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 2](#_Toc515793944)

[Design 2](#_Toc515793945)

[Use Case View 2](#_Toc515793946)

[Development View 3](#_Toc515793947)

[Physical View 5](#_Toc515793948)

[Proccess View 6](#_Toc515793949)

[Udførsel - (Sammenligning af platforme måske, Service Fabric vs. Kubernetes) 7](#_Toc515793950)

[Særligt gode/bemærkelsesværdige implementeringer 8](#_Toc515793951)

[Mangler 8](#_Toc515793952)

[Konklusion 8](#_Toc515793953)

[Litteraturliste 8](#_Toc515793954)

[Bilag 9](#_Toc515793955)

# 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 følsgende afsnit:

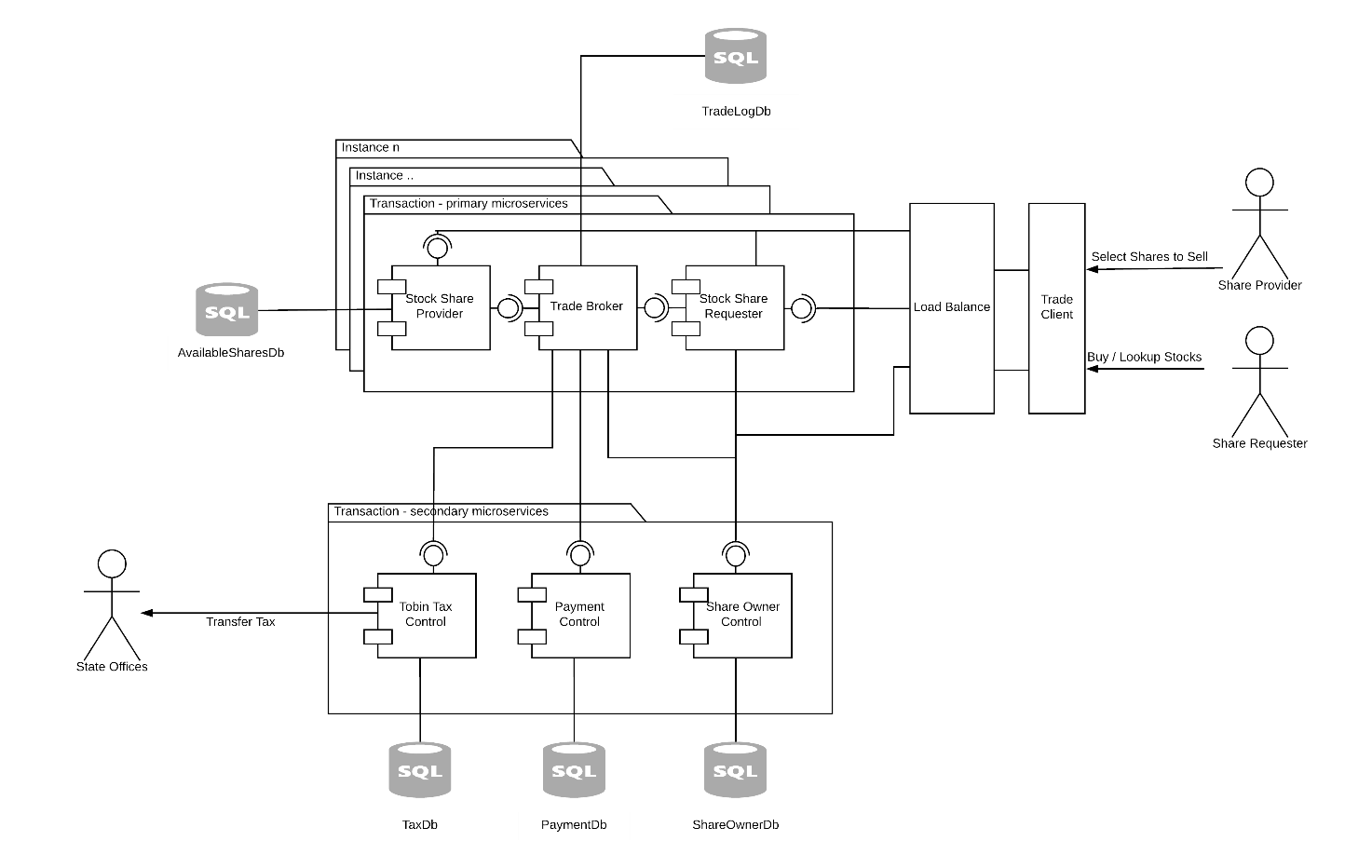
## Use Case View

Vores Use Case View tager sit udgangspunkt i oplægget til projektet, mere specifikt de opstillede Mountain-Goat krav til systemet. Disse ses nedenfor (Skal der tilføjes mere til UC-view??):

* 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

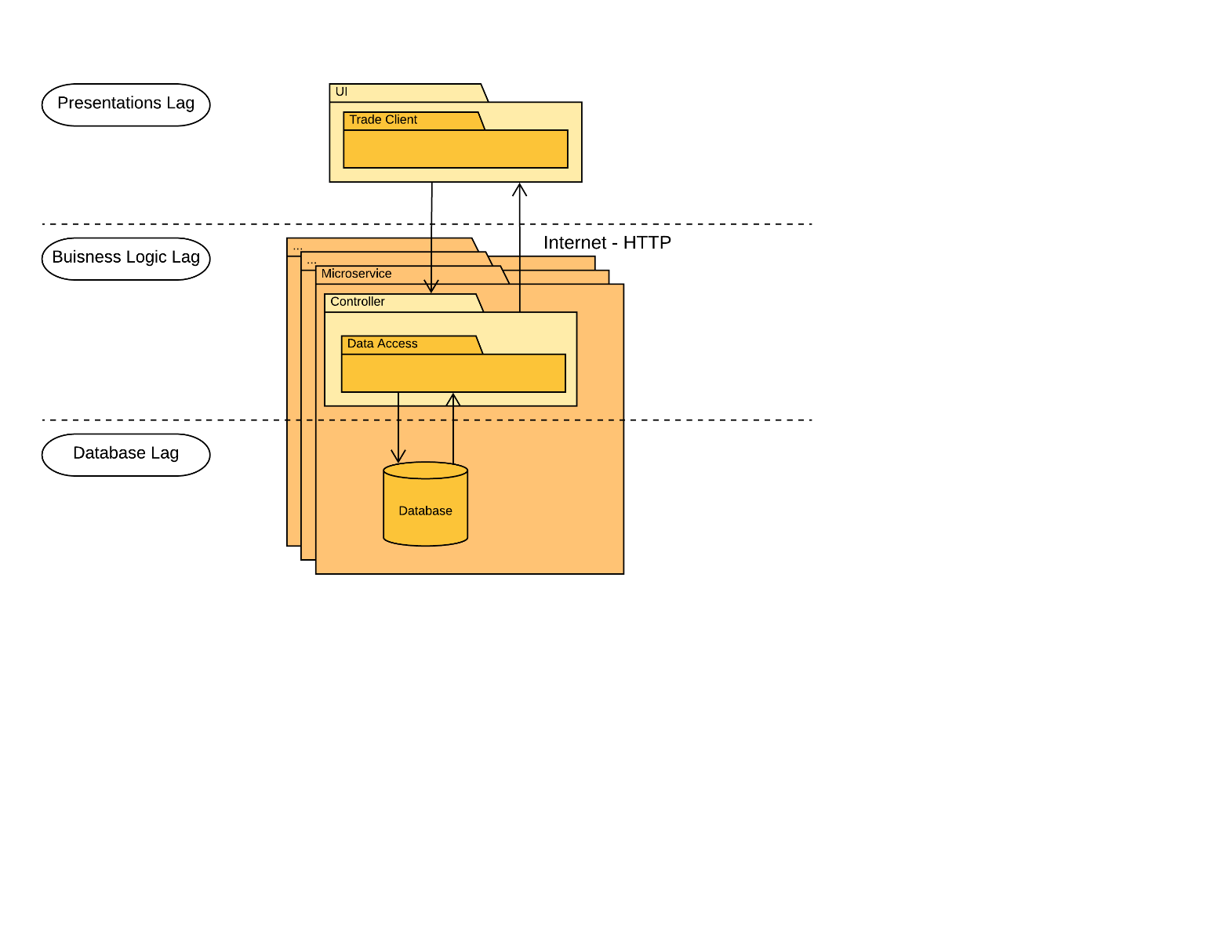
Figur 1 viser et component diagram for vores TSEIS applikation, det skal her bemærkes at kald imellem de forskellige microservices er udeladt, da dette kun ville skabe forvirring i diagrammet. I stedet kan en beskrivelse af alle de forskellige api-kald ses i bilagene:



Figur : Komponent 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.

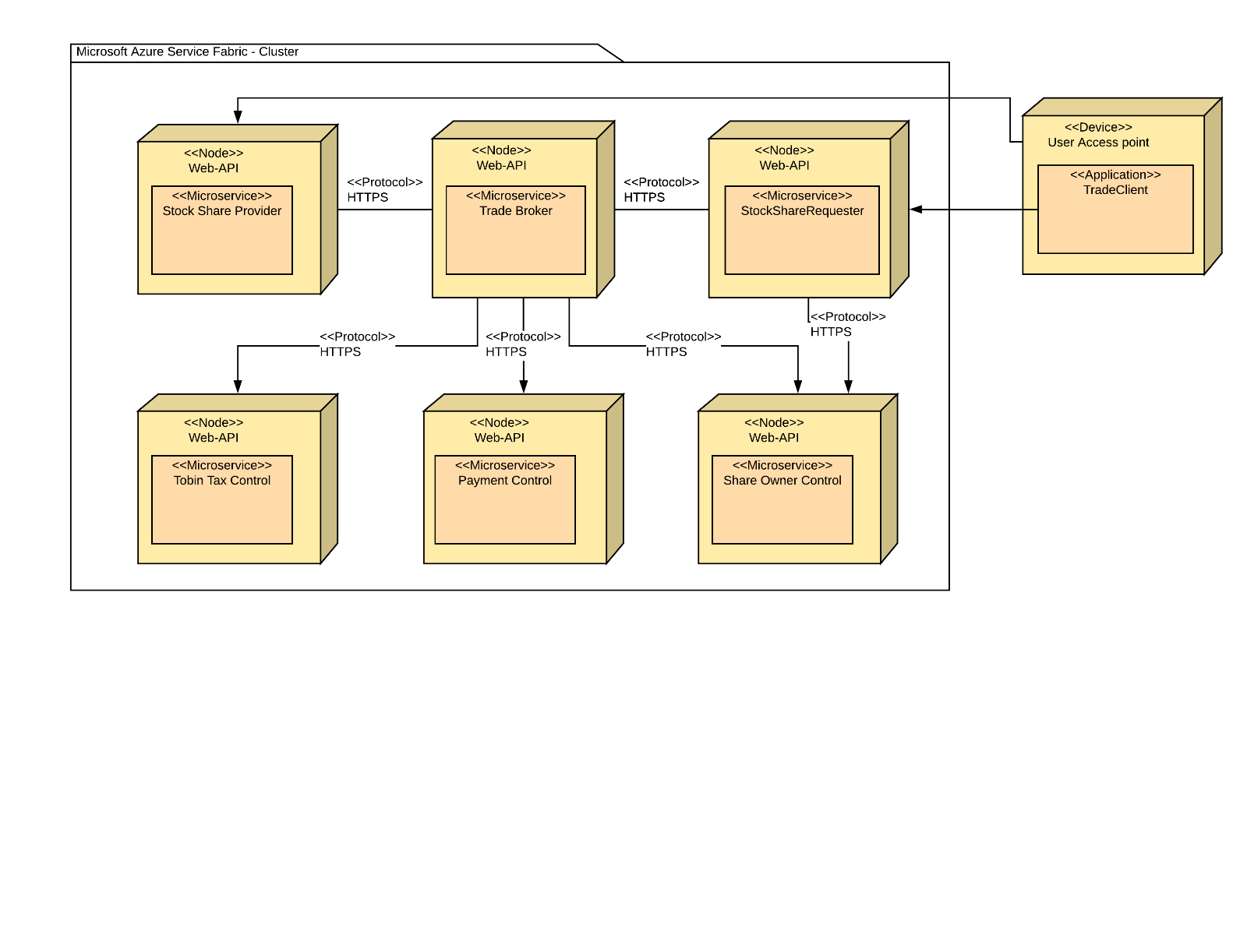
Figur 2 viser et package diagram for systemet, det ses her at vores præsentationslag består af den Trade-client UI der vises til brugeren. Vores buisness logic lag består af en række microservices, der kommunikerer ved hjælp af en http-protokol. Hver disse udstiller en række api-kald, der tillader CRUD-operationer 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 : Package diagram for TSEIS

## Physical View

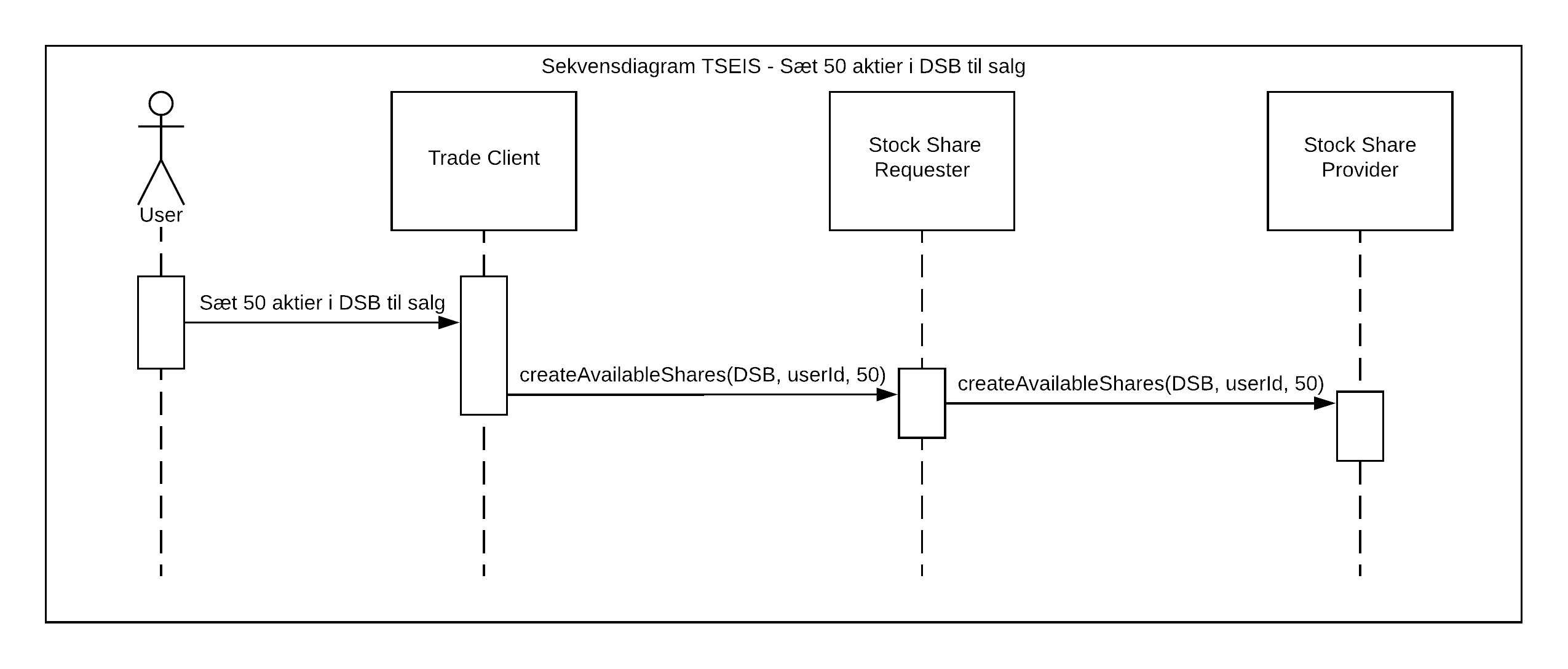
På figur 3 ses et deployment diagram over systemet. Hver microservice er her repræsenteret, som en node i en Azure Service Fabric Cluster. Det skal hertil nævnes at alle microservice ikke kører på sin egen node, men at en node godt kan have flere microservices kørende. Dette er også grunden til at det 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. (Måske tilføje load-balancer til diagrammet)



Figur : Deployment diagram for TSEIS

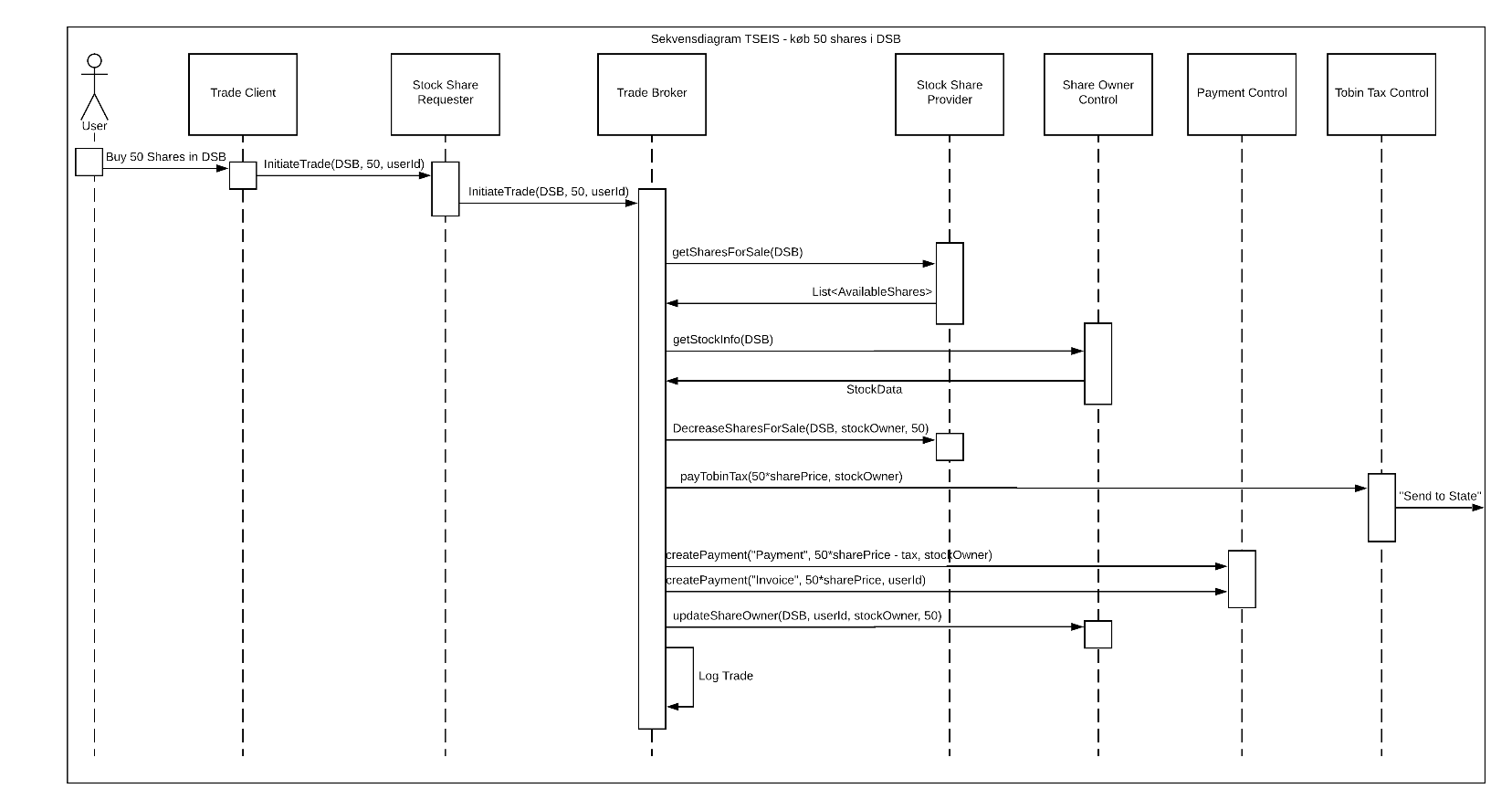
## Proccess View

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.



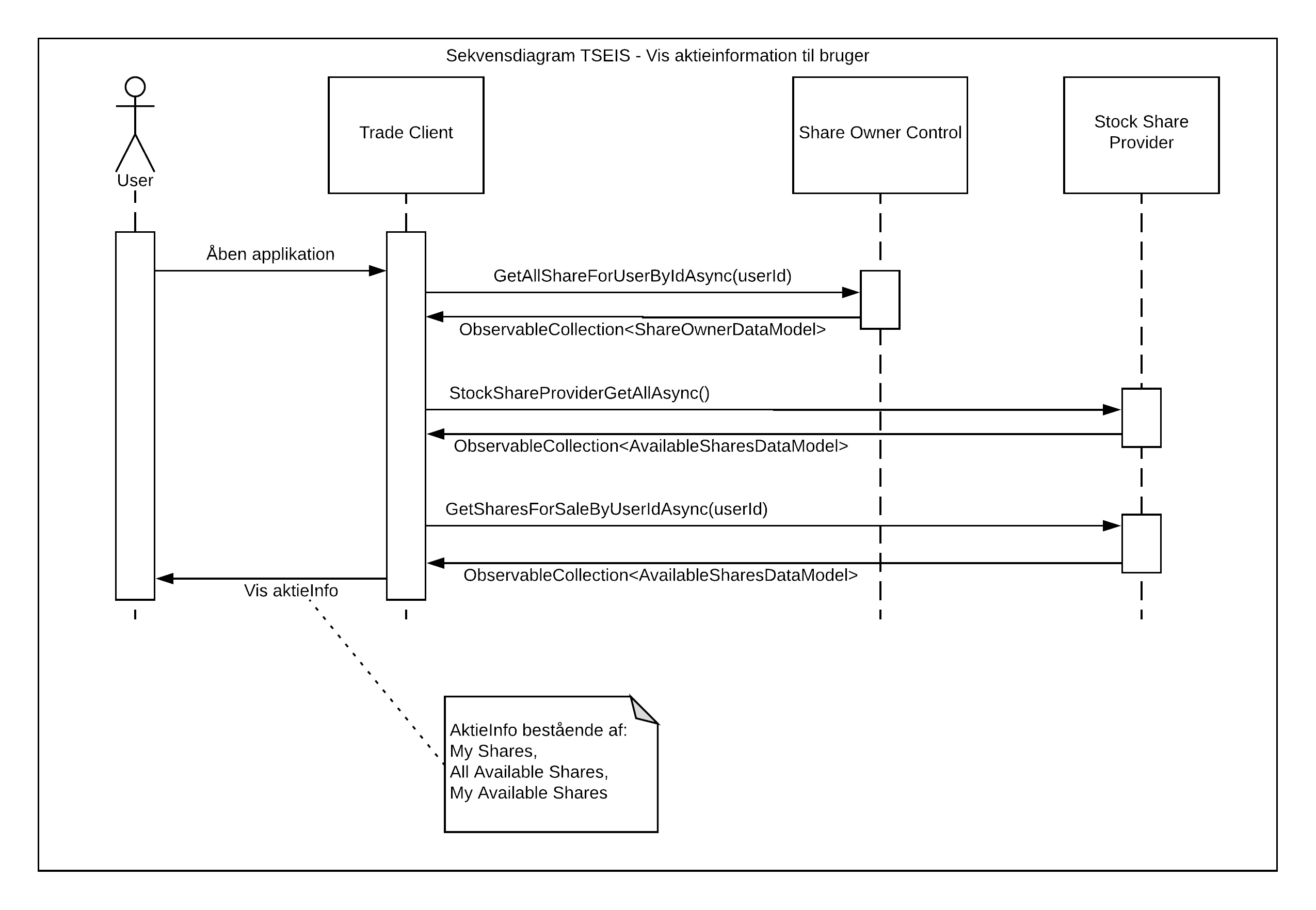
Figur : Sekvensdiagram - Sæt aktier til salg

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 : Sekvensdiagram - køb aktier

Diagrammet figur 6 illustrerer, hvilke kald der foretages fra Trade Client til de bagvedliggende microservices, når brugeren åbner applikationen. (ved ikke helt om det holder…)



Figur : Sekvensdiagram - vis aktieinformation

(Måske skrive noget mere om vores overvejelser undervejs og hvorfor vi tog de valg vi tog)

# 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.

(Kubernetes vs. Service Fabric) (Måske mere om udviklingmiljøet)

(Desuden skal Naming-service og Load-balancer nok også inddrages her….)

# Særligt gode/bemærkelsesværdige implementeringer

Til test af vores api-kald valgte vi at anvende Swagger, dette blev gjort ved brug af Nuget-pakken ”Swashbuckle”. Efter konfigurering af denne var det muligt at teste alle vores api-kald løbende ved blot at tilføje ”/swagger/” til url’en. Således kunne vi gennem dette framework nemt og hurtigt teste vores metoder og se hvilken http-respons disse returnerede. I sammenhæng med dette valgte vi til generering af vores clients, at benytte os af NSwag-Studio. Dette gjorde det simpelt at tilføje den ønskede business-logik og vi kunne ude de store problemer opsætte et funktionelt-system. Begge disse gjorde det nemt for at implementere alt business-logik og vores fokus kunne derfor ligges på microservice-arkitekturen.

Til persistering af data valgte vi at benytte os af ormLite-frameworket. Dette gjorde det muligt at opsætte alle vores databaser gennem de enkelte microservices controller-klasser. Ved brugen af ormLite var det muligt at generere alle de ønskede databaser og lave database-operationer på disse.

(Andre ting at fremhæve???)

# Mangler

Kigger man på det endelige design af vores system, ses det at Trade Broker microservicen har ansvaret for at initiere trade-funktionalitet, hvilket har resulteret i at den bliver monolitisk. Specielt metoden Intiate-trade går imod en del af design principperne for microservice-arkitekturer, da den gennem denne metode kommunikerer med alle de andre microservices. Yderligere har den som mediater til resten af systemet ansvar for initieringen af mange af systemets funktionaliteter, hvilket går imod SRP-princippet.

Desuden har Share owner control microservicen I det endelige design en database bestående af en OwnerDb, en StockDb og en relations database mellem disse kaldet ShareOwnerDb. Denne microservice er således det eneste access-point til disse tre databaser. Hvilket burde have været opdelt, således at de fungerede som tre separate databaser, hver tilhørende sin egen microservice.

(Skrive om load-balancing, DNS, Naming service, hvis det ikke kommer op at køre)

(Andre Problemer???) (Konkretiser måske betydningen af mangler)

# Konklusion

(Konkluder på lortet)

# Litteraturliste

(Find ud af hvilke reference vi har brugt somehow….)

# Bilag

(Andet shit at smide herunder??)

**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 |