Containerized Ecommerce Microservices with dotnet

Mathias Nielsen [mani320@student.sdu.dk](mailto:mani320@student.sdu.dk)

Bachelor Project in Software Engineering

June 2023

A picture containing black, darkness

Description automatically generated

The Maersk Mc-Kinney Moeller Institute

University of Southern Denmark

Supervisor:

Jesper Minor Melholt, TV2

Abstract

Contents

[Introduction 5](#_Toc135026265)

[Context 6](#_Toc135026266)

[2.1 Business Domain 6](#_Toc135026267)

[Analysis 7](#_Toc135026268)

[Requirements 7](#_Toc135026269)

[Adding to basket 8](#_Toc135026270)

[Source of stimulus 8](#_Toc135026271)

[Stimulus 8](#_Toc135026272)

[Response 8](#_Toc135026273)

[Response measures 9](#_Toc135026274)

[Handling orders 10](#_Toc135026275)

[Source of stimulus 10](#_Toc135026276)

[Stimulus 10](#_Toc135026277)

[Response 10](#_Toc135026278)

[Response measures 11](#_Toc135026279)

[Update product database 12](#_Toc135026280)

[Source of stimulus 12](#_Toc135026281)

[Stimulus 12](#_Toc135026282)

[Response 12](#_Toc135026283)

[Response measures 13](#_Toc135026284)

[Payment 13](#_Toc135026285)

[Source of stimulus 13](#_Toc135026286)

[Stimulus 14](#_Toc135026287)

[Response 14](#_Toc135026288)

[Response Measure 15](#_Toc135026289)

[Shipping 16](#_Toc135026290)

[Source of stimulus 16](#_Toc135026291)

[Stimulus 16](#_Toc135026292)

[Response 16](#_Toc135026293)

[Response Measure 16](#_Toc135026294)

[Resiliency 17](#_Toc135026295)

[Services 17](#_Toc135026296)

[Network Communication 17](#_Toc135026297)

[Direct Service Call 17](#_Toc135026298)

[Choreographed Pattern 18](#_Toc135026299)

[Observability 18](#_Toc135026300)

[Language 19](#_Toc135026301)

[Dapr 19](#_Toc135026302)

[Deployment 20](#_Toc135026303)

[Design 20](#_Toc135026304)

[Architecutre 20](#_Toc135026305)

[Broker 21](#_Toc135026306)

[Datastorage 22](#_Toc135026307)

[Events 23](#_Toc135026308)

[1. Implementation 25](#_Toc135026309)

[2. Test 26](#_Toc135026310)

[3. Discussion 27](#_Toc135026311)

[4. Conclusion 28](#_Toc135026312)

# Introduction

A structure is stable if cohesion is strong, and coupling is low. When designing software, the goal should be to have an architecture with low coupling between the components that make up the system. Traditional methods of designing software have limitations when it comes to the deployment and scalability of the system. This is where microservice architecture has its advantages in areas as deployment and scalability. Each service should be aimed to be independently developed, deployed and scaled from one another and designed around a specific business domain which in total forms the entirety of a system.

The goal for the ecommerce store backend, which serves as the business domain for the project, aims to implement a microservice architecture with a focus on low coupling between the services as well as the methods of communication between these services. Finally, the goal is to containerize these service that make up the ecommerce store and deploy them independently from each other.

# Context

The context for the system of this project is an e-commerce backend system where the main focus is on designing the architecture to have a low coupling between the components of the system while still maintaining a high cohesion. The focus of this project lies in the backend part of the e-commerce system with analysis, design, and development of key identified services of an e-commerce backend, where other services act as mock services.

## 2.1 Business Domain

The architecture behind the e-commerce system is a microservice architecture where each service is specifically modeled around the business domain of such a store. Each microservice has its boundaries where information specific to that service is hidden. Therefore, related behavior in the business domain is developed and deployed together while unrelated behavior is being handled elsewhere. The services that make out the system as a whole also need to be loosely coupled. A change to Service A should not require a change to Service B. Therefore, the services need to be modeled as loose couples of units that don’t require a ripple effect throughout the system whenever a change is made. The boundaries of the software can be drawn based on each of the parts of the system that handles a specific behavior. Meaning the boundaries are set around the domain itself which is commonly referred to as Domain Driven Design[[1]](#footnote-1) where the business domain is the core of the software. A clear boundary must be presented to the system as a whole whereas internal implementation is hidden and able to change without impacting other services of the system.

# Analysis

The ecommerce system has different business boundaries where each service should reseamble the business domain. A service that handles payment should not directly share information with another service which does not handle payment. The quality attributes of the system will be described with a flow from the start when browsing for products to when an order is ready for shipping to further investingate the requirements for each of the services. The attributes described in this process are:[[2]](#footnote-2)

* *Source of stimulus*
* *Stimulus*
* *Response*
* *Response measures*

## Requirements

The actions that are going to happen from the customer browses the shop to the order is ready for shipping split into five activities.

* The customer finds a product they want to purchase, and they add the item to the basket where the product will be reserved and deducted from the catalog database.
* The customer can decide to remove the product from the basket, or they can continue to checkout which initiates and order.
* The customer can decide to complete the order, or they can cancel the order. A cancellation of an order should add the stock back into the catalog database.
* When orders are submitted the money is reserved from the customer and packaging the order can beging.
* The reserved payment will finally be processed when the order is packed and ready for shipment

If any errors happen in each of these steps, the step previously conducted by a service should be rolled back to ensure consistency throughout the services.

## Adding to basket

The customer finds a product they want to purchase, and they add the item to the basket where the product will be reserved and deducted from the products database. The responsibility for containing and storing the products is handled by the basket service.

### Source of stimulus

The source of stimulus for this action is a human triggered event. A precondition for this stimulus is that the customer has logged in to their account to add products to their basket. The source is based on an event that is published whenever the customer adds an item to their basket.

### Stimulus

The stimulus comes in the form of endpoints triggered from the customer or events coming in from a message broker. The endpoints triggered from customers should handle adding products to the basket, removing products from the basket and checking out.

Stimulus can also come in the form of events from a message broker. These events should handle updating the basket based on information from other services such as errors or submitting orders.

### Response

When requests are made to store products in the basket, the basket service should store information regarding the customer id and the specific product id that was added to the basket. The datastorage for the basket service should be a short-term storage such as a Redis cache. Information regarding the product and the amount should be published to the message broker called *On\_Product\_Reserved*.

Customers can also trigger endpoints to update the basket, get information about the basket and delete the basket.

When customers are ready to checkout the product, a request is made which publishes an event called *On\_Checkout* with data concerning the customer and the information about the products from the basket.

The basket service also listens for messages called *On\_Order\_Submit* to be able to delete the entries in the basket when the order has been finalized.

|  |  |  |  |
| --- | --- | --- | --- |
| Event | Pub/Sub | Description | Data |
| On\_Product\_Reserved | Pub | Publish information about customer and products | Customer id  Product id  Quantity |
| On\_Product\_Reserved\_Failed | Sub | Get information if errors happen when trying to reserve products | Customer id  Product id  Quantity |
| On\_Checkout | Pub | Publish customer and product info for checkout | Customer id  Product id  Quantity |
| On\_Order\_Submit | Sub | Get information if orders where submitted, so the entries into the cache can get deleted. | Customer id |
| On\_Product\_Removed\_From\_Basket | Pub | Publish information about the product that was removed from the basket to add back into the catalog database | Product id  Quantity |

### Response measures

The basket service should consume events from the event called *On\_Product\_Reserved\_Failed* to be able to countermeasure any faults that has happened when trying to reserve the product. If messages are received on this topic, the service should remove the earlier entry to the cache about the products in the basket and publish an error message back to the frontend regarding not enough stock. This rolling back any changes to the database is a part of handling database interactions for a specific business process without having to implement some sort of distributed transaction.

## Handling orders

When customers are ready to checkout, the responsibility of purchasing a product moves from the basket service to the order service. This responsibility lies with the order service.

### Source of stimulus

The stimulus that triggers the order service arrives in the from of requests from customers and events from the message broker.

### Stimulus

Customers can initiate requests to submit the final order or to cancel the order. Stimulus can also come in the form of events from the message broker about newly created orders and when to update the status of a specific order based on activites from other services.

### Response

When the customers have checked out the order, information about the order is saved to the order service’s database with an order status of “*pending”*. The service should then publish data on the topic called *On\_Order\_Submit* for other services to consume.

The messages that are received in the event broker from the *On\_Order\_Submit* should be processed in a FIFO queue to accommodate race conditions when two individual customers are trying to purchase the same product with a low stock.

The customer can also choose to cancel the order. If this happens, an event should be published called *On\_Order\_Cancel* so other services can roll back any changes to their database.

If the order service receives events on the topic called *On\_Order\_Shipped*, the service should update the order status from *reserved* in the database to *shipped*.

|  |  |  |  |
| --- | --- | --- | --- |
| Event Name | Pub/Sub | Description | Data |
| On\_Checkout | Sub | Receive information about the products and the customer. | Customer id  Product id  Quantity |
| On\_Order\_Submit | Pub | Publish order information when order has been submitted by the customer. | Customer id  Name  Mail  Address |
| On\_Order\_Cancel | Pub | Publish if an order in the making is cancelled by the customer. | Order id |
| On\_Payment\_Reserved\_Failed | Sub | Receive information if errors happens when processing payment, to roll back changes in order database | Customer id  Order id |
| On\_Order\_Shipped | Sub | Receive information when the order has been shipped to update the payment status in order-service database. | Customer id  Order id  Order Status |

### Response measures

If errors from other services trying to process the order happens, the service receives information about the specific order on the event called *On\_Payment\_Reserved \_Fail.* Thereafter, the order service can initiate rollbacks to its database and update the order status for the specific order to *cancelled.*

## Update product database

When customers put orders into their basket or cancel orders, the quantity of the product in the database should adjust according to the action. This responsibility lies with the catalog service.

### Source of stimulus

The source of stimulus for this action is based on events emitted from the order service and the basket service.

### Stimulus

The stimulus for this service arrives in the form of customer initated requests or via messages received from the event broker.

### Response

When receiving messages on the topic *On\_Product\_Reserved*, the quantity of the product in question should decrease by the amount that the customer wish to buy if there is enough in stock. Since the logic of reserving products is a business logic that span multiple services, means that eventual consistency is adopted. The solution for accommodating reservations and eventually race condition edge cases where two or more customers reserve that same item is to have the message broker distribute the events in a FIFO order.

Messages can also be received on the topic *On\_Priduct\_Removed\_From\_Basket,* which should update the stock in the catalog database with the amount that was removed from the basket.

Messages can also be consumed on the topic *On\_Order\_Cancel* from the order service. This is received when orders are cancelled by the customer. Here, the order that have been cancelled should increase the stock for the product in question by the amount intended to purchase by the customer.

|  |  |  |  |
| --- | --- | --- | --- |
| Event | Pub/Sub | Description | Data |
| On\_Product\_Reserved | Sub | Should deduct available stock for specific product when products are added to the basket | Quantity  Product Id |
| On\_Order\_Cancel | Sub | Should add stock for specific product when orders are cancelled | Quantity  Product Id |
| On\_Product\_Removed\_From\_Basket | Sub | Should add stock for specific product when products are removed from basket. | Quantity  Product Id |
| On\_Product\_Reserved\_Failed | Pub | Should emit events if reservations have failed. | Quantity  Product Id |

### Response measures

If the reservation of an item fails, it should be published on an event called *On\_Product\_Reserved\_Failed*, so other services can handle roll back any changes to their own business logic or state store. This mainly happens if there is not enough stock in the catalog database.

## Payment

When the order from the customer has been submitted, the payment for the product is going to be reserved from the customer. This activity is handled by the payment service which is a mock service.

### Source of stimulus

The input that triggers this service, is based on events published from other services in the system.

### Stimulus

The stimulus are events, primarily published by the order service and the shipping service. The event is called *On\_Order\_Submit* from the order service and *On\_Order\_Shipped* from the shipping service.

### Response

The service will consume the event from the order service and publish when the payment from the customer has been reserved on the topic called *On\_Payment\_Reserved.*

When events are received from the shipping service, this service will publish events called *On\_Payment\_Finalized* which tells other services that the payment for the order has been processed.

|  |  |  |  |
| --- | --- | --- | --- |
| Event | Pub/Sub | Description | Data |
| On\_Order\_Submit | Sub | Payment reservation for the order will be executed when the order has been submitted. | Customer id,  Order id,  Order Status |
| On\_Order\_Shipped | Sub | Finalize payment when package is ready for shipping. | Customer id,  Order id,  Order Status |
| On\_Payment\_Reserved | Pub | Publish information when the payment has been reserved. | Customer id,  Order id,  Order Status |
| On\_Payment\_Reserved\_Failed | Pub | Publish if a payment has failed and what order has been affected. | Customer id,  Order id,  Order Status |
| On\_Order\_Paid | Pub | Publish information when the order has been fully paid. | Customer id,  Order id,  Order Status |

### Response Measure

Since this is going to be a mock service, this service should publish events where payments are either reserved successfully or if it fails to reserve money. If failure happens, an event should be emitted, and rollbacks should happen in the rest of the system. Successful or failed payments are determined by chance.

## Shipping

When payment from the customer has been reserved the order should physically be packed and made ready for shipping. This responsibility is handled by the shipping service.

### Source of stimulus

The trigger for this part of the system arrives in the form of events from the payment service.

### Stimulus

The event that come in from the payment service is called *On\_Payment\_Reserved,* notifying when the payment for the order has been reserved and is ready to get packed and shipped.

### Response

When events come in, the service should publish events on the topic called *On\_Order\_Shipped*, notifying other services that the order is ready for shipment.

|  |  |  |  |
| --- | --- | --- | --- |
| Event | Pub/Sub | Description | Data |
| On\_Payment\_Reserved | Sub | Get information when the payment from the customer has been reserved. | Customer id,  Order id,  Order Status |
| On\_Order\_Shipped | Pub | Publish information when the order is ready to be shipped. | Customer id,  Order id,  Order Status |

### Response Measure

Since this service is intended to be implement as a mock service, meaning the service reacts to events coming in, the response measures are limited to publishing an event, letting other service know that the specific order has been shipped so other services can update their database with this information.

## Resiliency

## Services

## Network Communication

The services would need to communicate with each other to perform the necessary actions to handle customers’ requests as they arrive and other internal communication between services. There are several ways to handle inter-communication between the services in the system. However, each method of communication between services has its pros and cons that need to be assessed early in the project so the services can be designed in a way that can make the system resilient to heavy traffic and potential network errors between the services.

## Direct Service Call

When designing the communication between services it can typically be done through direct service-to-service invocation or event streaming.

A service finishes its operation and sends a request to another service. This direct service call can be done synchronously or asynchronously. When sending a synchronous call to another service, the service is waiting for a response that results in a tighter coupling between the services. If the service that is requested is unavailable it will lead to failure in the request. The process can then be blocked while waiting for a response, even though it would be possible to implement some patterns to avoid blocking like circuit breakers.[[3]](#footnote-3)

Asynchronous request-response calls can mitigate this blocking issue and can carry on with other tasks. The service initiating the call can carry on with other tasks while it waits on a response from the called service. However, with the direct service call, it needs to know about other services and who to send the data to which results in tighter coupling between the services that would not be necessary with asynchronous event-based communication.

## Choreographed Pattern

The choreographed pattern aims to be fully independent of any upstream producers or downstream consumers[[4]](#footnote-4). Each microservice reacts to specific events as they arrive through an event broker. When aiming for highly decoupled microservices, this pattern is often preferred. A system based on the choreography pattern relies on the system to be event-driven that provides reusable event streams of business information. The producers of events of this system do not know where or who needs the information that the producing service emits. This style of communication reduces the coordination between services. It can also be easier to add new services to the system as they can start consuming specific events relevant to their business domain. This pattern is good when low coupling is the goal, but it also comes with its separate challenges[[5]](#footnote-5) as the message broker needs to be configured and discovered by all the services.

## Observability

Observability is an important aspect to see the different services and their interaction across the system and network. In decoupled services where no direct service call[[6]](#footnote-6) is involved, it can be a challenge to see the overall flow of data and track events in the system. Since the services are distributed it can be helpful to implement distributed tracing by collecting different events and operations across services. All the data that these services and events produce can be queried and visualized to understand relationships between services. Tools such as Zipkin can be configured and deployed in the same cluster as the services to trace data in the system.

## Language

Theoretically, each service could be written in different languages across the services as long as the language supports HTTP requests. A service that primarily is responsible for statistics and analyzing user behavior could utilize Python to make specific data visualizations. However, since the scope of the project concerns a few selected services that all strongly concern themselves with a specific business domain, it would be helpful to choose an object-oriented language to model this business domain. Languages such as Java and C# can be used. In this project, the services will be written in C# and .NET 6.

## Dapr

To help develop the microservices and communicate with other components of the system such as a message broker or other services, Dapr will be used. Dapr stands for Distributed Application Runtime which provides APIs that simplifies connectivity to other services. Dapr has 10 *building blocks[[7]](#footnote-7)* concerning things such as service-to-service invocation, state management, publish and subscribe, observability, and more. These components are configured in a separate yaml file which gets deployed together with the specific service. This can be self-hosted in a docker-compose environment or on a Kubernetes cluster.

These building blocks are independent of one another and do not require all of them to be used. Dapr is deployed as a sidecar with the service, so all traffic to and from the service goes through the Dapr sidecar.

Dapr utilizes the sidecar pattern to enable communication with other services, message brokers, or state management components. In the sidecar pattern, separate processes or components of an application are deployed separately to the application[[8]](#footnote-8). It is referenced as the sidecar pattern as it resembles the sidecar of a motorcycle. The motorcycle represents the application while the sidecar represents a related process that is attached to the application. The dapr sidecar could be logging or monitoring and it is handling all outgoing and incoming communication to and from other services or message brokers.

## Deployment

If done correctly, the microservices can be deployed and updated independently from each other. Each of the services will be deployed in a self-hosted docker-compose environment and not in a Kubernetes cluster. Dockerfiles are needed for each of the services that need to be deployed, but also for data storage, message broker, and observability. In Docker-Compose, each of the containers is discoverable to each other as a network is created where each service gets a DNS that is reachable to other containers in the network.[[9]](#footnote-9)

# Design

The system is utilizing an event-driven design where communication between the services are done by publishing and consuming different events that concern the specific services.

## Architecutre

The different microservice has its own business domain to handle without interfering our identifying other services. Requests for each of the service is based on requests from customers or messages from the event broker. Each service produce events for downstream services to consume without knowing which specific service will react to the event which makes the architecture more loosely coupled.

The overall architecture is built up of 5 services where 2 of the services serves as mock services that consumes and publishes specific events without handling any logic or storing specific business data in a database. The other 3 services have their own separate database which the other services does not know about. Events from the services are published to a message broker and distributed to specific services.A diagram of a blockchain

Description automatically generated with low confidence

## Broker

Kafka is used as the message broker. Messages arrive from the service in a FIFO queue where race conditions can be omitted if two or more customers are trying to purchase a product with low stock as the events from the producers are distributed by Kafka in the order they are produced to Kafka. If two or more orders of the same product with a low stock arrive, race conditions can then be dismissed as the first order to be arrive to Kafka is the one who gets to purchase the order.

Kafka was chosen as the event broker as it is capable of handling high amounts of data in a manner of real time processing instead of a batch-oriented processing style.[[10]](#footnote-10) Kafka is also capable of storing the messages when they are received by the broker and can then be stored on disk. This allows for sending messages to newly deployed consumers to receive past sent messages. Since the ecommerce store potentially could have thousands of customers triggering thousands of messages across the system, the message broker needs to handle this in a real time processing manner. This is where Kafka is chosen as the message broker over other systems such as RabbitMQ.

Kafka was chosen as several services would need to read the same event. These topics are partitioned which furthers the scalability as services can read and write to multiple instances at the same time.[[11]](#footnote-11)

## Datastorage

The services that are not intended as mock services have their own datastore that is only accessible to the specific service.

#### Catalog Service

This service is responsible for storing all the products available at the store. This service requires a persistence store where all the products are stored in.

The data that is going to live in the catalog service is going be for long term storage. The datastore should also be able to handle more complex queries. The data for this service should be able to be queried in separate ways for customers on the frontend to filter all the products by different measurements. Already, when more complex queries are required, the traditional relational SQL database are preferred over a NoSQL database.

Since this service should hold all the products, it does not have to store millions of entries. It could be possible for the catalog service to make use of a NoSQL database but making flexible queries can be a bit harder to perform compared to a relational database.[[12]](#footnote-12)

The NoSQL is by design very useful for distributed architecture[[13]](#footnote-13), but they come with a cost. That is, we sacrifice consistency between nodes when referencing the CAP Theorem. However, because of the requirement to make flexible queries the database selected is a MySQL database that is set up as a container.

When referring to the CAP theorem, the MySQL database would reside on the CP side, meaning all reads receive the most recent data or an error and partition tolerance at the cost of availability[[14]](#footnote-14). Since the catalog service holds all the product, a product shouldn’t be sold to the customer if it is not in store.

#### Basket Service

The basket service is responsible for storing products that the customer wants to purchase. This service needs a persistence store to store information about the products in the basket. The data that are stored by this service is not required to store the data long-term. It just needs to act as a cache to store products short-term for customers. This service should be required to handle fast reads and writes as customers use the basket service. Therefore, the datastore chosen for this service is a Redis where each entry in the cache is key/value based. Each customer has a customerid, that Redis uses as the key for storing and retrieving data. The value is information about the product that was saved. This means that only one basket can be used by the customer at the time since every customer only has one unique customer id. Redis was chosen as the cache for this service, as the data is short-lived with time to live configuration as it should not be persisted to disk and saved. Also, Redis is horizontally scaled which is good for scaling out in situations where the system needs to work under a loads from many customers where Redis can handle a high throughput. Redis does not require a trip to a disk which enables fast queries. [[15]](#footnote-15)

#### Order Service

The order service would be responsible for storing incoming orders made from customers. The data that is going to live in the database for this service is going to live there long-term meaning it should not be removed after shorter periods like the basket service’s data. The order service needs to store the orders in a database that can handle many writes during peak periods. Also, the data that is going to live here needs to be reliable data that is not going to be lost. The database should also be an SQL database but for different reasons compared to the catalog service. This service has a strict requirement on reliability and consistency as the service is dealing with purchases customers have made, which a NoSQL database cannot guarantee consistency across all nodes. Since these requirements for the database leans towards a CP type of databases in the CAP theorem[[16]](#footnote-16), the database chosen is also a MySQL database.

## Events

To better get an overview of all the events the different services publishes and consumes, a diagram of the relationship is made.

# Implementation

# Test

# Discussion

# Conclusion

1. Building Microservices page 61 [↑](#footnote-ref-1)
2. Software Architecture in Practice, Len Bass, Page 75 [↑](#footnote-ref-2)
3. Building Microservices page 95 [↑](#footnote-ref-3)
4. Building Event Driven Microservice page 136 [↑](#footnote-ref-4)
5. Building Microservices page 116 [↑](#footnote-ref-5)
6. Practical Event Driven Microservice Architecture page 184 [↑](#footnote-ref-6)
7. https://docs.dapr.io/developing-applications/building-blocks/ [↑](#footnote-ref-7)
8. https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar [↑](#footnote-ref-8)
9. https://docs.docker.com/compose/networking/ [↑](#footnote-ref-9)
10. Building Microservices page 139 [↑](#footnote-ref-10)
11. https://kafka.apache.org/documentation/#introduction [↑](#footnote-ref-11)
12. Big data page 6 [↑](#footnote-ref-12)
13. https://www.ibm.com/topics/nosql-databases [↑](#footnote-ref-13)
14. Building Microservices page 408 [↑](#footnote-ref-14)
15. https://developer.redis.com/explore/what-is-redis/ [↑](#footnote-ref-15)
16. Building Microservices page 411 [↑](#footnote-ref-16)