Microservices og Availability

Adam Lass og Rasmus Helsgaun

11. december 2020

Abstract

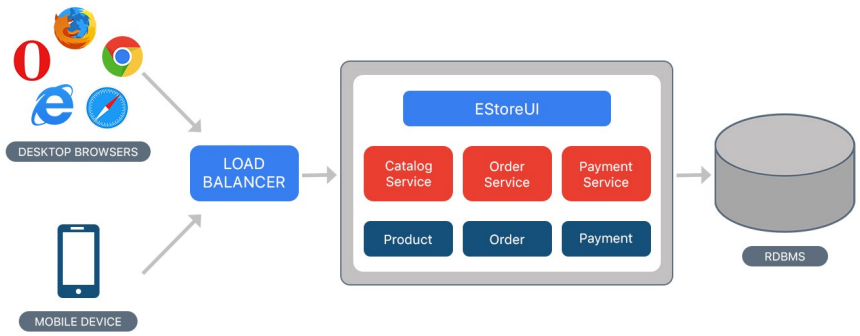
Ensuring high availability in a monolithic application is almost impossible due to its deployment challenges and lack of flexibility. This poses a big threat to businesses with a monolithic system today, that want to scale tomorrow. The solution to this problem is called microservices, and in this article we will explore why this is the architecture to use when building large systems with a lot of user traffic. By implementing microservices in the right places, businesses stand to win big on flexibility, scalability and availability of their systems.

Hvad indebærer availability?

Når der er tale om availability, refereres der til tilgængeligheden af software og dets evne til at udføre en opgave når der er brug for det. Konceptet bygger på reliability, som handler om hvor pålidelig et system er, og inkorporerer tilmed også konceptet om gendannelse af systemet ved fejl. I denne artikel vil der blive set nærmere på hvordan en microservice arkitektur kan skabe værdi for en kunde i form af availability og hvilken bekostning har det sammenlignet med en monolitisk arkitektur.

Hvad er en monolitisk arkitektur?

Den monolitiske arkitektur beskriver en software applikation, hvor alle komponenter er samlet i et enkelt system.



Figur 1: *Illustrerer opsætningen af den monolitiske arkitektur, med alle systemets komponenter samlet i ét modul bagved en load balancer.[1]*

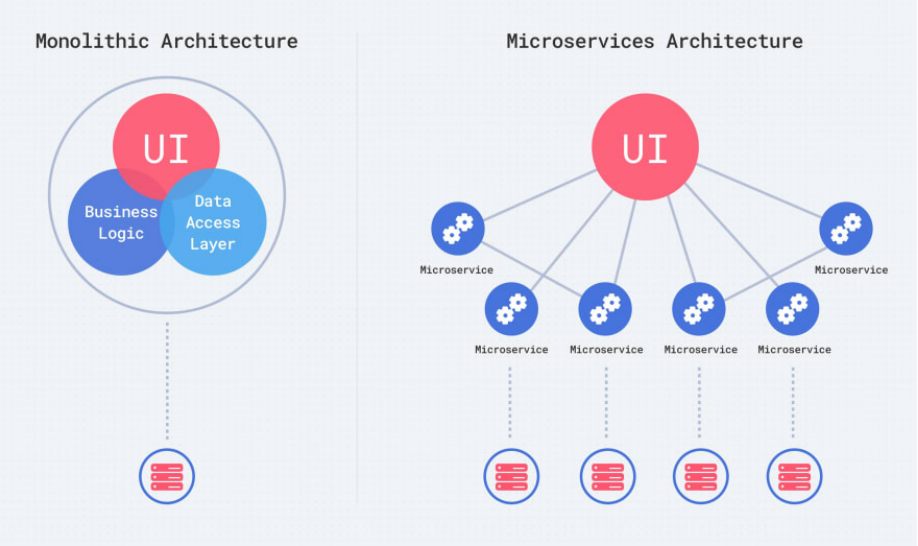
Fordelene ved at bruge en monolitisk arkitektur er blandt andet, at det i de tidlige stadier kan være enkelt at udvikle, deploye og teste, da man har alle komponenter samlet i én applikation.[2] Givet at ens monolitiske system er udviklet stateless, er det også forholdsvist simpelt at skalere horisontalt, ved at køre flere instanser af systemet bag en load balancer, som illustreret på figur 1.[1]

Der er dog også en række ulemper ved at udvikle og vedligeholde et monolitisk system, som bør overvejes inden den vælges som arkitektur. Eksempelvis kan det være sværere at vedligeholde et monolitisk system, da størrelsen af komponenterne og afhængighederne derimellem kan øge systemets kompleksitet. Den øgede kompleksitet medfører ofte en høj kobling, hvilket forværrer fleksibiliteten og overskueligheden af systemet.[3] Dette kan have store negative implikationer for udefrakommende udvikleres forståelse og overblik af systemet, hvilket kan resultere i øgede tidsmæssige og økonomiske udgifter ved videreudvikling.[4]

Samtidigt kan størrelsen have betydning for en længere opstartstid, hvilket er essentielt at holde til et minimum under opdateringer eller efter at systemet er crashet. I sådanne tilfælde bærer arkitekturen i særdeleshed præg af at en lille opdatering eller crash af et komponent kræver et fuldt genstart af alle komponenterne.[5] Hvis ikke disse scenarier bliver håndteret ordentligt, kan det resultere i at arkitekturen har en stor negativ indflydelse på reliability og availability af applikationen.

Hvad er Microservices?

Microservices er applikationer som følger en arkitektur hvor deres formål er at specialisere sig i forskellige egenskaber, ofte som en del af et større system. Dette er et direkte modstykke til den monolitiske arkitektur, hvor man har alle egenskaber i samme system.



Figur 2: *Illustrerer forskellen mellem den Monolitiske arkitektur og Microservices arkitekturen.[6]*

Den opdelte arkitektur set i figur 2 bringer en række fordele til områder af udviklingen i form af availability, scalability, testability, deployment m.fl., men har selvfølgelig samtidigt også en række ulemper med sig.[1] En meget dominerende fordel ift. availability er eksempelvis hvordan microservices gerne skal være små og afgrænset i forhold til deres egenskaber.[7] Dette har nemlig den betydning at potentiel downtime på den enkelte service bliver mindre alvorlig, da den nu kun eksisterer som en mindre del af et større system. Sådan en downtime kan være forårsaget af bugs der får servicen til at crashe, men også af almindelige opdateringer af servicen, begge ting som er forventelige i dagligdagen. I forhold til continuous deployment kan microservices til med drage fordel af rullende opdateringer såsom Kubernetes' Rolling Updates[8], og på denne måde helt undgå downtime forårsaget af opdateringer. Denne teknik udnytter i høj grad horisontal scaling, hvilket sørger for at flere instanser af samme service eksisterer ved siden af hinanden, så hvis den ene fejler kan en anden bare overtage. Dette forudsætter at man implementerer en loadbalancer der ligger foran disse horisontalt skalerede services, hvilket kan resultere i længere udviklings- og responstid. Dog tillader denne metode, sammen med et

framework såsom serverless, at man dynamisk kan skalere de enkelte microservices eller funktioner efter behov, hvilket både øger availability og kan give økonomiske besparelser.[9]
En anden betydelig fordel som microservices bringer er dets testability. Selvstændigheden og isolationen af komponenterne gør nemlig at de er nemmere at teste, da der findes de samme afhængigheder til andre komponenter sammenlignet med et monolitisk system.[10] Accepterer man det faktum at tests skaber bedre kode, der sjældnere fejler[11], så kan man sige at bedre testability i overført betydning også skaber bedre availability og reliability.

I det store hele kommer microservices altså med en masse fordele i forhold til availability, men der ligger selvfølgelig også en række udfordringer forbundet med at implementere arkitekturen. Udviklingsmæssigt skal man som udvikler ofte bruge meget tid på at hver enkelt komponent kan stå alene. Dette indebærer ofte ting som authentication, authorization, logging, osv., alle ting som man i et monolitisk system kun skal implementere én gang.[12] En anden udfordring er at de forskellige komponenter ofte skal snakke med hinanden via protokoller såsom HTTP, hvilket tilføjer ekstra netværkstrafik og compute hos eventuelle loadbalancers. Dette kan både resultere i længere svartider og større regninger hos cloud leverandøren, begge ting man gerne vil forsøge at undgå, især som en mindre virksomhed.[13]

Hvad vælger man?

Store tech giganter såsom Amazon, Netflix og Spotify startede ud med en monolitisk arkitektur, men det ses dog at den generelle udvikling går mod at bruge microservices.[3] Denne udvikling skyldes i høj grad at microservices er bedre egnet til at håndtere de udfordringer der kommer med at bygge store systemer som skal kunne håndtere meget brugertrafik. Dette kommer på en bekostning af ekstra udviklings- og responstid, men disse udgifter bliver nemt opvejet af de fordele der ligger i at bruge microservices såsom availability, scalability og maintainability. Den monolitiske arkitektur har dog stadig sine fordele ved mindre projekter hvor der ikke er store spikes i trafikken. Her er den nemlig hurtigere at udvikle og ofte billigere at hoste. Når der skal vælges mellem de to arkitekturer kommer det i sidste ende an på hvilke behov man har her og nu. Disse behov kan over tid ændre sig, dog vil de i de fleste tilfælde kun gå i en retningen mod at bruge microservices. Står man derfor i et valg mellem de to, og hælder mest til at bruge microservices, kan man lige så godt gøre det her og nu, i stedet for i morgen.

Litteratur

[1] S. ul Haq. Introduction to monolithic architecture and microservices architecture. https://medium.com/koderlabs/introduction-to-monolithic-architecture-and-microservices-architecture-b211a5955c63.

[2] J. Kanjilal. Pros and cons of monolithic vs. microservices architecture. https://searchapparchitecture.techtarget.com/tip/Pros-and-cons-of-monolithic-vs-microservices-architecture.

[3] A. Kwiecień. 10 companies that implemented the microservice architecture and paved the way for others. https://divante.com/blog/10-companies-that-implemented-the-microservice-architecture-and-paved-the-way-for-other

[4] P. Karwatka. Monolithic architecture vs microservices. https://divante.com/blog/monolithic-architecture-vs-microservices/.

[5] R. Mhetre. When to choose microservices architecture over monolithic? why? https://medium.com/@mhetreramesh/when-to-choose-microservices-architecture-over-monolithic-why-794aed04d8db.

[6] A. Barashkov. Microservices vs. monolith architecture. https://dev.to/alex_barashkov/microservices-vs-monolith-architecture-4l1m.

[7] Ambassador. Microservices. https://www.getambassador.io/learn/kubernetes-glossary/microservices/.

[8] Kubernetes. Rolling update deployment. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-update-deployment.

[9] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. https://www.cs.colostate.edu/~shrideep/papers/ServerlessComputing-IC2E-2018.pdf.

[10] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. M. F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow. https://arxiv.org/pdf/1606.04036.pdf.

[11] O. Mikhalchuk. The importance of unit testing, or how bugs found in time will save you money. https://fortegrp.com/the-importance-of-unit-testing/.

[12] T.v. Vignesh. Building a boilerplate for microservices — part 1. https://medium.com/techahoy/building-a-boilerplate-for-microservices-part-1-166ce00f5ce9.

[13] M. S. Nyfløtt. Optimizing inter-service communication between microservices. https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2479192/18340_FULLTEXT.pdf?sequence=1.