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

[1. Introduction 4](#_Toc134272088)

[2. Context 5](#_Toc134272089)

[2.1 Business Domain 5](#_Toc134272090)

[2.2 Services 5](#_Toc134272091)

[3. Analysis 6](#_Toc134272092)

[4. Design 7](#_Toc134272093)

[5. Implementation 8](#_Toc134272094)

[6. Test 9](#_Toc134272095)

[7. Discussion 10](#_Toc134272096)

[8. Conclusion 11](#_Toc134272097)

# 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 as it enables services to be deployed and scaled individually. Each service should aim to be independently developed 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 foundation business domain, 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 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.

## 2.2 Services

# 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:[[1]](#footnote-1)

* *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 should add the stock back into the catalog database.
* When orders are submitted the money is reserved from the customer
* The reserved payment will finally be processed when the order is packed and ready for shipment

### 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 events from the customer or other services. The events are called *On\_Added\_To\_Basket* and On\_Product\_Reserved\_Failed. The basket service should consume these events to act on whether products are added to the basket or if errors happen when trying to reserve products.

Response

When receiving such events, the basket service should store information regarding the customer ID and the products that was added to the basket in a short-term storage such as a Redis cache that only the basket service knows about. If the write to the cache was successful, an event should be emitted from the basket service called *On\_Product\_Reserved.* This event should contain data about the product and the quantity that was added to the basket.

When customers are ready to checkout the product, they can click on a button 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.

### 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 order service’s source of stimulus is through events from the basket service. The trigger for this activity is in the form of events coming in from the basket service.

### Stimulus

The stimulus comes in the form of events from the basket service which publishes messages on the topic *On\_Checkout.* The order service receives information about who is purchasing the product and what products are going to be purchased.

### Response

When the customer has stated relevant information about the order, the order is saved to the order service’s database and an event is published with the name *On\_Order\_Submit* with data regarding the order. This consists of the time the order was created, when the order was updated, and the payment status. The order itself consists of a unique id, a reference to the customer id, and a list of the items containing name and quantity.

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\_Paid*, the service should update the payment status in the database to paid.

|  |  |  |  |
| --- | --- | --- | --- |
| Event Name | Pub/Sub | Description | Data |
| On\_Checkout | Sub | Receive information about the products and the customer. | \* |
| On\_Order\_Submit | Pub | Publish order information when order has been submitted by the customer. | \* |
| On\_Order\_Cancel | Pub | Publish if an order in the making is cancelled by the customer. | \* |
| On\_Order\_Submit\_Fail | Pub | Publish if errors occur during submitting orders. |  |
| On\_Order\_Paid | Sub | Receive information when the order has been paid to update the payment status in order-service database. | \* |

### Response measures

If errors occur when trying to submit the order and save it to the services’ database, an error event should be published called *On\_Order\_Submit\_Fail.* This event should publish information about what went wrong, and to which order the error occurred so other services can roll back any changes in their database.

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

### 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.

3.2 Resiliency

3.3 Data storage

3.4 Network Communication

3.5 Language

3.6 Deployment

Design

Events

# Implementation

# Test

# Discussion

# Conclusion

1. Software Architecture in Practice, Len Bass, Page 75 [↑](#footnote-ref-1)