

Implementering af Microservices med Messaging og integration til eksterne systemer

Af Mads Mikel Rasmussen

studerende i faget Systemintegration på diplomuddannelsen i
softwareudvikling ved University College Lillebælt Erhvervsakademi og
Professionshøjskole

studienr.: mmra30000

1. februar 2021

Antal tegn: 19458, 8 normalsider.

Indholdsfortegnelse

1	Indledning	3
2	Problemformulering	3
3	Teori.....	4
4	Implementering.....	6
4.1	Arkitektur.....	6
4.2	Teknologier.....	8
4.2.1	gRPC med protobuf-net.....	8
4.2.2	ASP.NET web application som microservice vært.....	8
4.2.3	Klientens kald til en microservice.....	10
4.2.4	Push notifikation med SignalR.....	12
4.3	Workflowet	14
5	Konklusion.....	19
6	Perspektivering.....	19
7	Referencer	20

1 Indledning

Der er valgt at konstruere et nyt system med det formål at belyse min opnåede viden, færdigheder og kompetencer i kursets læringsmål, samt at illustrere proof-of-concept. Til dette formål er der lavet følgende case:

Systemet skal hente data fra Kystdirektoratet og Dansk Meteorologisk Institut og notificere brugere om potentielt høj vandstand og vindhastighed, ved én udvalgt del af de danske kyster, da netop disse to faktorer kombineret ofte resulterer i oversvømmelser

Rapporten fortsættes herfra med en problemformulering og den teori der er anvendt til belysning heraf, samt til systemets konstruktion. Casens implementering illustreres ved kodeeksempler, der anvender et udvalgt teoretisk grundlag, i konteksten af problemstillingen. Slutteligt konkluderes på problemstillingen, og der perspektiveres til de aspekter af kursets læringsmål der gør sig gældende for den anvendte teori i konteksten af det konstruerede system.

2 Problemformulering

Casen anvendes til illustration af én potentiel løsning på følgende problemstilling:

Hvordan konstrueres et nyt system, der understøtter messaging i en moderne service orienteret arkitektur?

Denne problemstilling rejser følgende spørgsmål, der søges besvaret:

1. Hvilken moderne arkitektur bør anvendes?
2. Hvilke moderne teknologier bør anvendes til sikring af skalabilitet og agilitet i udviklingsprocessen?
3. Hvilke metoder, mønstre og teknikker til systemintegration bør anvendes til casen?
4. Hvilke standarder bør overvejes som dataformat, således en langtidsholdbar løsning opnås?
5. Hvilke forretningsmæssige overvejelser bør drøftes ved etableringen af et nyt system, ud fra et systemintegrationsmæssigt synspunkt?

3 Teori

Jeg redegør her for, hvordan jeg ser Microservices som en moderne variant af Service Orienteret Arkitektur (SOA).

Et system i en SOA, skal ifølge Thomas Erl, opfylde følgende otte designmæssige principper (Erl, 2008):

- **Standardiserede service kontrakter:** Formålet og funktionaliteten af en service bør eksponeres til klienter på en standardiseret måde, herunder definitionen af datamodeller og datatyper, til sikring af styrbarheden, pålideligheden og konsistensen af servicens endpoint.
- **Løs kobling:** Både internt i en service og eksternt til andre komponenter, herunder andre services, bør afhængigheder være så begrænset som muligt, med det formål at opnå så stor uafhængighed som muligt i implementeringen, hvilket giver større fleksibilitet i udviklingen og stabilitet i driften.
- **Abstraktion:** Detaljer og information i implementeringen bør eksponeres i mindst mulige omfang, hvilket afspejles i sammensætningen med f.eks. andre services. Begrebet er meget lig det vi kender fra objekt orienteret programmering.
- **Genbrugelighed:** En service bør kunne opfylde det samme formål, men på differentieret vis, eksempelvis ved parametrisering, og herved kunne genbruges.
- **Autonomi:** En service bør kunne opfylde sit formål, i så stor grad af uafhængighed som muligt, af både andre services og driftsmiljøet. Dette fordrer at en service har en vis grad af styring med driftsmiljøet.
- **Tilstandsløshed:** En service bør eksponere mindst mulig grad af sin logiske tilstand til klienter, hvorved skalabilitet og tilgængelighed fordres. Dette begreb minder meget om det vi kender fra objekters tilstand i objekt orienteret programmering.
- **Opdagelighed:** En service bør af hensyn til både den tekniske og videnmæssige tilgængelighed, men også til den forretningsmæssige forståelse, være let at identificere som et IT aktiv, med de kapaciteter den indeholder.
- **Sammensættelighed:** En service bør kunne deltage i forskellige sammensætninger af services på forskellige tidspunkter, der tilsammen understøtter en positiv forretningsmæssig værdi.

I konteksten af moderne distribuerede systemer og systemintegration, er det naturligt at redegøre for hvordan Microservices bør anskues i konteksten af SOA, og særligt Erls otte principper. Martin Fowler argumenter, at SOA *begrebet*, med

tiden er for omfavnende og diffust (Fowler, 2014), hvor han eksemplificerer SOA med, at en Enterprise Service Bus, løser f.eks. problemet med integration af mange monolitiske applikationers kommunikation, og skaber herved forretningsmæssig værdi. Men Fowler mener ikke dette er ikke formålet med Microservices. I Microservices sammensættes forskellige services til at løse forskellige udfordringer, og skaber herved forretningsmæssig værdi. Den forretningsmæssige fordel ved microservices over traditionel SOA, er at en forretningsmæssig udfordring muligvis allerede kan løses, helt uden at udvikle ny funktionalitet, da kombinationen af eksisterende funktionalitet således allerede kan muliggøre en løsning på den pågældende udfordring. Microsofts officielle holdning er, citat (de la Torre, Cesar; Wagner, Bill; Rousos, Mike;, 2020):

»SOA was an overused term and has meant different things to different people. But as a common denominator, SOA means that you structure your application by decomposing it into multiple services that can be classified as different types like subsystems or tiers. «

Både Fowler og Microsoft, beskriver microservices, som *"en server applikation som et antal små services"* (de la Torre, Cesar; Wagner, Bill; Rousos, Mike;, 2020). Med denne viden, kan man således argumentere for, at en microservice arkitektur opfylder Erls otte principper for en SOA. Dette søges illustreret ved følgende implementering af casen.

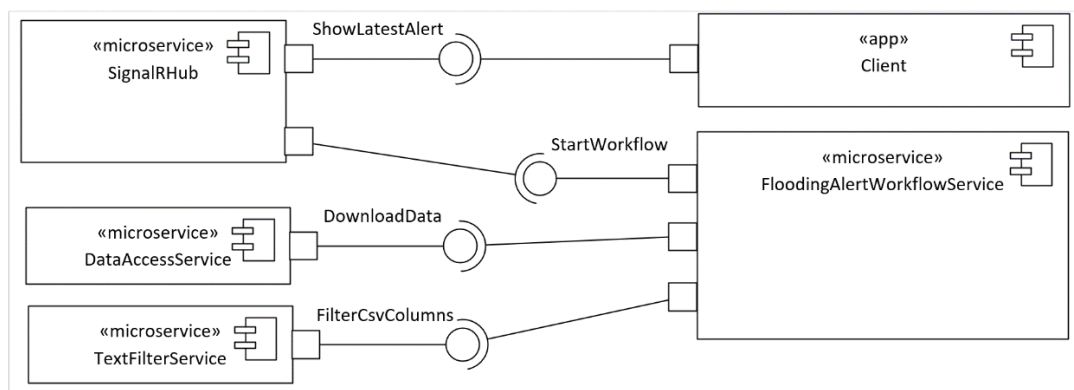
4 Implementering

Casen er implementeret som applikationer i et localhost driftsmiljø. Først præsenteres den valgte arkitektur, og dernæst de teknologier, der er valgt som nødvendige for implementeringen. Vi tager bagefter en roundtrip fra serveren igangsætter workflowet, til klienten modtager notifikationen.

Namespace Corp er anvendt som root namespace.

4.1 Arkitektur

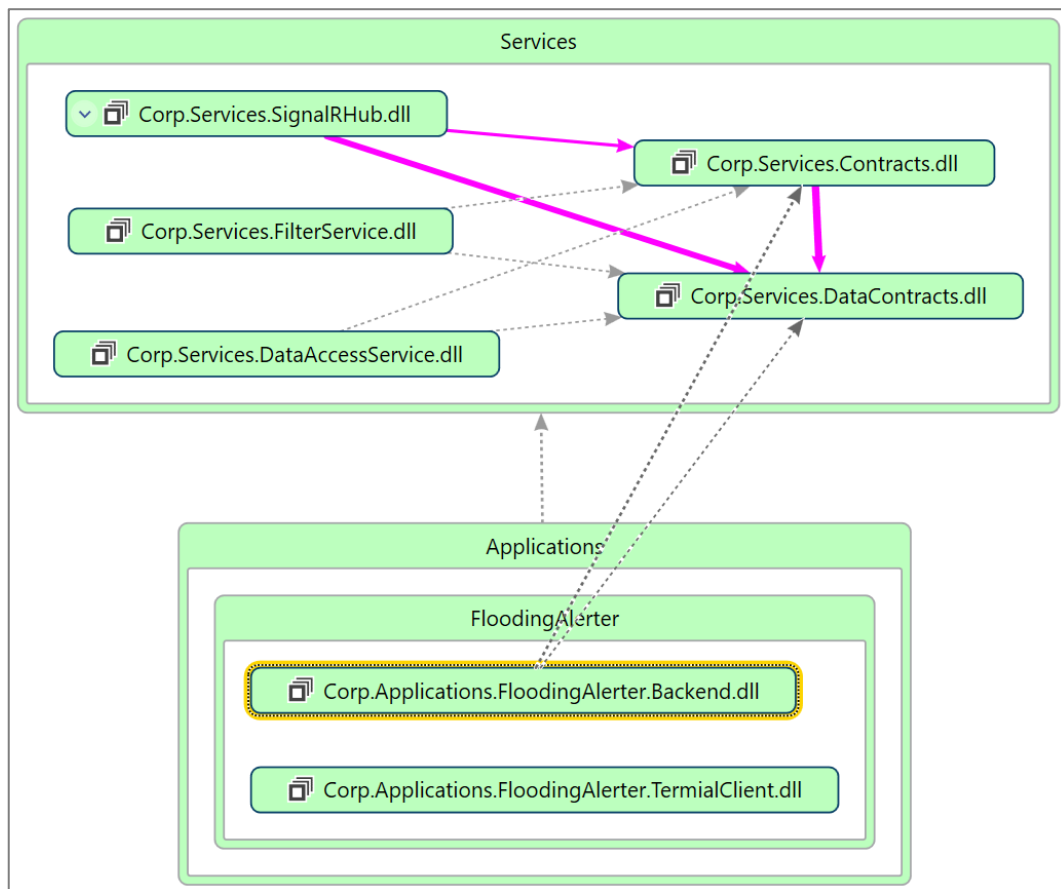
På baggrund af ovenstående teori, er der til casen valgt følgende microservice arkitektur, vist som UML Component diagram:



Figur 1: UML Component diagram over systemets arkitektur

Komponenten SignalRHub agerer server, der automatisk sender varslinger til forbundne klienter. I et nærmere defineret tidsinterval, starter SignalRHub komponenten et workflow, der har til formål at indhente data fra henholdsvis Kystdirektoratet og Dansk Meteorologisk Institut, samt at filtrere og aggregere disse data til en varslingsbesked, inden de forbundne klienter notificeres med denne. Klienten kan være både en hjemmeside, mobilapp eller en hvilken som helt anden type applikation, der understøtter SignalR.

Den logiske arkitektur fremgår af følgende diagram:



Figur 2: Code Map, genereret i Visual Studio. Diagrammet viser logiske referencer mellem .dll assemblies. Lilla pile: funktionskald; grå stiplede pile: assembly referencer. Dette diagram indeholder samme information som et UML package diagram.

Servicekontrakter og deres datakontrakter er separeret ud i hver sit assembly. Dette giver en fordel når der skal tilføjes ny funktionalitet, da der således ikke laves breaking changes. Hver microservice har en assembly reference til disse to assemblies, men foretager ikke nødvendigvis funktionskald (lilla), herunder instantiering af typer. Dette umiddelbare misforhold rundes under næste afsnit.

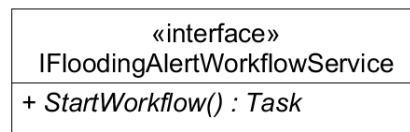
Bemærk at Corp.Applications.FloodingAlerter.Backend også er implementeret som microservice, men placeret logisk som applikation. Dette er valgt, da denne microservice har et domænespecifikt formål, i modsætning til de microservices vi ser i Services. Disse microservices er generelle services, og kan anvendes af alle andre microservices, både de i Services, og de i Applications.

4.2 Teknologier

Her uddybes de anvendte teknologier i proof-of-concept løsningen til casen.

4.2.1 gRPC med protobuf-net

Alle applikationer og biblioteker er skrevet i .NET 5.0 med C# 9.0. Kommunikationen mellem komponenterne foregår med RPC (Remote Procedure Call) i en message channel, og den konkrete teknologi gRPC (Google RPC) er valgt, med en tredjeparts wrapper til .NET (protobuf-net). Denne kombination af teknologier muliggør en væsentlig reduktion af udviklingstiden, da gRPC uden denne wrapper, kræver udvikling af såkaldte *proto* filer, der har til formål at være servicekontrakten. Disse proto filer skal først kompileres af en protoc compiler, der udover .proto filerne, også genererer platformspecifikke filer, der også agerer som servicekontrakt. protobuf-net abstraherer dette og reducerer processen til kun at skulle skabe servicekontrakten, i platformens eget sprog, her C# — på samme måde som man kan gøre i WCF (Windows Communication Foundation). Derfor er det eneste kode der skal genereres til en microservices servicekontrakt, selve interface typen. Her vises UML Class Diagram og koden der anvendes til servicekontrakten for komponenten FloodingAlertWorkflowService, som eksempel herpå:



```

1 using Corp.Services.DataContracts;
2 using System.ServiceModel;
3 using System.Threading.Tasks;
4
5 namespace Corp.Services.Contracts
6 {
7     [ServiceContract]
8     public interface IFloodingAlertWorkflowService
9     {
10         [OperationContract]
11         Task<FloodingAlertWorkflowResponse> StartWorkflow();
12     }
13 }
  
```

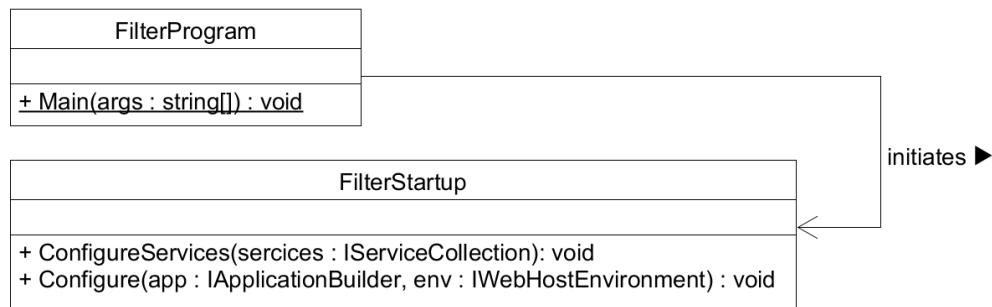
Figur 3: UML klassediagram og kode til IFloodingAlertWorkflowService.

Vi bemærker attributterne ServiceContract og OperationContract på henholdsvis typen og metoden.

4.2.2 ASP.NET web application som microservice vært

Applikationerne der agerer microservices er implementeret som ASP.NET web applikationer med Kestrel, der fungerer som letvægts webserver. Dette gør applikationen i stand til at modtage HTTP forespørgsler ved en angiven port. Når

værten er startet, kan konfigurationen til anvendelse af forskellige services køres. Helt generelt for alle microservices i denne løsning, er det udelukkende opstart og konfiguration af servicen, der er applikationens ansvarsområde. Selve *implementeringen* af servicekontrakten, er separeret ud i et klassebibliotek, Corp.Services.Contracts. Vi ser her et klassediagram for FilterService applikationen, som eksempel og denne fremgangsmåde er gældende for alle microservices i løsningen:



Figur 4: Implementering af microservice værtsapplikation som ASP.NET Core Web Application.

Og den tilhørende kode:

```

1 using Microsoft.AspNetCore;
2 using Microsoft.AspNetCore.Hosting;
3 using Microsoft.AspNetCore.Server.Kestrel.Core;
4 using Microsoft.Extensions.Hosting;
5 using static Corp.Resources.Infrastructure.Endpoints.Services;
6
7 namespace Corp.Services.FilterService
8 {
9     public class FilterProgram
10    {
11        static void Main(string[] args)
12        {
13            CreateHostBuilder(args).Build().Run();
14        }
15
16        public static IWebHostBuilder CreateHostBuilder(string[] args) =>
17            WebHost.CreateDefaultBuilder(args)
18                .ConfigureKestrel(options => options.ListenLocalhost(
19                    FilterServicePort,
20                    ListenOptions => ListenOptions.Protocols = HttpProtocols.Http2)
21                )
22                .UseStartup<FilterStartup>();
23    }
24 }
    
```

Figur 5: Kode til opstart af microservice.

Metoden CreateHostBuilder har til formål at starte applikationen med infrastrukturkonfigurationen, i metodens indhold. Bemærk her, at FilterServicePort er et portnummer, samt at kommunikationsprotokollen sættes til HTTP/2.

```
1 using Corp.Services.Contracts;
2 using Microsoft.AspNetCore.Builder;
3 using Microsoft.AspNetCore.Hosting;
4 using Microsoft.Extensions.DependencyInjection;
5 using Microsoft.Extensions.Hosting;
6 using ProtoBuf.Grpc.Server;
7
8 namespace Corp.Services.FilterService
9 {
10     public class FilterStartup
11     {
12         public void ConfigureServices(IServiceCollection services)
13         {
14             services.AddCodeFirstGrpc();
15         }
16
17         public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
18         {
19             if(env.IsDevelopment())
20             {
21                 app.UseDeveloperExceptionPage();
22             }
23
24             app.UseRouting();
25
26             app.UseEndpoints(endpoints =>
27             {
28                 endpoints.MapGrpcService<TextFilterService>();
29             });
30         }
31     }
32 }
```

Figur 6: Kode til konfiguration af microservice.

ConfigureServices metoden har til ansvar at tilføje protobuf-net som service til applikationen. Denne ene linje sørger for at generere de nødvendige servicekontrakter til brug for den underliggende gRPC teknologi. Linje 28 i Configure metoden, tildeler en bestemt service til applikationens endpoint. Det var dette endpoint der blev opsat i FilterProgram (se forrige figur). Den konkrete klasse TextFilterService implementerer interfacet ITextFilterService. Bemærk at FilterStartup anvender et namespace Corp.Services.Contracts, der er en assemblyreference. Heri er både ITextFilterService og TextFilterService implementeret. Derved er der fuld adskillelse mellem microserviceapplikationen og servicekontraktens implementering.

4.2.3 Klientens kald til en microservice

Når en klient skal gøre brug af en service skal den gøre følgende:

1. Lave en message channel fra klientens endpoint til servicens endpoint.
2. Lave en instans af den pågældende service, ved anvendelse af denne message channel.

3. Kalde en metode på serviceobjektet.

Klienten kan være en ekstern applikation eller en anden microservice. Som en del af workflowet skal der eksempelvis hentes data fra Kystdirektoratet. Servicekontrakten ser således ud:

```
1 using Corp.Services.DataContracts;
2 using System;
3 using System.Net;
4 using System.ServiceModel;
5 using System.Threading.Tasks;
6
7 namespace Corp.Services.Contracts
8 {
9     [ServiceContract]
10    public interface IDownloadDataService
11    {
12        [OperationContract]
13        Task<DownloadDataResponse> DownloadWith(DownloadDataRequest request);
14    }
```

Figur 7: Servicekontrakt for DownloadWith metoden på IDownloadDataService i DataAccess microservicen.

Vi ser at servicekontrakten anvender to typer i request/response mønstret, nemlig DownloadDataRequest og DownloadDataResponse:

```
1 using System;
2 using System.Runtime.Serialization;
3
4 namespace Corp.Services.DataContracts
5 {
6     [DataContract]
7     public class DownloadDataRequest
8     {
9         [DataMember(Order = 1)]
10        public virtual Uri Uri { get; set; }
11    }
12
13    [DataContract]
14    public class DownloadDataResponse
15    {
16        [DataMember(Order = 1)]
17        public virtual byte[] Data { get; set; }
18
19        [DataMember(Order = 2)]
20        public virtual string Message { get; set; }
21    }
22 }
```

Figur 8: Request/Response typer som DataContracts.

Disse typer indkapsler request/response mønstret og implementerer servicekontraktens dataabstraktion.

Her vises hvordan microservicen `FloodingAlertWorkflowService` kalder den eksponerede `DownloadWith` metode, på microservicen `DataAccessService`:

```

38     private async Task<DownloadDataResponse> GetCurrentWaterLevelData()
39     {
40         string url = GenerateCoastDirectorateUrl();
41         Uri uri = new Uri(url);
42         DownloadDataRequest request = new DownloadDataRequest() { Uri = uri };
43         string localhostAddress = $"http://localhost:{DataAccessServicePort}";
44         GrpcChannel channel = GrpcChannel.ForAddress(localhostAddress);
45         GrpcClientFactory.AllowUnencryptedHttp2 = true;
46         DownloadDataResponse response;
47         using(channel)
48         {
49             IDownloadDataService downloadDataService =
50                 channel.CreateGrpcService<IDownloadDataService>();
51             response = await downloadDataService.DownloadWith(request);
52         }
53         return response;
54     }

```

Figur 9: Eksempel på hvordan en klient kalder en eksponeret metode på en microservice.

Linje 44 laver en ny message channel. Linje 49 laver en ny instans af den eksponerede service. Linje 50 foretager et asynkront kald med et request objekt, og modtager et response objekt. Bemærk igen, at selve `DataAccess` microservice kun er refereret gennem `DataAccessServicePort` i URL parameteren til den anvendte message channel.

4.2.4 Push notifikation med SignalR

SignalR er et software framework udviklet af Microsoft, der giver mulighed for nemt at kunne foretage push notifikationer til forbundne klienter, ved metodekald. Altså igen en variant af RPC. SignalR anvender et begreb der kaldes for 'Hub'. En sådan hub eksponerer funktionalitet til at kalde enten alle forbundne klienter, grupper af klienter eller specifikke klienter. Klienten skal blot implementere en metode, hvis navn Hub'en kender og er implementeret som et interface:

```

1  using Microsoft.AspNetCore.SignalR;
2  using System.Threading.Tasks;
3
4  namespace Corp.Services.SignalRHub
5  {
6      public interface IFloodingAlerter
7      {
8          Task ShowLatestAlert(string alert); // The method to be invoked on SignalR clients.
9      }
10
11     public class FloodingAlerterHub: Hub<IFloodingAlerter> { } // Marker class
12 }

```

Figur 10: Specifikation af kontrakt mellem SignalR server og klient. SignalR skal bruge en konkret instans af interfacet på runtime, derfor er der lavet en marker class på linje 11.

Interfacet her specificerer hvilken metoder der skal kaldes på klienten. Klienten implementerer ikke interfacet eksplicit via Roslyn compileren, men implicit (linje 35):

```
21     class Client
22     {
23         HubConnection connection;
24         readonly string url = $"http://localhost:{HubPort}{FloodingAlertHub}";
25
26         public string Id { get; set; }
27
28         public async Task Initialize()
29         {
30             connection = new HubConnectionBuilder().WithUrl(url).Build();
31             connection.On<string>(nameof>ShowLatestAlert), ShowLatestAlert);
32             await connection.StartAsync();
33         }
34
35         public Task ShowLatestAlert(string alert)
36         {
37             Console.WriteLine($"Client {Id}: {alert}");
38             return Task.CompletedTask;
39         }
40     }
```

Figur 11: En SignalR klients implementering af specifikationen for push notifikationer. Koden er fra Console applikationen til casen.

Selve datatransporten foregår ved WebSockets og er derved persistent over tid. Dataformatet er JSON, med metodens navn og eventuelle parametre som indhold. På samme vis som med microservices, er denne server implementeret som en ASP.NET Core Web Application hvorpå der kører en intern service, en instans af Worker der arver fra Background Service, der igangsætter workflowet med hentning af data og efterfølgende notifikation af klienter:

```

15 public class Worker: BackgroundService
16 {
17     private readonly ILogger<Worker> logger;
18     private readonly IHubContext<FloodingAlerterHub, IFloodingAlerter> floodingAlerterHub;
19
20     public Worker(ILogger<Worker> logger, IHubContext<FloodingAlerterHub, IFloodingAlerter>
21         floodingAlerterHub)
22     {
23         this.logger = logger;
24         this.floodingAlerterHub = floodingAlerterHub;
25     }
26
27     protected override async Task ExecuteAsync(CancellationToken stoppingToken)
28     {
29         while(!stoppingToken.IsCancellationRequested)
30         {
31             logger.LogInformation("Worker starting flooding alert workflow at {Time}",
32                 DateTime.Now);
33
34             FloodingAlertWorkflowResponse workflowResult = await GetFloodingAlertData();
35             string alert = ConstructAlertMessage(workflowResult);
36             await floodingAlerterHub.Clients.All.ShowLatestAlert(alert);
37             await Task.Delay(15000); // 15 sec
38         }
39     }
40
41     private async Task<FloodingAlertWorkflowResponse> GetFloodingAlertData()
42     {
43         string localhostAddress = $"http://localhost:{FloodingAlerterWorkflowPort}";
44         GrpcChannel channel = GrpcChannel.ForAddress(localhostAddress);
45         GrpcClientFactory.AllowUnencryptedHttp2 = true;
46         FloodingAlertWorkflowResponse response;
47         using(channel)
48         {
49             IFloodingAlertWorkflowService service =
50                 channel.CreateGrpcService<IFloodingAlertWorkflowService>();
51             response = await service.StartWorkflow();
52         }
53         return response;
54     }
55
56     private string ConstructAlertMessage(FloodingAlertWorkflowResponse r)
57     {
58         return $"Vandstanden er {r.WaterLevel} cm og vinden er {r.WindSpeed} m/s.";
59     }
60 }

```

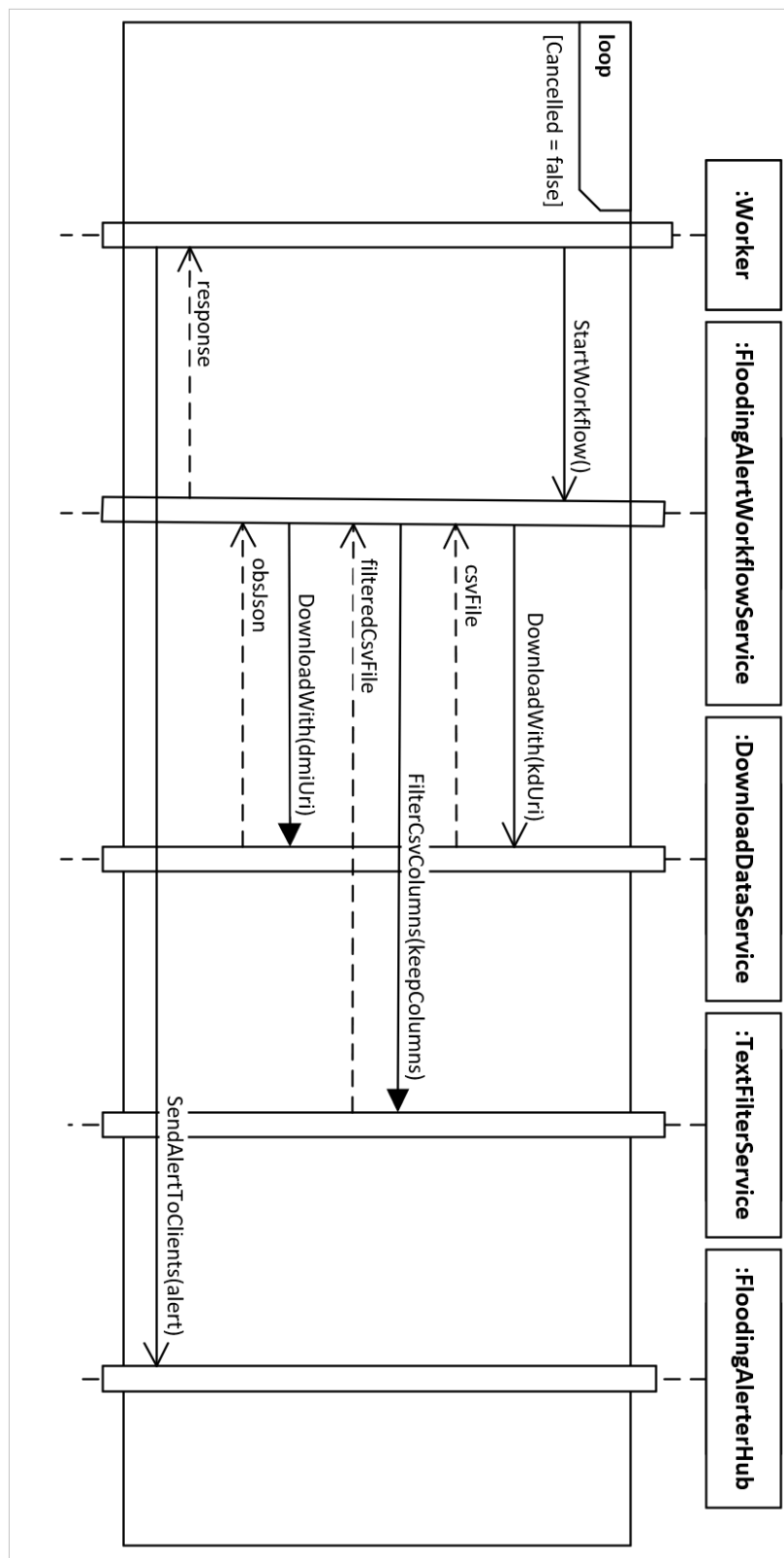
Figur 12: Intern service på SignalR serveren, der igangsætter workflowet og notificerer klienten.

ExecuteAsync metodens ansvar er at igangsætte workflowet og notificere klienterne med en varsling. Vi ser at workflowet startes i linje 32, hvor data hentes med et kald til StartWorkflow metoden på FloodingAlertWorkflow microservicen i linje 48. Der konstrueres en besked på linje 33, der sendes til klienterne via Hub'en på linje 34. Derefter ventes i 15 sekunder inden næste workflow starter.

SignalR anvendt på denne måde, bliver således en dynamisk router (Woolf & Hohpe).

4.3 Workflowet

Det overordnede billede af workflowet illustreres bedst med et UML Sequence diagram:



Figur 13: UML Sequence diagram af workflowet. Som med så mange andre sekvensdiagrammer, er det nødvendigt at lægge dem på siden, for at det hele kan læses.

Vi bemærker først, at hele workflowet er indlejret i en løkke. Det er denne løkke der gentages hvert 15. sekund. Dette tidsinterval bør i realiteten være 10 minutter, da dette er opdateringsintervallet fra både Kystdirektoratet og Dansk Meteorologisk Instituts webservices. Men 15 sekunder er valgt, for hurtigt at kunne se en opdatering på klienten. Hver livslineje er i realiteten en applikation, og derfor eksisterer objektet både før og efter løkken. Alle kald mellem applikationerne er asynkrone på en gRPC message channel. Den sidste, `SendAlertToClients(alert)`, foregår dog som almindeligt asynkront metode kald i SignalR applikationen.

Efter `StartWorkflow` kaldet, kaldes `DownloadWith` metoden på `DownloadData-Service`n. Her foretages et asynkront kald til den eksterne REST webservice hos Kystdirektoratet, hvor parameteren `kdUri` angiver endpoint på websvicens. En csv fil returneres med dagens vandstandsmålinger. Her vises et uddrag:

```
Datotid, Vandstand, Source_ident , North , East
2021-02-01 00:00:00,-63,,6132670,479450
2021-02-01 00:10:00,-64,,6132670,479450
2021-02-01 00:20:00,-64,,6132670,479450
2021-02-01 00:30:00,-60,,6132670,479450
```

Figur 14: CSV fil fra Kystdirektoratet med vandstandsmålinger ved Ribe havn.

Da det kun er vandstanden der er interessant, skal denne besked filtreres for noget af dets indhold, altså et content filter. Til dette formål er en `FilterService` microservice konstrueret med en klasse `TextFilterService`, der eksponerer en metode `FilterCsvColumns(keepColumns)`, hvor `keepColumns` angiver de kolonner der ønskes bibeholdt. Kaldet sker fra `FloodingAlertWorkflowService`, og der ønskes kun at den anden kolonne beholdes. Dette giver følgende resultat:

```
-63
-64
-64
-60
-55
```

Figur 15: Filtreret CSV fil.

Dernæst hentes vindhastighed fra DMI's REST webservice, igen ved `DownloadData` servicens `DownloadWith` metode. Dette endpoint returnerer følgende JSON for den seneste måling af vindhastigheden, målt i Vester Vedsted, ikke langt fra Ribe:


```

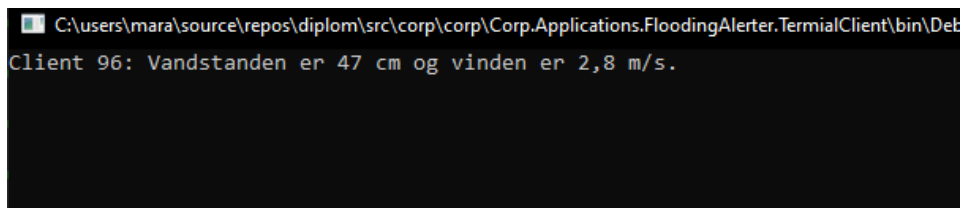
1 {
2   "_id": "33a75fe5-d1fc-c6f5-e9eb-f0a688ce66aa",
3   "parameterId": "wind_speed",
4   "stationId": "06093",
5   "timeCreated": 1.612204167683119E15,
6   "timeObserved": 1.6122042E15,
7   "value": 2.0
8 }

```

Figur 16: Returneret JSON svar fra DMI's webservice.

JSON svaret deserialiseres med Newtonsoft og kun "value" og værdien hertil beholdes, da dette er vindhastigheden i meter per sekund.

Nu er både vandstanden og vindhastigheden hentet fra eksterne systemer, og data aggregeres nu i én besked i typen `FloodingAlertWorkflowResponse`, der returneres fra `FloodingAlertWorkflowService` til Worker instansen i Hub'en. Worker instansen kalder metoden `SendAlertToClients` på hub instansen, hvorved SignalR forbundne klienter dernæst notificeres. Klienten viser den modtagne besked således:



The screenshot shows a terminal window with the following text:

```

C:\users\mara\source\repos\diplom\src\corp\corp\Corp.Applications.FloodingAlerter.TermialClient\bin\DeB
Client 96: Vandstanden er 47 cm og vinden er 2,8 m/s.

```

Figur 17: Klient til SignalR som konsol applikation, der modtager en push notifikation fra Hub'en.

`FloodingAlertWorkflowResponse` der indeholder den aggregerede besked ses her:

```

1 using System;
2 using System.Runtime.Serialization;
3
4 namespace Corp.Services.DataContracts
5 {
6     [DataContract]
7     public class FloodingAlertWorkflowResponse
8     {
9         [DataMember(Order = 1)]
10        public int WaterLevel { get; set; } // cm
11
12        [DataMember(Order = 2)]
13        public string WindSpeed { get; set; }
14
15        [DataMember(Order = 3)]
16        public DateTime Time { get; set; }
17
18        [DataMember(Order = 4)]
19        public string MessageInfo { get; set; }
20
21        [DataMember(Order = 5)]
22        public string WindDirection { get; set; }
23    }
24 }

```

Figur 18: Resultatet af dataaggregationen som DTO.

5 Konklusion

Den moderne microservice arkitektur giver et solidt fundament for driften og ikke mindst videreudviklingen af systemet. Den nuværende interne integration og messaging mellem microservices, udnytter arkitekturens muligheder for nemt vedligehold af kodebasen og fejlfinding. Herudover er skalering af ny funktionalitet i eksisterende microservices meget nem, både i implementering i Corp.Services.Contracts, samt konfigurationen af applikationsvæerten, hvor der blot skal tilføjes en ekstra service. Integrationen til eksterne systemer er tilsvarende overkommelig, idet nye microservices blot kan tilføjes til systemet efter behov. Dette kan tilsvarende udnyttes i en agil udviklingsproces i systemets livscyklus.

Casens implementering illustrer hvorledes moderne teknologier anvendes i fint samspil med etablerede integrationsmønstre, som message construction, -filter og -router. Moderne standarder for datatransport som HTTP og dataformat som JSON, finder umiddelbar anvendelse med de valgte teknologier, nemlig gRPC og SignalR, begge udviklet af markedsførende virksomheder, hvilket i høj grad sikrer den fremadrettede kompatibilitet.

De forretningsmæssige fordele ved greenfielding med disse teknologier i en microservicearkitektur, er til at få øje på når udviklingstiden kan reduceres markant ved anvendelse af god ansvarsadskillelse i de interne komponenter. Herved gøres også integrationstest og systemtest væsentligt nemmere, da kun få teknologier anvendes til så stor en del af systemets kernefokus.

Samlet set konkluderer jeg, at Erls otte principper for design af en service orienteret arkitektur, kan opfyldes ved anvendelse af en microservice arkitektur og ovennævnte valgte teknologier i samspil.

6 Perspektivering

Den præsenterede løsning er umiddelbart omsættelig til deploy på en cloud platform, og her er det nærliggende at tænke på Microsoft Azure. Dette giver reelle driftsmæssige fordele på infrastrukturelle skalering og monitorering af systemet.

Vi må samtidig overveje om ikke tiden er løbet fra traditionel SOA, når der skal konstrueres nye service orienterede systemer. Flexibiliteten og konfigurerbariteten i en microservice arkitektur, giver en række fordele i både udvikling, drift og vedligehold af systemet. Naturligvis skal valget af denne arkitektur ske under forudsætning af, at det er muligt at greenfelde med de her nævnte teknologier og standarder, eller tilsvarende.

7 Referencer

- de la Torre, Cesar; Wagner, Bill; Rousos, Mike;. (2020). Hentet 22. 01 2021 fra dotnet.microsoft.com: <https://dotnet.microsoft.com/download/e-book/microservices-architecture/pdf>
- Erl, T. (2008). *SOA: Principles of Service Design*. Pearson Education, Inc.
- Fowler, M. (25. 03 2014). Hentet 18. 01 2021 fra martinowler.com: <https://martinfowler.com/articles/microservices.html#MicroservicesAndSoa>
- Woolf, B., & Hohpe, G. (u.d.). *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*.