Institutionen för datavetenskap
Department of Computer and Information Science

Examensarbete

Portering från Google Apps REST API till Microsoft Office 365 REST API

av

Tina Danielsson

LiTH-IDA/ERASMUS-G--15/003--SE

2015-06-03

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköpings universitet
Institutionen för datavetenskap

Examensarbete

Portering från Google Apps REST API till Microsoft Office 365 REST API

av

Tina Danielsson

LiTH-IDA/ERASMUS-G--15/003--SE

2015-06-03

Handledare: Johan Åberg
Examinator: Peter Dalénius

Portering från Google Apps REST API till Microsoft Office 365 REST API

Tina Danielsson
danielsson.tina@gmail.com

SAMMANFATTNING

Stress på arbetsplatsen relaterat till många inkommande och utgående kommunikationskanaler är ett reellt problem. Applikationer som samlar alla kanaler i samma verktyg kan hjälpa till på det här området. För att förenkla vid utveckling av en sådan applikation kan ett modulärt system skapas, där varje modul ser liknande ut och enkelt kan kopplas in i en huvudapplikation. Den här studien undersöker de problem som kan uppstå när flera tjänster ska integreras, mer specifikt genom att titta på hur en befintlig modul för e-post via Google Apps kan porteras för att stödja e-post via Microsoft Office 365. Arbetet har skett enligt metoder för testdriven portering och varje steg i porteringen har dokumenterats noggrant. Ett antal problemområden har identifierats och möjliga lösningar föreslås. Utfrån de problem som uppstått dras slutsatsen att de är av en sådan karaktär att de inte utgör något hinder för en portering.

Author Keywords

e-post; portering; testning; Unified Communications; Office 365; Google Apps; Gmail; API

INLEDNING

Motivering

I dagens samhälle har elektronisk kommunikation blivit en allt större del av både mångas privatliv och arbetsliv. Framför allt i arbetslivet förväntas en person att vara tillgänglig via många olika kanaler, exempelvis via e-post, chatt, sms och telefoni. Men dessa korta avbrott stör förmågan att arbeta effektivt [13] och det finns även studier som visar hur våra stressnivåer ökar när vi blir splittrade och får in information från många olika källor [3].

Ett sätt att hantera denna överbelastning är att samla alla källor i en kommuniktionslösning, ett koncept som kallas Unified Communications (UC) [17], för att på så sätt minimera risken för mottagaren att bli splittrad i sin koncentration och därmed kunna behålla bättre fokus och arbeta mer effektivt. När trenden går mot att fler och fler leverantörer erbjuder sina applikationer över internet [1] så blir processen att integrera källor enklare då plattformarna är likartade.

Bland de ledande aktörerna på internet vad gäller kontorssviter finns Microsoft och Google att hitta med sina produkter Office 365 respektive Google Apps [23]. Det gör att de är lämpliga mål att samla i en UC-lösning som fokuserar på e-post, kalender, kontakter och dokumenthantering. Arbetet med att integrera dessa två sviter kan tyckas enkel men problem kan uppstå som kräver mycket tid och resurser. För att inte slösa resurser på att utveckla samma sak två gånger så kan en svit i taget integreras så att delar därifrån kan användas för integrationen av nästa svit.

Syfte

Företaget Briteback har inspirerats att skapa en webbaserad kommunikationslösning för att lösa problemet med stress i samband med multipla kommunikationskällor. Produkten är uppbyggd av fristående delar, så kallade moduler, vilket gör att när en ny källa ska läggas till så finns redan en sorts mall på plats som anger en grundläggande struktur och innehåll för den nya modulen. Den initiala produkten hade stöd för Google Apps e-post samt kalender och Briteback befann sig därefter i ett skede där de var redo att börja integrera flera källor i sin produkt. Näst på tur att integreras i produkten var Microsoft Office 365. Utvecklingen av den modulen har gett upphov till den fallstudie som den här rapporten beskriver, nämligen att studera likheter och skillnader mellan Google Apps och Office 365:s REST API:er i samband med portering mellan dessa.

Frågeställning

Fokus för studien är den tekniska implementationen av en lösning som använder sig av Office 365 REST API, vilket återspeglar sig i de specifika frågeställningarna nedan.

1. Vilka problem kan uppstå vid portering från Google Apps REST API till Microsoft Office 365 REST API?

2. Hur kan dessa problem hanteras?

Avgränsningar

Den här rapporten vänder sig främst till personer med åtminstone några års praktisk eller teoretisk kunskap inom programmering samt de som är intresserade av REST API:er i allmänhet eller mer specifikt Microsoft Office REST API eller Google Apps API.

På grund av den tid som fanns allokerad för att utföra studien så begränsades den till att endast inkludera e-post och fokuserar därmed på Gmail REST API och Outlook Mail REST API. En fungerande inlogging var ett krav för att det skulle vara möjligt men den funktionaliteten utvecklades separat i förväg av en annan utvecklare.

Som utgångspunkt för porteringen har applikationen Briteback använts. Det är en webbaserad applikation skriven i HTML5, CSS3 och JavaScript. Serversidan är skriven i Java. Under den här studiens arbete har all utveckling skett med JavaScript.

Författaren, som har utfört porteringen samt samlat in all data, har drygt tre års högskolestudier inom flera olika paradigmer och programmeringsspråk bakom sig men inte någon större praktisk erfarenhet av det programmeringsspråk som användes under studien. Arbetet har utförts i nära samarbete med ett team av utvecklare som arbetat med applikationen Briteback under en längre tid.

TEORI

Det som kommer studeras i den här rapporten är uppgiften att portera mellan REST API:er för molntjänsterna Google Apps och Microsoft Office 365. Nedan beskrivs dessa begrepp och miljöer mer detaljerat.

Portering

Att portera ett program innebär att skriva om den befintliga koden till ett annat format. Ett vanligt scenario är att en applikation som är utvecklad till ett operativsystem även ska fungera i ett annat operativsystem. Om orginalapplikationen inte är skriven i ett plattformsoberoende programmeringsspråk så måste helt ny kod skrivas, ofta i ett annat språk. Det innebär att all funktionalitet ska ”kopieras” från den tidigare applikationen till den nya. Det kan i många fall vara svårt då olika språk kan ha helt olika syntaxer och metoder för att utföra samma moment.

Vad gäller denna studie var porteringen något enklare då samma programmeringsspråk användes både i den ursprungliga och den nya produkten. Skillnaden låg istället i att produkterna skulle använda sig av olika REST API:er som anslöt mot varsin molntjänst.

API

API står för Application Programming Interface och översätts på svenska till applikationsprogrammeringsgränssnitt. Syftet med ett API är att ge programmeraren ett gränssnitt mot en viss applikation eller tjänst [10]. Det är inte ovanligt att en applikation är skriven i ett språk men att det finns möjlighet att göra programanrop med flera olika språk via applikationens API.

Eftersom ett API kan komma att användas i många olika applikationer och av många olika programmerare så är det viktigt att det sker så få ändringar som möjligt i det. Om förändringar sker i API:et så innebär det att applikationerna där det är använt kan behöva skrivas om för att fortsätta fungera. de Souza et al beskriver ett API som ett ”kontrakt’ mellan API-utvecklare och API-användare [8]. API-utvecklaren garanterar en viss funktionalitet via det så kallade kontraktet som API-användaren måste kunna förlita sig på.

Det finns ingen standard som anger hur anropen i ett API ska konstrueras eller döpas, därför kan API:er skilja sig åt väsentligt även om de bakomliggande applikationerna eller tjänsterna är snarlika. Vad ett anrop gör kan också skilja sig, även om namnen är lika. Exempelvis kan ett anrop i ett API kallas *doLogin()* och innehålla både inloggning och hämtning av mail medan ett liknande anrop i ett annat API kan kallas *initLogon()* och det endast innehålla inloggning.

Anropen i ett API döljer den faktiska implementationen av den eller de interna metoder som körs vilket kan leda till problem. Exempelvis kan anrop vara beroende av varandra genom att dela på interna resurser. Ett scenario skulle kunna vara att ett anrop till *doLogin()* görs utan att ange någon användare men inloggningen går oväntat att slutföra ändå på grund av att ett användarnamn angetts tidigare i anropet *verifyUserExists()* och API:et har sparat undan den informationen i en intern datastruktur. Om det finns beroenden så bör dessa vara dokumenterade.

Det är viktigt med tydlig och omfattande dokumentation som förutom beskrivning av anrop även kan bestå av exempelvis scenarios och kodexempel [12]. I en studie från 2009 som handlade om vad som krävs för att lära sig ett API så upplevde 63% av studiens deltagare att bristfällig dokumentation hindrade dem från att lära sig väl [18]. Samma studie anger även att för att förebygga problem bör dokumentationen uppfylla följande krav:

- innehålla bra exempel
- vara komplett
- visa många komplexa användningsscenarion
- vara lämpligt organiserat
- inkludera relevanta designelement

Molntjänster

Historiskt sett har applikationer installerats lokalt på de datorer där de ska användas. Under många år har det däremot varit vanligt, speciellt på företag, att data som används av applikationerna, exempelvis dokument som öppnas i en dokumenthanterare, ligger på en nätverksplats istället för lokalt. Motiveringen till detta är ofta någon typ av säkerhetsaspekt som att kunna enklare kunna styra över behörigheter eller att förhindra dataförlust. Under de senaste åren har många applikationsleverantörer valt att ta det här steget längre och erbjuda även sina applikationer via nätverk, eller mer specifikt över internet. Det senare är ett exempel på en molntjänst[1].

Det finns många olika sorters molntjänster, exempelvis IBM SmartCloud Enterprise[2], Microsoft Azure Cloud Services[3] och Dropbox[4]. Det är vanligt att man delar in dem i segmenten IaaS (Infrastructure as a Service), PaaS (Platform as a Service) och SaaS (Software as a Service) [20]. Skillnaden ligger i vilken nivå i installationen som blir levererad till användaren.

För att mycket förenklat förklara skillnaderna mellan segmenten beskrivs de generella åtgärderna som krävs inom varje nivå kopplat till följande exempel: *en användare ska kunna skicka och ta emot e-post*.

Inom IaaS-segmentet levereras endast en virtuell maskin. Först måste då ett operativsystem installeras och därefter måste en e-postklient installeras. Därefter kan användaren börja arbeta.

[1] Molnet syftar förenklat beskrivet till internet eller ett nätverk
[2] http://www.ibm.com/cloud-computing/us/en/iaas.html
[3] http://azure.microsoft.com/en-us/services/cloud-services/
[4] http://www.dropbox.com/

*Inom PaaS-segmente*t levereras också en virtuell maskin men den här gången så är ett operativsystem redan installerat. Det krävs fortfarande att en e-postklient installeras innan användaren kan börja arbeta.

Inom SaaS-segmentet levereras en komplett lösning hela vägen fram till e-postklienten. Användaren behöver inte ens tänka på vad som ligger bakom programmet, det är bara sätta igång att arbeta.

Inom varje segment finns det ytterligare uppdelningar, inom SaaS finner man exempelvis kontorssviter.

Figur 1: Segment inom molntjänster. Varje segment innefattar och bygger på det underliggande segmentet.

Representional State Transfer (REST)

REST är en uppsättning regler som beskriver hur kommunikation mellan klient och server bör designas i webbsammanhang. Grundtanken är att göra det möjligt att hämta data genom att ansluta mot resurser via en URL [14]. Den data som skickas tillbaka kan antingen vara statisk eller dynamisk [14]. Varje resursanrop måste innehålla all nödvändig information för att genomföra det, det får alltså inte vara beroende av tidigare anrop [5].

Office 365

Microsoft Online Services är en molntjänst som innefattar ett antal SaaS-produkter, bland annat Office 365. Office 365 är en webbaserad kontorssvit som innehåller en e-postklient med kalender och ett antal verktyg för dokumenthantering inklusive lagringsyta[5].

Office 365 finns tillgänglig i tre fjärdedelar av världens alla länder [15] och enligt en undersökning från 2014 använder 7,7 procent av småföretag och 8,8 procent av stora företag produkten [6].

Microsoft tillhandahåller ett REST API för Office 365 som innehåller stöd för inloggning, e-post, kontakter, kalender och filhantering [11]. Det finns både referensdokumentation och exempel för vanliga uppgifter som exempelvis att logga in, att skicka meddelanden och att svara på meddelanden. Utöver det finns det även en sandlåda där API:et kan testas interaktivt.

Google Apps

Google Apps innehåller motsvarande webbaserade applikationer för e-posthantering, kalender och dokumenthantering samt lagringsyta som i Office 365[6].

För närvarande har Google Apps ett försprång på Office 365, 16,3 procent av småföretag använder Google Apps [6]. Men när det gäller större företag ligger de båda konkurrenterna lika på 8,8 procent.

Även de API:s som finns för Google Apps är baserade på REST och stöd finns för inloggning, e-post, kalender, kontakter, uppgifter, skript för dokument, backup av filer med mera [9]. Både referensdokumentation, scenarion och exempel finns tillgängliga och det finns möjlighet att interaktivt testa respektive anrop.

METOD

Det här kapitlet börjar med att beskriva utvecklingsmiljön, därefter beskrivs de metoder som användes under utvecklingen och slutligen beskrivs hur analysen av datan gick till.

Utvecklingsmiljö

Huvudprodukten består av en serverdel skriven i Java och Grails samt en klientdel skriven i JavaScript och HTML5. Det är i klientdelen som utvecklingen kopplat till den här rapporten har skett. Klientdelen kan delas in i en frontend och en backend, där frontend består av det visuella och backend består av de funktioner som krävs för att kommunicera med respektive molntjänst. Både Office 365 och Google Apps har REST API:er för JavaScript.

Implementation

Porteringen har skett enligt testdriven utveckling. Metoden för testdriven utveckling innebär kortfattat att tester skrivs innan någon kod finns på plats varefter man iterativt lägger till kod och testar på nytt tills man är nöjd med testets nytta. Kent Beck, skaparen av eXtreme Programming (XP) och nyupptäckare av testdriven utveckling [21], anger följande grundläggande regler [4]:

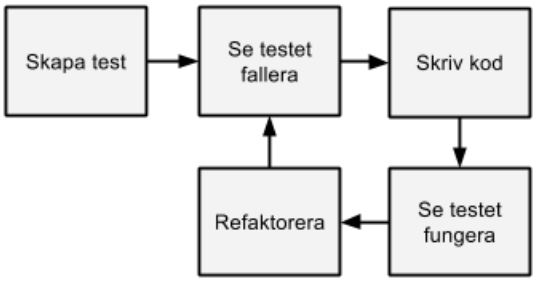
- Skriv inte en ny rad kod om du inte först har ett fallerande automatiskt test.
- Eliminera duplicering.

Utvecklingen av den nya modulen har genomförts enligt en liknande process som Bohnet och Meszaros använt sig av [7]. Processen anpassades enligt nedan.

I det första skedet av processen togs de övergripande testfallen, så kallade user stories, fram. I brist på domänexperter i form av användare av systemet, på grund av att det ännu inte var i drift, bidrog utvecklarna av Briteback till att ta fram

[5]https://products.office.com

[6]https://www.google.com/work/apps/business/products.html



Figur 2: Processen för testdriven utveckling.

större delen av testfallen. För samtliga user stories skapades automatiserade tester i Selenium, ett ramverk för att testa det grafiska gränssnittet hos webbsidor genom att exempelvis simulera musklick [19]. Testerna utvecklades mot den befintliga Google Apps-modulen i syfte att senare användas för att verifiera motsvarande funktionalitet i den nya Office 365-modulen.

Samtliga user stories dokumenterades i första hand i ett kalkylblad där även noteringar kopplade till respektive user story skrevs ner. Mer om det i stycket Dokumentation. I andra hand lades user stories in i JIRA, ett ärendehanteringssystem som kan användas som stöd vid agil utveckling [2].

Därefter prioriterades alla user stories och de började bearbetas en och en. Det första som skedde när en ny user story påbörjades var att utreda vilka metoder som exekverades i Google Apps-modulen. Varje funktion kopplades till respektive user story i kalkylbladet för att dokumentera även dessa. Slutligen utvecklades motsvarande funktioner för Office 365, även dessa enligt testdriven metod.

En user story ansågs vara klar då dess Seleniuimtest gick igenom helt eller då endast delar som inte gick att finna någon motsvarighet för i Office 365 var kvar.

Dokumentation

Ohly et al nämner två olika metoder att dokumentera en personlig upplevelse [16]. Den första metoden, att ta stickprov av händelser (eng. event sampling), handlar om att bemöta frågor knutna till en specifik händelse omedelbart. Den andra metoden, att föra dagbok (eng. daily diaries), handlar om att skriva ner erfarenheter under dagen utan att koppla dem till någon specifik händelse. Båda dessa metoder har använts för att dokumentera porteringen löpande under utvecklingens gång.

Den händelseknutna delen av dokumentationen skedde genom att logga tid med verktyget Toggl [22], och genom att göra kontinuerliga noteringar. Till skillnad från stickprovsmodellen så dokumenterades samtliga funktioner. Noteringen fungerade så att efter varje färdigutvecklad funktion så besvarades följande frågor:

- Hur lång tid tog porteringen av funktionen?
- Vilka API-anrop för Gmail användes i källkoden?
- Vilka API-anrop behövde göras för Outlook Mail för att få motsvarande funktionalitet?
- Vilka eventuella problem uppstod?
- Finns det några andra upplevelser, tankar, ideér med mera som är vara värdefulla att dokumentera?

I slutet av varje dag avsattes en halvtimme till reflektion över dagens arbete och i samband med det noterades ytterligare saker som kändes lämpliga att dokumentera. Dagsnoteringarna kopplades till respektive user story men var mer övergripande än noteringarna som gjordes tidigare under dagen.

Analys

När porteringen slutförts analyserades dokumentationen genom att först kategorisera de problem som uppstått. Följande kategorier valdes ut baserat på de problem som uppstått:

- Arkitekturproblem
  Problem som beror på att Google och Microsoft har valt olika arkitektur i sina e-postsystem.
- Funktionalitetsproblem
  Problem som beror på att motsvarande anrop eller funktionalitet saknas helt i Outlook REST API, trots att det finns i Microsofts Outlook 365-klient.

Därefter rangordnades problemen inom varje kategori genom att de fick poäng baserat på följande kriterier:

1p tog max 30 minuter att lösa

2p tog minst 30 minuter och upp till 1 timme att lösa

3p tog minst 1 timme och upp till 4 timmar att lösa

4p tog minst 4 timmar och upp till 8 timmar att lösa

5p tog minst 8 timmar eller längre att lösa

1p problemet var trivialt och kunde lösas utan någon dokumentation

2p problemet kunde lösas med hjälp av API:ets dokumentation

3p problemet kunde inte lösas med hjälp av API:ets dokumentation, men med hjälp annan dokumentation

4p problemet kunde inte lösas med hjälp av någon funnen dokumentation, trial-and-error krävdes

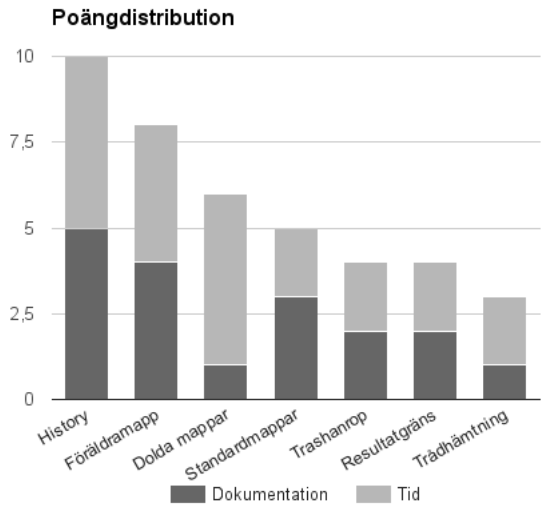
5p problemet kunde inte lösas alls

De problem som sammanlagt fick 5 poäng eller mer valdes ut för att besvara frågeställningarna.

RESULTAT

I det här kapitlet redovisas resultatet från porteringen.

Totalt registrerades 7 problem (se figur 3) varav 4 ingick i arkitekturkategorin. Av de 7 problemen var det 4 problem som fick 5 poäng eller mer och bland dessa var det 50% fördelning mellan kategorierna.



Figur 3: Problemens poängfördelning vad gäller tidsåtgång och brist på dokumentation.

Historik saknas (10p)

När något ändras i den mapp där man står så ska vyn uppdateras automatiskt. Ändringar kan till exempel vara att ett nytt mail kommer in i inkorgen eller att ett mail flyttas in eller ut ur mappen genom en annan klient. I Gmail API så finns det ett anrop kallat *history* som ger en lista med alla ändringar som skett. Genom att ange ett historyid är det möjligt att få veta exakt vad som skett sedan senaste kontrollen, genom att alla händelser med ett högre id listas. Varje ändring har en typ så som *messagesAdded* eller *labelsDeleted* vilket gör det mycket enkelt att ändra innehållet i vyn.

I Outlook Mail API saknas en motsvarighet. För att hitta nya eller ändrade meddelanden kan man hämta alla meddelanden i en mapp och filtrera ut de som modifierats efter en viss tid. Men det går inte att hämta ut någon sorts status på meddelanden som inte längre finns i en mapp. Lösningen blev därför att jämföra mot en lista med redan hämtade meddelanden för att se om något meddelande saknades och därefter ta bort dessa ur vyn.

Direktåtkomst till objekt saknas (8p)

Vanliga operationer i en e-postklient är att döpa om mappar eller ta bort meddelanden och andra liknande åtgärder. I Gmail räcker det att man har ett objekts unika id för att kunna göra dessa operationer. För att döpa om en mapp anger man mappens id och det nya namnet och för att ta bort ett meddelande anger man meddelandets id.

I Outlook Mail API måste mappen som objektet ligger i också skickas med. Lösningen blev att spara ner föräldramappen i applikationens interna datastruktur så att den alltid fanns att tillgå när anrop skulle göras.

Attribut för visade eller dolda mappar saknas (6p)

Användaren ska kunna välja vilka mappar som ska visas i mapplistan och i Gmails API finns det stöd för det genom att titta på attributet *labelListVisibility* för en mapp.

Exakt samma funktionalitet finns inte i Outlook Mail API men att ange en mapp som *favorit* är motsvarande funktion. Det går dock inte att läsa eller sätta det attributet genom API:et och ingen alternativ lösning togs fram till applikationen.

Namn på standardmappar saknas delvis (5p)

En standard i e-postklienter är att det till finns en mapp för inkommande meddelanden och en för skickade meddelanden, andra vanliga mappar är papperskorg och skräpmeddelanden. I Britebacks applikation separeras dessa standardmappar visuellt från användarens egna mappar. Applikationen tillåter heller inte att standardmappar döps om eller tas bort. I Gmails API har mappar ett attribut som anger om det är en så kallad *systemmapp* eller *användarmapp* och standardmapparna har även fått ett eneklt id som till exempel *INBOX*, *SENT* och *TRASH*. Dessa id:n används både när man kallar på mappen och när man får tillbaka ett svar med information om mappen vilket gör det enkelt att veta vilka mappar i en lista som är en standardmapp och bör särskiljas från de övriga.

I Outlook Mail API finns det också ett antal standardmappar men det finns inget attribut som anger vilka de är och de har heller inte enkla id:n. Precis som övriga mappar har de en lång unik nyckel (se figur 4). Kortnamnen *Inbox*, *Drafts*, *SentItems* och *DeletedItems* finns definierade men det är endast alias för de riktiga id:na och kortnamnen skickas aldrig tillbaka som svar.

```
AAMkAGI2NGVhZTVlLTI1OGMtNDI4My1iZmE5LTA
5OGJiZGEzMTc0YQAuAAAAAADUuTJK1K9aTpCdqX
op_4NaAQCd9nJ-tVysQos2hTfspaWRAAAAAAEJA
AA=
```

Figur 4: Exempel på ett mappid i Office 365.

Lösningen som valdes var att göra ett anrop mot respektive mapps kortnamn för att sedan spara undan det riktiga id:t och vid senare operationer jämföra mot det sparade id:t för att veta om en mapp i fråga var en av standardmapparna.

DISKUSSION

I det här kapitlet diskuteras först de problem som togs upp i resultatkapitlet. Därefter diskuteras den valda metoden samt Office 365 och Google Apps i allmänhet. Slutligen tas samhälliga aspekter och källkritik upp.

Resultat

Historik

Briteback har valt att kontinuerligt uppdatera det användaren ser i applikationen. Det kräver upprepade anrop mot tjänsteleverantörens servrar för att hämta hem eventuella ändringar. Då varje anrop kostar resurser så är det önskvärt att göra så få anrop som möjligt med så lite data som möjligt i överföringen, annars riskerar applikationen att upplevas som långsam och trög. Då är det bra med ett anrop med det specifika syftet att endast hämta information om ändringar.

```
/* Google API mail message retrieval */
url: "https://www.googleapis.com/gmail/v1/users/me/messages/" + messageId,
params: {
  fields: "historyId,id,labelIds,payload,snippet,threadId"
}
```

```
/* Office 365 API mail message retrieval */
url: "https://outlook.office365.com/api/v1.0/me/folders/" + folderId + "/messages/" + messageId,
params: {
  $select: "Id,ChangeKey,ParentFolderId,From,ToRecipients,CcRecipients,BccRecipients,BodyPreview,
      ConversationId,Subject,BodyPreview,DateTimeReceived,Body"
}
```

Figur 5: Jämförelse av API-anrop för att hämta e-postmeddelanden.

Om applikationen istället endast skulle tillåta manuella uppdateringar när användaren specifikt ber om det blir mindre kritiskt att minimera anropen och storleken på dataöverföringarna då det inte sker lika ofta. En kortare fördröjning i applikationen skulle dessutom i det läget vara lättare för användaren att förstå eftersom den just aktivt bett applikationen att utföra ett uppdrag.

En mellanväg kan vara att endast ha automatisk uppdatering av nya meddelanden i inkorgen och manuell uppdatering i övriga mappar.

Anledningen till att den implementerade lösningen valdes var för att få en så lik funktionalitet som möjligt jämfört med Google Apps-modulen. Lösningen resulterar dock i en avsevärt mycket större dataöverföring än vad ett rent historikanrop hade resulterat i eftersom anropet måste hämta hem alla meddelanden i mappens för att kunna jämföra med tidigare hämtade meddelanden för att se vad som inte längre finns.

Ett väldigt snarlikt alternativ till den valda lösningen hade varit att helt ignorera vilken data som hämtats hem tidigare, rensat allt innehåll i mappen och laddat in den nya datan på nytt. Men då hade alla interna dataobjekt också behövts skrivas om. Varje mapp och meddelande genomgår nämligen en omvandlig till en intern datastruktur i Briteback när de hämtats från tjänsteleverantörens servrar. Orsaken till det är att det då är enklare att arbeta med objekt från olika leverantörer i samma modeller. Den valda lösningen ansågs därför mer effektiv när man såg den stora bilden och den är dessutom förberedd för den dag då Office 365 förhoppningsvis får ett historikanrop.

En radikalt annorlunda lösning hade varit att flytta uppdateringen internt i applikationen från klienten till servern. Om servern hämtar hem all data och upprätthåller en cache så kan klienten ansluta mot cachen och på så sätt minimeras flaskhalsar i klienten. Antal anrop och datastorleken hade fortfarande varit lika stor men på serversidan är det möjligt att påverka hårdvaran så att det finns tillräckliga resurser att hantera det. Men när antal klienter växer så kan det ändå komma en tidpunkt då resursförbrukningen blir orimlig och en decentraliserad modell där klienterna står för större delen av beräkningskraften blir mer lämplig ändå.

Den valda lösningen känns rätt i nuvarande läge då det är troligt att Office 365 API kommer att få ett historikanrop, nuvarande API för Exchange har nämligen ett anrop kallat *SyncFolderItems*[7] som gör motsvarande som *history* i Gmails API.

Direktåtkomst till objekt

Det här problemet uppstod vid två tillfällen, först då mappar skulle listas och senare då meddelanden skulle listas. Var för sig hade inte problemen nått upp till en poäng så att de skulle tagits med bland de största problemen, de fick 5 respektive 3 poäng, men då det i grunden handlade om samma problem slogs de ihop. Egentligen ligger inte problemet i Outlook Mail API utan att applikationen skrivits mot Gmail där ingen mapp behövs. Koden behövde alltså skrivas om mer än för andra likvärdiga anrop.

I Briteback står man alltid i specifik mapp så det är känt vilken mapp ett meddelande ligger i, därför är det inte ett problem att ha med mappen. Teoretiskt sett skulle det vara möjligt att användaren försöker göra en ändring på ett objekt som inte längre finns kvar i mappen ifall att den inte är uppdaterad, men då blir det faktiskt ett extra skydd. Om objektet inte längre finns i mappen så ska operationen kanske inte tillåtas ändå.

Men det kan finnas tillfällen då man vill arbeta med ett specifikt objekt, utan att bry sig om vilken mapp som är överordnad. Ett fall skulle kunna vara om man vill ha en samling med alla meddelanden i samma vy. När man utför operationer mot objekten så spelar det då ingen roll vilken mapp det ligger i. Eftersom informationen finns kopplad på objektet så är det dock heller inte något problem att ha med den informationen. Ett annat fall då man kan vilja vara mapplös är om man söker efter meddelanden. Säg att man tagit fram ett antal meddelanden och man en stund senare vill se om något ändrats på det specifika meddelandet, exempelvis om det har

[7] https://msdn.microsoft.com/EN-US/library/office/aa563967%28v=exchg.150%29.aspx

flyttats till en annan mapp. Det blir då ett moment 22, meddelandet måste hämtas för att se vilken mapp det ligger i för att kunna hämta det.

Det finns inget sätt att komma runt att en överordnad mapp måste skickas med i anropet, antingen explicit eller indirekt. Om ingen mapp anges så blir inkorgen vald per default. Ett sätt att komma runt problemet helt vore att skicka upprepade förfrågningar via API:et och upprätthålla en intern cache där ett meddelande och dess förälder alltid finns registrerade. Men i dagsläget finns det ingen funktionalitet som kräver direktaccess så det finns ingen anledning att implementera en alternativ lösning.

Visade/dolda mappar

Hurvida användaren har nytta av att kunna dölja mappar i mapplistan är högst individuellt, det beror helt på hur man arbetar med mappar. Den som endast använder mappar för mycket enkla saker, som exempelvis att lägga meddelanden i en mapp när de ska arbeta vidare med dem och en annan mapp när de är klara med dem, har förmodligen inget behov av att dölja någon av sina mappar. Men när mappar används för att sortera och kategorisera meddelanden kan det bli så att vissa mappar används oftare än andra. En del mappar kanske bara används för att arkivera meddelanden som man sällan eller aldrig mer tittar på medan andra mappar kanske används för meddelanden man arbetar aktivt med. Det är då hjälpsamt om endast de mappar man använder aktivt syns i mapplistan, speciellt om man har många mappar.

Eftersom Outlook Mail har vad som kallas favoritmappar så borde det inte vara några problem att implementera det, men API:et är ännu i ett tidigt stadie i sin utveckling och där finns inte stödet.

För meddelanden skulle det vara möjligt att använda kategorier eller utökade egenskaper för att ange om objektet ska visas eller inte men dessa egenskaper finns inte på mappar. Enda alternativet idag är att hantera det internt i applikationen. Mappens id och visningsstatus skulle kunna sparas ner på applikationsservern, kopplat till användarens konto. En nackdel med den lösningen är att visningsstatusen inte skulle bli samma i Outlook, som använder sig av favoritflaggan, och i Briteback, som använder sig av intern visningsstatus.

Det är troligt att det kommer bli möjligt att läsa ut information om favoritmappar när API:et mognar och beslutet togs därför att vänta med den funktionaliteten istället för att lägga tid på en egen lösning.

Namn på standardmappar

En separation av specialmappar som inkorg, skickade meddelanden, papperskorg med flera är vanligt att se i e-postklienter. En ny företeelse idag är att frångå mappar helt, Googles Inbox by Gmail[8] och Dropbox Mailbox[9] är exempel på det. Med ett sånt tänk skulle det inte spela någon roll vilken mapp som ett meddelande låg i. Genom Outlook Mail API kan man via egenskaper på ett meddelande istället få veta om det tex är ett skickat meddelande eller ett utkast.

[8]http://www.google.com/inbox/

[9]http://www.mailboxapp.com/

Men Briteback har valt att behålla grupperingen av standardmappar och för att få liknande uppdelning av standardmappar i Office 365-modulen som i Google Apps-modulen så krävdes det att applikationen fick kännedom om vilka mappar som var standardmappar. Vissa interna funktioner i applikationen är också beroende av att veta om en mapp är en standardmapp. Den lösning som valdes löser frågan om vilka mappar som är de olika standardmapparna, men det är inte en optimal lösning då samma anrop måste göras upprepade gånger för att få ut den informationen.

En alternativ lösning vore att titta på mappens namn istället för dess id. Det man då behöver tänka på är att undersöka vilken föräldramapp mappen ligger i för det är möjligt att skapa en mapp med samma namn som en standardmapp i en annan mapp. Ett problem som uppstår med den här lösningen är att standardmapparna har olika namn på olika språk, därför måste språkstöd implementeras med en sån lösning. Språkstöd var något som fanns planerat för Briteback men längre fram i tiden, därför var inte en sådan lösning lämplig i dagsläget.

En annan teoretiskt möjlig lösning vore en liknande som den som diskuterades för favoritmappar, alltså att använda fältet utökade egenskaper som finns på meddelanden. I dagsläget är det inte en möjlig lösning eftersom fältet endast finns på meddelanden men i framtida versioner av API:et är det möjligt att det tillkommer och då är det en bättre lösning än den som används nu. Standardmappens namn skulle kunna sparas ner i de utökade egenskaperna första gången man loggar in via Briteback, vid senare inloggningar är det möjligt att genom ett enda anrop hämta alla mappar och läsa ut om en mapp är en standardmapp och vad dess id i Briteback är.

Metod

Metoden bör ha en hög replikerbarhet men reliabiliteten och validiteten är mer osäkra. Samma API-problem bör framkomma i en ny studie eftersom de är baserade på vad som finns implementerat i API:en men poängen kommer med största sannolikhet inte bli samma eftersom mätpunkterna är beroende av personlig kunskap och erfarenet. Även om samma problem hittas så är det inte säkert att samma lösningar tas fram då det till stor del är beroende av applikationens syfte.

Utvecklingsmiljö

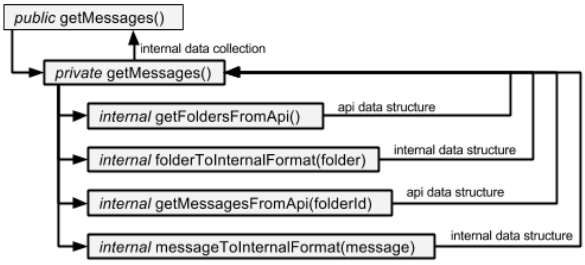
Vikten av att sitta i en utvecklingsmiljö anpassad för uppgiften blev tidigt påtaglig. Både applikationens server och klient kördes på en och samma dator i samband med utvecklingen. Det krävde att datorn hade mycket RAM-minne, vilket tyvärr inte var fallet under den största delen av projektets gång. Mycket tid gick åt till att vänta på att webbsidor och applikationer skulle laddas färdigt eller att starta om datorn när minne tog slut och många dagar avslutades i stor frustration över att tidsplanen blivit lidande som ett resultat. Det var lätt att göra misstag när applikationer inte uppdaterades ordentligt, vilket därmed ledde till än större förseningar.

Implementation

Testdriven utveckling bedrevs under porteringen. Eftersom utgångspunkten för testdriven utveckling är det önskade läget

så passade metoden utmärkt för portering då syftet med portering är att skapa en ny version av något som redan existerar.

Upplägget visade sig vara svårt att genomföra väl, men det var främst på grund av utvecklarens ovana med TDD. Varje test var väldigt begränsat i sin omfattning för att kunna implementera ett anrop i taget, men det innebar att det ofta var väldigt få rader att portera och det var då lätt hänt att den iterativa processen som beskrivs i teorin förbisågs och klassisk utveckling, anropet först och behandlingen av data därefter, skedde istället. Där processen följdes kändes det dock som en lämplig metod, i alla fall för större kodbitar, då strukturen i applikationen var uppbygd med en övergripande funktion som kallade på en funktion som bearbetade data som i sin tur kallade på en eller flera interna funktioner för bland annat själva anropen (se figur 7).



Figur 7: Exempel på funktionsflöde. Returen från samtliga funktioner kan ersättas av mockdata.

Det fanns en tanke om att göra enhetstester för alla funktioner som kördes i samband med respektive Seleniumtest men den idéen övergavs då det inte fanns några namngivna funktioner i koden. Om den vägen hade varit möjlig så hade det blivit ännu tydligare med användandet av mockdata och en iterativ utvecklingsprocess.

Den främsta anledningen att förespråka den valda metoden är Seleniumtesterna. Då fungerande tester skapas innan nu kod skrivs så blir det väldigt tydligt när en portering blivit helt klar, det finns ingen risk att något inte verifieras så länge som testet är korrekt skapat från början.

Dokumentation

Porteringen dokumenterades löpande under utvecklingens gång, dels genom att besvara förutbestämda frågor för varje funktion, dels genom en reflektionsdagbok. Metoden kändes lämplig då det är viktigt att få ner konkreta tankar och idéer så snart som möjligt för att inte glömma något väsentligt. Reflektionen bidrog med mer genomtänkta tankar, det hände vid flera tillfällen att andra idéer och lösningar dök upp efter en stunds distans.

Den dagliga reflektionen var svår att få in en bra rutin för. Det var lättare att bryta för reflektion vid ett naturligt tillfälle, som exempelvis efter att en del av porteringen fallit på plats, än mitt i ett problem. Det hände vid flera tillfällen att reflektionen skrevs på morgonen istället och det fanns både för och nackdelar med det. Om reflektionen skedde i slutet av dagen så att det inte gått någon tid mellan senaste utveckligen och reflektionen så förlorades hela idén med att få distans. Om reflektionen skedde i början av dagen så hade det ibland gått för lång tid sedan delar av den föregående dagens utveckling och de reflektioner som dök upp under dagen hade hunnit glömmas. En möjlig förbättring av metoden vore att ha ett reflektionspass i början av morgonen och ett efter lunch. Då sammanfaller det med ett naturligt avbrott och det går alltid lite tid mellan utveckling och reflektion.

Anteckningarna var väldigt värdefulla för analysen. Även då minnen av att ett problem som hade uppstått fanns kvar så var det ofta som detaljerna hade glömts bort.

Analys

Samtliga problem kategoriserades och bedömdes genom att de fick poäng baserat på hur lång tid problemet tog att lösa och hur bra dokumentation det fanns att tillgå för att lösa problemet. Dessa två mätpunkter valdes för att det är viktigt att veta om en sak är tidskrävande och hurvida det går att planera sin design i förväg med hjälp av tillgänglig dokumentation.

För att få en större spridning hade flera kriterier kunnat användas, ett exempel vore att räkna hur många rader kod som behövdes för varje motsvarande funktion i respektive modul. Kriterier skulle kunna få olika vikt baserat på hur väsentliga de är för sammanhanget.

Outlook Mail REST API vs Gmail REST API

En stor skillnad mellan API:erna är förhållningssättet till meddelanden och konversationer. Gmails API utgår ifrån konversationer som innehåller information om tillhörande meddelanden. Outlook 365:s API fungerar tvärt om och utgår istället från meddelanden som innehåller information om vilken konversation de tillhör. Det gör att det inte nödvändigtvis är möjligt att byta anropen rakt av. Om man vill ha så lik kod som möjligt runt API-anropen så bör man tänka till ordentligt innan man börjar designa sin lösning för att undvika att koden blir för hårt knuten till den ena eller den andra utgångspunkten.

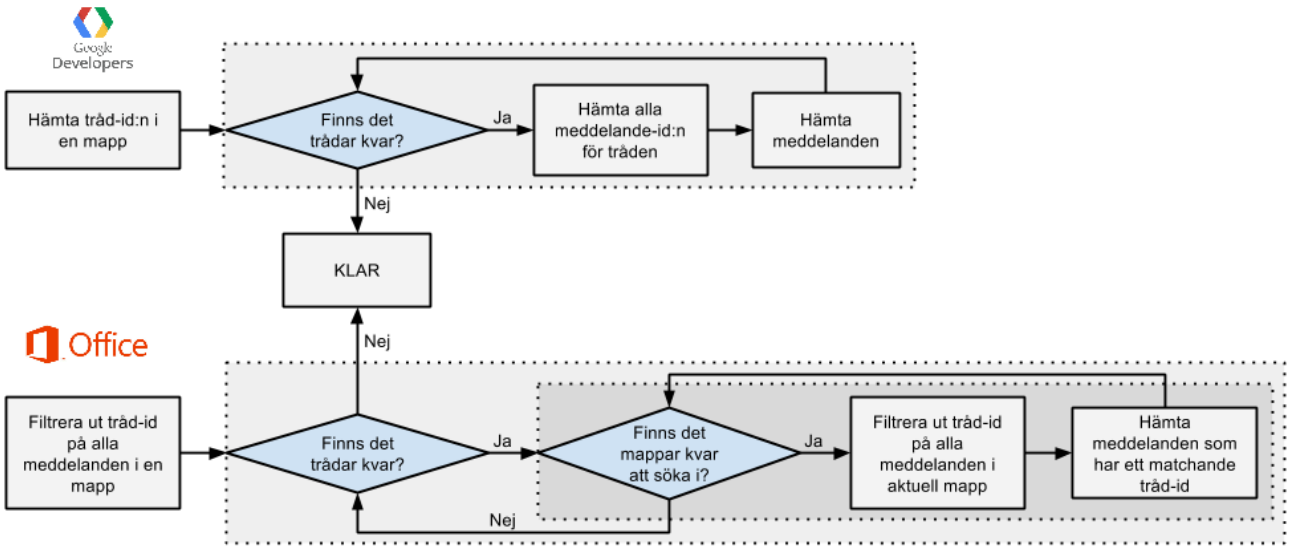
Ett exempel i Briteback är deras hämtning av meddelanden. I Google Apps-modulen hämtar de först alla konversationsid:n i en mapp, därefter hämtar de alla meddelandeid:n för respektive konversation och slutligen hämtar de alla dessa meddelanden. Det blir omständigt att göra samma sak i Office 365-modulen då konversationens id måste läsa ut ifrån ett attribut på ett meddelande. Så först får alla meddelanden i en mapp tas fram, därefter undersöks vilka konversationsid:n dessa har och slutligen söks alla meddelanden i alla mappar igenom för att se var det finns matchande konversationsid:n (se figur 8).

Arbetet i ett vidare sammanhang

Briteback har chansen att förbättra vardagen för många människor som dagligen arbetar i en miljö där de blir utsatta för teknikstress i form av många kommunikationskanaler. Applikationen skulle kunna hjälpa till att återfå en stor del av den tid som går förlorad varje dag. Det i sin tur kan bidra till friskare arbetstagare och arbetsgivare som känner att de får större värde för den lön de betalar.

Källkritik

Det finns många vetenskapliga studier och böcker att finna inom områdena stress, portering, testdriven utveckling och



Figur 8: Flöde för att hämta meddelanden tillhörande trådar i Gmail REST API och Outlook Mail REST API.

molntjänster och flera av författarna till de valda källorna är erkända inom sitt område.

När det gäller tekniska aspekter så har i första hand tillverkarens uppgifter använts. Risk finns då att den informationen är partisk, därför har andra källor använts som stöd för att bekräfta uppgifterna.

För statistik har webbaserade undersökningar främst använts som källor då det är där man hittar färskast data. Det svåra här har varit att utvärdera hur trovärdig källan är. Som hjälp har omdömen om källorna på andra webbsidor och diverse forum använts.

SLUTSATSER

På det stora hela så är Gmail REST API och Outlook Mail REST API kompatibla och det är en indikation på att även resterande REST API:er i Google Apps och Office 365 är det med. Det finns därmed ingen anledning att avråda från en portering. Det saknas dock motsvarighet för flera anrop i de utvärderade API:erna vilket gör det nödvändigt att skapa workarounds för att få samma funktionalitet. Att det inte finns möjlighet att lista alla meddelanden i en mailbox, utan att behöva ange en specifik mapp att titta i upplevs som mycket ineffektivt, framför allt vid arbete med trådar då de tillhörande meddelandena kan ligga i olika mappar.

Office 365 REST API är än så länge endast släppt som en version för förhandsgranskning. Alla, eller åtminstone de flesta, av de problem som upptäckts i API:et under arbetets gång bör bli lösta i framtida versioner.

Framtida arbeten

En jämförelse mellan ett färdigt API och en förhandsgranskning är inte helt rättvis. När en färdig version av Office 365 REST API släpps så vore det intressant att göra om samma utvärdering på nytt för att se om några av problemen kvarstår.

Även om kompabiliteten mellan Gmail REST API och Outlook REST API indikerar en kompabilitet mellan övriga applikationers API:er så vore det lämpligt med en utökad studie vad gäller kompabilitet för kalender och kontakter med mera för att bekräfta antagandet.

För att få en mer övergripande bild över hur kompatibla REST API:er är så skulle fler produkter behöva jämföras. Det kan antas att de är snarlika med tanke på REST-regelverket, därför vore det även intressant att jämföra med ett icke-REST API som exempelvis Microsoft Exchange.

REFERENSER

1. Apps Run The Cloud. 2014. *Worldwide Cloud Applications Market Forecast 2014-2018*. Technical Report. 27 pages. https://www.appsrunthecloud.com/app/webroot/img/pdfdownlaod/WorldwideCloudApplicationsMarketForecast2014-2018.pdf
2. Atlassian. 2015. JIRA Agile. (2015). Hämtad 23 februari, 2015 från https://www.atlassian.com/software/jira/agile.
3. Ramakrishna Ayyagari, Varun Grover, and Russell Purvis. 2011. Technostress: Technological Antecedents and Implications. *MIS Quarterly* 35, 4 (2011), 831–858. DOI:http://dx.doi.org/10.1109/TSE.2013.12
4. Kent Beck. 2003. *Test-Driven Development By Example*. Vol. 2. 176 pages. DOI: http://dx.doi.org/10.5381/jot.2003.2.2.r1
5. Fatna Belqasmi, Roch H. Glitho, and Fu Chunyan. 2011. RESTful Web Services for Service Provisioning in Next-Generation Networks: A Survey. *IEEE Communications Magazine* 49, 12 (2011), 66 – 73. DOI: http://dx.doi.org/10.1109/MCOM.2011.6094008]

6. Bitglass. 2014. *Cloud Adoption Report*. Technical Report. 7 pages. http://pages.bitglass.com/five_essentials_cloud_adoption.html
7. Ralph Bohnet and Gerard Meszaros. 2005. Test-driven porting. In *Proc. Agile Conference 2005*. IEEE Computer Society, 259–266. DOI: http://dx.doi.org/10.1109/ADC.2005.46
8. Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. 2004. How a Good Software Practice Thwarts Collaboration: The Multiple Roles of APIs in Software Development. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (SIGSOFT '04/FSE-12)*. ACM, 221–230. DOI: http://dx.doi.org/10.1145/1029894.1029925
9. Google Developers. 2015. Google Apps Platform. (2015). Hämtad 16 februari, 2015 från https://developers.google.com/google-apps/.
10. Walid Maalej and Martin P. Robillard. 2013. Patterns of knowledge in API reference documentation. *IEEE Transactions on Software Engineering* 39 (2013), 1264–1282. DOI: http://dx.doi.org/10.1109/TSE.2013.12
11. Microsoft Office Dev Center. 2015. Office 365 API reference. (2015). Hämtad 24 april, 2015 från https://msdn.microsoft.com/office/office365/APi/api-catalog.
12. Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. 2012. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering* 17, 6 (2012), 703–737. DOI: http://dx.doi.org/10.1007/s10664-011-9186-4
13. Susan L. Murray and Zafar Khan. 2014. Impact of interruptions on white collar workers. *Engineering Management Journal* 26, 4 (2014), 23–28. DOI: http://dx.doi.org/10.1080/10429247.2014.11432025
14. Deborah Nolan and Duncan Temple Lang. 2014. REST-based Web Services. In *XML and Web Technologies for Data Sciences with R*. Springer New York, Chapter 10, 339–379. DOI: http://dx.doi.org/10.1007/978-1-4614-7900-0_10
15. Office 365 Team. 2014. Office 365—now available in 140 markets. (2014). Hämtad 18 mars, 2015 från http://blogs.office.com/2014/11/03/office-365-now-available-140-markets/.
16. Sandra Ohly, Sabine Sonnentag, Cornelia Niessen, and Dieter Zapf. 2010. Diary Studies in Organizational Research: An Introduction and Some Practical Recommendations. *Journal of Personnel Psychology* 9, 2 (2010), 79–93. DOI: http://dx.doi.org/10.1027/1866-5888/a000009
17. Kai Riemer and Frank FrößÎer. 2007. Introducing Real-Time Collaboration Systems: Development of a Conceptual Scheme and Research Directions. *Communications of the Association for Information Systems* 20 (2007), 204–225.
18. Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *Software, IEEE* 26, 6 (2009), 27–34. DOI: http://dx.doi.org/10.1109/MS.2009.193
19. Selenium. 2015. Web Browser Automation. (2015). Hämtad 23 februari, 2015 från http://www.seleniumhq.org/.
20. Barrie Sosinsky. 2010. *Cloud Computing Bible*. John Wiley & Sons. DOI: http://dx.doi.org/10.1002/9781118255674
21. Three Rivers Institute. 2008. Who is Kent Beck. (2008). Hämtad 23 februari, 2015 från http://www.threeriversinstitute.org/KentBeck.htm.
22. Toggl. 2015. About Toggl. (2015). Hämtad 24 februari, 2015 från https://www.toggl.com/about.
23. William Van Winkle. 2014. Cloud Office Suites: Comparison of Online Solutions - Top 3 Solutions Compared. (28 January 2014). Hämtad 17 februari, 2015 från http://www.tomsitpro.com/articles/cloud-office-suites,2-690.html.