CHALMERS

Modulär hantering av produktdata

En mikrotjänst i programmeringsspråket Elm för en e-handelswebbapplikation

Examensarbete inom högskoleingenjörsprogrammet Datateknik

MARTIN BERTNSSON
NARIMAUN NOVAK

INSTITUTIONEN FÖR DATA-OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA

Göteborg, Sverige 2025
www.chalmers.se

Modulär hantering av produktdata: En mikrotjänst i programmeringsspråket Elm för en e-handelswebbapplikation

Martin Berntsson, Narimaun Novak

June 3, 2025

Examensarbete 2025

Modulär hantering av produktdata: En mikrotjänst i programmeringsspråket Elm för en e-handelswebbapplikation

Martin Berntsson, Narimaun Novak

Institutionen för Data- och informationsteknik
Chalmers tekniska högskola
Göteborg 2025

Innehållsförteckning

1 Inledning **1**
1.1 Syfte . . . 1
1.2 Mål . . . 2
1.3 Avgränsningar . . . 2

2 Teknisk bakgrund **3**
2.1 E-handel . . . 3
2.2 Mikrotjänstarkitekturer . . . 3
2.3 Funktionella programmeringsspråk . . . 4
2.4 Programmeringsspråket Elm . . . 4
2.5 Web workers . . . 6
2.6 Google Sheets API . . . 6

3 Metod **7**
3.1 Etiska aspekter . . . 7
3.2 Design och implementering . . . 7
3.3 Continuous integration och Continuous deployment . . . 8
3.3.1 Fördelar med CI/CD paradigmet . . . 8
3.3.2 Genomförande av projektet . . . 8
3.4 Agil metodologi . . . 8
3.4.1 Genomförande av projektet . . . 9

4 Systemkonstruktion **10**
4.1 Modulerna . . . 11
4.2 Inhämtning av data . . . 12
4.2.1 Get-request . . . 13
4.2.2 Get Spreadsheet . . . 13
4.3 Utdata och versionshistorik . . . 15
4.3.1 Konstruktion och hashing av utdata . . . 15
4.3.2 Jämförelse med tidigare sparad data och versionshistorik . . . 15
4.3.3 Sparande av data på disk . . . 16
4.4 Uppdatering av nuvarande version . . . 17

5 Resultat och diskussion **18**
5.1 Modularitet . . . 18
5.2 Återkoppling av fel i kunddata . . . 19
5.3 Arbetsprocess . . . 19
5.4 Hantering av ett stort antal filer . . . 19
5.5 Versionshistorik . . . 19
5.6 Förslag till förbättring . . . 20

6 Slutsats **21**

Sammanfattning

E-handel har växt fram i samband med internets utveckling från tidigt 1990-tal och har ökat kraftigt i Sverige under de senaste åren. En utmaning som medföljer e-handeln är den stora mängd produktdata som behöver hanteras och organiseras för att slutligen presenteras till potentiella kunder. Webbhuset I Sverige AB tillhandahåller e-handelslösningar som är skräddarsydda efter varje kunds behov och detta projekt genomfördes i samband med dem. Det förekommer stor variation i den data som olika företag behöver hantera och presentera för sina kunder. På grund av detta har Webbhuset behövt göra en ny implementation av logik för behandling och cache-lagring av varje enskild kunds data vilket har lett till mycket upprepat arbete som borde gå att effektivisera. Med detta som bakgrund har författarna tillsammans med Webbhuset tagit fram en modulär mikrotjänst för hämtning, konvertering och cache-lagring av kunders produktdata. Mikrotjänsten implementerades i det reaktiva och funktionella programmeringsspråket Elm som är specialiserat för att ta fram webbapplikationer.

Den framtagna mikrotjänsten integrerades i Webbhusets befintliga arkitektur och klarade av att läsa in, bearbeta och felhantera genererad testdata från ett kalkylark i Google Spreadsheets. Datan cache-lagrades sedan enligt Webbhusets interna format och funktionalitet för versionshistorik implementerades utöver de ursprungliga målen. Mikrotjänsten behöver anpassas när det kommer till inläsning och bearbetning av kunddatan medan cache-lagring och versionshistorik är helt modulär. Den slutgiltiga mikrotjänsten uppfyllde de mål som sattes upp vid projektets start men författarna anser att vidare utveckling behövs för att förbättra återkoppling till användare av mikrotjänsten och för en mer generaliserad hantering av kunddata.

Kapitel 1

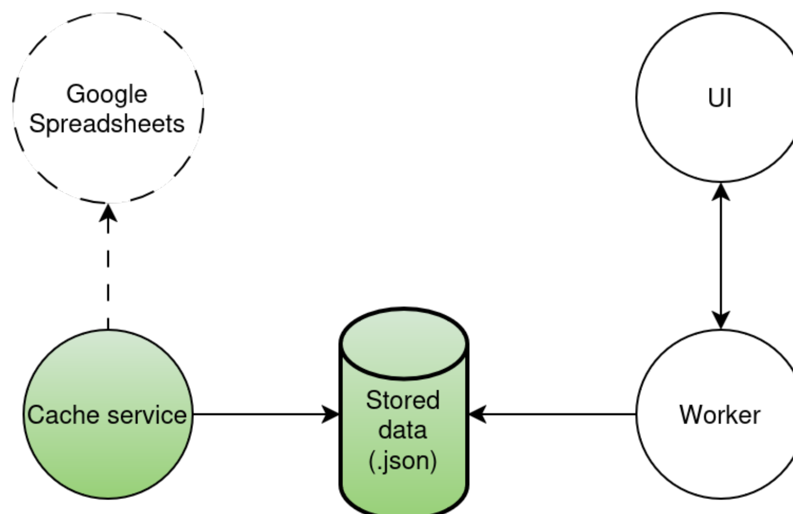
Inledning

Denna rapport undersöker huruvida analys och lagring av produktdata inom e-handel kan underlättas genom skapandet av en mikrotjänst vars uppgift är att hantera en cache över produktdata. Mikrotjänsten blir ett tillskott till en redan befintlig mikrotjänstarkitektur och ska konstrueras med hjälp av det funktionella programmeringsspråket Elm. Projektet genomförs i samarbete med Webbhuset i Sverige AB (Webbhuset) som inriktar sig på att leverera skräddarsydda e-handelslösningar till företag inom olika branscher [1].

1.1 Syfte

Varje kund som Webbhuset arbetar med har produktdata som är unikt utformad. På grund av detta har Webbhuset behövt göra en ny implementation av logik för behandling och cache-lagring av varje enskild kunds data vilket har lett till mycket upprepat arbete som borde gå att effektivisera. Med detta som bakgrund ska författarna tillsammans med Webbhuset ta fram en mikrotjänst för hämtning, konvertering och cache-lagring av kunders produktdata.

Den mikrotjänst som ska tas fram namnges ”Cache service” och ska vara en del av Webbhusets befintliga mikrotjänstarkitektur. Ett exempelflöde för hur Cache service ska fungera illustreras i figur 1.1. Pilarna i figuren illustrerar kommunikation mellan komponenter. De grönmarkerade komponenterna (Cache service och filer sparade i databasen) ska designas och implementeras som en del av projektet medan komponenterna ”Worker” och ”UI” är befintliga mikrotjänster i Webbhusets arkitektur. Den streckade komponenten ”Google Spreadsheets” representerar produktdata i ett kalkylark som har skapats av en användare hos kunden, exempelvis en administratör.



Figur 1.1: Övergripande struktur av Cache service och relaterade mikrotjänster.

Flödet för en cachelagring av produktdata ska starta genom att en användare som vill initiera en ny cachelagring klickar på en knapp tillhörande kalkylarket. Detta ska leda till att Cache service hämtar produktdatan, bearbetar denna och slutligen lagrar den som JSON filer i cachen. Efter detta ska en annan mikrotjänst, i detta exempel den som heter Worker, kunna serva den lagrade datan till Webbhusets andra mikrotjänster efter behov, denna del är dock ingenting som ska ingå i projektets omfattning. Den data som sparats i cachen behöver alltså vara i ett format som kan användas av övriga mikrotjänster i Webbhusets e-handelslösningar.

Cache service ska vara anpassningsbar för att underlätta hanteringen av olika utformad data från de olika företag som Webbhuset jobbar mot och behöver därför vara modulär.

Utöver ovannämnd funktionalitet kommer projektet i mån av tid att implementera en lösning för versionshistorik av den hanterade data som sparas hos Webbhuset genom mikrotjänsten. Detta för att undvika onödig dataöverföring och sparande av oförändrad data.

1.2 Mål

Författarna kommer att designa och implementera en prototyp av en mikrotjänst som ska cachelagra kunddata i programmeringsspråket Elm. En användare ska kunna skicka en förfrågan om att cache-lagra kunddata till cache-servicen. Datan ska vara lagrad i ett Google Sheets kalkylark och kunna läsas in via Google sheets API. Efter hämtning ska datan kunna parsas och felsökas och rapportera tillbaka eventuella fel till kalkylarket. Datan ska sedan skrivas ut som JSON-filer enligt ett format som används av Webbhuset i andra projekt. Slutligen ska användaren som efterfrågat cache-lagringen få ett svar om skrivningen lyckats eller inte. Prototypen ska fungera med testdata som erhålls från Webbhuset och som efterliknar autentisk kunddata men i en mindre omfattning. Prototypen ska kunna återanvändas i Webbhusets olika projekt.

1.3 Avgränsningar

På grund av projektets begränsade omfattning kommer författarna att implementera en lösning för den typ av data som förekommer i Webbhusets testmiljö. Utveckling av versionshistoriksfunktionalitet kommer även att begränsas beroende på hur lång tid av projektet som kvarstår.

Kapitel 2

Teknisk bakgrund

2.1 E-handel

Konceptet av e-handel har växt fram i samband med internets utveckling från tidigt 1990-tal och har ökat kraftigt i Sverige under de senaste åren. År 2022 utgjordes 17 procent av Sveriges totala detaljhandel av e-handel [2]. E-handeln ser mycket annorlunda ut idag jämfört med när konceptet först var nytt. Till en början använde företag de nya möjligheterna internet förde med sig till att sprida information och marknadsföra sina produkter vilket ledde till ett utökat intresse för att utveckla konceptet vidare. Detta resulterade sedan till ytterligare möjligheter för företag att sälja sina produkter direkt över internet vilket idag även innefattar digitala produkter [3].

En utmaning som medföljer e-handeln är den stora mängd produktdata som behöver hanteras och organiseras för att slutligen presenteras till potentiella kunder. Så kallade E-kataloger är ett sätt att försöka samla ihop denna data och kan vanligen innehålla tusentals produkter [4]. Alla produkters tillhörande produktdata behöver analyseras och lagras på ett effektivt sätt vid skapande av en e-katalog eller webbshop för att underhåll av katalogen ska vara hanterbart. Eftersom produkternas egenskaper kan variera kraftigt från företag till företag, både när det kommer till kvalité, omfattning och format, är det viktigt att lagra sin data på ett sätt som underlättar indexering och sökning [5].

En annan utmaning när det kommer till att katalogisera ett företags produkter är att tillverkaren av en produkt ofta är intresserad av andra produktattribut jämfört med en distributör. Exempelvis kan tillverkaren lägga mer vikt vid material och processer involverade vid tillverkningen av en produkt medan distributören ser annan data som är mer relevant för potentiella köpare av produkten som mest betydelsefulla. Klassificeringen av produkter är därför centralt för att bygga en användbar e-katalog och behöver ta hänsyn till alla parters behov [4].

2.2 Mikrotjänstarkitekturer

I takt med att webbaserade applikationer och tjänster har blivit mer komplicerade samtidigt som behovet av skalbarhet har kvarstått så har industrin övergått från att främst använda monolitiska systemarkitekturer till att istället ge plats åt mikrotjänstbaserade system [6]. En mikrotjänstbaserad arkitektur innebär att strukturen av en applikation bryts ner till mindre beståndsdelar eller tjänster som kan köras självständigt och kommunicera med andra tjänster genom enkla protokoll [7].

Det finns flera fördelar med att använda en mikrotjänstarkitektur för ett system då varje mikrotjänst är tydligt avgränsad från de övriga mikrotjänsterna i systemet. Detta ger systemet en ökad flexibilitet då varje mikrotjänst kan lanseras och hanteras oberoende från andra moduler. En annan fördel är att olika teknologi kan användas för olika mikrotjänster utefter vilka behov tjänsten har. För en mikrotjänst kan ett särskilt programspråk passa bra medan en annan tjänst kräver andra egenskaper och därför kan skrivas i ett annat programspråk eller använda andra programbibliotek [8].

Däremot menar Fowler [8] att det även finns en rad utmaningar med att använda en mikrotjänstarkitektur. Eftersom mikrotjänster använder ett distribuerat system så tillkommer en ökad grad av komplexitet när det kommer till att behålla en god prestanda och tillförlitlighet.

Ytterligare problem som kan uppstå vid användning av mikrotjänster är seghet i systemet eller logiska fel till följd av synkroniseringsproblem. Dessa problem kan förekomma eftersom de decentraliserade mikrotjänsterna i ett system hanterar data var för sig vilket kan leda till att en tjänst blir beroende och behöver vänta på en annan tjänst eller i värsta fall agerar på förlegad information. Dessa fel tenderar att vara svåra att felsöka i efterhand då tjänsterna kan ha hunnit synkroniseras när felsökningen påbörjas. Det är därför viktigt att försöka förhindra att dessa problem kan uppstå vid design och implementation av systemet. Fowler lyfter även att ett system som består av hundratals mindre mikrotjänster kan vara svårare att använda och hantera i praktiken eftersom varje mikrotjänst drivs och förändras individuellt. Arbetssättet ”Continuous Delivery” förespråkas därför av Fowler som ett viktigt verktyg för att kunna hantera och uppdatera de många tjänster som ingår i ett mikrotjänstbaserat system [8].

2.3 Funktionella programmeringsspråk

Funktionella språk som Elm och Haskell har länge kompletterat andra paradigmer inom programmering. Men varför just dessa språk? Anledningarna är många, men en av de främsta är renheten (“purity”) hos deras funktioner [9].

En funktion är ren om samma indata alltid ger samma utdata. Omgivningen kan alltså inte påverka funktionen, vilket innebär att resultatet alltid blir detsamma för identiska indatavärden. En ren funktion saknar dessutom sidoeffekter, såsom diskåtkomst eller nätverksförfrågningar. Denna egenskap minskar avsevärt behovet av bugghantering i Elm och Haskell, eftersom deras rena funktioner inte påverkar något annat än sig själva.

Visserligen går det att skriva rena funktioner även i språk som Python och Java, men då förloras ofta den koncisa syntaxen och de statiska kompilatorgarantier som säkerställer att en funktion verkligen är ren [9].

2.4 Programmeringsspråket Elm

Ytterligare en lösning för hantering av data i webbapplikationer kan väcka skepsis hos vissa, särskilt när den påstås förenkla själva utvecklingen. Men i jämförelse med andra lösningar, till exempel Angular, Vue eller React, är Elm funktionellt. Det faktum att alla värden är oföränderliga, att funktioner inte har några sidoeffekter och att språket är statiskt typat, gör att många fel fångas vid kompilering istället för under körning [10].

Om man blandar Haskell (den funktionella delen), TypeScript (som transpileras till JavaScript), Python (enkelt men med strikt syntax) och till slut Rust (med vänliga felmeddelanden) kan man säga att man får Elm [10].

Elm är ett funktionellt reaktivt programmeringsspråk (FRP) som syftar till att förenkla skapandet av responsiva grafiska användargränssnitt (GUI) och riktar sig specifikt mot webbapplikationers GUI. FRP fungerar genom att vara signalberoende. Detta innebär att språket ständigt lyssnar på signaler och utför beräkningar baserade på de inkomna signalerna. Problemet med många FRP-språk är att semantiken antar att signaler ändras kontinuerligt, vilket leder till att beräkningar sker konstant. I verkligheten ändras signaler sällan, vilket resulterar i onödiga beräkningar och slöseri med resurser. Elm antar istället att alla signaler är diskreta och använder detta paradigm för att upptäcka förändringar i de inkommande signalerna [11].

Allt som skrivs i Elm översätts till JavaScript. För att detta ska vara möjligt används en Elm-till-JavaScript-kompilator, skriven i Haskell och bestående av cirka 2700 rader kod. Detta medför att det tar längre tid att kompilera ett Elm-program jämfört med ett handskrivet JavaScript-program, men kompilatorn är tillräckligt effektiv för att hantera alla Elm-program oavsett hur stor kodbasen är [11].

Sammanfattningsvis är Elm ett praktiskt asynkront programmeringsspråk som låter programmeraren bestämma när en signal bör användas i beräkningarna med andra ord, vad som räknas som

en ändring i signaler. Genom att kombinera asynkron FRP med funktionell utformning av grafiska element förenklar Elm data hanteringen som körs i bakgrunden av webbapplikationer [11].

Frågan ställs ofta om Elm inte längre används och om det har blivit inaktuellt, men det finns inte riktigt bevis för detta. Elm är byggt för att hålla genom tiderna tack vare:

Stabilitet och långsiktighet: Elm fungerar i sitt eget ekosystem, vilket gör att det kommer vara effektivt under lång tid. En bra jämförelse av Elms utvecklingscykel är med C-språket, som också har långa intervall mellan större versioner. Sällsynta uppdateringar innebär inte att språket är inaktuellt.

Hög paketkvalitet: Även om Elm har ett betydligt mindre paketbibliotek jämfört med exempelvis npm, håller Elm-paketen en hög standard med strikt semantisk versionshantering som säkerställer kompatibilitet.

Lätt att lära sig: Elm är också lätt att lära sig, vilket möjliggör intern utbildning av utvecklare och därmed minskar oron kring tillgången på kompetent personal.

Elm är mycket stabilt, den enda större kraschen inträffade 2018 med utgåva 0.19. Språket är också förutsägbart. Eftersom det är rent funktionellt, om det återanvänder en utdata i produktionen, kommer samma utdata att produceras i alla webbläsare hos användaren [12].

Men såklart finns det nackdelar med Elm också. Till exempel är integrationen mellan Elm och externa JavaScript-bibliotek svårare. Allt som hanterar och manipulerar DOM tenderar att krocka med Elm. ML-syntaxen som används i Elm kan också ibland vara svår att förstå för utvecklare som är vana vid imperativa programmeringsspråk som Java och Python, vilka är mer konventionella. Eftersom Elm inte är så konventionellt i industrin, kan det även begränsa anställbarheten. Både juniora och seniora utvecklare väljer vilka språk de ska lära sig baserat på hur eftertraktade de är, och vilka språk som gör att det blir lättare att få anställning i [12].

2.5 Web workers

Web workers, som namnet antyder, används för att utföra tunga beräkningar som kan behövas i backgrunden av en webbapplikation. I varje applikation finns en huvudtråd som kör alla moduler och sätter ihop dem. Om applikationens huvudsakliga syfte är att utföra tunga beräkningar, kommer dessa om de körs på huvudtråden att göra sidan icke-responsiv under beräkningarna [13].

Av denna anledning används web workers för att hantera tyngre uppgifter. När man skapar en web worker startar man en extra tråd som kör parallellt med huvudtråden och väntar på uppgifter därifrån [14].

Web workers används flitigt inom industrin, vilket har lett till att de stöds i olika ramverk, till exempel JavaScript, React, Angular och Elm. Sammanfattningsvis är syftet med web workers att flytta tunga beräkningar från huvudtråden, utföra dem utan att påverka huvudprogrammets responsivitet och returnera resultatet när beräkningarna är klara [14].

2.6 Google Sheets API

Google Sheets API används flitigt i industrin för att automatisera arbetsflöden, möjliggöra realtidsdataflöden och öka produktiviteten. Detta API är RESTful som gör det möjligt att läsa in data från ett kalkylark. Med detta API går det att skapa kalkylark, samt att läsa från och skriva till olika blad samt uppdatera formatering. Trots sina styrkor har Google Sheets API också vissa begränsningar, exempelvis kvotgränser och en maximal cellgräns på två miljoner celler per kalkylblad [15].

Data hanteras via ett primärt objekt som kallas Spreadsheet. Spreadsheet-objektet innehåller ett spreadsheet-ID och alla blad som det specifika kalkylarket har. Ett blad innehåller ett sheet-ID och en sheet title bland sina egenskaper. Själva datan hanteras i enheter som kallas celler. Celler har inga ID:n, istället identifieras de med unika rad- och kolumn koordinater. I Google sheets API, för att specificera vilka celler som ska importeras eller manipuleras (till exempel genom att skrivas till), används en så kallad "range" som anger vilka rader och kolumner som operationen berör. Rader benämns med siffror som börjar med 1, och kolumner benämns med bokstäver som börjar med A.
För att skriva tillbaka till celler, till exempel om det finns felaktiga eller korrupta data, kan man använda Google Apps Script. Apps Script är ett molnbaserat programspråk baserat på JavaScript. Det använder JavaScript-syntax och har flera inbyggda bibliotek för att interagera med kalkylblad. En viktig funktion i det här projektet har varit möjligheten att köra skriptet med tidsstyrda utlösare, till exempel dagligen eller varje timme. Det gör det möjligt att hämta data från ett kalkylblad regelbundet och skriva till de celler som innehåller felaktiga värden [16].

Google Sheets API är gratis att använda, och för att kunna hantera kalkylark med det behöver man ha tillgång till Google Cloud Platform (GCP) för att kunna skapa inloggningsuppgifter som senare används för att ansluta till API:et [17].

Kapitel 3

Metod

3.1 Etiska aspekter

Under projektets genomförande användes endast artificiell testdata för representation av ett företags potentiella produktdata. Detta inkluderade inga känsliga uppgifter eller annan autentisk personinformation.

Den testdata som användes i projektet konstruerades av den industriella handledaren hos Webbhuset med syfte att efterlikna strukturen hos den kunddata som används i många av Webbhusets projekt. I detta projekt bestod testdatan av ett kalkylark från Google Sheets där produkter, kategorier, prisinformation och annan relevant data lagrades i separata blad. Under projektets gång förväntades författarna utöka mängden data för att kunna testa prestandan hos den framtagna mikrotjänsten men strukturen och utformningen av datan var oförändrad. Vid framtida användning av den framtagna mikrotjänsten kommer dock sannolikt behandling av känslig användarinformation att förekomma. Det var därför av vikt att mikrotjänsten behandlade aktuell data med integritet och säkerhet i åtanke. För att säkerställa att känslig data hanteras korrekt följer Webbhuset relevanta ISO-standarder för GDPR [18].

3.2 Design och implementering

Projektet inleddes med att författarna spenderade två veckor med instudering av programmeringsspråket Elm eftersom ingen av författarna tidigare hade arbetat med detta. Detta följdes av ytterligare en veckas förberedelser då författarna satte sig in i Webbhusets befintliga systemarkitektur och bekantade sig med andra mikrotjänster som kom att samspela med den nya mikrotjänsten projektet skulle ta fram.

Mikrotjänsten som implementerades namngavs "Cache service" och utformades enligt en "top-down-design", det vill säga att författarna först formulerade det övergripande huvudproblemet och sedan successivt bröt ner detta i mindre och mindre beståndsdelar. Efter att de viktigaste komponenterna som utgör mikrotjänsten hade identifierats började författarna att konstruera motsvarande moduler och typer i Elm. Elms statiska typsystem och låga förekomst av sidoeffekter utnyttjades till att först verifiera att alla komponenter i mikrotjänsten kunde interagera med varandra som planerat. Först efter att detta var säkerställt implementerades funktionaliteten för interaktion med övriga mikrotjänster samt för kommunikation utanför systemet.

Utformningen av mikrotjänstens beståndsdelar evaluerades kontinuerligt mot övriga mikrotjänster som ingick i samma system. Detta med målet att återanvända så mycket som möjligt av de redan befintliga typer och funktionalitet som skulle delas mellan olika mikrotjänster.

3.3 Continuous integration och Continuous deployment

I det här projektet planerade vi att utföra arbetet på samma sätt som medarbetarna arbetade på Webbhuset. Arbetet genomfördes enligt metodiken Continuous Integration och Continuous Deployment (CI/CD).

Continuous Integration, eller på svenska kontinuerlig integration, är ett arbetssätt där utvecklarna utvecklar och testar små kodbitar regelbundet, så snart de är klara. När alla tester passerar sammanfogas koden till ett versionshanterat kodlager, vilket underlättar för utvecklarna att återgå till tidigare versioner vid behov. Det viktiga med kontinuerlig integration är att utvecklarna inte väntar på att en större mängd kod ska skrivas innan tester genomförs, istället testas mindre kodbitar kontinuerligt [19].

Continuous Deployment, eller på svenska kontinuerlig driftsättning, innebär att ha en fungerande produkt som utvecklarna alltid kan flytta till automatiserade testningsmiljöer eller visa upp för användare och kunder även om det eventuellt bara är en Minimal Viable Product. Med andra ord resulterar kontinuerlig driftsättning i en produkt som kan distribueras till olika miljöer så snart som möjligt [20].

3.3.1 Fördelar med CI/CD paradigmet

Bättre kvalitet på lösningar och kod: Att skriva mindre kodbitar gör det lättare att debugga och hitta fel. Detta leder till att fel och buggar upptäcks mycket tidigare, vilket höjer kodens kvalitet.

Tydliga riktlinjer och processer: CI/CD paradigmet är mycket tydligt med vad utvecklarna behöver tänka på i varje steg, vilket underlättar planeringen av vad som exakt behöver göras.

Snabbare återkoppling: Eftersom paradigmet bygger på att det alltid finns en produkt att visa, får utvecklarna återkoppling från användare eller kunder mycket snabbare.

3.3.2 Genomförande av projektet

Vi planerade att genomföra vårt projekt enligt CI/CD-paradigmet. Detta gjordes genom att vi delade upp arbetet mellan oss, försökte lösa problemen och sedan testade de lösningarna i små delmoment. Varje vecka hade vi ett tekniskt möte med vår industrihandledare där vi utvärderade föregående veckas arbete och diskuterade vilka problem som måste åtgärdas med tanke på att ta korta steg. Det viktiga var att problemen ska vara tillräckligt små för att kunna lösas inom en vecka, vilket möjliggjorde en kontinuerlig integration. När ett problem hade lösts skapades en merge request till repot, som visades för vår handledare i rollen som användare/kund. Eftersom uppgifterna tilldelades varje vecka och vi försökte lösa dem innan veckans slut, underlättade detta även felhanteringen.

3.4 Agil metodologi

Vi valde också att jobba agilt i detta projekt. Att jobba agilt innebär att man, istället för att arbeta rigidt och linjärt, arbetar i kortare cykler som kallas “sprints”. Det finns fyra viktiga kärnvärderingar med agilt arbetssätt: [21]

1. **Individ och mänskliga interaktioner framför processer:**
   Detta fokuserar på hur viktigt det är att upprätthålla kommunikationen med klienten. Det allra viktigaste för teamet är att klientens krav uppfylls. Eftersom det är klienten som driver projektet är denne naturligtvis viktigare än verktyg och processer.

2. **Väl fungerade mjukvara framför omfattande dokumentation:**
   Innan agilt arbetssätt togs i bruk inom industrin lades för mycket tid på att dokumentera – till exempel teknisk dokumentation, specifikationer, osv. Detta ledde till förseningar i

utvecklingen av projektet. Inom agilt arbetssätt är det viktigaste att leverera till användaren och klienten, vilket effektiviserar både dokumentationen och leveransen.

3. **Kundsamarbete framför kontraktsförhandling:**
Tidigare skedde förhandlingar med kunden endast i början och i slutet av projektet, vilket inte var särskilt effektivt. Med ett agilt arbetssätt är kunden involverad i hela processen från början till slut. Detta leder till att den färdigställda produkten ligger mycket närmare kundens önskemål.

4. **Att kunna anpassa sig till förändringar framför att följa en plan:**
Att kunna anpassa sig till förändringar är viktigt i ett agilt arbetssätt. Det viktiga med ett team som arbetar agilt är att förstå att förhållanden kan ändras och att teamet måste kunna anpassa sig därefter. Det tidigare arbetssättet såg förändringar som kostsamma och något som måste undvikas, men tyvärr fungerar inte produktutveckling på det sättet [22].

3.4.1 Genomförande av projektet

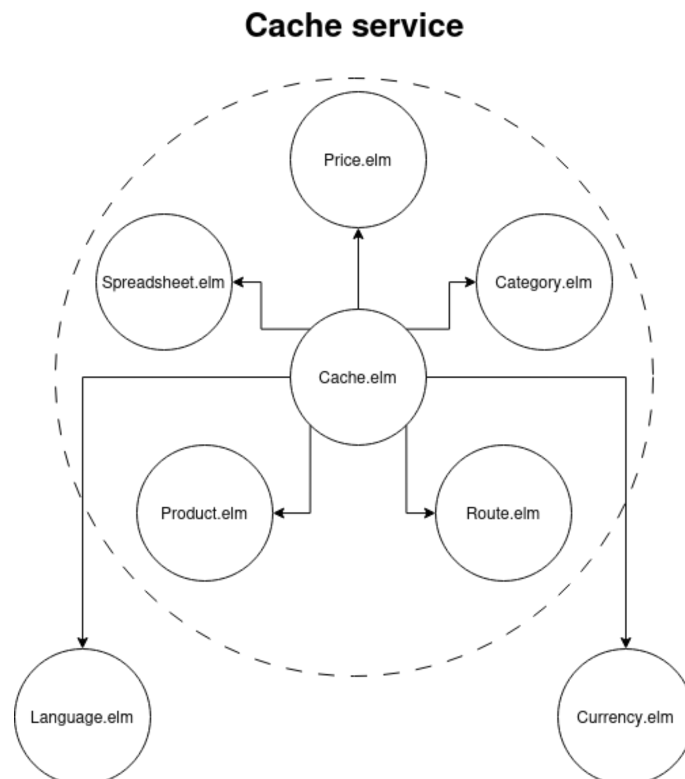
Som sagt, vi planerade att arbeta agilt i detta projekt. Först och främst var vår utveckling av produkten fokuserad på kundens behov och önskemål. I detta fall var kunden vår industrihandledare, och vi arbetade väldigt nära och agilt med honom. Vi hade också ett veckovis planerat arbete samt ett möte varje fredag med industrihandledaren, vilket fungerade likt “sprints” och det dagliga scrum mötet i teamet. Vi var dessutom mottagliga för förändringar från kunden eller andra justeringar som kunde uppstå under produktutvecklingen, och teamet kanske behövde anpassa sig därefter.

Kapitel 4

Systemkonstruktion

Mikrotjänsten "Cache service" har implementerats och integrerats i webbhusets mikrotjänstarkitektur enligt strukturen i figur 1.1. Cache servicens huvuduppgift är att läsa in produktdata från ett Google Spreadsheet, bearbeta denna och skriva ut datan som en cache i Webbhusets databaser där andra mikrotjänster i systemet sedan kan läsa av datan. Vid bearbetningen kontrolleras även datan för fel och vid förekomst av dessa avbryts cachningen och felmeddelanden skrivs tillbaka till det Google Spreadsheet datan hämtats ifrån. Om produktdatan inte innehåller några fel konverteras den till specifika typer för utdata som följer en önskad struktur för att slutligen sparas i JSON-format i Webbhusets databaser. Den sparade datan i databasen används bland annat av en Web Worker som presenterar den till användare genom någon form av UI, exempelvis ett företags webbshop.

Cache service består av flera Elm-moduler. Cache.elm innehåller den huvudsakliga logiken för hämtning, verifiering och sparande av data och är beroende av flera andra moduler vilket beskrivs i figur 4.1. Varje cirkel med heldragen linje representerar en Elm-modul och pilarna visar beroenden mellan moduler. De moduler som ligger inom den streckade cirkeln används endast av den framtagna mikrotjänsten medan de moduler som ligger utanför cirkeln används globalt i Webbhusets arkitektur.



Figur 4.1: Beroendediagram över Cache Service.

4.1 Modulerna

Cache.elm: Modulen Cache.elm är, som tidigare nämnt, huvudkomponenten i mikrotjänsten som hanterar förfrågningar och håller koll på servicens tillstånd. Den hanterar inhämtning av data genom en beständig process som lyssnar på alla inkommande förfrågningar och anropar motsvarande metoder. Cache.elm är uppbyggd som en web worker, vilket innebär att alla beräkningar och tungt arbete sker här. Arbetet består i att hämta produktdata från ett Google Spreadsheet genom att anropa API:t, parsa den inhämtade datan samt konvertera den till korrekt utdata. Cache.elm-modulen står i centrum av den implementerade lösningen och har tillgång till alla nödvändiga typer som krävs för att genomföra arbetet.

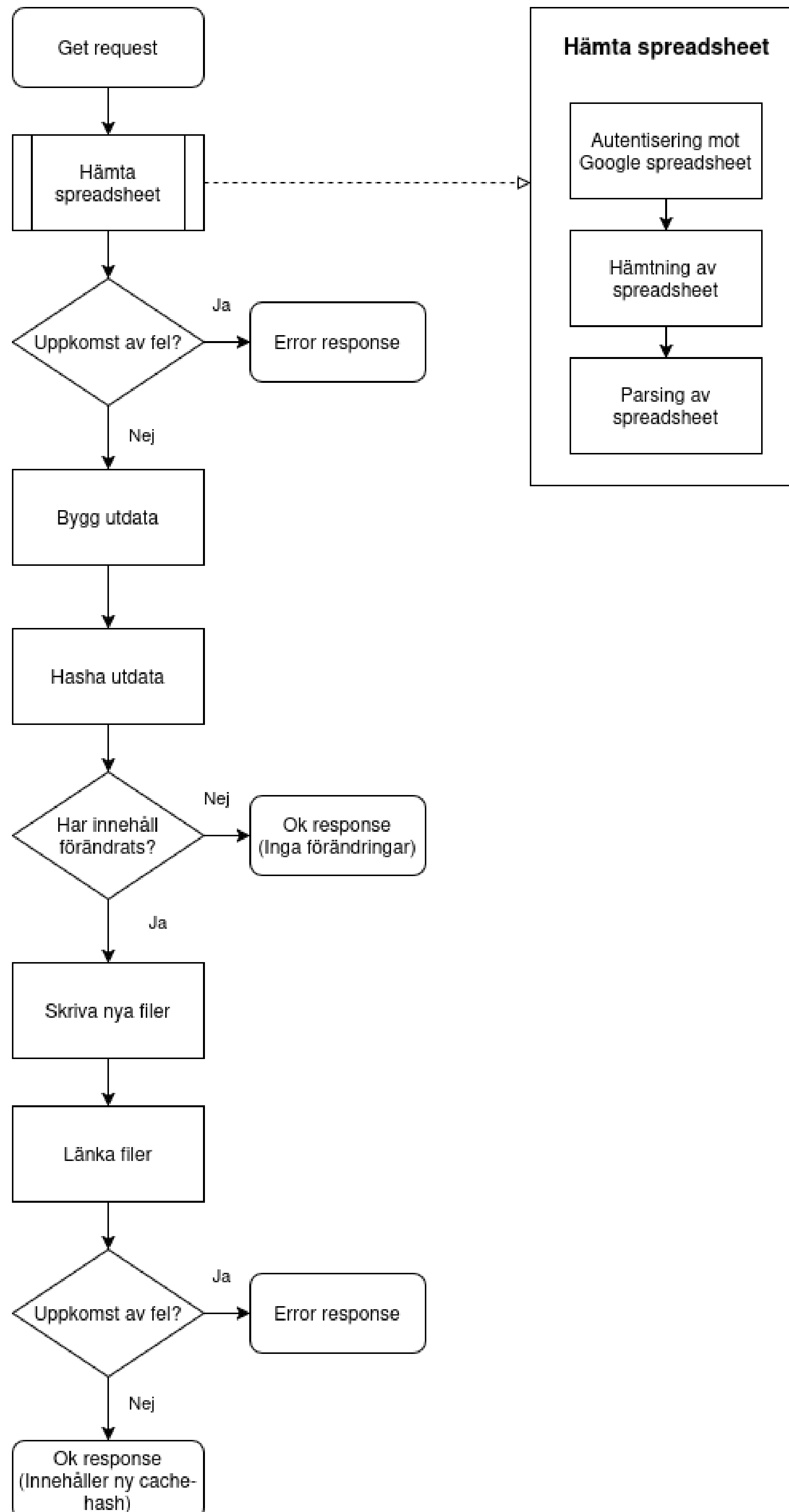
Spreadsheet.elm: Denna modul, som namnet antyder, är kopplad till hantering av inhämtad data från ett Google Spreadsheet. Den innehåller alla typer och funktioner som används då inhämtad produktdata parsas.

Typer för utdata: Cache servicen använder sig av fyra moduler för att representera den utdata som ska sparas i Webbhusets databas: Category.elm, Price.elm, Product.elm och Route.elm. Varje modul innehåller en typ i form av ett record och även tillhörande encoder och decoder funktioner, vilka underlättar översättning av data till korrekt strukturerad utdata.

Language.elm och Currency.elm: För att kunna arbeta dynamiskt och undvika hårdkodade lösningar används Webbhusets globala moduler för språk och valutor. Exempelvis ansvarar modulen Language.elm för att lagra alla språk som kan förekomma i indata från ett kalkylark. På samma sätt används modulen Currency.elm för att lagra de valutor som kan förekomma i indata. Dessa moduler används även av andra mikrotjänster i Webbhusets system vilket underlättar interaktionen mellan dem eftersom när ett nytt språk eller valuta ska introduceras så behöver endast ändringar i respektive modul göras.

4.2 Inhämtning av data

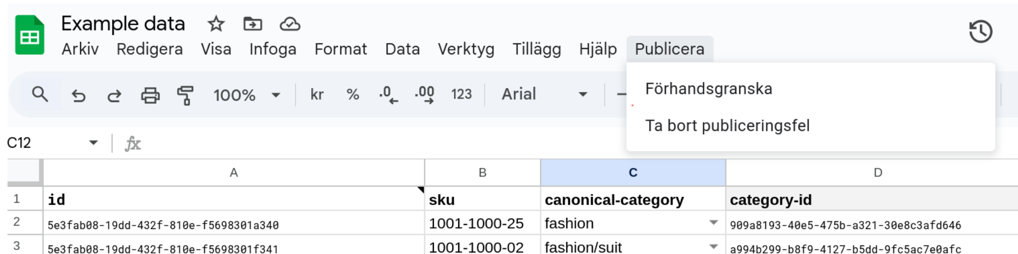
Då mikrotjänsten tar emot en GET-request hämtas data från kalkylarket via Google Sheets API. Nedan följer en steg-för-steg-beskrivning av datahämtningen. Samtliga steg återges även i figur 4.2.



Figur 4.2: Flödesschema över cache-processen.

4.2.1 Get-request

Användaren kan välja att förhandsgranska data i kalkylarket vilket visas i figur 4.3. När detta görs skickas en GET-request till cache-servicen, vilket initierar hämtningen av datan.



Figur 4.3: Knapp för att efterfråga en cache lagring

Mikrotjänsten "Cache.elm" är anpassad för hur Elm-server fungerar. Servern i "Cache.elm", implementerad som en webworker i Elm, lyssnar kontinuerligt på olika meddelanden och agerar utifrån deras typ. I figur 4.4 visas alla meddelanden som används för autentisering, hämtning av kalkylarket, parsing av inhämtad data och slutligen överföring av den avkodade datan till utdata-processen.

```
type Msg
    = GotInitOutputDirResult (Result File.Error ())
    | GotCredentials (Result String Spreadsheet.Credentials)
    | GotRequest Server.Request
    | GotProductImportResult Server.Request (Result String (Dict String Spreadsheet.Sheet))
    | GotCacheDirCheckResult Server.Request CacheData (Result File.Error ())
    | GotFileWriteResult Server.Request CacheData (FD.Dict Crypto.Hash Encode.Value) (Result File.Error ())
    | GotMakeDirResult Server.Request CacheData (Result File.Error ())
    | GotFileLinkResult Server.Request CacheData (FD.Dict OutputLink Crypto.Hash) (Result File.Error ())
    | GotUpdateMasterResult Server.Request (Result File.Error ())
    | GotSignal Process.Signal
    | NoOp
```

Figur 4.4: Alla meddelanden i Cache.elm

4.2.2 Get Spreadsheet

För att kunna hämta data krävs att cache-servicen autentiserar sig mot det kalkylark som har skickat en GET-request. En auth-metod i Elm skickar en privat nyckel till API:et för att verifiera att användaren har behörighet att läsa och skriva till kalkylarket. Som svar får cache-servicen antingen en token eller ett felmeddelande. Token används sedan för att ladda hela kalkylarket som en dictionary, där keys är bladnamn och values innehåller motsvarande data.

När data har hämtats som en dictionary parsas den till ett format som är enklare att felsöka och omvandla till utdata-typer. För detta definieras olika Elm-records som visas i figur 4.5. Själva kalkylarket representeras av en record som innehåller listor av rätt typ för varje blad, t.ex. 'Product' för produktsheets och 'Attribute' för egenskapssheets. Eftersom kolumnstrukturen varierar mellan bladen används specifika parse-metoder till varje blad. Varje parser kontrollerar att värdet i varje cell matchar den förväntade typen. Om allt stämmer returnerar parsern datan, annars returneras ett felmeddelande som underlättar felsökning. När alla parsers lyckas sparas datan i respektive record.

Figur 4.5: Alla typer för inhämtade data

När datan parsas kontrolleras att alla inlästa värden har de datatyper som cache-servicen förväntar sig, till exempel att ett produkt-ID är en sträng och inte ett booleskt värde. Vissa kalkylblad innehåller dessutom referenser till data i andra blad. Varje produkt tillhör exempelvis en kategori vars ID måste matcha motsvarande kategori-ID i kategoribladet. Under parsningen verifieras därför även att alla sådana korsreferenser mellan blad är giltiga.
Genom att skriva i Apps Script i kalkylarket kan vi sedan skriva tillbaka dessa fel till kalkylarket. Först samlar vi in parser-resultaten, om data är korrupt anges cell-range och feltyp. Cellen kan till exempel markeras med röd bakgrund, och en kommentar läggs till som beskriver felet, så att användaren enkelt kan åtgärda det.

Produktexporten misslyckades:


```
{
  "message": "Failed to parse spreadsheet",
  "errors": [
    {
      "range": "Price!E2:E2",
      "message": "Problem with the given value:\n\n\"abc\"\n\nExpecting a FLOAT\nE
    }
  ]
}
```


Close

Figur 4.6: Exempel på felmeddelande vid parsningsfel

E	F	G
sale-price:SEK	sale-price:EUR	sale-price:USD
abc		
10 034,00 kr		
1 089,00 kr		
1 010,00 kr		
1 565,00 kr		
1 269,00 kr	€109,00	$119,00

Problem with the given value:

"abc"

Expecting a FLOAT
Expected float value.

Figur 4.7: Exempel på återkoppling av fel i kalkylark

4.3 Utdata och versionshistorik

Då den inlästa datan har parsats och felsökts utan att några fel har uppstått påbörjas konverteringen till utdata som ska sparas på Webbhusets servrar. I denna process ingår även versionshistorik och kontroll av vilken data som har förändrats sedan senaste skrivning till databasen.

4.3.1 Konstruktion och hashing av utdata

Den data av typ kalkylark som har byggts upp av cache-servicen enligt figur 4.5 används nu till att bygga en dictionary för de filer som ska sparas till databasen. Värdena i dictionaryn är JSON-kodade objekt som ska sparas i databasen och värdets tillhörande nyckel är en hashsumma beräknad på värdet självt. Hashsummorna beräknas med hash-funktionen SHA-256 och kodas URL-säker Base 64. Efter att alla värden lagts till i dictionaryn beräknasa en ny hashsumma från alla nycklar i dictionaryn, det vill säga en hashsumma av hashsummor. Hashsumman över alla utdatans hashsummor kallar vi för en cache-hash och används vid versionshistoriken och skrivningen av utdatan.

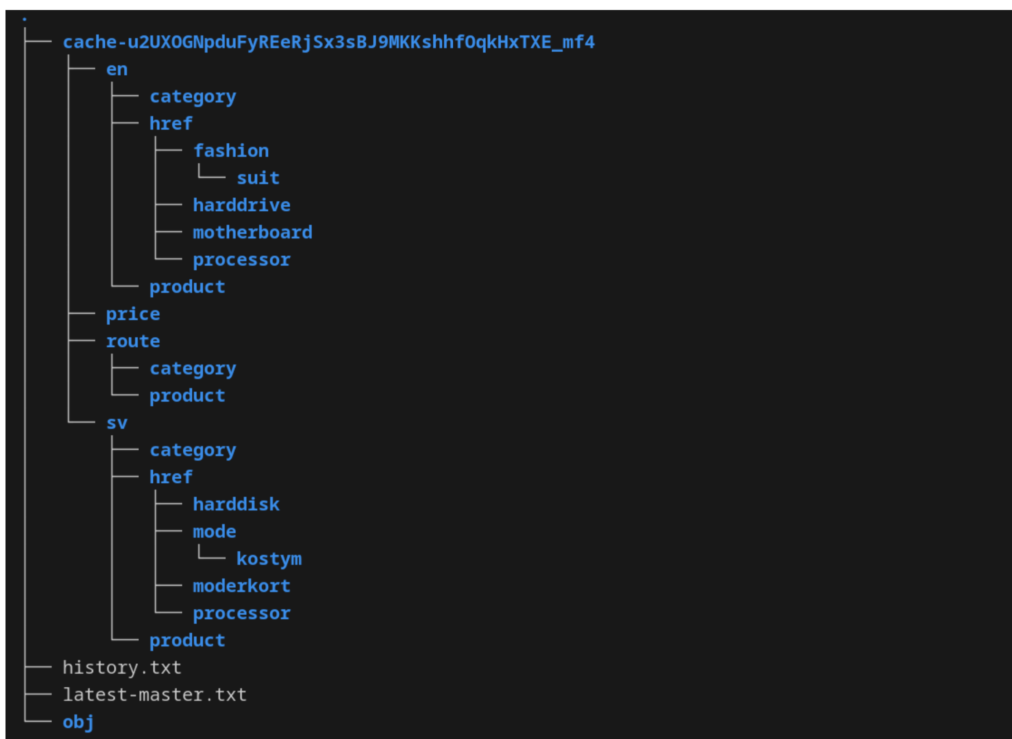
4.3.2 Jämförelse med tidigare sparad data och versionshistorik

Då hela utdatan är färdigkonstruerad och hashad jämförs cache-hashen med tidigare sparade cache-hashar. Om det inte redan finns en cache sparad med samma cache-hash kan vi direkt avgöra att den nya utdatan inte är identisk med någon tidigare version och skriver då ut den data som är ny till databasen. Existerar en cache med samma cache-hash skickas istället direkt ett svar tillbaka till avsändaren av GET-requesten som innehåller motsvarande hash.

4.3.3 Sparande av data på disk

Om den nya utdatans cache-hash inte är identisk med någon tidigare sparad utdatas cache-hash påbörjas istället skrivningen av den nya utdatan till databasen. Utdatan sparas som JSON-filer enligt strukturen i figur 4.8. Mappen obj innehåller alla JSON-filer som tidigare sparats till databasen. JSON-filerna är namngivna utifrån sina hashar och sparas i en undermapp till obj-mappen som matchar de två första tecknen i en JSON-fils hash. Detta för att undvika prestandaproblem då stora antal objekt sparas i obj-mappen, det kan exempelvis vara så att en kund har ett sortiment med hundratusentals produkter vilka kommer sparas som enskilda objekt. Då en ny cache ska skrivas jämförs all utdatas individuella hashar med de filer som redan är sparade i obj mappen. Om det redan finns en fil med samma hash i obj-mappen behöver inte filen skrivas om på nytt utan istället kan en hard link skapas i cache-mappen som pekar på motsvarande fil i obj-mappen för att spara på diskutrymme. Om det inte finns någon fil med samma hash som en av de filer som ska sparas skrivs en ny fil i obj-mappen som då kan användas på liknande sätt för framtida cachar.

Efter att alla filer med en ny hash är skrivna till obj-mappen byggs en ny cache i mappen "Caches". En cache är namngiven "cache-" följt av sin cache-hash. Eftersom all ny data redan har skrivits till obj-mappen sparas innehållet i varje cache som en hard link som pekar på motsvarande fil i obj-mappen. Varje cache följer en struktur som används i Webbhusets befintliga projekt: Priser och routes till olika produkter och kategorier för den aktuella webbshoppen sparas i separata mappar medans språkberoende information om produkter och kategorier sparas i språkspecifika mappar, i detta fall engelska och svenska.

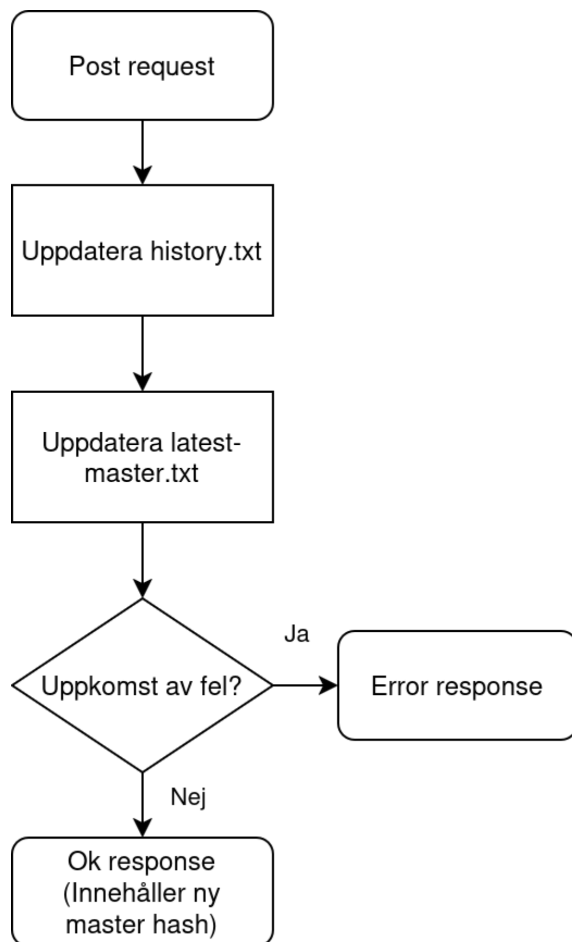


Figur 4.8: Struktur av databas

Efter att alla nya filer har skrivits till obj-mappen och länkningen till den nya cache-mappen har slutförts skickas ett svar till avsändaren av GET-requesten som innehåller den nya cache-hashen.

4.4 Uppdatering av nuvarande version

Efter att en lyckad cachning har genomförts och användaren har fått tillbaka ett svar innehållande den nya cache-hashen har användaren möjlighet att förhandsgranska hur ändringarna ser ut på webbshoppen. Väljer användaren sedan att publicera ändringarna skickas en POST-request till mikrotjänsten. Detta triggar mikrotjänsten att uppdatera filen ”latest-master.txt” till att innehålla den nya cachens hash och föregående master hash läggs till i ”history.txt” som innehåller alla tidigare master hashar. Slutligen skickas ett svar till användaren med den nya master-hashen som meddelar att publiceringen av den nya cache har lyckats. Detta illustreras i figur 4.9.



Figur 4.9: Flödesschema för uppdatering av aktuell master cache.

När en annan mikrotjänst i Webbhusets arkitektur läser data från databasen tittar den på vilken hash som är angiven i ”latest-master.txt” och läser in datan från motsvarande cache. Detta gör att det enkelt går att följa historiken för cachar som har använts och återställa aktuell cache till en tidigare version genom att endast ändra på innehållet i ”latest-master.txt”.

Kapitel 5

Resultat och diskussion

Den implementerade mikrotjänsten uppfyller de krav som sattes upp i projektets planeringsfas. Tjänsten hämtar in, bearbetar och felsöker kunddata från ett Google Spreadsheet vilket triggas genom en GET-request. Om kalkylarket innehåller några fel återkopplas dem till kalkylarket genom ett felmeddelande som hanteras av ett Apps script som är kopplat till själva kalkylarket. Om kalkylarket parsas utan fel påbörjas struktureringen av utdata som följer ett format som Webbhuset använder för sina projekt. Efter att all utdata som ska tillhöra cachen har konstruerats skapas hashsummor för varje enskilt objekt samt en övergripande hashsumma för hela cachen. Detta låter mikrotjänsten avgöra vilka filer som inte tidigare har sparats i databasen och dessa filer sparas sedan. Användaren får sedan möjlighet att förhandsgranska ändringarna utifrån den nya cachen och om användaren välja att publicera dem uppdateras slutligen en historik av hashsummor för cachelagringar vilket används för hålla koll på versionshistoriken. Kunddatan som har lagrats i databasen kommer sedan läsas av andra mikrotjänster i Webbhusets system. Detta innefattar bland annat presentation av olika produkter till potentiella kunder genom exempelvis en webbshop.

5.1 Modularitet

En av de mest centrala delarna i det här projektet är att kunna hämta data från ett kalkylark på det sätt som Webbhuset AB önskar. Detta har vi lyckats med: mikrotjänsten kan hämta rätt formaterad data ur kalkylarket och parsa indata till korrekta datatyper som sedan används för att bygga utdata.

Ett annat viktigt krav från Webbhuset är att lösningen ska vara så modulär som möjligt, den skulle kunna återanvändas på flera ställen utan större ändringar. I den framtagna lösningen är dock själva inläsningen av data inte särskilt modulär. Skälet är att datan skiljer sig mellan olika företag och därför måste lösningen anpassas efter hur datan ser ut i respektive kalkylark. Till exempel är alla parsers skrivna utifrån datans struktur i varje blad och måste justeras om datan förändras. Detta är ändå en förbättring jämfört med tidigare lösningar, tidigare behövde hela lösningen skrivas om, men nu räcker det att anpassa datainläsningen, till exempel genom att skriva om parsers så att de följer det nya dataformatet. På så sätt sparas mycket tid, eftersom den del som behöver justeras har blivit betydligt mindre än förr. Däremot är vissa delar av felhanteringen modulära, till exempel lösningen i Apps Script som skriver tillbaka felmarkeringar direkt i kalkylarket. Den delen kan återanvändas i olika kalkylark och visar tydligt var felet uppstod.

Trots att delar av mikrotjänsten kommer att behöva anpassas utifrån vilken typ av kunddata ett projekt tar emot så är själva cache-lagringen och versionshistoriken av bearbetad indata modulär. Detta då den lösning som vi har designat och implementerat under projektets gång enbart är beroende av att den indata som mikrotjänsten tar emot översätts till de Elm-typer som används för att representera utdata. Så länge mikrotjänsten anpassas till att göra detta kommer den kunna återanvändas i samtliga av webbhusets projekt eftersom de lagrar kunddata på samma sätt i samtliga projekt. Skulle strukturen på Webbhusets databaser få nya behov och ändras på behövs alltså endast mikrotjänsten anpassas utifrån detta för att ändringarna ska ske och vara kompatibla i samtliga projekt. Detsamma gäller även mikrotjänstens funktionalitet för versionshistorik som också är oberoende av vilken typ av indata som tas emot.

Sammanfattningsvis kan mikrotjänsten beskrivas som delvis modulär. Inläsning, parsning och fel-

hantering av kunddata behöver anpassas utifrån varje individuellt projekt medan cache-lagringen och versionshistoriken kan återanvändas utan att behöva justeras. Detta innebär att den prototyp som vi har designat och implementerat med enkelhet och minimala antal ändringar kan återanvändas till projekt där kunddata erhålls från Google Sheets då endast delar av parsningen behöver justeras. Om kunddatan kommer från någon annan källa behöver dock fler ändringar för inläsningen av kunddata utföras.

5.2 Återkoppling av fel i kunddata

När det gäller felhantering tar lösningen hänsyn till några vanliga fel, exempelvis kontrolleras att korrekt datatyp ligger i produkt och kategori blad men omfattningen är fortfarande begränsad. Det finns sannolikt andra feltyper som inte upptäcks i nuläget, något som skulle kunna åtgärdas med mer tid.

5.3 Arbetsprocess

Vi planerade att arbeta med CI/CD (Continuous Integration och Continuous Deployment) i kombination med agil metodik. CI/CD-arbetssättet var givande i inledningsskedet av projektet: redan första veckan kunde tester köras kontinuerligt och lösningarna förbättras iterativt. De veckovisa mötena med industrimentorn innebar tydliga mål varje vecka, i slutet av veckan kunde en fungerande del lösning demonstreras och vidareutvecklas.

Det agila arbetssättet gjorde det också enklare att snabbt hitta och åtgärda fel. Den täta dialogen med kunden som i detta fall var industrimentorn, höll kundens behov i fokus och minskade risken för avvikelser. Sammantaget bidrog CI/CD-strukturen och det agila samarbetet till att den framtagna lösningen ligger nära Webbhusets önskemål och kan användas i deras framtida projekt.

5.4 Hantering av ett stort antal filer

Mot projektets slut testades mikrotjänstens prestanda vid användning av större set av kunddata. Ett problem som uppstod var att Node.js kraschade i runtime då antalet filer som skulle cachelagras närmade sig 500 000. Detta beror på att instruktionerna för filskrivningarna och skapandet av hardlinks skickades från Elm till Node.js runtime som en enda enorm IO-instruktion. Detta leder i sin tur till att Node.js gräns för tilldelat minne för en enskild process överskrids vilket orsakar kraschen. För att lösa detta problem strukturerades mikrotjänsten om till att istället skicka varje IO-instruktion individuellt till Node.js runtime. Mikrotjänsten inväntar sedan ett resultat av instruktionen innan nästa instruktion skickas till Node.js. Denna lösning tillät bearbetning av en större mängd data utan någon större skillnad i exekveringstid.

5.5 Versionshistorik

Vid projektets början sattes möjligheten för cache-servicen att ha någon form av versionshistorik över databasen upp som ett delmål som skulle genomföras i mån av tid. Då projektet mer eller mindre följde den bestämda tidsplanen ansågs detta vara möjligt att genomföra och två huvudsakliga problemområden identifierades för den föreslagna implementationen: Prestanda och kollisionsrisk mellan hashsummor.

Implementationen av versionshistoriken bygger på att varje fil i en cache har en tillhörande hashsumma. Varje cache tilldelas i sin tur sedan en hashsumma baserad på alla hashsummor i cachen. Då en hashsumma per definition ska vara deterministisk och unik för varje indata som körs genom hashfunktionen kan vi därmed avgöra om innehållet i en ny cache är identiskt som innehållet i en tidigare sparad cache, det vill säga om deras hashsummor stämmer överens. På så vis kan mikrotjänsten undvika att skriva om redan tidigare lagrade cachar men även individuella filer som redan sparats och delas mellan cachar. I teorin ansåg vi att detta var en rimlig och robust lösning

att implementera men eftersom kunddata i ett projekt kan innefatta hundratusentals eller fler produkter och annan data som ska sparas vid varje cache-lagring behövde valet av hash-funktion göras med kollisionsrisk mellan hashsummor i åtanke. Ju större storleken är av en beräknad hashsumma desto mindre är risken för kollision och utifrån detta valde vi att använda hashfunktionen SHA-256 som anses vara en kollisionsresistent hashfunktion, dock var vi osäkra på hur tidseffektiv SHA-256 skulle vara att använda. Ytterligare en anledning till att vi valde att använda just SHA-256 som hashfunktion var att den redan fanns implementerad i Webbhusets miljö vilket ledde till att vi kunde spara tid och arbete som skulle ha gått åt till att implementera en annan hashfunktion. Vid testning av att hasha en stor mängd kunddata från ett spreadsheet (över 500 000 rader) tog det endast cirka en minut att parsa, felsöka och hasha datan. Detta ansågs vara acceptabelt och därför valde vi att inte byta hashfunktion till ett mer tidseffektivt alternativ. SHA-256 är även en kryptografiskt säker hashfunktion vilket i detta sammanhang egentligen inte är en nödvändig egenskap eftersom hashsummorna i första hand används för att bygga upp en datastruktur.

5.6 Förslag till förbättring

I mån av ytterligare tid hade vi prioriterat att utveckla återkopplingen av status och eventuella fel till användaren som startat cache-sparandet. Utöver att få korta felbeskrivningar eller ett ok meddelande som mikrotjänsten idag återkopplar till användare hade det kunnat vara av nytta att få en mer utförlig beskrivning av resultatet för cache-sparandet. Exempel på detta kan vara om det faktiskt skedde några ändringar i cachen eller om ingen ny cache behövde skrivas samt vilken som är aktuell cache. Ytterligare en förbättring skulle vara att implementera någon form av förhandsgranskning så att användaren kan se hur resultatet av den nya cache-skrivningen ser ut på den webbshop eller liknande som cachen representerar och att användaren utifrån detta skulle kunna välja att publicera eller kasta den nya versionen.

Vi hade gärna utfört mer utförliga tester på olika former av kunddata för att få en uppfattning om hur stort arbetet skulle bli med att anpassa mikrotjänsten utifrån detta. Detta hade kunnat vara att testa med fler kalkylark som var organiserade på ett annat sätt än vad som använts under projektet eller någon helt annan form av kunddata.

Kapitel 6

Slutsats

Författarna har lyckats designa och implementera en prototyp för den efterfrågade mikrotjänsten. Mikrotjänsten inväntar förfrågningar om att påbörja cache-lagring och läser då in testdatan från kalkylarket. Datan parsas och felsöks och återrapporterar eventuella fel tillbaka till kalkylarket. Den bearbetade datan sparas sedan enligt den bestämda strukturen i Webbhusets databaser. Användare får sedan ett svar om huruvida lagringen lyckades eller inte.
Trots att funktionalitet med en versionshistorik över cache-lagringar inte ingick i de huvudsakliga målen implementerades detta av författarna. Versionshistoriken dokumenterar vilka cachar som har varit aktuella och uppdateras efter varje lagring. Implementationen för versionshistorik möjliggör återställning till tidigare versioner.

Under projektets gång uppstod flera utmaningar, bland annat kopplade till att programmeringsspråket Elm var helt främmande för författarna samt till att mycket hänsyn behövde tas för att se till att mikrotjänsten skulle vara kompatibel med Webbhusets arkitektur. Genom att författarna arbetar agilt och med CI/CD så kunde dock problem delas upp i mindre beståndsdelar och lösas i små iterationer.

Den framtagna prototypen fungerar med testdata som ska efterlikna verklig kunddata Webbhuset hanterar. Författarna anser dock att ytterligare arbete med återkoppling till användare och hantering av en mer generaliserad kunddata behövs.

Bibliography

[1] "Om oss." Webbhuset. (Mar. 2025), [Online]. Available: https://webbhuset.se/om-oss.

[2] "E-handeln ökade under pandemin." Statistikmyndigheten SCB. (Mar. 2025), [Online]. Available: https://www.scb.se/hitta-statistik/temaomraden/sveriges-ekonomi/fordjupningsartiklar_Sveriges_ekonomi/e-handeln-okade-under-pandemin/.

[3] V. F. dos Santos, L. R. Sabino, G. M. Morais, and C. A. Gonçalves, "E-commerce: A short history follow-up on possible trends," *International Journal of Business Administration*, vol. 8, no. 7, pp. 12 573–12 573, Nov. 2017. DOI: 10.5430/ijba.v8n7p130.

[4] D. Kim, S. Lee, J. Chun, and J. Lee, "A semantic classification model for e-catalogs," in *Proceedings. IEEE International Conference on e-Commerce Technology, 2004. CEC 2004.*, Jul. 2004, pp. 85–92. DOI: 10.1109/ICECT.2004.1319721.

[5] A. Jhingran, "Moving up the food chain: Supporting e-commerce applications on databases," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 29, no. 4, pp. 50–54, Dec. 2000. DOI: 10.1145/369275.369287.

[6] F. Tapia, M. A. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," *Applied Sciences*, vol. 10, no. 17, Aug. 2020. DOI: https://doi.org/10.3390/app10175797.

[7] A. M. Gutiérrez-Fernández, M. Resinas, and A. Ruiz-Cortés, "Redefining a process engine as a microservice platform," in *Business Process Management Workshops*, Springer International Publishing, 2017, pp. 252–263, ISBN: 978-3-319-58457-7.

[8] M. Fowler. "Microservice trade-offs." Accessed April 2, 2025. (2015), [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.htmlguides/concepts.

[9] K. Downs. "What's a pure function?" (Jul. 2020), [Online]. Available: https://dev.to/kjdowns/what-s-a-pure-function-205e (visited on 04/01/2025).

[10] O. Olsson. "Elm — a beautiful language for web development." Medium, accessed April 2, 2025. (2022), [Online]. Available: https://medium.com/better-programming/elm-a-beautiful-language-for-web-development-bc06fd0b570f.

[11] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, ACM, 2013, pp. 411–422. DOI: 10.1145/2499370.2462161. [Online]. Available: https://doi.org/10.1145/2499370.2462161.

[12] S. Ritchie. "The case for Elm." Accessed April 2, 2025. (2024), [Online]. Available: https://samritchie.net/posts/the-case-for-elm/.

[13] "Web worker overview." Accessed April 2, 2025. (2023), [Online]. Available: https://web.dev/learn/performance/web-worker-overview.

[14] B. Güler. "Web workers in a nutshell." Accessed April 2, 2025. (2022), [Online]. Available: https://hwclass.medium.com/web-workers-in-a-nutshell-809f1dbe7fbf.

[15] B. Heyns, *Revolutionize data with google sheets api automation*, https://smythos.com/ai-industry-solutions/business-automation/google-sheets-automation-with-the-api/?utm_source=chatgpt.com, Last updated January 28, 2025. Accessed April 17, 2025, 2025.

[16] Google Developers. "Apps Script Overview." (2025), [Online]. Available: https://developers.google.com/apps-script/overview (visited on 04/30/2025).

[17] Google Developers. "Google Sheets api concepts." Accessed April 2, 2025. (), [Online]. Available: https://developers.google.com/workspace/sheets/api/guides/concepts.

[18] Svenska Institutet för Standarder. "GDPR och standarder." Accessed April 3, 2025. (2025), [Online]. Available: https://www.sis.se/iso27001/gdpr-och-standarder/.

[19] ServiceNow. "What is CI/CD pipeline?" Accessed April 2, 2025. (), [Online]. Available: https://www.servicenow.com/products/devops/what-is-cicd-pipeline.html.

[20] Right People Group. "Vad är CI/CD, automatisering och varför behöver företag det?" Accessed April 2, 2025. (2025), [Online]. Available: https://rightpeoplegroup.com/sv/blog/vad-ar-ci-cd-automatisering-och-varfor-behover-foretag-det.

[21] K. Beck, M. Beedle, A. van Bennekum, *et al.*, *Manifesto for Agile Software Development*, https://agilemanifesto.org/, Accessed: 2025-05-14, 2001.

[22] Institute Project Management. "Agile methodologies and framework." Accessed April 7, 2025. (2022), [Online]. Available: https://instituteprojectmanagement.com/blog/agile-methodologies-and-framework/#null.

INSTITUTIONEN FÖR DATA- OCH INFORMATIONSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2025
www.chalmers.se

CHALMERS