CASE STUDY

Problem Statement:

We're building an application to maintain a product catalog that offers a REST API.

We should be able to perform CRUD operations on the product

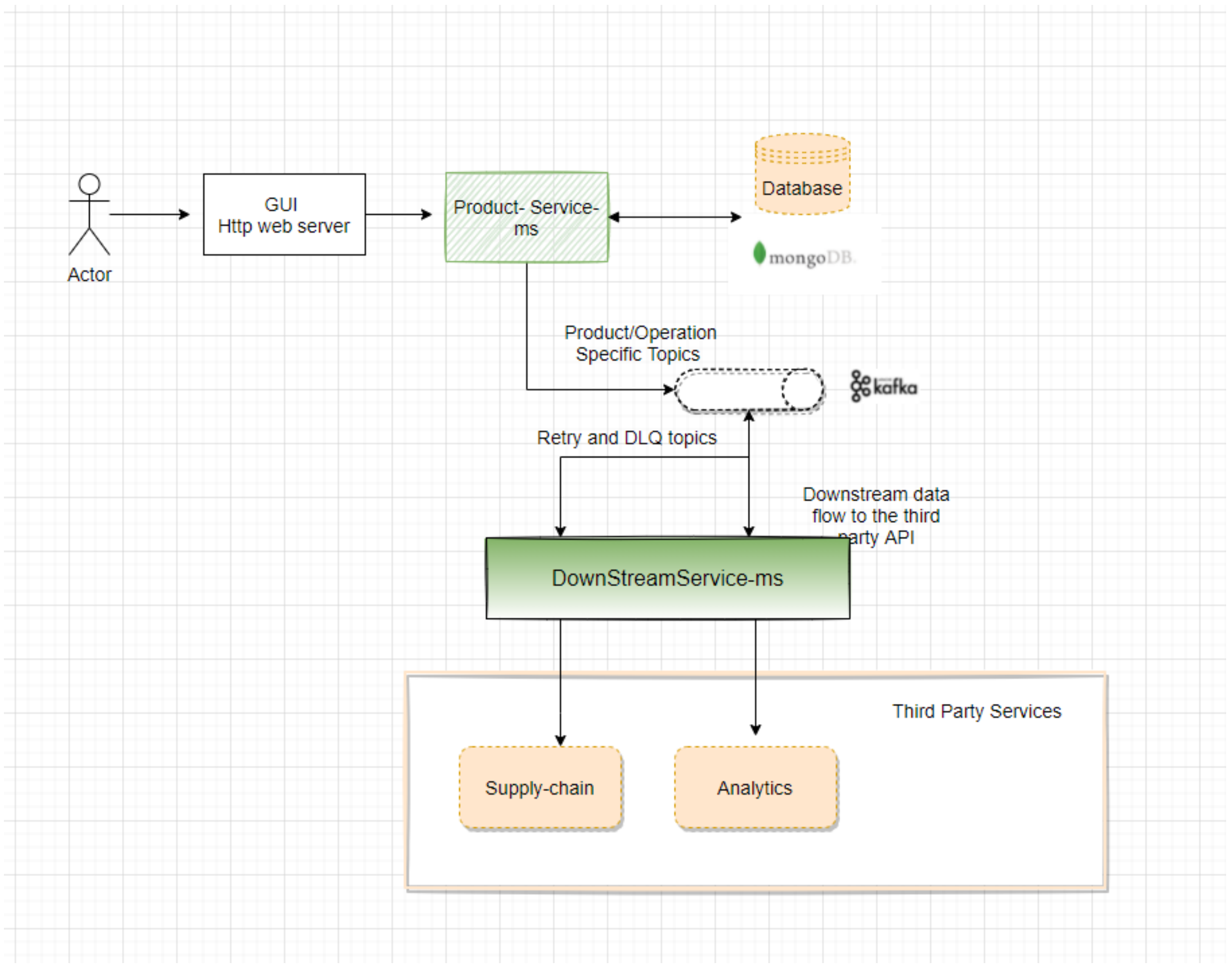
Changes to the catalog need to be propagated to 3rd party downstream services via a REST interface.

For example: Consumer interest on specific products or usage on a specific date. These info should be communicated to the third party API for analytics. OR sale related supply chain account management, inventory etc :

The partner's supply chain APIs are unreliable, and can return unexpected random errors, hence our application needs to be fault tolerant. In case the API is down or returns an error, the system should ensure the message is properly processed once it recovers.

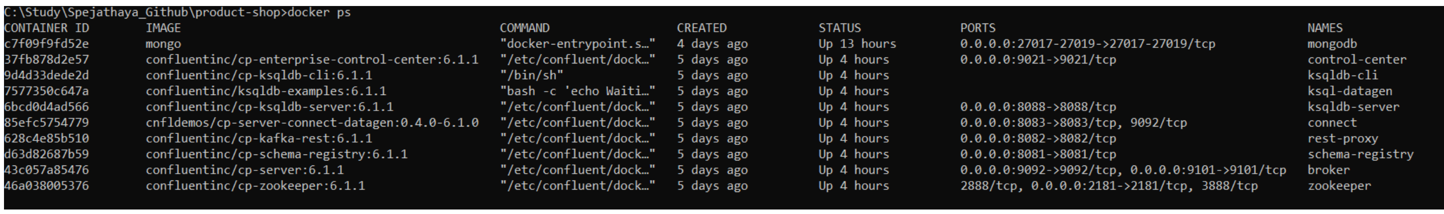
CASE STUDY

How the solution works :



Prerequisites to set up locally:

1. Web server
2. Java 11+ /Docker
3. Kafka v2.13-2.8.0 +ZK – port: 9092
4. MongoDB v4.4.6 port: 27017

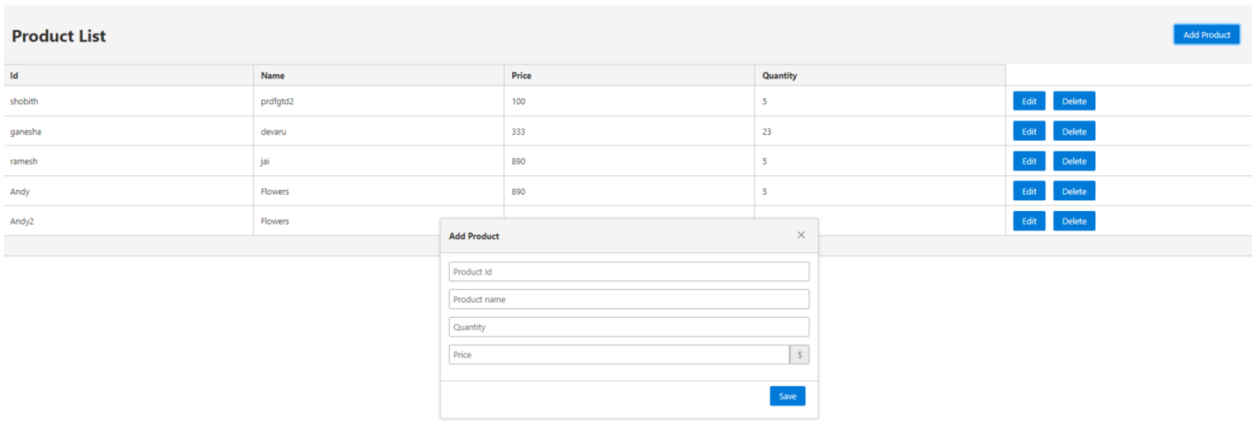


```
C:\Study\Spejathaya_Github\product-shop>docker ps
CONTAINER ID   IMAGE                                          COMMAND                  CREATED      STATUS         PORTS                                                NAMES
c7f09f9fd52e   mongo                                          "docker-entrypoint.s…"   4 days ago   Up 13 hours    0.0.0.0:27017-27019->27017-27019/tcp                 mongodb
37fb878d2e57   confluentinc/cp-enterprise-control-center:6.1.1  "/etc/confluent/dock…"   5 days ago   Up 4 hours     0.0.0.0:9021->9021/tcp                               control-center
9d4d33dede2d   confluentinc/cp-ksqldb-cli:6.1.1               "/bin/sh"                5 days ago   Up 4 hours                                                          ksqldb-cli
7577350c647a   confluentinc/ksqldb-examples:6.1.1             "bash -c 'echo Waiti…"   5 days ago   Up 4 hours                                                          ksql-datagen
6bcd0d4ad566   confluentinc/cp-ksqldb-server:6.1.1            "/etc/confluent/dock…"   5 days ago   Up 4 hours     0.0.0.0:8088->8088/tcp                               ksqldb-server
85efc5754779   cnfldemos/cp-server-connect-datagen:0.4.0-6.1.0  "/etc/confluent/dock…"   5 days ago   Up 4 hours     0.0.0.0:8083->8083/tcp, 9092/tcp                     connect
628c4e85b510   confluentinc/cp-kafka-rest:6.1.1               "/etc/confluent/dock…"   5 days ago   Up 4 hours     0.0.0.0:8082->8082/tcp                               rest-proxy
d63d82687b59   confluentinc/cp-schema-registry:6.1.1          "/etc/confluent/dock…"   5 days ago   Up 4 hours     0.0.0.0:8081->8081/tcp                               schema-registry
43c057a85476   confluentinc/cp-server:6.1.1                   "/etc/confluent/dock…"   5 days ago   Up 4 hours     0.0.0.0:9092->9092/tcp, 0.0.0.0:9101->9101/tcp       broker
46a038005376   confluentinc/cp-zookeeper:6.1.1                "/etc/confluent/dock…"   5 days ago   Up 4 hours     2888/tcp, 0.0.0.0:2181->2181/tcp, 3888/tcp           zookeeper
```

CASE STUDY

Components :

1. Web Application - product-shop-ui



Product List

Id	Name	Price	Quantity	
shobith	prdfgtd2	100	5	Edit Delete
ganesha	devaru	333	23	Edit Delete
ramesh	jai	890	5	Edit Delete
Andy	Flowers	890	5	Edit Delete
Andy2	Flowers			Edit Delete

Add Product

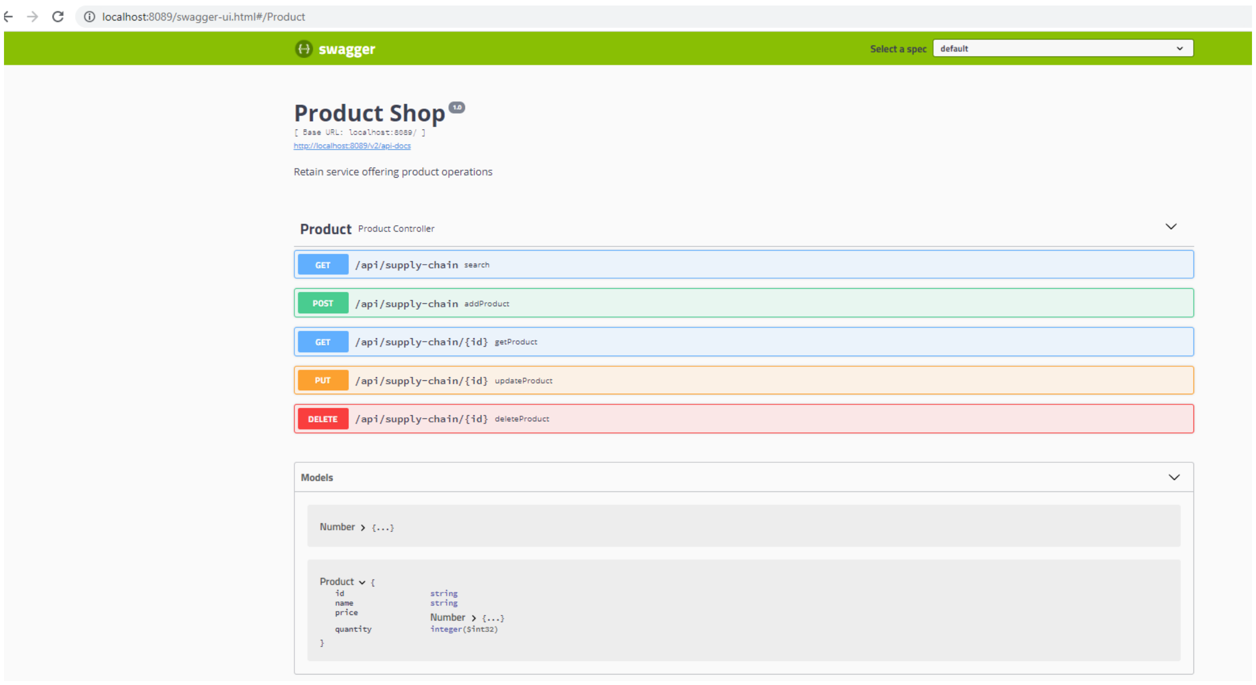
This is the launchpad application to perform the CRUD operations on the product. Developed in TypeScript - Angular - framework

CASE STUDY

2. **Product-Service-ms :** This is the microservice providing the CRUD operations running in port 8089.

   Provides the below feature:

   1. Offers API endpoints to perform CREATE | DELETE | UPDATE | FIND | SEARCH operations on the product data.
   2. MongoDB configured to persist the product data into the Document based database.
   3. Caching implemented to optimize the read operation.
   4. Kafka integration: Pushes the incoming operational data (CREATE | UPDATE | DELETE) into the Kafka broker – "product-operations" which can be read by the relevant services.



localhost:8089/swagger-ui.html#/Product

Product Shop 1.0

[Base URL: localhost:8089/]

http://localhost:8089/v2/api-docs

Retain service offering product operations

Product Product Controller

```
GET     /api/supply-chain  search
POST    /api/supply-chain  addProduct
GET     /api/supply-chain/{id}  getProduct
PUT     /api/supply-chain/{id}  updateProduct
DELETE  /api/supply-chain/{id}  deleteProduct
```

Models

```
Number > {...}

Product v {
   id                   string
   name                 string
   price                Number > {...}
   quantity             integer($int32)
}
```

3. **DownStreamService-ms**

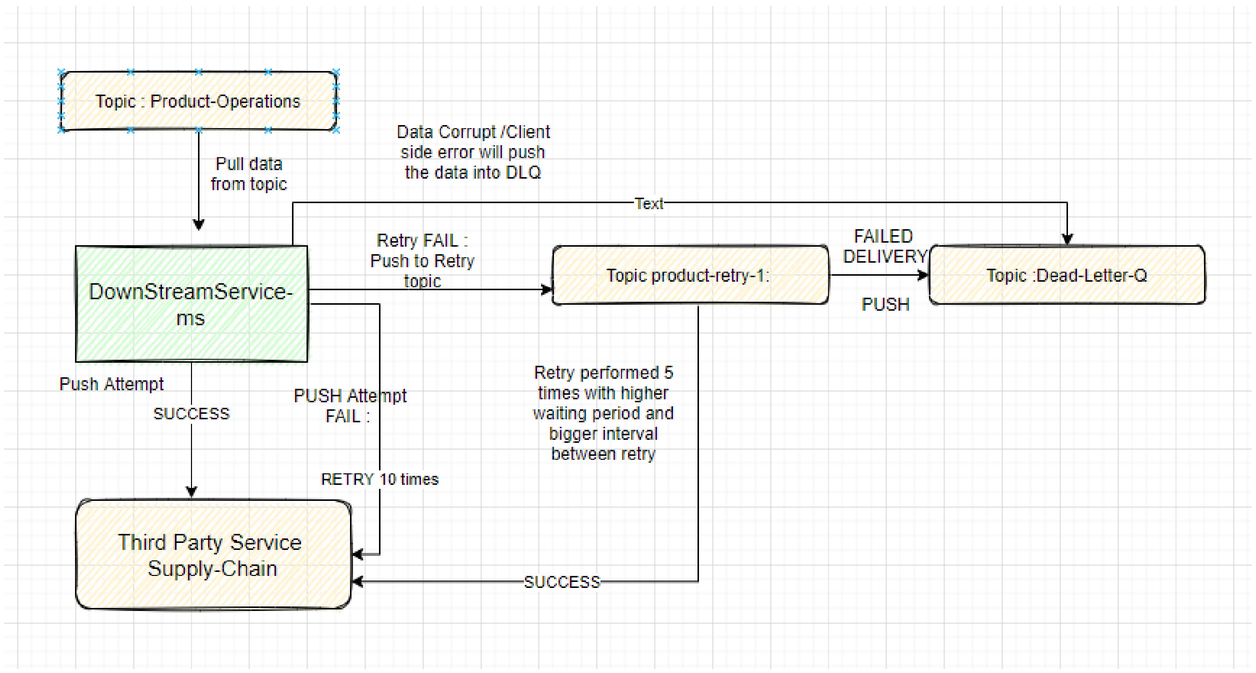
   This is a microservice handling messages from the kafka consumer.

   For the purpose of demonstration below assumptions are made :

   1. Any CREATE| UPDATE data is consumed by the service and pushed to the supply chain URL mock given https://ev5uwiczj6.execute-api.eu-central-1.amazonaws.com/test/supply-chain
   2. DELETE is pushed into a url which is not available.

CASE STUDY

Below diagram clearly represents the policy used t perform the third party integration


Topic : Product-Operations
Pull data from topic
Data Corrupt /Client side error will push the data into DLQ
Text
DownStreamService-ms
Retry FAIL : Push to Retry topic
Topic product-retry-1:
FAILED DELIVERY
PUSH
Topic :Dead-Letter-Q
Push Attempt
SUCCESS
PUSH Attempt FAIL :
Retry performed 5 times with higher waiting period and bigger interval between retry
RETRY 10 times
Third Party Service Supply-Chain
SUCCESS

1. The data is retried several times in the first attempt .to push into third party API
2. Failure will push the data into retry topic : Where it will be tried at a bigger duration between the push attempt (exponential increase in retry attempt). Here the failure is only considered if it's a server error . Client-side error is not considered for retry and data is directly pushed to DLQ
3. Still failure to deliver will push the data into DLQ . Which can be configured further based on the business needs.

CASE STUDY

How the Design Evolved:

Though it felt initially that the case study requires me to write a front end GUI which attempts to write data to downstream API's , I had several questions around the workflow.

Hence the purpose of the rest of the document is to convey the thought process involved in arriving at the solution and the considerations made. This is more of a design board for the product-shop which helped me to think through how I should proceed. I have jolted them down through the diagrams and explanations.

This is the high-level diagram on the expectations from the case study.

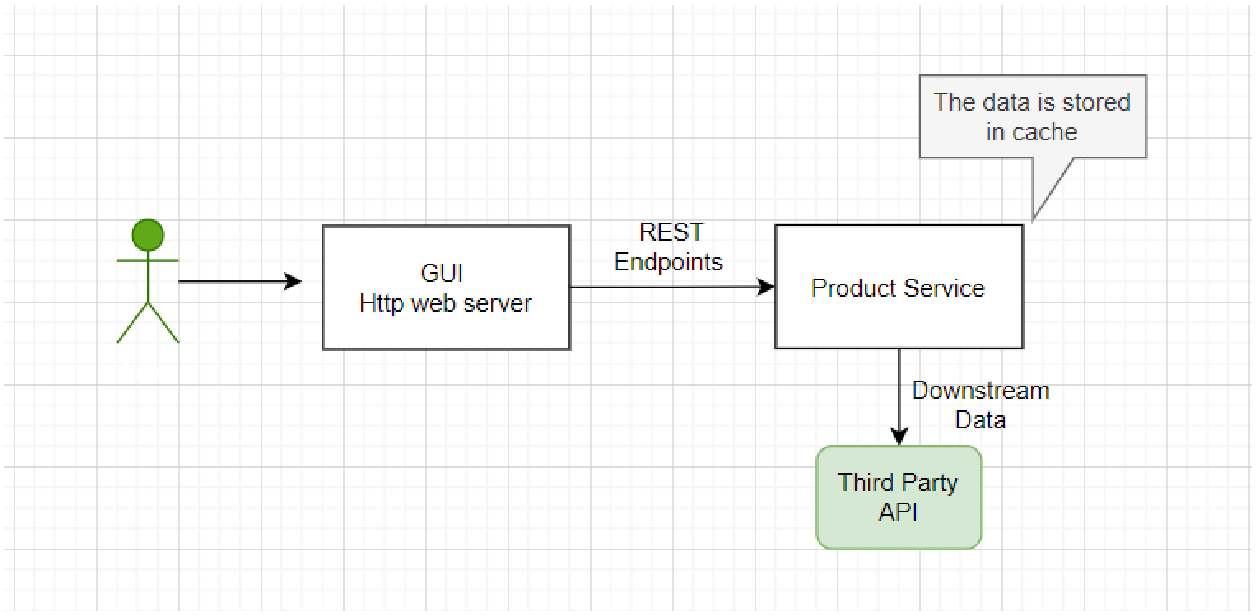


Diagram: 1

The web application performs an HTTP call as per the REST standards to the product service in order to achieve the GUI actions.

The product service stores the data in cache and also passes it along to a third-party service., say supply-chain, syncing or orders.

Drawbacks:

The above design has many flaws when it comes to **Scalability and performance** which are the focus areas of this case study

1. The product service is handling multiple responsibilities hence throttling the design and thus becoming a single point for failure.

2. Data is stored in cache. A volatile memory which might be lost in case of service crash, thus resulting in inconsistency across systems.
3. The application is not secure and real time monitoring of data and processing statistics is not possible
4. Bulk product Data import or migration is not possible.

Third Party Integration:

In the traditional approach, we usually place the third-party API integration logic in the main application codebase, but I feel it has the below challenges.

1. During peak hours, there will be a sudden network traffic on Product or purchase flow which will increase overall application latency and can eventually bring the system down
2. Third party API stability and availability is not consistent; there is no adequate retry mechanism available.
3. If there is an update to the third-party API integration mechanism or version, then it would require entire codebase to redesign and redeploy.

To avoid such issues, I preferred to decouple the third-party API integration from the main code base and handle it via a dedicated service operating on a message broker service.

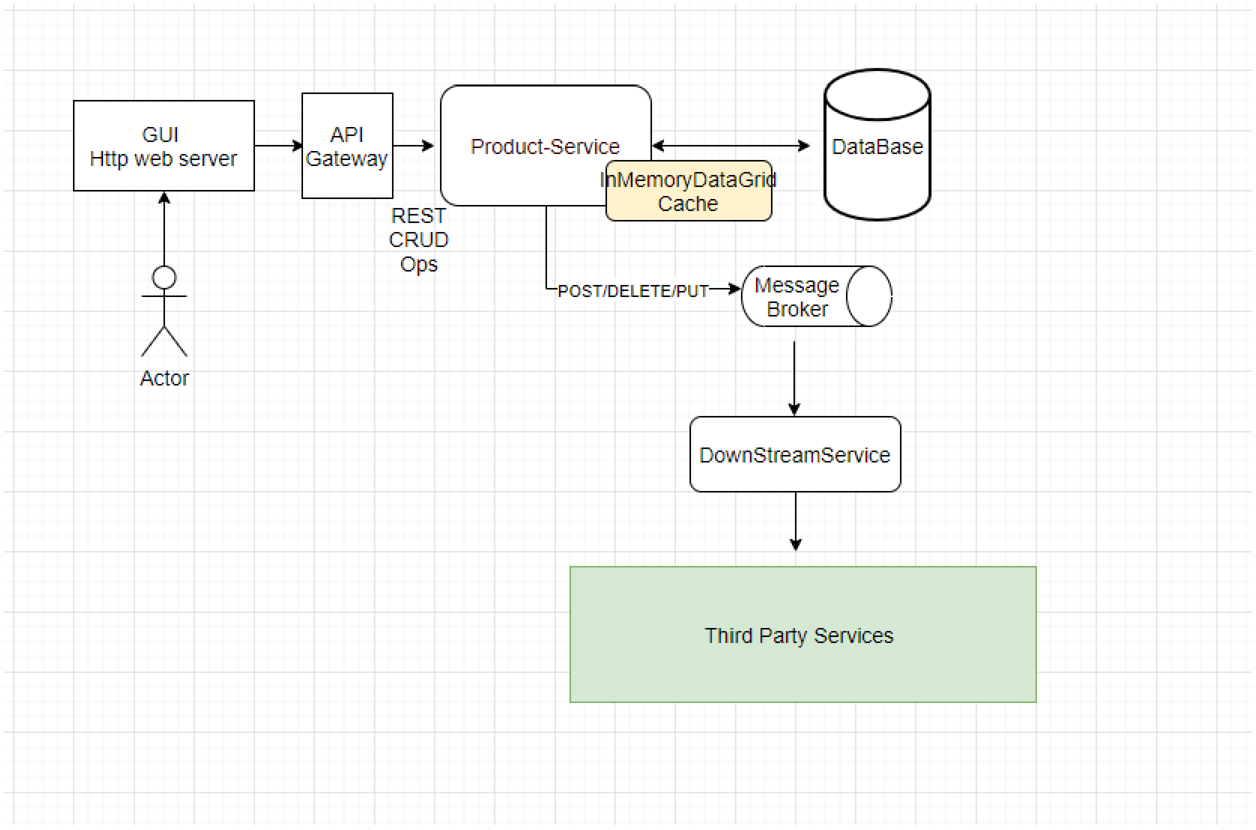


Diagram 2:

One of the drawbacks I can list out from the above design is "**DownStreamService**" can be overloaded and choked if there are multiple API's of third parties need to be integrated. Hence it would be better to have individual responsible services per API integration.

FaaS (ex: AWS Lambda) would better fit here !!

Tools and Services: With these considerations lets move on to tools evaluation

Cost, availability, learning curve, official and community support, Licensing would be some of the main criteria to consider. Product roadmap also plays a major part with the evaluation which is assumed to grow as a full-scale retail service in the future.

I have listed some tools I have come across which can fit the design. I have not really tried to do a comparison here as I know it's a subject very vast and opinionated. 😊

GUI	Angular application
Cloud Hosting infra	AWS
API Providers	Java /Spring based microservices. Java 11
In Memory Data Grid	Hazel Cast
Database	NoSQL database, MongoDB
Messaging platform	Kafka Streams
Cloud tools	Load balancer - Ribbon Service registry - Eureka Circuit breaker - Hystrix API gateway - Zuul
Service Monitoring	Spring Micrometer with Prometheus and Grafana
Security	OAuth 2.0 protected API's : Spring Security
Logging	Log4J with ELK configured
Downstream service	Serverless deployment such as AWS Lambda.
Deployment pipeline	Docker and Git pipeline

CASE STUDY

Detailed Design:

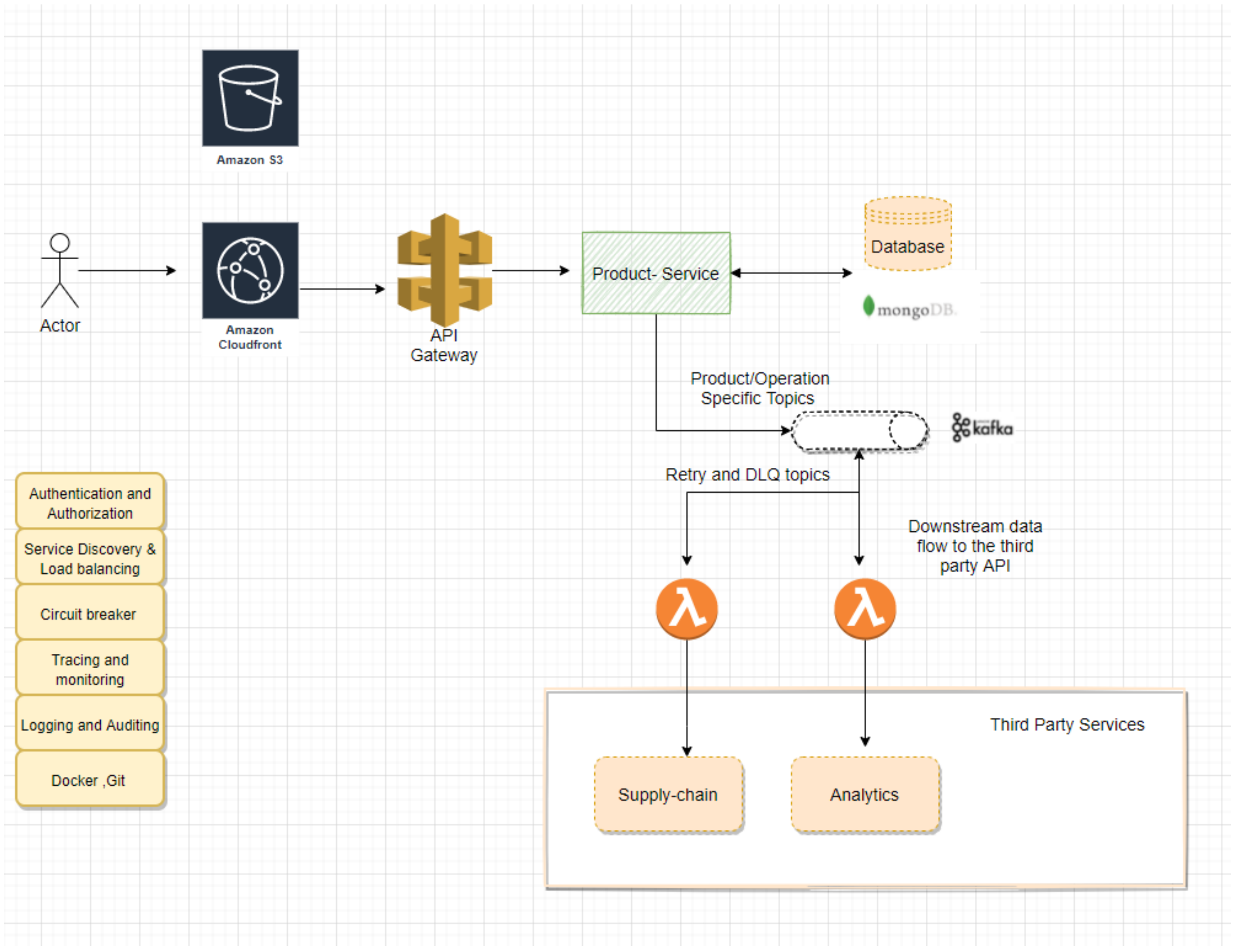


Diagram3:

Overall workflow:

- To handle third party API downstream services, we've created an AWS Lambda to consume Product addition and updation data via the kafka streams.
- Lambda function establishes connection to the third party API and passes on the data. In case of error while transfer or unexpected error from the third party services it can hold the offset data and retry at a later time. This can be achieved via kafka retry and DLQ topic integration.
- It also work 2 ways , if the supply chain or Sync services from the third party needs to write a data into the product DB , it can utilize the kafka integration to publish the data.
- Third party authorization can be done via client credentials oauth2.0 spec as it's a service to service interaction. Federated authentication is another option if the whole eco system offers a SSO.

CASE STUDY

Problem that our architecture solves:

- The logic for consuming third-party API could be isolated from the main codebase. Hence individual load can be handled operation wise at optimum cost.
- AWS Lambda (or any serverless FaaS offering) can instantly scale up to a large number of parallel executions. Hence the overall solution is highly scalable.
- Gives flexibility of deploying the third-party API update independently. I assume fine-tuning of AWS Lambda could be done for high availability and fault tolerance.

The following section is my thoughts on further expanding/bettering the design. I probably wouldn't advocate it right away as ***"Premature optimization is the root of all evil -***
- DonaldKnuth" 😊

One problem I see from the above design is: It is not optimized for read. In a typical retail service scenario READ is more frequently performed than write. Even if its other way round we need to decouple the READ operation from WRITE so that system is not throttled during a sale day or peak hours. "

Detailed Design:

In ideal design, DB per microservice is expected or preferred. But it's very rarely we get the luxury to dedicate a DB per µS .Even more tough if there is a Relational DB involved.

In the below design we can have the writer and reader services per a logical group of µS and thus scale and optimize based on the user requests.

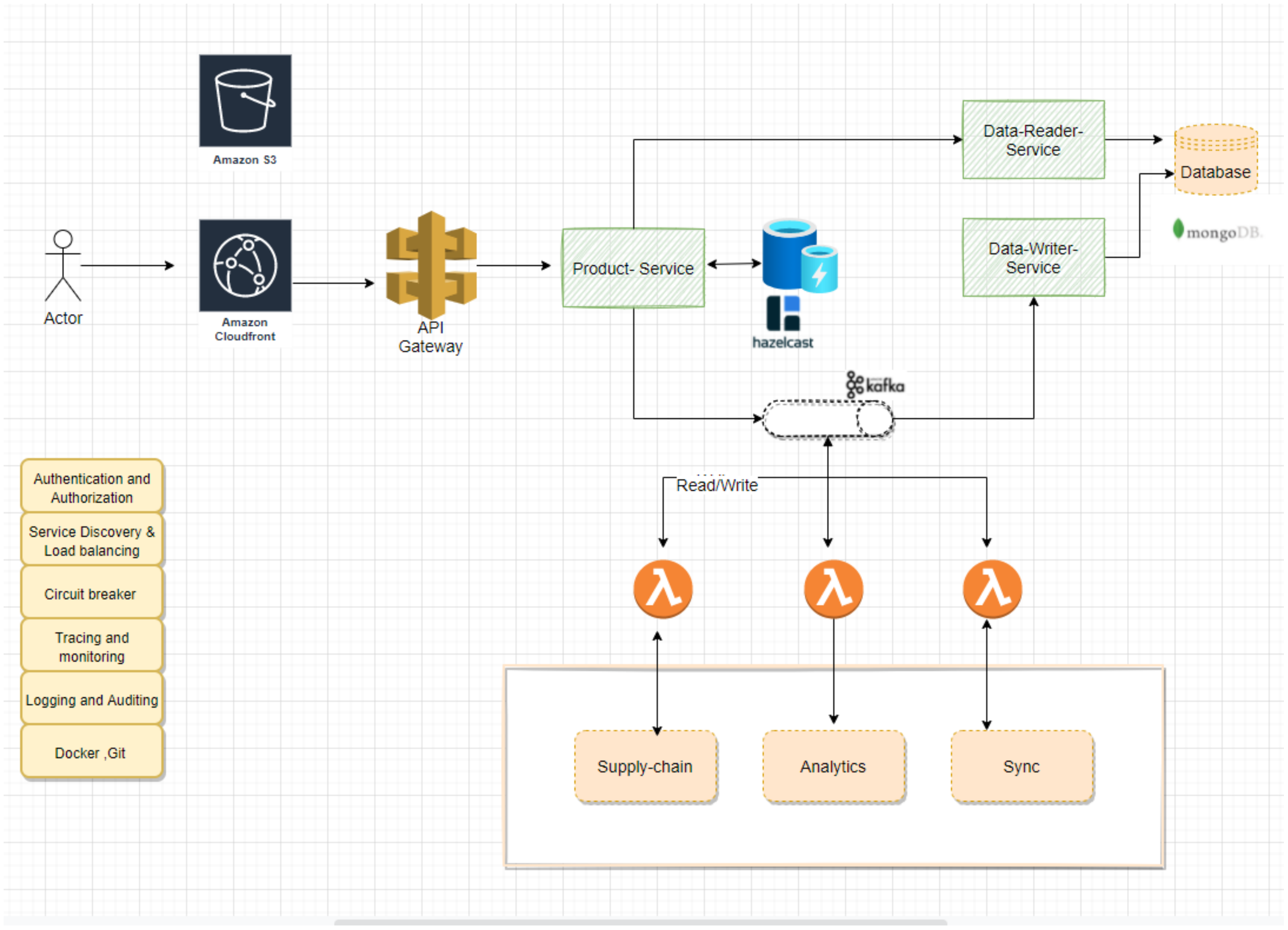


Diagram4:

- Shobith Pejathaya