Online Shopping Platform

1. Introduction

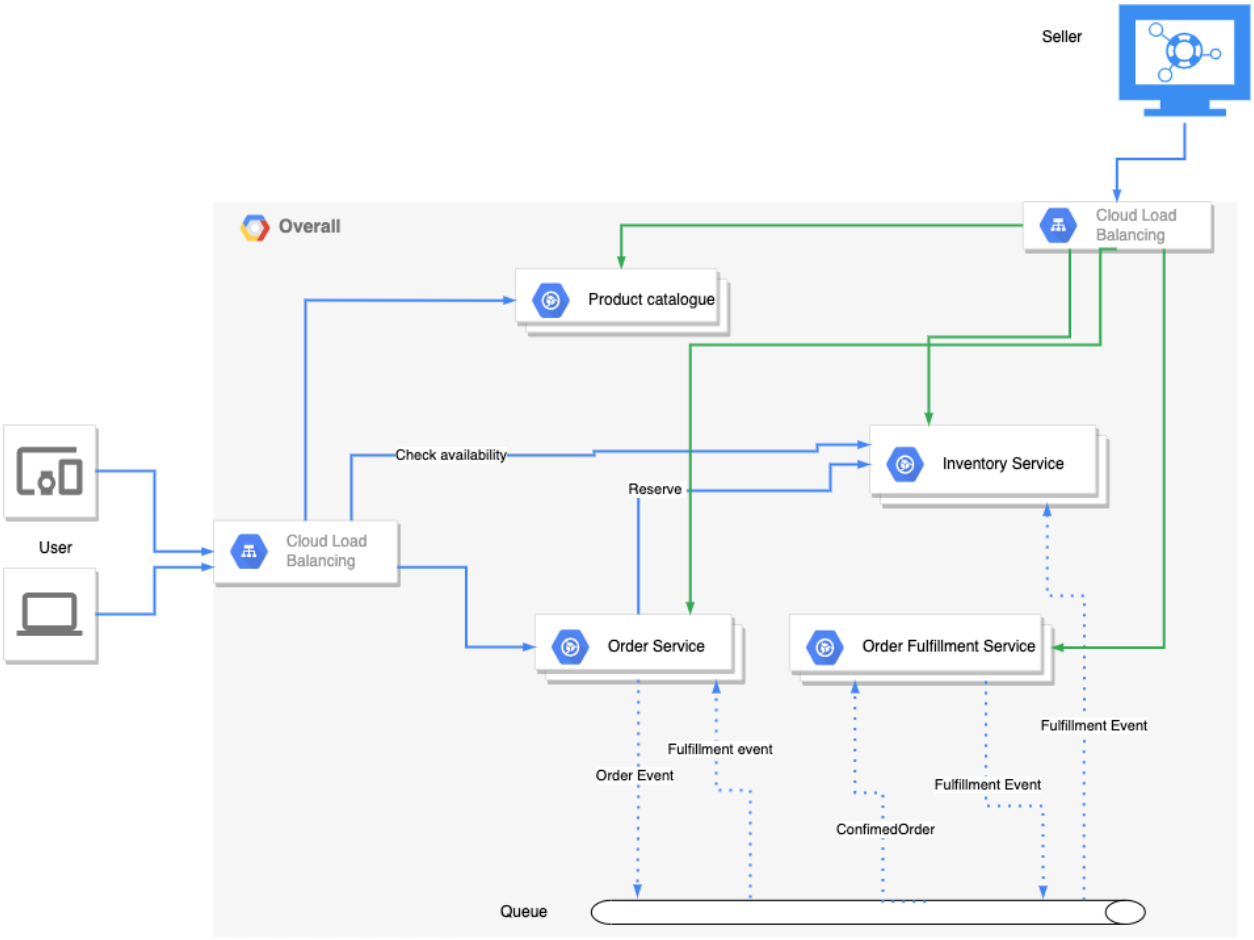
This is a simple architectural design for an online shopping platform. For the purpose of this document, it only focuses on main modules of an online shopping system which are product catalogue, order management, inventory and delivery management.

In this document we only describe the overall architecture of the system and then focus on 2 of main services of the system which are product catalogue and inventory service.

For readers:

- This document only focuses on main backend modules
- This is an technical architectural design which is independent of any special programming language or infrastructure
- For each module, it may recommend a specific tool like MySQL, Elasticsearch, MongoDB,...
- Symbols in this document are borrowed from online drawing tools and some of them are from google cloud icons. This does not mean that the design in this document is stuck with google cloud tools. It is just for convenience.

2. Overall architect



To provide a simple design for an online shopping platform which is focused on the main business flow: sellers will post products to the platform and customers will use any devices to search and buy products that they want.

For this purpose, the system will be decomposed into 4 main services:

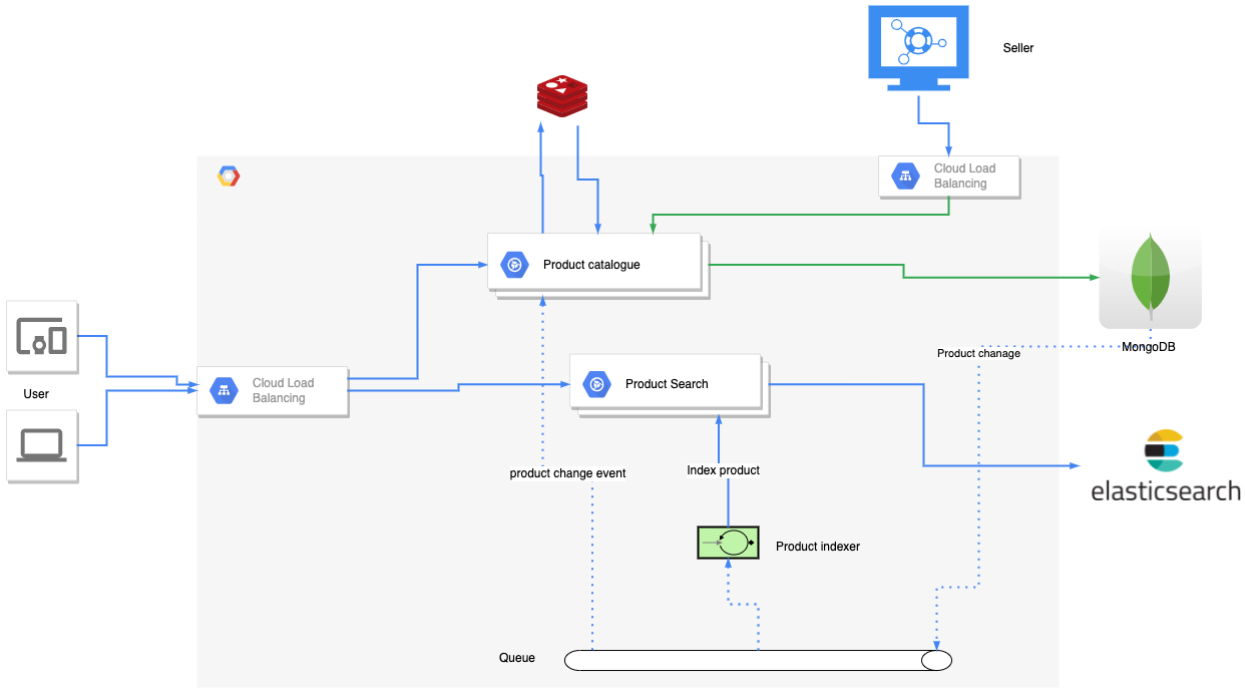
- Product catalogue service
- Cart and Order service
- Inventory management service
- Fulfillment service (delivery)

In order to increase the independent deployment capacity of each service, this design leverages a queue system to employ async communication between services. With async communication between services, we prevent services from tight-coupling and make the system easy to scale.

The main flow:

- Customers will use a mobile app or the platform website to search for their desired products through product catalogue API.
- After choosing products, they will add them to the shopping cart and then do the checkout by order API.
- Before checkout, order API will call (sync) the inventory service to check for product availability.
- If all chosen SKUs have enough quantity, order service will call inventory service to reserve product quantity as in user order.
- An event will be sent to the queue for an succeed order
- The fulfillment service will pick succeed order event and an fulfillment order will be created
- (Optional) then the fulfillment service will send back an event with fulfillment order ID, order event will use this ID and update to order data for later query
- Seller will operate through the fulfillment API. After fulfillment order is finished (success or fail), it will dispatch an event to the queue
- If success, inventory service will pick the event and update reserved quantity to reduce available quantity and vice-versa
- Order service will also use this event to update information on customer order

3. Product catalogue

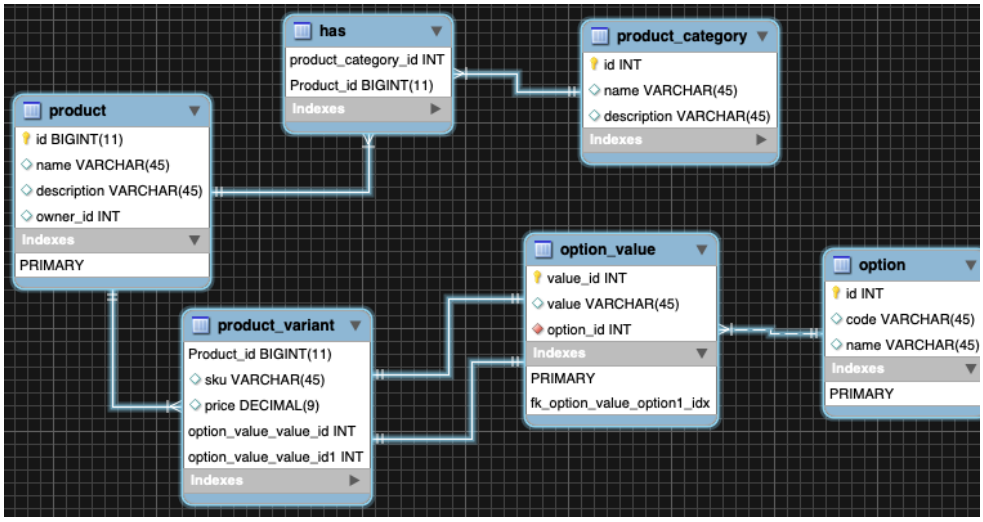


Product catalogue architecture

Product catalogue will be divided into 2 services: product service and product search service.

- Product service: Handle all CRUD request of products and product categories
- Product search service: Build to optimize search request with help of Elasticsearch

3.1. Product service



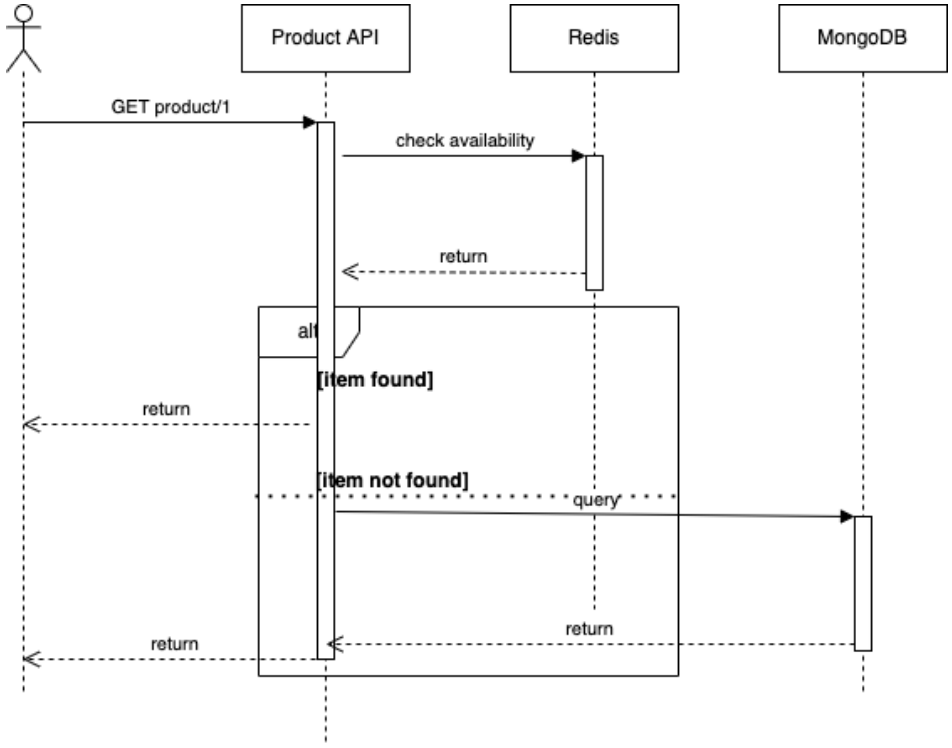
Product service conceptual ERD

Product service handles all CRUD requests for products and product categories. To support complicated and unpredictable data on a product, I recommend using MongoDB (or any suitable document DB) to store products' main data. Above diagram just illustrates the conceptual relationships between entities in product service:

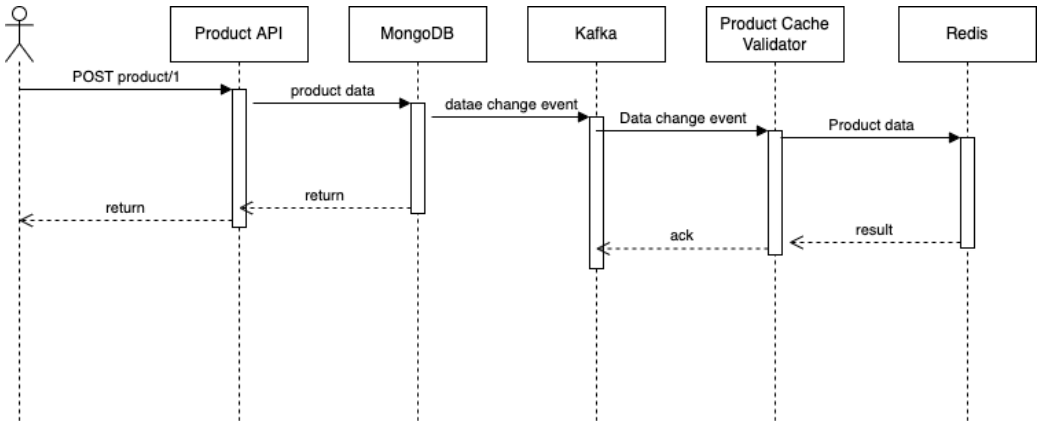
- A product may belong to one or many product categories
- A product may have one or many variants
- In this design, we assume that we will support up to 2 variant options

When we implement this service, we will use a MongoDB (document DB) to provide more flexibility to handle a variety of products and product information. When we choose to use a Document DB, it means we increase the flexibility but decrease the level of consistency. With this kind of service, we don't need a very strict ACID database system. It is a very rare situation that there are a lot of concurrent requests to update some information on a product. If it happens, it is not a big deal if there is some inconsistency for a short time on some information of a product. Note that the price stored in this API is just a comparable price and for comparison purposes only. We need a separate service for managing tradable prices.

Service implementation also divides read and write to products to two separated flows.



A product read handling sequence diagram



A write request handling sequence diagram

3.2. Product search service

For the purpose of this document, the product search service is just a simple RestFul API that handles HTTP requests from clients and uses Elasticsearch to provide a high performance and rich feature search service.

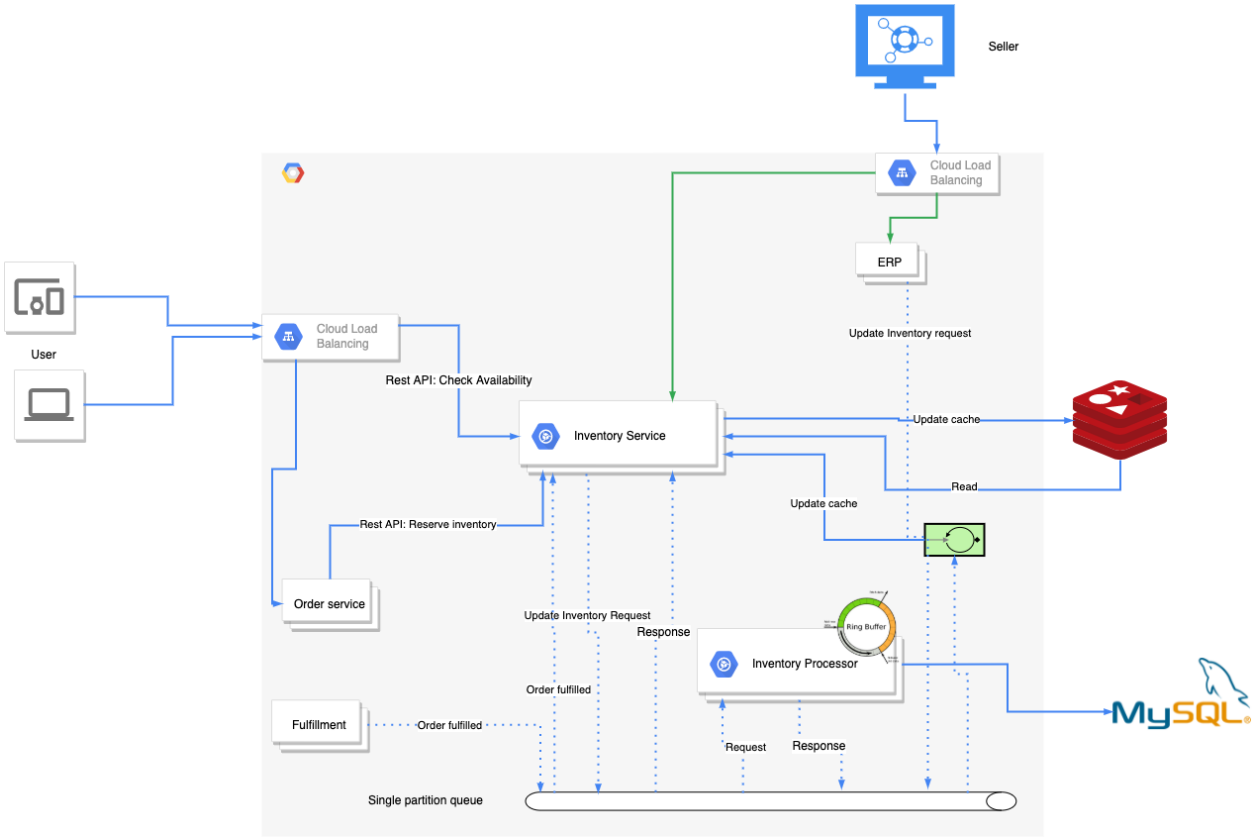
In order to keep the data up to date, we add a data indexer which keeps listening to product data change events from kafka (or any queue) and update data to elasticsearch with whatever optimization that we want.

4. Inventory Management

Inventory management service will be divided into 2 sub components: Inventory Restful API and Inventory processor.

- Inventory Restful API is the public interface handles all request form external systems
- Inventory processor is the internal component to handle business logic

The Restful API and internal processor will communicate in an async model through a message queue. Because the Restful API needs to respond to clients about result of their request in sync way (check availability, reserve inventory for order), the Restful API send a message (update inventory request) to the message queue and wait for the response message from the processor, also through the message queue.

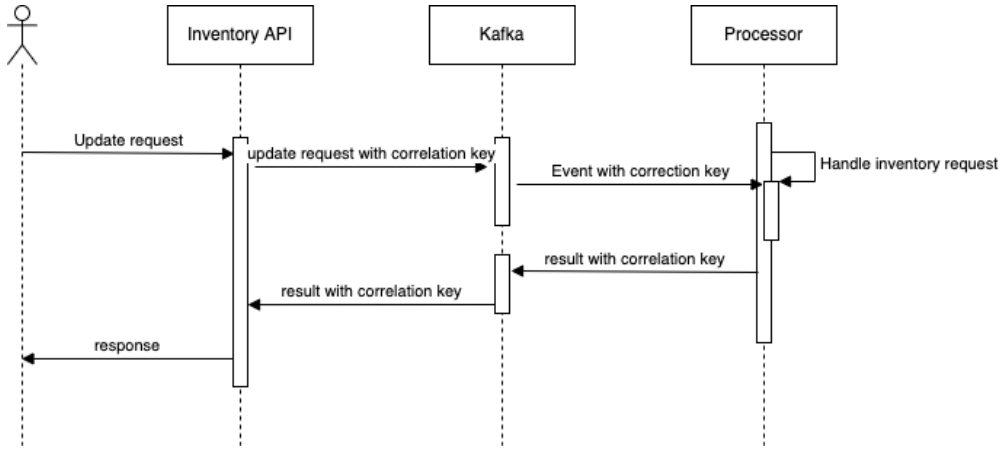


4.1. Inventory Restful API

Inventory API is just a simple Restful API that handles HTTP requests and uses redis as main data source for performance.

- For read requests: it just simply read data from Redis and send response back to client
- For write requests:
    - API get HTTP request from client, parse request and keep the connection waiting
    - Prepare update message with a correlation key and send to kafka (single partition)
    - Wait for message from kafka and check the correlation key
    - If a response and request with the same correlation key are found, it use data from kafka message and prepare a response to send to client

The request and reply communication through kafka can be easily implemented using Spring Kafka if you are using Spring Framework. But it is also easy to implement in any other language. In the last project, I used Go for the Restful API.



4.2. Inventory processor

The most interesting part of the system is the inventory processor. In this design we don't use database transactions to handle inventory updates. The traditional way to handle transactional requests like inventory updates is using database transactions but it requires lock and is hard to scale.

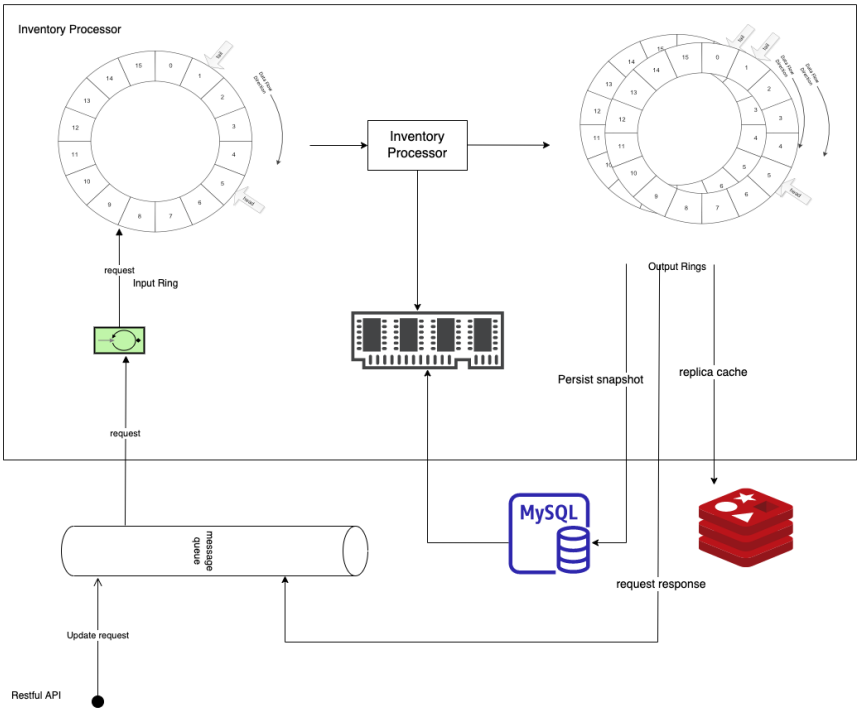
The main idea of database transactions is to ensure the correct order of update operations by using locking. We can prevent using locks but still ensure update order by using a single thread for all update requests. We don’t need any lock if we have only one thread “Inventory processor” to handle all update request because there is no concurrent operation.

To ensure performance of the system with single business processor thread, we remove all I/O operations from this processor. The processor will only use data in RAM memory and update inventory in RAM only. The whole state of inventory is stored in RAM,

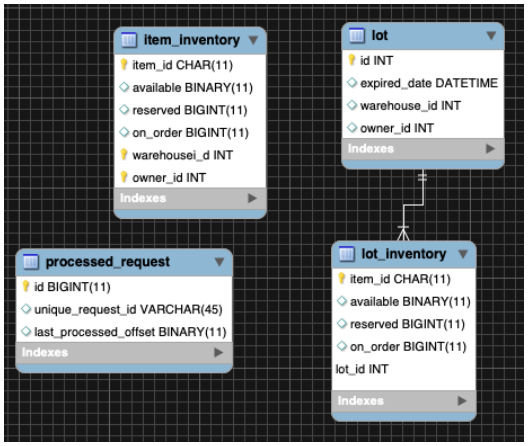
MySQL is used to store snapshots of inventory for read and recovery purposes. To separate all I/O operations from the main business processor without any blocking, we use a very high performance queue backed by Ring Buffer algorithm. In this design we use LMax Disruptor (https://lmax-exchange.github.io/disruptor/disruptor.html).

Another important part of this design is the request queue. The state of the whole system is the state stored in RAM + unprocessed messages in queue. The queue also acts as the event store. The whole state of the system can be restored by replaying all messages persisted in the queue. In order to ensure all updates are in correct order, we use a single partition kafka message topic for receiving all update requests.

One of the advantages of this design is that it is very easy to test. With only one thread and processes all data in RAM, testing becomes very simple. You don't need a web server or any database server to prepare the test environment.



Failure Recovery: With all data stored in RAM memory, we can ensure a very high performance processor with only one thread. However, this faces the risk of losing all data if the system crashes because all data is stored in RAM. In order to recover from any crash, we store the latest offset of processed messages into MySQL anytime we sync a snapshot from RAM to MySQL. If any crash happens, after restarting the system, it will read the latest processed offset and replay all messages from that offset to restore the whole system state into RAM memory.



ERP for managing inventory

5. Summary

5.1. Product catalogue

- Language: any
- Man database: MongoDB
- Search: Elasticsearch
- Cache: redis (or can be an local cache to increase more read performance)
- Queue: kafka

5.2. Inventory management

5.2.1. RestAPI

- Language: Any
- Queue: kafka
    - All Update requests are sent to a single partition request topic
    - All responses are received from response topic with correlation id
- Cache: redis

5.2.2. Processor

- Language: Java (not use any framework)
- Request and response queue: kafka
- Process queue: LMAX Disruptor
- Database: MySQL (any relational database)