28. maj 2019

Gruppe 2
Semesterprojekt 4
Ingeniørhøjskolen Aarhus

# Photobook
# Systemarkitektur

| **Deltagere** | | |
|---|---|---|
| **Navn** | **Initialer** | **Studienummer** |
| Andreas Harfeld Jakobsen | AHJ | 201608930 |
| Oskar Thorin Hansen | OTH | 201700132 |
| Morten Rosenquist | MLR | 201706031 |
| Jonas Hingeberg Hansen | JHH | 201508749 |
| Troels Østergaard Bleicken | TØB | 201705588 |
| Frederik Fuglsang Midtgaard | FFM | 201704982 |
| Nina Nguyen | NN | 201506554 |

# Indhold

# 1 Introduktion

## 1.1 Formål og omfang

Formålet med dokumentet er at gennemgå arkitekturen og designet for Photobook. Der bliver set på alle dele af systeme/t og kan bruge dette dokument til at få et overblik over Photobook. Dokumentet er blevet lavet i forbindelse med I4PRJ4. Derfor forventes det, at læseren er på teknisk niveau som en IKT studerende på 4. semester. Dokumentet er primært skrevet til vejleder og sensor, men kan læses af alle, som er interesseret i Photobook's design.

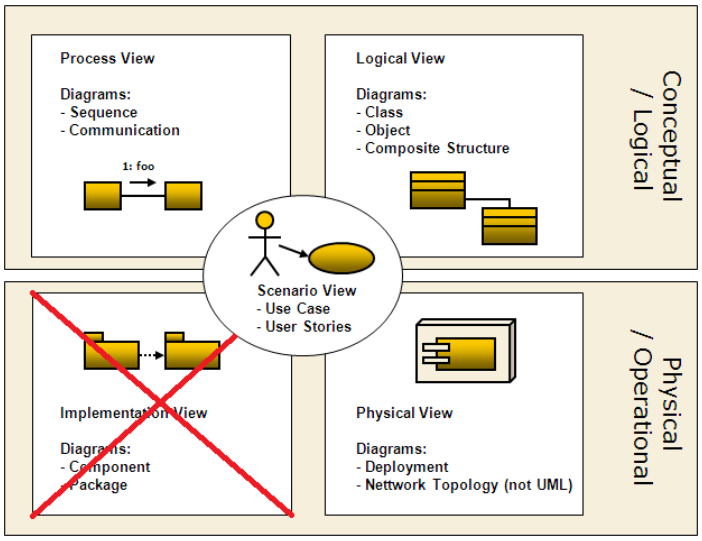
## 1.2 Ordforklaring

| **Forkortelse:** | **Forklaring**: |
|---|---|
| PB | Photobook |
| DAL | Data Access Layer |
| BLL | Business Logic Layer |
| PL | Presentation Layer |
| IF | Interface |
| EFC | Entity Framework Core |
| APP | Application |
| MVVM | Model View View Model |
| CRUD | Create, Read, Update og Delete |
| ORM | Object-Relational Mapping |
| DI | Dependency Injection |
| Mobilapp | Mobilapplikation |
| Webapp | Webapplikation |
| Host | Arrangør |
| Guest | Gæst |
| Event | Begivenhed |

## 1.3 Dokumentstruktur og læsevejledning

Igennem dokument er dele af den arkitekturiske 4+1 model anvendt. Modellen er opbygget ved, at man kan fortolke et scenarie på fire forskellige måder(views). Scenariet bliver i dette projekt beskrevet ved user stories. De fire forskellige måde at illustrere scenariet på gør det muligt for forskellige stakeholders, at se på det view, der omhandler dem. På figur 1 kan modellen ses[1]. Det er dog ikke alle views fra 4+1 der anvendes. Implementering view er udeladt da det ikke menes relevant. Der er også tilføjet et ekstra view til at belyse de databaser der bliver brugt, dette er beskrevet i Data view.

[1] https://wiki.cantara.no/display/dev/4+plus+1+View+Model



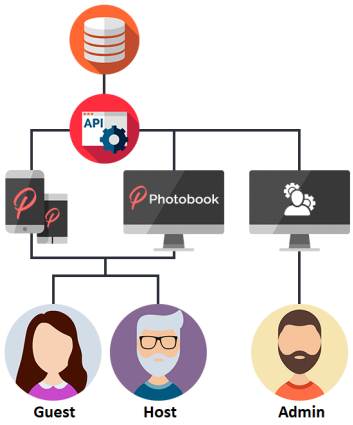

Figur 1: 4+1 modellen med tilhørende UML diagrammer

## 1.4 Dokumentets rolle i en iterativ udviklingsproces

I forbindelse med projektet arbejdes der iterativt ved brug af scrum. Dette dokument følger ligeledes iterationerne igennem forløbet. Derfor er dokumentet meget dynamisk, og vil blive ændret i forbindelse med erfaringer, der opnåes igennem iterationerne.

# 2 Systemoversigt

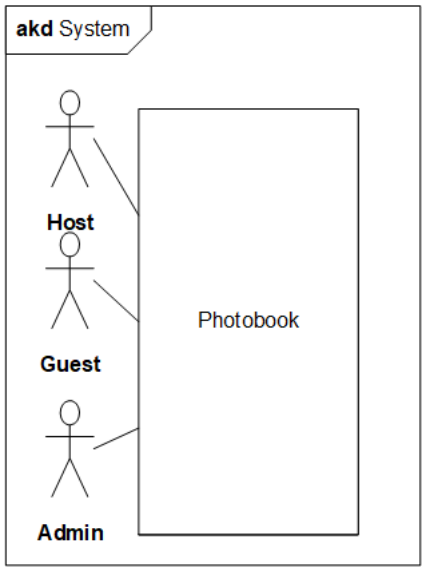
De forskellige dele Photobook består af kan ses på figur 2. Figuren illustrerer en rig skitse af systemet.




Figur 2: Rig skitse der illustrerer systemet Photobook

## 2.1 Systemkontekst

Der kan på figur 3 ses et aktør kontext diagram, der viser hvilke aktøre, der påvirker systemet. De primære aktøre, der tilgår systemet er host, guest og admin.




Figur 3: Aktør kontext diagram for systemet

## 2.2 Systemintroduktion

Målet med systemet er at give en gruppe af mennesker ved et event mulighed for at samle deres billeder et overskueligt sted. Måden det skal fungere på er, at en person i gruppen skal oprette sig som host i

systemet og skal derefter oprette et event. Eventet er tilknyttet en pin som hosten kan give til de resterende personer(guests). Hosten og guests kan derefter se, slette, tilføje billeder som de ønsker. Desuden kan de up-/downloade billeder.

Derudover er der også en admin som har adgang til en administrator side. På denne side vil det være muligt at se alt der er oprettet i databasen, det vil være muligt at hente en log der indeholder samtlige kald til web API'et og det vil være muligt at se en oversigt over alle kald der kan laves til web API'et.

# 3 Systemets Grænseflader

## 3.1 Grænseflader til person aktører

Som beskrevet i tidligere afsnit er der to almindelige person aktøre: host og guest. Der er også en administrator aktør: admin. De tre forskellige aktøre tilgår systemet igennem tre UI's. Brugergrænsefladerne er som det kan ses på den rige skitse en mobil- og webapplikationer. Grunden til at der både er en mobil- og en webapplikation er for at gøre det muligt at uploade billeder til et event på en computer, hvis en guest har taget billeder med et kamera eller andre enheder, der ikke har mulighed for at uploade billeder på en mobilapplikation. Opgaverne de to almindelige aktøre kan udføre er næsten identiske på de to UI's. På den tredje UI, den hvor admin aktøren har adgang der er det muligt at se hvilke actions der bliver tager og hvilket data der er blevet oprettet.

For at en person bliver host skal han registrere sig. Opgaverne hosten kan udføre er:

- Opret/slet event
- Se, slette, uploade og downloade billeder

Før en person bliver en guest skal han logge ind med event pin og oplyse sit navn. Opgaverne guest kan udføre er:

- Se, uploade og downloade billeder

En admin kan udføre følgende opgaver

- Se data der oprettet i databaserne
- Se alle kald der kan laves til web API'et
- Se loggen over alle kald der er blevet lavet til web API'et

Den eneste forskel på, hvad der kan gøres fra mobilapplikationen og webapplikationen er, at der kan tages billeder til et event igennem mobilapplikationen.

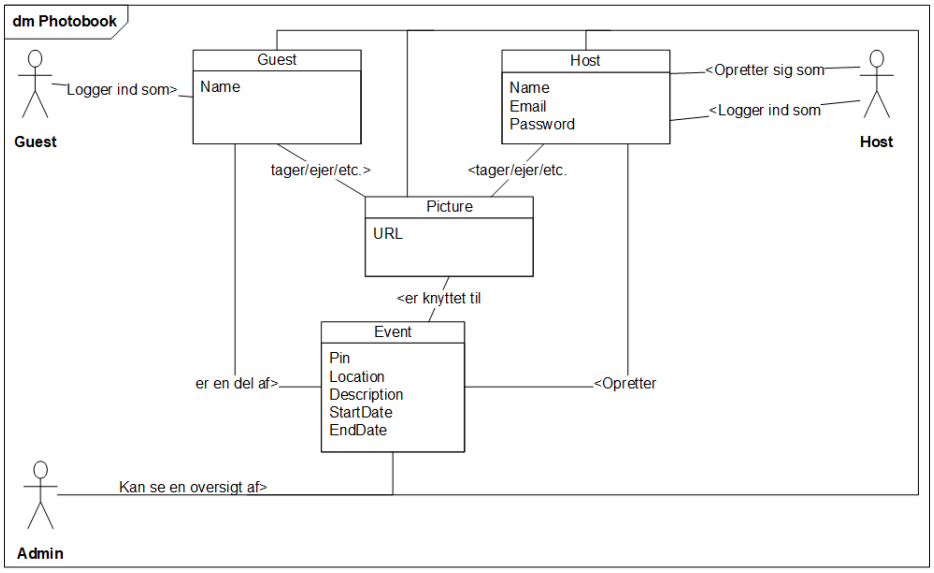
# 4 Use Case View

Se kravspecifikation for Photobook [2]

[2] Bilag/Kravspecifikation.pdf

# 5 Domænet

Der er lavet en domænemodel for systemet. Modellen bruges til at vise hvilke conceptuelle klasser systemet består af. Modellen kan i dette system ikke bruges til at fremvise boundary / controller klasser. Men bruges i stedet til at vise hvilke domæne(model) klasser, der er eksisterende i systemet. Modellen kan ses på figur 4.

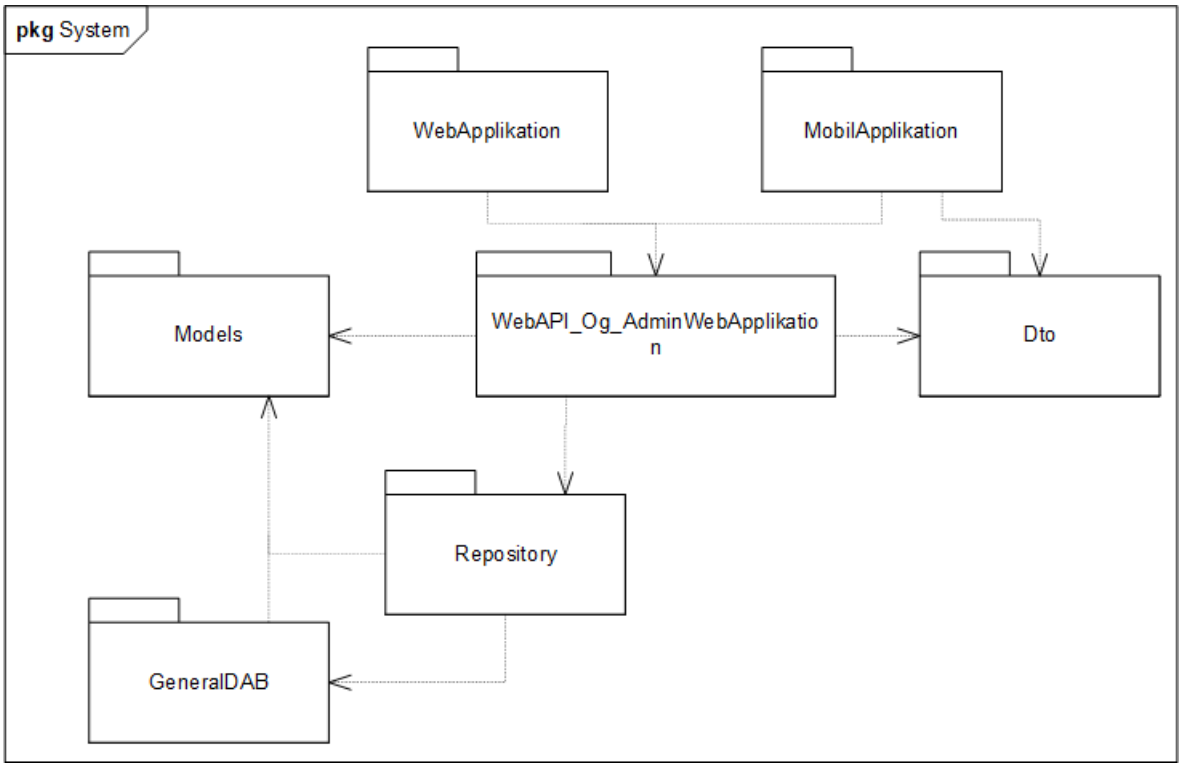



Figur 4: Domænemodel for systemet

# 6 Logisk View

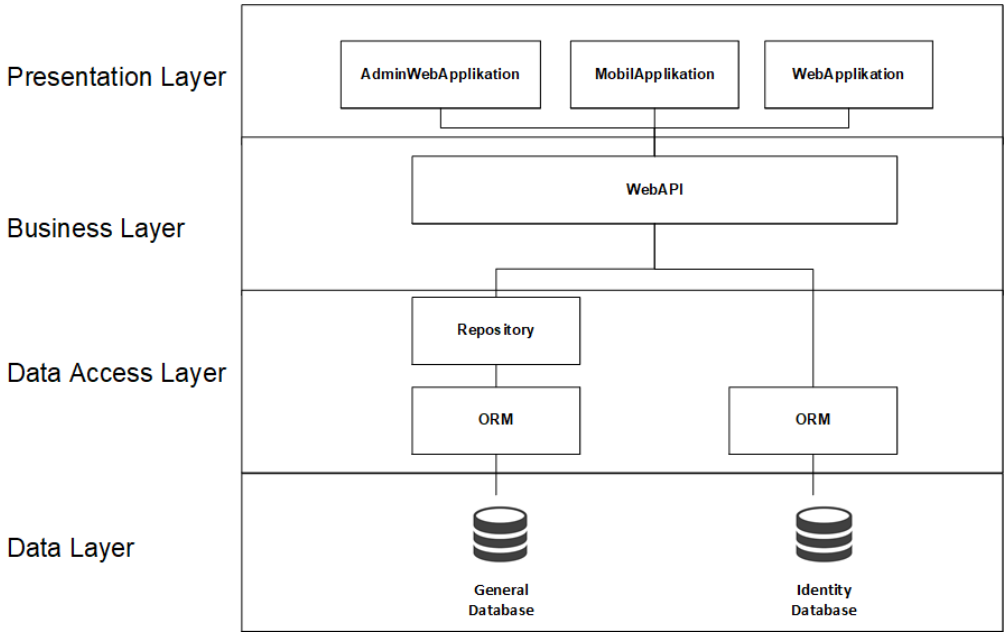
I dette afsnit vil der blive set på det logiske view af systemet. Der ses på hvilke pakker systemet består af. Derefter bliver der set på indholdet af de enkelte pakker. Afsnittet indeholder også beskrivelse af relevante arkitekturiske beslutninger.

## 6.1 Oversigt




Figur 5: Pakkediagram for systemet

Der er valgt at bruge et N-tier arkitektur mønster til systemet. De fire lag er Data Layer, Data Access Layer, Business Layer og Presentation layer. På figur 6 kan opdelingen ses. Denne arkitektur er en client-server arkitektur, hvor hvert ansvarsområde i systemet er separeret fra hinanden. Denne arkitektur medfører også at business laget sikrer at det kun er gyldig data der indsættes i databasen. Derudover har en client heller ikke direkte adgang til databasen, hvilket øger sikkerheden for systemet.

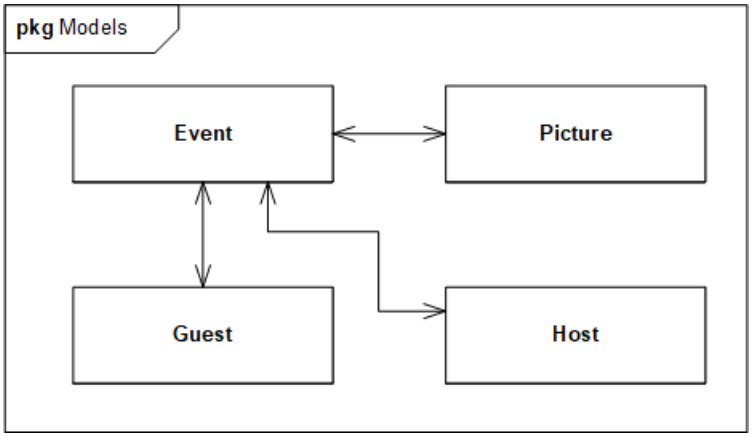



Figur 6: Systemet ud fra et layered mønster

## 6.2 Arkitektursignifikante designpakker

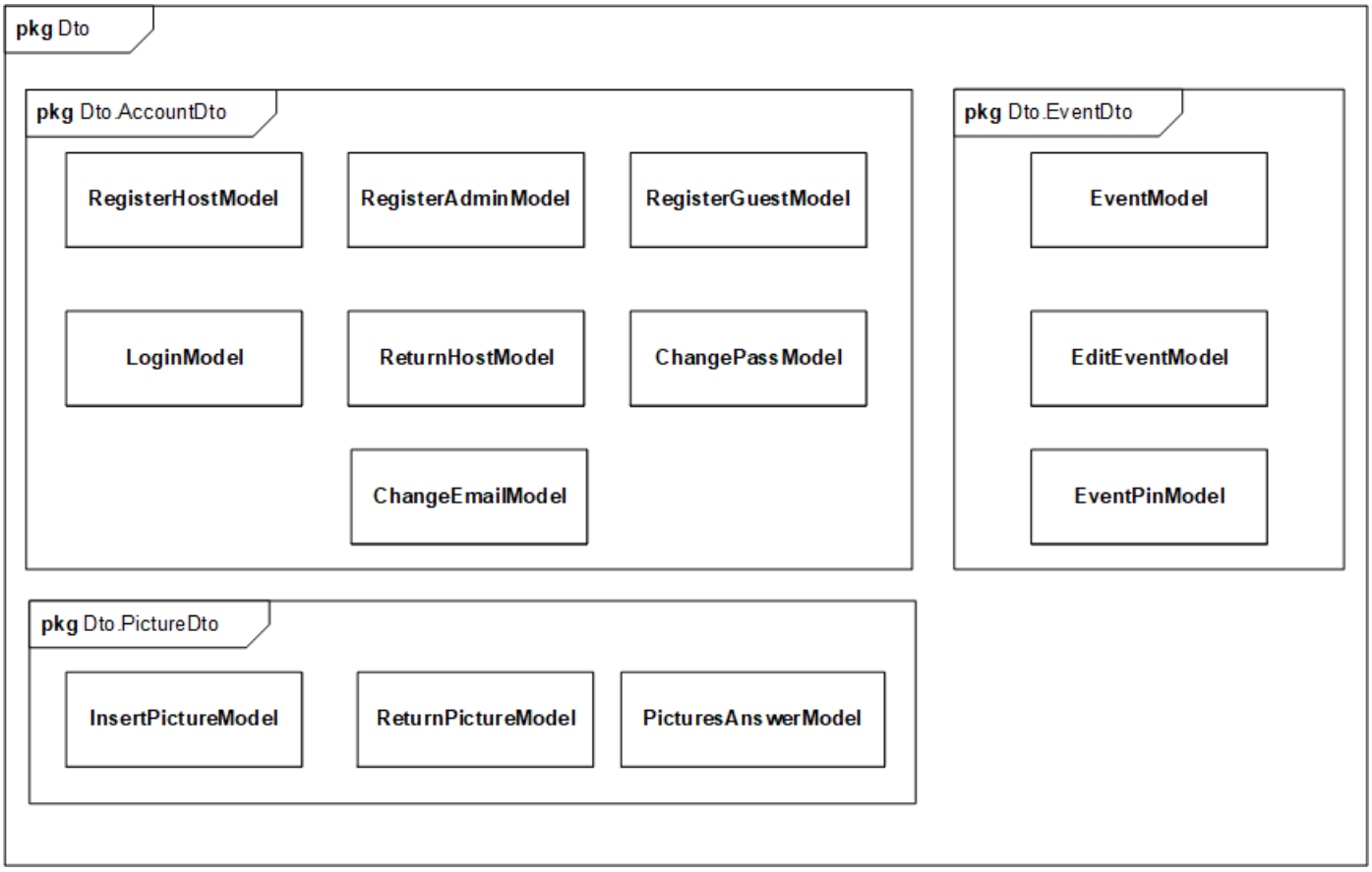
### 6.2.1 Models

Denne pakke indeholder modelklasserne i systemet. Klasserne i pakken er alle value types. Pakken kan ses på figur 7.




Figur 7: Pakke diagram for Models

### 6.2.2 Dto



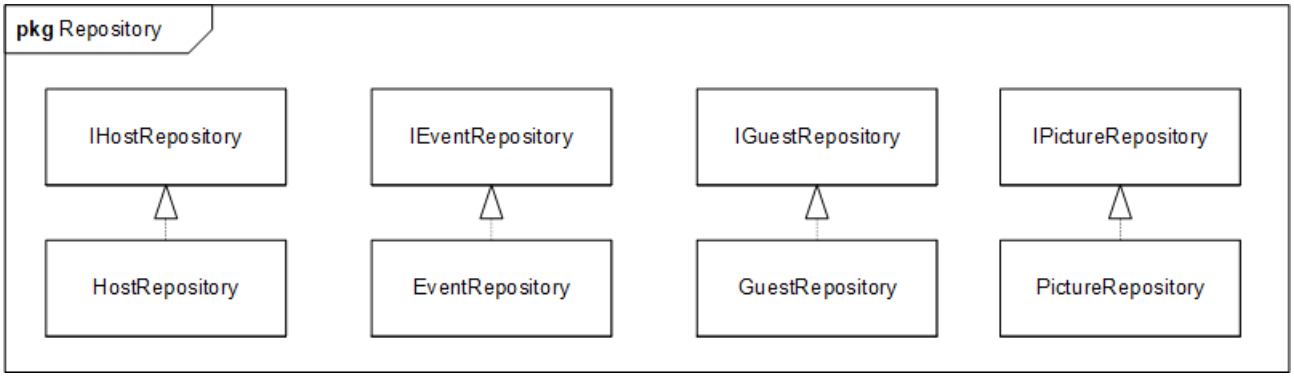

Figur 8: Pakke diagram til Dto

### 6.2.3 GeneralDAB

Denne pakke indeholder det nødvendige til oprettelsen af den generelle database. Det laves ved brug af EF core code first. Det vil sige pakken indeholder context klassen med tilhørende migrations. Der kan læses mere om databasen under Data View.

### 6.2.4 DAL

Denne pakke indeholder adgangen til databasen. Det bliver lavet ved brug af repository mønstret. Det laves ikke generisk, så der er et repository pr. entitet. Indholdet i pakken kan ses på figur 9.



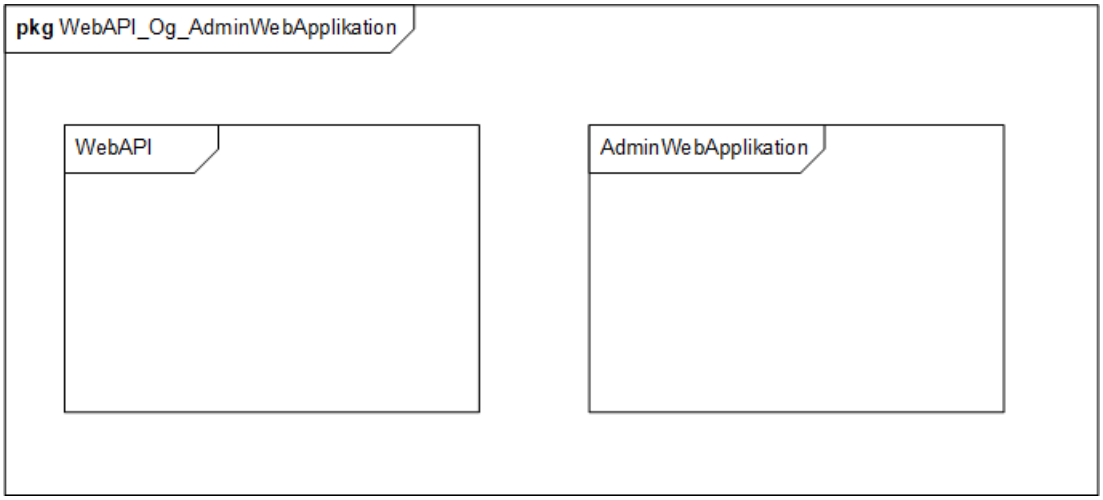

Figur 9: Pakke diagram for PB.DAL

### 6.2.5 IdentityDAB

IdentityDAB er lavet ved brug af Identity Framework. Dette gøres ved at have en DBContext som arver fra IdentityDbContext og en AppUser som arver fra IdentityUser. Den eneste ting der varierer fra den default user der arves fra Identity framework er at der er tilføjet en Name attribut. Der kan læses mere om denne database i Data view.

### 6.2.6 WebAPI_Og_Adminwebapplikation

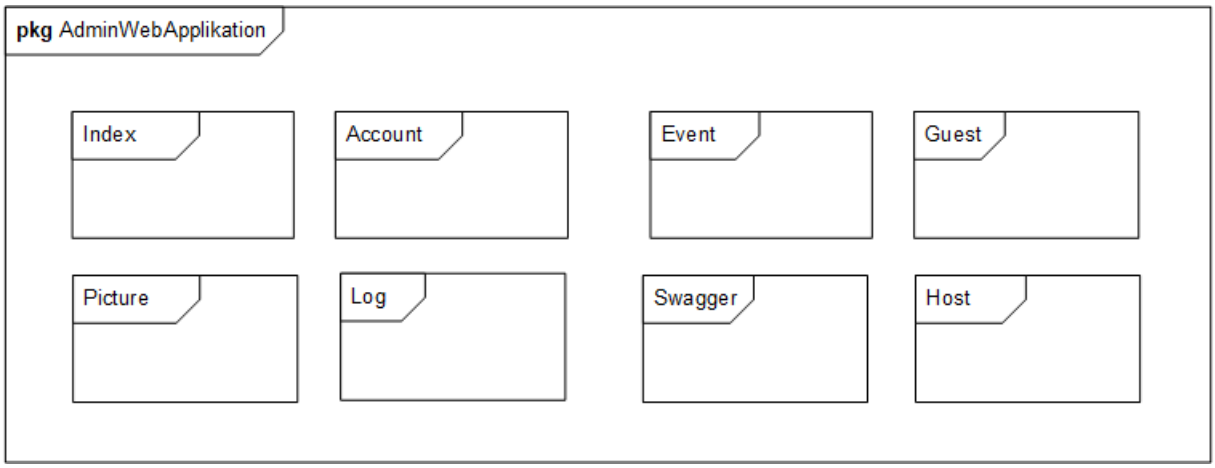
I denne pakke er der to under pakker. Den ene pakke er en Admin side til at se hvilket data der er i databasen denne er kaldet Adminwebapplikation. Den anden pakke er selve web API'et som består af de controllers der kan sendes http requests til.




Figur 10: Pakke diagram for WebAPI_Og_Adminwebapplikation

På figur 11 kan under pakken Adminwebapplikation ses. Denne pakke indeholder 8 pakker som hver representerer et view. Dette er hovesageligt styret med MVC pattern. Undtagelserne er Index og Swagger. Index er bare en statisk side, det er denne side der er hovedsiden og det er her administratoren skal logge ind. Swagger

er sat op ved brug af Swashbuckle UI [3].

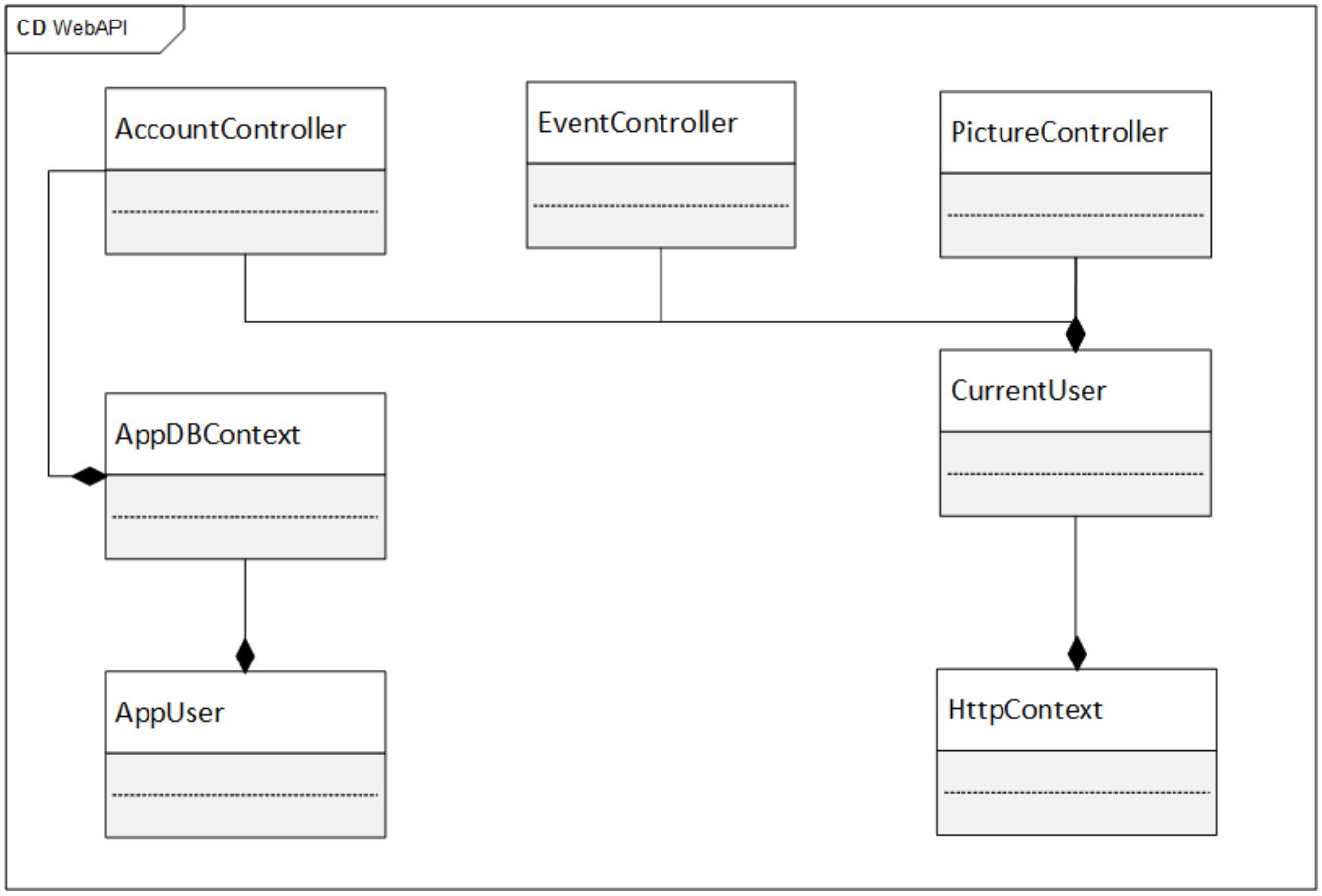



Figur 11: Pakke diagram for Adminwebapplikation i WebAPI_Og_Adminwebapplikation

På figur 12 kan klasserne i WebAPI delen ses. Der er 3 controllers som alle gør brug af en HttpContext til at få information om hvilken bruger der er logget ind. For at undgå at skabe en dependency direkte til sådan en stor og kompliceret klasse som HttpContext gøres der brug af en wrapper klasse "CurrentUser". Dette gør at der kan laves alle de kald der er brug for til HttpContext gennem wrapper klassen hvilket gør designet meget mere testbart.

AccountController er den controller som styrer alt oprettelse af brugere samt login osv. Derfor tilgår denne controller DbContexten.
Alle kald der kan laves til web API'et kan ses i swagger ui på https://Photobookwebapi1.azurewebsites.net/swagger/index.html

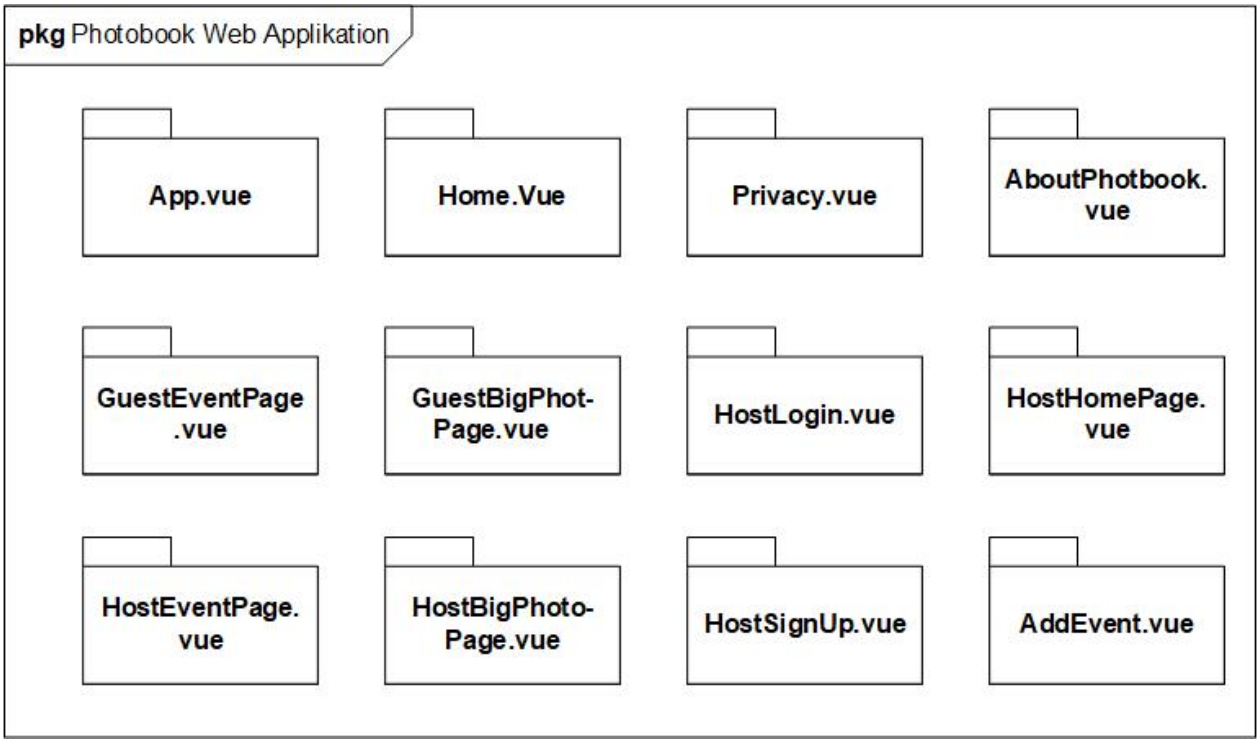
[3]https://docs.microsoft.com/en-us/aspnet/core/tutorials/getting-started-with-swashbuckle?view=aspnetcore-2.2tabs=visual-studio




Figur 12: Klasse diagram for WebAPI

### 6.2.7 Webapplikation

På figur 13 ses et overblik over indholdet af software pakken til webapplikation. Pakken består af en lang række underpakker, hvor hver pakke tilsvarer et Vue komponent. Hver pakke tilsvarer en side i webapplikationen, og indeholder derfor samlet hele webapplikationens funktionalitet. Hver pakke indeholder et stykke html kode og et tilhørende Vue objekt med Javascript kode.




Figur 13: Pakkediagram for Photobook webapplikation

**App.Vue**
Denne pakke er webapplikationens grundpakke. Denne pakke indeholder en navigationsbar, samt det element, hvor alle andre sider renderes i.

**AddEvent.Vue**
Pakken gør det muligt for Host at tilføje et nye events.

**Home.Vue**
Dette er en første pakke som brugeren møder, og giver brugeren mulighed for at logge ind som guest, eller clicke videre til host login.

**Privacy.Vue and AboutPhotobook.Vue**
Disse to pakker indeholder tekst beskrivelser som Photobook, med information til brugeren.

**GuestEventPage.Vue**
Denne pakke indeholder siden som en guest først møder. Her kan alle billeder fra et event ses, og der kan tilføjes og og downloades billeder.

**GuestBigPhotoPage.Vue**
Denne pakke indeholder siden som fremkommer når der trykkes på et billede. Den viser billedet i fuld størrelse, med mulighed for at bladre igennem billeder i fuld opløsning.

**HostLogin.Vue**
Pakke som sørger for at en host kan logge ind i Photobook.

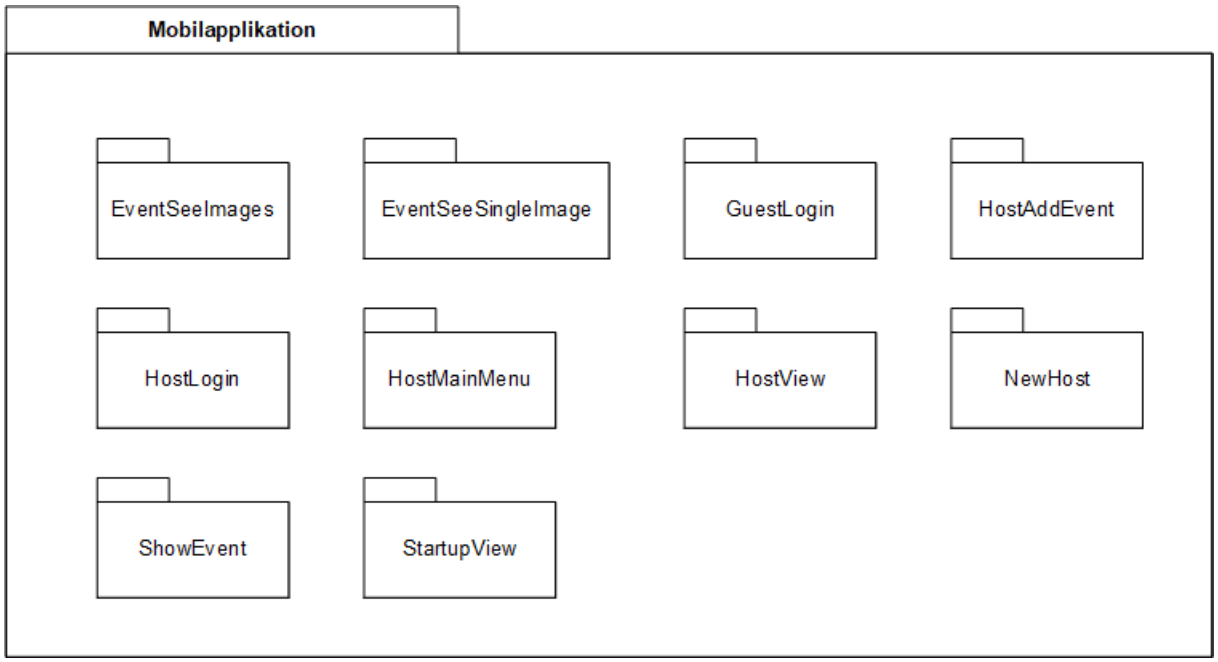
**HostSignUp.Vue**
Denne pakke gør det muligt at oprette sig som Host, således at man kan tilgå HostHomePage.

**HostHomePage.Vue**
Pakken indeholder en side med oversigt over de events som en host har oprettet, med mulighed for at oprette nye events.

**HostEventPage.Vue**
Denne pakke er ligesom GuestEventPage, blot med mulighed for at slette eventet.

**HostBigPhotoPage.Vue**
Denne pakke er ligesom GuestBigPhotPage, blot med mulighed for at slette billedet.

### 6.2.8 Mobilapplikation




Figur 14: Pakkediagram for MobilApplikation, der viser de pakker denne pakke består af

Pakken "Mobilapplikation"består af flere underpakker, der til sammen udgør hele mobilapplikationen. Hver underpakke, der er illustreret på figur 14, består af et view, en viewmodel og de tilhørende models viewmodellen bruger i henhold til MVVM arkitekturen, der forklares i et senere afsnit. Dvs. at hver underpakke består af nogle klasser og en xaml fil, der til sammen udgør noget funktionalitet og et view i mobilapplikationen.

**StartupView**
Denne pakke sørger for at vise det første man ser når man åbner appen. Dvs. at den sørger for at vise startskærmen og giver en bruger mulighed for at logge ind som enten host eller guest.

**HostView**
Funktionaliteten i denne pakke giver en bruger mulighed for at logge ind (hvis brugeren allerede er registreret) eller oprette en ny bruger.

**GuestLogin**
Denne pakke giver brugeren mulighed for at logge ind som guest, ved at indtaste sit fulde navn og PIN koden til det event, brugeren ønsker at tilgå.

**ShowEvent**
Denne pakke sørger for at vise et event og de tilhørende informationer om et givet event, som enten en guest eller en host kan tilgå. Her er det muligt at se alle billeder, at uploade billeder eller at tage et billede til det givne event.

**EventSeeImages**
Pakken EventSeeImages gør det muligt at se alle de billeder der er taget/uploadet til et givet event. Denne pakke gør det også muligt at downloade alle billeder, eller at trykke på et billede for at se dette i et større format.

**EventSeeSingleImage**
Denne pakke viser det billede, der er trykket på i EventSeeImages pakken. Billedet vises i et stort format, og der er mulighed for at trykke på "Download"eller "Slet".

**NewHost**
Denne pakke giver mulighed for at tilføje en ny host til mobilapplikationen. Her skal brugeren angive sit fulde navn, kodeord og email. Når dette er udfyldt, registreres brugeren som en host i systemet.

**HostLogin**
Denne pakke gør det muligt at logge ind som en allerede eksisterende host. For at logge ind skal brugeren angive email og kodeord.

**HostMainMenu**
Denne pakke giver en host mulighed for at trykke på en knap 'opret event' og for at tilgå de events, som brugeren har oprettet (hvis der er oprettet nogle).

**HostAddEvent**
Denne pakke giver en host mulighed for at oprette et nyt event, og skrive detaljerne til det givne event. Dvs. at en host opretter en beskrivelse, navn og tidspunkt for eventet.

## 6.3 User story realiseringer

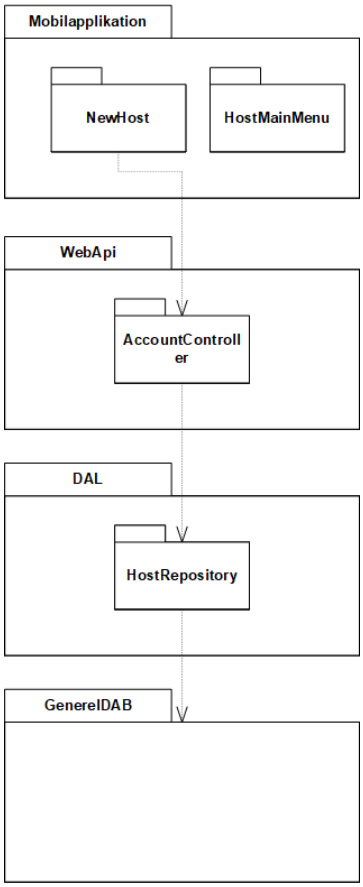
Til dokumentation af vores user stories realiseringer er der lavet sekvensdiagrammer og et tilhørende klassediagram til hver user story.

Nogle af de definerede user stories ligner hinanden funktionalitetsmæssigt, og derfor undlades de.
Det drejer sig om de user stories hvor der blot foretages et kald fra en af brugergrænsefladerne (webapp el. mobilapp) til web API'et videre til data access lag og derefter database laget. Disse kald foretages hyppigt i vores user stories, og derfor vises der blot ét eksempel på dette.

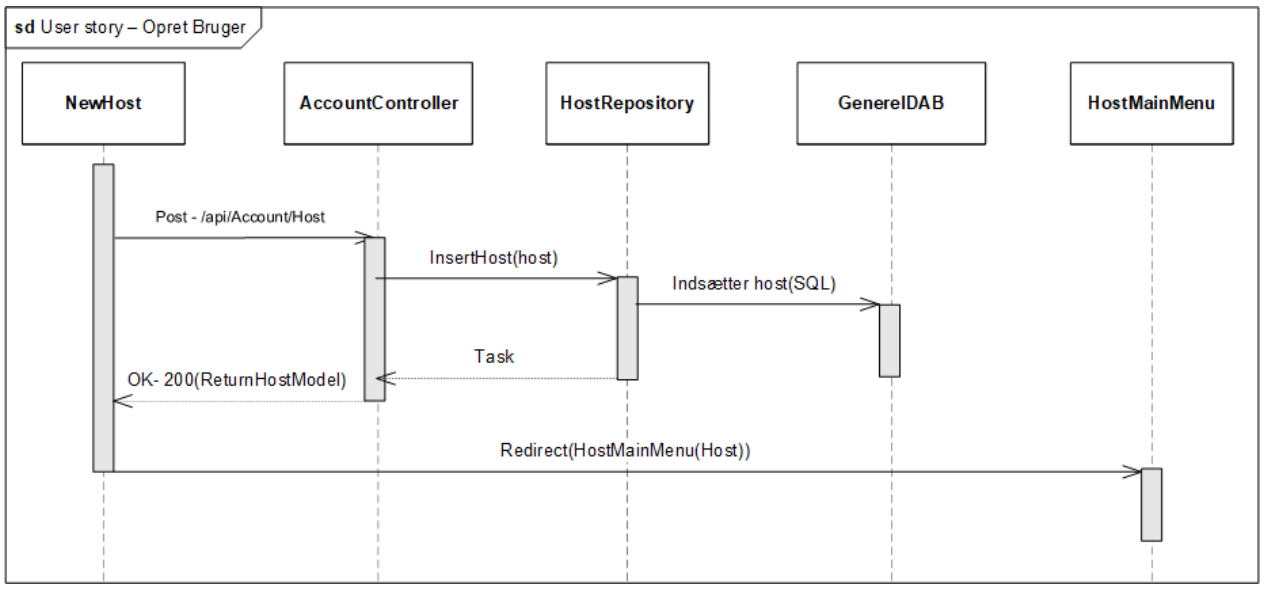
Det er valgt at vise et overblik af to af vores user stories, hvor der vises kommunikationen mellem de forskellige pakker. Dette gøres for at få indsigt i de forskellige processer der sættes i gang, ved en given user story.
Dette gøres kun for 2 user stories, da disses realisering i forhold til software arkitekturen, er meget dækkende for resten af systemets. Flere diagrammer vil derfor ligne disse to meget. Det er ligeledes lavet sådan at en user story realisering tager udgangspunkt i mobil app, og en tager udgangspunkt i Webapp, hvorved det gerne skulle fremgå at systemets agering er den sammen ligegyldigt hvorvidt kaldet kommer fra Webappen eller Mobilappen.

### 6.3.1 User story - Opret bruger

**Fra mobilapp**



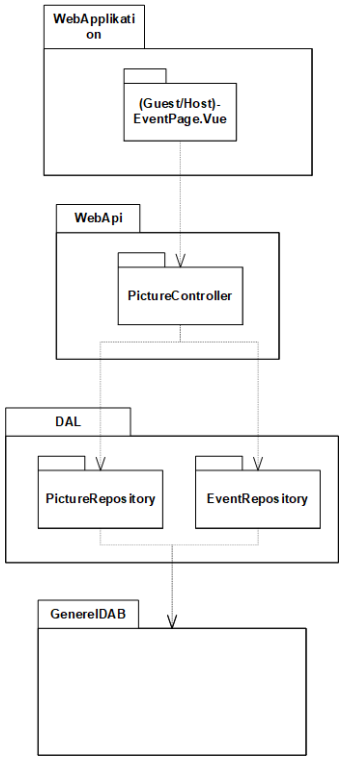

Figur 15: Realisering af user story 'opret bruger' - Klassediagram




Figur 16: Realisering af user story 'opret bruger' - Sekvensdiagram

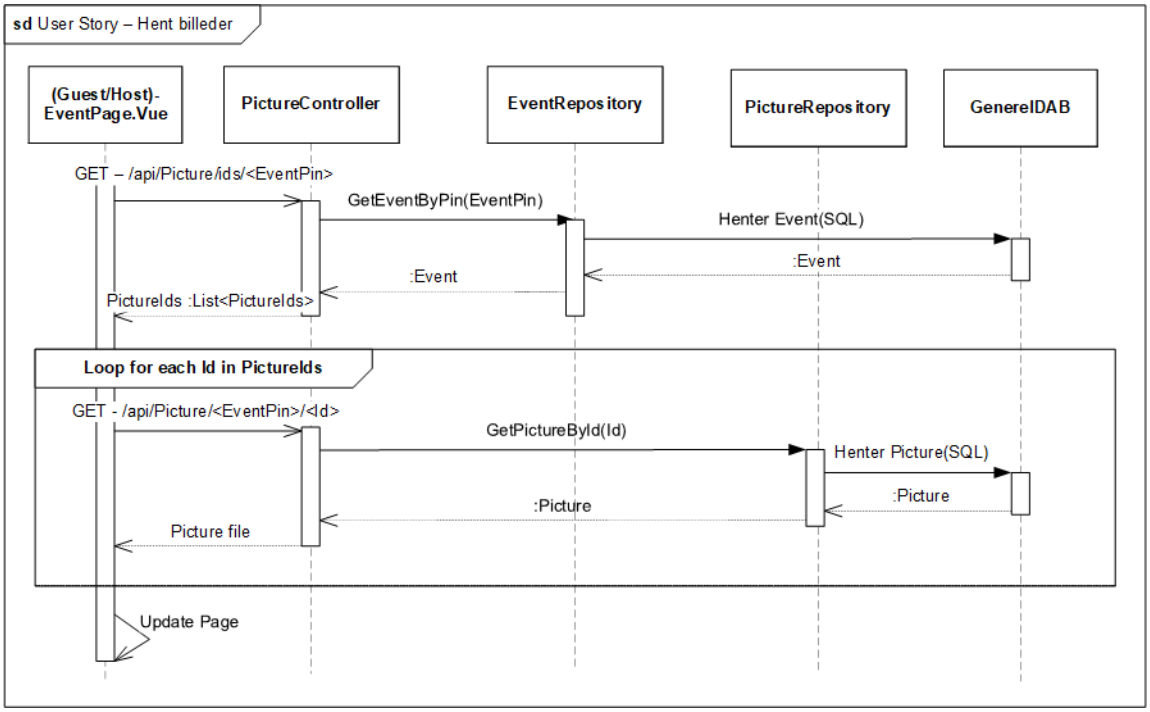
### 6.3.2 User story - Se billeder

**WebApp**




Figur 17: Pakke diagram med dependecies for User story 'se billeder - se billeder

På figur 17 ses en pakkeafhængighedsdiagram for user story 9, med udgangspunkt i webapplikationen. Det ses tydeligt at systemets afhængigheder bevæger sig fra front end og ned i mod backend.




Figur 18: Sekvens Diagram for user story 'se billeder'

På sekvensdiagrammet på figur 18 ses en oversigt over systemets kommunikation når der hentes billeder fra database til frontend. Først hentes en liste over Ids, hvorefter hvert enkelt billede hentes individuelt som en fil.

# 7 Proces/Task View

## 7.1 Oversigt over processer/task

I næsten alle underdele af projektet benyttes internt multi processering og tasks. Disse styres internt i de enkelte software pakker, hvor kommunikationen mellem dem også styres. Desuden kører ingen underdele af projektet på samme tråd, men på forskellige tråde på forskellige maskiner, som kommunikerer sammen over ved hjælp af kald til hinanden.
I dette afsnit følger en uddybning af brugen af processer og tasks i systemet.

## 7.2 Proces/task i Web APi

Til de fleste actions i WebApi'et anvendes der Tasks. Dette gøres for at få systemet til at være mere responsivt når der er flere brugere der laver kald til WebApi'et på samme tid. De fleste actions er async og anvender await i alle kald til DAL. Når der anvendes tasks gør det, at selvom et kald kan tage lang tid at returnere så blokerer det ikke for andre brugere.

## 7.3 Proces/task i repository

Alle metoder WebAPI'et kan tilgå igennem repository interfaces er implementeret med async/tasks og await. Ved alle kald til databasen ved brug af EFC, er der brugt den async version, som bliver await'et.

## 7.4 Proces/task i Web Applikation

### 7.4.1 Proces/task implementering

I systemets Web Applikation benyttes javascript til at gøre hjemmesiden interaktiv. Javascript kører som udgangspunkt kun enkelttrådet, og den basale brug af hjemmesiden kører derfor på en enkelt tråd. Ved kommunikation med Web API benyttes der dog fetch kald, som kan køre parallelt. Dette sker eks. ved udførsel af user story 9 - hvor der hentes billeder fra hjemme siden i en for løkke. Disse fetch kald vil kører uafhængigt af hinanden, og billederne vil derfor hentes i tilfældig række følge, efter hvilket fetch kald som returnere først.

### 7.4.2 Proces/task kommunikation og synkronisering

For at styrer fetch kaldene benyttes promises og async/await.

#### 7.4.2.1 Promises

Promises benyttes i javascript til at håndtere asynkrone kald. Med promises kan asynkrone kald kobles sammen med en resolve eller reject funktion, alt efter om det asynkrone kald fejler eller ej. Herved kan der ventes på at en funktion færdiggøres før der arbejdes videre. Dette gøres i praksis ved at benytte et .then() kald efter et promise.

Et fetch kald returnere som standard et promise, og der kan derfor benyttes et .then() kald til at reagere når fetch kaldet returnere. Dette benyttes til at arbejde på det data som fetch kaldene til web Api returnere,

og til at kalde funktioner, som skal arbejde videre på returneret data, som først må kaldes når fetch kaldet returnere.

#### 7.4.2.2 Async/Await

async await er en javascript syntaks som kan benyttes til at arbejde med promises. En async funktion returnere et promise, og inde i en async funktion kan await kaldes på et promise. Ved et await vil javascript vente på at dette promise returnerer, før koden eksekvere videre. Da fetch kald returnerer et promise, kan disse awaites. Dette benyttes eks. når der hentes billeder fra et event, da rækkefølgen af billeder ellers vil blive bestemt efter hvilket billede som hentes først, og ikke efter billedernes ID.

## 7.5 Proces/task i MobilApp

I mobilapplikationen gøres der brug af flere tråde, når billeder skal håndteres og når et nyt view skal loades. Når et nyt view skal loades benyttes en anden tråd ved at awaite en asynkron funktion, der initialiserer det næste view der skal renderes hvorefter der skiftes til det ønskede view. Det resulterer i at UI tråden forbiliver responsiv, mens det nyew view initialiseres.
De to klasser MediaUploader og MediaDownloader håndterer overførsel af billedfiler til og fra app'en. Begge bruger en HttpClient til dataoverførslen. Download af billeder sker igennem tråde. For hvert billede der downloades eller uploades, bliver der startet en ny tråd. Dette gøres for ikke at fryse app'en under dataoverførslen, så brugeren fortsat kan navigere rundt. Derfor har MediaDownloader et event, der aktiveres hver gang en et billede er hentet. På denne måde kan en eventhandler håndtere hvor og hvordan, billedet skal gemmes. MediaUploader uploader også hvert enkelt billede i én tråd af samme grund.

### 7.5.1 Proces/task implementering

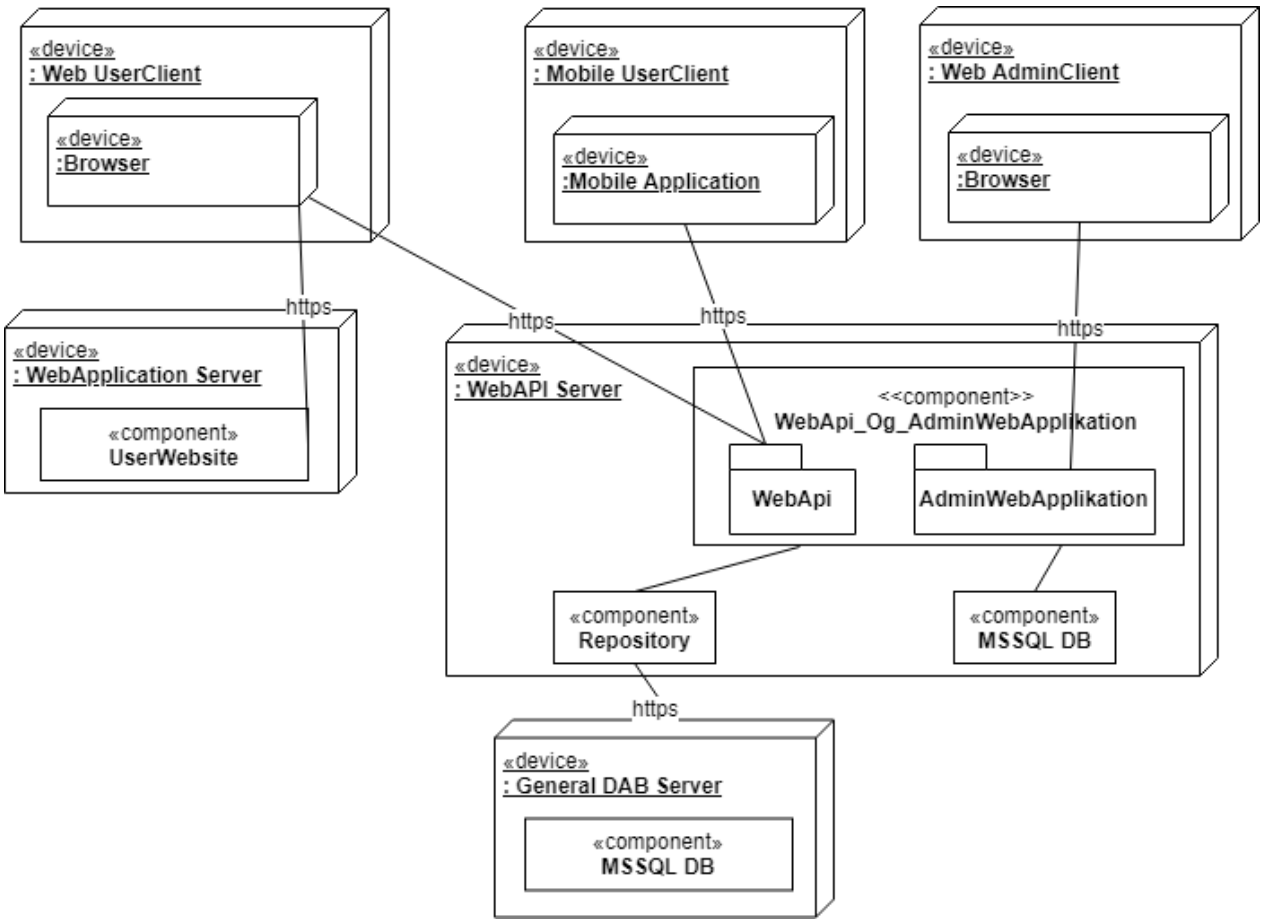
Processer og tasks bliver implementeret ved C#'s Thread og Task klasser og asynkrone metodekald i mobilapp'en. Thread klassen anvendes i forbindelse med de førnævnte tråde, der henter billeder og uploader billeder. Task klassen anvendes i store dele af projektet, da det er .Net standarden for parallel programmering. Derfor bruger mange C# klasser et Task objekt eller et Task<> objekt som returtype i deres metoder. Det medfører at async/await operatorerne anvendes i lige så stor omfang som Task klassen.

### 7.5.2 Proces/task kommunikation og synkronisering

Som tidligere nævnt har MediaDownloader klassen et event, der bruges til at kommunikere mellem MediaDownloader og den viewmodel/klasse, der anvender den. Events anvendes også i høj grad gennem hele projektet, da en stor del af GUI programmering afhænger af events. Dog er disse events ikke altid implementeret som C# events (delegates), men ofte som ICommands fra Prism frameworket, på grund af MVVM arkitekturen.
I forbindelse med synkronisering/trådsikkerhed, bliver lock operatoren kun anvendt i et enkelte tilfælde i forbindelse med download af billeder. Når billederne skal gemmes lokalt på telefonen, anvendes lock for kun at gemme ét billede af gangen. Dette gøres blandt andet fordi billedets filnavn er afhængigt af antallet af gemte billeder (billede1.png, billede2.png, osv.). Hvis to billeder gemmes på samme tid, kan de risikere at få samme navn, hvilket vil resultere i, at et af billederne bliver overskrevet.

# 8 Deployment View

Dette afsnit beskriver hvordan systemet deployeres. Der kan på figur 19, som er et deployment diagram, ses hvilke noder systemet består af og hvordan de er tilkoblet.




Figur 19: Deployment diagram for systemet

## 8.1 Node-beskrivelser

### 8.1.1 Web UserClient

Det er brugeren, der tilgår systemet via. en browser. Den henter hjemmesiden ned fra webapplikation server og kommunikerer herefter med web API'et på WebAPI serveren.

### 8.1.2 WebApplication Server

Det er serveren hvor websitet bliver hentet ned fra, den placeres på Azure.

### 8.1.3 Mobile UserClient

Det er brugeren, der tilgår systemet via. en Mobil Application. Den kommunikere med web API'et på WebAPI serveren. .

### 8.1.4 WebAPI server

WebAPI serveren indeholder et web API som brugeren tilgår igennem mobil- og webapplikationen. Det indeholder også IdentityDAB, der indeholder login data. WebAPI serveren består også af et Adminwebapplikation, som er et Website en admin vil tilgå igennem sin browser.

### 8.1.5 Web AdminClient

Det er en admin, der tilgår systemet via. en browser. Den kommunikerer med WebAPI serveren, hvor den henter siden ned fra Adminwebapplikation.

### 8.1.6 General DAB Server

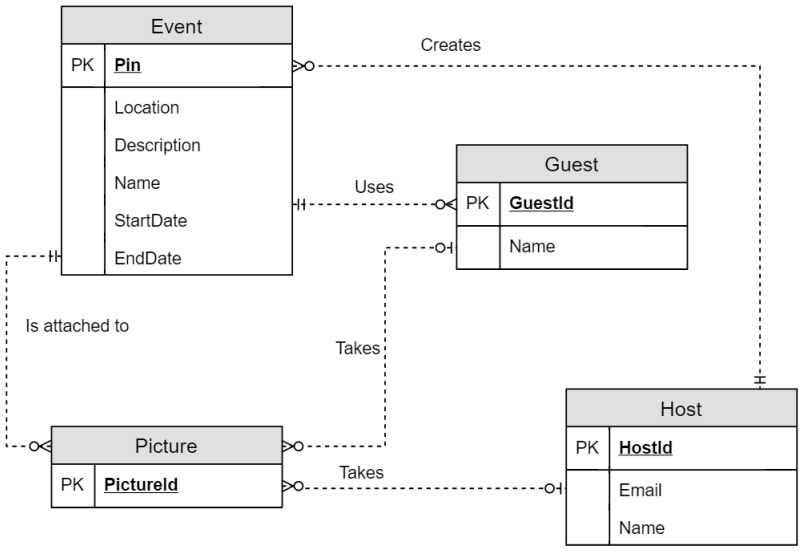
Den indeholder en MSSQL DB, hvor data fra systemet persisteres. Den bliver tilgået og manipuleret af WebAPI Serveren.

# 9 Data View

Det afsnit omhandler, hvordan systemet persisterer data. Som beskrevet tidligere i arkitekturen bruger systemet to databaser.

## 9.1 Generel database

Den generelle database persisterer alt data, der ikke omhandler login. Der bruges en MSSQL database. Der kan på figur 20 ses et Entity-Relationship diagram, der viser hvilke entiteter og tilhørende relationer, der skal være i databasen.




Figur 20: Entity relationship diagram general database

Som det kan ses på figuren, så er der en cyklisk reference. Det gør det mere omstændigt at slette/ændre i databasen, men lettere at tilgå data.

## 9.2 Identity Database

Til at styre login anvendes identity frameworket til ASP.Net Core. Dette gør det muligt at oprette brugere og logge ind og ud. I vores identity database er der en række entities som automatisk bliver oprettet når DbContext arver fra IdentityDbContext, nogen af de entities som er interesante for os er AppUser og UserClaims. AppUser indeholder information om useren som Name, UserName og Email. En AppUser har et en til mange forhold til UserClaims. Det er claims der bruges til at tjekke på om en bruger har ret til at kalde den action den prøver at kalde. UserClaims bliver tilføjet til en user ved oprettelse.

# 10 Generelle design/arkitektur beslutninger

I dette afsnit beskrives design beslutninger, der er taget for systemet og dets dele.

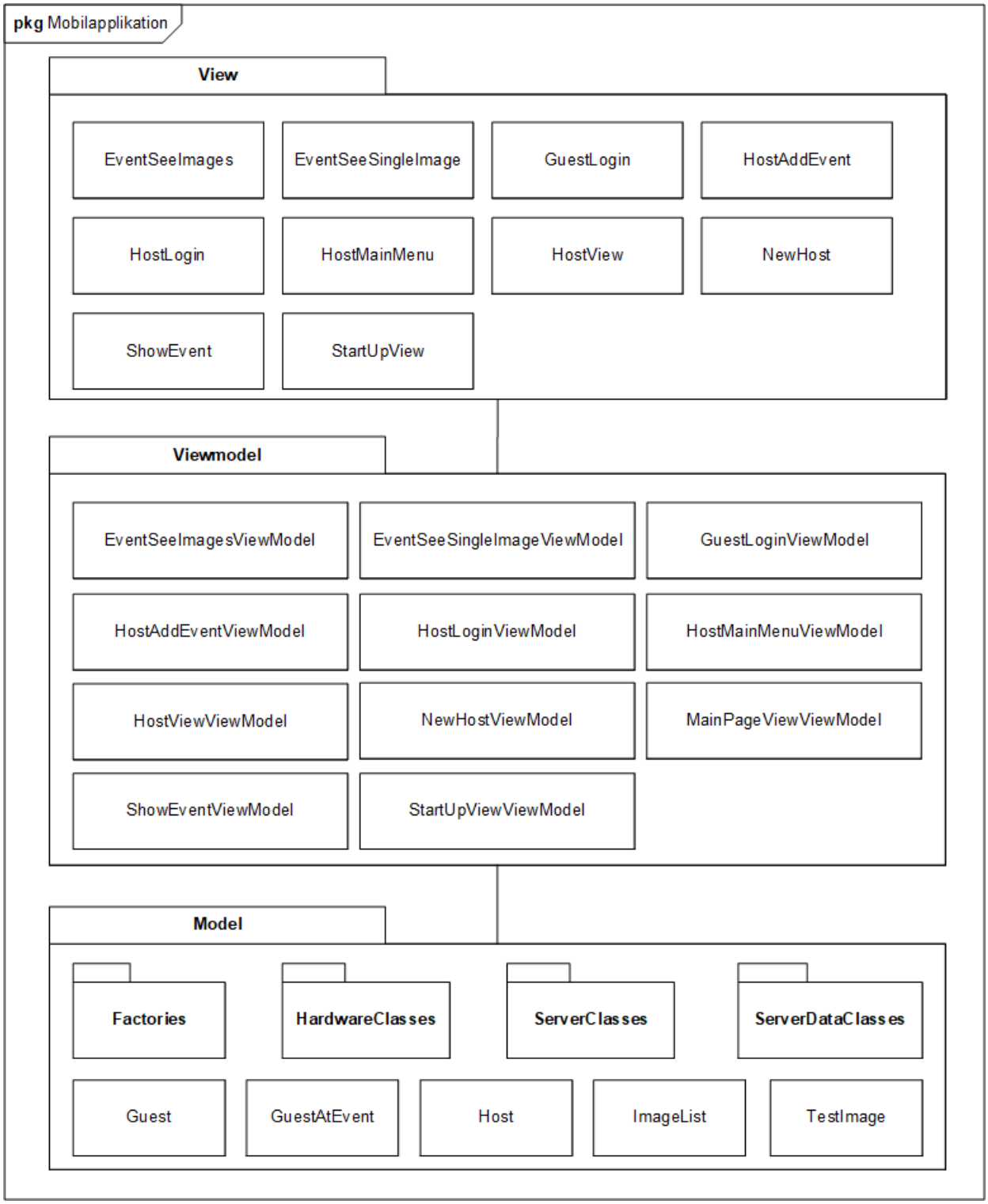
## 10.1 Mobilapplikation

Mobilapplikationen implementeres med MVVM arkitekturen.
Det betyder at strukturen af koden er delt op i 3 lag:

- View
  - Består af en .xaml og en code-behind xaml.cs fil. Her implementeres det visuelle/brugergrænsefladen som en bruger møder. Det er altså det, der kan ses i appen.
- Viewmodeller
  - Viewmodellerne er klasser, der hver især hører til et respektivt view. Viewmodellerne står for bindingen mellem et view og en model. Viewmodellerne står ligeledes for at binde data til et view, og det står for logikken bag et view. En viewmodel kan bruge nul eller flere model klasser. Eksempelvis bruger en viewmodel en model, der har adgang til databasen. Derefter bruges det data, der er hentet fra en model, til at blive renderet på det view, som viewmodellen hører til. Viewmodellen står altså for logikken bag et view.
- Modeller
  - Modellerne består blandt andet af klasser og factories som viewmodellerne bruger. Derudover er der også modeller der står for tilgang til databasen, eller hardware tilgang til den enhed som appen kører på. I modellerne er der også implementeret helper-klasser, som viewmodellerne bruger.

Da arkitekturen for mobilapplikationen er MVVM, gøres det lettere at teste systemet, debugge og implementere Pga. denne opdeling betyder det også at der ikke er en direkte database tilgang fra et view. Derimod bruger viewmodellerne en model, der er implementeret som er repository, og denne har adgang til databasen.

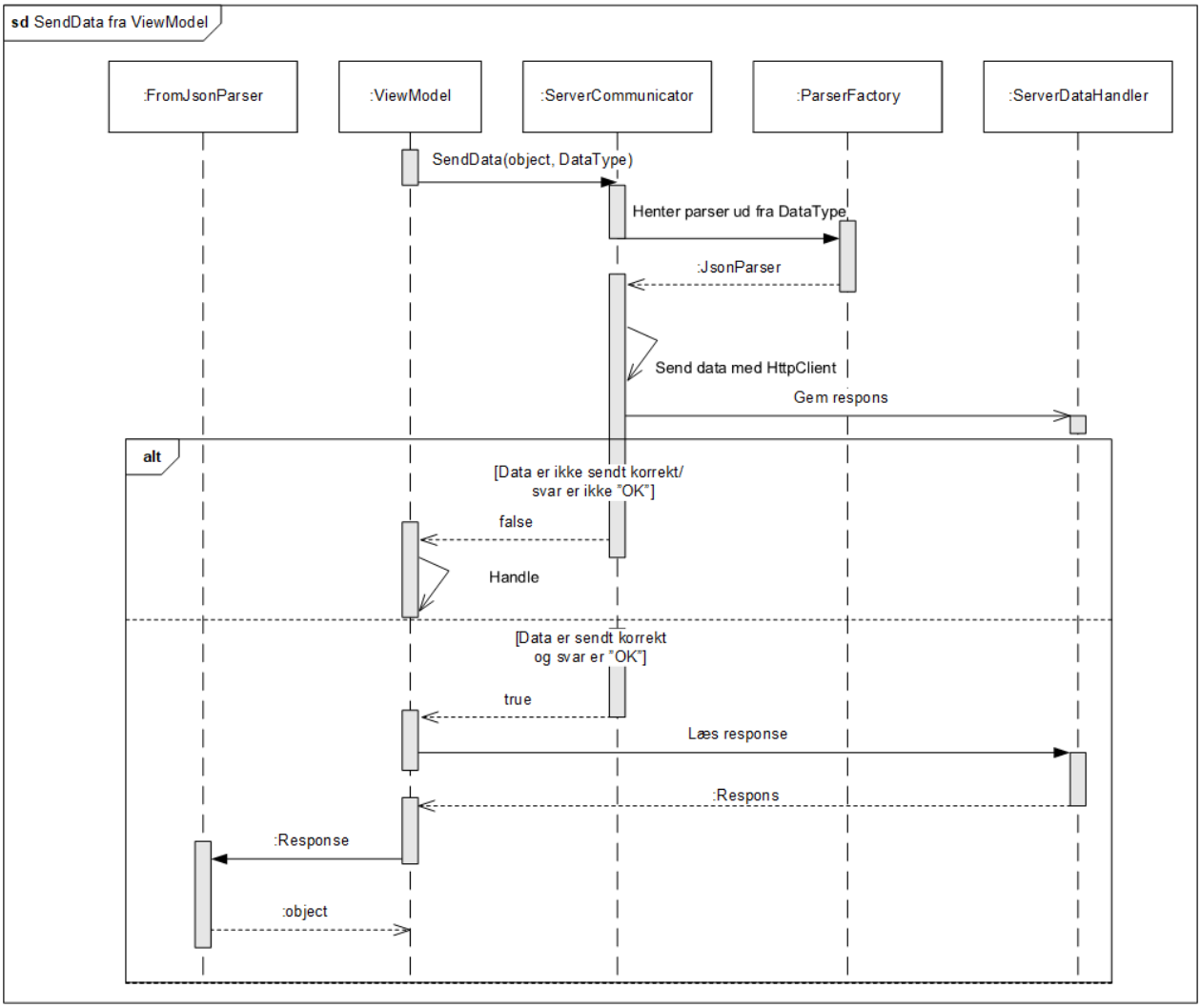
Understående på figur 21 ses der pakker, der viser de 3 lag i applikationen. Her ses der i pakkerne views og de tilhørende viewmodels. Til sidst ses de modeller, som viewmodellerne benytter. I modellaget ses der nogle pakker, som hver især indeholder klasser. De vises som pakker i diagrammet for overskuelighedens skyld.




Figur 21: Pakke diagram, der viser klasserne i mobilapplikationen

### 10.1.1 Serverkommunikation

Klassen ServerCommunicator er hovedansvarlig for kommunikation med serveren igennem web API'et. Den sender og modtager oplysninger om brugere og events og kan ligeledes slette brugere, events og billeder. Download og upload af billeder er den eneste del af serverkommunikationen, der ikke varetages af ServerCommunicator. Kommunikationen med serveren er forsøgt lavet så generisk som muligt, så alle ViewModels kan anvende den samme metode fra en implementering af "IServerCommunicator"interfacet. Der er derfor anvendt factories og strategy pattern for at generere de korrekte JSON filer run time.
Interaktionen mellem en ViewModel og ServerCommunicator er forsøgt forklaret igennem sekvensdiagrammet på figur 22




Figur 22: Interaktion mellem ViewModel og ServerCommunicator

Det essentielle ved sekvensdiagrammet er genereringen af JSON filen igennem JsonParser og håndtering af serverens respons med FromJsonParser. Disse to interfaces implementeringer vælges run time ud fra hvilken data der sendes og hvilken data, der forventes modtaget.
Implementeringerne af IJsonParser anvender en Dictionary til at generere JSON filerne, da dataklasserne

ellers var 100% koblet til serverens forventede data. Der bliver dannet key-value par med de korrekte navne og data, som serveren forventer, der derefter blevet lavet om til JSON filer. På denne måde kan serverens API ændres, uden dataklasserne skal ændres alle steder i app'en.
Både JsonParser og FromJsonParser bruger NuGet packagen NewtonSoftware til henholdsvis at generere og læse JSON filer.

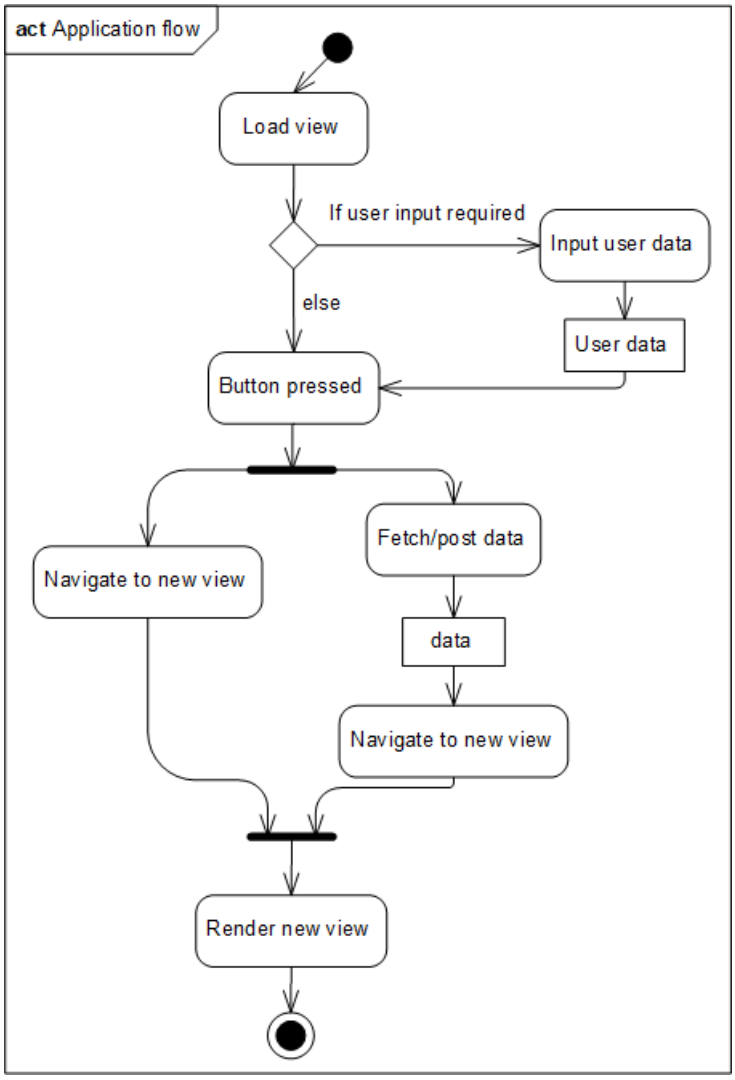
### 10.1.2 Medie up- og download

De to klasser MediaUploader og MediaDownloader håndterer overførsel af billedfiler til og fra app'en. Begge bruger en HttpClient til dataoverførslen. Når et billede skal uploades, forventer serveren at modtage en base64 streng med billedets data. Dette sker i den specifikke JsonParser. Download af billeder sker igennem tråde. For hvert billede der downlodes, bliver der startet en ny tråd. Dette gøres for ikke at fryse app'en under dataoverførslen, så brugeren fortsat kan navigere rundt. Derfor har MediaDownloader et event, der aktiveres hver gang en et billede er hentet. På denne måde kan en eventhandler håndtere hvor hvordan, billedet skal gemmes. MediaUploader uploader også billeder i en tråd af samme grund.

### 10.1.3 Flowet i applikationen

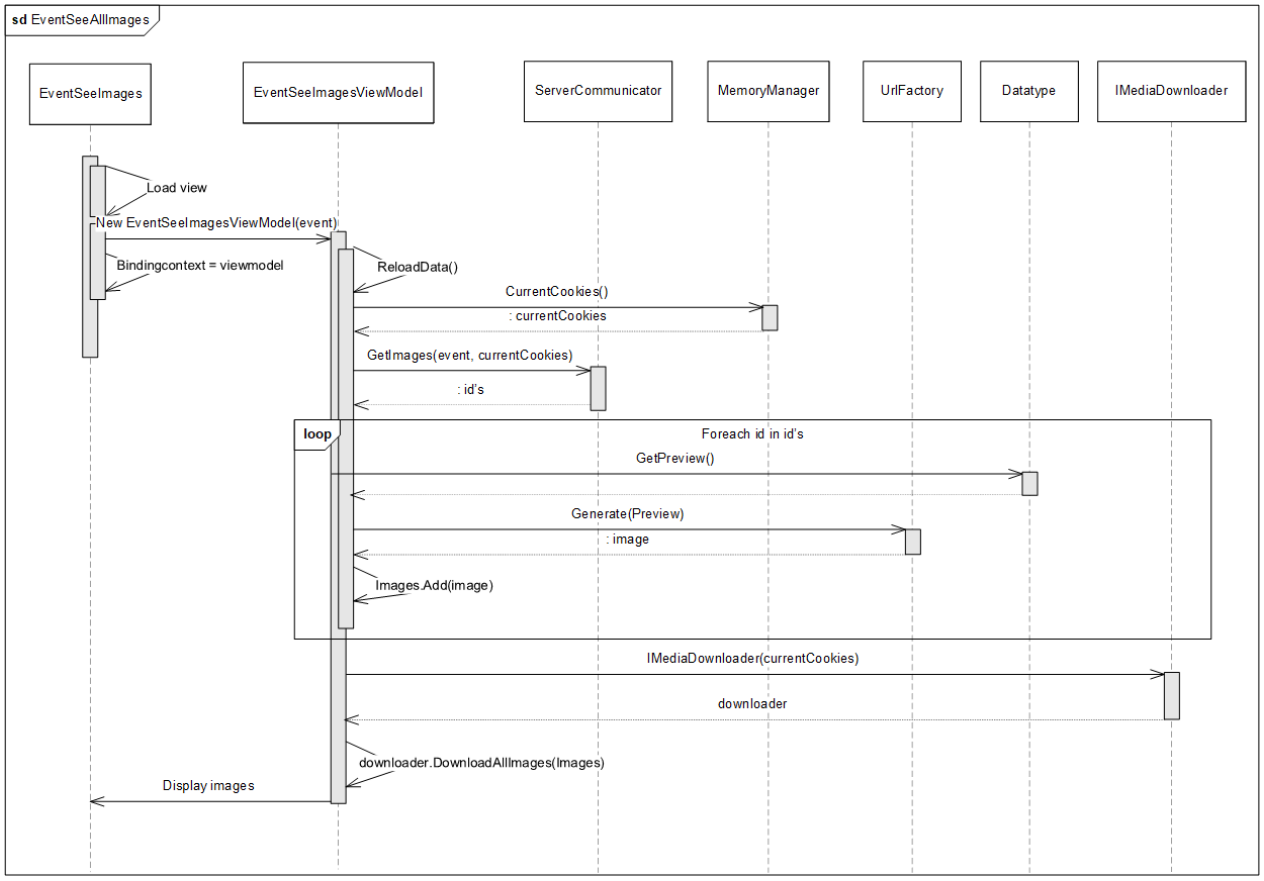
Understående ses et flow diagram og et sekvensdiagram der viser adfærden og flowet i applikationen. Denne funktionalitet er den samme i næsten alle views, hvorfor der kun vises denne ene illustration.
Der er dog en undtagelse for det viste aktivitetsdiagram. Hvis man eksempelvis er inde på et event, er det muligt at trykke "tag billede". Dette åbner kameraet så det er muligt at tage et billede, og efterfølgende er det muligt at se billede under "se alle billeder".

Flowet for mobilapplikationen kan ses understående på figur 23 Diagrammet viser funktionaliteten af mobilapplikationen.




Figur 23: Aktivitetsdiagram der viser flowet i applikationen

Sekvensdiagrammet nedenstående viser det view, hvor det er muligt at se alle billeder. Denne viser generelt det flow der er i applikationen. Dvs. at alle views binder til deres tilhørende viewmodel, hvor man ser, at det er viewmodellen der bruger modellerne, hvorefter viewmodellen opdaterer view'et.




Figur 24: Sekvensdiagram der viser den typisk adfærd i applikationen

### 10.1.4 Dependencyservices

Xamarin har en funktionalitet kaldet dependencyservices, der tillader forskellige implementeringer for det samme interface baseret på styresystemet. Det bliver brugt i flere af de anvendte NuGet packages, men også af modellen IFileDirectoryAPI, som returnerer forskellige filstier baseret på styresystemet. Denne funktionalitet anvendes i stedet for PCLStorage (som også vedrører filmanipulering), fordi filstierne i PCLStorage er gemt for brugeren på Android. Dette er ikke hensigtmæssigt, når brugeren skal kunne finde billederne på sin telefon. Derfor anvendes PCLStorage til at gemme tidligere brugere og deres cookies, mens IFileDirectoryAPI anvendes til billeder.

## 10.2 Webapplikation

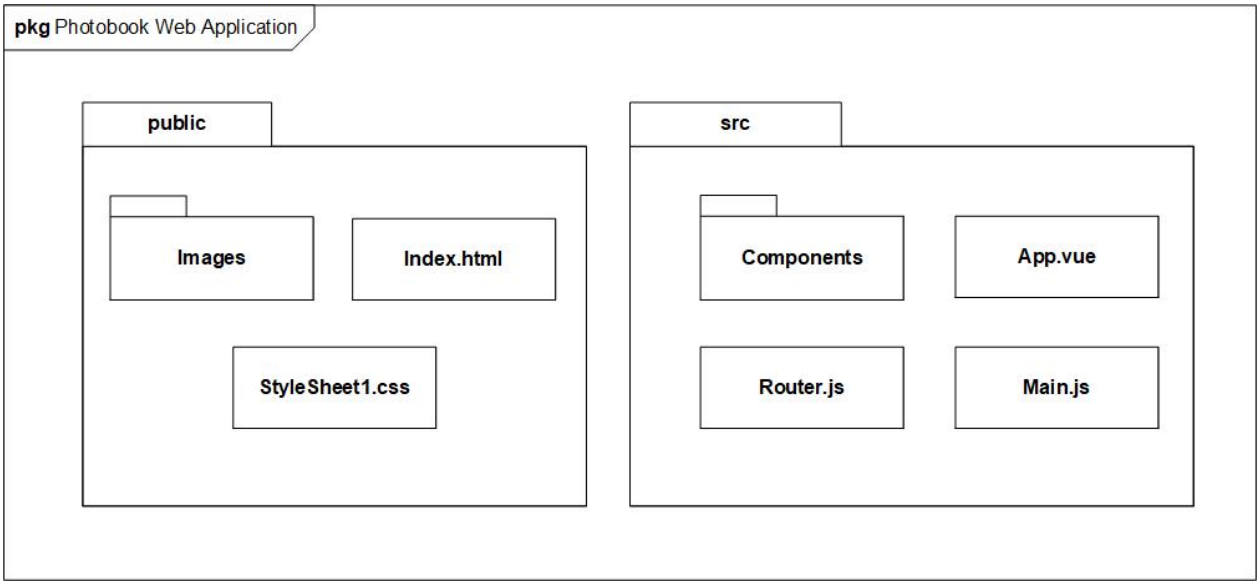
### 10.2.1 Webapplikation design

Webapplikationen implementeres som en single page applikation ved hjælp af Javascript frameworket Vue. Til dette opsættes et Vue projekt i Visual Studio, som benytter Vue single file components. Derved bliver hver side som skal laves i webapplikationen implementereret, som en Vue fil [6].

Udover Vue benyttes bibliotekerne Vue Router[7] og Vue Cookies[4]. Router benyttes til at skifte mellem

de forskellige komponenter i webapplikationen, og cookies benyttes til at gemme data som skal bruges i flere komponenter, eks. brugernavne eller event pins. Her kunne alternativt bruges VueX, men i VueX gemmes instanser ikke når siden genindlæses, og der er derfor set bort fra denne løsning[5]. Der er desuden benyttet fetch kald til kommunikation med web API. Dette er valgt over en almindelig httprequest da de benytter promises, og derved tilføjer ekstra funktionalitet i forhold til brugen af asynkrone/synkrone kald[2].

Udover Vue components består webapplikationen af en række software pakker, som hver især er grundlæggende for at Vue komponenterne kan fungerer. Herunder ses et skema for de pakker som webapplikationen består af:




Figur 25: Pakkediagram for webapplikation

**Public**

Public pakken indeholder de ressourcer, som kan tilgåes fra hele applikationen. Deres funktion er som følgene:

- Images
  - Images indeholder de billeder, som skal være tilgengægelige for hele webapplikationen, eks. logoer.
- Index.html
  - Her placeres den html fil, som hentes først når webapplikationen hentes startes. Denne side vil indeholder Vue komponentet App.Vue fra SRC pakken.
- Stylesheet1.css
  - Her gemmes den css kode som skal være hele applikationen skal dele. Dette gøres sådan at der for hele applikationen gælder samme styling, så der opnåes et uniformt design.

**SRC**

SRC pakken indeholder de ressourcer som danner kernen i webapplikationen. Her findes Vue komponenter, samt de filer hvor Vue og Vue router konfigureres.

- Components

  - Components indeholder de Vue komponenter, som hver tilsvare en side på webapplikationen med tilhørende funktionalitet. For en oversigt over Vue komponenterne henvises til afsnit 6.2.7.

- App.Vue
  - App.Vue er et Vue komponent, som vil indeholde det som skal være fælles for alle komponenter. Dette indebærer en navigations bar og et router view, hvor de enkelte Vue komponenter vises.

- Router.js
  - Her konfigureres Vue routeren, som benyttes til at navigere mellem de forskellige Vue komponenter. Dette gøres ved at oprette stier til hvert komponenet.

- Main.js
  - Her konfigureres selve Vue, med de afhængigheder som Vue skal bruge.

**Vue Components**
Hver Vue fil i componenets indeholder HTML kode og tilhørende Javascript kode. Hvert Vue component er opdelt i følgende underdele:

- Data
  - Data indeholder de variabler som Vue componentet skal kunne gemme og bruge, her gemmes eks. en liste med fotos, når disse hentes fra web API'et.

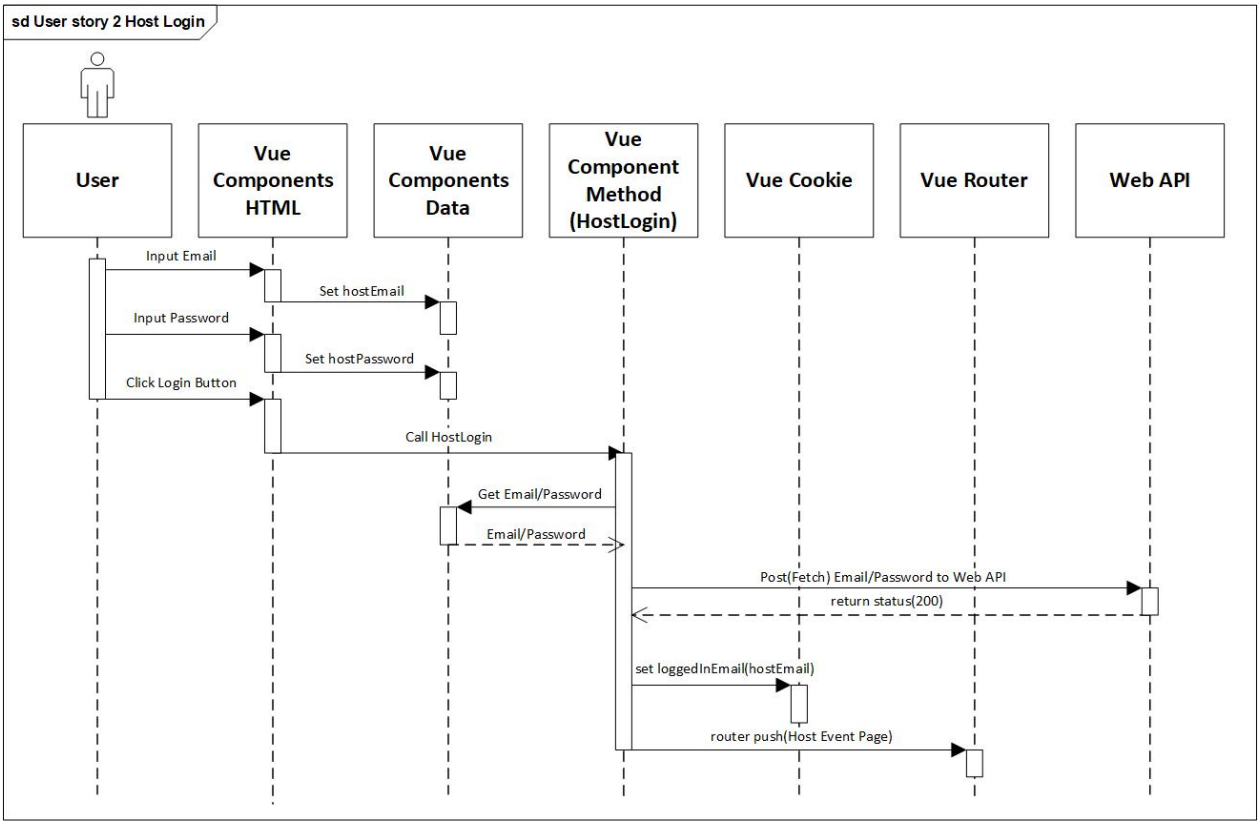
- Methods
  - Her ligges alle metoder metoder, som kan tilgåes fra et Vue component, disse kaldes når der trykkes på eks. knapper i html koden, eller ved andre metoder.

- BeforeMount
  - Dette er en funktion som kaldes når Vue komponentet loades. Her indlæses og sættes eks. cookies og kaldes nødvendige opstarts funktioner.

I det grafiske design af brugergrænsefladen, er det valgt at lave et responsivt design, som tilpasser sig skærmens størrelse ved hjælp af procentvise størrelser i css koden og med ekstra plads i siderne af skærmen. Dette betyder at webapplikationen kan fungere på flere forskellige computerskærme, uden at det grafiske ændres for meget. Der er dog valgt at bruge specifikke størrelser til eks. knapper og input felter, disse er holdt forholdsvist små, men ville blive henholdsvis meget store og meget små hvis de også skulle skalere.

### 10.2.2 Eksempel på systemets adfærd

På figur 26 ses et eksempel på adfæren i en Vue komponent. Bemærk at sekvensdiagrammet ikke er teknisk præcist, men blot giver indblik i hvordan komponenten fungerer, på et konceptuelt plan.




Figur 26: Eksempel på adfærd i et Vue komponent, vist ved User Story 2 Host Login

I sekvens diagrammet indtaster User først de nødvendige oplysninger i Vue komponentets HTML kode. HTML koden binder dette data til Vue komponentets data. Herefter trykker brugeren på login knappen, som er 'bindet' til Vue metoden Host Login. Denne metode tager dataet i Vue data, pakker det til en JSON fil, poster til web API'et med et fetch kald,og ved hjælp af et promise, checkes det at der returneres med en succes kode (200). Herefter sættes en cookie med Email, og Vue komponentet skiftes ud med et Vue komponent, som indeholder en start menu for en Host, ved hjælp af Vue Router.

Dette eksempel beskriver det generelle flow i langt de fleste Vue komponenter. Der kan dog være forskel på, om der fetches først, hvorefter modtaget data gemmes, eller om der gemmes data først, hvorefter der fetches. Enkelte funktioner benytter heller ikke et fetch kald, men gemmer blot lokalt data. Dette er eks. tilfældet når der uploades filer, hvor en funktion først gemmer billederne lokalt i data, hvorefter en anden funktion uploader billederne til databasen med et fetch kald.

### 10.2.3 Ekstra biblioteker

Til implementeringen er desuden benyttet en række ekstra biblioteker, herunde JSZip, JSZip utils, filesaver.js, progressbar.js og picsum.photos. De første 3 er benyttet for at gøre det muligt at hente alle billeder på en gang. Disse biblioteker kan zippe filerne til en mappe, og downloade dem til brugerens pc.

progressbar.js benyttes til at give brugeren en ide om, hvor langt forskellige processer i systemet er, eks. upload af billeder.

picsum.photos er en tjeneste som returnere et tilfældligt billede ved hver request. Dette bruges til at sætte forskellige baggrunde på Photobook ved hver instans.

## 10.3 WebAPI

Til design af WebAPI anvendes AspNetCore Mvc Controllers. Som nævnt tidligere ligger der to funktionaliteter i en solution. Der er selve web API'et og der er Administrator siden. Administrator siden anvender Index actions fra hver controller til at navigere fra view til view. Disse view's præsenterer blot data som er gemt i databasen.

web API delen af projektet anvender tre controllers, alle mulige API kald kan ses på https://Photobookwebapi1.azurewebsites.net/swagger/index.html:

- AccountController

  Denne controller er designet til at styrer oprettelse af brugere samt login og logout. Til dette skulle et system bruges til at logge ind og ud. Det er blevet valgt at identity framework bruges til dette system. Dette er også grunden til at der i systemet er to databaser, en til identity brugere og en til systemet generelt. Ud over at skulle oprette identity brugere opretter denne controller også guests og hosts i den generelle database. Der blev forsøgt i første omgang at dele funktionaliteten til at udføre CRUD operationer ud på guest og host entities til de tilsvarende controllers men det endte med at gøre det mere besværligt end det var værd. Derfor blev det ændret til at AccountControlleren håndterer alt der har med alle slags brugere at gøre.

- EventController

  Denne controller indeholder CRUD funktionaliteten for alt der har med events at gøre. For at kunne give guest adgang til at kunne uploade billede til et event skal der til hvert event genereres et unikt eventpin. Denne pin er valgt til at være 10 cifre lang bestående af både bogstaver og tal. Den er valgt til at være så lang med henblik på at kunne skalere det til at systemet indholder mange events på samme tid. Det hjælper også på sikkerheden af systemet at denne pin er sværere at knække. Denne pin bliver genereret i den action der laver Create.

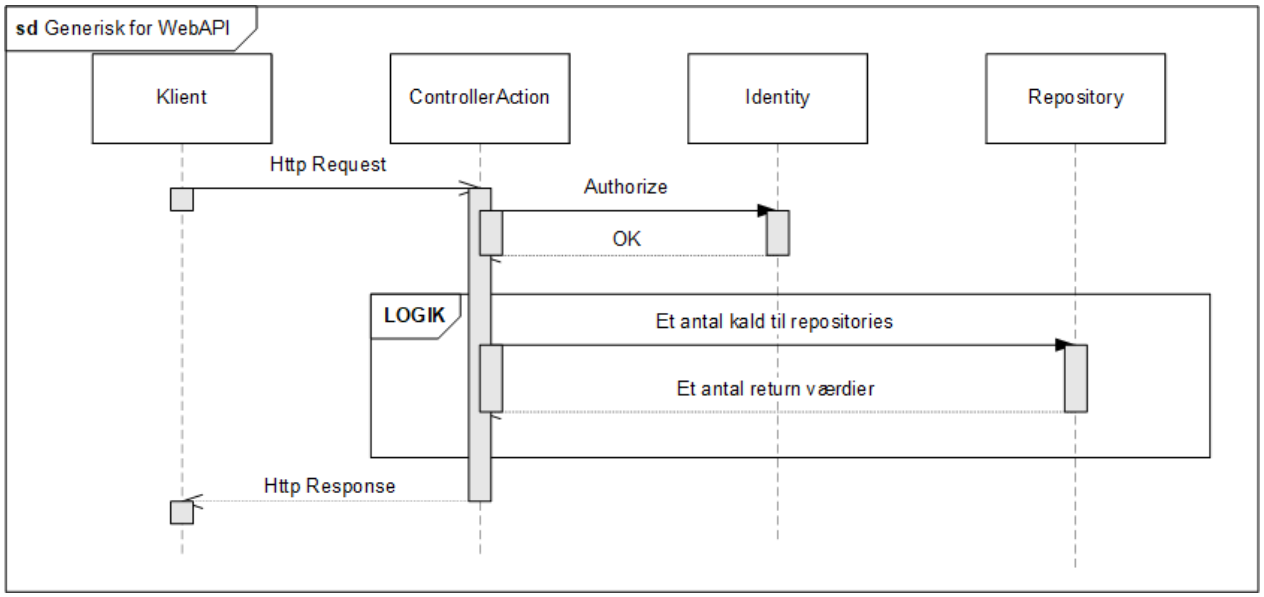
- PictureController

  PictureController står for alt der har med billeder at gøre. Til at starte med var det tænkt at når der blev uploadet billeder skulle de alle sammen gemmes i root mappen hvilket ville sige at de ville være tilgængelige for omverdenen. Dette sikkerheds problem var et af de større design beslutninger skulle tages. Den måder man henter billeder på er ved først at bede om at få alle billede id'er til et specifik event, dette er en action i controlleren. Når man så har hentet alle billede id'er kan man gennem en action hente selve billedet gennem en anden action som returnere en PhysicalFile. Når der gøres på denne måde kunne billederne ligges et sted som ikke er offentligt tilgængeligt og man skal dermed igennem en action for at kunne tilgå et billede. Når man skal igennem en action kan det styres hvem der har lov til at hente billeder.

En anden større beslutning for denne controller var noget der først blev opdaget da blev nødvendigt. Problemet der blev opdaget var at det tog alt for lang tid at overføre billederne når brugerne ønsker at se alle billeder tilhørende et event. Løsning blev at der laves to actions til at hente billder, En preview version hvor billedet er gjort meget mindre og en med det fulde billede. Dette gør at når brugeren ønsker at se billederne i appen så bliver preview versionen hentet og når brugeren ønsker at downloade et billede så bliver den fulde version hentet.
Måden preview versionen af billedet bliver oprettet sker i Create funktionen. Der bruges et bibliotek kaldet MagicScaler [4] til at gøre billedet mindre.
Grunden til at oprettelsen af preview billeder sker i Create funktionen frem for når billederne hentes er fordi det fordeler arbejdstiden bedre, da man sjældent uploader mere end et billede af gangen, hvorimod man downloader mange på en gang.

### 10.3.1 Sekvensdiagrammer

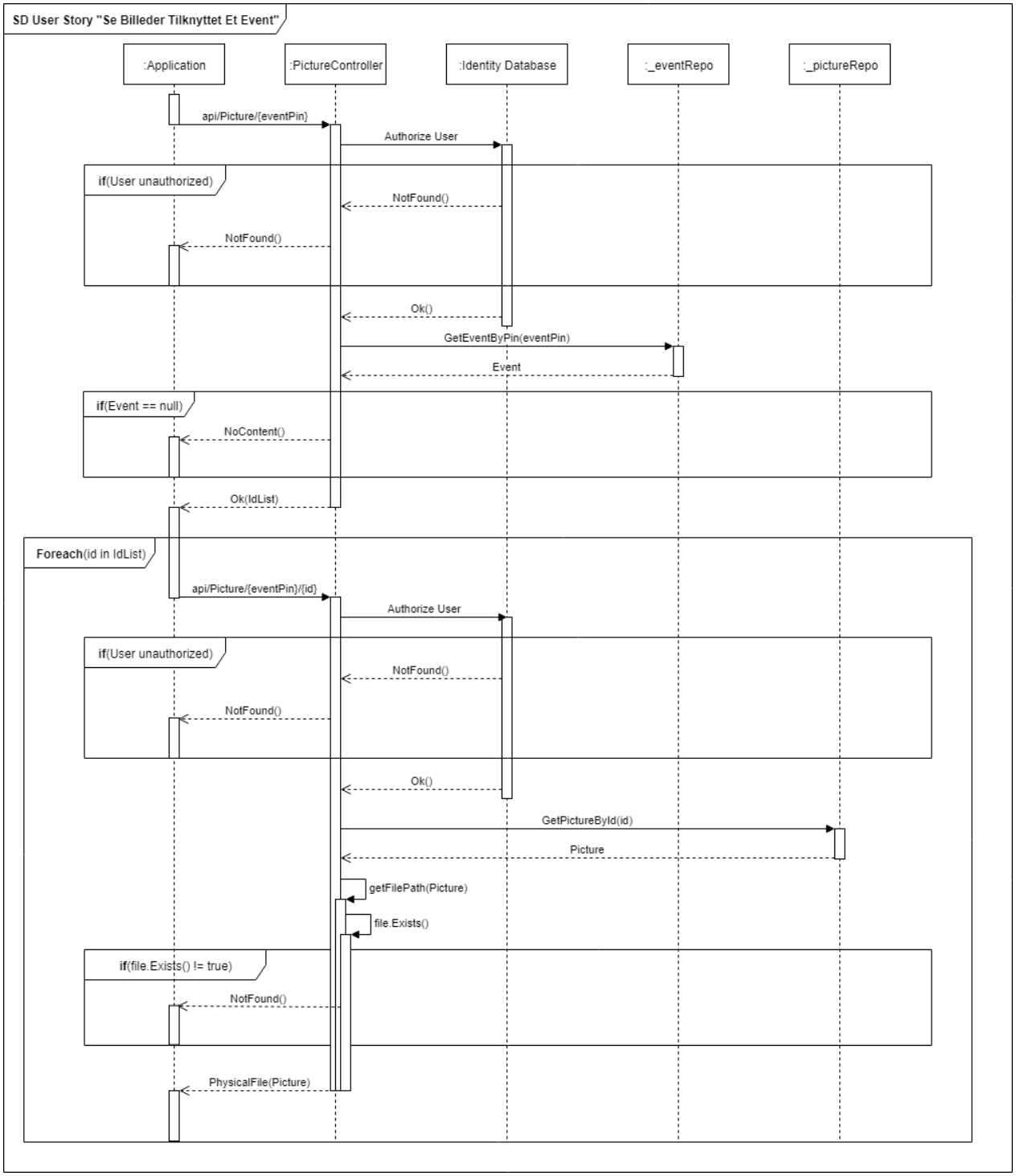
Figur 27 viser et overblik over hvad der sker når der laves kald til WebAPI.




Figur 27: Generisk Sekvens diagram for WebAPI

Figur 28 viser sekvensdiagram for User Story 'se billeder' denne user story er valgt da den er den mest komplekse for WebAPI.

[4] https://github.com/saucecontrol/PhotoSauce




Figur 28: Illustration af hvad der sker i systemets API når en applikation vil se billederne fra et event.

### 10.3.2 General Database

Den generelle database indeholder alt data for systemet, der ikke omhandler login. Databasen er en relationel MSSQL database. Databasen designes code first ved brug af Entity Framework Core.

### 10.3.3 IdentityDatabasen

Identity databasen indeholder alle brugere i systemet der omhandler login. Databasen er en relationel MSSQL database som er designet code first ved brug af Entity Framework Core, Identity. Et af de store valg der blev træffet i forbindelse med brugen af denne database er hvordan det vælges at authorize på hvad den nuværende bruger har lov til at gøre. Der er to oplagte muligheder bygget ind i identity frameworket, Claims og Roles. Der blev valgt at bruge Claims. Grunden til at claims blev valgt var at det tilbyder noget fleksibilitet som Roles ikke gør og da designet og implementeringen påbegyndes var vides det ikke hvilke slags authorization der var behov for. Det har så vist sig at den måde claims bruges på sagtens kunne være gjort med roles i stedet. Claims var dog stadig en god beslutning da det kan flere ting som der kunne blive brug for i viderudvikling af systemet.

### 10.3.4 Repository

Den service der bruges af web API'et til at tilgå den generelle database er implementeret med et ikke generisk repository mønster. For at gøre web API'et testbart så implementere hvert repository et tilhørende interface. ORM'en repositoriet bruger til at tilgå den generelle database er Microsoft's eget EFC. I forbindelse med EFC, så skal et repository bruge en context. Der er skrevet en artikel af Maedi El Gueddari, der beskriver om gode standarder, når der arbejdes med context's i EFC [3]. Derfor bliver en context Dependency injectet i et repository, så contexten's levetid bliver lige langt som et web API request.

I forbindelse med repositoriet ud fra et performance synspunkt. Så er der specificeret i kravene forskellige krav vedrørende reaktionstider. Repositoriet's rolle i den sammenhæng betyder, at databasen kaldene ikke skal tage unødvændigt tid. Derfor er repositori'ets metoder implementeret asynkrone ved brug af tasks. Det gør at web API'et kan starte nye tråde, når den bruger repositoriet. Dette giver web API'et alle muligheder til at optimere programmet, så det kan udføre andet arbejde imens det afventer database kaldene.

## 10.4 Exception og fejlhåndtering

### 10.4.1 WebAPI

Ved brug af systemets web API mange muligheder for at der kan ske fejl, funktioner kan kaldes med ugyldige værdier, med værdier som godtages ad selve funktionen men ikke er gyldige ved de kald som laves videre ned i systemet, eller brugeren kan ske ikke at have tilladelse til at lave det forsøgte kald. Generelt er det vigtigt at brugeren har en idé om hvad der gik galt i operationen. Til dette formål bruges der et væld af forskellige HTTP resultat koder, med forskellige passende fejl- og successbeskeder. For at kunne returnere de rigtige beskeder bliver der tjekket på om de udførte kald videre har returneret korrekt, hvis ikke det forventede kommer retur håndteres det ved at returnere en HTTP-fejlkode. Et godt eksempel på dette er Delete Picture som returnerer Unauthorized() hvis ikke brugeren har ret til at slette billedet, NotFound() hvis ikke det billede som skal slettes eksisterer, eller NoContent() hvis det lykkes at slette billedet.

### 10.4.2 Webapplikation

Ved brug af webapplikationen er der mange parametre som alle skal spille sammen, for at systemet fungerer pålideligt. Der skal gemmes data lokalt, der skal kaldes til WebAPI og der skal opdateres brugergrænseflader.

Der er til disse ting i et vidst omfang benyttet fejlhåndtering.

I webapplikationen er der benyttet fejlhåndtering i forbindelse med de fleste fetch kald til web API'et. Dette er ofte et sted, hvor der er opstået fejl i eksekveringen, da to forskellige systemer skal tale sammen. Måden dette gøres på er ved at kigge på respons status fra WebAPI, og alt efter status, laves så de pågældende kald. Er en status eks. 200, betyder det at kaldet er succefuldt, og der kan fortsættes i eksekveringen af koden. Er status derimod eks. 400 eller 500, er der sket en fejl i henholdsvis klient eller server. Dette udskrives typisk som en alert til brugeren og funktionen stoppes. Da langt hovedparten af funktionalitet i webapplikationen omhandler at lave kald til WebAPI, fanges langt størstedelen af fejl også heri.

### 10.4.3 Mobilapplikation

Ved login på mobilappen, både som guest og host, kan der ske fejl. Det kan enten være forkert password, et guest navn som allerede er brugt, eller en serverfejl. For at give brugeren en idé om, hvad der er gået galt, er der designet et interface kaldet IServerErrorCodeHandler. Dette interface har flere implementeringer, der passe til de forskellige scenarier. Interfacet har en enkelt metode (*Handle()*), der håndterer fejlkoden specifikt for implementeringen af interfacet.
Efter *Handle()* er kaldt, vil klassen have genereret en string (Message), der eksempelvis kan bindes til et label, som brugeren kan læse og derefter fejlhåndtere. En implementering af dette interface anvendes blandt andet på denne måde, når brugeren logger ind som guest.
Inden brugeren sender sine oplysninger (hvad end det er som host eller guest) bliver brugerens email valideret. Hvis emailen ikke er i korrekt format, (indeholder ikke '@' eller '.') bliver der kastet en IncorrectEmailException. Denne kan så gribes og en fejlmeddelelse kan udskrives til brugeren.

## 10.5 Implementeringsværktøjer- og sprog

Følgende værktøjer er benyttet:

**Til implementering:**

- Microsoft Visual Studio 2017
- Vue CLI

**Til dokumentering:**

- Overleaf (LaTeX)
- Swagger

**Til test:**

- Postman

**Andre:**

- Tortoise Git

**Sprog:**

- C#
- Javascript
- HTML
- CSS

## 10.6 Biblioteker

**Til implementering:**

- Entity Framework Core
- ASP.net core
- Photosauce.Magicscaler
- NLog
- Prism
- Vue
- Vue Router
- Vue cookies
- JSZip
- JSZup utils
- Filessaver.js
- Progressbar.js
- Newton software
- PCL Storage
- Xamarin
- Plugin.media
- Plugin.permissions

**Til test:**

- NUnit
- NSubstitute

# 11 Kvalitet

Der vil i dette afsnit blive gennemgået, hvilken påvirkning kvalitet har haft for designet af systemet. Der vil blive set på nøgleattributter inden for software produkt kvalitet, og gennemgået hvordan det har haft indflydelse på Photobook [1].

## Ydeevne effektivitet:

Det er en central egenskab for en Web- og MobilApplikationer at have en hurtig reaktionstid. Derfor er systemet designet med et web API, der kan håndtere flere henvendelser på samme tid. Desuden kalder web API'et database serveren asynkront, så den ville kunne udføre andet arbejde imens den venter.

### Ydeevne ved webapplikation

Webapplikationens ydevne afhænger af flere del elementer. Først og fremmest er webapplikationens ydeevne begrænset af dens usability. Dette er eks. tilfældet når der skal hentes billeder. Her kunne fetch kald godt kaldes asynkront, men for at billederne skal komme i kronologisk rækkefølge er det valgt at vente på hvert fetch kald. Dette sænker ydeevne betragteligt. Det ville være muligt at sætte billeder i rækkefølge baseret på deres meta data, men denne løsning er væsentlig mere omfattende i forhold til den valgte, og ville kræve ekstra udviklingstid.

I forbindelse med download af billeder, kan ydeevnen af systemet falde drastisk, hvis mængden af billeder stiger. Dette skyldes at de eksterne biblioteker, som er benyttet her, håndterer mængde af data meget langsomt. Ved eks. 100 billeder er der tale om ca. 350 mB data, hvilket naturligt forsinker processen. Alternativt skulle man udvikle sin egen løsning på det, som eks. kunne kører på serveren i stedet for klienten, men dette ville igen påvirke udviklingstiden af systemet.

Når der køres et production build af webapplikationen, bundles applikationen, så mængden af data klienten skal hente og bearbejde begrænses. Dette gør at brugen af systemet bliver hurtigere, men det gør det væsentligt svære at debugge og udvikle på. Derfor er der i udviklingsfasen benyttet en lokal development server, som gør at hele source koden kan tilgåes.

### Ydeevne ved mobilapplikation

For at optimere ydeevnen i mobilapplikationen bruges der C#'s Thread og Task klasser og asynkrone metodekald. De asynkrone metodekald anvendes når et nyt view skal loades. Derfor initialiseres et nyt view asynkront hvorved UI tråden forbliver responsiv. Dette er blandt andet gjort for at optimere ydeevnen, og sikre en bedre usability af app'en.

Det der særligt kunne optimeres ifht. ydeevnen i mobilapplikationen, er håndteringen af billeder. På nuværende tidspunkt renderes billederne alle sammen på én gang. Det ville her i stedet have været fordelagtigt at gøre brug af et pipeline designmønster, der igennem forskellige stages ville bearbejde et billede og til sidst rendere ét billede ad gangen når billedet var igennem pipelinen. Her ville det også være muligt at benytte én tråd pr. stage i pipelinen, hvilket ville medføre at flere billeder kunne håndteres ad gangen og dermed optimere ydeevenen. Dette ville medføre at ét billede blev renderet/loadet ad gangen, og derved ville brugeren ikke

skulle vente på at alle billeder var loadet og renderet.
En anden mulighed for at øge ydeevnen ifht. billeder ville være at gøre brug af lazy loading. Det ville medføre, at der kun blev loadet de billeder som ville være synlige for brugeren. Derved ville der nødvendigvis ikke blive loadet alle 100 billeder til et event, hvis brugeren kun scrollede ned til 50 af billederne.
Denne sammenhæng af pipeline designmønstret og lazy loading ville gøre ydeevnen betragteligt bedre, ifht. håndtering af billeder i appen.

### Anvendelighed:

I forbindelse med anvendelighed er mobilApplikationen implementeret, så den kan fungerer på flere OS. Desuden er både Mobil- og webapplikationen implementeret responsivt, så de kan håndtere forskellige skærmopløsninger. Webapplikationen virker også på flere forskellige browsers. Begge brugergrænseflader er designet, så de er lette at bruge. Dette er eftertestet ved bruger tests, hvilket er uddybet i test dokumentet.

### Sikkerhed:

Systemet er ikke designet til at håndtere de problemer, der er til stede ved online services. Men der er brugt Identity Framework på web API'et, der automatisk kryptere vigtige brugeroplysninger(password).
Systemet giver ikke mulighed for at bestemme, hvem som har adgang til et event udover pin tilladelse. Det er heller ikke muligt at ændre password for en bruger, eller benytte en 2 faktor godkendelse ved oprettelse. Dette er altsammen ting, som burde implementeres senere i fremtidigt arbejde, men som er blevet undladt nu, da der blot er tale om en teknisk prototype.

### Vedligeholdelse:

Den tydelige opdeling af ansvarsområder og funktionalitet i systemet gør det let at vedligeholde og modificere. Der er lavest mulig kobling i hele systemet som sikre den gode mulighed for vedligeholdelse. Alt dette er lykkes ved at bruge N-tier arkitektur for systemet.

# Litteratur

[1] Nikolay Ashanin. Quality attributes in software architecture. https://hackernoon.com/quality-attributes-in-software-architecture-3844ea482732, 2018.

[2] developer.mozilla.org. Fetch api. https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API, 2019.

[3] Mehdi El Gueddari. Managing dbcontext the right way with entity framework 6: An in-depth guide. https://mehdi.me/ambient-dbcontext-in-ef6/, 2014.

[4] Alf Henderson. Vue cookie. https://www.npmjs.com/package/vue-cookie, 2017.

[5] Katashin. Vuex. https://vuex.vuejs.org/, 2019.

[6] Evan You. Vue guide. https://vuejs.org/v2/guide/, 2019.

[7] Evan You. Vue router. https://router.vuejs.org/, 2019.