Tjänstebaserad arkitektur: från monolit till mikrotjänster

Daniel Vainio

Kandidatavhandling i datavetenskap

Handledare: Ivan Porres

Fakulteten för naturvetenskaper och teknik

Åbo Akademi

2022

Referat

TODO...

Innehållsförteckning

1 Inledning

En applikation har två kategorier av krav. Den första kategorin innehåller de funktionella kraven och den andra kategorin innehåller kvalitetsegenskaperna. De funktionella kraven är oftast de krav man har på vad användaren skall kunna uträtta med mjukvaran, dessa är oftast beskrivna i så kallade ”*user stories*”. Med dessa försöker man fastslå inom vilka ramar mjukvarans funktionalitet kommer att existera. Händelser defineras genom att skapa en fiktiv händelse som återspeglar vad en användare bör ha möjligheten att uträtta[1].

Kvalitetsegenskaperna kommer direkt att påverkas av valet av mjukvaruarkitektur. Vissa typer utav arkitekturer kommer att bemöta kraven på vissa system bättre än andra på grund av dess teknologiska överlägsenhet inom specifika områden.

Den troligen vanligste arkitekturen inom systemutveckling för webben idag är den monolitiska arkitekturen. En monolit defineras oftast som en applikation där de olika delarna är beroende av varandra och de olika modulerna i en applikation paketeras och distribueras i en enkel applikation[2]. Den monolitiska arkitekturen använder även oftast en delad databas och delad kod mellan modulerna.

Monoliter är oftast en enkel Java WAR fil, en enkel hierarki av NodeJS mappar eller en Python/Django applikation. Den monolitiska arkitekturen använder oftast en skiktad design med separata lager för användargränssnitt, programlogik och datahantering samt datalagring[3].

Mikrotjänster är en arkitektur som bygger på att applikationen delas upp i mindre självständiga moduler som kallas för tjänster. Tjänsterna har oftast ett specifikt användningsområde och de bör inte vara sammankopplade genom delad kod eller delad databas. Tjänsterna bör heller inte känna till varandras existens, kommunikation bör ske så obundet som möjligt. En mikrotjänst distribueras enskilt från alla andra tjänster och kommunicerar med andra tjänster över nätverk. Syftet är att uppfylla de kvalitetsegenskaper som den monolitiska arkitekturen inte hanterar lika bra.

Mikrotjänster är vad man kan kalla för en typ utav tjänstebaserad arkitektur som är distribuerad i sin natur.

Kvalitetsegenskaperna kan delas in i många kategorier. I denna text kommer jag inte att ta upp alla dessa. Jag kommer att fokusera främst på skillnader som rör de krav som direkt påverkas genom att jämföra en monolitisk arkitektur med mikrotjänster.

Några av de viktigaste av kraven som rör dessa arkitekturer är följande[4]:

- Skalbarhet
- Stabilitet
- Tillgänglighet
- Heterogen teknologi
- Distribution
- Organisatorisk anpassning

Mikrotjänster är enklare att skala upp med inkommande trafik således att man inte behöver skala upp hela applikationen utan man kan skala upp en tjänst åt gången efter behov. Detta innebär att de delar av applikationen som får, eller förväntas få, en större mängd trafik kan skalas upp för att möta trafiken istället för att skala upp hela applikationen som man hamnar göra med en monolit.

Om en tjänst blir otillgänglig så kommer det inte att ha lika grova implikationer på grund av tjänsternas lösa beroende av varandra. En del eller delar av applikationen kommer inte att vara tillgänglig men de övriga tjänsterna fortsätter fungera. Detta ger stabilitet och gör applikationen mera robust om man jämför med en monolit vars funktionalitet avbryts då något oförväntat sker[4].

Då tjänsterna är löst beronde av varandra så kan man välja den teknologi man anser vara bäst för respektive tänst. Ingen tjänst är bunden till en viss teknologi som exempelvis programmeringsspråk, ramverk eller databaser. Detta ger mera flexibilitet än en monolitisk arkitektur där man är mera låst i en specifik teknologi[4].

Det är även enklare för ett team inom organisationen att ta ägarskap över en eller flera tjänster istället för att fokusera på ett lager inom en mera traditionell arkitektur.

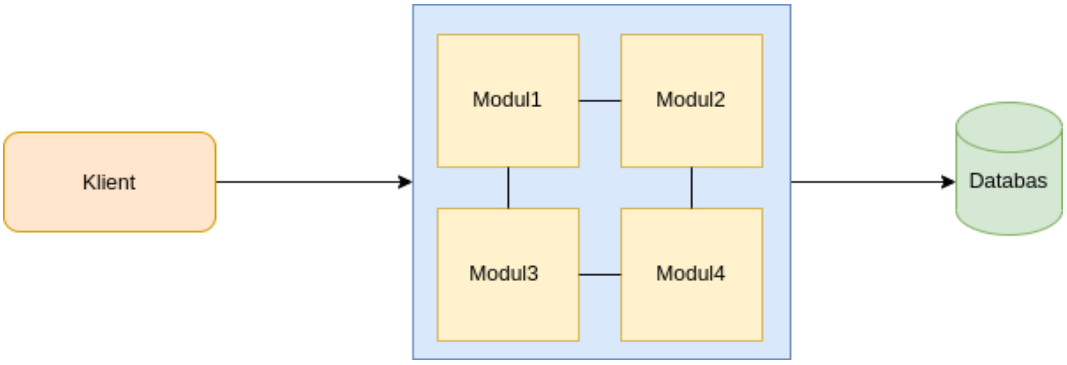
2 Design och arkitektur

Mjukvaruarkitektur är en viktig disciplin inom programvaruutveckling där man genom att fatta beslut på högre nivå försöker påverka utkomsten av mjukvarans kvalitet[5]. Genom att bryta upp mjukvaran i modulära delar så kan man förbättra mjukvarans flexibilitet och göra mjukvaran enklare att förstå[6]. Den grundläggande tanken bakom mjukvaruarkitektur bygger på den fundamentala idén att mjukvara sammansätts av flera olika element och dess relationer till varandra. En modell föreslogs av *Perry et. al.* i texten *Foundations for the Study of Software Architcture* där modellen för mjukvaruarkitektur kan brytas ner i de element som sköter behandlingen av data, de element som innehåller data och de element som binder samma dessa element. En viktig del av mjukvaruarkitekturen är den logiska grund som arkitekturen baserar sig på. Detta skall utgöra motivationen för valet av vilken stil utav arkitektur man väljer för att binda samman elementen[9].

2.1 Monolit

I en monolitisk arkitektur så är all funktionalitet inkapslad i enbart en applikation. Oftast så är monoliten uppdelad i olika moduler som inte kan fungera oberoende av varandra utan monoliten måste köras som en enhet sammansatt av en eller flera moduler. Sammankopplingen mellan modulerna är väldigt stark och det finns ett beroende av logik och data mellan dessa moduler[10].

I *figur 1* kan man se en grundläggande struktur för en monolitisk arkitektur av en webbapplikation som är sammansatt av flera olika moduler som har en stark sammankoppling och delar databas. *Sam Newman* hänvisar i boken *Building Microservices* till denna typ utav lösning som en modulär monolit. Då en sådan lösning kan bryta ner koden i en sammansättning så varje modul representerar ett användningsområde så saknar den samma löst sammansatta komposition av oberoende moduler man finner i mikrotjänster.



Figur 1: Modulär monolit med delad databas där modulerna är starkt beroende av varandra

Det finns både fördelar och nackdelar med en monolitisk arkitektur. Det brukar oftast vara relativt enkelt att påbörja ett projekt med denna arkitektur, den är relativt enkel att förstå och utveckling av en monolit har relativt enkla processer i utvecklingsstadiet. Det är även enklare att sätta en monolit i bruk för användning. Återanvändning av kod är mycket enklare i en monolit då dess struktur redan innebär att det finns ett naturligt beroende av modulerna och kod kan återanvändas fritt från andra moduler.

Nackdelarna blir uppenbara då monoliten växer och kodbasen blir större. Koden blir oftast svårare att förstå. Den är svårare att skala upp och man är bunden till den teknologi som monoliten är skapad i då det inte finns någon flexibilitet vad beträffar programmeringsspråk eller annan teknologi[10].

2.2 Tjänsteorienterad arkitektur (SOA)

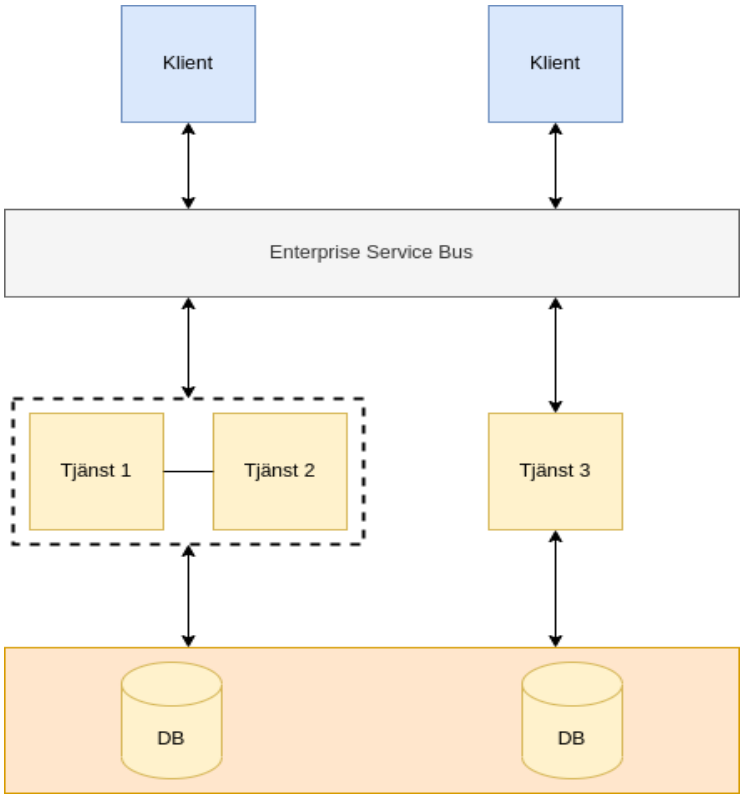
En tjänsteorienterad arkitektur är en arkitektur där flera oberoende tjänster kommunicerar med varandra genom servicegränssnitt. Varje tjänst fungerar självständigt från de andra tjänsterna. Dessa tjänster kan anropas med väldigt lite eller ingen vetskap om de andra tjänsternas funktionalitet[11].

Tjänsterna exponeras genom olika nätverksprotokoll som exempelvis SOAP eller REST för att manipulera data. Tjänsterna i en tjänsteorienterad arkitektur kan byggas från grunden men ofta används funktionalitet som finns tillgänglig sedan tidigare[11].

Ett simplistiskt sätt att sammanfatta en tjänsteorienterad arkitektur är att det är en arkitektur, inte en sätt att designa enskild mjukvara, i ett bredare omfång. Ett mål kan vara att integrera flera olika existerande system så att de kan fungera i en större helhet eller att man skapar mindre komponenter som kommunicerar med varanda över HTTP/S. Detta gör givetvis att det finns ett beroende mellan tjänsternas enskilda komponenter. Man kan även bygga tjänster genom att sammanfoga flera tjänster[12].

En tjänsteorienterad arkitektur definerar inte hur själva tjänsten skall byggas. Detta saknar definition då detta är en arkitektur snarare än en stil för att designa enskild mjukvara.

Klienterna kommunicerar ofta med tjänsterna genom en ESB (Enterprise Service Bus). Detta är en central mjukvarukomponent som styr integrationen, kommunikation och meddelanden mellan tjänster och applikationer som kommunicerar med varandra. En ESB har en väldigt stor uppgift i denna arkitektur och är en väldigt central del inom SOA. En stor nackdel med ESB är att det är mjukvara som tenderar att bli väldigt komplicerad på grund av dess berda användningsområde[17].



Figur 2: Exempel på en SOA arkitektur med en ESB som sköter kommunikation och mellan klienter och tjänster.

2.3 Mikrotjänster

Nästa steg i evolutionen som bygger på idén om enskilda tjänster i den tjänsteorienterade arkitekturen är mikrotjänster. Mikrotjänster är oberoende tjänster som är utformade efter en affärsdomän. En tjänst kapslar in funktionaliteten och gör den tillgänglig för andra tjänster genom kommunikation över nätverk. Mikrotjänster är en typ av distribuerad tjänsteorienterad arkitektur som fokuserar på hur man strukturerar mjukvaran på applikationsnivå. Mjukvaran byggs till en helhet av små tjänster som har ett klart definerat användningsområde där varje tjänst har egen egen databas och kan distribueras enskilt. Det får inte finnas något beroende mellan tjänsterna då ett av målen är att det skall vara oberoende av varandra.

Genom att bryta beroendet mellan tjänsterna så finns det fördelar som kan utnyttjas där exempelvis monoliten inte är fullt lika kapabel[13]:

- Genom löst sammansatta tjänster så behöver inte hela applikationen tas ur bruk om en tjänst får tekniska problem. Problemet är isolerat till en tjänst.
- Utveckling av tjänsterna kan även isoleras i mindre team som är ansvariga för en eller flera tjänster. Man kan strukturera organisationen ganska långt efter de tjänster som finns. Ett team blir således experter på området tjänsten eller tjänsterna sköter om.
- Varje tjänst kan distribueras enskilt. Detta leder till att en tjänst kan skalas upp efter behov. Skalning av tjänsterna kan ske i flera riktningar än den monolitiska arkitekturen.
- Varje tjänst ”äger” sitt eget tillstånd genom en databas per tjänst.
- Varje tjänst kan utnyttja olika teknologier utan att påverka de andra tjänsterna.
- Tjänsterna är inte beroende av hur datan konsumeras. Datan kan konsumeras av mobila applikationer, webbapplikationer och desktop applikationer.

Kommunikationen mellan klient och mikrotjänster sker oftast över HTTP/S eller gRPC. I *figur2* ser man ett exempel på mikrotjänster som använder en så kallad ”*API Gateway*” för att kommunicera. En ”*API Gateway*” sitter mellan klienten och tjänsterna och diregerar begäranden från klienter till tjänsterna. Utan en *API Gateway* så är ett alternativ att kommunicera direkt med tjänsterna[14].

Tjänsterna kan kommunicera med varandra asynkront eller synkront. Att använda en synkron metod vore dock att skapa beroende mellan tjänsterna då de i sådana fall skulle vara tvugna att ha kännedom om de andra tjänsternas API. Asynkron kommunikation är att föredra då binder inte tjänsterna till varandra.

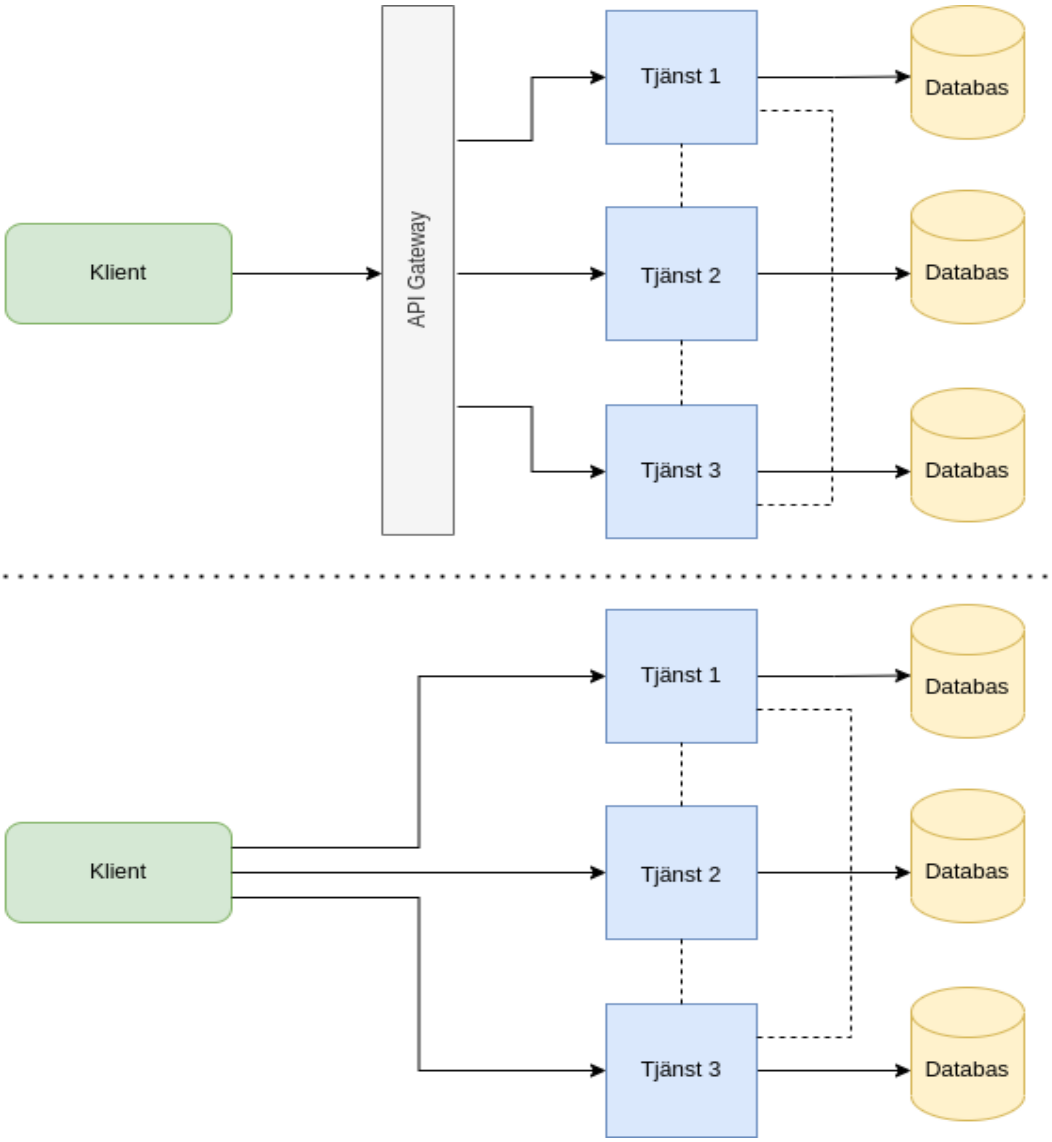


Figure 3: Mikrotjänster med och utan en API Gateway

3. Kommunikation mellan tjänster

Tjänster kommer att behöva meddela händelser som sker eller åberopa funktionalitet från en annan tjänst. På något sätt måste det alltså vara möjligt att kommunicera mellan dessa tjänster. Det kan röra sig om exempelvis att en användare vill skapa ett nytt användarkonto som sköts av en specifik tjänst, *tjänst A*. När denna händelse har ägt rum så vill man kanske skicka ett e-post meddelande till den nya användaren genom en annan tjänst, *tjänst B*. Då måste *tjänst A* meddela *tjänst B* att användaren har blivit skapad och skicka e-post adressen vidare till *tjänst B* som sedan skickar ett e-post meddelande med informationen från *tjänst A* som har förmedlats genom ett meddelande.

Det finns en mängd teknologier och protokoll som tjänsterna kan använda sig av för att kommunicera med varandra. En av de viktigaste sakerna att ha i åtake när man väljer teknologi är att tjänsterna inte får gå sönder om man gör modifieringar i källkoden eller att tjänsterna är tillgängliga för kommunikation. Om tjänsten i fråga samt andra tjänster blir lidande av förändringar så har man skapat ett beroende mellan tjänsterna som direkt påverkar stabiliteten. Därför är det viktigt att separera de interna funktionerna i en tjänst från omvärlden, andra tjänster skall inte behöva ändra sina interna funktioner på grund av att något har förändrats i en annan tjänst[15].

Några av de vanligaste teknologier som används för kommunikation mellan tjänster är följande:

- RPC
- REST
- Meddelandeförmedlare (Message Broker)

Flera olika mönster finns tillgängliga beroende på vilken typ utav implementation som projektet kräver. Det är inte ovanligt att använda flera av dessa olika teknologier i ett och samma projekt.

3.1 Stilar av kommunikation

Det finns några grundläggande stilar av kommunikation som kan appliceras för kommunikation mellan mikrotjänster. Alla stilar kommer inte att tas upp i detta stycke men man kan på den högsta nivån dela in några av dem i följande kategorier[16]:

Synkron blockerande kommunikation

En tjänst kallar på en annan tjänst och blockerar(är otillgänglig) i väntan på ett svar.

Asynkron icke-blockerande kommunikation

Tjänsten som kallar på en annan tjänst kan fortsätta utan att blockera oavsett om ett svar mottages eller ej.

Förfrågan-Svar (Request-Response) modellen

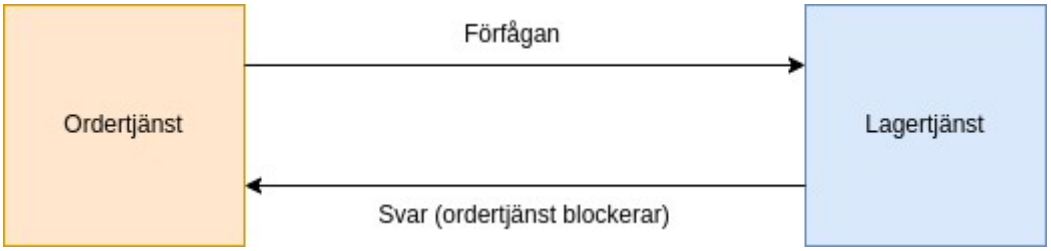
En tjänst kallar på en annan tjänst och förväntar sig ett svar då den andra tjänsten är klar. Denna stil kan vara både synkron och asynkron.

Händelsestyrd kommunikation (Event-Driven communication)

En tjänst meddelar händelser som andra tjänster kan konsumera. Tjänsten som meddelar händelsen känner inte till vilka andra tjänster som konsumerar denna händelse.

3.2 Synkron kommunikation

Med synkron kommunikation så skickar en tjänst en förfrågan till en annan tjänst och väntar sedan på ett svar. Tjänsten som skickade förfrågan blockerar tills tjänsten har fått ett svar från den andra tjänsten. I *figur 3* skickar ordertjänsten en förfrågan till lagertjänsten och blockerar i väntan på svar från lagerstjänsten.



Figur 4: En synkront meddelande mellan två mikrotjänster

Ett problem med denna typ utav kommunikation är att det kopplar samman tjänsterna på ett sätt man gärna vill undvika då man bygger mikrotjänster. Denna typ utav tvåvägskomunnikation leder till att man måste ha någon form utav plan för att hantera potentiella problem som kan uppstå då tjänsterna inte är tillgängliga. Om lagertjänsten slutar fungera så kommer i detta fall svaret att gå förlorat. Sammankopplingen mellan tjänsterna är därmed mellan två specfika instanser av tjänster och inte enbart mellan tjänsterna på högre nivå. Om flera tjänster än i exemplet kommunicerar på samma sätt så kan detta i sin tur skapa en kaskad av problem om en eller flera tjänster slutar att fungera.

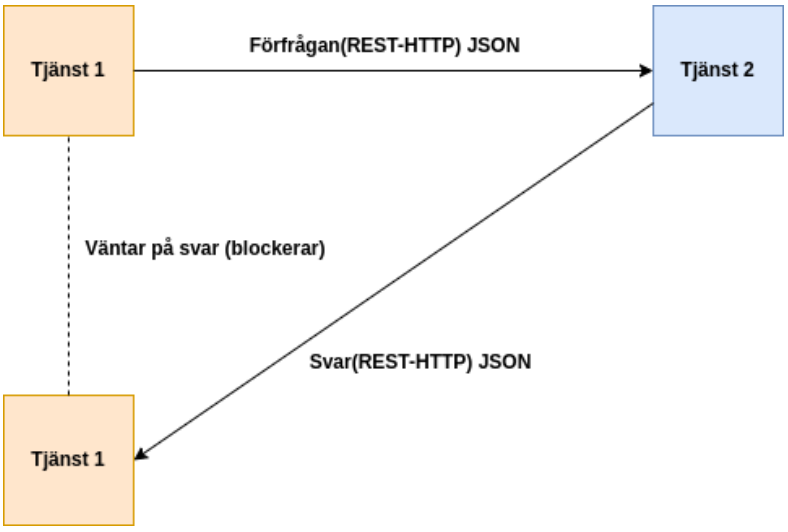
Två synkrona teknologier som använder sig utav *förfrågan-svar* modellen är REST över HTTP och RPC.

3.2.1 REST

REST är en stil av kommunikation som är inspirerad av webben. Kärnan i REST kommunikation är resurser. En resurs är en entitet som oftast representerar ett objekt som enbart tjänsten känner till. En resurs kan exempelvis vara en *Användare* i en tjänst som hanterar användarinformation. JSON är det vanligaste sättet att förmedla resursen från sändaren till mottagaren. Detta gör att JSON representationen av *Användaren* inte är sammankopplad med hur den är lagrad i systemet. Det finns ingen exakt definition för vilket underliggande protokoll som REST bör använda, men det allra vanligaste är HTTP/S. Detta ger REST möjligheten att använda sig utav HTTP verben ”get”, ”post”, ”put” och ”delete” för att hämta, spara, ändra och radera data.

Ett av problemen med att använda REST är att det skapar en stark sammankoppling mellan tjänsterna. Tjänsterna måste känna till de andra tjänsternas API och därmed så kommer förändringar att potentiellt skapa problem. Då REST även är beroende av HTTP som kommunicerar över TCP så kommer begränsningarna av dessa protokoll spela en roll i kommunikationen mellan tjänster.

REST finner sin plats då man skapar en API för extern kommunikation mellan tjänster och olika klienter, som exempelvis webbapplikationer elller mobilapplikationer. Om tjänsterna inte lider av en modell med synkrona förfrågningar och svar så lämpar sig även REST för kommunikation mellan tjänsterna, men detta är inte att föredra om man har möjligheten att använda sig utav icke-blockerande metoder.



Figur 5: REST kommunikation mellan tjänster

3.2.2 RPC

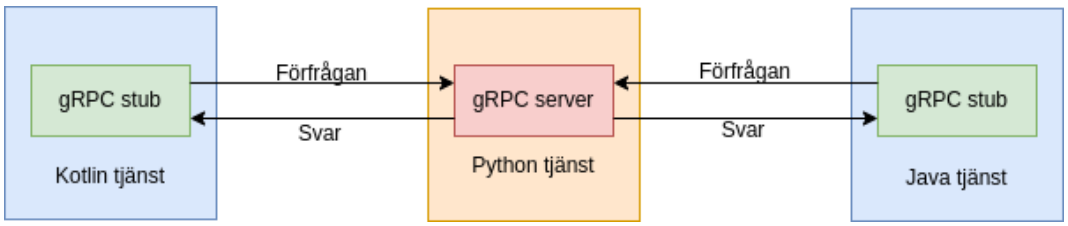
RPC (Remote Procedure Call) är en teknik för att kalla en funktion som körs på en annan tjänst. RPC är i grunden väldigt likt REST, men istället för att åberopa funktionalitet över kommunikation med hjälp av en API som är baserad på HTTP verben så använder sig RPC av ett gränssnittsdefinitionsspråk (IDL) för att definera kommunikationen mellan tjänster. RPC används för det mesta med hjälp av ett ramverk som implementerar RPC funktionaliteten. Ett populärt ramverk är *gRPC* som är skapat av Google. Ramverket gömmer den mera komplexa kommunikationen över nätverket och försöker få nätverksanropet att se ut som att en lokal funktion åberopas istället för en API med hjälp av HTTP som är implementerad på en extern tjänst.

Definitionen för gränssnittet i *gRPC* är inte beroende av vilket språk som används. Definitionen innehåller typsignaturer som sedan generar den kod som behövs för att åberopa och svara på en förfågan i form av så kallade ”*stubs*”.

Denna teknologi skapar ett beroende mellan tjänsten som kallar på funktionen och den tjänst som åberopas. I Googles implementation *gRPC* så kör den mottagande tjänsten en server som implementerar gränssnittet som kan ta emot förfrågningar från andra tjänster. Tjänsterna som skall kalla på denna tjänst implementerar de tidigare nämnda ”*stubs*” som innehåller samma funktionsdefinitioner som servern.

gRPC använder *”protocol buffers”* som är Googles egna implementation av ett binärt format för kommunikation men kan även använda JSON. Olika implementationer använder dels något binärt format eller ett textbaserat format (som exempelvis JSON) som skickas mellan tjänsterna.

RPC är heller inte bundet till TCP protokollet som exempelvis REST är, utan kan även använda UDP protokollet för kommunikation.



Figur 6: RPC kommunikation med Googles gRPC protocol buffers

3.3 Asynkron kommunikation

Aysnkron kommunikation över nätverk, till skillnad från synkron kommunikation, blockerar inte tjänster som kommunicerar med andra tjänster. Tjänsten kan fortsätta processen utan att vänta på svar. Det finns flera typer utav asynkron kommunikation som kan användas för kommunikation mellan tjänster:

Kommunikation genom gemensam data

Tjänsen modifierar gemensam data i en databas eller genom en gemensam fil i filsystemet som sedan en eller flera tjänster kan dra nytta av. Denna typ utav kommunikation är relativt ovanlig i praktiken[15][16].

Asynkron Förfrågan-Svar

Liknande som den synkrona kommunikationen så skickas en förfrågan från en tjänst till en annan. Tjänsten som skickar förfrågan kommer inte att blockera i väntan på svar utan fortsätter sin process. Detta kan implementeras genom bland annat en meddelandekö som sedan en annan tjänst konsumerar meddelanden ifrån[15][16].

Händelsestyrd kommunikation

Genom en meddelandeförmedlare så meddelar tjänsterna en händelse till en eller flera tjänster. Detta kan ske som *point-to-point* kommunikation, det vill säga från en tjänst direkt till en annan eller genom att skicka ett meddelande med en händelse som sedan flera tjänster kan ta del av[15][16].

3.3.1 Meddelanden

Ett meddelande är ett generellt koncept som definerar informationen en tjänst skickar till en eller flera tjänster. Meddelandet kan innehålla en förfrågan, ett svar eller en händelse. Istället för att en tjänst kommunicerar direkt med en annan tjänst så kan tjänsten skicka ett meddelande till en meddelandeförmedlare med information som konsumenterna(en eller flera tjänster) konsumerar[18].

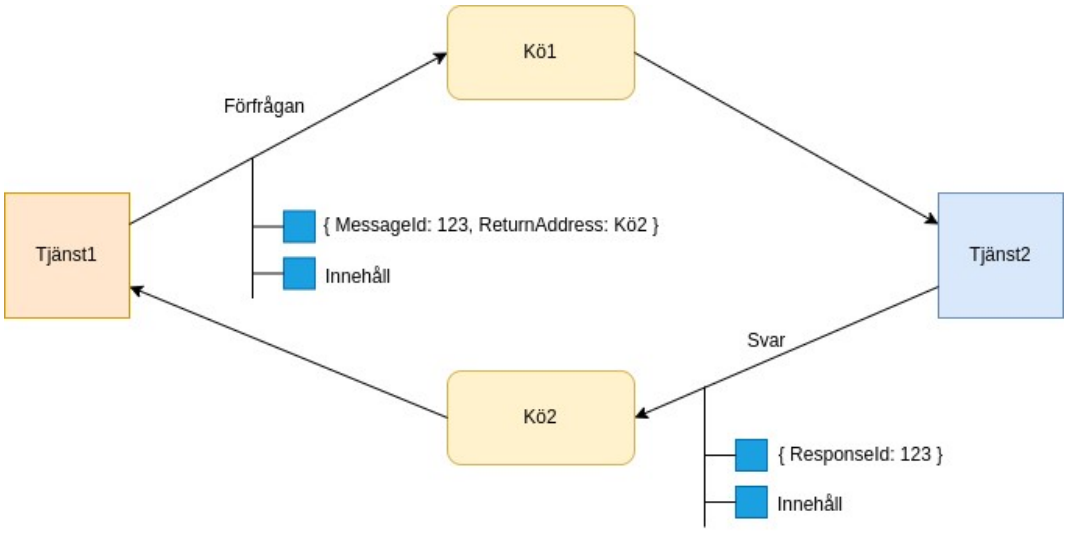
Ett meddelande består av två delar, rubrik (header) och innehåll (body). Rubriken innehåller metadata angående vad som skickas, ett unikt meddelande id, mottagaren av meddelandet och information om hur svaret skall returneras om ett svar krävs. Datan som skickas är i textformat eller binärformat[18].

Ett meddelande kan innehålla en förfrågan som skall åberopa en funktionalitet med dess parametrar eller så kan meddelandet vara en händelse som representerar en förändring i en mikrotjänst. Detta kan exemepelvis vara en förändring i databasen eller dylik händelse som andra tjänster kan vara i behov att agera på[18].

3.3.2 Meddelandeförmedlare

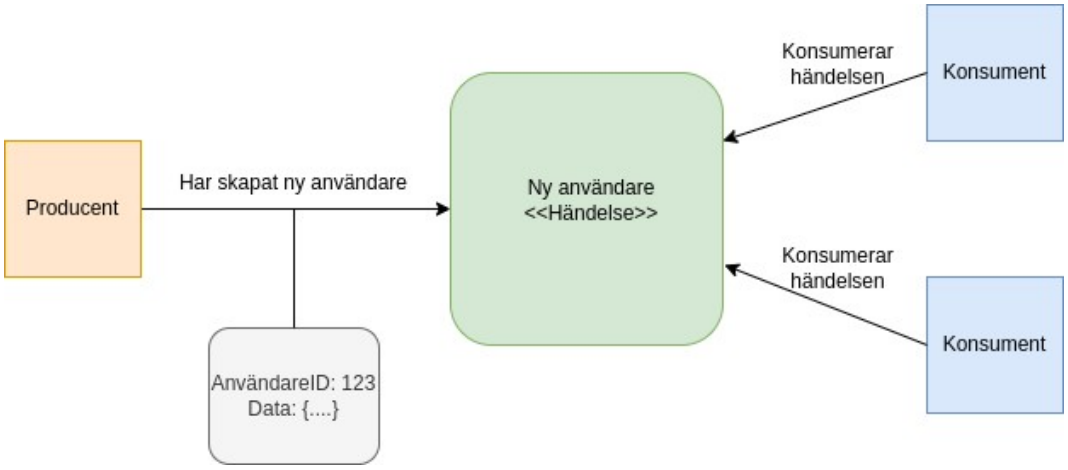
Kommunikationen av asynkron förfrågan-svar kommunikation och händelsestyrd kommunikation implementeras oftast med en meddelandreförmedlare för att styra kommunikationen mellan tjänsterna. Meddelandeförmedlare är så kallad ”*middleware*” som sköter kommunikationen mellan processer. De är ett populärt val för att sköta asynkron kommunikation mellan mikrotjänster då de erbjuder olika funktioner[15][16].

Meddelandeförmedlare har ofta två olika funktioner, *publish-subscribe*(*ämne*) eller *queue*(köer). Köer är oftast så kallade *point-to-point*, det vill säga att en tjänst kommunicerar direkt med en annan genom meddelandeförmedlaren. En sändare skickar ett meddelande(en förfrågan) till en kö och en specifik konsument konsumerar detta meddelande. En stor skillnad mellan köer och ämnen är att ett meddelande som skickas genom en kö till en konsument innehåller informationen angående vilken konsument som skall konsumera meddelandet. Detta innebär att tjänsterna måste veta vilken tjänst som skall konsumera meddelandet i förväg, vilket innebär att det finns en viss grad av beroende[15][16][18].



Figur 7: Förfrågan-Svar kommunikation med köer, point-to-point

När tjänster kommunicerar genom *publish-subsribe* mönstret så utsänder sändaren ett meddelande till meddelandeförmedlaren som innehåller en händelse. När en händelse skickas så meddelas ett fakta, någonting har hänt och de andra tjänsterna skall reagera på denna händelse. Det kan exempelvis röra sig om att en ny användare har skapat ett konto genom en tjänst och denna information används sedan av ytterligare tjänster för att genomföra andra åtgärder. Fördelen i jämförelse med köer är att tjänsterna är helt obundna. Tjänsterna som använder denna modell utsänder oftast meddelanden till ett ämne(topic) som andra tjänster konsumerar. En tjänst som utsänder en händelse till ett ämne behöver inte veta vilka tjänster som skall konsumera denna händelse. Man överför därmed ansvaret till de andra tjänsterna att agera på händelserna[15][16].



Figur 8: Kommunikation med hjälp publish-subscribe mönster med ett ämne

4. Struktur och designmönster av mikrotjänster

5. Migrera från Monolit till Mikrotjänster

6. Skalning

7. Distribution

8. Diskussion

Källförteckning

[1] Richardson Chris, Microservices patterns (s.37), Manning Publications, 2019.

[2] Newman Sam, Building microservices (s.14-18), O'Reilly, 2021.

[3] Migrera ett monolitiskt program till mikrotjänster med domändriven design, Microsoft Azure Dokumentation, Hämtad 17.03.2022 från: https://docs.microsoft.com/sv-se/azure/architecture/microservices/migrate-monolith

[4] Newman Sam, Building microservices (s.22-26), O'Reilly, 2021.

[5] Agile Software Architecture : Aligning Agile Processes and Software Architectures, edited by Muhammad Ali Babar, et al., Elsevier Science & Technology, 2013

[6] D.L Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, D.L. Parnas, Carnegie Mellon University, 1972. Hämtad 17.03.2022 från: https://www.win.tue.nl/~wstomv/edu/2ip30/references/criteria_for_modularization.pdf

[9] Dewayne E. Perry and Alexander L. Wolf, Foundations for the Study of Software Architecture, 1992, Hämtad 17.03.2022 från: http://users.ece.utexas.edu/~perry/work/papers/swa-sen.pdf

[10] Ponce Mella, Francisco & Márquez, Gastón & Astudillo, Hernán. Migrating from monolithic architecture to microservices: A Rapid Review, 2019, Hämtad 17.03.2022 från: https://www.researchgate.net/publication/335716451_Migrating_from_monolithic_architecture_to_microservices_A_Rapid_Review

[11] IBM Cloud Education, *SOA (Service Oriented Architecture)*, 2021. Hämtad 17.03.2022 från: https://www.ibm.com/cloud/learn/soa

[12] Kim Clark, *Microservices vs. SOA: How to start an argument,* IBM Integration Community, 2020. Hämtad 17.03.2022 från: https://community.ibm.com/community/user/integration/viewdocument/microservices-vs-soa-how-to-start?CommunityKey=77544459-9fda-40da-ae0b-fc8c76f0ce18&tab=librarydocuments

[13] Newman Sam, Building microservices (s.22), O'Reilly, 2021.

[14] Microsoft Azure Dokumentation, *Använda API-gatewayer i mikrotjänster,* Hämtad 17.03.2022 från: https://docs.microsoft.com/sv-se/azure/architecture/microservices/design/gateway

[15] Newman Sam, Building microservices (kapitel 5), O'Reilly, 2021.

[16] Newman Sam, Building microservices (kapitel 4), O'Reilly, 2021.

[17] IBM Cloud Lean Hub, *What is an ESB,* Hämtad 29.03.2022 från: https://www.ibm.com/cloud/learn/esb

[18] Richardson Chris, Microservices patterns (kapitel 3), Manning Publications, 2019.