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](#_Toc135472683)

[Context 6](#_Toc135472684)

[Business Domain 6](#_Toc135472685)

[Analysis 7](#_Toc135472686)

[Requirements 7](#_Toc135472687)

[Adding to basket 8](#_Toc135472688)

[Source of stimulus 8](#_Toc135472689)

[Stimulus 8](#_Toc135472690)

[Response 8](#_Toc135472691)

[Response measures 9](#_Toc135472692)

[Handling orders 10](#_Toc135472693)

[Source of stimulus 10](#_Toc135472694)

[Stimulus 10](#_Toc135472695)

[Response 10](#_Toc135472696)

[Response measures 11](#_Toc135472697)

[Update product database 12](#_Toc135472698)

[Source of stimulus 12](#_Toc135472699)

[Stimulus 12](#_Toc135472700)

[Response 12](#_Toc135472701)

[Response measures 13](#_Toc135472702)

[Payment 13](#_Toc135472703)

[Source of stimulus 13](#_Toc135472704)

[Stimulus 13](#_Toc135472705)

[Response 14](#_Toc135472706)

[Response Measure 14](#_Toc135472707)

[Shipping 15](#_Toc135472708)

[Source of stimulus 15](#_Toc135472709)

[Stimulus 15](#_Toc135472710)

[Response 15](#_Toc135472711)

[Response Measure 15](#_Toc135472712)

[Network Communication 16](#_Toc135472713)

[Direct Service Call 16](#_Toc135472714)

[Choreographed Pattern 16](#_Toc135472715)

[Observability 17](#_Toc135472716)

[Language 17](#_Toc135472717)

[Dapr 17](#_Toc135472718)

[Deployment 18](#_Toc135472719)

[Design 18](#_Toc135472720)

[Architecture 18](#_Toc135472721)

[Broker 19](#_Toc135472722)

[Data storage 20](#_Toc135472723)

[Events 22](#_Toc135472724)

[Implementation 23](#_Toc135472725)

[Project 23](#_Toc135472726)

[Configuring Dapr 23](#_Toc135472727)

[Endpoints 24](#_Toc135472728)

[Basket Service 25](#_Toc135472729)

[Order Service 26](#_Toc135472730)

[Repository Pattern 27](#_Toc135472731)

[Submitting orders 27](#_Toc135472732)

[Payment Errors and Order Service 28](#_Toc135472733)

[Test 29](#_Toc135472734)

[Reserving Products 30](#_Toc135472735)

[Submitting Orders 32](#_Toc135472736)

[Discussion 35](#_Toc135472737)

# 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 such 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 e-commerce 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 services that make up the e-commerce 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.

## 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 e-commerce system has different business boundaries where each service should resemble the business domain. A service that handles payment should not directly share information with another service that 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 investigate 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 browsing the shop to the order being 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 an order.
* The customer can decide to complete the order, or they can cancel the order. 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 begin.
* 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 by the customer or events coming in from a message broker. The endpoints triggered by customers should handle adding products to the basket, removing products from the basket, and checking out.

The stimulus can also arrive 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 data storage for the basket service should be 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 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 customers 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 were 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 have 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 form 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. The stimulus can also arrive 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 activities 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 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 orders has been submitted by the customer. | Customer id  Name  Mail  Address |
| On\_Order\_Cancel | Pub | Publish if an order in the making is canceled by the customer. | Order id |
| On\_Payment\_Reserved\_Failed | Sub | Receive information if errors happen 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 the 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 *canceled.*

## 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-initiated 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 wishes to buy if there is enough in stock. Since the logic of reserving products is a business logic that spans 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 canceled by the customer. Here, the order that has been canceled 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 products when products are added to the basket | Quantity  Product Id |
| On\_Order\_Cancel | Sub | Should add stock for specifics products when orders are canceled | Quantity  Product Id |
| On\_Product\_Removed\_From\_Basket | Sub | Should add stock for specific products when products are removed from the 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 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 is 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 the 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 arrives 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 implemented as a mock service, meaning the service reacts to events coming in, the response measures are limited to publishing an event, letting other services know that the specific order has been shipped so other services can update their database with this information.

## 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 is done by publishing and consuming different events that concern the specific services.

## Architecture

The different microservice have their own business domain to handle without identifying other services. Requests for each of the service is based on requests from customers or messages from the event broker. Each service produces 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 serve 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 do 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

Figure 1 - High-level architecture of each service with sidecars

Every service has a Dapr sidecar deployed with it. The services only communicate with this sidecar where it is the sidecar’s responsibility to send and receive data to the message broker. The broker sends events to the sidecar and the sidecar sends them to the service. Retries occur if the services are not responding to requests, Dapr performs retries in a fixed interval over a certain period. Communication with the MySQL databases is not performed with Dapr but with an object relation mapper.

## 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 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 e-commerce 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)

## Data storage

The services that are not intended as mock services have their own data store 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 persistent 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 data store 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 can have an advantage over a NoSQL database since it is possible to join tables when querying the 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 products, a product shouldn’t be sold to the customer if it is not in the store.

#### Basket Service

The basket service is responsible for storing products that the customer wants to purchase. This service needs a persistent store to store information about the products in the basket. The data that are stored by this service are 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 customer id, 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 a 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 load 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 by 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. The database could be a 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, where a NoSQL database cannot guarantee consistency across all nodes. Since these requirements for the database lean towards a CP type of database 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 publish and consume, a diagram of the relationship is made.

# Implementation

Key identified actions that are central to the system and their events will be described. These are the actions of handling orders and submitting orders and reserving products in the basket.

## Project

Each microservice has its own dotnet 6 project where all the projects are bundled into a solution. Every project is a Web API Project[[17]](#footnote-17), where each project manages its own dependencies such as Entity Framework Core for mapping objects to the MySQL databases. All of the services depend on the Dapr SDK which is a Nuget package[[18]](#footnote-18) as it makes development easier. Although it is possible to not have any dependencies on the Dapr SDK and only configure the deployments to route specific requests from the sidecar to the correct endpoint.

## Configuring Dapr

The sidecars that Kafka and Redis need to communicate with are configured in separate YAML files which are deployed together with the service they are trying to communicate with. Dapr components have an interface definition so it is possible to swap components with the same interface. A scenario could be that a local Redis instance is used in testing but a separate deployed Redis server is used in a production environment.

When configuring the component for Kafka, the kind of component is specified along with where the Kafka instance can be found. Since Kafka is running in a docker environment, the service name for Kafka is specified.

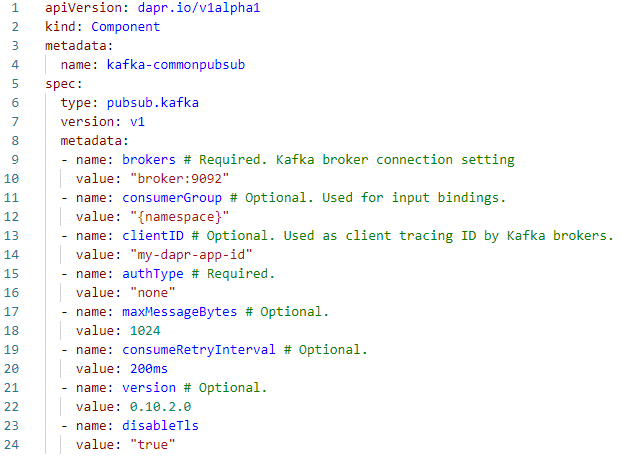


Figure 2 - Dapr configuration of Kafka pub/sub component

In the component specification, the consumer group is set to the namespace. A level of resiliency can also be specified in these components. If the sidecars cannot reach the Kafka instance, retries are performed. The metadata name is important, as this is the name which all the services will publish and consume events from.

## Endpoints

The naming of the endpoints of each service’s API is set up in the same way with an API tag, followed by a version and the service responsibility. An example of reserving products in the basket service:

*http://baseurl:5002/api/v1/reserve*

## Basket Service

The basket service is responsible for storing customer products in a Redis cache and initiating orders. The Basket Services publishes events to the broker based on the different kinds of products reserved and quantities.

To even initialize an order, the product first needs to be in the basket. The reservation of the product is initialized by the customer based on HTTP requests to the endpoint */api/v1/reserve.* Each customer should have a customer id which would be assigned when logging in. Since no service handles authentication, every service assumes the customer is logged in and provides a customer id.

The request is made to the URL with the customer id, product id, and quantity as the body of the request. This information serves as the foundation for reserving products.

A picture containing text, font, screenshot

Description automatically generated

Figure 3- Reserving product code snippet from BasketController.cs

The annotation on line 28 specifics which method is triggered. When making an HTTP request the async method AddProductToBasket is invoked which takes a Reservation object from the body as input. A check is made in the Redis cache on line 36 to see if any products already are present in the basket as the customer should be able to add to the basket later. On lines 47-52, new events are prepared to be published to Kafka with details of the products ready to be reserved which the catalog service consumes. Line 50 is an example of how to use the Dapr client to publish an event. It is important to publish the event to the correct name of the Kafka component as described in the components section. The topic is specified which is the *On\_Product\_Reserved* topic that the catalog service consumes. An Ok result is returned with the status code 200.

The method is marked as async as it ensures that the await keyword on lines 36, 44, and 50 can be used which tells the compiler that the method cannot continue until the process is complete. This turns the method into a state machine meaning it is possible to know the state of an async operation such as the result of the operations.

If an error occurs when trying to reserve the products the basket service will get notified as the service consumes events on the topic *On\_Products\_Reserved\_Failed.* If any events are received, the entry in the Redis cache is deleted as the order cannot be initiated as the product has not been reserved.

A picture containing text, font, line, screenshot

Description automatically generated

Figure 4 - Countermeasures if product reservation fails from BasketService.cs

The deletion of the entry in the Redis cache is handled by the Dapr client. The entry that belongs to the customer is deleted asynchronously and is logged and returns a status code of 200. Since the services can't access each other's databases, a choreographed pattern needs to be orchestrated so each service can update its database to have consistency across the system.

## Order Service

This service has a MySQL database where orders are stored. Communication with the database is not handled by Dapr but with a separate persistence abstraction and Entity Framework Core

### Repository Pattern

The Order Service implements the repository pattern which can help minimize code duplication when accessing the database[[19]](#footnote-19). An interface is defined with the methods for creating, reading, updating, and deleting orders which are implemented in a separate class. The aim is to make the repository implementation persistence ignorant, meaning the best database can be selected for each repository. The interface along with the implementation is referenced in the *program.cs* file, which is where services are registered and the dependency injection container is responsible for managing the lifecycle of the service.

### Submitting orders

When checking out from the Basket Service, an event is published that the Order Service consumes. An entry into the order database is made with an order status of pending. This is for orders which have been initialized but not yet confirmed or shipped.

A picture containing text, screenshot, font

Description automatically generated

Figure 5 - Submitting orders by HTTP requests in OrderService.cs

To submit an order, the endpoint */api/v1/order/{orderid}* is called with information such as name, address, and email which comes from the body of the request. On line 79, the order that is going to be submitted is retrieved from the database by a reference to the repository.

On line 80 the data from the HTTP request is mapped to an object which can be stored in the database. The API uses data transfer objects which can help avoid exposing database entities to the client and generally shape the data that the client sees. The mapping from the DTO to the entity that is saved in the database is done with Automapper which is a library for mapping one object to another.

When submitting the order, the order status on the order in the database is changed from pending to reserved and an event is published via Dapr on the topic *On\_Order\_Submit.*

## Payment Errors and Order Service

The responsibility of handling payments lies with the payment mock service. The payment service consumes events from the topic *On\_Order\_Submit* from the Order Service. When receiving such an event, the mock payment service decides to either accept the payment or publish an error. This was done to show choreography across several services when an error in some part of the system occur. An event is published on the topic *On\_Payment\_Reserved\_Failed* which is consumed by the Order Service. If the payment is successful, an event is published on the topic *On\_Payment\_Reserved* which the mock-service Shipping Service consumes.

A picture containing text, font, screenshot

Description automatically generated

Figure 6 - Handling payment errors from the Payment Service in OrderService.cs

When a payment fails, the Order Service is alerted, but the specific order which it concerns, is not deleted from the database but its order status is set to *canceled.* For every product that belonged to the order, an event is triggered on the topic *On\_Order\_Cancel* which the Catalog Service consumes to make sure the products and the quantity from the order are no longer reserved. This serves as a ripple effect throughout the different services to update their databases to have consistency across the system.

# Test

To test the events that are sent on the specific topics, the system has to be running to verify the correct events are triggered and the correct services are consuming these events. In a distributed system, this can be harder to verify as the services are decoupled. Distributed tracing with Zipkin is used to trace the events to and from different services. Zipkin is running as its own service in the docker-compose deployment and consumes data from the microservices. The services’ APIs are triggered with Postman[[20]](#footnote-20) to verify the responses while the events are analyzed with Zipkin. In total, there are 10 topics that can be tested in the same manner as sending requests to Postman and verifying the events arrive at the correct services in an acceptable time period. Reserving products and submitting orders are two essential parts of the system in which tests are described.

To see all the microservices, the dependency graph in Zipkin can be used to highlight the services and their relation to each other. This relation shows the events that are being distributed to each other. It does not mean that the basket service knows about the catalog service, but it only displays where the events are sent from and to who receives them.

A screenshot of a computer

Description automatically generated

Figure 7 - Zipkin Distributed Tracing across the services

The flow of data can further be analyzed to see the requests, duration, and potential errors. Each blue dot is a service and events between the services travel along the dark lines. The basket-api is the only service to communicate with the basket store which is the intended behavior from the system.

## Reserving Products

When reserving products, the expected result is that the client receives a status 200 on their request, and events on the topic *On\_Product\_Reserved* should be published and consumed by the catalog service. The initial trigger for reserving products is done by sending an HTTP post request to the endpoint using Postman.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 8 - Reserving products using Postman

When the endpoint is triggered, the response for the request is a status code 200 with an execution time of 396 ms. The status code is what was expected. The endpoint should have published the data from the request on the topic *On\_Product\_Reserved* which can be searched using Zipkin.

By searching traces only related to the basket-api it is possible to analyze the flow of data when reserving products.

A screenshot of a computer

Description automatically generated

Figure 9 - Inspecting tracing when reserving products

The flow when saving products in the basket is:

* Check Redis Cache for existing products.
* Save state.
* Publish event.

The duration of this flow took 42 ms. seconds to complete including publishing the event. The longest action was publishing the events which took 30 ms. to complete. The duration for the overall trace is an acceptable amount of time to complete.

*A screenshot of a computer

Description automatically generated with medium confidence*

Figure 10 - Inspecting receiving service of messages on topic On\_Product\_Reserved in Zipkin

The message that was sent on the topic *On\_Product\_Reserved* was received by the correct service as seen on the service-name in the annotation which is the intended behavior. To check the actions of the event the catalog service can be queried to see that the amount of product reserved has been deducted. In the request body, the quantity was set to 2.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 11 - Reducing stock when reserving using Postman

In the response body, the stock has been reduced to 98 on line 16 as expected. The basket is ready for checkout and an order can be initiated.

## Submitting Orders

The checkout is triggered by an HTTP request to the endpoint */api/v1/basket/{id}.* The expected response should be a 200 status code with no content returned. The endpoint is triggered in Postman.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 12 - Checkout with status code 200

As expected, the status code is 200 while no content is returned. It is expected that an event on the topic *On\_Checkout* has been published which only the order service should consume.

A screenshot of a computer

Description automatically generated with medium confidence

Figure 13 - Checkout products in Zipkin

The event was published on the topic *On\_Checkout* and only the order-api was the consumer of this event which is what was expected. The duration for getting the data in the Redis cache, publishing the event, and having the order-api receive the event took 137 ms. to complete. This operation was longer to complete compared to reserving the products, but as the requests still are under 1 second to complete, it is a satisfactory result. When checking out orders, the order should be saved in the order database when data arrive at the order service. Specifically, the order-status for each newly created order should be set to *pending.* This can be verified using Postman on the endpoint *api/v1/order/{id}*

A screenshot of a computer

Description automatically generated with medium confidence

Figure 14 - Verifying newly created orders with correct status in Postman

On line 4, it can be observed that the order status field is set to Pending which is the correct behavior for the order service when initiating new orders.

# Discussion

The catalog service is missing some features that normally would be expected by an e-commerce system. The current implementation can only query all the products or a single product from the database. A potential frontend would mainly be responsible for calling the endpoint to get these products. However, customers should be able to filter the products based on price, availability, color, brand, etc. Not just customers, but also the frontend would need to call the API with pagination to not show everything in the store at once. This is not the case right now. The MySQL database was selected based on its ability to make complex queries easier, but the implementation for utilizing these queries is still lacking.

The order service saves customers into a table when submitting orders. As of the current implementation, authentication and customer handling have not been a part of the design which makes the customer and people aspect of the e-commerce store tricky. When submitting orders, the request requires a name, an address, and email but in the current system, this information only gets saved to a table in the order database, but the data is not being used for anything. This was a step that was being left out of the scope of this project as the workload would be too much for one person. Ideally, the orders related to customers should be able to get queried from the database by the correct person, meaning customers should only be able to query their own orders. This also touches on authentication and authorization which has not been a part of the scope for this project.

Security has not been designed into the services in any way. This would be the next step as it is very important for any system. All the services assume that the customer is logged in to provide a customer id. An authentication service or an API gateway would need to handle the responsibility of authenticating users and verifying identities. Additional steps should be implemented into each existing service to make sure that they cannot be reached directly without passing the API gateway. External authentication providers can also be used, so customers authenticate with Google Accounts, etc. This would be an important step for any further development of the system.

Docker-Compose is good for setting up prototypes and making proof of concepts of a system. This project has 17 services running in docker-compose including Kafka, all the services, databases, tracing, and sidecars. This can quickly be hard to configure and read in one docker-compose file. These services could be deployed in a Kubernetes cluster locally or by using one of the cloud providers such as Azure or AWS which would be better suited when running all these services. Kubernetes brings on a whole separate layer of complexity, but it would be a natural next step for deploying the services as Kubernetes brings several important functionalities such as load balancing, container management, and auto-scaling.

A frontend would be the next step to displaying the data instead of using Postman to call the different services. A prototype can rather easily be implemented using .Net Blazor[[21]](#footnote-21) to keep all development in C# context. A layer between the frontend and the backend would also be needed to not have the frontend directly call the backend services. The backend for frontend pattern could be utilized especially if specific user interfaces are implemented such as mobile apps and web apps.

For a production-ready system, measures for backing up customer data such as orders and the business data such as all the products should be implemented into the system. One way of doing so could be to write Azure Functions that take any new entries and updates from the production database such as the order database and replicates the data in a separate database. Mistakes can happen and databases can be deleted. Therefore, it is very important to have backups of the data in the system.Conclusion

The aim was to design and implement an e-commerce store using a microservice architecture with a focus on a low coupling between the services. To fulfill the goal of having low coupling, modern practices such as event-driven communication were used to keep the coupling low. The current implementation with asynchronous messaging being used as the main form of communication and a well-defined bounded context between the service leads to a system where further development is easier, as newly created services can consume existing events without the need to redeploy the whole system as the services do not have dependencies on each other. The API endpoints and the events for the services were tested to verify the intended behavior using Postman and Zipkin. Finally, all the parts of the system such as the individual services, databases, message broker, and more were each deployed individually in a local docker-compose environment.

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)
17. https://dotnet.microsoft.com/en-us/apps/aspnet/apis [↑](#footnote-ref-17)
18. https://www.nuget.org/profiles/dapr.io [↑](#footnote-ref-18)
19. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design [↑](#footnote-ref-19)
20. https://www.postman.com/ [↑](#footnote-ref-20)
21. https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor [↑](#footnote-ref-21)