

M I N I P R O J E K T

# TicketHub

Online Eventbilletsystem

Microservices med C#, Dapr og Aspire

---

Varighed: 5–7 uger

Teknologier: .NET / C#, Dapr, .NET Aspire

Op til 5 microservices med messaging, REST og SAGA

# 1. Introduktion

I dette miniprojekt skal I designe og implementere et online eventbilletsystem kaldet **TicketHub**. Systemet gør det muligt for arrangører at oprette events (koncerter, konferencer, teater), og for kunder at søge, bestille og betale billetter.

Projektet strækker sig over 5–7 uger og fører jer gennem hele processen: fra domæneanalyse med Bounded Contexts, over design af kommunikationsmønstre, til implementering af op til 5 microservices med C#, Dapr og .NET Aspire.

## 1.1 Læringsmål

- Analyser et domæne med Bounded Context-metoden og identificere microservices
- Implementere microservices i C# med .NET Aspire som orkestreringsværktøj
- Anvende Dapr til pub/sub-messaging og service invocation
- Implementere en SAGA med kompenserende transaktioner via Dapr Workflow
- Træffe og begrunde valg mellem asynkron messaging og synkrone REST-kald
- Observere og fejlfinde distribuerede systemer via Aspire Dashboard
- Strukturere en microservice internt efter Clean Architecture med adskillelse af Domain, Use Case, Facade og Infrastructure

## 1.2 Teknologistak

**.NET / C#** – Alle services bygges som ASP.NET Core-projekter.

**Dapr** – Anvendes til pub/sub (messaging), service invocation (REST mellem services) og workflows (SAGA).

**.NET Aspire** – Orkestreringsværktøj der starter alle services, Dapr sidecars og infrastruktur (f.eks. Redis, RabbitMQ) samlet. Giver et dashboard til at observere logs, traces og metrics på tværs af services.

## 2. Casebeskrivelse

TicketHub er en platform med følgende overordnede funktionalitet:

- Arrangører kan oprette events med navn, dato, sted, beskrivelse og billetkategorier (f.eks. "Standard", "VIP") med hvert sit antal og pris.
- Kunder kan browse og søge i tilgængelige events.
- Kunder kan bestille billetter. En bestilling involverer reservation af billetter, betaling og bekræftelse.
- Hvis betaling fejler eller timer ud, skal reserverede billetter frigives automatisk (kompenseringe transaktion).
- Kunder modtager notifikationer ved bekræftelse og annullering.

### **Forretningsregel: Billetter er en begrænset ressource**

Hver billetkategori har et fast antal billetter. Når en kunde starter en bestilling, skal billetterne reserveres midlertidigt, så de ikke kan sælges til andre. Hvis bestillingen ikke gennemføres, skal reservationen ophæves. Dette er kernemotivationen for SAGA-mønstret.

## 3. Sprint plan

### 3.1 Sprint 1

Formål:

1. Identificer services
2. Udarbejd og dokumenter det eksterne design imellem de indgående services
3. Opret kode skelet med domain og facade delen realiseret for én service.

Acceptkriterier:

- 1) Identificer services:
  - a) Analysér domænet med Bounded Context-metoden.
  - b) Identificér contexts, definér ubiquitous language per context.
  - c) Tegn et Context Map der viser relationer mellem contexts.
- 2) Gennemfør det eksterne design:
  - a) Design kommunikationen imellem services
    - i) Design af kontrakter (REST og Messaging)
  - b) Design ordreoprettelses SAGA
  - c) Tegn et BPMN diagram for ordreoprettelsen.
- 3) Opret kode skelet:
  - a) Opsæt Aspire-løsning med tomme C#-projekter f.eks. en Event Catalog service
  - b) Kod Domain og Facade og lad resten være tomme projekter.

Nedenstående er medtaget lidt hjælp:

#### 3.1.1 Designniveauer

Arkitekturen i TicketHub kan beskrives på tre niveauer, som tilsammen dækker hele designet fra det store billede ned til den enkelte service:

**3.1.2 Overordnet design (afsnit 3.1)** handler om at opdele domænet i selvstændige services. Her bruger I Bounded Context-analysen til at identificere hvilke microservices systemet består af, og hvad hver service ejer og er ansvarlig for.

**3.1.3 Eksternt design (afsnit 4–5)** handler om kommunikationen mellem services. Her beslutter I om en given interaktion skal være synkron (REST via Dapr service invocation) eller asynkron (pub/sub via Dapr), og I designer SAGA-flowet der orkestrerer bookingprocessen på tværs af services.

**3.1.4 Internt design (afsnit 6)** handler om den interne opbygning af hver enkelt service. Her følger I Clean Architecture-principperne, så hver service er struktureret i lag (Domain, Use Case, Facade, Infrastructure) med tydelig adskillelse af forretningslogik og tekniske detaljer.

Disse tre niveauer hænger tæt sammen: Det overordnede design bestemmer hvilke services der findes. Det eksterne design bestemmer hvordan de taler sammen. Og det interne design bestemmer hvordan hver service er bygget op indeni.

#### 3.1.4.1 Overordnet design: Domæneanalyse med Bounded Contexts

Før I begynder at kode, skal I analysere domænet og identificere de Bounded Contexts der skal blive til microservices. Start med at genlæse casebeskrivelsen i afsnit 2 og læg mærke til de forskellige ansvarsområder: Hvem ejer stamdata om events? Hvem holder styr på hvor mange billetter der er tilbage? Hvem koordinerer en bestilling fra start til slut? Hvem håndterer betaling? Og hvem sender beskeder til kunden?

Et godt udgangspunkt er at kigge efter substantiver og handlinger i casebeskrivelsen. Når I finder et område der har sin egen data, sine egne regler og kunne fungere uafhængigt, er det sandsynligvis en selvstændig Bounded Context. Tænk også over: Hvis to begreber lyder ens (f.eks. "billetkategori"), men betyder noget forskelligt alt efter konteksten – så er det et stærkt hint om at de hører til i hver sin context.

Nedenfor er analysen beskrevet – I forventes at gennemarbejde den og dokumentere jeres forståelse.

### 3.1.5 Eksternt design: Kommunikationsmønstre

Systemet anvender både asynkron messaging (pub/sub) og synkrone REST-kald. Valget er bevidst og begrundet for hvert enkelt tilfælde.

Nøglespørgsmålet er: Skal afsenderen vente på et svar for at kunne gå videre? Hvis ja, er synkron REST det rigtige valg – f.eks. når Ordering har brug for en pris fra Event Catalog, før den kan oprette ordenen. Hvis afsenderen derimod bare skal meddele at "noget er sket" og ikke behøver et svar, er asynkron pub/sub bedre – f.eks. når en ordre er bekræftet, og Notification blot skal reagere på det.

Tænk også over kobling: Med pub/sub kender afsenderen ikke modtageren, hvilket gør det nemt at tilføje nye lyttere senere. Med REST er der en direkte afhængighed mellem to services. Gå jeres kommunikationer igennem én for én og overvej: Er der tale om en kommando ("gør dette og fortæl mig resultatet") eller en hændelse ("dette er sket")?

#### 3.1.5.1 Serviceoversigt

Lav et skema over de enkelte services – det hjælper meget i kode fasen.

Service	Port (forslag)	Database	Dapr-funktioner
A	5001	SQL/SQLite	Pub/sub (publish)
B	5002	SQL/SQLite	Pub/sub (subscribe), Service invocation
C	5003	SQL/SQLite	Workflow, Pub/sub (publish), Service invocation
D	5004	Ingen (stateless)	Service invocation
E	5005	Ingen (log-baseret)	Pub/sub (subscribe)

### 3.1.6 Eksternt design: SAGA – Place Order

Kernen i systemet er bookingflowet, som implementeres som en SAGA med Dapr Workflow i Ordering-servicen. SAGA'en orkestrerer reservation, betaling og bekræftelse – og kompenserer hvis noget fejler.

Hvorfor en SAGA og ikke bare en almindelig transaktion? I et monolitisk system kan man pakke reservation og betaling ind i én database-transaktion. Men i microservices ejer hver service sin egen database, så der findes ingen fælles transaktion. SAGA-mønstret løser dette ved at udføre hvert trin for sig og definere en kompenserende handling, hvis noget går galt undervejs.

Tænk over det konkret: En kunde bestiller 2 VIP-billetter. Først reserveres billetterne i Inventory, så ingen andre kan købe dem. Dernæst forsøges betaling. Men hvad hvis betalingen fejler? Så sidder Inventory med 2 låste billetter, som ingen kan købe. Løsningen er kompensation: Ordering beder Inventory om at frigive billetterne igen. Spørg jer selv ved hvert trin: "Hvad kan gå galt her, og hvordan ruller vi tilbage?"

### 3.1.7 Eksternt design: BPMN – Place Order

En god teknik til at synliggøre opdelingen er BPMN swimlane-diagrammer. I et swimlane-diagram får hver aktør (eller service) sin egen vandrette bane, og processens aktiviteter tegnes i den bane der ejer dem. Når en pil krydser fra én bane til en anden, repræsenterer det kommunikation mellem services.

### 3.1.8 Internt design: Clean Architecture

Hvor afsnit 3–5 handler om det overordnede og eksterne design (hvilke services findes, og hvordan taler de sammen), handler dette afsnit om det interne design: Hvordan er den enkelte service bygget op indeni?

Hver microservice i TicketHub skal internt følge Clean Architecture-principperne, som kombinerer idéer fra Onion Architecture, Domain-Driven Design (DDD) og Command Query Separation (CQS). Det centrale princip er Dependency Rule: kildekode-afhængigheder må kun pege indad – fra de ydre, tekniske lag mod de indre, forretningsnære lag. Det sikrer at forretningslogikken er uafhængig af frameworks, databaser og andre tekniske detaljer.

#### 3.1.8.1 Lagmodellen

Arkitekturen består af fire lag plus et Api-projekt. Hvert lag afspejles som et selvstændigt projekt i jeres .NET Solution:

**Domain-laget (inderst)** – Arkitekturens hjerte. Indeholder kerneforretningslogikken i sin reneste form: entity-klasser med DDD-principper (private setters, tilstandsændringer via metoder), value objects og domain services. Dette lag har ingen afhængigheder til andre lag eller frameworks overhovedet. I Inventory-servicen vil f.eks. entity-klassen TicketStock med logik for reservation og frigivelse høre til her.

**Use Case-laget** – Orkestrerer applikationslogikken. Hvert use case repræsenterer én konkret handling (f.eks. “Placér ordre” eller “Reserver billetter”). Use cases kalder Domain-laget for forretningsregler og definerer repository-interfaces (f.eks. IOrderRepository, ITicketStockRepository), som Infrastructure-laget implementerer. Med CQS-princippet er use cases udelukkende commands – de ændrer tilstand, men returnerer ikke data. Læsning af data sker via separate query-interfaces (se Facade-laget).

**Facade-laget** – Kontraktlaget der udgør den eneste indgang til kernen. Her findes udelukkende interfaces og DTO'er – ingen implementering. Interfaces opdeles efter CQS-princippet: ét interface per use case (command) og ét interface per query-gruppe. Alle parametre og returværdier er DTO'er – domæne-entities eksponeres aldrig. Facade-laget sikrer at API-controllere, test-klienter og andre konsumenter aldrig taler direkte med use cases eller domæne-entities.

**Infrastructure-laget (yderst)** – Indeholder al teknisk implementering: databaseadgang (Entity Framework, SQLite), Dapr-integrationer, eksterne API-kald og messaging. Infrastructure implementerer både Use Case-lagets repository-interfaces og Facade-lagets query-interfaces. Med CQS kan query-handlers gå direkte til databasen med optimerede forespørgsler uden at involvere domæne-entities.

**Api-projektet** – ASP.NET Core Web API-controllere der eksponerer servicens funktionalitet. Controllerne refererer kun til Facade-laget (use case-interfaces og query-interfaces) og bruger dependency injection til at resolve de konkrete implementeringer.

#### 3.1.8.2 Dependency Rule

Den vigtigste regel i hele arkitekturen er: kildekode-afhængigheder må kun pege indad. Domain-laget kender intet til de andre lag. Use Case-laget afhænger kun af Domain, men implementerer Facade-lagets interfaces. Facade-laget definerer kontrakten (interfaces og DTO'er) og kender ikke til andre lag. Infrastructure-laget implementerer interfaces fra både Facade og Use Case, og afhænger desuden af Domain (til materialisering og persistering). Api-projektet refererer kun til Facade.

Konkret: Når et Use Case har brug for at gemme data, definerer det et repository-interface (f.eks. IOrderRepository). Infrastructure-laget implementerer dette med Entity Framework og SQLite. Ved opstart registreres implementeringen via dependency injection, så Use Case-laget aldrig har en direkte reference til Infrastructure-projektet. På samme måde definerer Facade-laget query-interfaces (f.eks. IEventQueries), som Infrastructure implementerer med optimerede læseforespørgsler.

### *3.1.8.3 Referencemateriale*

Følg Clean Architecture-noten fra kursusmaterialet for den fulde beskrivelse af lagene, DDD-entity-mønstret, CQS-princippet og Facade-lagets rolle:

<https://github.com/kbr-ucl/DMU-2-Semester-Programmering/blob/main/Noter/Clean-Architecture.md>

## 3.2 Sprint 2

Formål:

1. Etablering af pub/sub imellem to services

## 3.3 Sprint 3

Formål:

1. Oprettelse af ordre uden SAGA

## 3.4 Sprint 4

Formål:

1. Oprettelse af ordre med SAGA

## 3.5 Sprint 5

Formål:

1. Implementering af notifikation

## 3.6 Sprint 6 (optional)

Formål:

1. Implementering af robusthed

Note:

- Tilføj retry-policies på Dapr-kald.
- Implementér idempotens på alle endpoints.
- Opsæt dead letter topics til fejlede beskeder.
- Tilføj InventoryLow-event

## 4. Krav til løsningen

### 4.1 Minimum (uge 1–5)

1. Dokumenteret domæneanalyse med Bounded Contexts, ubiquitous language, Context Map og BPMN diagram.
2. Minimum 3 kørende microservices.
3. Mindst ét pub/sub-flow.
4. Mindst ét REST-kald via Dapr service invocation (f.eks. prisopslag).
5. En fungerende SAGA med mindst én kompenserende transaktion implementeret som Dapr Workflow.
6. Aspire-løsning der starter alle services samlet.
7. Minimum én service struktureret efter Clean Architecture med separate projekter for Domain, UseCases, Facade, Infrastructure og Api. Dependency Rule skal overholdes (se afsnit 6).

### 4.2 Udvidet (uge 6–7)

- Payment- og Notification-service (op til 5 services i alt).
- Retry-policies og idempotens.
- Dead letter topics.
- Simpelt UI.
- Afsluttende dokumentation med arkitekturbeskrivelse og refleksion.

## 5.

# Tips og Gode Råd

## Start småt

Begynd med to services og ét kommunikationsmønster. Få det til at virke før I tilføjer mere. Aspire gør det nemt at tilføje services løbende.

## Brug Aspire Dashboard aktivt

Dashboard'et viser logs, traces og metrics på tværs af alle services. Det er jeres vigtigste værktøj til at forstå hvad der sker i systemet, især når noget fejler.

## Simuler fejl bevidst

Gør Payment-servicens fejlrate konfigurerbar (f.eks. via en miljøvariabel eller et konfigurationsendpoint). Så kan I nemt teste både happy path og fejlscenarier.

## Husk idempotens

Messaging-systemer kan levere en besked mere end én gang. Overvej fra starten hvad der sker hvis en handler modtager den samme event to gange. Brug f.eks. et unikt EventId eller OrderId til at sikre idempotens.

## Hver service ejer sin data

Services må aldrig dele database. Hvis Ordering har brug for prisdata, kalder den Event Catalog – den læser ikke direkte i Event Catalogs database. Dette princip er fundamentalt for microservices.

## Dokumenter jeres valg

Skriv ned hvorfor I valgte pub/sub vs. REST for hver kommunikation. Det er mindst lige så vigtigt at kunne begrunde designet som at implementere det.

— *God fornøjelse med projektet!* —

