Distributed Software Architecture Lab 3 Report

# GOALS

* Orchestrate multiple microservices with Docker Compose
* MongoDB basics
* Reliability concepts: Retry & Circuit-Breaker
* Availability: Data Caching & Request Queuing

# SOURCE CODE

<https://github.com/tvph1996/third_lab>

# SYSTEM SETUP

## Supports 4 methods

* AddItem (POST): Duplicate 'id' and 'name' is not allowed
* GetItem (GET): Search by 'id' gets single item, search by 'name' gets item stream (wrap around)
* UpdateItem (PUT): Change name (no duplicate) of an item by 'id'
* DeleteItem (DELETE): Remove item by 'id'

## REST-service

* Provides human response to CURL request
* Uses FastAPI, continue from Lab 2
* Calls gRPC-service using Environment Variable in Docker Compose
* Retry & Circuit-Breaker
  + Only in AddItem method
  + Design as the lab guideline with the only differences:
    - 2 retries after 1 sec & 2 sec to avoid IO display issue with minimal time
    - Shorten reset timeout to 6 sec for easier testing
* Supports: Request Queuing when Circuit-Breaker is opened
  + When gRPC-service or MongoDB down, after 2 times Retry, requests are put in a Queue and will be processed when services are healthy again.
    - Checking health of gRPC & MongoDB-service is done by a process sending dummy request periodically in the background every 15 seconds.

## gRPC-service

* Rule check for 4 methods
* Calls MongoDB using Environment Variable in Docker Compose
* Data Caching of MongoDB using python dictionary for AddItem & GetItem
  + Whenever an item is added, the same item will be cached
  + GetItem will attempt to search in Cache first (to reduce call to MongoDB)
    - If there is no result in Cache (gRPC-service is restarted), then query MongoDB

## MongoDB

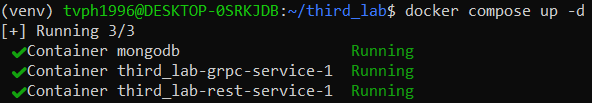
* Data is stored locally using Docker Volume
  + Configured in Docker Compose

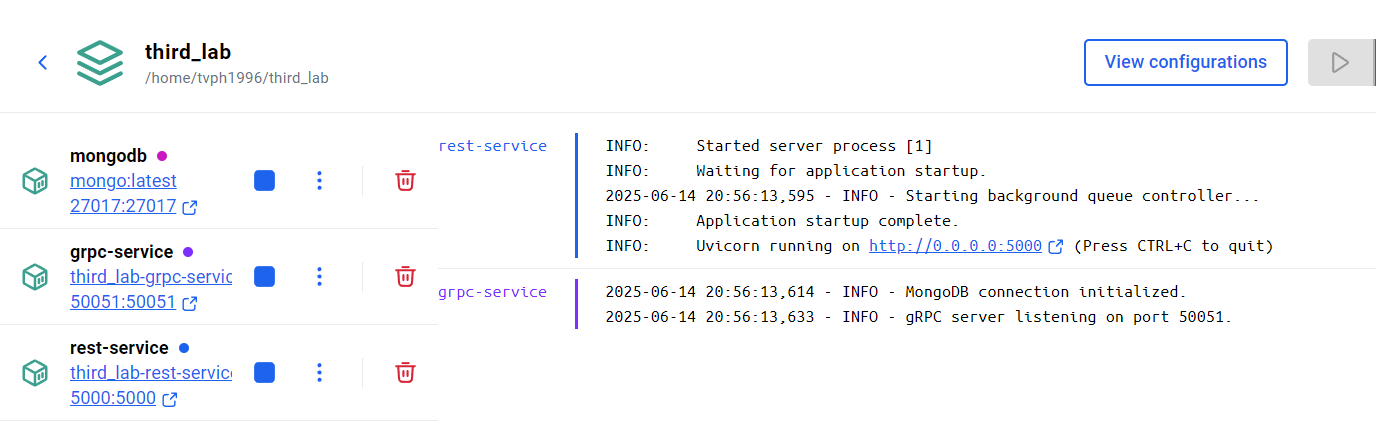
## OTHER NOTES

* Request should be in correct format, no checking is implemented
* Proto file is simplified by reusing message

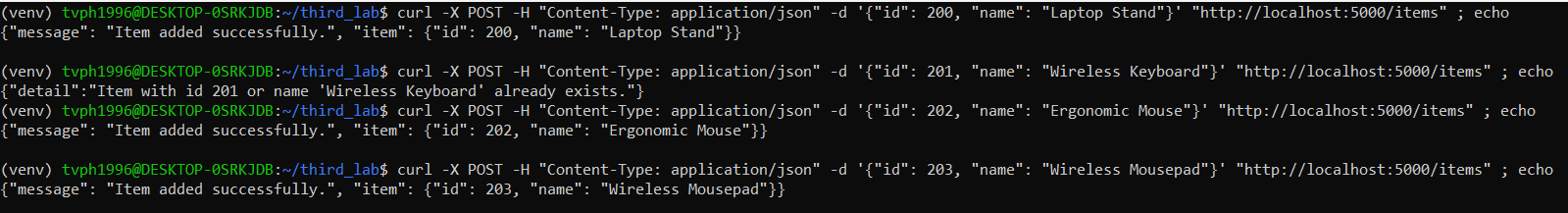
# OUTPUT

## Run by Docker Compose

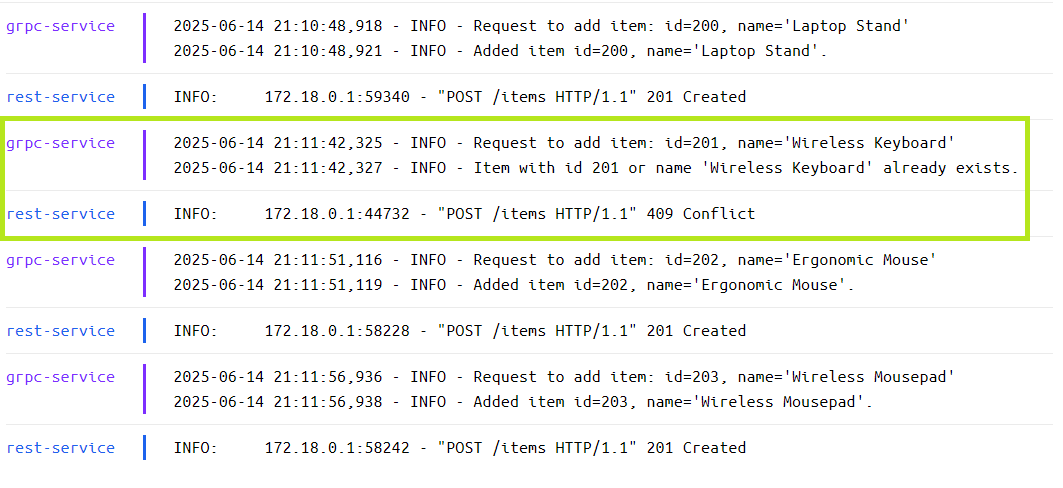




## Add items



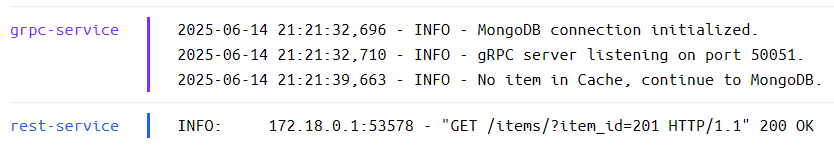
One item is duplicate here



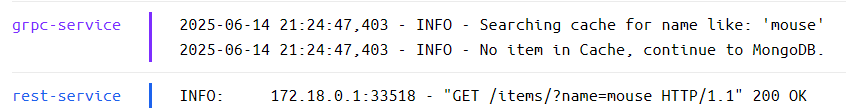
## GetItem

### Search in Cache first

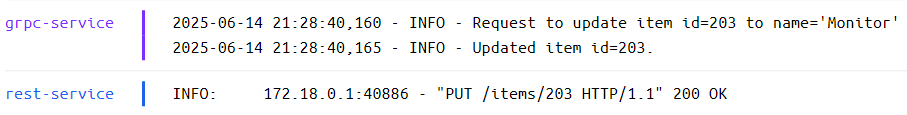
### Restart gRPC-service -> Cache lost -> Query MongoDB



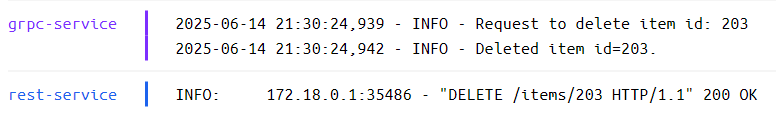
### Search by name -> get multiple results



## UpdateItem



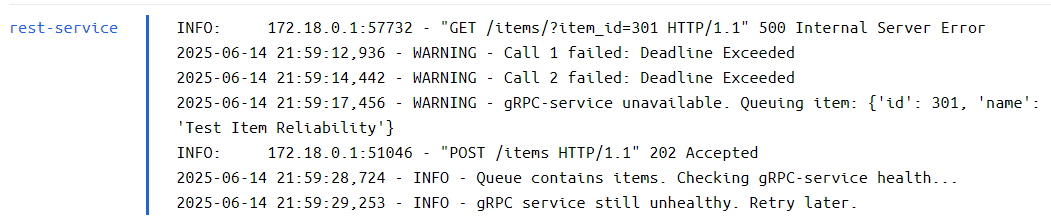
## DeleteItem



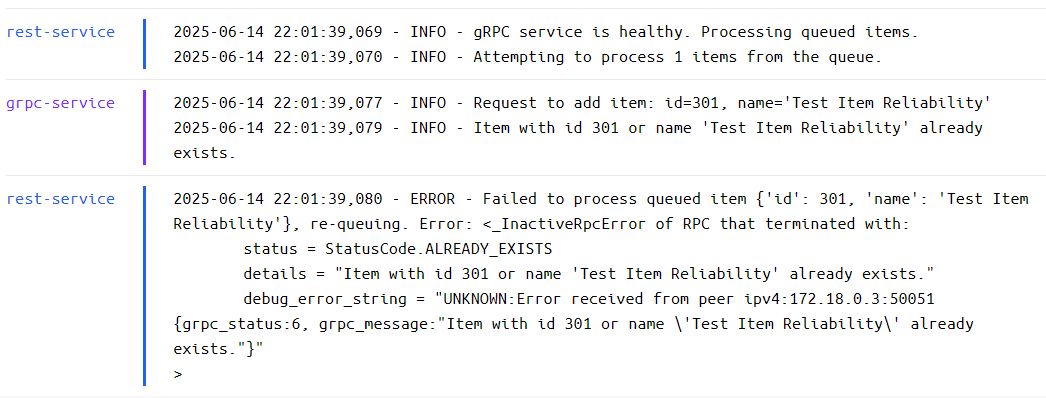
## Retry & Circuit-Breaker with Request Queuing’

### gRPC-service down

REST-service doing 2 times Retry and Circuit-Breaker which adds calls into Queue



After gRPC-service starts again, queued command is executed (this item is duplicate in this test case)



### MongoDB down

Retry & Circuit-Breaker work similarly like when gRPC-service is down.

In the screenshot we can see connection to MongoDB lost



# REFLECTION

1. When using ''docker stop'' on gRPC-service to simulate Reliability mechanism, the gRPC channel inside REST-service will lose its connection because gRPC channel have different internal logic to maintain its connection that very difficult for user to control.

This can be fixed by

1. Creating a new gRPC Channel for each call (implemented). This works because CircuitBreaker from pybreaker is a separate object running parallel and is mapped with gRPC Stub call

1. Using ''docker pause'' instead, but this is not useful in real life scenario.
2. To avoid the overhead of creating gRPC Channels in solution a, the only solution is to have multiple gRPC-service and create the logic to monitor and switch to channels from alive services.
3. MongoDB saves data to an internal folder inside the container /data/db. If we ‘docker stop/start’, the data is not lost. However, if we ‘docker down/up’ which is real life scenario when Docker Engine is down, the data is lost.

The solution is to using Docker Volume to store the data locally on Docker host PC.

Data from MongoDB can not be read in gRPC-service (client), which is why Data Caching is required. Maybe other database can, then we only need to save data using Docker Volume and share it between the containers.

1. For this only simple system, it is already very hard to find the route of 1 call from begin to end.

Dedicated solution is preferred.

1. For the purpose of learning, here AP is implemented by using mechanism like Data Caching or Request Queuing.

Decide between CP & AP depends on how consistent the data must be. For important financial or government data, CP is more suitable. For nice-to-have data, AP fits better.