Kandidatarbete

vårterminen 2019

Linköpings universitet

---

# Egna hälsodata

## Teknisk dokumentation

---

2019-06-04

**Utvecklad av**
Noah Hellman
David Lantz Sofie Liljedahl
Erik Norrestam Held
Matilda Olsson Kaalhus
Mattias Salo
Alexander Vestin

**Utvecklad åt**
Region Östergötland

# Innehåll

# 1 Inledning

Detta dokument syftar till att ge en inblick i hur webbapplikationen för inrapportering av hälsodata är strukturerad och vilka utvecklingsmöjligheter som finns för produkten. Produkten är skapad av en grupp på sju studenter som en del av deras kandidatarbete på uppdrag av Region Östergötland och har skett i samråd med dem.

## 1.1 Syfte

Denna tekniska dokumentation ämnar att underlätta vid underhåll och vidareutveckling av webbapplikationen för att rapportera in hälsodata. En översiktlig arkitektur kommer presenteras som tolkningsstöd av koden och specifika exempel för vidareutveckling kommer beskrivas. Dessa exempel baseras på vad den ursprungliga utvecklingsgruppen ansåg vara mest centralt vid vidareutveckling.

Den tänkta målgruppen för detta dokument är utvecklare som inte tidigare arbetat med applikationen. Därför läggs fokus först på att ge en överblick som ska tillåta läsaren att snabbt sätta sig in i strukturen som beskrivs i sektion 3. I sektion 5 följer sedan konkreta guider av de steg i vidareutvecklingsprocessen som originalutvecklarna anser behöver en djupare beskrivning.

## 1.2 Installation och användning

Applikationen är byggd med ramverket Angular och använder därmed dess uppsättning av verktyg känt som *Angular CLI* för byggning och testning. Applikationen använder *npm* för att hantera paket. Kommentarer i koden är formaterade så att TypeDoc går att använda för att generera dokumentation.

### 1.2.1 Bygga applikationen

För att först installera alla paket som behövs körs kommandot

```
npm install
```

i rotmappen av projektet. Det skapar en mapp `node_modules` som innehåller alla dependencies till applikationen.

Därefter kan *Angular CLI* användas för att kompilera applikationen. För att skapa en distruerbar kompilerad version används kommandot

```
ng build
```

i rotmappen. Då skapas en kompilerad variant och läggs i mappen `dist` med endast statiska filer.

### 1.2.2 Utveckling

För utveckling och felsökning går det att använda kommandot

```
ng serve
```

som bygger applikationen och serverar den på `localhost:4200` så att applikationen går att komma åt med en webbläsare. Om någon av filerna ändras byggs applikationen om automatiskt och sidan laddas om.

Kommandot

```
ng test
```

används för att köra alla enhets- och integrationstester.

För mer övergripande tester körs kommandot

```
ng e2e
```

som används för att köra alla end-to-end tester.

För att visa och eventuellt fixa kodstilsfel används kommandot

```
ng lint --fix
```

i applikationens rotmapp. Kodens stilguide och lintingens regler konfigureras i filen `tslint.json`

### 1.2.3 Dokumentation

Kommentarerna i koden kan genereras till dokumentation med hjälp av TypeDoc. Detta utförs genom att anropa kommandot

```
npm run docs
```

i applikationens rotmapp. Därefter läggs dokumentationen i mappen `docs` som kan läsas genom att öppna `docs/index.html` med en webbläsare. Konfiguration till TypeDoc finns i `typedoc.json` i projektets rotmapp.

### 1.2.4 Google API key

För nuvarande används en API-nyckel som skapats på ett konto som specifikt användes för olika testningssyften under projektet. Id:t för API-nyckeln återfinns i `google-fit-config.ts`. Eftersom denna nyckel används för att definiera vilka URL:er som är auktoriserade att användas för hemsidan kan det vara av intresse att skapa en ny API-nyckel där nya URL:er kan auktoriseras. Detta görs enklast genom att följa stegen på `https://developers.google.com/maps/documentation/javascript/get-api-key`. Därefter måste `client_id`-fältet i `google-fit-config.ts` ändras till det genererade id:t.

### 1.2.5 Konfiguration av EHR

Filen `ehr/ehr-config.ts` innehåller ett EhrConfig-objekt som väljer inställningar till journalsystemet. I objektet finns ett fält `baseURL` som ska sättas till den URL som API-anropen för journalsystemet ska göras mot. Det finns även ett fält `templateId` som specificerar vilken *template* som ska användas för att skapa kompositioner.

# 2 Teori och referenser

De resterande kapitlena av dokumentet är skrivna utifrån att läsaren är bekant med openEHR och ramverket Angular. Om det inte är fallet följer i det här kapitlet en kort beskrivning av viktiga koncept och referenser till mer detaljerad dokumentation.

## 2.1 openEHR

OpenEHR är en väldigt stor standard men det här projektet behandlar endast en liten del av den. Alla standardens olika delar går att läsa på `https://specifications.openehr.org/`. Koncept som är centrala för applikationen är framförallt kompositioner, arketyper, *templates* och elements datatyper.

Standarden är uppdelade i flera komponenter. Det här projektet behandlar framförallt "Reference Model" (RM) och "Archetype Model" (AM). Mer information om standardens arkitektur finns i openEHR:s arkitekturöversikt.[1]

**Arketyper** är datastrukturer som definierar hur all data ska struktureras i en journal. Arketyper för noteringar (*entry*) kan vara av typen *observable*, *evaluation*, *instruction* och *action*. Eftersom applikationen använder sig av självrapporterad data är det endast *observable* som används för noteringar. Ett exempel på en *observable* som används är "Blood pressure". Arketyper kan också beskriva kompositioner. Arketyper definieras väldigt allmänt och är tänkta att användas på global nivå. Arketyper definieras i AM.[2]

**Kompositioner** är ett inlägg i en journal. En komposition består av en kompositionarketyp som kan innehålla noteringar såsom "Blood pressure". I applikationen används endast arketypen "Self monitoring" eftersom all data är självrapporterad. Kompositioner defineras i RM.[3]

**Templates** bestämmer vad en komposition kan och ska innehålla. Den sätter ytterligare begränsningar på kompositionen än vad arketyperna gör. Exempelvis har en *template* använts i projektet som lägger till en *observable*-arketyp för varje tillgänglig kategori och därefter tar bort alla element som inte är relevanta. *Templates* definieras i AM.[4]

**Datatyper** definierar hur individuella element av arketyper kan se ut. Exempel på datatyper är `DV_QUANTITY` och `DV_TEXT` som representerar kvantiteter och strängar som exempelvis en vikt eller en kommentar. Datatyper är definerade i "Data Types Information Model" som är del av RM.[5]

## 2.2 Angular

Projektets applikation är byggd i ramverket Angular. Nedan följer en kort beskrivning av de koncept från Angular som använts mycket i projektet. Referenser till vidare dokumentation tillhandahålls också ifall mer information eftersökes.

[1] `https://specifications.openehr.org/releases/BASE/latest/architecture_overview.html`
[2] `https://specifications.openehr.org/releases/AM/latest/Overview.html`
[3] `https://specifications.openehr.org/releases/RM/latest/ehr.html#_compositions`
[4] `https://specifications.openehr.org/releases/AM/latest/AOM2.html#_templates`
[5] `https://specifications.openehr.org/releases/RM/latest/data_types.html`

### 2.2.1 Components och services

Applikationer består av *Components* eller komponenter som definierar skärmelement som modifieras enligt komponentens logik. Varje komponent beskrivs med hjälp av TypeScript, HTML och CSS. Applikationer har även *services* eller tjänster som komponenter kan använda för funktionalitet som inte är direkt specifik till den komponenten, exempelvis HTTP-förfrågningar. Ett exempel på en komponent i applikationen är `toolbar` som bestämmer hur panelen i toppen av applikationen ska se ut och fungera. Ett exempel på en tjänst är `EhrService` som hanterar kommunikation mot journalen.

Vidare information om komponenter och tjänster finns i Angulars officiella dokumentation om dess arkitektur.[6]

### 2.2.2 Dependency Injection

För att kunna använda en tjänst måste komponenten injicera den. Detta är ett mönster inom Angular som kallas för `Dependency Injection`. Injicering sker genom att lägga till tjänsten som ett argument till konstruktorn och därefter tilldela den till en instansvariabel så att metoder kan komma åt den. Mer information om mönstret finns i Angulars dokumention.[7]

### 2.2.3 Filstruktur

Angular har starka åsikter och konventioner. En av dessa konventioner som upprätthålls av Angular och som utvecklare tvingas följa är applikationens filstruktur.

Varje kompontent består av åtminstone tre filer; exempelvis `toolbar.component.html`, `toolbar.component.scss`, `toolbar.component.ts` som innehåller komponentens HTML, CSS respektive TypeScript. CSS kan även beskrivas med en `.css`-fil men i det här projektet används SASS. Tjänster består av åtminstone en fil såsom `ehr.service.ts`.

Nästan alla applikationens komponenter och tjänster har även en motsvarande specifikationsfil såsom `toolbar.component.spec.ts` och `ehr.service.spec.ts`. Dessa filer specificerar enhetstester för komponenten eller tjänsten. Integrationstester definieras i mappen `e2e/src`.

Mer detaljer om Angulars filstruktur finns i Angulars stilguide.[8]

### 2.2.4 RxJS och Observable

Angular använder sig av ett bibliotek som heter RxJS (Reactive Extensions for JavaScript) för att hantera asynkron programmering. RxJS använder sig av så den så kallade reaktiva programmeringsparadigmen. Projektet använder sig av `Observable`-objekt från RxJS för att utföra HTTP-förfrågningar till hälsoplattformar och till journalsystemet.

När en metod måste utföra en HTTP-förfrågan för att utföra sin uppgift returneras alltid en `Observable`. Ett exempel är `getPartyId` i `ehr/ehr.service.ts` som ska få tag på ett `partyId` utifrån ett personnummer. Metoden utför inte sin förfrågan och returnerar strängen direkt. Istället skapar den en `Observable` som anroparen kan prenumerera till genom att anropa dess metod `subscribe`. Först när `Observable`-objektet prenumereras på utförs förfrågan och strängen returneras.

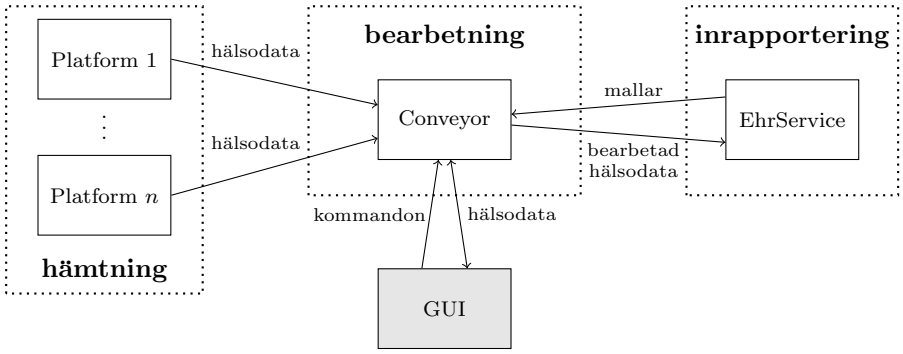
[6] `https://angular.io/guide/architecture`
[7] `https://angular.io/guide/dependency-injection`
[8] `https://angular.io/guide/styleguide`

För mer information om RxJS hänvisas läsaren till Angulars introduktion till RxJS[9] samt utvecklargruppens egna dokumentation som finns tillgänglig på deras hemsida `http://reactivex.io/`

[9] `https://angular.io/guide/rx-library`

# 3 Översikt

Programmet kan delas upp i tre delar: hämtning, bearbetning och inrapportering. Figur 1 visar vilka moduler som ingår i varje del och vilken data som flödar mellan dem.




Figur 1: Översiktliga delar och dataflödet mellan dem.

Alla händelser styrs utifrån kommandon från det grafiska gränssnittet i GUI-modulen. GUI-modulen kommunicerar endast med Conveyor, som fungerar som en fasad, vilket innebär att arkitekturen till viss grad följer ett fasadmönster. Fasaden ser efter vilka datakategorier och plattformar som finns tillgängliga.

## 3.1 Programflöde

Vid ett typiskt programflöde frågar GUI-modulen fasaden först vad det finns för plattformar att hämta data ifrån. Därefter väljer GUI-modulen en plattform och frågar fasaden vilka kategorier av hälsodata som finns tillgängliga för den plattformen. Fasaden får en lista av mallar för kategorier från EhrService. Fasaden kollar därefter vilka av dessa mallar som stöds av den valda plattformen och returnerar dem för utskrivning i GUI:et.

GUI-modulen väljer därefter en kategori och ber fasaden att hämta datan. Då anropas implementationen för plattformen och datan lagras i fasaden. Därefter kan GUI-modulen hämta och modifiera datan i fasaden för uppvisning och komplettering från användaren. När användaren har bearbetat och granskat hälsodatan ber GUI-modulen fasaden att skicka upp datan vilken i sin tur låter EhrService skicka datan till patientens journal.

## 3.2 Moduler och filstruktur

Den övergripande filstrukturen kan ses i figur 2. Applikationen följer huvudsakligen Angulars filstruktur som beskrivet i sektion 2.2.

```
src
|-- app
|   |-- app.component.ts|html|scss|spec
|   |-- conveyor.service.ts|spec
|   |-- ehr
|   |   |-- datalist.ts|spec
|   |   |-- datatype.ts|spec
|   |   |-- ehr-config.ts
|   |   `-- ehr.service.ts|spec
|   |-- gui
|   |   `-- ..
|   |-- platform
|   |   |-- dummy.service.ts
|   |   |-- gfit.service.ts|spec
|   |   `-- platform.service.ts
|   `-- shared
|       `-- ..
|-- index.html
|-- main.ts
`-- ..
```

Figur 2: Övergripande filstruktur av applikationens källkod.

Utöver det har programmets filer delats upp i tre delar: `platform`, `gui` och `ehr`. Hämtningen av hälsodata hanteras i `platform`, som innehåller en superklass `Platform` samt en specifik underklass för varje plattform såsom `GfitService`. Inrapporteringen av hälsodata och hanteringen av all kommunikation mot RÖD sköts i `ehr`, som även innehåller de datastrukturer som är baserade på openEHR. Till sist återfinns applikationens grafiska komponenter i `gui`.

`app` innehåller även en mapp `shared` som i sin tur innehåller filer som ej är bundna till en specifik modul, exempelvis en klass som specificerar en tidsperiod.

# 4 Systembeskrivning

I denna sektion följer en ingående beskrivning av systemets olika moduler och komponenter. Först beskrivs openEHR-modulen samt dess strukturer och gränssnitt, följt av plattformsmodulen och de specifika hälsoplattformarna som är implementerade, och till sist behandlas det grafiska användargränssnittet.

## 4.1 EHR: mot journal

EHR-modulen hanterar framförallt inskickning av hälsodata till journalen. Den innehåller också datastrukturer och funktioner som är baserade på openEHR-standardens datatyper. Totalt innehåller modulen fyra filer: `datatype.ts`, `datalist.ts`, `ehr.service.ts` och `ehr-config.ts` som beskrivs i detalj i den här sektionen.

### 4.1.1 datatype.ts

Filen `datatype.ts` definierar de datatyper som är baserade på openEHR. Det finns två huvudsakliga objekt: den abstrakta klassen `DataType` och gränssnittet `CategorySpec`.

**DataType** definierar vad en datatyp ska innehålla och hur den hanteras. Varje underklass motsvarar en DataType i openEHR. Exempelvis motsvaras `DV_QUANTITY` och `DV_DATE_TIME` av underklasserna `DataTypeQuantity` och `DataTypeDateTime`. `DataType` har metoder som `isValid` som avgör om ett visst värde är ett giltigt värde för den datatypen. För `DataTypeQuantity` ser den till värdet är ett tal inom det angivna intervallet. Datatypsklassen sköter även hur värden jämförs med `equal` och `compare` samt hur värden konverteras till JSON med `toRest`.

Instanser av datatypsklasser motsvarar ett element i en arketyp eller kompositions-*template*. Elementets egenskaper anges vid instansiering. Olika datatyper kan ha olika egenskaper: `DataTypeCodedText` måste exempelvis ha en lista av möjliga koder medan `DataTypeQuantity` måste ha en enhet och ett intervall. Det finns också egenskaper som är gemensamma för alla datatyper, bland annat namn, beskrivning och plats i *template*. Alla gemensamma egenskaper läggs först in i en `DataTypeSettings` och skickas därefter till konstruktorn för datatypen tillsammans med de specifika egenskaperna.

Figur 3 visar ett exempel på en instansiering av elementet för det diastoliska trycket inom blodtrycksarketypen. Här anges att elementet ligger direkt under `any_event`, borde visas som "Diastoliskt" samt en beskrivning. Det anges också att elementet måste finnas, och att det kan finnas fler än ett, samt att det ska vara synligt i tabellen för användaren. Eftersom det är en kvantitet anges även att enheten är "mm[Hg]" och måste ligga mellan 0 och 1000.

```
let diastolicDataType = new DataTypeQuantity(
  {
    path: 'any_event',
    label: 'Diastoliskt',
    description: `Det minsta systemiskt arteriella blodtrycket uppmätt
    diastoliskt eller i hjärtcykelns avslappningsfas.`,
    required: true,
    single: false,
    visible: true,
  }, 'mm[Hg]', 0, 1000
)
```

Figur 3: Instantiering av en datatype för diastoliskt tryck.

**CategorySpec** är ett gränssnitt som motsvarar en *template* eller del av *template* i openEHR. Varje instans av `CategorySpec` representerar en kategori av hälsodata, såsom blodtryck. En instans innehåller huvudsakligen en lista av instansierade datatyper som motsvarar element men även namn, beskrivning och en identifierande sträng.

Ett exempel på en instansiering av en kategorispecifikation kan ses i figur 4. Här anges kategorins ID, namn, beskrivning samt ett antal element som kategorin innehåller.

```
let bloodpressureSpec = {
      id : 'blood_pressure',
      label : 'Blodtryck',
      description : `Den lokala mätningen av artärblodtrycket som är ett
      surrogat för artärtryck i systemcirkulationen.`,
      dataTypes : new Map<string, DataType>([
        ['time', timeDataType],
        ['systolic', systolicDataType],
        ['diastolic', diastolicDataType],
        ['position', positionDataType]
      ])
}
```

Figur 4: Instantiering av en kategorispecifikation för blodtryck.

Notera att tid här tolkas som ett element, men det är egentligen del av `Event`-objektet i arketypen. Alla kategorier innehåller därmed ett element för tid. Detta är dels på grund av hur JSON-formatet för inskickning är strukturerat, samt för att det är enklare att arbeta med generella element än att hantera tid specifikt.

### 4.1.2 datalist.ts

Filen `datalist.ts` innehåller de två klasserna `DataPoint` och `DataList`.

**DataPoint** En instans av en datapunkt motsvarar ett mätvärde för en kategori. Varje datapunkt innehåller ett värde för varje fält som kategorin har. En datapunkt för blodtryckskategorin i figur 4 innehåller därmed en tidpunkt, ett diastoliskt värde, ett systoliskt värde och en position. Eftersom positionsfältet inte är markerat som `required` kan det värdet vara tomt.

Implementationen av `DataPoint` är en endast en `Map` som mappar fältens ID mot fältens värde. Klassen har även metoderna `equals` och `compareTo` som jämför punktens alla fält med en annan punkts motsvarande fält.

**DataList** är en lista av datapunkter. En `DataList` kan därmed tolkas som en stor matris eller tabell av mätvärden. Varje datalista har en kategorispecifikation som definerar vilka fält som listan kan innehålla. Klassen är enkapsulerad och klienter måste använda metoder för att komma åt eller modifiera dess innehåll. Punkter läggs till med `addPoint` eller `addPoints` och deras giltighet verifieras enligt kategorins datatyper.

En datalista har även egenskaper som `width` och `mathFunction` som bestämmer hur värdena ska filtreras eller sammansättas. Om exempelvis `width` är `DAY` och `mathFunction` är `MEAN` så sammanställs alla punkter för varje dag till det genomsnittliga värdet för den dagen.

Implementationen av `DataList` innehåller två listor vilket möjliggör att filtreringar kan återställas. Instansens fält `points` innehåller de ursprungliga punkterna som lagts till medan `processedPoints` innehåller filtrerade punkter. Listan returnerar endast de behandlade punkterna med metoden `getPoints` medan de ursprungliga punkterna ej är åtkomliga utifrån.

### 4.1.3 EHR service

EHR-tjänsten agerar som en fasad mot EHR-modulen. Det finns två uppgifter för EHR Service: tillhandahålla kategorispecifikationer för programmet samt skicka in hälsodata till journalen.

**Kategorispecifikationer** hämtas genom att först anropa metoden `getCategories` för att se vilka kategorier som finns tillgängliga. Därefter hämtas kategorispecifikationen för en given kategori med hjälp av `getCategorySpec`.

För tillfället finns samtliga implementerade kategorier i `ehr-config.ts` och hämtas helt enkelt därifrån.

**Inskickning** utförs genom att först autentisera med `authenticateBasic` och därefter anropa `sendData` med en `DataList` och personnummer som inparameter och prenumerera till `Observable`-objektet som den returnerar. `sendData` skapar en JSON-formaterad komposition utifrån datalistan med hjälp av bland annat datatypernas `toRest`-metoder.

`Observable`-objektet som `sendData` returnerar utför totalt tre anrop mot journalen. Först måste ett `partyID` hittas genom att söka på individer som har det givna personnumret med en `GET`-förfrågan till `/demographics/party/query`. Därefter kan ett `ehrId` fås genom att göra en `GET`-förfrågan till `/ehr`. När ett `ehrId` väl har mottagits går det att skicka den skapade kompositionen med en `POST`-förfrågan till `/composition`.

### 4.1.4 ehr-config.ts

Filen `ehr-config.ts` innehåller alla inställningar som är specifika till det journalsystem som används. Just nu inkluderar detta ID:t till den *template* som används för alla inskickade kompositioner, URL:en till journalens REST-API samt *template*:s struktur beskriven med instanser av `CategorySpec`.

## 4.2 Platform: mot hälsoplattformar

Plattformsmodulen sköter all kommunikation mot hälsoplattformar. Dess huvudsakliga uppgift är att hämta data från hälsoplattformar och konvertera det till instanser av `DataPoint`.

Eftersom olika plattformar har olika API:er och dataformat måste varje hälsoplattform implementeras separat. Varje implementation för en hälsoplattform är en service som är en underklass till `Platform`.

### 4.2.1 Platform service

`Platform` är den superklass som definierar det gemensamma gränssnittet för alla plattformar. Utåt finns det fyra metoder tillgängliga; `signIn`, `signOut`, `getAvailable` och `getData`.

Metoden `signIn` är tänkt att autentisera användaren, och bör anropas när användaren väljer en specifik plattform. Väljer användaren Google Fit kommer en pop-up där användaren loggar in på Google och blir förfrågad om att dela med sig av sin hälsodata. Först när denna autentisering är klar kommer nästa sida att laddas. Med anledning av detta behov för att vänta in metodens exekvering så är `signIn` just

nu märkt med `async`-prefixet, vilket innebär att det är en asynkron funktion. När autentiseringen väl har skett kan `getAvailable` anropas för att avgöra för vilka kategorier som användaren har tillgänglig hälsodata som går att hämta. Slutligen kan `getData` användas för att hämta all hälsodata från en viss kategori inom ett visst tidsintervall.

`Platform` innehåller ett fält `implementedCategories` som är ämnat att innehålla plattformsspecifika detaljer om varje kategori som är implementerad, exempelvis motsvarande id hos plattformen. Fältet består av en `Map` som mappar varje kategori-id till ett `CategoryProperties`-objekt. Det objektet mappar i sin tur olika datatyper för de olika kategorierna till de funktioner som används för att hämta ut och konvertera datan från HTTP-förfrågan till det interna formatet.

### 4.2.2 Google Fit service

Klassen `GfitService` är en implementation för hälsoplattformen Google Fit. Utöver de fält som nämns i 4.2.1 så har `GfitService` två mappar av intresse: `commonDataTypes` som mappar vanligt förekommande datatyper såsom datum och enhet till den funktion som konverterar dem till det interna formatet, samt `categoryDataTypeNames` som mappar Googles namn på datatyper till de interna namnen.

I konstruktorn initieras sedan `implementedCategories`, och url:erna för de olika kategoriernas dataströmmar kopplas till ett `CategoryProperties`-objekt. Förutom vilken ström data ska hämtas ifrån så definieras även funktioner för att konvertera data till det interna formatet här.

För att hämta hälsodata från Google Fit så används i ett första steg metadata för användaren. Denna metadata hämtas och analyseras i `getAvailable`. Från metadatan extraheras alla tillgängliga kategorier och använda enheter. Efter detta kan `getData` användas för att hämta data för en implementerad kategori via en HTTP-förfrågan. I samband med hämtning av kategoridata kallas även funktionen `convertData`, som använder sig av `CategoryProperties`-objektet för att konvertera all hämtad data till det interna formatet.

## 4.3 Conveyor

Conveyor är en service som sköter all interaktion mellan EHR-modulen, plattformsmodulen och gränssnittet. Den kallas Conveyor eftersom den kan liknas lite till ett rullband. Hälsodata kommer från plattformar och lagras i Conveyor. Därefter behandlas och modifieras datan medan den är i Conveyor. Slutligen tas hälsodatan från Conveyor till EHR-modulen där den skickas vidare till journalen.

Eftersom Conveyor endast överför och lagrar data består en av en väldigt liten implementation. Conveyor är nästan helt agnostisk till de implementerade plattformar. Conveyor behöver endast injicera varje plattform och därefter kommunicera med det gränssnitt som är gemensamt för alla plattformar. Detta gränssnitt beskrivs närmre i avsnitt 4.2. Conveyor är även helt oberoende av vilka kategorier som är implementerade eftersom den endast lagrar och överför datalistor som själva implementerar behandling och modifiering.

### 4.3.1 Gränssnitt

Metoden `getPlatforms` returnerar ett id i form av en sträng för varje plattform som har injicerats av Conveyor och som går att använda. Det är det här id:t som alltid används i gränssnittet för att specificera en plattform till någon av metoderna hos Conveyor.

När en tillgänglig plattform väl är vald kan metoden `signIn` anropas för att autentisera mot den givna plattformen. I fallet av Google Fit öppnas här en extern pop-up som användaren kan logga in genom.

När autentiseringen är genomförd går det att anropa `getAvailableCategories` för att få reda på vilka kategorier som finns tillgängliga hos den inloggade användaren på den valda plattformen. Även här ges ett id i form av en sträng för varje tillgänglig kategori som används för att specificera kategorin.

För att därefter hämta hälsodatan används funktionen `fetchData` som tar emot plattformens id, *ett* id för en kategori samt ett tidsintervall. Metoden hämtar då en `Observable` från plattformen i fråga. Den binder därefter den i en ny `Observable` som inte innehåller något och returnerar den. När den sedan exekveras utförs hämtningen av hälsodatan och lagras därefter i Conveyor. Klienten kan därmed få reda på när hämtningen är klar, men datan sparas hos Conveyor istället för hos klienten. Klienten kan sedan få en referens till datan genom att anropa `getDataList` med kategorins id som inparameter.

Därefter kan klienten modifera datan fritt. Modifieringen av data sker ej via Conveyor, utan görs direkt utifrån datalistan som Conveyor returnerar en referens till när `getDataList` anropas.

När klienten väl är färdig med hämtning och modifiering kan den använda `authenticateBasic` och `sendData` för att skicka vidare datan till journalen.

## 4.4 GUI: grafiskt gränssnitt

Alla grafiska element i applikationen består av Angular-komponenter. Hela sidan som visas är applikationens AppComponent. Den innehåller statiska komponenter som syns hela tiden såsom menypanelen. Den innehåller också en Routing-modul som innehåller olika komponenter beroende på vilken URL som användaren befinner sig på. Routingmodulen kan innehålla komponenter för huvudsidor eller vyer i inrapporteringsflödet.

Det finns även en modul DataViewer som innehåller komponenter som används i flera vyer. DataViewer ansvarar för att visualisera och interagera med användarens data och används i både redigeringsvyn och inspektionsvyn.

Det grafiska gränssnittet använder sig till stor del av biblioteket Angular Material för att få färdigbyggda komponenter. När en specifik komponent föregås av prefixet `Mat` betyder det att det tillhör Materialbiblioteket (t.ex. `MatCard`, `MatDialog` o.s.v.).

### 4.4.1 Statiska komponenter

Applikationen har tre komponenter som alltid ligger i AppComponent oberoende av nuvarande URL.

**progress-bar** ProgressBarComponent visar hur långt man kommit i inrapporteringsstegen och vilka steg som återstår och syns då man startat en inrapporteringsprocess. Den finns alltid i AppComponent men döljs om användaren inte befinner sig i inrapporteringsflödet.

Den består av 3 knappar med motsvarande beskrivning. Knapparnas färg indikerar vilket steg man är på och vilka steg som gjorts. Den använder nuvarande URL för att bestämma vilket steg användaren är på.

**ToolbarComponent** visas alltid högst upp i applikationen, och innehåller en logogtyp samt länkar till start, info och hjälpsidan. Består av en `MatToolbar` som innehåller en logotyp och knappar som länkar till de olika sidorna.

**FooterComponent** är applikationens permanenta sidfot och innehåller för tillfället information om licenser för ikoner.

### 4.4.2 Komponenter för huvudsidor

Applikationen har tre huvudsidor som ej är del av inrapporteringsflödet.

**HomePageComponent** är applikationens startsida. Härifrån kan man navigera till plattformsväljaren (se *sources*) för att välja hälsoplattform eller välja att gå till en hjälpsida som förklarar hur hemsidan fungerar (se *help*).

**HelpPageComponent** innehåller helt enkelt en kort användarguide som beskriver de olika stegen en användare behöver göra för att skicka hälsodata.

**InfoPageComponent** är tänkt att visa information om applikationen, till exempel info om GDPR och eventuellt kontaktuppgifter.

### 4.4.3 Komponenter för inrapporteringsflöde

Applikationen har fyra olika vyer och därmed fyra olika komponenter för inrapporteringsflödet.

**platform-selection** visar vilka hälsoplattformar som finns tillgängliga så att användare kan välja vilken plattform (t.ex. Google Fit) de vill hämta data ifrån.

För varje tillgänglig plattform visas ett `MatCard` med namnet på plattformen samt dess logotyp. När man klickar på ett sådant `MatCard` kallas typescriptfunktionen *selectPlatform*, som då använder Conveyor för att sköta inloggning till den valda plattformen.

Använder Conveyor för att hämta de hälsoplattformar som är tillgängliga samt för att sköta inloggning då användare valt en plattform att hämta data.

**category-selection** används för att välja de kategorier som ska hämtas från en källa. Den använder sig av `Conveyor` för att hämta de kategorier som finns tillgängliga att välja mellan, och sparar de id:n som är valda i de checkboxes som ritas ut. Den väljer automatiskt en månad bakåt i tiden som det tidsintervall för vilket data ska hämtas.

Komponenten består av två `MatCard`, en för att visa alla valbara kategorier, och en för att välja datum. Kategorierna ritas ut m.h.a. en for-loop som går igenom de kategorier som har hämtats från `Conveyor`.

**editor-view** hanterar vyn som tillåter användaren att redigera och komplettera sin hälsodata. Den låter användaren redigera och se sin data genom att inkludera en DataContainerComponent från Data-ViewerModule för varje kategori som användaren har hämtat data för.

Dess AddNewDataModalComponent innehåller knappar för att lägga till kategorier eller hämta ny data. Dess BottomSheetCategoriesComponent används för att välja vilken kategori att lägga till om användaren vill mata in egen data.

**inspection-view** är den komponent som ger granskningsvyn, det sista steget innan man skickar in hälsodata till RÖD. Här kan man se all den data man är på väg att skicka in, men man kan ej göra några ändringar. Alla kategorier som man har fyllt med data visas i paneler som kan expanderas för att visa en lista med all den data som lagts in på den kategorin. Denna komponent innehåller även funktionalitet för autentisering mot RÖD. Här är tanken att autentiseringen ska ske via BankID, men i nuläget sker den helt enkelt med vanligt användarnamn och lösenord.

Består av en `mat-expansion-panel` för varje kategori som innehåller data. I varje panels `mat-expansion-panel-header` visas både namnet på kategorin och hur många datavärden den innehåller. När en användare klickar på en panel så expanderas den och visar en lista med alla datavärden för den aktuella kategorin. Listan med data utgörs helt enkelt av en DataContainerComponent, men med `editable` satt till false så att man ej kan manipulera datavärdena på något sätt.

Resten av komponenten består av inputs för användarnamn och lösenord samt en knapp som kallar på autentiseringsfunktionen i Conveyor med det användarnamn och lösenord som fyllts i, samt funktionen *sendData* i Conveyor.

Använder sig av Conveyor för att kunna hämta kategorier, se om en viss kategori innehåller data eller ej, autentisera mot RÖD samt skicka hälsodata.

### 4.4.4 DataViewerModule

DataViewerModule innehåller komponenter för att visa och redigera hälsodata. DataViewerModule inkluderar komponententerna DataTableComponent, DataChartComponent och DataContationerComponent.

**data-table** är den lista som visar alla datapunkter från en vald kategori. Listan kan bli satt `editable`, för att kunna ändra på kommentarer och dylikt. Listan har en `paginator` för att den inte ska bli allt för stor, samt beskrivningar av allt den visar upp som hints.

Komponenten består av en `MatTable` med en `paginator`. Komponenten använder en for-loop för att få fram alla kolumner. Komponenter använder en for-loop för att sätta visningen av alla datavärden, om det är text, datum, drop-down-lista m.m.. Dessa är även olika om värdena är `editable` eller ej. För att bestämma visningstyp används if-satser.

Komponenten hämtar `CategorySpec` från `Conveyor` och sätter dessa i `displayedColumns` som används för uppritning.

**data-chart** är ett diagram som ritar ut alla numeriska värden för varje filter i den givna kategorin.

**data-container** består av flikar där varje flik består av en DataTableComponent för varje filter som användaren valt. En av flikarna är också en DataChartComponent som visualiserar värdena för alla filter.

Komponenten innehåller också knappar för att lägga till, ta bort eller filtrera punkter.

**data-point-dialog** används som en pop-up, den är en `MatDialog` och öppnas när användare vill lägga till eller modifiera en datapunkt i en kategori.

Eftersom `data-point-dialog` är en `MatDialog` används MatDialogs header, content, och footer. I headern visas kategorins titel. I content finns ett antal if-satser som kollar vilken sorts data som ska matas in och visar rätt sorts input. I footern finns två knappar: en för att lägga till data, och en för att avbryta inmatningen, vilket även stänger komponenten.

DataPointDialogCompenent använder sig av en `CategorySpec` för att veta vilka sorters data som ska läggas in i varje kategori. Kategorinamnet injekteras av den komponent som använder sig av `data-point-dialog`, där `data-point-dialog` sedan använder sig av det namnet för att hämta `CategorySpec` från `Conveyor`. Det finns en `ErrorStateMatcher` som används för att skicka error-meddelanden till de input-fält som inte är ifyllda.

### 4.4.5 Responsiv design

För att implementera en responsiv design användes till viss del Angulars Flex-Layout bibliotek. Detta används genom att sätta attribut i HTML taggarna. t.ex. kommer `<div fxHide fxShow.lt-sm>small screen</div>` gömmas som default, och visas när enhetens skärm uppfyller lt-sm (vilket motsvarar mediaQuerien `screen and (max-width: 599px)`). Mer info om olika storlekar finns här `https://github.com/angular/flex-layout/wiki/Responsive-API`. Ett problem som uppstod med denna taktik var t.ex. i tabellen. På den mobila versionen behövde vissa kolumner tas bort, för att visas i en modal istället. Att använda fxHide gömmer endast elementet, vilket i och för sig är intuitivt men detta gjorde att kolumnerna som skulle flyttas till modalen fortfarande dök upp, dock med gömda element. Istället slopades kolumntitlarna i TypeScript filen genom kolla ifall ovanstående villkor uppfylldes. I HTML-komponenten användes istället Angulars inbyggda villkorliga utritning ngIf som använde samma taktik för att bestämma ifall ett element skulle ritas ut.

Ett problem som kvarstår för den mobila vyn är att det på vissa enhter ej är uppenbart att huvudytan (mellan header och kategorimenyn längst ned) är scrollbar, och att det finns knappar under huvudtabllen. Det hittades ingen bra lösning på detta, förutom att möjligen inkludera det i instruktionenerna som visas när användaren först laddar denna vy.

# 5 Vidareutveckling

För att underlätta vid vidareutveckling av applikationen har gruppen valt att ta upp viss funktionalitet som de anser vara sannolika. Förhoppningen är att det kan hjälpa utvecklare som är nya till systemet snabbt bli bekväma med systemet och hur det kan breddas.

## 5.1 Utökning av kategorier

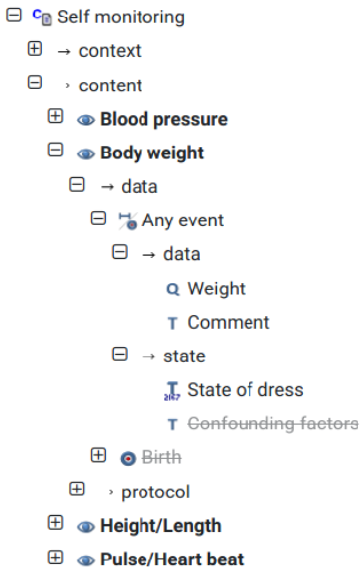
För att lägga till ytterligare kategorier i systemet så måste saker göras dels hos journalen men även i alla de olika modulerna.

### 5.1.1 Hos journalsystemet

För att kunna lägga till en kategori måste det först finnas en lämplig `Observable`-arketyp i openEHR. Den måste även läggas till i den *template* som applikationen används i. I figur 5 går det att se den övergripande strukturen av den nuvarande *template* som används för att skapa kompositioner.

För att lägga till en ny kategori läggs en *Observable* till under `content`-grenen. Om arketypen innehåller klustret *Medical Device* bör den behållas så att applikationen kan ange insamlingsenhet.

När kategorin är tillagd i ens *template* går den att ladda upp med en POST-förfrågan till API-anropet `/template`. [10]




Figur 5: Nuvarande template för inrapportering.

### 5.1.2 Ehr

Nästa steg är att skapa en kategorispecifikation för kategorin. Dessa definieras i filen `ehr-config.ts`. Som nämnt i avsnitt 4.1.1 måste denna innehålla ett id för kategorin, ett namn, en beskrivning och en lista av datatypinstanser som beskriver varje element. Alla id:n måste matcha de som används i JSON-formatet vid inrapportering. De använder alltid samma namn som i *template*:n, med skillnaden att alla bokstäver är gemener samt att mellanrum ersätts med understreck.

Namnet och beskrivningen motsvaras av de svenska översättningarna i arketypen. Id:t ska vara namnet av arketypen för kategorin fast med understreck, exempelvis ersätts "Body weight" med `body_weight`. Id:t bör även läggas till i `Enum`:en `Categories` så att plattformarna kan referera till den.

Slutligen måste en datatypinstans läggas till för varje element som *template*:n tillåter och som önskas användas. All information som ska gå att hämta från en plattform eller kompletteras av användaren måste läggas till som ett element i kategorispecifikationen.

I viktexemplet i figur 5 finns det tre element `weight`, `comment` och `state_of_dress`. Det måste utöver detta alltid finnas ett fält för tid. Detta fält bör även ligga först så att sortering i första hand utgår från mätvärdets tidpunkt. Instanser för tid och kommentar finns redan definierade i `ehr-config` som `TimeField` och `CommentField`. Det finns även definerade instanser för att ange information om mätenheten. Om kategorins arketyp innehåller klustret *Medical Device* bör dessa läggas till i kategorispecifikationen.

[10] https://www.ehrscape.com/reference.html#_template

För resterande element; kontrollera att de datatyper som används är implementerade i `datatype.ts`, lägg annars till datatyperna enligt 5.3. Skapa därefter en instans av datatypen och lägg till den i specifikationen. En `Enum` med datatypernas id bör även läggas till för att undvika att stavfel sker när de ska refereras till av plattformens implementation.

### 5.1.3 Platform

När kategorin finns i en *template* hos journalsystemet och en motsvarande kategorispecifikation har skapats återstår att implementera hämtning av hälsodatan i aktuella plattformar. Detta är förstås väldigt beroende av plattformens specifika implementation.

**Google Fit** Mycket av det som tas upp nedan beskrivs i större detalj på `https://developers.google.com/fit/rest/v1/reference/`. Detta steget kan vara lite bökigt då inget annat sätt än att manuellt identifiera dataströmmar för olika kategorier och de fält som innehåller mätvärden hittats. För att lägga till en kategori måste nedanstående göras i `GfitService`:

- Mappa Googles kategorinamn mot det interna kategorinamnet i `categoryDataTypeNames`. Googles standardtyper kan exempelvis hittas på `https://developers.google.com/fit/android/data-types#public_data_types`
- I konstruktorn initieras som tidigare nämnt `implementedCategories`. Här måste ny entry med den nya kategorin föras in. Förutom det interna kategorinamnet behövs en url till den dataström som data ska hämtas ifrån, samt en `Map` som beskriver vart de olika datatyperna för kategorin kan utläsas i det JSON-format som Google Fit returnerar.

Att lägga till url:en till den dataström som datan ska hämtas ifrån har i projektet gjorts genom att gå in på `https://developers.google.com/oauthplayground/`, välja 'Fitness v1' och de scopes som är av intresse, och sedan skicka en `GET`-request för metadata. Metadatan kan sedan genomsökas med exempelvis det kategorinamn som ska implementeras. Det brukar finnas ett antal olika dataströmmar för varje kategori, och i vår implementation används de dataströmmar som innehåller den data som visas upp i Google Fit-appen. Dessa brukar ungefär vara på formen:
`derived:<kategorinamn>:com.google.android.gms:merge_<kategorinamn>`. Det bör även påpekas att endast de kategorier och dataströmmar som den inloggade användaren aktivt laddat upp data på kommer att synas. Detta innebär att det kan bli svårt att hitta vissa dataströmmar med denna metod. Under projektet har tyvärr ingen dokumentation som visar alla existerande dataströmmar hittats.

När den efterfrågade dataströmmen är funnen så är det sista steget att identifiera vilka fält som innehåller data av intresse. För att hitta dessa fält kan man hämta och granska den specifika dataströmmen för kategorin som ska implementeras. Detta är exakt vad som görs i metoden `getData` när kategorin sedan är färdigimplementerad. I OAuth playground görs det genom följande anrop:
`https://www.googleapis.com/fitness/v1/users/me/dataSources/<dataström>/datasets/startTimeNanos-endTimeNanos/?access_token=<token>`
där `startTimeNanos` och `endTimeNanos` är Unix-timestamp i nanosekunder och `<token>` är den unika access token som gäller för sessionen. Väl här så bör det synas vilket fält som innehåller själva mätvärdet. Oftast är det under `point.value[0].fpval` eller liknande. Några av de vanligaste fälten såsom tidpunkt och enhet är gemensamma för samtliga kategorier och finns redan definierade i `commonDataTypes`. Mer noggrann dokumentation om olika fält och datasets finns på `https://developers.google.com/fit/rest/v1/reference/users/dataSources/datasets`

Övrig dokumentation om `DataPoint`, `Dataset` och `DataSource` som kan vara av intresse:
`https://developers.google.com/android/reference/com/google/android/gms/fitness/data/DataSource`

## 5.2 Utökning av plattformar

För att lägga till en ny plattform måste ändringar göras i flera av programmets moduler.

### 5.2.1 Platform

För att implementera en ny plattform måste en ny service skapas i mappen `platform` som ärver superklassen `Platform`. Därefter måste metoderna `signIn`, `signOut` och `getData` och `getAvailable` implementeras.

Metoden `signIn` ska autentisera användaren så att klienten därefter kan anropa `getAvailableCategories` och `getData`. `getAvailableCategories` ska returnera en `Observable` i form av en lista av id:n för alla de kategorier som finns tillgängliga för användaren. `getData` ska returnera en `Observable` i form av en lista av `DataPoint`-instanser för alla datapunkter inom ett visst tidsintervall för en kategori. Plattformen kommer antagligen inte föra in data i alla tillgängliga fält, utan endast de som det finns data för. För exempelvis kroppsvikt bör tid och vikt alltid fyllas i av plattformen. Kommentar och klädsel kanske inte finns implementerat i plattformen, och kompletteras av användaren om den önskar. Plattformen bör även fylla i fälten angående mätenheten om den informationen finns tillgänglig.

Plattformen kan använda sig av fältet `implementedCategories` för att mappa de implementerade kategoriernas id:n till funktioner som konverterar kategorispecifik data från hälsoplattformens format till det interna formatet. De id:n som används måste matcha kategorispecifikationernas id:n. Därmed bör `Enum`:s med id:n från `ehr-config` användas för att se till att det är exakt samma id.

### 5.2.2 Conveyor

När plattformen väl är implementerad återstår att se till att Conveyor har tillgång till den. Detta görs genom att injicera Conveyor med den nya plattformens service. För att åstadkomma detta måste klassen importeras och en referens till klassen måste skapas i konstruktorn för Conveyor. Därefter måste instansen av klassen läggas till i fältet `platforms` så att Conveyor känner till att den finns. Ett id behöver även bestämmas för att kunna specificera plattformen.

## 5.3 Utökning av datatyper

Datatyper behöver dels definieras i Ehr-modulen men stöd behöver även skapas i gränssnittet.

### 5.3.1 Ehr

Att skapa stöd för en ny datatyp innebär att skapa en ny klass. Klassen definieras i `ehr/datatype.ts` och måste ärva den abstrakta superklassen `DataType`. Nedan beskrivs hur varje metod därefter bör implementeras.

**Konstruktorn** ska ta emot alla egenskaper som är specifika till en datatyp. Exempelvis intervall och enhet för `DV_QUANTITY`. Konstruktorn sätter därefter dessa egenskaper i instansvariabler så att resterande metoder kan använda dessa.

**isValid** måste implementeras och har som uppgift att testa om ett värde är giltigt eller inte. Värdet måste vara av rätt enhet och uppfylla alla krav som satts av arketypen och kompositions-*template*.

**equal** ska jämföra två giltiga värden och avgöra om de är lika eller inte. Om det räcker att jämföra värdena med === behöver den inte implementeras eftersom den ärvs från `DataType`.

**compare** ska jämföra två giltiga värden och avgöra vilket av värdena som är störst. Om det första värdet är störst ska den returnera ett positivt tal. Om det andra värdet är störst ska den returnera ett negativt tal och om båda är exakt lika ska den returnera 0. Om det räcker att jämföra med < och > behövs den ej implementeras eftersom den ärvs från `DataType`.

**Matematikfunktioner** behöver också implementeras om inte implementationen från `DataType` är lämplig. Metoderna motsvarar matematikfunktionerna i openEHR och inkluderar just nu `median`, `mean`, `total`, `min` och `max`. Metoderna ska ta en lista av värden och returnera *ett* värde som är en sammanställning av alla värden med en viss funktion. Implementationen för `DataType` är densamma för alla dessa; om alla värden är samma används det värdet, annars används ett nollställt värde. Denna implementationen är lämplig för datatyper som matematiska funktioner inte är rimliga för; exempelvis `DV_TEXT`. Om matematiska funktioner går att applicera på datatypen bör metoderna implementeras så att beteendet från `DataType` inte används.

**toRest** är en metod som måste implementeras. Den ska ta emot ett giltigt värde och skapa ett motsvarande JSON-objekt som ska användas i anropet till REST-API:n. Detta är väldigt beroende på datatypen och därför demonstreras endast ett exempel. I figur 6 visas ett exempel på hur en komposition med ett värde för kroppsvikt kan se ut på JSON-formatet som används mot API:n THINK-EHR. Varje element har en lista av objekt, och `toRest` har i uppgift att ta ett giltigt värde och returnera det JSON-objektet som ska läggas i listan. Exempelvis returnerar `toRest` för `DataTypeQuantity` i detta fallet `{'|magnitude': 79.85, '|unit': 'kg'}` då värdet "79.85" ges.

```
"self_monitoring": {
  "body_weight": [
    {
      "any_event": [
        {
          "body_weight": [
            {
              "|magnitude":
                  79.85,
              "|unit": "kg"
            }
          ],
          "state_of_dress": [
            {
              "|code": "at0011"
            }
          ]
        }
      ]
    }
  ]
}
```

Figur 6: Exempel av en kompositon för kroppsvikt i JSON.

### 5.3.2 GUI

All hantering av data för gränssnittet sker inom modulen DataViewerModule.

**DataTableComponent** måste uppdateras för att avgöra hur datatypen ska visas och matas in.

I funktionen `getFormattedTextFromPoint` måste en sträng skapas utifrån en datatyps värde som därefter ska visas i tabellen. Exempelvis blir ett värde för datatypen `DataTypeQuantity` värdet i sig plus dess enhet.

I funktionen `getInputType` måste hur datatypen ska matas in i tabellen specificeras genom att ange en sträng som motsvarar inmatningstypen, exempelvis `'text-input'` och `'text'`. `'text'` används om datatypen ej ska gå att ändra i tabellen. Därefter skapas ett `mat-form`-objekt för varje typ av inmatning i komponentens HTML-fil `data-table.component.html`.