1. Instructions for the teaching assistant

System is implemented as monolith application. There is only one build. The User select microservice by starting with correct parameter.

Parameter	location	Service
HttpServ	internal	Get log of all messages
ORIGMain	internal	Server which sends messages
OBSEMain	internal	Obser messages and save them to database
IMEDMain	internal	Message forwarder
ApiGateWay	external	System which exposes internal service to world
Monitor	internal	Service which tells if all systems are running.

More granular configuration is done with environment variable parameters:

environment variable	info
queueCreated	Boolean, if false queue is created in startup of service.
mainqueue2	Queue compse140.i
exchanges	compse140.o-ex
fanoutqueue1	Queue compse140.o
fanoutqueue1	Queue compse140.o-2
rabbitmq	Rabbit MQ server connection string
rabbitmqUrl	www site url
rabbitmqToken	Authentication token
mongoDbURL	Database url connection string
dbName	Database name
Port	Service port
httpServ,	Service url for monitor service and Api gateway service.
httpOrigv	Service url for monitor service and Api gateway service.
httpObse,	Service url for monitor service and Api gateway service.
httpImed	Service url for monitor service and Api gateway service.
httServiceStats	Service url for monitor service and Api gateway service.

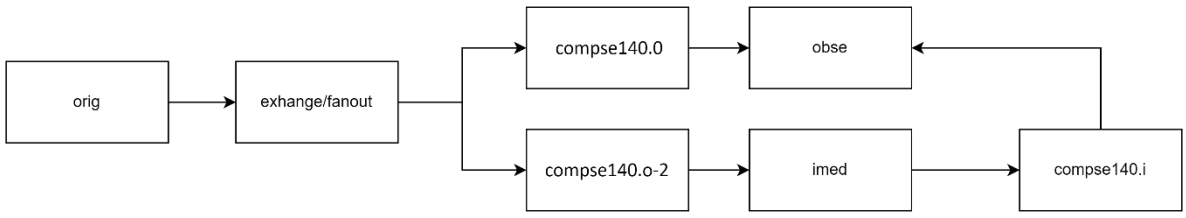


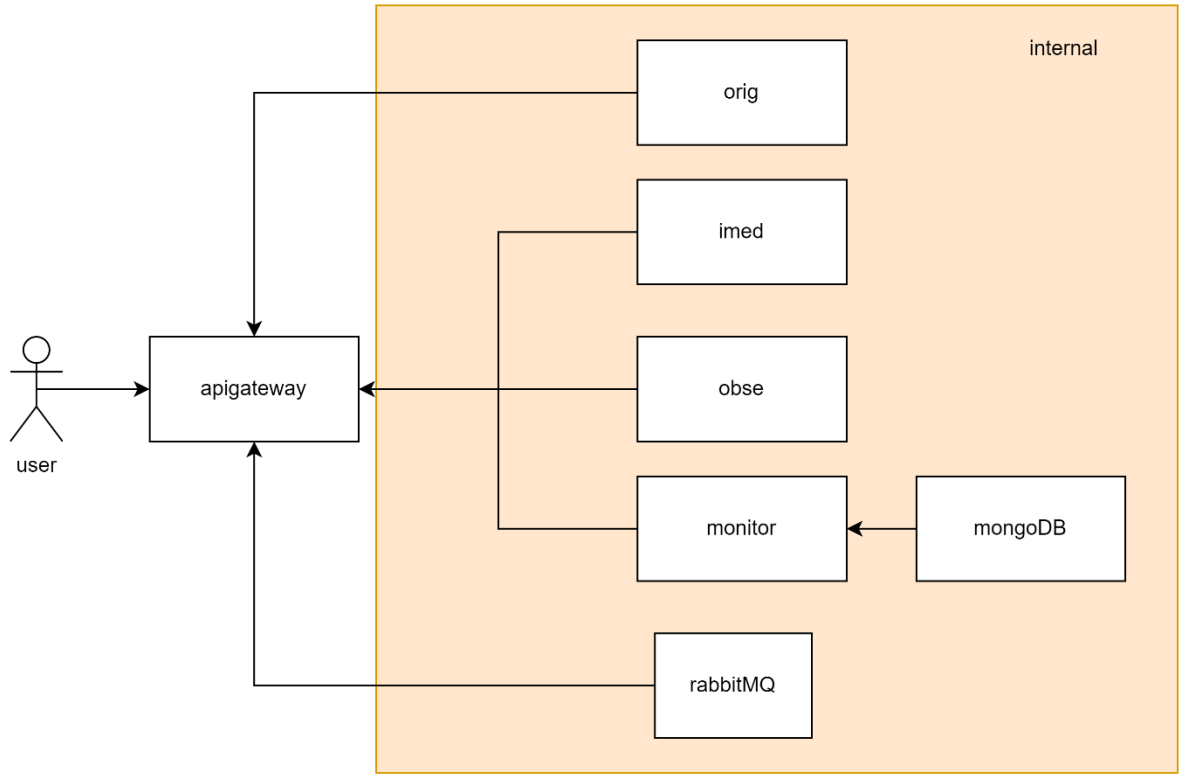
Figure 1 messaging structure.

Orig service send message into exchange queue which fanout messages into compse140.0 and compse140.0-2 queue.

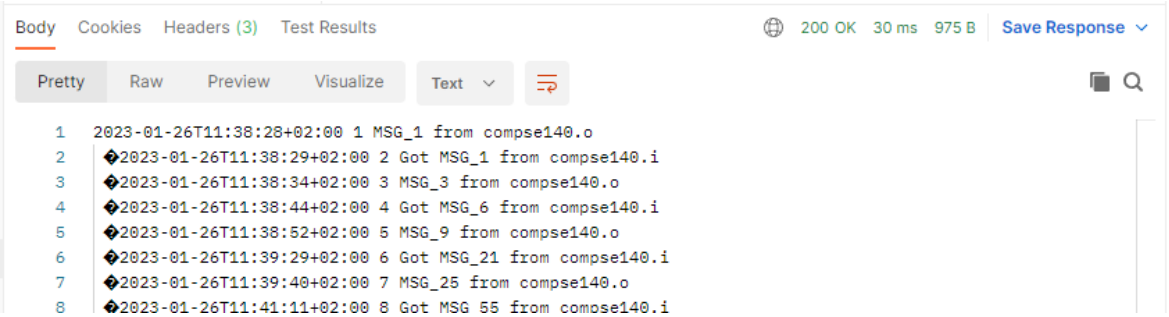
Each service has ping api for testing if service is running.

RabbitMq is used as message broker. MongoDB is used as databases.

Each microservice is behind API gateway.



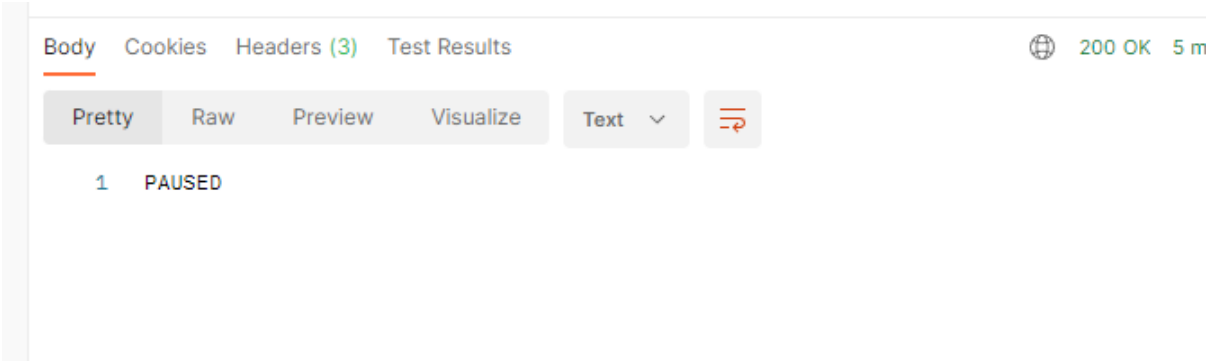
Apigateway run in port 8083
/message



/state

Empty response with http code 200

With http POST method:

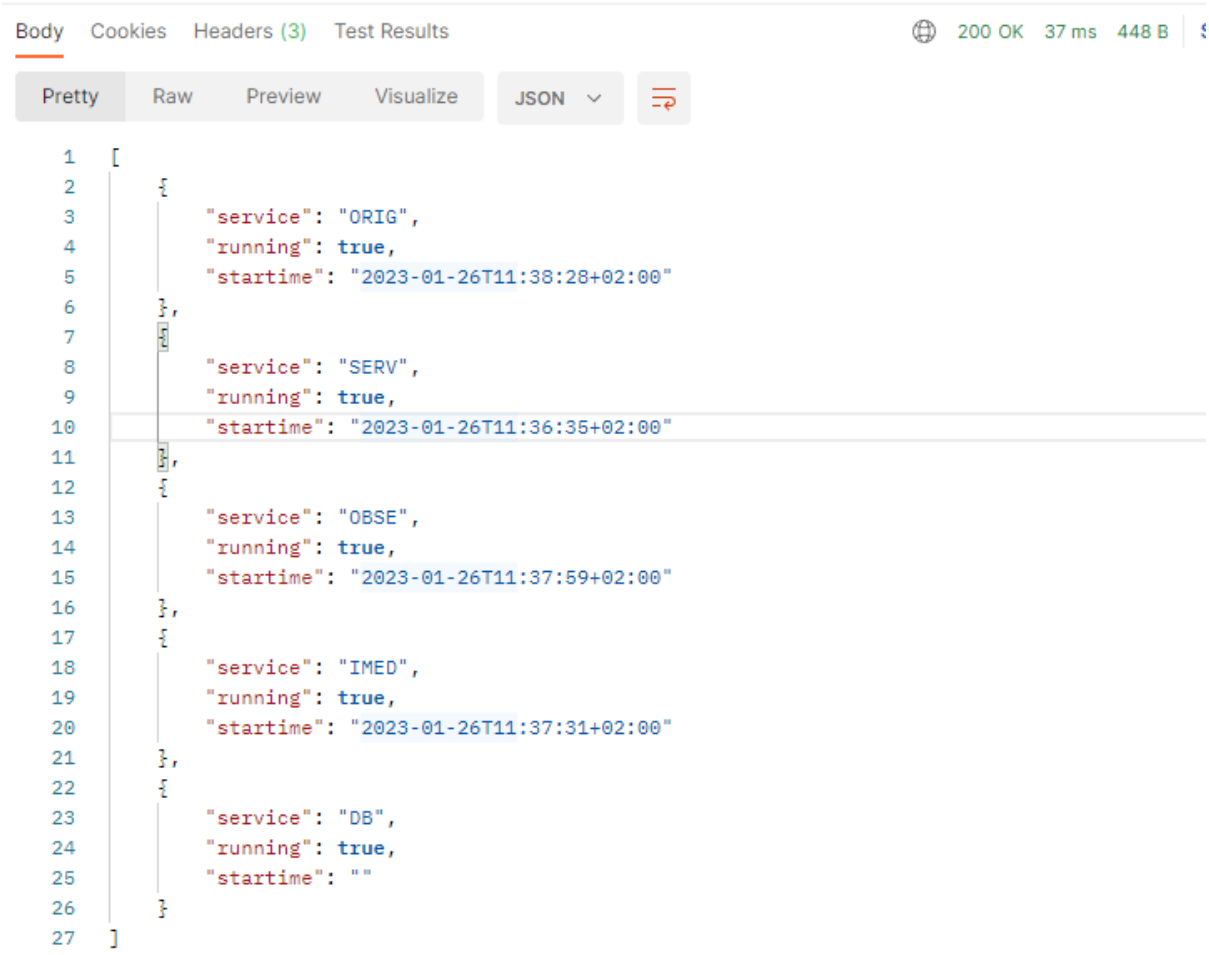


/node-statistic

/queue-statistic


```
[
    {
        "Name": "compse140.i",
        "delivery_rate": 0,
        "messages_delivered": 0,
        "messages_delivered_recently": 31,
        "messages_published": 0,
        "messages_published_recently": 31
    },
    {
        "Name": "compse140.o",
        "delivery_rate": 0,
        "messages_delivered": 0,
        "messages_delivered_recently": 498,
        "messages_published": 0,
        "messages_published_recently": 498
    },
    {
        "Name": "compse140.o-2",
        "delivery_rate": 0,
        "messages_delivered": 0,
        "messages_delivered_recently": 498,
        "messages_published": 0,
        "messages_published_recently": 498
    }
]
```

/serviceRunning



Implemented optional features

List of optional features implemented.

GET /node-statistic	implemented
GET /queue-statistic,	implemented
implement a static analysis step in the pipeline by using tools like jlint, pylint or SonarQube.	Not implemented
deployment to an external cloud (Ansible exercise, Heroku or similar)	Not implemented
Testing of individual components	Not implemented
monitoring and logging for troubleshooting	Partially implemented, User can monitor systems using /serviceRunning

Instructions for examiner to test the system.

Pay attention to optional features.

Start systems:

docker-compose up -d

Stop service:

docker-compose down

2. Description of the CI/CD pipeline

Git work as VCS. Project code is in project branch.

Code is build by using Go own build tools.

Application is built as docker image. Customization is done by using environmental variable.

Pipeline build docker image.

Testing is done in integration level by using integrationTest.sh

3. Example runs of the pipeline

4. Reflections Main learnings and worst difficulties

Generally, exercise was okey. Good way to learn GO language. Basic principles were not hard. Biggest obstacle were more Sysadmin thing when trying to set up Gitlab.

Amount effort (hours) used

60h maybe