Aleksander Konnerup

[Mailadresse]

Resumé

[Indfang din læser med et engagerende og kort resumé af dokumentet.   
Når du er klar til at tilføje dit indhold, skal du klikke her og skrive.]

IT6ONK - TSEIS

[Dokumentets undertitel]

Indhold

[Indledning 1](#_Toc515273490)

[Design 1](#_Toc515273491)

[Udførsel 1](#_Toc515273492)

[Særligt gode/bemærkelsesværdige implementeringer 1](#_Toc515273493)

[Mangler 1](#_Toc515273494)

[(Sammenligning af platforme måske, Service Fabric vs. Kubernetes) 1](#_Toc515273495)

[Konklusion 1](#_Toc515273496)

[Litteraturliste 1](#_Toc515273497)

Antal sider 10 ca. afhængig af brug af figurer o. Lign i rapporten, herudover kommer sider til indholdsfortegnelse, reference- og litteraturlister, forside og bilag. Husk sidenummerering!

Format PDF, navngivet ITONKF18Gr[nr].pdf eks ITONKF18Gr5.pdf.

Rapporten beskriver

# Indledning

I faget IT6ONK arbejdes der med distribuerede systemer, mere specifikt omhandler faget forbindelserne mellem objekter/microservices. I sammenhæng med dette er der i løbet af semesteret udviklet på et distribueret system til udveksling af aktier, kaldet The Stock Exchange Interaction System (TSEIS). Denne applikation har baseret sig på de opstillede Mountain-Goat krav til systemet og det oplæg til design af applikationen der blev skrevet i starten af kurset. Følgende rapport vil indeholde en beskrivelse af designet af TSEIS-applikationen, hvordan dette er blevet udført gennem Azure Service Fabric, en opsummering af særlige gode features, ligeledes en beskrivelse af systemets fejl/mangler og slutteligt en konklusion på hele processen og det færdige system.

# Design

Til at beskrive design af vores distribuerede system har vi opsat en N+1 arkitektur. Denne er inddelt i et Development view, et Physical view, et Proccess view og et Use case view. Vi har valgt at udlade det logiske view, da vi ikke følte det bragte værdi til beskrivelsen af designet. De førnævnte views vil blive beskrevet i føgende afsnit:

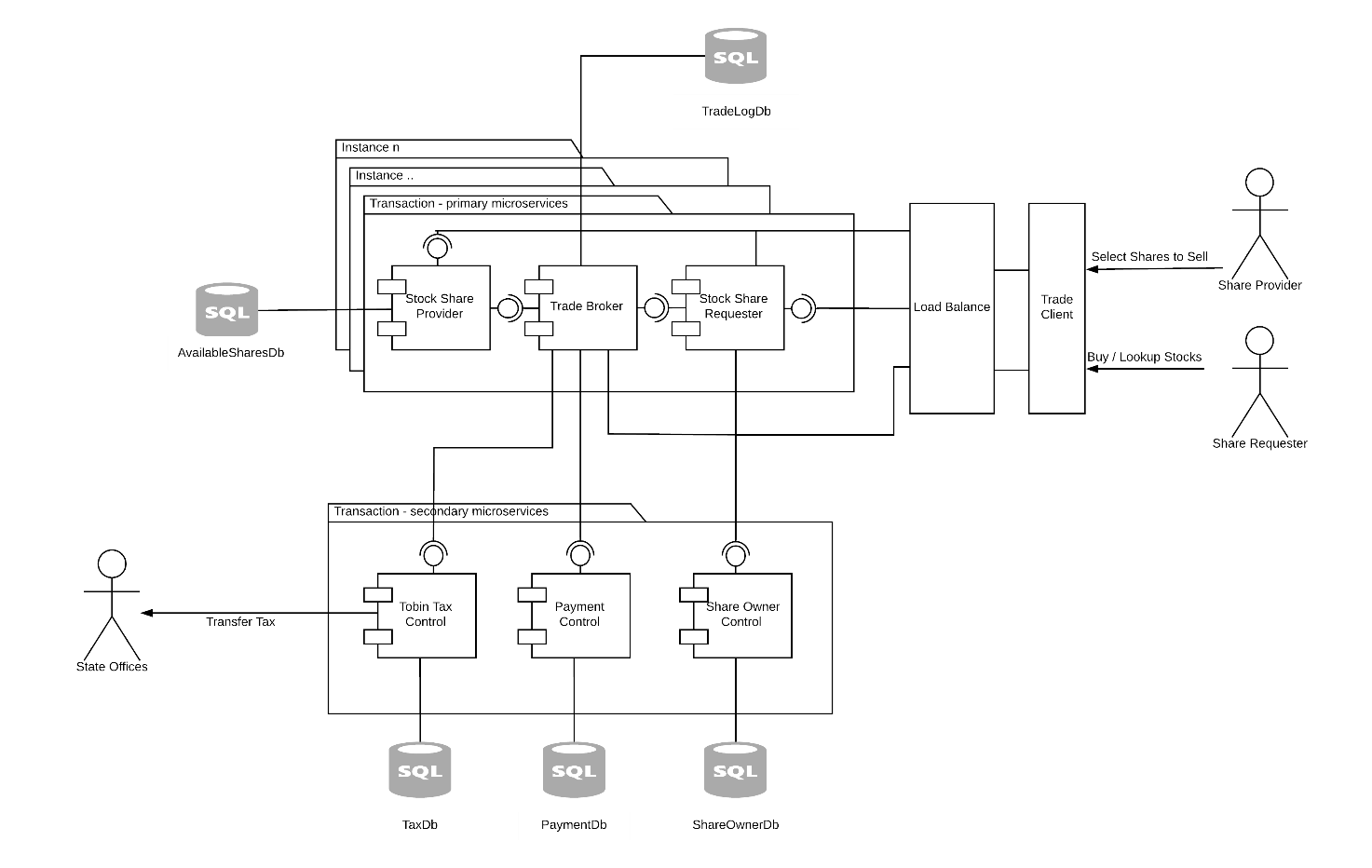
## Use Case View

Vores Use Case View tager sit udgangspunkt i oplægget til projektet, mere specifikt de opstillede Mountain-Goat krav til systemet. Disse ses nedenfor:

* As a "Tobin Taxer" I want to tax **all** transaction with a charge af 1%  of the total transaction value for then to be able to send the tax to the state.
* As a "Public Share Own Control" I want to be informed of any changes in the share ownerships for then to be able to track who is the owner of any share and series of shares.
* As a "Stock Trader Broker" I want to be the broker between the seller (provider) og the buyer (requester) for then to be able to intermediate the trade of one or more shares or one or more series of shares between one or more providers and/or one or more requesters.
* As a "Stock Trader Broker", "Stock Share Provider" or a "Stock Share Requester", I want to inform the Tobin Tax Control, that a transaction of a certain value has been commited, for then to be able obey the public rules about taxing and paying the 1% Tobin Tax.
* As a "Stock Trader Broker", "Stock Share Provider" or a “Stock Share Requester", I want to inform the The Public Share Owner Control, that a share, many shares or a series of shares or many series of shares has changes ownership, for the to able to inform the public about who own the shares.

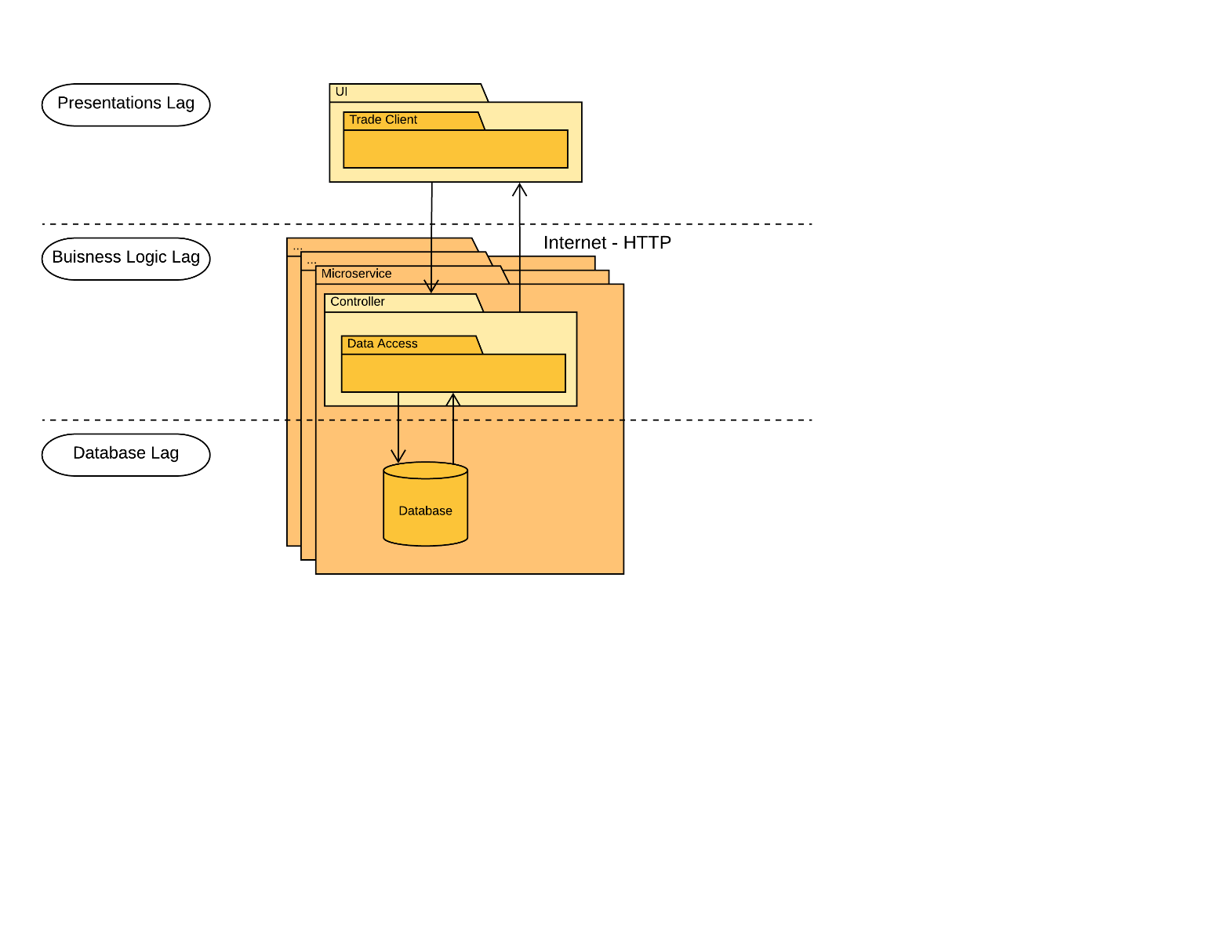
## Development View

Figur 1 viser et component diagram for vores TSEIS applikation, det skal her bemærkes at kald imellem de forskellige microservices er udeladt, da dette kun ville skabe forvirring i diagrammet. I stedet kan en beskrivelse af alle de forskellige api-kald ses i bilagene:



Figur 1: Component diagram for TSEIS

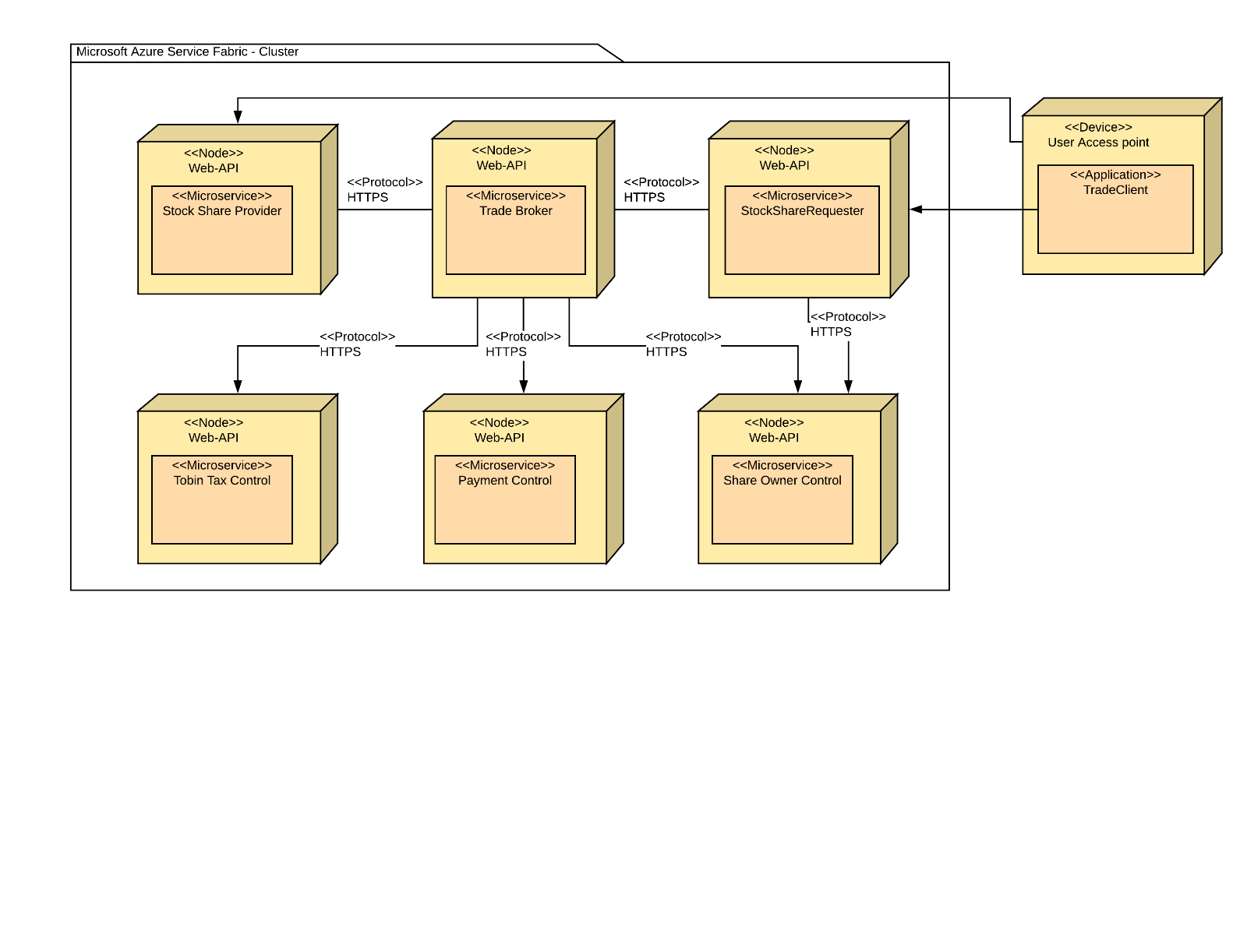
I diagrammet på figur 1 ses det at vi har inddelt vores system i 6 microservices, hvoraf 5 af dem har en tihørende database. Stock-share requesteren fungere lige nu som mediater til resten af systemet og er det primære acces point for Trade Client, som er systemets UI. Figur 2 viser et package diagram for systemet, det ses her at vores præsentationslag består af den Trade-client UI der vises til brugeren. Vores buisness logic lag består af en række microservices, der kommunikerer ved hjælp af en http-protokol. Hver disse udstiller en række api-kald, der tillader CRUD-operationer på deres respektive databaser, derfor er data-access også placeret i microservicenes controller klasser. Database laget består af de 5 databaser tilhørende hver microservice.



Figur 2: Package diagram for TSEIS

## Physical View

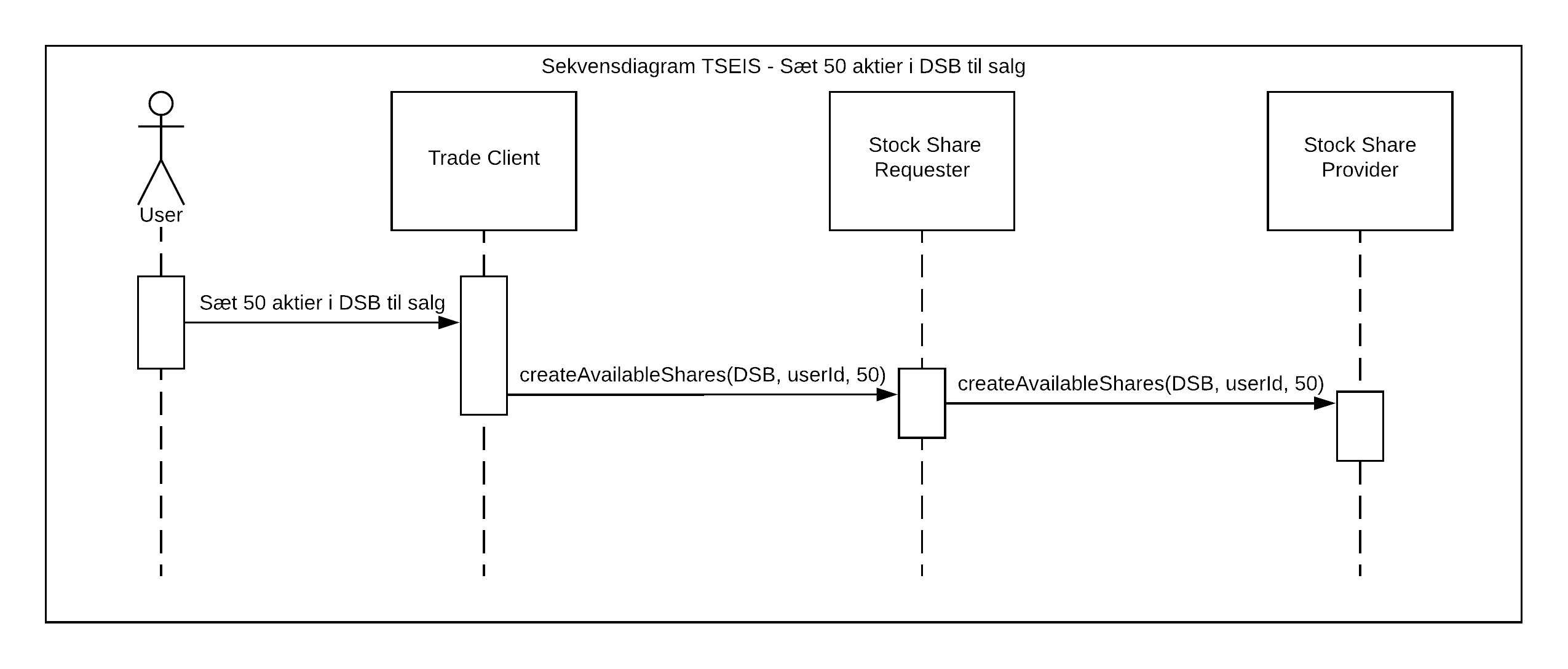
På figur 3 ses et deployment diagram over systemet. Hver microservice er her repræsenteret, som en node i en Azure Service Fabric Cluster. Det skal hertil nævnes at alle microservice ikke kører på sin egen node, men at en node godt kan have flere microservices kørende. Dette er også grunden til at det er muligt at køre 6 microservices med kun 5 noder tilgængelig i clusteren. Yderligere ses det at Trade Client er opstille som User Access point, dette er således de eneste endpoints der udstilles til brugeren i form af en WPF-applikation.



Figur 3: Deployment diagram for TSEIS

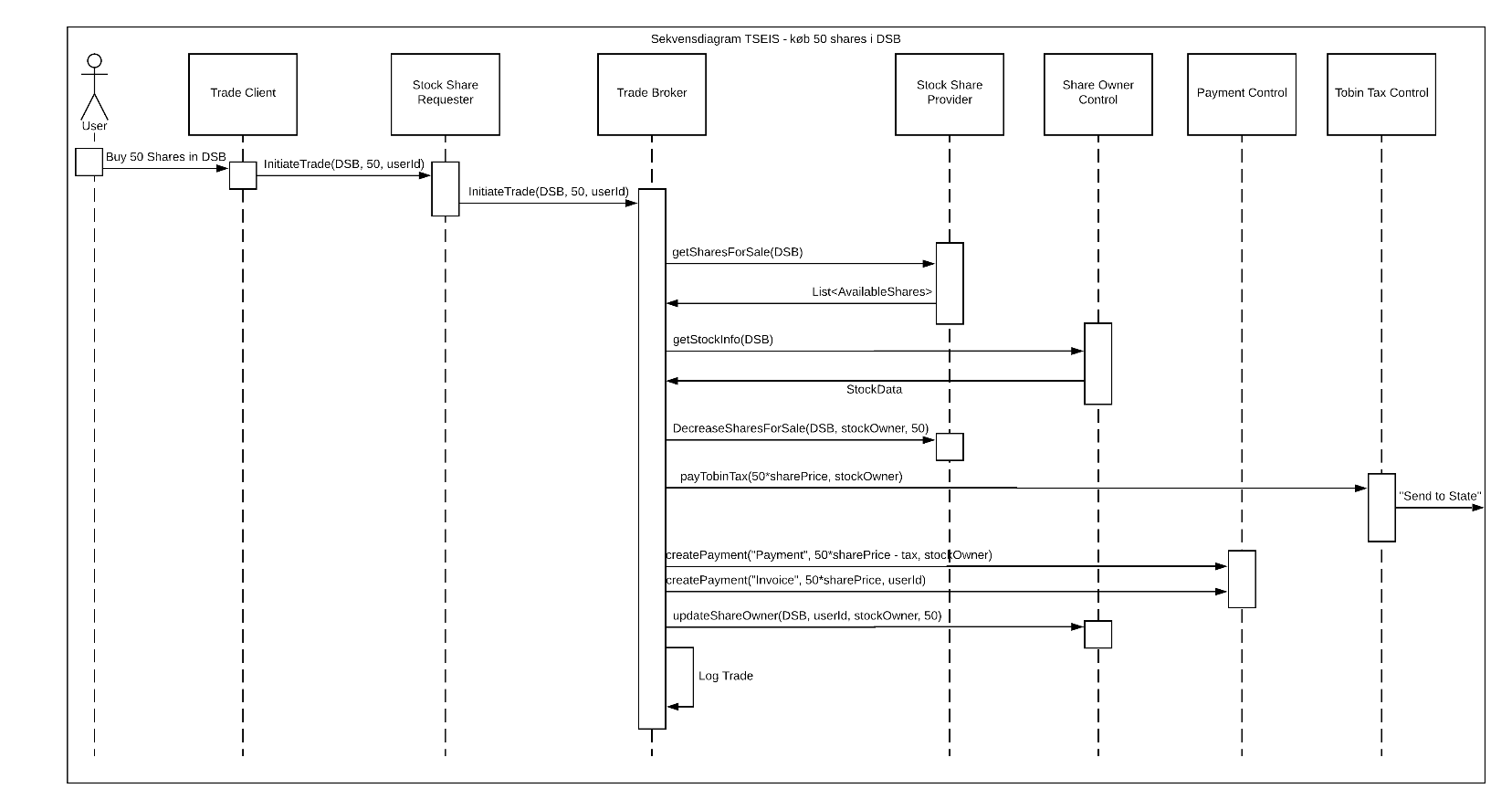
## Proccess View

Vores process view består af en række sekvensdiagrammer baseret på de opgivede Mountain-Goat krav. Diagrammet på figur 4 illustrerer hvilke kald der foretages når en bruger ønsker at sætte 50 af sine aktier til salg.



Figur 4: Sekvensdiagram - Sæt aktier til salg

Diagrammet figur 5 viser scenariet hvor en bruger ønsker at købe 50 aktier i et specifikt firma. Det ses her Trade brokeren fungere som mediater ud til resten af systemet og foretager kaldene ud til de forskellige microservices, dette startes af metoden InitiateTrade.



Figur 5: Sekvensdiagram - køb aktier

* I design lægges der primært vægt på at beskrive opbygningen af applikationens infrastruktur på den valgte cluster samt overvejelser for opdelingen af funktionalitet og microservices.
  + Selve ”business logic’en” er som sådan sekundær (kræver ikke den store dokumentation), idet den forventes at være brugt, som ”driver” for design af infrastruktur (microservices og cluster).
* Til UC- Diagram (not needed..)
  + I would say if the DB is internal to your app (i.e. it is used solely by this app and the external world doesn't interact with it directly in anyway, only via your app), you shouldn't represent it as a separate actor.
  + If the DB is shared and used directly by external users/systems, it is probably better to represent it as a separate actor.

# Udførsel - (Sammenligning af platforme måske, Service Fabric vs. Kubernetes)

Vores applikation er udviklet på Azure Service Fabric, en platform til distribuerede systemer lavet af Microsoft. Gennem platformen er det muligt at oprette og styre applikation sammensat af microservices, der kører på en fælles samling af delte maskiner, bedre kendt som et cluster. Man kan således nemt opsætte en række skalerbare stateless eller stateful microservices, med et tilhørende applikationsinterface til at håndtere levering, implementering, overvågning, opgradering og sletning af de opsatte microservices. Service Fabric tillader altså omfattende runtime- og lifecycle-håndterings kapabiliteter til microservice-applikationer. Den hoster hver microservice inde i containere, der bliver deployed og activated på Service Fabric clusteren, dette gør det muligt at håndtere store stigninger i densitet.

Vores applikation er sammensat af 6 stateless microservices oprettet i Visual-studio og deployed til en Service Fabric Cluster, bestående af 5 noder. Disse udstiller hver et api og kommunikerer således internt gennem http-protokollen. Yderligere har 5 af de 6 micoservices en database som kun de kan tilgå og lave operationer på.

* Udførelse på valgte platform, Azure Service Fabric eller/og Google Kubernetes, og inddrag her en overordnet en beskrivelse af udviklingsmiljø.

# Særligt gode/bemærkelsesværdige implementeringer

* Fremhævelse af ”guldkorn (/ …bamsefrø) ” for gennemførelse af tekniske løsning og en eller flere givne problemstillinger, dvs. beskriv en særlig smart, elegant eller på anden måde ”bemærkelsesværdig” løsning. Et eksempel herpå er inddragelsen af eksisterende standard services/komponenter til f.eks. persistering.
* Persistering er opsat via ormLite, alt opsætning findes derfor i controlleren
* Swagger og auto-generering af clients (NSwag)

# Mangler

* Opsummering af problemer, som ikke er løst eller udeståender som kan ses som mangler for projektet. Konkretiser om muligt betydningen heraf.
* ShareOwnerControl
* TradeBroker

# Konklusion

# Litteraturliste

* Brug referencer til brugte artikler og hjemmesider. Og gør det gerne i ”overdrevent” omfang, dels af hensyn til den videre brug af jeres projekt men også i forhold til andres læsning af rapporten (Eksaminator).

# Bilag???

* Synopsis???
* Mountain goat
* Beskrivelse af alle Api-kald skal nok flyttes her.
* Feedback fra anden gruppe mybe? Don’t fucking know…

**ShareOwnerControl API-kald:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HttpType:** | **Metode-navn:** | **Parametre:** | **Returværdi:** |
| Get | GetAllSharesForUser | Guid userId | List<ShareOwnerDataModel |
| Get | VerifyShareOwnership | String stockId,  Guid userId,  Int sharesAmount | Bool |
| Get | GetStockInfo | String stockId | StockDataModel |
| Get | GetAllStocks |  | List<StockDataModel> |
| Get | GetAllUsers |  | List<OwnerDataModel> |
| Put | UpdateShareOwnership | String stockId,  Guid requester,  Guid provider,  Int sharesAmount | Void |
| Post | CreateShareOwnership | String stockId,  Guid userId,  Int sharesAmount | Void |
| Post | CreateStock | String stockId,  IntSharePrice | Void |
| Post | CreateOwner | Guid shareHoldeId | Void |

**PaymentControl Api-kald:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HttpType:** | **Metode-navn:** | **Parametre:** | **Returværdi:** |
| Get | GetPaymentInfoForUser | Guid userId | PaymentDataModel |
| Post | CreatePaymentInfo | String paymentType,  Int paymentAmount,  Guid counterParty | Void |

**StockShareProvider Api-kald:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HttpType:** | **Metode-navn:** | **Parametre:** | **Returværdi:** |
| Get | GetShareForSaleByStock | String stockId | List<AvailableSharesDataModel> |
| Get | GetSharesForSaleByUser | Guid userId | List<AvailableSharesDataModel> |
| Get | GetAllSharesForSale |  | List<AvailableSharesDataModel> |
| Put | IncreaseSharesAmountForSale | String stockId,  Guid userId,  Int sharesAmount | Void |
| Put | DecreaseSharesAmountForSale | String stockId,  Guid userId,  Int sharesAmount | Void |
| Post | CreateAvailableShares | String stockId,  Guid userId,  Int sharesAmount | Void |

**StockShareRequester Api-kald:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HttpType:** | **Metode-navn:** | **Parametre:** | **Returværdi:** |
| Get | GetSharesForSaleByUserId | Guid userId | List<AvailableSharesDataModel> |
| Get | GetAllSharesByUserId | Guid userId | List<ShareOwnerDataModel> |
| Get | GetAllSharesForSale |  | List<AvailableSharesDataModel> |
| Get | GetAllUsers |  | List<OwnerDataModel> |
| Post | CreateAvailableShares | String stockId,  Guid userId,  Int sharesAmount | Void |
| Post | InitiateTrade | String stockId,  Int sharesAmount,  Guid requesterId | Void |
| Post | CreateStock | String stockId,  Int sharePrice | Void |
| Post | CreateOwner | Guid shareHoldeId | Void |

**TobinTaxControl Api-kald:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HttpType:** | **Metode-navn:** | **Parametre:** | **Returværdi:** |
| Post | PayTobinTax | Float transactionValue,  Guid shareProviderId | Void |

**TradeBroker Api-kald:**

|  |  |  |  |
| --- | --- | --- | --- |
| **HttpType:** | **Metode-navn:** | **Parametre:** | **Returværdi:** |
| Get | GetTradeData | Int tradeDataId | TradeDataModel |
| Post | InitiateTrade | String stockId,  Int sharesAmount,  Guid requesterId | Void |