What is the tradeoff when opting for asynchronous communication within a microservice architecture?

*S*tudy Programm Digital Sciences - Master
at the Faculty of Computer Science and Engineering
of the Technical University of Cologne

submitted by:

Name:	Marc Kevin Zenzen
Student-No.:	11131724
Name:	Luca Stamos
Student-No.:	11132237
Name:	Stefan Steinhauer
Student-No.:	11132517
submitted to:	Prof. Dr. Stefan Bente

Gummersbach, 16.02.2023

Abstract

When making decisions within a software architecture, there is no right or wrong, there are only tradeoffs. This also applies to the communication between microservices. In this paper, the tradeoff of asynchronous communication is examined using various architectural properties in the example of an eCommerce application. After evaluating these properties, the decisions to be made are explained. Then, guiding principles are given based on which the decision making is supported. Finally, the eCommerce application is developed to prove the conclusions drawn in this thesis.

Contents

List of Figures

1 Introduction

When planning projects, software architects have to decide which services of their application should work synchronously and which asynchronously. For the majority of cases, however, it is not possible to simply Google for an answer as to which decision is the right one, since such a decision is never unambiguous, but always involves a compromise. So the decision always depends on the current circumstances. But on which exactly and how should these influence the decision of a software architect? Using a simplified eCommerce application, this paper examines various architectural properties and their influence on decisions regarding communication between microservices, more specifically: the advantages or disadvantages of asynchronous versus synchronous communication of services.

With synchronous communication within microservices „Everything fails, all the time“ (Vogels, 2022). Such and similar statements are frequently encountered, not least from people like Werner Vogels, the CTO and vice president of Amazon. Indeed, some advantages can be assumed for asynchronous communication. Processes are not blocked and services are less strongly coupled.

First, the basics of microservices and the various forms of communication are explained. The various microservices options are presented. In addition, the differences between the various types of communication between microservices and the advantages and disadvantages are shown. Then, the present domain of the simplified eCommerce application is defined and described. For this purpose, a diagram of the domain is used, which shows the different classes used and divides them into aggregates representing the services. There is also a use-case diagram for the functions that a corresponding application must have. On the basis of the present domain, various architectural properties are determined which are to be examined in the following. Afterwards, individual aspects of the tradeoff between synchronous and asynchronous communication are clarified and explained on the basis of these determined properties. In the next section, the decisions to be made are shown, which type of communication the service uses to each other. Reasons are collected and summarized, which should help with the decision. Here the individual Business Events are enumerated and explained before one is decided justified for one of the communication forms. In the last chapter, a coding experiment is started in which the information gained is used to implement the simplified eCommerce application. The evaluation of the experiment will confirm or contradict the decisions made in advance.

2 Fundamentals

2.1 Event based architecture

The event-based architecture style is an asynchronous architecture style according to Richards, 2020. This finds application in both small and large applications. „An event-based architecture consists of decoupled event processing components that receive and process events asynchronously (Richards, 2020, p. 183)“.

In most applications, the request-based model is used. Here, events are passed to a request orchestrator so that actions can be executed later. The request orchestrator distributes these requests deterministically and synchronously to different services. In these services, the requests are finally processed, e.g. by reading or updating data from a database. An example would be if a customer wanted to have his orders of the last six weeks displayed. In the deterministic query, the data is returned to the customer. It is not an event to which the system must respond.

An event-based model, on the other hand, responds to a specific situation and executes one or more actions in response. An example of this would be when a customer places a bid in an online auction. The bid is not a request to the system, but an event that occurs after the customer has placed his bid. The system has to compare this bid with other bids to find out who is the highest bidder.

The event-based style is a standalone style, but can also be used in other architectural styles, such as the event-based microservice architecture.

2.2 Microservices

Microservices are an architectural concept for the realization of an application. In contrast to monolithic applications, the functions are decoupled from one another and distributed across different applications. Due to the loose coupling, changes to a microservice have no direct impact on other services and cannot lead to the failure of the entire application (Hat, 2018). Microservice architecture is a mixture of request-based and event-based architecture.

An important feature of the microservice architecture is that the entire application does not need to be shut down for updates and maintenance. Due to the scalability of individual services, updates can be carried out by only partially replacing the running services. This means that the application can be updated gradually without having to shut it down. The resources used can also be dynamically adapted to the current workload by scaling the individual services.

Teams can work independently and develop services using minimal communication. Tests can also be limited to individual services instead of having to check the entire application for functionality after a change.

Since the services are now no longer all in one place, a means of communication between the individual microservices must be ensured.

2.3 Communication

In an architecture with multiple services, these can communicate with each other in two different ways, synchronously and asynchronously. While in synchronous communication the sender and receiver are always active at the same time, in asynchronous communication the connection is terminated after the request has been transmitted. To transmit the corresponding response a new connection is opened (Kröner, 2022). This way there is no need to wait for a response before the sender can continue. A good example of synchronous communication from everyday life would be a meeting at work, since it takes place in real time, or a phone call. For asynchronous communication, an email can be used as an example because there is no need to respond directly to the email.

For communication between services, a message can be addressed to one or more recipients. Either all recipients can be notified one after the other or an additional service (e.g. broker or mediator) is used, which receives the message and distributes it to the appropriate recipients or from which the individual recipients can obtain the information.

When designing an application, it is important to decide where to communicate synchronously or asynchronously between services.

2.3.1 Communication vs. Protokoll

Synchronous protocols include, for example, HTTP/HTTPS. The client sends a request and waits for a response from the service. This is independent of the execution of the client code, which can be synchronous (thread is blocked) or asynchronous (thread is not blocked, and the response reaches a callback at some point). The important thing here is that the protocol (HTTP/HTTPS) is synchronous and the client code cannot continue its task until it receives the response from the HTTP server (Microsoft, 2022).

Other protocols such as AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages. The client code, or the sender of the message, typically does not wait for a response. It simply sends the message, as when sending a message to a RabbitMQ queue or other message broker (Microsoft, 2022).

The choice of protocol used ultimately does not matter for data transmission. A synchronous protocol like HTTP can be used for synchronous communication as well as for asynchronous communication and vice versa.

2.3.2 Synchronous Communication

With synchronous communication, a connection remains active until a request has been answered. After sending data, no further action can be taken within the application until the response arrives. Accordingly, the running thread is blocked and does not respond to further requests until the first operation that was triggered has been completed. So that a server can nevertheless accept several requests, a separate thread is started for each incoming request. This thread can then be blocked without effects on other inquiries and running procedures up to the reaction of the communication partner. The advantage of this is that each client has its own "contact person" who waits for its requests and reacts to them as quickly as possible.

2.3.3 Asynchronous Communication

In contrast to the synchronous approach, in asynchronous communication the client sends its request to the application without the certainty of when and if it will be answered. The connection is closed immediately after the request is transmitted. A new connection is established for the respective response. Since it is not possible to wait for an immediate response, asynchronous communication offers the option of using a queue. All requests are stored there and retrieved by the receiver one after the other. With the help of vertical (threads) or horizontal (replicas) scaling, parallel processing can be achieved. The advantages of asynchronous communication can be summarized in principle under the term separation (To, 2022). Services, which do not need to know each other and thus work anonymously in a certain way, are more strongly decoupled from each other. In addition, the strong decoupling makes horizontal scaling easier than with synchronous communication.

There are several different styles of asynchronous communication:

- Request/response - a service sends a request message to a recipient and expects to receive a reply message promptly
- Notifications - a sender sends a message a recipient but does not expect a reply. Nor is one sent.
- Request/asynchronous response - a service sends a request message to a recipient and expects to receive a reply message eventually
- Publish/subscribe - a service publishes a message to zero or more recipients
- Publish/asynchronous response - a service publishes a request to one or recipients, some of whom send back a reply

Richardson, 2021

3 Investigation using the example of an eCommerce application

In this chapter, important architectural characteristics of an eCommerce application are identified and then examined in more detail.

3.1 Domain description

This domain contains domain objects for the realization of a simplified eCommerce application in the form of a proof of concept (PoC). In domain-driven design, aggregates are formed for a specific domain. An aggregate is a group of domain objects that can be treated as a single entity. Figure 1 shows the UML class diagram consisting of four aggregates. Each of these aggregates represents exactly one microservice in the microservice architecture. An aggregate can contain one or more classes, of which exactly one is always defined as the aggregate root. The aggregate root is always the location from which a microservice can be accessed externally.

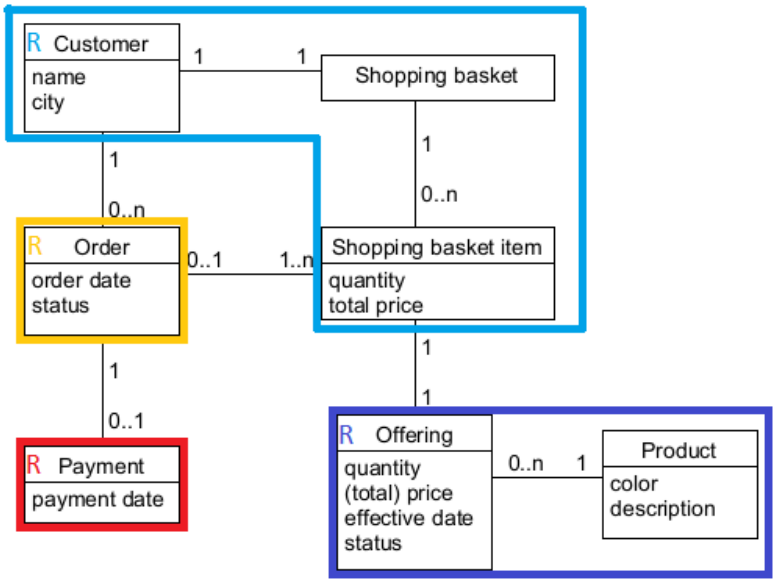

Figure 1 ECommerce Domain

The Customer aggregate (1) contains customers, shopping baskets and shopping basket items (classes: *Customer*, *Shopping Basket*, *Shopping Basket Item*). A shopping basket item contains information about the respective quantity and the corresponding total price. A customer contains as information the name as well as the city. Each customer has exactly one shopping basket and each shopping basket is assigned to exactly one customer. While a shopping cart can contain any number of items, each item is in exactly one shopping cart.

The Offering aggregate consists of offerings (*Offering*) and products (*Products*). A product contains its respective color as well as the product description. An offer consists of the

quantity of a respective product as well as the resulting total price, a date and the status it is currently in. An offer always refers to exactly one product. On the other hand, a product can also appear in different offers.

The order aggregate contains only the orders (class *Order*). An order can be placed on a specific date. In addition, it has a status such as "Paid".

The payment aggregate also consists of exactly one class, the *Payment* class. A payment is processed on a specific time (date).

An item of a shopping cart always refers to exactly one offer. An order always contains at least one item of a shopping cart while an item is either contained in an order or not. Furthermore, an order is always assigned to exactly one customer, while a customer can place any number of orders. Each order is either paid or not and a payment is always assigned to exactly one order.

In Figure 2, the available business events are shown as use cases in a diagram.

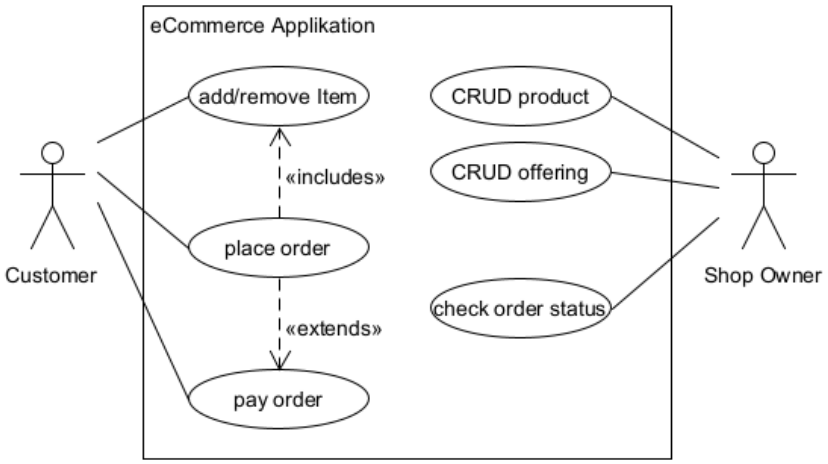

Figure 2 Use Case Diagram

The customer and the shop owner are actors in the application. In the eCommerce application, the customer should be provided with the function of adding and removing items from his shopping cart. In addition, the customer should be able to purchase items in the form of an order, which can then be paid for. There the customer has the possibility to pay directly online or by invoice.

The right side of the use case diagram shows the business events of the shop owner. He should be able to create, delete or edit products. He should also be able to create offers from the products. An additional function that should be provided for the shop owner is the ability to view the status of a customer's orders.

3.2 Derive architectural properties from domain-specific requirements

To make meaningful decisions, domain-specific goals and needs must be translated into architectural properties. In the following section, the domain at hand is examined for

these properties. Ultimately, the most important factor in an eCommerce application is the user experience (UX). However, in today's fast-paced environment, it is difficult to achieve a good UX. The architectural properties studied for this use case ultimately lead to an improvement in UX (Vogels, 2006, FX, 2022, Zenith, 2020).

A good UX in eCommerce is created especially by the successful completion of a purchase process. The final screen „Thank you for shopping“ creates a sense of accomplishment for the user. Since errors in a communication channel are fundamentally not completely avoidable, a procedure must be in place to deal with errors. If an error occurs, the user must be informed immediately. As soon as the user can no longer be sure that the process has really been completed successfully (for example, if he receives an email message about an error after the fact), this feeling is lost. Once a user has had the experience that the process has not really been completed in reality, he is conditioned not to expect this the next time. In this way, the inhibition threshold to order something is greatly increased.

Due to the wide range of eCommerce applications, small inconveniences can have a direct effect on the profitability of an eCommerce application. If the application is not available or an error occurs during a transaction, as a result of which a customer has to wait or perform other additional actions, in most cases the customer has the option to fall back on another eCommerce application. Accordingly, performance in terms of response time and availability is an important characteristic for an eCommerce application.

In a software application, errors of any kind should be avoided as far as possible. Especially in eCommerce, there are always users who have a special interest in exploiting errors that occur to their advantage. Good testability is therefore an important feature. To ensure the successful outcome or termination of a transaction, it is necessary, for example, to check whether either the monetary consideration has been paid or the product is back in the system in order to rule out any negative effects on the company. Testability therefore goes beyond testing the actual functionality.

Nowadays, software solutions are getting bigger and bigger, including eCommerce stores. Especially in an eCommerce application, maintainability plays a very important role, as the application must be easily modifiable to fix errors or adjust attributes. In this way, it is possible to react quickly to problems, which usually have an immediate economic impact.

Users of an eCommerce application perform payment transactions, among other things. Any error at this point leads directly to real consequences in the form of money loss. This makes the issue of data loss an important feature of an eCommerce application.

A modern corporate structure is often based on the principle of expandability in order to offer live service products. Accordingly, it is very important nowadays to be able to follow current trends and developments in order not to be left behind by the competition. It is also often the case with companies that when you grow, the resources gained should be reinvested in the company. In this case, it is a great disadvantage if the application is difficult to expand and thus the growth of the company would be held back.

The requirements for an eCommerce application are very dynamic, so it should always have good scalability. For example, during the Christmas season, significantly more orders

are placed, while shortly after the Christmas season a high number of complaints arise, so it is advantageous to be able to dynamically adapt the number of replications of services to the requirements.

In many cases, a user's actions trigger processes in the background that the user should not have to wait for. For example, when a user adds an item to the shopping cart, he or she should not have to wait for an archiving process to be completed. Good responsiveness is therefore also a feature of the eCommerce application that needs to be investigated.

In case of a change of data, all corresponding modules must be informed about this change in order to keep the data consistent. Otherwise, for example, an incorrect stock of goods would be displayed or, after a change of address by the customer, the goods might be delivered to an incorrect address when an order is placed.

If an error occurs in a service, it is important that the entire application does not block or collapse (with reference to statements such as those made by Werner Vogels, see section 1). Accordingly, the resilience of the application is an important property of an eCommerce application.

In addition to the technical features, the pure business aspect also plays a role. Technologies used always require a corresponding competence, which means that someone must be paid to implement functionalities. The competencies required and the time needed to implement a functionality depend on the technology used.

In summary, the following properties of the application are now to be examined:

- Error Handling
- Performance
- Availability
- Testability
- Maintainability
- Data Loss
- Extensibility
- Scalability
- Responsiveness
- Consistency
- Resilience
- Business Aspect

3.3 Tradeoff ASYNC

In the following, the tradeoff of asynchronous communication is examined based on the identified properties.

3.3.1 Error handling

„The main problem with asynchronous communication is error handling“ (Richards, 2020, p.201). When a sender sends a message asynchronously, it only receives a response that its message has arrived, a kind of promise that its request will be processed. After that, the communication channel to the client is closed, in contrast to synchronous communication. If further information has to be exchanged due to an error or similar, another communication channel has to be opened. However, if a problem arises with his request, he himself is initially unaware of it and must be informed via an external mechanism. With time-critical procedures this can be perceived thus as a disadvantage of asynchronous communication (Richards, 2020, p. 200).

3.3.2 Performance

In places where an operation must be performed as quickly as possible, synchronous transmission of the request is generally safer because the response is immediate. As already described in section 3.3.1, with asynchronous communication there can initially only be a promise that the request will be processed. However, in terms of overall processing time or performance, there is basically no significant advantage or disadvantage between synchronous and asynchronous communication as long as the number of threads is the same. While synchronous starts a thread for each request, which remains blocked until the answer is generated and sent, with asynchronous transmission all requests arrive in a queue and threads are generated internally in the program, which take these for processing. Since thereby the tasks do not change it has no substantial influence on the performance (illustrated in section 3.3.8 figure 3) (Richards, 2020, p. 200ff). This was confirmed e.g. also by Mikulich, 2021 in several experiments.

However, with asynchronous transfer there is an additional intermediate element (queue). This only has to lead to a transfer time (processing time) and thus to a deterioration in performance, albeit minimal, due to the operations added as a result with the same number of threads. In complex microservice applications, however, these minimal effects can add up and create noticeable impacts.

However, an event-based architecture can achieve high performance due to a combination of asynchronous messaging and highly parallel processing (Richards, 2020, p. 213). Consequently, performance may ultimately be perceived as an advantage of asynchronous communication, provided it is an event-based architecture.

3.3.3 Availability

A significant difference between synchronous and asynchronous communication arises at the point where the capacity limit of a service is reached. If no further thread can be started in the synchronous model, no more requests can be accepted. The caller and the called are connascent, that means the number of services, which receive data must at least correspond to the number of services, which send data (Richards, 2020, p.95). Asynchronous services are not connascent due to the use of queues, requests end up in a queue to be picked up later.

3.3.4 Testability/Maintainability

There is a significant difference in testability between synchronous (request-based model) and asynchronous (event-based model) communication. While deterministic processes are quite easy to test in the request-based model because the paths and results are known, the testability of non-deterministic processes and dynamic events is considerably more complex in the event-based model because it is sometimes not clear how services react to dynamic events and which messages are generated in the process. These "event tree diagrams" can turn out to be very complex and generate thousands of different scenarios, which are difficult to monitor and test (Richards, 2020, p.213).

3.3.5 Data loss

In asynchronous communication, data can be lost at various points. These places are typically:

- Path from an event processor to the queue, or the broker fails before the message reaches the next event processor.
- An event processor crashes after loading the message from the queue - but before processing it.
- Error with database connection

[Richards, 2020 p.205 Prevent data loss]

Data loss can therefore be perceived as one of the problems/disadvantages of asynchronous communication. With synchronous communication, there is always a direct wait for a response and it would be immediately noticeable if the data does not arrive. There are concepts to counteract the problem of data loss in asynchronous communication such as the workflow event pattern (described in Richards, 2020 p.202), but these are always accompanied by a performance penalty.

3.3.6 Extensibility

In an event-based architecture (asynchronous), if a new service or function is implemented in an existing service that requires a specific piece of information that already exists, it can simply be subscribed to the corresponding topic. With synchronous communication,

extending a new functionality is more difficult because synchronously communicating services must have information about the receiver or sender of the current data. Thus, all services that interact with the new service or function must first be informed about it. Accordingly, extensibility is an advantage of asynchronous (event-based) communication.

3.3.7 Scalability

Due to the connascence described in section 3.3.1, synchronously communicating services can only be scaled in dependence on each other, i.e., they can be scaled meaningfully only in a certain relationship to each other. In contrast, asynchronously communicating services can be scaled independently by using a queue or an event-based broker.

3.3.8 Responsiveness

In synchronous communication, a connection is established from the sender to the receiver and a request is sent. The connection remains open until the corresponding response has been sent. The sender must therefore wait until the receiver has performed all the necessary steps to be able to respond. In the asynchronous model, on the other hand, the connection is closed as soon as the sender has transmitted his request. Thus, it is only active for the duration of the transmission and there is no additional active waiting time. Figure 3 shows a comparison using an event-based architecture as an example. Accordingly, responsiveness can be perceived as a strong advantage of asynchronous communication. (Richards, 2020, p.200)

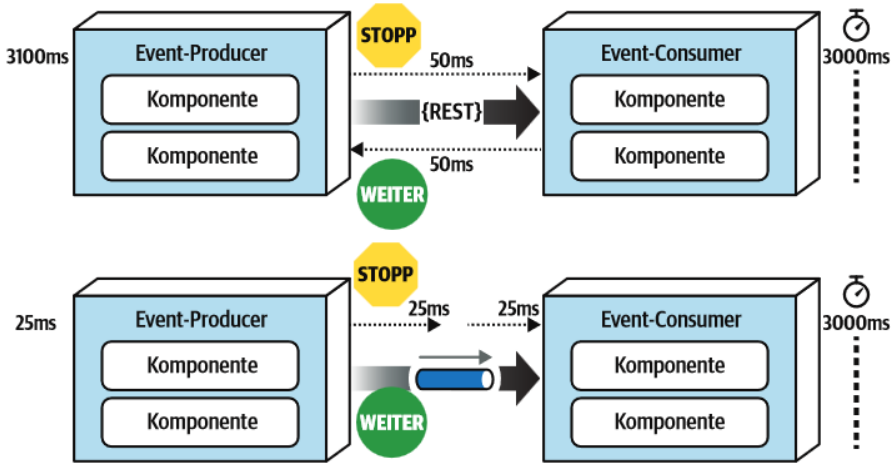

Figure 3 Responsiveness between two services with a async communication (Richards, 2020, p. 201)

3.3.9 Consistency

Since, as described in the sections 3.3.1 and 3.3.5, error handling and data loss can be a problem with asynchronous communication, this also endangers the consistency of the data. Precautions and control mechanisms must exist, which lead to eventual consistency and in turn cost computing power. Since with synchronous processing the data is kept directly consistent, this is to be noted as an advantage of synchronous communication.

3.3.10 Resilience

Because the sender and receiver must always be known in synchronous communication, all other communication partners must be informed if one of these services fails. At a certain level of complexity, this is difficult to implement and not particularly practical. If there is a communication chain in which several services are involved, synchronous communication also runs the risk of this chain being completely blocked in the event of a failure. Asynchronous communication therefore significantly increases resilience.

3.3.11 Business aspect

For asynchronous communication, additional competencies are needed in the team. Especially for an event-based architecture with e.g. Kafka, someone must be involved who is familiar with this technology. The implementation is also more time-consuming than the realization e.g. with a synchronously working REST interface.

4 Decisions

In the following, decisions regarding the communication between the services of the eCommerce application as well as with clients are made and justified on the basis of the available business events. The different communication channels are shown in Figure 4.

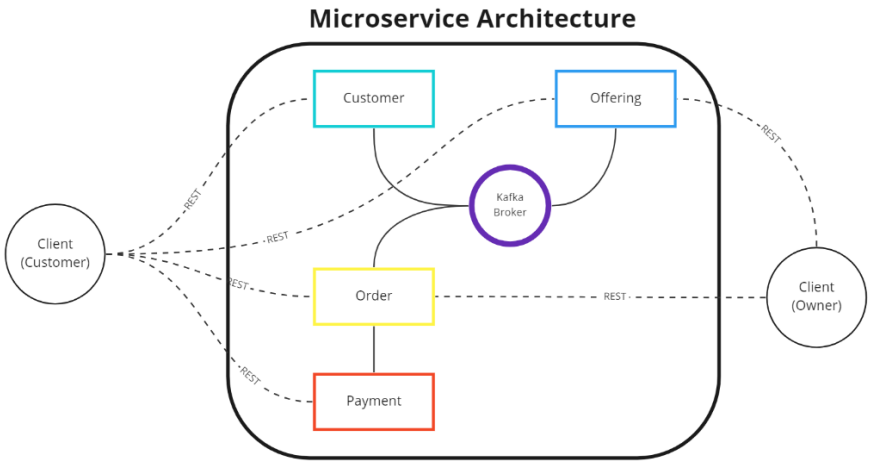

Figure 4 Microservice Architecture

For a decision, the following circumstances or guiding questions are considered in addition to the tradeoffs identified:

- frequency of operation
- duration of the operation
- Importance of the order of operations
- Are other services in a communication chain blocked?
- Does a service need a direct response to continue?
- Are there multiple receivers for the sender?

In cases of asynchronous communication, an event-based architecture with a broker is used. For synchronous communication between clients and services, REST interfaces are used.

4.1 Shop owner adds, updates or removes product or offering

In this case, there is communication between the client and *Offering*. Since this process occurs comparatively infrequently and there is no other chain of communication, neither responsiveness nor performance is particularly important. Asychronous communication does not provide any advantages at this point. The synchronous approach is therefore chosen here, which is easier to implement, test and maintain. Synchronous communication also makes it easier to guarantee data consistency.

4.2 Customer adds or removes item to/from shopping basket

Here there is a communication between client and *offering*, and between *offering* and *customer*. First, the Offerings have to be displayed in the frontend of the client. To do this, the information from *Offering* is transferred to the client. If the client receives a request from *Offering* to add a particular item to the shopping cart, *Offering* communicates with *Customer* to fill the shopping cart corresponding to the customer. This process is very common as many customers are viewing offers and processing their shopping cart at the same time. Good responsiveness and performance are important so that items can always be quickly displayed to the customer. In addition, scalability is very important at this point, as requirements vary greatly depending on the season. So everything speaks for asychronous communication between *Offering* and *Customer* at this point. The communication of the client with *Offering* still remains synchronous.

4.3 Customer places order

For this event, there is communication between the client and *Customer*, and between *Customer* and *Order*. In this business event, the client communicates primarily with the customer service. The customer is in his shopping cart and now wants to place the order. To do this, a request is sent to the Order Service. The order contains the items specified by the client. This process is performed frequently and by many clients, so it is advantageous to ensure a good response time as well as performance. Furthermore, a good availability as well as scalability is important, since the requests are seasonally dependent and fluctuate strongly, similar to the use case „Customer adds or removes item to/from shopping basket“. For the reasons described above, asynchronous communication is chosen and the tradeoff for testability and maintainability is accepted in this case.

4.4 Customer pays order

The payment transaction takes place between the client and *Payment*. The client selects its payment transaction and contacts the *Payment* service through it, which confirms the transaction and forwards this to *Order* to complete the order. Both of these communication channels are synchronous, due to the high relevance of this process and possible direct monetary damages in case of data loss. In addition, the customer gets direct feedback on whether the process was completed successfully. Since this process cannot be divided into smaller subprocesses, a high degree of parallelization does not bring any great advantage in this case.

4.5 Customer does not pay order

The *Order* service has a mechanism which periodically checks for an order whether a payment is present or not. To do this, *Order* must communicate with *Payment*. In this case, data loss is not a major concern, as the process is repeated periodically, so if the data has not arrived before, it can be fetched on the next attempt. Also, this process is highly parallelizable since it involves many small requests that do not necessarily have to happen

in a specific order. For the reasons described above, asynchronous communication is used at this point.

4.6 Shop owner checks order status

The business event describes a manual status check of a particular order. In this case, there is communication only between the client and *Order*. The decision at this point behaves like „Shop owner adds, updates or removes product or offering“ and therefore synchronous communication is chosen for this event.

4.7 Decision Tree

With the information gained, this section attempts to develop an initial concept for a decision tree. Possibly, after further development, this could lead to a way to support the decision process between synchronous and asynchronous communication. The figures 5 and 6 show two possible decision trees or respectivly two parts of a single decision tree.

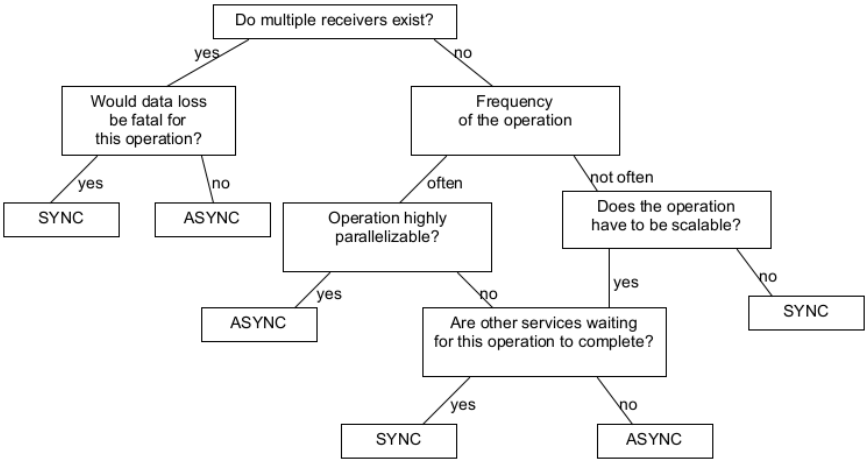

Figure 5 Decision Tree Guiding Questions

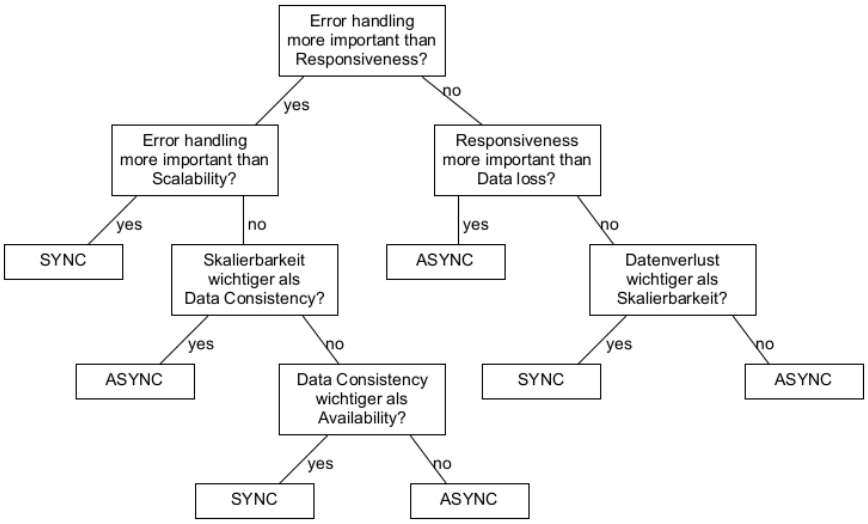

Figure 6 Decision Tree Properties

4.8 Preparing the Coding Exercise

With the information gained and decisions made, a coding experiment is now started, which is described in this chapter and in which the present eCommerce application is realized in a first version.

It was decided to implement the project with .NET Core (C#). For clients, a frontend with Angular is provided. For the asynchronous communication channels, Kafka is used as broker. The microservice architecture is containerized locally with Docker for simplification. Figure 8 shows the architecture considering the previous decisions.

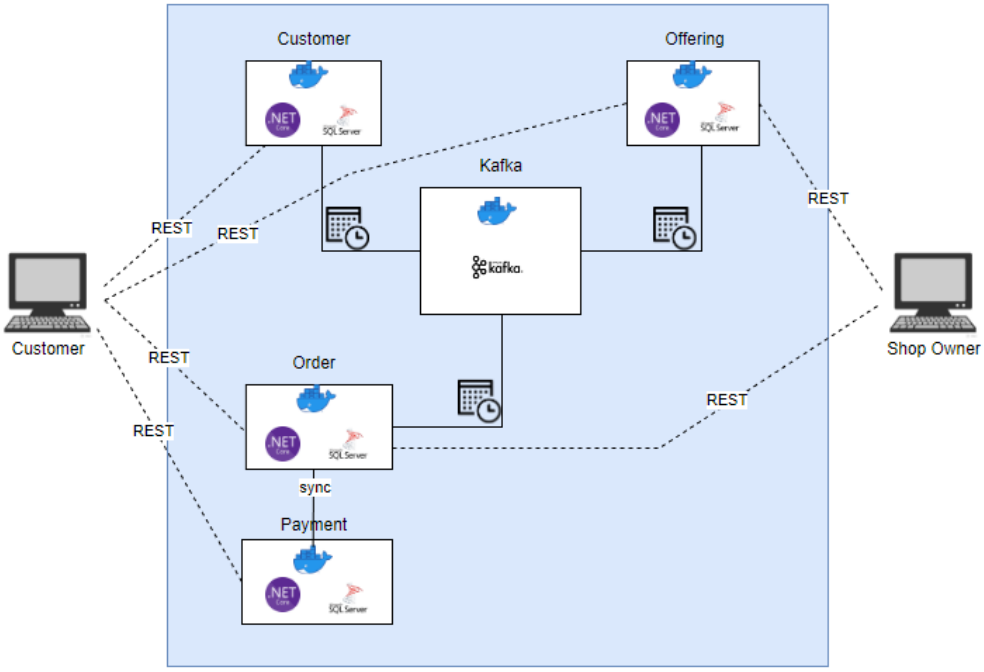

Figure 7 Microservice Architecture with broker

5 Coding Exercise and Evaluation

Now that the research is complete, the practical part of the module is tackled. For this purpose, the corresponding IDEs were installed and configured for the technologies to be used (see chapter 4.8).

During the development it turned out that it makes sense to adapt the planned architecture in certain points. A direct REST connection of the client to all services does not go along with the planned use of the event broker. Therefore, the architecture is modified in that the customer client communicates exclusively with the customer service. The customer service thus acts as a kind of API gateway and in turn communicates with the other services via the event broker.

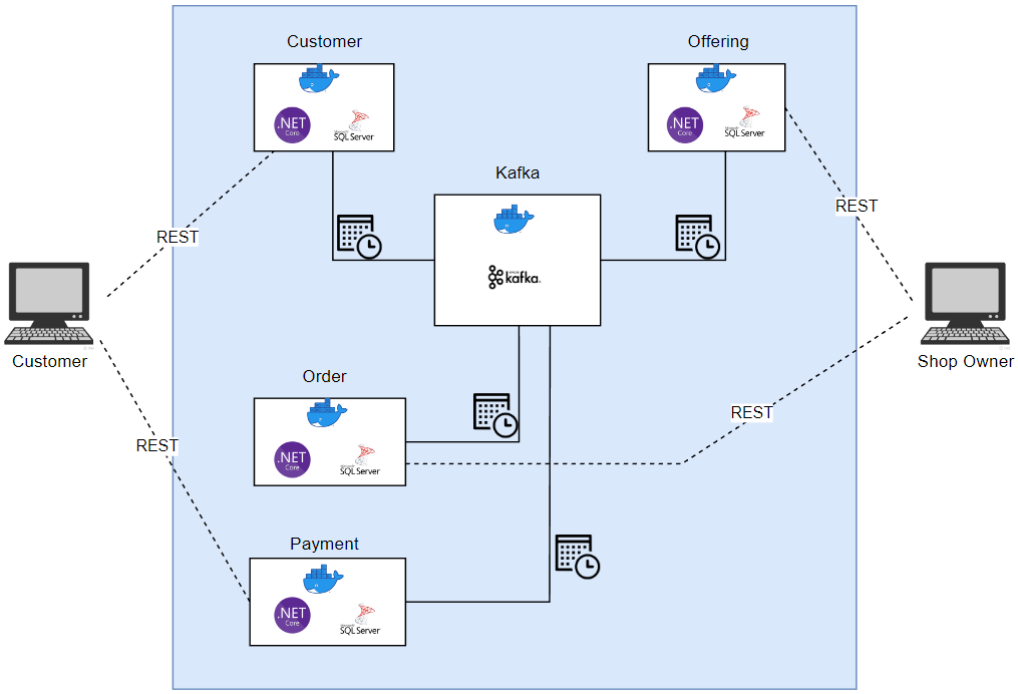

Figure 8 Microservice Architecture with broker

To evaluate the project, the following points are considered:

- To investigate the extensibility of the application, an additional event-based service is added.
- To evaluate the resilience of the application, a series of tests are performed in which targeted individual services are shut down to simulate a failure and determine the impact on the application's ability to run.
- To determine the effort, a developer diary is kept during the project, which contains tasks, times, lines of code and problem descriptions.

5.1 Extensibility

5.1.1 Adding an additional service: Order archiving

The event-based order archiving service is used to archive a history of all orders placed by the customer service. It listens for the Kafka channel where new orders are announced and adds the corresponding customer number and order date to a list for each order placed.

5.2 Resilience-Tests

5.2.1 1. Test: Order archiving service fails

In the event of a failure of the archiving service, the order will not be archived, but the order itself can be successfully executed regardless. The failure has no effect on the rest of the application. The order data is available in the corresponding Kafka channel and will be retrieved as soon as the archiving service is running again. Orders placed in the meantime are archived retrospectively.

If the communication between the customer service and the order archiving service were synchronous, an error would occur during the creation of an order in the customer service and the order process could not be completed successfully, even though the order itself had already been created.

5.2.2 2. Test: Order service fails

If the order service fails, customers can continue to place orders. The order data is stored by the customer service in the Kafka channel "order data" and retrieved by the order service as soon as it is up and running again. Only then is the order created. In the meantime, the customer service can continue to work normally. The order placed by the customer is already archived even though, strictly speaking, it does not yet exist. If a payment is created during the outage, the order service will receive the data from Kafka when it is operational again and will update the order status. If an order is created and paid during the outage, there is a risk of a race condition. It must be ensured that the order service first checks orders placed before processing payments.

If there was synchronous communication between the customer service and the order service, no order could be placed by the customer during the outage. The customer service would report an error when trying to place the order and the customer service would be blocked.

5.2.3 3. Test: Payment service fails

Since this is a synchronous communication between client and payment service, no payments can be made in case of a failure. Otherwise, there is no further impact on the application.

5.2.4 4. Test: Offering service fails

For the shop owner functions (create/remove products and offerings), there is synchronous communication between the client and the offering service. The shop owner functions can therefore not be executed in the event of a failure of the offering service. Furthermore, the customer service can no longer call up the offerings, which means that a customer has no way of displaying the offerings or adding them to his shopping cart. Since the communication between the customer service and the offering service is request/reply-based via Kafka, the customer service's request is stored in the corresponding Kafka channel. However, the response in the form of the required Offerings will not be sent until the Offering service is running again. So it does not cause an error in the customer service but the Offerings will not be displayed during the outage of the Offering service. If the Customer Service has already requested and received the Offerings before the outage, it will have its own Offering list. This (not current) list would be able to be displayed to the client in this case. This creates the risk that a client will see or add to the shopping cart an Offering that may no longer exist.

In an event-based model, the offering data could be stored redundantly in the customer service. When a new offering is created, the information would be stored in a corresponding Kafka channel. From there, the customer service could take all the required information of the created offering and maintain its own reduced database of offerings ("Better have data than need data"). In this model, the customer service would be independent and could continue to work even in the event of a failure of the offering service. However, with a large assortment, i.e., a large volume of Offerings data (e.g., Amazon), this would mean a very large redundant volume of data.

5.2.5 5. Test: Customer service fails

Since the customer communicates directly with the customer service as a client, he cannot access the application in the event of a failure. The customer service serves as an API gateway, so to speak. The only function that remains available is "pay order", as this is implemented by a separate REST call to the payment service. The shop owner functions are also still available and are not affected.

The direct communication of the client with the application should be synchronous, since the customer must receive a direct response. Therefore, asynchronous or event-based communication is not useful at this point.

5.2.6 6. Test: Kafka fails

Only the shop owner functions are still available, as they are the only ones that do not involve communication via Kafka. All other functionality of the application fails. If Kafka is already down when the services are started, errors occur directly because the Kafka Topics are not accessible.

5.2.7 Resilience Tests Conclusion

In figure 9 the results of the resilience tests are summarized in tabular form.

Function / failed service	Customer	Offering	Order	Payment	Archiv	Kafka
getAllOfferings	✗	◷	✓	✓	✓	✗
getShoppingBasketItems	✗	✓	✓	✓	✓	✓
addToCart	✗	✓	✓	✓	✓	✓
removeFromCart	✗	✓	✓	✓	✓	✓
createCustomer	✗	✓	✓	✓	✓	✓
placeOrder	✗	✓	◷	✓	✓	✗
payOrder	✓	✓	◷	✗	✓	✓
gettAllOrders	✓	✓	✗	✓	✓	✓
deleteOrder	✓	✓	✗	✓	✓	✓
createOffering	✓	✗	✓	✓	✓	✓
createProduct	✓	✗	✓	✓	✓	✓
getAllProducts	✓	✗	✓	✓	✓	✓
deleteProduct	✓	✗	✓	✓	✓	✓
deleteOffering	✓	✗	✓	✓	✓	✓
order is archived	✓	✓	✓	✓	✗	✗

✓	runs
✗	error
◷	delayed

Figure 9 The figure shows the runnability of the individual functions in relation to crashed services, where the x-axis contains the service that crashed and the y-axis describes the behavior of the functions. Runs means, the function continues to run without degradation, Error means, the function is no longer available. Delayed means, the function can be executed, but the processing of the data happens only as soon as the corresponding service is running again.

5.3 Effort

This section deals with the process of the coding exercise, problems encountered during the execution as well as the time spent and the amount of code generated for each element. During the course of the exercise, a detailed developer diary was kept, written in german language, in which the individual steps and considerations were recorded (see Appendix A.1).

Initially, a simple test project was created with 2 services communicating using Kafka (1 producer + 1 consumer, 1 Kafka topic). Initially, several problems arose with the use of Kafka, especially when trying to configure a service to be both a consumer and a producer of events. The solution to this problem in this case was to use threads.

At the beginning of the project, various approaches were attempted to implement the microservice architecture in conjunction with a message broker. These included two different Kafka variants (Docker images), various approaches in .NET, and an approach in Python.

The implementation of the Phython application was initially faster and easier to realize because it is a very easy-to-use scripting language. In terms of modularity as well as file structure and file management, .NET delivered clear advantages thanks to its dedicated development environment with corresponding features. With .NET Core, however, a suitable version first had to be agreed upon and the IDE of the other team members had to be adapted accordingly. This led to various version and package conflicts in the first steps. In .NET Core, IDE-related variables or connection strings for databases are swapped out in a so-called *appsettings.cs*. This mechanism formed an additional advantage to work together on the project from different machines.

One problem with using Kafka was monitoring. While one of the variants of Kafka used included many additional features, including its own Control Center, this same multitude of functionality created additional overhead and problems with error-free use. The other Kafka variant, on the other hand, was very lean and easy to run in comparison, but did not provide an accompanying way to monitor the various topics. The solution was a self-made Python script, which subscribes to all available topics and outputs the content to the console.

The plan was to use Angular to create a frontend to execute the various business events in the form of functions and visualize the results. This involved familiarization with the corresponding framework, which turned out to be time-consuming and effort-intensive. The creation of the frontend was therefore ultimately discontinued in order not to exceed the time scope of the project.

While comparatively little code was produced at the beginning of the project or discarded at the end, it was possible to develop more and more efficiently from step to step as the project progressed. While typical REST calls could be implemented easily and without problems from the beginning, the development of a service that communicates via Kafka was much more difficult at the beginning. As soon as the principle of producer and consumer was understood and functionally implemented, the creation of further services and

functionalities became much easier.

The exact descriptions and timings as well as lines of code for the individual work steps are listed in the developer diary (see Appendix A.1). A summary of some essential work steps can be found in the following table:

Task	Time in Mins	Lines of code
Testproject	107	80
First .NET App (customer, offering)	1078	233
Python Application (client, customer, offerings, monitor)	407	269
.NET order-service	52	89
create/delete product/offering	40	53
Payment-service incl. event processing	33	36
Order-archive-service incl. test	28	61
Split project into individual projects	60	-

The following tables show the final scope of the project based on the included files, as well as the breakdown of the time spent on coding, research and testing:

File	Lines of code
Product class	10
Payment class	5
Order class	8
Offering class	10
Customer class	9
Customer service	87
Offering service	110
Order archive service	61
Order service	120
Payment service	36
Customer controller	120
Order controller	32
Payment controller	32
Product controller	53
Program.cs	30
Total	723

Table 1 Final project size in lines of code

Framework	Time
.NET	28.8 h
Python	6.7 h
Angular	5.5 h
Total	41 h

Table 2 Pure coding times

Task	Time
Coding	41 h
Research	5.5 h
Test	2.05 h
Total	48.55 h

Table 3 Workload

With the data obtained, the average productivity is 14.8 lines of code per hour, resulting in approximately 118 lines per 8-hour workday. On average, a programmer produces 10-50 lines in 8 hours. It can therefore be seen that, as soon as a functioning infrastructure is in place, above-average productivity can be achieved.

5.4 Evaluation Conclusion

During the resilience tests, in which individual services were shut down, it became apparent that if a service operates exclusively event-based, the impact on the rest of the application resulting from a failure can be minimized or completely prevented.

Due to the given event-based part of the infrastructure, additional event-based services can be implemented with comparatively little effort.

By using asynchronous communication with the help of a message broker, functions that would block or crash if a service fails can be processed retrospectively at the moment when the corresponding service is operational again. However, it also means that if the broker itself crashes, there is an impact on all services that need it to communicate.

One measure against the effects of service failures is to scale them. In addition, this provides protection against failures due to excessive load.

While productivity was comparatively low at the beginning, it increased steadily once the infrastructure was in place.

A particular problem in the implementation of an event-based architecture is the paradigm shift from a synchronous thought model (Request/Reply) to an event-based way of thinking. Communication can be implemented asynchronously and still be message-based with the request/reply pattern. It turned out to be remarkably difficult to change over mentally.

6 Conclusion

This work has confirmed that a decision on whether a service should communicate synchronously or asynchronously is not always easy or clear-cut. There will always be a trade-off. The most important thing is to know the requirements and limitations of its domain and incorporate these into the design decisions of the application.

Indeed, asynchronous communication offers significantly better resilience, scalability and responsiveness. Nevertheless, there may always be places where asynchronous communication cannot exploit its advantages. For example, if no communication chain exists, none can be blocked, or if a process only happens very rarely, scalability is not necessary. Since clients from the outside should never have to wait until related processes are completed in the background, synchronous communication is the best option at these points.

This work should help to see these requirements more clearly and to be able to make the decisions more easily. A useful extension of this work would be to develop two eCommerce applications. One of these applications would be developed completely with asynchronous methods and the other one completely with synchronous methods. This would allow a direct comparison of each communication channel, confirming or disproving the claims of this thesis through measurements.

So what is the answer to Mr. Vogels Question?

There are places where communication should be synchronous because an immediate response is needed and it only makes sense to deliver that response immediately. Outside of this, we agree with Mr. Vogels. We would even go further and say make it event based if possible. But do it where it makes sense and where you are willing to make the tradeoff. In the example of the present project, this was well illustrated by the handling of the offerings with the tradeoff of data redundancy.

7 Outlook

In a second version of the application, the client would not communicate directly with the respective service via REST, but via an API gateway that forwards the requests to the respective services. An asynchronous request/reply pattern is avoided; instead, the message broker is used exclusively for event-based communication. Each service has its own database and data redundancies are accepted. All communication channels where an immediate response is required are implemented synchronously with REST calls. Additionally the implementation of the planned frontend for the eCommerce application could be resumed.

Figure 10 shows an architecture in which this could be implemented:

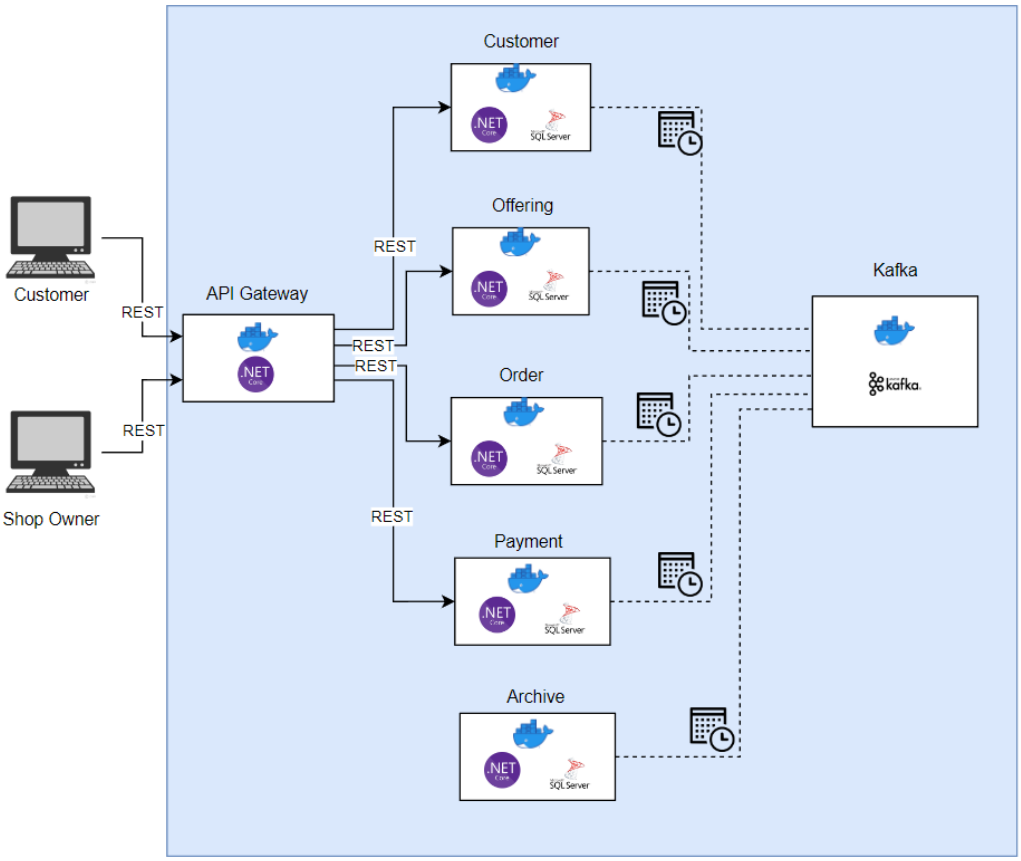

Figure 10 Future microservice architecture

References

FX, W. (2022). Why does ux matter for ecommerce? Retrieved November 25, 2022, from https://www.webfx.com/web-design/ecommerce/why-does-ux-matter-for-ecommerce/#:~:text=UX%20is%20critical%20to%20ecommerce,offers%20the%20best%20UX%20possible.

Hat, R. (2018). Was sind microservices? Retrieved November 10, 2022, from https://www.redhat.com/de/topics/microservices/what-are-microservices

Kröner, K. (2022). Was unterscheidet asynchrone und synchrone kommunikation? Retrieved November 25, 2022, from https://www.repetico.de/card-34295922

Microsoft. (2022). Communication in a microservice architecture. Retrieved November 25, 2022, from https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

Mikulich, A. (2021). Async vs sync benchmark (.net). Retrieved November 20, 2022, from https://artemmikulich.medium.com/async-vs-sync-benchmark-net-f1e752a57755

Richards, M. (2020). *Handbuch moderner softwarearchitektur - architekturstile, patterns und best practices*. John Goerzen, Brandon Rhodes. https://doi.org/10.1007/978-1-4302-3004-5

Richardson, C. (2021). Pattern: Messaging. Retrieved November 24, 2022, from https://microservices.io/patterns/communication-style/messaging.html

To, J. Y. (2022). Microservices architecture: Asynchronous communication is better. https://www.sysaid.com/blog/sysaid-tech/microservices-architecture-asynchronouscommunication-better

Vogels, W. (2006). A conversation with werner vogels. Retrieved November 25, 2022, from https://queue.acm.org/detail.cfm?id=1142065

Vogels, W. (2022). Werner vogels. Retrieved November 25, 2022, from https://en.wikipedia.org/wiki/Werner_Vogels

Zenith, V. (2020). Importance of ui/ux for e-commerce. Retrieved November 25, 2022, from https://www.linkedin.com/pulse/importance-uiux-e-commerce-viswam-zenith

Appendix

Appendix Material

.1 Developer Diary (devlogs)

```
ENTWICKLERTAGEBUCH
==================
------------------------------------------------------------
11.12.22 - 11:00 Uhr - Kafka Testprojekt

Testprojekt: 2 services kommunizieren über Kafka:
.NET 6
docker-compose
1 producer
1 consumer
confluent kafka
message sending test mit swagger

1.Test - Nachrichten mit swagger produzieren während beide services laufen:
1 nachricht kam durch, danach nichts mehr
services neugestartet
keine Besserung -> rechner komplett neustarten
1. nachricht kommt durch - danach nichts
swagger gibt 201 (created) zurück
docker container laufen (zookeeper + kafka)
nach docker neustart kamen die alten Nachrichten plötzlich an, neue testnachricht
 kommt wieder nicht an -> namespace bei consumer hinzugefügt, neustart von visual
 studio (consumer) scheint zu funktionieren, 3 Nachrichten hintereinander kommen an,
 kurzer idle,
 3 weitere Nachrichten kommen an [snip01] lag es am namespace?

2.Test - Consumer abschalten, 3 Nachrichten senden, consumer wieder hochfahren:
Erfolg, einige Sekunden nach dem Start erreichen die Nachrichten den Consumer
 (14:41 Uhr)




11.12.22 - 12:47 Uhr -  insgesamt ~80 Zeilen Code
1 producer, 1 consumer, kafka, docker-compose


------------------------------------------------------------
```

12.12.22 - 18:02 Uhr - Modellierung des Domain Message Flow Diagramms

-> Funktion zum Anzeigen aller Products/Offerings für ShopOwner?
-> Funktion zum Anzeigen der Offerings für Customer?
-> Was passiert wenn Payment beim Check kein payment date liefert?
(Kunde hat nicht bezahlt)
-> Shop owner checkt nur direkt über order id oder auch anhand eines customers?
(19:15 Uhr)

13.12.22 - 14:11 Uhr - Test 1+2 wiederholen: nur 1. msg kommt durch
offering neustarten - alle nachrichten kommen durch

Service: Offering -> Klassen: Offering (id, name, quantity, totalPrice, effectiveDate, status, product) + Product (id, name, color, description, price) anlegen

-> entweder nur price bei offering oder totalPrice und dann price bei product
Daten mocken, 2 products, 2 offering
(15:30 Uhr)

14.12.22 - 17:00 Uhr - Git Repo erstellen

14.12.22 - 17:00 Uhr - frontend mit angular aufsetzen (18:20 Uhr)

15.12.22 - 10:20 Uhr - angular components für frontend erstellen

15.12.22 - 11:20 Uhr - klassen anpassen (preis für product), refactoring

15.12.22 - 11:50 Uhr - GET Test-Message
Client -> Customer -> Kafka -> Offerings (erstellt message)
Offerings -> Kafka -> Customer -> Client

Customer+Offering sind gleichzeitig Producer und Consumer des selben Topics

Controller zum Senden, Handler zum empfangen

15.12.22 - 13:07 Uhr - GET offerings
Client -> Customer -> Kafka -> Offerings -> Kafka -> Customer -> Client
Kafka topic: offerings_ch

PROBLEM: simple_topic funktioniert, offerings_ch nicht, Fehlersuche
console logs, break-points
topics in der compose-file ändern
namen und namespaces prüfen
Leerzeichen in CREATE_TOPICS, reihenfolge ändern
kafka konsole meldet topics created, aber keine messages, wie monitoring?
Service CustomerConsumerHandler in Program.cs definiert
alle services neustarten
Reihenfolge der ConsumerHandler ändern
builder service für ConsumerHandler ändern von Singleton auf AddScope
topics tauschen, dann funktoiniert offerings_ch aber simple_topic nicht mehr
topics in verschiedenen variablen anlegen und auf beide subscriben
AddScope auf AddTransient ändern
nach wie vor funktioniert immer nur 1 topic
wieder zurück auf Singleton
LÖSUNG: 1 ConsumerHandler, topics als Liste, beide topics funktionieren
(15:01 Uhr)

15.12.22 - 15:03 Uhr - 1 producer, 1 consumer, 2 topics - 115+10+10+90+40 = 265 Zeilen Code

15.12.22 - 15:06 Uhr - Ziel: 1 producer, 1 consumer, 2 topics (1x "Jemand will alle Offerings", 1x "Alle Offerings")
Test: getMessage kommt rein, Test-Message geht raus an Kafka
getMessage kommt an, Test-Message geht raus, aber Test-Message kommt nicht
- beim Customer an Customer kann bisher nur senden, beim Versuch den Handler-Service zu implementieren startet swagger nicht mehr

Offerings: getOfferings kommt rein, liste von offerings geht raus an kafka

---

15.12.22 - 15:40 Uhr - Testprojekt mit Python
probleme mit netzwerkgeräten von virtualbox lösen (virtualbox
- neuinstallation)

kafka image, dockerfile und compose yml von wurstmeister
confluent kafka - certify
test mit 1 producer + 1 consumer + kafka
test erfolgreich
15.12.22 - 16:48 Uhr - Dockerfile, composefile, 2 python scripts ->
4 dateien mit 10+38+20= 68 Zeilen Code

15.12.22 - 16:51 Uhr - Customer+Offerings Kommunikation über Kafka

2 topics: offerings, offerings_data
Customer-Service sendet "customer needs offerings" an kafka(topic:
offerings),
Offering-Service consumed offerings-topic
Offering-Service sendet Liste mit Offerings (gemockt) an kafka(topic: offerings_data)
Customer-Service consumed offerings_data-topic
Customer-Service bekommt Liste von Offerings und gibt sie aus
dazu existiert 1 kafkaConsumer-Service welcher alle Topics subscribed
(monitoring)

PROBLEM: Anfrage kommt an, offering-liste erreicht den kafkaConsumer
aber nicht den Customer
LÖSUNG: mit producer.flush() sicherstellen dass daten raus gehen, funktioniert
(17:20 Uhr)
Aber ist die Lösung gut? Besser mit Threading?
Code Anpassen mit Threads
PROBLEM: offering-liste kommt nicht an
Kafka neustart, neue IP
ip von kafka selbst festlegen, im code anpassen
poll timeout entfernt
es dauert teilweise sehr lange (~30sek) bis nachrichten ankommen
kafka admin api research (um in kafka direkt reinschauen zu können)
problem bleibt bestehen: offering-liste erreicht den customer nicht

15.12.22 - 18:37 Uhr - Customer, Offerings, KafkaConsumer,
Composefile mit 51+10+37+52+20 = 170 Zeilen Code

16.12.22 - 10:11 Uhr - .NET-Projekt - offerings kommen nicht beim customer an
namespaces anpassen, refactoring
automatischer browser-launch auf false
namespaces entfernen

research: message-broker vs event-sourcing
partitionen, commits

16.12.22 - 11:50 Uhr - Angular Frontend, components (12:40 Uhr)

```
------------------------------------------------------------
17.12.22 - 12:00 Uhr - Angular Frontend, mock-data, offerings
Design, offerings (gemockt) werden gelistet, addToBasket-button (13:10 Uhr)


17.12.22 - 18:01 Uhr - Python Backend, async commits, message count
PROBLEM: offering-liste kommt bei monitoring an, aber nicht bei customer-service
einrückungen korrigiert
es dauert einige zeit (ca. 10sek), aber offering-liste kommt bei customer
service an
input für getOfferings im Customer-Service einbauen (nicht mehr
automatisch)
customer-service sendet "getOfferings" an kafka, monitoring und offering-
service bekommt Anfrage und sendet offering-liste an kafka, customer bekommt liste
LÖSUNG: kafka-gruppen angepasst, immer nur 1 aus einer Gruppe bekommt die
Nachricht
funktioniert (18:40 Uhr)
test wiederholen, alles neustarten -> funktioniert, keine verzögerung mehr
wichtig: warten bis kafka komplett initialisiert ist
github commit (18:47 Uhr)

17.12.22 - 18:56 Uhr - API Endpunkt mit Flask für customer-service
Test mit curl (getOfferings an customer-service) - funktioniert
return einbauen, offering-liste an client zurücksenden
Client (PythonClient.py) für http-request an die API
PROBLEM: Client bekommt leeres Paket
json nicht vor dem Senden an Client decoden
Problem mit event?
LÖSUNG: data variable global deklarieren
test wiederholen, funktioniert, sehr wenig delay
github commit (19:40 Uhr)

17.12.22 - 19:40 Uhr - 5 Dateien (compose, client, customer, offerings, monitoring)
mit insgesamt 20+15+76+57+43 = 211 Zeilen Code


------------------------------------------------------------
18.12.22 - 19:20 Uhr - Listen: Offering (id, name, quantity, (totalPrice),
effectiveDate, status, product_id) + Product (id, name, color, description, price)
anlegen
```

products[(1, "Pencil", "black", "Faber Castell Limited Edition HB", 4.95),
(2, "Toaster", "gray", "Miele Toaster Deluxe 600W", 49.90),(3, "Jacket", "red",
"Hally Hanson best Down", 109.99)]
offerings[(1, "Pencil (single)", 1, "18.12.2022", "in stock", 1),(2,
"Pencil (set)", 6, "18.12.2022", "in stock", 1),(3, "Toaster", 1, "18.12.2022", "not
available", 2),(4, "Jacket", 1, "18.12.2022", "in stock", 3)]

gemockte Daten werden in .txt-Dateien gespeichert
getOfferings auf das Auslesen der Datei anpassen

18.12.22 - 19:51 Uhr - put in shopping basket, architektonische Überlegungen
Customer-Client addToCart(offeringID, customerID) -> an Customer-Service -
> Customer-Service getOfferings() über Kafka (wie immer), Customer-Service schreibt
entsprechendes Offering anhand der Id in den ShoppingBasket des entsprechenden
Customers
besser: Customer-Service sendet getOfferingByID(id) und bekommt nur das
entsprechende Offering zurück
rename: getOfferings zu getAllOfferings (um Verwechslung zu vermeiden)
customers.txt enthält customer+shoppingBasket Daten
addToCart() schreibt offeringID in die entsprechende Zeile von
customers.txt
PROBLEM: offering wird nicht in Datei geschrieben, keine Fehlermeldung
prints einbauen für debugging
addToCart() findet customerID nicht
customerID als string lesen
LÖSUNG: jsonLine in anderer Variable speichern, Datei neu schreiben
kleiner \n fix
addToCart() funktioniert (21:04 Uhr)

pythonClient.py erweitern mit Menü für Funktionen (show/add/remove)
removeFromCart(itemID)

18.12.22 - 21:17 Uhr - Github commit, 5 Dateien (compose, client, customer,
offerings, monitoring) mit insgesamt 20+47+98+61+43 = 269 Zeilen Code

---

19.12.22 - 19:17 Uhr - customerclient: show shopping basket
! Designentscheidung: Customer-Service bildet die alleinige Schnittstelle
für Customer-Client, keine Verbindung mehr zu Offering-Service
customer-service und offering-service kommunizieren intern über kafka
(19:40)
siehe auch: event message flow model "Scenario: Customer adds item to

shopping basket"
event: z.B. offeringID oder name ändert sich oder wird gelöscht

---

20.12.22 - 11:50 Uhr - Angular Frontend, json-server (simuliertes backend)
button: AddToBasket
frontend design-überlegungen (2 views + navigation, warenkorb (mit remove button) und offeringliste (mit addToBasket button)) (13:00 Uhr)

20.12.22 - 14:27 Uhr - Kafka research, topics, partitions, consumer groups, replication factor
überlegungen zu kafka-funktionsweise und threads
(!) consumer-loop in eigenem thread
pause (15:45)

20.12.22 - 16:50 Uhr - neues projekt mit confluent kafka + controll center
2 services + kafka
producer sendet message, kommt in kafka an, kafka-logging um message einzusehen, noch kein consumer (17:30 Uhr)

21.12.22 - 13:30 Uhr - .NET backend (v2, weather) senden
2 topics (weather+confirmation), zweiter service soll mit "ok" antworten
kafka ist sehr schnell wenn alles läuft, aber vorher dauert es wenn topics angelegt/geändert werden bis alles initialisiert ist

---

21.12.22 - 14:00 Uhr - Meeting mit Michael
sind threads der richtige denkansatz? => beides geht aber threads sinnvoll
manche services in .net andere in python => eher abgeraten
informationen von consumer-thread in den main-thread
anderer ansatz: customer-service gibt statische antwort, danach (unabhängig) die nächsten schritte
wiederholbare tests schreiben (code- oder textform) und festhalten
kleine schritte machen
interne synchrone kommunikation der services mit socket? socket, pipes, in datei schreiben oder rest-endpoint, vielleicht sogar auch asynchron über kafka

(15:10 Uhr)

21.12.22 - 15:13 Uhr - kafka + 2 Services, beide producer und consumer, cunsumer-loop läuft in background thread, anderer thread produced
neu:
1.Service: produce im main-thread, consume-loop in background-thread
2.Service: consume-loop in main thread, bei event wird produce-thread gestartet
PROBLEM: 2.Service daten kommen nicht in kafka an
while-statement der consume-loop geändert (wartet nicht mehr auf eingabe)
while-loop entfernt (daten werden nun einmalig gesendet)
LÖSUNG: producer.flush() um zu warten bis nachricht raus ist bevor thread beendet wird
funktioniert (16:29 Uhr)
21.12.22 - 16:43 Uhr - (Service1,Service2,compose -> 84+74+170 = 328) eigentlich falsch die compose mit reinzurechnen, eigentlich spielt nur ne rolle wieviel wir gecoded haben, ändern!!

22.12.22 - 13:00 Uhr - .NET Backend, producer, customer-service, GetAllOfferings, GetMessage
customer-service mit api -> sendet an offering-serice ("alter" consumer wird umgebaut), swagger als client
ergebnis eines threads an main-thread zurückgeben
ergebnis an swagger senden
PROBLEM: erster durchlauf geht, bei der zweiten Anfrage wird scheinbar der thread nicht gestartet und swagger erhält nur die dummy-antwort
thread-liste implementiert
static thread-variable implementiert
git commit (customer-service (offering-controller), KafkaConsumerHandler(Thread), Consumer => 110+50+75 = 235 Zeilen Code)

22.12.22 - 13:05 Uhr - neubau im sinne des python-ansatzes, nun aber komplett mit .NET (in Windows) und API+Swagger, Kafka läuft in einer Linux-VM
klassen für offering und product
speicherung von offerings und products in dateien als json-objekt
problem mit rest-api (max retries received) (18:23 Uhr)

22.12.22 - 18:50 Uhr - .NET backend (orginal-Version)

Client -> Customer-Service - erhält http request, sendet an kafka in offeringRequest, hört auf topic offeringResponse
Offering-Service subscribed auf offeringRequest - sendet offerings auf topic offeringResponse
Customer-Service erhält Offerings sendet als http response an Client -> funktioniert! (19:03 Uhr)
PROBLEM: dauert sehr lange, irgendwo auf dem rückweg zwischen kafka und offering-serivice
scheinbar muss in der konsole des jeweiligen services eine taste gedrückt werden damit es weitergeht? manchmal?!
überlegungen zur persistenten speicherung (txt-datei oder sqllite) (19:15)

23.12.22 - 12:30 Uhr - klärende Gespräche zur Einigung welcher Ansatz final verwendet wird, ab jetzt Arbeit aller Teammitglieder an genau einem Projekt
.NET6, Visual Studio, Kafka in Linux-VM, 1 Projekt für alle Services (1 cs.-Datei pro Service), swagger für http-request
probleme mit sdk-versionen beheben, verschiedene pakete installieren
NET-SDK unterstützt .NET6.0 nicht, neues SDK
versionskonflikte zwischen visual code 2019 und 2022, Verwendung von 2022 löste die Probleme
customer-service, offering-service, getAllOfferings funktionert komplett (15:08 Uhr)

23.12.22 - 15:16 Uhr - github commit (customer-service, offering-service, customer-controller, program.cs -> 61+81+63+28 = 233 Zeilen Code)

WINTERPAUSE

08.01.23 - 12:50 Uhr - customer-klasse (int id, string name, list<offerings>shoppingBasket)
POST customer legt offering in warenkorb
createCustomer
Diskussion: sollte der client auf die verschiedenen Services zugreifen können oder hat er genau einen "Ansprechpartner"?
Nutzen wir Kafka falsch?
eigener payment service sinnvoll? (mit wolf klären)
Architektonische Überlegungen

removeFromCart() (14:50 Uhr)

08.01.23 - 14:52 Uhr - Angular Frontend repository
view für offerings und warenkorb (navigationsleiste)
design-überlegungen

08.01.23 - 15:23 Uhr - Backend: order-service, order-Klasse
place order [von client an customer-service] (order enthält customerID und offeringliste des warenkorbs)
client -> customer-service -> order-service
id-vergabe für orders
(16:15 Uhr)

08.01.23 - 16:16 Uhr - github commit (customer-service, offering-service, order-service, customer-controller, program.cs, order-klasse, product-klasse, offering-klasse, customer-klasse ->
96+81+81+99+29+8+10+10+9 = 423 Zeilen Code)

11.01.23 - 12:16 Uhr - Angular Frontend
mit backend verbinden (getAllOfferings)
Klick auf + eines offerings sendet customerID und offeringID ans backend -> addToCart(cID,oID)
Warenkorb anzeigen führt für jedes Item getOfferingByID() aus um an die Offering-Daten zu kommen.
Shop-Owner Funktionen möglicherweise nur über swagger (extra view für shop owner auf die optionale liste).

11.01.23 - 13:05 Uhr - .NET Backend showOrders()
getOfferingByID()

(14:14 Uhr)

11.01.23 - 14:20 Uhr - Meeting mit Wolf und Michael
beide uml (vorher/nachher) vergleichen, wenn etwas ausfällt, passiert...
(restfull vs kafka)
einzeln durchspielen: order-service fällt aus, offering-service fällt aus, ...
analyse auf resilienz-verbesserung

"richtig"-eventbasiert (z.B. Order-Archivierungsservice)
ist event-kafka besser als request/response-kafka
selbst große architektur-teams machen ähnliche "fehler" wie mir, synchrone
denkweise vs event-denkweise , paradigmen-wechsel/denkmodell
-> resilienzfrage beantworten
-> archiv-service
-> vogel-frage
-> weg über rest/synchron über kafka-request/response zu wirklich event-
basiert
order und payment sind entkoppelt, microservice-bedingt trennen,
kundendaten getrennt von zahldaten, separation of concerns
order+payment auch asynchron
(16:11 Uhr)

11.01.23 - 16:12 Uhr - .NET Backend übertragene Funktion verursacht Fehler gefixt.
(16:21 Uhr)

13.01.23 - 10:10 Uhr - crud offering/product (create und delete) (10:50 Uhr)
payment service -> nimmt orderID entgegen und erzeugt payment (zusammen
mit date) -> sendet event an kafka dass orderXY bezahlt wurde
order-Service subscribed auf payment-channel, kommt etwas rein wird die
entsprechende order auf "paid" gesetzt
(11:23 Uhr)

14.01.23 - 10:40 Uhr - Überlegungen zur Erweiterung eines Event-basierten Services
Order-Archivierung: hört auf kafka order-channel und zieht sich nur die
orderID+customerID+Datum (10:57)
Order-Archiv-Service: 61 Zeilen Code
Archiv-Service Test erfolgreich (11:08 Uhr)

Finaler Umfang:
Product-Klasse -> 10 Zeilen
Payment-Klasse -> 5 Zeilen
Order-Klasse -> 8 Zeilen
Offering-Klasse -> 10 Zeilen
Customer-Klasse -> 9 Zeilen
Customer-Service -> 87 Zeilen
Offering-Service -> 110 Zeilen

```
Order-Archive-Service -> 61 Zeilen
Order-Service -> 120 Zeilen
Payment-Service -> 36 Zeilen
Customer-Controller -> 120 Zeilen
Order-Controller -> 32 Zeilen
Payment-Controller -> 32 Zeilen
Product-Controller -> 53 Zeilen
Program.cs -> 30 Zeilen
---------------------------------
    -> 723 Zeilen Code Total

(11:33 Uhr)
-------------------------------------------------------------


14.01.23 - 11:39 Uhr - Überlegungen zur Evaluation:
Unterscheidung zwischen "(lupenrein) Eventbasiert", "Pseudo-Eventbasiert"
und sync

PROBLEM: Services laufen in einem Projekt und können nicht unabhängig
voneinander gestoppt und gestartet werden
ANSÄTZE: Globale Variable die zur Laufzeit geändert wird
 CheatEngine
 Auskommentieren der StartService() zur Laufzeit
LÖSUNG:  getrennte Projekte für Services (git commit 12:40 Uhr)
 ports anpassen, kleinere fixes
(13:07 Uhr)


-------------------------------------------------------------

17.01.23 - 14:27 Uhr - Planung der Tests für Evaluation:

1. Test:
Order-Archiv-Service fällt aus:
-> Keine Auswirkung, der Rest läuft ganz normal weiter, alle Funktionen
verfügbar, Kafka speichert die Daten, sobald Archiv-Service wieder da erhält er die
Daten
-> wäre es synchron, gäbe es beim Versuch eine Order zu erstellen im
customer-service einen fehler oder er wäre blockiert (error/timeout/retry)

2. Test:
Order-Service fällt aus:
```

-> Order wird erst erstellt, sobald Service wieder läuft, kafka verwahrt die Daten, Customer-Service läuft normal weiter
-> wird während des ausfalls ein payment erstellt bekommt order-service die info von kafka wenn er wieder oben ist
-> wäre es sync, fehler im customer-service, orders können nicht erstellt werden, customer-service blockiert

3. Test:
Payment-Service fällt aus:
-> (synchron zwischen client und payment-service) "pay order" kann nicht ausgeführt werden, fehler beim client, sonst keine auswirkung

4. Test:
Offering-Service fällt aus:
-> shop-owner funktionen (synchron client+offering-service) können nicht ausgeführt werden
-> "get all offerings" funktioniert nicht, offerings können beim client nicht geladen werden
-> event-basiert könnten (z.B) neue products erstellt und in kafka zwischengespeichert werden
-> offerings könnten redundant im customer-service gespeichert werden (event-basiert könnte customer-service auf den channel subscriben in dem neue offerings gepostet werden)

5. Test:
Customer-Service fällt aus:
-> (customer-) Client kann nicht auf Applikation zugreifen da customer-service quasi als API-Gateway dient
-> event-basiert hier möglich/sinnvoll?

6. Test:
Kafka fällt aus:
-> Auswirkungen auf sevices die an kafka senden wollen?
-> "getAllOfferings", "placeOrder", "payOrder"

(15:00 Uhr)

17.01.23 - 15:01 Uhr - Evaluations-Abschnitt im Paper aufsetzen und erste Texte (15:50 Uhr)

```
------------------------------------------------------------
18.01.23 - 11:30 Uhr - Durchführung der Resilience-Tests (siehe oben) und
Überlegungen
1. Test (archivierungs-service fällt aus):
- Archivierung wird wie erwartet nachgeholt (quasi null wartezeit)

2. Test (order-service fällt aus):
- Order wird archiviert obwohl sie strenggenommen noch nicht exisitert!
- sobald order wieder läuft wird order erstellt wie erwartet (quasi
null wartezeit)
- kommt ein payment in der zeit des ausfalls wird es später
nachgetragen ('paid') wie erwartet
- sollte eine order während des ausfalls erstellt und bezahlt werden -
> RACE CONDITION!!!

3. Test (payment-service fällt aus):
- payment kann vom client nicht durchgeführt werden, sonst keine
auswirkung wie erwartet

4. Test (offering-service fällt aus):
- keine shopowner funktionen
- getAllOfferings gibt "alte" liste falls customer-service bereits
offerings angefortert hat (customer-service speichert sich eigene liste)
- im falle einer nicht aktuellen liste sähe der client unter umständen
offerings, welche z.b. nicht mehr existieren

5. Test (customer-service fällt aus):
- kein zugang zur applikation
- only payments möglich
- shopwoner funktionen möglich

6. Test (kafka fällt aus):
- shopowner funktionen gehen
- timeout für quasi alles andere

(13:33 Uhr)

18.01.23 - 13:33 Uhr - Übertragung der Testergebnisse ins Paper
(14:39 Uhr)

------------------------------------------------------------
18.01.23 - 15:20 Uhr - Strukturierung des Papers
```

Evaluierung des Tagebuchs
Tabelle: ausgefallener Service -> Funktionen
(16:37 Uhr)

19.01.23 - 11:35 Uhr - openapi spec file(s) für den anhang
wo kommen schemas her?
asyncapi files(?)
(12:05 Uhr)

19.01.23 - 12:42 Uhr - Paper Abschnitt über Zeiten/Code/Probleme
zeiten erfassen für verschiedene schritte
(14:15 Uhr)

26.01.23 - 11:20 Uhr - Paper Abschnitt über Zeiten/Code/Probleme
(14:30 Uhr)

27.01.23 - 11:00 Uhr - Paper Conclusion, Outlook
Architektur v2 Diagramm
Übersetzungen
Tagebuch Anhang
(14:11 Uhr)