DESIGNDOKUMENT

PatientCare – Patientkald med tilknyttet årsag

Indholdsfortegnelse

1 Indledning

Læsevejledning

Designdokumentet har til formål at give en detaljeret beskrivelse af hvordan de tekniske krav skal løses og henvender sig til udviklere af systemet.

Designdokumentet er delt op i to dele: *Systemarkitektur* og *Design*. I *Systemarkitekturen* beskrives forbindelserne mellem systemets moduler og opbygningen af hvert modul. I *Design* beskrives implementeringen af de funktionelle krav der blev specificeret i *Kravspecifikationen* og hvordan de enkelte moduler udfører operationer.

Designdokumentet indeholder følgende afsnit under *Systemarkitektur*:

- Overordnet systemarkitektur
- Deployment view
- Sekvensdiagrammer for PatientCare
- 3-lags arkitektur

Dernæst beskrives arkitekturen for de enkelte moduler: PatientApp, PersonaleApp og AdminApp.

Designdokumentet indeholder følgende afsnit under *Design*:

- PatientApp
- PersonaleApp
- AdminApp
- Database
- WebAPI

PatientCare systemet er opdelt i 3 moduler:

- PatientApp
    - Installeres på en smartphone der benyttes af en patient
    - Er udviklet til Xamarin
    - Kommunikerer med et WebAPI via http protokollen og det som sendes og modtages er i json-format

- PersonaleApp
    - Installeres på en smartphone der benyttes af personalet
    - Er udviklet til Android
    - Kommunikerer med et WebAPI via http protokollen og det som sendes og modtages er i json-format

- AdminApp

  - Er en webapplikation der administreres af en administrator af PatientCare systemet
  - Er udviklet i ASP.NET MVC
  - Kommunikerer med et WebAPI ved brug af http protokollen.

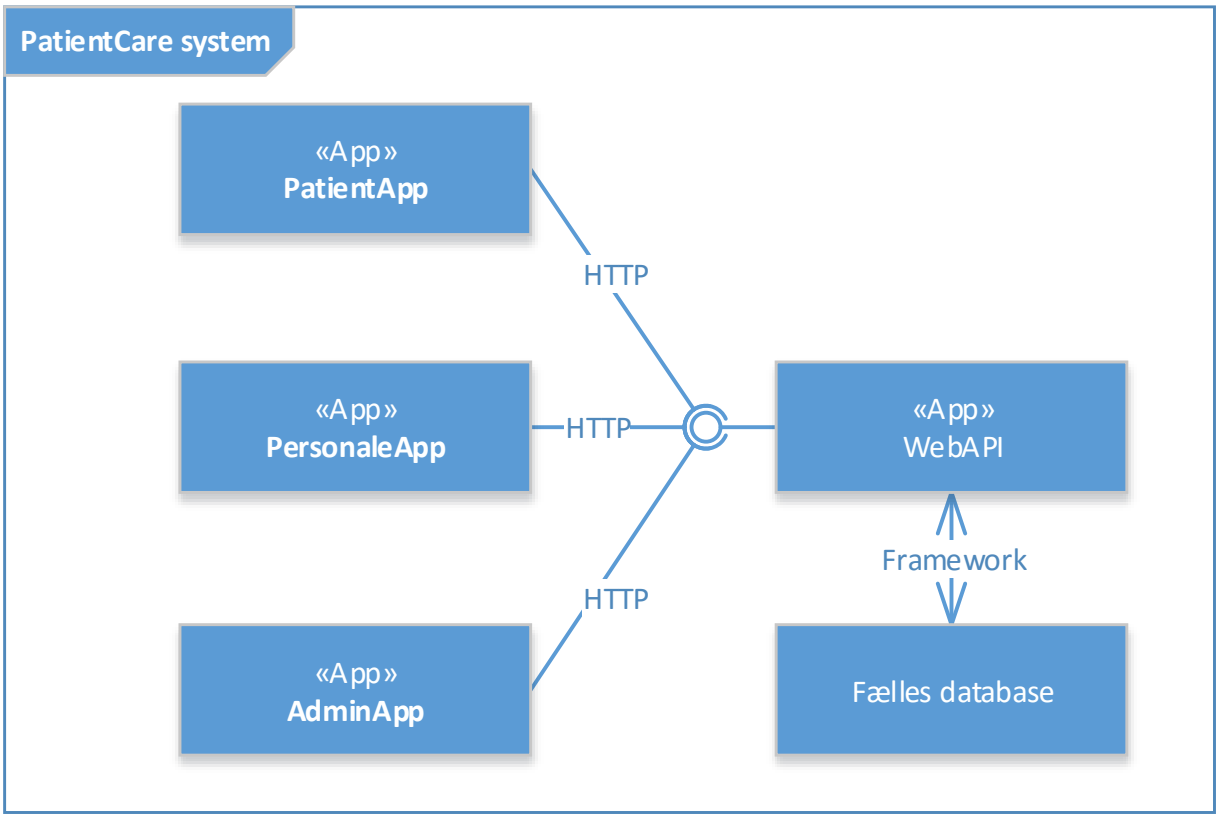
Modulerne kommunikerer med hinanden gennem et WebAPI.

- WebAPI
  - Er et API der fungerer som integrationen mellem alle moduler og er den eneste vej ind til databasen
  - Er udviklet med ASP.NET

2 Systemarkitektur

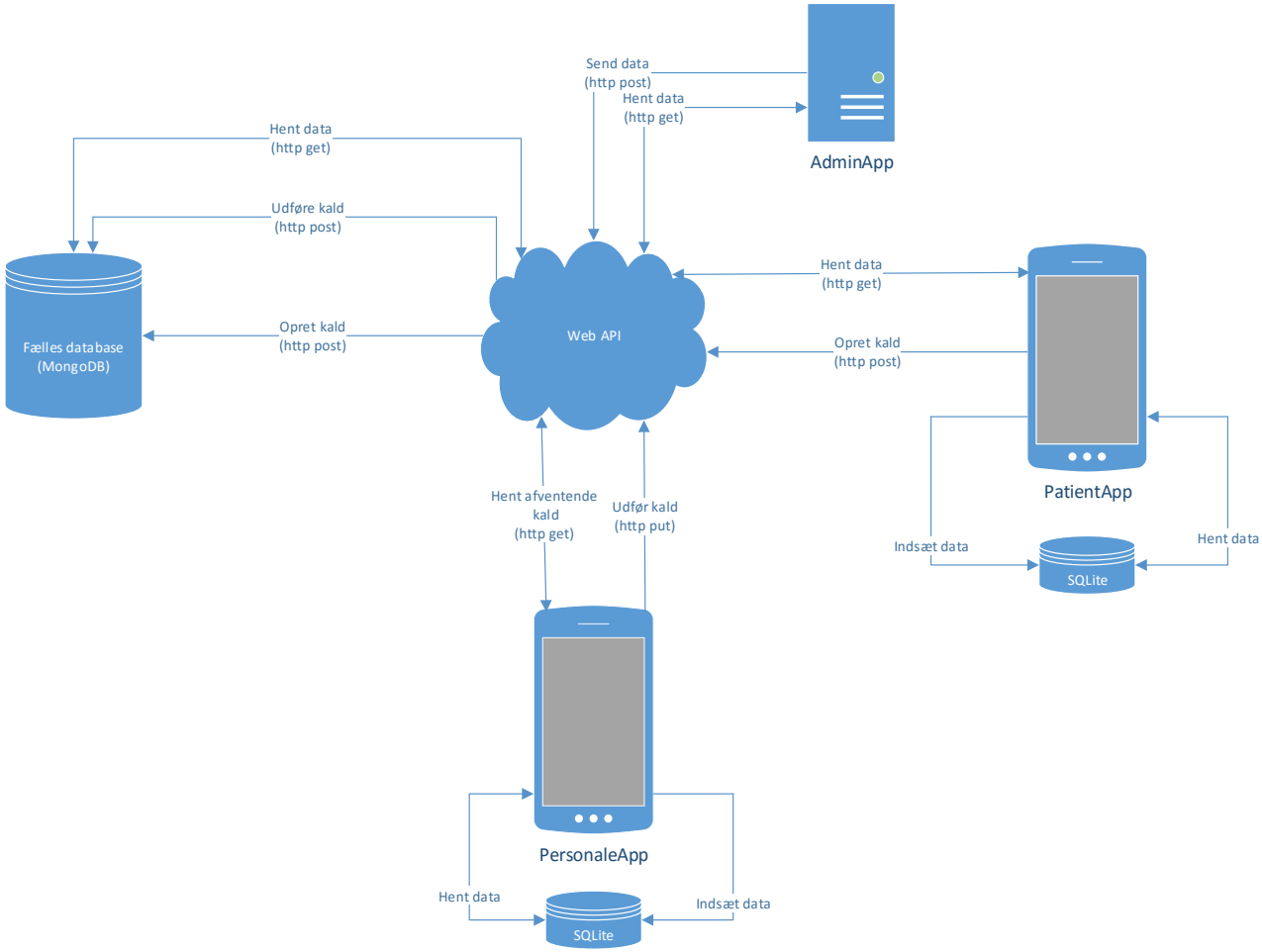
Dette afsnit beskriver kommunikationen mellem PatientCare's subsystemer, først overordnet og efterfølgende detaljeret for de enkelte subsystemer.

Overordnet systemarkitektur



Figur 1 - Overordnet systemarkitektur

Figur 1 viser den overordnede systemarkitektur for PatientCare systemet samt hvilket format der benyttes ved kommunikation mellem de forskellige subsystemer. Det ses, at der benyttes json imellem WebAPI'et og de tre moduler, PatientApp, PersonaleApp og AdminApp. WebAPI'et har forbindelse til MongoDB via MongoDB frameworket.



Figur 2 Diagram for den overordnet kommunikation mellem modulerne for PatientCare

I figur 2 ses den overordnet kommunikation mellem WebAPI og PatientApp, PersonaleApp, og AdminApp.

PatientApp'en sender et json objekt til WebAPI'et som består af følgende parametre:


```
{
   "PatientCPR":"1111111118",
   "Category":"Smerter",
   "Choice":"Efter operation",
   "Detail":"Moderate",
   "CreatedOn":"12:37 PM"
}
```


Figur 3 - Json objekt som sendes fra PatientApp'en til WebAPI'et.

Json objektet bliver deserialiseret til et konkret objekt med json.net. Dette konkrete objekt bliver brugt rundt ind i PatientApp'en og persisteret til den lokale database sammen med det id, der blev returneret fra http response message fra WebAPI.

PersonaleApp benytter sig af en service, der gør brug af en pull-funktionalitet til at hente det patientkald fra WebAPI'et, som netop er blevet sendt til fra PatientApp'en. Der refereres til "Bound Service(Modtag kald)" i designafsnittet for detaljeret beskrivelse af, hvordan denne service fungerer.

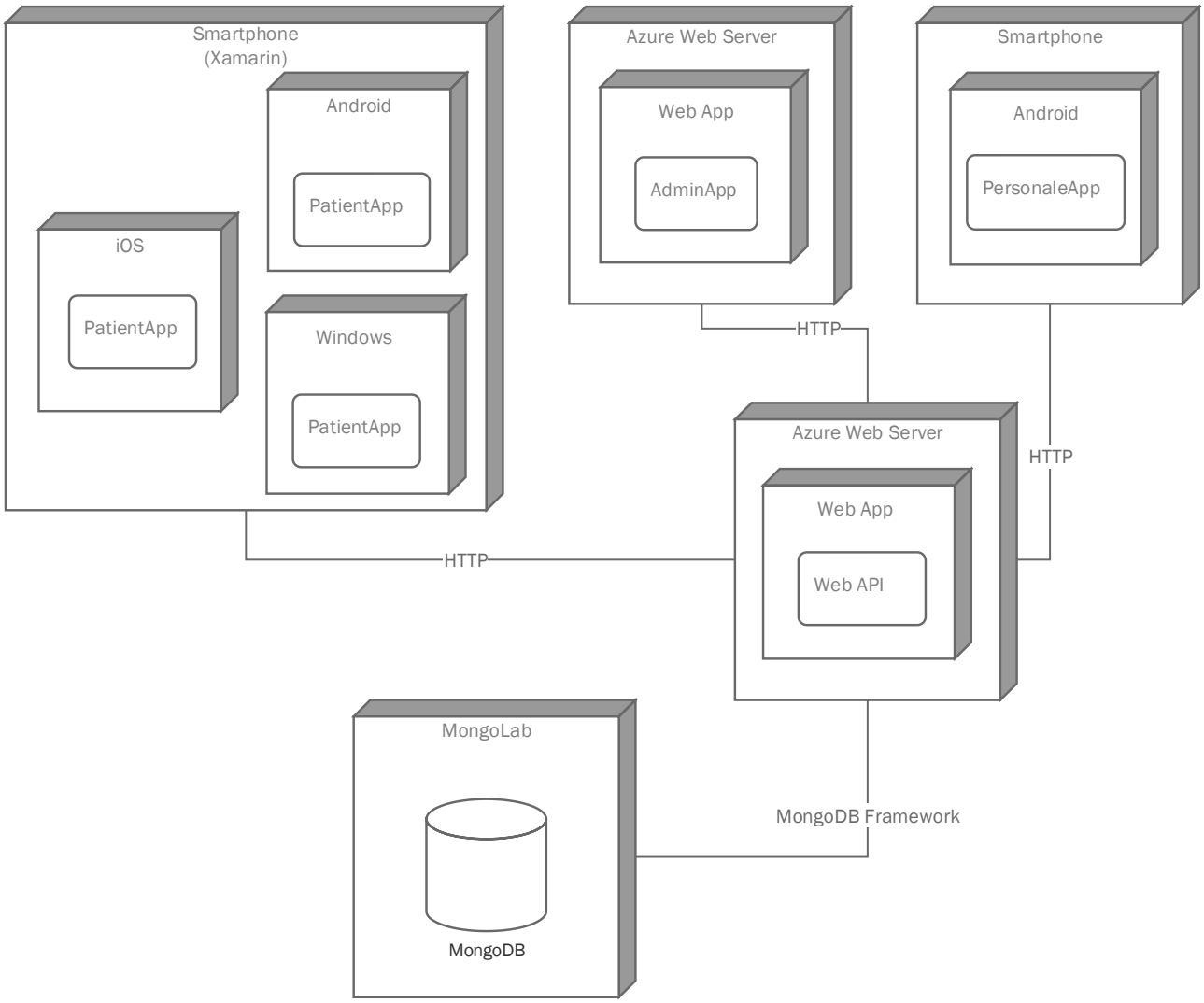
Det som modtages er også et json objekt med følgende parametre:


```
{
   "_id":"564c62e34ca8e93380157bdf",
   "PatientCPR":"1111111118",
   "PatientName":"Louise Andersen",
   "Room":"Stue 2",
   "Bed":"1",
   "Department":"Gyn-obs",
   "Category":"Smerter",
   "Choice":"Efter operation",
   "Detail":"Moderate",
   "CreatedOn":"12:37 PM",
   "ModifiedOn":"Wednesday, November 18, 2015 11:38:14 AM",
   "Status":1
}
```


Figur 4 - Json objekt som PersonaleApp'en modtager fra WebAPI

De parametre som ses i json objekterne er bestemt ud fra hvad PersonaleApp'en forventer, og hvad der i den forbindelse skal bruges til at blive vist på devicet. Id'et som ses i overstående json objekt er det samme id som PatientApp'en har og dermed opnås der en vis konsistenthed i hele PatientCare systemet, fordi det kald der sendes kan identificeres overalt i PatientCare og på den måde kan refereres til, når kaldet eksempelvis skal udføres eller fortrydes på et senere tidspunkt i systemet. Det skal dog bemærkes at de viste data ikke er rigtigte data men er mock data, der er lavet til formålet.

Deployment view



Figur 5 - Deployment View

Figur 5 er et deployment diagram over PatientCare systemet. Det viser systemets fysiske domæne, hvor softwaren ligger, hvilke dele der kommunikerer sammen, og hvilke protokoller der bliver benyttet ved kommunikation. Det vides ikke hvordan MongoDB frameworket fungerer, og derfor ikke specificeret.

PatientApp'en er udviklet i Xamarin og kører på en smartphone som har henholdsvis iOS, Android eller Windows. Dog er der i dette projekt kun lagt vægt på implementering af iOS version 8.4 for hurtigst muligt at få vist det konceptuelle af projektet.

WebAPI'et kører som en web App på en Azure web server som har det formål at være integrationen mellem systemets subsystemer og er den eneste vej ind til databasen. WebAPI'et udstiller en række tjenester som PatientCare's subsystemer benytter sig af.

PersonaleApp kører på en smartphone som kører android version 4.0 eller nyere. PersonaleApp er den applikation personalet på hospitalet bruger til at håndtere patientkald med årsag.

AdminApp er en webapplikation der kører som en Azure web App på en Azure web server. AdminApp er den applikation, en administrator af PatientCare systemet, bruger til at konfigurere de forskellige valgmuligheder en patient kan vælge at tilknytte et kald til på PatientApp'en.

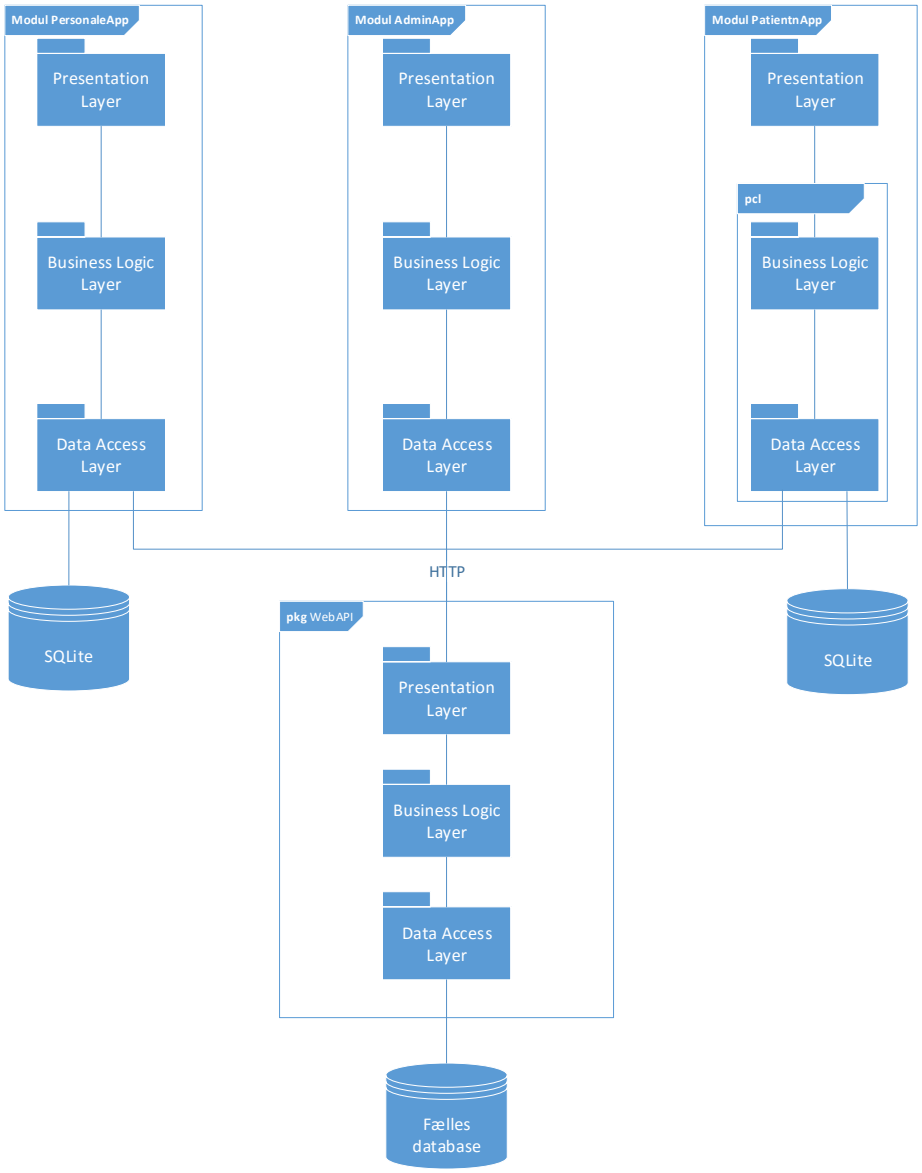
PatientApp, PersonaleApp og AdminApp kommunikerer med WebAPI'et ved brug af http protokollen.

MongoDB er systemets fælles database som står for at gemme alle data der bliver sendt frem og tilbage i systemet. Databasen bliver hostet på mongolab.com og kommunikerer med WebAPI'et gennem MongoDB frameworket. WebAPI'et er databasens eneste tilgang til resten af systemet.

Trelagsarkitektur

Arkitekturen i systemets moduler er lavet ud fra trelagsmodel tankegangen, hvor de tre lag er: *presentation layer, business logic layer* og *data access layer*. Dette gør det muligt for udvikleren at fokusere på et område ad gangen, når kode skal implementeres, da de tre lag så vidt muligt holdes adskilt.

Presentation layer er det lag brugeren interagerer med og håndterer modtagelse og præsentation af data. *Business logic layer* er det lag der håndterer udvekslingen af data mellem *presentation layer* og *data acces layer*. Når brugeren interagerer med App'en kaldes der funktioner i *business logic layer*. F.eks. når en patient indtaster CPR-nr I PatientApp vil CPR valideringen ligge I dette lag.
Data acces layer håndterer udvekslingen af data til det sted data lagres. Trelagsarkitekturen for hvert modul ses på figur 6.



Figur 6 - PatientCare systemet opdelt i 3-lags arkitekturen

PatientApp

På baggrund af Xamarin platformen skal der bygges en systemarkitektur for PatientApp for at fastlægge byggeklodserne der tilsammen udgør en funktionel app. Arkitekturen for PatientApp er også vigtig for PatientCare systemet, da den skal være åben for nye ændringer og fremtidigt arbejde. Det indebære også evt. nye udviklere der skal forstå PatientApp'en og dens kommunikation med resten af PatientCare systemet og betydningen heraf.

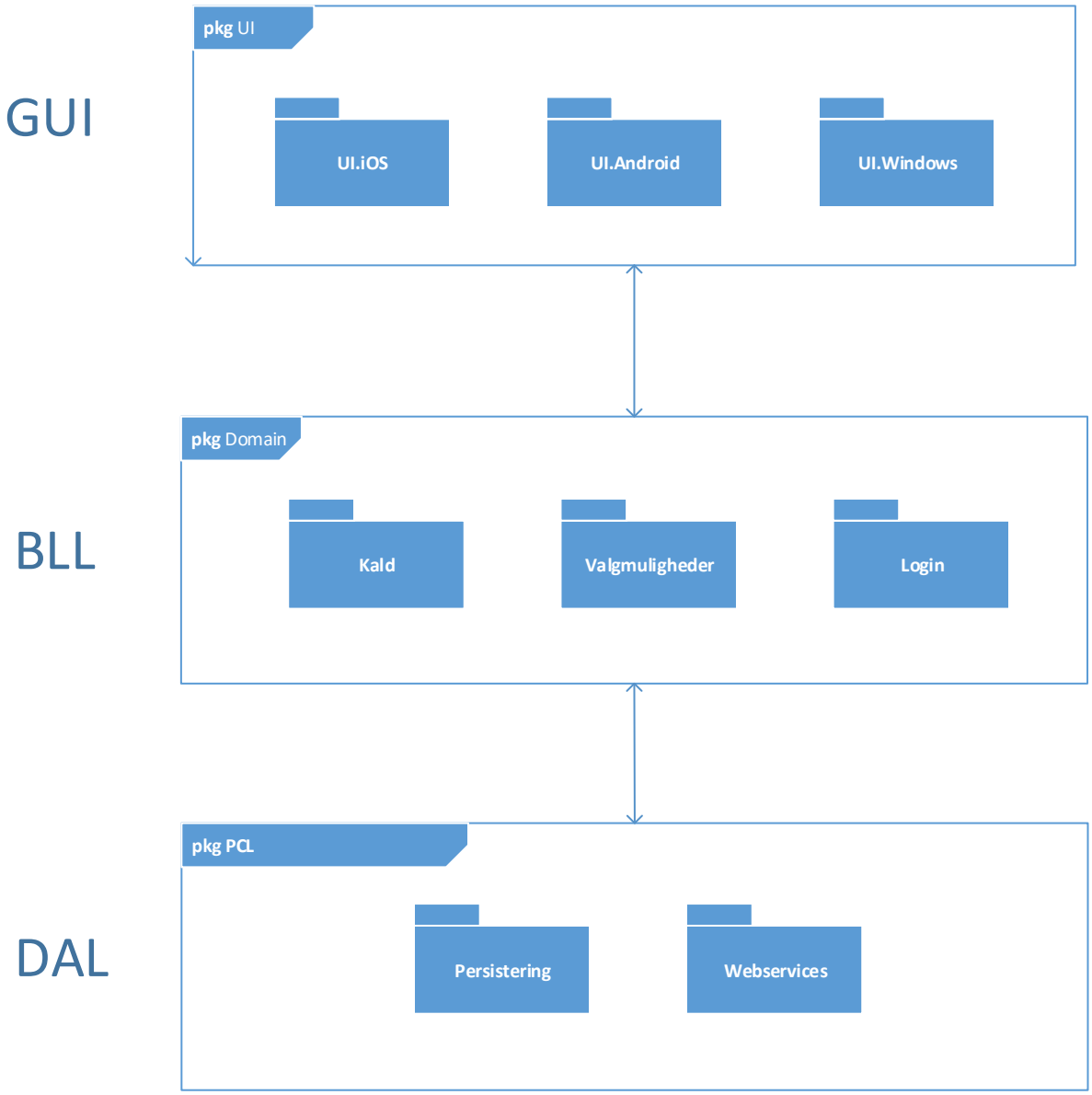
Da den tradionelle Xamarin løsning allerede sætter nogle tanker i gang omkring arkitekturen, er det alligevel vigtigt at forstå, hvad denne løsning og dens indre struktur helt præcist består af. For den tradionelle Xamarin løsning, er brugergrænsefladen platformsafhængig pga. de hver især har deres API og bibleoteker til at kunne implementere f.eks. en tabelliste. På iOS laves en tabel via en UITableViewController og et UITableView og på Android laves det med en Adapter og et ListView. Men domænet og baggrundslogikken er den samme. F.eks. når listen af valgmuligheder blive indlæst fra lokal databasen. Backenden dvs. alt tilgangen til en lokal database og forespørgsler til omverdenen kan via den tradionelle Xamarin løsning også være den samme. Det kræver blot en PCL.

For at kort opsummere kan den tradionelle Xamarin løsning byde på:

- Forskellig implementering af brugergrænsefladen.
- Fælles logik og kommunikation til omverdenen, heraf http request til en webserver.
- Fælles backend til at persistere data.

Package diagram

De tre overstående punkter lægger op til en arkitektur beskrevet i figur 6 over den overordnet arkitektur tredelt i et GUI, BLL, DAL lag. Fordelen ved at have dette, er, at man har "separation of concerns", og det reducer også koblingen og afhængigheder. Derudover giver det også et potentiel "reuse". Det betyder f.eks. at de nederste lag nemt kan blive brugt i andre applikationer. Og da der skal laves en mobil applikation til iOS, Android og Windows vil dette være til stor gavn, fordi data'en kan tilgås fra samme lag. De tre lag er beskrevet i følgende diagram:



Figur 7 - Package diagram for PatientApp

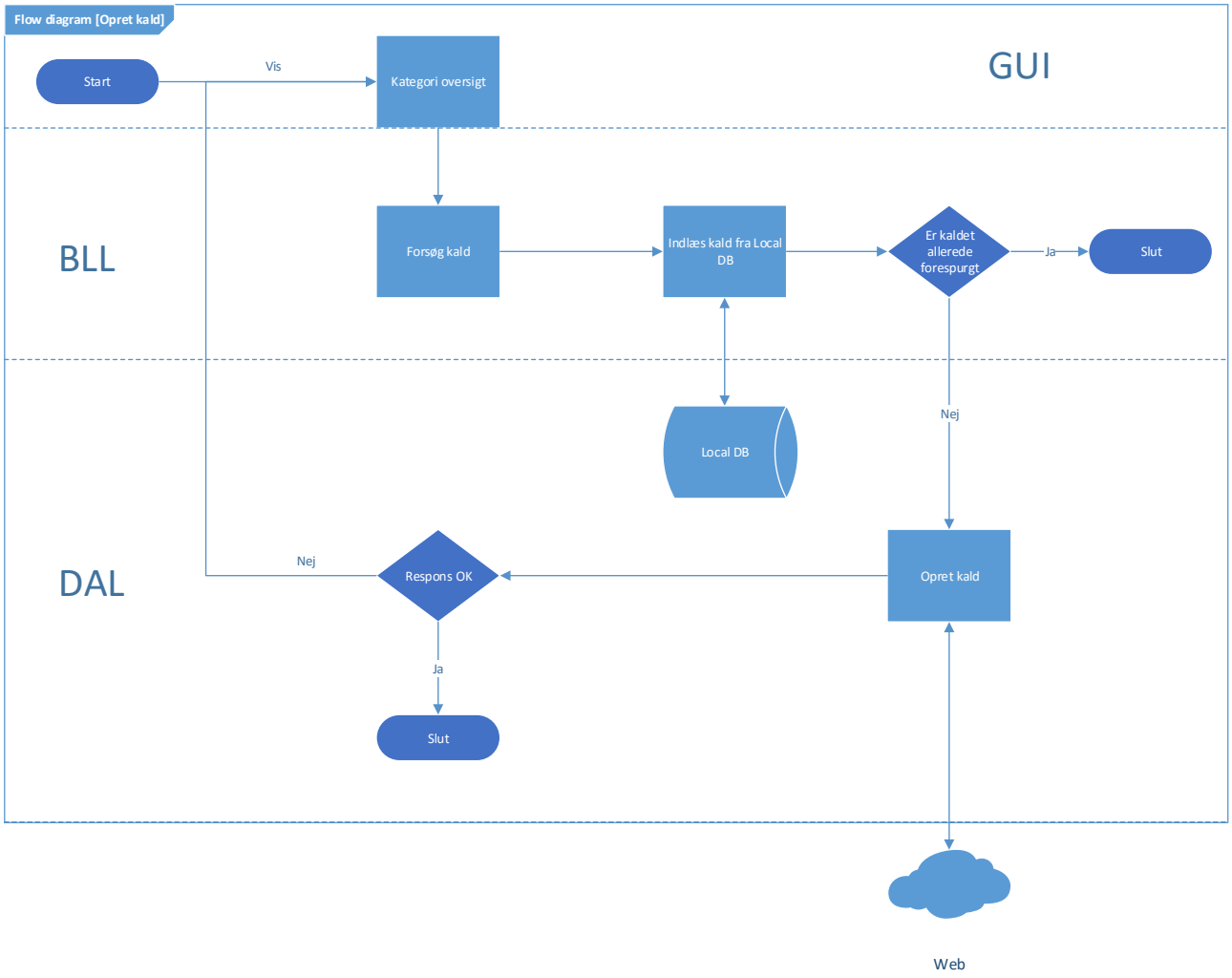
GUI (Også kaldt for PLL – Presentation Logic Layer) er laget, der indeholder en række forskellige pakker af forskellige API'er og biblioteker der sammen med tilhørende logik udgør en brugergrænseflade, som patient kan se. Dette lag vil blive vist forskelligt afhængig af hvilken platform, PatientApp'en er installeret på.

BLL (Også kaldt for Business Logic Layer) er laget, der indeholder alt logikken for PatientApp'en med udgangspunkt i use casene, som patienten vil kunne interagere med. Patienten har f.eks. som mål at oprette et kald og den logik, der sørger for dette ligger, som det ses i figur 7, i en pakke kaldt "Kald".

DAL (Også kaldt for Data Access Layer) er laget, der indeholder pakker som simplificere adgangen til data, persisteret i en database samt kommunikationen til omverdenen. F.eks. vil pakken "Persistering" som i figur 7 returner en reference til et objekt med attributter i stedet for en række med felter fra en database tabel. Dette kan f.eks. være et objekt der indeholder attributter omkring en valgmulighed: Drikke som kategorien,

Kaffe som typen og Mælk som detaljen. Dette lag skjuler dermed kompleksiteten af den underliggende datalagring fra omverdenen og kommunikationen indimellem.

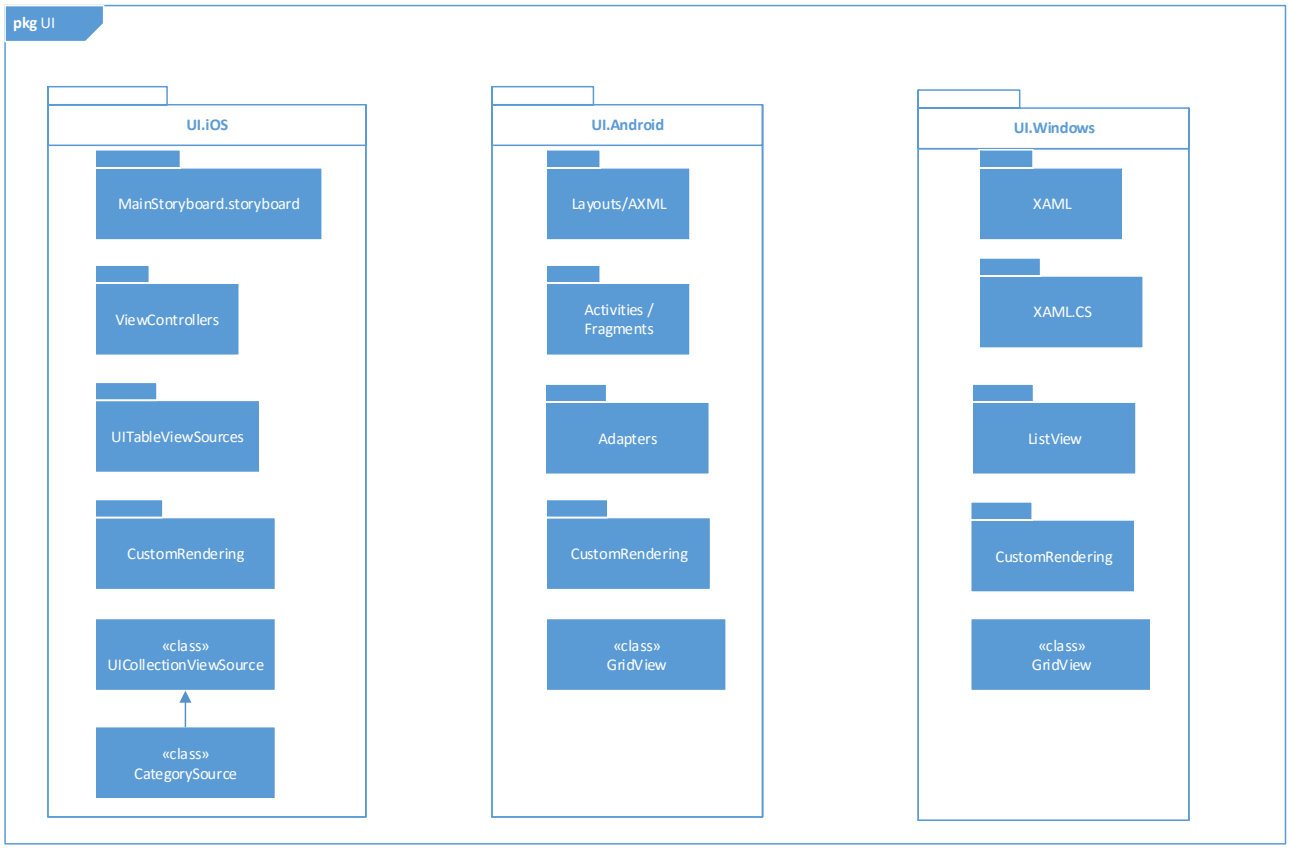
Med udgangspunkt i at patient opretter kald, kan flowet for kommunikationen mellem de tre lag beskrives i følgende diagram:



Figur 8 - Flowdiagram Opret kald

Som start får patienten vist kategorioversigten på brugergrænsefladen. Patienten vælger dernæst en årsag til et kald og trykker på en control i GUI'en der udløser et event. Dette event benytter sig af relevant logik i BLL laget for derefter at lave et tjek på lokal databasen. Kaldet til lokal databasen sker i DAL laget hvor der tjekkes på om kaldet allerede er forespurgt. Hvis kaldet ikke allerede er forespurgt, oprettes kaldet ved at lave en forespørgsel på WebAPI, som giver et svar tilbage. Er svaret "OK" er casen slut, ellers får patienten vist kategorioversigten og kan prøve igen.

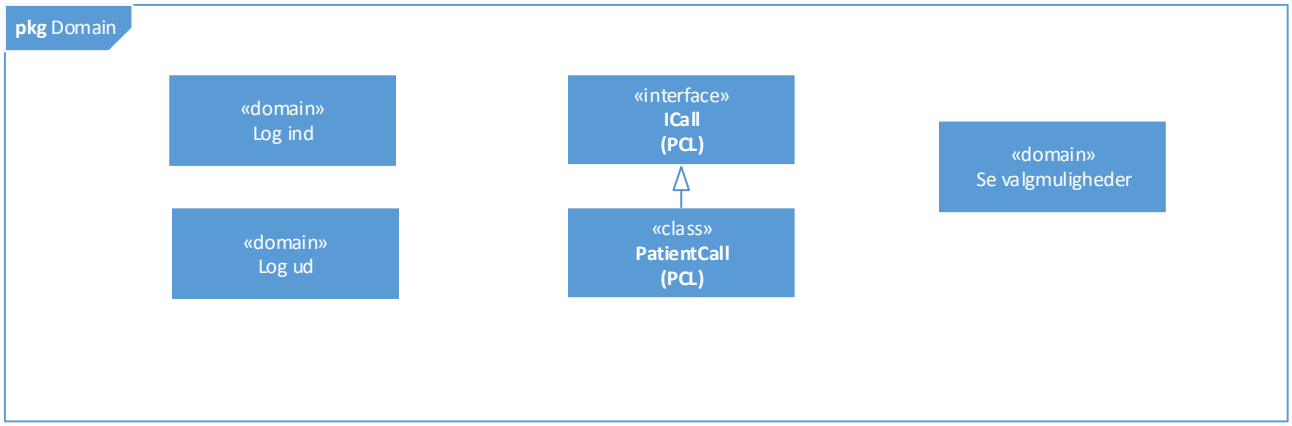
Som beskrevet i figur 8 kan pakkerne i hvert lag pakkes ud til nogle flere pakker. GUI laget har en pakke for hver platform men disse pakker indeholder en række forskellige controllers og andet logik til at få vist en brugergrænseflade.



Figur 9 - Package diagram for UI

Hver platform har som sagt sin egne API'er og bibleoteker til at f.eks. fremstille et view. På iOS bruges Storyboards. På Android bruges Layouts/AXML og på Windows bruges XAML. Logikken bag hver view ligger i controllers. På iOS bruges ViewControllers, på Android Activites og/eller Fragments og på Windows Code-Behind c-sharp filer. Derudover er der andre custom renderinger dvs. forskellige måde at customisere en menu knap eller en tabel.

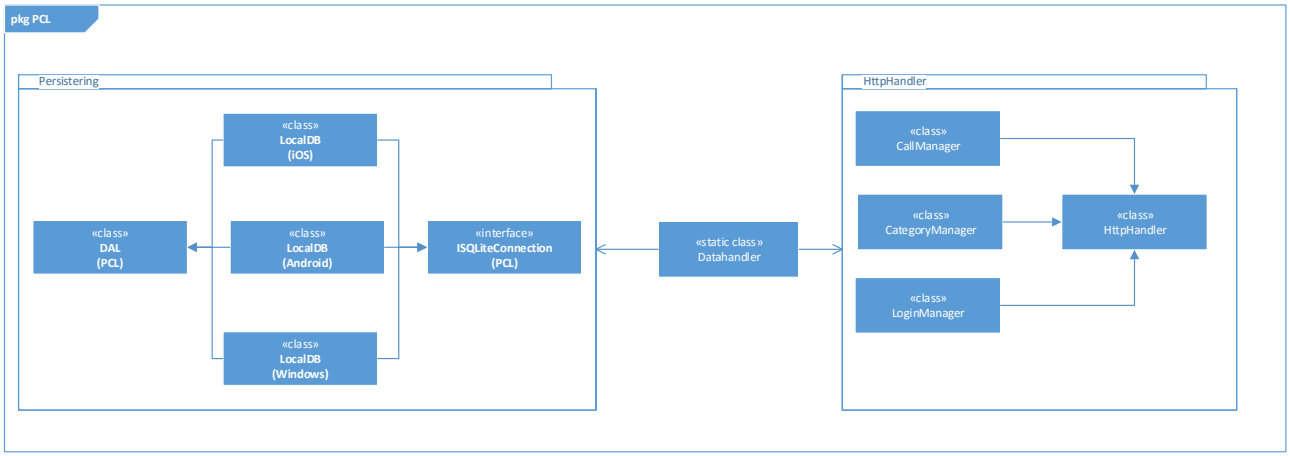
Ligesom GUI laget har pakker med andre pakker, har BLL laget det på samme vis.



Figur 10 - Package diagram for domain

Figur 10 tager udgangspunkt i kaldedomænet, hvor patienten opretter et kald. Det ses at ICall er et interface og at PatientCall er klassen som implementerer dette interface. Årsagen til at ICall er lavet som et interface, er fordi, at det er tænkt som et kald både i form af et portør kald fra Columna systemet fra Systematic men også i form af et patient kald. Det opstiller nogle definitioner for en gruppe af relateret funktionaliteter som en klasse skal implementer. I dette tilfælde relaterede til et kald, i form af et portørkald, patientkald eller noget helt tredje.

DAL laget for PatientApp består af to pakker. En pakke der håndterer persistering af data, og en pakke der står for kommunikationen til omverdenen. Hver af disse er pakket ud til følgende klasser:



Figur 11 - Package diagram for PCL

Metoderne til at oprette tabeller, indlæse og gemme data kan alt sammen tilgås via den samme klasse. Denne klasse ud fra figur 23 hedder LocalDB. Dette kan lade sig gøre fordi, at der findes et API[1] til SQLite, der gør muligt at gemme data i sqlite3 databaser med C# og kan kompileres til at virke på alle platforme. (Mono for Android, .NET, Silverlight, WP7, WinRT, Azure, osv).

Pakken der hedder HttpHandler sørger for at lave http requests ud til et WebAPI, der sender response tilbage i JSON-format, der derefter vil blive deserialiseret til konkrete objekter i DAL laget.

Alt data udtræk fra WebAPI'et og lokal databasen holdes styr på i en klasse kaldt "Datahandler". Derved når man som udvikler har behov for enten at indlæse "mine kald" fra lokal databasen eller sende et kald til WebAPI'et, kan udvikleren finde den rette metode i denne klasse.

Fælles Backend(PCL)

Den traditionelle Xamarin fremgangsmåde indebære brugen af en fælles backend, også kaldt Portable Class Library. Her skal alt PatientApp logikken ligge og tages med overalt. Ud fra de funktionelle og ikke-funktionelle krav, er der blevet stillet følgende punkter op for hvad logik der kan deles i PCL'en, og hvad hver platform skal implementer:

PCL funktioner:

- Indlæs/gem data herunder valgmuligheder og mine kald
- HTTP requests til og fra WebAPI herunder Object Relationel Mapping(ORM)
- CPR validering via WebAPI

[1] https://github.com/oysteinkrog/SQLite.Net-PCL

- Andet logik (klasse til at pakke valgmuligheder til et kald, der senere skal persisteres)

Hver platform skal implementere:

- Egen lokal database
- Filstien til hvor hver database lokalt skal ligge.
- Oprettelse til databasen
- Brugergrænseflade (menuknapper, dialogbokse osv.)

På trods af at hver platform skal have deres egen lokale database, vil database opbygningen og implementeringen være den samme.

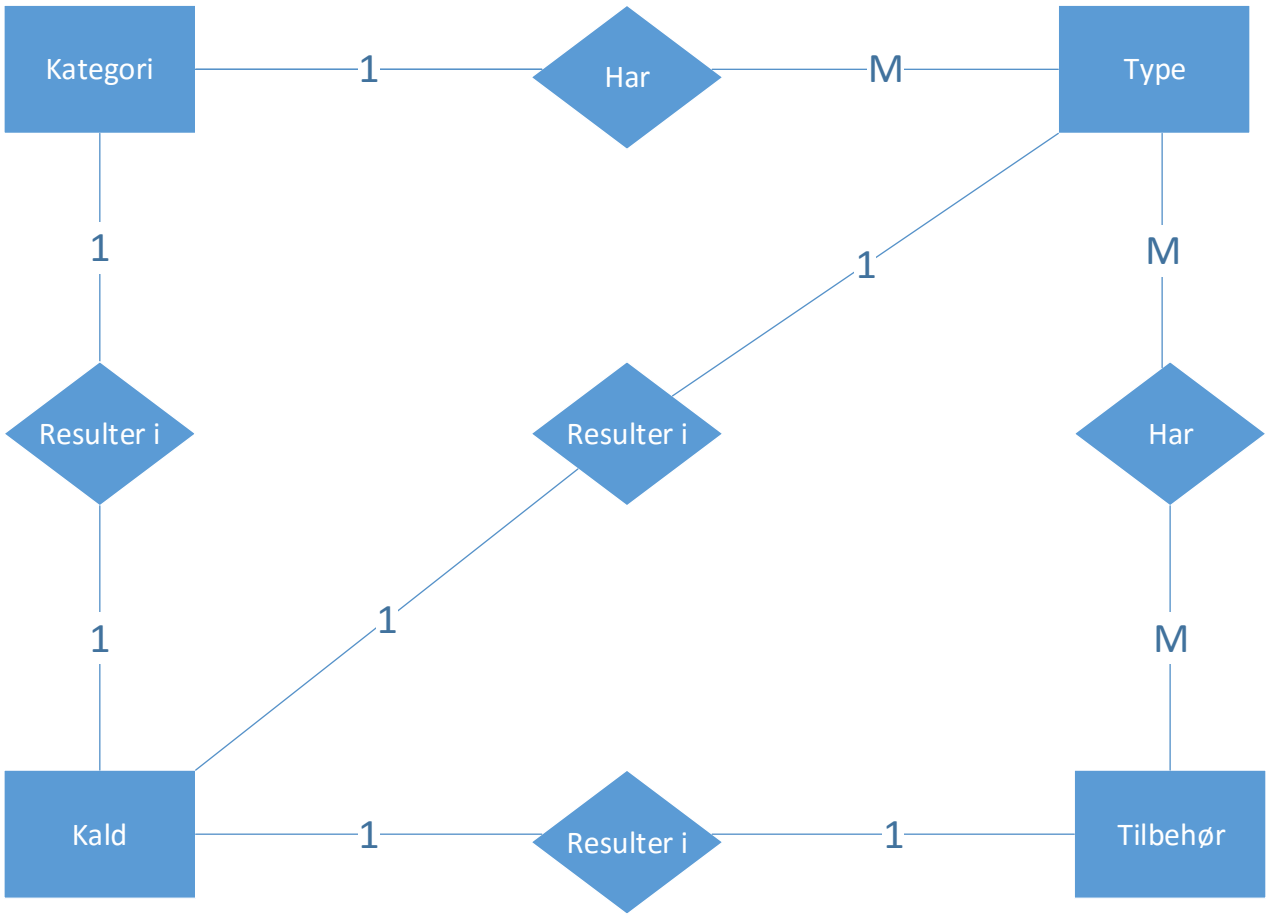
Fælles Database opbygning

Fremgangsmåden i at opbygge en fælles database tager udgangspunkt i objekt-orienterede systemudviklingsmetoder. Først løbes kravene igennem for PatientCare systemet, hvor entiterne identificeres, der i OO svarer til et objekt af bestemt type. Entiteterne er altså et objekt, som vi ønsker at modellere og gemme information om. De entiteter som er relevant for at blive persisteret lokalt på PatientApp'en er opstillet ud fra Use Casene. De er som følger:

- Valgmuligheder
    - Kategori
    - Type
    - Tilbehør
- Mine kald

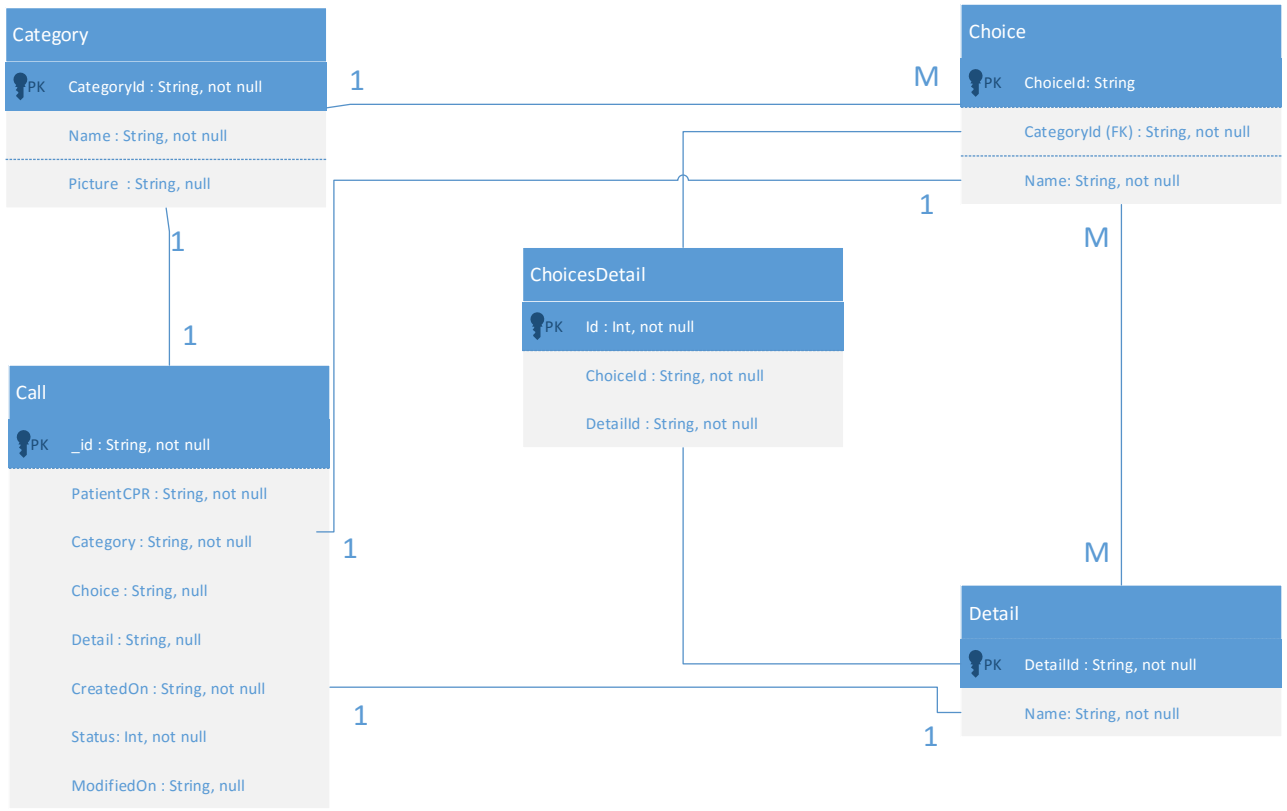
Årsagen til at Valgmuligheder og mine kald skal persisteres er, at man som patient stadig skal have mulighed for at navigere rundt i app'en selvom internettet er ustabilt eller ikke til rådigt. Det er hensigtmæssigt for brugervenligheden og det opretholder gennemsigtigheden, dvs. at app'en stadig kan tages i brug men at funktionaliteten er begrænset.

Entiteterne og deres relationsships opstilles i følgende ER diagram:



Figur 12 - Entity Relationship diagram for fælles SQLite database

Af figur 12 ses det at en Kategori har en-til-mange Typer og en Type har mange-til-mange Tilbehør. Derudover er der en stærk binding mellem Kategori og Type og Type og Tilbehør. Med andre ord, en type skal tilhør mindst én kategori og én tilbehør skal tilhør mindst én type. Hver af disse valgmuligheder, som patient kan vælge, giver en årsag til et kald. Dette er wrappet i en entitet for sig selv kaldt ”Kald”. Relationerne er fundet ud fra en logisk tankegang men også på baggrund, hvad der er mest hensigtsmæssigt i forhold til hvad, en Patient kan vælge af serviceydelser.
Alt data der er relateret til en entitet er vist via attributter. En attribut er en property på en entitet og dens værdi kan findes ud fra dens domæne. Domænet er dette tilfælde valgmuligheder, herunder kategori, type og tilbehør og hver har en attribut af typen String med et navn på. Dette fører frem til følgende diagram:



Figur 13 – Tabelopsætning

Fremmednøglen for en-til-mange relationsship skal defineres i mange entiteten af relationsshippet. En-til-mange relationsships supporter Lister og Arrays, derfor har f.eks. Category en liste af choices, hvor hver af dem har en fremmednøgle tilbage til Category. Mange-til-mange relationsships kan ikke udtrykkes ved at bruge en fremmednøgle i en af entiterne, fordi fremmednøgler repræsenterer X-til-en relationsships. I stedet kræves en mellemliggende entitet, hvor disse fremmednøgler er erklæret. I figur 13 ses det at ”ChoicesDetail” er dette mellemliggende entitet. Denne vil dog aldrig blive brugt direkte i app’en, og der vil derfor ikke blive oprettet en tabel for den. Entiterne som er defineret her ender ud i klasser, som skal implementeres i en PCL og inkluderes i hver platform, specifikke projekt. Men hvordan får PatientApp’en persisteret data’en fra disse klasser og indlæst igen, når der er behov for det? Dette løses ved at benytte en teknologi som SQLite gør brug af. Her vil det være muligt at mappe data’en fra lokal databaserne direkte til disse klasser helt automatisk. Hvorefter disse klasser kan traversere rundt på tværs af iOS, Android og Windows via PCL’en og benyttes i logikken og vises på brugergrænsefladen i app’en.

SQLite-Extension

For PatientApp’en har projektgruppen valgt at persistere data lokalt i en SQlite database. Da logikken for PatientApp skal bygges på en PCL, skal persistering af data dermed også være af samme fremgangsmåde og bygges i samme PCL. Til dette er der valgt en SQLite-Extension[2], som er en meget simpel ORM der tilbyder alle de relationer databasemodellen har brug for, af et SQLite-net[3] bibliotek. Det er open source, der

[2] https://bitbucket.org/twincoders/sqlite-net-extensions
[3] https://github.com/praeclarum/sqlite-net

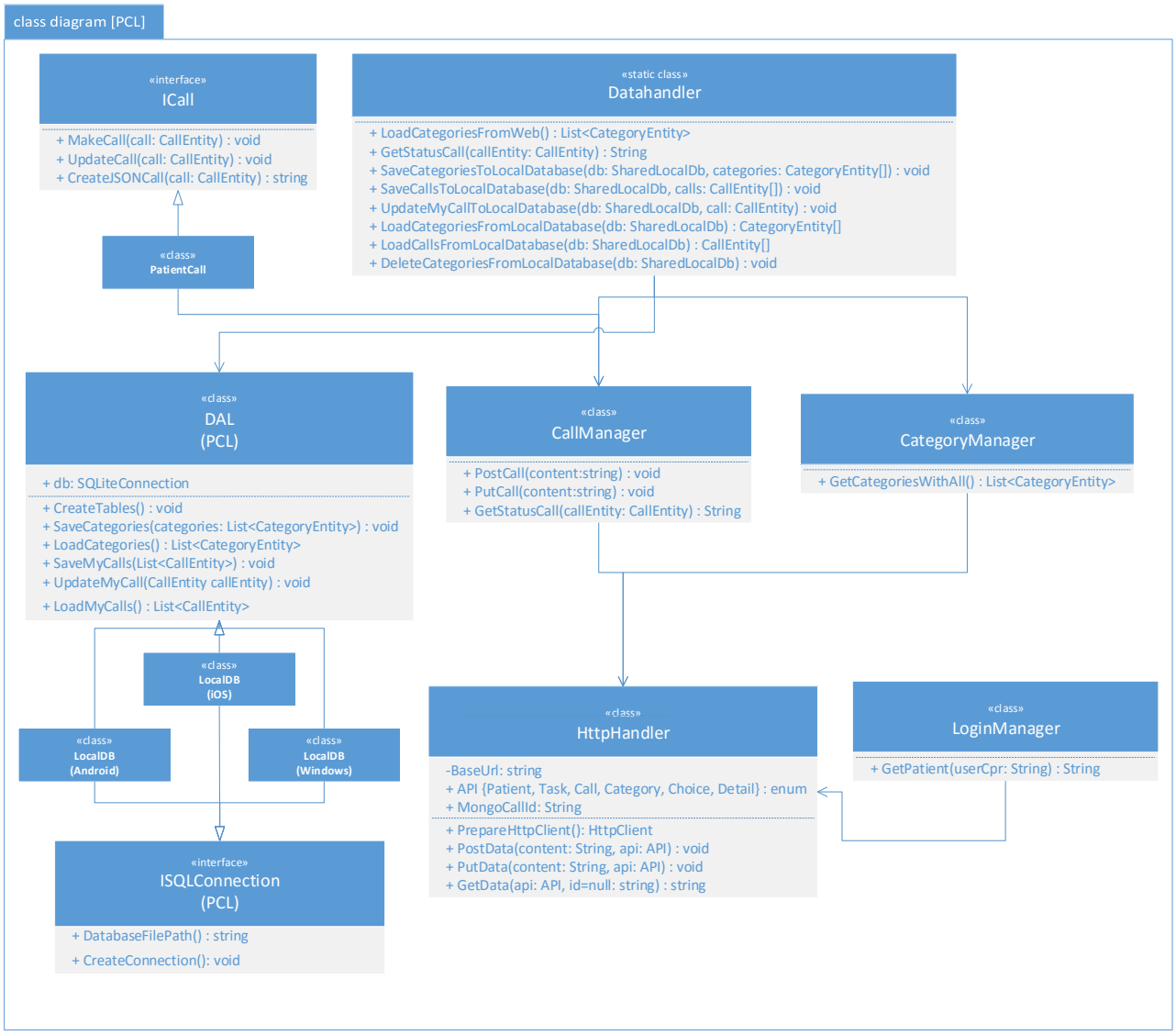
tillader .NET og monoapplikationer at gemme data i SQLite3 databaser. Med SQLite - Net Extension udvides SQLite3's funktionalitet og hjælper bedre udvikleren til at håndtere relationer mellem SQLite-net entiteter. Med SQLite – Net Extension kan man via attributter på sine properties angive relationerne. Man skal ikke oprette tabeller eller kolonner i databasen og derfor har man fuld kontrol over databasens skema til at persistere entiteterne. SQLite-Net Extensions kræver kun, at man angiver fremmednøgler, der anvendes til at håndtere relationer og resten finder SQLite3 den selv ud af.

Et eksempel på brugen af SQLite-Net Extension er metoden *GetAllWithChildren<Category>*. Metoden kigger på alle de relationer der er specificeret i databasemodellen, finder eventuelle fremmednøgler og automatisk fylder properties på entiteten, der er i dette tilfælde er *Category*. Man slipper altså for at skrive queries, som man normalt ville med SQLlite3.

Hver platform implementere en metode til at returnere filstien for hvor SQLite databasen lokalt skal ligge. Samtidig skal hver platform implementere en metode til at oprette forbindelse til SQLite databasen. Dette kræver et interface som implementeres af tre platformsspecifikke klasser med hver af deres SQLiteConnection. I *Data Access Layer* laget ligger der en klasse, der har en række metoder som skal bruges ved indlæsning og persistering af data fra og til SQLite databaserne. Denne klasse skal også bruges i de tre platformsspecifikke klasser.

Klassediagram for PCL

For at beskrive strukturen i PatientApp vises nedstående et klassediagram for PCL'en. Dette viser PatientApp's klasser for PCL'en, deres attributter, metoder og relationen mellem objekterne. Dette klassediagram tager udgangspunkt i figur 11, hvor DAL er beskrevet.

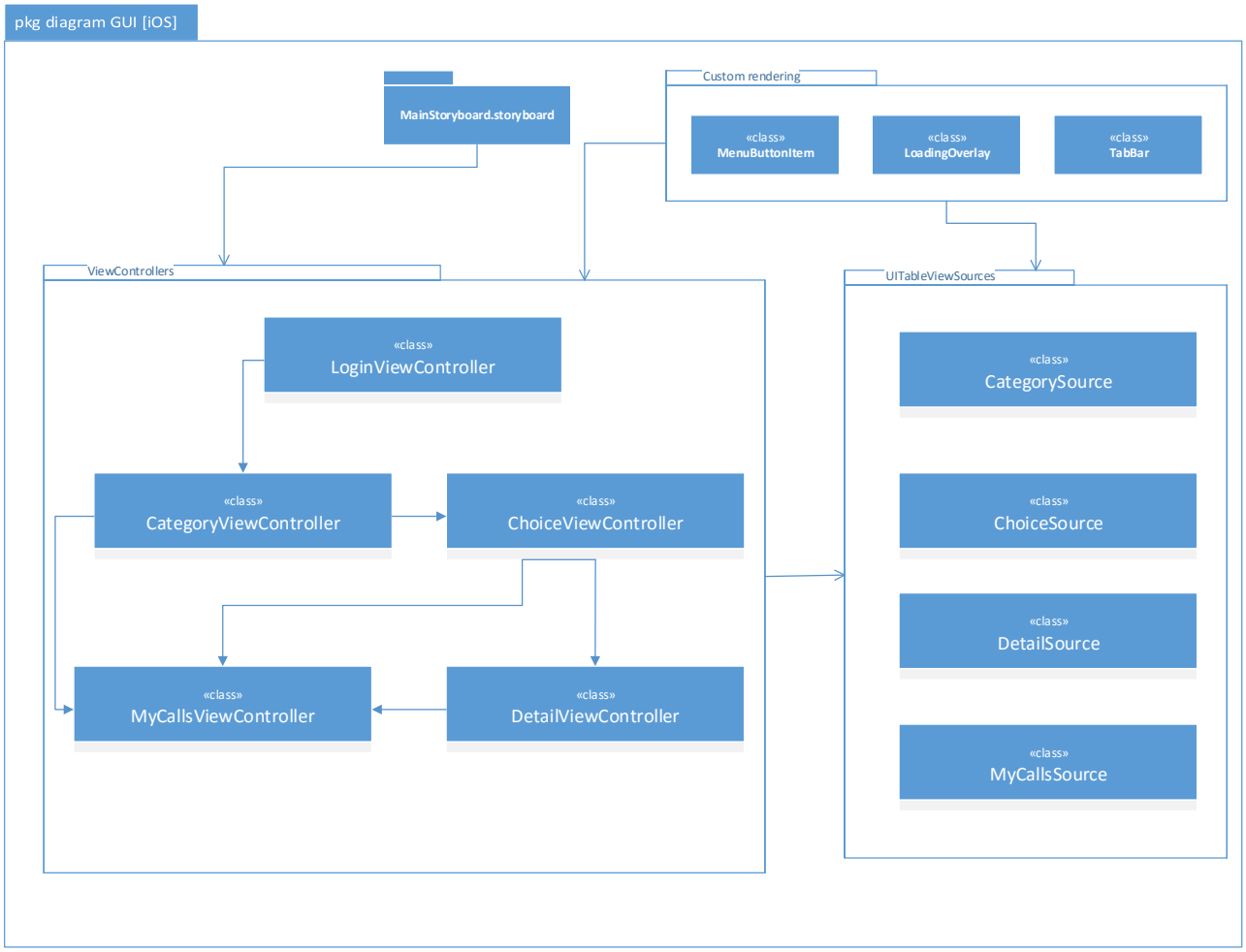


Figur 14 – PCL klasse diagram

Klassen "Datahandler" som kort beskrevet i Figur 11 under DAL laget har en association til CallManager, som håndtere oprettelsen og opdateringen af et kald. Den sørger også for at hente status (Aktiv, Udført, Fortrudt) på et specifikt kald. Ligeledes har "Datahandler" en association til "CategoryManager" som henter alle valgmuligheder fra administrations modulet. Hver service har association til en "HttpHandler" klasse hvor den konkrete implementering af httprequest og httprespons til et WebAPI sker.

Klassediagram for GUI (iOS)

Nedstående viser klassediagram for GUI'en for PatientApp med udgangspunkt i prototypen, som er udviklet til Xamarin.iOS.



Figur 15 - GUI iOS package diagram

På iOS arbejder man med storyboards[4] som er en visuel repræsentation af brugergrænsefladen og flowet af applikationen. Det indeholder en sekvens af alle scener hvor hver scene repræsenterer en viewcontroller og dens views. Disse views vil indeholde bestemte objekter og kontroller som gør at brugeren kan interagere med applikationen. I figur 15 ses det at "MainStoryboard.storyboard" har en association til en pakke med viewcontrollers. Essensen er at hver viewcontroller er tilknyttet et view eller en scene i storyboadet. Disse viewcontrollers er logikken bag viewet og alt håndtering af brugerinteraktion sker her. Det ses at pakken med viewcontrollers består af 5 viewcontrollers:

- LoginViewController – står for at håndtere login, heraf CPR validering og tjek af patient indlæggelse
- CategoryViewController – står for at håndtere visning og valg af kategorier
- ChoiceViewController – står for at håndtere visning og valg af typer tilknyttet en kategori
- DetailViewController – står for at håndtere visning og valg af detailer tilknyttet en type.

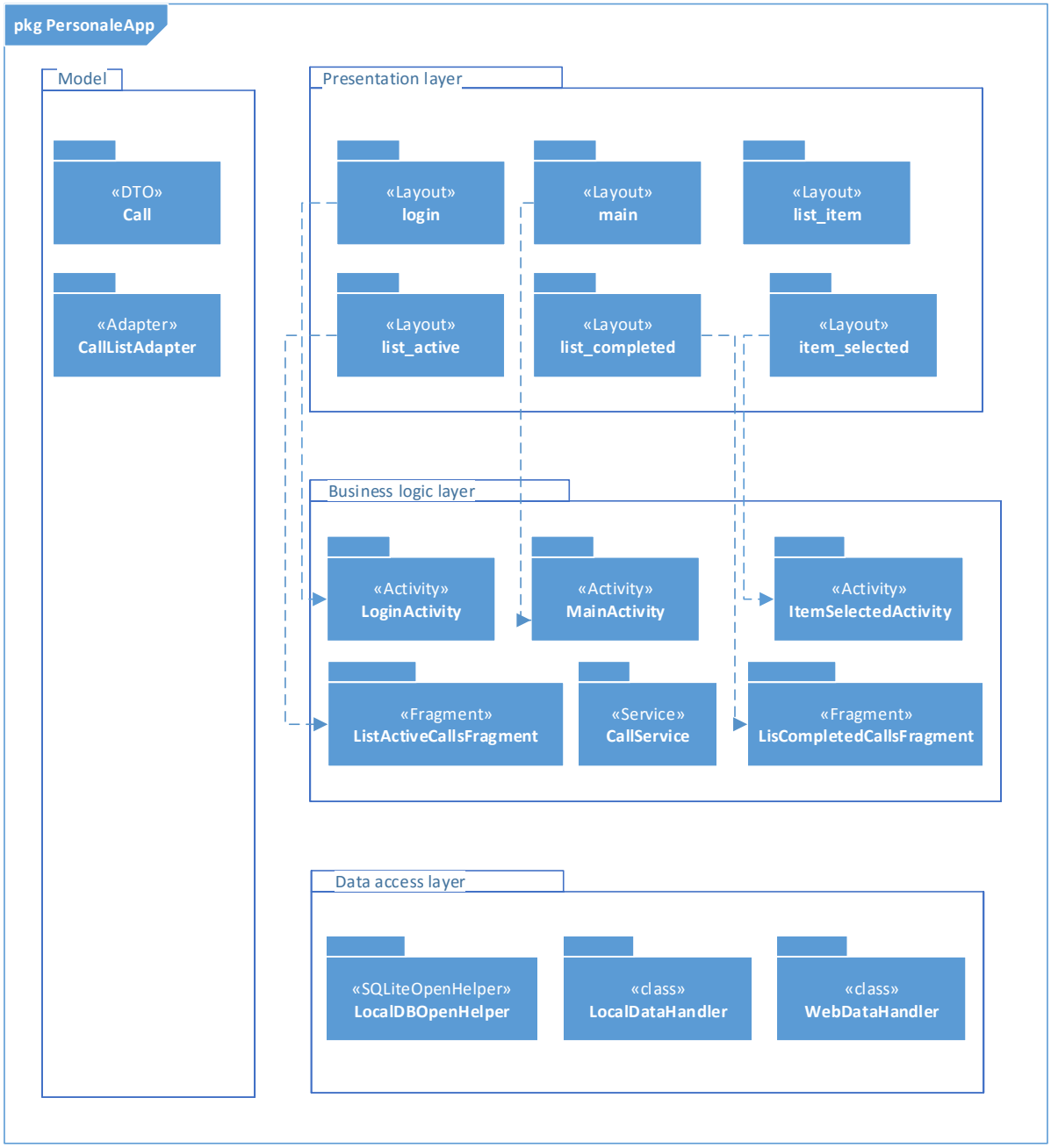
[4] https://developer.xamarin.com/guides/ios/user_interface/introduction_to_storyboards/

- MyCallsViewController – står for at håndtere visning af mine kald og holder styr på status på et kald (om det er aktivt, udført eller fortrudt)

Hver viewcontroller har en association til en ”UITableViewSource” som er en klasse der fungerer som ”code behind” mht. visning af en tabel, for hvor hver valgmulighed bliver vist i.

PersonaleApp

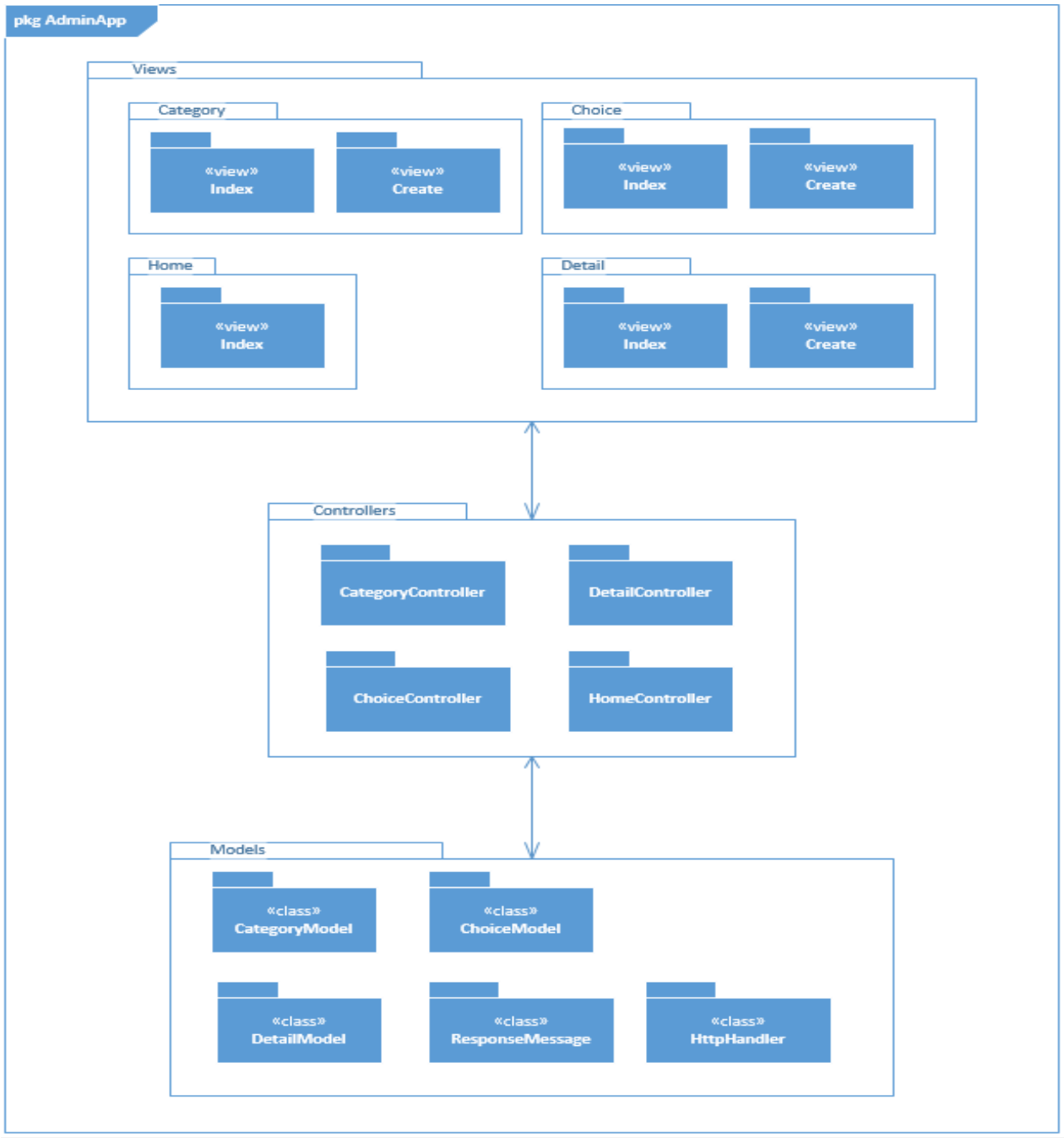
Figur 16 viser package diagrammet for PersonaleApp opdelt efter 3-lags arkitekturprincippet. Præsentationslaget består af layouts som skal definere den visuelle struktur af UI. Business logik laget består af activities og fragments som skal håndtere logikken bag layoutsne og dermed de operationer der aktiveres gennem præsentationslaget. I dette lag skal der også køre også en service, som skal håndtere modtagelse af patientkald uforstyrret af brugerens interaktion med App’en. Data access laget består af klasser der skal sørge for adgang til udveksling af data lokalt på personalets smartphone og globalt med resten af systemet via WebAPI’et. Ud over de tre lag er der også modeller der kan opererer frit mellem lagene. Dette gælder en DTO for Call der skal transportere data for et kald samlet og sikkert gennem App’ens lag og en adapter der skal bruges til at indsætte den rette data i et bestemt view for hvert kald i præsentationslaget.



Figur 16 - Package diagram for personale app

AdminApp

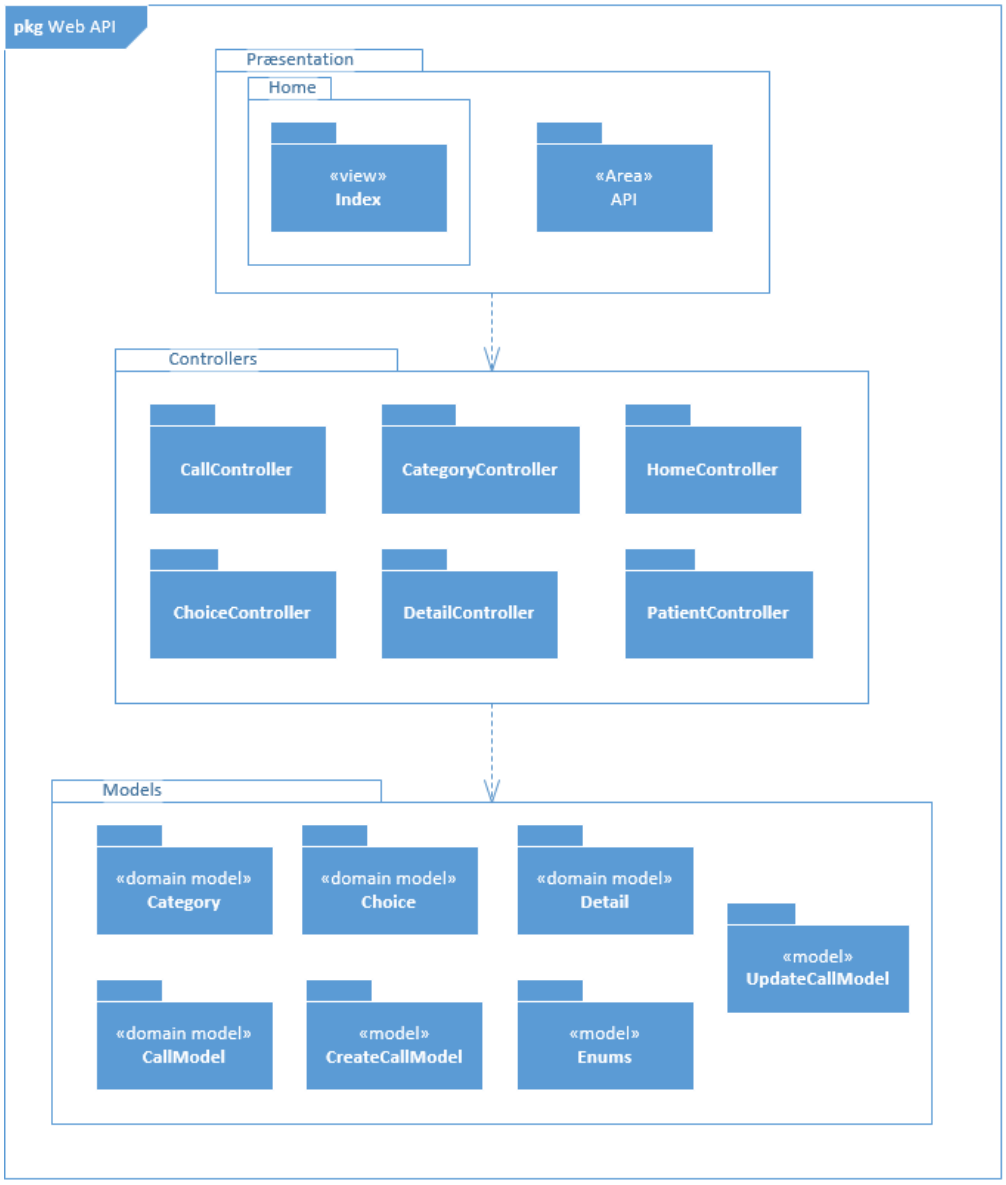
Figur 17 viser trelags-arkitekturen for AdminApp som bygger på MVC strukturen. MVC strukturen består af Views, som er det brugeren bliver præsenteret for og kan interagere med. Controllers, som den logik der tager sig af brugerens interaktion med præsentations lagets views. Det kan fx være form data der bliver sendt til controlleren. MVC strukturen bygger på én controller for hver view pakke. Models, er de modeller som indeholder controller benytter sig af til at vise data til viewet eller gemmet data sendt fra viewet til controlleren. Models, består også af evt. hjælper klasser til fx at sende data til WebAPI'et eller hente data ind fra WebAPI'et.



Figur 17 Pakkediagram for AdminApp der viser en opdeling af pakker i MVC

WebAPI

Figur 28 viser trelags-arkitekturen for WebApp som bygger på MVC strukturen. MVC strukturen består af views, som i denne figur er beskrevet som præsentation. Det er gjort fordi API'et ikke benytter sig af views på samme måde som fx MVC strukturen for AdminApp. Præsentation ses her som værende WebAPI'ets dokumentations side, som beskriver kommunikationen med WebAPI'et. Controllers, som tager sig af http requests til WebAPI'et. Models, som er WebAPI'ets datamodeller som bliver modtaget fra PatientApp eller fra PersonaleApp, eller de modeller som bliver persisteret i databasen.



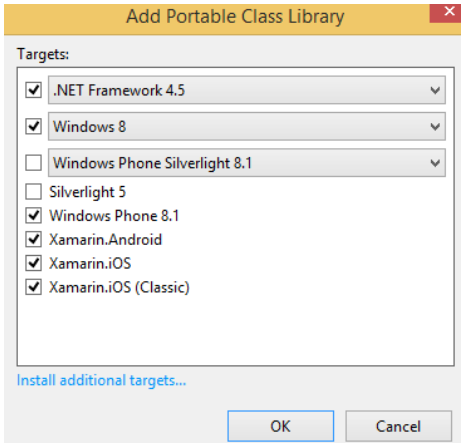
Figur 18 Pakkediagram for WebAPI der viser en opdeling af pakker i MVC.

3 Design

PatientApp

Opsætning af cross-platform projekt

På Xamarins hjemmeside[5] kan man installere et Xamarin-plugin til Visual Studio, hvilket er blevet gjort for PatientCare systemet, da denne IDE er den mest kendte. Xamarin installerer automatisk de nødvendige SDK'er for Android og iOS – dog kræver det, at man har en Mac for at kunne oprette og bygge et Xamarin.iOS projekt. I guiden[6] til at opsætte et cross-platform projekt laves et Portable Class Library projekt. Her bliver man spurgt hvilke targets denne PCL skal henvende sig imod:



Figur 19 Oprettelse af et PCL projekt

Dernæst oprettes et projekt for iOS, Android og Windows. Da prototypen for PatientApp'en med udgangspunkt udvikles til iOS, er der en række udfordringer:

1. En Mac er nødvendig
2. iOS applikationer kan ikke bygges uden Apple's kompiler, og de kan ikke deployes til et device uden Apple's certifikater og code-signing værktøjer.
3. Xamarin.iOS SDK
4. Apple's Xcode(7+) IDE og IOS SDK med den nyeste version.

Dette betyder at Xamarin.iOS projektet for Visual Studio kræver en forbindelse til en netværksforbundet Mac OSX computer for at udføre disse opgaver.

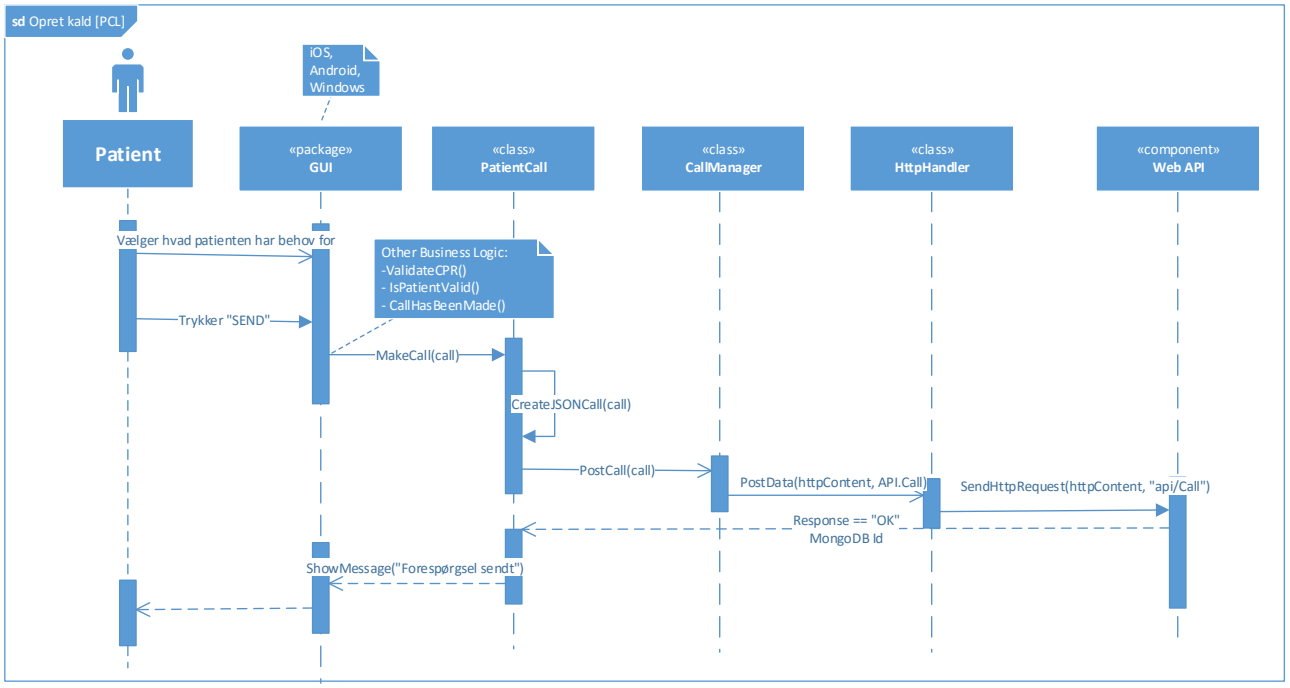
Derudover kræver det en Apple Developer konto for at kunne deploye applikationen til App Store.

Ud over disse restriktioner er det super nemt at oprette og rediger iOS projekter og cross-platform løsninger som inkluderer Xamarin.Android og Xamarin.Windows projekter. For PatientCare systemet udvikles PatientApp'en på en privat, Macbook Pro med fuld mulighed for at oprette, bygge og deploye et Xamarin.iOS projekt.

[5] https://xamarin.com/download
[6] http://developer.xamarin.com/guides/cross-platform/getting_started/visual_studio_with_xamarin/cross_platform_visual_studio_project_setup_walkthrough/

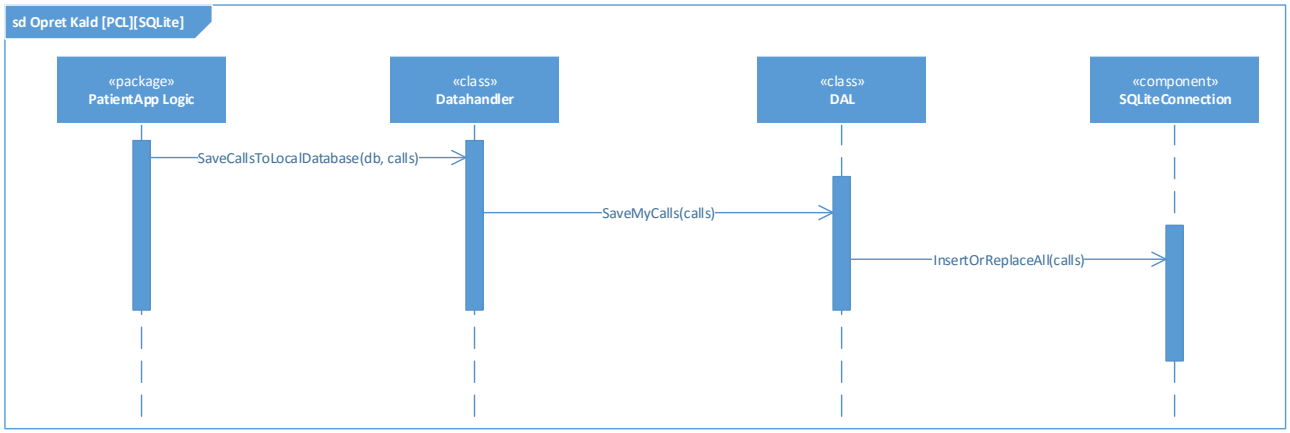
Sekvensdiagram (Opret kald)

For at illustrere hvordan de forskellige lag snakker sammen i PatientApp'en er der taget udgangspunkt i use case 1.3: Opret kald. Kommunikationen mellem disse tre lag vises med et sekvensdiagram på figur 20 der beskriver sekvensen fra en patient opretter et kald til patienten kan se kaldet er oprettet. Brugergrænsefladen (GUI) er en pakke af forskellige komponenter på hver af de forskellige platforme som initialiserer logikken i PCL. Komponenten, WebAPI, er ikke en del af PCL, men er taget med i diagrammet for at vise forbindelsen til resten af systemet.



Figur 20 Sekvensdiagram for Opret kald (PCL)

På figur 21 vises sekvensen for hvordan kaldet persisteres i lokaldatabasen efter det er blevet oprettet. Komponenten, *SQLiteConnection*, er et API til SQLite.Net der er inkluderet i PCL'en.

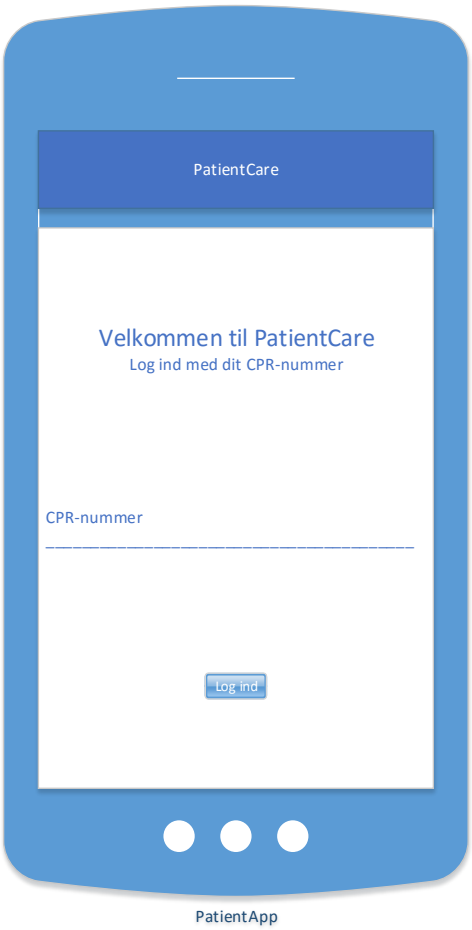


Figur 21 Sekvensdiagrao for Opret Kald (PCL) ved brug af SQLite

Layouts

Efter at backenden nu er beskrevet skal PatientApp'ens layout designes. Projektgruppen har prioriteret det konceptuelle af PatientCare systemet, og derfor er der valgt et meget simpelt men elegant design til PatientApp. Nedenfor ses det første som man som bruger af PatientApp'en vil kunne se.

Layout for log ind side



Figur 22 Layout for log ind side

Grunden til at Patienten som bruger af PatientApp'en skal se en log ind side, er for at han/hun kan identificere sig selv – for så at lade systemet finde ud af resten. Det skal tydeligt fremgå af log ind siden at Patienten skal logge ind med CPR-nummer for at få adgang til app'en. Figuren ovenfor viser et udkast til Log ind siden for PatientApp.

Layout for valgmuligheder

Figur 23 Layout for valgmulighede

Når brugeren er logget ind skal der være en menu for at patienten nemt kan navigere rundt i App'en. Menuen skal bestå af to tabs den ene med teksten *Valgmuligheder,* den anden med eksten *Mine kald. Mine kald* tab'en skal have et lille rødt batch icon, som skal gøre følgende:

- Tælle op hver gang patienten opretter et kald
- Tælle ned hver gang et kald bliver udført
- Tælle ned hver gang et kald bliver fortrudt

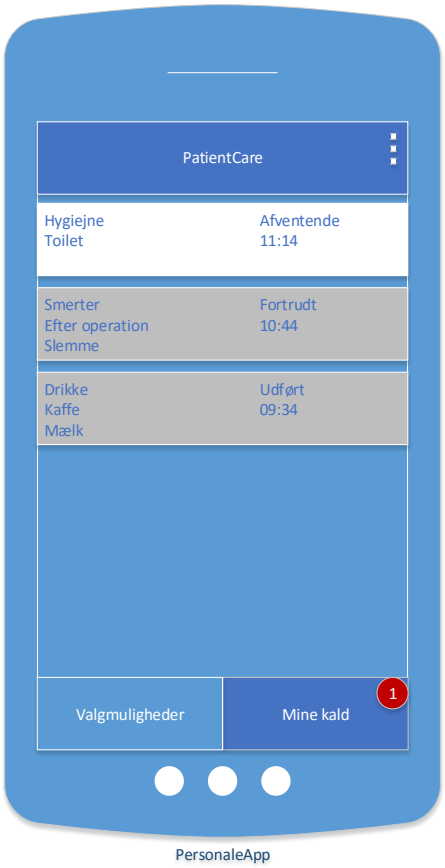
På den måde bliver det tydeligere for patienten hvor mange afventende kald patienten har.

Der skal være et layout når tab'en *valgmuligheder* er valgt. Et udkast til dette layout ses på figuren ovenfor og indeholder:

- Et felt med teksten "Hvad kan vi hjælpe med?"
  som vejledende tekst til patienten når han/hun har behov for hjælp
- Et ikon for hver valgmulighed patienten har at vælge imellem,
  for at gøre valgmulighederne visuelle
- Under hvert ikon skal der stå et navn på valgmuligheden,
  som vejledende tekst

Når man trykker på et ikon skal det fremgå tydeligt hvor man er. Det skal ligeledes være tydeligt at man er i færd med at sende et kald afsted, og hvad kaldet indeholder – for dermed at tilknytte en årsag til kaldet.

Layout for mine kald



Figur 24 Layout for mine kald

Patienten skal kunne se en liste over de kald som han/hun har sendt afsted. Dette giver gennemsigtighed for patienten. Denne liste kaldes ”Mine kald”. Et udkast af dette layout ses på figuren ovenfor. På listen af ”Mine kald” skal der være et item for hvert kald. Dette item er illustreret som en række i listen.
Item skal indeholde:

- Kaldets kategori
- Kategoriens type (afhængig af om kategorien har type)
- Typens detalje (afhængig af om typen har detalje)
- Tidspunkt for hvornår kaldet blev oprettet af patienten i form af:
    - Timer, minutter og sekunder
- Status på kaldet i form af tekst (afventende, udført, fortrudt)
- Farveindikation af status på kaldet
    - Ventende kald har hvide items
    - Udførte og fortrudte kald har grå items

Menu øverst i højre hjørne

Der skal være en tab i en menu øverst I højre hjørne hvor patienten kan logge af. Her skal det være tydeligt hvilken patient der er logget ind.

PersonaleApp

I systemarkitekturen blev systemets packages identificeret og det blev beskrevet hvad de skulle bruges til. I dette afsnit beskrives designet af PersonaleApp og hvordan funktionerne skal implementeres i den fastsatte struktur.

Layouts, activities og fragments

Layoutsne i præsentationslaget er designet i XML-filer, én fil for hvert layout der er behov for. Det har resulteret i syv xml-filer. Den tilhørende logik i business-logik-laget er designet som activities, der som udgangspunkt håndterer logikken for hvert layout, men der er også anvendt fragments. Fragments lever i en aktivitet men har deres egen livscyklus. Dermed behøver man kun at udskifte det der er forskelligt på layoutsene og på den måde undgås kodedublikering. Fragments er anvendt i tilfælde hvor der skal tilgås samme funktioner på flere layouts. Dette er tilfældet for layoutsne for *afventende*- og *udførte* patientkald hvorfra man skal kunne tilgå samme menu. Disse kan med fordel leve i hver deres fragment, men i samme aktivitet, så menuen kan tilgås uagtet af hvilket layout brugeren ser.

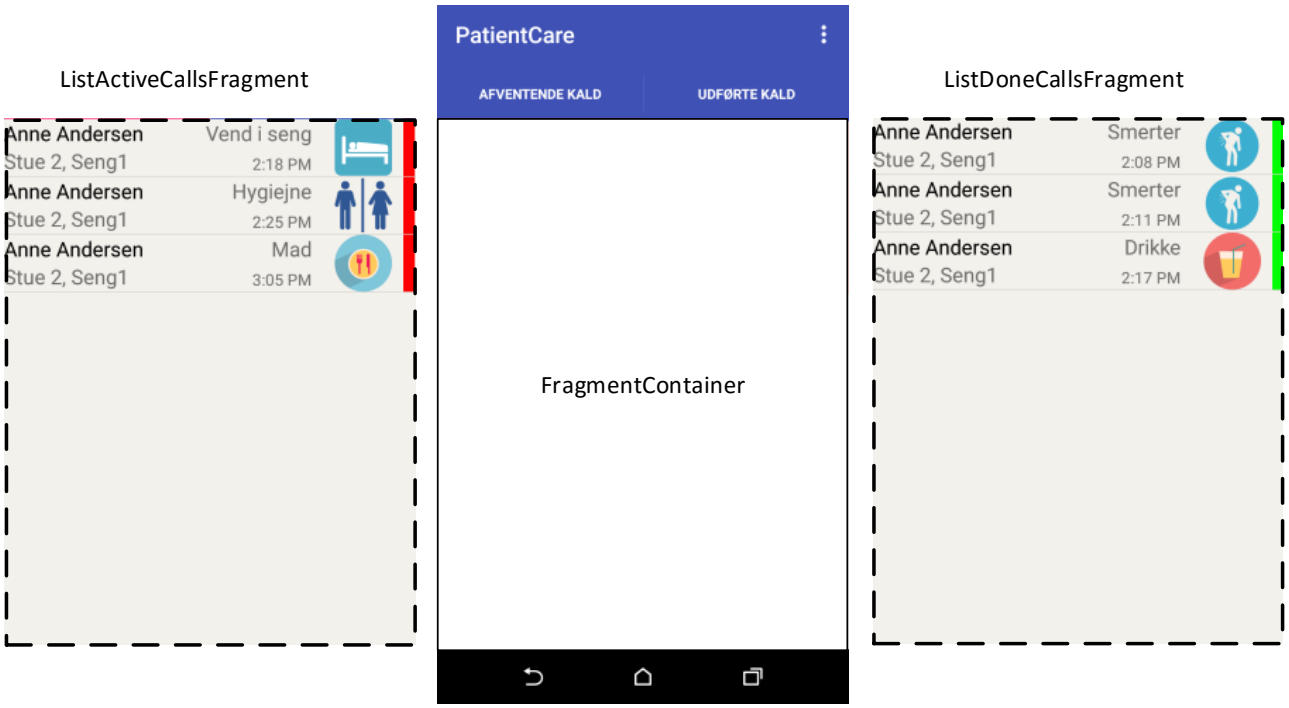
Flere lag i et lag

Når personalet skal kunne se oversigt over *afventende*- og *udførte* kald og tilgå knapper i en menu på samme tid bruges to fragments i en activity (MainActivity).

Her skal man forestille sig at præsentationen har flere layoutlag:

1 Nederste layout (main): fragmentContainer
2: Midterste layout (list_active eller list_completed): container til en liste af afventende eller udførte kald afhængig af hvilken tab der er valgt i menuen.
3: Øverste lag (list_item): layout til hvert item i en liste. Et item er en komponent for hvert patientkald i en liste.

Figur 27 viser at MainActivity har en FragmentContainer, hvor de to forskellige fragments kan køre i.



Figur 25 FragmentContainer

SQLite Database

For PersonaleApp'en har projektgruppen valgt at persistere data lokalt i en SQlite database. SQLite databasen har en relationel databasestruktur med tabeller og kolonner som er anvendt fordi der er behov for at gemme flere parametre for et kald lokalt (se hvilke parametre der gemmes på PersonaleApp i under afsnittet *Systemarkitektur*). Disse parametre kan nemt tilgås ved kendskab til et id, da der er lavet en tabel kun for kald. En anden måde at gemme data på er at gøre brug af Shared Preferences hvor data gemmes i key-value pairs, hvilket ikke er at foretrække, da det kræver en nøgle for hver eneste parameter og ikke gør det muligt at hente flere parametre samlet i én omgang.

Data acces laget

Data acces laget håndterer kommunikationen med den lokale SQLite database som instansieres i klassen *LocalDBOpenHelper*. Alt data udtræk fra SQLite databasen holdes der styr på i en klasse kaldt *LocalDataHandler*. Laget håndterer også adgangen til udveksling af data globalt til og fra WebAPI'et i klassen *WebDataHandler*.

```
ublic class LocalDBOpenHelper extends android.database.sqlite.SQLiteOpenHelper {
    private static final int DATABASE_VERSION = 1;
    private static final String DATABASE_NAME = "personale.db";

    //Table of calls
    public static final String TABLE_CALL = "personale";

    //Columns of the tabel of calls
    public static final String COLUMN_ID = "_id";
    public static final String COLUMN_PATIENT_CPR = "PatientCPR";
    public static final String COLUMN_PATIENT_NAME = "PatientName";
    public static final String COLUMN_ROOM = "Room";
    public static final String COLUMN_BED = "Bed";
    public static final String COLUMN_DEPARTMENT = "Department";
    public static final String COLUMN_CATEGORY = "Category";
    public static final String COLUMN_CHOICE = "Choice";
    public static final String COLUMN_DETAIL = "Detail";
    public static final String COLUMN_CREATED_ON = "CreatedOn";
    public static final String COLUMN_MODIFIED_ON = "ModifiedOn";
    public static final String COLUMN_STATUS = "Status";
    public static final String COLUMN_ICON = "Icon";

    //Table of personnel
    public static final String TABLE_LOGIN = "Logind";

    //Columns og the table of personnel
    public static final String COLUMN_LOGIN_ID = "login_id";
    public static final String COLUMN_USERNAME = "username";
    public static final String COLUMN_PASSWORD = "password";

    //Constructor
    public LocalDBOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
```

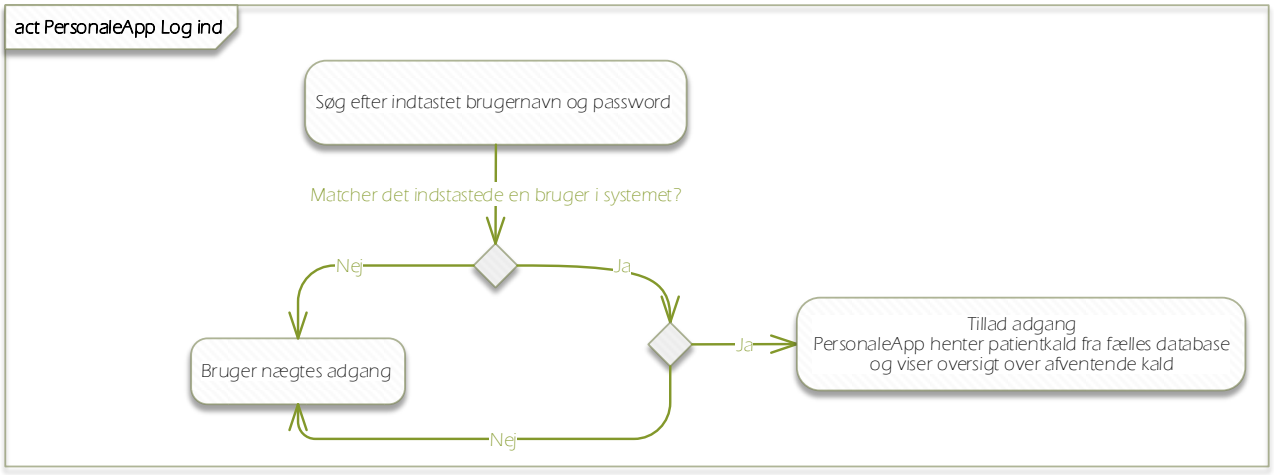
Figur 26 Tabelstruktur for SQLitedatabasen for PersonaleApp

I figure 28 ses tabelstrukturen i klassen "LocalDBOpenHelper, der håndtere oprettelsen af lokaldatabasen. Der benyttes følgende fremgangsmåde til at håndtere SQLite operationer:

1. Opret tabelstrukturen I LocalDBOpenHelper
2. Opret DTO der bruges til at gemme data ind på
3. Opret SQLite Database Handler som en klasse der indeholder metoder der skal bruge dataen
4. Lav alle CRUD operationerne create(Create, Read, Update, Delete)

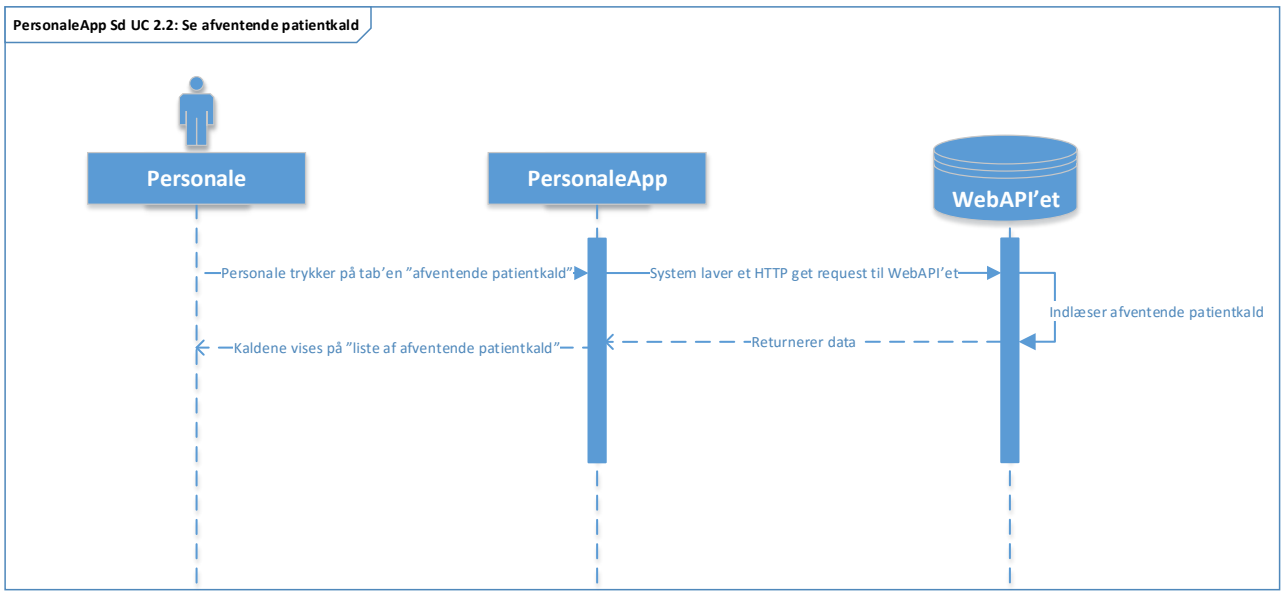
Fra Use Cases til designdiagrammer

Figur 29 viser et aktivitetesdiagram for Log ind.



Figur 27 Aktivitetsdiagram for log ind

Figur 30 viser et sekvensdiagram for use casen "Se afventende kald"



Figur 28 Sekvensdiagram for use case 2.2

Design af funktioner

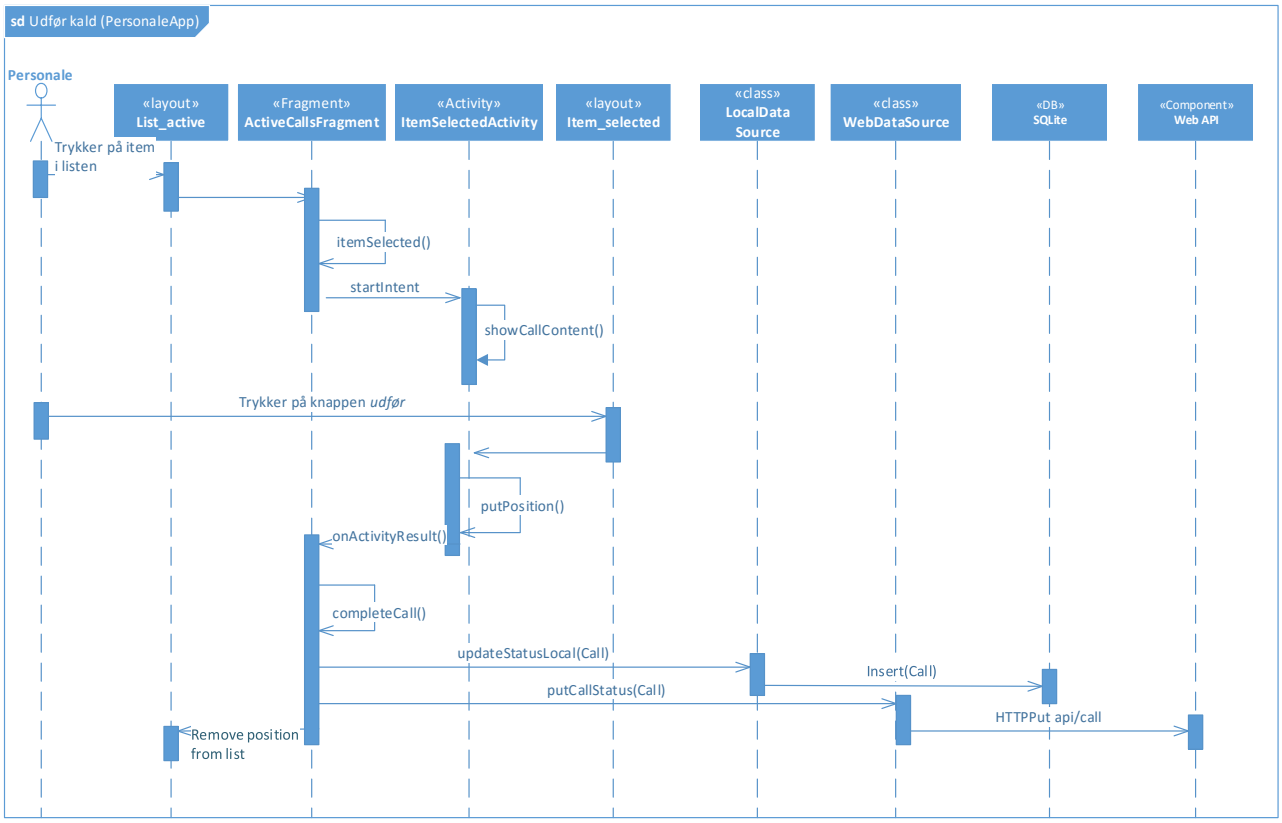
Nogle af de vigtigste funktioner PersonaleApp'en skal tage sig af er at modtage patientkald og gøre det muligt for personalet at udføre dem.

Sekvensdiagram (Udfør kald)

Med udgangspunkt i Use Case 2.3 "Udfør kald" laves et sekvensdiagram der beskriver sekvensen for at et kald udføres i PersonaleApp, hvilket er en af de primære sekvenser for dette modul.

Figur 31 viser et sekvensdiagram over hvordan et kald udføres. Sekvensdiagrammet indeholder to sekvenser.
Når patienten har sendt et patientkald og personalet har modtaget det via WebAPI'et, kan personalet vælge at udføre det. Den første sekvens starter ved at patientkaldet ligger som et item i en liste af

afventende kald (list_active) og personalet trykker på dette item. Herefter ledes personalet videre til en ny side (item_selected), hvor der står detaljer omkring patientkaldet. Metoden ”showCallContent” kaldes, efter at personalet har trykket på et item. Da items i listen er indekseret fra 0 til antallet af items, vil data fra den valgte position blive ført med over via en intent der igangsætter den nye aktivitet, (ItemSelectedActivity). (Læs nærmere om design af PersonalepApp’en i Android studio i Designdokumentet). Fra siden item_selected har brugeren mulighed for at udføre kaldet ved at trykke på en knap ”udfør”. Den anden sekvens starter med når personalet trykker på ”udfør”. Her bruges positionen af det valgte item til at identificere patientkaldet som skal udføres. Ud fra denne information opdateres kaldets status fra afventende til udført, hvilket indsættes i den lokale SQLite database på smartphonen (Læs mere om denne længere nede). Derefter sendes ændringen til WebAPI’et med en http protokol. Det pågældende patientkald fjernes fra listen af afventende kald ud fra den givende position og føjes til listen af udførte kald (list_completed). På den måde vil personalet se at kaldet er udført.



Figur 29 Sekvensdiagram for Udføre kald for PersonaleApp

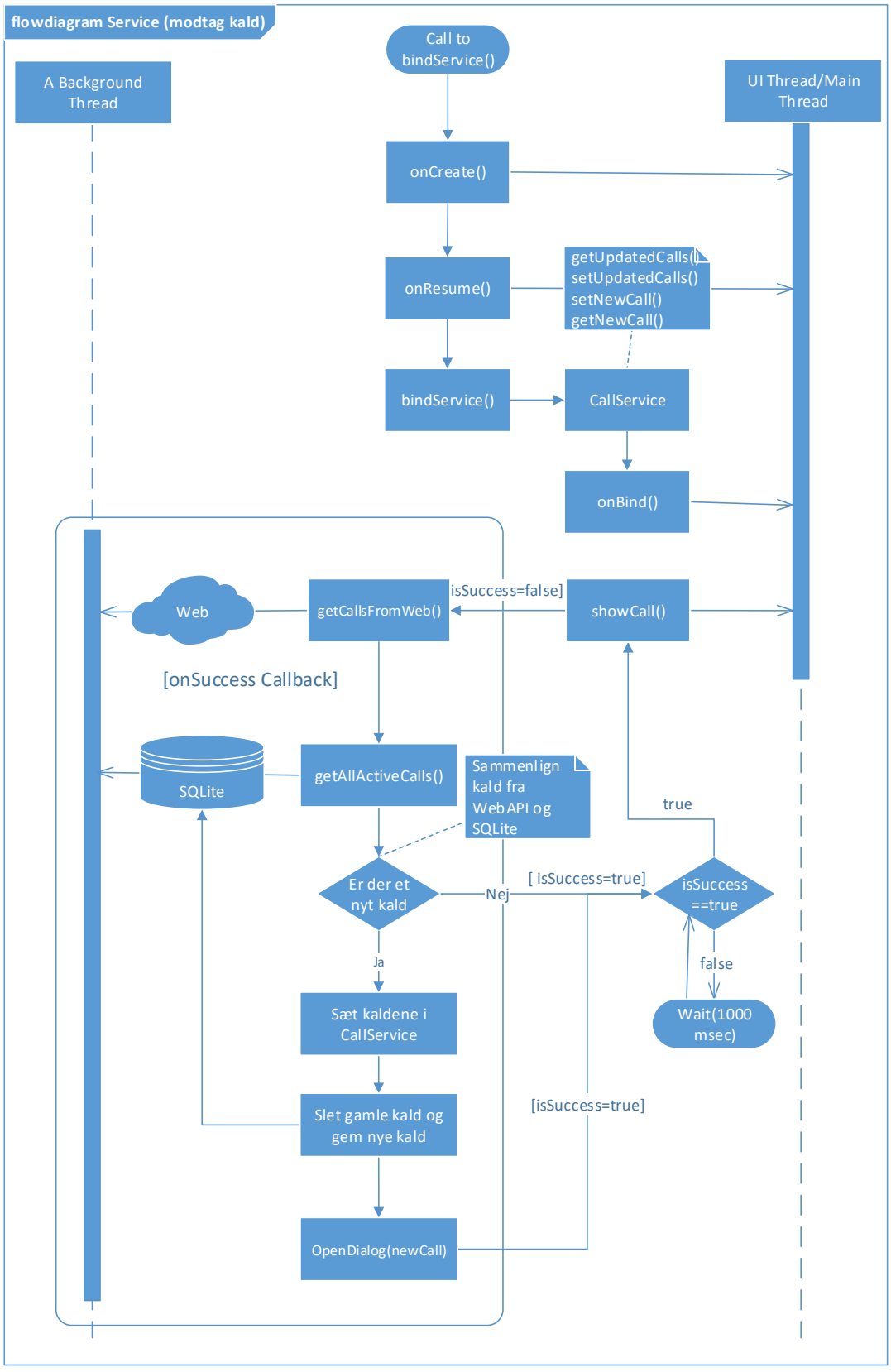
Bound Service (Modtag kald)

PersonaleApp modtager patientkaldene fra PatientApp via WebAPI’et løbende. Det er der en service i App’en der håndterer.

Når et nyt kald fra patienten er blevet sendt fra PatientApp’en, er der brug for en teknologi til at notificere personalet. Da det er prioriteret at vise konceptet frem for performance, er der valgt en pull funktionalitet til at indlæse nye kald fra systemet. Denne pull funktionalitet kan enten laves ved at brugeren der interagerer med systemet gør noget aktivt eller det kan håndteres med en service. Da en service kan håndtere baggrundsopgaver, vil der ikke være behov for at personalet manuelt skal interagere med app’en for at holde sig opdateret med de indkomne kald. En service i android kommer i to udgaver: En Started

service og Bound Service. En Started service startes af en aktivitet ved at kalde startService() hvorefter servicen uafbrudt vil udføre én operation i baggrunden uden at returnere et resultat til kalderen. I en Bound service bindes en aktivitet eller en komponent til servicen, der tillader et client-server interface, hvilket gør det muligt at interagere med servicen, sende requests og modtage resultater på tværs af processer. I PatientApp'en er der behov for det sidstnævnte, da der løbende skal hentes patientkald fra en webserver via WebAPI'et. Derfor er der valgt en Bound service.

Servicen starter op i "MainActivity", som er den aktivitet der starter når brugeren af PersonaleApp er logget ind. Servicen håndterer opgaven at hente alle afventede patientkald fra WebAPI'et via et http request til en URL. WebAPI'et returnerer en string af JSON objekter som deserialiseres til konkrete objekter.



Figur 30 Flowdiagram Service (Modtag kald)

På figur 32 ses det hvordan servicen bliver oprettet i PatientApp. Til højre i diagrammet ses tidslinjen for aktiviteter og andet logik der kører i hovedtråden (MainActivity). Og til venstre ses tidslinjen for en separat tråd hvor de aktiviteter servicen udfører kører på. Servicen kører på en anden tråd end hovedtråden så den ikke laver blokerende operationer på hovedtråden. Bindingsprocessen ses i figur 32 hvor android frameworkets ”Service” implementeres af ”CallService” og for at tillade bindingen, skal onBind() implementeres. Denne metode returnerer et IBinder objekt, der gør det muligt for klienten at interagere med servicen. Når bindingen er sket, kaldes en metode ”showCall()” der vil lave en række operationer der sker i baggrundstråden i stedet for at blokere hovedtråden:

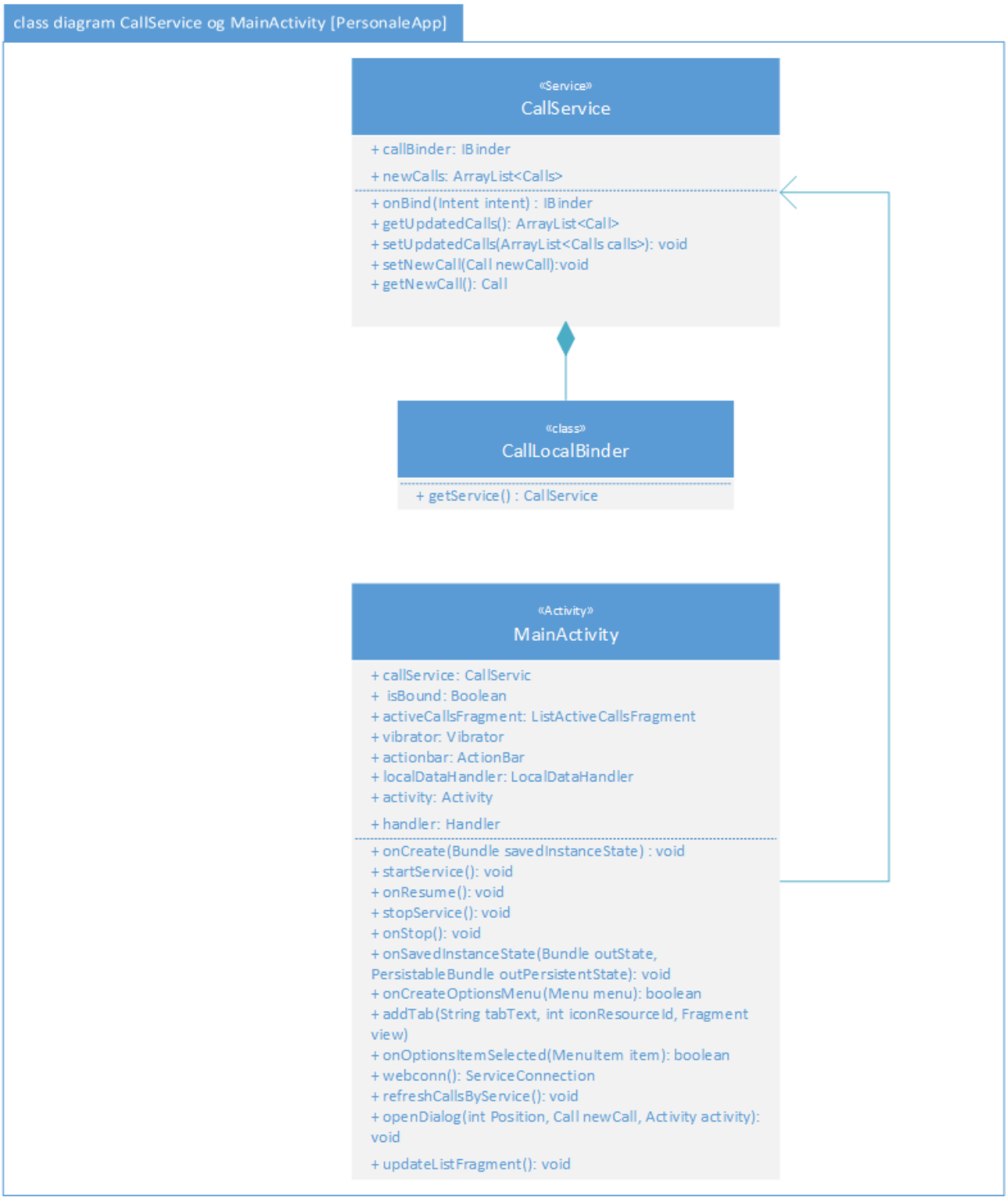
1. Hente nye kald fra en webserver via et WebAPI.
2. Indlæse eksisterende kald fra lokal databasen via SQLite.
3. Gemme resultaterne i servicen
4. Åbne en ny dialogboks med information omkring det nye kald

Eksisterende afventende kald indlæses fra fælles databasen hvor eventuelle nye kald er gemt. Det indlæste sammenlignes med afventende kald fra lokal databasen. Hvis det indlæste afviger fra det lokale data er det udtryk for at der er kommet et nyt afventende kald. Resultatet gemmes i servicen, så det kan tilgås via den aktivitet, som har bundet sig til den, for herefter at vise en dialogboks tilbage på hovedtråden med informationer om det nye kald.
Disse operationer sker i baggrundstråden fordi de indebærer en række intensive operationer og requests, som man som bruger af PatientApp’en ikke vil vente på, hvis de blokkerede hovedtråden.

Da requests til webserveren sker asynkront grundet det framework der benyttes til det, og da det kan tage noget tid at få respons tilbage, må baggrundstråden stå og vente på at operationerne er færdige, før den kan starte næste iteration. Her står den og venter hvert sekund til dette er sket. Dette er illustreret i figur 29 ved at der tjekkes på en bool variable ”isSuccess”.

Servicen er til stor gavn for PatientApp’en og en fremragende løsning hvor det ikke kræver en brugerinterkation for at gøre brug af pull funktionaliteten. Performancemæssigt vil det være en bedre løsning gøre brug af push funktionalitet, da man på den måde ikke forgæves brænder PatientAppen’s CPU resurser og tid af i en situation, hvor der ikke er kommet et nyt patientkald. Dette kræver dog at push funktionen implementeres i WebAPI’et som skal sørge for at patientkaldet der modtages fra PatientApp sendes videre til PersonaleApp. Konceptet er prioriteret højere end performance i det tilfælde.

For at vise implementering af de to kritiske klasse MainActivity og CallService vises nedstående klassediagram. På figur 33 ses deres attributter, metoder og relationen mellem objekterne som det er blevet implementeret.



Figur 31 - Klassediagram for CallService og MainActivity

Layouts

I dette afsnit beskrives de layouts som brugeren af PersonaleApp visuelt ser og dermed er det userinterface som brugeren kan interagere med. Der er defineret syv forskellige XML-filer for layoutsne.

Når personalet skal logge ind ser de layoutet *login.xml* som indeholder noget tekst med informationer og tekstfelter hvor personalet kan indtaste brugernavn og password samt en knap som igangsætter log ind af personalet. Når personalet er logget ind oprettes layoutet *main.xml* som indeholder en fragmentcontainer, hvor fragmentklasserne *ListActiveCallsFragment* og *ListCompletedCallsFragment* bliver oprettet. *ListCompletedCallsFragment* består af layoutet *list_active.xml* og *ListCompletedCallsFragment* består af layoutet *list_completed.xml*. Begge layouts indeholder en liste med items som er tiltænkt at være en liste af patientkald. Hvert item består af layoutet *listitem.xml* som indeholder tekstfelter hvor information om patienten og patientens kald skal stå. Heriblandt patientens navn, hvilken stue og seng patienten ligger på, tidspunktet patienten sendte kaldet og årsagen patientkaldet. Der er også et billede som viser et billede af hvilken kategori årsagen hører til. Fx hvis patienten skal på toilet er der et billede af et toilet. Der er helt til højre af item'et en farveindikation til personalet om patientkaldets status. Hvis personalet mangler at behandle kaldet har kaldet farven rød og hvis det er blevet behandlet som udført har det farven grøn. Hvis personalet har brug for yderligere information omkring et kald der skal udføres kan der trykkes på et item og layoutet *itemselected.xml* vil blive vist. Dette layout indeholder yderligere information om det valgte kald. Hvis der ikke er nogle kald der kan blive vist på listen vises layoutet *emptylist.xml* som giver brugeren besked om at der ikke er nogle kald i listen. Dette gøres for at brugeren ikke får en oplevelse af at applikationen ikke fungerer når der ikke er nogle kald på listen.

Layoutsne er sat op efter LinearLayout (vertikalt) og RelativeLayout.

LinearLayout: main.xml, list_active.xml og list_completed.xml

RelativeLayout: login.xml, list_item.xml, item_selected.xml og empty_list.xml

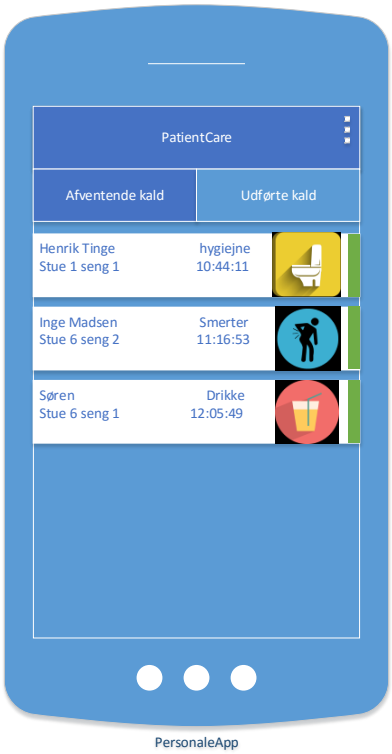
Layout for log ind side

Figur 32 Layout for log ind side

Det skal tydeligt fremgå af log ind siden at Personalet skal logge ind med brugernavn og password for at få adgang til app'en. Dette layout viser et udkast til Log ind siden for PersonaleApp.

Layout for afventende- og udførte kald

Figur 33 Layout for afventede



Figur 34 Layout for udførte kald

Når brugeren er logget ind er der en menu hvor personalet nemt kan navigere rundt i App'en. Menuen består af to tabs, den ene med teksten *Afventende kald,* den anden med teksten *Udførte kald.*

På listen af "Afventende kald" og listen af "Udførte kald" er der et item for hvert kald.
Dette item indeholder:

- Patientens navn
- Stuenummer
- Sengepladsnummer
- Kaldets kategori
- Et billede der hører til kategorien
- Tidspunkt for hvornår kaldet blev oprettet af patienten
- En farveindikation af status på kaldet
    - Ventende kald er røde
    - Udførte kald er grønne

Layout for detaljeside

Figur 35 Layout for detaljeside

Når brugeren har trykket på et item i listen af afventende kald er der være en detaljeside.
Detaljesiden indeholder:

- Patientens navn

- Stuenummer
- Sengepladsnummer
- Kaldets kategori
- Kategoriens type (afgængig af om kategorien har type)
- Typens detalje (afhængig af om typen har detalje)
- Et billede der hører til kategorien
- Tidspunkt for hvornår kaldet blev oprettet af patienten i form af:
    - Timer, minutter og sekunder

Menu øverst i højre hjørne

Der er en tab i en menu øverst I højre hjørne hvor personalet kan logge af.

AdminApp

Sekvensdiagrammer for Use Case (Opret kategori)

Følgende tager udgangspunkt i Use Case 3.1 Opret kategori. Der laves to sekvensdiagrammer for denne Use Case. Det ene viser flowet fra hvor en kategori oprettes på AdminApp og requestes til WebAPI'et. Det andet viser hvordan requestet til WebAPI'et fører til at kategorien bliver gemt i MongoDB. Flowet for de andre Use Cases vises i Bilag 4.

WebAPI'et indgår ikke direkte i Use Case definitioner, men er en nødvendighed for at PatientCare's devices kan interagere.

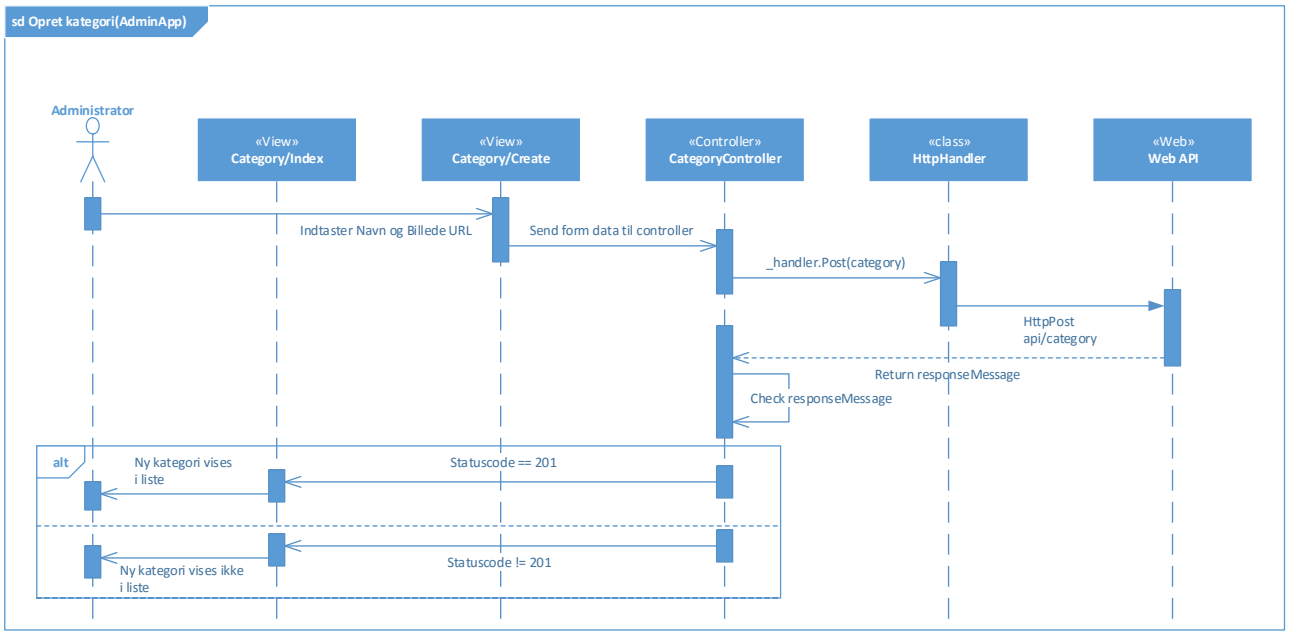
For sekvensdiagrammerne for AdminApp er WebAPI'et en black box, da det eneste AdminApp kender til WebAPI'et, er hvad der skal sendes til API'et og hvilken URL der skal rammes.

For sekvensdiagrammet for WebAPI'et gælder det at indgående trafik til WebAPI'et ses som en black box. Dette er fordi WebAPI'et ses som en isoleret del af systemet, da WebAPI'et fungerer som et web interface. Dog vides det for hver Use Case, hvilken applikation der kalder WebAPI'et og er markeret med <<web>>. Dette indikerer at kaldet kommer ude fra internettet og hvilken applikation der laver kaldet.

Information om hvordan der kan kommunikeres med WebAPI'et kan findes her:
http://patientcareapi.azurewebsites.net/Help

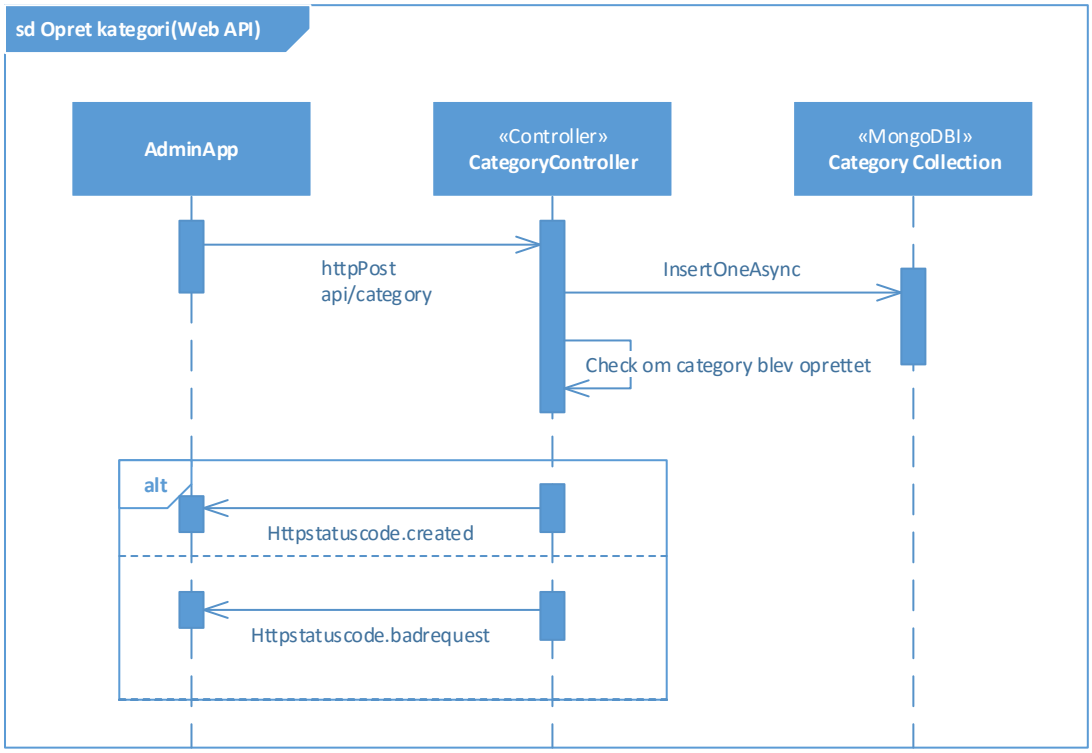
Implementeringen af disse sekvensdiagrammer kan findes i koden for AdminApp, der er at finde i zip filen som kan findes på digital eksamen afleverings siden.

Opret kategori(AdminApp)



Figur 36 - Sekvensdiagram for Use case 3.1 Opret kald for AdminApp

Opret kategori(WebAPI)



Figur 37 - Sekvens diagram for Opret kategori WebAPI

Database

Nedenstående billeder er udklip af MongoDB databasen


```
{
    "PatientCPR":"1111111118",
    "Category":"Smerter",
    "Choice":"Efter operation",
    "Detail":"Moderate",
    "CreatedOn":"12:37 PM"
}
```


Figur 38 Patientdata

I ovenstående figur ses de data som bliver sendt fra PatientApp'en til WebAPI'et, ved oprettelse af et nyt kald. WebAPI'et går efterfølgende ned i *Patient Collectionen*, i databasen og kigger efter PatientCPR. Hvis det ikke findes bliver der smidt en exception og fejl besked bliver givet til brugeren. Hvis en Patient med det sendte CPR nummer findes, bliver yderligere information om Patienten sat tilføjet til objektet og bliver efterfølgende gemt i databasen's *Call Collection.*

```
{
   "_id":{
      "$oid":"562e5073e4b06ba894cee07b"
   },
   "PatientCPR":"1111111118",
   "PatientName":"Louise Andersen",
   "Department":"Gyn-obs",
   "Room":"Stue 2",
   "Bed":1,
   "ImportantInfo":"Acne"
}
```


Figur 39 - Mock data

Patient Data fra *Patient Collection* i databasen.


```
{
   "_id":"564c62e34ca8e93380157bdf",
   "PatientCPR":"1111111118",
   "PatientName":"Louise Andersen",
   "Room":"Stue 2",
   "Bed":"1",
   "Department":"Gyn-obs",
   "Category":"Smerter",
   "Choice":"Efter operation",
   "Detail":"Moderate",
   "CreatedOn":"12:37 PM",
   "ModifiedOn":"Wednesday, November 18, 2015 11:38:14 AM",
   "Status":1
}
```


Figur 40 - Patientinformation

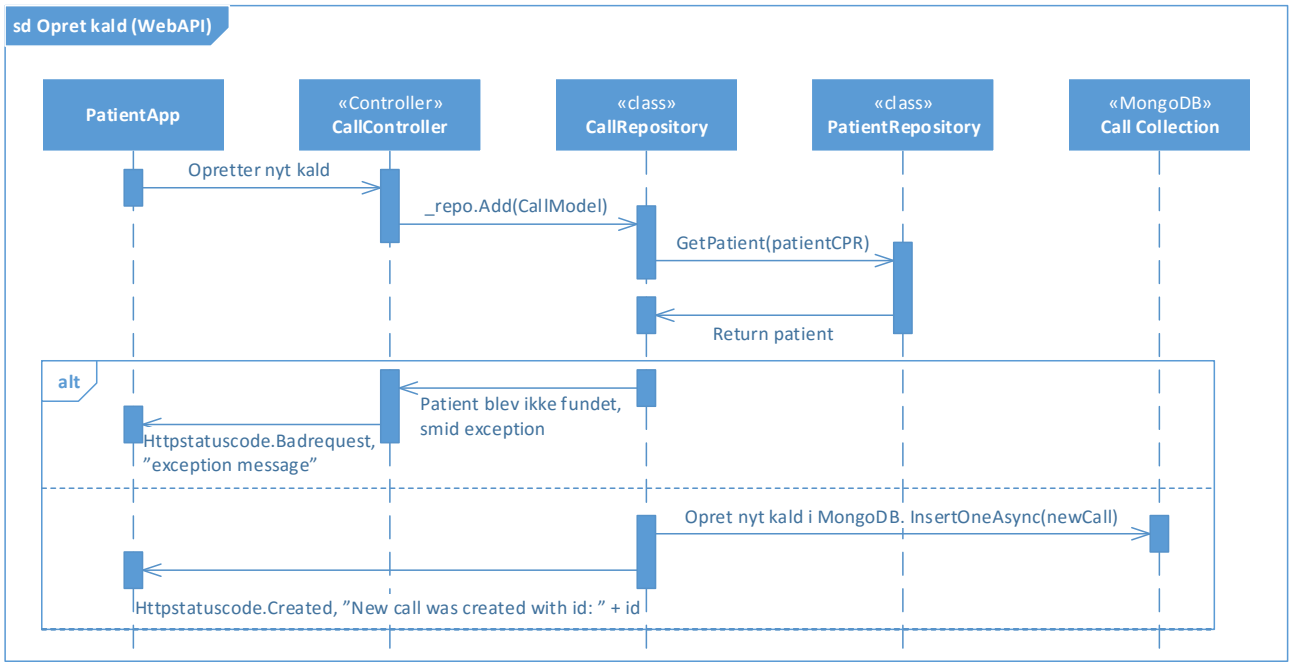
Figur 40 er det færdige objekt som bliver gemt i *Call Collection* og er samtidig det objekt som PersonaleApp'en modtager.

WebAPI

Sekvensdiagrammer for Use Case (Opret kald)

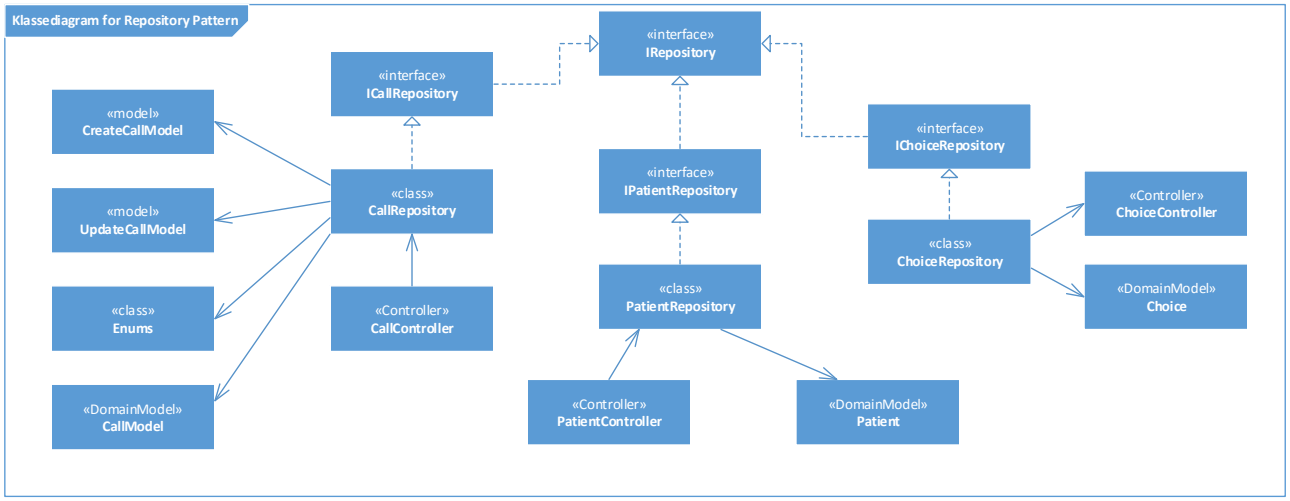
For at beskrive kommunikationen ud fra f.eks. PatientApp'en og til WebAPI'et er det nødvendigt at opstille et sekvensdiagram.

Der tages udgangspunkt i Use Case 3.1: Opret kald, da der her vises flowet fra PatientApp'en til WebAPI'et og dens vej tilbage:



Figur 41 - Sekvensdiagram for Use Case 3,1: Opret kald for WebAPI

I figur 41 oprettes et kald fra PatientApp'en som sendes til WebAPI'et. Først undersøges om patienten er indlagt. Hvis patienten ikke er blevet fundet, smides en exception fra WebAPI'et tilbage til PatientApp'en. Ellers hvis patienten er fundet, oprettes et nyt kald i MongoDB. Ved oprettelsen laver MongoDB et id på kaldet. Dette id sendes i en responsbesked tilbage til PatientApp'en.



Figur 42 - Klasse diagram for WebAPI'ets Repository Pattern

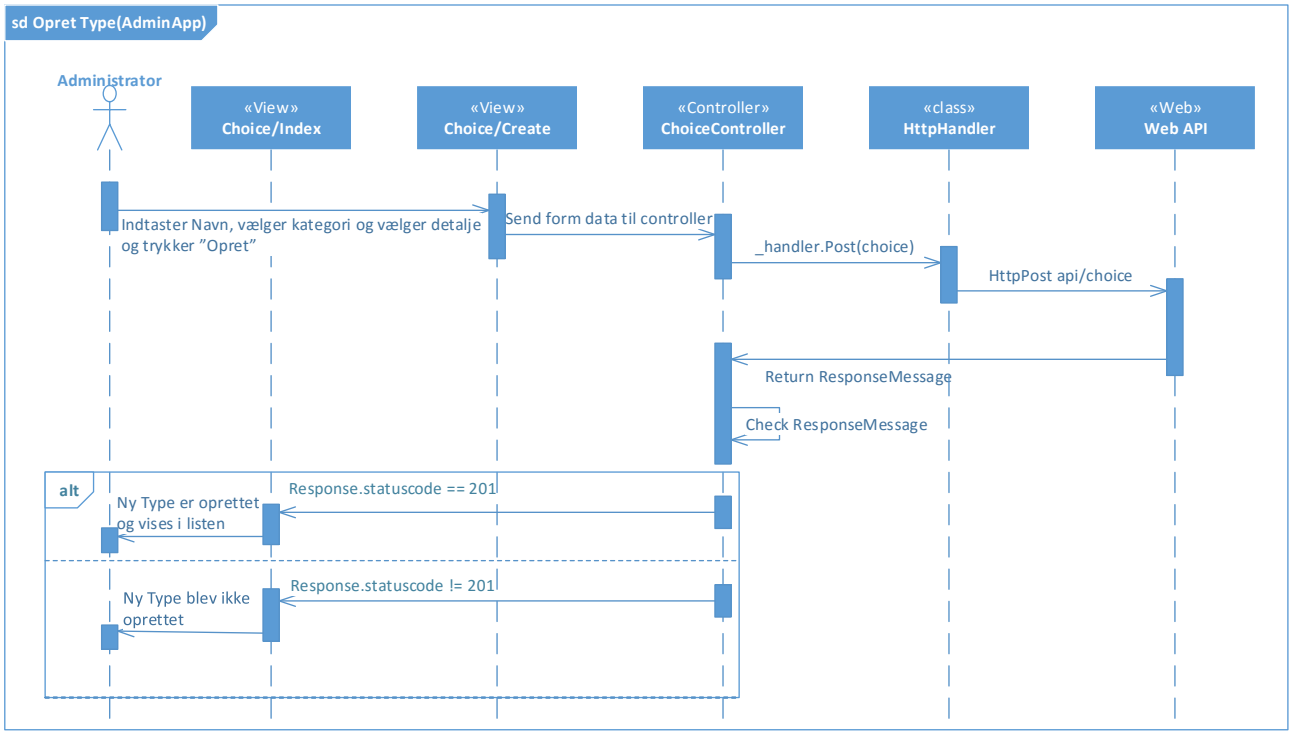
Figur 42 viser et klassediagram for WebAPI'ets repository pattern. Der er på nuværende tidpunkt ikke inddraget alle controllers i WebAPI'ets repository pattern.

4 Bilag

Sekvensdiagrammer for Use Cases (AdminApp og WebAPI)

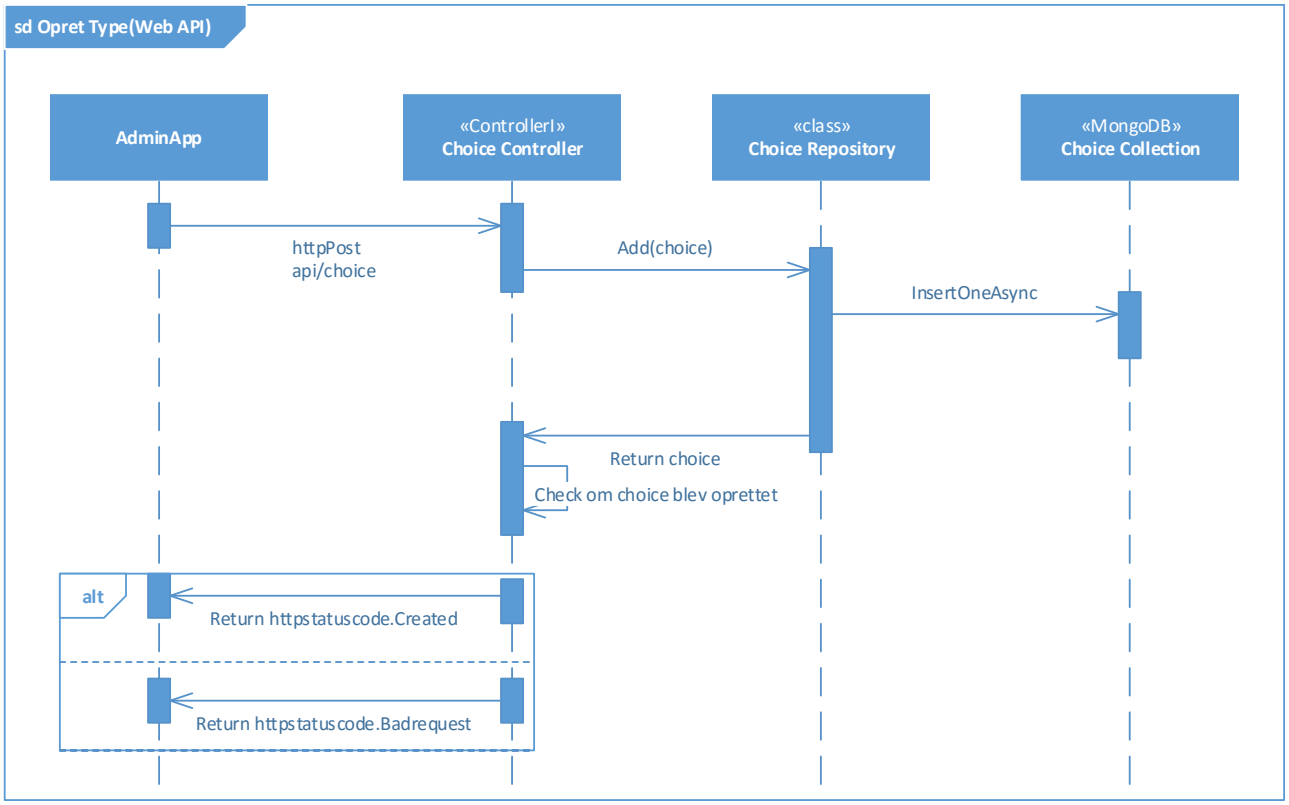
Diagrammer i dette bilag er lavet i forbindelse med udvikling af use casene for AdminApp. Hver use case består af to sekvensdiagrammer hvor det første altid vil være diagrammet for AdminApp efterfuldt af et sekvensdiagram for WebAPI for den givne use case. I diagrammerne for AdminApp ses WebAPI'et som en black box og omvendt diagrammer omhandlende WebAPI'ets sekvens i en use case og viser således sekvensen igennem use casen.

Opret type(AdminApp)



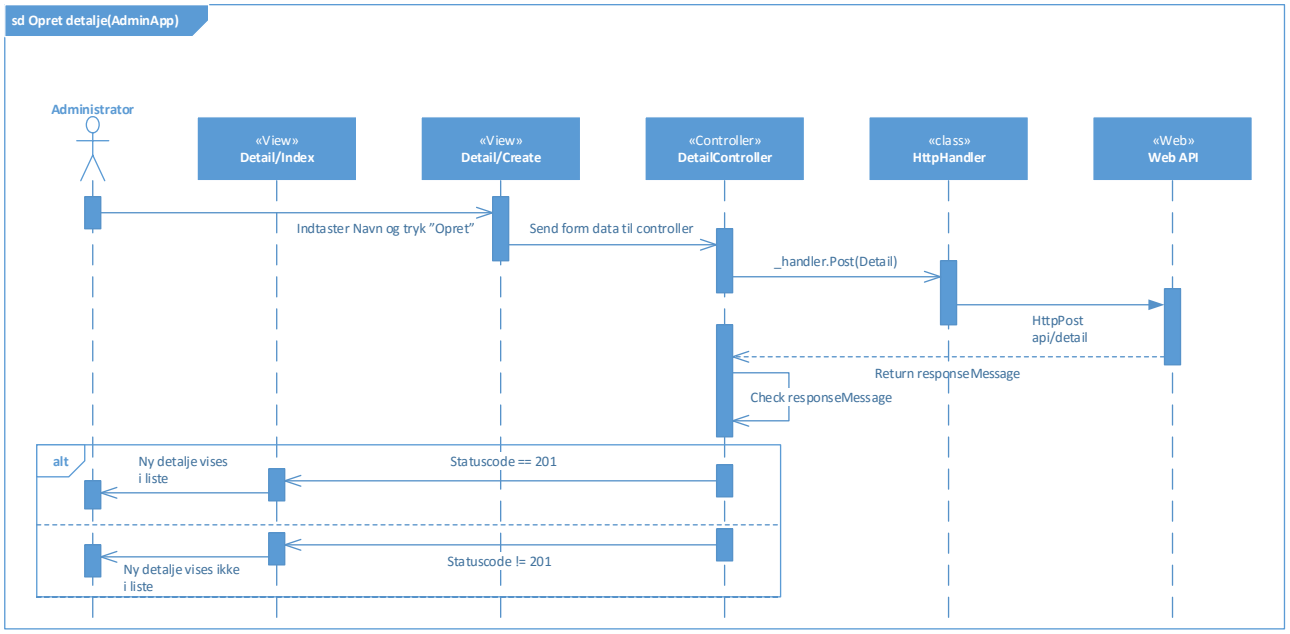
Figur 43 – Sekvensdiagram for Opret type AdminApp

Opret type(WebAPI)



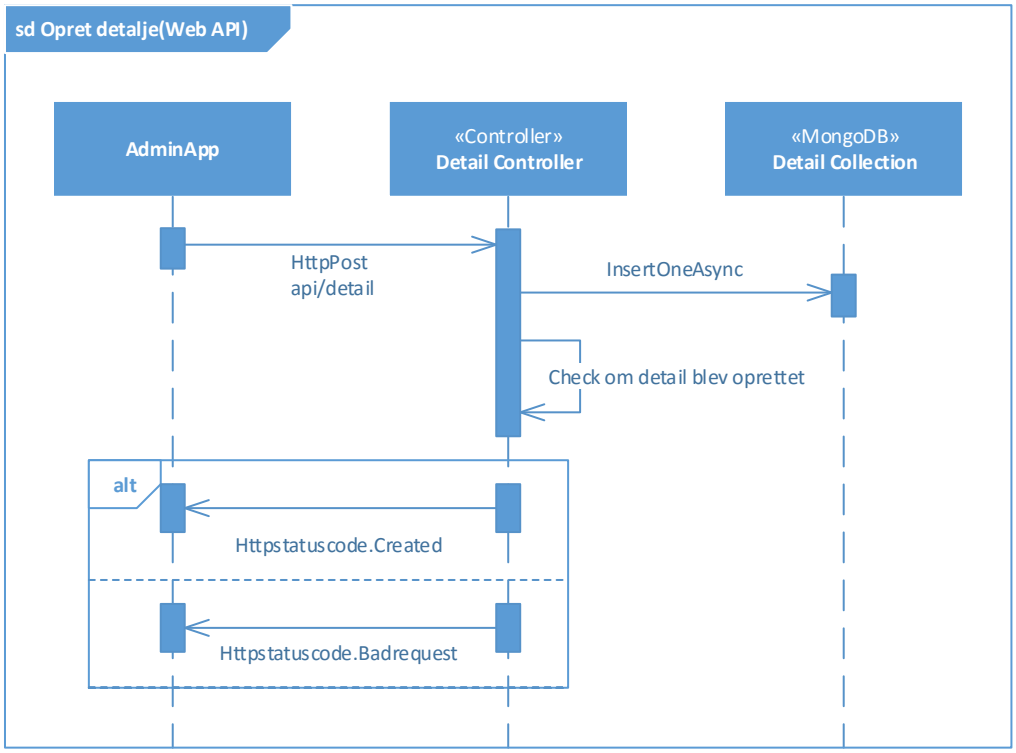
Figur 44 – Sekvensdiagram for Opret type WebAPI

Opret detalje (AdminApp)



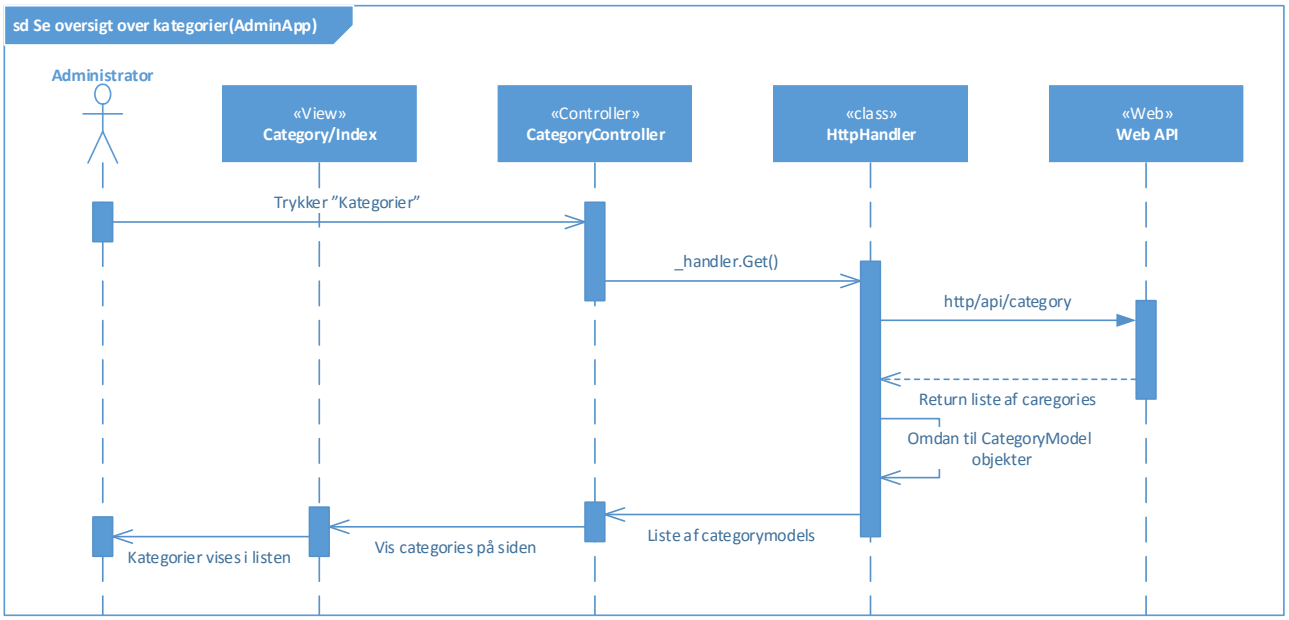
Figur 45 – Sekvensdiagram for Opret detalje AdminApp

Opret detalje (WebAPI)



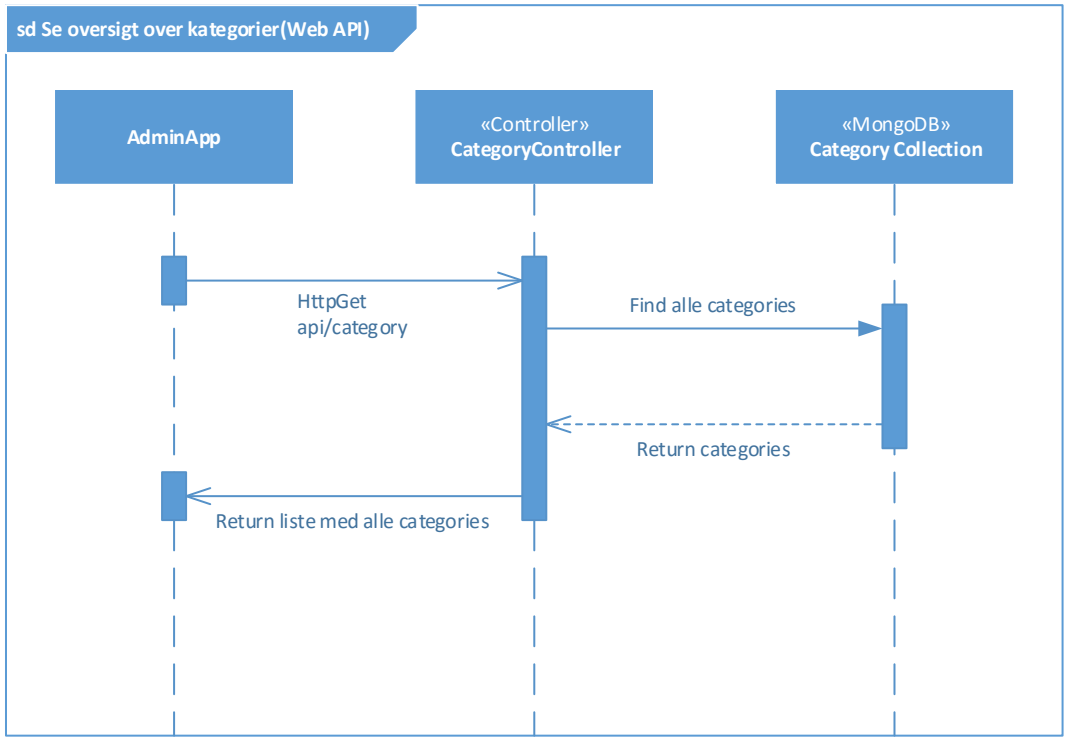
Figur 46 - Sekvensdiagram for opret detalje WebAPI

Se oversigt over kategorier (AdminApp)



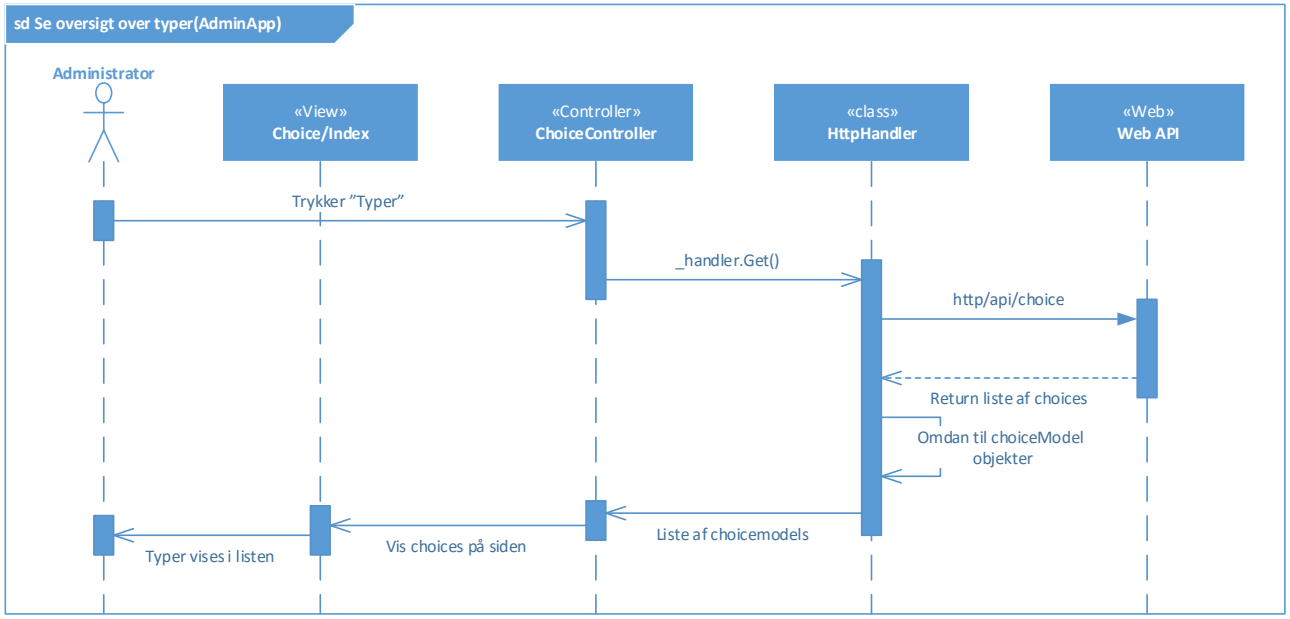
Figur 47 - Sekvensdiagram for se oversigt over kategorier AdminApp

Se oversigt over kategorier (WebAPI)



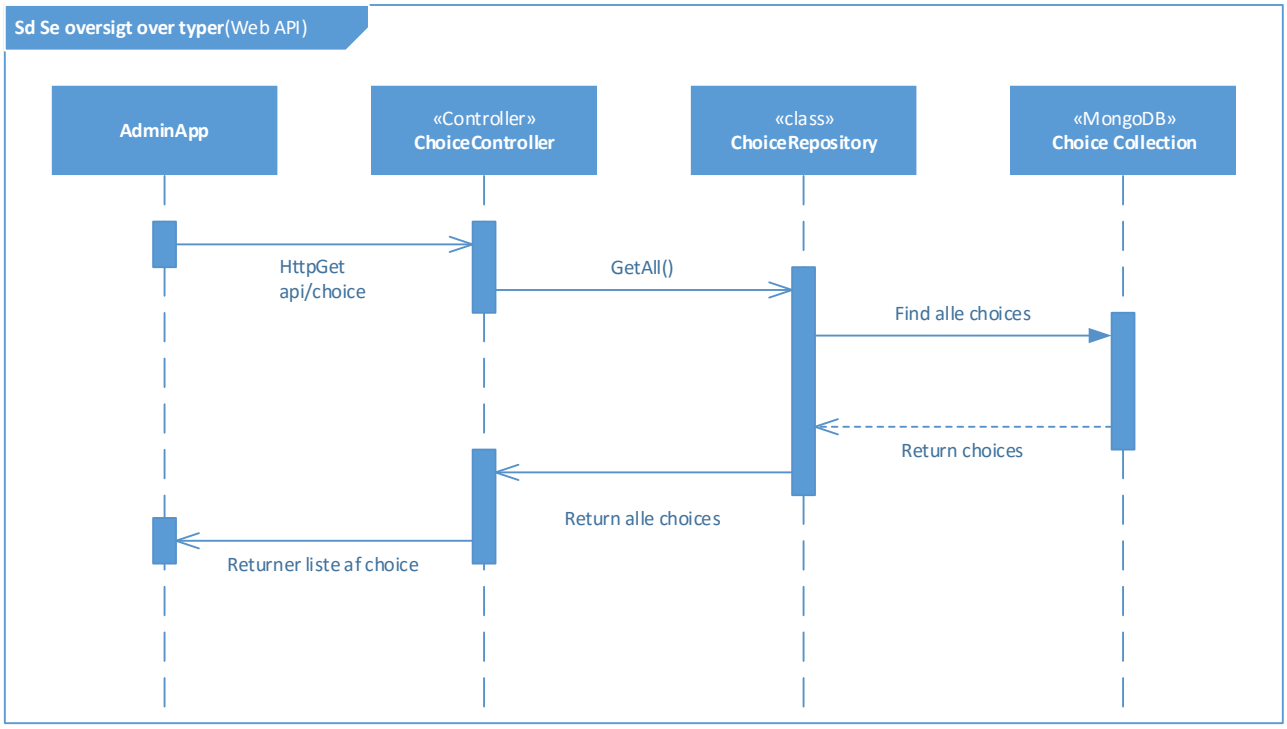
Figur 48 - Sekvensdiagram for se oversigt over kategorier WebAPI

Se oversigt over typer (AdminApp)



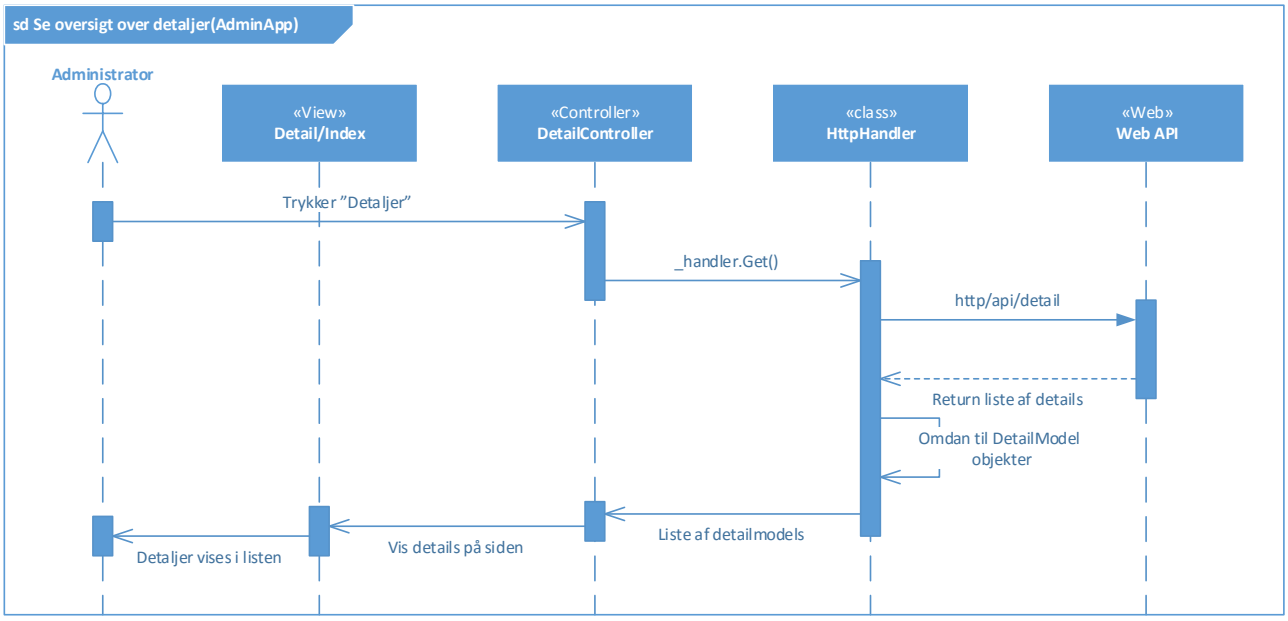
Figur 49 - Sekvensdiagram for se oversigt over typer AdminApp

Se oversigt over typer (WebAPI)



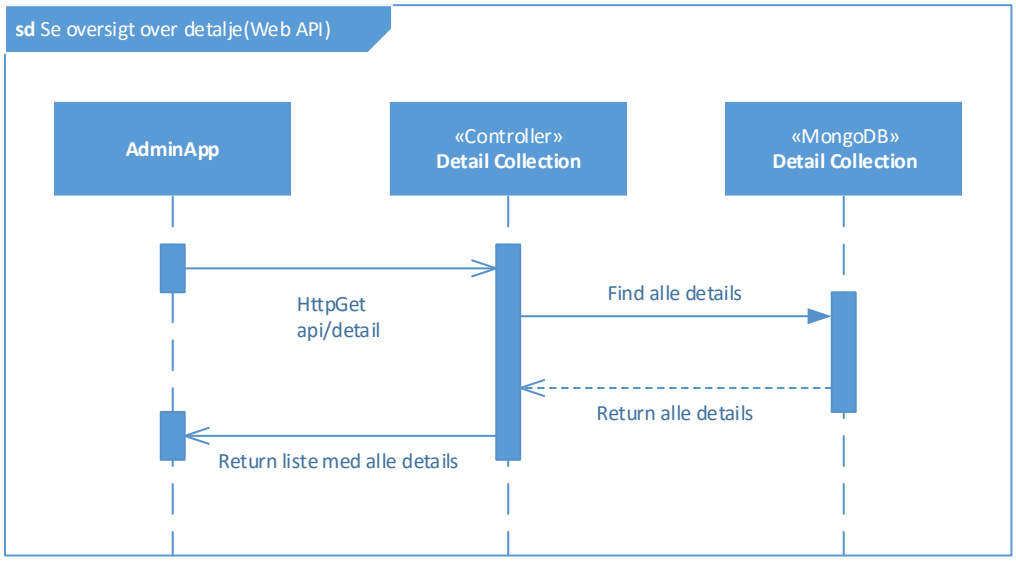
Figur 50 - Sekvensdiagram for se oversigt over typer WebAPI

Se oversigt over detaljer (AdminApp)



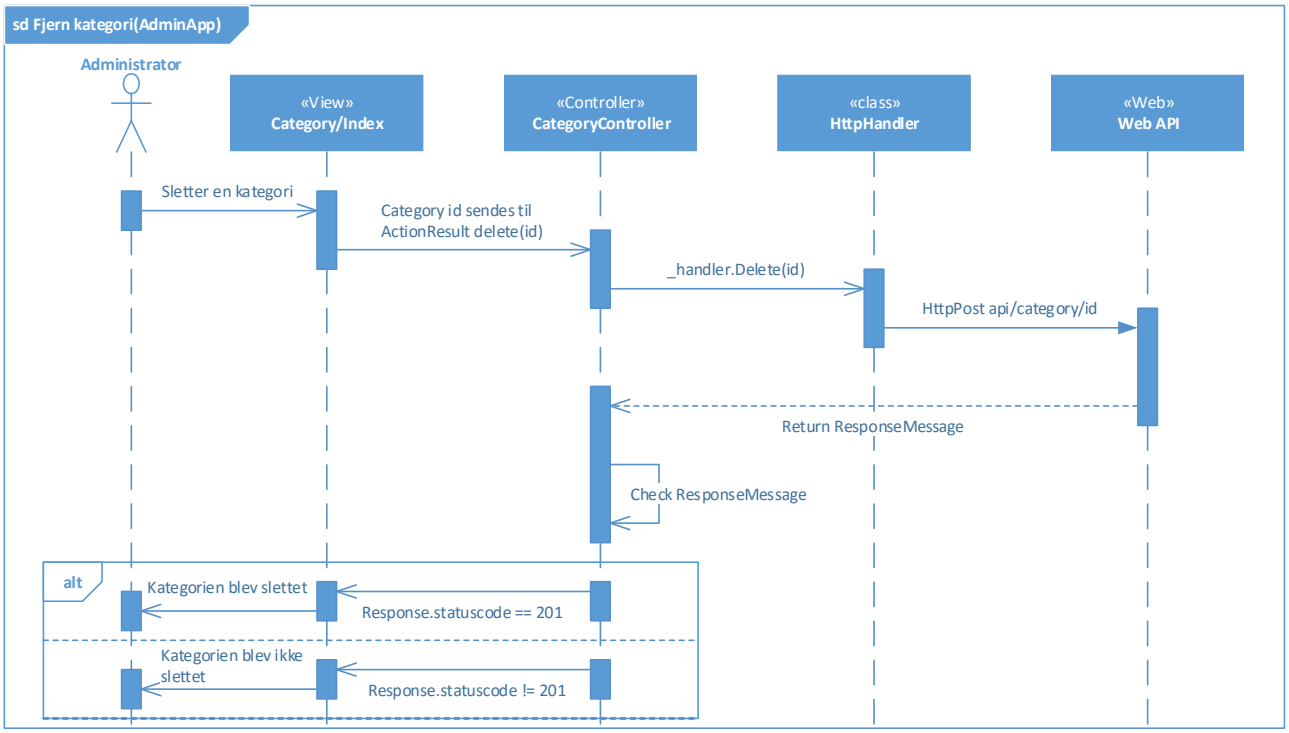
Figur 51 - Sekvensdiagram for se oversigt over detaljer AdminApp

Se oversigt over detaljer (WebAPI)



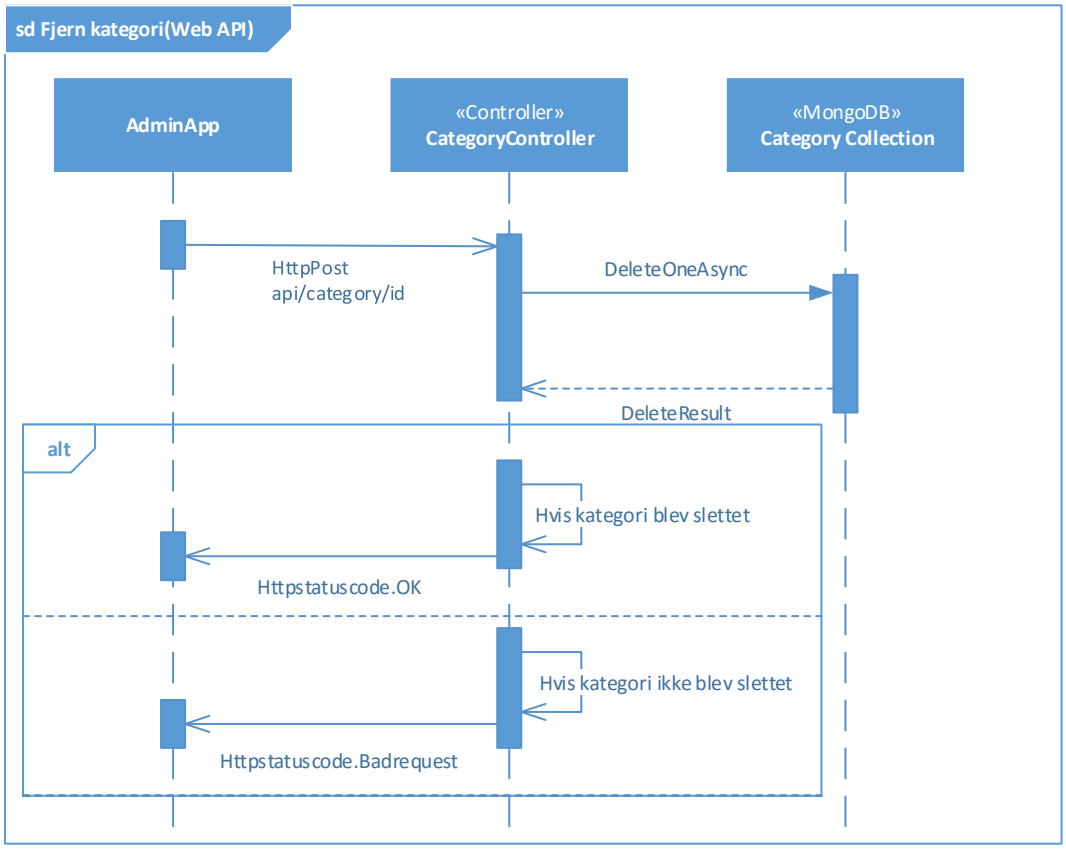
Figur 52 - Sekvensdiagram for se oversigt over detaljer WebAPI

Fjern kategori (AdminApp)



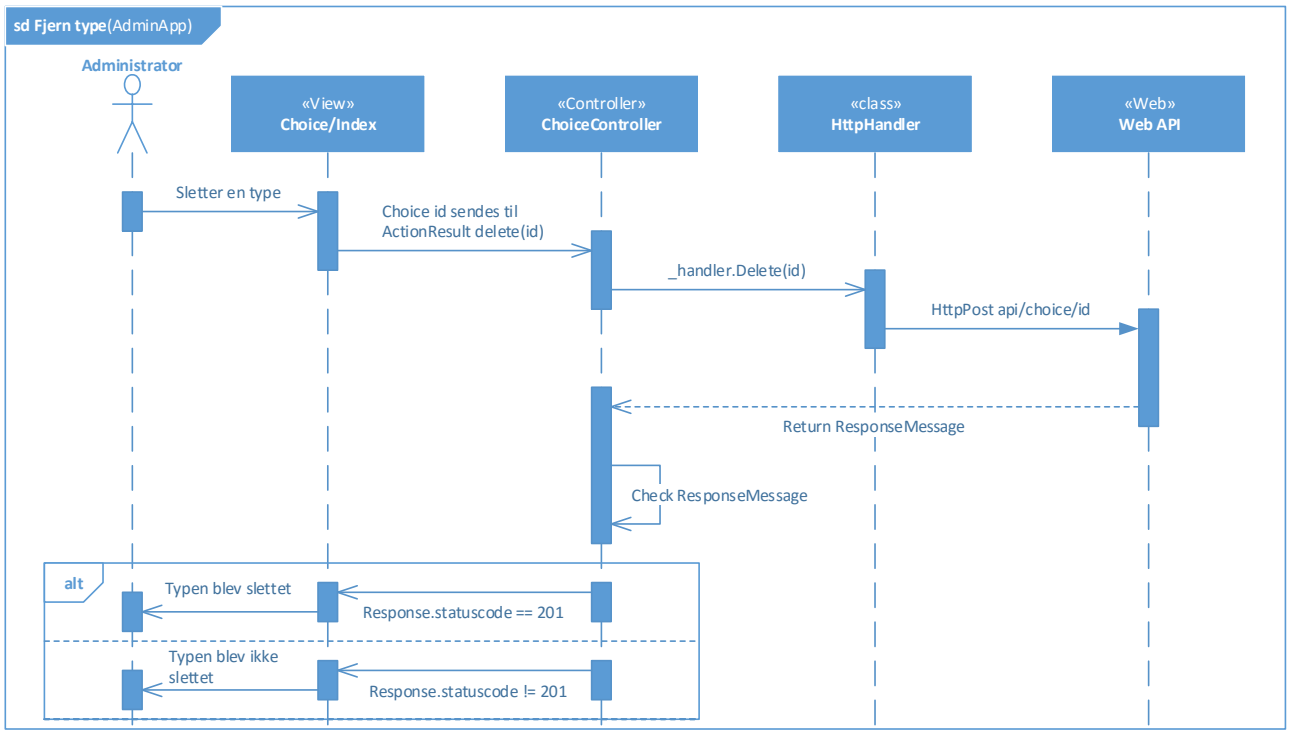
Figur 53 - Sekvensdiagram for fjern kategori AdminApp

Fjern kategori (WebAPI)



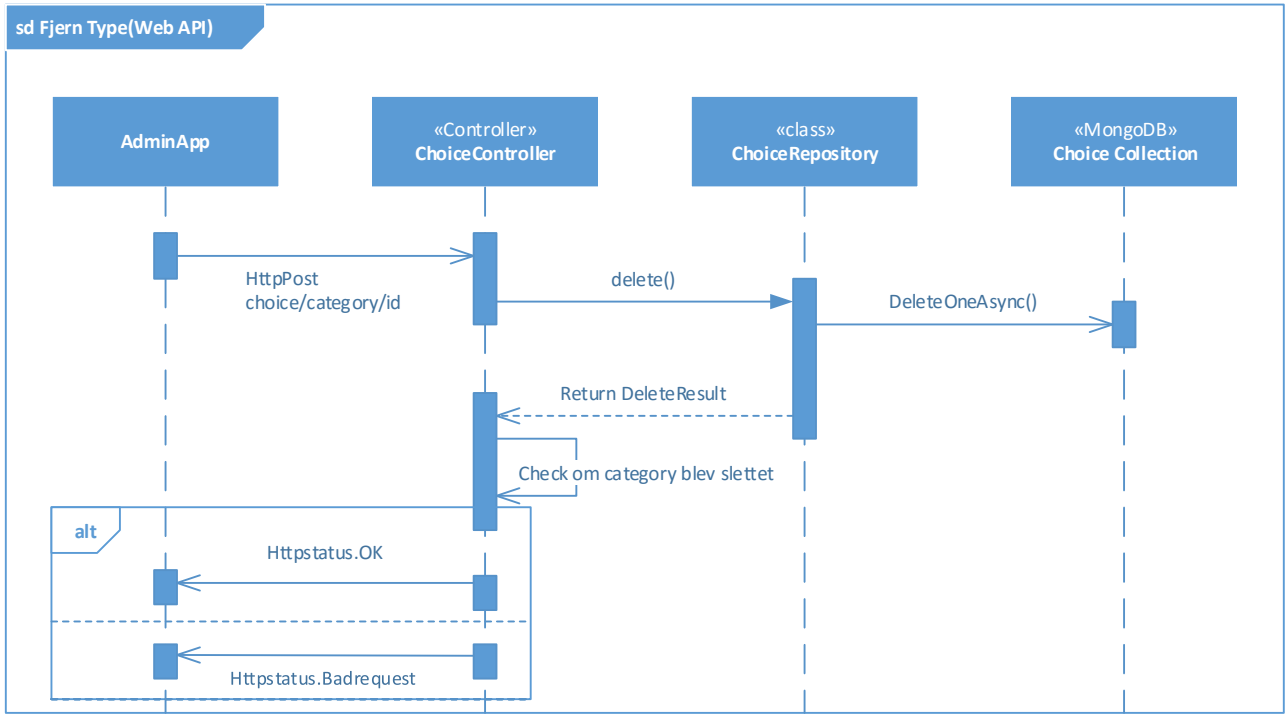
Figur 54 - Sekvensdiagram for fjern kategori WebAPI

Fjern type (AdminApp)



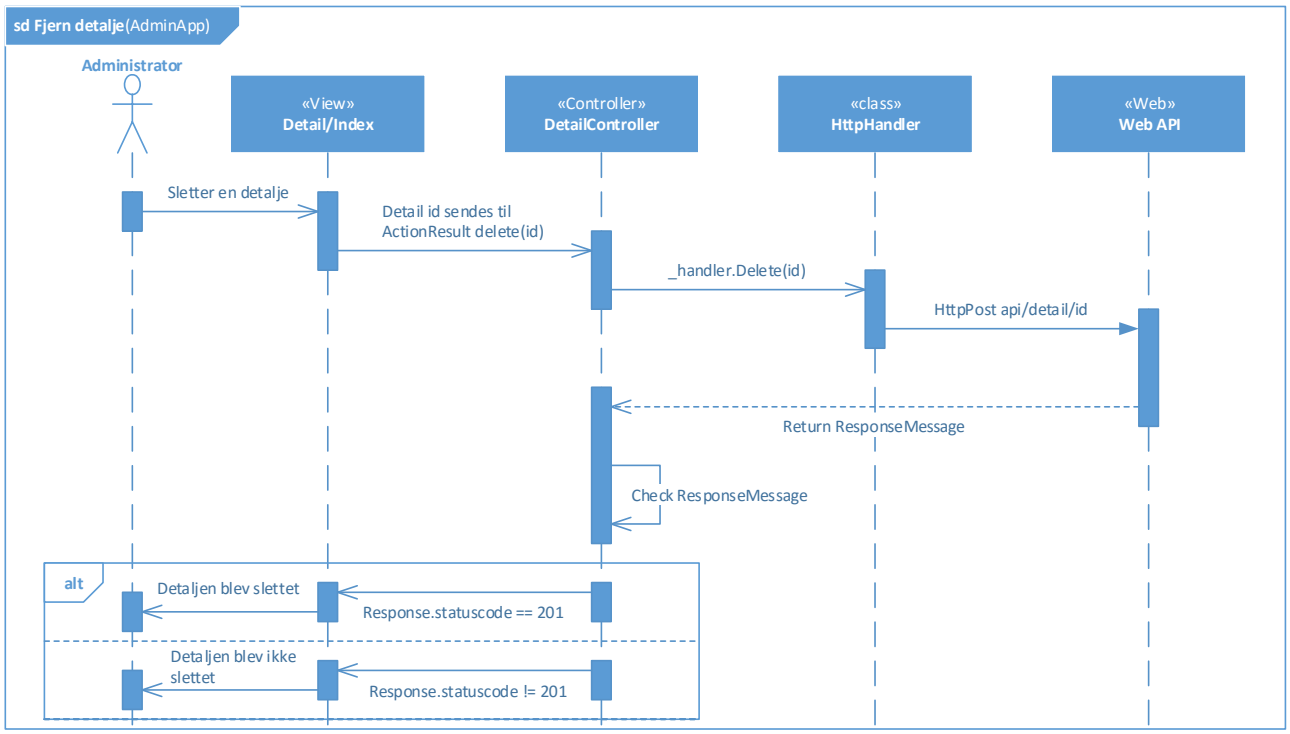
Figur 55 - Sekvensdiagram for fjern type AdminApp

Fjern type (WebAPI)



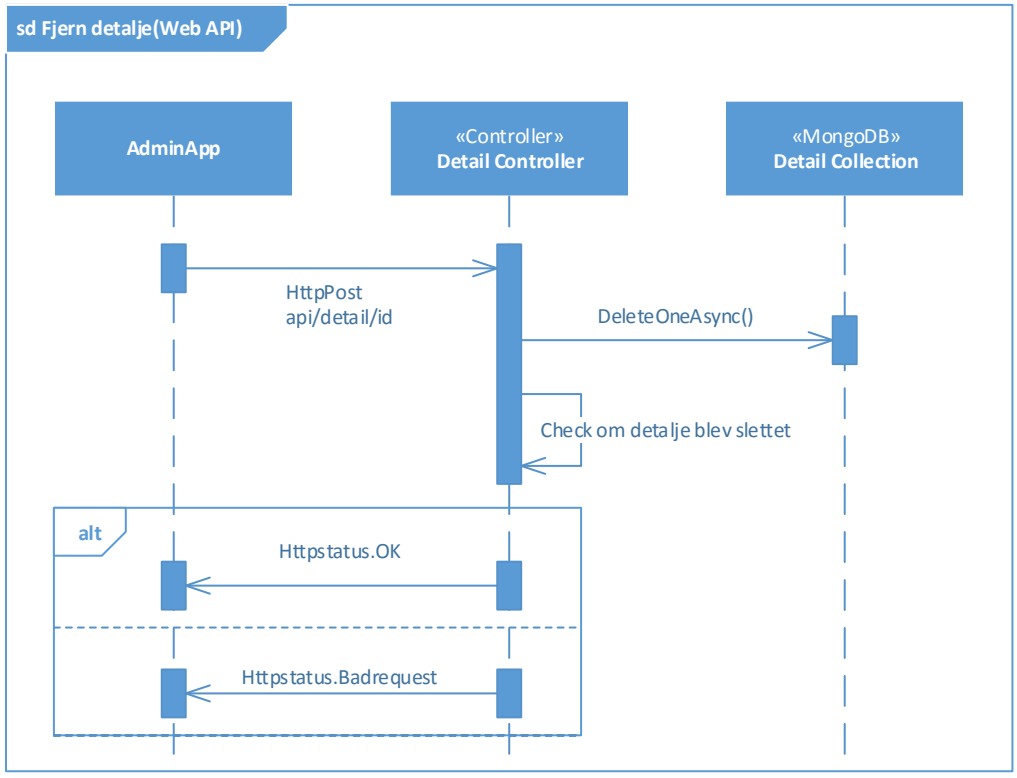
Figur 56 - Sekvensdiagram for fjern type WebAPI

Fjern detalje (AdminApp)



Figur 57 - Sekvensdiagram for fjern detalje AdminApp

Fjern detalje (WebAPI)



Figur 58 - Sekvensdiagram for fjern detalje WebAPI