Cloud Computing Technologies - Project

Project name: e-commerce microservice app
Author: Aleksander Chotecki, ID: V11141
Project link: https://github.com/heyimjustalex/e-commerce-microservices
Deployed at: https://cloudcomputingtechnologies.pl/
Presentation videos: https://youtu.be/FpN1-x2qDtk?feature=shared

Introduction

The aim of this project was to develop a distributed microservice system designed to facilitate fundamental shop functionalities like: listing/adding products, listing/adding orders, user registering/logging-in. The system consists of microservice applications that collaborate by transmitting events through the message broker. Due to its distributed nature of functionalities the system demonstrates resilience to a certain extent. In case of failure it consumes lacking events from the broker and makes the databases consistent. In the case of failure of microservices, it retrieves missing events from the broker and ensures the consistency of databases.

Database credentials: admin / pass
App admin credentials: admin@admin.com **/** admin@admin.com
App user credentials: aaa@aaa.com / aaa@aaa.com

System architecture

The system consists of a reverse proxy that serves the frontend application and passes API requests from frontend to backend. The backend consists of 4 microservices: gateway-ms, authentication-ms, products-ms, orders-ms. There are 3 document databases for each of the microservices except for gateway-ms which has some in-code data embedded. Message broker is used by two of the microservices products-ms and orders-ms in order to keep the state of orders consistent.

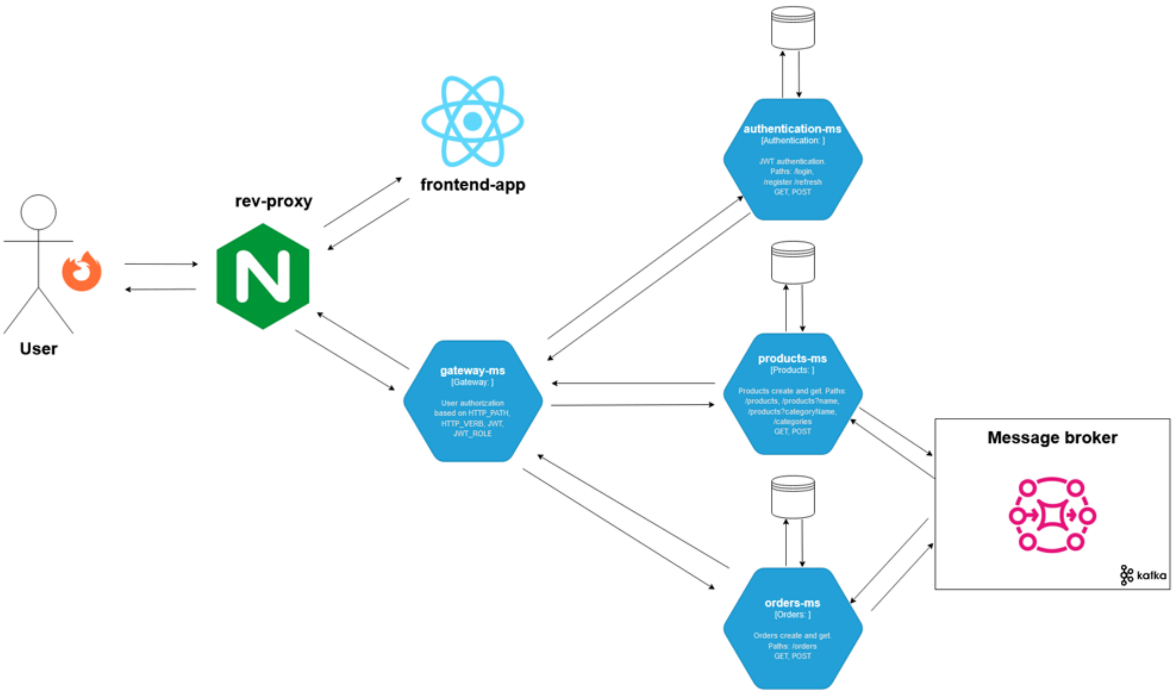

Figure 1 - Basic system architecture

Backend

Backend architecture

All of the services are developed with REST API Python Framework FastAPI. Each of the services have different purposes and therefore, conceptually separated functionalities. Three of the microservices use separate document databases and two of these keep data consistent by using events and brokers.

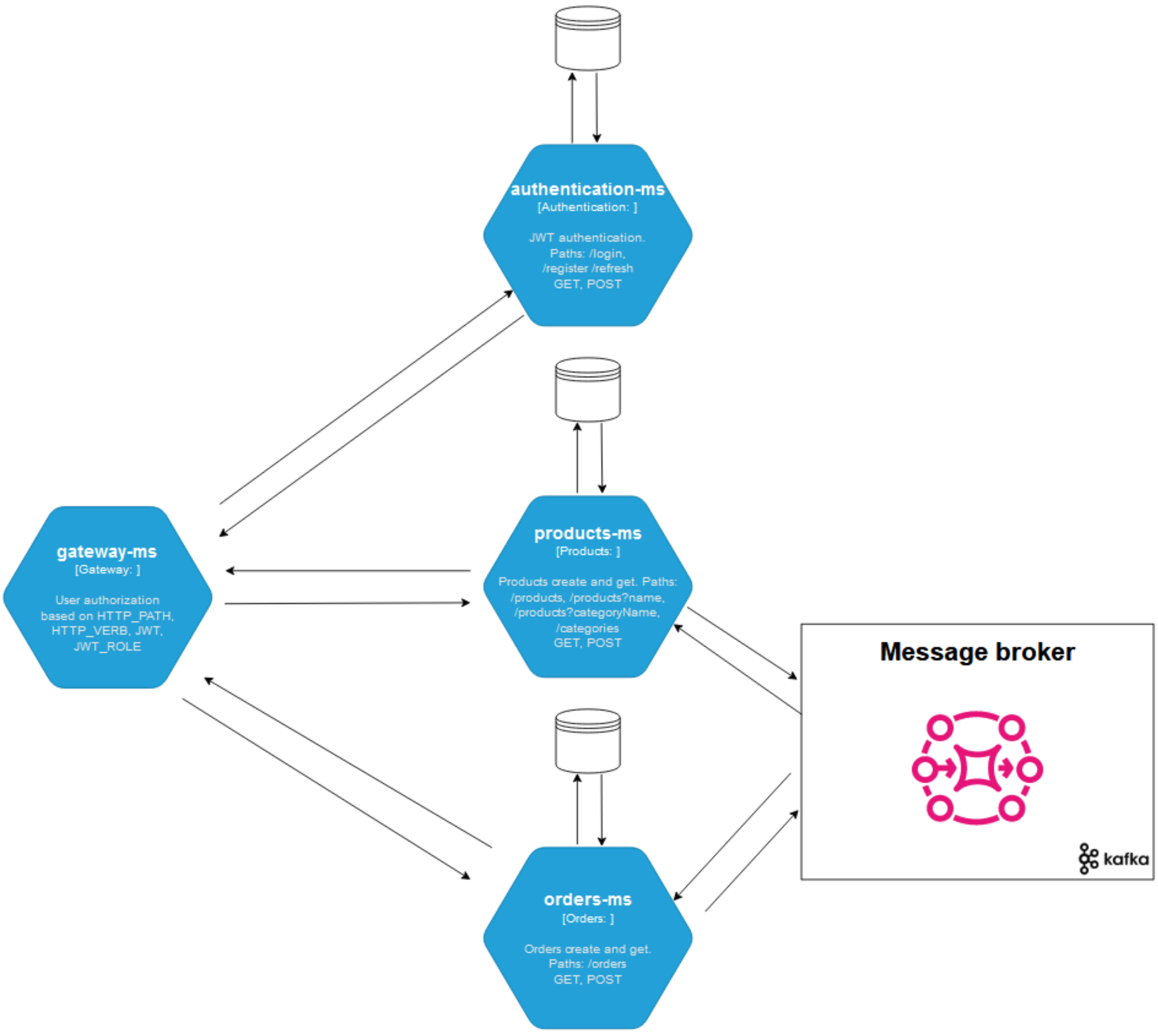

Figure 3 - Backend architecture

Microservices

Backend system consists of:

- **gateway-ms -** Microservice responsible for passing the request to relevant microservice (based on requested path) and for authorization of users based on the token that user attaches to the requests. JWT token is decoded by the microservice and verified using private secret. Based on the role and the endpoint that the user tries to access (HTTP_VERB and PATH) the decision is made regarding passing the access or returning relevant HTTP_CODE.

```
backend > gateway-ms > app > const > const.py > ...
endpoint_redirect_map: dict[str, str] = {
        "/api/products": "http://products-ms:8000",
        "/api/categories": "http://products-ms:8000",
        "/api/register": "http://authentication-ms:8000",
        "/api/login": "http://authentication-ms:8000",
        "/api/refresh": "http://authentication-ms:8000",
        "/api/orders":"http://orders-ms:8000",
        "/api/products_error":"http://products-ms:8000",
        "/api/authentication_error":"http://authentication-ms:8000",
        "/api/orders_error":"http://orders-ms:8000",
    }

endpoint_access_map: dict[tuple[str,str,str],bool] = {
            # If not listed access is allowed
            #     ENDPOINT    HTTP_VERB   ROLE
            ("/api/products", "POST", "user"): False,
            ("/api/products", "POST", "visitor"): False,
            ("/api/orders", "GET", "visitor"): False,
```

Figure 4 - Relevant access code for all of the endpoints

- **authentication-ms -** Microservice responsible for registering and logging-in users by providing JWT authentication. There is also a refresh functionality to renew an access token by using refresh token. User credentials (email and password) and roles (user or admin) are saved in the document database.

```
backend > authentication-ms > app > controllers > authentication_controller.py > error
    tags=['auth']
)

@router.post("/register", response_model=UserRegisterResponse, status_code=status.HTTP_201
def register(data: UserRegisterRequest, service:UserService = Depends(get_user_service)):
    user:User = service.create_user(data,"user")
    return UserRegisterResponse(email=user.email, role=user.role)

@router.post("/login",response_model=UserLoginResponse, status_code=status.HTTP_200_OK)
def login(data:UserLoginRequest, response: Response, user_service:UserService = Depends(ge
     user:User = user_service.check_user_credentials(data.email,data.password)

     access_token :str = token_service.create_access_token(user)
     refresh_token : str = token_service.create_refresh_token(user)

     response_model : UserLoginResponse = UserLoginResponse(access_token=access_token, ref
     response.headers.append("Token-Type","Bearer")

     return response_model
```

Figure 5 - Example endpoints implementation from authentication-ms controller

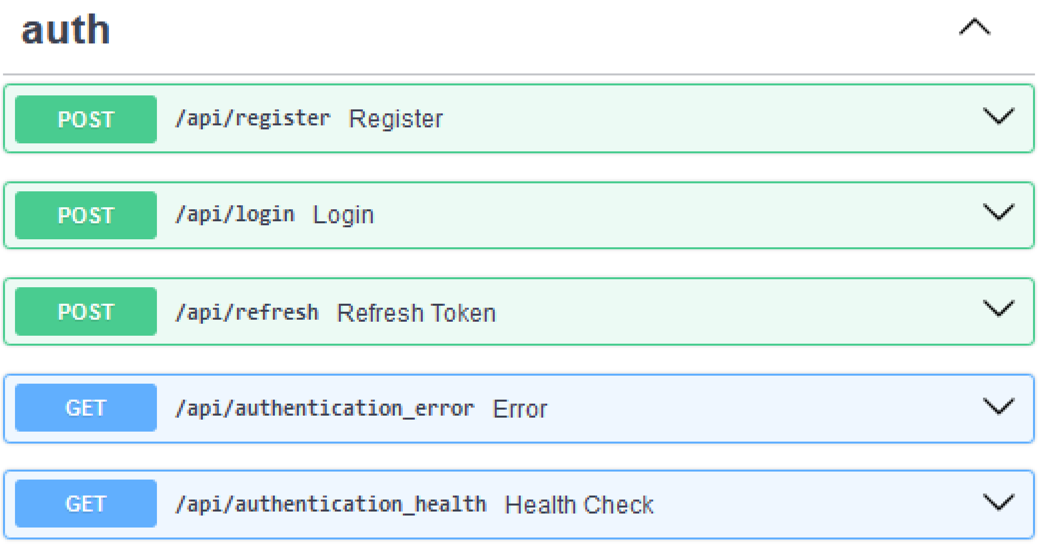

Figure 6 - Authentication-ms endpoint documentation

- **products-ms** - Microservice responsible for listing the products, getting single product by name or multiple by category. Administrator of the shop is also able to add a new product. This microservice is able to generate/consume relevant events: ProductCreateEvent, OrderCreateEvent, OrderStatusUpdateEvent, ProductsQuantityUpdateEvent. It checks whether the event has already been consumed in order to avoid any duplicates and inconsistencies in the database.

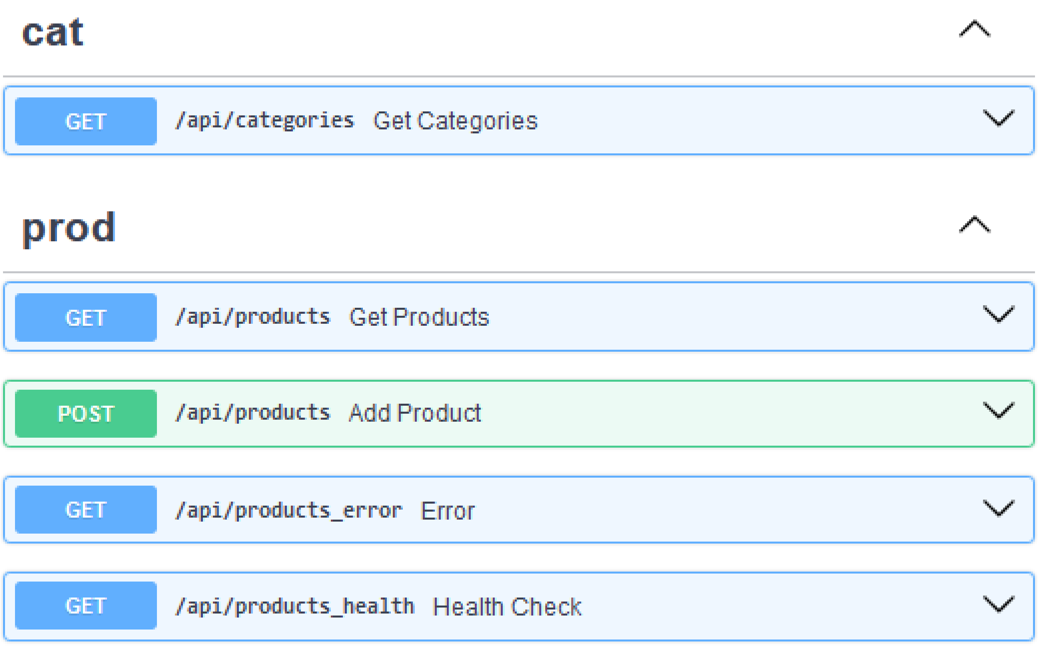

Figure 7 - products-ms endpoint documentation

- **orders-ms** - Microservice responsible for listing the orders and accepting a new order. This microservice is able to generate/consume relevant events: ProductCreateEvent, OrderCreateEvent, OrderStatusUpdateEvent, ProductsQuantityUpdateEvent. It also checks whether the event has already been consumed in order to avoid any duplicates and inconsistencies in the database.

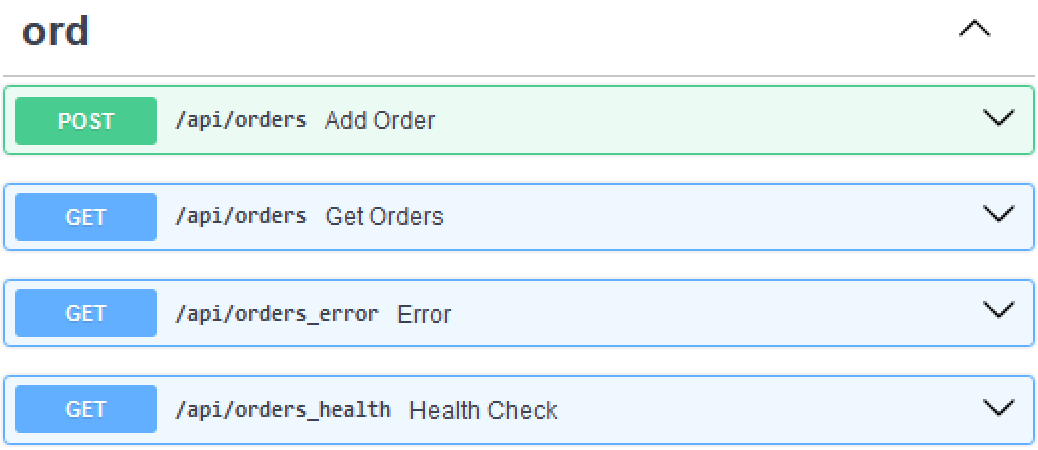

Figure 8 - orders-ms endpoint documentation

Layered architecture of each service

All of the microservices except for gateway-ms have a layered structure with 3 main layers: controller, service and repository. Abstraction helps to separate endpoints definition, business logic and data manipulation functionalities.

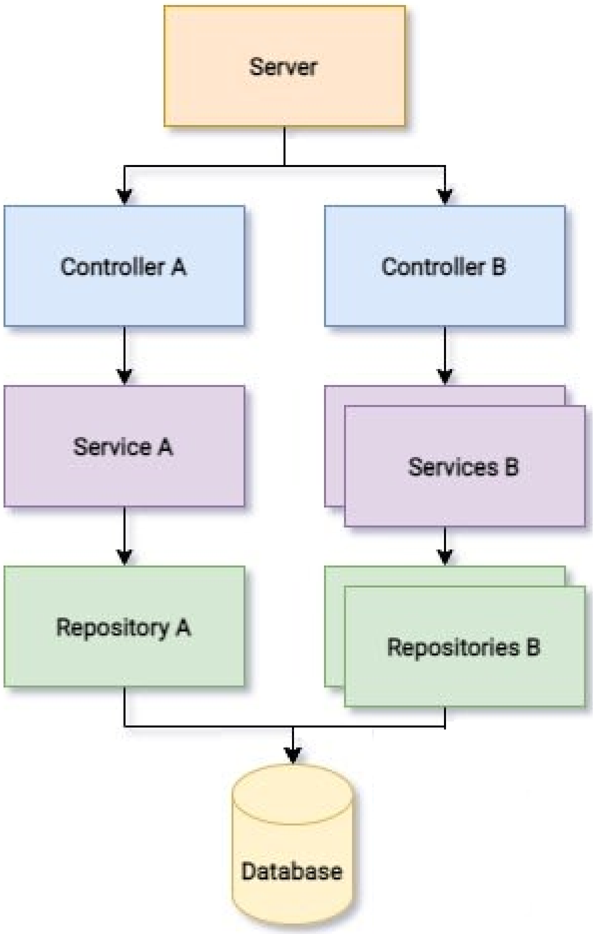

Figure 8 - Basic backend layered architecture for each for the microservices [1]

Event Handling

Two of the microservices - products-ms and orders-ms communicate through the Kafka message broker by producing and consuming relevant events. Event model classes are serialized by one service and deserialized by the other one. In order to make the system consistent both of the microservices use local transactions to save data in local databases and at the same time publish it to the broker. If either of these operations fail, the whole transaction fails.

.

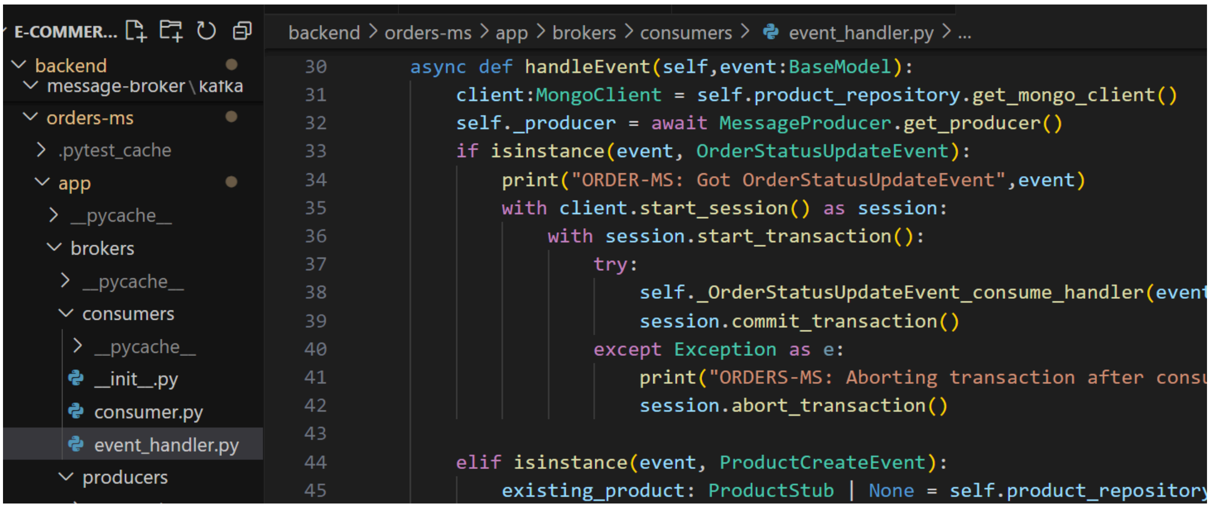

Figure 9 - Example transaction in orders-ms

Services products-ms and orders-ms have also consumer and producer classes that work asynchronously using aiokafka package, so sending or receiving messages does not block REST API functionality.

Containerization

Each of the backend microservices is containerized with Docker. Services have Dockerfile and Dockerfile.prod versions of Dockerfile and they differ with package requirements that are embedded in Docker image and stages (production includes stage of testing). For development purposes docker-compose is used with hot-reload of the code and bind mounts.

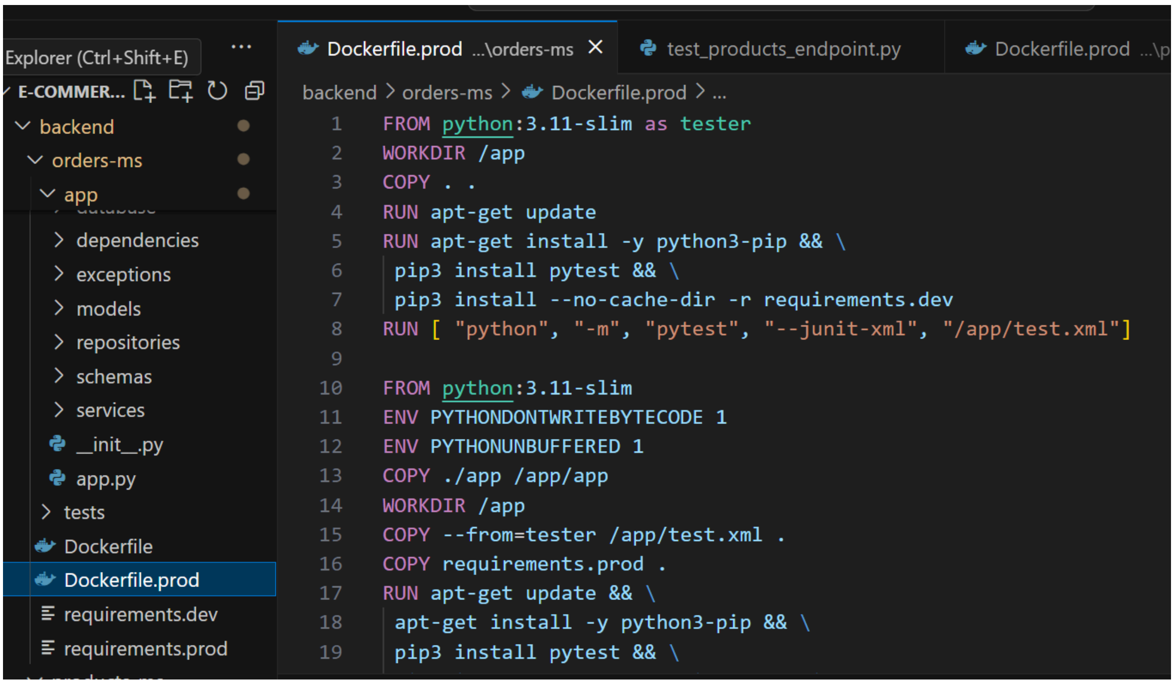

Figure 10 - Production dockerfile for orders-ms that uses 2-stage build with testing using pytest

Tests

Three of the microservices (orders-ms, products-ms and authentication-ms) are tested with pytest and mongomock. The DB that is used by endpoints is MongoMock which is an in-memory database, so the whole testing process might be performed in a single container and not a specialized environment (get_db function mocked).

```python

def test_Given_ProperUser_When_RegisteringTheUser_Then_CreatedRequestIsReturned(
    inmemory_database_creation_function: Callable[[], Database[Any]],

) -> None:
    # Mock DB
    app.dependency_overrides[Connector.get_db] = inmemory_database_creation_function

    # Given
    user_data: dict[str, str] = {
        "email": "test@test.com",
        "password": "password123",
        "role":"user"
    }
    # When
    response: Response = client.post(API_AUTHENTICATION_PREFIX+"/register", json=user_data)

    # Then
    assert response.status_code == status.HTTP_201_CREATED
    assert response.json() == {"email": "test@test.com", "role":"user"}

```

Figure 11 - Example tests (written with the GivenWhenThen convention)

There are some downsides with regards to using mongomock. I was not able to mock methods that are responsible for transactions, due to Mongomock's lack of support for a session object [2] that I use for transactions (for saving in local databases and publishing events). That makes me unable to test endpoints that use transactions.

backend > orders-ms > app > services > order_service.py > OrderService > create_order_with_event_OrderCreate

```python

    async def create_order_with_event_OrderCreate(self, data:OrderCreateRequest) -> Order:
        self._verify_create_request_format(data)
        email : str = data.email

        #Bought product
        products: List[ProductStub] = [ProductStub(name=product.name.lower(), price=self._get_product_price(produc
        self._check_products_existance_and_quantity(products)
        order_cost:float = self._calculate_order_cost(products)
        order : Order = Order(client_email=email.lower(),cost=order_cost, status="PENDING", products=products)

        client:MongoClient = self.product_repository.get_mongo_client()
        with client.start_session() as session:
            with session.start_transaction():
                try:
                    created_order: Order = self.order_repository.create_order(order, session)
                    await self._publish_OrderCreateEvent_to_broker(created_order)
                except Exception as e:
                    session.abort_transaction()
                    raise OrderPlacingFailed()
        return created_order
```

Figure 12 - Untestable case

Databases

There are 3 distributed databases in the system. Each of them is a Bitnami MongoDB document database that supports transactions. All of them have different schemas, but the schema creation and the initialization of data is done in the same way with Docker and the db/init.js file for each of them. Schemas for each of the microservices:

authentication-db

Inside authentication-db there is a 'shop' document database and inside of it there is a collection 'users'. Each record in the collection has 4 attributes *_id, email, role password_hash*. Email is used as username and Role may be user or admin and hashing algorithm SHA-256 is used for the password transformation.

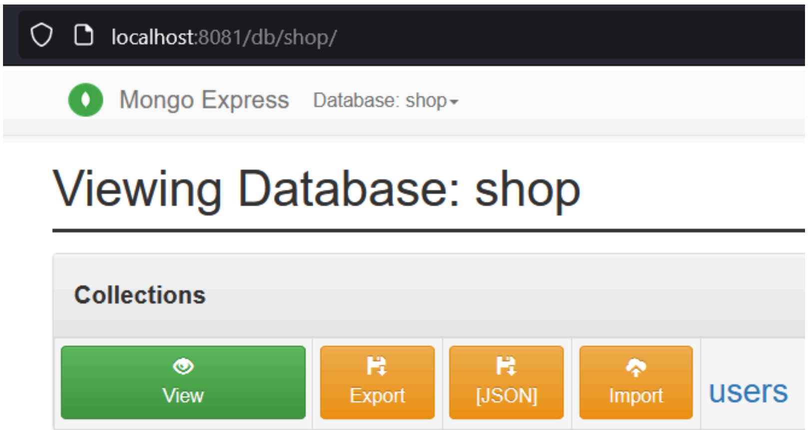

Figure 13 - collections in authentication-db

products-db

There is a 'shop' document database and inside there are collections 'orders', 'products', 'categories'.

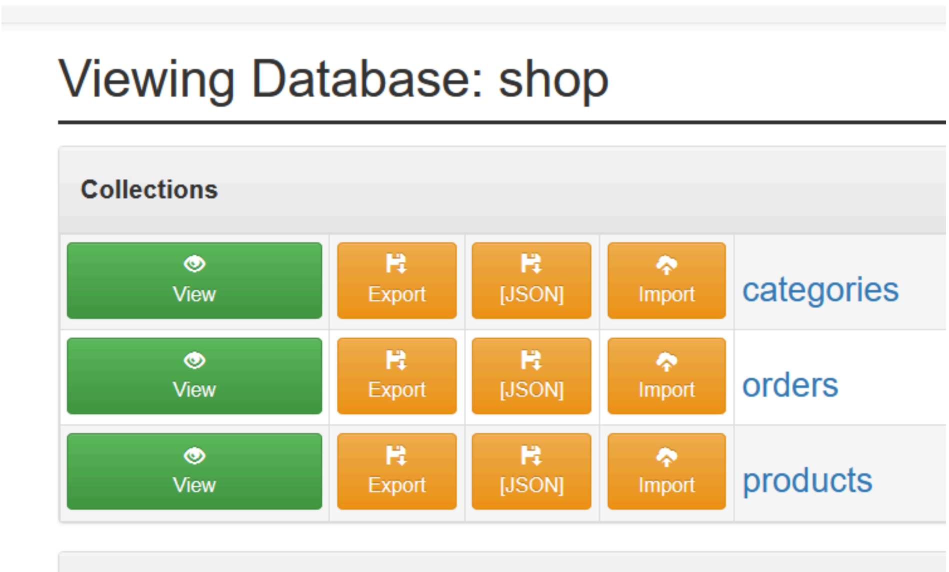

Figure 14 - collections in products-db

Each record in the 'products' collection has 6 attributes *_id, name, description, price, quantity, categories*. Categories is an array of _id from categories collection.

_id	name	description	price	quantity	categories
10fefe4a1cad4140785928a4	cutlery	An interesting set of cutlery	5.99	200	22fefe4a1cad4140785928a4
11fefe4a1cad4140785928a4	chair	A comfortable armchair	29.99	30	23fefe4a1cad4140785928a4,22
12fefe4a1cad4140785928a4	laptop	A powerful computing device	1299.99	1	21fefe4a1cad4140785928a4
13fefe4a1cad4140785928a4	headphones	Wireless noise-cancelling headphones	99.99	1	21fefe4a1cad4140785928a4

Figure 15 - records in products-db products collection

The 'categories' collection has _id and name attributes. Ids are bound to product records and we are able to see the name of the category when listing products from the backend.

_id	name
21fefe4a1cad4140785928a4	electronics
22fefe4a1cad4140785928a4	kitchen
23fefe4a1cad4140785928a4	furniture

Figure 16 - records in products-db categories collection

Orders stub collection has information about which of the events have been already consumed by products-ms. If there is OrderCreateEvent sent from orders-ms then products-ms consumes the event, save status as ACCEPTED if there is enough quantity for the products. Then responds with another event about the product's quantity change.

_id	status
71fefe4a1cad4140785928a4	ACCEPTED
72fefe4a1cad4140785928a4	ACCEPTED
73fefe4a1cad4140785928a4	REJECTED

Figure 17 - records in products-db orders collection

orders-db

Inside orders-db there is a 'shop' document database and inside there are collections 'orders', 'products' that are responsible for order data handling.

Each record in the 'orders' collection has 4 attributes *_id, client_email, status, cost, products.* The contents of the product collections are stub versions of products embedded in the order record.

_id	client_email	status	cost	products
71fefe4a1cad4140785928a4	aaa@aaa.com	ACCEPTED	83.94	[{ "name": "cutlery", "price": 5.99, "quantity": 4 }, { "name": "chair", "price": 29.99, "quantity": 2 }]
72fefe4a1cad4140785928a4	bbb@bbb.com	ACCEPTED	1299.99	[{ "name": "laptop", "price": 1299.99, "quantity": 1 }]

Figure 18 - records in orders-db orders collection

Each record in the 'orders' collection has 4 attributes: name, price, quantity. This collection is only to make an initial decision about the sale of products. If orders-ms does not have required product quantity i reject the user request and does not try to generate events. If the quantity of bought products in the local products collection is available, it creates an OrderCreate event and the order in the 'orders' collection in the pending state. If products-ms is available it can either accept or reject the order.

_id	name	price	quantity
10fefe4a1cad4140785928a4	cutlery	5.99	200
11fefe4a1cad4140785928a4	chair	29.99	30
12fefe4a1cad4140785928a4	laptop	1299.99	1
13fefe4a1cad4140785928a4	headphones	99.99	1

Figure 19 - records in orders-db products collection

Events

Microservices products-ms and orders-ms produce events in order to keep their databases consistent. Available event types are listed as models in the code in both of the applications.

backend > products-ms > app > models > models.py > OrderSta

```python
# Event models
class ProductCreateEvent(BaseModel):
    type: str
    product:ProductStub
    class Config:
        include_private_fields = True

class OrderCreateEvent(BaseModel):
    type: str
    order:Order

class OrderStatusUpdateEvent(BaseModel):
    order_id:PyObjectId
    type: str
    status: str

class ProductsQuantityUpdateEvent(BaseModel):
    type: str
    products: List[ProductStub]
```

Figure 20 - Available events

ProductCreateEvent

This event is generated when the administrator of the shop adds a new product to the inventory, and products need to be populated into both of the ‘products’ collections in ‘orders-db’ and ‘products-db’.

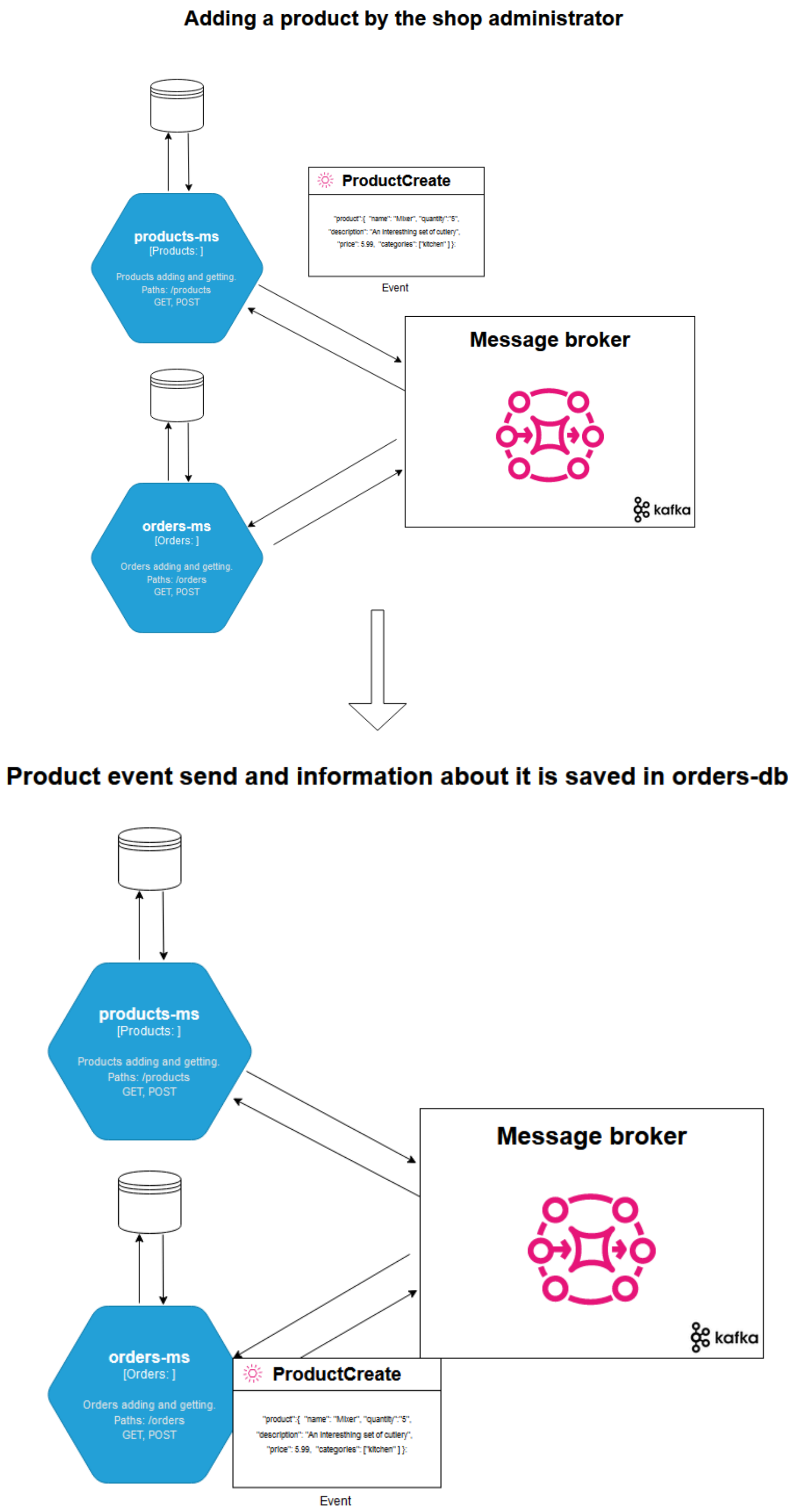

Figure 21 - Adding new product to the inventory, ProductCreateEvent flow

OrderCreateEvent, OrderStatusUpdateEvent, ProductsQuantityUpdateEvent

OrderCreateEvent event is generated by orders-ms when the user makes the order. The order is initially in the PENDING state (status) and if products-ms consumes the event and approves it (checks availability of products), then OrderStatusUpdateEvent, ProductsQuantityUpdateEvent are sent from products-ms for orders-ms to update its collections.

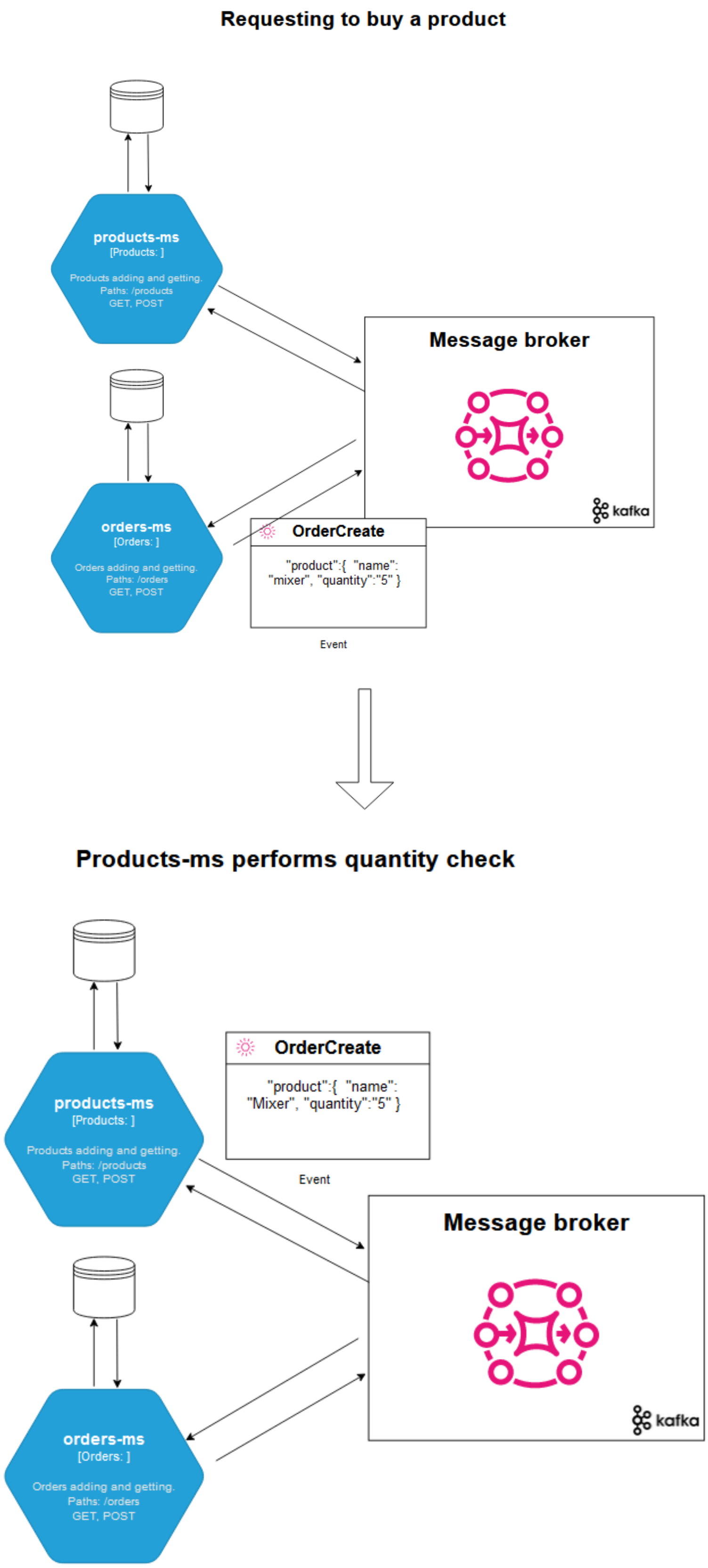

Figure 22 - Order placing by a client. OrderCreateEvent is created

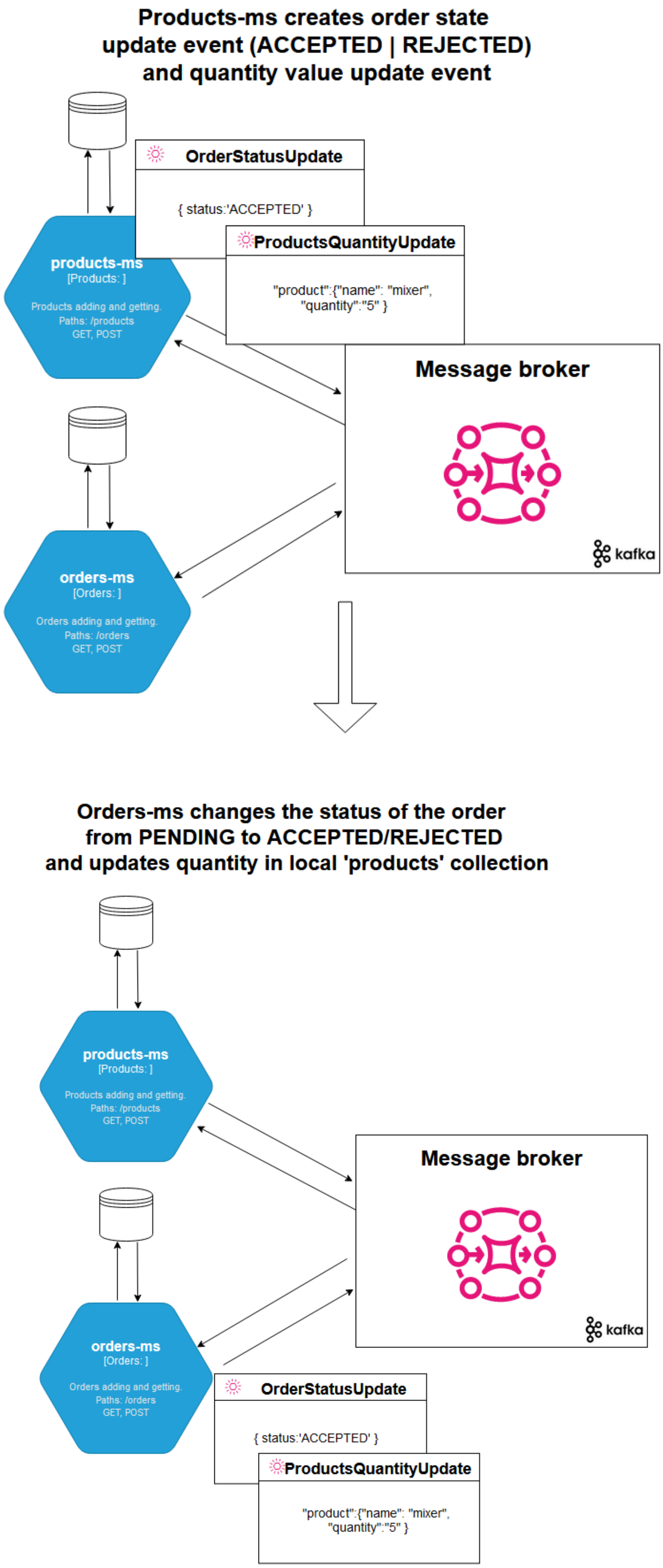

Figure 23 - Order accepted by products-ms. Product-ms generates relevant events. Orders-ms updates order status and quantity.

Broker

For the message broker a containerized version of Kafka is used with zookeeper software providing broker's configuration. To communicate between Kafka and backend services aiokafka package is used by backend. Kafka broker is a single point of failure for the microservices that use it for event emitting (orders-ms, products-ms). There is only a single instance of Kafka broker in the infrastructure.

infrastructure > kubernetes > gcp > ! zookeeper.yaml > {} spec > {} template > {} spec >
io.k8s.api.core.v1.Service (v1@service.json) | io.k8s.api.apps.v1.Deployment (v1@deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: zookeeper-dep
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zookeeper
  template:
    metadata:
      labels:
        app: zookeeper
    spec:
      containers:
        - name: zookeeper
          image: confluentinc/cp-zookeeper:7.3.2
          ports:
            - containerPort: 2181
          env:
            - name: ZOOKEEPER_CLIENT_PORT
              value: "2181"
```

Figure 24 - Zookeeper K8S production yaml file

io.k8s.api.core.v1.Service (v1@service.json) | io.k8s.api.apps.v1.Deployment (v1@deployment.json)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: message-broker-dep
spec:
  replicas: 1
  selector:
    matchLabels:
      app: message-broker
  template:
    metadata:
      labels:
        app: message-broker
    spec:
      containers:
        - name: message-broker
          image: confluentinc/cp-kafka:7.3.2
          imagePullPolicy: "Always"
          ports:
            - containerPort: 9092
              name: kafka
            - containerPort: 19092
```

Figure 25 - Kafka K8S production yaml file

Frontend

Frontend is executed in the client's browser (client-side rendering) and was developed with React.js + Vite + VanillaJS + Bootstrap. The code uses a custom hook for http request and global contexts (AuthContext, CartContext) for JWT and cart management. Token is stored in LocalStorage and the design is responsive.

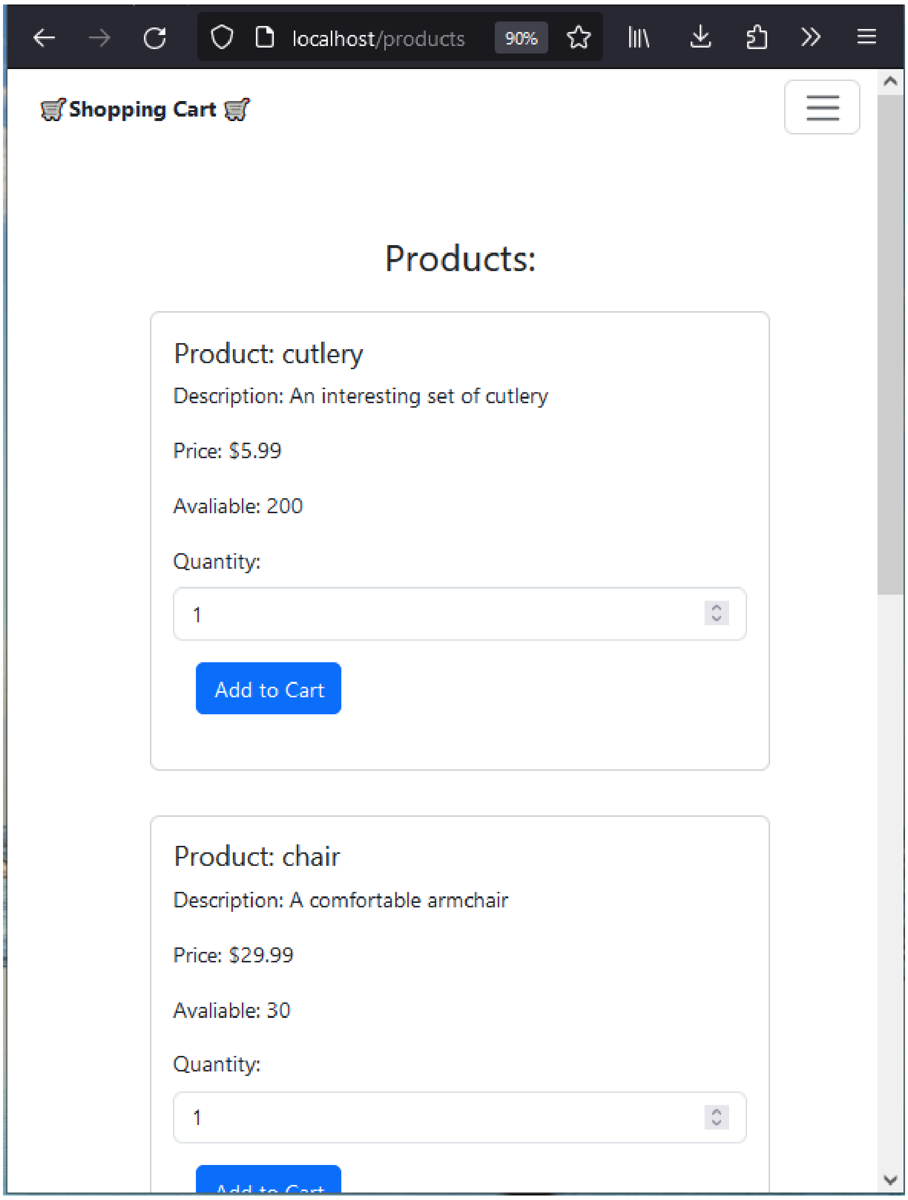

Figure 26 - Responsive UI design

Production version of frontend is deployed with 2-stage build and is hosted with nginx server. Production build is generated with '*vite build'* command underneath.

frontend > app > Dockerfile.prod > ...

```
FROM node:20-alpine as builder
WORKDIR /app
COPY . .
ARG VITE_APP_BACKEND_ADDRESS
ENV VITE_APP_BACKEND_ADDRESS $VITE_APP_BACKEND_ADDRESS
RUN npm install
RUN npm run build

FROM nginx:1.25.4-alpine-slim as prod
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf  /etc/nginx/conf.d
RUN rm /etc/nginx/conf.d/default.conf
EXPOSE 3000
CMD ["nginx", "-g", "daemon off;"]
```

Figure 27 - 2-stage Dockerfile for production build

Reverse-proxy / Ingress

Reverse proxy is used to distribute requests between frontend and API. For development there is an nginx used with configuration presented below.

nginx_proxy > nginx.conf

```
}

http {
server {
    listen 80;
    server_name example.com;

    # Frontend configuration
    location / {
        proxy_pass http://frontend:3000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    # Backend configuration
    location /api {
        proxy_pass http://gateway-ms:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
```

Figure 28 - Reverse proxy - nginix.conf

For production there is an ingress configuration also based on nginx with in-line certificates (base64 encoded) and in-line config file.

```yaml
infrastructure > kubernetes > gcp > ! ingress.yaml > {} spec > [ ] rules >
apiVersion: v1
kind: Secret
metadata:
  name: nginx-tls-secret
  namespace: default
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1
  tls.key: LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ing
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  tls:
    - hosts:
        - cloudcomputingtechnologies.pl
      secretName: nginx-tls-secret
  rules:
```

Figure 29 - Ingress.yaml used for k8s GCP production

Deployment

For deployment I use GCP (GKE) with IAC configuration files. The infrastructure creation is not fully automated due to DNS configuration and dynamic LoadBalancer public IP, but steps for configuration are listed in README.md.

IAC

For IAC I use Terraform with hashicorp/google 4.0 provider. The configuration allows nodes to be scaled up dynamically. The nodes are e2-medium instances.

```hcl
resource "google_container_node_pool" "spot" {
  name    = "spot"
  cluster = google_container_cluster.primary.id

  management {
    auto_repair  = true
    auto_upgrade = true
  }

  autoscaling {
    min_node_count = 0
    max_node_count = 10
  }

  node_config {
    preemptible  = true
    machine_type = "e2-medium"

    labels = {
      team = "devops"
    }

```

Figure 30 - Ingress.yaml used for k8s GCP production

CI/CD

For CI/CD CircleCI is used that is connected to Github. After every commit to the main branch workflows are triggered and new Docker images are deployed to Dockerhub. Then the images are downloaded by GCP K8S to the infrastructure and pods are recreated. Relevant secrets for GCP are stored in the CircleCI platform and used by scripts to authenticate and modify the GCP cluster.

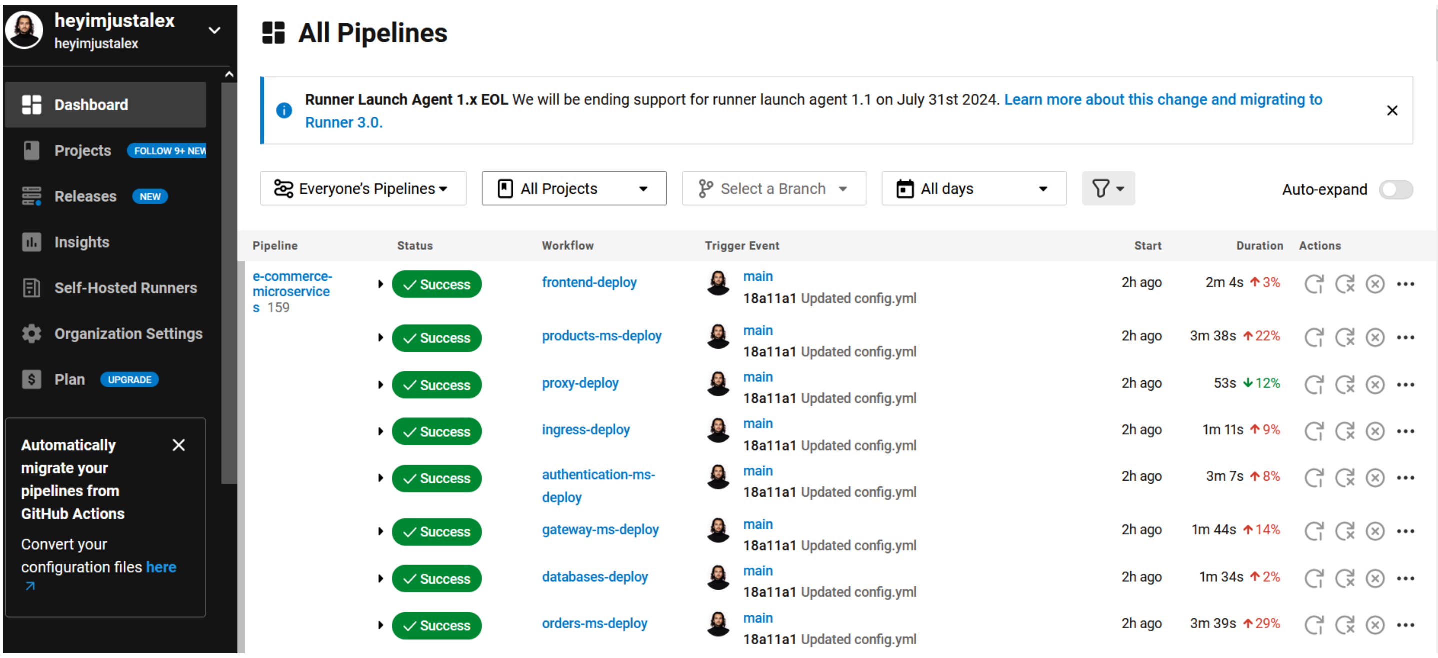

Figure 31 - successful deployment

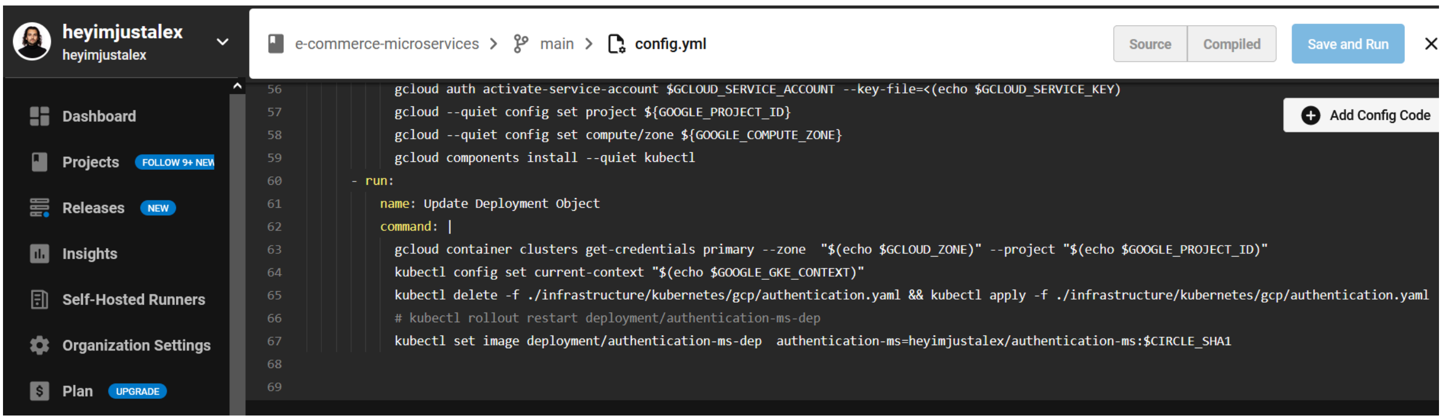

Figure 32 - A fragment of script for GCP deployment

Non-functional requirements [3]

According to the first report I have chosen availability/resiliency and security requirements. Availability/resiliency is achieved by designing microservices in a stateless way and changing the number of replicas in the configuration of K8S deployment.

```
infrastructure > kubernetes > gcp > ! orders.yaml
io.k8s.api.core.v1.Service (v1@service.json) | io.k
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-ms-dep
spec:
  replicas: 2
  selector:
    matchLabels:
      app: orders-ms
  template:
    metadata:
      labels:
        app: orders-ms
    spec:
```

Figure 33 - multiple replicas in the K8S deployment object

```
alex on alex-PC via gke_peppy-citron-421113_europe-north1-a_primary() at …
NAME                                     READY   STATUS    RESTARTS   AGE
authentication-db-dep-86c447bf69-nc9lr   1/1     Running   0          18h
authentication-ms-dep-779f64bd84-dt5tc   1/1     Running   0          5h
frontend-dep-786c55f4c8-4jfwk            1/1     Running   0          21s
gateway-ms-dep-55db8c45bd-6qhs5          1/1     Running   0          5h38m
orders-db-dep-0                          1/1     Running   0          5h
orders-ms-dep-56f6bff99b-jrjxr           1/1     Running   0          5h
orders-ms-dep-56f6bff99b-wstxs           1/1     Running   0          5h
products-db-dep-0                        1/1     Running   0          5h
products-ms-dep-85b54cfb7b-2rv54         1/1     Running   0          5h36m
products-ms-dep-85b54cfb7b-bmfhp         1/1     Running   0          5h
zookeeper-dep-8dd689f78-ss4jh            1/1     Running   0          5h
```

Figure 34 - multiple pods run by deployments

Security is achieved by implementing a certificate to ingress with a ‘secret’ k8s object.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: nginx-tls-secret
  namespace: default
type: kubernetes.io/tls
data:
  tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS
  tls.key: LS0tLS1CRUdJTiBQUklWQVRFIEtFWS
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx-ing
  namespace: default
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  tls:
    - hosts:
      - cloudcomputingtechnologies.pl
      secretName: nginx-tls-secret
```

Figure 35 - ingress TLS implementation

Sources:
[1] https://dev.to/blindkai/backend-layered-architecture-514h
[2] https://github.com/mongomock/mongomock/issues/614
[3] https://en.wikipedia.org/wiki/Non-functional_requirement