Utveckla med Microservices

Mikael Svahnberg*

2024-03-15

1 Introduktion

- Thomas & Hunt: Kapitel 5, Bend or Break
  - Topic 28 Decoupling
  - Topic 29 Juggling the Real World
  - Topic 30 Transforming Programming
  - (Topic 31 Inheritance Tax)
  - Topic 32 Configuration
- Thomas & Hunt: Kapitel 6, Concurrency
  - Topic 33 Breaking Temporal Coupling
  - Topic 34 Shared State is Incorrect State
  - Topic 35 Actors and Processes
  - Topic 36 Blackboards
- Microservice-arkitekturer
- Kommunicerande Microservices, REST-API:er.

2 Principer för Distribuerade Arkitekturer

Underhållbarhet Separata moduler som kan underhållas var och en för sig

Skalbarhet Separata exekveringsenheter som kan skala upp och ner efter behov

Vertikal Skalbarhet Mer minne, snabbare processor, fler processorkärnor, mm.

Horisontell Skalbarhet Fler noder

Tillförlitlighet Separata exekveringsenheter som kan "ta över" vid behov

*Mikael.Svahnberg@bth.se

3 Cloud Computing

Cloud Computing för utvecklare:

(Illusionen av) Oändliga Resurser Lätt att skala horisontellt

Elasisk Skalning Skala upp vid behov, men också skala ner när du inte längre behöver resurserna

Betala bara för det du använder (Metered Billing) Det upplevs som "Billigt" att skala upp (det är det inte egentligen)

Automatiserat Verktygsstöd för att skala.

Cloud Computing för moln-leverantör:

Delade Resurser Flera användare delar på gemensam hårdvara.

Virtualisering Möjliggör delade resurser

Billig Hårdvara Flera billiga enheter som går sönder snarare än få dyra enheter med garanterad drift-tid

Notera att de första principerna *möjliggör* cloud-applikationer, medans de senare *begränsar* lösningarna.

- Det är billigt att skala, men du måste utveckla feltoleranta applikationer.

4 För- och Nackdelar med Distribuerade System

Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: a systematic mapping study. In CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science, SciTePress, Setúbal, 2018.

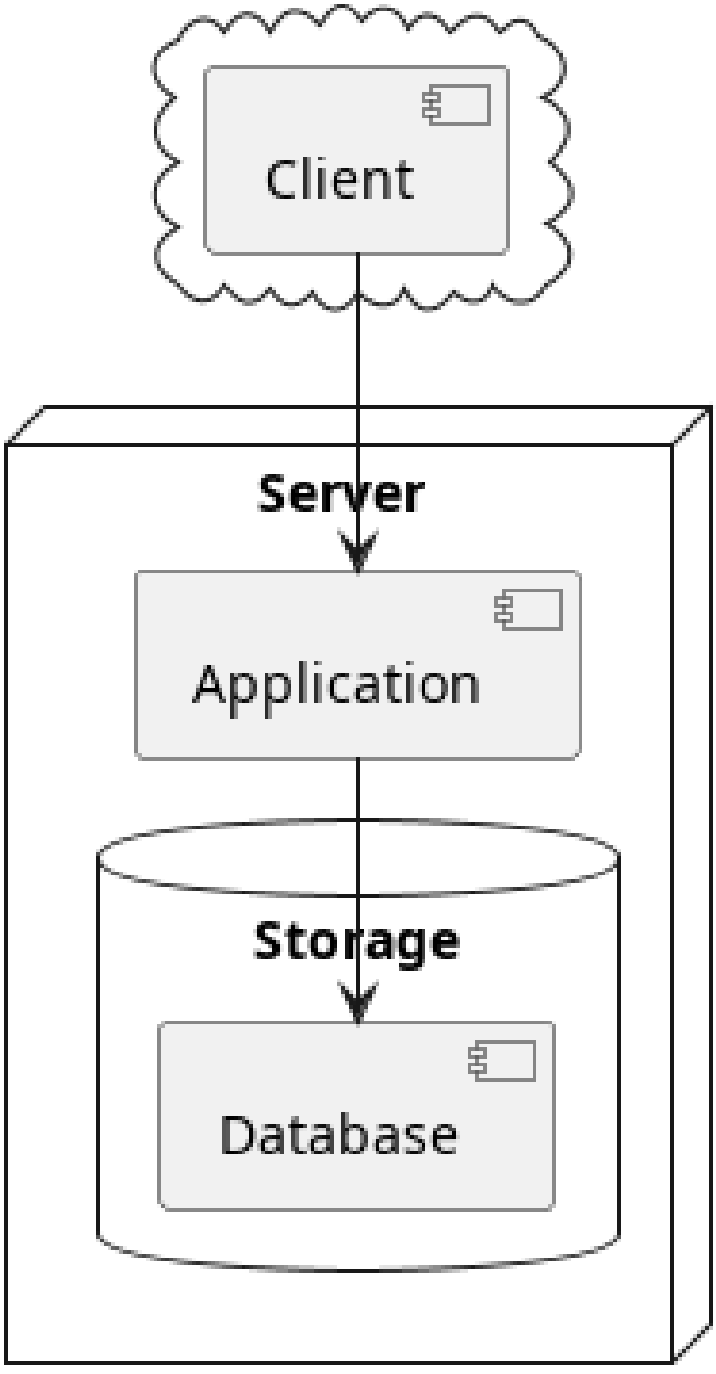
Fördelar

- Lätt att Underhålla
- Lätt att Återanvända enskilda komponenter
- Lätt att Skala pga. isolerade komponenter
- Lätt att Driftsätta
- Lätt att Övervaka varje enskild del/komponent i systemet
- Lätt att Starta om komponenter (Driftsäkerhet)
- Kräver en robustare design där enskilda komponenter kan utveckla fel
- Möjligt att använda olika teknologier och programspråk för varje komponent
- Kan (i teorin) skalas oändligt

Nackdelar

- Ökad komplexitet för testning
- Större utvecklingsansträngning, kräver mer erfarna utvecklare.
- Nätverksrelaterade frågor kräver större uppmärksamhet
- Mer invecklad användar-autentisering
- Större behov av verktyg för automatiserad driftsättning

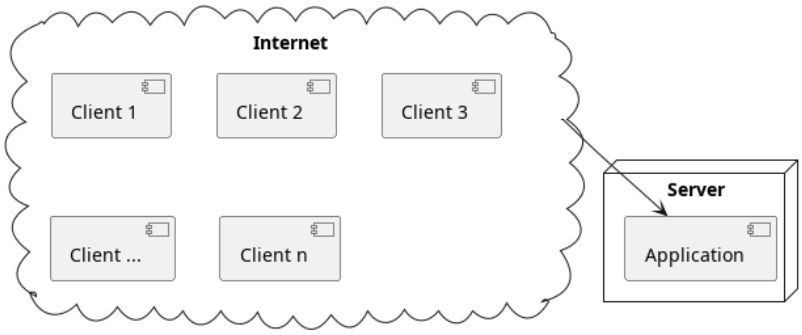
5 Grundläggande Distribuerad Arkitektur



- *Layered Architecture Style*

- Man kan bygga detta som en skrivbords-applikation, med alla tre delar i samma binär
- `Storage/Database` kan ligga på en separat maskin
- `Application` kan bestå av många samarbetande komponenter
- `Client` kan vara ett egenbyggt gränssnitt, eller (vanligare idag) en webklient

6 Kommunikation mellan Komponenter



- Socket-Programmering
- Remote Procedure Calls
- REST API
- Databas/Filsystem

6.1 Socket-programmering:

- `bind()` → `listen()` → `accept()`
- Behöver hantera samtidiga uppkopplingar
  - Snabbt lösa uppgiften så du kan komma tillbaka till `accept()` igen
  - Skapa en ny tråd för varje uppkoppling så du snabbt kan gå tillbaka till `accept()`
- Behöver själv hantera fel, kommunikationsprotokoll, osv.

6.2 Remote Procedure Calls (RPC)

- Ser nästan ut som ett vanligt funktionsanrop
- Protokollet färdigt
- Felhantering inbyggt
- Svårt att felsöka (måste förstå protokollet)

6.3 *REST-API*

- *Representational State Transfer*
- Enhetligt protokoll: *HTTP* och *JSON*
- Tillståndslöst, varje uppkoppling/begäran hanteras för sig
- Löst Kopplat:
  - Klienter behöver bara känna till adressen till servern
  - Servern behöver inte veta något om klienterna
  - Varje begäran är en ny uppkoppling
- Inte längre transparent: Särskild kod behövs för att anropa andra delar av applikationen
- Enkelriktad kommunikation `Client` → `Server`

6.4 Databas/Filsystem

- Vanligt om man har långsamma jobb där klienten inte vill låsa sig i väntan på ett svar
- Alla hittar till databasen, kan använda den som en gemensam anslagstavla

7 Mer om REST-API

- REST är inte en formell standard, det är mer en allmän överenskommelse
- Upprepar:

  Enhetligt protokoll HTTP och JSON (mer om detta)

  Tillståndslöst Inget delat tillstånd mellan olika anrop; ingen förväntad ordning på anrop.

7.1 HTTP och JSON

HTTP

- `POST`, `GET`, `PUT` (replace), och `DELETE`
  - Kallas också CRUD – Create, Read, Update, Delete
- Väldefinierat protokoll för svarskoder
  - ex. `200 OK`, `403 Forbidden`, `404 Not Found`, `418 I'm a teapot`, `500 Internal Server Error`, `503 Service Unavailable`

JSON

- Text-baserade objekt:

```
{
    field: "value",
    "other field": 42,
    nested: {
        anInt: 1,
        aFloat: 1.2
    }
    anArray: ["First", "Second", { val: "third" }],
}
```

7.2 Exempel

- Samma address kan ha flera funktioner:
  - GET `https://www.zombo.com/Users/1011`
  - POST `https://www.zombo.com/Users/1011`
- Resursen som delar av sökvägen:
  - GET `https://www.bth.se/utbildning/program-och-kurser/paasw/`
- Parametrar till en viss resurs:
  - GET `https://studentportal-old.bth.se/api/exams?fromDate=2024-03-21&toDate=2024-10-07&courseId=pa1458`

Svaren på de här frågorna kan vara vad som helst:

- ett tomt HTTP-svar
- ett JSON-objekt
- en HTML-sida
- en binär-fil
- ...

8 Arkitekturer för Skalbara Distribuerade System

(Kanske mer *tekniker* än arkitekturer i dess vanliga bemärkelse)
B Wilder, *Cloud Architecture Patterns*, O'Reilly, 2012:

- Horizontally Scaling Compute Pattern
- Auto-Scaling Pattern
- Queue-Centric Workflow Pattern
- MapReduce Pattern
- Node Failure Pattern
- Busy Signal Pattern

- Colocate Pattern
- Multisite Deployment Pattern
- Content Delivery Network (CDN) Pattern
- Valet Key Pattern

Taibi, D., Lenarduzzi, V., & Pahl, C. (2018) *Architectural patterns for microservices: a systematic mapping study.*:

- Orchestration and Coordination
  - Service Composition, API-Gateway
  - Service Discovery
    - Client Side Discovery (Workers Register themselves in gateway)
    - Server-Side Discovery (Load Balancer)
  - Hybrid – Combine service registry and gateway
- Deployment
  - Multiple Service Per Host
- Data Storage
  - Database-per-Service
  - Shared Database
  - Database Cluster

9 Horizontally Scaling and Auto-Scaling

Föredra att automatiskt skapa nya noder i stället för att skriva multitrådad kod

- Enkeltrådad kod är lättare att underhålla
- Behöver hålla koll på vad som kostar (fler trådar → kraftfullare CPU, fler noder → fler noder)
- Behöver hantera *tillstånd* i applikationen
- Behöver hantera *autenticering* i alla delar av applikationen
- Behöver hantera *tillförlitlighet* när enskilda noder försvinner
- Behöver veta hur platformen stödjer automatisk/programmatisk skalning

Från Taibi et al. (2018) lär vi oss också:

- API Gateway Pattern
  - Klienten begär en resurs från en gateway (“portal”)
  - Portalen skickar vidare uppgiften till rätt resurs/nod.

  - Ex. olika noder för Produktkatalog, Kundvagn, Rekommendationssystem

Det här kan lösas på i huvudsak två sätt:

- Client-Side Discovery Pattern
  - Klienten begär en resurs från ett register
  - Får en direkt-länk till resursen för all framtida kommunikation
- Server-Side Discovery Pattern
  - Alla resurser registrerar sig hos en lastbalanserar
  - Klienten begär en resurs från lastbalanserareen
  - Lastbalanseraren skickar vidare frågan till en ledig resurs
  - (Klienten vet aldrig direkt-addressen till resursen)
- Hybrid pattern
  - Klienten postar sin begäran på en "anslagstavla"
  - Alla resurser läser meddelande från samma anslagstavla
    * Den resurs som tror sig kunna lösa uppgiften tar ner den från anslagstavlan.

10 Queue-Centric Workflow

Låt applikationen kommunicera via meddelande-köer

- I stället för att hänga sig på ett anrop, lägg uppgiften i en meddelande-kö
- Lösare kopplad applikation
  - Komponenter behöver bara hitta till sin kö
  - Flera komponenter kan lägga meddelanden i samma kö
  - Flera komponenter kan plocka meddelanden från samma kö
  - Man behöver inte (kan inte) vänta på svar innan man fortsätter (Asynkron kommunikation)
- Robust med lite trixande; man behöver inte ta bort ett jobb från kön innan man är färdig
- Använd inte om man behöver svar direkt; bättre då med *REST* och en lastbalanserare.
- Jämför med Taibi's Hybrid-pattern med "Anslagstavlan"

11 MapReduce

Map *Tillämpa en funktion på varje element i en sekvens med data*

- En funktion från en mängd element till en annan mängd med element.

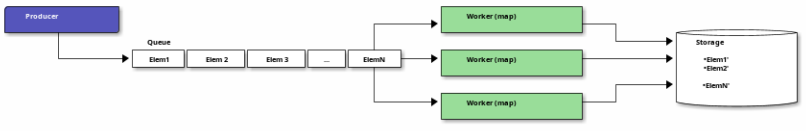
Reduce *Tillämpa en funktion för varje element och samla resultatet*

- En funktion från en mängd element till ett enskilt element.

```
            map                   reduce
(a b c)  -------> (a' b' c') ----------> X
```

- En gammal idé
  - finns i *lisp*-programspråk sedan 1950-talet
  - finns i moderna språk som JavaScript och Java också.
- Ofta ser man en sekvens av flera *map* -steg innan man (kanske) hakar på en *reduce*.
- *map* -steget är trivialt parallelliserbart; varje element kan behandlas var och en för sig.
- Paralelliseringen är användbar om man har tillgång till all data från början.
  - Man *kan* använda MapReduce på ett element i taget också, men det är mindre användbart.
- Kräver att datan och problemet är strukturerat så att man kan hantera varje element för sig.
  - En stor del av Big Data Analytics går ut på att förändra problemet så att MapReduce funkar.

12 Kombinera mönster



MapReduce Varje Worker-nod är en *map*

Queue-Centric Varje element hämtas från en kö

Horizontal Scaling Behövs fler Worker-noder kan man lätt skala upp; de är oberoende av varandra

Auto-Scaling Justera antalet Worker-noder baserat på storleken på Queue.

13 Felhantering – Node Failure och Busy Signal

- Grundprincipen är: *Varje nod skall kunna misslyckas när som helst*
  - Korollarium: Alla noder måste vara beredda på att alla andra noder misslyckas när som helst.

Arbetssätt:

- Spara data i persistent lagring (Databas) så snart det är färdigt
  - Markera inte jobbet som färdigt innan data är lagrat
- Behåll jobb i medelande-kön tills det är färdigt
  - Markera i stället att det är påbörjat av en Worker, med en tidsstämpel
- Överväg att ha ett antal redundanta noder som är beredda att ta över arbetet
- Om du kan (t.ex. vid planerat underhåll), låt noder flagga för att de inte är tillgängliga
- Ha en timeout på alla anrop mellan noder
- Ha en plan för vad som skall hända när du får en timeout på ett anrop
  - Försök igen
  - Vänta ett tag och försök sedan igen
  - Byt till en backup-nod och försök igen
  - Lita på att lastbalanseraren byter åt dig, och försök igen
- Skilj på noder som är upptagna `503 Service Unavailable` och noder som inte längre svarar alls.
  - Noder behöver programmeras så att de kan ge ett `503`-meddelande

14 Effektivisering – Colocate Pattern och Multisite Deployment

- *Noder som arbetar tätt tillsammans skall också driftsättas nära varandra*
- *Duplicera delar av applikationen för att komma närmre användaren*
- Gäller noder internt i applikationen, men också kopplingen användare $\leftrightarrow$ applikation.
- Kan finnas lagkrav att data om medborgare skall behållas inom landets gränser.
  - e.g. Schrems-lagarna inom EU
- Duplicering ger möjlighet till lastbalansering och driftsäkerhet.

- Synkronisering mellan duplicerade delar av applikationer ger krångligare programmering
- Extra utmaning är att automatiskt skala applikationen runt om klotet för att passa användningsmönster
  - t.ex. mer aktivitet tidiga kvällar, mindre under natten.

15 Mer Effektivisering – Content Delivery Network och Valet Key

Statiska resurser skall levereras av en statisk webserver

- Spara resurserna i din applikation till det som faktiskt behöver beräknas i realtid.
- Web-ramverk är sällan de bästa webservarna, det är bättre att ha en dedikerad nod för detta.
- Ibland skall innehåll bara vara tillgängligt för en viss användare eller en viss tid
  - Lås in det bakom en unik URL som bara existerar en viss tid eller för ett visst inlogg
  - Kan också låsas mot ett visst IP-nummer, eller en viss krypteringsnyckel
  - Kan också handla om en plats för att ladda upp innehåll, inte bara nedladdning
  - *Valet Key* - en särskild nyckel till bilen med begränsad funktionalitet

16 Datalagring

Taibi et al. (2018) beskriver tre huvudsakliga sätt att lagra data:

Database-per-Service Varje resurs ansvarar för sin egen data och väljer lämplig lagring för det

Shared Database Alla resurser delar på en central databas (med separata collections i den)

Clustered Database Alla resurser delar på en distribuerad databas, för att få bättre prestanda.

Ett begrepp till: *Database Sharding*

- Om man behöver dela på en databas kan man göra det på olika sätt
  - Vissa tabeller ligger i en databas, andra tabeller ligger i en annan
  - Vissa kolumner av en tabell ligger i en databas, andra kolumner ligger i en annan.
    - Löses "enkelt" med två tabeller och samma nyckel.
  - Vissa *rader* ligger i en databas, andra rader ligger i en annan.
- Sharding görs t.ex. för prestandaskäl eller lagkrav

17 Sammanfattning

- **Thomas & Hunt: Kapitel 5, Bend or Break**
  - Topics 28, 29, 30, och 32: Decoupling, Juggling the Real World, Transforming Programming, och Configuration
  - Löst kopplade komponenter är lättare att underhålla
    - Containers och distribuerade arkitekturer är synnerligen löst kopplade
    - *Microservice* - arkitekturer drar detta till sin spets; varje uppgift är en separat container.
  - Låt komponenterna *reagera* snarare än *leta efter arbete*
    - Ha ett tydligt gränssnitt, e.g. ett *REST-API* för att anropa komponenter.
  - Kan du se ditt program som en serie transformeringar av data?
    - Använd detta för att skapa skalbara system, e.g. med *MapReduce*
- **Thomas & Hunt: Kapitel 6, Concurrency**
  - Topics 33–36 Breaking Temporal Coupling, Shared State is Incorrect State, Actors and Processes, och Blackboards
  - *Tillstånd* skall hanteras försiktigt. Vi kan inte styra i vilken ordning vårt vackra API anropas.
    - Skriv programmet så att vilken ordning som helst är ok.
  - Varje Container kör till synes på sin egen processor och med sitt eget minne.
    - Skriv programmet så att man *kan* köra olika delar parallellt.
    - *Undvik att dela tillstånd*, låt varje container bara bero på just sin indata.
    - *Dela inte tillstånd* Undvik situationer där flera containers behöver uppdatera samma data samtidigt.
      - Dela *data* är såklart ofta nödvändigt, och då behöver du vara försiktig.
  - Utveckla *microservices* där varje container representerar en *Actor*
    - Kanske inte lika strikt på "varje aktör har en egen brevlåda"...
  - Delad kommunikation: *Blackboard*

18 Nästa Föreläsning

- Ett tolkat språk med rötter i funktionell programmering: `JavaScript`
- JavaScript som server-språk: `node.js`
- Ett rikt ekosystem: `npm`

Vi har väntat med detta för att kunna utveckla inuti en container

- Vanlig målplatform för node.js - applikationer idag
- Vi slipper installera mer programvara

Två resurser för att få en överblick över JavaScript och node.js:

- `https://javascript.info/`
- `https://www.freecodecamp.org/news/the-definitive-node-js-handbook-6912378afc6e/`

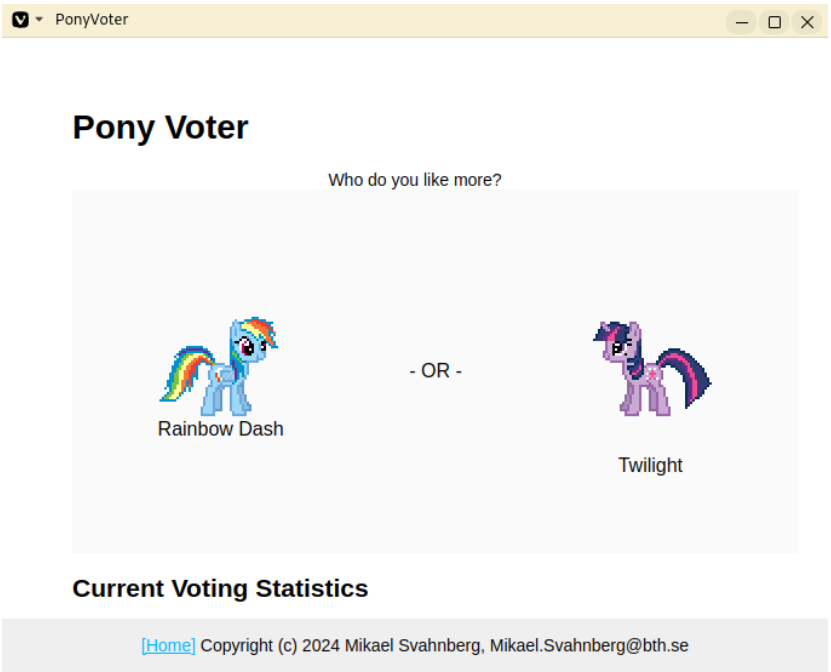
19 Övning: Microservices

19.1 Introduktion: PonyVoter

- Vi stannar i Equestria, men med en enkel röstningsapp den här gången.
- PonyVoter presenterar två alternativ åt gången, och man röstar genom att klicka på en av dem.
- Rösterna räknas i en databas, så att man över tid kan se vilken ponny som är mest populär.

Kom igång:

1. Ladda ner projektet: `https://codeberg.org/mickesv/PonyVoter.git`
2. Studera filerna, försök bilda din egen uppfattning om vad du har laddat ner.



19.2 Teknisk Översikt

- PonyVoter består av tre containers och en databas:

  PonyVoter "Framsidan" på applikationen som servar webbsidor till användarna

  VoteCounter Registrerar röster och sparar dem till databasen

  StatsPresenter Räknar ihop hur många röster respektive ponny har och sammanfattar detta

  MongoDB Databasen där rösterna lagras

- PonyVoter är hopplöst överdesignat och samtidigt underimplementerat
  - `VoteCounter` och `StatsPresenter` är extremt enkla, och hade antagligen inte behövt ha egna Containers i nuläget.
  - Mycket är hårdkodat.
  - För att inte kräva för många extra resurser används ingen renderingsmotor (såsom `Pug`) för att generera HTML-koden.
  - För att hålla projektet litet finns det bara sex ponnies att välja mellan.
  - Fullständighet? Bara det allra nödvändigaste finns implementerat.
  - Skalbarhet, vad händer när det totala antalet röster ökar?
  - Buggar! De finns såklart.
  - Säkerhet?

Fundera på:

1. Vilka containers skall vara tillgängliga för användaren?
2. Hur ser du till att bara dessa blir tillgängliga?
3. Hur kan du starta alla containers med ett enda kommando?

19.3 Starta och Testa

1. Filen `ponyvoter.yaml` används av `docker compose` för att bygga och starta applikationen.
   - Hur är den uppbyggd?
   - Vad finns angivet för varje container?
   - Är `volumes` - blocken nödvändiga? Vad gör de?
   - Kan du se hur man kommer åt respektive container?
2. Starta applikationen: `docker compose -f ponyvoter.yaml up`
3. Gå in på `http://localhost:8080` och testa applikationen
   - Håll ett öga på terminalen när du kör; vad skrivs ut?
4. Avbryt genom att trycka `Ctrl-C` i terminalen.

- Vad händer?
- Kolla med `docker images` vilka images som du har
- Kolla med `docker ps -a` vilka containers som körs respektive inte längre är igång

5. Starta igen (samma kommando)
   - Vad händer?
   - Notera att statistiken nollställs inte, trots att alla containrar startats om.
     - Varför?
     - Hur kan du ta reda på mer om detta?

19.4 Hitta Databasen

1. Kontrollera vilka volymer som docker har skapat `docker volume ls`
   - Det borde finnas två med långa icke-namn, ex. `aa5972d833f74bc8085bafdc32aa279e45c8d29cf63`
   - Kan det vara dessa som är databasen?
   - går det få mer information? `docker volume inspect aa5972d833f74bc8085bafdc32aa279e45c8`
2. Gå bakvägen. `docker ps -a` visar att databasen heter `ponyvoter-mongodb-1`
   - Vad får du för information från `docker inspect ponyvoter-mongodb-1`?
   - Leta efter "Mounts" i utskriften, eller filtrera lite först: `docker inspect -f '{{.Mounts}}' ponyvoter-mongodb-1`

Vår misstanke stämmer alltså. MondoDB använder alltså två volymer:

- `/data/configdb` och `/data/db` .

Uppgift:

1. Läs på om *Volumes* i dokumentationen till docker compose.
2. Modifiera `ponyvoter.yaml` så att `mongodb` använder två *namngivna* volymer; `db-data` och `db-config`.
3. Rensa bort de båda gamla med `docker volume prune` .

19.5 Skala applikationen

- Eftersom alla containers håller sig till REST-principerna, så går det enkelt att skala.
- I `ponyvoter.yaml` kan man ange hur många `replicas` en viss service skall ha i en viss driftsättning.
  - Det är lite mer invecklat än så; läs på i den officiella dokumentationen först.

Att göra

1. Uppdatera `ponyvoter.yaml` så att den driftsätter 3 st replicas av `votecounter`.
2. Starta om applikationen.
3. Rösta på ett antal ponnys och håll koll på terminalen: vad händer?
   - Finns det någon ordning i hur dina tre replicas används?
4. Fundera på:
   - Kan du ändra till 5 replicas *utan* att starta om applikationen? Hur? Prova!
   - Tips 1: Räcker det att ändra i yaml-filen?
   - Tips 2: `up` tar flaggan `--detach`
   - Tips 3: Du kanske inte ens behöver ändra i yaml-filen...
     - kolla vad du kan göra med `docker compose --help`

19.6 Erbjuda och Använda REST

Det är dags att titta inuti applikationen också.

`Containers/StatsPresenter`

- Har en enda kodfil: `src/index.js`
- Det finns i huvudsak fyra delar:
  1. Skapa en express-webserver
  2. Koppla upp mot databasen
  3. Ställ in och sätt igång alla REST-ändpunkter som applikationen skall lyssna på
  4. Funktioner för varje ändpunkt

`Containers/VoteCounter`

- Ser i princip likadan ut.

`Containers/PonyVoter`

- Lite fler funktioner, men i stort sett samma struktur.

Att göra:

- Vilka REST-ändpunkter erbjuder respektive container?
- Är de GET, POST, PUT, eller DELETE? Vad borde de vara?
- Vilka typer av svar ger respektive ändpunkt?
- Hur kan du testa det?

19.7 Testa API:et

- Bara `PonyVoter` är tillgänglig från värd-datorn...
- hur kan vi testa de andra containrarna?

Att göra

1. Studera `Containers/APITester` så att du vet vad den gör.
   - Studera även `test.yaml`.
2. Starta `PonyVoter` - applikationen
3. Kör `docker compose -f test.yaml up` och se vad som händer.
   - Notera att du har tre olika typer av svar, med olika `Content-Type`.
   - Hur kan du använda detta när du bygger ett REST-API?

19.8 Fundera på / Ta reda på

- Kan du kontrollera om en container är frisk?
  - Hur skriver du en sådan *healthcheck* i din docker compose-fil?
  - Måste du alltid ha en särskild ändpunkt i ditt REST-API för detta?
    - När måste du definitivt ha en särkskild ändpunkt?
    - Finns det andra lösningar?
- Vissa driftsättningsplatformar har begreppet *Init Containers*
  - Vad använder man init containers till?
  - Hur kan du åstadkomma detta med docker compose?
- Vad är docker compose *Secrets* ?
  - När skall du använda dem?
  - Hur?
- Vad behöver du göra för att din docker compose-file skall bli färdig att driftsättas (*Production Ready*)?

19.9 Sammanfattning

- Du har nu arbetat med en *microservice* - applikation
- Varje komponent (Container) har sitt eget *REST-api*
- Du har använt flera olika programspråk (`JavaScript/Node.js` och `bash`)
- Du har skalat delar av din applikation upp och ner