Bruse - en webbdatabas för filhantering
Projektrapport, TNM094

Grupp J
Erik Olsson
Ronja Grosz
Klas Eskilson
Daniel Rönnkvist
Therése Komstadius

6 juni 2015

Sammanfattning

Den här rapporten beskriver hur en webbdatabas för filhantering utvecklades i kursen Medietekniskt Kandidatprojekt, TNM094, på Linköpings universitet. Den presenterar arbetsprocessen och de tekniska lösningar som applicerats.

Syftet med projektet var att gruppen skulle följa ett agilt arbetssätt för att utveckla ett önskat system beställt av en kund med specifika krav. Produkten som kunden önskade beskrevs som en databas för att kunna lagra och strukturera personlig information som dokument, bilder och musik. Dessa filer skulle sedan taggas med ett eller flera nyckelord för att kunna söka reda på önskad information på ett effektivt sätt. Serversystemet utvecklades i ramverket Ruby on Rails och databasens klientsida utvecklades i ramverket Angularjs. Resultatet är ett system, Bruse, som innehar den prioriterade funktionalitet som kunden önskat.

Innehåll

Figurer

Tabeller

Typografiska konventioner

Följande typografiska konventioner används i denna rapport.

Utseende	**Förklaring**	**Exempel**
Kursiv text	Engelska termer	Systemets *models* används för...
`Fast teckenbredd`	Klasser och tekniska komponenter	Systemets *model* `Identity` används för...

Tabell 1: Typografiska konventioner i rapporten

Förkortning	**Förklaring**
API	Application Program Interface
CSS	Cascading Style Sheets
HTML	HyperText Markup Language
SDK	Software Development Kit

Tabell 2: Ofta användna förkortningar

Kapitel 1

Inledning

Eftersom mycket information idag finns i elektronisk form är det viktigt att kunna lagra denna data på ett enkelt vis. Ofta används flera olika lagringstjänster och användaren har dålig översyn på var alla filer ligger och vad de heter. Därför är det önskvärt att skapa ett system där en användare kan få åtkomst till olika typer av filer, från olika lagringstjänster på ett snabbt och smidigt sätt.

1.1 Bakgrund

Idén till detta projekt fick kunden av boken *Getting things done* [1]. I boken presenteras principer för att organisera pappersdokumet genom att kategorisera dem i tillstånd (klart, påbörjat, ej påbörjat) istället för indelning enligt ämne (fakturor, kontrakt och så vidare). Inspirerad av detta hade kunden som önskemål att enkelt kunna sortera digitala filer och skapa en egen struktur med hjälp av att ge varje fil en eller flera taggar. Med taggar åsyftades ett eller flera nyckelord som för användaren hade en koppling till filen. På detta sätt kunde användaren komma åt filerna med hjälp av en begränsande sökning istället för att behöva leta igenom en djup mappstruktur för att hitta önskat objekt.

Kravspecifikationen från kunden var

- en webbdatabas som kan hantera lokal fillagring
- en teknisk kunnig ska kunna installera systemet
- filer ska kunna lagras och sökas efter med hjälp av taggar
- ett enkelt GUI som framhäver sökningen
- filerna ska kunna öppnas i webbläsaren
- spara filer och länkar genom en drag och släppa-metod.

Krav med lägre prioritet var

- e-post ska kunna sparas
- systemet ska fungera på olika enheter
- välja mellan olika vyer för filvisning.

1.2 Syfte

Syftet med projektet är att gruppen ska utveckla en webbtjänst för lagring av filer som användaren ska kunna komma åt via ett sökfält. Sökningen begränsas genom att söka efter taggar eller metadata som den eftersökta filen innehåller. Tjänsten ska vara användarvänlig så att vem som helst ska kunna ha nytta av den. Det ska både gå att lagra lokala filer och filer från externa lagringstjänster. Poängen med att synkronisera externa lagringstjänster till webbtjänsten är att användaren kan hantera alla sina uppladdade filer på ett och samma ställe.

Förutom att utveckla en produkt åt kunden var syftet med projektet att tillämpa en agil utvecklingsmetodik. Agil utveckling innebär att utvecklarna arbetar efter att ständigt ha en fungerande produkt, så att kontinuerlig återkoppling med kund kan ske. Detta för att säkerställa kundens belåtenhet och för att utveckla ett flexibelt system som ständigt förblir relevant [2].

Denna rapport redogör för utvecklandet av detta projekt, dess framgångar och dess motgångar. Rapporten tar upp detaljer kring den tekniska implementationen, vad som verkställts och vad som skulle ha genomförts om mer tid funnits.

1.3 Frågeställning

Arbetet med systemet och denna rapport har kretsat kring ett par centrala frågor.

Hur ska en webbdatabas struktureras för att sökning efter sparade objekt ska kunna levereras enligt en användares förväntningar gällande hastighet och resultat för webbtjänster med liknande funktionalitet?

För ett databasfilsystem är sökning en central komponent. Användaren ska enkelt och utan att behöva vänta hitta det denne letar efter. Då en databas växer utgör antalet rader i den ett potentiellt problem för sökningar – många rader i tabellen skall sökas igenom utav servern. Frågan handlar alltså om hur systemet kan utvecklas för att tidigt lösa problemet gällande hastighet och resultat. Hur går det att säkerställa att användarens sökord snabbt genererar resultat, oavsett hur komplex sökfrågan är eller databasens omfattning? Och hur kan användarens söktext användas för att hitta filer utifrån exempelvis taggar och filtyper?

Hur går det att säkerställa att en användares information som lagras i en webbdatabas inte är tillgänglig för någon som inte är den specifika användaren eller har blivit auktoriserad av den specifika användaren?

Säkerhet är ett ständigt aktuellt ämne då webbtjänster diskuteras. Ämnet är dessutom känsligt då ett system ämnar att lagra en användares personliga och potentiellt känsliga uppgifter. Hur går det att säkerställa att ingen obehörig person får tillgång till användarens filer? Och hur kan systemet ge ett tryggt intryck?

Hur kan filer i en webbdatabas presenteras i en webbtjänst på ett sätt som gör de överskådliga, hanterbara och lättillgängliga för en användare, i en filstruktur utan kataloger eller annan hierarki?

Denna fråga handlar om hur databasbaserade filsystem på webben kan presenteras. Systemet bör klara av användare som har flera hundra sparade filer, både ur prestanda- och användarvänlighetsperspektiv. Hur mycket information kan presenteras utan att det blir överväldigande?

På vilket sätt kan en webbtjänsts användargränssnitt utformas för att demonstrera all funktionalitet ett system besitter och göra det intuitivt för en användare?

Systemet ska göra som användaren förväntar sig, trots att filstrukturen kan vara något användaren är

ovan vid. Hur går det att anpassa en webbtjänst till hur exempelvis filhanteringsprogrammet användaren är van vid beter sig?

1.4 Avgränsningar

Utgående från kundens krav riktar sig installation av systemet mot en användare som har tillräckligt stor teknisk kompetens för att kunna hantera ett UNIX-baserat operativsystem. Vidare utvecklas systemet också för en slutanvändare med viss erfarenhet av liknande system. Det är alltså inte anpassat för ovana datoranvändare.

Kapitel 2

Relaterat arbete

I projektet har metoder och ramverk för såväl arbete som mjukvara använts. Nedan kommer en förklaring på de största delarna som använts och implementerats.

2.1 Scrum

Scrum är ett ramverk som innefattar olika roller, aktiviteter och tekniker för att förenkla utvecklingsprocessen. Det finns nyckelroller inom teamet för att se till att alla delar ses över utan att lägga allt ansvar på en individ. Det finns även en rad förutbestämda möten och uppdateringar som är till för att ge hela gruppen bra överblick och en chans att påverka arbetet.

Inom Scrum delas hela utvecklingsperioden upp i tidsrutor, så kallade sprintar. Dessa är förutbestämda tidsperioder som strukturmässigt är uppbyggda på samma sätt varje omgång. De inleds med ett planeringsmöte där fokus och mål för förestående sprint bestäms. Inför varje ny arbetsdag hålls ett kort scrummöte där alla i teamet får presentera det de gjort sen senast och vad de ska göra kommande arbetsdag. Detta för att alla ska få en bra bild över hur arbetet ligger till tidsmässigt och om det finns några problem som måste lösas.

I slutet av varje sprint hålls två möten. En sprintgranskning och en sprintåterblick. Sprintgranskningen går ut på att alla i utvecklingsteamet sätter sig ned, eventuellt även med kund, och går igenom vad som åstadkommits under senaste sprint. Arbetet demonstreras och utvecklarna svarar på eventuella frågor, presenterar problem och hur dessa har lösts. Detta möte är till för att utvärdera de tekniska lösningar som applicerats. Sprintåterblicken är ett internt möte för utvecklingsteamet. Även här ska senaste arbetet utvärderas men ur ett socialt perspektiv istället för ett tekniskt. Här ska gruppens relationer diskuteras, hur användandet av valda verktyg fungerat och det finns även tillfälle för alla enskilda individer att utvärdera egen arbetsinsats.

Alla krav som finns för projektet sparas i en så kallad backlogg. Det är en rörlig lista där krav rangordnas utefter prioritet. När ett krav blir mer aktuellt eller mer väldefinierat flyttas det uppåt i listan. [3]

2.2 Mjukvaruramverk

Två stora mjukvaruramverk som använts i projektet är Ruby on Rails samt Angularjs.

2.2.1 Ruby och Ruby on Rails

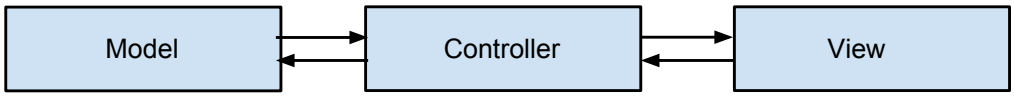
Ruby on Rails är ett ramverk skrivet i språket Ruby [4]. Ramverket är skrivet med öppen källkod som används utav flera stora tjänster på nätet, bland annat mikrobloggstjänsten Twitter, sammarbetsverktyget Github och boendeförmedlingstjänsten Airbnb. Ramverket är skrivet enligt designmönstret MVC. Detta står för *model*, *view*, *controller* och är ett sätt att strukturera ett projekts logik på sådant vis att olika komponenter har egna tydliga platser och syften.

I Ruby on Rails skapar utvecklaren *models*. Detta är en samling klasser vars syfte är att hantera data som användaren eller systemet interagerar med, och fungerar som ett lager ovanpå databasen. Ett exempel på en *model* kan vara en klass för användare eller filer. Samtliga databasoperationer sker genom systemets *models*. En *model* skrivs i Ruby med syntaxen *CamelCase* medans klasstabellerna i databasen är namngivna med *snail_case*.

Den komponent som sköter interaktionen mellan det som användaren interagerar med och systemets *models* är systemets *controllers*. Här finns logik för att hantera användares handlingar och hämta data från systemets *models*. Ett exempel kan vara att användaren klickar på en knapp i webbläsaren. Syftet med knappen är att visa en viss data. Användarens handling skickas till en *controller* som tar emot vad det är för data som användaren efterfrågar och hämtar den datan från en *model*. Här finns också logik för att hantera undantag, till exempel om datan saknas. Den *controller* som anropats skickar sedan resultatet av användarens begäran vidare till användaren.

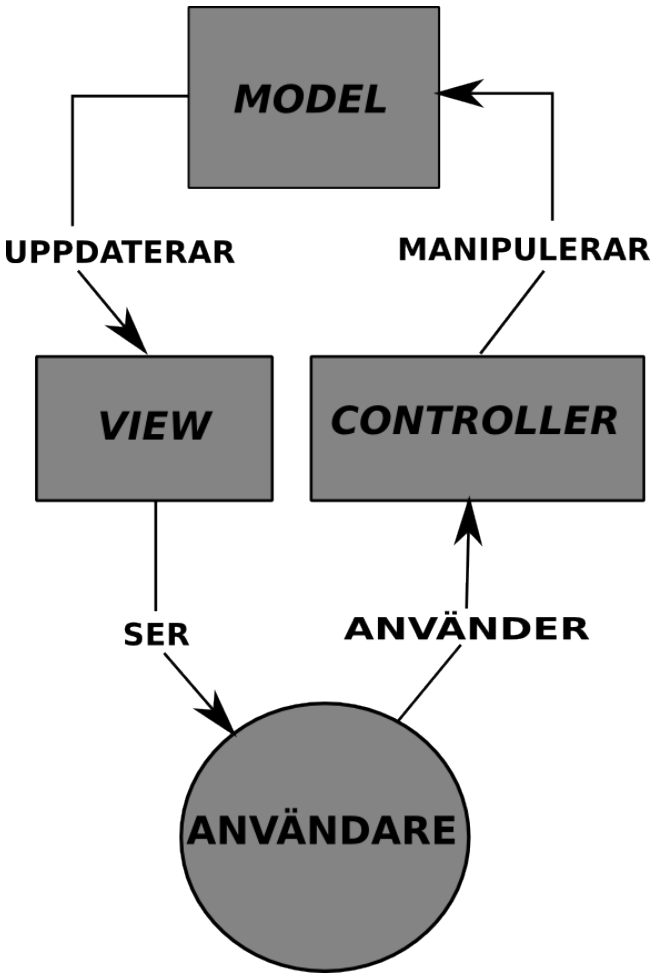
Det som användaren i sin tur interagerar med är systemets *views*. Här presenteras det som systemets *controllers* producerat. Knappen som användaren trycker på, i exemplet i stycket ovan, skapas i en *view*. Här kan det också finnas länkar, texter, textfält och alla andra komponenter som utgör det som renderas av en webbläsare.

En mer överskådlig figur för hur data och interaktioner generellt sett färdas genom ett MVC-system visas i figuren nedan. Data transporteras från *models* till *views* via *controllers*. Interaktioner färdas mellan *views* och *models* via *controllers* och direkt mellan *controllers* och *views* samt mellan *controllers* och *models*, se figur 2.1.



Figur 2.1: *MVC-systemets struktur.*

En annan figur som snarare fokuserar på MVC ur ett användarperspektiv visas i figur 2.2.



Figur 2.2: *MVC ur ett användarperspektiv.*

2.2.2 Angularjs

Angularjs är ett så kallat MVW-ramverk, vilket står för *Model-View-Whatever* [5]. För att effektivt använda detta ramverk bör även systemet byggas upp efter den designen. Då *whatever* innebär att det finns flera olika typer av sätt att kontrollera eller manipulera data på, kan varje komponent i systemet få specialanpassade lösningar.

I Angularjs finns så kallade *factories*. Detta är en samling hjälpfunktioner vars syfte är att undvika koduppprepning och göra det enkelt för koden att återanvändas på flera platser i systemet.

2.3 Övrig tredjepartsmjukvara

Då Ruby on Rails och Angularjs är mjukvara med öppen källkod har det bildats globala nätverk kring dessa ramverk. Många tillägg, verktyg och bibliotek har skrivits för att underlätta utvecklandet i dessa ramverk.

För att underlätta för utvecklare har Ruby on rails byggts med en mängd olika verktyg. Möjligheten finns även för utvecklare att bygga egna verktyg som kallas för *gems* och är ofta fria att använda och även de skrivna med öppen källkod.

För att fortsätta bidra till detta nätverk kring Ruby on Rails och Angularjs utvecklades även projektet med öppen källkod, för att kunna dela de lösningar som tagits fram.

2.4 Taggar och sökning

För att uppnå en effektiv sökstruktur i en databas finns det två olika metoder. Den första metoden kallas professionell indexering och är hierarkisk. Den andra metoden kallas folksonomi vilket bygger på att taggar sätts på innehållet utan någon särskild hierarki. Sammanfattat kan de beskrivas som "*Indexering strävar efter precision medan taggning strävar efter hantering*" [6].

Genom professionell indexering ramas den eftersökta informationen in med hjälp av huvudkategorier och underkategorier. Det går alltid att ta sig fram i databasens olika kataloger och till slut hitta den sökta filen. Folksonomi går ut på att hitta den eftersökta filen genom taggar. Nyckelorden behöver inte följa någon speciell hierarki eller struktur sinsemellan.

2.5 Gränssnitt

För att designa ett användavänligt gränssnitt kan Normans Principer[7] tillämpas för att åstadkomma en god användarupplevelse. Dessa utgår ifrån sex punkter: synlighet, återkoppling, begränsningar, mappning, konsekvens och affordans.

2.5.1 Synlighet

Det är viktigt att utforma en design som gör att användaren känner sig trygg i gränssnittet. Användaren ska helst kunna förutspå vad som händer när denne klickar på en ikon eller en rubrik. Ikoner ska tala för sig själv, till exempel en ikon med en soptunna betyder kasta och en penna betyder redigera. För att göra det lättare för användaren att hitta det mest relevanta på sidan kan tomma utrymmen användas [8]. Dessa tomma utrymmen är grafiska utrymmen som inte innehåller någon typ av information. De är till för att skapa mer "luft" i gränssnittet. Detta gör att användares fokus begränsas kring ett visst område och det blir lättare att navigera.

2.5.2 Återkoppling

Att ständigt ge återkoppling till vad som sker eller vad som har skett gör att användare känner sig tryggare i systemet. Dialogrutor, laddningssymboler och felmeddelande är exempel på viktig återkoppling [8]. Utan en laddningssymbol kan användaren få känslan av att ingenting händer och går därför kanske tillbaka och upprepar steget, vilket förstör hela processen.

2.5.3 Begränsningar

För att användaren inte ska göra oönskade saker i systemet eller ha för många alternativ, är det en bra idé att begränsa alternativen [8]. Till exempel förvirrar för många menyalternativ på en hemsida.

2.5.4 Mappning

Mappning innebär att liknande innehåll samlas på samma plats för att hålla en god struktur [8]. Användare ska snabbt kunna hitta filens tillhörande funktioner. Detta kan åstadkommas genom att visa alternativen i anslutning till filerna.

2.5.5 Konsekvens

För att göra det lättare för användaren att minnas hur denne använde en funktion är det bra att ha en liknade design på hela hemsidan. Det blir lätt rörigt för användren om designen inte följer ett mönster. Att använda sig av samma teckensnitt och färgschema hjälper till att hålla sidan konsekvent [8].

2.5.6 Affordans

Affordans betyder att ett objekt själv ska kunna beskriva vad det ska användas till. Till exempel är det tydligt att det går att hänga kläder på en klädkrok och ingen ytterligare beskrivning krävs [8]. Samma sak gäller på en hemsida. Användaren ska inte behöva fundera över vad som kommer hända när denne klickar på exempelvis en papperskorg, det vill säga något kommer att raderas.

2.6 Databaserat filsystem

Ett databasbaserat filsystem använder sökning för att hitta filer genom metadata eller nyckelord. De flesta filsystem använder kataloglagring. Ett examensarbete på University of Twente i Nederländerna hade som uppgift att undersöka om ett databasbaserat filsystem kan ersätta filsystem med kataloglagring med avseende på användbarhet och förmåga att lära sig att använda systemet [9]. Genom användartester drogs slutsatsen att ett databasbaserat filsystem presterade bättre utifrån dessa aspekter.

Kapitel 3

Webbdatabas för hantering av filer

I detta kapitel kommer de olika delar som utvecklingen av systemet bestod av att presenteras.

3.1 Inledande möten

Projektet inleddes med ett antal interna gruppmöten där projektets vision diskuterades utifrån projektets beskrivning. Diskussionen ledde till många frågor, dels kring tekniska bitar och dels kring kundens krav. På första kundmötet besvarades dessa frågor i den mån kunden kunde. Kunden hade inte alltid en bestämd åsikt om hur ett problem skulle lösas och lät därför utvecklarna hitta en lösning. Efter kundmötet hölls ett nytt gruppmöte där projektvisionen förtydligades och tekniska lösningar diskuterades och ritades upp.

I inledningen av projektet diskuterades det mycket om hur systemet skulle utvecklas. Faktorer som språk och ramverk togs upp. Beslutet som fattades var att språket Ruby och ramverket Ruby on Rails skulle användas som grundstomme i projektet. Vidare användes också Javascript och ramverket Angularjs för att göra upplevelsen av systemet snabbare för användaren.

När gruppen hade en tydlig idé om hur databasen skulle se ut och vilken teknik som skulle användas skrevs en projektplan (se bilaga []) som skulle ligga till grund för arbetsrutinerna, ansvarsfördelningen och tidsplanneringen (se bilaga []).

3.2 Hantering och strukturering av databaser

Systemet använder sig utav en SQL-databas, där SQL står för *Structured Query Language*. Det är ett programmeringsspråk för att lagra, bearbeta och hämta information i en databas [10].

3.2.1 Databasstruktur

Inledningvis togs en grundläggande databastruktur fram. Databasen innehöll användare, användares olika externa konton (till exempel Dropbox eller Google Drive), filer och taggar, se figur 3.1.

3.2.2 Databashantering

En stor komponent utav Ruby on Rails är modulen som heter `ActiveRecord`. Detta är en modul vars syfte är att förenkla databashantering [11] och skapades efter ett designmönster med sam-

ma namn [12]. Istället för att skriva databasfrågor manuellt kan en utvecklare använda systemets `ActiveRecord`-*models* för att förenkla arbetet. Rent praktiskt kan detta innebära att ersätta SQL-frågan "`SELECT * FROM table WHERE column1='value'`" med "`Table.where(column1: 'value')`".

Med hjälp av detta kunde komplexa relationer mellan olika tabeller i databasen förenklas, och en objektorienterad struktur med Ruby-klasser skapades utifrån tabellerna. En extern nyckel i en SQL-tabell kan i sin `ActiveRecord`-form liknas med en pekare till ett annat objekt. Detta ledde till att arbetet kunde skyndas på, och att avancerad kunskap om SQL-frågor inte var nödvändig för utvecklingen av sidan.

I och med att Rails `ActiveRecord`-modul hade inbyggt stöd för SQL-databaser användes dessa för systemets relationella tabeller. Mer specifikt användes SQL-databasen Postgresql i systemet. En fördel med Postgresql är dess indexering som bygger på bland annat B-trees [13]. (Redogörelse för indexeringsmetoder och datastrukturerna bakom dem går bortom räckvidden för denna rapport.) Denna indexeringsmetod möjliggjorde snabba sökningar baserat på olika parametrar (exempelvis namn eller etiketter), vilket ansågs viktigt för systemet.

3.3 Systemarkitektur

Systemet kan sägas vara uppdelat i två huvudsakliga delar. Dels finns det som körs på en server i Ruby-kod och dels finns det som körs hos klienten (webbläsaren) i Javascript.

3.3.1 Javascript och Angularjs

För att skapa ett responsivt system, i bemärkelsen att det var snabbt för användaren, valdes det att implementera mycket utav funktionaliteten hos klienten med hjälp av Javascript. Systemet innehåller många olika komponenter som ska interagera med varandra. Ett exempel är då flera filer ska listas med verktyg för att hantera filen, samtidigt som den filen har flera taggar som också ska kunna hanteras.

För att skriva Javascriptkod användes Coffeescript, vilket är ett språk som kompileras till Javascript. Coffeescript bidrar till en mer kompakt kod, har en syntax som är mer lik Rubys syntax och ämnar att underlätta vissa bitar av Javascript. [14]

Ramverket Angularjs användes för hanteringen av datan i gränssnittet. *Controllers*, *factories*[15], och en särskild fil för presentationen av filerna skapades. Funktioner implementerades på klientsidan för att få ett gränssnitt som uppdaterades snabbt när olika operationer utfördes. Operationer som sökning, sortering och visning av filer sköts med hjälp av Angularjs.

Systemets *factories* underlättade också för utvecklingen av vissa av systemets asynkrona komponenter. Asynkront innebär att koden inte nödvändigtvis väntar på att samtliga kodrader exekveras uppifrån och ned. Istället kan vissa funktioner ta den tid de behöver. Resultatet blev ett snabbt system, samtidigt som utvecklaren fick en utmaning att hantera detta på ett bra sätt.

3.4 Tredjepartsmjukvara

Några *gems* som använts under utvecklandet av detta system är:

Byebug, ett avlusningsverktyg som möjliggör för utvecklare att sätta brytpunkter i koden. När systemet kör och stöter på den rad där denna brytpunkt finns pausas systemet. I konsolen kunde sedan variabler granskas och utvecklarna kunde steg för steg följa vilken väg koden följde.

Letter opener, ett verktyg för att hantera e-post som skickas av systemet. Istället för att utvecklaren sätter upp en e-postserver som skickar e-postmeddelandena via den, öppnar Letter opener e-posten i webbläsaren. Detta gjorde att utvecklingen av systemets e-postrelaterade komponenter blev betydligt enklare.

För utvecklandet av Angularjs och annan Javascript användes Chrome Developer Tools och tillägget Angularjs Batarang. Dessa är verktyg som gör det lättare för utvecklare att följa med i vad som händer då de använder webbläsaren Google Chrome. Med funktioner som brytpunkter och möjligheten att bevaka variabler blir utvecklandet betydligt enklare.

3.5 Filhantering

För att hantera tjänstens filer implementerades tre olika fillagringssätt. En modulär filhantering skapades för att lätt kunna implementera fler sätt att lagra filer. Dropbox implementerades först eftersom det ansågs vara ett enkelt sätt att hantera användarens filer. När det väl implementerats upptäcktes det att Dropbox krävde en krypterad anslutning till systemet, vilket kostar pengar. På grund av detta infördes även Google Drive som lagringssätt. Efter ett möte med kunden flyttades fokus till att också implementera lokal fillagring.

3.5.1 Dropbox

Dropbox implementerades med hjälp av Dropbox SDK. SDK står för *Software Development Kit* vilket är en uppsättning av utvecklingsverktyg för mjukvaruutveckling mot specifika ramverk eller programpaket, i det här fallet till utveckling mot Dropbox tjänster. Dropbox SDK ger verktyg för att utveckla nya tjänster som använder sig av Dropbox olika funktioner. Webbtjänsten som utvecklades använder sig av ned- och uppladdning av filer till och från Dropbox. För att nå användarens filer måste en autentisering till Dropbox ske, vilket görs med hjälp av Omniauth, en *gem*.

Med hjälp av Dropbox SDK hämtas metadata för filerna i rot-mappen för att sedan visas för användaren. Om en mapp sedan klickas på hämtas metadatan för filerna i den mappen med hjälp av sökvägen. En fil laddas också ned med hjälp av dess sökväg. För att ladda upp filer till Dropbox krävs det att de läggs till i en existerande mapp eller i rot-mappen. För att hålla filerna som laddats upp till Dropbox samlade, skapas automatiskt en mapp med namnet “Bruse” där filerna lagras.

3.5.2 Google drive

Likt Dropbox användes Google Drive SDK för att kunna använda deras funktionalitet i systemet. För att nå användarens filer krävs en autentisering för att ansluta till användarens konto hos Google Drive. Det här görs med hjälp av två *gems*, Drive SDK samt Omniauth. Filhanteringen sker likt metoden för Dropbox, förutom att Drive inte strukturerar sina filer med hjälp av sökvägar. Varje fil har istället en parameter för *parent* och varje mapp har en parameter *child*, där eventuella filer i mappen lagras. Om en fil ligger i en mapp är mappen filens *parent* och filen är mappens *child*, se figur 3.2. För att identifiera det här förhållandet kommer alla filer som ligger i samma mapp ha mappens id i dess parent parameter. Liknande kommer mappens parameter child innehålla alla id tillhörande filer som ligger i mappen.

3.5.3 Lokal fillagring

Den lokala filhanteringen infördes för att ge användaren ett alternativ utan extern kostnad och utan filutrymmesbegränsningar från externa tjänster. För att sköta uppladdningen, som är en central del av den lokala filhanteringen i och med att det inte går att importera filer likt från Google Drive eller Dropbox, implementerades en så kallad drag och släpp-uppladdning samt en formulärsuppladdning. Drag och släpp-uppladdningen innebär att användaren kan markera en eller flera filer på sin dator, dra dem till webbläsarfönstret där systemet är öppet och släppa dem. Systemet tar där vid och laddar upp filerna. När servern har tagit emot all uppladdningsdata från webbläsaren får filen ett slumpat filnamn och placeras i en mapp på servern.

3.6 Gränssnitt

För att utveckla gränssnittet användes främst HTML, CSS och Javascript. För att skriva CSS användes verktyget SASS[1] (*Syntactically Awesome Style Sheets*), vars syfte är att förenkla avancerade stilstrukturer genom att bland annat tillåta loopar, nästlade stilregler samt funktioner i CSS. Javascript användes för att göra gränssnittet snabbare för användaren genom att placera viss logik för utseendet i användarens webbläsare. Detta ledde till att webbläsarens DOM (*Document Object Model*) kunde modifieras, och på så sätt snabbt ändra innehållet som presenteras. För detta användes Angularjs.

Sidans CSS kompletterades av ett par SASS- och CSS-ramverk för att underlätta utvecklandet. Bourbon[2] användes för att tillhandahålla smidiga hjälpfunktioner, så kallade *mixins* till SASS. För att kunna fylla tomma rutor i tjänsten användes kolumnsystemet Jeet[3]. För att nollställa webbläsares olika standardinställningar, exempelvis länkars färger eller typsnitt, användes Normalize.css[4]. Ikoner lades till för att göra knappar mer intuitiva vilket biblioteket Font Awesome[5] användes för. För att visa att systemet laddar hem data, eller skicka data, användes Nprogress [6]. Vidare användes knappar och textfält från Skeleton [7].

CSS-koden strukturerades upp i flera filer för att skapa en mer överskådlig struktur. Strukturen var uppdelad i följande kategorier: knappar, textfält, generell layout-struktur, huvudmeny, länkar, fillistor samt små individuella komponenter.

3.7 Testning

Sedan version 1.9 utav Ruby har standardverktyget för testning varit Minitest [16]. Minitest erbjuder många delar som behövs för att kunna hantera testning. Det som användes mest var så kallade *fixtures*, en exempelinstans utav en *model* i systemet. Dessa användes för att säkerställa att systemet producerade de förändringar mot modellen som var väntade. Men även att de *controllers* som fanns i systemet utförde rätt logik beroende på datan.

För att kontinuerligt säkerställa att projektet levde upp till de krav testerna specificerade användes tjänsten Travis CI, som testar att starta systemet och köra dess tester. Varje gång ny kod laddades upp

[1] http://sass-lang.com/
[2] http://bourbon.io
[3] http://jeet.gs
[4] http://necolas.github.io/normalize.css/
[5] http://fontawesome.io
[6] http://ricostacruz.com/nprogress/
[7] http://getskeleton.com

gavs då ett resultat om hur testerna gick. Med denna kontinuerliga uppdatering gick det att via Github följa projektets status utan att ladda ned och starta det.

För testning av gränssnittet krävs dock ett program som kan simulera en webbläsares funktion genom att rendera Javascript, HTML och CSS. Då Travis CI redan hade ett sådant program som heter Phantomjs installerat valdes den. För att integrera Phantomjs med Minitest användes Capybara. Capybara är ett testningsverktyg och finns som tillägg till Minitest. Capybara erbjuder inte bara stöd för rendering utav javascript med Phantomjs utan ger också hjälpfunktioner för att göra tester som simulerar användarbeteende i gränsnittet, något som täcker in stora delar av både servern och gränsnittet.

De tester som utvecklarna lade mest tid på att implementera var tester för *models*, *controllers* samt integrationstester. Integrationstester är tester som testar ett användarbeteende och har väldigt enkla instruktioner, till exempel gå till startsidan och fyll inloggningsformuläret med dessa uppgifter och tryck sedan på logga in. Med dessa tester gavs en helhetsblick och en väldigt stor del av systemet testades med ett och samma test.

Tester skrevs efter att en funktion hade implementerats för att säkerställa att den önskade funktionaliteten skulle hålla i framtida utveckling av andra funktioner.

3.8 Utvecklingsmetodik

Genom att använda Scrum som utvecklingsmetodik gavs en bra grund och tydliga riktlinjer att följa under projektets gång.

3.8.1 Roller

Utvecklingsteamet bestod av fem personer där varje person fick en nyckelroll:

3.8.1.1 Scrummästare

Dennes ansvar innefattade att se till att teamet höll sig till de riktlinjer som fanns, sköta kommunikationen runt de mer administrativa bitarna (boka sal, bestämma arbetsdagar och så vidare) och hålla i scrummöten.

3.8.1.2 Produktägare

Produktägarens ansvar var att hålla kontakten med kund och se till att kundens krav omvandlades till scenarion och uppgifter. Det var även dennes ansvar att hålla ordning i produktbackloggen.

3.8.1.3 Kodansvarig

Dennes ansvar gick ut på att hitta ett bra system för att integrera nya delar i det befintliga systemet och sedan se till att detta system upprätthölls.

3.8.1.4 Testansvarig

Dennes uppgift var att se till att systemet testades för att säkerställa att de krav som fanns uppnåddes och för att kontinuerligt jobba för att upptäcka brister i systemet.

3.8.1.5 Dokumentansvarig

Dokumentansvarig hade till uppgift att säkerställa att projektet blev ordentligt dokumenterat. Det var önskvärt att ha protokoll från kundmöten, gruppmöten med beslutsfattande karaktär och dokumentera skisser och liknande vid diskussioner kring systemets uppbyggnad.

3.8.2 Händelser

Hela utvecklingsperioden delades upp i mindre sprintar. Dessa sprintar hade en förutbestämd tid som inte kunde förlängas även om teamet upplevde att tiden inte räckte till. Varje sprint hade samma struktur. Vid uppstart hölls en sprintplanering där alla i teamet satte sig tillsammans för att gå igenom vilka scenarion från produktbackloggen som skulle ligga till grund i förestående tidsperiod. Det var viktigt att försöka hålla en god balans mellan tid och antalet valda scenarion. Teamet fick tillsammans utvärdera om de ansåg det möjligt att utföra alla scenarion innan sprintens avslut. När samtliga scenarier var valda bröts de ned i mindre delar och formades till specifika uppgifter. Det var önskvärt att hålla dessa uppgifter små och konkreta för att ha möjlighet att genomföra dessa under en arbetsdag.

Inför varje arbetsdag i en sprint höll teamet ett dagligt scrummöte. Detta var ett kort möte på max 15 minuter där alla i teamet stod upp, utan datorer. Varje gruppmedlem fick under detta möte berätta vad de gjorde senast och vad de skulle göra under dagen. Detta gav alla en bra inblick i arbetet och en kort avstämning på hur teamet låg till tidsmässigt.

Den sista arbetsdagen i varje sprint ägnades åt de två avslutande sprintmötena. Det första av dessa var en sprintgranskning. Detta var ett tillfälle då hela teamet kunde diskutera det senaste projektinkrementet och hur arbetet hade fortskridit. Det första som gjordes på dessa möten var att gå igenom de scenarion som fanns inför sprinten och om dessa blivit avslutade. Eftersom målet med varje sprint var att ha en fungerande produkt med de krav som scenarierna representerade, gav denna genomgång en tydlig bild om hur effektivt gruppen arbetat. Nästa del i mötet var att diskutera de problem som uppstått under sprintens gång och hur dessa blivit lösta. Den sista delen av dessa möten ägnades åt att diskutera den dåvarande backloggen, vad som skulle göras härnäst och vilka faktorer som spelade roll inför nästa produktinkrement.

Den sista aktiviteten i varje sprint var en sprintåterblick. Detta var ett möte där gruppen lade fokus på sina egna prestationer, de sociala relationerna i gruppen och de verktyg som använts. Detta var ett tillfälle där alla fick möjlighet att diskutera och påverka dessa aspekter för att jobba mot ett bättre arbetsklimat.

För att kunna skapa en realistisk bild över projektets storlek, vilka tidsramar projektet skulle delas upp i och hur mycket som skulle kunna åstadkommas under projektets gång skapades en tidsplan. För att ge en tydlig bild skapades ett Gantt-schema [2] som ett kalkylark i teamets gemensamma Google Drive. Där samlades alla dagar i projektperioden. I schemat kunde deadlines, tänkta kundmöten, sprintarna och lediga perioder åskådliggöras på ett smidigt sätt. (BILAGA [])

För att få en överblick över vilken sprint som påbörjats och vilka uppgifter som skulle slutföras användes webbtänsten Trello, se figur 3.3, som scrumboard[17]. Produktägaren skötte uppdateringen av pruduktbackloggen och sprintbackloggen i denna tjänst. Samtliga i gruppen hade fria händer att påbörja vilken uppgift som helst från sprintbackloggen. För att veta vem som påbörjat en uppgift märktes uppgiften med gruppmedlemmens namn. Övrig dokumentation som till exempel protokoll från sprintåterblick och sprintgranskning sköttes med hjälp av Google Drive. Även skisser på klassdiagram och programfunktionalitet sparades som bilder i Google Drive.

Här presenteras kort det fokus som fanns för varje sprint. Fullständiga sprintbackloggar går att se i (Bilaga []).

3.8.2.1 Sprint 1 - MVP

Målet med denna sprint var att ha en *Minimum Viable Product*, MVP[18] efter sprintavslutet. Något som var funktionellt och kunde visa på de problem som systemet skulle kunna lösa i ett senare skede. Fokus låg på att skapa en användare med Dropbox som fildatabas, importera filer från Dropbox till systemet och skapa nyckelord för filerna.

3.8.2.2 Sprint 2 - Utveckla den tekniska funktionaliteten

Fokus här var att göra filerna lättåtkomliga och sökbara. Dock uppkom det ett problem gällande Dropbox som fildatabas. En kostnad för ett SSL-certifikat (*Secure Sockets Layer*), som används för att visa att en anslutning är krypterad, gjorde att teamet fick ändra riktning efter samråd med kund. Utöver sökningen blev det även fokus på att skapa ett mer modulärt system för att kunna lägga till flera typer av tjänster. Det teamet kom överens om var att implementera Google Drive och även skapa en lokal databas för egen server.

3.8.2.3 Sprint 3 - Tänka på användaren

Under sprint tre hamnade fokus mer på systemets användare än den tekniska funktionaliteten. Ett flöde för systemets användande togs fram [BILD?], sökresultatens visning diskuterades [BILD?] samt utvecklades, och funktionalitet för drag och släpp-uppladdning påbörjades.

3.8.2.4 Sprint 4 - Färdig produkt

I den sista sprinten var det önskvärt att skapa ett användargränssnitt, få klart alla öppna scenarion och förbättra redan implementerad kod.

3.9 Versionshantering och kodgranskning

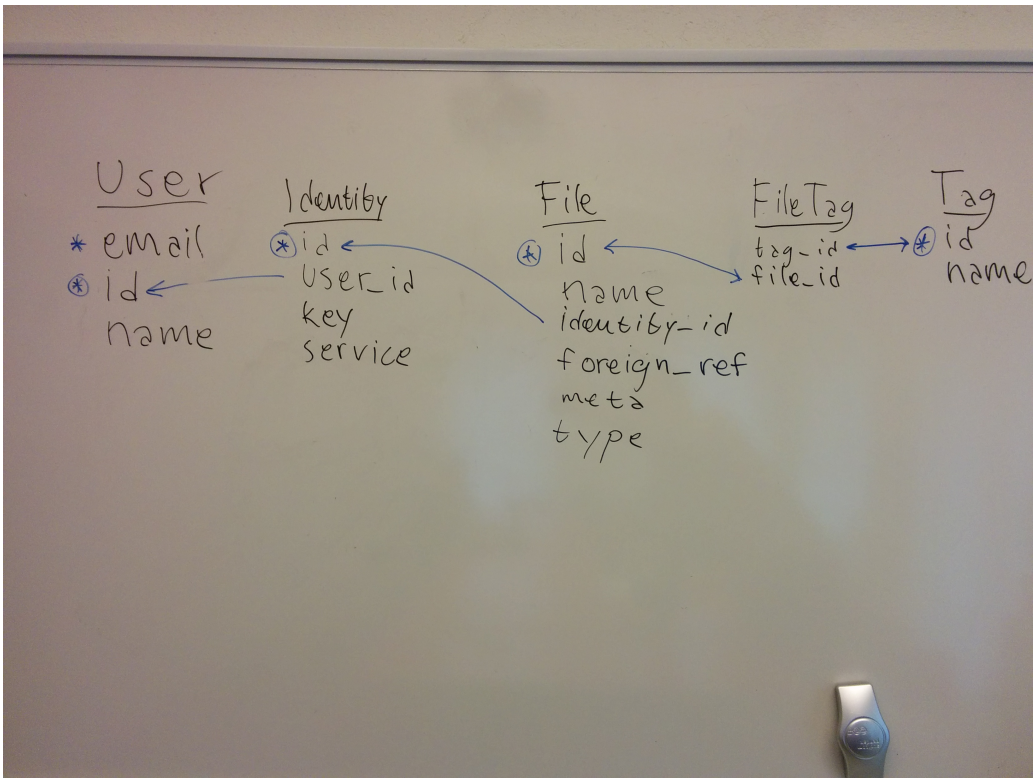
För versionshantering användes Git i kombination med Github genomgående i projektet. Git är ett distribuerat versionshanteringssystem där samtliga utvecklare har en komplett lokal kopia av hela projektet på dess egna datorer [19].

För att underlätta vid konflikter och andra problem som uppstår då parallell utveckling av närliggande delar av projektet skedde användes Gits förgreningsfunktion. Förgreningar är en metod för att lösa problem som uppstår vid en linjär utveckling, där allt måste ske i en viss ordning. Flera utvecklare kunde jobba på olika komponenter utifrån en gemensam grund, och Git organiserade sedan hur dessa kunde sammanfogas på enklast möjliga vis [19].

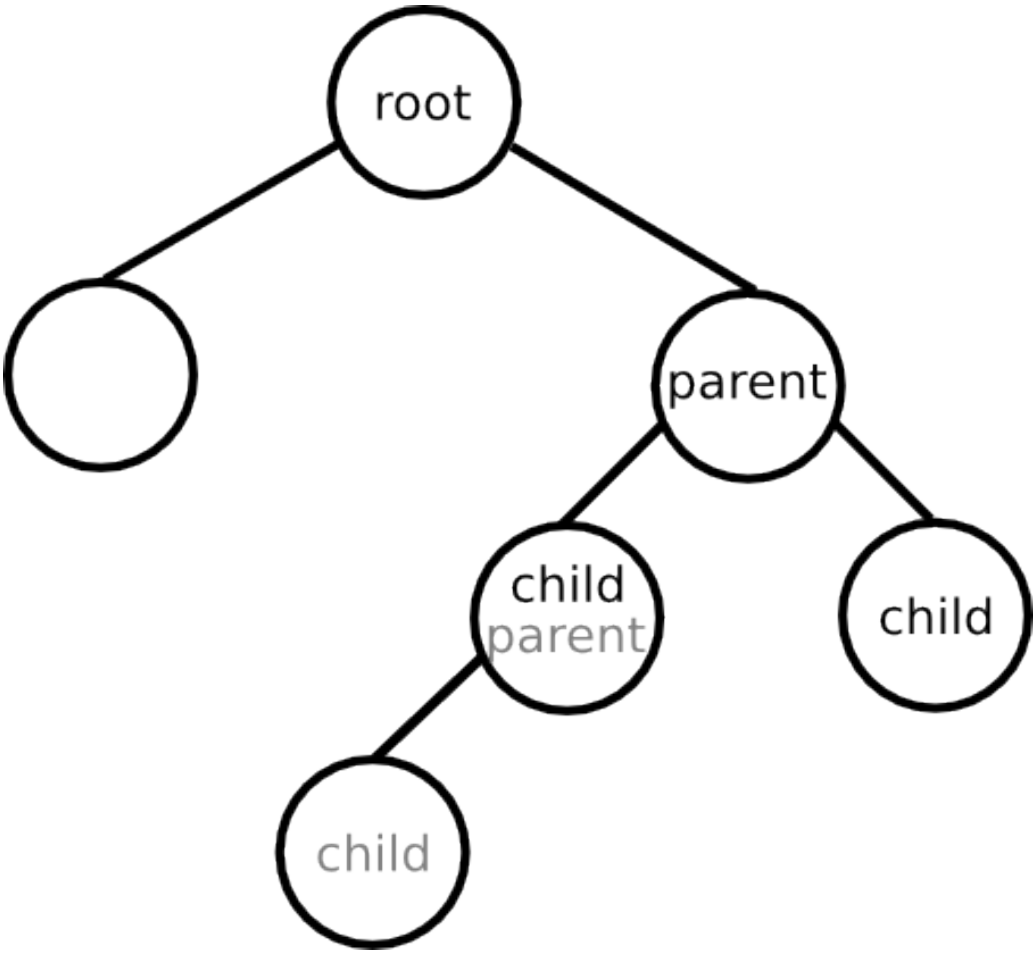
Inom förgreningen implementerades också en metod som populärt kallas *feature branches*, som ungefär kan översättas till funktionalitetsgrenar [20]. Då en ny funktionalitet skulle implementeras i systemet skapade den ansvarige utvecklaren en förgrening. När utvecklaren var klar med det aktuella arbetet, skapades en förfrågan på Github (en *pull request*) om att sammanfoga koden för den nya funktionaliteten med den befintliga. Innan koden sammanfogades fick den ansvarige utvecklaren återkoppling från resterande medlemmar i utvecklingsteamet om de blivande förändringarna och tilläggen. På så sätt skapades en trygghet för utvecklargruppen att det fanns en gemensam förståelse och konsensus kring det som skulle bli en del av den befintliga kodbasen.

3.10 Refaktorering

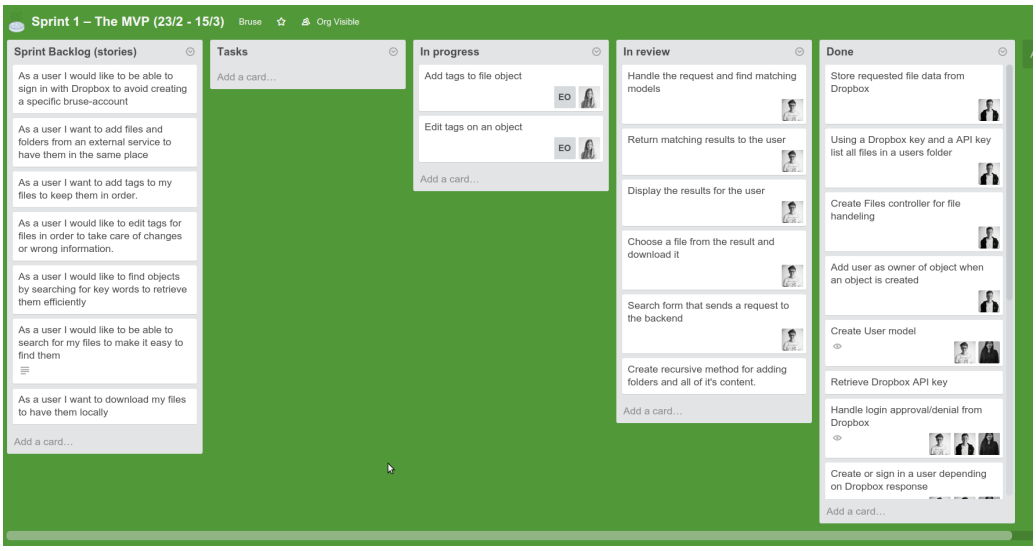
Refaktorering, det vill säga omstrukturering av kod, skedde med jämna mellanrum då till exempel en *controller* blev för stor eller innehöll för många olika funktionaliteter. Den delades då upp för att tydliggöra vad de olika delarna hanterar. Exempelvis delades *controllern* som hanterar användarnas filer upp i sex delar då den innehöll logik för olika delar av filhanteringen.



Figur 3.1: Visar strukturen för parametrarna *parent* och *child* i Google drive.



Figur 3.2: Visar strukturen för parametrarna *parent* och *child* i Google drive.



Figur 3.3: Scrumboard över sprint ett.

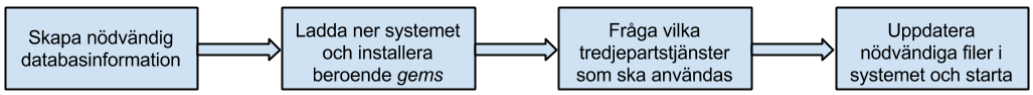
Kapitel 4

Resultat

Nedan presenteras det system som utvecklats.

4.1 Installation av systemet

Ett av kraven från kunden var att kunna installera systemet på sin egna server och låta det köra därifrån. För att underlätta för kunden vid installationen skapades en installationsfil som laddar ner systemets filer, fixar inställningar för databasen och även ställer frågor om vilka tredjepartstjänster som ska användas. Installationsfilen anpassar sedan systemet utefter de svar som fås av användaren vid installation. Stegen för installationen kan ses i figur 4.1.



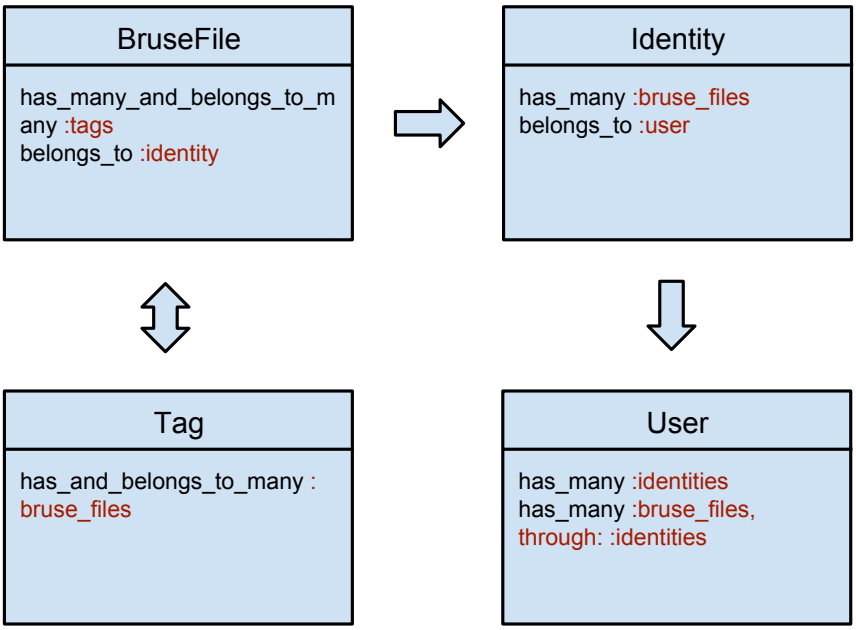
Figur 4.1: De steg som installationsfilen tar för att installera systemet.

4.2 Programdesign

Systemet är uppdelat i två huvudsakliga delar: server och klient. Javascriptklienten skickar och tar emot data från servern, som presenterar resultatet i JSON (*Javascript object notation*).

4.2.1 Ruby on Rails

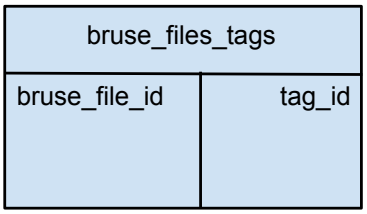
Följande *models* valdes att användas för systemet. Relationerna mellan dem är specificerade i figur 4.2. Modellen `BruseFile` fick det namnet eftersom `File` är ett reserverat ord i Ruby.



Figur 4.2: *Systemets* models *och dess relationer.*

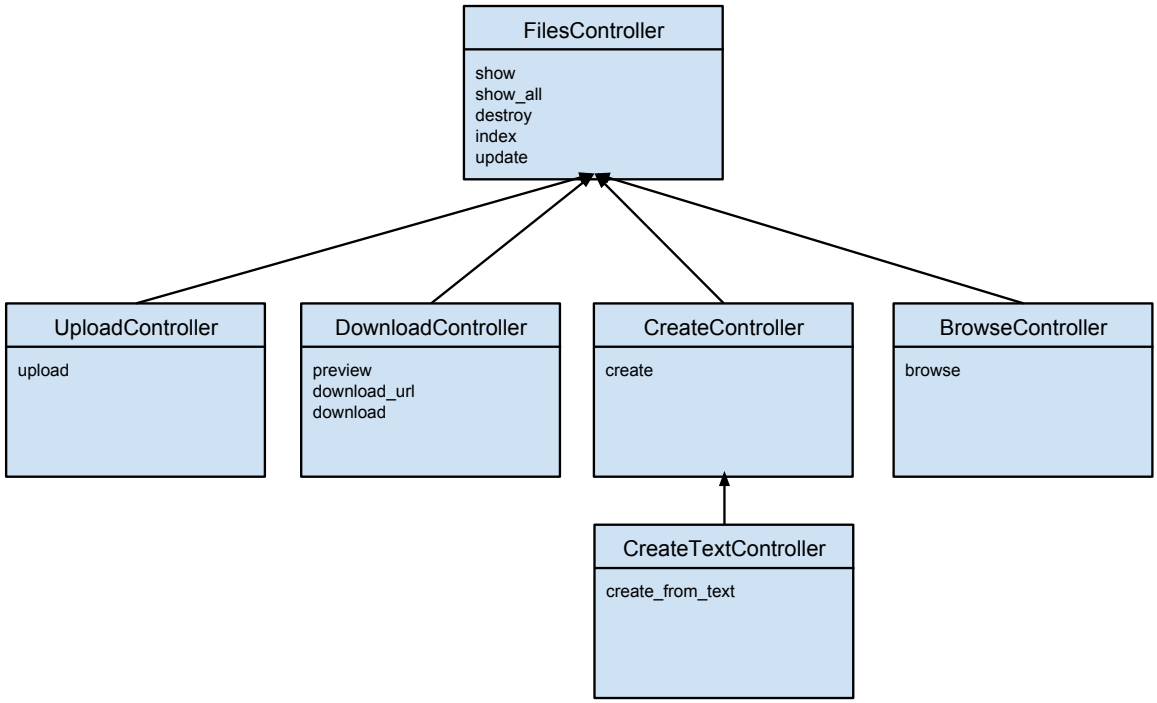
`Identity` är en av systemets viktigaste models. En `Identity` är en av tredjepartstjänsterna Google Drive eller Dropbox. Men även den egna filhanteringen hanteras som en `Identity` för att hålla det konsekvent och modulärt. Modellen är viktig för att kunna skilja på vilken tredjepartstjänst filen tillhör och kunna hantera den därefter.

För att kunna presentera en fil och dess taggar äger `BruseFile` flera taggar, men för att även kunna lista filer beroende på en tagg äger varje tagg flera `BruseFiles`. Rails löser relationen mellan dessa två genom att skapa en tabell som heter `bruse_files_tags` enligt figur 4.3.



Figur 4.3: *Relation mellan* `BruseFiles` *och* `Tags`.

Den *controller* som hanterade `BruseFiles` delades först upp i tre delar efter en första refaktorering för att sedan bli sex vid utökad funktionalitet och ytterligare refaktorering. Detta för att filhanteringen behandlar olika aspekter och behöver då delas upp för att skapa en tydlig översikt. I figur 4.4 beskrivs de olika *controllers* som används för `BruseFiles`.



Figur 4.4: *Struktur för filrelaterade* controllers.

4.2.1.1 `FilesController`

Denna *controller* är den som alla ärver av, som kan ses i figur 4.4. Den hanterar de vanligaste metoderna för en fil, de som handlar om att hitta eller redigera den information om filer som redan är sparad i databasen. Skillnaden mellan funktionerna *index* och *show_all* är att *index* visar alla filer för en identity vid import medans *show_all* visar alla filer för alla användarens identities.

4.2.1.2 `BrowseController`

Hanterar inläsning av mappar från en extern tjänst.

4.2.1.3 `CreateController`

Skapar ett nytt inlägg i databasen från den information som fås av klienten. Här finns även validering så att endast tillåtna variabler skickas vidare till att sparas i databasen. Valideringen av vilken typ variablerna är sker senare i systemets *model* `BruseFile`.

4.2.1.4 `CreateTextController`

Ärver av `CreateController`. Denna *controller* skapades då `create_text` kräver många fler hjälpfunktioner och valet gjordes att dela upp dem för att hålla det strukturerat. Många av de funktionerna handlar om att hantera textsträngen om den är en URL.

4.2.1.5 `DownloadController`

För att ladda ner filer på ett säkert sätt som kräver att rätt användare är inloggad skapades en *controller* för att säkerställa detta med ett antal funktioner. Funktionen `preview` fick även vara i denna då den

hanterar filer på ett liknande sätt som `download` gör. `download_url` skapades för att eventuellt implementera en delningsfunktion där flera användare skulle kunna få tillgång till filen.

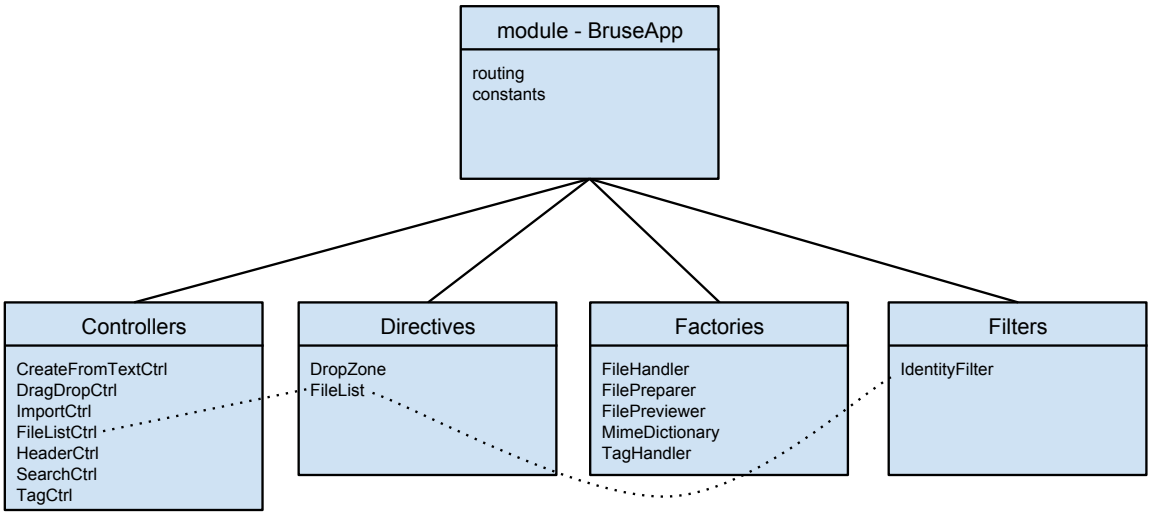
4.2.1.6 `UploadController`

Hanterar filer som ska laddas upp till de olika tjänsterna. Om filen laddas upp via drag och släpp-metoden eller via ett vanligt formulär tas filerna emot i olika format. Via drag och släpp-metoden fås filen kodad i *base64* vilket måste göras om till en så kallad `Tempfile` och sedan till ett objekt som är samma som det som fås via formulärsuppladdning [26].

4.2.2 Klient – Angularjs

Tillsammans bildar filerna olika komponenter som används för att hantera sina områden, se figur 4.5. De områden som styrs utav Angularjs är

- lägga till filer
- lägga till taggar till nyligen tillagda filer
- söka efter filer
- skapa filer från text
- drag och släpp-uppladdning
- lista filer.



Figur 4.5: *Struktur för Angularjs-komponenter.*

Vid import av filer sparas alla de valda filerna i en *array* i Javascript, sedan när användaren väljer att spara filerna skickas en förfrågan till serven per fil för att lägga till dem i databasen. Denna *array* sparas i en global variabel som Angularjs tillhandahåller. Användaren omdirigeras genom att systemet byter HTML-*template* och URL för att ge intrycket av att användaren kommer till en ny sida. Men då det är samma session och ingen ny förfrågan mot serven har gjorts av webbläsaren kan denna nya sida komma åt samma globala variabel och på så sätt använda sig av de nyligen tillagda filerna.

Denna lösning använder sig av ett verktyg för Angularjs som heter Angular Router. Detta verktyg kan hantera olika URL-förfrågor i Javascript och presentera *templates* och *controllers* som utvecklare

specificerat. Men då en förfrågan av en webbläsare sker först till serven måste även Angular Router-systemet där vidarebefodra vissa förfrågor till en speciell URL så att Javascript och Angular Router kan ta över istället för servern.

Angularjs *factories* användes för att: uppdatera och skapa filer, för att visa filer, för att uppdatera taggar, för att förbereda filer för visning samt för att hantera olika filtyper. Detta kan exempelvis innebära att `MimeDictionary.prettyType()` användes för att skriva ut en fils MIME- typ (*Multipurpose Internet Mail Extensions*) i ett mer läsbart format [27]. Exempelvis "excel spreadsheet" istället för "application/vnd.ms-excel" och "c++ file" istället för "text/x-c".

Factories användes även för att hantera asynkrona anrop mot servern. Detta gjorde att utvecklaren kunde veta när anropen var färdiga, till exempel när en fil hade uppdaterats och vad resultatet blev.

4.2.3 Sökning

För att låta en användare söka efter filer i systemet på ett enkelt och effektivt sätt delades denna funktion upp så att både servern och klienten användes för att hantera användarens önskemål. Hos klienten skrev användaren in sin söksträng som en kombination av ord, ord föregångna av ett nummertecken "#" och ord föregågna av en punkt ".". Denna söksträng delades upp efter respektive typ av ord, och skickades till servern. Servern tog då hand om denna samling som generella ord att söka efter, taggar att söka efter samt filtyper att söka efter. Då en användare skrev söktermen "hej #hälsning .txt" sökte alltså servern efter filer och taggar som innehöll ordet "hej" i sitt namn, hade taggen hälsning och var en textfil. Resultatet presenterades sedan för användaren utav klienten.

4.3 Tredjepartsmjukvara

Devise är ett tillägg till Ruby on Rails som ger ett paket för användarhantering, färdigt att användas. Då beslutet om att användare även skulle kunna logga in med e-postadress och lösenord insågs det samtidigt att en tredjepartsmjukvara skulle vara nödvändigt för att hålla tidsramen. Devise valdes för att det gav den mest kompletta lösningen med färdiga *views*, modifierbara *controllers* och enkel *model*-påbyggnad. Devise har också inbyggda funktioner som tillsammans med Rails e-postssystem skickar e- post till användare då de till exempel registrerar sig eller behöver hjälp med att återställa sitt lösenord.

Carrierwave ger utvecklarna verktyg för att hantera uppladdningen av filer. Genom att endast specificera inställningar för var filen ska sparas och så vidare fås en modul som kan användas i de *controller*-metoder som behövs.

Fuzzily skapar för varje ny instans av specificerade models så kallade *trigrams* över valda kolumner. *Trigrams* delar upp ett ord och grupperar det om tre tecken i taget [25]. För varje ny gruppering förflyttas den ett steg vilket gör att det skapas flera olika kombinationer. För ett ord skulle det resultera i flera olika bokstavskombinationer av grundordet. Exempelvis skulle ordet "bruse" med hjälp av trigrams bli `[b, br, bru, rus, use, se, e]`. Under implementationen av Fuzzily upptäcktes det att numeriska eller nordiska tecken inte sparades i trigrams. Tack vare att Fuzzily är skrivet i öppen källkod kunde problemet åtgärdas genom att ändra koden.

Delayed Job användes för att kunna skjuta upp skapandet av *trigrams*. Genom att spara kommande jobb i en lista kunde skapandet ske utan att blockera en förfrågan från användaren. Istället skedde detta i en ny process i bakgrunden.

Jstag är ett tillägg till Angularjs som konverterar användarens inskrivning utav taggar till visuella objekt för att tydliggöra för användaren hur dess inskrivning tolkas utav systemet. Även detta tillägg

hade brister då olika objekt kopierades utav systemet utan att samtliga tvåvägsbindningar behölls som kunde åtgärdas tack vare att det var skrivet med öppen källkod.

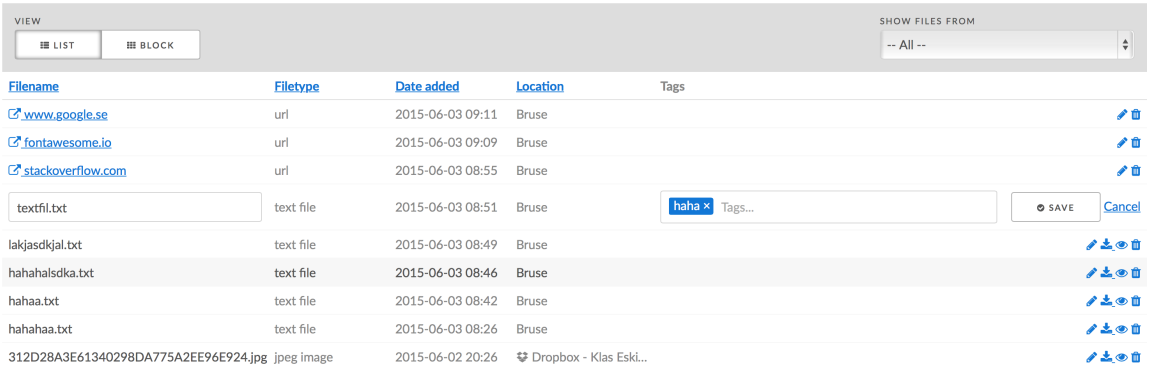
Magnific Popup gör att filerna som laddas upp till systemet går att förhandsgranska i samma vy som användaren befinner sig på. För att en viss filtyp ska kunna förhandsgranskas krävs det att man lägger till en definition för filtypen i hjälpfunktionen `MimeDictonary`. Magnific Popup använder samma teknik som webbläsaren gör för att förhandsgranska en fil vilket gör att filtyper som bilder, filmer, musik och textfiler går att öppna.

4.3.1 Filhantering

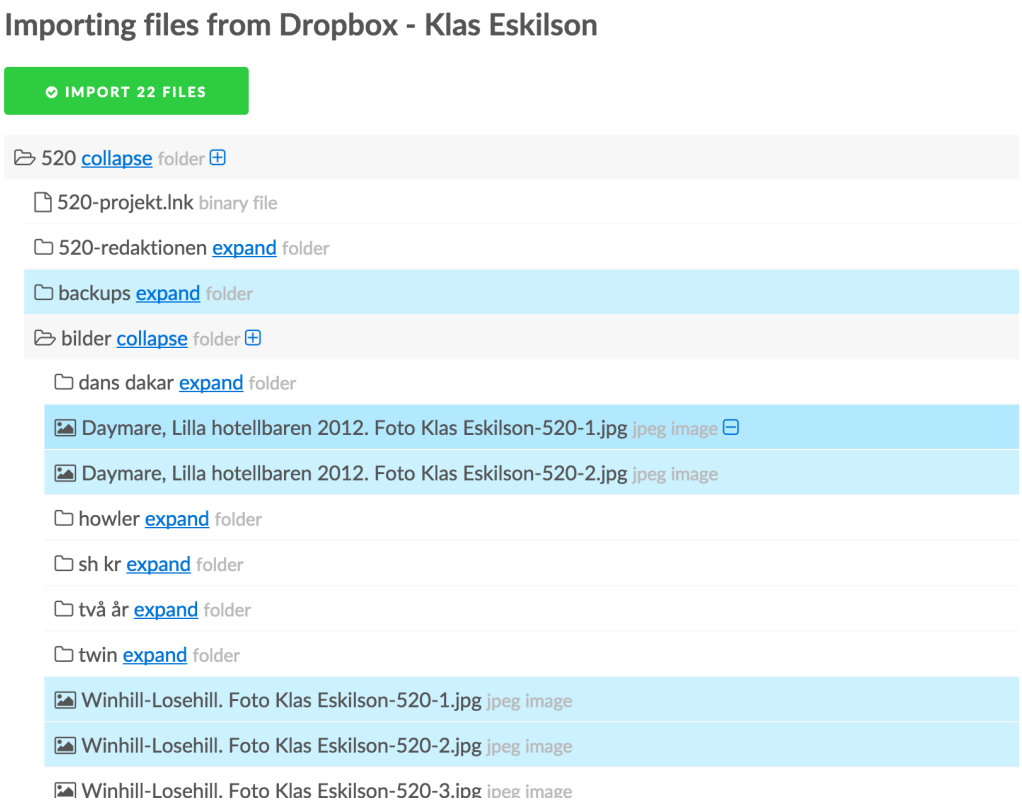
Då en fil lagrades i systemet var det två attribut som var centrala. Dels varifrån filen sparats – om den importerades från en extern tjänst som Google Drive eller Dropbox, eller om den lagrades i systemets lokala fillagring. Dels också en textsträng som motsvarar hur den aktuella tjänsten som filen tillhör separerar filen från övriga filer . Denna textsträng kallas i systemet för `foreign_ref`. För Google Drive är detta en slumpmässig bokstavs- och sifferkombination, för Dropbox är detta sökvägen till filen. För den lokala filhanteringen är detta namnet på filen efter att den sparats på servern, också detta en slumpad bokstavs- och sifferkombination.

4.3.2 Gränssnitt

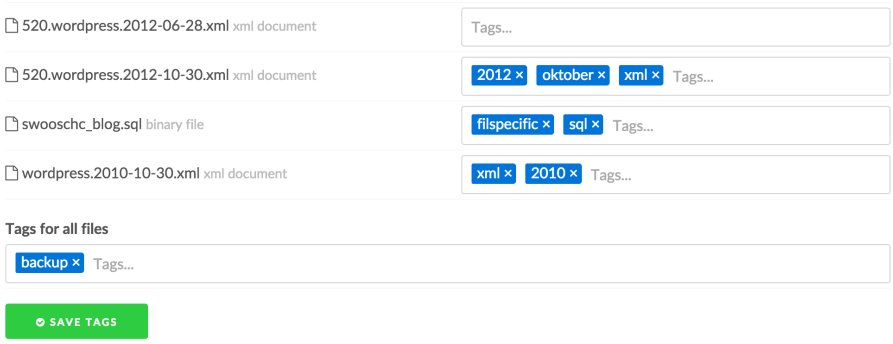
Systemet utvecklades för att ge användaren en enkel och tillgänglig upplevelse med ljusa färger och tydliga uppdelningar mellan systemets komponenter. Nedan visas skärmdumpar från systemets olika delar. I figur 4.6 till 4.10 visas skärmdumpar från systemet.



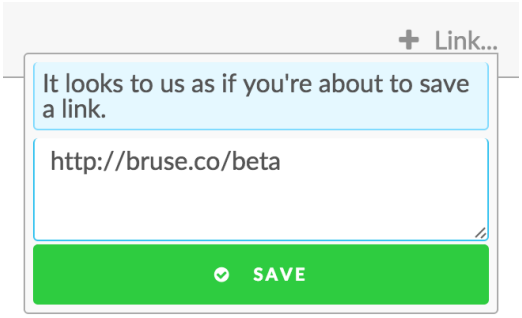
Figur 4.6: *Systemets lista över dess filer. En fil redigeras.*



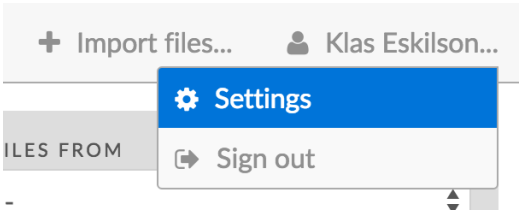
Figur 4.7: *Filimport.*



Figur 4.8: *Taggning av filer efter import.*



Figur 4.9: *Sparande av bokmärke via textfält.*



Figur 4.10: *Öppen undermeny.*

4.4 Utvecklingsmetodik

Arbetet följde Scrum som utvecklingsmetodik vilket har resulterat i en tydlig och strukturerad arbetsgång. Korta scrummöten i början av varje arbetsdag har förhindrat onödigt arbete samt att alla utvecklare är insatta i arbetet. Varje sprintgranskning och sprintåterblick har givit en tydlig bild om vad som gjorts och behöver göras för att nå nästa förbättring av produkten.

4.5 Versionshantering och kodgranskning

Användningen av Github gjorde att det alltid fanns en fungerande programversion i *develop*-förgreningen. Systemet växte i funktionalitet allt eftersom nya tillägg från *pull requests* godkändes. Om det inte fanns behov av kodning fanns det alltid behov av att granska *pull requests*. Antal *pull requests* och *commits* varierade från person till person. *Commits* från varje gruppmedlem och övrig statistik finns i bilaga [].

Kapitel 5

Analys och diskussion

Under projektets gång fattades många beslut och det stöttes även på en del problem. Grunden till dessa beslut och lösningar på problemen diskuteras i detta kapitel.

5.1 Metod

Här diskuteras utvecklingen av projektet både ur ett tekniskt perspektiv och ett metodikperspektiv.

5.1.1 Systemarkitektur

I projektets början diskuterades vilka programmeringsspråk och ramverk som kunde passa. De språk som ansågs vara lämpliga för systemet var Javascript, Google Dart, Phyton, PHP och Ruby. För- och nackdelar mellan de olika språken diskuterades. Slutledningen blev att PHP gick bort ganska snabbt på grund av att det ansågs inte vara det optimala språket då det ansågs onödigt svårarbetat samt att PHP är inte ett fullständigt objektorienterat språk till skillnad från Ruby on Rails [21]. Google Dart gick bort på grund av att gruppen inte hade någon tidigare erfarenhet av språket. Av de kvarstående språken var det Ruby med ramverket Ruby on Rails som var mest attraktivt eftersom det redan fanns kompetens om språket och ramverket i gruppen.

Beslutet om att använda Ruby on Rails hade både för och nackdelar. Nackdelarna var att språket har en hög inlärningströskel. Eftersom det är ett fullständigt objektorienterat språk kräver det att programmeraren förstår hur alla *controllers* och *models* fungerar. Som nybörjare var det svårt att förstå principen för designmönster MVC. Det var också svårt att förstå och börja använda Rubys syntax i början. Då Ruby är ett så kallat script-språk är det svårt att applicera kunskaper från högnivåspråk som C++.

En av fördelarna med att använda Ruby on Rails var att små förändringar i koden kan göra stora förändringar i systemet. Till exempel finns det många hjälpfunktioner man kan dra nytta av vilket gör att koden kan hållas kort och koncis. Det går också väldigt lätt att dela upp koden i olika filer vilket gör att varje fil aldrig innehåller särskilt många rader kod. Dock kan det ta lång tid att hitta rätt fil när hela projektet har delats in i många små filer om det inte finns en tydlig struktur och namngivning.

Ytterligare fördelar med att använda ett webbramverk finns och kan generaliseras med att säga att det är onödigt att uppfinna hjulet på nytt. Säkerhet, effektivitet och struktur är aspekter som förstärks utav många ramverk. En mer konkret fördel med Ruby on Rails är modulen `ActiveRecord` som alla modeller i systemet ärver av. `ActiveRecord` har funktioner för att hantera relationer och frågor mot databasen.

När Angularjs började användas höjdes svårighetsgraden ytterligare. Efter att gränssnittet började ta form gick det nästan inte att göra förändringar i systemet utan att använda Angularjs. Därför blev det en förutsättning att varje gruppmedlem hade en förståelse för ramverket eftersom samtliga utvecklare jobbade på hela systemet.

Fördelen med Angularjs är att detta ramverk hanterar datan som ska presenteras på ett mycket smidigt sätt [5]. Utan att använda ett ramverk som Angularjs hade det tagit mycket lång tid för webbläsaren att presentera datan och koden hade blivit väldigt komplicerad och ineffektiv. Nackdelen med Angularjs var den samma som Ruby on Rails, att det tar tid att förstå och är svårt att hantera i början.

En annan fördel med Angularjs är dess stöd för så kallade templates och för dess tvåvägsbindning för variabler [5]. Tvåvägsbindningen gjorde det enklare att utveckla komponenter som beror på hur användaren interagerar med textfält eller knappar. Till exempel krävdes det väldigt lite ansträngning för att uppdatera en rubrik till det som användaren skrev i ett textfält samtidigt som denne skrev, eller att uppdatera både textfältet och rubriken samtidigt.

5.1.2 Filhantering

I inledningen av projektet utvecklades systemet endast med tanken att det skulle vara ett lager ovanpå Dropbox. Då kundens krav förtydligades efter insikten att Dropbox kräver kostnadsbelagda servertekniker (krypterad och certifierad anslutning) fyra veckor in i projektet, var gruppen tvungna att göra en omvändning och prioritera om vad som skulle utföras och i vilket skede. Istället för att fokusera på enbart Dropbox lades då fokus på Google Drive. Detta visade sig dock otillräckligt enligt kund på grund av de begränsningar i lagringsutrymme Google Drive har. Det var först där efter som arbetet med den lokala filhanteringen inleddes. Det blev då tydligt att gruppen fokuserat på fel funktionalitet sett till kundens önskemål under flera veckor. Detta misstag kan ha berott på bristande kundkontakt och teknisk studie. Om en teknisk studie angående Dropbox och Google Drive hade påbörjats tidigare, hade dessa implementationer fått lägre prioritet medans lokal lagring hade fått högre prioritet. På så sätt hade projektets fokus blivit rätt från början.

Ett alternativ som övervägdes för att sköta filhanteringen var Mongodb. Detta är en dokumentdatabas som istället för att lagra tabeller med information lagrar dokument med identifierade nycklar som indexeras av databasen. Mongodb har dessutom stöd för att ha en kolumn med binär fildata och inte bara siffror eller strängar. Denna kolumn med fildata skulle alltså användas för att lagra filerna, snarare än att skriva dem i serverns filsystem. Detta valdes dock bort för att minska insteget till att utveckla systemet samt för att hålla nere antalet mjukvaror som systemet berodde av.

Skapandet av *trigrams* för filer skedde från början vid samma tillfälle som när en fil skapas i databasen. Detta gjorde så att varje förfrågan blockerades av systemet för att skapa dessa. När det var många filer som lades till samtidigt kunde väntetid bli påtaglig för användaren, för att lösa detta implementerades Delayed Job, en *gem*. Istället för att det skapas direkt i vid förfrågan sköts skapandet upp och skedde senare i en separat process. Skillnaden i tid gick från att en förfrågan tog 392 millisekunder till 55 millisekunder vilket ansågs vara en markant förbättring.

5.1.3 Taggning och sökning

För att uppnå en effektiv sökstruktur efter filer i en databas finns det två olika metoder: professionell indexering och folksonomi som tidigare nämnts i kapitel 2.4.

Fördelarna med professionell indexering är att den är mer precis jämfört med folksonomi eftersom det går att rama in den eftersökta informationen med huvudkategorier och underkategorier. Det går alltid att ta sig fram i databasens olika kataloger och till slut hitta den sökta filen. Folksonomi, till

skillnad från professionell indexering, baserar sin sökning på taggar och på grund av det är det inte säkert att nyckelorden leder rätt. Användare kan ha olika uppfattning om vad basnivån är på innehållet vilket gör att antingen onödiga taggar läggs till eller att det finns brist på taggar. Om till exempel en Javascript-fil ska taggas kan det vara onödigt för en användare att tagga filen med "webb" medan det är nödvändigt för en annan. Därför kan ett traditionellt klassificeringssystem ge ett bättre resultat eftersom det är helt opersonligt.

Eftersom systemet främst är till för den enskilde användarens strukturering av filer för att snabbt kunna finna dem igen är folksonomi att föredra. Ett kundkrav var dessutom att hierarkin skulle vara platt, vilket professionell indexering inte uppfyller.

Då det inte finns någon stavningskontroll när taggen skrivs in kan användaren använda slanguttryck och dialektala uttryck som kan precisera innehållet. Fördelen med att låta användaren stava taggarna precis som denne vill är att det blir lättare att skräddarsy taggarna för användarens egna ändamål. Det är också mindre kostsamt att använda sig av folksonomi till skillnad från professionell indexering. Professionell indexering kräver att det finns en noga genomtänkt katalogstruktur och regler för vart innehållet ska placeras. I ett taggningssystem krävs det inte alls lika mycket eftertanke från användaren gällande vilken katalog denne ska börja leta i. Det krävs bara att söka efter ett någorlunda passande nyckelord.

5.1.4 Hantering och strukturering av databaser

Systemet använder sig idag utav en relationell SQL-databas för dess data; användare, taggar samt filer lagras som rader i olika tabeller. Ett alternativ till relationella SQL-databaser av denna typ som övervägdes är så kallade NOSQL- databaser. Ofta står detta för *Not Only SQL* och är en samlingsterm för databaser som inte nödvändigtvis är tabellbaserade [24]. Exempel på typer som ingår i denna samling är dokument-baserade databaser (databasen lagrar indexerade filer) och nyckel-värde-lagringar (data lagras i listor med nyckelord som används för att komma åt dem). Gemensamt för denna samling databastyper är att de inte har en fast struktur och undviker relationer till andra databaser eller uppsättningar [22]. Dessa typer av databaser valdes tidigt bort på grund av dess icke-relationella struktur, samt på grund av deras mindre utvecklade stöd för användning i en Ruby on Rails-program.

5.1.5 Gränssnitt

Tanken med gränssnittet var att fokusera på filpresentationen. Eftersom alla filer presenteras i en och samma lista hade systemet blivit för långsamt och överväldigande om alla filer skulle visas samtidigt. Därför skapades en knapp för att ladda in fler filer successivt.

Överst i gränssnittet har en huvudmeny fixerats för att användaren snabbt ska kunna nå funktioner som utloggning, inställningar och filimportering. Det är funktioner som används ofta och bör vara lättillgängliga. Eftersom varje fil innehåller så pass mycket metadata som namn, datum, filtyp, `Identity` och taggar krävs det att fil-listan breder ut sig över hela skärmen för att kunna visa denna information. Ändringar av filer i systemet görs med ikonerna längst ut till höger i listan. I de flesta lagringstjänster kommer användaren åt dessa funktioner genom att högerklicka på filen. Genom att ha ikonerna synliga från början blir det färre klick för att utföra dessa operationer.

Figur 5.1: *Förhandsgranskning av fil.*

Förhandsgranskningen av filer sker genom att filen visas i webbläsaren med hjälp av en gem som heter Magnific Popup. Magnific Popup gör att filen som ska förhandsgranskas lägger sig som ett lager över allt annat. Förhandgranskaren gör det möjligt att bläddra fram och tillbaka mellan alla filer som lagts till och för att stänga ner förhandsgranskningen är det bara att klicka på det skuggade området eller på krysset, se figur 5.1. Den här typen av förhandsgranskning är mycket vanlig bland andra tjänster och är väldigt praktisk på grund av att piltangenterna går att använda för att bläddra mellan filerna och att det går snabbt att komma tillbaka till fillistan.

5.1.6 Testning

Under projektets början följdes planen att skriva tester efter varje funktionsimplementering, men vid förändrade förutsättningar i sprint två uppstod problem med testningen. Då användarhanteringen byggdes om med hjälp av tredjepartsbiblioteket Devise förändrades hur sessioner hanterades av systemet och följderna blev att de redan skrivna testerna misslyckades.

Det tog för lång tid att hitta en lösning för hur den nya sessionshanteringen skulle kunna testas. Därför valdes att nedprioritera testning. Tester togs bort och eftersom problemet aldrig löstes försvann testning från arbetsflödet och togs aldrig upp igen. *Build*-servern var kvar under utvecklingen som fortsatt kontroll att inget i systemet kraschar.

Testningen gav en bekräftelse vid kodgranskning att systemet och det nyligen utvecklade funktionerna inte förstörde något tidigare. Dock krävdes mycket tid för att få igång många system. Att testa javascript med hjälp av Phantomjs var viktigt då stora delar av systemet byggdes i Javascript men det tog tid att installera och implementera. Även *fixtures*, som tas upp i kapitel 3.7, tog tid att få igång på det sätt som gjorde att systemets funktionalitet gick att testa.

5.1.7 Utvecklingsmetodik

För att fungera som ett team är det en förutsättning att man fördelar ansvarsområdena så att alla får en roll i projektet. Enligt Scrum måste det finnas en scrummästare och en produktägare . Utöver dessa roller utsågs en testansvarig, dokumetansvarig och kodansvarig. scrummästaren hade lite svårt i början med att komma in i sin roll och se till att mötena genomfördes enligt scrumprinciperna men blev bättre efter återkoppling i samband med en sprintåterblick.

Det uppstod problem vid projektets början på grund av dålig kommunikation inom gruppen vilket gjorde att två personer började arbeta på samma funktion samtidigt utan att veta om det. Det hade räckt att bara en person arbetade med funktionen och låtit den andra arbeta med något annat för att vara effektiva. Detta problemet togs upp på ett möte och det bestämdes att gruppen skulle prata mer

med varandra och vara mer aktiva på Trello för att samma misstag inte skulle upprepas.

Arbetsrutinerna var att arbeta cirka tre hela dagar varje vecka, motsvarande 60% av en arbetsvecka. Hade någon inte möjlighet att närvara under bestämd tid fick denne ta igen arbetet vid något annat tillfälle. Mycket av tiden gick åt att sätta sig in i hur teknikerna som användes fungerade. Det ledde till att de lägre prioriterade kraven inte implementerades som till exempel e-posthantering och implementationen av Mongodb.

Fördelen med Scrum var att gruppen ständigt uppdaterade varandra om vad som var gjort och vad som skulle göras. Detta gav utvecklingsteamet en uppfattning om hur arbetet låg till tidsmässigt. Sprintåterblickar var speciellt bra då problem som uppstått både inom gruppen och kodmässigt togs upp och kunde åtgärdas.

Ibland var det svårt att tidsuppskatta en uppgift från ett scenario vilket ledde till att vissa scenarier aldrig hann slutföras innan sprintavslutningen. Helst ska alla scenarier i sprintbackloggen vara avklarade innan sprinten avslutas för att hålla sig till tidsplanen. Om så inte var fallet löstes det genom att skicka över ej avslutade scenario till nästa sprint. I sprint ett var det särskilt många scenarior som inte blev slutförda vilket berodde på att inlärningen för nya verktyg tog tid och att rutinerna för kodgranskningen gick långsamt i början.

Valet att avsluta sprint två tidigare än vad som var planerat visade sig vara ett bra beslut. Efter mötet med kunden fanns det ny funktionalitet som skulle implementeras vilket ledde till stora förändringar jämfört med vad som var planerat för denna sprint. Det krävdes diskussioner kring hur den lokala fillagringen skulle fungera, men till slut hade alla i gruppen en ganska klar bild hur detta skulle lösas vilket underlättade för nästa sprintplanering.

Under arbetet jobbade gruppen utan att ha några milstolpar för projektet. Det sattes inte upp mål för när vissa funktioner skulle vara klara annat än att varje scenario hörde till en sprint. Om gruppen hade satt upp tydligare mål för varje sprint och större, sprintöverskridande mål, hade det med stor sannolikhet varit lättare att hålla tidsmålen. Om detta även hade kombinerats med fler kundmöten hade projektet med stor sannolikhet också blivit mer framgångsrikt.

En förbättringpotential finns också för gruppen då de dagliga scrummötena inte utnyttjades fullt ut. Dessa mötens fulla potential finns i att det som sägs tas med under resten av dagen. Med detta menas att varje gruppmedlem måste tänka efter noggrant kring vad som faktiskt gått bra sedan det senaste mötet, vad som gått sämre och vad som skall göras den kommande dagen. Dessa möten blev rutinmässiga istället för eftertänksamma, vilket hämmade syftet med dem.

5.1.8 Versionshantering och kodgranskning

Github har varit till fördel för utvecklarna genom möjligheten till versionshantering och kodgranskning. Varje utvecklare kunde arbeta på en egen komponent utan att grunden ändrades. Det här arbetssättet medförde dock att det ibland blev en konflikt då två utvecklare arbetat inom samma komponent, men i två olika förgreningar. Detta löstes med att grenarna antingen slogs ihop innan de sammanfogades med grunden, eller att ena grenen sammanfogades med grunden först.

Arbete genom förgrening kan också bli problematisk då en gren bygger på en annan. Detta gav mestadels problem för kodgranskningen, eftersom det var svårt att sätta sig in i den påbyggda grenen utan att ha granskat grenen den bygger på. Detta löstes genom att teamet avvaktade med den senare grenen.

Ett annat problem har varit när en gren har blivit för stor och därmed blivit svårgranskad på grund av dess omfattning. Det här har utvecklingsteamet försökt undvika genom att hålla förändringarna små samt kontinuerligt och ofta sammanfoga varje förgrening med grunden.

5.1.9 Källkritik

Mycket av informationen angående programmeringen har hämtats från olika webbsidor. Både Ruby on Rails och Angularjs har väldokumenterade kodexempel på sina hemsidor som pedagogiskt förklarar grundprinciperna. Det fanns även dokumentation för alla *gems* som användes, dock med varierande kvalitet. Denna dokumentation fanns dels tillgänglig via Github och dels från diverse internetguider och artiklar. Trots att det fanns bra dokumentation var det inte alltid tillräckligt med hjälp. Det behövdes ofta göras eftersökningar på fler kodexempel för att kunna använda till exempel en *gem* eller Google Drives API (Application Programming Interface). Kodexempel och diskussioner som inte kommer direkt ifrån bibliotekens utgivare bör användas mer kritiskt. I vissa fall var information från forum där vem som helst kunnat föreslå lösningar den enda hjälpen som gick att få tag på.

Inledningsvis var internetguider i form av filmer till stor nytta, för att öka förståelsen för Ruby on Rails. Ett exempel på en sida som användes var Railscast [1] som var till hjälp eftersom sidan har ett brett filmarkiv som behandlar både triviala och avancerade implementationer i Ruby on Rails. Problemet med internetguider är att programmeringsspråken uppdateras relativt ofta vilket gör att guiderna blir föråldrade och ibland odugliga trots att de kan behandla rätt ämne.

5.2 Resultat

Nedan diskuteras hur väl det färdiga systemet levde upp till de förväntningar som inledningsvis fanns på det. Vad kunde gjorts annorlunda vad hade detta kunnat leda till?

5.2.1 Filhantering

Som nämnts tidigare hade gruppen inledningsvis fel fokus vad gäller filhantering. Exempelvis kunde mer tid ha lagts på att utveckla den lokala fillagringen ytterligare. Istället för att skriva direkt till samma server som sköter systemets logik kunde en annan tjänst använts om mer tid hade funnits. Några exempel på detta är Mongodb, vilket tas upp i kapitel 3.5.3, eller exempelvis Amazons Simple Storage Service, som är en tjänst för att lagra stora volymer data i molnet.

Vidare hade systemet kunnat göras effektivare genom att minimera det lagringsutrymme som krävs utav tjänsten. En metod för detta är att se till att en fil aldrig lagras av systemet mer än en gång. Säg till exempel att flera användare lagrar en textfil som innehåller texten “Hej”. I dagsläget lagrar systemet en kopia av denna fil för varje användare, istället för att lagra en gemensam referens till filen för alla. Genom att generera så kallade hashar för samtliga filer, textsträngar som beror på innehållet i en fil och skapas med hjälp av matematiska operationer, kan filers likhet konstateras [28]. Detta skulle kunna användas för att se om användaren laddar upp en fil som redan har en identisk like i systemet.

En annan aspekt som kunde utvecklats ytterligare är säkerheten kring den lokala fillagringen. Filerna är idag inte publikt åtkomliga, endast systemet kan läsa och skriva till dem och de finns inte att komma åt publikt via internet. Vidare döps filerna om till en slumpad kombination av siffror och bokstäver, vilket gör dem svårare att identifiera vid ett eventuellt intrång. För att veta vem en fil tillhör behövs således både databas och fil med andra ord behövs fler pusselbitar behövs för att få hela bilden. Om någon obehörig person skulle få tag på en fil fanns det dock inget som hindrade denne person från att öppna den.

[1] railscasts.com

5.2.2 Tredjepartsverktyg och bibliotek

Användning av *gems* hjälpte till att öka funktionaliteten i systemet. Gems till exempel som Devise sparade mycket programmeringsarbete till skillnad från om funktionaliteten hade skrivits från grunden. Även om det fanns svårigheten med att hantera vissa tillägg under utvecklandet ansågs det att den tid som lades på att förstå sig på tillägget var väl värd. Detta eftersom det var en relativt liten ändring, jämfört med tilläggets storlek och totala funktion, som behövdes åtgärdas eller justeras.

5.2.3 Krav

Kundens alla krav uppfylldes inte då de prioriterades olika och därmed hann inte vissa krav bli verklighet. Ett av dessa krav var att kunna spara e-post och att tagga dessa.

5.3 Arbetet i ett vidare sammanhang

Här diskuteras hur det färdiga systemet ställer sig i relation till omvärlden. Vilka etiska krav ställer systemet på användare och utvecklare?

5.3.1 Fildelning

I dagens elektroniska samhälle är illegal fildelningen ett problem, upphovsrättsskyddat material delas utan tillstånd från upphovsrättsinnehavaren. Samtidigt är det svårt att kontrollera illegal fildelning utan att inkräkta på användarens integritet. Därför är det upp till användaren att inte använda webbtjänsten till att dela upphovsrättsskyddade filer. Liknande tjänster som Dropbox eller Google Drive har större resurser för att motverka att användaren använder tjänsten på otillåtna sätt jämfört med vad detta system har [29].

5.3.1.1 Filhantering

Det är viktigt att ha en tjänst som hanterar filer på ett säkert sätt så att obehöriga inte kan få tillgång till dem. Därför är det viktigt som utvecklare att se till att systemet inte har några säkerhetsbuggar som kan leda till att känslig information kan nås från obehöriga användare.

5.3.1.2 Olagliga filer

Ansvaret för vad som laddas upp ligger enbart hos användaren. I Sverige är det olagligt att publicera information som bryter mot: hets mot folkgrupp, uppvigling, barnpornografi samt olaga våldsskildring [23]. Om det upptäcks att en användare sprider den här typen av information är det ett lagbrott vilket kommer leda till att polisen får utreda ärendet.

5.3.1.3 Personlig integritet

Då tjänsten är gjord för att kunna spara vilka filer som helst har ägaren utav servern där systemet körs en skyldighet att låta den informationen som sparas i databasen vara fortsatt hemlig för utomstående

eller andra användare utav systemet. Koder för att komma åt användares Google Drive- och Dropbox-konton sparas i databasen. Även om dessa är temporära och databasen är skyddad med lösenord kan alltid ägaren av servern se innehållet i databasen.

Kapitel 6

Slutsatser

Här diskuteras vilka slutsatser som kan dras kring frågeställningarna för systemet. Vilka lärdomar teamet drog och vad dessa innebar.

6.1 Frågeställningar

Hur ska en webbdatabas struktureras för att sökning efter sparade objekt ska kunna levereras enligt en användares förväntningar om hastighet och resultat?

Genom att dela upp taggar för sig och filer för sig och skapa en relation mellan de två som förklaras i kapitel 4.2.1 kan användaren specificera om den vill söka på taggar, filer eller båda två. Om det specificeras vilken typ som eftersöks kommer systemet att kunna leverera ett snabbare resultat.

Hur går det att säkerställa att en användares information som lagras i en webbdatabas inte är tillgänglig för någon som inte är den specifika användaren eller har blivit auktoriserad av den specifika användaren?

Genom att arbeta med en säker sessionshantering på servern som gör att en användare inte kan ändra en serverförfrågan för att modifiera data som utbyts. För att ingen ska kunna få ut någon annan användares information måste systemet försäkras om att rätt användare är inloggad och inte vara beroende på vilket id som skickas med i en serverförfrågan. På så sätt kan endast de inlägg i databasen som är kopplade till rätt användare säkert presenteras.

Hur kan filer i en webbdatabas presenteras i en webbtjänst på ett sätt som gör de överskådliga, hanterbara och lättillgängliga för en användare, i en filstruktur utan kataloger eller annan hierarki?

Filer i en webbdatabas kan presenteras genom en databasbaserad filstruktur i en webbtjänst. En databasbaserad filstruktur innebär att filerna visas vartefter de söks efter med hjälp av taggar eller metadata. På så sätt kan filerna lätt hittas, så länge användaren vet på ett ungefär vad den letar efter. Filer som stämmer överens med alla sökord genom en lös sökning visas. Användaren kan också redigera filens taggar till att passa sökningen bättre.

På vilket sätt kan en webbapplikations användargränssnitt designas för att demonstrera all funktionalitet ett system besitter och göra det intuitivt för en användare?

Det är viktigt att hålla en konsekvent design för att användaren ska känna sig trygg i systemet. Funktioner som hanterar gränssnittets data ska vara lätta att hitta för att underlätta användarens användning av systemet.

6.2 Framtida arbete

Den framtida visionen för systemet är att fler användare ska kunna interagera med varandra genom att dela filer, projekt eller exempelvis *mood boards* med varandra. Detta kan implementeras genom att användaren kan redigera synligheten för filer, så att de kan visas för andra genom att till exempel skicka en länk som ger läs- och nedladdningsrättigheter. Att kunna göra *mood boards* måste också utvecklas, vilket kan göras med hjälp av delningsbara taggar särskild vy.

För att öka säkerheten och kunna lagra filer på ett effektivare sätt kan den framtida databashanteringen ske med hjälp av Mongodb. Detta kan göras genom att byta ut all befintlig databaslogik till att ske med hjälp av Mongodb.

Fillagringstjänsten Box kan implementeras som ett alternativ till de externa tjänsterna Google Drive och Dropbox. Detta skulle förverkligas genom att utöka redan befintlig logik för hantering av filer från externa tjänster.

I dagsläget har systemet ett API för att låta utvecklaren skapa ett snabbt och välfungerande gränsnitt som kan skicka och ta emot data från servern utan att ladda om hela sidan. Denna del av systemet har utvecklats under hela projektets gång utan någon gemensam strategi eller struktur. I vidare utveckling av systemet bör detta göras om, så sökvägar (så kallade *endpoints*) för systemets API följer en gemensam konvention och att såväl server- som klientkod kan struktureras på ett mer logiskt sätt.

Litteraturförteckning

[1] D. Allen, *Getting Things Done: The Art of Stress-Free Productivity*, Penguin Books 2001

[2] S. Lawrence Pfleeger och J. M. Atlee, *Software Engineering, Fourth Edition, International Edition*, Pearson 2010

[3] K. Schwaber, J. Sutherland, *Scrumguiden*, 2013-07, hämtad: 2015-05-13
http://www.scrumguides.org

[4] D. Flanagan, Y. Matsumoto, *The ruby programming language*, O'Reilly Media, Inc. 2008

[5] R. Branas, *AngularJS Essentials*, Packt Publishing Ltd. 2014

[6] J. Granström, *Social taggning*, Högskolan i Borås 2007, hämtad: 2015-04-08
http://bada.hb.se/bitstream/2320/2178/1/07-56.pdf

[7] C. Kroner Grogarn, K. Olin, M. Sun Bursjö, *Hur långt når Norman?*, Göteborgs universitet 2011, hämtad: 2015-05-14
https://gupea.ub.gu.se/bitstream/2077/26683/1/gupea_2077_26683_1.pdf

[8] K. Lahtinen, *Skapandet av en modern webbdesign*, Arcada 2014, hämtad: 2015-05-14
http://www.theseus.fi/handle/10024/73920

[9] O. Gorter, *Database File System: An Alternative to Hierarchy Based File Systems*, University of Twente 2004, hämtad: 2015-05-14
https://www.sphinux.org/misc/docs/references/papers/dbfs.pdf (hämtad 2015-04-27)

[10] Encyclopædia Britannica. Encyclopædia Britannica Online. Encyclopædia Britannica Inc., 2015. Web. 12 maj. 2015 <http://global.britannica.com/EBchecked/topic/569684/SQL>.

[11] E. Płuciennik-Psota, *Object relational interfaces survey*. Studia Informatica, 2012.

[12] C. Pytel, Yurek, J., och Marshall, K.. *Pro Active Record: Databases with Ruby and Rails*. Apress, 2007.

[13] P. M. Aoki, *Implementation of extended indexes in POSTGRES.* SIGIR Forum, vol. 25, no. 1, ACM, 1991.

[14] J. Ashkenas, *A Little Book on CoffeeScript*, 2012

[15] *Angularjs.org*, hämtad 2015-06-04
https://docs.angularjs.org/guide/providers

[16] *Minitest in Ruby*, hämtad 2015-05-14.
http://svn.ruby-lang.org/repos/ruby/tags/v1_9_1_0/NEWS

[17] L. Pries-Heje, *Why Scrum Works*, 2011, hämtad 2015-06-04 http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6005502&tag=1

[18] *Minimal Viable Product*, hämtad 2015-06-04 http://www.techopedia.com/definition/27809/minimum-viable-product-mvp

[19] S. Chacon och B. Straub, *Pro Git*, 2nd ed., 2014

[20] Preißel, R., Stachmann, B., *Git: Distributed Version Control–Fundamentals and Workflows*, dpunkt.verlag, 2014.

[21] Dailey Paulson, L., *Developers Shift to Dynamic Programming Languages*, Computer 40.2, 2007.

[22] Lith, A., Mattsson, J., *Investigating storage solutions for large data*, Chalmers University of Technology, 2010.

[23] Polisen, *Lagar och fakta* Hämtad: 2015-05-15 https://polisen.se/Lagar-och-regler/Om-olika-brott/It-relaterade-brott/Diskussionsforum/

[24] Cattell, R., *Scalable SQL and NoSQL data stores*. ACM SIGMOD Record, 39(4), 2011.

[25] Gencosman, BC., Ozmutlu, HC., *Character n-gram application for automatic new topic identification*, Information Processing and Management vol. 50, 2014.

[26] Josefsson, S. *The base16, base32, and base64 data encodings.* Internet Engineering Task Force, 2006. Hämtad 2015-06-05. http://tools.ietf.org/html/rfc4648/

[27] Freed, N., Borenstein, N. *Multipurpose Internet Mail Extensions (MIME) Part Two*. Internet Engineering Task Force, 1996. Hämtad 2015-06-05. https://tools.ietf.org/html/rfc2046

[28] Weiss, A. M. *Data Structures & Algorithm Analysis in C++* Pearson, 2013.

[29] Kumparak, G. *How Dropbox Knows When Youlere Sharing Copyrighted Stuff (Without Actually Looking At Your Stuff)*, TechCrunch, hämtad 2015-06-04. http://techcrunch.com/2014/03/30/how-dropbox-knows-when-youre-sharing-copyrighted-stuff-without-actually-looking-at-your-stuff/

Bilaga A

Involverade i projektet

Bilaga B

Projektplan

Projektplan, TNM094

Team Bruse
Daniel Rönnkvist
Klas Eskilson
Ronja Grosz
Erik Olsson
Therése Komstadius

4 mars 2015

Sammanfattning

Detta är en plan för utvecklingen av en webb-baserad databas för hantering av filer på *Dropbox*.

Innehåll

Kapitel 1

Kort beskrivning av projektet och dess mål

Projektet går ut på att skapa en multifunktionell webb-databas som är tillgänglig för flera olika användare. Det ska vara möjligt för användaren att installera tjänsten lokalt på sin egen dator. Målet med webb-databasen är att hantera olika typer av filer och kunna hitta dem på ett lätt och smidigt sätt med hjälp av nyckelord. Dessa nyckelord kommer vara sökningsbara och med hjälp av dessa ska olika typer av filer kunna visas upp enkelt och sorterade efter olika parametrar, exempelvis nyckelord, filtyp, namn eller liknande.

1.1 Frågeställningar

- Hur ska en webbdatabas struktureras för att sökning efter sparade objekt ska kunna levereras enligt en användares förväntnignar om hastighet och resultat?
- Hur går det att säkerställa att en användares information som lagras i en webbdatabas inte är tillgänglig för någon som inte är den specifika användaren eller har blivit auktoriserad av den specifika användaren?
- Hur kan filer i en webbdatabas presenteras i en webbapplikation på ett sätt som gör de överskådliga, hanterbara och lättilgängliga för en användare, i en filstruktur utan kataloger eller annan hierarki?
- På vilket sätt kan en webbapplikations användargränssnitt designas för att demonstrera all funktionalitet ett system besitter och göra det intuitivt för en användare?

Kapitel 2

Tidsplan

För att snabbt få en överblick över projektet och se distribution gällande till exempel resurser och arbetsfördelning är *Gantt*-scheman ett bra verktyg. Varje dag under projektets gång specificeras här, och samtliga händelser under projektets gång markeras i schemat. Projektets Gantt schema finns att se i bilaga A.

2.1 Tid för planering, efterforskningar och tekniska studier

En övergripande efterforskning görs under projektuppstarten. Under projektets gång kommer sedan ytterligare tekniska studier ske inför och under varje *sprint*, för att ta reda på hur vissa krav ska realiseras.

2.2 Sprints och deras möten

En sprint pågår i tre veckor. I början på varje sprint kommer det hållas ett planeringsmöte på sex timmar inför kommande sprint. Där bestämmer gruppen tillsammans vilka scenarion som ska genomföras inför kommande sprint. Ett sprintmål kommer att formas efter dessa scenarier som gruppen kommer jobba mot att nå. Det är önskvärt att ha en fungerande produkt efter varje sprint. De scenarion som är intressant flyttas sedan över från projektets produkt-backlogg till aktuell sprint-backlogg tillsammans med de uppgifter som finns för att utföra arbetet.

Varje arbetsdag kommer ett 15 minuter långt dagligt Scrum-möte att hållas. De kommer att vara stående möten utan datorer för att hålla dem fokuserade och effektiva. Där diskuteras vad gruppmedlemmarna gjorde förra arbetsdagen och vad de ska göra den kommande dagen för att hjälpa till att nå målet för aktuell sprint. Där utvärderas också möjligheten att slutföra de krav som finns i nuvarande sprint backlogg.

Som avslutning i varje sprint kommer två möten att hållas. Först hålls en sprintgranskning, som är tidsbestämd till tre timmar, där det arbete som precis genomförts utvärderas. Här är det även bra att ha med kunden som kan få en demonstration av systemet som hittills utvecklats. Här redovisas även de problem som kan ha stötts på och hur de har blivit lösta. Mötet ger också en möjlighet att uppdatera projektets produkt-backlogg om nya krav eller förändringar i existerande krav dykt upp.

Till sist hålls en sprintåterblick som är ett möte på två timmar. Här går gruppen igenom den sprint som genomförts utifrån sina egna arbetsinsatser, verktyg, processer och kommunikationen i gruppen. Det framförs konstruktiv kritik och görs upp en plan för hur arbetet kan förbättras ur denna aspekt.

2.3 Milstolpar och leverabler

Efter varje sprint skall produkten vara ett inkrement på produkten från föregående sprint. Detta innebär att en förbättrad version av produkten skall finnas redo för leverans efter varje sprint.

Efter den första sprinten är målet ha en så kallad *minimal viable product*. Användaren ska kunna skapa ett konto, eller logga in på ett befintligt, och lägga till filer från dennes Dropbox-mapp samt ge filerna nyckelord. Andra sprinten kommer till stor del handla om att göra filerna lättåtkomliga och sökbara för användaren. Tredje sprinten kommer att fokusera på att förbättra användagränssnittet för att gör det mer attraktivt för användaren. Användartester ska vara genomförda innan starten för sprint fyra. Resterande sprintar kommer att användas för att utveckla lägre prioriterade önskemål som delning av filer.

2.4 Avstämningsmöten med ledningsgruppen

Bör hållas ett par gånger i samband med att en sprint avslutas. Tanken är att detta kan hållas efter sprintgranskning och sprintåterblick, så att dessa möten kan hålla fokus på det som är viktigt för utvecklingsprocessen.

Kapitel 3

Infrastruktur för programmering och systemutveckling

3.1 Utvecklingsplattform

Alla i projektet utvecklar i en *UNIX*-baserad miljö. Detta valdes för att majoriteten hade vana i denna miljö och för att förenkla samarbetet.

3.2 Kravhantering

De kravspecifikationer som finns gällande systemets funktionalitet från en användarsynpunkt kommer från kunden. Här är det önskvärt att kraven blir tydligt definierade och att alla menar samma sak. Det ligger i detta skede inget fokus på hur kraven ska uppnås. Dessa krav diskuteras sedan av utvecklingsteamet för att ta fram kravspecifikationer, alltså de krav som krävs för att uppnå kundens krav. Alla kravdefinitoner sparas som scenarion i projektets produkt-backlogg. Dessa bryts ner i mindre kravspecifikationer i form av uppgifter. Det ska vara möjligt att genomföra ett scenario över en sprint och en uppgift på en arbetsdag. Det är viktigt att det finns mycket detaljer för att arbetet ska bli rätt från början. Det är önskvärt att ha ett möte med kunden efter varje sprint för att visa upp arbetet så långt i projektet och se om några krav har förändrats.

3.3 Scenario- och uppgiftshantering

Varje krav kommer att förvandlas till ett scenario som kommer sparas ner i backloggen. Dessa scenarion kommer sedan att brytas ner till mindre uppgifter. För att få en bra översikt på hur backloggen och uppgifter hanteras kommer webbtjänsten *Trello* att användas.

3.4 Versionshanteringssystem och rutiner

För att hantera projektets kod kommer *Git* användas tillsammans med *Github*. Git är ett etablerat verktyg för versionshantering av kod. För att lätt kunna dela koden lagras den på Github. Versionshantering sker genom förgreningar från huvud- och utvecklingsgrenen där all utveckling sker i grenarna. När utvecklingen i någon förgrening är klar ska koden granskas av minst två andra personer innan den

sammanfogas med huvudgrenen. Koden får endast sammanfogas med huvudgrenen av en person för att minska risken för att trasig kod.

3.5 Testningsprinciper och rutiner

Inför varje sprint kommer den testansvarige att säkerställa att det finns tester som kan möta sprintens krav. Testen kommer att skrivas för att upptäcka fel i den kod som skrivs för kraven, även för att upptäcka fel som skapas. Detta kommer att ske löpande under varje sprint efter att en funktion har implementerats.

I projektets början kommer enhetstester att implementeras. Senare i projektet när ett användargränssnitt har byggts kommer integrationstester att tillämpas genom användartester. Användartesterna utförs genom att filma en testperson och skärmaktiviteten samtidigt medan systemet manövreras.

3.6 Dokumentationsprinciper och rutiner

All dokumentation i form av textdokument och filer samlas i *Google drive*. Vid varje möte förs ett kort mötesprotokoll och alla tavelanteckningar sparas som bilder. Backlogg samt sprints dokumenteras på Trello som kommer agera som Scrumtavla. Backloggen beskrivs genom olika fall där det finns kategorier för fall som är antingen vedertagna, idéer eller slutförda. Varje sprint har en egen gren som innehåller tillhörande scenarion, uppgifter, pågående uppgifter, uppgifter som ska godkännas och godkända uppgifter.

För att generera lättöverskådlig dokumentation från koden används *TomDoc*. Då skrivs exempelvis klass- och funktionskommentarer i koden, som sedan genereras till en *HTML*-sida.

3.7 Modelleringsstandard och rutiner

Systemet kommer att modelleras upp som UML-modeller på tavla. Det ger hela teamet en möjlighet att vara delaktiga och få samma bild av systemet. Inför varje ny implementation bör nya strukturer modelleras för att säkerställa att samtliga utvecklare har en gemensam bild av hur strukturerna skall komma att se ut.

Kapitel 4

Organisation

4.1 Teammedlemmar och kontaktuppgifter

- Klas Eskilson, klaes950@student.liu.se, 073-730 73 56
- Ronja Grosz, rongr946@student.liu.se, 076-218 64 26
- Therése Komstadius, theko867@student.liu.se, 073-940 42 28
- Erik Olsson eriol726@student.liu.se, 073-840 68 72
- Daniel Rönnkvist, danro716@student.liu.se, 076-396 74 24

4.2 Ansvarsfördelning och arbetsgrupper

4.2.1 Erik Olsson - *Scrummästaren*

Det är Scrummästarens uppgift att se till så att Scrum-teamet efterföljer Scrum-teorin och få de att förstå hur den fungerar. Scrummästaren hjälper också produktägaren med att hantera produkt backloggen på rätt sätt. Scrummästaren ska även ansvar för att produktutvecklingen flyter på och att det råder en god arbetsro.

4.2.2 Therése Komstadius - *Produktägare*

Dennes uppgift är att hålla kontakt med kunden och i takt med det hålla kraven uppdaterade. Största ansvarsområde är att se till att projektets produkt-backlogg är uppdaterad och organiserad på ett sätt som gör att projektmålen går att nås. Det är även viktigt att alla förstår kraven som finns i den och att den är synlig och tydlig för alla.

4.2.3 Ronja Grosz - *Dokumentansvarig*

Dokumentansvarige har hand om att dokumentationen och dokumentationsprinciperna av projektet följs och utförs korrekt. Så som att se till att mötesprotokoll alltid förs vid varje möte av beslutsgrundande karaktär och att samla alla relevanta dokument.

4.2.4 Daniel Rönnkvist - *Testansvarig*

Som testansvarig har denne som uppgift att säkerställa att tester skrivs som kontrollerar att kraven för den aktuella sprinten uppfylls. Det åligger även denne att se till så att testning sker kontinuerligt och att tester skrivs inför och under varje sprint. Denne ska även jobba för att skriva tester för att upptäcka brister i systemet.

4.2.5 Klas Eskilson - *Kodansvarig*

Uppgiften för den kodansvarige är huvudsakligen ansvarig för kodgranskningen inom gruppen. Koden som begärs att sammanfoga med redan befintlig kod skall hålla en god kvalité och inte upprepa sig för mycket. Det är den kodansvariges ansvar att se till att denna granskning görs av minst två personer som inte var involverade i utvecklingen av det som ska sammanfogas. Till sin hjälp har den kodansvarige verktyg som automatiserar detta, tex *Code Climate*.

4.3 Mötesprinciper och rutiner

Dagar börjar 08:30 och slutar 17:00 om inte annat anges. För principer kring Scrum-relaterade möten, se kapitlet 2.2. Mötena kommer att ske varje onsdag till fredag under vårens första period, diskussion om vilka arbetsdagar som kommer vara aktuella under andra perioden sker senare. All information angående salsbokning och mötestider sker via *Facebook*-gruppen. Personen som bokar sal varierar beroende på hur många timmar denne har att utnyttja i salsbokningsystemet.

Kapitel 5

Teknisk beskrivning

5.1 Målplattform, behov av programvara, grundläggande systemarkitektur, standarder och APIer

Systemet kommer att utvecklas i *Ruby* med ramverket *Ruby on Rails*. Detta ramverk är byggt enligt designmönstret *Model-View-Controller*, där olika delar i systemet är nedbrutna i olika komponenter. I *model* hanteras databasen och dess innehåll. I *view* lagras alla olika vyer, tex hur det ser ut när en användare ombeds logga in eller när någon besöker startsidan. *Controller* fungerar som spindeln i nätet. Här finns logik, och denna komponent förser view med data från model.

För databashantering används Ruby on Rails inbyggda *ActiveRecord*-modul, som bygger på *object relation model*. Detta innebär att det är enkelt att strukturera upp relationer mellan olika tabeller och inlägg. Exempelvis kan man säga att en användare har många filer, och på så sätt låta systemet enkelt lista en användares filer, utan att skapa avancerade *SQL*-frågor.

För att användarens användarupplevelse inte ska avbrytas utav att sidan laddas om för ofta kommer många anrop till ett eget API att ske via *Javascript* och med hjälp av *AngularJS* presentera detta för användaren. Detta för att sträva efter en snabb och effektiv upplevelse för användaren. Till exempel vid sökning så kommer anrop ske löpande och resultaten renderas i webbläsaren.

5.2 *Build*-miljö, *IDE*, kompilator, *debug*-verktyg, profileringsverktyg

Då Ruby on rails inte kräver en kompilator kommer valet av textredigerare vara upp till programmeraren. Ett alternativ till en vanlig textredigerare är att använda *IDE*:n *Rubymine*, som går att använda gratis som student.

De *debug*-verktyg som redan är inbyggda i rails kommer i huvudsak att användas, men även verktyg så som; *byebug*, *better-errors* och *did-you-mean*.

För testning kommer verktyget *minitest* att implementeras och tredjepartstjänsten *Travis CI* kommer att kopplas till systemet för att användas som ett komplement till kodgranskning för att försäkra kodstandard. Minitest kommer även att användas för att köra test för att mäta systemets prestanda. För att kolla *test-coverage* och *code smells* osv. ska tredjepartstjänsten Code Climate användas. Detta för att automatisera processen att hitta duplicerad kod och på så sätt snabba på och underlätta för refaktorering.

5.3 Systembegränsningar, responstid, körtid

För att systemet inte ska uppfattas som långsamt och irriterande för användaren är målet att en sökning får ta max tre sekunder. Efter lätta undersökningar på hur liknande tjänster levererar sin sökning är tre sekunder en rimlig begränsning.

Programmet kommer endast att utvecklas för att kunna köras i UNIX-miljö. *Windows* prioriteras inte på grund av att ingen av de tänkta servrar där projektet kommer att köras är en Windows server. Windows-maskiner kommer fortfarande att kunna utnyttja tjänsten via en webbläsare.

Ansvaret för vad som laddas upp på servern ligger endast på användaren. Detta krav måste användaren acceptera för att få ta del av webbtjänsten. Alltså är det användarens skyldighet att läsa på om reglerna som gäller för att använda vår webbtjänst. Olagliga filer har systemadministratörerna rätt att ta bort.

5.4 Standarder, metoder och tredjeparts-APIer

Som standard skall GitHubs stilguide för Ruby, *JavaScript* och *CSS* användas. Den finns här.

För att hantera användares filer via Dropbox kommer deras *API* att användas. Detta genom den Ruby-klient som Dropbox tillhandahåller för att enkelt uppdatera, lista och skapa filer på användarens Dropbox-konto. På sikt skall även *Google*:s och *Box*:s API för filhantering undersökas för att se om det kan användas.

5.5 Systemmiljö, filer/filformat, input och output

Dropbox kommer användas som server för att underlätta hantering av filer. Därav kommer webbtjänsten kunna hantera de filtyper Dropbox stödjer. I dagsläget innebär detta i princip samtliga filer som går att lagra på en dators hårddisk. Inledningsvis kommer fokus ligga på att kunna presentera bilder, länkar, *pdf*-filer, ljud-filer samt videor. För att kunna redigera en fil måste den laddas ner från webbtjänsten till sin dator sen är det upp till användaren att själv hitta ett passande program för att redigera filen.