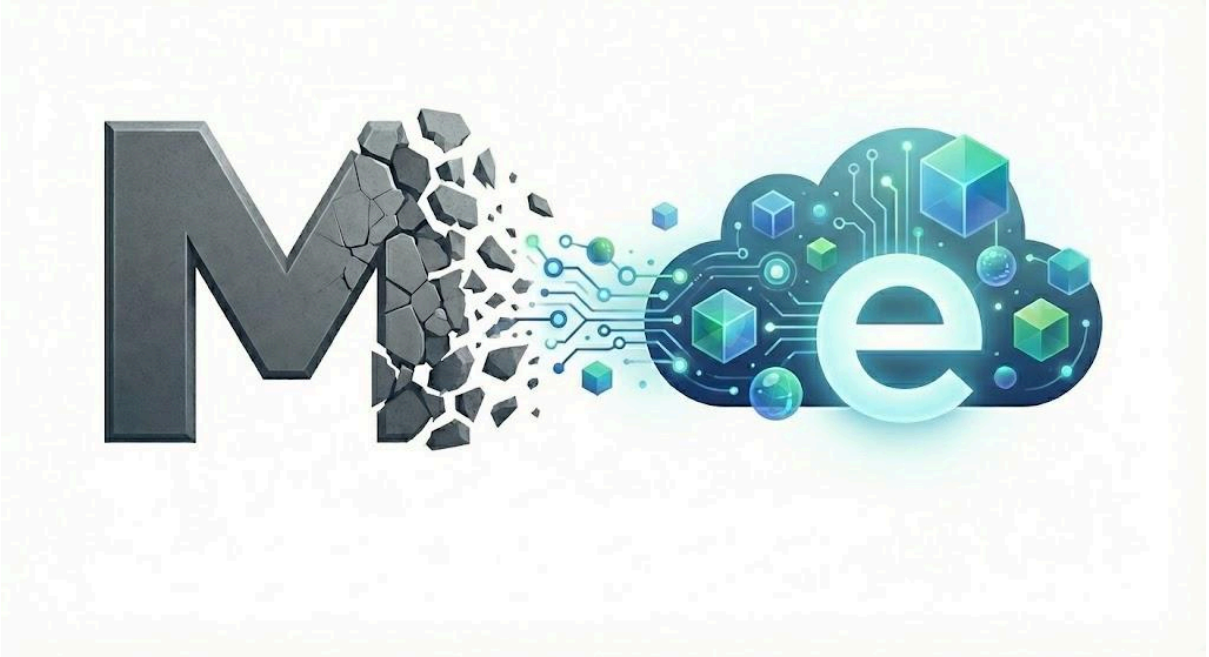
eShopOnWeb



Store systemer og Systemintegration

Underviser:

Morten Sabro Damgaard og Jakob Varring Jensen

Hold: SWOF25

Udarbejdet af

Ihab Dabaan, Seymen Kiran, Nouraldin Habbal og Adil Nazir

GIT repository: https://github.com/IHD85/eShopProject.git

Antal tegn: 53,509

1.Introduktion

Moderne softwaresystemer er i stigende grad distribuerede, skalerbare og løst koblede. Monolitiske arkitekturer er ofte vanskelige at udvide, teste og drifte, fordi hele systemet er pakket ind i én applikation, der deployes samlet. Dette projekt tager udgangspunkt i den eksisterende applikation **eShopOnWeb**, der er bygget som en monolit.

Formålet med projektet har været at konvertere centrale funktioner til en **microservice-arkitektur** ved anvendelse af **Strangler Pattern**, hvor dele af systemet gradvist flyttes fra monolitten til selvstændige services.

Der er udviklet fem microservices:

- **Catalog Service** – produktkatalog
- **Basket Service** – kurv, cache og checkout
- **Order Service** – ordreoprettelse og ordrebehandling
- **Identity Service** – autentifikation og autorisation (JWT)
- **Inventory Service** - Lagersystem til Catalog.

Desuden er der implementeret en API Gateway baseret på YARP, som fungerer som fælles *entry point* for al trafik til Microservices. Microservices kommunikerer via RabbitMQ events, og hver service har sin egen PostgreSQL-database, mens Basket anvender Redis som cache. Alt kører i Docker Compose.

Projektets mål er at demonstrere en komplet, skalerbar og moderne arkitektur, der kan erstatte dele af monolitten uden nedetid og samtidig sikre sikkerhed, integritet og god systemkvalitet.

1.1 Problemformulering

Hvordan kan den monolitiske eShopOnWeb-applikation migreres til en moderne microservice-arkitektur ved anvendelse af Strangler Pattern, således at systemet opnår løst koblede services med selvstændigt ansvar, samtidig med at drift, performance, skalerbarhed og sikkerhed opretholdes?

1.2 Funktionelle og ikke-Funktionelle Krav

1.2.1 Funktionelle Krav

Disse krav beskriver, **hvad** systemet konkret skal kunne gøre, og hvilke funktioner der skal implementeres.

- **Opdeling af Monolit:** Systemet skal opdeles i mindst 3 uafhængige microservices.
- **Lagerstyring:** Der skal implementeres et nyt lagerstyringssystem som en selvstændig microservice.
- **Strangler Pattern:** Funktionalitet skal gradvist flyttes fra monolitten til microservices. Trafik til de implementeret dele, skal rutes til de nye services, mens resten rutes til monolitten.
- **API Gateway Routing:** En API Gateway skal fungere som fælles indgangspunkt ("Single Point of Entry") og dirigere anmodninger til enten microservices eller monolitten.
- **Autentificering:** Identity service skal håndtere brugerautentificering.

1.2.2 Ikke-Funktionelle Krav

Disse krav beskriver, **hvordan** systemet skal opføre sig, herunder performance, arkitektur og drift.

- **Arkitektur:** Der skal anvendes en microservice-arkitektur med løs kobling.
- **Dataseparation:** Hver microservice skal have sin egen database (*Database-per-service pattern*).

- **Kommunikation:** Der skal anvendes asynkron, beskedbaseret kommunikation (RabbitMQ) mellem services.
- **Skalerbarhed:** Services skal kunne skaleres uafhængigt af hinanden. API Gateway skal understøtte load balancing.
- **Sikkerhed og Drift:** API Gateway skal håndtere rate-limiting.
- **Robusthed (Resilience):** Der skal implementeres fejlhåndtering, herunder *Circuit Breaker* mønsteret, for at håndtere nedbrud i services.
- **Overvågning (Observability):** Der skal opsættes centraliseret logging og overvågning (Prometheus/Grafana).
- **CI/CD:** Der skal etableres en pipeline til automatisk bygning, test og containerisering (Docker).

1.2 Monolit til Microservice Arkitektur

I eShopOnWeb ligger al forretningslogik samlet i én applikation. Denne arkitekturstil har længe været udbredt, fordi den er enkel at starte med, let at teste i små projekter og nem at deploye, når funktionaliteten er begrænset. Når et system vokser i omfang og kompleksitet, begynder denne struktur dog at skabe betydelige begrænsninger. Tæt kobling mellem domæner gør det vanskeligt at videreudvikle systemet uden at påvirke andre funktioner, og ændringer ét sted kan introducere fejl i andre dele af applikationen. Samtidig bliver skalerbarheden kun mulig ved at opskalere hele applikationen, også selvom kun ét område er belastet. Det reducerer fleksibilitet, øger driftsrisici og gør systemet mindre robust over tid.

En microservice-arkitektur løser disse udfordringer ved at nedbryde systemet i selvstændige, domæne orienterede services, som hver kan udvikles, deployes og skaleres uafhængigt. Denne tilgang muliggør klar ansvarsfordeling, dedikeret datalagring per service og langt bedre isolering af fejl. Kommunikationen mellem services håndteres gennem veldefinerede API'er og asynkrone events, hvilket sikrer løs kobling og reducerer afhængigheder. Samlet giver dette en mere fleksibel og skalerbar arkitektur, som bedre understøtter moderne krav til performance, drift, sikkerhed og kontinuerlig udvikling.

1.2.1 Monolitten

eShopOnWeb er opbygget som en klassisk monolit, hvor alle domæner ligger samlet i én kodebase og deler den samme database. Katalog, kurv, ordrelogik og brugerhåndtering er implementeret i ét projekt, og hele applikationen deployes som en samlet enhed. Denne struktur betyder, at en ændring i ét domæne kan påvirke andre områder, og fejl kan forplante sig på tværs af hele systemet.

I eShopOnWeb viser det sig konkret ved, at:

- alle domæner deler de samme database-tabeller
- ændring i en model (fx CatalogItem) påvirker Basket, Order og UI samtidig
- nye features kræver ændringer flere steder i monolitten
- opskalering kun kan ske ved at skalere hele applikationen, selv hvis kun fx kataloget er belastet
- deployment er risikofyldt, fordi hele systemet skal genstartes ved hver ændring

Denne struktur har fungeret godt i små projekter, men bliver en begrænsning, når domænerne vokser, og når systemet skal skaleres, udvikles og driftes mere modent.

1.2.2 Microservices – begrundelse

Microservices introducerer en arkitektur, hvor systemet opdeles i uafhængige services med hvert deres ansvar og hver deres database. Denne tilgang løser flere af de udfordringer, som monolitten skaber.

Microservices giver følgende fordele:

- **Uafhængig deployment**
  Hver service kan opdateres uden at påvirke de andre.
- **Uafhængig skalering**
  Services som Catalog eller Basket kan skaleres individuelt baseret på belastning.
- **Løst koblede funktioner**
  Hver service er isoleret i sin egen kodebase og har sit eget livscyklusforløb.
- **Dedikeret datalagring**
  Hver service har sin egen PostgreSQL-database, hvilket reducerer risikoen for data konflikter og gør domænerne tydeligt afgrænset.
- **Asynkron kommunikation via RabbitMQ**
  Events sikrer, at microservices kan reagere på hændelser uden direkte afhængighed.

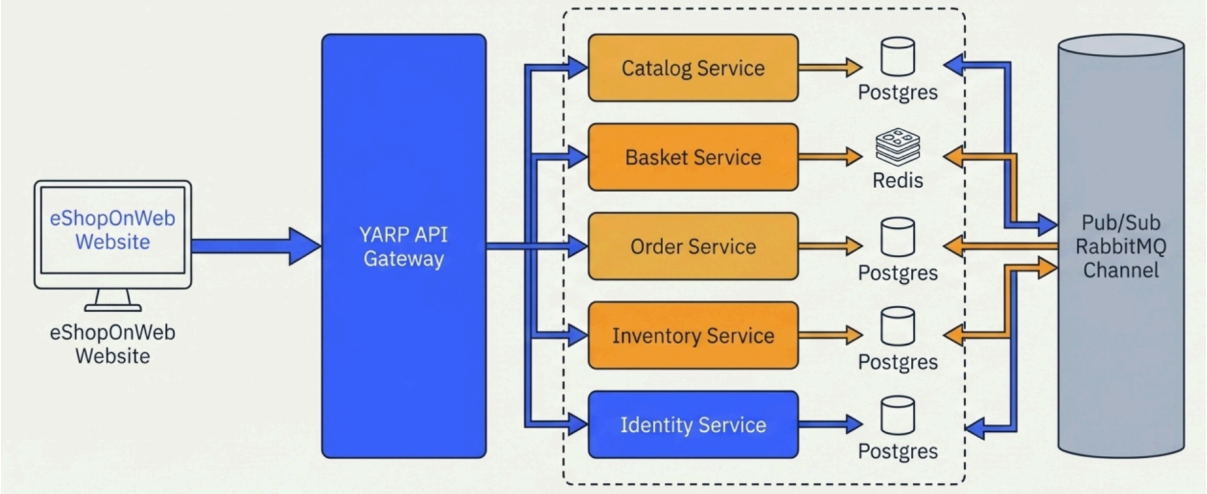
- **Bedre fejlisolation**

  Nedbrud i én service påvirker ikke de andre, hvilket øger systemets robusthed.

Denne arkitektur skaber et mere fleksibelt, skalerbart og driftssikkert system, hvor udvikling kan foretages modulært og distribueret. Det danner samtidig det tekniske fundament for at anvende Strangler Pattern til gradvis migration.

1.2.3 Strangler Pattern – valg af metode

Strangler Pattern bruges til gradvis udskiftning af et system uden at stoppe driften. Frontend (monolitten) peger først på monolittens endpoints. Efterhånden som nye microservices erstatter funktioner, rutes trafikken via Gateway til microservices.



Figur 1 - Systemet med Microservices (Med Gateway, Services og Message Broker Pub/Sub)

1.3.1 Arkitektur og afhængigheder i monolitten

Monolitten indeholder flere domæner (Catalog, Basket, Order, Identity), men disse er ikke separeret i selvstændige moduler. De er lagt i samme kodebase og anvender de samme datamodeller og samme database. Afhængighederne er stærkt koblede:

- Controllers kalder services direkte
- Services kalder EF Core repositories direkte
- Repositories arbejder på en fælles database

Dette gør systemet komplekst og svært at udvide.

1.3.2 Begrænsninger i monolitten

Analysen af monolitten viser følgende problemer:

- Manglende skalerbarhed: hele applikationen skal skaleres, selvom kun én funktion (f.eks. kurv) har høj belastning.
- Dårlig deploybarhed: en lille ændring kræver redeployment af hele systemet.
- Single point of failure: en fejl i et modul kan bringe hele systemet ned.
- Kompleksitet i vedligehold: alle udviklere arbejder i samme kodebase som kan føre til merge conflicts, regressioner.
- Delt database: risiko for locking, race conditions og brud på dataintegritet.

1.3.3 Identifikation af microservices (bounded contexts)

Ved analyse af monolitten kunne vi identificere følgende domæner, som naturligt kunne udskilles:

- Catalog – produktinformation, brands, kategorier
- Basket/Payment – indkøbskurv, checkout, cache
- Order – ordrer, ordrestatus, historik
- Inventory - lagerstatus
- Identity – brugere, login, roller, JWT

Hvert domæne har eget ansvar, egne data og egen forretningslogik, derfor er de ideelle kandidater til microservices.

1.3.4 Konklusion på analysen

Monolitten fungerer stabilt i sin nuværende form, men arkitekturens begrænsninger gør den uegnet til videre skalering og moderne drift. Analysen viser, at tæt kobling mellem domæner, fælles database og

samlet deployment kræver store indgreb ved selv små ændringer. Dette øger risikoen for fejl og gør udvikling, test og drift unødigt kompleks.

En microservice-arkitektur adresserer disse problemer ved at adskille domæner, isolere data og tillade uafhængig skalering og deployment. Det giver mulighed for hurtigere iterationer, bedre fejl isolering, stærkere sikkerhedslag og en arkitektur, der kan udnytte cloud-infrastruktur effektivt. Migreringen er derfor ikke kun en teknisk forbedring, men en nødvendighed for at systemet kan udvikles og driftes på et professionelt niveau.

1.4 Strangler Pattern – praktisk implementering

Strangler Pattern blev brugt til gradvist at migrere logikken ud af monolitten, uden at afbryde driften og uden at ændre frontend (eShopOnWeb). I praksis betyder det, at nye microservices overtager funktionalitet område for område, mens monolitten gradvist "kvæles".

1.4.1 Gateway som indgang og kontrolpunkt

Første skridt var at placere en API Gateway (YARP) foran monolitten.
Al trafik går nu gennem Gatewayen:

Frontend → Gateway → Microservice

På den måde kan vi kontrollere hvilke funktioner der håndteres hvor.

1.4.2 Gradvis ruteomlægning

Når en funktion migreres til en microservice, ændres kun Gatewayens ruter.
Frontend er uændret.

Eksempel:
Før migration:

- /Basket/AddToBasket → Monolit MVC-controller

Efter migration:

- /basket/Basket/add → Basket.API (Docker container)

1.4.3 Event-drevet afkobling

Efter checkout i Basket-service publiceres der events i RabbitMQ:

- BasketCheckedOutIntegrationEvent
- OrderCreatedIntegrationEvent
- ProductPriceChangedIntegrationEvent

Disse events flytter forretningslogikken ud i microservices.

1.4.4 Kontrolleret udskiftning af funktioner

Efter hvert domæne blev migreret, blev den tilhørende monolit-kode deaktiveret eller fjernet

Migreringen fra monolit til microservices blev gennemført gradvist og kontrolleret for at sikre, at systemet fungerede under hele processen. For hvert domæne fulgte vi samme fremgangsmåde:

1. **Identificer funktionalitet i monolitten**
   F.eks. katalog, kurv, ordre eller brugerhåndtering.

2. **Løft funktionaliteten ud i en ny microservice**
   Domænets logik blev implementeret som en selvstændig service med sin egen database, event-integration og API.

3. **Udfasning af logik i monolitten**
   Når en microservice overtog et område, blev den tilsvarende kode i monolitten deaktiveret eller fjernet for at undgå duplikeret forretningslogik og data-infrastruktur.

4. **Omlægning af trafik via Gateway**
   API Gateway (YARP) blev opdateret, så ruten ikke længere pegede på monolitten, men på den nye microservice.
   Dette gjorde migreringen usynlig for frontend, som fortsat kaldte de samme endpoints.

5. **Event-baseret afkobling mellem domæner**
   Funktioner, der tidligere var afhængige af hinanden i monolitten, blev nu koblet sammen gennem RabbitMQ-events.
   F.eks. sender Basket BasketCheckedOutIntegrationEvent, Order opretter ordren og sender

OrderCreatedIntegrationEvent, som Inventory bruger til at opdatere lageret.

Denne proces blev gentaget for hvert domæne, indtil hele forretningslogikken var løftet ud af monolitten og distribueret i selvstændige microservices.

Det centrale i denne metode er, at **alt skete trinvis**, uden at brugeren på noget tidspunkt oplevede nedetid eller ændringer i frontend.

Gateway ruten pegede nu permanent på microservice

1.4.5 Resultat

Frontend bruger stadig monolitten som UI, men al backend-logik ligger nu 100% i microservices.
Efter fuld gennemførsel af Strangler Pattern står systemet i en ny arkitektur:

- **Frontend (eShopOnWeb) fungerer stadig uændret**, præcis som i monolitten.
- **Al backend-forretningslogik er flyttet ud** i microservices:
    - Basket
    - Order
    - Catalog
    - Identity
    - Inventory (som ekstra krav - lagerstyring)
- **Kommunikationen mellem services er 100% asynkron** via RabbitMQ.
- **Hver microservice har sin egen PostgreSQL-database**, og Basket anvender Redis til kurvdata.
- **Monolitten bruges kun som UI**, og alle relevante endpoints rutes via API Gateway til microservices.
- Monolitten indeholder **ingen kritisk backend-logik længere**, og kan nu udfases helt, uden risiko for systemnedbrud.

Det betyder, at vi i praksis har opnået det præcise formål med Strangler Pattern:

> At erstatte en monolitisk applikation med en moderne microservice-arkitektur uden at lukke systemet ned eller påvirke brugere undervejs.

2. Build/Deployment

2.1 Docker

Til at deploye projektet anvendes Docker og Docker Compose. Hele systemet består af flere selvstændige containere, som kører parallelt i et fælles Docker-netværk. Hver microservice (Basket, Order, Catalog, Identity, Gateway) har sit eget image, egen konfiguration og egen databasecontainer (PostgreSQL), hvilket sikrer fuld isolation af ansvar og data.

Compose-filen definerer alle services samlet, inkl. RabbitMQ, Redis, Gatus og Grafana. Når projektet startes med:

```
docker compose up -d
```

oprettes alle containere automatisk, bindes til interne netværk og får tildelt faste porte.

Billedet nedenfor viser det endelige deploymentsmiljø:

- **gateway** fungerer som eneste offentlige indgangspunkt
- **basket-api**, **order-api**, **catalog-api**, **identity-api** kører som separate microservices
- **postgres-containere** lagrer data for hver service via persistente Docker-volumes
- **redis** bruges af Basket-servicen til caching
- **rabbitmq** bruges til event-kommunikation mellem services
- **grafana** og **gatus** leverer observability (logs, health checks)

Hver container kan stoppes, startes og genstartes individuelt uden at påvirke resten af systemet. HealthChecks sikrer, at services først markeres som "ready", når deres afhængigheder (fx database og message-broker) svarer korrekt. Det gør hele miljøet mere stabilt og nemt at drifte.

Denne opbygning gør systemet reproducerbart: det kan startes på enhver maskine uden manuelle installationer, og alle services kører i præcis samme miljø hver gang.

Figur 2 - Services der kører i Docker Desktop

2.1.1 Driftmiljø og container-arkitektur

Projektets microservice-arkitektur er fuldt containeriseret. Hver service kører i sin egen Docker-container for at sikre isolation, skalerbarhed og uafhængig deployment. Dette understøtter Strangler Pattern-tilgangen, fordi funktioner kan flyttes væk fra monolitten uden at påvirke resten af systemet.

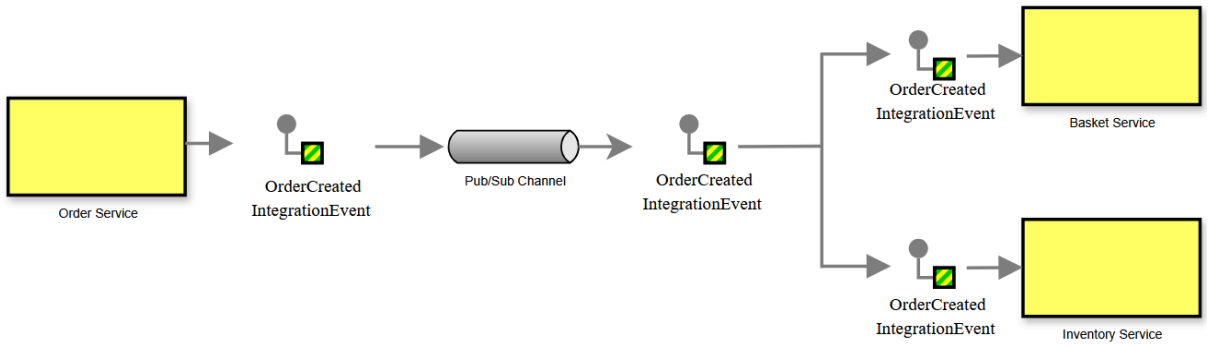
- eShop.Basket.API

  Basket-servicen håndterer kundens indkøbskurv, cache og checkout-flow. Den kører i en separat container for at kunne skaleres uafhængigt af de andre services, f.eks. ved høj trafik belastning under checkout. Basket anvender Redis til hurtig lagring af kurv produkter.

- eShop.Order.API

  Order-servicen modtager checkout-events fra Basket og opretter ordrer. Den kører isoleret i sin egen container, som gør det muligt at opdatere ordrer eller database uden nedetid, for de øvrige services. Den publicerer desuden OrderCreatedIntegrationEvent til RabbitMQ, som

Inventory-servicen bruger til lageropdatering og Basket bruger den til at slette den pågældende Basket.



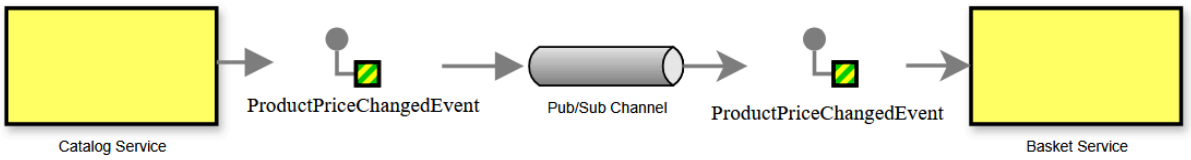
Figur 3 - EIP Diagram for OrderCreatedIntegration Event Message

- Identity.API

Identity-servicen håndterer autentifikation, brugere og JWT-tokens. Ved at køre i egen container holdes sikkerhedslag og brugerhåndtering adskilt fra forretningslogik. Det gør også sikkerhedsopdateringer mere kontrollerede.

- Catalog.API

Catalog-servicen styrer produktkatalog og lagerantal. Den modtager OrderCreated-events fra Order-servicen og opdaterer lageret asynkront. Den kører i en separat container, så produktkatalog og lagerstyring kan skaleres eller ændres uden at påvirke checkout eller login.



Figur 4 - EIP Diagram for ProductPriceChangedEvent Event Message

- Gateway.Api

API Gateway fungerer som eneste indgangspunkt for klienter og frontend. Den ligger i egen container og håndterer routing, rate-limiting og trafikstyring. Gateway'en muliggør Strangler Pattern, fordi den gradvist kan omdirigere endpoints fra monolitten til microservices uden at ændre frontend.

2.1.2 Databaser i separate containere

Hver microservice har sin egen PostgreSQL-database, der kører i sin egen container. Det betyder, at Basket, Order, Catalog og Identity ikke deler tabeller, skema eller *ConnectionStrings*, men kun taler sammen via deres API'er og RabbitMQ-events. Dette følger princippet "database per service" og sikrer, at hver service har fuldt ejerskab over sine data og har sit eget datamodel-design.

Data adskillelsen har flere konkrete fordele i projektet:

- **Ingen kryds-skema afhængigheder:**
  Der findes ingen direkte foreign keys på tværs af databaser. Order-databasen refererer fx til **CustomerId** og **ProductId** som rene værdier, uden at være bundet til Identity- eller Catalog-databasen. Hvis vi ændrer Identity-skemaet (fx tilføjer felter til bruger tabellen), påvirker det ikke Order-databasen eller dens migrationer.

- **Uafhængige migrationer og releases:**
  Hver service har sine egne EF Core migrationer. Vi kan fx ændre **CatalogItems**-tabellen (fx tilføje **AvailableStock**, nye felter til pris eller kategori) og køre migrationer kun på Catalog-databasen uden at skulle lave en stor "big bang"-ændring på tværs af hele systemet. Tilsvarende kan Order-servicen få nye felter på **Orders** og **OrderItems** uden at risikere at bryde med **Basket** eller **Identity**.

- **Mindre risiko for låsninger og skema konflikter:**
  I en monolitisk database ser man ofte problemer med låsninger på tabeller (fx **Orders** eller Basket), når flere funktioner konkurrerer om de samme rækker. I vores løsning skriver Basket kun til sin egen database/Redis, Order kun til sin egen Postgres, osv. Det reducerer risikoen for, at én tung operation (fx rapportering på ordrer) blokerer kurv-opdateringer eller login.

- **Klar ansvarsfordeling og bounded contexts:**
  Hver database afspejler et bounded context:

  - **Catalog-databasen** indeholder produktkatalog og lagerfelter (**CatalogItems** med bl.a. **AvailableStock**).

  - **Order-databasen** indeholder ordrer og ordrelinjer (fx Orders og OrderItems).

- **Identity-databasen** indeholder brugere, roller og relationer mellem dem (fx Users, Roles, UserRoles).

- **Basket-databasen** bruges sammen med Redis til kurv/checkout-relaterede data og evt. outbox/integrationsevents.

Konkrete eksempler fra projektet

- Ved checkout læser Basket-servicen kurven fra Redis og publicerer et **BasketCheckedOutIntegrationEvent**.

  - Eventet indeholder kun de data, Ordre har brug for (kunde-id, totalpris).

  - Order-servicen modtager eventet via RabbitMQ og skriver en ny ordre til **Order-databasen** (fx Orders + OrderItems). Den skal ikke lave JOIN mod Catalog- eller Identity-tabeller; alt foregår inden for dens egen database.

- Når en ordre er oprettet, publicerer Order-servicen et **OrderCreatedIntegrationEvent**.

  - Catalog-servicen modtager eventet og opdaterer lageret (**AvailableStock**) i **Catalog-databasen**.

  - Igen sker der ingen direkte databasekald til Order-databasen; Catalog reagerer kun på eventet og ændrer sine egne tabeller.

Det er netop disse flows (Basket → Order → Inventory), hvor vi har et meget konkret bevis på "database per service" i praksis: data bevæger sig som events mellem services, mens hver service gemmer sin egen sandhed i sin egen PostgreSQL-database, uden delte skemaer eller direkte database afhængigheder.

- **Redis (Basket)**

  Redis (*Basket*) er implementeret som et in-memory cache-lag, der ligger foran Basket-servicens PostgreSQL-database. I stedet for at gemme Basket data direkte i en persistent PostgreSQL database, lagres den midlertidigt i Redis, hvor alle data ligger i RAM. Det betyder, at læse- og skriveoperationer på Basket, typisk sker på få millisekunder, hvilket

er vigtigt, fordi brugeren ofte opdaterer *Basket* (lægger varer i, fjerner, ændrer antal) mange gange, før der gennemføres et checkout. Samtidig undgår vi unødvendig belastning på databasen (fx postgres), som kun skal opbevares midlertidigt i kurven.

I Basket-servicen lagres kurver typisk med en nøglestruktur som fx basket:{customerId}, hvor hele kurven serialiseres som JSON i Redis. Da kurvdata ikke skal persistere data i lang tid, og derudover kan de også konfigureres med TTL (time-to-live), så ubrugte kurve automatisk udløber og bliver ryddet op. Ved checkout læser Basket kurven fra Redis, publicerer et integrationsevent til Order-servicen og kan derefter slette kurven fra cachen. På den måde bruges Redis kun til hurtig, midlertidig tilstand, mens de endelige ordrer og betalingsoplysninger gemmes sikkert og permanent i PostgreSQL.

En vigtig årsag til at vælge Redis frem for fx ASP.NETs lokale **IMemoryCache** er, at Redis kører som en selvstændig container og dermed kan deles på tværs af flere instanser af Basket-servicen. Hvis Redis-containeren skulle genstartes, mistes kun de midlertidige kurvdata, mens de "rigtige" domænedata (ordrer, produkter, brugere) fortsat ligger i deres egne PostgreSQL-databaser. Dermed fungerer Redis som et hurtigt, men ikke forretningskritisk, in-memory cache-lag, der både forbedrer brugeroplevelsen og aflaster resten af systemet.

- **RabbitMQ**

  RabbitMQ containeren fungerer som central message-broker. Den muliggør asynkron kommunikation mellem services, så Basket, Order og Catalog kan arbejde uafhængigt, og fejl i én service ikke stopper resten af flowet. Det reducerer kobling og forbedrer systemets robusthed.

2.2 Overvågning og drift

- **Gatus**

  Gatus kører i egen container og overvåger *health checks* for Gateway og alle microservices. Den sikrer, at systemets status kan monitoreres løbende, og at fejl opdages tidligt. Den bruges primært til *Black Box* overvågning.

- **Grafana**

  Grafana bruges til visualisering af logs, metrics og performance-data (via Serilog → OTLP). Ved at køre i separat container kan overvågningen skaleres eller udskiftes uden at påvirke applikationerne.

2.2 Versionsstyring - GitHub

Til versionsstyring anvendes GitHub som fælles repository for hele projektet. GitHub gør det muligt at håndtere ændringer i kildekoden struktureret gennem branches, commits og pull requests. Det sikrer, at flere udviklere kan arbejde parallelt uden konflikter, og at alle ændringer er dokumenteret og kan spores i projektets historik. GitHub fungerer samtidig som en sikker ekstern backup, og understøtter god udviklingsdisciplin i projektgruppen.

2.3 CI/CD - GitHub Actions

Til CI/CD har vi gjort brug af GitHub Actions.[1]

Projektet benytter GitHub Actions til at automatisere build, test og containerisering af microservices. Workflowet er konfigureret til automatisk at køre, når der sker ændringer i master-branch eller når der åbnes en pull request.

Det første job bygger hele .NET-løsningen, gendanner NuGet-pakker og kører alle tests for at sikre, at koden er stabil, før den må gå videre i pipeline. Kun hvis alle tests består, fortsætter workflowet.

Det andet job bygger Docker-images for hver microservice (Gateway, Basket, Catalog, Identity, Inventory og Order) via en matrix-strategi. Dette betyder, at hver service containeriseres separat, og at ændringer kun påvirker de nødvendige services. Derefter pushes images automatisk til Docker Hub ved hjælp af projektets GitHub-secrets.

Workflowet sikrer dermed konsistent kvalitet og gør projektet klar til senere deployment, selv om fuld deployment ikke er implementeret endnu.

[1] Findes i Bilag 1

2.4 Deployment

Selvom fuld deployment ikke er implementeret, er arkitekturen forberedt til at kunne automatisere udrulning af microservices ved hjælp af GitHub Actions og Docker-baserede miljøer.

Strukturen gør det muligt at bygge Docker-images for hver service, hvorefter de kan publiceres til et container registry (Docker Hub). Disse images kan senere anvendes til deployment på en server, i en container-orchestrator eller i en cloud-platform. Deployment-sektionen fungerer dermed som en teknisk ramme for, hvordan systemet kan idriftsættes på en kontrolleret og skalerbar måde, når det ønskes.

3. Lagerstyringssystem

3.1 Implementering

Lagerstyringssystemet er implementeret som en selvstændig service med ansvar for antallet af produkter og lager opdateringer. Servicen eksponerer et API til læsning af produkter og deres aktuelle lagerantal, og den har sin egen PostgreSQL-database, hvor hvert produkt er knyttet til et felt for tilgængelig beholdning (AvailableStock).

Lageret opdateres ikke direkte fra frontend, men via det event-drevne ordreflow. Når en kunde gennemfører checkout i Basket-servicen, publiceres et BasketCheckedOut-event til RabbitMQ. Order-servicen konsumerer dette event, opretter en ordre i sin egen database og publicerer derefter et OrderCreated-event. Lagerstyringsservicen abonnerer på OrderCreated-eventet, opslår de berørte produkter i sin database og reducerer lagerantallet svarende til de bestilte mængder.

Denne implementering betyder, at lageret altid afspejler de faktisk oprettede ordrer, og at opdateringen sker asynkront uden direkte kobling mellem ordrelogik og lagerlogik. Systemet er samtidig forberedt til at kunne udvides med egentlig reservation ved checkout (f.eks. midlertidig reservering af beholdning, før en ordre endeligt bekræftes), uden at det kræver ændringer i de øvrige microservices.

4. YARP - API Gateway

I eShop-projektet varetages ansvaret som API Gateway af **YARP (Yet Another Reverse Proxy)**. Gatewayens formål er at agere som en samlet indgang (et *single point of entry*) for klienten, der skal

kommunikere med de bagvedliggende microservices. Denne implementering er konfigureret i projektet Gateway/API.Gateway.

4.1 Implementering

4.1.2 Konfiguration af Services

Opsætningen af YARP i projektets Program.cs

```
8           builder.Services.AddReverseProxy()
9               .LoadFromConfig(builder.Configuration.GetSection("ReverseProxy"));
10
```

Figur 5 - Kodeeksempel for Konfigurering af Gateway

Aktivering af reverse proxy i Program.cs:

```
29          app.MapReverseProxy();
```

Figur 6 - Kode eksempel for MapReverseProxy metoden.

Analyse af Program.cs:

1. **AddReverseProxy()**: Denne metode registrerer YARP's nødvendige services i applikationens DI-container.
2. **LoadFromConfig(...)**: Denne metode henter konfigurationen – dvs. definitioner af ruter (rutes) og klynger (clusters) – direkte fra applikationens konfigurationsfiler (appsettings.json) under JSON-nøglen "ReverseProxy".
3. **MapReverseProxy()**: Denne metode tilføjer YARP-middlewaren til applikationens request pipeline, hvilket aktiverer selve proxy-funktionaliteten.

4.2 Rute- og Clusters

Den specifikke routing-logik er defineret i **appsettings.Docker.json**. Denne fil indeholder den centrale konfiguration for, hvordan YARP skal håndtere trafik i et Docker-miljø.

Konfigurationen er opdelt i Routes (ruter) og Clusters (klynger).

4.2.1 Routes

Ruterne definerer, hvordan indgående anmodninger matches, og hvad der skal ske med dem. Projektet anvender en unik sti, der er mappet til hver microservice.

Der er defineret fem primære ruter:

- **catalog-route**: Matcher alle anmodninger til stien /catalog/{**catch-all}.
- **basket-route**: Matcher alle anmodninger til stien /basket/{**catch-all}.
- **order-route**: Matcher alle anmodninger til stien /order/{**catch-all}.
- **identity-route**: Matcher alle anmodninger til stien /identity/{**catch-all}.
- **inventory-route**: Matcher alle anmodninger til stien /inventory/{**catch-all}.

Hver rute anvender **Transforms** til at omskrive den offentlige sti til den interne sti, som de bagvedliggende services forventer. For eksempel omskrives identity-route's offentlige sti /identity/{...} til den interne sti /api/{...} via denne transformation:

```
26              "Transforms": [
27                { "PathRemovePrefix": "/identity" },
28                { "PathPrefix": "/api" }
```

Figur 7 - Transformering af rute

4.2.2 Clusters

Hver rute videresender trafik til en "Cluster". En cluster er en logisk gruppe af en eller flere destinationer (endpoints). I appsettings.Docker.json er der defineret fem klynger, som korresponderer med de fem ruter:

- **catalog-cluster**: http://catalog-api:8080
- **basket-cluster**: http://basket-api:8080
- **order-cluster**: http://order-.api:8080
- **identity-cluster**: http://identity-api:8080
- **inventory-cluster**: http://inventory-api:8080

Disse adresser (f.eks. http://catalog-api) er ikke offentlige DNS-navne, men derimod de interne service navne, som anvendes af Docker Compose til service discovery.

4.3 Rate Limiting

Rate limiting bruges i API Gateway'en til at beskytte microservices mod misbrug og overbelastning. Ideen er, at en klient kun må sende et vist antal requests i et givent tidsinterval (fx 20 requests pr. 10 sekunder). Hvis grænsen overskrides, returnerer Gateway'en en **429 (Too Many Requests)**, og kaldet bliver aldrig sendt videre til den bagvedliggende service.

I vores løsning er rate limiting implementeret direkte i Gateway-projektet med ASP.NET Core Rate Limiting middleware. Vi konfigurerer en fast vindues-politik med navnet "fixed":

```
builder.Services.AddRateLimiter(options =>
{
    options.AddFixedWindowLimiter("fixed", limiterOptions =>
    {
        limiterOptions.Window = TimeSpan.FromSeconds(10); // hver 10 sekunder, et nyt vindue
        limiterOptions.PermitLimit = 20; // maks 20 anmodninger per vindue (per klient)
        limiterOptions.QueueLimit = 0; // ingen kø (429, hvis for mange request)
    });
});
```

Figur 8 - Kode eksempel for rate limiting.

Derefter aktiveres middleware med:

```
app.UseRateLimiter();
```

Figure 9 - Rate limiting metode eksempel.

I appsettings.json knytter vi denne *policy* til hver route via RateLimiterPolicy:

```
"ReverseProxy": {
  "Routes": {
    "basket-route": {
      "ClusterId": "basket-cluster",
      "Match": { "Path": "/basket/{**catch-all}" },
      "RateLimiterPolicy": "fixed",
      "Transforms": [
        { "PathRemovePrefix": "/basket" },
        { "PathPrefix": "/api" }
      ]
    },
```

Figur 10 - Rate Limiter Policy.

Det betyder, at alle kald til fx /basket/, /catalog/, /identity/, /inventory/ og /order/ går gennem den samme rate limiter-politik. I praksis resulterer det i, at en enkelt klient ikke kan oversvømme Gateway'en og lægge Basket, Order, Identity eller Catalog ned ved at sende tusindvis af requests på kort tid. Rate limiting indgår dermed som en del af vores "beskyttelseslag" sammen med authentication, authorization, logging og overvågning.

4.4 Load Balancing

Load balancing er konfigureret i YARP-Gateway'en på cluster-niveau. For hver microservice (basket, catalog, identity, inventory, order) er der oprettet et cluster med LoadBalancingPolicy sat til "RoundRobin":

```
"Clusters": {
  "basket-cluster": {
    "LoadBalancingPolicy": "RoundRobin",
    "Destinations": {
      "primary": { "Address": "https://basket-api:5001" }
    }
  },
  "catalog-cluster": {
    "LoadBalancingPolicy": "RoundRobin",
    "Destinations": {
      "primary": { "Address": "https://catalog-api:5002" }
    }
  },
```

Figur 11 - Load Balancing.

I den nuværende konfiguration peger hvert cluster kun på én destination (fx primary → https://basket-api:5001), men ved at have LoadBalancingPolicy = RoundRobin er Gateway'en allerede forberedt til horisontal skalering. Hvis vi senere tilføjer flere instanser, fx basket-api-2, kan vi blot tilføje endnu en destination under samme cluster, og YARP vil automatisk fordele trafikken i round-robin: første request til instans 1, næste til instans 2, osv.

Det betyder, at Gateway'en både fungerer som indgang til systemet og som central load balancer, der kan fordele load jævnt mellem flere containere for hver microservice – uden at klienterne behøver kende til de enkelte instanser.

4.5 Circuit Breaker

I projektets arkitektur er **Circuit Breaker**-mønsteret implementeret i **API Gateway**-laget (API.Gateway) ved hjælp af *YARP*. Implementeringen er konfigureret i filen appsettings.json og anvender en strategi kendt som "Passive Health Checks" til at overvåge og afbryde trafikken til fejlramte mikroservices.

I modsætning til et traditionelt kode-baseret Circuit Breaker (som fx brug af *Polly*-biblioteket direkte i C#-koden), udnytter gatewayen her infrastrukturens evne til at analysere den faktiske trafikstrøm. Konfigurationen ses under HealthCheck-sektionen for de enkelte klynger (*Clusters*), eksempelvis for basket-cluster:

```
58          "HealthCheck": {
59            "Passive": {
60              "Enabled": true,
61              "Policy": "TransportFailureRate",
62              "ReactivationPeriod": "00:00:10"
63            }
```

Figur 12 - Circuit Breaker.

Følgende elementer i konfigurationen gør, at den opfylder Circuit Breaker mønsteret:

- **Fejldetektering:** Feltet "Policy": "TransportFailureRate" angiver, at gatewayen overvåger de mislykkede anmodninger til den pågældende service (fx. basket-api). Hvis fejlraten overstiger en intern tærskelværd (sat til “TransportFailureRate”), slår den forespørgslerne fra.
- **Afkølingsperiode:** Feltet "ReactivationPeriod": "00:00:10" definerer, hvor længe Circuit Breakeren skal forblive åben. I dette tilfælde vil gatewayen stoppe med at sende trafik til den pågældende service i 10 sekunder, hvilket giver servicen tid til at komme sig (også kaldet Cool-down perioden).
- **Genaktivering:** Efter de 10 sekunder vil gatewayen forsøge at sende trafik igen. Hvis anmodningerne lykkes, vil normalt drift genoptages.

Dette design sikrer, at systemet er robust (resilient), da det forhindrer kaskadefejl. Ved at stoppe trafikken til fejlslagen service undgår systemet at spilde ressourcer på kald, der alligevel ville fejle, og forhindrer samtidig at overbelaste en service.

5. Kommunikation mellem Microservices

I vores projekt har vi valgt asynkron kommunikation som en central del af arkitekturen. Vi bruger en message broker (RabbitMQ) til at sende beskeder mellem services.

Vi har valgt at bruge en **Publish/Subscribe** tilgang til det. Herunder har vi taget et eksempel med:

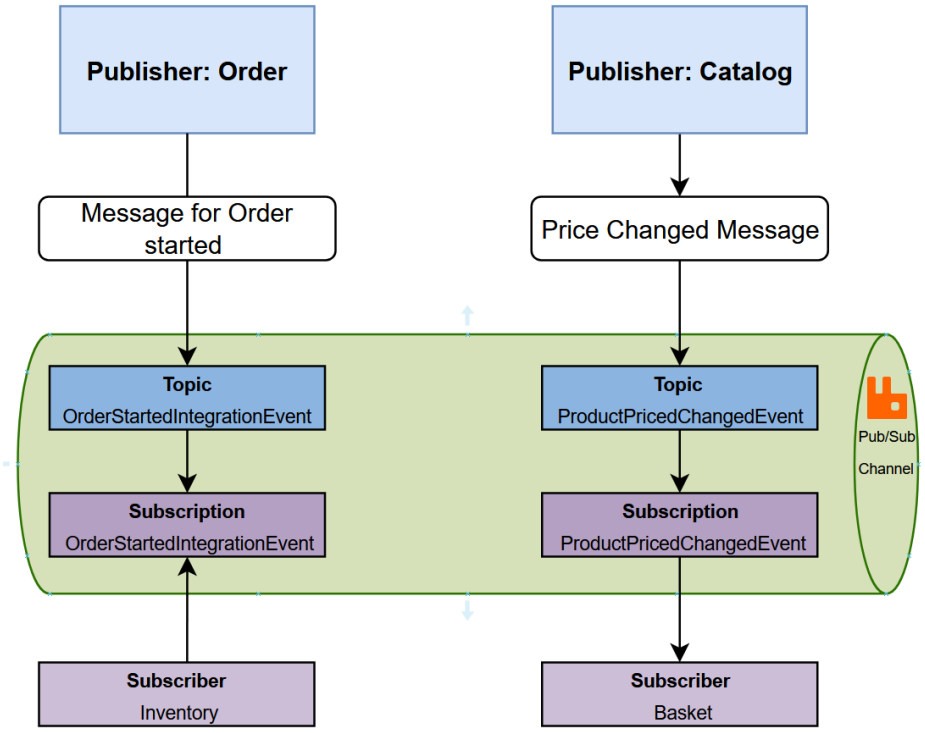


Figure 13 - Pub/Sub Channel.

Når en ordre oprettes, publicerer **Order-servicen** et *OrderStartedIntegrationEvent*.
Når en pris ændres i kataloget, publicerer **Catalog-servicen** et *ProductPriceChangedEvent*.

Begge events sendes til et tilsvarende **topic** i den fælles *Pub/Sub-kanal*, som repræsenteres af det store grønne område i diagrammet (implementeret i RabbitMQ).

Hver eventtype har sit eget topic, og under hvert topic findes én eller flere **subscriptions**. Services, der har brug for eventet, abonnerer på den relevante subscription:

- **Basket-servicen** abonnerer på *ProductPriceChangedEvent* og opdaterer dermed priser i brugerens basket, når den modtager beskeden.

- **Inventory-servicen** abonnerer på *OrderStartedIntegrationEvent* og kan eksempelvis reservere lager, når en ordre påbegyndes. Dette gør Basket-servicen også, men dette er ikke vist her.

Vi har i projektet udarbejdet flere af disse, men valgte at tage disse to med som eksempler.

Vi valgte denne tilgang, fordi den giver nogle vigtige fordele i vores hybride setup med både monolit og microservices:

Afkobling og robusthed: Services bliver ikke afhængige af at være oppe på samme tid. Hvis en *Consumer* (Subscriber) er nede, bliver beskeden liggende sikkert i *Queue*, indtil den kan behandles. Afsenderen kører videre uden at blive påvirket.

5.1 Enterprise Integration Patterns

Systemets kommunikation er designet ud fra *Enterprise Integration Patterns* (EIP). Diagrammet viser dataflowet mellem microservices og de centrale integrationselementer: API Gateway, RabbitMQ og de enkelte services. Kommunikationsmønsteret er baseret på Publish/Subscribe, hvor events sendes til en fælles message broker[2] (RabbitMQ), hvorefter relevante services modtager og behandler dem. Dette sikrer løs kobling og gør det muligt at udvide eller ændre services uden at påvirke resten af systemet.

Typen af Messages vi har sendt er **Event Messages**, hvilket bruges til pålidelig asynkront event notifikationer mellem applikationer.

5.2 Event-Driven Design

Arkitekturen anvender et event-drevet design, hvor microservices ikke kommunikerer direkte med hinanden, men udveksler information gennem domænespecifikke events. Når en service ændrer sin tilstand (f.eks. ved checkout eller ordreoprettelse), publicerer den et event, som andre services kan abonnere på. Dette giver høj skalerbarhed, reducerer afhængigheder og gør det muligt at tilføje nye funktioner uden at ændre eksisterende API'er. Event-Driven Design understøtter samtidig en robust og fejltolerant arkitektur, da services kan være midlertidigt utilgængelige uden at stoppe hele systemet.

[2] https://www.rabbitmq.com/tutorials/

5.3 Implementering

Kommunikationen mellem services implementeres via RabbitMQ som message broker. I stedet for at bruge eksterne biblioteker som MassTransit, er der valgt at bygge videre på Microsofts oprindelige EventBus-implementering fra eShop[3], som er blevet opdateret og moderniseret til at fungere med den nuværende .NET-version og projektets behov. EventBus-laget håndterer oprettelse af exchanges, routing af beskeder samt registrering og håndtering af event handlers.

Denne tilgang giver fuld kontrol over integrationslaget, minimerer kompleksitet og gør det nemt at udvide systemet med nye events eller nye microservices uden at introducere yderligere afhængigheder.

6. Test strategier

En robust teststrategi er essentiel for at sikre både de enkelte services' korrekte funktion, og hele systemets samlede pålidelighed. En effektiv strategi bygger på testpyramiden[4], der balancerer tre niveauer af tests: unittests, integrationstests og end-to-end tests. Hvert niveau har et specifikt formål.[5]

6.1 Unit Testing

Den primære teststrategi, der er implementeret i projektet, er unittests.

Formål: Formålet med en unittest er at validere den mindste logiske enhed af kode – typisk en enkelt metode eller klasse – i fuldstændig isolation fra eksterne afhængigheder.

Implementering i Projektet: Dette er demonstreret i Catalog.API.UnitTests-projektet. I CatalogBrandControllerTests.cs testes CatalogBrandController's metoder.

Eksempel fra Koden:

For at opnå isolation fra databasen, "mockes" CatalogDbContext ved hjælp af Moq-biblioteket.

- I GetCatalogBrands_ReturnsOkWithBrands() testen oprettes en Mock<CatalogDbContext>, og dens Brands DbSet konfigureres til at returnere en foruddefineret liste af brands:
    - var testBrands = new List<Brand>....
- Controlleren instantieres derefter manuelt, og blev mockede med database-context:

[3] https://github.com/dotnet/eShop/tree/main/src/EventBusRabbitMQ
[4] https://automationpanda.com/2018/08/01/the-testing-pyramid/
[5] Newman, S. (2021). *Building Microservices(Testing)CHAPTER 9*

- var controller = new CatalogBrandController(mockContext.Object);.

- Til sidst kaldes metoden controller.GetCatalogBrands(), og resultatet (der returneres en OkObjectResult) valideres med Assert.

```
20          [Fact]
            0 references | noural07, 9 days ago | 1 author, 2 changes
21          public async Task GetCatalogBrands_ReturnsOkWithBrands()
22          {
23
24              var testBrands = new List<Brand>
25          {
26              new Brand { Id = 1, BrandName = "Nike" },
27              new Brand { Id = 2, BrandName = "Adidas" }
28          }.AsQueryable();
29
30
31              var mockSet = testBrands.AsMockDbSet();
32
33
34              var mockContext = new Mock<CatalogDbContext>();
35              mockContext.Setup(c => c.Brands).Returns(mockSet.Object);
36
37              var controller = new CatalogBrandController(mockContext.Object);
38
39
40              var result = await controller.GetCatalogBrands();
41
42
43              var okResult = Assert.IsType<OkObjectResult>(result);
44
45              var returnedBrands = Assert.IsAssignableFrom<List<Brand>>(okResult.Value);
46              Assert.Equal(2, returnedBrands.Count);
47          }
```

Figure 14 - Unit Test Eksempel.

6.2 Perspektivering: Integrationstests

Integrationstests er det næste logiske skridt til hver enkelt microservice.

- **Formål:** At teste en komplet microservice (f.eks. Catalog.API) som en "sort boks", inklusiv dens rigtige infrastruktur-afhængigheder. For Catalog.API betyder det, at testen skal validere, at API'et kan skrive til og læse fra en **rigtig PostgreSQL-database**.
- **Forslag til Implementering:**
    1. **Test Framework:** Man ville anvende Microsoft.AspNetCore.Mvc.Testing-pakken. Denne pakke kan starte Catalog.API-servicen i hukommelsen ved hjælp af en WebApplicationFactory.
    2. **Database:** I stedet for at mocke CatalogDbContext, ville man bruge et bibliotek som **Testcontainers** til at starte en midlertidig, "ægte" PostgreSQL-database i en Docker-container for hver testkørsel.
    3. **Testflow:** Testen ville:

- Starte test-databasen.
- Konfigurere WebApplicationFactory til at overskrive appens ConnectionString til at pege på test-databasen.
- Køre db.Database.Migrate() på test-databasen for at sikre, at skemaet er opdateret.
- Bruge en HttpClient til at sende et POST-kald til f.eks. /catalog/catalog-items.
- Sende et efterfølgende GET-kald for at verificere, at varen blev gemt korrekt i databasen.

Denne tilgang validerer hele service-laget, fra controller-endpoints til Entity Framework Core-mappings og database-forespørgsler.

6.3 Perspektivering: End-to-End Tests med UI-Automatisering

Toppen af testpyramiden er E2E-tests, som validerer hele systemets værdikæde set fra slutbrugerens perspektiv. I stedet for manuelt at teste systemet, kan dette automatiseres ved hjælp af browser-automatiseringsværktøjer som **Selenium WebDriver**.

Formål: At simulere et reelt brugerflow gennem applikationens grafiske brugergrænseflade (GUI) på tværs af alle microservices. Dette verificerer ikke kun, at services samarbejder korrekt (synkront via Gateway og asynkront via RabbitMQ), men også at frontend-logikken, routing, JavaScript og HTML-elementer fungerer som forventet for brugeren.

Forslag til Implementering:

1. **Testmiljø:** Den mest pålidelige metode er at køre hele applikations-stacken ved hjælp af docker compose up. Dette starter alle backend-services (api-gateway, basket-api, catalog-api, order-api, identity-api, rabbitmq, postgres) samt frontend-applikationen.
2. **Værktøjer og Indgangspunkt:** Testen fungerer som en "robot-bruger". Vi anvender et test-projekt med **Selenium WebDriver**, der styrer en rigtig browser (f.eks. Chrome eller Edge). Testen tilgår applikationen via dens offentlige URL (f.eks. http://localhost:7163 gennem gatewayen), præcis som en rigtig kunde ville gøre.
3. **Testflow (Eksempel: "Kunde Checkout"):**
    - **Trin 1 (Browsing):** Selenium åbner browseren på forsiden. Den finder et produkt i kataloget (f.eks. ved at søge efter CSS-klassen .esh-catalog-item) og klikker på knappen "Add to Basket".

- **Trin 2 (Kurv):** Browseren navigerer til kurven (/Basket). Testen verificerer visuelt, at varen ligger i listen.
- **Trin 3 (Checkout):** Selenium finder og klikker på knappen "Checkout".
- **Bag scenen:** Når knappen aktiveres, sender frontend-appen et kald til Basket.API, som via RabbitMQ trigger Order.API til at oprette ordren asynkront.
- **Trin 4 (Login - hvis påkrævet):** Hvis systemet kræver login, vil Selenium udfylde brugernavn og adgangskode i login-formularen og indsende den.

4. **Verificering:** Testen bestås, når browseren omdirigeres til en "Tak for din ordre"-side, og testen kan finde et specifikt HTML-element, der bekræfter ordren (f.eks. en <h1> med teksten "Order completed" eller lignende succes-besked i Success.cshtml).

Disse tests giver den højeste grad af sikkerhed, da de beviser, at hele systemet virker. De er dog langsommere at afvikle end API-tests, da en browser skal startes og rendere siderne, og de kan være mere sårbare over for små ændringer i HTML-strukturen.

7. Observability (Overvågning og Drift)

Et distribueret system kræver en robust tilgang til driftsmæssig indsigt. Efterhånden som den monolitiske arkitektur er transformeret til uafhængige microservices, er centraliseret overvågning blevet et fundamentalt krav for at opretholde systemets pålidelighed og ydeevne.

7.1 Formål

Formålet med monitoring- og logging er at give et samlet overblik over hele systemets tilstand, performance og stabilitet. Dette er f.eks. statistikker, data og realtids målinger som hjælper udviklere med at identificere fejl. Derudover bruges det også til at optimere ydeevnen og reagere hurtigt når en service eller flere går ned.

- **Samlet overblik**: Det er vigtigt at samle data fra alle services ét sted, så udviklere ikke skal tilgå hver enkelt microservice individuelt.
- **Mean Time To Recovery**: Ved at gøre brug af logs, metrics og traces, kan tiden for fejlfinding reduceres markant.
- **Performance**: Med målinger kan man bla. se om der performance problemer. Dette kan bla. være responstider.

7.2 Logs[6]

Hver microservice gør brug af Serilog til logging. Dette gør det muligt at indsamle logdata fra de forskellige microservices. Vores logs bliver gemt på to forskellige måder: Vi skriver dem både til en tekstfil og sender dem videre til *Loki*, som gør det muligt at søge og analysere i logdata. Til visualisering af disse logs benyttes *Grafana*.

Effektiv og hyppigt logging er essentielt for fejlsøgning. Hver microservice i projektet implementerer *Serilog* til håndtering af logs.

Log flowets design:

1. **Indsamling af Logs:** De forskellige microservices generer logs med *Serilog*. Metadata som CorrelationId, Microservice navn og environment udfyldes automatisk på hver log entry.

2. **Lagring af logs**: Logs i systemet sendes til to primære *sinks*:
    a. **Tekstfil**: Der bliver oprettet en tekstfil for hver Microservice. På denne fil bliver alle logs skrevet.
    b. Loki: Logs sendes asynkront til Grafana Loki.

3. **Visualisering**: Til visualisering har vi gjort brug af Grafana. I Grafana kan udviklere søge i logs med LogQL. Og da vi også gør brug af OpenTelemetry, kan en log direkte linke til en specifik trace, som giver konteksten bag den request, der forårsagede loggen.

7.3 Monitoring, Metrics og Tracing

Vi har gjort brug af kombinationen af Prometheus, Grafana og OpenTelemetry.

Her er Docker imagen (i compose) som vi har gjort brug af:

[6] Se Bilag 2

```yaml
otel-lgtm:
  container_name: grafana-aio
  image: grafana/otel-lgtm:latest
  ports:
    - "3000:3000"
  volumes:
    - grafana-data:/var/lib/grafana
    - prometheus-data:/var/lib/prometheus
    - loki-data:/var/loki
    - tempo-data:/var/tempo
  restart: unless-stopped
```

Figur 15 - Docker Compose eksempel af otel-lgtm imagen.

Dette er et docker image som inkluderer alle nødvendige værktøjer til *Observability*. Den deployer Grafana, Prometheus, Loki og Tempo alle på en gang.

7.3.1 OpenTelemetry [7](Instrumentation)

Laget hvor der opsamles data inde i koden. OpenTelemtry sørger for at *metrics* som f.eks. 'Antal Requests' og traces som fx. "hvor lang tid tog X" indsamles ensartet på tværs af microservices, men også på tværs af programmeringssprog og frameworks.

7.3.2 Prometheus[8] (Metrics)

Prometheus fungerer som en *Time Series* Database. Konfigureret til at indsamle metrics fra de forskellige microservices der er i projektet. Dette overvåger kritiske punkter som:

- CPU og RAM forbrug
- Request Rates
- Error Rates

7.3.3 Grafana Tempo[9]

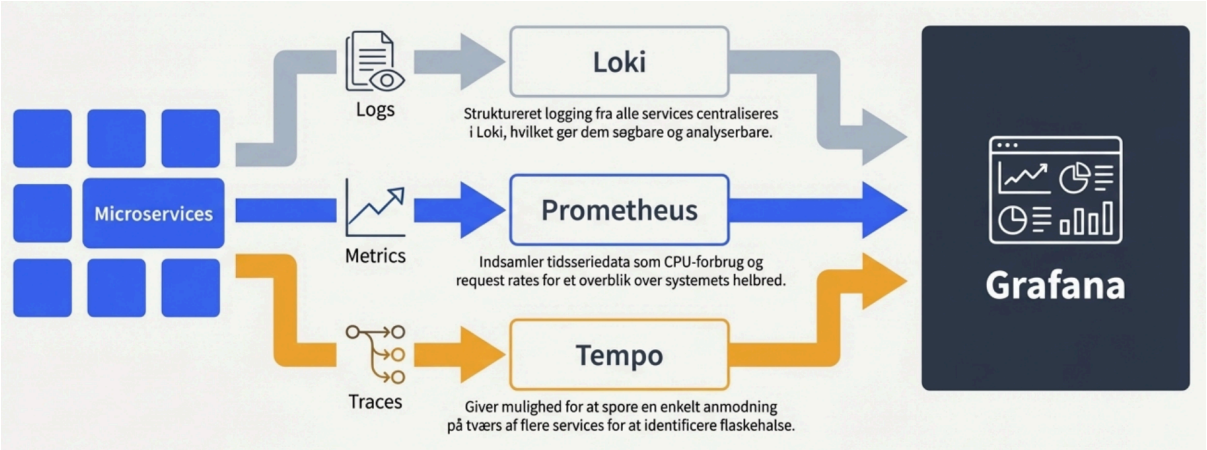
Dette sørger for at spore en request gennem flere services. En handling tildeles en *TraceId*. Det vil sige at man kan følge en handling gennem hele systemet og præcist se hvor lang tid hvert led i kæden tager og hvor en eventuel forsinkelse opstår.

[7] https://opentelemetry.io/ - Open Telemetry
[8] https://prometheus.io/ - Prometheus
[9] https://grafana.com/docs/tempo/latest/ - Tempo for Traces

7.3.4 Grafana Visualisering

Alle data fra Prometheus, Loki og Tempo samles i Grafana Dashboards. I Grafana kan man se grafer over systemets helbred, og hvis der er behov for det, kan man dykke ned i detaljerne. Dette kaldes også for "Single Pane of Glass" hvor man sørger for at samle alt ét sted. Dette gør man for at gøre det nemmere for en udvikler at fejlfinde, og se eventuelle udfordringer med systemet.

7.3.5 Overordnet Diagram for Observability



Figur 16 - Observability Diagram.

Som man kan se på figuren,

7.4 Black Box Monitoring (Gatus)

Der er blevet anvendt Gatus[10] til uptime- og health monitoring. Prometheus og Loki giver dybere indsigt i systemets indre (som f.eks. logs og traces). Man kan dog bruge Gatus til et hurtigere overblik, nærmest set ud fra en brugers tilgang eller perspektiv. Vores formål med Gatus har været at kunne få et overblik over de forskellige services hurtigt og se responstiden fra dem.

7.4.1 Health Checks med Gatus

Gatus konfigureres med en YAML-fil og udfører health checks på:

- **Availability**: Ser om servicen overhovedet svarer tilbage. Hvis en service returnerer en Ok status code (fra HTTP), så betyder at den svarer. Hvis ikke, så er der noget galt.
- **Perfomance**: Gatus tjekker også om services svarer indenfor de forventede tidsrammer. Det vil sige at man kan konfigurere den til f.eks. at en service **skal** svare indenfor 150ms.

[10] https://gatus.io/

- **Integrity**: Man kan tjekke for en specifik body i responsen. Her tjekker man om servicen returnerer det forventede data.

Her er YAML-filen som vi har anvendt:

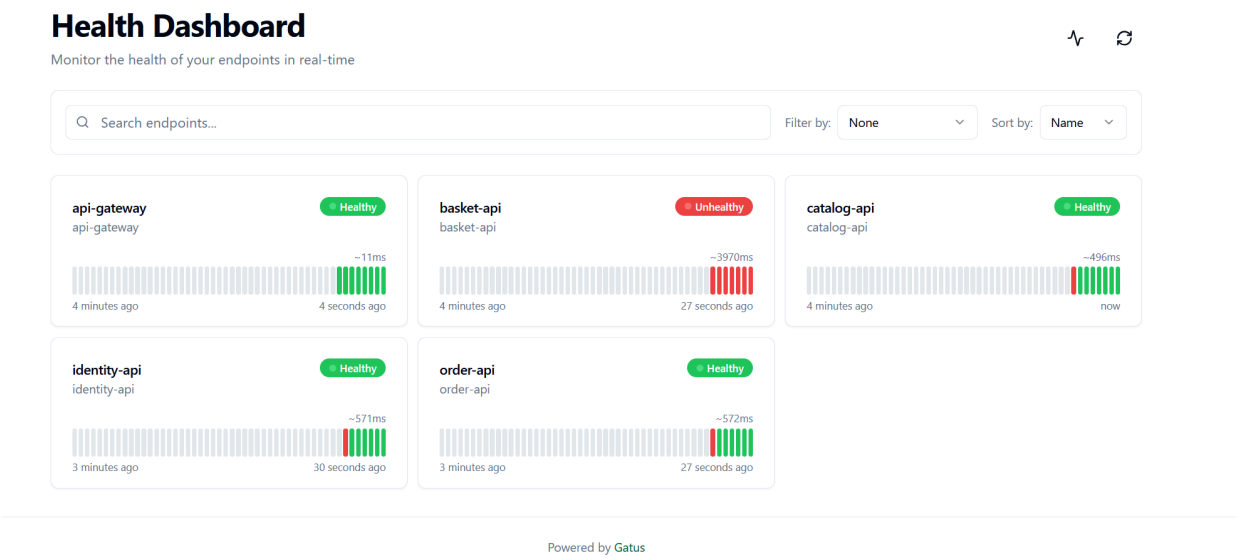
```yaml
endpoints:
  - name: basket-api
    url: "http://basket-api:8080/health"
    interval: 30s
    conditions:
      - "[STATUS] == 200"
      - "[BODY] == pat(*Healthy*)"
      - "[RESPONSE_TIME] < 150"

  - name: api-gateway
    url: "http://api-gateway:8080/health"
    interval: 30s
    conditions:
      - "[STATUS] == 200"
      - "[BODY] == pat(*Healthy*)"
      - "[RESPONSE_TIME] < 150"

  - name: catalog-api
    url: "http://catalog-api:8080/health"
    interval: 30s
    conditions:
      - "[STATUS] == 200"
      - "[BODY] == pat(*Healthy*)"
      - "[RESPONSE_TIME] < 150"

  - name: identity-api
    url: "http://identity-api:8080/health"
    interval: 30s
    conditions:
      - "[STATUS] == 200"
      - "[BODY] == pat(*Healthy*)"
      - "[RESPONSE_TIME] < 150"

  - name: order-api
    url: "http://order-api:8080/health"
    interval: 30s
    conditions:
      - "[STATUS] == 200"
      - "[BODY] == pat(*Healthy*)"
      - "[RESPONSE_TIME] < 150"
```

7.4.2 Alerting med Gatus

Når en uregelmæssighed bliver registreret i produktionen, kan Gatus sende notifikationer i f.eks. Slack eller Discord, så udviklere kan se hvilken service der er gået ned og efterfølgende handle på det.

7.4.3 Gatus Health Dashboard

Her er et eksempel af hvordan Gatus ser ud i projektet:



Figur 17 - Gatus Health Dashboard.

Værktøjet er godt til at få et hurtigt overblik over hvilke services der er nede eller oppe.

7.5 Traces

I projektet anvendes *distributed tracing*[11] til at følge en request på tværs af flere microservices. Formålet er at kunne se hele "rejsen" for et request – fra API Gateway, gennem Basket, videre til Order og Catalog – samlet i ét trace med ét fælles trace-id. Det gør det langt nemmere at fejlsøge performanceproblemer og forstå, hvor i kæden en flaskehals eller fejl opstår.

OpenTelemetry bruges til at instrumentere .NET-services. Når en HTTP-forespørgsel rammer f.eks. Basket.API, oprettes der automatisk et *span*, som repræsenterer den enkelte operation (endpoint, responstid, statuskode).

Når Basket derefter publicerer et RabbitMQ-event (BasketCheckedOutIntegrationEvent), og Order.API modtager det, videreføres samme trace-kontekst. Order publicerer efterfølgende OrderCreatedIntegrationEvent, som Catalog.API modtager. På den måde kan man visualisere hele flowet gennem systemet:

Gateway → Basket → RabbitMQ → Order → RabbitMQ → Inventory

Traces exporteres via OpenTelemetry som OTLP[12]-data til en tracing backend, hvor **Grafana Tempo** anvendes som lagringsmotor for *spans*. Tempo er optimeret til at gemme mange traces billigt og uden behov for en tung database. I Grafana kan udviklere derefter:

- Søge på trace-id, operation (f.eks. GET /basket/api/Basket/{customerId}) eller service-navn.
- Se et "gantt"-lignende overblik over et helt trace, hvor hvert span (Gateway, Basket, Order, Catalog) vises med start-/sluttid.
- Identificere, hvilken service der bruger mest tid, og om der er fejl (5xx, exceptions) i et bestemt led.

En typisk brugssituation i projektet er fejlsøgning på checkout-flowet: Hvis en bruger oplever, at "Checkout hænger", kan man i Grafana finde et trace fra API Gateway, se at Basket svarer hurtigt, men at Order eller Catalog bruger længere tid (eller returnerer en fejl). Fordi traces er korreleret med

[11] YARP Distributed tracing:
https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/yarp/distributed-tracing?view=aspnetcore-10.0
[12] OpenTelemetry Protocol

logs (Serilog → OTLP → Grafana), kan man fra et trace hoppe direkte til de tilhørende loglinjer for den samme request.

Samlet set giver OpenTelemetry + Tempo + Grafana et samlet observability-lag, hvor vi ikke kun kan se om en service er oppe (Health Checks/Gatus), men også præcist se, hvor i et distribueret flow tiden bruges, og hvor eventuelle fejl opstår. Det er særligt vigtigt i en microservice-arkitektur, hvor et enkelt bruger *request* ofte involverer flere services og flere asynkrone hop via RabbitMQ.

8. Konklusion

8.1 . Konklusion

Projektet har demonstreret, hvordan en klassisk monolitisk applikation som eShopOnWeb kan migreres til en moderne microservice-arkitektur ved anvendelse af Strangler Pattern. Gennem arbejdet er centrale dele af systemet blevet til selvstændige services, uden at frontend er ændret, og uden at systemet har mistet funktionalitet undervejs. Dette viser, at en gradvis migration er både teknisk mulig og praktisk anvendelig i større systemer.

Projektets arkitektur understøtter alle krav til løst koblede services. Hver microservice har sit eget domæneansvar, og sin egen PostgreSQL-database (dog har Basket en Redis cache), og data udveksles ikke direkte mellem services, men via asynkrone RabbitMQ-events. Denne tilgang reducerer afhængigheder, minimerer risikoen for fejl og gør systemet robust over for nedetid i enkelte komponenter.

Strangler Pattern har vist sin styrke ved at muliggøre en kontrolleret og sikker overgang. API Gateway (YARP) fungerer som en fleksibel indgang, hvor eksisterende endpoints fra monolitten gradvist kan omdirigeres til microservices. Dette gør det muligt at udskifte funktioner uden at påvirke brugeroplevelsen eller ændre frontend-koden.

Projektet har også vist vigtigheden af en helhedsorienteret arkitektur. Logging (Serilog, OTLP, Grafana), overvågning (Gatus), health checks, CI/CD og containerisering (Docker Compose) har sikret, at systemet kan driftes, monitoreres og videreudvikles på et professionelt niveau. Implementeringen af asynkron kommunikation mellem Basket, Order og Catalog har været central for at håndtere lagerstyring og ordreflow uden stram kobling mellem services.

Lagerstyringssystemet blev implementeret som en selvstændig service, der reagerer automatisk på ordreoprettelse. Dette viser, hvordan event-drevet arkitektur kan udvide funktionalitet uden at ændre eksisterende services, og samtidig løse konkrete forretningsbehov som lageropdatering og data-konsistens.

Derudover opfylder det de både faglige mål for **Store Systemer og Systemintegration** faget samt alle tekniske krav i opgavebeskrivelsen[13]. Samlet set har projektet demonstreret en fuld, realistisk og skalerbar microservice-løsning, der erstatter centrale funktioner i en monolit uden at kompromittere drift, sikkerhed eller systemkvalitet.

[13] se 1.2 for krav

9. Refleksion

9.1 Refleksion over Migreringsstrategi

I forbindelse med refaktoriseringen af eShopOnWeb-monolitten til en mikroservice-arkitektur, har vi benyttet os af **Strangler Fig Pattern**. Formålet med dette mønster er at udskifte funktionalitet i monolitten inkrementelt, fremfor at foretage et "Big Bang"-rewrite, hvilket minimerer risikoen for systemnedbrud og datatab.

Inden for Strangler Pattern kunne vi vælge tre primære implementeringsstrategier: **UI Strangler**, **Gateway First** og **Lazy Migration**.

1. **UI Strangler:** Denne tilgang indebærer, at migreringen sker for frontend-laget, hvor en ny brugergrænseflade kan sammensætte data fra både monolitten og microservices, eller genbruge den gamle frontend. På den måde kan frontend dirigere trafik til både monolitten og microservices.
2. **Gateway First:** Ved denne strategi placeres en API Gateway foran systemet som det allerførste skridt. Gatewayen agerer facade og router trafikken enten til monolitten eller til de nye services. Dette giver mulighed for at skjule ændringer i arkitekturen for klienterne.
3. **Lazy Migration:** Vi valgte denne tilgang, da den fokuserer på at udtrække funktionalitet "efter behov" eller domæne-vis. Funktionalitet flyttes kun ud af monolitten, når det giver værdi, eller når det specifikke domæne skal implementeres.

9.2 YARP

En API Gateway, I modsætning til ældre gateways som Ocelot, har vi valgt at implementere **YARP**[14] som vores reverse proxy. Valget af YARP er begrundet i dets tætte integration med .NET og høje performance. I Program.cs for gatewayen har vi desuden implementeret *Rate Limiting* ("fixed window"), der tillader op til 20 anmodninger per 10. sekund. Dette tilføjer et nødvendigt sikkerhedslag, der beskytter de bagvedliggende microservices mod overbelastning.

[14] YARP Gateway - https://learn.microsoft.com/en-us/aspnet/core/fundamentals/servers/yarp/getting-started?view=aspnetcore-10.0

9.3 Infrastruktur og Containerisering:

Monolitten kunne køre som en enkelt proces, kræver den nye løsning orkestrering af flere dele (Gateway, Identity, Catalog, Basket, RabbitMQ, PostgreSQL). Brugen af **Docker** og Docker Compose har været essentiel for at sikre et ensartet udviklingsmiljø, hvilket har betydet, at den samme version af projektet har virket på alles computer. Opsætningen af netværk mellem containere og styring af miljøvariabler (som ses i vores compose.yaml og appsettings.Docker.json filer) kræver betydeligt mere vedligeholdelse end den oprindelige monolit, men er en forudsætning for at kunne skalere services individuelt.

9.4 Centraliseret Identitet:

I monolitten var bruger-sessionen og autentificeringen tæt koblet til applikationen via cookies. I den nye arkitektur skal gatewayen og de enkelte services kunne validere tokens (JWT). Dette stiller større krav til styring af secrets og certifikater, men muliggør til gengæld, at vi kan have forskellige klienter (web, mobil, 3. part) koblet på samme API i fremtiden.

10. Litteraturliste

Hohpe, G., & Woolf, B. (2004). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley.

Newman, S. (2021). *Building Microservices* (2nd Edition). O'Reilly Media.

Microsoft. *ASP.NET Core Documentation*.
https://learn.microsoft.com/aspnet/core/

Microsoft. *YARP – Yet Another Reverse Proxy*.
https://microsoft.github.io/reverse-proxy/

Microsoft. *eShopOnWeb Reference Application*.
https://github.com/dotnet-architecture/eShopOnWeb

RabbitMQ. *RabbitMQ Tutorials – .NET Client*.
https://www.rabbitmq.com/tutorials/

PostgreSQL Global Development Group. *PostgreSQL Documentation*.
https://www.postgresql.org/docs/

Redis. *Redis Documentation*.
https://redis.io/documentation

Docker Inc. *Docker Documentation*.
https://docs.docker.com/

Grafana Labs. *Grafana Documentation*.
https://grafana.com/docs/

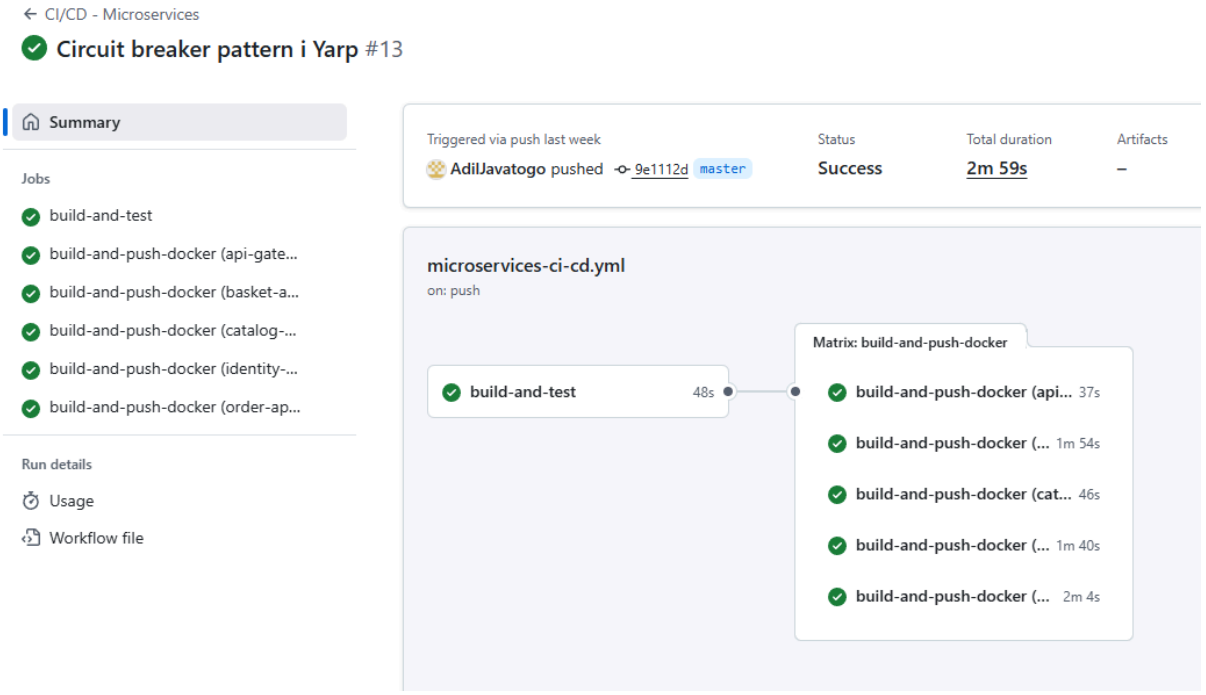
OpenTelemetry. *OpenTelemetry Documentation*.
https://opentelemetry.io/docs/

UCL Erhvervsakademi. *Store Systemer og Systemintegration – Studieordning* (2025).

11. Bilag

Bilag 1

Git actions



Figur 18 - GitHub Action CI/CD Bilag

YML Fil

```yaml
name: CI/CD - Microservices

on:
  push:
    branches: [ master ]
    paths:
      # Lyt efter ændringer i alle relevante microservice- og gateway-mapper
      - 'Gateway/API.Gateway/**'
      - 'Microservices/Basket/eShop.Basket.API/**'
      - 'Microservices/Catalog/Catalog.API/**'
      - 'Microservices/Identity/Identity.API/**'
      - 'Microservices/Inventory/Inventory.API/**'
      - 'Microservices/Order/eShop.Order.API/**'
      # Lyt også på testprojekter og solution-filen
      - 'Catalog.API.UnitTests/**'
      - 'eShopProject.sln'
      # Og workflow-filen selv
      - '.github/workflows/microservices-ci-cd.yml'

  pull_request:
    branches: [ master ]
    paths:
      # Samme mapper som ovenfor
      - 'Gateway/API.Gateway/**'
      - 'Microservices/Basket/eShop.Basket.API/**'
      - 'Microservices/Catalog/Catalog.API/**'
      - 'Microservices/Identity/Identity.API/**'
      - 'Microservices/Inventory/Inventory.API/**'
      - 'Microservices/Order/eShop.Order.API/**'
      - 'Catalog.API.UnitTests/**'
      - 'eShopProject.sln'
      - '.github/workflows/microservices-ci-cd.yml'

```

Figur 19 - YML fil for Workflow.

```yaml
  workflow_dispatch:

jobs:
  # ----------------------------------------------
  # JOB 1: Byg og test hele .NET-løsningen
  # ----------------------------------------------
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      # Trin 1: Hent koden
      - name: Hent koden (Check out repository)
        uses: actions/checkout@v4

      # Trin 2: Sæt .NET 8 SDK op
      - name: Sæt .NET 8 SDK op (Setup .NET)
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '8.x'

      # Trin 3: Gendan NuGet pakker (for hele solution)
      - name: Gendan NuGet pakker (Restore dependencies)
        run: dotnet restore eShopProject.sln

      # Trin 4: Byg hele løsningen
      - name: Byg løsningen (Build solution)
        run: dotnet build eShopProject.sln --configuration Release --no-restore

      # Trin 5: Kør ALLE tests i solution-filen
      - name: Kør enhedstests (Run unit tests)
        run: dotnet test eShopProject.sln --no-build --configuration Release

  # ----------------------------------------------
```

Figur 20 - YML Fil for Workflow.

```yaml
  # JOB 2: Byg og push Docker images (via Matrix)
  # ------------------------------------------------
  build-and-push-docker:
    # Kør KUN hvis 'build-and-test' jobbet er succesfuldt
    needs: build-and-test

    # Kør KUN på 'master' branchen for at undgå at pushe fra PRs
    if: github.event_name == 'push' && github.ref == 'refs/heads/master'

    runs-on: ubuntu-latest

    strategy:
      # Definer en matrix for hver microservice med en Dockerfile
      matrix:
        include:
          - service-name: 'api-gateway'
            dockerfile-path: './Gateway/API.Gateway/Dockerfile'
            context-path: '.' # .NET Dockerfiles har ofte brug for solution root som context
          - service-name: 'basket-api'
            dockerfile-path: './Microservices/Basket/eShop.Basket.API/Dockerfile'
            context-path: '.'
          - service-name: 'catalog-api'
            dockerfile-path: './Microservices/Catalog/Catalog.API/Dockerfile'
            context-path: '.'
          - service-name: 'identity-api'
            dockerfile-path: './Microservices/Identity/Identity.API/Dockerfile'
            context-path: '.'
          - service-name: 'inventory-api'
            dockerfile-path: './Microservices/Inventory/Inventory.API/src/Dockerfile'
            context-path: './Microservices/Inventory/Inventory.API' # Node.js context er projektmappen
          - service-name: 'order-api'
            dockerfile-path: './Microservices/Order/eShop.Order.API/Dockerfile'
            context-path: '.'
```

Figur 21 - YML Fil for Workflow .

```yaml

    steps:
      # Trin 1: Hent koden
      - name: Hent koden (Check out repository)
        uses: actions/checkout@v4

      # Trin 2: Sæt Docker Buildx op
      - name: Sæt Docker Buildx op (Set up Docker Buildx)
        uses: docker/setup-buildx-action@v3

      # Trin 3: Log ind på Docker Hub
      - name: Log ind på Docker Hub (Login to Docker Hub)
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      # Trin 4: Byg og push Docker image (dette trin kører for hver service i matrixen)
      - name: Byg og push image for ${{ matrix.service-name }}
        uses: docker/build-push-action@v5
        with:
          # Brug context-stien fra matrixen
          context: ${{ matrix.context-path }}
          # Brug Dockerfile-stien fra matrixen
          file: ${{ matrix.dockerfile-path }}
          push: true
          tags: |
            ${{ secrets.DOCKERHUB_USERNAME }}/${{ matrix.service-name }}:latest
            ${{ secrets.DOCKERHUB_USERNAME }}/${{ matrix.service-name }}:${{ github.sha }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

Figur 22 - YML fil for Workflow.

Bilag 2

Grafana - logs

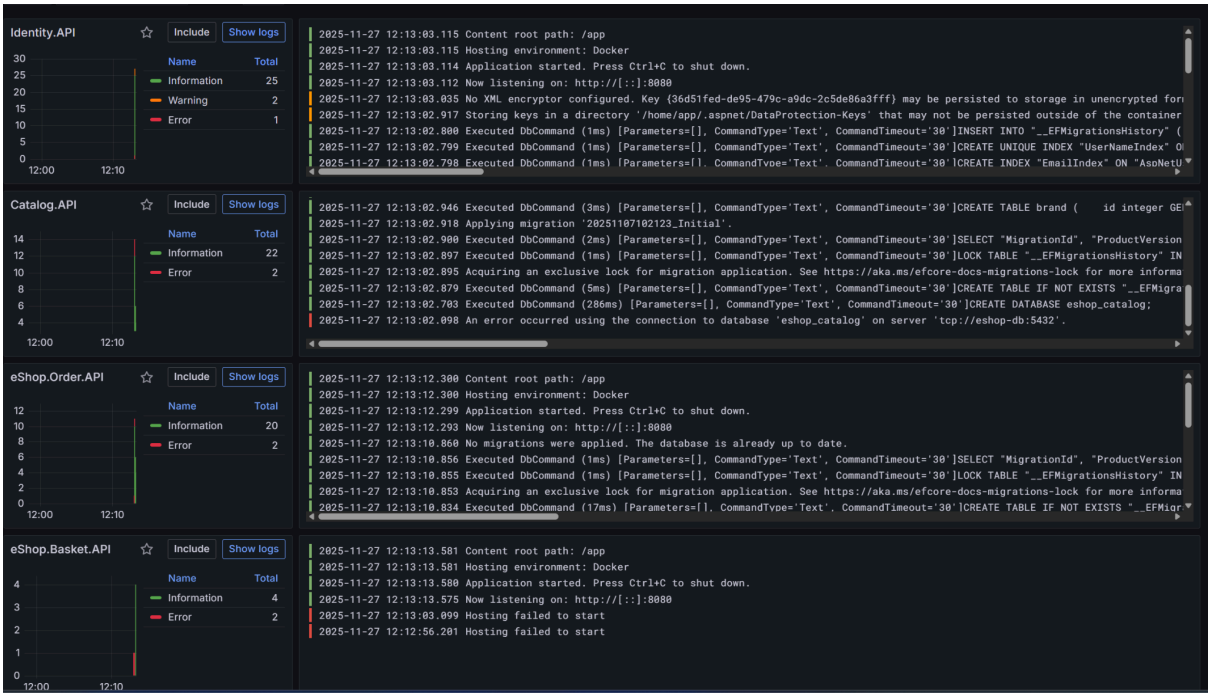


Figure 23 - Grafana Logs Bilag.