Assignment lab 01

YHL-18 CMT
By: Henrik Ählström

Task 1:

- **Iaas** – Infrastructure as a service – En instans som hyrs av en provider .Till skillnad från Paas och Saas har man I Iaas möjlighet att konfigurera/justera allt från applikation till nätverk. Providern har hand om hårdvaran endast I princip.
- **Paas** – Platform as a service – En platform hyrd av en provider där möjlighet finns att deploya en egenutvecklad eller anskaffad applikation. Möjlighet att konfigurera sträcker sig till det som krävs för att hosta applikationen.
- **Saas** – Software as a service – En mjukvarutjänst som körs på en providers infrastruktur. Inga möjligheter till konfigurering/justering på något utom möjligtvis användare och vissa inställningar I själva mjukvarutjänsten.
- **OS-level virtualization** – Containers. En typ av virtualisering där varje instans delar kärna med det operativsystem där den virtuella instansen körs.
- **cgroups –** En kernelfeature I Linux kärnan. Används för att begränsa, fördela och isolera hårdvaruresurser för processer.
- **CoW & snapshot –**
  CoW – Vid modifiering av data(tex en fil) skapas en kopia, på så sätt är orginalet alltid tillgängligt.
  Snapshot – En snapshot sparar ett systems eller en disks tillstånd vid en given tid. Denna snapshot kan användas för att återställa systemet/disken till det tillstånd det var I när snapshoten skapades. Filsystem som ZFS och volymhanterare som LVM har denna feature inbyggt. Kan användas som en typ av backup(åtminstonde som fail-safe tex vid förändringar I konfiguration, går det fel återställer man snapshoten och allt är som det var innan gjorde förändringar)
- **High availability –** Syftar att säkerställa tillgänglighet av ett system eller applikation genom att eliminiera single-point-of-failure och/eller implementera self-healing(uppstart av ny instans om en instans går ner) och med det minimera risken för nertid.
- **Idempotency –** Samma operation kan genomföras flera gånger med samma resultat. Tex: ta bort en rad med ett unikt ID från en databas, första gången operationen utförs tas raden bort, följande gånger må svaret från databasen se annorlunda ut (en rad med det IDt existerar inte) men resultatet är det samma(raden är borta).
- **Mutable vs Immutable infrastructure :**
  Mutable – Kan förändras efter att den skapats dvs förändringar appliceras på en existerande konfiguration(på själva target servern). En fördel är tex att man slipper förflytta data om det är en stateful applikation, nackdelen är risken för nertid om konfigureringen går snett.
  Immutable – Konfiguration appliceras inte på en existerande konfiguration, en ny instans skapas istället som den nya konfigurationen appliceras på, när detta är klart pekas trafiken till den nya instansen och den gamla avvecklas.

- **Configuration management vs orchestration :**
  Configuration management – För att hantera konfiguration och mjukvara på existerande servrar och snabbt kunna ändra state på servern. Kan göras på minst en server och görs ofta repeterat. Exempel på mjukvara: Chef, Ansible, Puppet. (Verktyg av denna typ har ingen historik om hur den existerande infrastrukturen ser ut) till skillnad från orchestration har config management endast hand om konfigurationen.
  Orchestration – Jag tänker att I sammanhanget syftar orkestrering på att kordinera flertalet instanser samtidigt, om vi tar kubernetes som exempel: Kubernetes har till skillnad från ett config management tool inte endast hand om konfiguration, utan även instansers blotta existens(dvs starta och stoppa instanser) samt saker som det nätverkets infrastruktur och dess samspel med instanser. Jag hävdar att config management är en del I orkestrering, man kan se orkestrering som mer omfattande än config management.
- **Procedural vs Declarative :**
  Procedural – Här måste exakta instruktioner I processen defineras för att uppnå önskat resultat. Tänk: *"Ge mig det här genom att gå dit och ta den där och gör så här"*.
  Här defineras alltså vad man vill ha och hur det ska göras.
  Declarative – Man definerar ett mål eller önskat resultat(tex en server konfigurerad så här och så här), men inte vilka steg som ska tas för att få resultatet. Tänk: *"Ge mig det här"*.
  Här defineras alltså endast vad man vill ha.
- **Git submodule –** Används för att inkludera ett git repository som ett directory I ett annat git repository(tex inkludera ett visst bibliotek I ens eget projekt), detta för att kunna behandla dessa repositories som 2 separata projekt men kunna använda dom med varandra.
- **Ansible :**
  - **Inventory –** Här deklareras de servrar Ansible har hand om, kan vara enstaka servrar eller grupper. Inventory kan byggas statiskt av administratörer eller dynamiskt med hjälp av tex scripts/python program.
  - **Playbook –** Här deklareras tasks Ansible ska utföra, på vilka targets det gäller och vilken/vilka användare som ska utföra detta.
    Enkelt förklarat: "På grupp X, ladda ner <paket> med apt och kopiera configfil Y till detta"
- **Kubernetes :**
  - **Stateful vs stateless applications:** Refererar till om en applikations arkitektur är stateful eller stateless. Stateful betyder att det finns data som ska vara persistent(behöver sparas och åtkomlig I ett senare skede), dvs om applikationen startas om ska datat finnas tillgängligt I samma skick det var innan omstart.
    Om applikationen är stateless är den inte beroende av persistent data för att fungera, inte heller skriver applikationen data som behöver vara persistent.

- **ReplicaSets,Deployments, Pods & Services :**
  - **Replicasets** – Kontrollerar hur många replikor av en pod som ska köras och att detta antal replikor alltid upprätthålls.
  - **Deployments –** Har hand om uppdateringar/förändringar på pods och replicasets på ett deklarativt sätt. Exempel på uppgifter: Uppdatering av pods till ett definerat desired state, roll-back om uppdatering misslyckas eller systemet blir ostabilt och utför status(health checks) på pods.
  - **Pods –** En pod är en eller flera containers som är relevanta för varandra och som grupperats ihop. Tex: En container som kör en tjänst och dess “sidekick” container vars uppgift är att understödja “main containern” I någon form tex en databas eller insamling av loggdata. Containers I en pod delar lagring, nätverk(IP och portar), cgroup och namespace.
  - **Services –** Syftar till kommunikation. Exponerar deployments/pods utåt samt möjliggör inter-pod kommunikation. En service är också den komponent som har hand om IP adressering, portar och DNS records.
    Detta hanteras dynamiskt för att säkerställa att en deployment kan upprätthållas även om en pod skulle restarta eller gå ner.
    Det vill säga om vi har en färdig deployment med DNS records och IP adresser på plats, alla hittar alla, men så blir en pod omstartad eller dör, risken finns att DNS och IP adresserna inte stämmer längre, en likadan pod existerar, men det är inte samma.
    En service uppgift är då alltså att säkerställa att även om podar går upp/ner dynamiskt, ska de fortfarande kunna kommunicera.

Task 2:

Subtask 1:

Question 1: Why might you want to make a multi-stage build

Svar:

- Separera källkod och kompilerad/byggd kod.
- Minimera imagens storlek genom att undvika att alla saker som krävs för byggandet (kompilator, libs, dependencies) följer med I produktions imagen.

Subtask 2:

Question 2: Mounting database dumps like this can be a good solution to use for a test database, but why might you not want to do it like this for a production database.

Svar:
Dels måste man bygga om imagen varje gång en ändring har skett I sql filen samt om man inte har någon typ av versionshantering på sql filerna känns det som en överhängande risk att en gammal databas blir deployad.
Sedan krävs det ju att man gör en mysqldump, vilken kan betyda att databasen man tar dump ifrån måste låsas, kan bli knivigt I en prodmiljö.

LabAssignment1 - Task 5: Essay
Av: Henrik Ählström
CMT YHL-18

Microservices

Med microservices menar man en utevecklingsteknik där en applikation bryts ner till flera självständiga och oberoende tjänster, där varje tjänst har en viss funktion.
Notera dock att micro inte behöver referera till att servicen ska vara liten, utan snarare att den är decentraliserad och distrubierad.
Detta till skillnad från en monolitisk applikation som agerar som en hel entitet, den innehåller alla funktioner och alla delar av applikationen inom ett och samma omsvep.
Microservices är inte att förväxla med en uppsättning monolitiska applikationer som är tight integrerade, detta är inte microservices utan består som sagt av monolitiska applikationer även om de skulle kunna misstas för microservices.

Dessa microservices kommunicerar med varandra, ofta med APIs över HTTP eller via meddelande(med en message broker exempelvis) och utgör tillsammans en applikation(tjänst).

En stor fördel kontra monolitiska applikationer är **agilitet.** Microservice arkitekturen är mycket mer agil och flexibel vad det gäller utveckling, val av programmeringsspråk, CD(Continuous Deployment) samt flexibel skalning.
Varje service kan bli mindre mödosam att utveckla(tex implementera features) och man kan skriva varje service I ett språk som bäst lämpar sig för ändamålet och inte behöva kompromissa och välja ett språk som “ish passar ändamålet men det blir jobbigt därför att …….”, detta kan dock även vara en fälla, att ha varje service skrivet I ett eget språk lär göra att det blir svårt att få en överblick.
En monolitisk applikation blir ofta krävande I en CD(Continuous Deployment) implementation, tex vid varje förändring måste hela applikationen kompileras/byggas och deployas på nytt, medans man I microservice tekniken endast behöver göra detta på en per-service basis där varje enskilda service troligtvis är mindre än motsvarande monolitiska applikation.
Allt detta tack vare att man jobbar med en mindre avknoppning av en applikation som bör vara så självständig som möjligt från övriga services.

Flexibel skalning vill jag säga sammanhänger med *resurseffektivitet* då I vissa fall kanske inte alla tjänster behöver skala ut, det kanske finns vissa tjänster som är mer belastade än andra, med microservices flexibla skalning kan man då endast skala de belastade delarna.
Jämför man detta med monolotiska applikationer är fallet annorlunda, det enda sättet att skala ut en sådan är att deploya flera instanser av applikationen, vilket om man jämför med exemplet för microservices kan betraktas som onödigt ockuperade resurser på grund av att man skalar även de delar I applikationen som inte har ett behov av att skalas.

En annan faktor till microservices fördel är **enklare underhåll.** Jämfört med en stor monolitisk applikation som består av ett betydande antal rader kod och med det kan vara svår att underhålla, bugfixa och implementera features på, särskilt för nya rektryter tänkta att jobba med applikationen, bristen på tydliga avgränsingar gör arbetet mödosamt, vad är egentligen vad I koden?
I och med att applikationen även måste byggas på nytt vid varje förändring kan det ta tid innan man får ett kvitto på om exempelvis en bugfix eller dylikt löste det problem den skulle.
Jag tänker ett mardrömsscenario där en stor applikation som med grädde på moset även är urkasst dokumenterad, leder till att den överhängade risken att introducera nya buggar vid förändringar I koden gör sig till känna. CI? Javisst, men återigen, applikationen måste byggas varje gång, vägen från “build now” till kvitto präglas av fördröjningar.
I och med att man I microservices arbetar med en del av applikationen kan detta minimera dessa negativa aspekter.

En nackdel med microservices är dess komplexitet. I och med att applikationen är uppdelad och distrubierad kan saker som kommunikation processerna emellan bli mer komplexa än en monolitisk applikation, felhantering lär också behövas läggas till, tänk: hur ska en tjänst agera om dennes request till en annan service timar ut tex?
En annan nackdel är risken att behöva **uppdatera på flera ställen.** I fall där services har någon sorts beroende till varandra kan det bli tidsödslande att uppdatera, dvs om du gör en förändring på service X som även påverkar service Y osv. Det här understryker vikten av att hålla varje service så självständig och oberoende som möjligt.

Det är inte alltid som en microservices är en fördel, fördelar och nackdelar bör noga övervägas vid val av teknik, tex: Vad vinner vi på att använda microservice kontra monolitisk? Hur ser det ut på sikt, är applikationen tänkt att växa så det finns risk för det så kallade *monolith hell* I framtiden, kan vi förutse det redan nu? Kan vi använda någon microservice till fler än en sak?

Use case varierar. ett exempel kan vara om man tillhandahåller en tjänst som består av flera icke-relaterade funktioner, dessa funktioner kan då delas upp till varsin tjänst enligt microservice principen.

Microservicearkitekturen implementeras med flera design mönster, nedan beskrivs de 2 stycken jag tycker var mest intressant, motiveringen till mina val:

1. Jag har tidigare nämnt vikten av att microservices är oberoende(för oberoende skalning och enklare underhåll), exemplet med databaser är en av flera saker som bidrar till detta.
2. Hur hittar services varandra I en ever-changing miljö?

Databasens plats I en microservice miljö,

Vissa delar av en microservice kan vara beroende av att läsa in data för att fungera, eller så är tjänstens uppgift att hämta och vidarebefodra data vidare till en annan tjänst, I vilket fall kan denna data vara lagrad I en databas.

Om man endast inkluderar en databas(monolit) I sin miljö som alla tjänster använder kan saker som table locking vara ett problem.
Table locking betyder att sessioner har ensamrätt till att använda en tabell så länge de behöver, andra sessioner har då inte tillgång till samma table(det är låst) tills sessionen med ensamrätt låst upp tabellen igen.
Problemet beror mycket på vilken databasmotor man använder, vissa motorer använder sig av row based locking istället för att ge möjlighet till bättre samkörning, men I fall där man använder en databasmotor utan stöd för row-based locking, kan detta bli ett problem. [1]

Ett annat problem är att I en miljö med en monolitisk databas har vi skapat oss ett enorm beroende samt en single-point-of-failure.

Det bör tilläggas att en monolitisk databas som flera services delar inte på något sätt är omöjlig, jag menar att monolitisk vs distrubierad databas beror på use case samt databasens storlek.

Så hur kan man göra I en microservice implementation?

Här finns flera alternativ:

Ett alternativ är att man ger ett eget DBMS till varje microservice. Fördelen med det är att man kan välja typ av databas utifrån berörd microservice behov, dvs ha en SQL databas för vissa microservices och NoSQL för andra.
Man kan också ge varje microservice ett DBMS med en databas som endast innehåller den tabell servicen behöver. Nackdelen är om joins som sträcker sig över flera tabeller behöver göras, dessa kan bli kostsamma.

Vid större implementationer kan man implementera databas sharding I syfte att skala. Vilket betyder att en tabell delas upp(ofta I en range av rader) och placeras på individuella noder.
En nackdel med sharding är att man på ett eller annat sätt måste se till att applikationer “vet om” den underliggande databasstrukturen.

En annan nackdel är att databasen I och med att den är distrubierad därmed blir komplex, samt att korrupt data kan förstöra hela tabellen.[2]

Något man bör ha I åtanke vid en distrubierad databas är att joins som sträcker sig över flera shards/partitioner/distrubierade tabeller kan vara kostsamma, I sådana fall man bör försöka arbeta bort behovet av join queries från sina applikationer.

Ett annat design mönster är **service discovery**. I och med att microservices nästan utestlutande deployas som virtuella instanser I någon form av dynamisk miljö tex Docker containers eller instanser I AWS cloudtjänst, där services kan uppstå och försvinna per automatik(tex vid skalning för att matcha worload) måste man kunna garantera kommunikation mellan dessa, med andra ord services måste veta varandras IP nummer och port för att kunna kommunicera.
Att hårdkoda in en IP/URL I koden är dålig practice.
Detta är inte ett problem I monolitiska applikationer, allt deployas på en och samma instans, även om monolitiska system också kan ha problem att hitta externa resurser (tex databaser) pga att dessa har dynamiska IP nummer osv, men det är utanför scopet för applikationens arkitektur.

Detta begrepp kallas service discovery och har två modeller:

- Client-side discovery
- Server-side discovery

Notera: Klient betyder här microservice.

Dessa två utesluter varandra då de båda syftar till att lösa samma problem med olika approaches. De har dock en sak gemensamt, de använder sig av ett service registry, en databas med information om vart services finns.
Services informerar registret som noterar vart de finns. Sedan kan andra services fråga registret om vart andra services återfinns. [4]

I Client-side discovery ligger ansvaret på klienten att ta reda på vart remote servicen finns. Innehåller två calls, ett till service registry där registret svarar med efterfrågad services information och efter det ett till servicen den vill kontakta.
Den största nackdelen här är dels att två calls behövs, kan I stora miljöer med flera replikor av samma service skapa mycket trafik och varje call sker över nätverket vilket är mer kostsammt än ett lokalt call. För det andra behöver man implementera själva service discoverien I applikationens kod.

I Server-side discovery modellen kommunicerar klienter med en mellanhand (tex en lastbalanserare eller router) där denne mellanhand sedan gör uppslaget mot service registret och forwardar trafik vidare.
Det första som slår en när man kollar på det här är: Jaha? Vad löser det här för problem? Du gör lika många calls med skillnaden att det är en mellanhand som gör det istället för klienten.
Sant men kanske inte, här spekulerar jag fritt, men varför inte lägga registriet tillsammans med lastbalanseraren/routern? Då slipper vi ett extra remote call åtminstonde.
En fördel är dock att vi inte behöver anpassa koden I våra applikationer.

Viss mjukvara har redan löst service discovery åt oss, vilket gör det enklare att implementera, tex Kubernets som rekommenderar att man kör “deras” DNS baserade service discovery. Services slår alltså upp FQDN – Fully Qualified Domain Name och hittar varandra på det sättet. [5]

Ett nyckelord man bör ta med sig från denna läsning om microservices är planering. Planering är nyckeln till en lyckad implementation. Vid första anblick har microservices fler fördelar än nackdelar, vilket till stor del är sanning.
Men man ska väga fördelar och nackdelar mot varandra, titta på sin nuvarande situation: Utvecklar vi nytt? Ska vi migrera nuvarande applikationer till microservices eller ska vi vänta tills applikationen är ansedd utdaterad och bygga en helt ny och då använda microservice arkitekturen? Vad vinner vi på microservices? Vad är företagsnyttan?

Frågorna är många, möjligtvis svaren likaså, men helt klart är microservices ett intressant ämne som varken bör hamna I skymundan men inte heller sväljas med hull och hår vid val av arkitektur.

Källor:

[1] https://dev.mysql.com/doc/refman/8.0/en/table-locking.html
[2] https://www.digitalocean.com/community/tutorials/understanding-database-sharding
[3] https://microservices.io/patterns/data/database-per-service.html
[4] https://microservices.io/patterns/service-registry.html
[5] http://kubernetesbyexample.com/sd/