

Einstieg in die Implementierung

MEA, Workshop 3, 29.10.2026

Der Tag

10:00	Ankommen und Überblick
10:15	AsyncAPI live, danach Contracts am Bildschirm
11:30	Walking Skeleton: der reduzierte Ablauf der Fallstudie
11:45	Die Probe
12:00	Mittagspause
12:45	Event-Katalog: die Grenzen, dann der Happy Path
14:30	Clients
15:45	Einstieg ins Coding - entwickeln und testen ohne Infrastruktur
16:45	Ausblick und Hausaufgabe

Walking Skeleton: was ist drin?

Wir lesen die Fallstudie noch einmal, mit Markierungen:

- normal: im Walking Skeleton
- orange: im Walking Skeleton, aber vereinfacht (simuliert, Testdaten)
- ~~durchgestrichen~~: nicht im Walking Skeleton

Fallstudie: How to Visit the Restaurant

Guests can use the Delicious App to search for a nearby restaurant, select it by clicking, and then book seats at a table. They need to specify the day, time, and the number of persons. The app then either confirms the reservation and displays the table number or asks the customer to select another time or day, or change the number of persons. Guests can only reserve timeslots that are more than 2 hours in the future.

Alternatively, guests can walk in without a reservation. Every seat at a table has a QR code printed on it, with the table and seat number. A reservation blocks the reserved seats for 90 minutes from the reserved time. Once a guest has ordered food, their seat is taken for 90 minutes from the order, with or without a reservation. Walk-in guests can only sit down on a seat that is free for the next 90 minutes; when they scan the seat's QR code, the app tells them whether it is.

If guests with a reservation have not ordered any food 30 minutes after the reserved time, the reservation lapses and their seats are free again.

Fallstudie: How to Order Food

There are no waiters in the restaurant. The only way to order food is via the Delicious App. Every guest orders (and pays for) their own meal; there is no ordering for a whole group.

A guest first needs to scan the QR code on their seat. Then they select food items from a "menu catalog" and pay via credit card in the app. The app then displays an estimated waiting time, which is updated every minute. If no cooking station is manned at that moment, the app does not accept the order. Nor does it accept an order when the seat has an upcoming reservation and the time left is too short for preparing and eating the food.

Fallstudie: How the Food is Prepared

For cost reasons, the cooks at Delicious restaurants do not need extensive training. The kitchen is organized into a number of "cooking stations." Each station contains a stove, the necessary cooking utensils, and space for used pots, spoons, etc. The used utensils are cleared regularly by a kitchen helper who puts them into the dishwasher and brings clean utensils to the cooking stations. This process does not require any digital support and is not modeled in the restaurant system.

Cooks shouldn't use their smartphone for hygiene reasons. Therefore, each cooking station has a **Cooking Station Terminal: a large monitor above the stove and a big mechanical switch that the cook can operate even with greasy hands**. The system assigns each food order to one of the manned cooking stations. When the cooking station receives a food order, the corresponding recipe is fetched and displayed in easy steps on the monitor.

The cook working at the station fetches the ingredients from the adjacent storage/cooling room and starts cooking. **At the beginning of the cooking process, and after each recipe step, the cook presses the switch to move on to the next step.** (This information is used in the waiting time estimation for the guest's app.) After the last step, the monitor tells the cook in which compartment of the pickup station (see below) the food goes. Once the food is placed there, the cooking station is free to receive the next preparation order.

Fallstudie: How the Food is Picked Up

There is no counter where cooks hand over food to the guests. Instead, between the kitchen and the guest area, there is a **Pickup Station**: a wall of numbered, heated compartments. Each compartment has two doors, one towards the kitchen and one towards the guests. The cook puts the finished food into the assigned compartment through the kitchen door and closes it again.

As soon as all dishes of an order are in their compartments, the **Delicious App** of the guest who placed the order displays a notification and a QR code. The guest goes to the pickup station and scans the QR code there. The guest doors of the matching compartments open, and the guest takes out the food. When both doors are closed again, the compartment can be used for the next dish.

If food is not collected within 15 minutes, or if a cook has finished a dish and no compartment is free, the manager gets a notification on the **Admin Client**. The manager then clears compartments by hand. The **Admin Client** shows who ordered which dish at which seat, so the manager can bring the food to the guest personally if need be.

Fallstudie: How the Storage Room is Managed

Each recipe lists the ingredients it needs, and the system keeps track of what is in the storage/cooling room. As soon as a guest has ordered a dish, its ingredients are reserved in the stock, and when the cook prepares the dish, they are taken out. If the stock for an ingredient runs low, the system sends an alarm to the manager's Admin Client. If the stock that is not yet reserved no longer covers a food item, the system automatically marks it as "temporarily unavailable" in the guests' Delicious App.

For now, ordering supplies is done outside the restaurant management system. The manager estimates the required amount of ingredients per day and ensures the storage is adequately stocked. In the medium term, however, we want to automate this: each restaurant works with a list of trusted suppliers, and the system should order ingredients that run low from these suppliers by itself, in reasonable quantities. The delivery itself and putting the supplies away remain outside the system.

The manager can correct the stock by hand in the Admin Client, for instance when a delivery has been put away, or when an ingredient has gone bad and has to be thrown away.

Ganz draußen, und was dafür im Seed steht

- ~~How Cooks Check In and Out at a Cooking Station~~: eine Station ist bemannt
- ~~How Food Items and Recipes are Managed~~: ein Gericht mit Rezept und genug Lager, kein Editor
- ~~Admin Client~~: Alarme, Eingriffe und Korrekturen entfallen
- ~~Drinks und Personnel~~: schon im Original außerhalb des Systems

Der Happy Path

- 1 Gast scannt den QR-Code am Sitzplatz, sieht die Speisekarte, bestellt und zahlt (Zahlung simuliert)
- 2 Die Zutaten werden im Lager reserviert
- 3 Die Bestellung geht an die bemannte Station, die Rezeptschritte erscheinen
- 4 Die Schritte werden nacheinander bestätigt, nach dem letzten wird ein Fach zugewiesen
- 5 Das Essen liegt im Fach, die Bestellung ist fertig
- 6 Gast bekommt einen QR-Code, scannt ihn an der Abholstation, das Fach öffnet sich
- 7 Das Fach ist wieder frei, die Zutaten sind verbraucht

Event-Katalog: Grenzen und Happy Path

1. **Paare** Je ein Team-Paar pro Pfeil der Context Map: Name und Payload der Events an dieser Grenze abstimmen, Änderungen gleich im Contract
2. **Pfad** Alle gemeinsam: den Happy Path Schritt für Schritt durchgehen
3. **Zeigen** Das Team, das ein Event erzeugt, zeigt seinen Contract
4. **Prüfen** Die Teams, die es nutzen: Kann ich damit arbeiten? Fehlen Daten, die ich brauche?
5. **Liste** Was offen bleibt, kommt auf eine gemeinsame Liste, mit Namen und Termin

Ein Event ist ein Fakt, der passiert ist, kein Aufruf in Verkleidung. Nur die Daten, die ein Consumer wirklich braucht.

Clients: welche gibt es, und wer ist zuständig?

1. **Akteure** Wer löst im EventStorming-Board und auf der Context Map Events aus, oder will sie sehen?
2. **Teams** Jedes Team markiert: Welche Events muss mein Context zeigen oder auslösen?
3. **Bündeln** Akteure zu Client-Kandidaten zusammenfassen
4. **Zusagen** Zwei Listen: Client gesucht, Client-Owner. Ein Client geht an das Team, dessen Context die meisten Daten liefert

Clients, die niemand übernimmt, bleiben sichtbar auf der Liste stehen.

Die einfachste Client-Architektur

- eine Hülle, die Bereiche pro Client, getrennt (Tabs oder Routen)
- keine Integration zwischen den Bereichen
- jeder Bereich spricht nur mit seinem Backend
- Backend for Frontend: am 05.11.

Prinzipien für Microservices

- 1 Bounded Context = 1 Service = 1 Team
- lieber zu groß anfangen, später teilen
- Aggregate = Einheit der Konsistenz im Service
- Event = ein Aggregate hat sich geändert
- Kommunikation vor allem asynchron, über Events
- Transport: Kafka
- Konventionen + Spezifikation pro Event: Event-Contract

Ein Service pro Bounded Context als Startpunkt: Vaughn Vernon, berichtet in InfoQ, 31.07.2016 (infoq.com/news/2016/07/microservices-ddd-vernon); Microsoft, .NET Microservices Architecture, Kapitel "Identifying domain-model boundaries". Grenzen zwischen Services sind schwer zu verschieben: Martin Fowler, MonolithFirst, 03.06.2015 (martinfowler.com/bliki/MonolithFirst.html).

Entwickeln und testen ohne Infrastruktur

- der Service läuft allein, ohne Kafka und ohne die Nachbarn
- Eingänge: REST und Event-Consumer, Ausgang: ein Port zum Veröffentlichen von Events
- Tests gegen den Application-Layer, mit einem Fake für das Veröffentlichen
- Events der Nachbarn von Hand einspeisen

Womit Ihr Team anfangen kann

- Repository klonen, Projektgerüst anlegen
- den Happy-Path-Ausschnitt Ihres Bounded Context als Test beschreiben, dann umsetzen
- Events passend zum Contract in docs/asyncapi.yaml
- Events der Nachbarn als Stubs
- Konventionen für Claude Code im Repository ablegen

Hausaufgabe bis 05.11.

- Contract fertig: docs/asyncapi.yaml im Service-Repo, passend zum Event-Katalog
- Service läuft allein: Happy-Path-Ausschnitt, lokale Tests, Stubs für die Nachbarn
- Client, falls Ihr Team einen hat: Entwurf der Oberfläche, simuliertes Backend, Architekturentscheidung als ADR

