Michelle Broens

S33 | GSO

KillerApp –

SAD

# Document historie

|  |  |  |  |
| --- | --- | --- | --- |
| **Versie** | **Wijzigingen** | **Auteur** | **Datum** |
| **0.1** | Document opgezet d.m.v. template in grote lijnen in te vullen. | Michelle Broens | 15/11/2017 |
| **0.2** | Klassendiagram toegevoegd. | Michelle Broens | 22/11/2017 |
| **0.3** | Domeinmodel en klassendiagram losgetrokken, oud domeinmodel is klassendiagram, nieuw domeinmodel gemaakt. | Michelle Broens | 29/11/2017 |
| **0.4** | Inleiding ingevuld, aanpassingen gemaakt om te matchen met gegeven voorbeeld. | Michelle Broens | 1/12/2017 |
| **0.5** | Hoofdstuk componenten, communicatie, persistentie ingevuld. | Michelle Broens | 2/12/2017 |
| **1.0** | Niet functionele eisen, deployment, specificaties van de interfaces en detailed design per component ingevuld. | Michelle Broens | 3/12/2017 |

Inhoudsopgave

[Document historie 1](#_Toc500098602)

[Inleiding 4](#_Toc500098603)

[Context 4](#_Toc500098604)

[Applicatie 4](#_Toc500098605)

[Niet-functionele eisen 4](#_Toc500098606)

[Doel van dit document 5](#_Toc500098607)

[Domeinmodel 6](#_Toc500098608)

[Domeinmodel 6](#_Toc500098609)

[Klassendiagram van het domein 7](#_Toc500098610)

[Bijlage 8](#_Toc500098611)

[Afbakening 8](#_Toc500098612)

[Componenten 9](#_Toc500098613)

[Componenten diagram 10](#_Toc500098614)

[Bijlage 10](#_Toc500098615)

[Koppeling tussen componenten 11](#_Toc500098616)

[Synchronisatie tussen componenten 11](#_Toc500098617)

[Services per component 11](#_Toc500098618)

[Allocatie van objecten 11](#_Toc500098619)

[Remote objecten 12](#_Toc500098620)

[Packagestructuur 12](#_Toc500098621)

[Communicatie 13](#_Toc500098622)

[Klassendiagram voor inloggen met RMI 14](#_Toc500098623)

[Bijlage 15](#_Toc500098624)

[Klassendiagram voor communicatie met server met RMI 16](#_Toc500098625)

[Bijlage 17](#_Toc500098626)

[Persistentie 18](#_Toc500098627)

[Eigenschappen van objecten m.b.t. persistentie 18](#_Toc500098628)

[Data-opslag 18](#_Toc500098629)

[Detailed design per component 19](#_Toc500098630)

[Sequence diagram login 20](#_Toc500098631)

[Sequence diagram checkAvailability 21](#_Toc500098632)

[Realisatie niet-functionele eisen 22](#_Toc500098633)

[Deployment 23](#_Toc500098634)

[Deployment diagram 23](#_Toc500098635)

[Bijlage 24](#_Toc500098636)

[Specificatie van interfaces 25](#_Toc500098637)

[ILogic 25](#_Toc500098638)

# Inleiding

*In dit architectuur document worden aanwijzingen gegeven in italic (schuinschrift).*

*Beschrijf in de inleiding:*

* *De context van het project*
* *De applicatie*
* *De niet-functionele eisen met betrekking tot betrouwbaarheid, performance, beveiliging en schaalbaarheid*
* *Het doel van dit document*

## Context

De applicatie is gemaakt voor mensen die vaak naar cons gaan en hun tickets in een applicatie bij willen houden.

## Applicatie

Het gedistribueerd ticket bestelprogramma zal bestaan uit drie onderdelen een klant, een login server en een server die al de overige data bij gaat houden. De klant kan inloggen (op de inlog server, die hem door zal sturen naar de andere server), de inlogserver houdt de gegevens bij van de username, email en password van een klant en als deze gegevens matchen wordt de klant doorgestuurd naar de andere server. Hier staat per klant de gekochte tickets en hier worden de nieuwe tickets ook opgeslagen als de klant deze koopt.

## Niet-functionele eisen

De onderstaande niet-functionele eisen uit het URS worden in dit document besproken:

**Functionaliteit:**  
De code is uitbreidbaar (NF1)  
De gegevens van de klant worden opgeslagen op een zogenaamde “inlogserver” (NF2).  
De gekochte tickets staan op een andere server opgeslagen dan de gegevens van de klant, hier staan de tickets gekoppeld aan de username van de klant (NF3).

**Uiterlijk:**  
De GUI is duidelijk, hier zijn verder geen onduidelijkheden in mogelijk vanuit de kant van de klant (NF4).

**Betrouwbaarheid:**   
Het systeem heeft een uptime van tenminste 90% (NF5).

**Schaalbaarheid:**  
Het aantal deelnemende werkstations kan worden uitgebreid tot maximaal 10 (NF6).

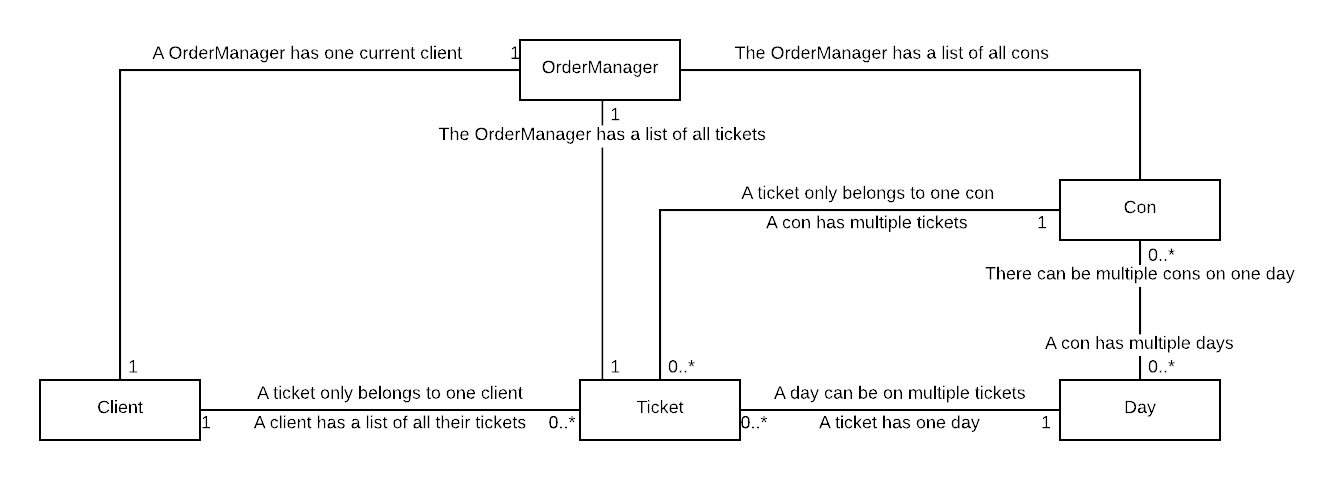
## Doel van dit document

Dit document verantwoordt de architectuur voor de servers en de klanten. Daarnaast worden de interfaces voor de communicaties met de servers en klanten beschreven met betrekking tot interoperabiliteit. Er worden zeven aspecten van de architectuur beschreven: domeinmodel, persistentie, communicatie, realisatie van niet-functionele eisen, componenten, deployment en specificatie van interfaces.

# Domeinmodel

*In dit hoofdstuk worden een of meer klassendiagrammen van het domein getoond. Daarnaast wordt een toelichting gegeven en worden gemaakte keuzes verantwoord. Tot slot wordt de afbakening van het project besproken.*

## Domeinmodel



## C:\Users\miche\Downloads\Klassendiagram van het domein v3.pngKlassendiagram van het domein

## Bijlage

Een **Client** heeft een username, email en een password. Deze zijn de gegevens die gebruikt worden bij het inloggen en registreren. Een **Client** heeft een aantal **Tickets**, dit zijn de **Tickets** die hij besteld heeft.

Een **Ticket** heeft een **Day**, dit is een klasse die bestaat uit een **DateTime** en een String van de dag naam dus bijvoorbeeld: day: Wednesday; date: 22/11/2017. Ik heb dit gedaan zodat je een datum en een dag kan meegeven zodat je meteen weet bij welke datum welke dag hoort. Een **Ticket** heeft ook een **Con** en een kon kan meerdere **Tickets** hebben, op de **Ticket** zelf zal de naam van de **Con** staan.

De **OrderManager** is de façade klassen die communiceert met de Gui. Alle methodes die deze klassen heeft staan in het interface **ILogic**. Dit zijn alle methodes de Gui van de Logic ooit nodig zal hebben.

## Afbakening

Het volledige ontwerp van de bestaande ticketverkoop app wordt niet bespreken in dit document. Alleen de interfaces ten behoeve van interoperabiliteit worden besproken. Dus de onderdelen **IRemotePublisherForListener**, **RemotePublisher**, **Remote** en **IRemotePropertyListener** worden in dit document niet behandeld, maar worden in de verdere diagrammen wel gebruikt.

Verder hebben alle private attributen en getter en een setter, ik heb dit niet in de diagrammen opgenomen om er voor te zorgen dat het zo overzichtelijk mogelijk te houden.

# Componenten

*In dit hoofdstuk wordt de opdeling in softwarecomponenten besproken aan de hand van een Componentendiagram met toelichting. Daarnaast wordt koppeling en synchronisatie tussen de componenten besproken en de opdeling in software packages.*

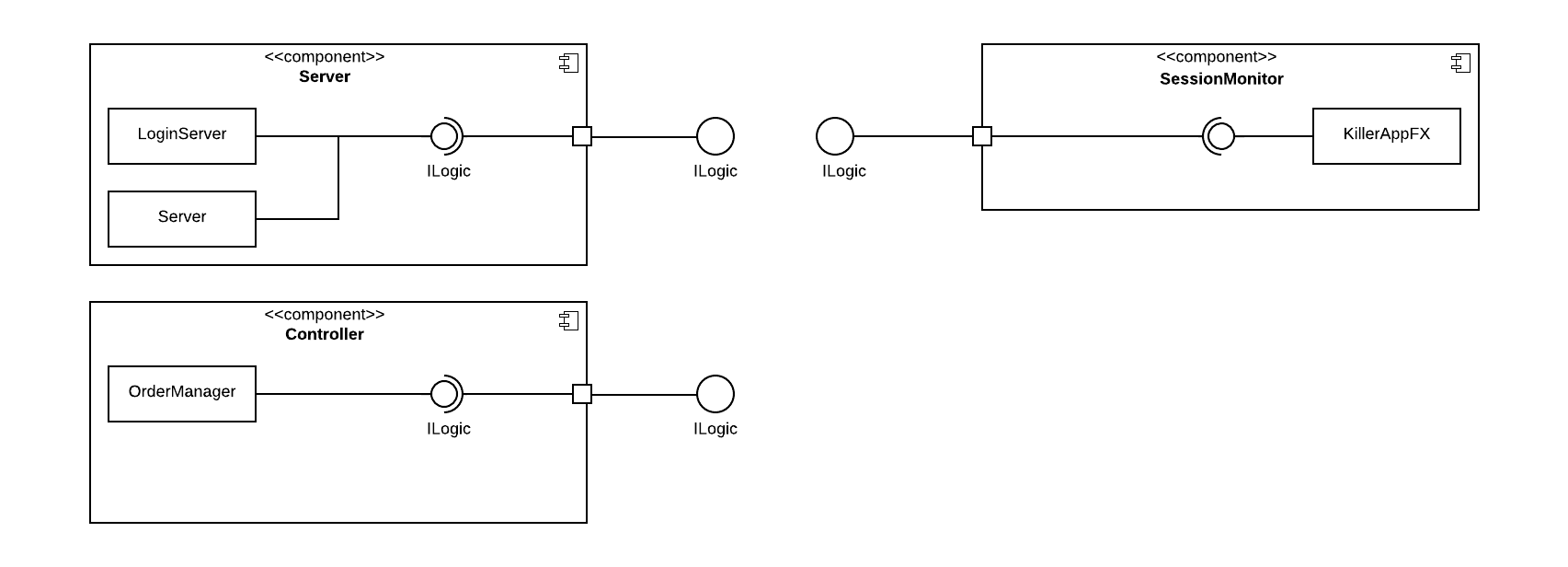
*Een component is een modulair deel van het systeem dat beschreven wordt in termen van aangeboden en gevraagde interfaces. In principe kan een component vervangen worden door een andere component, mits die andere component aan dezelfde interfaces voldoet. Er wordt onderscheid gemaakt tussen componenten en subsystemen: een component is altijd een executeerbare eenheid, maar een subsysteem is geen executeerbare eenheid. Tip: kies voor iedere component een naam die duidelijk verschilt van de namen van de klassen die in die component zitten.*

*De volgende onderwerpen worden in dit hoofdstuk besproken:*

* *Componentendiagram (opdeling componenten)*
* *Soort koppeling (RMI, HTTP, SQL, …)*
* *Synchronisatie (zie ook persistentie/communicatie)*
* *Service(s) per component (voor welke component interessant; aanduiding van behoeften (naam methode) per interface)*
* *Allocatie objecten (binnen welke component)*
* *Remote objecten (welke objecten worden op afstand aangesproken)*
* *Packagestructuur*

*Er zijn verschillende manieren om componentdiagrammen te tekenen, zelfs binnen UML. Hier is gekozen voor componenten opgebouwd uit klassen en interfaces. Klassen kunnen interfaces realiseren of hebben behoefte aan interfaces. Deze interfaces kunnen binnen een component met elkaar verbonden worden en/of via een zogenaamde poort beschikbaar worden gesteld aan andere componenten. Een poort (engels: port) wordt weergegeven door middel van een vierkantje op de rand van de component. Een poort kan een interface aanbieden (aangeboden poort) of behoefte hebben aan een interface (benodigde poort). Een poort die zowel een interface aanbiedt als een interface nodig heeft noemen we een complexe poort. Meestal wordt een poort gerealiseerd door middel van een object dat de aanvraag van services doorgeeft naar een ander object binnen de component. Tijdens uitvoering van het programma zal een benodigde poort een verbinding leggen met een aangeboden poort van een andere component. Voor meer informatie over componentendiagrammen, zie Hoofdstuk 14 van Praktisch UML van Jos Warmer en Anneke Kleppe (vijfde editie, uitgever Pearson).*

## Componenten diagram



## Bijlage

De **Server** beheert de taken zoals de updates versturen en ervoor te zorgen dat alle data die van die **Client** afkomt afhandelt.

De **Client** stuurt als het ware date naar de **Controller** die dit communiceert met de **Server**.

De **Controller** is de verantwoordelijk voor de communicatie tussen de **Client** en de **Server**.

## Koppeling tussen componenten

Voor onderlinge real-time communicatie wordt Remote Method Invocation (RMI) gebruikt. De koppeling tussen de componenten wordt gerealiseerd door een interface, ILogic. Omdat alle componenten informatie nodig hebben van alle andere componenten zijn deze verbonden met de interface.

## Synchronisatie tussen componenten

De componenten worden gesynchroniseerd door het interface dat de controller implementeert. Deze wordt door de server extend zodat de server deze data kan afhandelen.

## Services per component

Services worden gerealiseerd door een interface, **ILogic**. **ILogic** verbind deze drie componenten:

**Client**

* Vraagt een instantie van **localClient** aan de **Controller**.

**Controller**

* Vraagt een instantie op van een **Client** voor de **localClient**.
* Vraagt een lijst van alle tickets op van de **Server**.
* Vraagt een lijst van alle tickets van de **LocalClient** op van de **Server**.

**Server**

* Server stuurt een lijst van alle tickets naar de **ILogic**.
* Server stuurt een lijst van alle tickets van de **localClient** naar de **ILogic**.
* Server stuurt een lijs van alle dagen naar de **ILogic**.

## Allocatie van objecten

Allocatie van objecten per component

* **Client**: instantie van de huidige klant of wel de **localClient**.
* **Controller**: instanties van de lijst van alle **Clients**, alle bestaande tickets, alle tickets voor de huidige klant.
* **Server**: instanties van **Clients** en **Tickets**.

## Remote objecten

Remote objecten per component:

* **Client**: remote instantie van **localClient**.
* **Controller**: remote instanties van **Ticktes**, **Days** en **Cons**.
* **Server**: remote instanties van **Client** en **Tickets**.

## Packagestructuur

Er is een package Logic waar alle algemene klassen in zitten. Er is een package GUI waar de klassen in komen die iets met de GUI te maken hebben. Er is een package Login waar de klassen in komen die te maken hebben met het inloggen. En als laatst is er een package Server waar de klassen in komen die de server ondersteunen.

Klassen/interfaces per package:

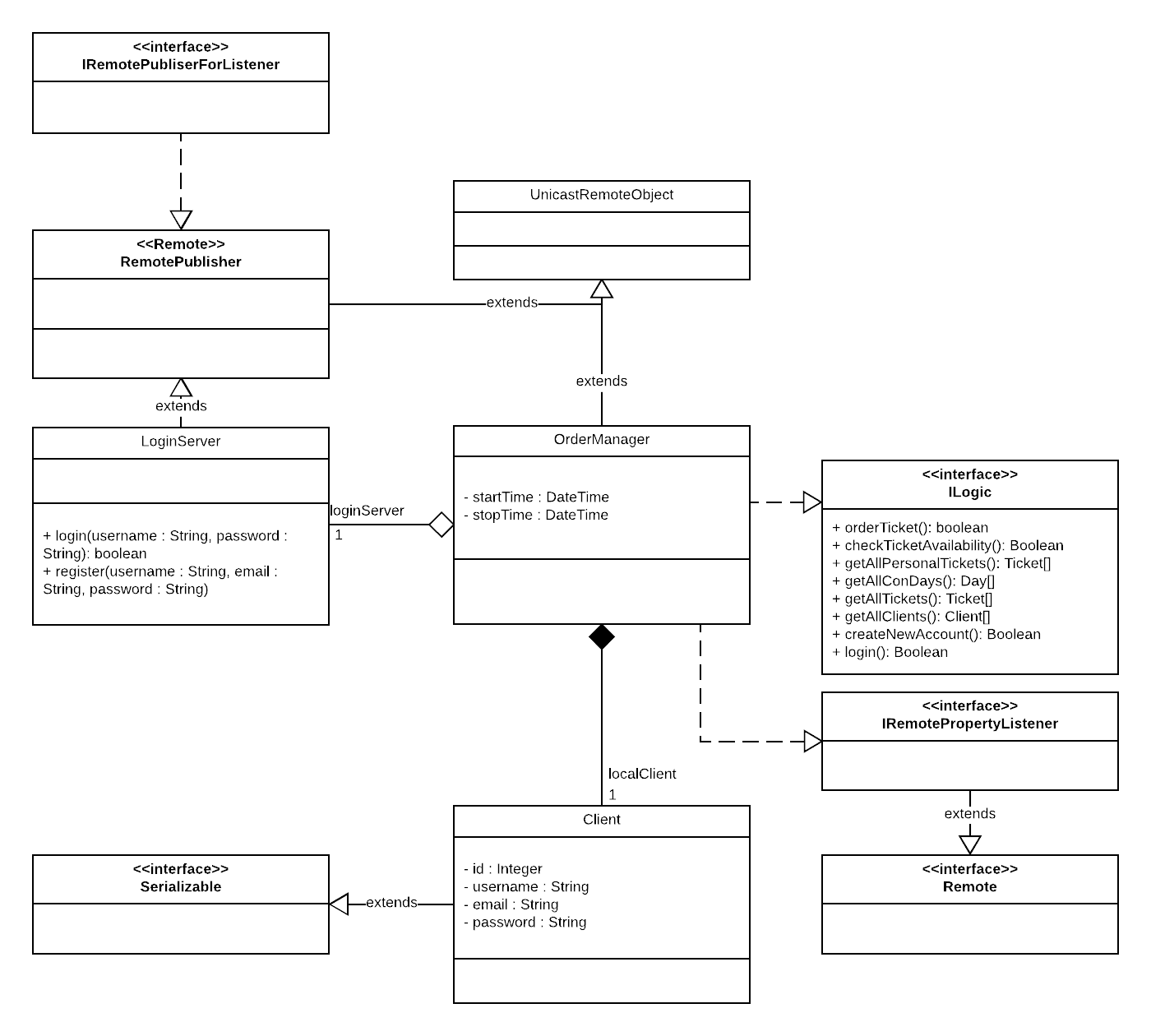
* **Logic**: Ticktet, Day, Con, OrderManager en Client.
* **Interfaces**: ILogic
* **GUI**: LoginScreenFX en HomeScreenFX.
* LoginServer: LoginServer.
* **Server**: Server.
* **FontysPublisher**:, IRemotePropertyListener, IRemotePublisherForListener, Publisher en RemotePublisher.

# Communicatie

*In dit hoofdstuk wordt de communicatie van en tussen objecten van verschillende componenten beschreven. Beschrijf welke eigenschappen van welke objecten moeten worden gecommuniceerd. Beschrijf hoe communicatie wordt gerealiseerd (bijvoorbeeld m.b.v Remote Method Invocation, RMI). Dit kan verschillen per eigenschap. Beschrijf tot slot wanneer communicatie plaatsvindt (push-pull, welk object is leidend? etc.) Ook dit kan verschillen per eigenschap.*

*In het geval RMI wordt gebruikt voor communicatie tussen componenten: stel een klassendiagram op en geef per klasse/interface aan of deze UnicastRemoteObject, Remote of Serializable is. In het geval gebruik gemaakt wordt van (SOAP) webservices: stel een klassendiagram op en geef aan welke services door welke klasse worden geleverd.*

## Klassendiagram voor inloggen met RMI



## Bijlage

De klassen die van de **FontysPublisher** zijn afgeleid zijn in dit diagram niet ingevuld. Dit zijn: **IRemotePropertyListener**, **IRemotePublisherForListener**, **Publisher** en **RemotePublisher**.

De **OrderManager** is vormt de communicatie tussen de **LoginServer** en de rest van de applicatie. De LoginServer extends de **RemotePublisher** klasse, deze functionaliteiten zorgen ervoor dat de klasse kan werken als een daadwerkelijke server. De **RemotePublisher** maakt op zijn beurt weer gebruik van de **IRemotePublisherForListener**, deze interface zorgt ervoor dat de **Listener** zich kan aanmelden om up to date te blijven van een property, als er iets veranderd aan deze property krijgt de **RemotePublisher** dit binnen en kan de property binnen de **RemotePublisher** geüpdatet worden.

De **OrderManager** implementeert ook het interface **ILogic**, hierin staan alle methodes die de gegevens uit een database halen en deze worden door de **OrderManager** doorgegeven naar de GUI zodat de klant al zijn data kan zien.

Ook heeft de **OrderManager** het interface **IRemotePropertyListener** geimplementeerd. Dit interface is aangemeld op een property, dit is de property waar de **IRemotePublisherForListener** ook op aangemeld is. De Publisher update deze property en de **IRemotePropertyListener** vangt deze update op en handelt de data af. Deze interface extends **Remote**, dit is een klasse die standaard in Java zit.

De **RemotePublisher** en **OrderManager** extenden beide **UnicastRemoteObject**. Deze klasse wordt gebruikt om ervoor dat deze klassen kunnen communiceren met het **Remote** object.

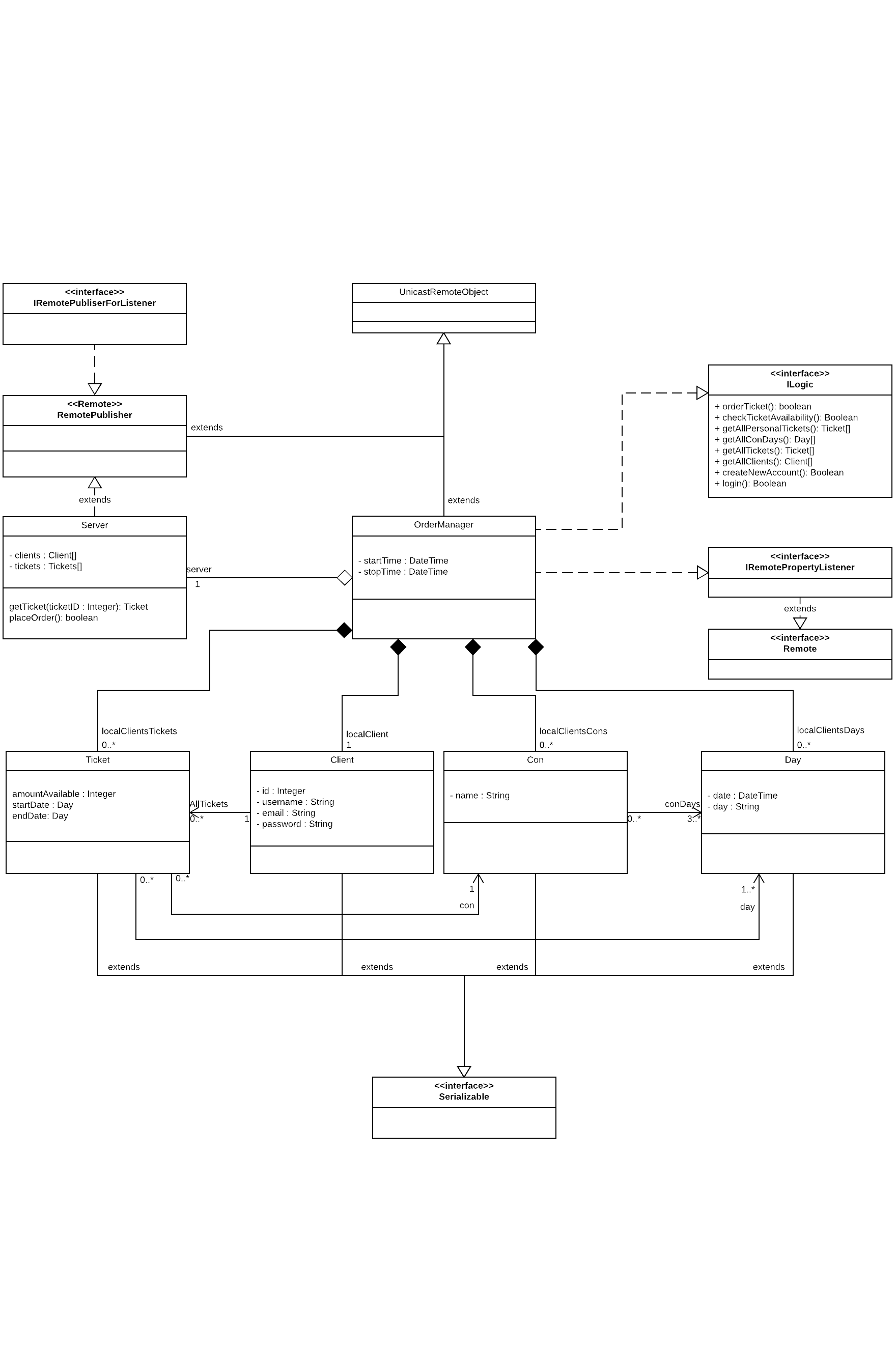
Verder is de **Client** **Serializable**, dit is nodig om hem te kunnen versturen van de **Client** naar de **LoginServer**. De **OrderManager** heeft een instantie van **Client**, dit is de huidige klant ofwel de **localClient**.

Een **Client** heeft een username, email en een password. Deze zijn de gegevens die gebruikt worden bij het inloggen en registreren. Een **Client** heeft een aantal **Tickets**, dit zijn de **Tickets** die hij besteld heeft.

Een **Ticket** heeft een **Day**, dit is een klasse die bestaat uit een **DateTime** en een String van de dag naam dus bijvoorbeeld: day: Wednesday; date: 22/11/2017. Ik heb dit gedaan zodat je een datum en een dag kan meegeven zodat je meteen weet bij welke datum welke dag hoort. Een **Ticket** heeft ook een **Con** en een kon kan meerdere **Tickets** hebben, op de **Ticket** zelf zal de naam van de **Con** staan.

De **OrderManager** is de façade klassen die communiceert met de Gui. Alle methodes die deze klassen heeft staan in het interface **ILogic**. Dit zijn alle methodes de Gui van de Logic ooit nodig zal hebben.

## Klassendiagram voor communicatie met server met RMI



## Bijlage

De klassen die van de **FontysPublisher** zijn afgeleid zijn in dit diagram niet ingevuld. Dit zijn: **IRemotePropertyListener**, **IRemotePublisherForListener**, **Publisher** en **RemotePublisher**.

Voor dit klassendiagram gelden voor en groot deel dezelfde klassen, maar een aantal dingen zijn hier anders.

De **OrderManager** is vormt de communicatie tussen de **Server** en de rest van de applicatie. De **LoginServer** extends de **RemotePublisher** klasse, deze functionaliteiten zorgen ervoor dat de klasse kan werken als een daadwerkelijke server. De **RemotePublisher** maakt op zijn beurt weer gebruik van de **IRemotePublisherForListener**, deze interface zorgt ervoor dat de **Listener** zich kan aanmelden om up to date te blijven van een property, als er iets veranderd aan deze property krijgt de **RemotePublisher** dit binnen en kan de property binnen de **RemotePublisher** geüpdatet worden.

De **OrderManager** implementeert ook het interface **ILogic**, hierin staan alle methodes die de gegevens uit een database halen en deze worden door de **OrderManager** doorgegeven naar de GUI zodat de klant al zijn data kan zien.

Ook heeft de **OrderManager** het interface **IRemotePropertyListener** geïmplementeerd. Dit interface is aangemeld op een property, dit is de property waar de **IRemotePublisherForListener** ook op aangemeld is. De Publisher update deze property en de **IRemotePropertyListener** vangt deze update op en handelt de data af. Deze interface extends **Remote**, dit is een klasse die standaard in Java zit.

De **RemotePublisher** en **OrderManager** extenden beide **UnicastRemoteObject**. Deze klasse wordt gebruikt om ervoor dat deze klassen kunnen communiceren met het **Remote** object.

Verder is de **Client** **Serializable**, dit is nodig om hem te kunnen versturen van de **Client** naar de **LoginServer**. De **OrderManager** heeft een instantie van **Client**, dit is de huidige klant ofwel de **localClient**. De klassen **Ticket**, **Con** en **Day** zijn ook **Serializable**. De klassen zijn nodig om door te sturen zodat op de database de gegevens van de **Client** geüpdatet kunnen worden.

# Persistentie

*In dit hoofdstuk wordt de persistentie van de objecten beschreven voor iedere component. Beschrijf welke eigenschappen van welke objecten moeten worden bewaard. Beschrijf hoe opslag wordt gerealiseerd (database, serialiseren, XML, etc.). Dit kan verschillen per eigenschap. Voeg het ERD met beschrijving toe indien gegevens worden opgeslagen in een database. Beschrijf tot slot wanneer opslag plaatsvindt. Ook dit kan verschillen per eigenschap.*

## Eigenschappen van objecten m.b.t. persistentie

Om een onderbroken sessie op een later tijdstip te kunnen vervolgen dienen er een aantal eigenschappen van objecten opgeslagen te worden:

* **Client**: id, username.
* **Ticket**: day(s), con.
* **Con**: name, day(s).

## Data-opslag

Gegevens van de **Client** worden opgeslagen in een database. Hieronder vallen de id, username, email en passwoord, maar ook de tickets die ze gekocht hebben voor welke datum. Verder staan er in de database een aantal dagen in de **Day** tabel, dit zijn alle dagen wanneer er een **Con** is, deze dagen worden alleen maar uitgelezen en worden in de database al gekoppeld aan een con en een ticket.

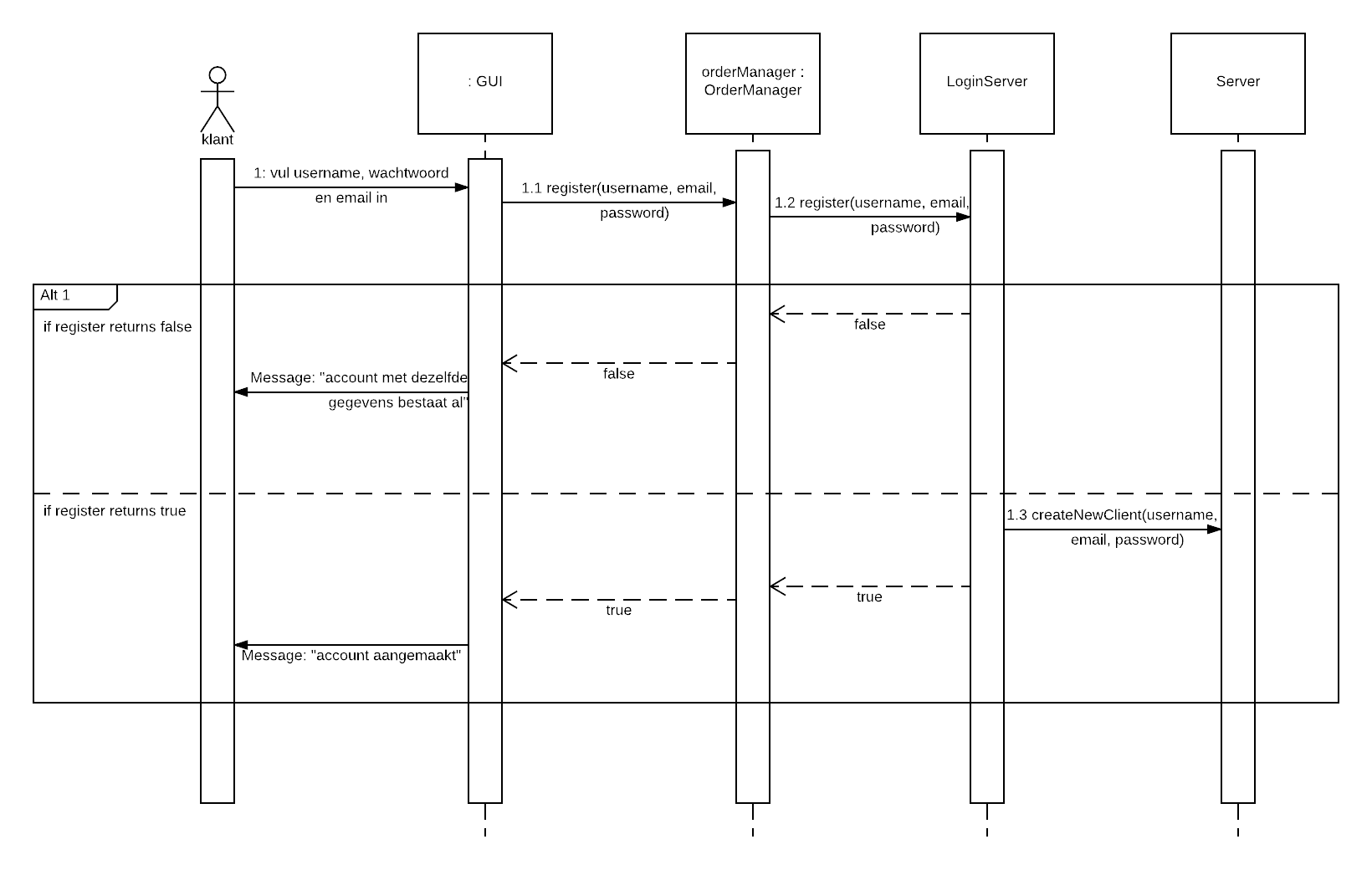
Als een klant inlogt worden zijn gegevens opgeslagen in de **localClient**, dit is steeds per systeem maar een klant. Als de klant een kaartje aanklikt wordt deze meteen van het totaalaantal beschikbare kaartjes afgehaald in de database. Als er een kaartje niet beschikbaar meer is krijgt de klant een melding te zien.

# Detailed design per component

*In dit hoofdstuk wordt het detailed design (implementatiemodel) per component beschreven. Maak hiervoor gebruik van klassendiagrammen en sequence diagrammen. Voor specifieke doeleinden kun je gebruik maken van objectdiagrammen, statemachinediagrammen, activiteitendiagrammen, etc. Voor meer informatie over UML-diagrammen, zie Praktisch UML van Jos Warmer en Anneke Kleppe (vijfde editie, uitgever Pearson).*

## C:\Users\miche\Downloads\Sequence diagram login v2.pngSequence diagram login

## Sequence diagram register



# Realisatie niet-functionele eisen

*In dit hoofdstuk wordt de realisatie van (overige) niet-functionele eisen besproken. De volgende onderwerpen dienen tenminste te worden besproken: betrouwbaarheid, performance, beveiliging, schaalbaarheid. Eventueel kunnen nog extra niet-functionele eisen worden besproken.*

**Functionaliteit**

De functionaliteiten eisen (NF2 en NF3) worden geregeld in de bijbehorende databases. De uitbreidbare code (NF1) wordt o.a. gecontroleerd door programma’s zoals SonarLint en SonarQube, ook wordt hier rekening mee gehouden tijdens het ontwerpen (alle klassendiagrammen en het domeinmodel).

**Uiterlijk**

De eisen die aan het uiterlijk staan kunnen gecontroleerd worden door mensen door de applicatie te laten lopen en als er iets onduidelijk is voor de tester kan dit aangepast worden voor de definitieve oplevering van de applicatie.

**Betrouwbaarheid**

De eis van dat het systeem een uptime heeft van minstens 90% (NF5) kan getest worden door mensen de applicatie te laten gebruiken en dan waarnemen hoe veel uptime het systeem heeft. De manier waarop dit in code onderhouden wordt is om ervoor te zorgen dat de code niet rommelig is, zodat de clients vrij weinig vragen van de pc die de klant gebruikt en alleen nuttige informatie naar de server te sturen zodat hier ook zo min mogelijk data overdracht zit.

**Schaalbaarheid**

De eis aan de schaalbaarheid, dat het aantal werkstations uitgebreid kan worden tot maximaal 10, kan getest worden door de servers te runnen en 10 clients en dan te kijken of het systeem dit aankan.

**Performance**

N.v.t.

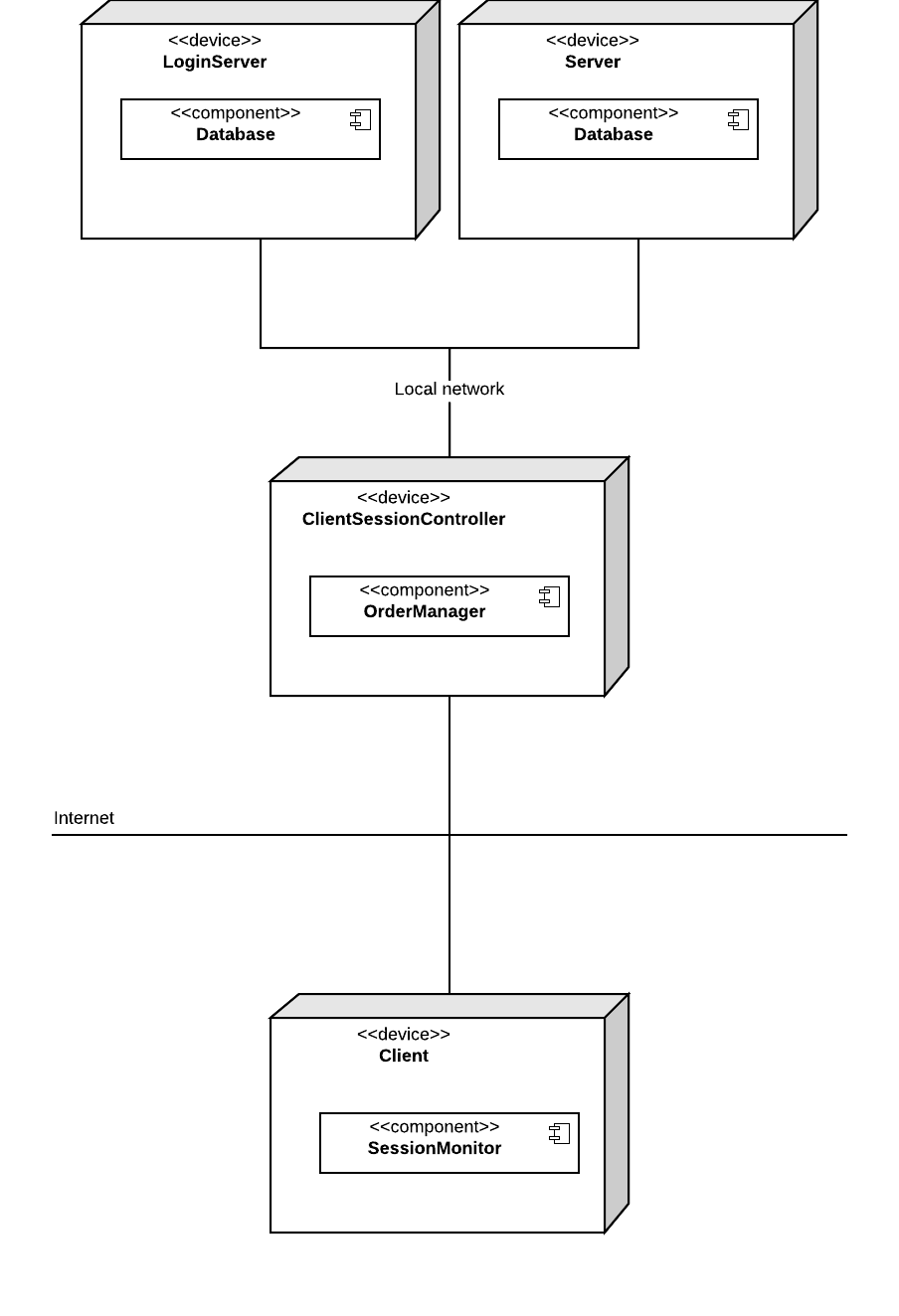
**Beveiliging**

N.v.t.

# Deployment

*In dit hoofdstuk wordt de toekenning van softwarecomponenten aan hardware besproken aan de hand van een Deploymentdiagram met toelichting. Daarnaast wordt de koppeling (lokaal netwerk, internet, etc.) tussen de nodes besproken.*

## Deployment diagram



## Bijlage

De **LoginServer** heeft een database met daarbij de inloggegevens die horen bij de **Client**.

De **Server** heeft een database met alle overige gegevens en de koppeling met de klant, hier staat alleen de gebruikersnaam en het unieke ID in van de klant.

De **ClientSessionController** geeft de data door van de **Client** naar de **Server**.

De **Client** geeft input en stuurt data via de **ClientSessionController** naar de **Servers**.

# Specificatie van interfaces

*In dit hoofdstuk wordt de specificatie van interfaces besproken. Voor ieder interface wordt per methode gedefinieerd:*

* Naam methode
* Naam en type argumenten
* Precondities
* Type returnwaarde
* Beschrijving
* Aanleiding voor excepties

## ILogic

**Methode:** orderTicket(localClient : Client, currentTicket : Ticket): Boolean

**Preconditie:** een localClient en een ticket die de klant gekozen heeft.

**Beschrijving:** geeft een boolean of het kopen van de ticket gelukt is.

**Excepties:** NullReferenceException, als de ticket uitverkocht is tijdens dat de client de bestelling wil plaatsen.

**Methode:** getAllPersonalTickets(localClient : Client): Ticket[]

**Preconditie:** een localClient.

**Beschrijving:** voor de localClient worden al zijn tickets binnen gehaald.

**Excepties:** NullReferenceException, als de client nooit tickets besteld heeft.

**Methode:** getAllConDays(): Day[]

**Preconditie:** niets is nodig voor deze methode.

**Beschrijving:** alle bestaande cons worden binnengehaald.

**Excepties:** NullReferenceException, als er geen cons zijn.

**Methode:** getAllTickets(): Ticket[]

**Preconditie:** niets is nodig voor deze methode.

**Beschrijving:** alle bestaande tickets worden binnengehaald.

**Excepties:** NullReferenceException, als er geen tickets zijn.

**Methode:** getAllClients(): Client[]

**Preconditie:** niets is nodig voor deze methode.

**Beschrijving:** alle bestaande klanten worden binnengehaald.

**Excepties:** NullReferenceException, als er geen klanten zijn.

**Methode:** createNewAccount(username : String, email : String, password : String): Boolean

**Preconditie:** een username, emal en een wachtwoord om het account mee kunnen te maken.

**Beschrijving:** Een nieuw account wordt aangemaakt voor de klant.

**Excepties:** EntityExistsException, als het account al bestaat in de database.

**Methode:** login(username : String, password : String): Boolean

**Preconditie:** een username en een wachtwoord om de klant als uniek te kunnen identificeren.

**Beschrijving:** de klant met een bestaand account logt in.

**Excepties:** NotInitializedException, als de klant niet bestaat in de database.