

# 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://www.youtube.com/watch?v=FpN1-x2qDtk&list=PLngi0ogU-DkGDOLVHYbnTXp1Kfpk1ow0D>

## 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.

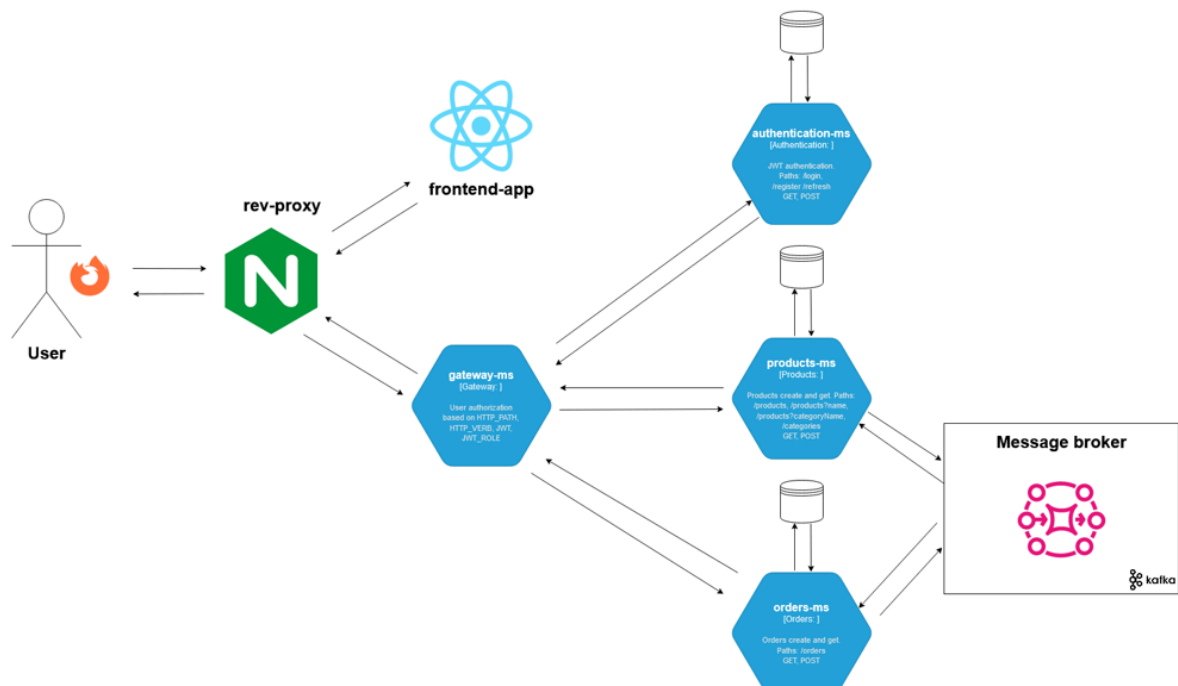


Figure 1 - Basic system architecture

# 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.

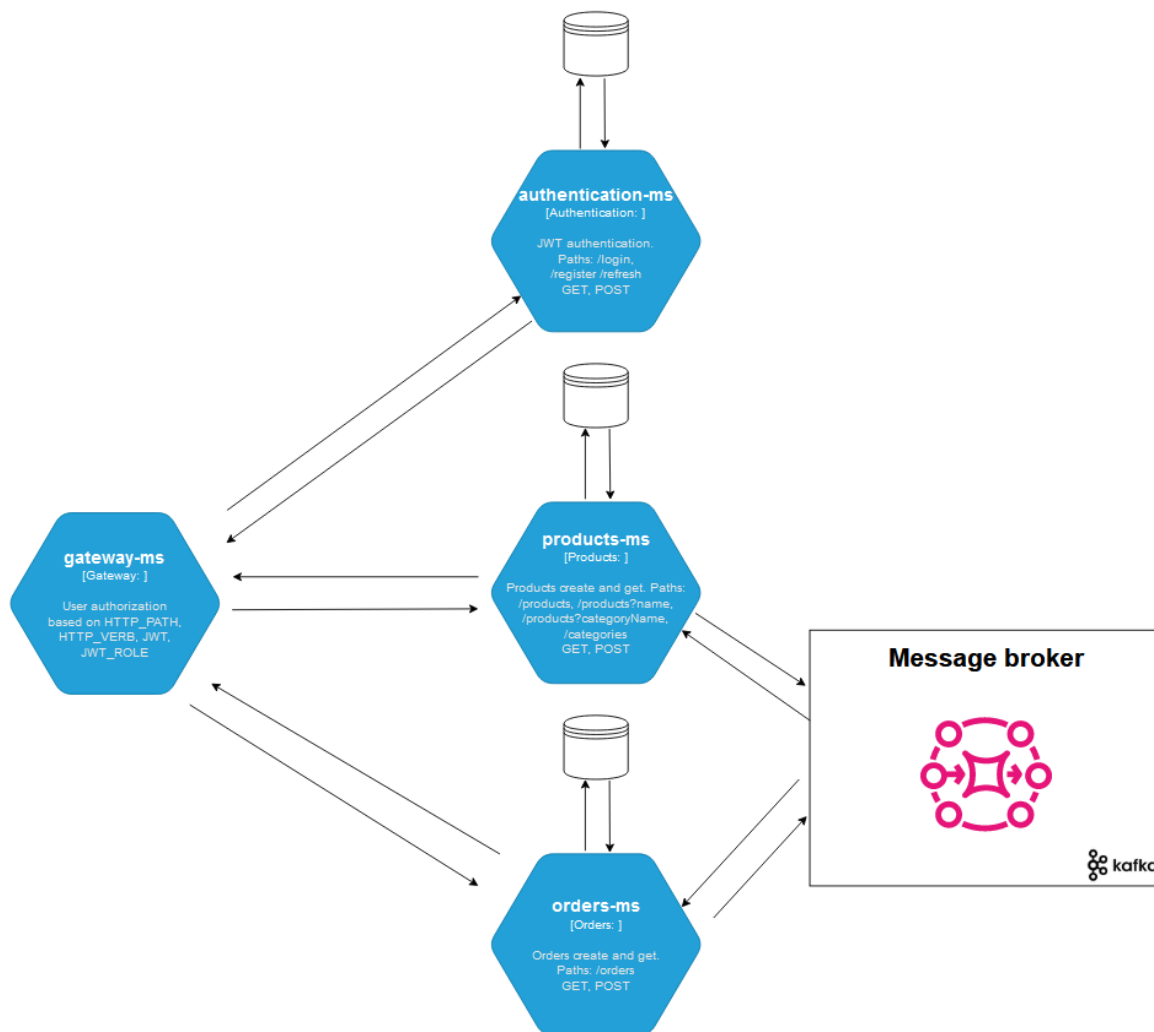


Figure 3 - Backend architecture

## 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 > ...
3  endpoint_redirect_map: dict[str, str] = {
4      "/api/products": "http://products-ms:8000",
5      "/api/categories": "http://products-ms:8000",
6      "/api/register": "http://authentication-ms:8000",
7      "/api/login": "http://authentication-ms:8000",
8      "/api/refresh": "http://authentication-ms:8000",
9      "/api/orders": "http://orders-ms:8000",
10     "/api/products_error": "http://products-ms:8000",
11     "/api/authentication_error": "http://authentication-ms:8000",
12     "/api/orders_error": "http://orders-ms:8000",
13 }
14
15  endpoint_access_map: dict[tuple[str, str, str], bool] = {
16      # If not listed access is allowed
17      #   ENDPOINT   HTTP_VERB   ROLE
18      ("/api/products", "POST", "user"): False,
19      ("/api/products", "POST", "visitor"): False,
20      ("/api/orders", "GET", "visitor"): False,
21      ("/api/orders", "POST", "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
14     tags=['auth']
15 )
16
17 @router.post("/register", response_model=UserRegisterResponse, status_code=status.HTTP_201_CREATED)
18 def register(data: UserRegisterRequest, service: UserService = Depends(get_user_service)):
19     user: User = service.create_user(data, "user")
20     return UserRegisterResponse(email=user.email, role=user.role)
21
22 @router.post("/login", response_model=UserLoginResponse, status_code=status.HTTP_200_OK)
23 def login(data: UserLoginRequest, response: Response, user_service: UserService = Depends(get_user_service)):
24     user: User = user_service.check_user_credentials(data.email, data.password)
25
26     access_token: str = token_service.create_access_token(user)
27     refresh_token: str = token_service.create_refresh_token(user)
28
29     response_model: UserLoginResponse = UserLoginResponse(access_token=access_token, refresh_token=refresh_token)
30     response.headers.append("Token-Type", "Bearer")
31
32     return response_model

```

Figure 5 - Example endpoints implementation from authentication-ms controller

auth			^
POST	/api/register	Register	▼
POST	/api/login	Login	▼
POST	/api/refresh	Refresh Token	▼
GET	/api/authentication_error	Error	▼
GET	/api/authentication_health	Health Check	▼

*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.

cat			^
GET	/api/categories	Get Categories	▼

prod			^
GET	/api/products	Get Products	▼
POST	/api/products	Add Product	▼
GET	/api/products_error	Error	▼
GET	/api/products_health	Health Check	▼

*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.

ord			^
POST	/api/orders	Add Order	✓
GET	/api/orders	Get Orders	✓
GET	/api/orders_error	Error	✓
GET	/api/orders_health	Health Check	✓

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.

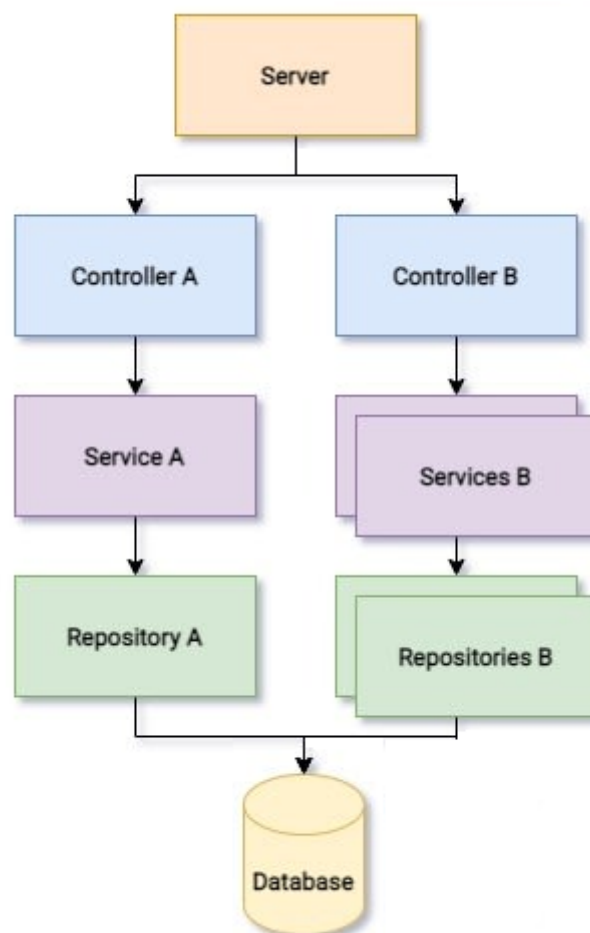


Figure 8 - Basic backend layered architecture for each for the microservices [1]

## 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.

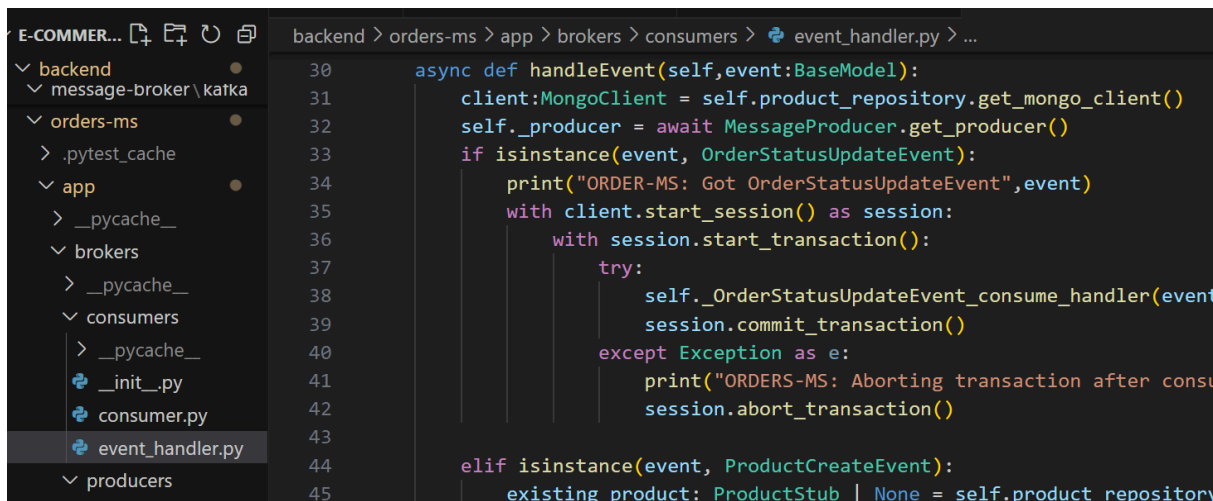


Figure 9 - Example transaction in orders-ms

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.

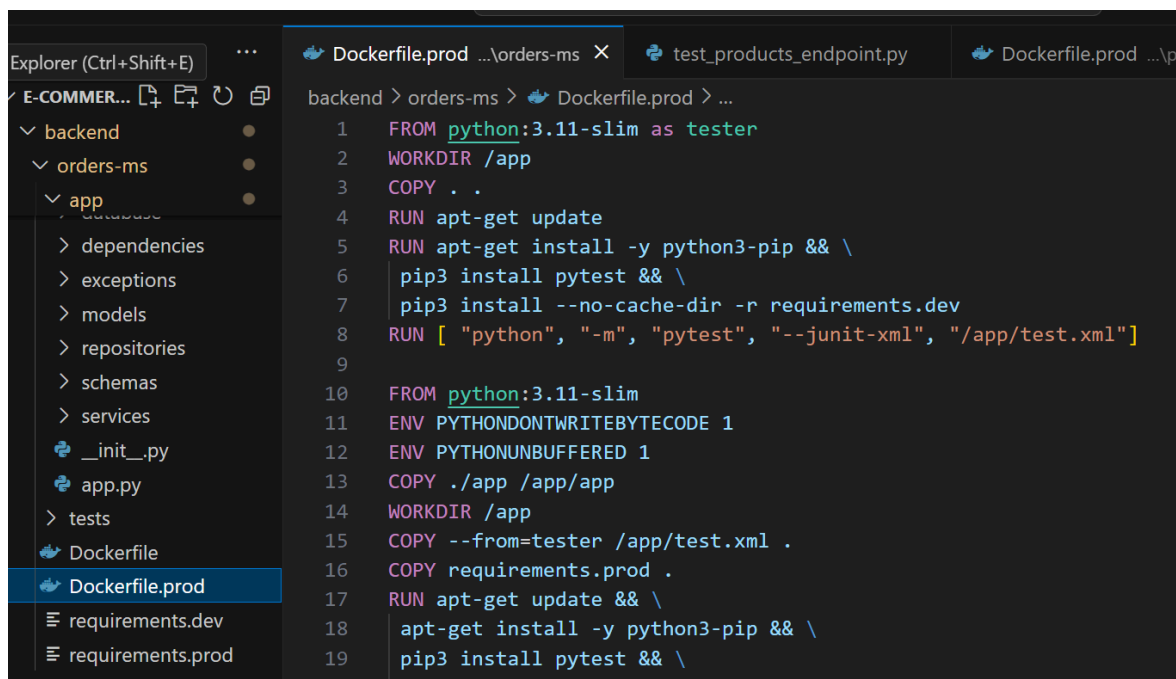


Figure 10 - Production dockerfile for orders-ms that uses 2-stage build with testing using pytest

## 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).

```
7
8 v def test_Given_ProperUser_When_RegisteringTheUser_Then_CreatedRequestIsReturned(
9     inmemory_database_creation_function: Callable[[], Database[Any]],
10
11 ) -> None:
12     # Mock DB
13     app.dependency_overrides[Connector.get_db] = inmemory_database_creation_function
14
15     # Given
16 v user_data: dict[str, str] = {
17     "email": "test@test.com",
18     "password": "password123",
19     "role": "user"
20 }
21     # When
22     response: Response = client.post(API_AUTHENTICATION_PREFIX+"/register", json=user_data)
23
24     # Then
25     assert response.status_code == status.HTTP_201_CREATED
26     assert response.json() == {"email": "test@test.com", "role": "user"}
27
```

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
76
77 async def create_order_with_event_OrderCreate(self, data: OrderCreateRequest) -> Order:
78     self._verify_create_request_format(data)
79     email : str = data.email
80
81     #Bought product
82     products: List[ProductStub] = [ProductStub(name=product.name.lower(), price=self._get_product_price(product.name.lower()),
83     self._check_products_existance_and_quantity(products)
84     order_cost:float = self._calculate_order_cost(products)
85     order : Order = Order(client_email=email.lower(),cost=order_cost, status="PENDING", products=products)
86
87     client:MongoClient = self.product_repository.get_mongo_client()
88     with client.start_session() as session:
89         with session.start_transaction():
90             try:
91                 created_order: Order = self.order_repository.create_order(order, session)
92                 await self._publish_OrderCreateEvent_to_broker(created_order)
93             except Exception as e:
94                 session.abort_transaction()
95                 raise OrderPlacingFailed()
96     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.

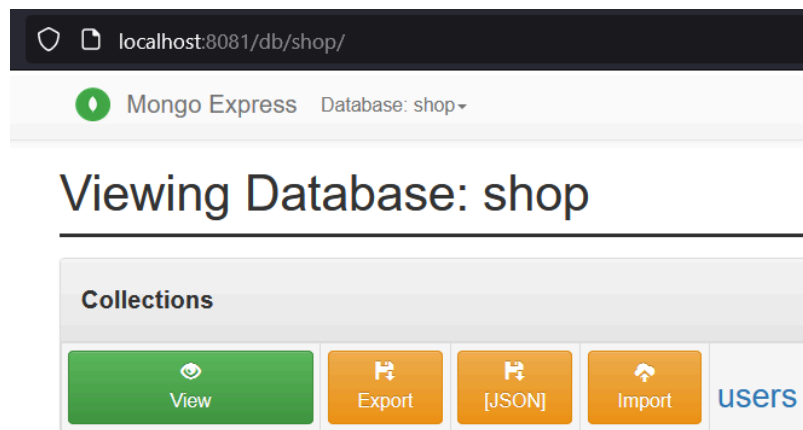


Figure 13 - collections in authentication-db

### 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	23fefe4a1cad4140785928a4,22fefe4a1cad4140785928a4
  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	<pre>ⓔ[   ⓔ{     "name": "cutlery",     "price": 5.99,     "quantity": 4   },   ⓔ{     "name": "chair",     "price": 29.99,     "quantity": 2   } ]</pre>
  72fefe4a1cad4140785928a4	bbb@bbb.com	ACCEPTED	1299.99	<pre>ⓔ[   ⓔ{     "name": "laptop",     "price": 1299.99,     "quantity": 1   } ]</pre>

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.

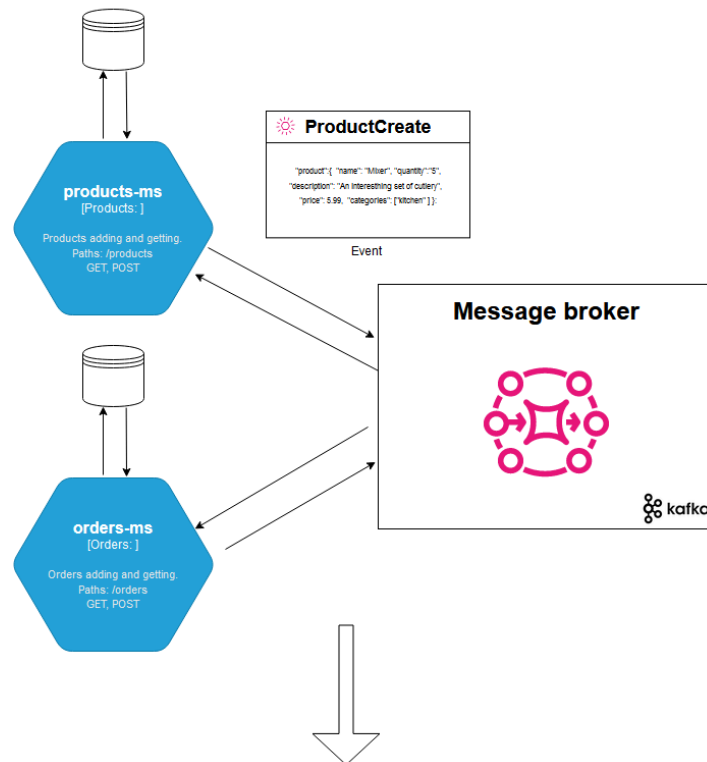
```
backend > products-ms > app > models > models.py > OrderSta
39 # Event models
40 class ProductCreateEvent(BaseModel):
41     type: str
42     product: ProductStub
43     class Config:
44         include_private_fields = True
45
46 class OrderCreateEvent(BaseModel):
47     type: str
48     order: Order
49
50 class OrderStatusUpdateEvent(BaseModel):
51     order_id: PyObjectId
52     type: str
53     status: str
54
55 class ProductsQuantityUpdateEvent(BaseModel):
56     type: str
57     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'.

## Adding a product by the shop administrator



## Product event send and information about it is saved in orders-db

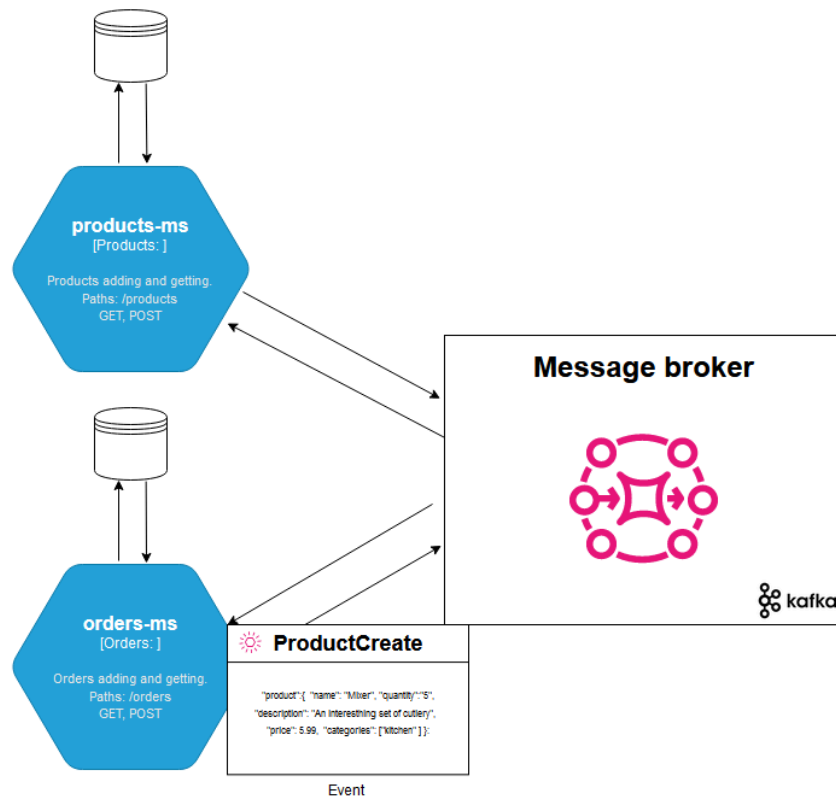
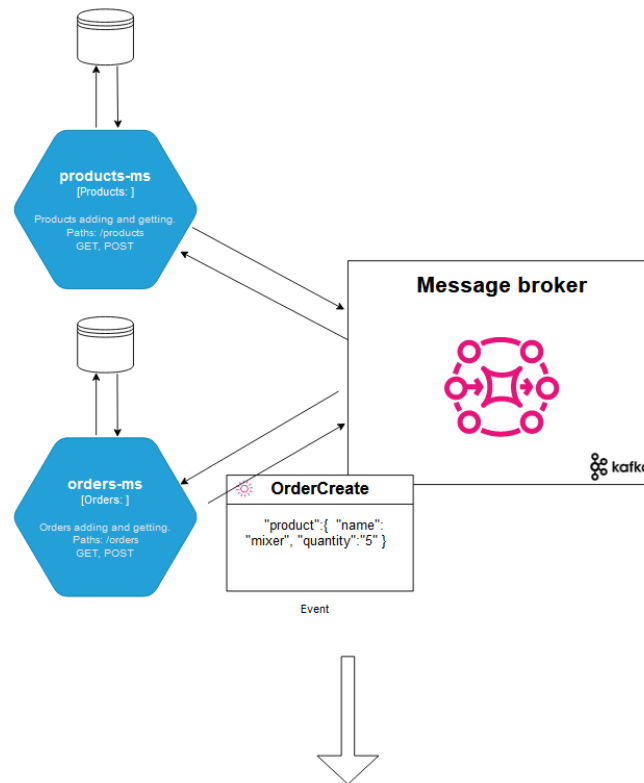


Figure 21 - Adding new product to the inventory, ProductCreateEvent flow

## 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.

### Requesting to buy a product



### Products-ms performs quantity check

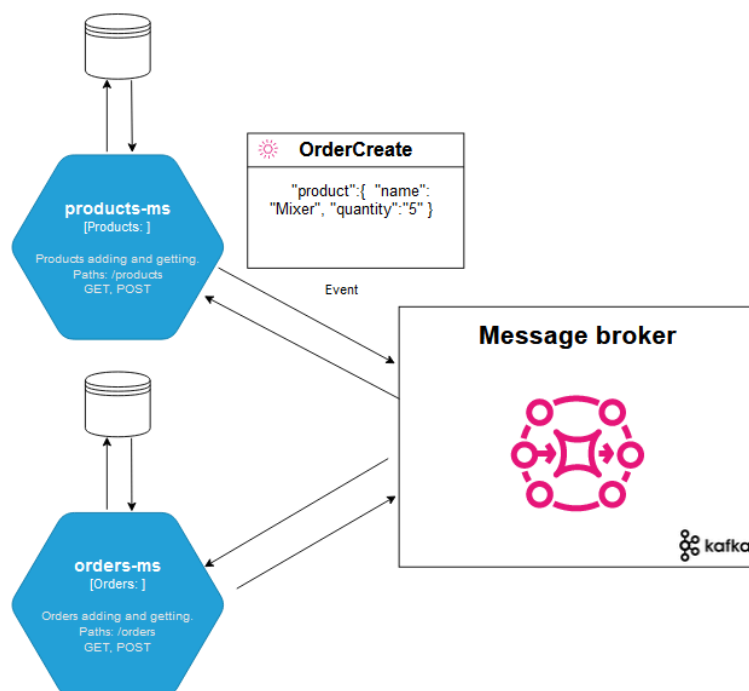
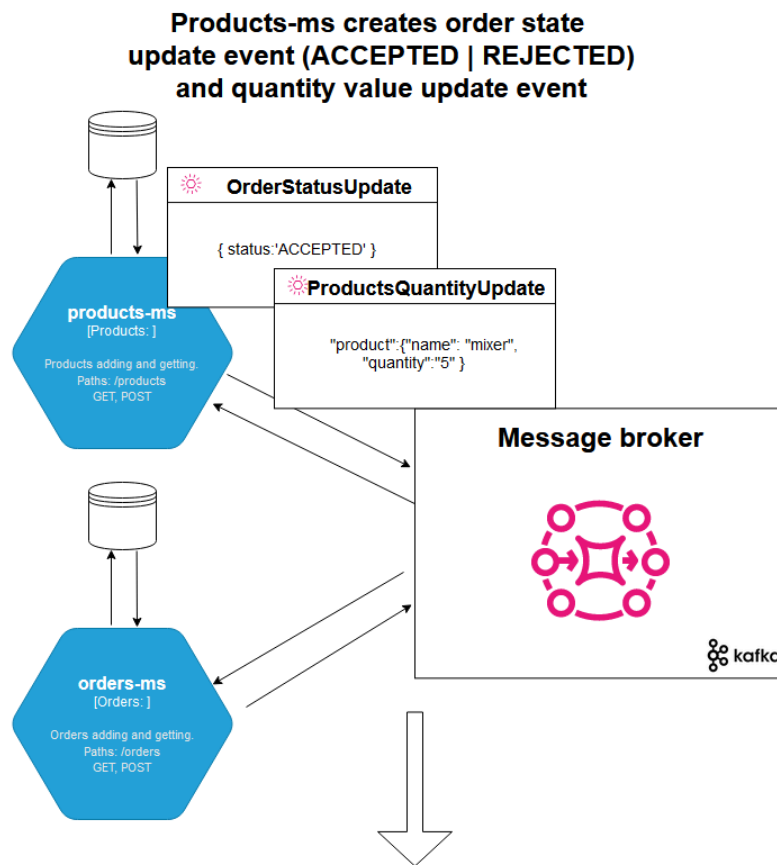
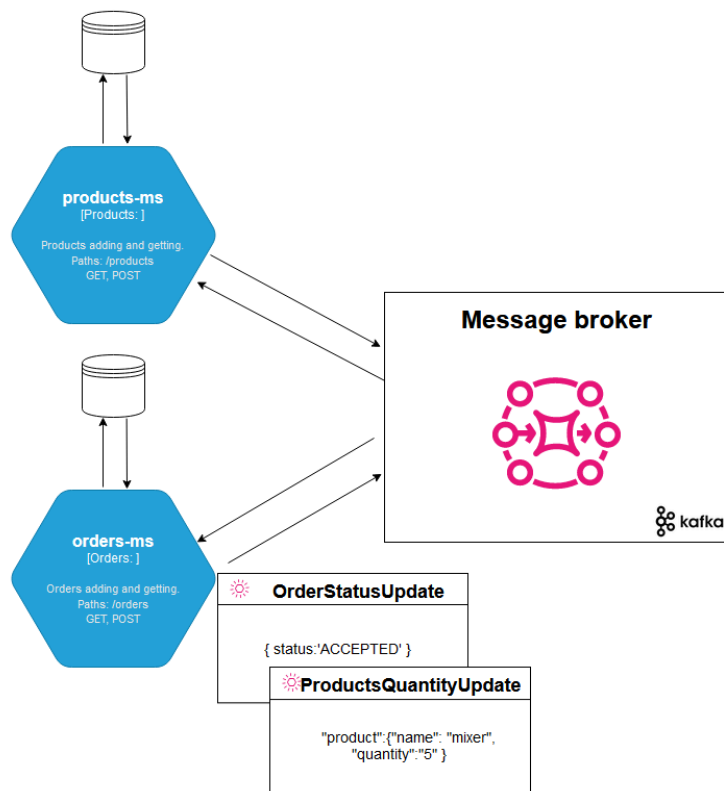


Figure 22 - Order placing by a client. OrderCreateEvent is created



**Orders-ms changes the status of the order from PENDING to ACCEPTED/REJECTED and updates quantity in local 'products' collection**



*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)
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: zookeeper-dep
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: zookeeper
10   template:
11     metadata:
12       labels:
13         app: zookeeper
14     spec:
15       containers:
16         - name: zookeeper
17           image: confluentinc/cp-zookeeper:7.3.2
18           ports:
19             - containerPort: 2181
20           env:
21             - name: ZOOKEEPER_CLIENT_PORT
22               value: "2181"
```

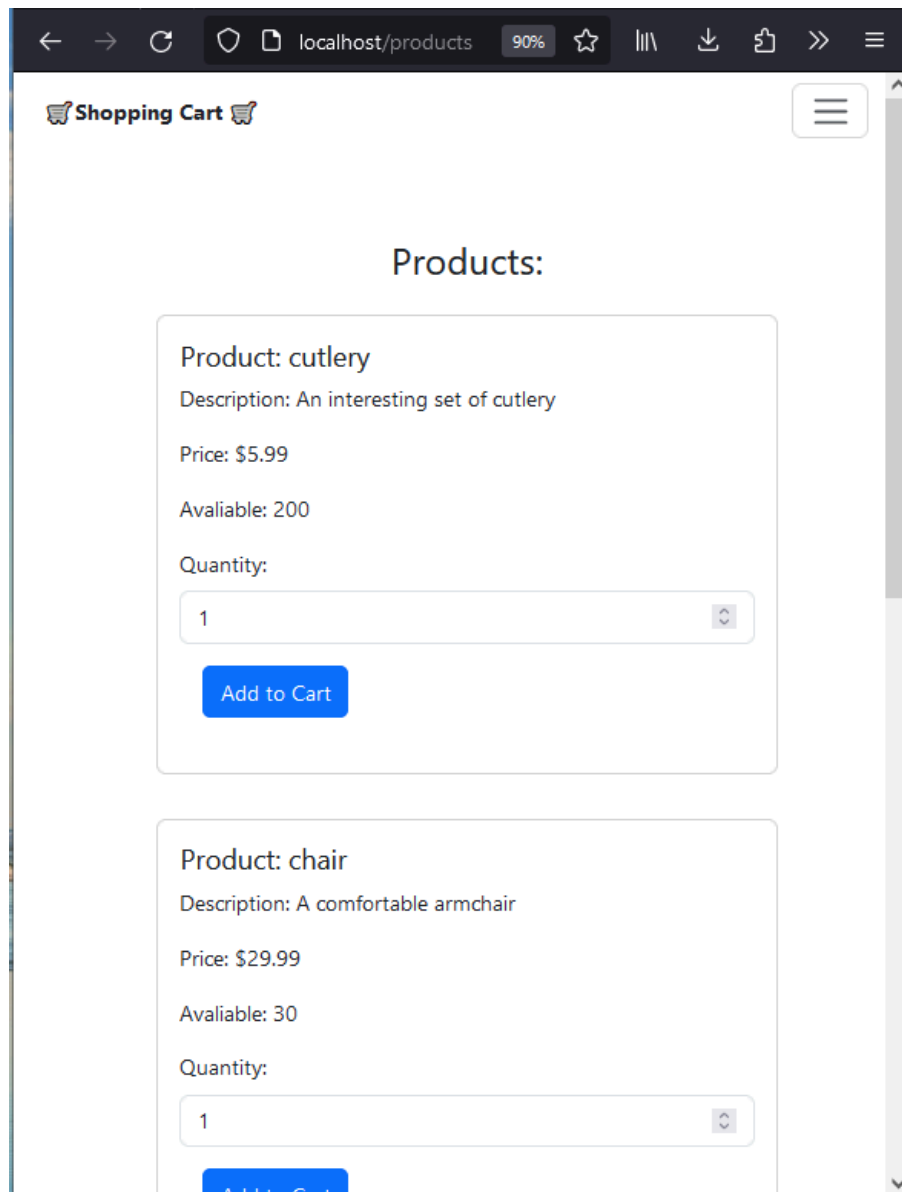
Figure 24 - Zookeeper K8S production yaml file

```
io.k8s.api.core.v1.Service (v1@service.json) | io.k8s.api.apps.v1.Deployment (v1@deployment.json)
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: message-broker-dep
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: message-broker
10   template:
11     metadata:
12       labels:
13         app: message-broker
14     spec:
15       containers:
16         - name: message-broker
17           image: confluentinc/cp-kafka:7.3.2
18           imagePullPolicy: "Always"
19           ports:
20             - containerPort: 9092
21               name: kafka
22             - 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 > ...
1  FROM node:20-alpine as builder
2  WORKDIR /app
3  COPY . .
4  ARG VITE_APP_BACKEND_ADDRESS
5  ENV VITE_APP_BACKEND_ADDRESS $VITE_APP_BACKEND_ADDRESS
6  RUN npm install
7  RUN npm run build
8
9  FROM nginx:1.25.4-alpine-slim as prod
10 COPY --from=builder /app/dist /usr/share/nginx/html
11 COPY nginx.conf /etc/nginx/conf.d
12 RUN rm /etc/nginx/conf.d/default.conf
13 EXPOSE 3000
14 CMD ["nginx", "-g", "daemon off;"]
15

```

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
3  }
4
5  http {
6  server {
7      listen 80;
8      server_name example.com;
9
10     # Frontend configuration
11     location / {
12         proxy_pass http://frontend:3000;
13         proxy_set_header Host $host;
14         proxy_set_header X-Real-IP $remote_addr;
15         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
16         proxy_set_header X-Forwarded-Proto $scheme;
17     }
18
19     # Backend configuration
20     location /api {
21         proxy_pass http://gateway-ms:8000;
22         proxy_set_header Host $host;
23         proxy_set_header X-Real-IP $remote_addr;
24         proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
25         proxy_set_header X-Forwarded-Proto $scheme;
26     }
27 }

```

Figure 28 - Reverse proxy - nginx.conf

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

```
infrastructure > kubernetes > gcp > ! ingress.yaml > {} spec > [ ] rules >
19  apiVersion: v1
20  kind: Secret
21  metadata:
22    name: nginx-tls-secret
23    namespace: default
24    type: kubernetes.io/tls
25  data:
26    tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB0tLS0tCk1
27    tls.key: LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1
28  ---
29  apiVersion: networking.k8s.io/v1
30  kind: Ingress
31  metadata:
32    name: nginx-ing
33    namespace: default
34    annotations:
35      kubernetes.io/ingress.class: nginx
36  spec:
37    tls:
38      - hosts:
39        - cloudcomputingtechnologies.pl
40        secretName: nginx-tls-secret
41  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.

```
158 resource "google_container_node_pool" "spot" {
159   name      = "spot"
160   cluster   = google_container_cluster.primary.id
161
162   management {
163     auto_repair = true
164     auto_upgrade = true
165   }
166
167   autoscaling {
168     min_node_count = 0
169     max_node_count = 10
170   }
171
172   node_config {
173     preemptible = true
174     machine_type = "e2-medium"
175
176     labels = {
177       team = "devops"
178     }
179   }
180 }
```

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.

The screenshot shows the CircleCI interface for user 'heyimjustalex'. The left sidebar contains navigation links: Dashboard, Projects (with a 'FOLLOW 9+ NEW' button), Releases (with a 'NEW' button), Insights, Self-Hosted Runners, Organization Settings, and Plan (with an 'UPGRADE' button). A notification box at the top of the main area states: 'Runner Launch Agent 1.x EOL We will be ending support for runner launch agent 1.1 on July 31st 2024. Learn more about this change and migrating to Runner 3.0.' Below this, filters are set to 'Everyone's Pipelines', 'All Projects', 'Select a Branch', and 'All days'. The main table lists pipelines for the 'e-commerce-microservice' (s 159). All listed pipelines are in a 'Success' state. The workflows include frontend-deploy, products-ms-deploy, proxy-deploy, ingress-deploy, authentication-ms-deploy, gateway-ms-deploy, databases-deploy, and orders-ms-deploy. Each pipeline was triggered by a 'main' branch event from user '18a11a1' updating 'config.yml' 2 hours ago. The durations and percentage changes are: frontend-deploy (2m 4s, +3%), products-ms-deploy (3m 38s, +22%), proxy-deploy (53s, -12%), ingress-deploy (1m 11s, +9%), authentication-ms-deploy (3m 7s, +8%), gateway-ms-deploy (1m 44s, +14%), databases-deploy (1m 34s, +2%), and orders-ms-deploy (3m 39s, +29%). Each row has icons for rerun, cancel, and delete.

Pipeline	Status	Workflow	Trigger Event	Start	Duration	Actions
e-commerce-microservice s 159	Success	frontend-deploy	main 18a11a1 Updated config.yml	2h ago	2m 4s <span>↑3%</span>	...
	Success	products-ms-deploy	main 18a11a1 Updated config.yml	2h ago	3m 38s <span>↑22%</span>	...
	Success	proxy-deploy	main 18a11a1 Updated config.yml	2h ago	53s <span>↓12%</span>	...
	Success	ingress-deploy	main 18a11a1 Updated config.yml	2h ago	1m 11s <span>↑9%</span>	...
	Success	authentication-ms-deploy	main 18a11a1 Updated config.yml	2h ago	3m 7s <span>↑8%</span>	...
	Success	gateway-ms-deploy	main 18a11a1 Updated config.yml	2h ago	1m 44s <span>↑14%</span>	...
	Success	databases-deploy	main 18a11a1 Updated config.yml	2h ago	1m 34s <span>↑2%</span>	...
	Success	orders-ms-deploy	main 18a11a1 Updated config.yml	2h ago	3m 39s <span>↑29%</span>	...

Figure 31 - successful deployment

The screenshot shows the CircleCI configuration file 'config.yml' for the 'e-commerce-microservices' project on the 'main' branch. The file contains a 'run' step with a 'name' of 'Update Deployment Object' and a 'command' block. The commands in the block are: 'gcloud auth activate-service-account \$G\_CLOUD\_SERVICE\_ACCOUNT --key-file=\$(echo \$G\_CLOUD\_SERVICE\_KEY)', 'gcloud --quiet config set project \${GOOGLE\_PROJECT\_ID}', 'gcloud --quiet config set compute/zone \${GOOGLE\_COMPUTE\_ZONE}', 'gcloud components install --quiet kubectl', 'kubectl config set current-context "\${echo \$GOOGLE\_GKE\_CONTEXT}"', 'kubectl delete -f ./infrastructure/kubernetes/gcp/authentication.yaml && kubectl apply -f ./infrastructure/kubernetes/gcp/authentication.yaml', '# kubectl rollout restart deployment/authentication-ms-dep', and 'kubectl set image deployment/authentication-ms-dep authentication-ms=heyimjustalex/authentication-ms:\$CIRCLE\_SHA1'. The interface includes a left sidebar with navigation links, a top bar with 'Source', 'Compiled', and 'Save and Run' buttons, and a right sidebar with an 'Add Config Code' button.

```
56 gcloud auth activate-service-account $G_CLOUD_SERVICE_ACCOUNT --key-file=$(echo $G_CLOUD_SERVICE_KEY)
57 gcloud --quiet config set project ${GOOGLE_PROJECT_ID}
58 gcloud --quiet config set compute/zone ${GOOGLE_COMPUTE_ZONE}
59 gcloud components install --quiet kubectl
60
61 - run:
62   name: Update Deployment Object
63   command: |
64     gcloud container clusters get-credentials primary --zone "${echo $G_CLOUD_ZONE}" --project "${echo $GOOGLE_PROJECT_ID}"
65     kubectl config set current-context "${echo $GOOGLE_GKE_CONTEXT}"
66     kubectl delete -f ./infrastructure/kubernetes/gcp/authentication.yaml && kubectl apply -f ./infrastructure/kubernetes/gcp/authentication.yaml
67     # kubectl rollout restart deployment/authentication-ms-dep
68     kubectl set image deployment/authentication-ms-dep authentication-ms=heyimjustalex/authentication-ms:$CIRCLE_SHA1
69
```

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
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: orders-ms-dep
5  spec:
6    replicas: 2
7    selector:
8      matchLabels:
9        app: orders-ms
10   template:
11     metadata:
12       labels:
13         app: orders-ms
14     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.

```

19   apiVersion: v1
20   kind: Secret
21   metadata:
22     name: nginx-tls-secret
23     namespace: default
24   type: kubernetes.io/tls
25   data:
26     tls.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSB
27     tls.key: LS0tLS1CRUdJTiBQUk1WQVRFIETFWSE
28   ---
29   apiVersion: networking.k8s.io/v1
30   kind: Ingress
31   metadata:
32     name: nginx-ing
33     namespace: default
34     annotations:
35       kubernetes.io/ingress.class: nginx
36   spec:
37     tls:
38       - hosts:
39         - cloudcomputingtechnologies.pl
40       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](https://en.wikipedia.org/wiki/Non-functional_requirement)