# Implementering

Her beskrives implementeringen af funktionerne til de moduler som projektgruppen har stået for udviklingen af. Først beskrives den overordnet systemarkitektur for PatientCare. Dernæst beskrives hvilke designløsninger der er blevet gjort for hvert modul for hhv. PatientApp, PersonaleApp og AdminApp.

## Systemarkitektur

Følgende beskriver den overordnede Systemariktektur med udgangspunkt i dokumentationen.



Figur X 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 snakker med MongoDB via dens framework.

### Deployment view



Figur x - Deployment View

Figur X 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 er skruet sammen, og det er 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 for hurtigst muligt at få vist det konceptuelle af projektet - projektgruppen har dog en erfaren iOS udvikler. Derudover kommunikerer SmartPhone(Xamarin) med et WebAPI ved brug af http protokollen.

WebAPI’et udstiller en række tjenester som PatientCare’s subsystemer benytter sig af og 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 fællesdatabasen i MongoDB.

PersonaleApp kører på en smartphone som benytter sig Android styresystem. PersonaleApp er den app personalet på hospitalet bruger til at håndtere patientkald med årsag. Denne app kommunikerer med WebAPI’et ved brug af http protokollen.

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. Denne app 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.

### 3-lags arkitektur

Udover at have et deployment diagram, der viser systemets fysiske domæne for PatientCare, er det en god fremgangsmåde at koncentrere alt kode relateret til fælles logikken og kommunikationen til omverdenen i hvert sit lag. Dette gør nemt at lægge hver partition af systemet i sit eget ansvarsområde og det hjælper til at nedbryde PatientCare i delsystemer. Dette er ofte opbygget som en lagdelt model af pakker og deres indbyrdes sammenhænge. Fordelen ved dette er man har ”separation of concerns”, og det reducer også koblingen og afhængigheder. Dette er illustreret af følgende diagram:



Figur x - PatientCare systemet opdelt i 3-lags arkitekturen

Det ses at PatientCare er opdelt i en 3-lags arkitektur. Hvert modul eller subsystem har sit eget lag. F.eks. ses det i dette tilfælde at PersonaleApp er en pakke, som ligger i præstentationslaget. Åbnes denne pakke vil man kunne se en række andre pakker, som vedrører dette lag.

WebAPI’et er opdelt lidt anderledes da alle 3 moduler tilgår WebAPI’et præsentationslag med http requests for efterfølgende at lave et kald ned i databasen.

## Design

//Beskrivelse af designprocessen. Hvilke designløsninger man har overvejet samt de valg man har gjort og begrundelsen herfor.

Som følge af den beskrevet arkitektur er design af PatientCare næste fase i udviklingsprocessen. For hvert modul beskrives hvilke designløsninger, der er overvejet, samt de valg man har gjort og begrundelsen herfor.

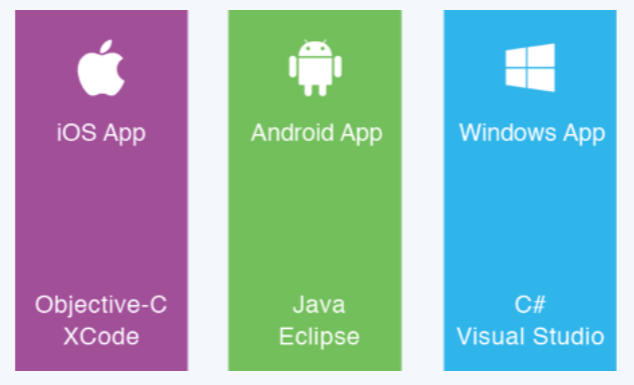
### PatientApp

#### Cross platform

Ud fra kravspecifikationen og modulbeskrivelsen skal PatientApp’en udvikles i et miljø der gør det muligt at understøtte de tre største smartphone styresystemer iOS, Android og Windows, så de fleste af patienternes egne smartphones kan køre App’en. Den traditionelle løsning ville være at udvikle app’en hver for sig specifikt til hver af disse platforme og deres operativ systemer. Dette vil kræve en teknologisk viden til at programmere i 3 forskellige IDE’er, sprog og API’er:

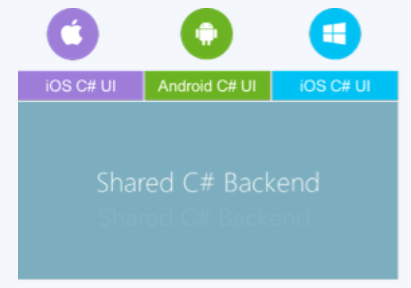
* Xcode skrevet i enten objective-c eller swift.
* Android Studio skrevet i Java.
* Visual Studio skrevet C#

Fordelen ved at benytte denne form for implementering for vores system, er at app’en vil kunne distribureres på alle platforme. Patienten vil ikke skulle være afhængig af at have en bestemt telefon for at kunne installere app’en, og det vil være fremragende for brugervenligheden. Ulempen vil være, at det rent teknisk vil koste meget udviklingstid, da hver styresystem er forskellig og som udvikler kræver det en bred, teknisk viden. Designmæssigt skal der designes en backend og en UI for hver platform.



Figur 1 Løsning uden brug af Xamarin

I stedet for overstående løsning, vil en endnu bedre, men ligeså fremragende løsning være brug af Xamarin platformen. Med Xamarin er det muligt at udvikle native iOS, Android og Windows apps i C# på en fælles kodebase. Her kan bruges den samme IDE, sprog og API alle steder. Alt hvad der kan udvikles i Objective-C, Swift eller Java, kan udvikles i C# med Xamarin. Med Xamarin apps har man adgang til the fulde spektrum af funktionalitet tilknyttet den underliggende platform og device, heriblandt platform specifikke API’er, såsom iBeacon og Android Fragments. Det dækkes 100 % med C#, dvs. man får det fulde udbytte for hver platform. Xamarin platformen kommer i to udgaver: En udgave hvor backenden er delt på tværs af alle platforme, men hvor UI’en skal udvikles for hver. Dette er den traditionelle Xamarin fremgangsmåde. Og en anden udgave er hvor både backenden og UI’en er delt. Dette er med Xamarin.Forms. Den løsning som PatientCare systemet benytter sig af, er den tradionelle Xamarin fremgangsmåde. Den ser således ud:



Figur 2 Løsning med den tradionelle Xamarin fremgangsmåde

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.

De tre overstående punkter lægger op til en arkitektur ligesom der er beskrevet i figur X over den overordnet arkitektur 3-delt 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. Hvert lag indeholder en række pakker, der relaterer sig til hhv. Brugergrænsefladen, domænet og use casene samt de mere kompliceret grænseflader imf. persistering og kommunikationen frem og tilbage imellem WebAPI’et. Dette er beskrevet mere detaljeret i designdokumentet, hvor der også tages udgangspunkt i en use case for at illustrere flowet igennem de forskellige lag.

Den tradionelle Xamarin løsning, hvor der skal være en fælles implementering af backenden og en forskellig implementering af brugergrænseflade, platformsspecifikt, har lagt op til nogle beslutninger vedrørende arkitektur og design. Specielt har designet af en PCL(Portable Class Library) været yderst vigtigt, fordi den afspejler funktionalitet vedrørende persistering og kommunikationten til omverdenen og deles på alle tre platforme. I designdokumentet kan det ses hvordan den fælles database struktur, der ligger i PCL’en er opbygget.

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:

**Hvad skal deles?**

* Indlæsning/gem af data herunder valgmuligheder og mine kald.
* HttpRequest til og fra adminsmodulet (WebAPI) og brugen af Object Relationel Mapping(ORM)
* CPR validering i native koden men også serverside validering via adminsmodulet.
* Andet logik, f.eks. en wrapper klasse til at pakke valgmuligheder til et kald, der senere skal persisteres.

**Hvad skal hver platform implementer?**

* Egen lokal database
* Filstien til hvor hver database lokalt skal ligge.
* Oprettelse til databasen
* UI og andet custom rendering såsom en menuknap eller en dialogboks. (da disse f.eks. jo er forskellig fra platform til platform)

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

De designmæssige valg, der er blevet taget i forbindelse med at persisteringen kommer frem til en teknologi som SQLite gør brug af. Her vil det være muligt at mappe data’en fra lokal databaserne direkte til 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**

Logikken i PatientApp skal kunne bygges på en PCL og deles for et Xamarin.iOS, Xamarin.Android og Xamarin.Windows projekt. Da logikken for PatientApp’en skal kunne bygges på en PCL skal persistering af data dermed også være af samme fremgangsmåde og bygges i samme PCL. For dette er valgt en SQLite-Extension[[1]](#footnote-1), som er en meget simpel ORM der tilbyder alle de relationsships, databasemodellen har brug for på toppen af et sqlite-net[[2]](#footnote-2) bibliotek. Det er open-source, der tillader .NET og Mono applikationer at gemme data i SQLite3 databaser. Med SQLite - Net Extension udvides sqlite3’s funktionalitet og hjælper bedre udvikleren til at håndtere relationsships mellem sqlite-net entities. Med SQLite – Net Extension kan man via attributer på sine properties angive relationerne. Og man skal ikke oprette tabeller eller kolonner i databasen. Af denne grund, har man fuld kontrol over databasens skema til at persistere entiterne. SQLite-Net Extensions kræver kun, at man angiver fremmede nøgler, der anvendes til at håndtere relationer og resten finder SQLite den selv ud af.

Et eksempel på brugen af SQLite-Net Extension er metoden GetAllWithChildren<Category>. Metoden kigger på alle de relationships der er specificeret i databasemodellen, finder eventuelle fremmednøgler og automatisk fylder properties på entititen. Man slipper altså for at skrive queries, som man normalt ville med sqlite3.

Som nævnt, skal hver platform implementere en metode til at returnere filstien for hvor sqlite databasen lokalt skal ligge. Og hver skal implementer en metode til at oprette forbindelse til deres sqlite database. Dette kræver et interface som implementeres af tre platformsspecifikke klasser med hver af deres SQLiteConnection. I DAL laget ligger der en klasse, der har en række metoder som skal bruges ifm. Indlæsning/persistering af data fra SQLite databaserne. Denne klasse skal også bruges i de tre platformsspecifikke klasser. Dette kan ses i klassediagrammet for PCL’en i designdokumentet.

For at illusterer hvordan de forskellige lag(GUI, BLL, DAL) snakker sammen er der taget udgangspunkt i Use Case 1.3: Opret kald. Kommunikationen imellem disse tre lag kan vises med et sekvensdiagram der beskriver sekvensen fra at en patienten opretter et kald, til at patienten kan se at kaldet er gået igennem systemet med success. Klasserne der kommunikerer med hinanden kan genses i designdokumentet i arkitekturen over klassediagrammet for PCL og GUI.



Figur 3 Sekvensdiagram for Opret kald

Sekvensdiagrammet følger følgende flow: Patienten har behov for at oprette et kald. Patienten vil finde ud af, at der er flere steps han/hun skal igennem afhængig af, om kaldet har behov for at blive specificeret yderligere. Dette foregår via brugergrænsefladen via en tabeloversigt illustreret i Figur X som en pakke af grafiske elementer, da tabellernes opbygning er platformsspecifikt. Efter patienten har valgt en valgmulighed på brugergrænsefladen, sker der en række andre ting som vedrører business logikken, bl.a. CPR validering og tjek på, om patienten er registreret i PatientCare systemet. Derefter oprettes kaldet i ”PatientCall” klassen, hvor der laves en JSON repræsentation af kaldet. Denne JSON repræsentation sendes videre til klassen ”CallManager”, der sørger for at poste det videre til ”HttpHandler” klassen. Denne klasse står for at opstille http klienten med request headers, body og url’en til WebAPI’et sammen med det indhold, der skal sendes af Content-Typen ”application/json”. WebAPI’et giver derefter en response kode. Er denne ”OK” betyder det, at kaldet er blevet oprettet. I responsen fra WebAPI’et gives der et id på kaldet, der netop er blevet oprettet i MongoDB. Dette id skal gemmes sammen med dette kald, der netop er blevet oprettet af patienten. På denne måde er id’et på kaldet ens i både lokaldatabasen men også på fællesdatabasen på MongoDB. Dette er vigtigt, da samme kald senere skal opdateres enten i form af, at det fortrydes af patienten eller udføres af personalet.

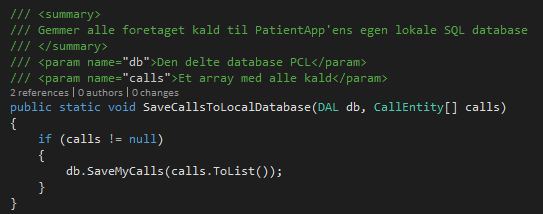
I Figur X vises ikke sekvensen for hvordan kaldet persisteres i lokaldatabasen efter det er blevet oprettet. Dette vises i et separat sekvensdiagram som følgende:



Figur 4 Sekvensdiagram der viser flowet for persistering af et kald

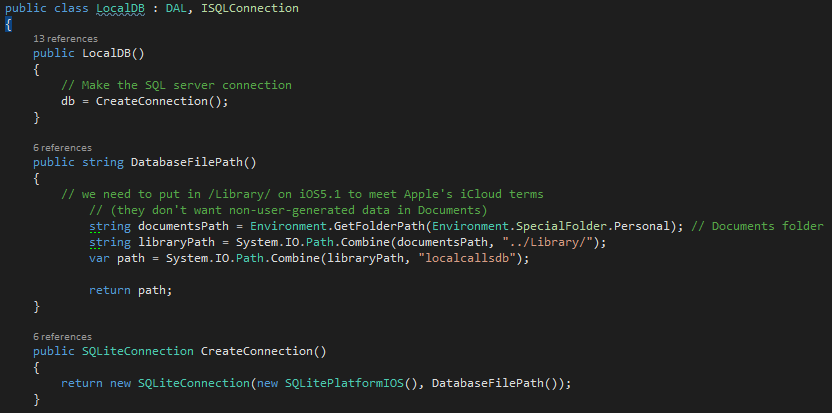
Efter responsen fra WebAPI’et vist i Figur X, tilføjes kaldet sammen med det returnerede MongoDB id til en liste af andre kald, som lokaldatabasen har persisteret. Listen sendes med metodekaldet til klassen ”Datahandler” sammen med en reference til den platformsspecifikke, lokaldatabase klasse. Herefter kaldes en metode på denne reference til at sende kaldet videre til den platform, som nedarver fra klassen ”DAL”. Sidst sker selve indsættelse af kaldet via en metode ”InsertOrReplaceAll” fra SQLite API’et.

Et kodeudsnit fra ”Datahandler” hvor ”SaveCallsToLocalDatabase” kaldes er vist her:



Figur 5 Metode til at persistere kald ligger I Datahandler klassen, der ligger I PCL’en.

”db” objektet som sendes med her er af typen DAL som hver platform skal nedarve fra. Et eksempel på hvordan det fungerer ses i klassen LocalDB:



Figur 6 Klasse som alle platforme skal benytte til at oprette en SQLiteConnection sammen med filstien til databasen.

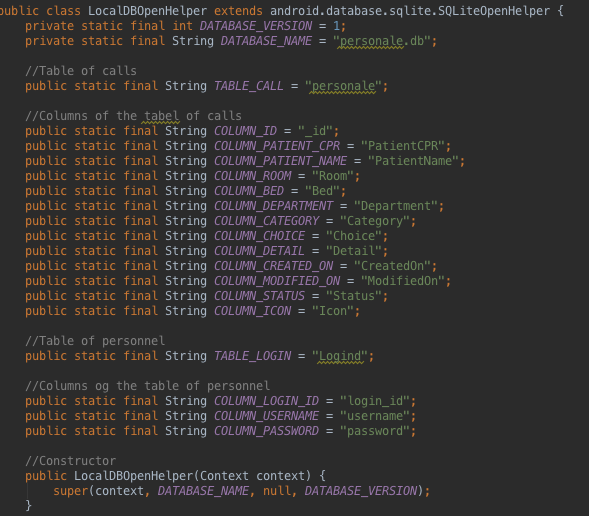
Eksemplet viser hvordan ”DAL” og ”SQLiteConnection” klassen implementeres, hvori constructoren angives en implementering af ISQLitePlatformen og filstien til lokaldatabasen. LocalDB klassen skal ligge sammen med Xamarin.iOS projektet og kopiers i Xamarin.Android og Xamarin.Windows projektet. Her skal ISQLConnection og DAL implementeres på ny og de samme metoder overskrives på samme måde som i Figur X. Sammen med overstående to metoder, som hver platform skal overskrive, får de også hver især de samme metode definitioner fra ”DAL” klassen. Disse metoder kan ses i klassediagrammet i Figur X.

### PersonaleApp

PersonaleApp er udviklet på baggrund af at Systematics opgavesystem til serviceopgave ikke kan håndtere patientkald. PersonaleApp skal håndterer modtagelsen af patientkald og ændringen af status på patientkaldene eftersom de opgaver der er forbundet med dem bliver udført. PersonaleApp modtager patientkaldene fra modulet PatientApp via WebAPI’et.

### SQLiteDatabase

Android har forskellige måder at gemme bruger-og applikationsdata. En måde er at gøre brug af SQlite database. Det er anvendt i PersonaleApp’en til persistering af data fordi der er behov for at gemme flere parametre for et kald lokalt og nemt tilgå det ved at kendskab til et id. En måde 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.



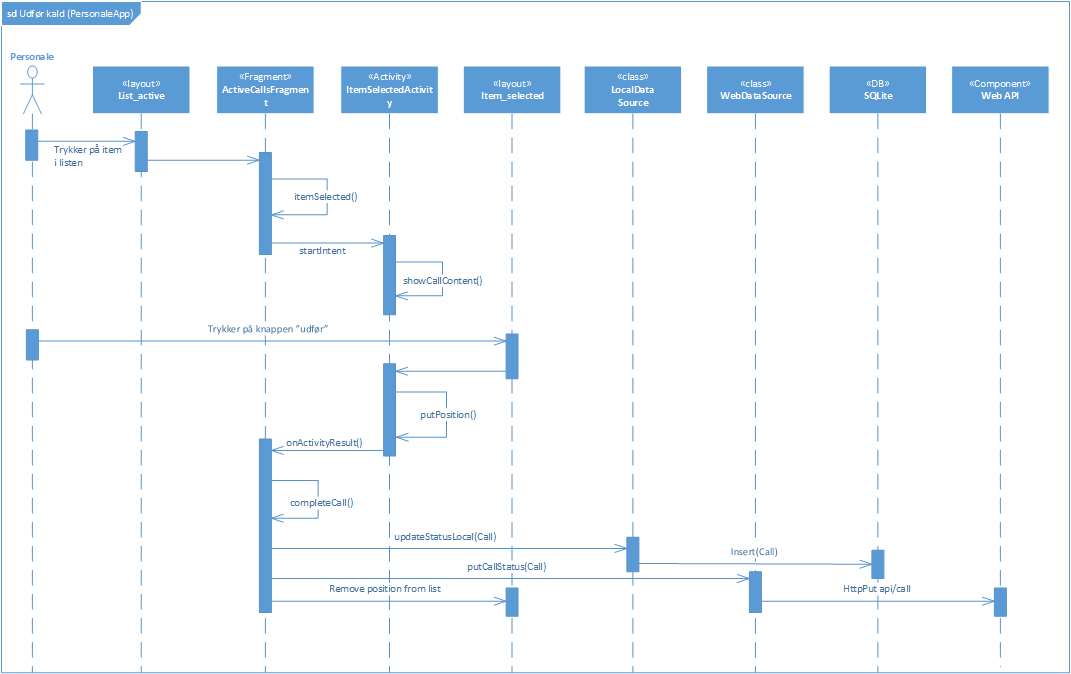
Figur 7 Tabelstruktur for SQLitedatabasen for PersonaleApp

I figure 7 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)

### 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 8 Sekvensdiagram for Udføre kald for PersonaleApp

### Bound Service (Modtag kald)

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 fremfor performance, er der valgt en pull funktionalitet til at indlæse nye kald fra systemet. Denne pull funktionalitet kan enten laves ved at der trykkes på en knap eller 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.

”MainActivity” er den aktivitet der starter op når brugeren af PersonaleApp er logget ind. I Mainactivity starter servicen også. Servicen henter alle afventede kald, dvs. de kald der ikke er blevet udført. De afventende kald hentes fra WebAPI’et via et http request til en URL. WebAPI’et returnerer en string af json objekter som deserialiseres til konkrete objekter – på samme måde som beskrevet for PatientApp’en .



Figur 9 Diagram der viser strukturen for en Bound Service for PersonaleApp

I figur 29 ses det hvordan servicen bliver oprettet i PatientApp. Til højre i diagrammet ses tidslinjen for aktiviteter og andet logik der kører i hovedtråden. Og til venstre ses tidslinjen for aktiviteter der kører i en separat tråd. Da en service som default kører i hovedtråden er der behov for at oprette en anden tråd, som servicen skal køre i, da servicen vil komme til at lave blokerende operationer. Bindingsprocessen ses i figur 29 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.

### AdminApp

For AdminApp har designprocessen taget udgangspunkt i kravspecifikationen i afsnittet for AdminApp fully dressed use cases hvilket har givet anledning til sekvensdiagrammer som er at finde i Bilag 4. Sekvensdiagrammerne er lavet ud fra tankegangen om at AdminApp var forudbestemt til at skulle være en web applikation og at applikationen skulle kommunikere med et web API for at kunne løse de opstillede use cases for AdminApp. Det er i sekvensdiagrammerne for AdminApp ikke muligt at identificere de implementerede funktionskald, da diagrammerne og implementeringen ville være hårdt koblet sammen og ikke give anledning til ændringer. Med andre ord, hvis en use case ændres så det vil ændre sekvensen af use casens udførelse, vil det kræve ændring i diagrammet også. Det eneste sted der i diagrammet er taget stilling til hvilket funktionskald der skal ske, er i klassen ”HttpHandler” som er her kommunikationen med web API’et sker og måden AdminApp kommunikerer med web API’et ændres ikke umiddelbart.

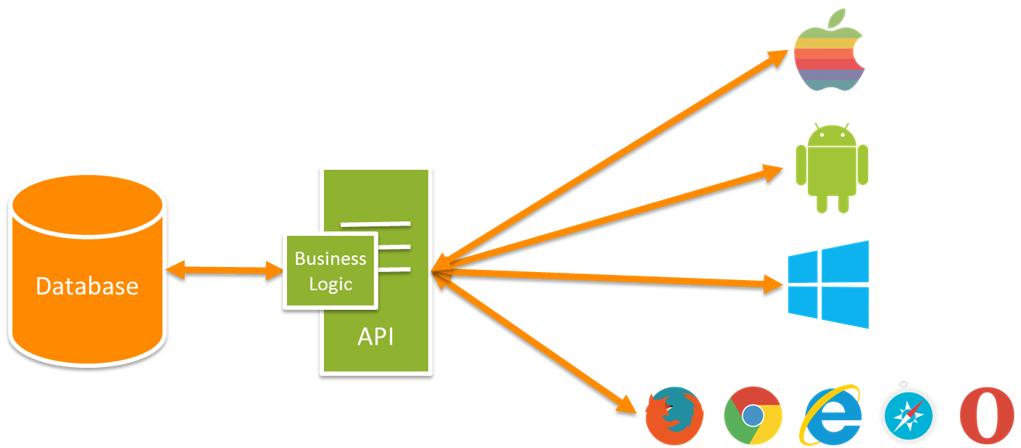
Views, Controllers og Models i AdminApp, følger MVC standarden og hvor views er opdelt efter hvilken del af applikationen de står for. F.eks. ligger alle views som har noget med ”category”(Kategori) at gøre, i en mappe der hedder ”Views/category”.

Selve designet for indholdet af de forskellige views er der ikke tænkt videre over. Der har været behov for at oprette, se og slette nogle data og sidernes layout er lavet efter dette behov.

### Web API

WebAPI’et udnyttes i kommunikationen mellem modulerne fx når patienten skal se de foruddefinerede valgmuligheder på PatientApp, når patienten opretter et nyt kald og personalet skal modtage og når personalet udfører kaldet på PersonaleApp. Kommunikationen foregår ved at data hentes fra eller leveres til en webside som WebAPI’et udstiller.

Figur X viser princrippet i et API Driven Development, som projektgruppen har udnyttet.



Figur 7 http://blogs.msdn.com/cfs-filesystemfile.ashx/\_\_key/communityserver-blogs-components-weblogfiles/00-00-00-56-73/2318.WithAPIArchitecture.PNG

Fordelen ved at bygge systemet op omkring et *API driven development*, er at modulerne gøres uafhængige af hinanden, så der opnås lav kobling, hvilket kan lede til bedre performance og bedre vedligeholdelse.

Hvis der skal implementeres en ny funktionalitet gør API Driven development det simpelt. Der tages udgangspunkt i et eksempel hvor kravet fra et should have krav i MoSCoW-prioriteringen   
SH1-5: som er et krav om at det skal sikres at patientkaldet kun kan foretages fra afdelingen hvor patienten er indlagt og altså ikke udenfor sygehusets område. Implementeringen af dette vil ske dels i PatientApp og dels i WebAPI’et.

Den måde WebAPI’et er struktureret med den fælles database på gør det nemt at benytte WebAPI’et til at håndtere valideringen af om patientkaldet sendes indenfor hospitalets rammer eller ej, mens den fejlbesked der skal gives til patienten hvis patienten ikke befinder sig på afdelingen når kaldet oprettes skal implemeneres i PatientApp’en.

For web API’et har designprocessen taget udgangspunkt i alle use cases i kravspecifikationen hvor en device har behov for en integration ud af eget system. Det er f.eks. ved oprettelse af et kald fra patientApp.

Det er forsøgt at udvikle web API’et først efter tankegangen om *API Driven Development*, og for dermed at have en klar datamodel for resten af systemet, men på grund af database skifte, hvilket der kan læses mere om i afsnit x, gjorde at udviklingen af web API’et måtte gentænkes og på daværende tidspunkt var udviklingen af resten af systemet godt i gang. I den efterfølgende udvikling af web API’et har der været løbende design og implementering efter systemet behov. F.eks. ved start af nu iteration, er der startet ved web API’et og arbejdet ud.

Web API’et er bygget i ASP.NET web API 2 som bygger på MVC frameworket, og derfor er opbygningen af mappe strukturen derefter. Denne template kommer med et forebygget *Area,* som står for at genererer hele dokumentations viewet. Dette *Area* benytter sig af Visual Studio *Summaries* og funktionsdefinitionerne i implementeringen til at danne et view der til dels virker optimalt.

### Database

Når en patient fx opretter et kald via PatientApp genereres der noget data om hvad årsagen til kaldet er. Dette skal kobles sammen med hvem patienten er og hvor patienten er indlagt før det modtages af personalet. Her beskrives oprettelsen af et patientkald for at vise princippet i hvordan data genereres på PatientApp’en, distribueres i databasen og benyttes på PersonaleApp’en.

Når et patientkald oprettes på PatientApp leveres det med tilhørende parametre til WebAPI’ets webside. Herfra gemmes kaldets parametre i den fælles database og knyttes sammen med patientens mock data ud fra CPR-nummeret som er tilknyttet kaldet. Fra WebAPI’et hentes alle parametrene for dette kald ned på PersonaleApp’en, hvorfra personalet modtager kaldet med alle oplysninger om hvem patienten er, hvor patienten er indlagt, hvad årsagen til kaldet er og hvornår det er oprettet med mere.

Hvis et patientkald oprettes med succes, afhængig af om patientens CPR-nummer findes i den fælles database, bliver kaldets status bliver sat til 0. Kaldets status er en C# enum, hvis denne er 0 er kaldet afventende, er den 1 er kaldet udført og er den 2 er fortrudt.



På figur X ses en oversigt over hvor de forskellige parametre kommer fra.

I designprocessen af databasen har projektgruppen skulle blive enige om, hvordan datamodellen skulle se ud for systemet med udgangspunkt i hvilke data en patient har behov for, for at kunne oprette et patientkald og med ligeledes hvilke data personalet har behov for, for at kunne udføre et patientkald. Denne datamodel ligger til baggrund for de modeller web API’et modtager og returnerer.

Det har givet anledning til 2 databaser, hvor den ene er en datamock som skal forestille data som det regnes med at kunne få fra Cetrea ved en evt. integration til deres systemer. Denne database består primært af en collection hvor information om patienten og hvor på sygehuset patienten er indlagt.

Den anden database indeholder de data som et patientkald benytter sig af ved oprettelse et patientkald, kategorier, typer og detaljer og har hver deres *collection* hvor disse data er opbevaret.

#### Datamock

PatientCare er udviklet med henblik på at trække data fra et system som Cetreas kliniske logistik hvor brugbare oplysninger i forvejen registreres. For at PatientCare kan blive en realitet i praksis er det derfor nødvendigt at være understøttet af et system som dette, der leverer de nædvendige oplysninger. Da det ikke har været muligt at integrere til Cetrea i dette procesforløb har projektgruppen opsat et mock data af det data der skal bruges til prototypen af PatientCare, som ligner det data Klinisk Logistic kan levere.

Datamocken består af data der indebærer følgende:



1. https://bitbucket.org/twincoders/sqlite-net-extensions [↑](#footnote-ref-1)
2. https://github.com/praeclarum/sqlite-net [↑](#footnote-ref-2)