Bruse - en webbdatabas för filhantering
Projektrapport, TNM094

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

14 maj 2015

Sammanfattning

Den här rapporten beskriver hur en webbdatabas för filhantering utvecklades i kursen Medietekninskt Kandidatprojekt, TNM094, vid Linköpings universitet. Syftet med projektet var att gruppen skulle följa ett agilt arbetssätt för att uppnå kundens krav. Kundens främsta krav var att databasen skulle användas för att 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 informationen snabbare. Serversystemet utvecklades i ramverket Ruby on rails och databasens klientsida utvecklades i ramverket Angularjs.

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 information på ett bra sätt. 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 filer på ett annorlunda sätt jämfört med den vanliga mappstrukturen. Kunden hade som önskemål att enkelt kunna sortera filer och skapa en egen hierarki med hjälp av att ge varje fil en eller flera taggar. Med taggar åsyftades ett eller flera ord 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 sig igenom en djup mappstruktur för att hitta önskat objekt.

1.2 Syfte

Syftet med projektet är att gruppen ska utveckla en webbtjänst för lagring av filer som användaren snabbt 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 sina lokala filer och synkronisera med andra lagringstjänster. Poängen med att synkronisera andra lagringstjänster till en webbtjänst är att användaren kan hantera alla sina uppladdade filer på ett och samma ställe.

För att utveckla den önskade slutprodukten jobbade gruppen enligt utvecklingsmetodiken Scrum. Syftet med att använda Scrum var att tillämpa ett agilt arbetsmönster. 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 nöjdhet och för att utveckla ett flexibelt system som förblir relevant. [2]

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

1.3 Frågeställning

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

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 kod skall gås igenom utav servern. Frågan handlar alltså om hur systemet kan utvecklas för att redan tidigt tackla detta. Hur går det att säkerställa att användarens sökord snabbt genererar resultat, oavsett hur komplex sökfrågan är? 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 känsligt ämne då webbtjänster diskuteras. Ämnet är dessutom extra 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 webbapplikation 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 ska gärna 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

2.1 Taggar

För att uppnå en effektiv sökstruktur på en databas finns det två olika metoder. Den första metoden kallas professionell indexering och är hierarkisk [3]. Den andra metoden kallas folksonomi vilket bygger på att man sätter etiketter/taggar på innehållet utan någon särskild hierki[3]. De två olika metoderna har både för och nackdelar men sammanfattat kan man beskriva det som *"Indexering strävar efter precision medan taggning strävar efter hantering"* [3].

Fördelarna med professionell indexering är att den är mer precis jämfört med folksonmoni 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 finner man den sökta filen. Till skillnad från folksnomi måste man ta sig fram via nyckelord (taggar) och det är inte alltid säkert att nyckelordet leder en rätt. Användare kan också ha olika uppfattning om vad basnivån är på innehållet vilket antingen gör att onödiga taggar läggs till eller att det finns brist på taggar. Om till exempel en Javascript-fil ska taggas är det kanske onödigt för en användare att tagga filen med "webb" medan det är nödvändigt för en annan. Därför kan ett traditionellt klassificerings system ge ett bättre resultat eftersom det är helt opersonligt.

Trots att det inte finns någon stavningskontroll när taggen skrivs in kan detta vara en fördel eftersom användaren då kan använda sig av slanguttryck och dialekter som kan precisera innehållet. Fördelen med att låta användaren stava taggarna precis som denne vill gör det lättare för personer med samma intresse att söka rätt på innehåll. 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 innhållet 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.

2.2 Gränssnitt

För att designa ett användavänligt gränssnitt bör man utgå från Normans principer[4] för att åstadkomma en viss användarupplevelse. Principerna grundar sig i hur en människa uppfattar och tar in information. Dessa utgår ifrån sex punkter: synlighet, återkoppling, begränsningar, mappning, konsekvens och affordans.

2.2.1 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 ett förstoringsglas betyder sök. För att göra det lättare för användaren att hitta det mest relevanta på sidan kan man använda mycket *white space*[5]. *White space* är grafiskt utrymme som inte innehåller någon typ av information, det 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, till exempel omringas sökfältet av mycket *white space* eftersom sökfältet är det mest relevanta på startsidan.

2.2.2 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 felmeddelanden är exempel på viktig återkoppling[4]. Utan en laddningssymbol får användaren känslan av att ingenting händer och kanske går tillbaka och upprepar steget vilket förstör hela processen.

2.2.3 Begränsningar

För att användaren inte ska göra oönskade saker i systemet är det en bra idé att begränsa alternativen[4]. Till exempel går det inte att skapa mappar för att användaren ska ha en bättre överblick på filerna. För att koppla på ett externt konto måste det först godkännas av användaren för att säkerställa att det inte sker av misstag. Om användaren glömt bort sitt lösenord går det alltid att återskapa det. Därför finns det ingen risk att man aldrig lyckas logga in igen på grund av att inloggningsuppgifterna har glömts bort.

2.2.4 Mappning

Mappning innebär att man samlar liknade innehåll på samma plats för att hålla en god struktur[4]. Användare ska snabbt kunna hitta filens tillhörande funktioner. I Bruse samlas alla externa lagringtjänster vid sidan om varandra eftersom de fyller samma funktion.

2.2.5 Konsekvens

För att göra det lättare för användaren att minnas hur denne använder 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[4].

2.2.6 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[4] ingen ytterligare beskrivning krävs. Samma sak gäller på en hemsida. Användaren ska inte behöva fundera över vad som kommer hända när denne klickar på exempelvis en papperskorg, något kommer raderas.

2.3 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 använda systemet [6]. Genom användartester drogs slutsatsen att ett databasbaserat filsystem presterade bättre utifrån dessa aspekter.

Kapitel 3

Webbdatabas för hantering av filer

3.1 Systemarkitektur

I inledningen av projektet hölls en diskussion 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.

3.1.1 Ruby och Ruby on rails

Ruby on rails är ett ramverk skrivet i språket Ruby [7]. Det är ett ramverk med öppen källkod som används utav flera stora tjänster på nätet, bland andra 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 sin egna tydliga plats och syfte.

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. Genom dessa klasser hanteras all information om just användare respektive filer, och samtliga databasoperationer sker genom klassernas metoder.

Den komponent som sköter interaktionen mellan användare 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 kan också logik för att hantera undantag, till exempel att datan saknas, finnas. Den *controller* som anropats skickar sedan resultatet av användarens begäran vidare till användaren.

Det som användaren i sin tur interagerar med är systemets *views*, på svenska vyer. Här presenteras det som systemets *controllers* producerat. Knappen som användaren trycker på i exemplet i stycket ovan skapas i en *view*. Här kan 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 figur 3.1. 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*.

En annan figur som snarare fokuserar på MVC ur ett användarperspektiv visas i figur 3.2.

Fördelarna med att använda ett webbramverk som Ruby on rails eller motsvarande är flera, och kan generaliseras med att säga att det är onödigt att uppfinna hjulet på nytt. Säkerhet, effektivitet och struktur är aspekter som förstärks utav Ruby on rails. En mer konkret fördel med Ruby on rails är klassen *ActiveRecord* som alla modeller i systemet ärver av. *ActiveRecord* har funktioner för att hantera relationer och frågor mot databasen. Mer om *ActiveRecord* och databashantering i kapitel 3.4.1.

3.1.2 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.

Ramverket Angularjs valdes bland annat på grund av dess stöd för så kallade templates men även för dess tvåvägsbindning för variabler. Tvåvägsbindningen gjorde det enklare att utveckla komponenter som beror på hur användaren interagrerar 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.

Angularjs är ett så kallat MVW-ramverk, vilket står för *Model-View-Whatever* [8]. 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.

3.2 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 fortsätta bidra till detta nätverk utvecklades även projektet med hjälp av öppen källkod, för att kunna dela de lösningar som tagits fram.

För att underlätta för utvecklare har Ruby on rails byggts med många verktyg för utvecklare. Möjligheten finns även för utvecklare att bygga egna verktyg som kallas för *gems* och är ofta fria att använda och även de är skrivna med öppen källkod. Några 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 på riktigt öppnar Letter opener mailen 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.

3.3 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.

3.3.1 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 blir klickad 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 från tjänsten till Dropbox samlade, skapades först en mapp i Dropbox dit filerna laddades upp till.

3.3.2 Google drive

Likt Dropbox användes Google drives 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*. 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, se figur 3.3.

3.3.3 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. Detta innebär att användaren kan markera en eller flera filer på sin dator, och dra dem till webbläsarfönstret där systemet är öppet, och släppa dem där. 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.

3.4 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 [9].

3.4.1 Databasstruktur

Inledningvis i projektet hölls ett möte för att ta fram en grundläggande databastruktur. Denna innehöll användare, användares olika externa konton (till exempel Dropbox eller Google Drive), filer och nyckelord. Vidare skapades också en tabell som länkade samman filer och nyckelord.

3.4.2 Databashantering

En stor komponent utav Ruby on Rails är modulen som heter `ActiveRecord`. Detta är en modul vars syfte är att abstrahera och förenkla databashantering [10] och skapades efter ett designmönster med samma namn [11, kapitel 1]. Istället för att skriva databasfrågor manuellt kan du använda systemets `ActiveRecord`-modeller för att förenkla arbetet. Rent praktiskt kan detta innebära att ersätta `SELECT * FROM table WHERE column1='value'` med `Table.where(column1: 'value')`.

Fördelen med detta var att komplexa relationer mellan olika tabeller kunde förenklas, och en objektorienterad struktur med Rubyklasser 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 nödvändigt.

I och med att Rails `ActiveRecord`-modul hade inbyggt stöd för SQL-databaser användes dessa för systemets relationella tabeller. Mer specifikt användes Postgresql i systemet. En fördel med Postgresql är dess indexering som bygger på bland annat B-trees [12]. (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.

3.5 Gränssnitt

...

3.6 Testning

Sedan version 1.9 utav Ruby har standardverktyget för testning varit Minitest [13]. Minitest erbjuder många delar som behövs för att kunna hantera testning. Det som användes mest var så kallade fixtures, en datarepresentation 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 build-servern Travis CI. Varje gång ny kod laddades upp gavs då ett resultat om hur testerna gick. Med denna kontinuerliga uppdatering går det att via GitHub följa projektets status utan att ladda ned och starta det.

För testning av gränssnittet krävs dock en renderingsmotor för Javascript. Då den valda build-servern redan hade en motor 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 end-to-end testning.

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

3.7 Utvecklingsmetodik

Genom att följa en bra utvecklingsmetodik gavs en bra grund och tydliga riktlinjer att följa under projektets gång. Den utvecklingsmetodik som låg till grund för projektet var Scrum [14]. Scrum är ett ramverk som innefattar olika roller, aktiviteter och tekniker för 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.

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

- 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.
- 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 som är den lista där alla scenarion sparas.
- 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.
- 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.
- Dokumentansvarig: Dokumentansvarig hade till uppgift att säkerställa att projektet blev ordentligt dokumenterat. Det var önskvärt att ha protokoll från kundmöten, teammöten med beslutsfattande karaktär och dokumentera skisser och liknande vid diskussioner kring systemets uppbyggnad.

Hela utvecklingsperioden delades upp i mindre tidsrutor, så kallade sprints. Dessa sprints 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. Detta för att hålla mötet kort och fokuserat. Varje

individ fick under detta möte berätta vad de gjorde senast och vad de skulle göra denna kommande dag. 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 två avslutande möten. Det första av dessa var en sprintgranskning. Detta var ett tillfälle då hela teamet kunde diskutera det senaste projektinkrement 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 uppfyllda, gav denna genomgång en tydlig bild huruvida sprintmålet uppnåtts eller inte. Nästa del i mötet var att diskutera de problem som uppstått under sprintens gång och hur dessa blivit lösta. Återstoden av dessa möten ägnades åt att diskutera den dåvarande backloggen, vad som skulle göras härnäst och vilka variabler som fanns 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 yttra 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] i ett Google sheet 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?!?!]

Här presenteras kort det fokus som fanns för varje sprint. Fullständiga sprintbackloggar går att se i [APPENDIX!!=!=?!].

3.7.1 Sprint 1 - MVP

Målet med denna sprint var att ha en *Minimal Viable Product*, MVP [KÄLLA] 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.

3.7.2 Sprint 2 - Utveckla den tekniska funktionaliteten

Fokus här var att göra filerna lättåtkomliga och sökbara. Dock stöttes det på ett problem gällande Dropbox som fildatabas. En kostnad för HTTPS-certifikat 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 databaser. Det teamet kom överens om var att implementera Google Drive och även skapa en lokal databas för egen server. Då förändringar ledde till ett väldigt spretigt och icke målbaserat arbete valdes det att avsluta denna sprint i förtid.

3.7.3 Sprint 3 - Tänka på användaren

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

3.7.4 Sprint 4 - Färdig produkt

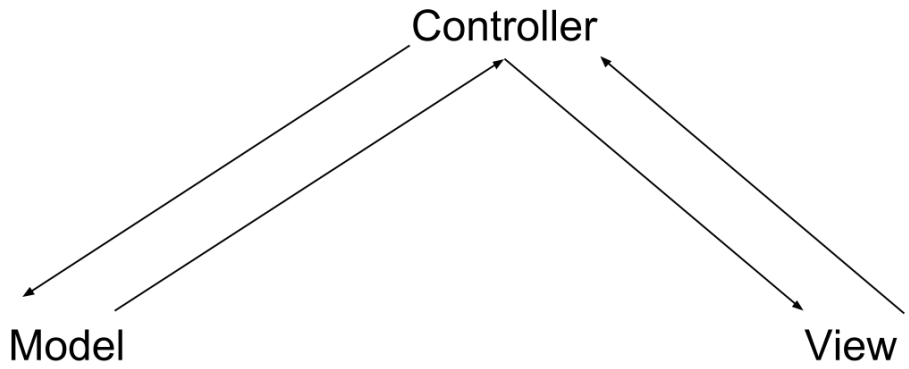
I denna sista sprint var det önskvärt att skapa ett användargränssnitt, få klart alla öppna scenarion och förbättra redan implementerad kod.

3.8 Versionshantering och kodgranskning

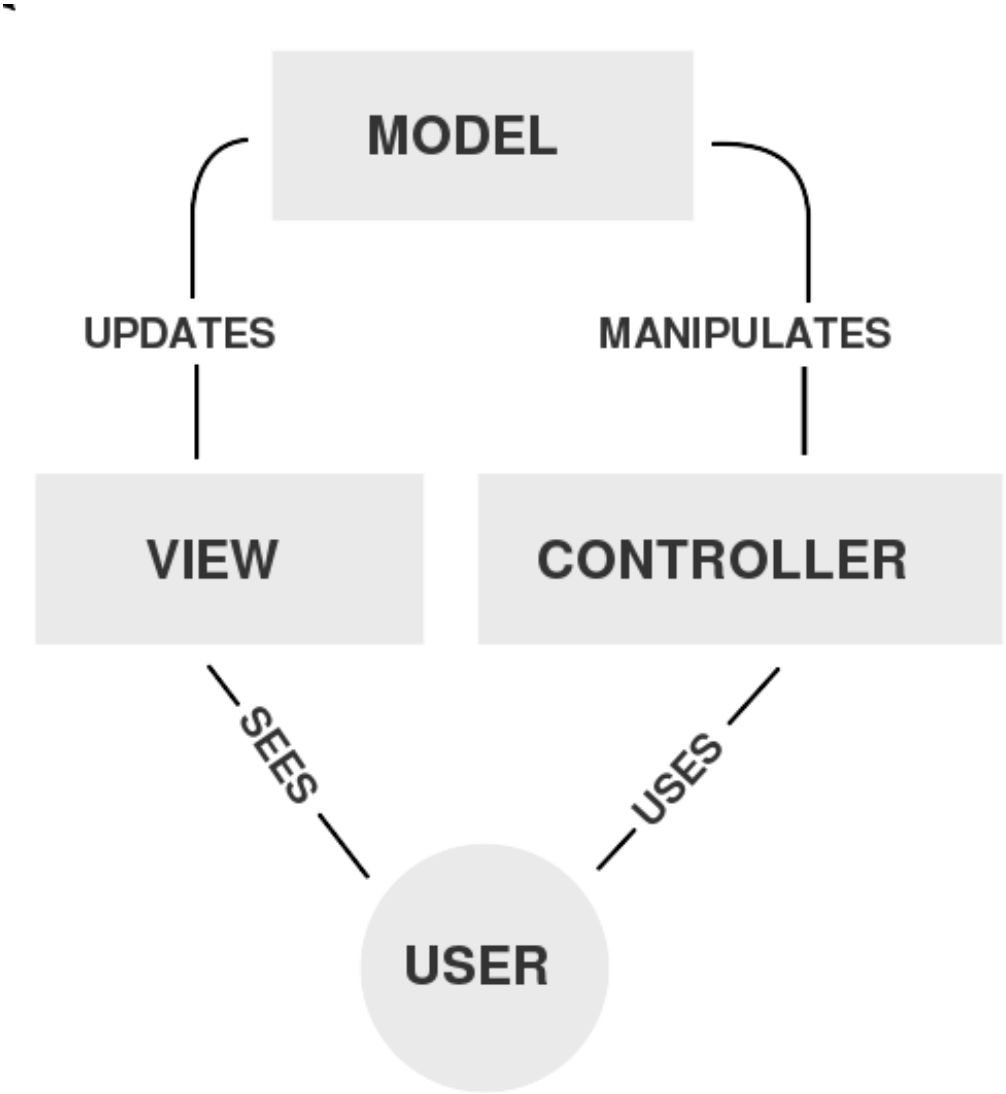
För versionshantering användes Git i kombination med Github genomgående i projektet. Git är ett distribuerat versionshanteringssystem där samtliga utvecklare har en komplett lokal kopia av hela projektet på dess egna datorer [15, kapitel 1.1]. Detta underlättade för en agil utveckling där målet var att alltid ha en fungerande produkt.

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 höll sedan koll på hur dessa kunde sammanfogas på enklast möjliga vis [15, kapitel 3.1].

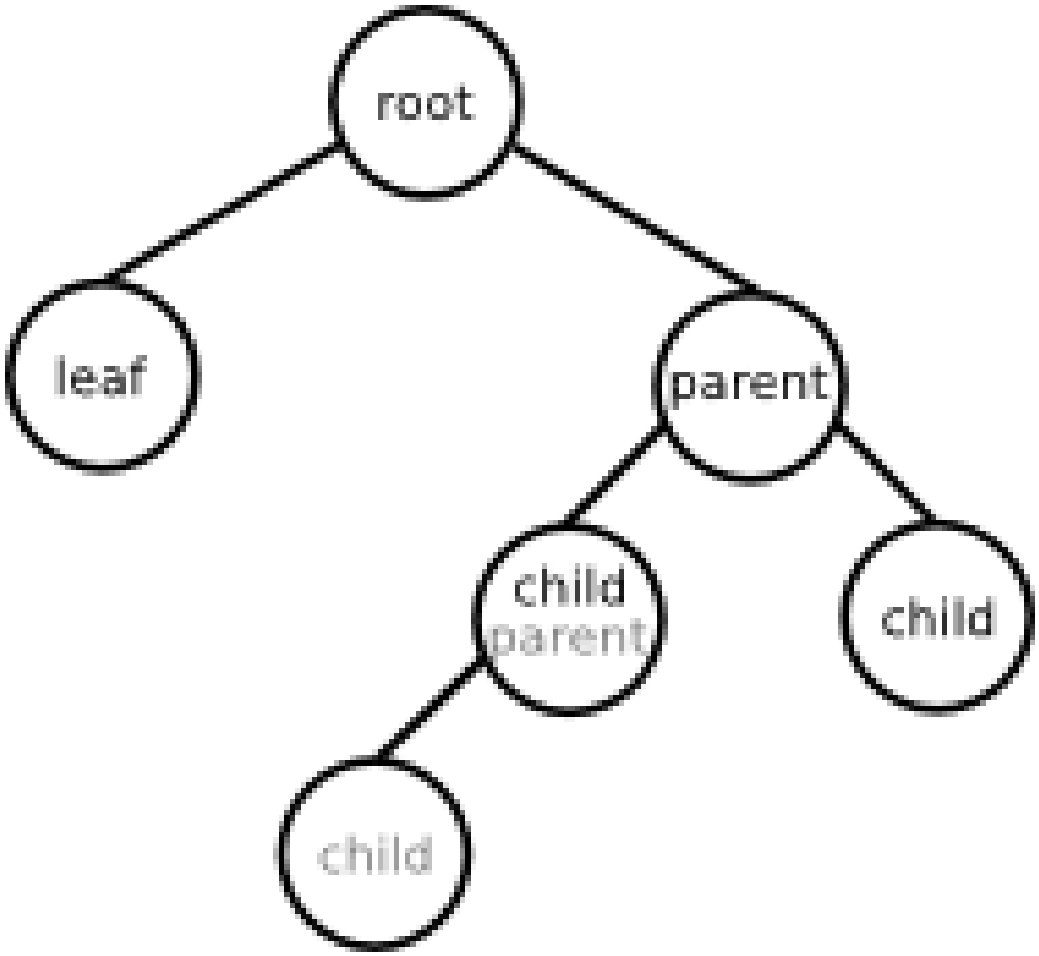
Inom förgreningen implementerades också en metodik som populärt kallas *feature branches* – funktionalitetsgrenar [16, kapitel 16]. Då en ny funktionalitet skulle implementeras i systemet skapade den ansvarige utvecklaren en förgrening. När utvecklaren var klar med det den jobbade med, skapades en förfrågan 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.



Figur 3.1: Visar på dataflödet i ett MVC-system



Figur 3.2: Visar på MVC-flödet ur ett användarperspektiv.



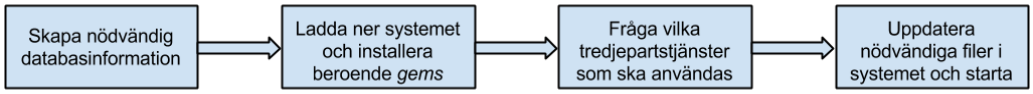
Figur 3.3: Visar strukturen för parametrarna *parent* och *child* i Google drive.

Kapitel 4

Resultat

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 demonstreras i bild 4.1.

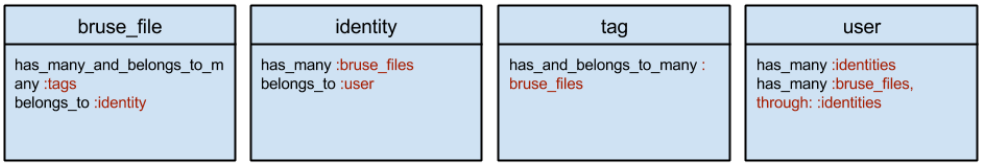


Figur 4.1: De steg som installationsfilen tar för att installera systemet.

4.2 Systemarkitektur

4.2.1 Ruby on rails

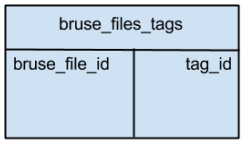
Följande modeller valdes att användas för systemet. Relationerna mellan de är specificerade i bild 4.2. Modellen *BruseFile* fick det namnet eftersom *File* är ett reserverat ord i Ruby och lösningen blev då att lägga till *bruse*.



Figur 4.2: De *models* som finns i systemet.

Identity är en av systemets viktigaste models. En *Identity* är antingen en av tredjepartstjänsterna Google Drive eller Dropbox. Men även den egna filhantering hanteras som en *Identity* för att hålla det konsekvent och modulärt.

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 bild 4.3.



Figur 4.3: Relationen mellan *BruseFiles* och *tags* i databasen.

I systemet finns det flera *controllers* för att hantera logiken. Varje *model* har en *controller*, men *bruse_file* delades upp i tre då den innehåller all logik kring uppladdning, nerladdning, visa filer, lista med mera.

4.2.2 Angularjs

De områden som styrs utav Angularjs är:

- Lägga till filer
- Lägga till taggar till nyligen tillagda filer
- Söka efter filer
- Skapa filer från text
- Dra och släppa-uppladdning
- Lista filer

För att inte behöva upprepa kod, för till exempel hämta information kring filer, skapades factories för att kunna delas utav flera *controllers*, där finns logik och rätt parametrar för att kunna skicka en *AJAX*-förfrågan till servern.

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 tredjeparts-tjänst skulle vara nödvändigt för att hålla tidsramen. Devise valdes för att det gav den mest kompletta lösningen med färdiga vyer, modifierbara *controllers* och enkel *model*-påbyggnad.

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 trigrams* över valda kolumner. *Trigrams* delar upp en instans och grupperar dem tre i taget. 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 märktes att numeriska eller nordiska tecken inte blev till *trigrams*. Men då Fuzzily är open source kunde problemet åtgärdas genom att ändra källkoden.

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 i koden som kunde åtgärdas tack vare att det var skrivet med öppen källkod.

4.4 Hantering och strukturering av databaser

Den första versionen av databasen hade följande struktur:

- User
  - id (heltal, unik nyckel)
  - name (sträng, användarens namn)
- Identity
  - id (heltal, unik nyckel)
  - user_id (heltal, referens till användare)
  - service (sträng, till exempel “Dropbox”)
  - token (sträng, nyckel som används för kontakt med extern API)
- File
  - id (heltal, unik nyckel)
  - identity_id (heltal, referens till identity)
  - name (sträng, filens namn)
  - foreign_ref (sträng, hur den externa tjänsten har koll på filerna)
  - filetype (sträng, filens typ)
  - meta (sträng, extra information om filen)
- Tag
  - id (heltal, unik nyckel)
  - name (sträng, nyckelordet)
- FileTag
  - file_id (heltal, referens till fil)
  - tag_id (heltal, referens till nyckelord)

4.5 Filhantering

Då en fil lagrades i systemet är 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. Vidare sparas också en textsträng som motsvarar hur den aktuella tjänsten som filen tillhör håller koll på filen. 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.

4.6 Gränssnitt

4.7 Testning

De tester som utvecklarna lade mest tid på att implementera var tester för *models*, *controllers* men även 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 fås en helhetsblick och väldigt stor del av systemet testas i ett och samma test.

För kunna testa viss funktionalitet krävs data i databasen, för detta har Minitest något som heter Fixtures. Det är data som man kan fejka och Minitest hanterar sedan dessa som databasinlägg.

4.8 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 sprints sprintgranskning och sprintåterblick har givit en tydlig bild om vad som gjorts och behöver göras för att nå nästa förbättring av produkten.

4.9 Versionshantering och kodgranskning

Kapitel 5

Analys och diskussion

5.1 Metod

5.1.1 Systemarkitektur

5.1.2 Filhantering

5.1.3 Hantering och strukturering av databaser

5.1.4 Gränssnitt

5.1.5 Testning

5.1.6 utvecklingsmetodik

5.1.7 Versionshantering och kodgranskning

5.2 Resultat

5.2.1 Filhantering

5.2.2 Tredjepartsverktyg

5.3 Arbetet i ett vidare sammanhang

5.3.1 Fildelning

5.3.1.1 Filhantering

5.3.1.2 Olagliga filer

5.3.1.3 Personlig integritet

Kapitel 6

Slutsatser

6.1 Frågeställningar

6.1.1 Struktur på webbdatabas

6.1.2 Säkerhet

6.1.3 Databasbaserad filstruktur

6.1.4 Användargränssnitt

6.2 Framtida arbete

6.2.1 Slutgiltigt rapportförslag

Litteraturförteckning

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

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

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

[4] 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

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

[6] 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)

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

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

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

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

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

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

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

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

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

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

[17]

[18]

[19]

[20]

[21]

[22]

[23]

[24]

[25]

[26]

Bilaga A

Bilaga