SOLARI

En universell presentasjonsløsning for flytrafikk

Skrevet av

Ole-Jørgen Andersen

Kandidatnummer 633

Studentnummer 202154

Innhold

Introduksjon

Denne dokumentasjonen er ment som en spesifikasjon for applikasjonen *"Solari"*.

Applikasjonen er i dette tilfellet skrevet i C# med hjelp fra .NET Core, ASP.NET Core, EF Core og WINUI3 med hjelp av Windows Template Studio.

Beskrivelse

Solari er en applikasjon for presentere ankomster og avganger ved en gitt flyplass. Målet med applikasjonen er å gi brukeren mulighet til å se og inspisere informasjon om flyvninger ved en hvilken som helst flyplass. Solari skal altså være en universell løsningen, som kan tas i bruk på alle flyplasser. Dette skal kunne eliminere behovet for totalt skreddersydde applikasjoner. Solari vil bare kreve et lite oversettings lag for å oversette datasettet ved flyplassen til data som applikasjonen kan lese. I en reell situasjon, er det tiltenkt at er en service som benytter applikasjons API for å legge inn, oppdatere og fjerne flyvninger, flyplasser og flyselskaper automatisk.

Applikasjonen skal i utgangspunktet brukes på presentasjons tavler, men vil også ha et brukergrensesnitt som en bruker kan anvende for å se og inspisere ytterligere informasjon om hver flyvning, samt å legge til og fjerne flyvninger.

Forkunnskaper

For å gi en bedre forståelse av prosjektet, er det noe domenekunnskap leseren burde sette seg inn i før man fortsetter. I hovedsak domene relaterte ord og uttrykk.

ICAO kode – Er en unik flyplass eller flyselskap identifikator på tre eller fire bokstaver og tall. Ofte brukt internt i luftfart av f.eks. flygeledere.

IATA kode – Er en flyplass eller flyselskap identifikator på tre eller to bokstaver og tall. Ofte brukt i kommersiell sammenheng, f.eks. sammen med flynummer på billetter og presentasjons tavler. Denne er ikke unik.

Første kjøring

For å åpne og kjøre prosjektet må man først og fremst ha alle de nødvendige avhengighetene. Disse finner du i seksjonen under.

Avhengigheter

Generelt krever prosjektet .NET 5 Visual Studio 2019. Pakkene som kreves er lagt inn som avhengigheter i prosjektfilen, men påse at du også at du har følgende komponenter installert via Visual Studio installatøren.

Komponenter:

- .NET desktop development *workload*
- Universal Windows Platform development *workload*
    - Windows 10 SDK 10.0.19041.0
- C++ (v142) Universal Windows Platform Tools *optional workload*
- Data storage and processing *workload*

Viktig informasjon

Følgende er noen viktige momenter for første gang du starter applikasjonen.

Advarsel *CS1591* er dempet da jeg har lagt opp til at dokumentasjonen i domenemodellen legges inn i *"Swagger"* automatisk.

Appen er i utgangspunktet ment for å kjøre i *dark mode*, jeg har satt dette som standard, men det kan endres via innstillinger i applikasjonen.

Siden appen har blitt utviklet med Windows 11, er "Project Reunion" pakken blitt byttet ut med "WindowsAppSDK", pga. kompatibilitet.

Påse at du er koblet til HiØ sitt VPN nettverk, slik at data aksess laget får tilgang til *"Donau*" databasen.

I noen tilfeller, hender det at appen ikke starter på første forsøk, noe relatert til en dll fil som mangler, men dette tror jeg har med noen midlertidige bygg-filer å gjøre. Dette løses enkelt ved å prøve å kjøre appen en gang til.

Starte applikasjonen

Når du er klar, kan du åpne prosjektfilen *"Solari.sln"* i Visual Studio 2019. Etter alt har lastet ferdig, må du starte Solari.Data.Api og Solari.Data.App (Package) (ofte i x86).

Lykke til!

Feil og mangler

For å redusere antall forespørsler til APIet, bestemte jeg meg for å sende med sub objekter i responsene, slik at jeg ikke måtte spørre om disse senere i egne forespørsler. Dette skapte dessverre rundganger i dataene, og derfor feilmeldinger. Jeg løste dette for det meste ved å ta i bruk *"referert"* JSON, men feilene gjenoppsto i mekanismen for å oppdatere data. Hvor jeg endte opp med å fjerne sub objektene før jeg sendte de oppdaterte verdiene tilbake.

Men, jeg tror en bedre løsning på dette ville være å ha egne request modeller (eller DTOs) for flyvninger, flyplasser og flyselskap, som ikke inkluderte disse sub objektene, da dette ville være mer oversiktlig og dermed enklere å vedlikeholde.

Noe som er ganske tilsynelatende er at applikasjonen mangler *"spinnere"* eller progresjon barer (loader) for asynkrone operasjoner. Dette, samt det om modellene ovenfor, er noe jeg ønsket å implementere, men har ikke fått gjort grunnet en klønete livssituasjon.

Om jeg skulle implementert en loader, ville jeg nok ha tatt utgangspunkt i *"Dialog"* grensesnittet mitt, og utvidet det med en metode for sjule dialogen. Deretter ville jeg ha laget en egen klasse på bakgrunn av grensesnittet, med en WINUI loader. Slik så kan jeg åpne og skjule dialogen før og etter et *await* kall, men fremdeles ha fleksibiliteten til å lage en mock for testing eller annen dialog på bakgrunn av det samme grensesnittet.

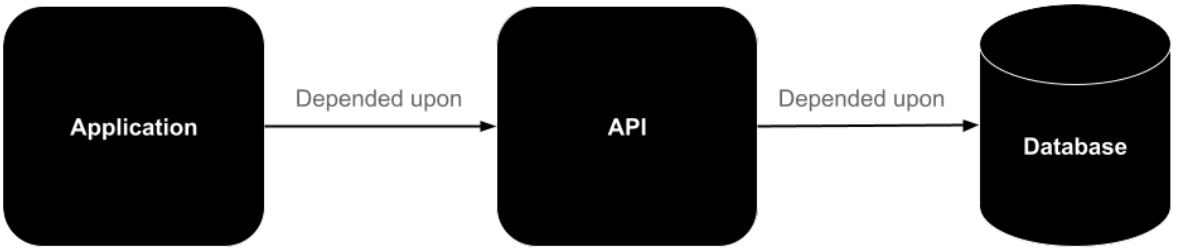
Videre

Hvis du ønsker å lese videre er det to kapitler til, kanskje med litt mindre relevant informasjon. Det første er en kravliste, som ble brukt for å definere funksjonaliteten. Det andre beskriver noen sentrale besluttninger, som ble tatt under utvikling.

Krav

Dette kapittelet tar for seg kravene satt til applikasjonen. Kravene er flytende, som vil si at krav kan endres, fjernes eller legges til under utvikling.

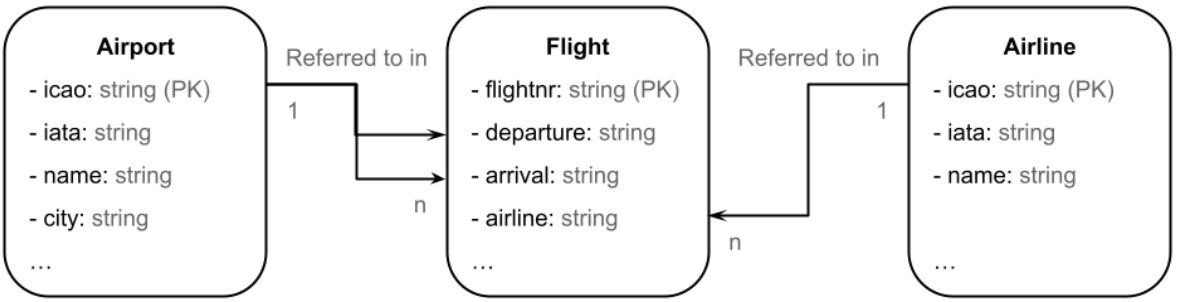
Krav til hele systemet



Figur 1: Overordnet oversikt over det tiltenkte systemet

1. **Systemet skal følge retningslinjer for industriell programmering**
2. **Systemet skal være delt i 3 uavhengige lag**
   - 2.1. Et datamodell lag, i form av en database
   - 2.2. Et datatilgang lag, i form av et api
   - 2.3. Et brukergrensesnitt lag, i form av en applikasjon

Krav til datamodell



Figur 2: Overordnet oversikt over den tiltenkte databasen

** (...) = Se kravlisten for en fullstendig oversikt over data*

1. **Databasen skal modellere en flyvning**
   - 1.1. En flyvning skal ha ett flynummer (unik string) (PK)
   - 1.2. En flyvning skal ha én avgangs flyplass (airport icao string) (FK)

1.3. En flyvning skal ha én ankomst flyplass (airport icao string) (FK)

1.4. En flyvning skal ha ett flyselskap (airline icao string) (FK)

1.5. En flyvning skal ha én avgangstid (datetime string)

1.6. En flyvning skal ha én ankomst tid (datetime string)

1.7. En flyvning kan ha én avgangs gate (string)

1.8. En flyvning kan ha ett bagasjebånd (string)

1.9. En flyvning skal ha én status (string)

1.9.1. Status skal være en egendefinert og informerende tekst

Eks.: *"Boarding"*, *"Gate closed"* eller *"New time 14:00"*

2. Databasen skal modellere en flyplass

2.1. En flyplass skal ha én ICAO-kode (unik string)

2.2. En flyplass skal ha én IATA-kode (string)

2.3. En flyplass skal ha ett navn (string)

2.4. En flyplass skal ha én by (string)

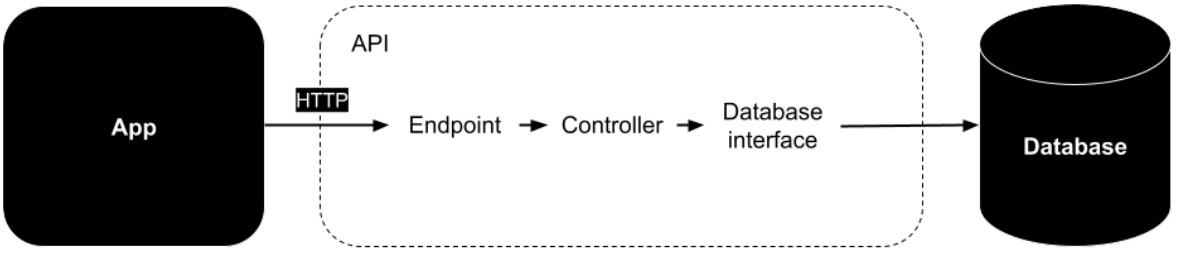
3. Databasen skal modellere et flyselskap

3.1. Et flyselskap skal ha én ICAO-kode (unik string)

3.2. Et flyselskap skal ha én IATA-kode (string)

3.3. Et flyselskap skal ha ett navn (string)

Krav til datatilgang



Figur 3: Overordnet oversikt over det tiltenkte APIet i form av et REST-API

1. APIet skal inkludere operasjoner mot datamodellene

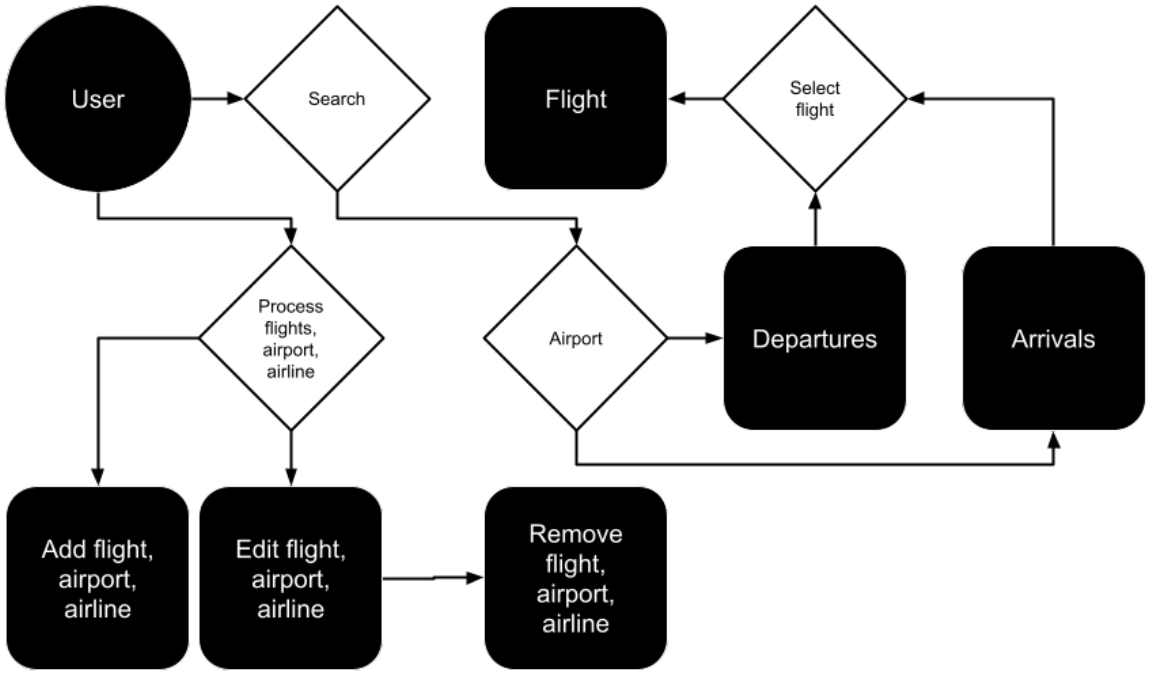
1.1. APIet skal inkludere flyplasser

1.2. APIet skal inkludere flyvninger

1.3. APIet skal inkludere flyselskaper

2. **APIet skal følge CRUD standarden for operasjoner mot hver datamodell som APIet eksponerer**

Krav til brukergrensesnitt



Figur 4: Overordnet oversikt over hoved flyten i det tiltenkte brukergrensesnittet

1. **Brukeren skal kunne søke etter flyplass**
   - 1.1. Brukeren skal kunne søke på navn
   - 1.2. Brukeren skal kunne søke på by
   - 1.3. Brukeren skal kunne søke på ICAO-kode
   - 1.4. Brukeren skal kunne søke på IATA-kode

2. **Brukeren skal kunne se alle flyvninger ved en flyplass i tabell**
   - 2.1. Brukeren skal kunne velge å se avganger eller ankomster
   - 2.2. Brukeren skal kunne velge å se mer informasjon om en flyvning

3. **Brukeren skal kunne behandle flyvninger**
   - 3.1. Brukeren skal kunne legge til en ny flyvning
   - 3.2. Brukeren skal kunne fjerne en eksisterende flyvning
   - 3.3. Brukeren skal kunne redigere en eksisterende flyvning

4. **Brukeren skal kunne behandle flyplasser**

4.1. Brukeren skal kunne legge til en ny flyplass

4.2. Brukeren skal kunne fjerne en eksisterende flyplass

4.3. Brukeren skal kunne redigere en eksisterende flyplass

5. Brukeren skal kunne behandle flyselskap

5.1. Brukeren skal kunne legge til et nytt flyselskap

5.2. Brukeren skal kunne fjerne et eksisterende flyselskap

5.3. Brukeren skal kunne redigere et eksisterende flyselskap

Utvikling

Dette kapittelet er et bonus kapittel som fremhever noen besluttninger tatt under utvikling i forhold til designet og oppbygging av applikasjonen.

Lagdelt arkitektur

Likt kravlisten, endte jeg opp med å separere appen opp i tre hoved lag, bestående av flere deler. Tanken bak dette er at det gir en naturlig oppdeling av ansvarsområder i det som fort kan bli et komplisert prosjekt. Lagene er avhengig av hverandre, men bare en vei. F.eks. er selve WINUI applikasjonen avhengig av data fra APIet, APIet er avhengig data aksess laget, som får sine data fra databasen. Alle piler peker altså innover mot domenet (business logikken) og dataene i databasen.

Det fine med dette er at alle endringer vil ha effekt utover, og siden alt er avhengig av domenet vil endringer her føres naturlig utover. En slik arkitektur vil være lettere å vedlikehold og vil stå tidens tann.

Repository pattern

I data aksess laget, laget jeg et repository grensesnitt og implementerte et eget repo for SQL database med Donau. Tanken bak å lage et slik abstraksjonslag, er først å fremst å ha et service lag. Slik at konkrete implementasjoner mot databasen kan endres enkelt, på ett sted. I tillegg, gir det oss fleksibiliteten til å lage andre repo implementasjoner senere med andre lagringsstrukturer.

Exception handling

For å håndtere feil, endte jeg opp med å konstruere mitt eget lille system, hvor tanken var å gjenbruke mest mulig og skrive DRY kode. Jeg gjorde dette ved å først definere et sett med grunnleggende exceptions i data aksess laget. Disse blir så tatt i bruk i repositoriet og sendt oppover lagene til det når sluttbrukeren sin skjerm. Underveis blir de kastet på nytt med ytterligere beskrivende feilmeldinger.

Når de når applikasjonslaget blir feilmeldingen presentert til brukeren ved hjelp av en en skreddersydd *error* implementasjon av “Dialog” service grensesnittet mitt. Uansett hva som skjer blir brukeren alltid presentert med feilmeldingen før noe data er prosessert av data aksess laget.