CHALMERS

MedImager

En webbapplikation för hantering av bilder inom oral medicin

Kandidatarbete inom data- och informationsteknik

Daniel Ahlqvist

Carl-Henrik Braw

Michel Folkemark

Renée Gyllensvaan

Marcus Linderholm

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, May 2017

En webbapplikation för hantering av bilder inom oral medicin

DANIEL AHLQVIST
CARL-HENRIK BRAW
MICHEL FOLKEMARK
RENÉE GYLLENSVAAN
MARCUS LINDERHOLM

© DANIEL AHLQVIST, 2017
© CARL-HENRIK BRAW, 2017
© MICHEL FOLKEMARK, 2017
© RENÉE GYLLENSVAAN, 2017
© MARCUS LINDERHOLM, 2017

Handledare: Olof Torgersson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Göteborg, Sweden 2017

Förord

Vi önskar rikta ett stort tack till Mats Jontell, professor i oral medicin vid Sahlgrenska i Göteborg, för hans synpunkter under utvecklingen. Vi vill även tacka vår handledare, Olof Torgersson, för hans engagemang och vilja att hjälpa oss, samt Anders Grip för den kontinuerliga hjälpen med MedView core.

Sammanfattning

Den här rapporten behandlar utvecklingen av MedImager, en webbapplikation vars syfte är att underlätta för odontologer och odontologstudenter att studera munsjukdomar och tänder. Applikationen utvecklas på beställning av Sahlgrenska akademin och bygger på en redan existerande databas innehållandes cirka 70 000 bilder.

MedImager består av både en front-end och en back-end, vilka båda beskrivs i rapporten. Front-end är sammansatt med hjälp av både mer traditionella webbutvecklingsmetoder, såsom HTML, CSS och Javascript, men använder sig även av modernare ramverk och bibliotek, till exempel Angular, SASS, Jquery och Materialize. Back-end är skriven i Java och använder sig av Javabiblioteket MedView core för hantering av databasen. Kommunikation mellan front-end- och back-end sker via ett REST API utvecklat med hjälp av ramverket Jersey. Datan som kommuniceras är på formatet JSON. Alla dessa tekniker gås igenom i rapporten. Även säkerhetsaspekter tas upp, däribland HTTPS, SSL, tokenautentiering och lösenordshantering.

Rapporten beskriver processen, de tekniska medel som använts, den slutgiltiga produkten i både text och bild samt diskuterar de val som gjorts och relevanta problem som stötts på under utvecklingens gång.

Abstract

This report describes the development of MedImager, a web application with the purpose of easing the ability to study oral diseases and teeth for oral experts and students. The application is developed on demand from Sahlgrenska akademin and is built upon an already existing database containing 70 000 images.

MedImager consists of a front- and back-end, which both are described thoroughly in the report. The front-end is put together with both traditional web development methods, such as HTML, CSS and Javascript, but also modern frameworks, for example Angular, SASS, Jquery and Materialize. The back-end is written in Java utilizies the Java library MedView core to manage the database. Communication between front- and back-end is accomplished through a REST API developed using the framework Jersey. The data communicated is in the format JSON. All these techniques are explained in the report, as well as security aspects, such as HTTPS, SSL, token authentication and password management.

The report describes the process, the technical methods aswell as the final product, both in text and images. It also discusses choices and problems the project has encounterd during development.

Innehåll

Förkortningslista

- AJAX – *Asynchronous JavaScript and XML (Extensible Markup Language)*
- API – *application programming interface*
- CSS – *Cascading Style Sheets*
- DOM – *Document Object Model*
- ER-diagram – *entity–relationship-diagram*
- GUI – *Graphical user interface*
- HTML – *HyperText Markup Language*
- HTTP – *HyperText Transfer Protocol*
- HTTPS – *HyperText Transfer Protocol Secure*
- JAX–RS – *Java API for RESTful Services*
- JSON – *JavaScript Object Notation*
- JWT – *JSON Web Token*
- REST – *Representational State Transfer*
- SASS – *Syntactically Awesome Style Sheets*
- SQL – *Structured Query Language*
- SSL – *Secure Sockets Layer*
- URL – *Uniform Resource Locator*

Ordlista

- back-end – den del av applikationen som kommunicerar med databas och som front-end gör förfrågningar till för att åstadkomma saker i applikationen, såsom sökning i databasen.
- Base64 – en metod som används för att koda binär data till skrivbara sjubitars ASCII-tecken, för distribution via till exempel e-post.
- examination – en representation av en klinisk undersökning i en mvd-databas.
- front-end – syftar till allting som syns på klientsidan.
- fält/parameter – sökbara parametrar i den givna databasen, såsom *Diag-def* (diagnos), *Dis-now* (nuvarande sjukdom) etc.
- header – en term som används för att beteckna ett avsnitt i början av en datafil som innehåller metadata.
- HTTP-förfrågan – meddelande som används för att kommunicera över internet enligt HTTP-protokollet
- klient – datorprogram/värddator som ofta styrs av en användare och blir tillhandahållen tjänster via kommunikation med en server över ett nätverk.
- metadata – data som är knutet till överordnad data som har syftet att agera beskrivande/förklarande för den överordnade datan
- mvd – namnet/filformatet på databasen innehållandes examinationer och bilder.
- relationsdatabas – en databas där data är organiserad i tabeller och sammankopplat via så kallade ”relationer”
- server – datorprogram/värddator som tillhandahåller klienter tjänster.
- term – ett annat namn för en nod/egenskap som en examination i en mvd-databas har ett flertal av, såsom *Diag-def* och *Dis-now*. Dessa kan exempelvis stå för diagnosen som utfärdats till patienten under undersökningen, vad patienten har för allergier etc.
- token – en sorts nyckel som, om den utfärdats av en webbtjänst, kan användas för att autentisera sig till webbtjänsten

1 Inledning

Den som någon gång varit hos en tandläkare i Sverige har med största sannolikhet blivit fotograferad i munnen. Dessa bilder ligger sedan kopplade till patientens journal i tandläkarnas datorsystem och kan egentligen inte användas till något mer användbart än att tandläkarna kan granska eventuella förändringar hos patienten mellan besöken. Bilderna bör dock, teoretiskt sett, vara mycket värdefulla, då de innehåller en mängd information. Exempel på användningsområden för bilderna kan tänkas vara utbildning av nya odontologer eller att ge odontologer möjligheten att granska liknande fall när de har en patient.

1.1 Bakgrund

MedView (1) (2) är ett projekt som drivs av institutionen för tillämpad IT vid Chalmers tekniska högskola och Göteborgs universitet, samt Oral medicin vid Sahlgrenska akademin och Högskolan i Skövde. Projektet har innefattat att bland annat med bild och text dokumentera patienters hälsa vid oral medicin. Med tiden har en databas bestående av cirka 70 000 bilder byggts upp. Något som dock har saknats är verktyg för att på ett effektivt sätt kunna dra nytta av denna databas. Verktyg för att kunna söka efter bilder baserat på vissa kriterier, spara dem i samlingar, dela dem med andra samt kunna samarbeta kring dem skulle kunna utöka användningsområdena för databasen ytterligare. Ett sådant verktyg har efterfrågats av odontologer vid Sahlgrenska akademin och skulle kunna hjälpa till att förbättra utbildningen vid universitetet.

1.2 Syfte

Syftet med projektet MedImager är att utveckla en webbaserad applikation för hantering av odontologiska bilder i en existerande databas. Applikationen ska underlätta för odontologer att lära sig av tidigare patienters sjukdomar så att denna information kan komma till användning vid exempelvis kommande fall. Den ska även kunna vara ett stöd för utbildningen av nya odontologer. Utöver detta ska applikationen vara snabb och smidig samt vara intuitiv nog att gemene man utan problem kan sätta sig in i den. Rapportens syfte är att presentera tillvägagångssättet för utvecklingen av applikationen samt den slutliga produkten.

2 Produktspecifikation

I detta kapitel gås det först igenom de problemställningar som har definierats för applikationen. Dessa har tagits fram i samband med kravspecifikation från kunden på Sahlgrenska akademin samt åsikter från handledare. Kapitlet avslutas med en del aspekter vad gäller användare som det måste tas hänsyn till vid utvecklingen av applikationen.

2.1 Önskad funktionalitet

I detta kapitel beskrivs mål för MedImager. Målen är baserade på önskemål från Sahlgrenska akademin i Göteborg. De önskemål som varit vaga har specificerats av gruppen, i kommunikation med en representant från Sahlgrenska akademin.

2.1.1 Sökning och filtrering

Den mest centrala funktionen i MedImager är möjligheten att söka efter bilder. Det ska vara möjligt att hitta bilder efter filtrering på en rad olika parametrar. Ett exempel kan vara att användaren vill hitta bilder på en kvinna som lider av lichen planus samt brukar cigaretter.

2.1.2 Samlingar

En användare ska kunna samla bilder i egenskapade samlingar, eller album. Detta för att kunna gruppera bilder som denna anser vara av intresse. Ett exempel på användningsområde kan vara att användaren grupperar alla bilder på lichen planus som denna anser vara bra. Varje samling, och dess individuella bilder, skall kunna beskrivas med korta komnmentarer. Samlingarna ska vara lätta att nå från samtliga vyer. Skapade samlingar kan sedan delas med övriga användare.

2.1.3 Egna anteckningar

Bilderna i databasen har viss information kopplad till sig. Det kan till exempel vara personens ålder, vilka sjukdomar och allergier den har, var bilden är tagen med mera. Dock kan det finnas behov för enskilda användare att skriva egna kommentarer till bilderna, såsom tankar och kom-ihåg-anteckningar. De egna anteckningarna ska endast visas för personen som skrivit dem.

2.1.4 Exportering till mvd-format

Bilddatabasen som MedImager ska hantera är i ett särskilt filformat vid namn mvd. Sedan länge har det funnits andra program som hanterar samma typ av databas. Ett exempel på ett sådant är MVisualizer (3), ett program som används för visualisering och statistisk analys av databasen. Funktionalitet för exportering av användarskapade samlingar till detta specifika format skulle möjliggöra för användning av dessa samlingar även i andra program som hanterar databaser av typen mvd.

2.1.5 Sekretess

Eftersom informationen i bilddatabasen är sekretessbelagd, samt det faktum att användarinformation kommer behöva lagras, behöver datasäkerhet tas i beaktning. Rätt metoder och tekniker kommer därmed behöva användas för att denna informations integritet inte ska äventyras.

2.2 Användaranalys

En viktig sak vid utveckling av programvara är att den måste vara anpassad för den tilltänkta målgruppen. I detta fall är målgruppen odontologer, vilka kan antas ha begränsade kunskaper inom datoranvändning, då deras utbildning inte innehåller programmering eller andra mer avancerade datortekniska kurser. Detta innebär att fokus behöver läggas på att göra användargränssnittet och funktionerna intuitiva för att alla användare ska kunna utnyttja tjänsten på ett effektivt sätt.

2.3 Avgränsningar

För att kunna fokusera på det som är viktigast för projektet och applikationen har följande avgränsningar gjorts.

- Applikationen kommer inte mobilanpassas, detta för att applikation främst kommer användas på datorer.
- Utvecklingen kommer ske mot webbläsaren Google Chrome och därför också vara den enda webbläsaren som stöds aktivt. Detta för att utveckling mot flera webbläsare skulle göra utvecklingsprocessen mer invecklad.
- Applikationen kommer bara stödja en datastruktur, vilken är bilddatabasen som blivit försedd.
- Datastrukturen kommer inte göras om på något sätt, utan enbart existerande lösningar kommer användas.

3 Teknisk bakgrund: Databasen med bilder

Databasen som MedImager är utvecklat för att söka igenom består av bilder samt så kallade ”trädfiler” (4), vilket är ett format för lagring av dokument. Denna databas lagras i ett format vid namn *mvd*, vilket gör att den även kan kallas för en ”mvd-databas”. I en mvd-databas är varje trädfil en representation av en klinisk undersökning. Därför kallas en sådan trädfil även för en ”examination” i databasen. Varje examination har ett antal så kallade ”noder”, vilka står för olika sorters ”egenskaper” eller ”värden” som antingen undersökningen eller själva patienten som undersökts besitter. Dessa egenskaper kan vara allt från allergier som patienten har till den diagnos som utfärdats vid undersökningen. En nod kan ha flera sådana värden givna till sig. Ett annat namn i databasen för en sådan nod är ”term”. Exempel på termer är *Allergy* och *Diag-def*, vars värden är just patientens allergier respektive diagnosen som utfärdades vid undersökningen.

En examination skapas och läggs in i databasen genom att den undersökande läkaren matar in information om undersökningen med hjälp av ett program utvecklat för detta ändamål. Ett exempel på ett sådant program är MedRecords (5). Beroende på vad läkaren matat in är det sedan möjligt att vissa termer står utan värde. Detta kan ske om läkaren inte bett patienten om en viss information, såsom dess yrke, eller om en term inte har något värde, såsom i fallet om patienten inte har några allergier. I det senare fallet skulle värdet *Nej* kunna ges till termen för att signifiera att patienten inte har några allergier, men beroende på vad läkaren matat in varierar det bland examinationerna hur detta hanteras. I figur 1 finns ett utdrag ur hur en sådan examination kan se ut.

```
NDis-past
LMultipel skleros##
NCheckup
LNej##
NDrug#
NAllergy
LNej##
```

Figur 1: Utdrag ur en trädfil.

Något förenklat står en rad som inleds med ”N” för en term (där N står för nod) medan en som inleds med ”L” står för ett värde till termen närmast ovanför (där L står för ett ”löv” till noden). Exempelvis innebär *Dis-past* patientens tidigare sjukdomar och *Drug* patientens aktuella medicinering. Ur figur 1 kan det alltså erhållas att patienten tidigare haft sjukdomen *Multipel skleros* samt att inget

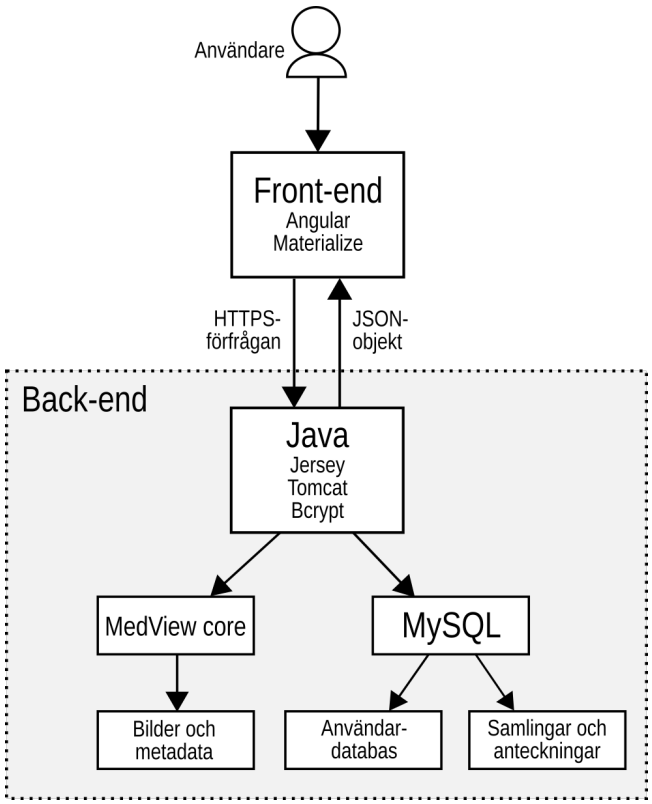
värde matades in för termen *Drug* vid insättning av examinationen i databasen. Se *Search: Available terms* i bilaga A för en fullständig lista över möjliga termer.

En av de ytterligare egenskaper en examination kan ha (och antagligen de viktigaste för MedImager) är de bilder som blev tagna under undersökningen som examinationen representerar. Examinationen innehåller på ett sätt länkar igenom vilka bilderna kan nås. Det är via dessa länkar som bilder kan hämtas ur databasen. Tack vare att en bild är länkad från en examination får den indirekt examinationens egenskaper knutna till sig, vilket bidrar till att bilden får en sorts metadata.

4 Använda tekniker och metoder

Det enda fysiska material som användes under utvecklingen var datorer för programmering samt papper och penna för skapandet av pappersmodeller för det grafiska användargränssnittet. I övrigt har en rad olika programmeringsspråk, ramverk och kodbibliotek använts. I vissa fall har ett par språk jämförts med varandra, innan gruppen bestämt sig för ett, men i de flesta fall har språk som ansetts uppfylla MedImagers krav valts utan närmare jämförelser med andra alternativ. Anledningen till att jämförelser med andra alternativ i de flesta fall inte gjorts, är att det ansetts överflödigt, då det inte påverkat prestandan och användarupplevelsen något. Det handlar främst om syntax och koduppbyggnad. Hur systemets komponenter hänger samman kan studeras i figur 2.

De använda språken och teknikerna kommer att presenteras i närmare detalj i detta kapitel.



Figur 2: Systemdiagram.

4.1 Virtuell server och domän

Digitalocean är en molnbaserad tjänst som tillhandahåller virtuella servrar (VPS) (6), vilka kan användas för att köra webbapplikationer. Digitalocean valdes över andra alternativ på grund en den stora mängden guider och dokumentation som finns för tjänsten (till skillnad från en del andra alternativ). För MedImager användes en VPS för att göra webbapplikationen tillgänglig för användartester.

Den VPS-server som användes är baserad på Ubuntu 16.04 LTS, med en baskonfiguration på två processorkärnor, fyra gigabyte internminne och 40 gigabyte lagring.

För att applikationen ska vara tillgänglig och lätt att nå, köptes en domän (medimager.com) som pekar på servern hos Digitalocean.

4.2 HTTP

HTTP står för *Hypertext Transfer Protocol* och är en av de standarder som används för transport av data över internet. I standarden finns olika typer av förfrågningar som kan skickas, där de vanligaste är GET, POST, DELETE och PUT. Deras funktion redovisas i listan nedanför. Förfrågningarna beskrivs som hur de fungerar när de görs till webbserver, vilket är en typisk situation där HTTP-förfrågningar görs.

- GET: Hämta data från en webbserver.
- POST: Skicka data till en webbserver. Ett typiskt användningsfall är för att exempelvis skapa en ny användare.
- DELETE: Ta bort data från en webbserver.
- PUT: Uppdatera data på en webbserver. Används exempelvis för att uppdatera enstaka värden hos en existerande datastruktur.

Ett HTTP-meddelande består av headers och en kropp. Headers kan innehålla både metadata för meddelandet samt information som ska förmedlas till mottagaren av meddelandet. Om ett meddelande är avsett för att skicka data, såsom ett POST- eller PUT-meddelande, brukar denna data transporteras i kroppen.

4.3 JSON

JavaScript Object Notation (JSON) är ett textformat för datautbyte på objektform. Detta används främst för att skicka data från back-end till front-end. En förfrågan om data skickas av front-end till back-end, vilken då svarar med den efterfrågade datan i ett JSON-format. Responsen avkodas sedan till ett Javascriptobjekt som kan användas på front-end. Figur 3 visar ett exempel på hur ett JSON-objekt kan se ut. I detta exempel finns en lista över bilar. För varje bil finns mer ingående data, som i detta fall är märke, modell och tillverkningsår.

```
{"cars":[
    { "manufacturer": "BMW", "model": "320i E46", "
        year": "2001" },
    { "manufacturer": "Ferrari", "model": "California
        ", "year": "2010" },
    { "manufacturer": "TVR", "model": "Cerbera", "
        year": "1999" }
]}
```

Figur 3: JSON-objekt.

Genom att omvandla JSON-objektet till ett Javascriptobjekt kan data extraheras som från en vanlig array av objekt. Låt säga att objektet kallas *cars*. Om den första modellen ska fås fram anropas `cars[0].model` vilket resulterar i ”320i E46”.

4.4 Front-end

Front-end motsvarar, inom webbutveckling, den kod som exekveras på klientens system. Detta innefattar i MedImagers fall HTML (*HyperText Markup Language*), vilket är ett märkspråk som utgör grundstrukturen för en webbsida, CSS (*Cascading Style Sheets*), som är ett stilmallsspråk för att ändra utseendet på komponenterna i HTML-filerna, samt Javascript, ett skriptspråk som används för att göra en webbsida dynamisk, det vill säga ge möjligheten att modifiera innehållet efter det att sidan laddats in i webbläsaren. Javascript kan även kommunicera med back-end via så kallad AJAX (*Asynchronous JavaScript and XML*) och på så vis hämta och visa information från databasen utan att hela webbsidan behöver laddas om. Samtliga dessa språk och metoder har länge varit standard för webbutveckling och det är därför de används för MedImager.

4.4.1 SASS

För att göra skapandet av CSS-filerna smidigare används språket SASS (*Syntactically Awesome Style Sheets*) (7), som kan ses som en förlängning av vanlig CSS. Syntaxen liknar CSS, men kan bespara arbete jämfört med detta, till exempel genom att kod inte behöver repeteras i samma utsträckning. Detta eftersom SASS bland annat har stöd för variabler, nästlade väljare (*selectors*), funktioner och väljararv. När SASS-filerna kompileras, vilket i detta fall utförs med hjälp av programvaran Koala, konverteras SASS-filerna till vanliga CSS-filer som kan läsas av webbläsaren.

I figur 4 visas ett exempel på skillnaden mellan vanlig CSS och SASS. I detta exempel kan det se ut som att i princip lika mycket kod behövs för SASS som vanlig CSS, men faktum är att exempelvis färgvariablerna och mixen `=popup-image-data-size` kan återanvändas till andra väljare och egenskaper och behöver således inte dupliceras, vilket hade behövts med vanlig CSS.

4.4.2 Bibliotek för utseende

Materialize (8) är ett fritt CSS-bibliotek som använts vid design av användargränssnittet. Det består helt enkelt av färdigdesignade komponenter, vilket innebär att mindre arbete behövts lägga på själva utseendet på exempelvis knappar och inmatningsfält. Materialize använder sig av det av Google framtagna designkonceptet *material design* (9), vilket utgår från platta, enkla färger, skarpa kanter och att komponenter ska se ut som att de ligger i lager, med subtila skuggor. *Material design* är anpassat för att fungera på olika plattformar, såväl datorer som mobiltelefoner. Samtliga Googles produkter är designade utifrån detta och en rad andra kända webbplatser och mobiltelefonapplikationer har tagit efter.

Storleken på viss text behöver variera beroende på skärmupplösning. För detta används enheterna *viewport width* (vw) och *viewport height* (vh), som förhåller sig till fönstrets bredd respektive höjd. Dock får texten inte bli hur stor eller hur liten som helst. Därför används biblioteket *mm-fontsize*, ett Javascriptbibliotek som skapar två nya CSS-parametrar kallade *min-font-size* och *max-font-size*, vilka inte existerar i vanlig CSS. Dessa sätter en minimal och maximal storlek för texten.

```
//Vanlig CSS:
#popup-image-data-buttons {
  max-width: 300px;
  width: 100%;
  min-width: 250px;
  position: fixed;
}

#popup-image-data-buttons div {
    width: 50%;
}

#popup-image-data-buttons div:first-of-type {
  background-color: #a7ffeb;
}

#popup-image-data-buttons div:first-of-type:hover
     {
    background-color: #1de9b6;
}

#popup-image-data-buttons div:nth-of-type(2) {
  background-color: #64ffda;
}

#popup-image-data-buttons div:nth-of-type(2):
    hover{
    background-color: #1de9b6;
}

//Samma fast med SASS:
$primary-colour: #1de9b6
$secondary-colour: #64ffda
$tertiary-colour: #a7ffeb

=popup-image-data-size
  max-width: 300px
  width: 100%
  min-width: 250px

#popup-image-data-buttons
  +popup-image-data-size
  position: fixed
  div
    width: 50%
    &:first-of-type
      background-color: $tertiary-colour
      &:hover
        background-color: $primary-colour
    &:nth-of-type(2)
      background-color: $secondary-colour
      &:hover
        background-color: $primary-colour
```

Figur 4: CSS och SASS.

4.4.3 Jquery

För Javascript används biblioteket Jquery, som, precis som SASS för CSS, innebär att mindre kod behöver skrivas. Som exempel visas i figur 5 hur ett DOM-objekt, det vill säga en komponent definierad i HTML-koden, kan döljas med hjälp av vanlig Javascript respektive med Jquery.

```
//Vanlig Javascript:
document.getElementById("the-object").style.
    display="none";

//Samma fast med Jquery:
$("#the-object").hide();
```

Figur 5: Javascript och Jquery.

4.4.4 Angular

Angular är ett ramverk som används för att skapa webbapplikationer (10). För MedImager används den andra utgåvan av ramverket, vilken bygger på Typescript istället för Javascript, vilket föregångaren gjorde. Angular kan endast köras på en webbserver, vilket betyder att utveckling av Angularapplikationer kräver en virtuell eller fysisk sådan. Notera att webbservern som Angular körs på är fristående från MedImagers back-endserver.

Ramverket bygger på att samtliga delar av webbsidan består av flertalet komponenter, som fogas samman. Därför delas exempelvis meny och sidfot upp i separata filer, eller komponenter, och läggs samman i en huvudfil för vyerna de skall synas på.

Om ett objekt ska repeteras ett stort antal gånger fast med olika innehåll, till exempel i form av rader i en lista, kan detta göras med en komponent som har platshållare för låt säga text och bilder.

En komponent består av huvudsakligen två saker: en HTML-mall och en Typescriptfil som innehåller logiken för mallen. Logiken kan till exempel innefatta villkorlig navigering beroende på vilken sida användaren befinner sig. För att en komponent ska kunna hämta data från en server använder sig Angular av en så kallad serviceklass. En sådan klass tillhandahåller funktionalitet för att hämta och skicka data mellan Angularapplikationen och en server.

Angular valdes dels för att det fungerar bra ihop med ett REST API, i och med

att Angular är helt fristående från back-end, och dels för att det har ett naturligt sätt att dela upp koden i olika moduler efter funktion.

4.5 Back-end

Back-end avser den del av en applikation vars kod exekveras på serversidan. Klienten kan således inte se källkoden för back-enddelen. I kontrast till front-end, som mer har hand om de delar av en applikation som användaren kan se, sköter back-end större delen av logiken i en applikation. I en back-end kan det exempelvis inrymmas funktioner för hantering av data och användarkonton samt databaser för att lagra denna information. I detta avsnitt gås det igenom de tekniker och metoder som använts för implementationen av back-end.

4.5.1 MedView core

MedView core (11) är ett bibliotek skrivet i Java för hantering av en mvd-databas. Biblioteket innehåller funktionalitet för en rad olika ändamål vid arbete med databasen. Bland annat finns funktionalitet för datahantering, såsom extrahering av examinationer och bilder, modifikation av trädfiler med mera, men även GUI-relaterad funktionalitet för att ge stöd för utveckling av program med användargränssnitt som ska arbeta med databasen.

4.5.2 Tomcat

Tomcat (12) är en webbserver för Java utvecklad av Apache. Tomcat valdes eftersom stöd hos alla de större utvecklingsmiljöerna gjorde det möjligt att utveckla lokalt. För utvecklarna känns tryggt att veta att resultatet blir detsamma då koden körs på servern.

4.5.3 REST API

REST (*Representational state transfer*) är en vida använd metod för en webbtjänst att tillgängliggöra funktionalitet för klienter via ett API bestående av URL:er och HTTP-förfrågningar. Det finns flera fördelar med ett REST API gentemot andra metoder som en webbtjänst kan bruka för att tillgängliggöra sina tjänster. En är att protokollet för kommunikation med webbtjänsten är relativt simpelt, då en klient som vill dra nytta av webbtjänsten i princip endast

behöver veta på vilken URL den kan nå en tjänst samt eventuella parametrar som behöver förses i HTTP-förfrågan. En annan fördel är att en webbtjänst med ett REST API typiskt sett inte upprätthåller någon sessionsinformation om klienten, något som bidrar till att öka prestandan och skalbarheten hos webbtjänsten.

Exempel på företag med webbtjänster vars funktionalitet erbjuds via REST API:er är Twitter (13) och PayPal (14). Som exempel följer nedan ett utdrag ur hur det via Twitters REST API kan utföras en sökning efter de mest populära inläggen om ”bilar” på Twitter.

```
GET https://api.twitter.com/1.1/search/tweets.json?q=bilar\&result
\textunderscore type=popular
```

4.5.4 Jersey

JAX-RS (*Java API for RESTful Services*) (15) (16) är en API-specifikation i Java som ger ett stöd för utveckling av webbtjänster enligt REST-modellen. Jersey (17) är en implementation av denna specifikation. Utöver funktionerna som JAX-RS specificerar erbjuder Jersey även extra funktionalitet som ytterligare underlättar utvecklingen av webbtjänster enligt REST-modellen.

Då back-end är skrivet i Java var det lämpligt att välja ett ramverk för utveckling av REST API:er som är kompatibelt med Java. Valet föll då naturligt på att använda sig av Jersey.

4.5.5 MySQL

En relationsdatabas används för att lagra exempelvis användarinformation, egenskrivna anteckningar, samlingar och annan användargenererad data. SQL (*Structured Query Language*) är ett databasspråk som innebär att förfrågningar om information skickas från back-endkoden till databasservern. Databasservern svarar med en lista innehållandes den efterfrågade informationen, alternativt modifierar tabellerna i databasen efter direktiv. I figur 6 kan ett exempel på en SQL-förfrågan ses.

```
SELECT userid, CONCAT(firstname, ' ',
    lastname) AS name FROM users WHERE
    userlevel < 2
```

Figur 6: SQL-exempel.

MySQL (18) är ett databashanteringssystem som använder sig av en version av SQL-språket. Anledningen till valet är att källkoden är öppen och därför fri att använda.

4.6 Tekniker och metoder för hantering av säkerhet

I detta avsnitt gås det igenom de tekniker och metoder som använts för att hantera säkerheten med applikationen. Eftersom informationen i databasen är sekretessbelagd behöver den skyddas, detta görs genom att vidta olika säkerhetsåtgärder.

4.6.1 HTTPS och SSL

Internet är ett okrypterat nätverk och data kan därför avläsas av icke behöriga personer. Detta går att undvika genom att använda HTTPS istället för HTTP när data skickas över internet. Det enda som skiljer HTTPS och HTTP är att HTTPS krypterar trafiken med hjälp av SSL.

SSL är ett protokoll för att kryptera trafik mellan två enheter. För att kunna identifiera avsändaren används ett så kallat SSL-certifikat, som fungerar som ett id-kort.

4.6.2 Kryptografisk hashfunktion och bcrypt

En kryptografisk hashfunktion (19) (20) är en algoritm som avbildar data, exempelvis en textsträng, mot en ny sträng av bestämd längd. Algoritmen är irreversibel, vilket betyder att det praktiskt taget är omöjligt att avbilda den nyskapade strängen mot den ursprungliga datan. Just därför är kryptografiska hashfunktioner användbara vid säkerställandet av ett lösenords integritet. Om den hashade versionen av lösenordet kommer i fel händer går det inte att utvinna det riktiga lösenordet ur det.

Ett enbart hashat lösenord är dock fortfarande sårbart för så kallade ordboksattacker, om den råkar hamna i fel händer. En ordboksattack kan gå till som så att angriparen av lösenordet besitter en ”ordbok” med ett stort antal ord ihopparade med sina hashade versioner. Ofta är dessa ord valda för att de dömts som sannolika att användas i ett lösenord. Det angriparen då kan göra är att matcha det hashade lösenordet med de hashade orden i ordboken. Om det hashade lösenordet matchar kan det riktiga lösenordet utvinnas. Innehåller ordboken väldigt många ord finns det en relativt stor risk att lösenordet kan äventyras.

För att skydda mot detta kan ett ”salt” användas. Ett salt är en slumpmässig sträng som konkateneras före lösenordet. Det är sedan denna nya konkatenerade sträng som hashas och lagras i användardatabasen. Eftersom saltet är slumpmässigt är det osannolikt att den konkatenerade strängen ingår i en ordbok avsedd för en ordboksattack.

bcrypt (21) är en kryptografisk hashfunktion. Utöver all den funktionalitet som har nämnts har bcrypt den ytterligare fördelen att den, i jämförelse med andra kryptografiska hashfunktioner, är relativt långsam. Det vill säga att det tar relativt lång tid att hasha en sträng. Detta är positivt ifall en angripare får tag på ett salt som använts vid hashning av ett lösenord. Det angriparen då kan tänkas försöka göra är att skapa en ny ordbok med samma ord som tidigare fast med saltet konkatenerat före alla ord. Men eftersom bcrypt är långsamt skulle denna nya ordbok ta alltför lång tid att skapa.

På grund av vikten att hålla datan i applikationen så säker som möjligt, samt bcrypts goda anseende i IT-världen, ansågs bcrypt vara rätt val för MedImager.

jBCrypt (22) är en implementation av bcrypt i Java. Förutom hashningsalgoritmen innehåller implementationen även funktionalitet för generering av salt och inkorporering av det i lösenord samt verifiering av ett givet lösenord med avseende på ett hashat lösenord.

4.6.3 Användarautentisering med tokens

Tokenautentisering (23) är ett sätt för en användare att autentisera sig för en resurs genom att användaren visar att den innehar en särskild token, vilken är en sorts nyckel i form av en textsträng. Denna token erhålls genom att användaren exempelvis går igenom en inloggningsprocess för att komma åt resursen. Processen går då i princip till på så vis att användaren först autentiserar sig till resursen med exempelvis användarnamn och lösenord. Om denna initiala autentisering

lyckas, utfärdar resursen ett token som användaren kan använda för framtida förfrågningar till resursen. Ofta har ett token en begränsad livslängd, bland annat för att begränsa den potentiella skadan som skulle kunna orsakas ifall den blivit komprometterad. Utgångstiden lagras då på detta token tillsammans med eventuell information som kan vara användbart för servern, såsom information om användaren som tokenet blivit utfärdad till.

JWT (*JSON Web Token*) (24) är en standardiserad metod för att säkert utbyta information mellan olika parter. Ett JWT-token består av tre delar: en header, en ”kropp” (engelska: *payload*) samt en ”signatur” (engelska: *signature*). Headern innehåller information som är relevant för den som utfärdat token. Kroppen kan innehålla information såsom tokenets utgångstid eller information om användaren som tokenet blivit utfärdat till. Signaturen är det som säkerställer att tokenet inte kan manipuleras utan att den som utfärdat tokenet blir varse om det.

Den vanligaste användingen av JWT är just för tokenautentisering. Utfärdandet av ett token går då till i princip som följande. Användaren har först lyckats autentisera sig till en server med användarnamn och lösenord. Servern börjar då skapa ett token genom att lägga till information i kroppen såsom tokenets utgångstid, användarinformation med mera. Servern fyller även i headern med rätt information. Signaturen skapas genom att man, med hjälp av en kryptografisk hashfunktion, krypterar headern och kroppen tillsammans med en hemlig nyckel som bara servern känner till. Slutligen sätts dessa delar ihop och omvandlas, med hjälp av metoden Base64, till ett format som är lämpligt att skicka över internet.

När servern väl får en förfrågan av en användare som innehar ett token avkodar servern tokenet med hjälp av den hemliga nyckeln. Om tokenet är giltigt godkänner servern förfrågan. Om den är ogiltig, vare sig det är för att utgångstiden har passerat eller för att tokenet blivit manipulerat, avfärdar servern förfrågan.

Valet föll på att använda sig av tokenautentisering för MedImager på grund av de många fördelar som finns med metoden. En är att användarens lösenord endast exponeras när användaren loggar in. Detta gör tokenautentisering mycket säkrare än exempelvis *Basic authentication*, vilket är en annan känd metod för autentisering där användarnamn och lösenord helt enkelt skickas med i varje förfrågan till servern. Ett token har även fördelen att den har en tidsbegränsning för hur länge den är giltig, vilket innebär att om det skulle äventyras så är den potentiella skadan som skulle kunna orsakas troligtvis mindre än om ett lösenord hade äventyrats. Ett token passar även utmärkt ifall en användare vill ge tillåtelse till en applikation att få tillgång till dess information på en annan

applikation. Istället för att dela med sig av sitt lösenord, vilket användaren kanske inte känner sig bekväm med, kan användaren helt enkelt dela med sig av ett token.

JWT-tokens i synnerhet har fördelen att de är relativt små i storleken jämfört med andra tokentekniker, samt att de är säkrare (25). Det konfererades även med en väldigt erfaren webbutvecklare som varmt rekommenderade JWT.

JJWT (26) är en implementation av JWT i Java. Implementationen erbjuder funktionalitet för bland annat generering av en hemlig nyckel samt skapande och verifiering av tokens.

5 Arbetsmetod, implementation och tillvägagångssätt

Gruppen har haft en bestämd arbetsfördelning. Två fokuserade på back-end, två på front-end och en främst på kommunikationen därimellan. Det har relativt sällan hållits helgruppsmöten, men delar av gruppen har träffats flera gånger per vecka, i de flesta fall. Kommunikation mellan gruppmedlemmarna har till största delen skett över meddelandetjänsten Slack.

Själva utvecklingen har skett iterativt, det vill säga att saker har förbättrats med tiden. Funktioner som kräver back-endfunktionalitet har i de flesta fall implementerats först på back-enddelen, innan front-endarbetet har satts igång.

För att på ett enkelt sätt kunna samarbeta kring samma kod har Git använts. Git är ett versionshanteringssystem som innebär att det går att ångra ändringar i efterhand och rulla tillbaka till tidigare versioner av koden. Det går även att redigera samma filer samtidigt som andra, då Git ser till att ändringarna slås ihop på ett korrekt sätt. Själva Gitsystemet körs lokalt på datorn och för att alla utvecklare ska ha tillgång till filerna används Github, en webbtjänst som bygger på Git. Efter att filerna sparats i Git lokalt skickas ändringarna upp till Github, från vilket andra utvecklare kan ladda ned koden.

I detta kapitel beskrivs hur samtliga delar av MedImager byggts upp, med hjälp av de tekniker som presenterats i föregående kapitel. Till att börja med beskrivs utvecklingen av användargränssnittet samt hur säkerhet och webbservern som applikationen kör på har hanterats. Därefter gås det djupare in på implementationen som gjorts i front-end och back-end.

5.1 Användargränssnitt

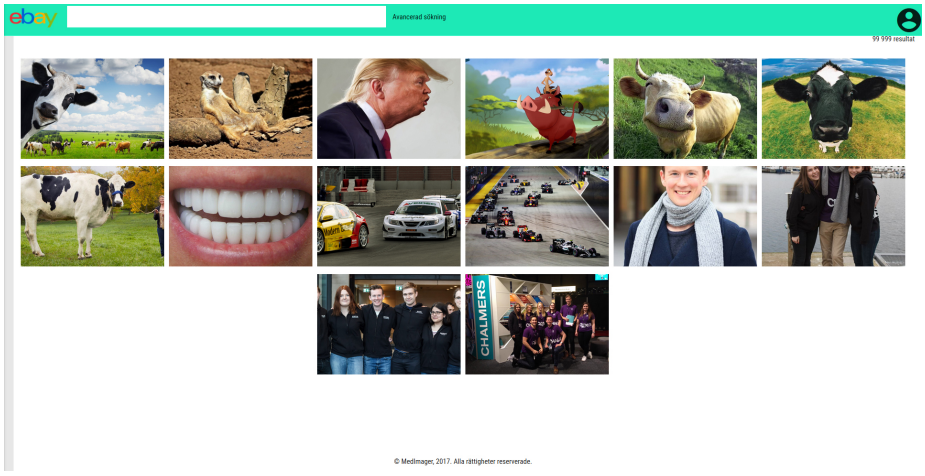
Designen av användargränssnittet för webbapplikationen har tagits fram iterativt. Som ett första stadie granskades populära webbplatser med liknande funktioner, för att hitta lämpliga sätt att implementera dessa på. Bland annat granskades bildtjänsten Flickr, reseplaneringsverktyget Rome2Rio, sökmotorn Google, det sociala nätverket Facebook och produktprisdatabasen Prisjakt. Olika idéer och funktioner från olika sidor har sedan vävts ihop för att skapa en ultimat användarupplevelse. Bland annat har funktionaliteten för att visa bilder både inspirerats av Flickrs bildsidor och Facebooks poppupprutor, samt sättet att visa sökresultat på av Googles bildsök. Inspiration från dessa sidor har tagits för att de alla är stora välkända sidor som kan antas ha testat sina funktioner väl. Dessutom används de nämnda tjänsterna av en stor mängd människor, vilket

innebär att funktionerna är välbekanta och lätta att förstå för de flesta. Detta uppfyller kriterierna för vad som av Jenifer Tidwell i boken *Designing interfaces* (27) kallar för *habituation*, det vill säga att användare föredrar när gränssnitt fungerar som något de är vana vid.

En viktig aspekt vid design av ett grafiskt gränssnitt är att det inte bör finnas för många störande komponenter och text som förvirrar användaren. Det bör även vara lätt att återgå till ett tidigare stadie om användaren skulle utföra en felaktig handling. Dessa aspekter är sammanfattade under begreppet *satisfaction* av Tidwell (27). Därför har ett minimalistiskt gränssnitt prioriterats, med väldigt lite text och få knappar.

Som ett andra steg skapades skisser med papper och penna. Dynamiska element, såsom poppuppfönster, klipptes ut för att kunna placeras över huvudvyerna. Dessa skisser visades sedan för kunden, för möjlighet till respons. I och med de urklippta komponenterna, kunde kunden interagera och testa användargränssnittet i ett mycket tidigt stadie. Kundens åsikter togs emot och gränssnittet kunde justeras innan allt för mycket arbete hunnit utföras.

Den iterativa utvecklingen av användargränssnittet har lett till att det förbättrats kontinuerligt. Till en början låg fokus på att få ihop ett funktionsdugligt gränssnitt för att funktioner skulle kunna testas. Komponenter har sedan lagts till, tagits bort och modifierats för att dels förbättra interaktionen och dels för att göra webbapplikationen mer estetiskt tilltalande.



Figur 7: Söksidan, så som den såg ut i slutet av februari.

Till en början hade front-end ingen koppling till databasen. För att användargränssnittet då skulle kunna utvecklas, lades statiska platshållarelement in, det vill säga andra bilder och påhittad metadata, vilket kan ses i figur 7. I annat fall hade all back-endkod behövt skrivas färdigt innan användargränssnittet kunnat påbörjas, vilket hade tagit betydligt längre tid.

5.1.1 Implementation av användargränssnitt

På grund av att MedImager är en webbapplikation har användargränssnittet implementerats med språk anpassade för visning i webbläsare. Standard för vanliga webbsidor är märkspråket HTML, som innehåller de element som används på sidan. Detta kan exempelvis vara textrutor, inmatningsfält, bilder och knappar. Dessa elements utseende specificeras sedan av stilmallar, vilka i MedImagers fall är skrivna i SASS, som kompileras till CSS. På grund av stilmallarna är det mycket enkelt att ändra utseendet i efterhand.

De delar av användargränssnittet som går att interagera med, såsom att visa och dölja menyer, är implementerade med Jquery och i vissa fall även till viss del genom Angular.

Samtliga filer på front-end är sammansatta med Angular. Varje undersida består av en Angularkomponent som klistras in på en huvudsida, vilken sedan returneras till webbläsaren. Komponenter finns även för element som repeteras, antingen flera gånger på samma sida, eller på flera sidor. Detta innefattar sidfot, toppmeny, sidopanel, formulär för avancerad sökning, tumnagelbilder och poppupprutor.

5.2 Implementation av säkerhet

För att kunna garantera att MedImagers data, såväl användarinformation som bilderna med tillhörande patientinformation, är skyddad användes flera metoder för att förhindra att obehöriga får tillgång till datan.

I första hand används inloggningsystemet, där en administratör manuellt godkänner användare, för att bara utvalda individer ska ha tillgång till informationen. Trots detta måste dock datan skyddas även under transport och under lagring.

För att all data ska vara skyddad krävs det att all trafik krypteras när den skickas från server till klient och vice versa. För att uppnå det skickas all data via protokollet HTTPS. HTTPS krypterar då all trafik, och förhindar därmed att någon utomstående kan läsa innehållet i trafiken.

5.3 VPS (Digitalocean)

Med hjälp av en guide på Digitaloceans webbsida (28), konfigurerades brandväggen och användare. Därefter installerades *nginx*, vilket är en HTTP-server med andra mer avancerade funktioner (29), och konfigurerades så att trafik till *www.medimager.com/api/...* skickas vidare till MedImagers back-endserver och all övrig trafik till Angularservern. Båda körs lokalt på maskinen men på olika portar. Detta gjordes med hjälp av en annan guide som tillhandahölls av Digitalocean (30).

5.4 Front-end

Kapitlet innehåller hur de kritiska funktionerna är implementerade i Angular och hur kommunikationen mellan front-end och back-end fungerar från användarens perspektiv.

5.4.1 Inloggningssystem

För att applikationen ska kunna skalas till en stor användargrupp använder sig inloggningsystemet av tokens som innehåller all sessionsinformation som tjänsten behöver. Denna token lagras sedan krypterad i webbläsaren, och skickas till servern i varje anrop där användaren behöver verifieras.

Ett token består i detta fall av ett JSON-objekt som innehåller den information som servern behöver för att kunna verifiera att användaren har rätt behörighet till resursen i fråga. Tokenet krypteras sedan för att det inte ska gå att manipulera det i webbläsaren av tredjepart, antingen användare eller någon som skulle komma över tokenet på annat sätt.

Front-enddelen av inloggningssystemet sköts av Angular. När användaren fyllt i formuläret skickar Angular en POST-förfrågan till back-end. Inloggningsuppgifterna jämförs med de som är lagrade i databasen. Om uppgifterna är korrekta svarar back-end med ett token och användaren skickas vidare till startsidan. Om inloggningsuppgifterna är inkorrekta visas ett felmeddelande.

Registrering av en användare sker på liknande sätt som inloggningen. En POST-förfrågan skickas till servern. Om informationen är giltig kommer användaren läggas i en kö i väntan på godkännande från administratör och ett meddelande visas. Om inloggningsuppgifterna är inkorrekta visas ett felmeddelande.

5.4.2 Hämtning av bilder

Utöver originalbilderna, finns även en komprimerad och förminskad version av varje bild. Den minskade filstorleken leder till att mindre mängd data behöver skickas och tas emot vid varje sökning.

Miniatyrbilden fås genom att skicka en GET-förfrågan till */api/thumbnail/examinationID/bildIndex*, där *examinationID* är id:t för den examination som bilden är från, och *bildIndex* är vilken av bilderna från examinationen som önskas. Detta eftersom en examination kan innehålla flera bilder. Miniatyrbilderna visas vid sökning och i samlingar.

På samma sätt fungerar det för att få originalbilden, fast istället skickas en GET-förfrågan till */api/image/examinationID/bildIndex*. Dessa bilder går att nå från front-end på */image/examinationID/bildIndex* alternativt genom att klicka på en miniatyrbild i sökningen.

5.4.3 Hämtning av examinationer

Specifik information gällande en särskild examination fås genom att skicka en GET-förfrågan till */api/examination/examinationID*.

Användare får ingen information som kan identifiera patienten i en examination. För att få alla examinationer tillhörande samma patient, skickas istället en GET-förfrågan till */api/patient/examinationsID*. Således behöver inte en *patientID*-variabel introduceras och patienternas identitet förblir anonym.

5.4.4 Sökning

Den enkla sökfunktionen söker igenom nuvarande diagnoser, med hjälp av en autokompletteringsfunktion som föreslår termer som finns i bilddatabasen. En term kan väljas åt gången.

Sökorden lagras i en serviceklass i Angular som sedan är ansvarig för att översätta dess interna datastruktur till en sträng som skickas med som URL-parametrar när GET-förfrågan skickas till servern. Detta sker när användaren klickar på sökknappen.

När GET-förfrågan når servern läses först sökorden in från URL:en och därefter gör Jersey ett metodanrop till Medcorebiblioteket som, utifrån bilddatabasen,

svarar med en lista med examinationer som matchar sökningen. Listan skickas sedan tillbaka till klienten som då skickar ytterligare GET-förfrågningar för varje individuell bild.

5.4.5 Avancerad sökning

Den avancerade sökningen fungerar på så sätt att användaren får välja, utifrån en lista, vilket fält och vilken term denna önskar söka på. Det går att lägga till och ta bort sökfält efter behov. De sökbara termerna genereras då servern startas, och ligger sedan statiskt i internminnet. Denna optimering leder till ökad tid för uppstart av server men sparar klockcykler då servern är aktiv.

Återigen lagras de fält och termer användaren valt i olika listor i en service som sedan översätts till en sträng som skickas med som URL-parametrar när GET-förfrågan skickas till servern då användaren klickar på sökknappen.

5.5 Back-end

I detta avsnitt gås det igenom hur de olika delarna av back-end har implementerats. Back-end är implementerat i Java, och körs på webbservern *Tomcat*. Funktionaliteten görs tillgänglig via ett REST API utvecklat med hjälp av Jersey. Sökning implementeras med hjälp av Javabiblioteket *MedView core*, vilket utnyttjas för extrahering av examinationer och bilder ur mvd-databasen. Autentisering åstadkoms via tokenautentisering med hjälp av JJWT. MySQL används för att upprätthålla en databas för användarinformation, bildsamlingar med mera. Säker lagring av lösenord uppnås genom hashning av lösenord med hjälp av jBCrypt.

5.5.1 REST API

Jersey används som ramverk för utformandet av REST API:t. I Java kan en annotation sättas innan en metod, som påverkar hur metoden kommer att användas. Jersey tillgängliggör funktionalitet i back-end till ett REST API genom att ett antal olika annotationer kan sättas på Javametoder. Jersey applicerar sedan rätt funktionalitet beroende på vilka annotationer som har satts.

Annotationen *@Path* binder en URL till en specifik metod. En sådan metod kan då kallas för en ”resurs” (engelska *resource*). Exempelvis annoteras resursen i back-end som gör det möjligt för en användare att logga in i applikationen med

@Path("/user/login") tas. Dessa URL:er är relativa till bas-URL:en för REST API:t, vilket är *medimager.com/api*.

@GET, *@POST*, *@DELETE* med flera är annotationer som används för att bestämma vilken HTTP-metod som ska användas för att få tillgång till en viss resurs. Denna annotation identifierar tillsammans med *@Path*-annotationen en unik resurs. Enligt HTTP-specifikationen används bland annat annotationen *@GET* för resurser som returnerar något och *@POST* för resurser till vilka information kan laddas upp.

Annotationen *@Produces* sätts på en resurs som producerar ett objekt av något slag som returneras i ett HTTP-meddelandes kropp. Annotationen får då HTTP-meddelandet att kommunicera till mottagaren vad innehållet i meddelandets kropp är. Dock används annotationen framförallt för att dra nytta av en inbyggd funktionalitet i Jersey som omvandlar ett Javaobjekt till en representation av detta objekt. Vad denna representation är beror på argumentet i *@Produces*-annotationen. *@Produces(MediaType.APPLICATION_JSON)*, exempelvis, omvandlar ett Javaobjekt till ett JSON-objekt som representerar det ursprungliga Javaobjektet. I MedImager är objekten som returneras uteslutande av typen JSON.

För att kunna returnera objekt i form av JSON på detta sätt måste dock Javaklassen för objektet ha en särskild struktur. Bland annat måste varje instansvariabel ha get- och set-metoder samt att en tom konstruktor måste finnas tillgänglig. Javaklasser som skapas för att representera exempelvis användare och samlingar måste därmed utformas på ett specifikt sätt. Utöver egenskapade klasser kan även en del av Javas egna klasser returneras på detta sätt. Ett exempel på en sådan klass är Javas *List*. Förutsättningen är dock att objekten inrymda i *List*-objektet även de uppfyller kravet för att kunna omvandlas till JSON.

På liknande sätt fungerar annotationen *@Consumes*. Ponera exempelvis att en resursmetod med annotationen *@Consumes(MediaType.APPLICATION_JSON)* har som metodargument ett objekt av typen *User*. Om resursen då skulle ta emot låt säga en POST-förfrågan innehållandes en JSON-representation av ett *User*-objekt, skulle detta objekt automatiskt omvandlas till dess Java-form.

Många resurser kräver att de blir försedda parametrar såsom användar-id, lösenord, autentiseringstoken med flera. Dessa går att förmedla till en resurs på en rad olika sätt. Ett sätt är att parametern förses direkt i URL:en. Även här har Jersey inbyggd funktionalitet för detta ändamål. Detta åstadkoms först genom att resursen i sin *@Path*-annotation explicit uttrycker att den förväntar sig en parameter. En sådan *@Path*-annotation är *@Path("/admin/getuser/{userid}")*, där *userid* är parametern i fråga. För att sedan extrahera parametern använder sig

resursmetoden av annotationen *@PathParam* i sin parameterlista. Javametoden för ovannämnda resurs extraherar då parametern enligt följande: *public User getUser(@PathParam("userid") String userid)*{...}.

Ett annat sätt att förse en resurs med parametrar är via headers i HTTP-förfrågan som skickas till resursen. Detta används bland annat vid registrering av en användare, vilket gås igenom i större detalj senare. Där krävs att användarinformation förses i ett antal headers. Användarens förnamn, exempelvis, kan anges enligt följande: *FirstName: Nils*. På liknande sätt som för en URL-parameter kan värdet på headern erhållas genom användning av annotationen *@HeaderParam* i resursmetodens parameterlista. Metoden kan då se ut enligt följande: *public void register(@HeaderParam("FirstName") String firstName ....)*{...}.

Många fler annotationer och funktioner finns att tillgå, varav ett flertal gås igenom nedan.

5.5.2 Överblick av sökning

En mvd-databas består, som beskrivet i kapitlet *Teknisk bakgrund*, av examinationer och tillhörande bilder, där en examination är en representation av en klinisk undersökning av en patient. Varje examination har ett antal termer såsom *Allergy* och *Drug*, vilka kan ha värden som {*Pollen, Stenfrukter*} respektive *Aspirin* givna till sig. Den för MedImager kanske viktigaste termen är *Photo*, vars värden är länkar till bilder i databasen associerade till examinationen i fråga. Bilderna kan inte direkt hämtas ur databasen, utan det är via länkarna i examinationerna de kan nås. Det som alltså söks efter i databasen är först och främst examinationer. Bilderna kan sedan fås via länkarna i examinationerna och de får indirekt examinationens egenskaper knutna till sig som en sorts metadata. *MedView core* tillhandahåller metoder för extrahering av examinationer ur databasen samt utvinnandet av bilder knutna till dessa examinationer.

Sökningen fungerar som så att examinationer filtreras på en rad olika parametrar specificerade av användaren. Om examinationen satisfierar parametrarna ingår den i sökresultatet (och därmed även bilderna knutna till examinationen). Parametrarna som filtreras på är användarens önskade värden på en examinations termer. Mer specifikt, en sökning skulle exempelvis kunna vara "Jag vill ha alla examinationer (och tillhörande bilder) där ett värde på termen *Dis-now* är *Diabetes*". Eftersom värdet på termen *Dis-now* är de aktuella sjukdomar som patienten som undersökts har, blir sökresultatet alla examinationer och bilder som matchar på patienter som har diabetes. Det ska noteras att endast

ett antal termer är av intresse för MedImager, detta efter vad som fastställdes under önskat funktionalitet (se 2.1).

Ovannämnda exempelsökning kan ses som att kravet *Dis-now=Diabetes* ställs på sökningen. En sökning kan ha hur många sådana krav som helst. Att kraven *Term1=a* och *Term2=b* ställs resulterar i sökningen "Jag vill ha alla examinationer där ett värde på *Term1* är *a* och där ett värde på *Term2* är *b*". Detta innebär att ju fler sådana krav som ställs desto stramare blir sökningen. Det kan även ställas flera krav på samma term. Kraven *Term1=a* och *Term1=b* innebär sökningen "Jag vill ha alla examinationer där värdena på *Term1* inkluderar *a* och *b*".

Ett krav som skiljer sig från resterande är det som specificerar vilken ålder patienten får ha. Detta krav uttrycks som ett spann i form av två delkrav: den lägsta åldern samt den högsta åldern. Om inget krav på ålder specificeras tas ingen hänsyn till ålder under sökningen. Om endast lägsta åldern specificeras tas ingen hänsyn till högsta åldern och vice versa.

5.5.3 Implementation av sökning

Resursen för sökning nås via URLen */search* och en GET-förfrågan. Parametrarna för sökningen förses i URL:en, men på ett något annorlunda än vad som beskrevs i avsnittet om REST API:t. Ett tillägg till en URL av formen *?a=b&x=y* tolkar Jersey som att parametrar har försetts till resursen i fråga. I ovannämna exempel är *a* och *x* namnet på parametrarna, medan *b* och *y* är de respektive parametervärdena. Det är just denna funktionalitet som används för att förmedla sökparametrar. Formen på URLen blir då enligt följande: */search?Term1=a&Term2=b& ...&ageLower=x&ageUpper=y*. Som beskrivet i föregående avsnitt står *Term1=a* för att användaren vill filtrera på examinationer vars värden på termen *Term1* inkluderar värdet *a*. *ageLower* och *ageUpper* används för att specificera ett åldersspann.

Parametrarna extraheras med hjälp av Javaklassen *UriInfo* som Jersey tillhandahåller, vilken erbjuder funktionalitet för utvinning av information ur en resurs URL. *UriInfo* tar parametrarna i URLen och skapar ett *MultivaluedMap*-objekt som associerar parameternamnen till sina värden. Detta *MultivaluedMap*-objekt används för att skapa ett *SearchFilter*-objekt, vilket organiserar parametrarna så att dessa kan användas vid filtrering av examinationer.

Med hjälp av funktionalitet i *MedView core* kan examinationer hämtas ur databasen och därefter även data extraheras från dem. Något förenklat går processen

till som så att alla examinationer först erhålls i ett fält. Examinationerna kan nås genom att iterera över detta fält. Varje examinations data testas sedan mot *SearchFilter*-objektet för att avgöra huruvida examinationen satisfierar sökningen eller inte.

Om en examination satisfierar sökningen skapas en representation av den i Java i form av ett *Examination*-objekt. Detta objekt är ingen fullständig avbild av examinationen då det enbart innehåller information som är relevant för MedImager. Objektet ska alltså inte förväxlas med den ursprungliga examinationen i databasen vilken innehåller ytterligare information om undersökningen och patienten som undersökts. Information som ingår i ett *Examination*-objekt är alla termer som går att filtrera på, såsom patientens ålder och diagnos, samt framförallt länkar till bilderna associerade med examinationen.

Ett *Examination*-objekt som satisfierar sökningen läggs till i ett Java *List*-objekt. När sökningen är fullbordad returneras slutligen detta *List*-objekt i form av ett JSON-objekt.

5.5.4 Användarhantering

Användarinformation lagras på back-end i en MySQL-databas. Information som lagras om en användare är användarid, användarnamn, lösenord (hashat), förnamn, efternamn samt användarbehörighet. En användare registrerar sig till MedImager genom att skicka en POST-förfrågan REST API-resursen */user/-register*. All användarinformation förutom användarbehörighet, vilket bestäms av en administratör för MedImager, och användarid, vilket sätts automatiskt i databasen, måste anges i särskilda headers. Exempelvis måste användarnamnet ges som ett värde till headern *Username*. Resursen erhåller användarinformation från dessa headers med hjälp av tidigare nämnda annotationen *@HeaderParam*.

Om användarnamnet inte redan är upptaget lagras användarinformation i användardatabasen i väntan på godkännande av en administratör för MedImager. Användaren blir tilldelad ett användarid och användarbehörigheten sätts till ”0” för att indikera att användaren ännu inte har behörighet att komma åt resurser på back-end. Om användarnamnet är upptaget returneras ett felmeddelande för att kommunicera detta. Lösenord lagras hashade för att inte äventyra dem vid ett eventuellt intrång i användardatabasen. Detta åstadkoms med hjälp av Javabiblioteket *jBCrypt*. Ett lösenord hashas med ett salt för extra säkerhet.

När en användare godkänts av en administratör (hur detta går till gås igenom i mer detalj senare) måste den logga in för att få tillgång till resurser på back-

end. Detta görs via resursen */user/login*. Användarnamn och lösenord används för inloggning och måste anges i särskilda headers på *POST*-förfrågan till resursen. Resursen extraherar denna information och kontrollerar om den matchar mot en användare i användardatabasen. Om användarnamnet inte existerar i databasen eller om lösenordet inte stämmer för användarnamnet returneras ett felmeddelande. Ett felmeddelande returneras även om en användare med behörighet ”0” försöker logga in.

Om inloggningen lyckas genereras ett token som användaren kan använda för att komma åt resurser på back-end. Detta token är ett *JWT-token* och genereras med hjälp av Javabiblioteket JJWT. I detta token inkluderas all användarinformation exklusive lösenord för den som tokenet utfärdats till samt tokenets livslängd. När användaren sedan drar nytta av en resurs kan denna information utnyttjas av resursen för att identifiera användaren. Mer om detta i avsnittet 5.5.5.

Det finns ett antal resurser som tillhandahåller funktionalitet för en användare att hantera sitt användarkonto. En *GET*-förfrågan till */user/getuser* returnerar till användaren all dess användarinformation exklusive lösenordet. En *PUT*-förfrågan till */user/updatepassword* updaterar användarens lösenord, där det nya lösenordet anges i headern *NewPassword*. Slutligen kan en användare avregistrera sig genom att skicka en *DELETE*-förfrågan till */user/unregister*.

Det finns även resurser avsedda för att tillåta en administratör att hantera användare. En *GET*-förfrågan till */admin/getuser/*{*userid*} returnerar användarinformation exklusive lösenord tillhörandes användaren med användarid *userid*. En *GET*-förfrågan till */admin/getusers* returnerar användarinformation för alla användare. En *PUT*-förfrågan till */admin/updateuserpermission/*{*userid*} uppdaterar en användares behörighet, där den nya behörigheten anges i headern *NewUserPermission*. Detta kan användas för att exempelvis godkänna en ny användare genom att uppdatera dess behörighet från ”0” till ”normal”. Slutligen kan en användare tas bort från databasen genom att skicka en *DELETE*-förfrågan till */admin/removeuser/*{*userid*}.

Mer information om alla REST API-resurser – inklusive vilka parametrar som måste förses, hur den returnerade datan kan se ut samt felmeddelanden som kan returneras – finns att tillgå i bilaga A.

5.5.5 Autentisering

För att få tillgång till resurser på back-end, förutom de för registrering och inloggning, måste en användare visa att den har fått tillstånd till detta. Första

steget är att användaren måste registrera sig och bli godkänd av en administratör. När användaren sedan loggar in får den ett token returnerat som kan användas för att komma åt alla resurser den har behörighet till. Detta token måste inkluderas i en särskild header på varje HTTP-förfrågan som användaren gör till back-end.

En resurs blockeras från obehöriga genom att ett sorts filter appliceras på resursen, vilket måste passeras innan resursen kan nyttjas. Detta åstadkoms med hjälp av Jerseys Javagränssnitt *ContainerRequestFilter*. Om en klass som implementerar detta gränssnitt deklareras som ett filter till en resurs måste en viss kod i klassen exekveras innan resursen kan nås. Det är i denna kod som autentiseringen av en användare utförs.

Första steget till att åstadkomma detta är att skapa en annotation i Java som kan användas som en sorts ”stämpel” på en klass. Annotationen i MedImager som skapats för detta ändamål heter *@Secured*. Denna sätts sedan på en klass som implementerar *ContainerRequestFilter* och därmed är avsedd att användas som ett filter. Klassen i MedImager som skapats för detta syfte heter *AuthenticationFilter*. Genom att annotera en resursmetod med *@Secured* skyddar man den därmed från obehöriga genom att det sätts ett krav att koden i *AuthenticationFilter* måste exekveras innan resursen kan nås. Om implementationen i *AuthenticationFilter* finner att användaren är obehörig avvisar den användaren med ett felmeddelande.

Autentiseringen går till som så att *AuthenticationFilter* extraherar tokenet (om ett sådant finns tillgängligt) ur HTTP-förfrågans header. Om tokenet är giltigt tillåts användaren att komma åt resursen som förfrågan gjorts till. Om inget token finns tillgängligt eller om det är ogiltigt, vare sig det är på grund av att dess livslängd gått ut eller av någon annan anledning, returneras ett lämpligt felmeddelande för att informera om vad som misslyckades vid autentiseringen. Mer information om felmeddelanden som kan returneras vid misslyckad autentisering finns att tillgå i bilaga A.

Hantering av användare med olika behörighet sköts med hjälp av inbyggd funktionalitet i Jersey. En förfrågan till en resurs har en livstid som pågår från det att förfrågan tagits emot av resursen fram till det att förfrågan behandlats färdigt (och resursen eventuellt returnerat något). Under denna livslängd erbjuder Jersey en del funktionalitet som körs i bakgrunden. Ett exempel på detta är ett objekt av klassen *SecurityContext* som alltid finns att tillgå. Detta objekt kan utnyttjas för att skicka runt information som kan komma till användning under en förfrågans livstid. Denna funktionalitet utnyttjas genom att behörigheten för användaren som ställt förfrågan extraheras ur tokenet den försett och sparas i *SecurityContext*-objektet, något som sköts av *AuthenticationFilter*. Genom att sedan sätta den

Jerseyförsedda annotationen *@RolesAllowed* på en resurs kan tillgång till resursen begränsas till enbart användare med en särskild behörighet. Att exempelvis annotationen *@RolesAllowed("admin")* sätts på en resurs åstadkommer att enbart användare med behörighet *admin* får tillgång till resursen.

5.5.6 Samlingar

Användare kan skapa bildsamlingar, där bilder av särskilt intresse kan grupperas och sparas. Varje samling har ett namn, en beskrivande text samt ett samlings-id. Samlingarna innehåller, i sin tur, samlingssobjekt. Dessa objekt består av ett examinations-id, en valbar notis och ett indexeringsheltal. Examinations-id tillhandahåller all information gällande examinationen samt dess bilder. Då en examination kan innehålla fler bilder, varav några inte hör hemma i samlingen, ges indexeringsheltalet för att specificera vilken bild som är relevant.

Samlingar kan delas med övriga användare, som får rättigheter att beakta samlingen men inte redigera innehållet.

Information om samlingarna och vilka bilder som finns i en samling lagras i två separata tabeller i en MySQL-databas. En tredje tabell i databasen bestämmer vilka samlingar som delas med vilka användare. Informationen hämtas vid inloggning och kan manipuleras via MedImagers API.

5.5.7 Exportera samlingar

Då användare förväntas använda andra applikationer som utnyttjar samma databas, se *3.1.4*, finns funktionalitet för att exportera en specifik samling till en egen, mindre mvd-databas. Denna ges till användare i ett komprimerat format.

6 Beskrivning av slutprodukt

I detta kapitel presenteras slutprodukten, med fokus på vad användaren ser och interagerar med. Även det framtagna API:t för att hämta information från back-end gås igenom.

6.1 Hantering av säkerhet

Applikationen använder sig av följande säkerhetsåtgärder:

- HTTPS (kryptering av trafik)
- Tokens (inloggning för att skydda resurser från obehöriga)
- Bcrypt (hashning av lösenord)

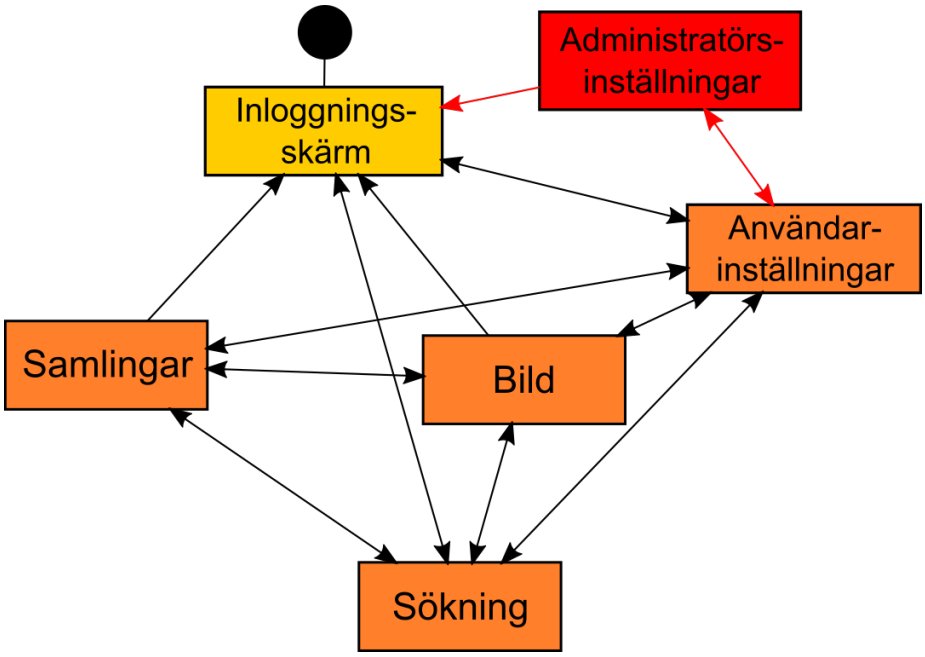
Dessa åtgärder borde teoretiskt sett skydda mot de vanligaste säkerhetsproblemen i en webbapplikation som MedImager.

6.2 Front-end

I det här avsnittet beskrivs utförligt hur applikationen fungerar.

6.2.1 Flöde

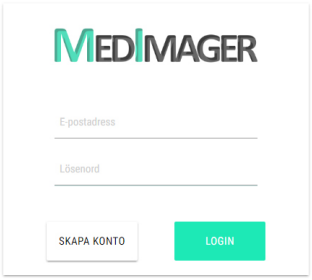
Endast en vy går att se som oinloggad. Detta är inloggningsskärmen. Alltså är MedImager helt obrukbar utan ett användarkonto. Efter att en användare loggat in, hamnar denna direkt på söksidan. Här kan användaren genomföra en sökning, eller växla till en samling via sidopanelen. Från en samling kan användaren sedan växla tillbaka till den senast gjorda sökningen. Från en sökning kan användaren gå vidare till de individuella bilderna, via en poppuppruta. Bilderna kan sedan exporteras till .mvd-filer. Från samtliga vyer kan en vanlig användare gå till sina användarinställningar. En administratör har även möjligheten att gå till en administratörsvy, där denna kan hantera användarförfrågningar och kontouppgifter. De vägar som är möjliga att ta kan ses i figur 8.



Figur 8: Diagram över flödet. Gult innebär att alla har behörighet till sidan, oranget att endast inloggade har behörighet och rött att endast administratörer har behörighet.

6.2.2 Inloggningssidan

Inloggningssidan är det första besökaren möts av när denna öppnar webbapplikationen. Det enda som här presenteras är en inloggningsruta med två inmatningsfält. Det första för användarens e-postadress och det andra för lösenordet som är kopplat till kontot. Det finns även en knapp som leder till sidan för att registrera ett nytt konto.



Figur 9: Inloggningssidan.

6.2.3 Sökning

Funktionaliteten för att söka efter bilder är huvudsakligen baserad på ett taggsystem. Varje bild har termer kopplade till sig. Det kan exempelvis handla om sjukdomar eller allergier som personen på bilden har.

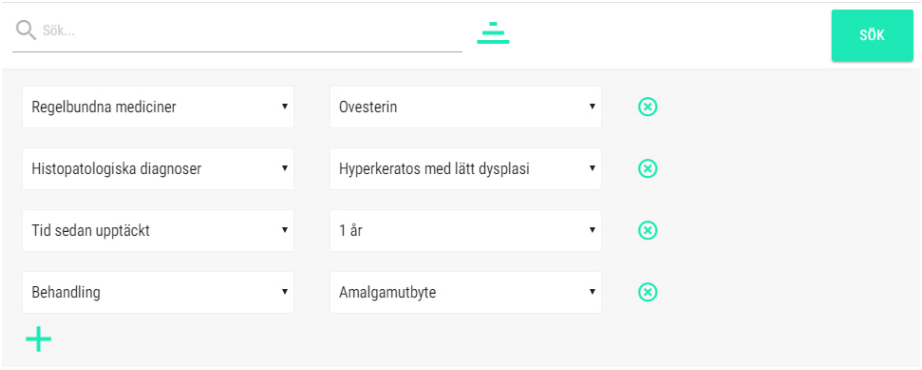
Figur 10: Sökförslag med automatisk komplettering.

På söksidan syns sökfältet i toppmenyn. När användaren börjar skriva söker applikationen automatiskt efter taggar som matchar strängen som matats in.

Dessa visas i en dropdownmeny under inmatningsfältet, vilket kan ses i figur 10. För att välja en tagg kan användaren antingen klicka på förslaget i listan, eller navigera med piltangenterna och välja tagg genom att trycka ned enter- eller returtangenten. När en tagg är vald visas den i form av ett ”chip”, som Materialize har valt att kalla det. Ett ”chip” är en ruta som innehåller dels taggens namn, men även en kryssknapp för att ta bort taggen och då även tillhörande ”chip”. Valda taggar placeras i sidomenyn under rubriken ”sökning”, vilket kan ses i figur 12, och när användaren fortsätter att skriva in fler ord i sökfältet, visas valda taggar inte längre som förslag, då det är överflödigt att välja samma tagg flera gånger.

Tagg- och söksystemet gjordes helt från grunden. Detta trots att Materializebiblioteket innehåller en funktion kallad *autocomplete*, som mer eller mindre gör samma sak som MedImagers sökförslagslista. Dock ansågs denna ha vissa brister, vilket gjorde att det valdes att göra en egen variant från grunden istället. En brist var att Materialize *autocomplete*-lista inte lade sig i ett lager över resten av sidan, utan istället knuffade ned allt innehåll för att få plats. Det hade även blivit mer komplicerat att hantera MedImagers taggsystem med detta.

Utöver att välja taggar kan användaren även filtrera sökningen på en rad andra parametrar. Detta innefattar bland annat tidigare diagnoser, läkemedel, kön på den fotograferade personen, huruvida personen brukar cigaretter och snus, med mera. Dessa typer av filtreringar kallas för ”avancerade sökningar” och reglagen för dessa fås fram genom att användaren klickar på en knapp med en filtreringssymbol, som är placerad bredvid inmatningsfältet i toppmenyn. En extrameny, vilken kan ses i figur 11 animeras då ned från toppmenyn och placeras under denna.



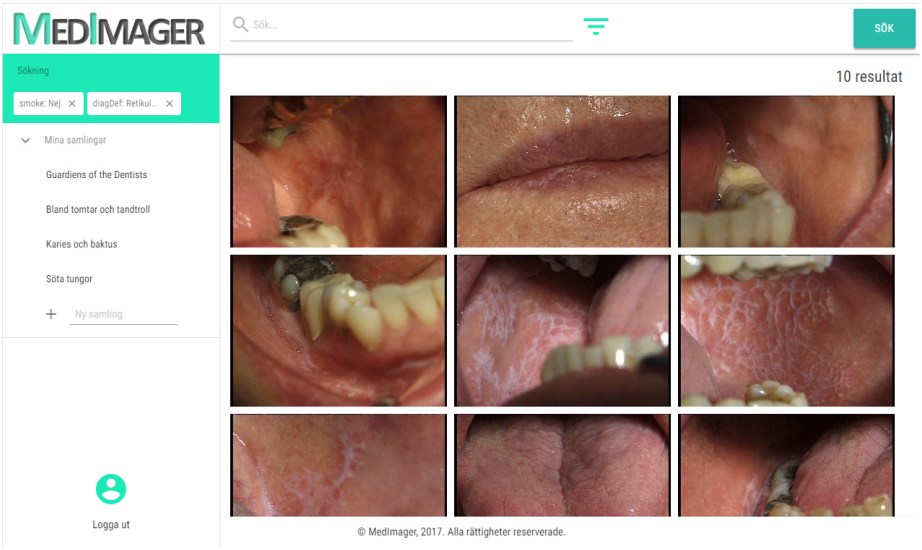
Figur 11: Inställningar för avancerad sökning.

I ett tidigt stadie användes olika typer av reglage för de avancerade inställningarna, beroende på vad som ansågs mest intuitivt för den aktuella parametern. Exempel på typer var kryssrutor, radioknappar och intervallreglage. Dock ändrades detta till den slutgiltiga produkten, efter önskemål från Sahlgrenska akademin. En önskan om att alla parametrar skulle filtreras på samma sätt fanns. Därför består den avancerade sökningen istället av rullgardinsmenyer uppdelade i två kolumner. I den vänstra kolumnen visas rullgardinsmenyer för val av filtreringsparametrar, såsom allergier. Motsvarande rullgardinsmeny i den högra kolumnen på samma rad uppdateras då med de alternativ som finns för allergifiltrering.

Sökresultaten visas på en sida som kan kallas för ”söksidan”. När en sökning påbörjas från en annan sida, vilket är möjligt från till exempel bildsidan, omdirigeras användaren automatiskt till söksidan. Redan gjord inmatning tas med från den föregående sidan till söksidan. Detta sker genom att inmatningen som redan gjorts sparas i en webbkaka (*cookie*) och läses in när söksidan laddas.

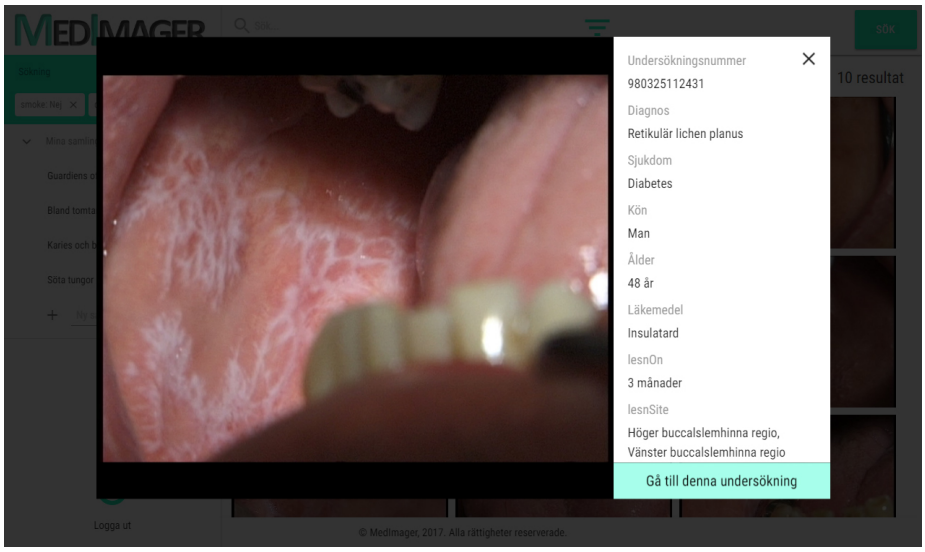
6.2.4 Söksidan

Söksidan består av ett rutnät, vilket kan ses i figur 12. I rutnätet visas bilderna som matchar den genomförda sökningen. Ingen övrig information om bilderna visas i standardläge. När användaren för musen över en bild, visas ett svart, svagt transparent lager, över bilden. I detta lager visas ett fåtal viktiga parametrar kopplade till bilden. Detta innefattar samtliga diagnoser som kopplats till bilden, hur gammal personen på bilden är samt eventuell annan viktig information. När användaren klickar på bilden dyker en poppuppruta upp, vilken kan ses i figur 13. Övriga sidan täcks av ett svart transparent lager, för att användaren ska fokusera på poppupprutan. I rutan visas bilden i stort format, och i en högerkolumn visas all metadata kopplad till bilden. Längst ned i högerkolumnen finns även en knapp som leder till bildsidan för bilden.



Figur 12: Söksidan.

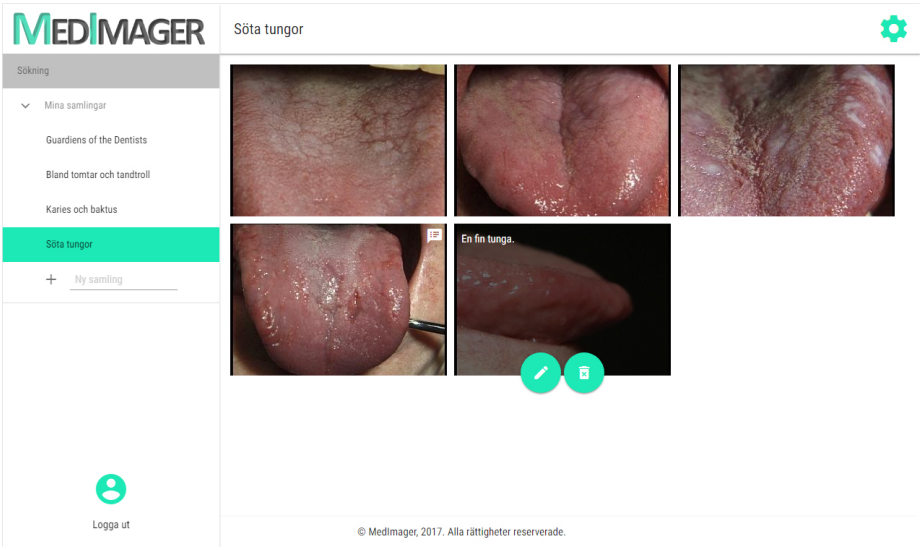
I poppuppvyn kan användaren navigera mellan bilderna i sökresultatet. Detta antingen genom vänster och höger piltangent på tangentbordet eller genom att klicka på pilknappar som visas till vänster och höger när användaren för musen över bilden. Detta följer kriteriet för *keyboard only* som definierats av Tidwell i *Designing interfaces* (27), för att minska behovet av att använad musen, och i MedImagers fall är beteendet inspirerat av det sociala nätverket Facebooks navigation i bildalbum. Användaren kan stänga poppupprutan antingen genom att klicka på ett kryss uppe i högra hörnet av rutan, eller klicka någonstans på den mörka ytan utanför poppupprutan.



Figur 13: Poppuppruta.

6.2.5 Samlingssidan och sidopanelen

Samlingssidan, vilken kan ses i figur 14, är egentligen bara en version av söksidan och fungerar på i princip samma sätt som den. Skillnaden är att namnet på samlingen visas istället för sökfunktionerna i toppmenyn, samt att bilderna som visas i rutnätet är de som tillhör samlingen. I toppmenyn finns även alternativ för att dela samlingen, radera den eller lägga till en beskrivning.



Figur 14: Samlingssidan.

Genom sidopanelen går det enkelt att växla mellan en sökning och en samling. Den senaste sökningen ligger alltid kvar som alternativ längst upp i sidopanelen. Där under visas samtliga samlingar som är skapade av eller delade med användaren. Det finns även en möjlighet att skapa nya samlingar.

Bilder kan läggas till i samlingar genom att användaren klickar och drar en bild till samlingen i sidopanelen. Detta är inspirerat av hur en användare kan lägga till en låt i en spellista på musiktjänsten Spotify. Bilder kan dras från söksidan, från en annan samling eller från bildsidan.

När en användare är inne på samlingssidan kan denna lägga till anteckningar till bilderna. Detta görs genom en redigeraknapp som är kopplad till varje bild. Anteckningarna är endast kopplade till bilderna när de visas i den valda samlingen, och kan alltså inte ses i andra samlingar eller från söksidan. De bilder som har en anteckning kopplad till sig får en liten symbol uppe i högra hörnet, för att det enkelt ska gå att se vilka bilder som har en anteckning. I figur 14 kan denna symbol synas på första bilden från vänster på andra raden. Implementationen av anteckningarna följer Tidwells tanke om *prospective memory* i boken *Designing interfaces*, det vill säga som en form av kom-ihåglapp för användaren (27).

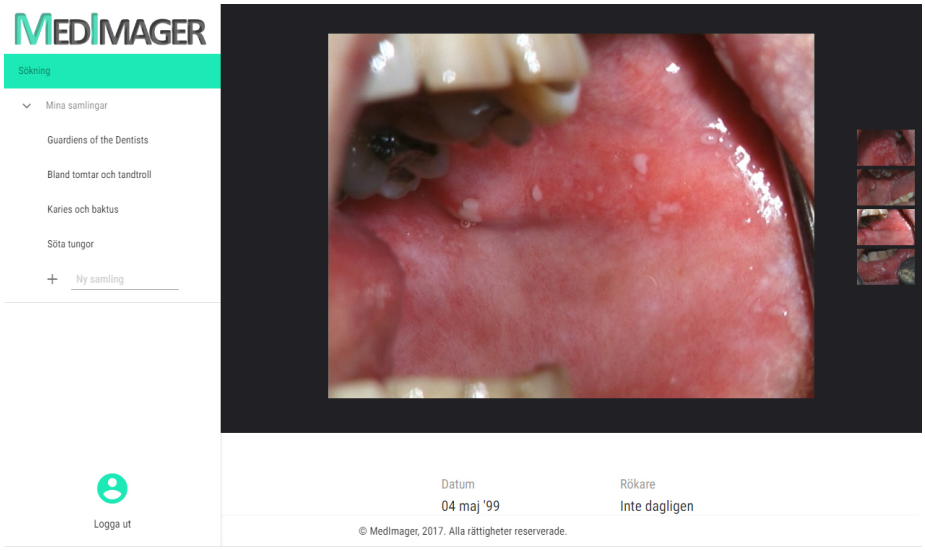
Högst upp i sidopanelen finns något som av Tidwell kallas för en *escape hatch* (27). Det innebär en knapp som leder tillbaka till startsidan, oavsett vilken sida

användaren befinner sig på. I MedImagers fall presenteras knappen i form av applikationens logotyp.

6.2.6 Bildsidan

Samtliga bilder och tillhörande data behöver kunna direktlänkas, via en permanent länk. Detta är inte möjligt i sökresultatsvyn. Därför har MedImager fått en statisk sida för varje bild, vilket kan ses i figur 15. På sidan presenteras exakt samma information som i poppupprutan på söksidan, men utöver detta även andra bilder som kan relateras till den aktuella. Detta inkluderar bilder från samma odontologbesök och andra bilder på samma patient från andra tillfällen.

Överst på sidan finns en mörk yta som spänner över hela skärmbredden. Denna yta fungerar som bakgrund för bilderna. Den valda bilden visas i ett så stort format som möjligt för den aktuella skärmupplösningen. Bilden kan inte bli högre än skärmens absoluta höjd eller bredare än skärmens bredd. Detta för att hela bilden ska kunna visas utan att användaren behöver skrolla. Vid den mörka ytans högra sida ligger små miniatyrbilder vertikalt centrerade. Detta är övriga bilder som tillhör samma undersökning. Anledningen är att det ska gå lätt att granska samma mun ur flera vinklar, vilket var något som önskades av Sahlgrenska akademin.



Figur 15: Bildsidan.

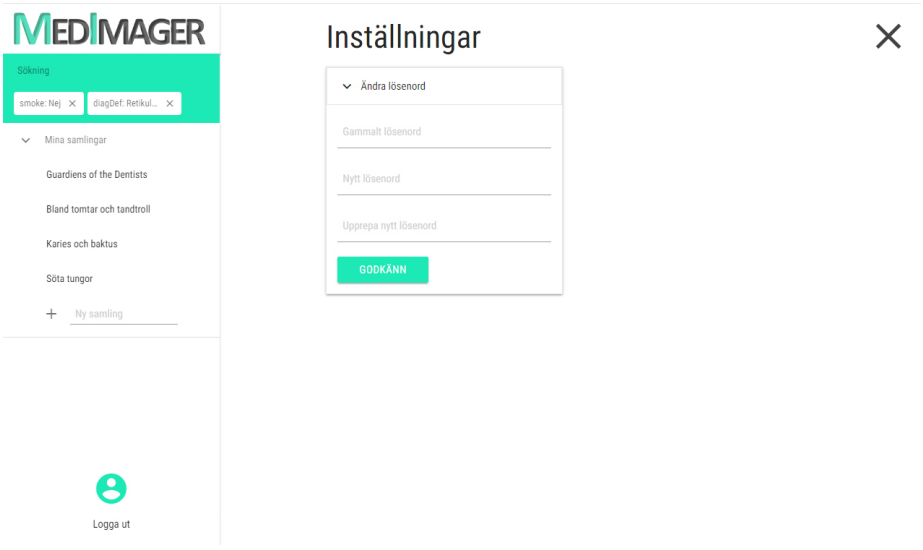
Nedanför den mörka ytan finns informationen som är kopplad till bilden. Datan är uppdelad i två kolumner för att minimera skrollbehovet. Informationen är densamma för samtliga bilder som tillhör en och samma undersökning och behöver därför inte hämtas om när användaren växlar mellan bilderna i den aktuella undersökningen.

Under informationsavsnittet visas miniatyrbilder i samma storlek som på sökresultatssidan. Dessa representerar andra undersökningar på samma patient. Undersökningarna är sorterade efter datumet ge genomfördes, med den tidigaste först. I själva miniatyrbilden står datumet samt antalet bilder som finns kopplade till undersökningen. Genom att klicka på dessa kan användaren navigera till bildsidorna för dessa undersökningar.

Upplägget för bildsidan är till viss del inspirerat av bilddelningstjänsten Flickr.

6.2.7 Inställningssidan

En funktion, som inte är primär för MedImager men ändå viktig, är att användarna ska kunna ändra sina inställningar. Genom att klicka på användarknappen längst ned i sidopanelen kommer användaren till inställningssidan. Här kan användarens lösenord bytas. Detta sker via tre inmatningsfält. I det första fältet måste det nuvarande lösenordet matas in, för att inte obehöriga ska kunna byta lösenord. I de två andra fälten ska det nya lösenordet anges. Anledningen till att två fält används är för att minska risken att användaren matar in fel.



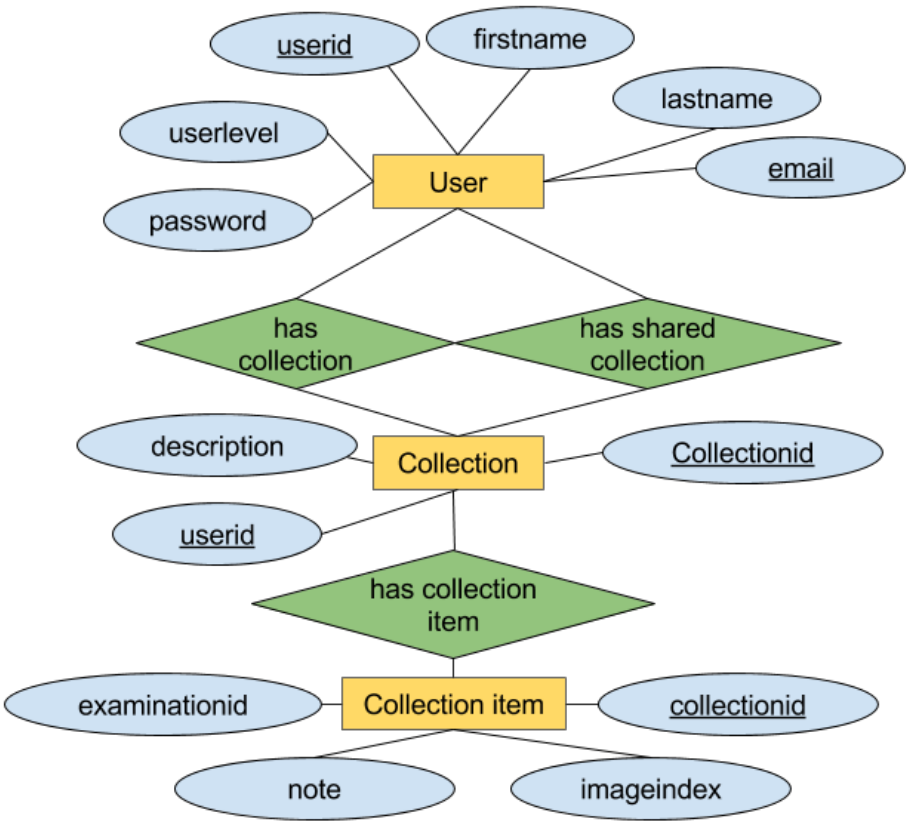
Figur 16: Inställningssidan.

6.3 Back-end

Då utveckling av back-end och front-end skett separerat har fokus vid back-end-utveckling legat på att ge logiska, enkelt tolkade resultat till förfrågningar från front-end. Dokumentation av API:t har skett kontinuerligt allt eftersom mer funktionalitet lagts till.

6.3.1 Databas

Relationsdatabasen som byggts upp innehåller information om användare, såsom kontaktuppgifter och lösenord, egna anteckningar, bildsamlingar och annan användargenererad data. Följande ER-diagram visar databasstrukturen. De gula rektanglarna motsvarar så kallade entiteter, vilka i databasen representeras av tabeller. Exempelvis finns det en tabell över användare, en tabell över samlingar och så vidare. De gröna fyrkanterna motsvarar relationer, som också representeras av tabeller i databasen. Därav uttrycket ”relationsdatabas”. Det kan sägas att relationstabellerna är vad som länkar ihop entitetstabellerna. De blåa cirklarna motsvarar attribut i tabellerna, som representeras av kolumner i databasen. Till exempel har användartabellen kolumner för användar-ID, namn, lösenord med mera.



Figur 17: ER-diagram över databasen.

6.3.2 API

Applikationens REST API har för avsikt att följa samma struktur för alla dess resurser. Då ett id-nummer används, läggs det sist i sökvägen. Flera resurser kan dela samma sökväg, och istället vara skilda beroende av vilken typ av förfrågan som skickas till resursen. Applikationens REST API återfinns i tabell 1.

Request type	**Route**	**Request body**	**Response body**
GET	search	-	[Examination]
GET	examination/:id	-	Examination
GET	patient/:id/	-	[Examination]
GET	image/examination:id/:index	-	Image
GET	thumbnail/examination:id/:index	-	Image
GET	collection	-	[Collection]
GET	collection/:id	-	Collection
GET	collection/share	-	[Collection]
GET	collection/share/:id	-	Collection
GET	collection/export/:id	-	Zip-file
POST	collection/	JSON	-
POST	collection/:id	JSON	-
POST	collection/share:id	JSON	-
PUT	collection/description	JSON	-
PUT	collection/note	JSON	-
DELETE	collection/	JSON	-
DELETE	collection/:id	JSON	-
POST	user/login	-	-
POST	user/register	-	-
DELETE	user/unregister	-	-
PUT	user/updatepassword	-	-
GET	user/getuser	-	User
GET	admin/getuser/:userid	-	User
GET	admin/getusers	-	[User]
DELETE	admin/removeuser/:userid	-	-

Tabell 1: Tabell beskrivande MedImagers REST API. Svar med hakparantes innebär lista av svar

Mer detaljerad information om alla REST API-resurser – inklusive header-parametrar som måste förses, hur den returnerade datan kan se ut samt felmeddelanden som kan returneras – finns att tillgå i bilaga A.

7 Diskussion

I detta kapitel diskuteras olika delar av projektet inklusive arbetet med vissa av de tekniker och metoder som använts under projektets gång, databasen med examinationer och bilder, feedback från kund samt slutprodukten.

7.1 Tekniker och metoder som använts

I detta avsnitt diskuteras några av de metoder och tekniker som utnyttjades under utvecklingen av applikationen. Det reflekteras över hur de varit att arbeta med inklusive vilka för- och nackdelar som upplevdes med dem med mera.

7.1.1 Server

Digitalocean gjorde det lätt att sätta applikationen i bruk tack vare att det fanns tillgängligt diverse guider för konfiguration av exempelvis nginx, Tomcat och MySQL.

Ett problem vi stötte på var att det var svårt att se hur vår applikation skalade över olika konfigurationer på servern eftersom det var onödigt komplicerat att ändra konfigurationen utan att förlora inställningar och dylikt. Därför valde vi att inte testa hur servern skalar utan fokusera på vidare utveckling.

7.1.2 Angular

För att implementera applikationens front-end har ramverket Angular använts. Angular är ett front-end-ramverk skrivet i Typescript som till skillnad från sin föregångare AngularJS behöver byggas till ett körbart format. Även om Angular kan användas utan Typescript valdes det som utvecklingsspråk för front-end-delen av applikationen, bland annat för sin skalbarhet och sin starka typning av variabler. Angularramverket bygger på en objektorienterad struktur av komponenter, tjänster och direktiv. Komponenter är de synliga delarna i applikationen i form av en vymall samt deras tillhörande modellklass. Tjänster är bakgrundsresurser, liknande designmönstret singleton, som kan anropas av alla komponenter eller direktiv som ber om en referens till dem. Direktiv är klasser som kan appliceras på komponenter eller HTML-element som kan manipulera sin komponent eller sitt element på olika sätt.

Att använda ett renodlat frontend-ramverk underlättade betydligt i och med möjligheten att dela upp front-end och back-end. Hade ett annat språk än Java använts för att kommunicera med den existerande MedView core-koden hade bindningar till MedView core behövt skapas, men i detta fall kunde hela back-end-koden skrivas i Java, och denna kunde sedan kommunicera direkt med front-end-delen av applikationen.

Då flera i utvecklingsgruppen hade tidigare vana vid serverspråket PHP och ingen hade erfarenhet av Angular råder delade meningar om huruvida detta var det mest effektiva valet. Exempelvis innebar Angulars separation av modell och vy huvudbry för de PHP-vana.

7.1.3 SASS

SASS har snabbat upp processen att ge HTML-elementen sitt rätta utseende genom möjligheten till nestade CSS-regler och användning av variabler. Variabler är extra användbart om man till exempel bestämmer sig för att ändra färgtemat.

Det enda negativa som upplevts med SASS är att koden behöver kompileras om när något ändrats. Detta innebär att SASS-filerna inte kan användas direkt som de är, till skillnad från vanlig CSS. Kompileringen genomfördes i MedImagergruppens fall med applikationen Koala.

7.1.4 Jersey

Arbetet med Jersey har i stort fungerat utan komplikationer. Detta tack vare den utförliga dokumentationen som existerar för ramverket samt den stora mängden guider som finns tillgängliga. Jersey innehåller mycket inbyggd funktionalitet som har underlättat utvecklingen av REST API:t oerhört mycket. Funktioner såsom enkel extrahering av parametrar i URL och headers, automatisk omvandling av Java-klasser till JSON-objekt med flera, har varit till stor nytta. Enda nämnvärda svårigheten som uppkom var när tokenautentisering skulle implementeras för applikationen, då Jersey inte har något direkt stöd för detta. Detta fick realiseras genom lämpligt utnyttjande av funktionalitet som Jersey erbjuder.

7.1.5 JWT

Implementering av tokenautentisering med JWT har fungerat väldigt bra tack vare Javabiblioteket JJWT, vilket erbjuder all funktionalitet som behövs för

att inkorporera JWT i applikationen. Bland annat tillhandahålls funktionalitet för automatisk generering av en hemlig nyckel, något som tagit bördan från gruppen att själva implementera detta. Dock var konceptet med JWT relativt komplicerad att begripa sig på till en början, så mycket tid fick läggas på att först bekanta sig med metoden innan arbetet med den kunde påbörjas.

7.1.6 bcrypt

Javabiblioteket jBCrypt har underlättat implementeringen av bcrypt mycket genom dess mykcet simpla gränssnitt som alltjämt tillhandahåller de funktioner som behövs för integrering av bcrypt i applikationen. Funktioner för automatiskt generering samt jämförelse av ett givet lösenord mot ett som finns i databasen har kommit till stor användning.

7.2 Mvd-databasen och MedView core

Till en början upplevdes en del svårigheter vid arbetet med databasen. Det uppfattades ganska snabbt att databasen inte såg ut som vedertagna databaser vilka till större delen utgörs av relationsdatabaser. Dessutom upplevdes dokumentationen för *MedView core* – Javabiblioteket som tillhandahölls för att hantera databasen – som inte lika väldokumenterad som konventionella Javabibliotek. Detta av naturliga skäl dock, med tanke på att biblioteket tidigare enbart använts för att utveckla applikationer som faller under projektet MedView. Mycket tid fick därför inledningsvis läggas på att experimentera med programvaran. Detta i samband med handledning från människor insatta i programvaran och databasen, vilka inkluderar handledaren till projektet, som till slut ledde till att en god förståelse kunde införskaffas. När det väl hade fåtts grepp på hur databasen var strukturerad samt hur programvaran skulle nyttjas var det relativt rättfram att implementera hur databasen skulle genomsökas.

Komplikationer dök dock upp under utvecklingsprocessen. Eftersom en examinations innehåll är beroende av vilka värden den undersökande läkaren valt att inkludera i den innebär det att olika examinationer kan ha olika struktur. Exempelvis kan en examination helt sakna värden på vissa termer eller överhuvudtaget inte ens innehålla termen. En examination kan alltså exempelvis innehålla termen *Allergy* med de tillhörande värdena *Stenfrukter, Pollen*, medan en annan kan helt sakna termen. Detta ställde till problem då det innebar att alla examinationer inte kunde hanteras på exakt samma sätt eftersom de inte är uppbyggda likadant. Hänsyn fick tas till detta under den fortsatta utvecklingsprocessen.

En annan komplikation uppkom i samband med att examinationerna i databasen var tvungna att representeras i form av objekt i Java, så att dessa sedan kan returneras till front-end. Problemet som uppstod var att namnen på termerna i examinationerna inte direkt kunde användas som namn på instansvariabler i Java, då dessa skulle strida mot Javas konventioner för namngivning. Resultatet av detta är att implementationen för överföringen av en examination till dess Java-representation innefattar en del hårdkodning för att hantera detta problem. Om databasen i framtiden skulle förändras genom att exempelvis nya termer tillkommer till en examination skulle den nuvarande implementationen behöva förändras för att möjliggöra detta.

7.3 Feedback från kund

Efter projektets genomförande hölls ett sista möte med MedImagers kund Mats Jontell där applikationen demonstrerades. Han var nöjd med applikationens funktionalitet, samt tyckte mycket om dess enkelhet. Jontell hade ett antal synpunkter på designen, de flesta relativt små. En större synpunkt han hade berörde hur söktermer representeras i den avancerade sökningen. Med hjälp från Anders Grip åtgärdades detta snabbt, och söktermer visas ej längre med databasens interna beteckning, utan skrivs ut med ett namn definierat i en konfigurationsfil. Istället för exempelvis *diagHist* står där nu *histopatologiska diagnoser*, vilket var det Jontell önskade.

7.4 Slutprodukt och utvecklingsprocess

Applikationen innehåller inte all funktionalitet som var tänkt från början. Detta är på grund av tekniska svårigheter som uppkom i samband med delar av applikationen som har med säkerhet att göra. Ett stor del av implementationen i front-end som gjordes under projektets tidigare stadier visade sig vara inkompatibelt med inloggningssystemet som senare implementerades. Främst för att det visade sig vara omöjligt att skicka med *authorization headers* när HTTP-förfrågningar görs i en HTML-mall, vilket var metoden som användes i tidigare stadier av projektet för att skicka HTTP-förfrågningar till back-end. Stöd för all önskad funktionalitet finns dock i back-end, så möjlighet för fortsatt utveckling av applikationen finns.

I och med kravet på att sökningar ska fungera med godtycklig databas kunde inte formuläret för avancerade sökningar utformas så smidigt och intuitivt som planerat. Det var istället tvunget att vara adaptivt beroende på vilken data som

fanns tillgänglig i databasen. Fördelen med detta är att vilken databas som helst som har samma struktur kan användas. Det negativa är just att inmatningstypen inte blir riktigt passande för viss data, men detta var den bästa lösningen för situationen som kunde åstadkommas. Ett exempel är att det kan anses mer intuitivt att tidsintervall filtreras med intervallreglage än med rullgardinsmenyer.

En annan tanke med applikationen var att den skulle kunna användas som stöd vid utbildning av nya studenter i odontologi. En idé som fanns till en början var att integrera stöd för Googles verktyg *Google Presentationer*, vilken fungerar på samma sätt som Microsofts välkända programvara *Powerpoint*. Meningen var att användaren med en enkel knapptryckning skulle kunna infoga en bild från MedImager i en bildskärmspresentation. Denna idé valdes dock att strykas på grund av att det var en lågprioriterad funktion som i slutändan ansågs vara alltför avancerad, och som även inte ingick i den ursprungliga kravspecifikationen. Det ansågs viktigare att lägga mer fokus på huvudfunktionaliteten i applikationen. Tanken var dock god och stöddes av en representant från Sahlgrenska akademin.

En annan begränsning i applikationen är att den inte kan presentera all information om en undersökning eller patient. Detta är en funktionalitet som återfinns i andra befintliga program som är utvecklade för att hantera en mvd-databas. Detta är ett resultat av kravspecifikationen som ställdes av kunden, vilken var att applikationen skulle fokusera på särskilda aspekter av en klinisk undersökning. Mer specifikt var önskemålet att det enbart skulle gå att filtrera sökningen på ett bestämt antal termer i en examination. Detta på grund av att dessa termer var de som kunden ansåg vara relevanta för applikationen. Det skulle gå att filtrera på termer som patientens diagnos, rökvanor med mera, men termer som patientens yrke och blödningsbenägenhet skulle ignoreras. Trots att implementation för detta saknas skulle det dock inte vara en alltför invecklad process att i framtiden anpassa applikationen så att den ackommoderar detta behov.

8 Slutsats

Projektets huvudsakliga mål var att skapa en snabb och smidig applikation för hantering av odontologiska bilder i en existerande databas. Denna applikation skulle vara till stöd för både nuvarande odontologer och studerande i odontologi. Detta mål har till stora delar uppnåtts då funktionalitet såsom sökning, filtrering samt möjligheten för en användare att ha egna samlingar finns tillgängligt. Applikationen är även relativt snabb och intuitiv i designen, något som stöddes av kunden när denne fick applikationen demonstrerad för sig.

Det saknas en del av de funktioner som från början var tänkt att ingå applikationen. Exempel på dessa är möjligheten för en användare att dela med sig av sina samlingar till andra användare samt en funktion för exportering av en samling till mvd-format. Dessa fick tyvärr utebli på grund av oförutsedda komplikationer som dök upp i projektets slutskede. Det saknas även vissa delar av applikationen som skulle behövas för att göra användarupplevelsen mer flytande. Trots det så finns det mesta som ursprungligen var tänkt att ingå i applikationen tillgängligt i back-end och är redo att byggas vidare på. Det skulle därför inte vara alltför komplicerat att åtgärda dessa brister i en eventuell framtida vidareutveckling av applikationen. Grundfunktionaliteten som byggts upp skulle även, med vissa modifieringar, potentiellt kunna utnyttjas vid utvecklingen av andra applikationer för vilka syftet är att arbeta med en mvd-databas.

En förhoppning med projektet är även att det ska kunna bidra till en mer hållbar utveckling, genom att applikationen förhoppningsvis kan hjälpa odontologer samt studerande i odontologi att arbeta effektivare. Ett resultat av detta skulle kunna vara att mer tid kan läggas på andra aspekter av arbetet eller utbildningen, vilket potentiellt skulle kunna öka kvaliteten på utbildningen eller arbetet med patienter. Att ha applikationen på en webbserver blir också mer hållbart i längden. Istället för att replikera alla data på varje system som applikationen ska användas på så har alla användare tillgång till samma databas, vilken huserar på servern.

För gruppen har detta projekt varit oerhört givande. För de i gruppen utan någon tidigare erfarenhet av webbutveckling har det varit väldigt roligt och bildande. För de med mer erfarenhet har det varit roligt att inte bara finslipa sina kunskaper men även att lära sig nya verktyg. Trots att projektet i många stunder varit frustrerande har det samtidigt varit belönande och att i slutändan komma ut med framgångsrikt resultat många gånger. Projektet har även gett insikt i hur det är att arbeta i ett för gruppen relativt stort mjukvaruprojekt. Lärdomar såsom vikten av kommunikation mellan olika delar av projektet kommer tas med in i framtida projekt som gruppmedlemmarna deltar i.

Referenser

[1] MedView. (u.d.) MedView Website. [Online]. Available: https://medview.se

[2] M. Jontell, U. Mattsson och O. Torgersson, "MedView: An instrument for clinical research and education in oral medicine," 2005.

[3] MedView. (u.d.) MVisualizer. [Online]. Available: http://www.cse.chalmers.se/research/group/medview/website/mvtour/mv4_4.html

[4] O. Torgersson. (2004) Trädfiler i MedView. [Online]. Available: http://www.cse.chalmers.se/research/group/medview/developer/docs/trees.pdf

[5] MedView. (u.d.) MedRecords. [Online]. Available: http://www.cse.chalmers.se/research/group/medview/website/mvtour/index.html

[6] DigitalOcean. (u.d.) DigitalOcean Website. [Online]. Available: https://digitalocean.com

[7] SASS. (u.d.) SASS Website. [Online]. Available: https://sass-lang.com

[8] Materialize. (u.d.) Materialize. [Online]. Available: http://materializecss.com/

[9] Google. (u.d.) Material Design. [Online]. Available: https://material.io/

[10] Angular. (u.d.) Angular Website. [Online]. Available: https://angular.io

[11] MedView. (u.d.) MedView - Developer. [Online]. Available: https://medview.se/developer

[12] Apache. (u.d.) Tomcat. [Online]. Available: http://tomcat.apache.org/

[13] Twitter. (u.d.) Twitter REST API. [Online]. Available: https://dev.twitter.com/rest/public

[14] PayPal. (u.d.) PayPal REST API. [Online]. Available: https://developer.paypal.com/docs/api/

[15] Oracle. (u.d.) JAX-RS. [Online]. Available: https://jax-rs-spec.java.net/

[16] J. Sandoval, *RESTful Java Web Services.* Packt Publishing, 2009, ch. 5. Jersey: JAX-RS.

[17] Jersey. (u.d.) Jersey. [Online]. Available: https://jersey.java.net/

[18] Oracle. (u.d.) MySQL. [Online]. Available: https://www.mysql.com/

[19] M. A. AlAhmad, I. F. Alshaikhli, "Broad view of cryptographic hash functions," *International Journal of Computer Science Issues*, vol. 10, no. 4, pp. 239–, 2013.

[20] R. Sobti, G. Geetha, "Cryptographic hash functions: A review," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 2, pp. 461–479, 2012.

[21] P. Sriramya, "Providing password security by salted password hashing using bcrypt algorithm," *ARPN journal of engineering and applied sciences*, vol. 10, no. 13, pp. 5551–5556, 2015.

[22] jBCrypt. (u.d.) jBCrypt website. [Online]. Available: http://www.mindrot.org/projects/jBCrypt/

[23] R. Zuccherato, *Encyclopedia of Cryptography and Security.* Springer, 2011, pp. 62–63.

[24] JWT. (u.d.) JWT website. [Online]. Available: https://jwt.io/

[25] ——. (u.d.) Introduction to JWT. [Online]. Available: https://jwt.io/introduction/

[26] Stormpath. (u.d.) Java JWT: JSON Web Token for Java and Android. [Online]. Available: https://github.com/jwtk/jjwt

[27] J. Tidwell, *Designing interfaces.* Sebastopol, California, USA: O'Reilly Media, 2011.

[28] DigitalOcean. (2016) Initial Server Setup with Ubuntu 16.04. [Online]. Available: https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-16-04

[29] ngix. (u.d.) nginx. [Online]. Available: https://nginx.org/en/

[30] DigitalOcean. (2016) How To Install Nginx on Ubuntu 16.04. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-install-nginx-on-ubuntu-16-04

Bilaga A Dokumentation för REST API

I denna bilaga presenteras det REST API som utvecklats för MedImagers back-end. REST API-resurser som finns tillgängliga inkluderar resurser för inloggning, registrering, användarhantering, sökning och hantering av bilder samt kollektioner.

Add description to collection

Allows a user to add a description to a collection

- **URL**

  /collection/description

- **Method:**

  `PUT`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "collectionDescr":"Interesting development",
        "collectionID": "12",
}
```

Add image to collection

Allows a user to add an image to a collection

- **URL**

  /collection/{collectionid}

- **Method:**

  `POST`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "examinationID": "22000300",
        "index": "2",
}
```

Add note to image in collection

Allows a user to add a note to an image in a collection

- **URL**

  /collection/note

- **Method:**

  `PUT`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "note":"Interesting development",
        "collectionID": "12",
        "collectionitemID":"123"
}
```

Create Collection

Allows a user to create a collection

- **URL**

  /collection/

- **Method:**

  `POST`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "userID": "2",
        "collectionName": "Karies",
        "collectionDescr": "Bilder på karies",
}
```

Delete collection

Allows a user to delete a collection

- **URL**

  /collection/

- **Method:**

  `DELETE`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "collectionID": "12",
}
```

Delete image from collection

Allows a user to delete an image from a collection

- **URL**

  /collection/{collectionID}

- **Method:**

  `DELETE`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "collectionitemID": "123",
}
```

Export Collection

Allows a user to export a collection to a ZIP, which contains an MVD-database.

- **URL**

  /collection/export/:id

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:
    `File: .zip`

Get Collection

Allows a user to get a certain collection

- **URL**

  /collection/{id}

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
{
        "examinationID": "9231239012",
        "note": "",
        "collectionID": "12",
        "collectionitemID": "122",
}
{
        "collectionName": "9231556012",
        "collectionDescr": "Se tandrad",
        "collectionID": "12",
        "collectionitemID": "123",
}
```

Get Collections

Allows a user to get its collections

- **URL**

  /collection/

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
{
        "collectionName": "Karies",
        "collectionDescr": "Bilder på karies",
        "collectionID": "1",
        "userID": "12",
}
{
        "collectionName": "Bakterier",
        "collectionDescr": "Bilder på Bakterier",
        "collectionID": "2",
        "userID": "12",
}
```

Get Shared Collection

Allows a user to get a certain shared collection

- **URL**

  /collection/share/{id}

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
{
        "examinationID": "9231239012",
        "note": "",
        "collectionID": "12",
        "collectionitemID": "122",
}
{
        "collectionName": "9231556012",
        "collectionDescr": "Se tandrad",
        "collectionID": "12",
        "collectionitemID": "123",
}
```

Get Shared Collections

Allows a user to get its shared collections

- **URL**

  /collection/share

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
{
        "collectionName": "Karies",
        "collectionDescr": "Bilder på karies",
        "collectionID": "1",
        "userID": "12",
}
{
        "collectionName": "Bakterier",
        "collectionDescr": "Bilder på Bakterier",
        "collectionID": "2",
        "userID": "12",
}
```

Get User (Admin)

Allows an admin to access a specific user's info

- **URL**

  /admin/getuser/{userid}

- **Method:**

  `GET`

- **URL Params**

  Required:

  `userid = <integer>`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
{
        "firstName": "Börje",
        "id": "4",
        "lastName": "Plåt",
        "userPermission": "admin",
        "username": "börje@medimager.com"
}
```

- **Error Response:**

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ User ID not valid }`

    OR

  - **Code:** 403 FORBIDDEN
    Headers: None

Message Body: None
Explanation: Occurs when the user does not have admin permission

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token must be provided`
  Message Body: `{ Token must be provided }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token not valid`
  Message Body: `{ Token not valid }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token expired`
  Message Body: `{ Token expired }`

OR

- **Code:** 409 CONFLICT
  Headers: None
  Message Body: `{ User permission updated, please login again }`

Get User

Allows a user to get its user info

- **URL**

  /user/getuser

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
{
        "firstName": "Börje",
        "id": "4",
        "lastName": "Plåt",
        "userPermission": "admin",
        "username": "börje@medimager.com"
}
```

- **Error Response:**

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token must be provided`
    Message Body: `{ Token must be provided }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token not valid`
    Message Body: `{ Token not valid }`

    OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token expired`
  Message Body: `{ Token expired }`

  OR

- **Code:** 409 CONFLICT
  Headers: None
  Message Body: `{ User permission updated, please login again }`

Get Users

Allows an admin to access a all the users' info

- **URL**

  /admin/getusers

- **Method:**

  `GET`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Example
```

```
{
        "firstName": "Börje",
        "id": "4",
        "lastName": "Plåt",
        "userPermission": "admin",
        "username": "börje@medimager.com"
}
{
        "firstName": "Nisse",
        "id": "5",
        "lastName": "Hult",
        "userPermission": "normal",
        "username": "nisse@medimager.com"
}
```

- **Error Response:**

  - **Code:** 403 FORBIDDEN
    Headers: None
    Message Body: None
    Explanation: Occurs when the user does not have admin permission

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token must be provided`
  Message Body: `{ Token must be provided }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token not valid`
  Message Body: `{ Token not valid }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token expired`
  Message Body: `{ Token expired }`

OR

- **Code:** 409 CONFLICT
  Headers: None
  Message Body: `{ User permission updated, please login again }`

Login

Allows a user to login

- **URL**

  /user/login

- **Method:**

  `POST`

- **Header Params**

  Required:

  `Username: <username>`
  `Password: <password>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body: `{ <token> }`

- **Error Response:**

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Username not valid`
    Message Body: `{ Username not valid }`

  OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Password not valid`
    Message Body: `{ Password not valid }`

  OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Not yet approved by admin`
    Message Body: `{ Not yet approved by admin }`

Register

Allows a user to register

- **URL**

  /user/register

- **Method:**

  `POST`

- **Header Params**

  Required:

  ```
  Username: <username>
  Password: <password>
  FirstName: <firstname>
  LastName: <lastname>
  ```

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body: `{ Preliminary registration complete, awaiting approval }`

- **Error Response:**

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ Required user info missing }`

    OR

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ Account already registered }`

Remove User

Allows an admin to remove a user

- **URL**

  /admin/removeuser/{userid}

- **Method:**

  `DELETE`

- **URL Params**

  Required:

  `userid = <integer>`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body: `{ User removed }`

- **Error Response:**

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ User ID not valid }`

    OR

  - **Code:** 403 FORBIDDEN
    Headers: None
    Message Body: None
    Explanation: Occurs when the user does not have admin permission

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token must be provided`

Message Body: `{ Token must be provided }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token not valid`
  Message Body: `{ Token not valid }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token expired`
  Message Body: `{ Token expired }`

OR

- **Code:** 409 CONFLICT
  Headers: None
  Message Body: `{ User permission updated, please login again }`

Search: Available terms

Below is a table of available terms for the resource Search and how they are translated in a returned Examination JSON object

Term	Translation
Allergy	allergy
Biopsy-site	biopsySite
Diag-def	diagDef
Diag-hist	diagHist
Diag-tent	diagTent
Dis-now	disNow
Dis-past	disPast
Drug	drug
Factor-neg	factorNeg
Factor-pos	factorPos
Family	family
Gender	gender
Lesn-on	lesnOn
Lesn-site	lesnSite
Skin-pbl	skinPbl
Smoke	smoke
Snuff	snuff
Sympt-now	symptNow
Sympt-site	symptSite
Treat-type	treatType

Term	Translation
Vas-now	vasNow

Search

Allows a user to search for examinations filtered by terms and age bounds

- **URL**

  /search?{Term1=a}&{Term2=b}&...&{AgeLower=x}&{AgeUpper=y}

- **Method:**

  `GET`

- **URL Params**

  Refer to documentation about available terms for a list of possible terms and more

  ```
  Term_ = <string>
  AgeLower = <integer>
  AgeUpper = <integer>
  ```

  Example URL:

  ```
  /search?Dis-now=Psoriasisartrit&Snuff=2 dosor/vecka&AgeLower=60
  ```

- **Header Params**

  Required:

  ```
  Authorization: <token>
  ```

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body:

```
// Excerpt from a possible response to the example URL above
{
  // ...
  // Possibly more examinations above
  {
    {
    "age": [
      "82"
    ],
    "allergy": [
      "Nej"
    ],
    "biopsySite": [
      "122"
```

```
    ],
    "diagDef": [],
    "diagHist": [],
    "diagTent": [],
    "disNow": [
      "Psoriasisartrit"
    ],
    "disPast": [
      "Knölros"
    ],
    "drug": [
      "Mycostatin mixtur"
    ],
    "examinationID": "123456789",
    "factorNeg": [
      "Nej"
    ],
    "factorPos": [
      "Nej"
    ],
    "family": [
      "Nej"
    ],
    "gender": [
      "1"
    ],
    "imagePaths": [
      "Database.mvd/Pictures/G0396/g03964.jpg",
      "Database.mvd/Pictures/G0396/g03963.jpg",
      "Database.mvd/Pictures/G0396/g03962.jpg",
      "Database.mvd/Pictures/G0396/g03961.jpg"
    ],
    "lesnOn": [],
    "lesnSite": [],
    "skinPbl": [],
    "smoke": [
      "Inte dagligen"
    ],
    "snuff": [
      "Nej"
    ],
    "symptNow": [],
    "symptSite": [],
    "treatType": [
      "Expectans"
    ],
    "vasNow": [
      "7.0"
    ]
  },
  // Possibly more examinations below
  // ...
```

```
}
```

- **Error Response:**

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token must be provided`
    Message Body: `{ Token must be provided }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token not valid`
    Message Body: `{ Token not valid }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token expired`
    Message Body: `{ Token expired }`

    OR

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ User permission updated, please login again }`

Share collection to user

Allows a user to share a collection to another user

- **URL**

  /collection/share/{targetuserID}

- **Method:**

  `POST`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Data Params :**

  - **JSON:**

```
{
        "collectionID": "12",
}
```

Unregister

Allows a user to unregister itself

- **URL**

  /user/unregister

- **Method:**

  `DELETE`

- **Header Params**

  Required:

  `Authorization: <token>`

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body: `{ "User unregistered" }`

- **Error Response:**

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token must be provided`
    Message Body: `{ Token must be provided }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token not valid`
    Message Body: `{ Token not valid }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token expired`
    Message Body: `{ Token expired }`

    OR

  - **Code:** 409 CONFLICT
    Headers: None

Message Body: `{ User permission updated, please login again }`

Update Password

Allows a user to update its password

- **URL**

  /user/updatepassword

- **Method:**

  `PUT`

- **Header Params**

  Required:

  ```
  Authorization: <token>
  NewPassword: <password>
  ```

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body: `{ Password updated }`

- **Error Response:**

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ No new password provided }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token must be provided`
    Message Body: `{ Token must be provided }`

    OR

  - **Code:** 401 UNAUTHORIZED
    Headers: `WWW-Authenticate: Token not valid`
    Message Body: `{ Token not valid }`

    OR

  - **Code:** 401 UNAUTHORIZED

Headers: `WWW-Authenticate: Token expired`
Message Body: `{ Token expired }`

OR

- **Code:** 409 CONFLICT
  Headers: None
  Message Body: `{ User permission updated, please login again }`

Update User Permission

Allows an admin to update a user's permission

- **URL**

  /admin/updateuserpermission/{userid}

- **Method:**

  `PUT`

- **URL Params**

  Required:

  `userid = <integer>`

- **Header Params**

  Required:

  ```
  Authorization: <token>
  NewUserPermission: <newuserpermission>
  ```

- **Success Response:**

  - **Code:** 200
    Headers: None
    Message Body: `{ User permission updated }`

- **Error Response:**

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ User ID not valid }`

    OR

  - **Code:** 409 CONFLICT
    Headers: None
    Message Body: `{ No new user permission provided }`

    OR

  - **Code:** 403 FORBIDDEN
    Headers: None

Message Body: None
Explanation: Occurs when the user does not have admin permission

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token must be provided`
  Message Body: `{ Token must be provided }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token not valid`
  Message Body: `{ Token not valid }`

OR

- **Code:** 401 UNAUTHORIZED
  Headers: `WWW-Authenticate: Token expired`
  Message Body: `{ Token expired }`

OR

- **Code:** 409 CONFLICT
  Headers: None
  Message Body: `{ User permission updated, please login again }`