

AARHUS UNIVERSITET

BeerBong Battle

Bilagsrapport

Gruppe 5

201711532	Andreas Elgaard Sørensen
201710717	Mathias Tørnes Pedersen
201710709	Mads Stengaard Jørgensen
201710702	Mark Højer Hansen
201710692	Umut Sahin

Vejleder

Jesper
jrt@ase.au.dk

17. december 2019



AARHUS UNIVERSITY

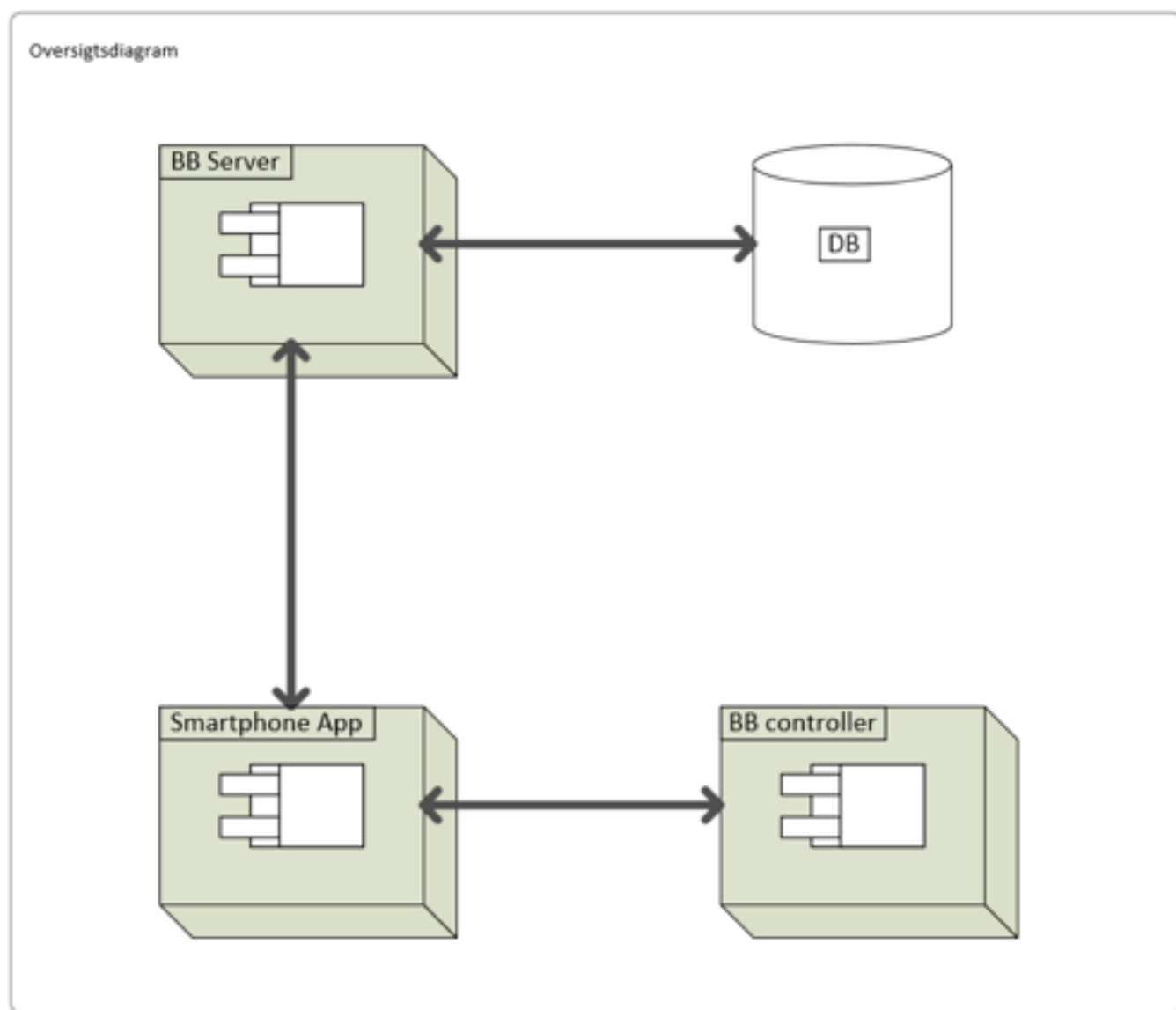
Indledning 1

1.1 Problemformulering

I Danmark er der en stigende tendens til at medbringe ølbonger til både fester og festivaler. Vi har som projektgruppe valgt at skabe bånd mellem alle slags fester og festdeltagere vha. af vores BeerBong. BeerBong er en digitaliseret ølbong, der skal gøre det muligt at konkurre i at drikke øl på tid på tværs af fester, festivaler m.m. Festdeltagerne skal downloade en mobilapplikation, som er trådløst tilknyttet en BeerBong. Her er det muligt at finde en modstander og ansøge om et spil. Når en modstander er fundet begynder spillet, hvor man på skift fylder sin BeerBong op, og efter appen anvisninger følger spillet gangs. Efter begge spillere har drukket deres øl, skal det være muligt at kommunikere med hinanden igennem en chat forum. Det er også muligt at lave et lokalt spil, hvor spillernes tider ikke uploades til en online topscore, men blot holdes lokalt på appen tilknyttet en BeerBong. I det lokale spil indtaster brugerne et antal spillere og deres navne, hvorefter de på tur drikker en øl i Beerbong og opnår deres tid. Efter alle har haft deres tur, er muligt at se på en offline topscore, som viser, hvem der har vundet spillet – den hurtigste øldrikker

1.2 Systembeskrivelse

BeerBong Battle er et system som overordnet består af 3 moduler. Det første modul er Smart-Phone Applikationen, som er en Xamarinudvikling android applikation udviklet i UWP. Det andet er BeerBong Server som er serveren der står for kommunikationen mellem databasen og SmartPhone applikationen. Dette modul indeholder også DB, som er databasen som indeholder brugernavne, passwords og tider. Det sidste modul er BeerBong controller som er en ølbong med en række sensorer som kontrolleres af en Raspberry Pi Zero Wireless.



Figur 1.1: Systemskitse over BeerBong.

Kravspekifikation 2

2.1 Overordnede krav

Der er til dette semesterprojekt blevet stillet en række overordnede krav som er gennemgående for alle semesterprojektergrupper. Disse krav dækker over hvad der skal til for at kunne bestå I4PRJ.

- Projektet skal inddrage faglige aspekter fra samtlige fag på 4. semester IKT. Dette skal dokumenteres i projektrapporten og bør inddrages af de studerende til eksamen.
- Projektet skal have et passende omfang, så alle i projektgruppen kan arbejde med projektet.
- Projektet skal være af en karakter der tillader at læringsmålene i faget opfyldes.

2.2 MoSCoW

Der er blevet udviklet en MoSCoW analyse til at definere de funktionelle krav til projektet. Analysen består af at opsætte en række funktionelle krav til projektet. Dette gøres ved at indele de funktionelle krav i 4 forskellige kategorier. Kategorierne afspejler vigtigheden af at få det pågældende punkt færdiggjort. Punkterne fra de MoSCoW modellen bliver anvendt i de funktionelle krav, der definere systemets opførelse.

M – Must

- BeerBong Server **skal** have et leaderboard med plads til minimum 100 tider.
- BeerBong Server **skal** give mulighed for at oprette et personligt login.
- BeerBong App **skal** have et leaderboard med plads til minimum 10 tider
- BeerBong App **skal** have en brugergrænseflade.
- BeerBong App **skal** give mulighed for brugeren at oprette et personligt login
- BeerBong App **skal** have en online feature.
- BeerBong App **skal** give mulighed for burgeren at logge ind med sit personlige login
- BeerBong Controller **skal** kunne detektere at indholdet i ølbongen er 33 cl $[+/- 3 \text{ cl}]$.
- Ølbongen **skal** overholde dimensionerne højde på 100 cm $[+/- 2 \text{ cm}]$ og en diameter på 10 cm $[+/- 1 \text{ cm}]$.
- BeerBong Controller **skal** have en kapacitet på 0,67 liter $[+/- 0.05 \text{ liter}]$.
- BeerBong Controller **skal** kunne registrere tider i tidsrummet 0,5-10 sek $[+/- 0,1 \text{ sek}]$.

S – Should

- BeerBong App **burde** have en chat funktionalitet mellem deltagere.

C - Could

- BeerBong App **kunne** visualisere tidsforskellen grafisk mellem spillerne.

- BeerBong App **kunne** have en lyd, som meddeler spillere om start.

W - Wont

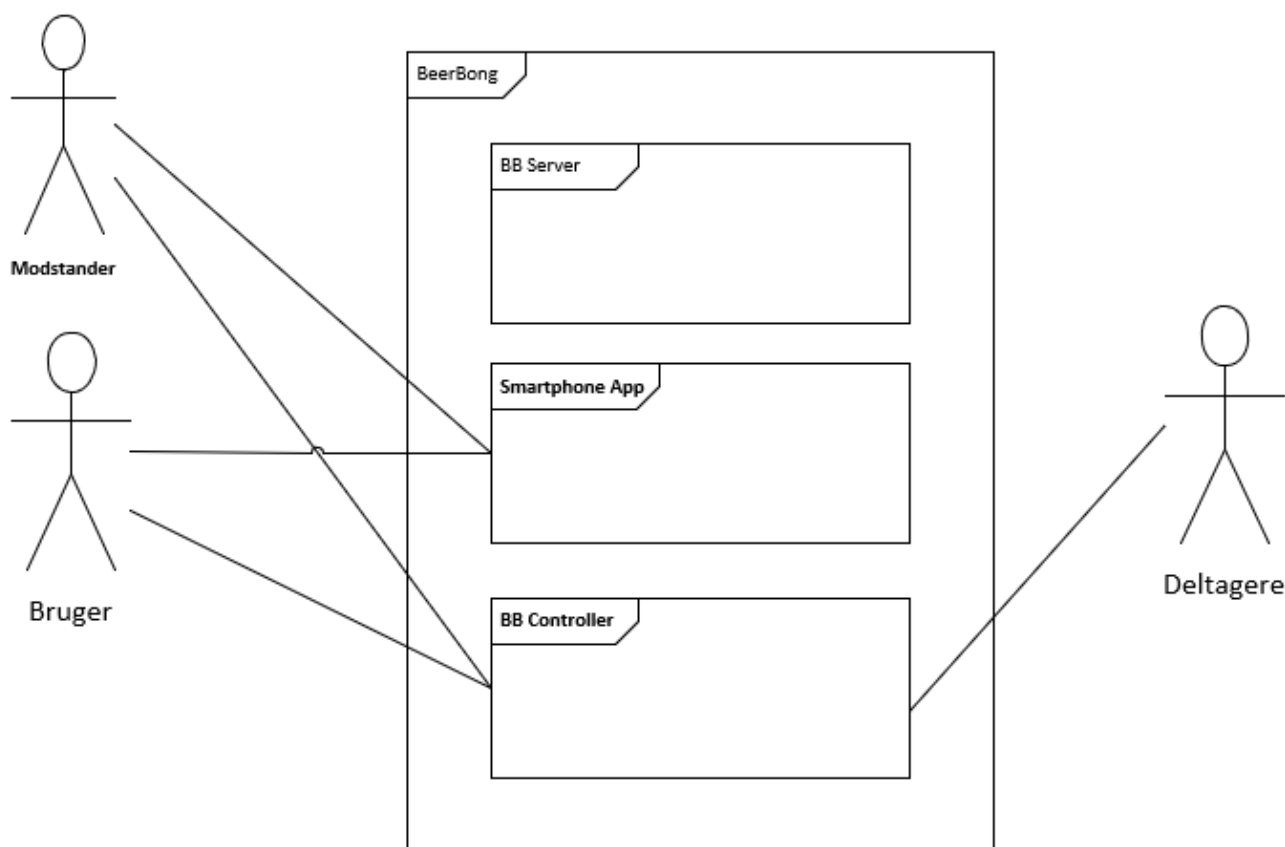
- BeerBong Server **vil ikke** være kompatible med andre systemer end Android.
- BeerBong Server **vil ikke** have et pointsystem
- BeerBongBattle **vil ikke** være kompatible med andre systemer end Android.
- BeerBong Controller **vil ikke** kunne kende forskel på forskellige væsker.

2.3 Funktionelle krav

2.3.1 Indledning

Dette afsnit omhandler de funktionelle krav for BeerBongBattle. Dette indebærer de overordnede diagrammer Aktør-kontekst diagram og use-case diagram, samt alle use cases fully dressed.

På Figur 2.1 ses aktør-kontekst diagrammet for BeerBongBattle. De to aktører er beskrevet i tabel 2.1 og tabel 2.2.



Figur 2.1: Aktør Kontekstst diagram

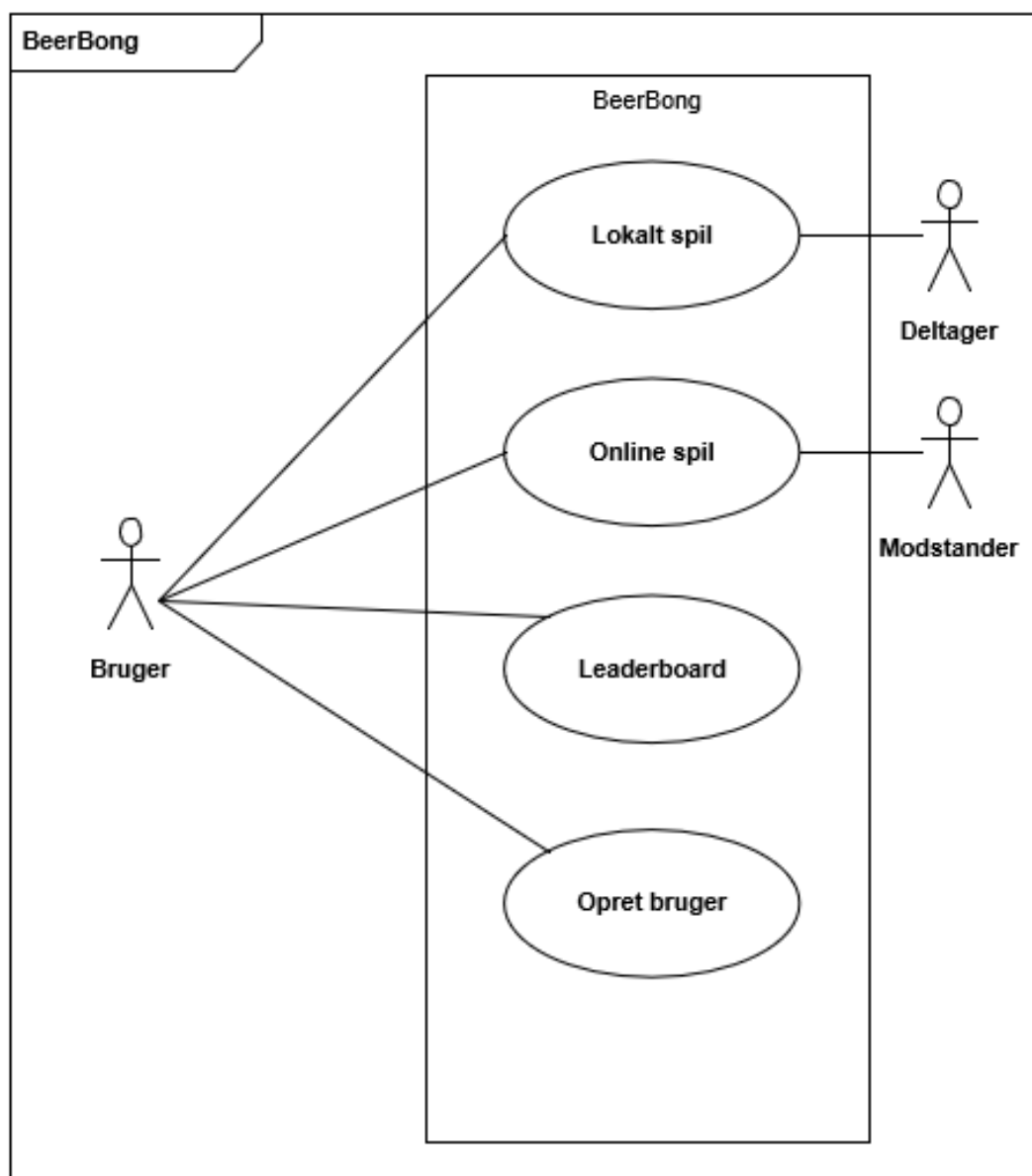
Aktørnavn	Bruger
Rolle	Primær
Beskrivelse	Brugeren skal kunne tage en øl i ølbongen, og få taget tid igennem appen.

Tabel 2.1: Aktørbeskrivelse af brugeren

Aktørnavn	Deltager
Rolle	Sekundær
Beskrivelse	Deltageren skal kunne tage en øl i bogen, men tilgår ikke appen.

Tabel 2.2: Aktørbeskrivelse af brugeren

I ovenstående tabeller beskrives aktørerne. Brugeren af systemet er også den der spiller spillet, hvor deltageren er en del af det lokale spil den primære aktør, brugeren, opretter. I tilfælde af multiplayer ville begge aktører være primære. På figur 2.2 ses use case diagrammet for BeerBongBattle. Som det ses på diagrammet bruger brugeren alle tre use cases, da den har bruger appen. Hvorimod Deltager kun bruger Lokalt spil appen, fordi de kun kan bruge ølbongen og få en tid derigennem.



Figur 2.2: Use case diagram

2.3.2 Fullydressed Use Cases

Use case 1 - Lokalt

Navn	Lokalt spil
Mål	At alle brugere har drukket en øl og fået tildelt en tid.
Initiering	BeerBongBattle initieres af brugeren.
Aktører	Primær: Bruger. Sekundær: Deltager
Antal samtidige forekomster	0
Prækondition	Applikationen på smartphonen er startet, og startside vises.
Postkondition	Alle tilmeldte spillere har drukket en øl, og fået vist deres tid.
Hovedscenarie	<ol style="list-style-type: none"> 1. Bruger vælger lokalt spil. 2. Bruger præsenteres for spillervalg på applikationen. 3. Bruger vælger antal spillere. 4. Bruger trykker 'næste'. 5. Bruger skriver navne på spillerne. 6. Bruger præsenteres for listen af spillere med mulighed for redigering. [Extension 1: Flere brugere] 7. Bruger starter spil. 8. Bruger fylder ølbongen. [Extension 1: Ikke nok øl] 9. Grænsefladen meddeler spilleren at ølbongen er klar. 10. Bruger drikker al øl i bongen. [Extension 2: Flere brugere] [Extension 3: Countdown udløber] 11. Tiderne for alle spillere vises på applikationen.
Extensions	[Extension 1: Flere deltagere] <ol style="list-style-type: none"> 1. Bruger trykker rediger og tilføjer flere deltagere. 2. Use Case genoptages fra punkt 7. [Extension 2: Ikke nok øl] <ol style="list-style-type: none"> 1. Grænseflade meddeler at der ikke er nok øl i bongen. 2. Use case gentages fra punkt 7. [Extension 3: Countdown udløber] <ol style="list-style-type: none"> 1. Bruger gives ugyldig, og use case startes fra punkt 5.

Tabel 2.3: Use Case 1 - Start lokalt spil

Use case 2 - Online Spil

Navn	Online spil
Mål	At to spillere har konkurreret mod hinanden på forskellige enheder.
Initiering	BeerBongBattle initieres af brugeren.
Aktører	Primær: Bruger. Sekundær: Modstander
Antal samtidige forekomster	2
Prækondition	Bruger er logget ind.
Postkondition	Bruger har drukket en øl og fået sin tid, samt modstanders tid.
Hovedscenarie	<ol style="list-style-type: none"> 1. Bruger trykker på 'Start Online Spil'. 2. Bruger trykker på 'Find modstander'. 3. Applikationen søger efter modstander. 4. Applikationen finder modstander. 5. Applikationen viser ny side med fundet modstanders navn. 6. Applikation starter nedtælling på et minut. 7. Bruger fylder ølbong. [Extension 1: Ikke nok øl/Ølbong ikke fyldt]

Use case 3 - Vis online leaderboard

Navn	Vis online leaderboard
Mål	Et leaderboard med forskellige tider vises til brugeren.
Initiering	BeerBongBattle initieres af brugeren.
Aktører	Primær: Bruger.
Antal samtidige forekomster	0
Prækondition	Applikationen er startet, bruger er logget ind og startskærm vises.
Postkondition	Applikationen viser leaderboard for online spil.
Hovedscenarie	1. Bruger vælger vis leaderboard. 2. Applikationen præsenterer brugeren for online leaderboard.
Extensions	

Tabel 2.5: Use Case 3 - Vis online leaderboard

Use case 4 - Opret bruger

Navn	Opret bruger
Mål	Brugeren får oprettet et personligt ID, i form af et brugernavn.
Initiering	BeerBongBattle initieres af brugeren.
Aktører	Primær: Bruger.
Antal samtidige forekomster	0
Prækondition	Brugeren har trykket 'Online spil' eller er på startskærmen.
Postkondition	Brugeren har oprettet sig.
Hovedscenarie	1. Bruger trykker "opret bruger" 2. Applikationen præsenterer brugeren for opret side. 3. Bruger indtaster et brugernavn og to ens kodeord. 4. Bruger trykker "Opret" 5. Applikation validerer brugernavn og kodeord. [Extension 1: Brugernavn allerede taget] [Extension 2: Kodeord matcher ikke] 6. Bruger sendes tilbage til start siden
Extensions	[Extension 1: Brugernavn allerede taget] 1. Applikationen viser 'Brugernavn optaget' 2. Der fortsættes fra punkt 3. [Extension 2: Kodeord opfylder ikke krav] 1. Applikationen viser 'Passwords matcher ikke'. 2. Der fortsættes fra punkt 3.

Tabel 2.6: Use Case 2 - Start online spil

2.4 Ikke-funktionelle krav

2.4.1 FURPS

Functionality

- BeerBong Controller **skal** have en kapacitet på 0,67 liter [+/- 0.05 liter].

Usability

- BeerBong Server **skal** have et leaderboard med plads til minimum 20 tider
- BeerBong Server **skal** give mulighed for at oprette et personligt login.
- BeerBong App **skal** have et leaderboard med plads til minimum 20 tider
- BeerBong App **skal** give mulighed for brugeren at oprette et personligt login.
- BeerBong App **skal** give mulighed for burgeren at logge ind med sit personlige login
- BeerBong Controller **skal** kunne registrerer væske niveau i ølbong.

Reliability

- Ølbongen **skal** overholde dimensionerne højde på 100 cm [+/- 2 cm] og en diameter på 10 cm [+/- 1 cm].

Performance

- BeerBong Controller **skal** kunne registrere tider i tidsrummet 0,5-10 sek [+/- 0,1 sek].

Supportability

- BeerBong Controller **skal** kunne forbinde til et internet netværk..

2.5 Udviklingsværktøjer

Dette afsnit viser alle udviklingsværktøjer som er blevet brugt i udviklingsprocessen af dette produkt.

- Visual Studio 2019 V.
- Github
- Zenhub
- Google Drive
- Nano Text Editor
- VIM text editor

3.0.1 Indledning

I dette afsnit vil de hardware og softwaremæssige overvejelser og valg i projektet gennemgås og begrundes. Alle refleksioner og over komponent valg vil blive opstillet. Her vil der blive diskuteret fordele og ulemper ved de beslutninger som er taget i projektet.

3.1 BeerBong Controller

3.1.1 Indledning

Dette afsnit omhandler designetvalget af de hardware mæssige moduler i projektet. Der gennemgås først en analyse af hardware modulet, og efterfølgende en analyse af softwaredelen af modulet. Da der kun er et hardware modul som er BeerBong Controlleren, er dette det eneste hardware som der vil laves analyse på.

3.1.2 Analyse

Hardware

Analysen af hardwaren startede ud med at opveje fordele og ulemper ved de mulige microcontrollere, som kunne blive anvendt. De oplagte valg var en version af Raspberry Pi eller Arduino. Begge mikrocontrollere har både fordele og ulemper. Der er tidligere blevet arbejdet meget med 'Raspberry Pi Zero W', hvorimod Arduino miljøet er mere ukendt og der er derfor flere ukendte variabler. Valget faldt derfor på en version af Raspberry Pi. Herfra skulle det besluttes hvilken version der skulle anvendes. De forskellige versioner af Raspberry Pis kommer i sidste ende an på størrelsen på printboardet. Her var det vigtigt at finde en der mødte vores krav bedst muligt. En lang batterilevetid er vigtigt for projektet. Derudover skulle mikrocontrolleren understøtte enten Bluetooth eller WiFi. Derfor faldt valget på den mindste version der var, nemlig en Raspberry Pi Zero W, som både har et lavt energiforbrug og mulighed for implementering af enten Bluetooth eller WiFi kommunikation.

Under analysen af BeerBong Controlleren blev der gjort mange overvejelser omkring om hvilke typer sensorer som skulle bruges til registrering af væske i røret. For at gøre hele designprocessen så overskuelig som muligt, skulle systemet have de samme spændingsniveauer på alle sensorer. Derfor blev der brugt meget tid på at finde sensorer, som kunne forsynes med 3.3 V fra Raspberry Pi. Denne løsning vil først og fremmest gøre batterilevetiden længere, desuden vil det ikke være nødvendigt at lave en levelconverter til dataoverførelse mellem sensor og Raspberry Pi.

Først blev det overvejet om der skulle bruges en flowsensor til at validere hvorvidt at der løb 33 cl igennem mundstykket på bongen. Dette valg ville mindske mulighed for at snyde, fordi brugeren hverken ville kunne hælde for lidt øl i bongen, men heller ikke ville kunne hælde det ud den forkerte vej, så de slap for at drikke det. Derudover er en flowsensor ekstremt præcis til både at måle væske og derfor kan den måle en meget præcis tid. Denne løsning blev desværre udelukket, fordi det var for svært at få en flowsensor som passede på mundstykket og samtidigt var præcis nok.

En anden mulighed som blev overvejet var en sonar sensor. Der er tidligere blevet arbejdet med en sonar sensor så dette var en oplagt løsning. Problemet opstod ved selve monteringen af sensoren, som skulle placeres ved åbningen af røret på ølbongen. Dermed blev denne løsning fravalgt pga. prioriteringen af brugervenlighed.

Den endelige løsning der blev valgt til registrering af væske samt tidtagning, blev en kombination af to løsninger. Til registrering af tid blev der monteret en magnet på håndtaget, til at registrere, om ventilen på ølbongen var åben. Dette gør det muligt at tage en forholdsvis præcis tid, når brugeren begynder at drykke fra ølbongen. Til registrering af væske i ølbongen blev der monteret to lasere som bruges til at tjekke om ølbongen er fuld og om der stadig er væske i ølbongen når brugeren er færdig med at drikke. En mere præcis beskrivelse af dette design valg kan ses i afsnit 7 'Hardwaredesign'.

For at overholde de tidligere beskrevet krav, blev der anvendt en magnetisk REED kontakt der er kompatibel med 3.3 V¹. Denne sensor er let at arbejde med, da den sender et højt eller lavt signal til Raspberry Pi. En anden løsning, kunne have været en Switch-1 pol (leverswitch) ². Denne switch kunne også let monteres på håndtaget og er kompatibel med en Raspberry Pi.

Laser receiveren som blev anvendt var en Laser receiver sensor modul ³, som opererer indenfor de ønskede spændingsniveauer (3.3 V). Derudover blev lige dette komponent valgt, da det havde en hurtig leveringstid, da det blev leveret af en dansk leverandør. Derudover var prisen også overkommelig for vores projekt.

Laser senderen som blev anvendt var typen KY-008

<https://arduinomodules.info/ky-008-laser-transmitter-module/>. Denne laser sender sender en rød laser stråle, som laser receiveren kan registrere. Der blev valgt dette komponent, eftersom det var let at anskaffe til en overkommelig pris. Derudover kunne den operere på 3.3 V og på omkring 40mA. En anden type laser kunne have været en TIM-201 laser modul⁴, som også kan forsynes fra en Raspberry Pi.

Software

Dette afsnit afdækker hvilke overvejelser som der har været omkring udviklingen af softwaren til BeerBong Controlleren. Der bliver gennemgået hvilke muligheder der har været omkring bl.a. programmeringssprog, Image til Raspberry Pi og udviklingsværktøjer. Først og fremmest skulle det vælges hvilket image som skulle bruges på den RaspberryPi som skulle fungere som controller. Mulighederne stod imellem Raspbian og et image udleveret fra undervisningen på 3. semester. Det image som blev udleveret ved undervisningen var et custom linux image som underviserne havde uarbejdet specifikt til undervisningen. Fordelene ved at bruge dette ville have været at gruppen allerede havde kendskab til dette, men noget negativt ville være at de præcise begrænsninger ikke er kendt for dette image. Fordelene ved at bruge Raspbian ville være at der er bred mulighed for at finde hjælp eller guides på nettet, for at løse problemer som måtte opstå. Raspbian blev valgt som image på RaspberryPi, fordi der var stor hjælp og hente online og det var muligt at finde de præcise begrænsninger ved det.

Det næste var at se på om det skulle udvikles i Linux eller Windows. Som udgangspunkt virkede det som et oplagt valg at udvikle det i Linux, og crosscompile til RaspberryPi. Men efter undersøgelse, blev det opdaget at der kun er visual code, som er en begrænset version af visual studio, til Linux. Derfor faldte valget i stedet på at udvikle i Windows. Til udviklingen af programmet var det oplagt at bruge Visual Studio 2019 som IDE, da det primært er dette udviklingsværktøj som er blevet brugt i vores studie. Vi er meget bekendt med dette værktøj, man kan hente nuget pakker igennem IDE'en,

¹Referer til bilag

²Datablad: Leverswitch170

³Datablad: LaserSensorModule

⁴Datablad: TIM-201 diode

frem for selv at skulle installere pakkerne individuelt. På baggrund af udviklingen i Windows, var det også nødvendigt at finde en ssh terminal til at forbinde til controlleren for at teste programmet og sætte controlleren korrekt op. Til denne process blev PuTTY valgt, som er en opensource terminal emulator, da det var den mest ubredte og anbefalede terminal emulator som blev fundet.

Ift. valg af programmeringssprog var der flere muligheder at vælge imellem. De overvejede sprog var C#, C++, C eller python. C og C++ blev hurtigt valgt fra på baggrund af at der fokuseres på højere niveau af programmeringssprog dette semester. Der blev som gruppe taget et valg om at udvikle i C# på alle platforme, primært fordi det er det som bliver brugt i alle fag dette semester, og dermed lever bedst muligt op til læringskravene. Men da det ikke er muligt at bruge alle biblioteker fra .Net frameworket på en Raspberry Pi, betød det at eksterne biblioteker ville være en nødvendighed. Forbindelse til GPIO portene, kunne ikke oprettes med .Net Framework standard biblioteker, så et eksterne bibliotek Unosquare ⁵ blev brugt. Dette bibliotek gjorde det muligt at oprette forbindelse til GPIO porte, selvom programmet var udviklet på Windows i C#. Da programmet blev udviklet på Windows og i C#, betød det at der var et godt alternativ til crosscompiling. Der findes nemlig et development framework ved navn af Mono, som gør det muligt at køre .exe filer på Raspberry Pi's.

Noget andet som skulle overvejes grundigt, var hvordan at BeerBong Controlleren skulle forbindes med BeerBong Applikationen. Under overvejelsen af dette blev 3 muligheder opsat, at forbinde til WebApi'et, så applikationen kunne hente dataen derfra. At forbinde direkte til Applikationen over WiFi var også en mulighed som blev taget med i betragtning, det ville være hurtigere end at skulle have Controlleren til at poste det til serveren, for at applikationen så skulle hente det, det ville betyde at Controlleren altid skulle være på nettet, og formindske brugervenligheden. Den sidste mulighed var at forbinde med bluetooth. Dette ville gøre det muligt at forbinde direkte til applikationen, uden at Controlleren skulle forbindes til et netværk. Forbindelse over bluetooth blev valgt som den metode der skulle bruges, pga. større brugervenlighed ved kun at skulle forbinde til applikationen, og at netværk ikke skulle sættes op til controlleren. Et stykke inde i udviklingsprocessen, gav bluetooth forbindelsen mange problemer, som tog for meget tid i forhold til hvad der var realistisk for at nå i mål. Derfor blev det en nødvendighed at skifte over til at bruge en Websocket forbindelse over WiFi. Mere specifikt blev der valgt at lave en node.js websocket med PM2. Node.js blev brugt på baggrund af tidspres og brugervenligheden i dette. PM2 er en Production Process Manager, gør brug og kørsel af node.js mere brugervenligt og hurtigt.

3.1.3 Delkonklusion

Efter analyse fasen var overstået var der blevet lagt et godt fundament for, hvordan BeerBong Controlleren skulle udvikles. Alle nødvendige sensorer var blevet fundet efter grundige overvejelser i forhold til alternativer. Derudover var der lavet et godt kendskab til det nye image (Raspian) på Raspberry Pi, som vil skabe fundamentet for udviklingen.

3.2 BeerBong Applikation

3.2.1 Indledning

Dette afsnit dækker overvejelserne til systemets applikation. Her vil overvejelserne og valg af blandt andet applikationen og udviklingværktøjet uddybes.

3.2.2 Analyse

Det primære fokus med spillet var brugervenlighed. Første overvejelse var en hjemmeside, men efter nærmere undersøgelse kom gruppen frem til at dette betød ølbongens controller skulle genere en kode som brugerne så kunne forbinde til, lidt á la Kahoot⁶. Dette var selvfølgelig en mulighed, men

⁵Referer til link

⁶<https://support.kahoot.com/hc/en-us/articles/360000109048-How-to-find-a-game-PIN>

da produktet var noget der skulle bruges i en festlig sammenhæng, hvor brugerne højst sandsynligt var under alkohols effekt, blev en hjemmeside krydset af listen med muligheder. Den generelle konsensus i gruppen var at det ville være nemmere med en mobil applikation, hvor en bruger kunne forbinde til ølbongen, som de ville med en trådløs højttaler⁷. En bruger skulle have mulighed for at kunne forbinde til sin ølbong og hurtigt starte et spil. Siden gruppen selv havde personlig erfaring med brug af applikationer til andre trådløse enheder e.g. højttalere, virkede dette som et åbenlyst valg.

3.2.2.1 Android og Xamarin.Forms

Applikationens platform blev besluttet relativt hurtigt. Det åbenlyse valg var Android, da en IOS applikation skal gennem en godkendelsesprocess, hvor længden af denne process er ukendt, og dette præsenterede sig som en mulig trussel for projektets deadline. Den tredje mulighed var en Windows applikation til smartphones der gør brug af Windows, men da Microsoft selv har meddelt at Windows telefoner ikke vil støttes længere⁸, virkede dette som intet andet end spild af tid.

Efter valget af platform skulle udviklingsværktøjet besluttes. Android applikationer bruger hovedsageligt Java⁹ eller Kotlin¹⁰. Da det hverken ville være studierelevant eller optimalt i forhold til tidsplanen at lære disse, måtte valget være noget tredje. Applikationen blev skrevet i Microsoft Visual Studio ved hjælp af Xamarin.Forms¹¹. Xamarin.Forms gør det muligt at udvikle applikationer til diverse platforme fra en enkelt, delt, implementering. Takket være dette kunne applikationen skrives i C# og XAML, hvilket gjorde det en del mere overskueligt takket være undervisningen i GUI.

3.2.2.2 Integration med resten af systemet

Det næste valg var hvordan appen skulle forbinde til selve ølbongen. Som tidligere skrevet, havde gruppen erfaring med trådløse enheder såsom højttalere og andre audio enheder. Disse enheder brugte som regel Bluetooth. Efter yderlige research viste det sig også at Bluetooth ofte blev brugt til disse former for teknologier hvor hastighed ikke var en absolut nødvendighed. Derudover krævede det også en del mindre i form af telefonens strøm, hvilket igen forbinder til det primære fokus med brugervenlighed. Til denne iteration af produktet måtte Wifi dog benyttes, da der opstod problemer med Bluetooth forbindelsen. Til viderudvikling af produktet er Bluetooth dog stadig en prioritet. For at læse mere om valget af Wifi frem for Bluetooth henvises til afsnit 3.1 'BeerBong Controller'.

3.2.2.3 Databaser til applikationen

Meningen med det lokale spil er at en bruger skal kunne spille med et brugerdefineret antal personer, med andre ord skal ejeren af produktet kunne starte et spil og indtaste det antal spillere der skal deltage. Til dette lokale spil implementeres en lokal database. Xamarin.Forms støtter dette med SQLite¹² hvilket gør det muligt at oprette og gemme et lokalt spils deltagere lokalt på en brugers enhed. Dataen fra et lokalt spil er midlertidig, så det skal ikke gemmes, hvilket også gør det fordelagtigt at bruge en lokal database, da dataen ikke skal overfylde vores remote database. Efter nærmere undersøgelse viser det sig også at de fleste appudviklere gør brug af dette, da det kan sættes op uden brug af nogen server¹³.

⁷E.g.: <https://www.sonos.com/da-dk/easy-to-use>

⁸<https://www.cnet.com/news/microsoft-isnt-making-another-windows-phone-for-one-simple-reason/>

⁹<https://www.zipcodewilmington.com/blog/i-want-to-develop-android-apps-what-languages-should-i-learn>

¹⁰<https://developer.android.com/kotlin>

¹¹For yderligere information om Xamarin.Forms: <https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin-forms>

¹²Kilde: <https://docs.microsoft.com/en-us/xamarin/xamarin-forms/data-cloud/data/databases>

¹³<https://medium.com/@patibrijesh/how-to-use-sqlite-in-a-xamarin-forms-app-2c6ec5894510>

Udover at spille lokalt skal en ejer af produktet også kunne spille online mod en anden spiller der også ejer det. Dette foregår over en remote server og database, som der kan læses mere om i nedenstående afsnit; 3.3 'BeerBong Server' og 3.4 'BeerBong Database'.

3.2.3 Delkonklusion

Xamarin.Forms tilbyder en velkendt måde at udvikle en applikation på. Med baggrundsviden fra semestrets kurser har det været muligt at implementere en applikation der kunne matche de fleste initielle planer. Desværre måtte nogle elementer nedprioriteres, da gruppen mistede to medlemmer under projektførelsen. Dette resulterede i en ukomplet version af det lokale spil, samt brug af Wifi i stedet for Bluetooth, som tidligere nævnt.

3.3 BeerBong Server

3.3.1 Indledning

Nedenstående afsnit omhandler de overvejelser som har været på spil i forhold til hvordan backend for systemet skulle laves. I afsnittet reflekteres der over de forskellige muligheder samt løsninger der blev udarbejdet igennem projektets forløb.

3.3.2 Analyse

Funktionaliteten af BeerBong serveren skulle bestå af at udbyde en datasynkronisering mellem platformen til en mobil applikation i form af en backend server. Udfra de forskellige muligheder for at implementere en bæredygtig og skalerbar backend for mobil applikationen. Blev det valgt at anvende et RESTful api til at definere hvordan data skulle overføres mellem applikationen og databasen. Generelt udfra research omkring backend, viste det sig at der var mange teknologier en API kunne udvikles i. Herunder PHP, Node.js og mange andre. Ud af disse teknologier blev det valgt at arbejde med asp.net core sammen med MVC frameworket for at udvikle BeerBong API'et. Dette var grundet at der var væsentlig større erfaring med Asp.net core, samt de andre teknologier var forholdsvis ukendte. Herudover har Asp.net core platformen med microsoft nogle suveræne deployings værktøjer, hvor det har været muligt med få klik at kunne deploy BeerBong serveren via microsofts cloud services. I forhold til udviklingsværktøjer blev projektet udviklet i visual studio, grundet dens understøttelse af Asp.net core platformen, samt tidligere erfaringen med dette udviklingsværktøj. Dette kunne dog have været blevet udviklet i et hav af andre miljøer.

For at kunne mappe .Net modeller i Asp.net core projektet og SQL databasen er der blevet valgt at anvende objekt-relational mapper. Til at implementere et persisten layer mellem databasen og beerbong serveren, er entityframework core blevet anvendt. Et alternativ til entityframework ville være NHibernate. Selvom der er fordele og ulemper ved begge frameworks, er det blevet valgt at anvende entityframework grundet tidligere erfaring med dette framework.

I forhold til sikkerhed på appen er det blevet valgt at anvende Asp.net cores membership system Identity, som giver login funktionalitet til BeerBong serveren. Da systemet ikke har været større har det været mest medgøreligt at implementere Identity servicen på Serveren. Alternativer var dog at man kunne have brugt tredje parts tjenester til at have en dedikeret API til login funktionalitet på API'et. Dette kunne f.eks. have været OAuth, der således ville have haft ansvaret for authentication og authorization. Dette ville have været et bedre alternativ hvis systemet havde været større med flere API'er og databaser. Funktionaliteten af login systemet på API'et er blevet gjort med tokens. Til at dokumentere API'et er det blevet anvendt swagger til at implementere swagger. Dette værktøj har bidraget til at dokumentere alle BeerBong serverens Endpoints. Herudover er swagger også blevet anvendt til at integrationsteste API'et og se om de forskellige endpoints har fungeret korrekt.

Til at teste API'et er blevet anvendt Xunit og moq frameworkene, grundet det er nyere og velintegreret sammen med .net core platformen. Moq frameworket er blevet anvendt mocks af de forskellige afhængigheder i unit testene. Alternativt kunne Nunit og Nsubstitute frameworkene været blevet anvendt. Men da Xunit frameworket anbefales til test af .net core platformen, samt microsoft anvender Xunit internt, har dette framework haft prioritet.

3.3.3 Delkonklusion

I forhold til valg af backend for mobil applikationen er det blevet valgt at anvende et RESTful API. Dette valg er grundet research som viste at et API ville være den bedste løsning til backend for mobil applikationen samt den overvældende dokumentation der kan findes om API. Asp.net platform, Entityframework core og visualstudio er blevet valgt grundet tidligere erfaringer med disse.

3.4 BeerBong Database

3.4.1 Indledning

Dette afsnit vil afdække hvordan vi har valgt at udvikle databasen til lagring af data for systemet. Der vil blive gennemgået refleksioner og generelle overvejelser over hvordan det er blevet besluttet at lave databasens struktur. Det vil fremstå som en diskussion hvor alle relevante alternativer bliver opvejet mod hinanden for at finde den bedste løsning.

3.4.2 Analyse

Til udvikling af databasen var der forskellige alternativer til udviklingen. Først skulle der besluttes hvorledes der skulle bruges en relationel database eller noSQL. Eftersom vores system ikke er stort og kompliceret var en noSQL database lavet vha. MonogoDB ikke nødvendigt. Derudover kunne en relationel database let og hurtigt laves, da vi blev introduceret til det tidligt på semestret, modsat noSQL og MongoDB, som først kom senere.

Efter at valget faldt på en relationel database skulle der vælges en server til at køre denne. Eftersom databasen altid skulle være funktionel for alle brugere som måtte spille BeerBongBattle, var det vigtigt at anvende en server der er fejlfri og altid funktionel. Det oplagte valg blev en database hosted af Azure. Valget faldt på denne server, eftersom den er gratis for studerende at anvende, og at den fungerer godt med Visual Studie udviklings værktøjet. Der findes mange alternativer til hosting af database, som f.eks. Oracle, men eftersom Microsoft har udviklet Azure, er der meget materiale til hjælp omkring database udvikling vha. EF-core.

Desuden skulle det besluttes hvorledes vi valgte at udvikle databasens entities og queries. Entity Framework (EF core) er Microsofts object-relationel-mapping værktøj (ORM) der giver programmøren nem adgang til at mappe database objekter til C# objekt modeller. Det er derfor nyttigt at anvende EF core frem for Direct SQL Query, da det fjerner en masse overflødig programmering.

Det blev derfor valgt at udvikle entities og queries vha. EF core, som herefter kunne mappes til en Azure database.

3.4.3 Delkonklusion

Efter analysen er færdiggjort for databasen til BeerBongBattle har gruppen fået en afklaring i forhold til valg af komponenter. Det er blevet fastlagt, at der skal anvendes Azure til server host og EF-core som udviklingsmiljø.

3.5 Konklusion

For at opfylde kravene til systemet måtte visse elementer analyseres før det kunne implementeres. At analysere de enkelte dele af systemet hjalp også med at danne et billede af hvad produktet egentlig kunne og hvordan det skulle fungere. Dette tidlige overblik gjorde processen mere overskuelig, og

på trods af visse uforudsete faktorer der resulterede i yderligere afgrænsning, lykkedes det at skabe et produkt der kunne opfylde de fleste af de initielle krav.

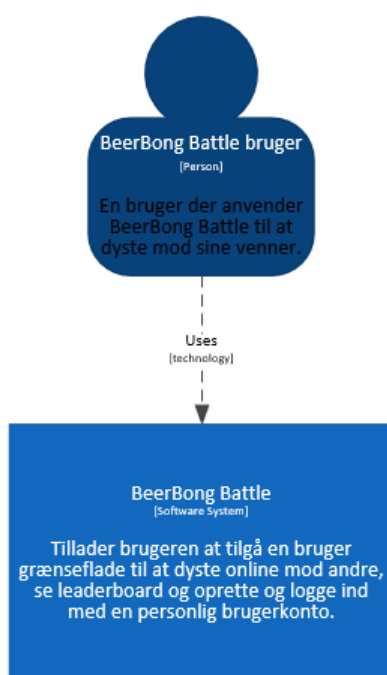
4.1 Indledning

I dette kapitel vil arkitekturen for hele systemet blive gennemgået. Der vil først blive gennemgået arkitektur få det overordnet system. Her vil de fire forskellige moduler blive vist. Herfra vil de forskellige moduler blive gennemgået og beskrevet i detaljer i henhold til C4 modellen. Alle moduler samt den overordnede arkitektur blive beskrevet vha. bla. containerdiagrammer, klassediagrammer og sekvensdiagrammer.

4.2 Overordnet arkitektur

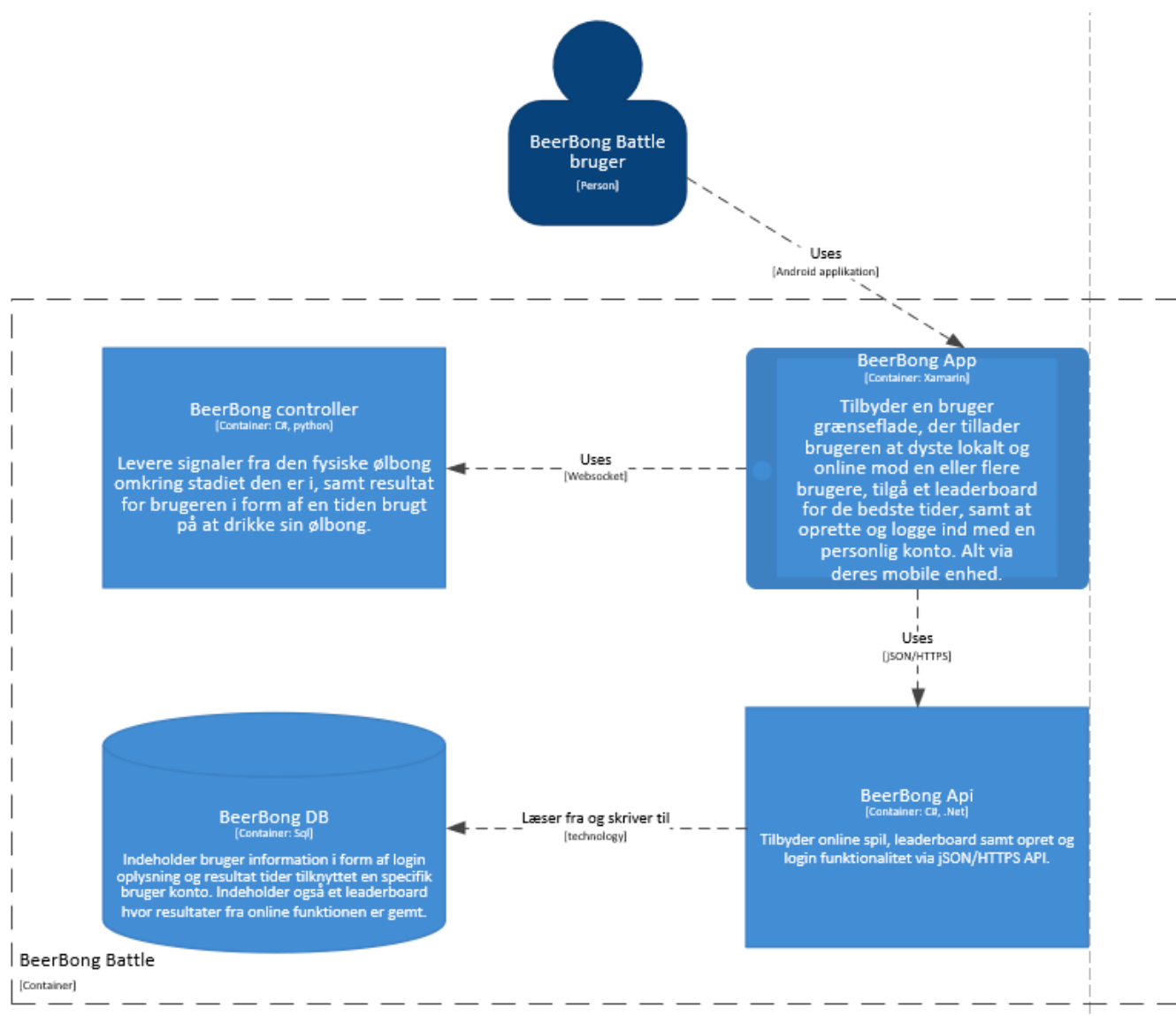
Dette afsnit dækker over den overordnede arkitektur for BeerBongBattle. Dette er medtaget for at give et bedre overblik over systemet som en helhed. På figur 4.1 ses et overordnet C4 system kontekst diagram af BeerBongBattle. På figur 4.2 ses hvordan det er blevet valgt at inddele systemet. Der er blevet lavet 4 forskellige moduler: BeerBong Applikation, BeerBong Controller, BeerBong Server og BeerBong Database. På figur 4.4, 4.5 og 4.6 ses de overordnede diagrammer for de usecases som systemet dækker. Disse skevensdiagrammer beskriver systemets opførsel ved hvert use case.

System Context



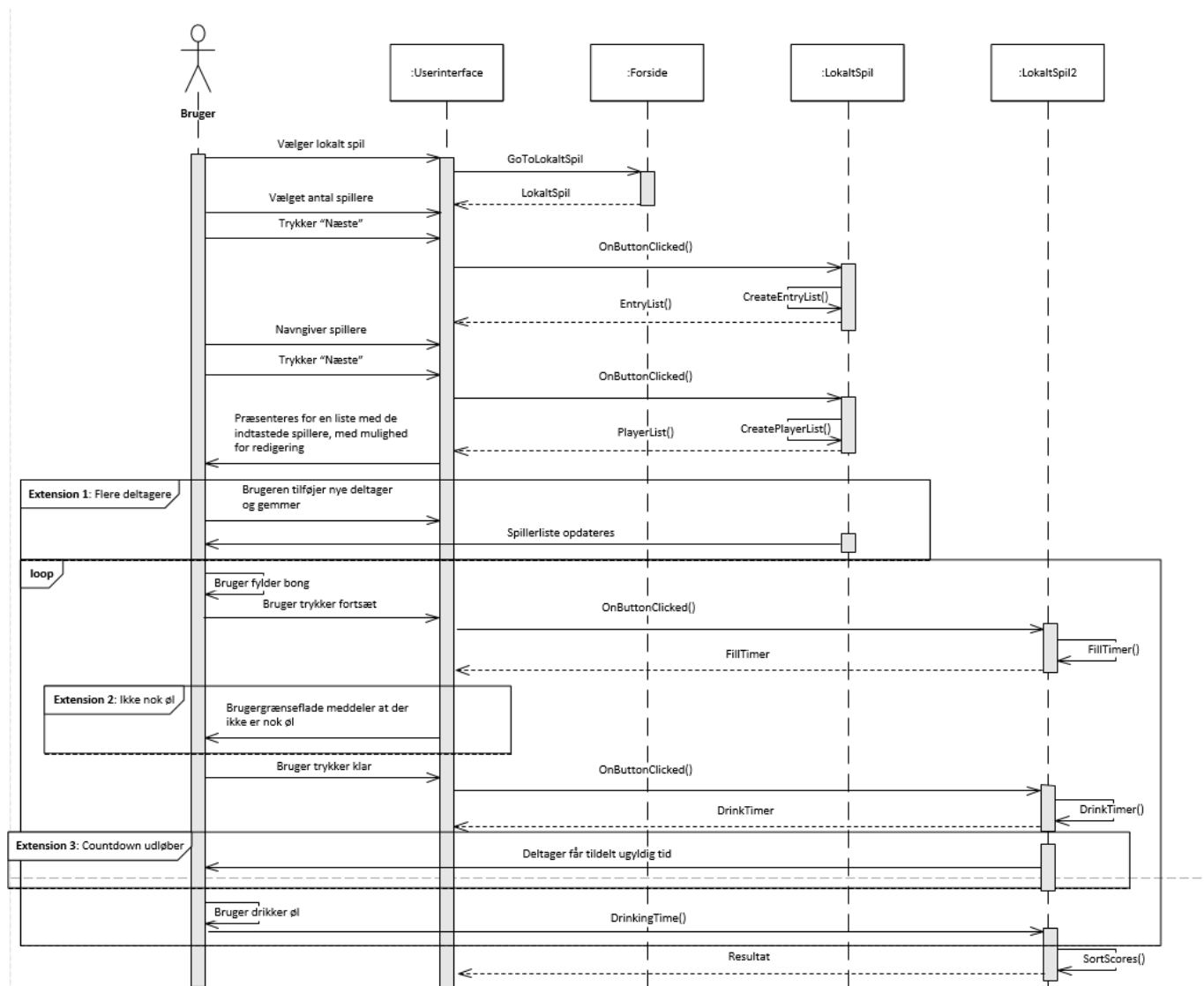
Figur 4.1: Overordnet C4 model

Container diagram



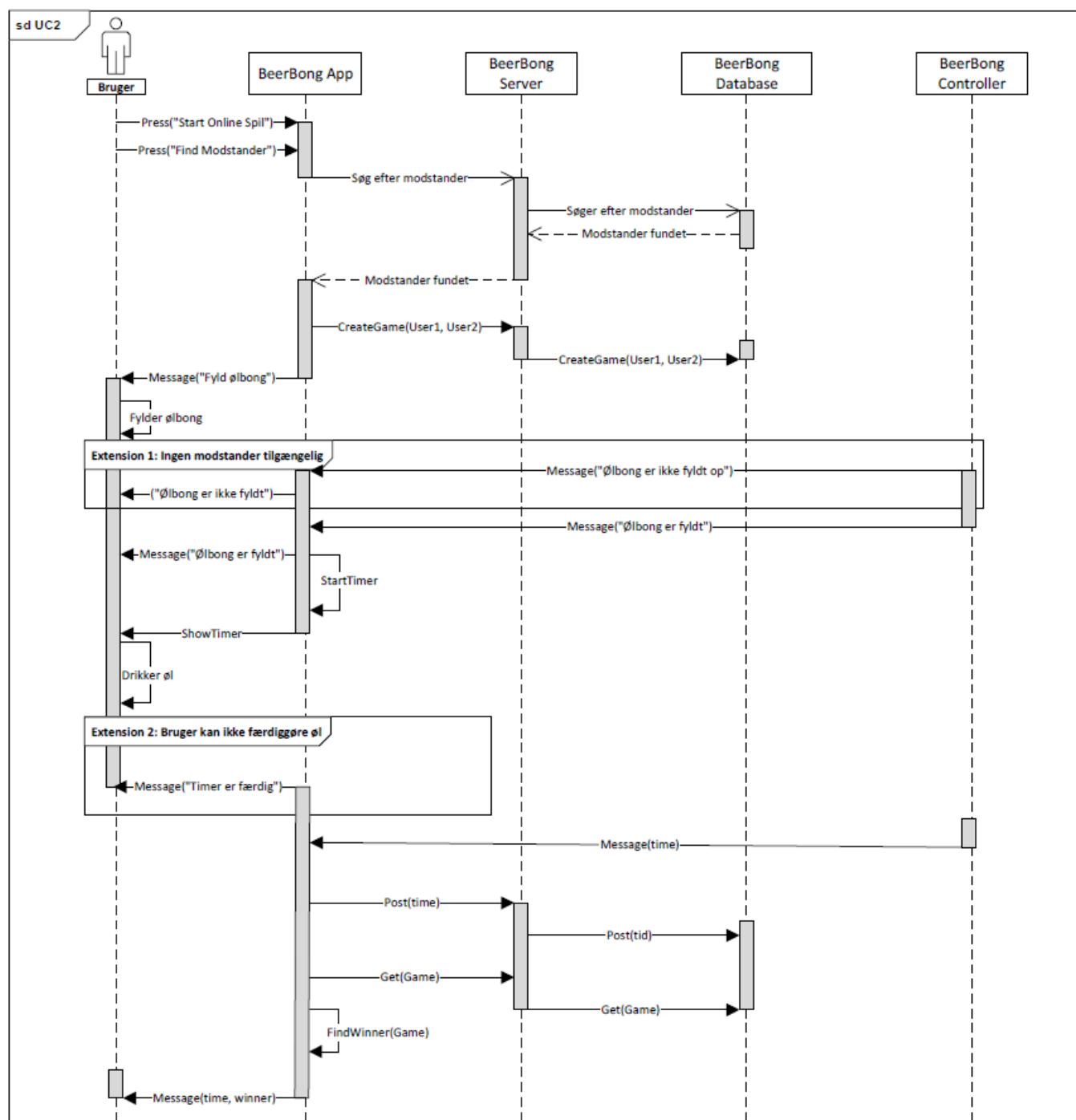
Figur 4.2: Diagram over alle containere i systemet

Overordnet sekvensdiagram Use Case 1



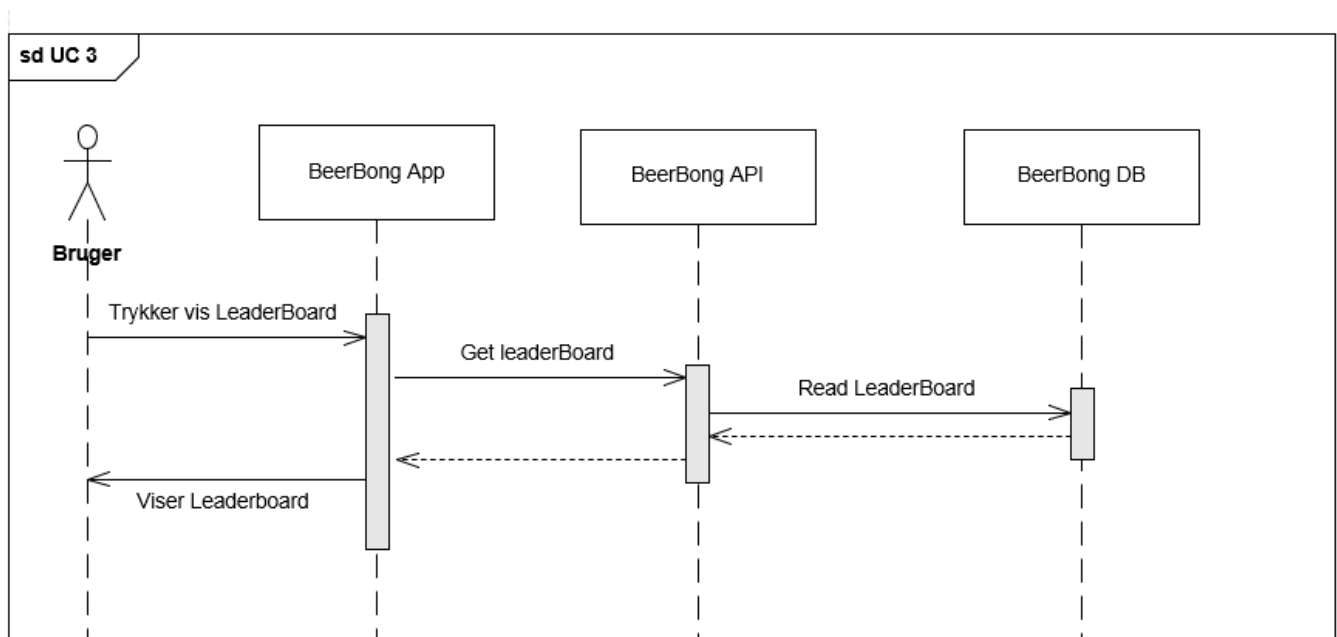
Figur 4.3: Overordnet sekvens diagram af use case 1

Overordnet sekvensdiagram Usecase 2



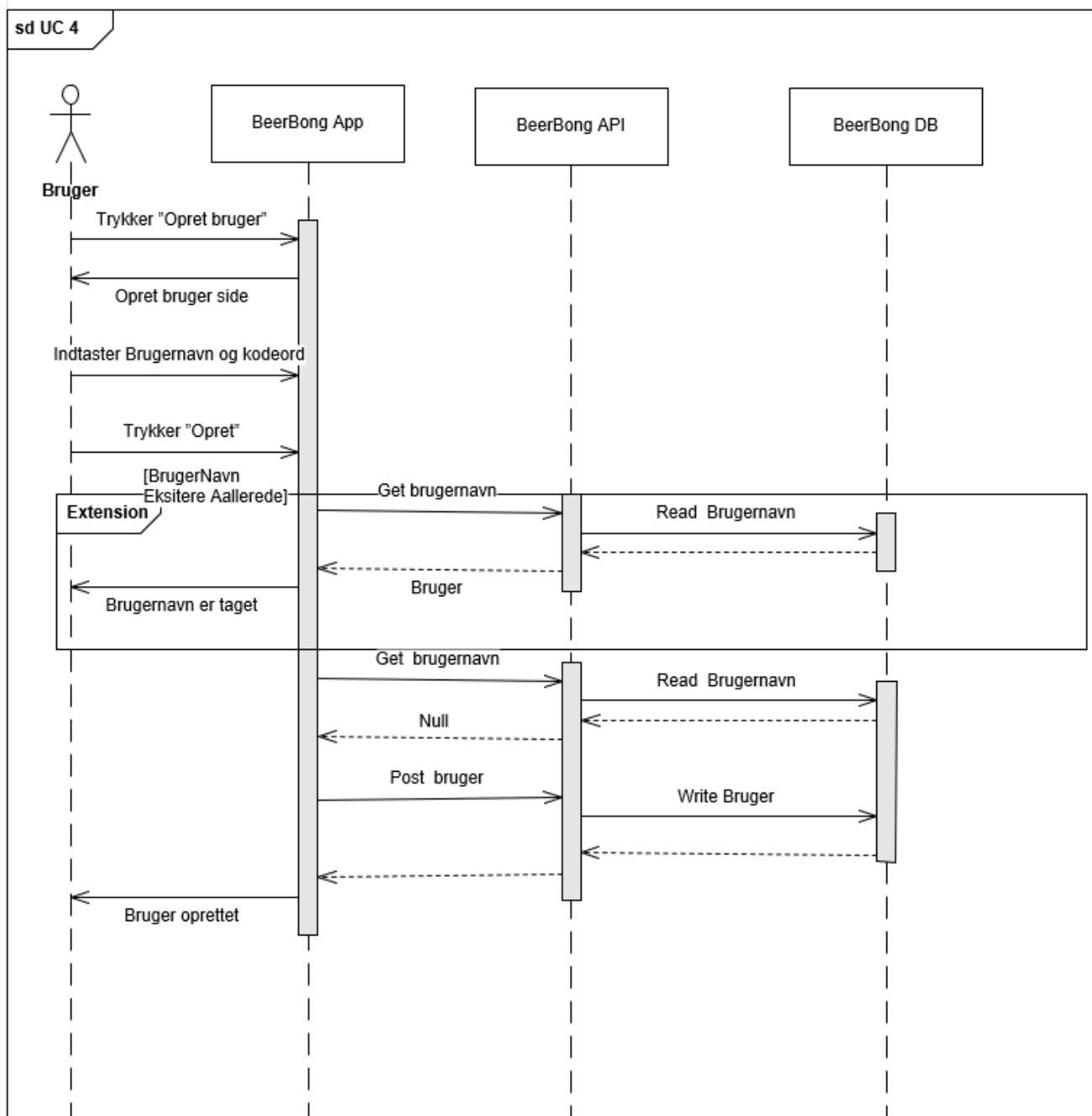
Figur 4.4: Overordnet sekvens diagram af usecase 2

Overordnet sekvensdiagram Usecase 3



Figur 4.5: Overordnet sekvens diagram af usecase 3

Overordnet sekvensdiagram Usecase 4

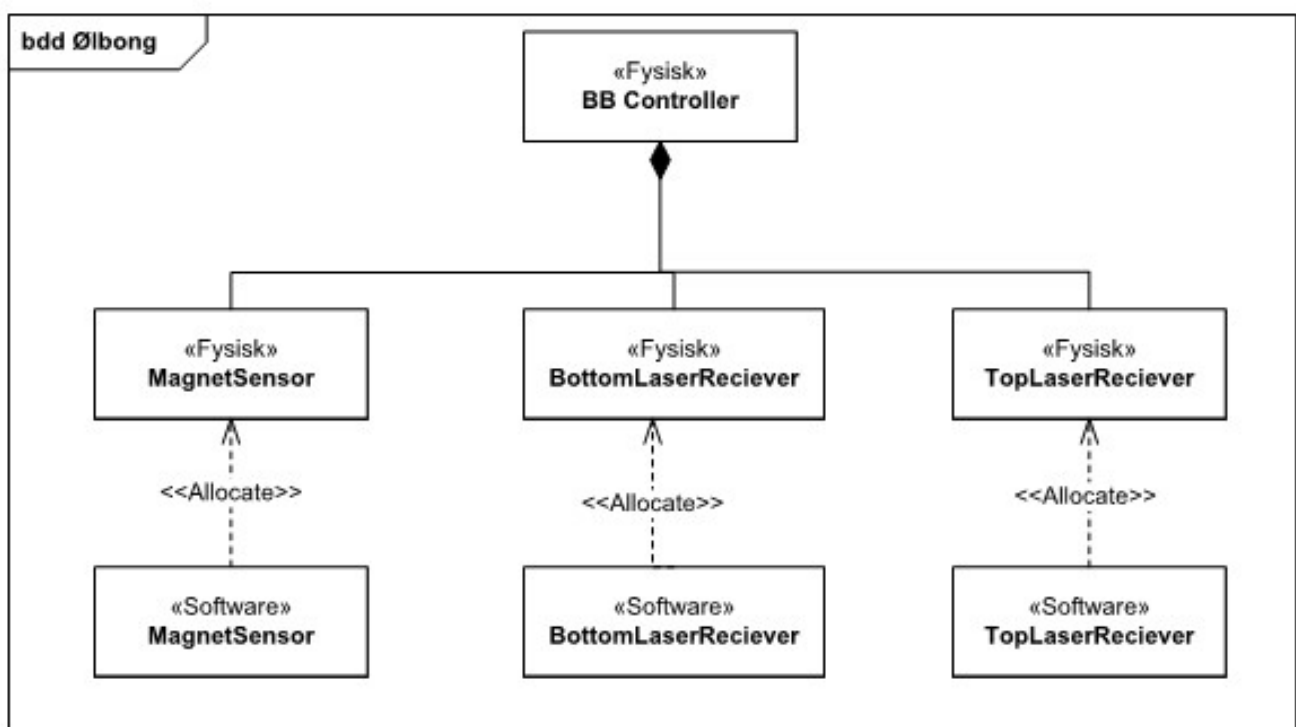


Figur 4.6: Overordnet sekvens diagram af usecase 4

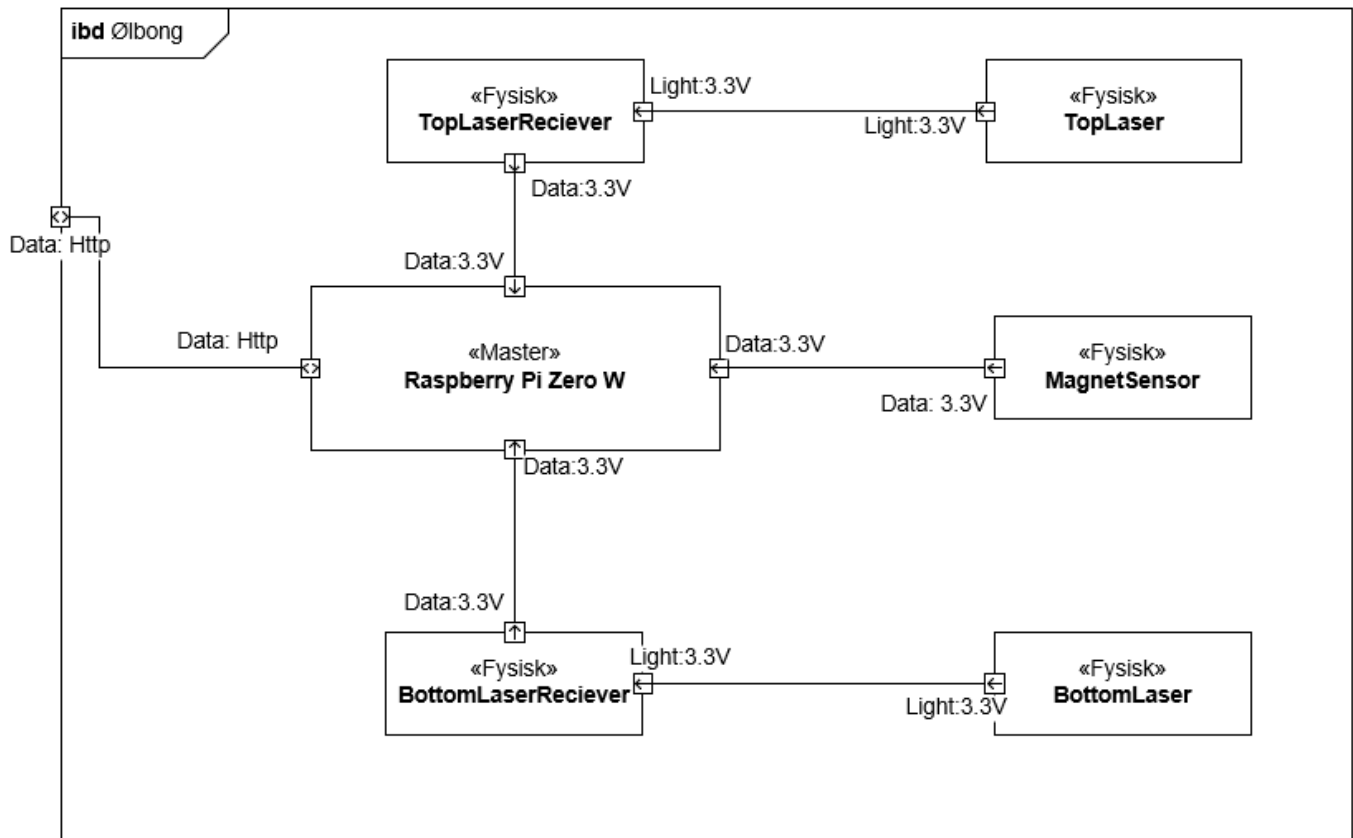
Hardwarearkitektur 5

BeerBong Controller

Denne sektion omhandler hardware arkitekturen for BeerBongBattle. Det eneste hardwaremæssige modul som er i systemet, er BeerBong Controller. På figur 5.1 ses BDD'et for BeerBong Controller. Her kan det ses at Controlleren indeholder 3 sensorer. MagnetSensor som er for at detektere om håndtaget er oppe eller nede. De to laser sensorer som skal detektere om der er øl i Ølbongen eller ej, og om der er nok øl i.



Figur 5.1: BDD over Ølbong



Figur 5.2: IDB over ølbong

Det kan ses på figur 5.1 og figur 5.2 at der ikke er allokeret noget hukommelse til henholdsvis top og bottom laser. Det skyldes, at de to lasere bare skal bruge et 3.3 V fra Raspberry Pi til at lyse.

5.1 Signalbeskrivelse

Bloknavn	Funktion	Portnavn	Type	Port specifikationer
Raspberry Pi Zero W	Input signal fra TopLaserReciever	Data	Analog	3.3 V trigger, input Lav i 1.5 V og Høj i 2.3 V max 3.3 V, max 100 mA
	Input signal fra BottomLaserReciever	Data	Analog	3.3 V trigger, input Lav i 1.5 V og Høj i 2.3 V max 3.3 V, max 100 mA
	Input signal fra MagnetSensor	Data	Analog	3.3 V trigger, input Lav i 1.5 V og Høj i 2.3 V max 3.3 V, max 100 mA
	Input/Output signal fra og til Websocket	Data	Digital	Http forbindelse mellem Raspberry Pi og websocket som er Full-duplex
TopLaser Reciever	Input signal fra TopLaser	Light	Analog	Ukendt følsomhed (check lux)
	Output signal til Raspberry Pi Zero W	Data	Analog	3.3 V output signal.
BottomLaser Reciever	Input signal fra BottomLaser	Light	Analog	Ukendt følsomhed (check lux)
	Output signal til Raspberry Pi Zero W	Data	Analog	3.3 V output signal.
Magnet Sensor	Output signal til Raspberry Pi Zero W	Data	Analog	3.3 V output signal.
BottomLaser	Output signal til BottomLaserReciever	Light	Analog	Forsynes med 3.3 V og sender kraftigt lys (rød)
TopLaser	Output signal til TopLaser	Light	Analog	Forsynes med 3.3 V og sender kraftigt lys (rød)

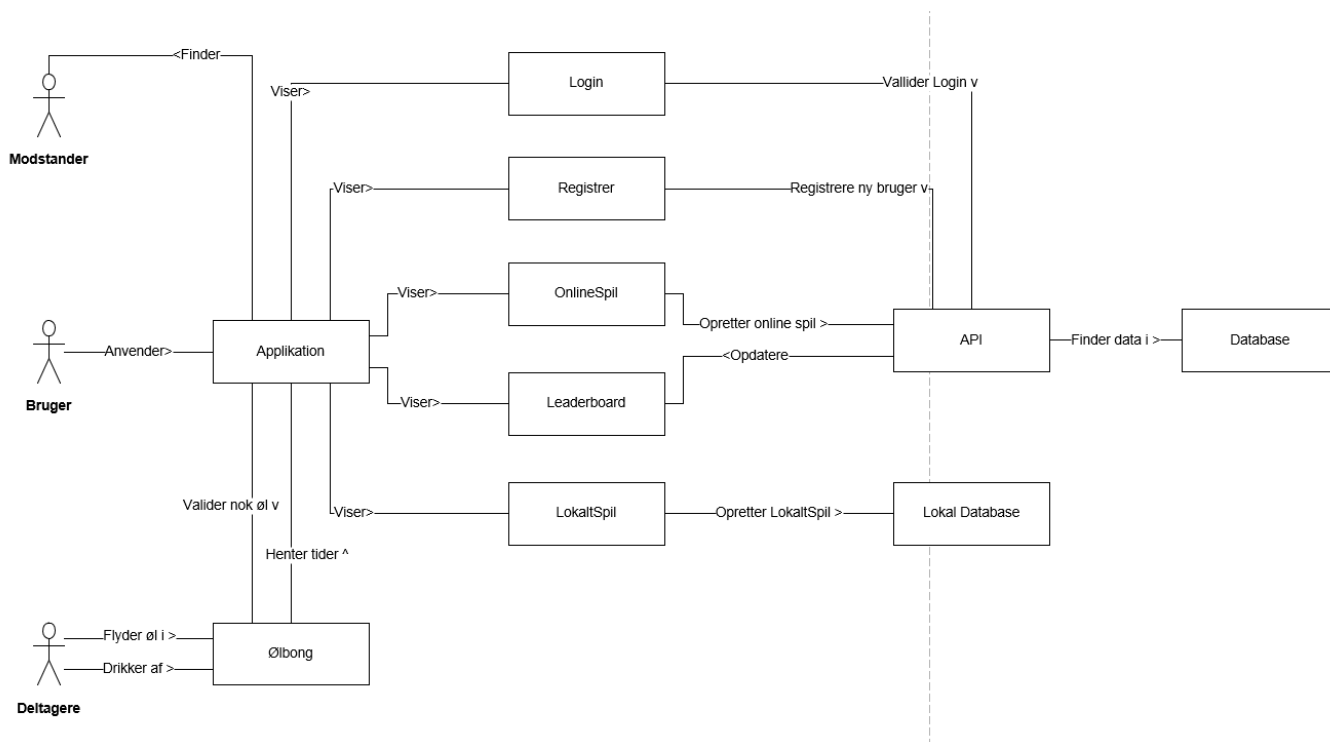
Softwarearkitektur 6

6.1 Indledning

Dette kapitel gennemgår softwarearkitekturen for hele systemet. Den overordnede arkitektur kan findes i afsnit 4.2 'Overordnet arkitektur'. Først gennemgås overordnede sekvensdiagrammer for alle use cases. Derefter gennemgås alle de individuelle moduler først med et component diagram, hvorefter der dykkes længere ned i arkitekturen med et klassediagram og til sidst et sekvensdiagram for at opsummere systemets opførsel.

6.1.1 Domænemodel

Nedenstående figur 6.1 viser domænemodellen for BeerBongBattle systemet. Herudover illustrere domænemodellen systemet opbygning, samt dets anvendelse. Domæne modellen afspejler domæner der fremgår i systemet, som findes i de opstillet usecases.



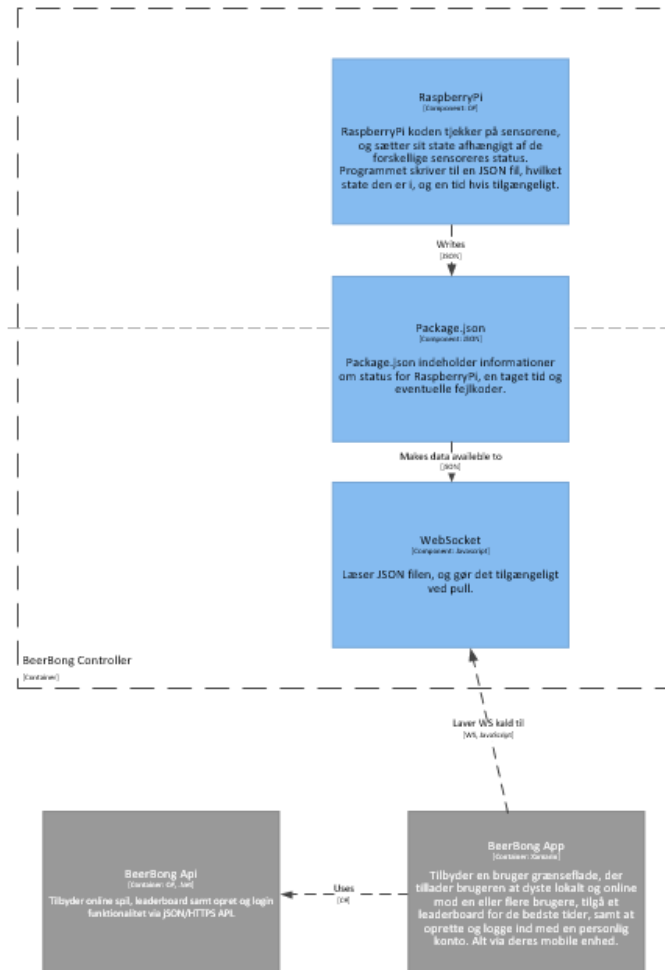
Figur 6.1: Domænemodel for BeerBongBattle systemet

6.2 BeerBong Controller

6.2.1 Component Diagram

På figur 6.2 ses et komponentdiagram for BeerBong Controller. Containeren indeholder 3 komponenter RaspberyPi som er softwaren der kører på Raspberry Pi Zero Wireless, som tjekker på

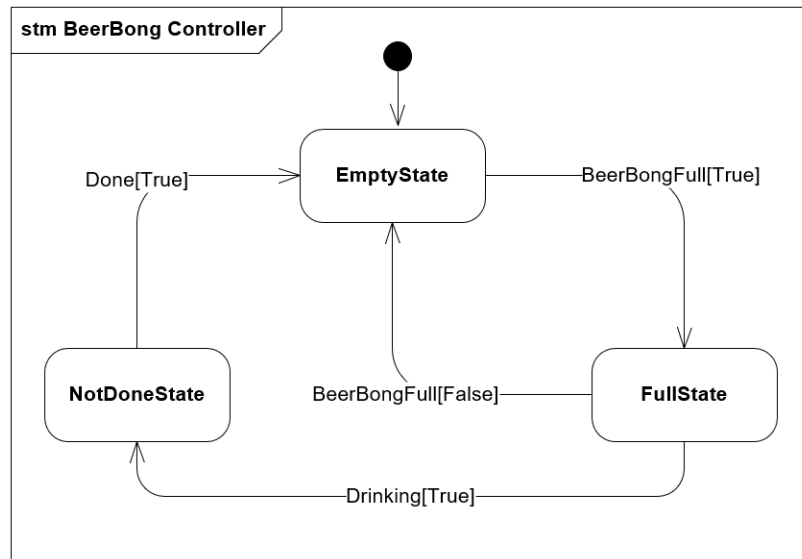
sensorene, og skriver til den anden komponent nemlig package.json. Dette er en json fil der indeholder hvilket state som RaspberryPi programmet er i. Derudover indeholder den også en tid, hvis ølbongen er taget, og som det sidste eventuelle fejlkode. Den sidste komponent er WebSocket, som kigger på package.json. Det er muligt for andre websockets at kalde pull og hente denne json fil, for at se status på beerbong controlleren.



Figur 6.2: Component diagram for BeerBong Controller

6.2.2 State Machine Diagram

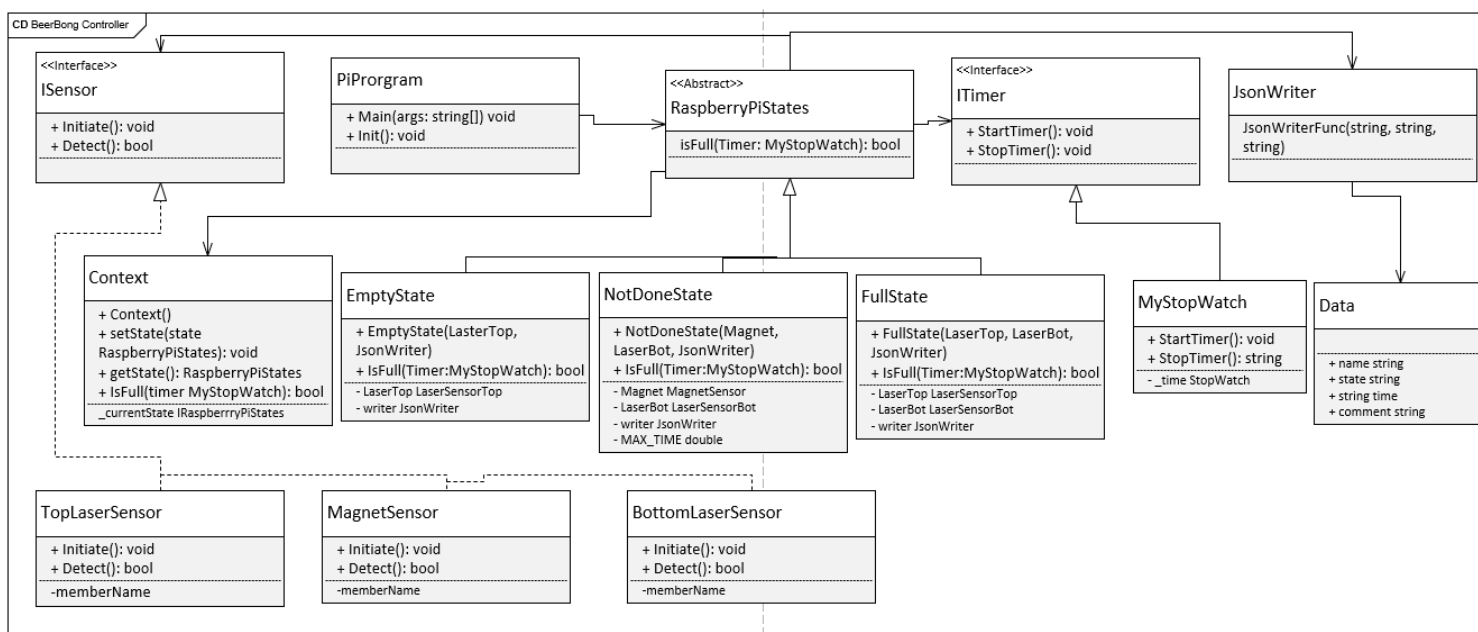
Figur 6.3 viser et state machine diagram over softwaren for BeerBong Controlleren. Diagrammet giver en god beskrivelse af hvordan controlleren fungerer og hvordan den går fra state til state.



Figur 6.3: State Machine Diagram over BeerBong Controller

6.2.3 Klassediagram

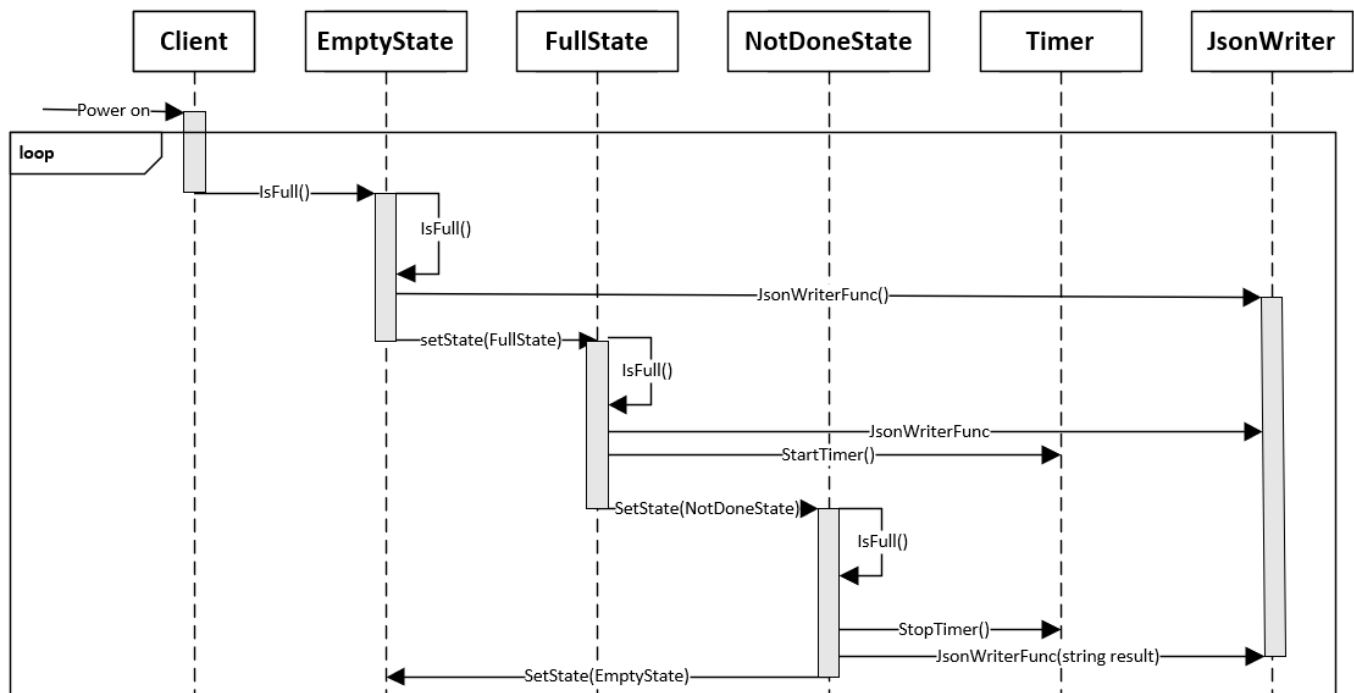
På figur 6.4 ses klassediagrammet for BeerBongControlleren, mere specifikt containeren RaspberryPi. Programmet er overordnet opbygget af 4 dele. Der er en sensor del, som består af et Interface og 3 implementerede klasser, som forbinder til sensorene og tjekker deres status. Det næste er state delen som hele programmet er opbygget omkring. Afhængigt af hvilken af de 3 states som programmet er i, tjekkes der på nogle bestemte sensorere. De sidste to dele er Timer og JsonWriter.



Figur 6.4: Klassediagram for software på BeerBong Controller

6.2.4 Sekvensdiagram

Sekvensdiagrammet for BeerBong Controller ses på figur 6.5. Dette sekvensdiagram er gennemgående for alle systemets use cases. Her gives et overblik over systemets funktionalitet. Ved ethvert scenarie starter programmet i **EmptyState**. Når bongen er fyldt, skrives **EmptyState** til json filen og programmet går til **FullState**. Når brugeren begynder at drikke skrives **FullState** til json filen, timeren startes og programmet går til **NotDoneState**. Når brugeren så er færdig med at drikke stoppes timeren, og tid samt **NotDoneState** sendes til json filen, og programmet går til **EmptyState**.

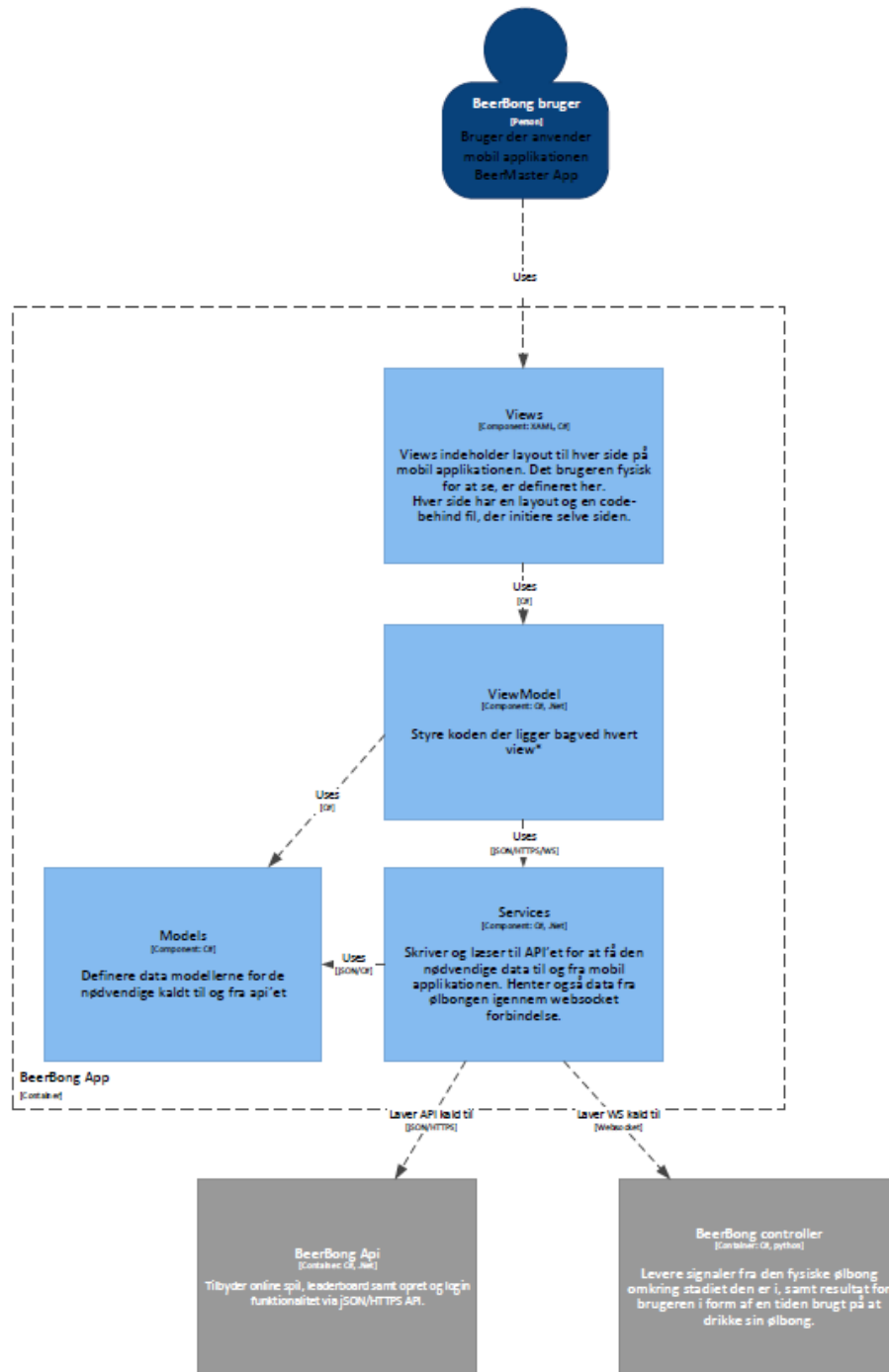


Figur 6.5: Sekvensdiagram for BeerBong Controller

6.3 BeerBong Application

6.3.1 Component Diagram

Component diagrammet på figur 6.2 er zoomet ind på den individuelle container BeerBong App som ses på figur 4.2. Component diagrammet viser de overordnede komponenter som BeerMaster app'en indeholder. Hvert komponent er en gruppe kode, hvor alle komponenterne til sammen udgøre alt source koden til applikationen. Views indeholder de filer som udgør layoutet på applikationen, og agere som brugerens grænseflade til systemet. ViewModel komponentet indeholder koden som delvist er med til at styre funktionaliteten for de views brugeren bliver præsenteret for på applikationen. ViewModel anvender service komponentet som læser og skriver data til og fra BeerBong api'et og BeerBong controlleren. Models komponentet indeholder de data modeller som skrives til enten fra viewmodellerne eller service komponentet. En dybere forklaring for hvert komponent og dets funktionalitet vil blive gennemgået næste afsnit.



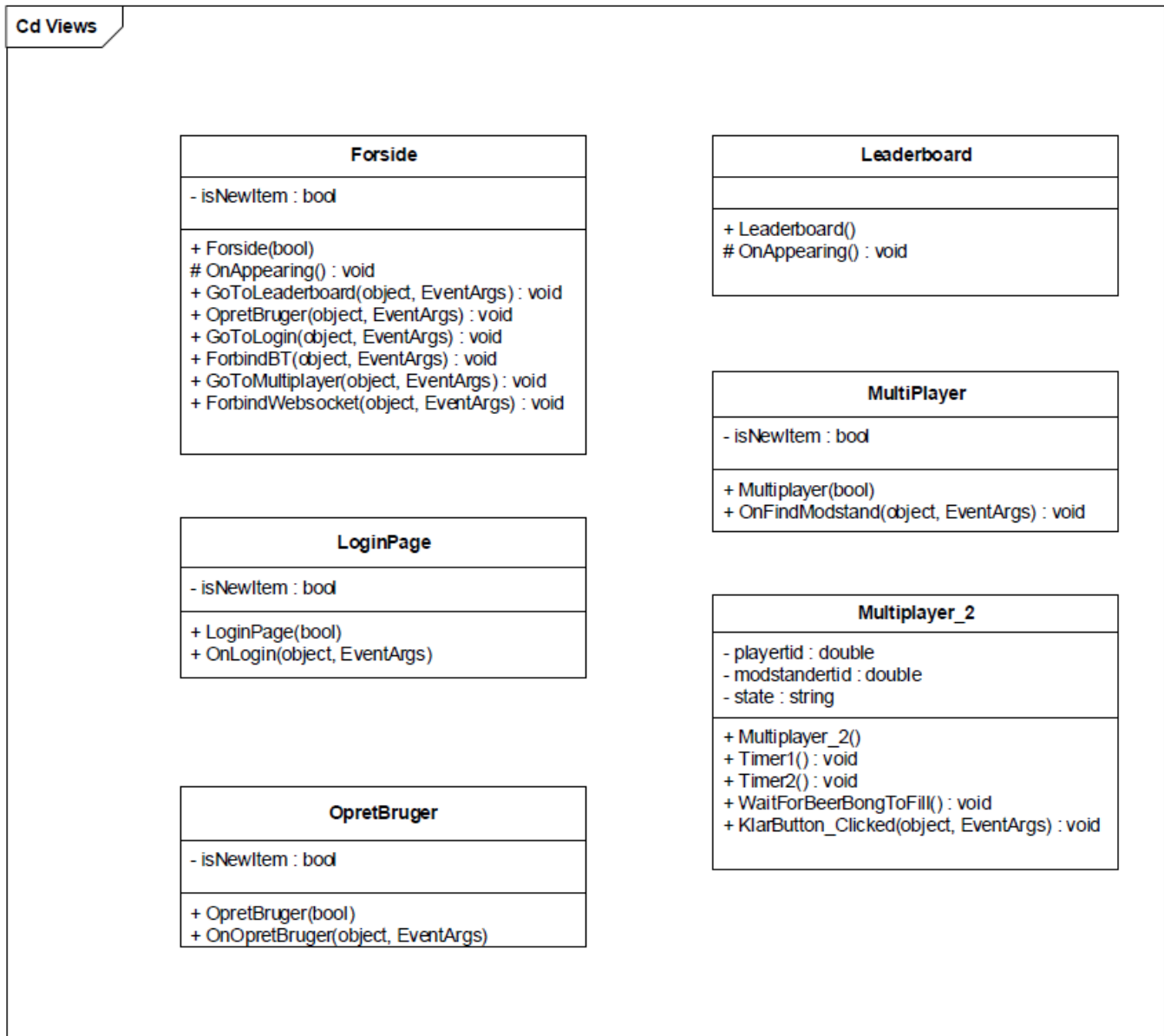
Figur 6.6: Component diagram for BeerBong Application

6.3.2 Klassediagram

Hvert komponent har et tilhørende klassediagram som ses i dette afsnit. Klasse diagrammerne vil indeholde alle klasser og deres tilhørende metoder for hvert komponent.

Klassediagram for views

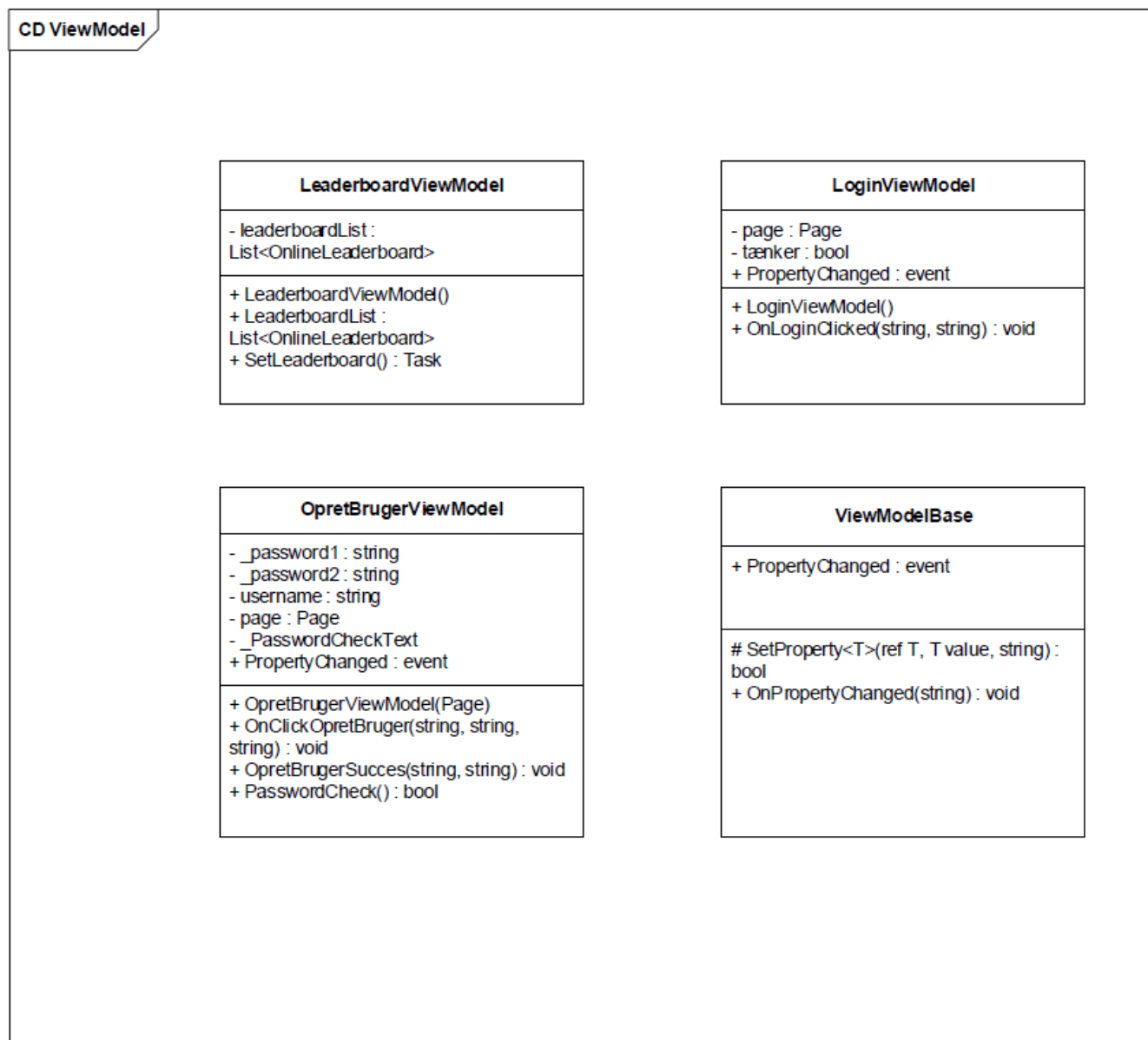
Klassediagrammet på figur 6.7 viser indholdet af komponentet views. Hver klasse har en tilhørende XAML fil som indeholder det layout brugeren bliver præsenteret for. Klasserne vist på figuren 6.7 er koden bagved layout filerne, da hvert view indeholder en XAML fil og en cs fil. XAML filen er layout, og cs filen er koden der initierer siden og evt. styre logikken for layoutet, eks. hvis der trykkes på en knap.



Figur 6.7: Klassediagram over komponentet Views

Klassediagram for ViewModel

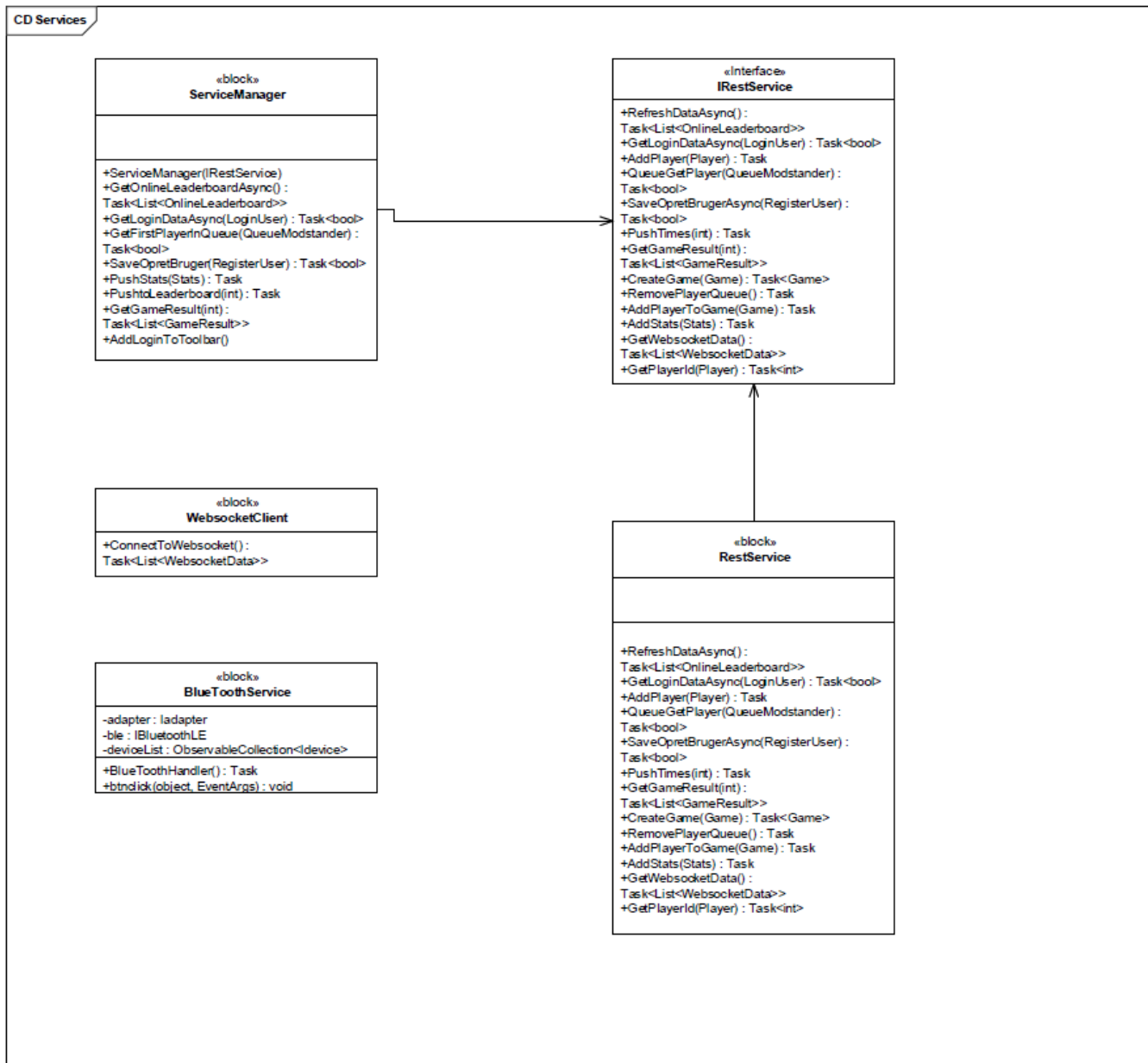
Viewmodel komponentet indeholder fire klasser, hvor tre af dem styre funktionaliteten bag deres tilhørende view. LeaderboardViewModel styre leaderboard viewet, og på den måde overholdes MVVM arkitekturen. Det samme er gældende for LoginViewModel og OpretBrugerViewModel. ViewModelBase er en klasse, som de tre andre klasser nedarver fra. Klassen sørger for at alle klasser implementere OnPropertyChanged. Dette kan også gøres igennem INotifyPropertyChanged interfacet, som det er gjort i nogle tilfælde hvis man kigger på sourcekoden REF.



Figur 6.8: Klassediagram over komponentet ViewModel

Klassediagram for Services

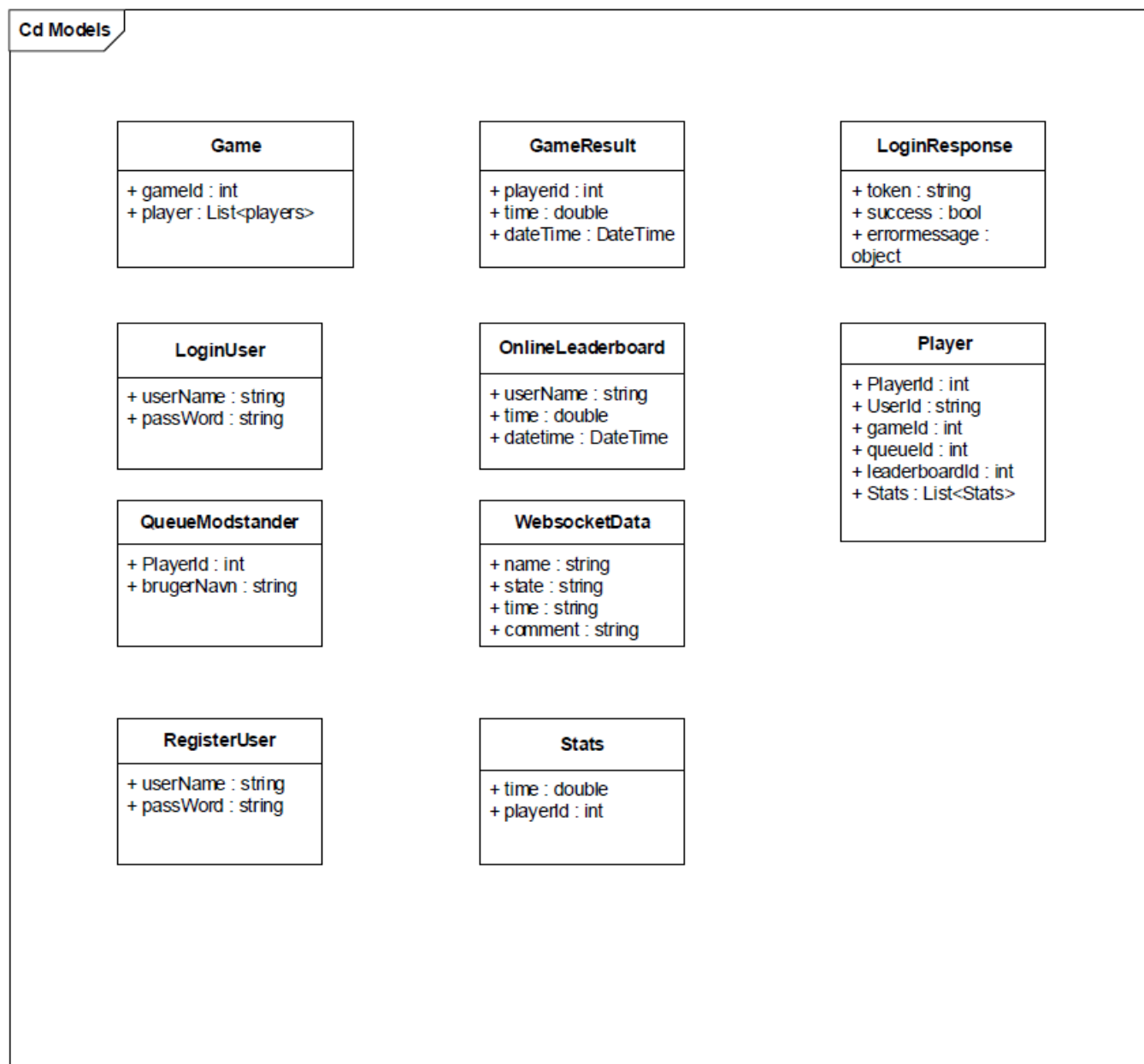
Service komponentet indeholder klasserne vist på figur 6.9. ServiceManager klassen er den som viewmodels opretter et objekt af, og igennem den kalder funktionerne i interfacet `IRestService`. `RestService` klassen implementere metoderne defineret i `IRestService` interfacet. `WebSocketClient` klassens ansvar er at oprette forbindelse til BeerBong controlleren via. websocket kommunikation.



Figur 6.9: Klassediagram over komponentet Services

Klassediagram for Models

Models komponentet indeholder klasserne/modellerne vist på figur 6.10. Hver model bliver brugt i form af kald til og fra BeerBong api'et, og definere kun hvordan model skal se ud, men implementere ingen logik. Logikken bliver håndteret i api'et eller i viewmodellen.

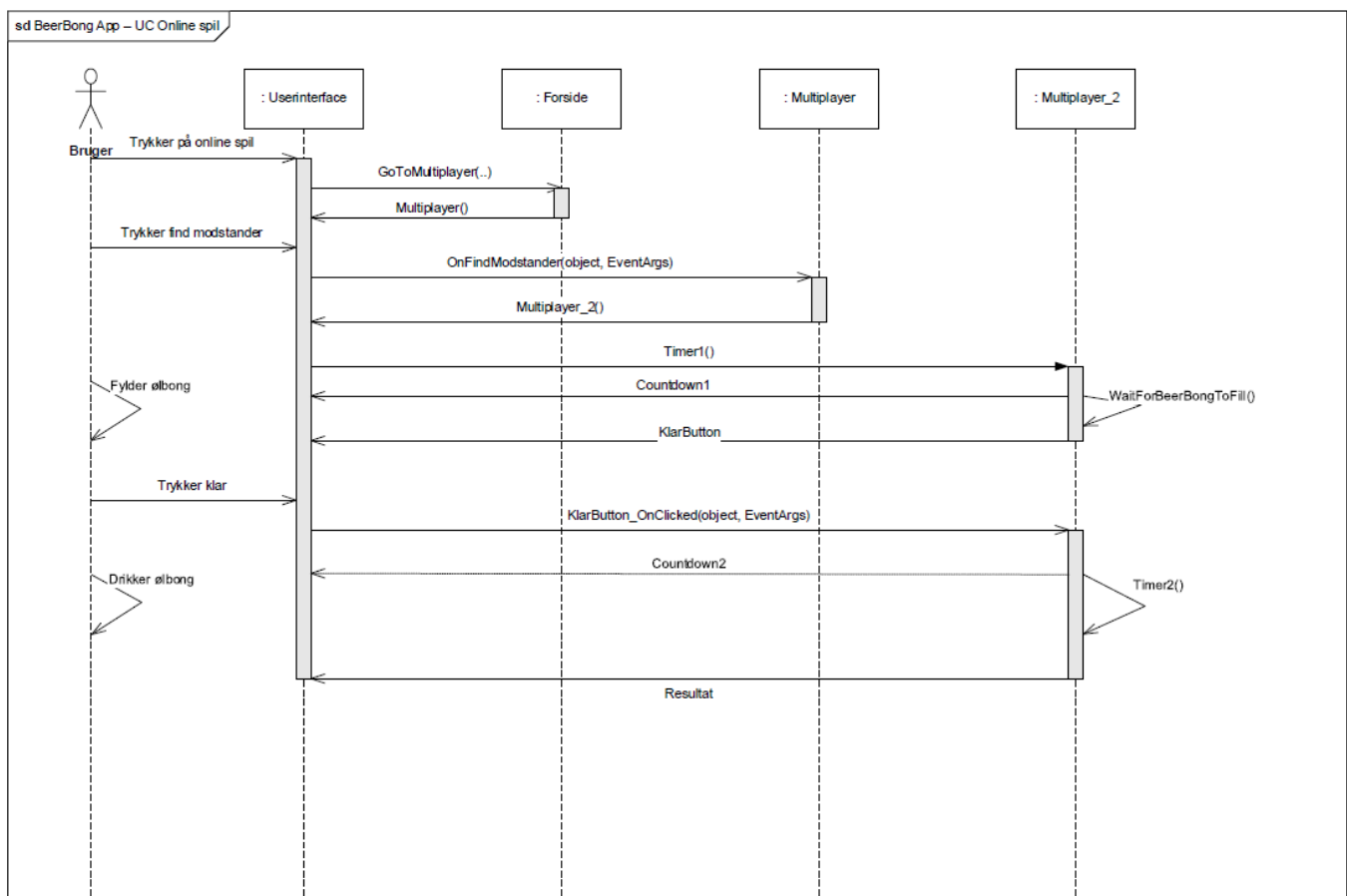


Figur 6.10: Klassesdiagram over komponentet Model

6.3.3 Sekvensdiagram

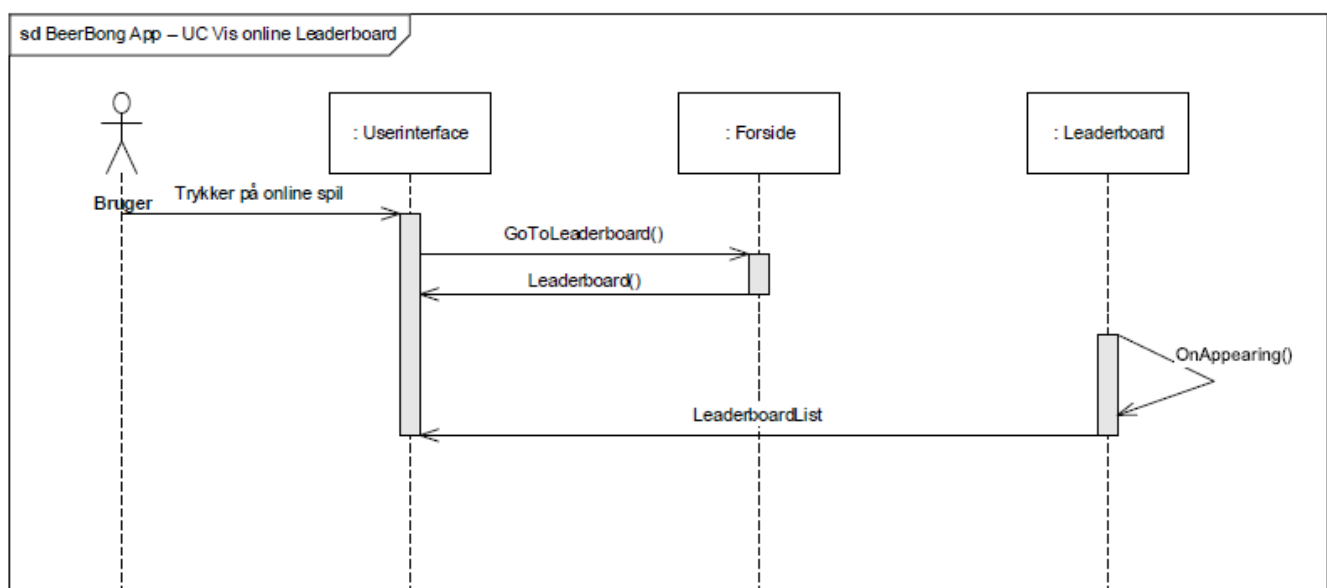
Sekvensdiagrammerne for BeerBong app'en er udarbejdet ud fra systemets usecases. Sekvensdiagrammerne vil vise den overordnet kommunikation imellem klasserne, dog uden at gå i dybden, da det ville gøre diagrammerne uoverskuelige. Der henvises istedet til REF BILAG SOURCE CODE, hvis man ønsker en større forståelse for hvordan softwaren fungerer intern i klasserne og imellem dem.

Sekvensdiagram for UC online spil



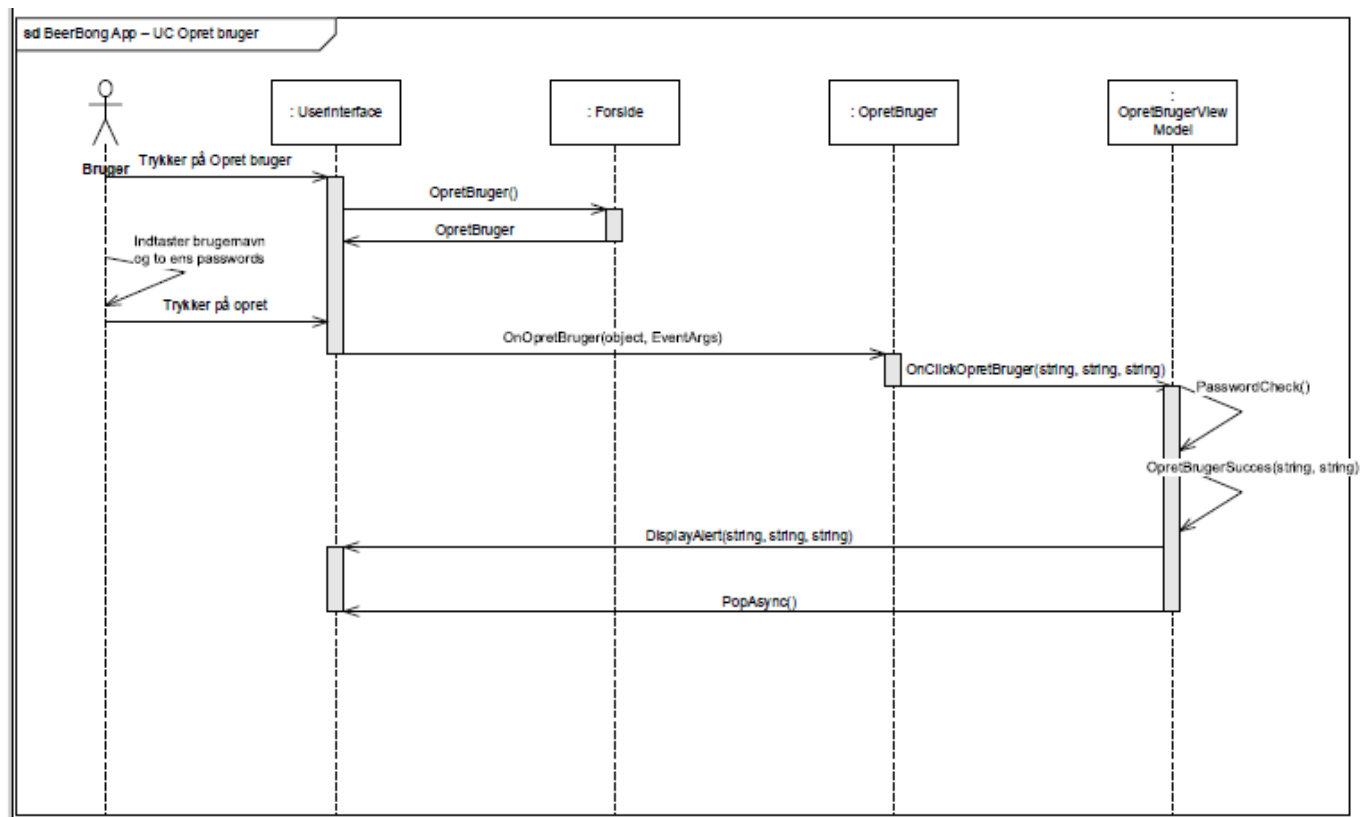
Figur 6.11: Sekevensdiagram for UC online spil for containeren BeerBong App

Sekvensdiagram for UC Leaderboard



Figur 6.12: Sekevensdiagram for UC leaderboard for containeren BeerBong App

Sekvensdiagram for UC opret bruger



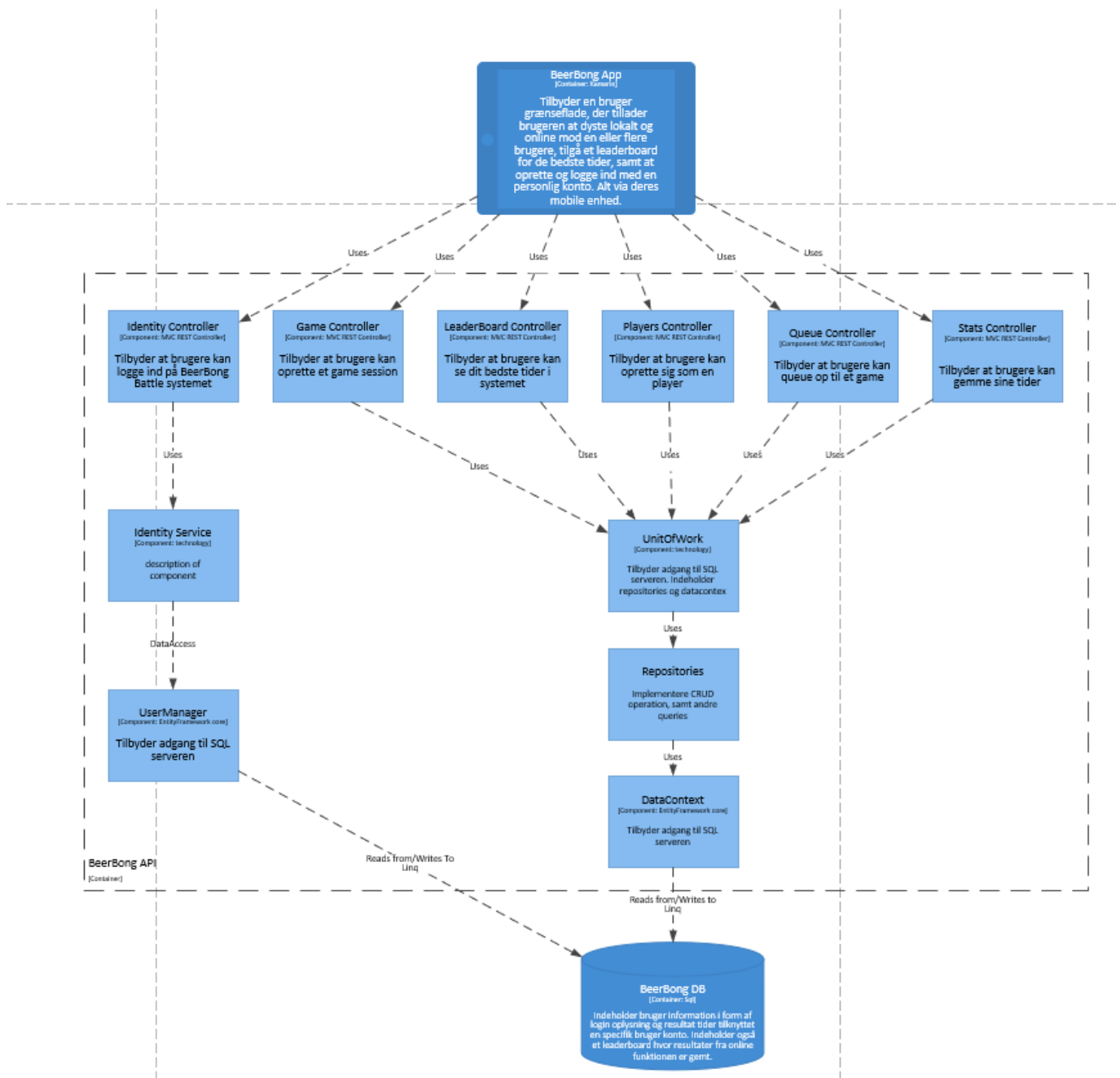
Figur 6.13: Sekevensdiagram for UC opret bruger for containeren BeerBong App

6.4 BeerBong Server

Nedenstående afsnit gennemgår software arkitekturen for BeerBong serveren.

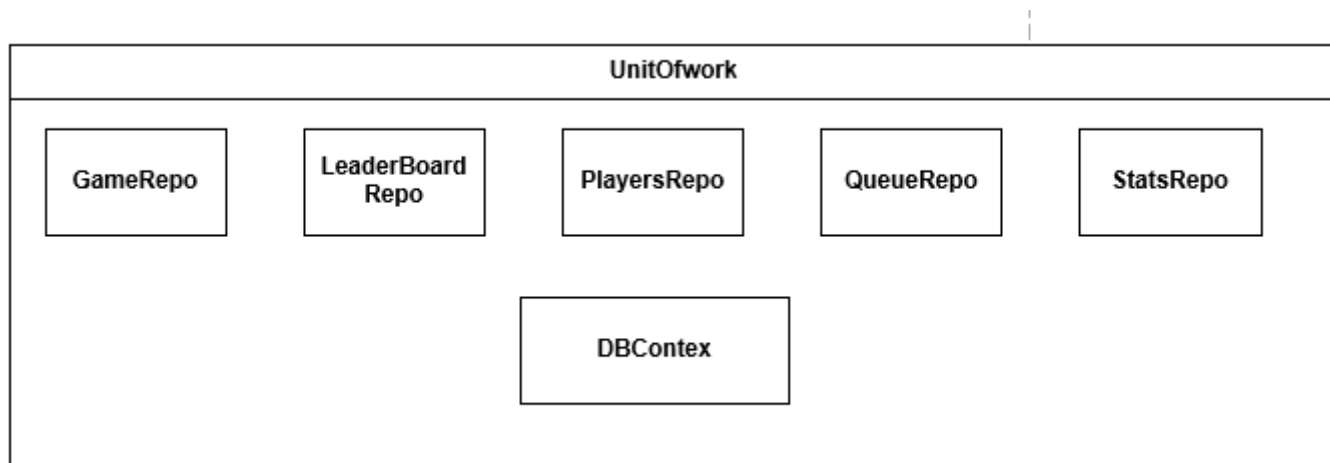
6.4.1 Component Diagram

Nedenstående figur 6.14 fremgår komponent diagrammet for BeerBong API'et. Af komponent diagrammet vises de største byggesten som containeren BeerBong API består af. Yderligere beskrives komponenternes interaktion med hinanden, samt deres ansvar og implementationsdetaljer.



Figur 6.14: Component diagram af BeerBong API

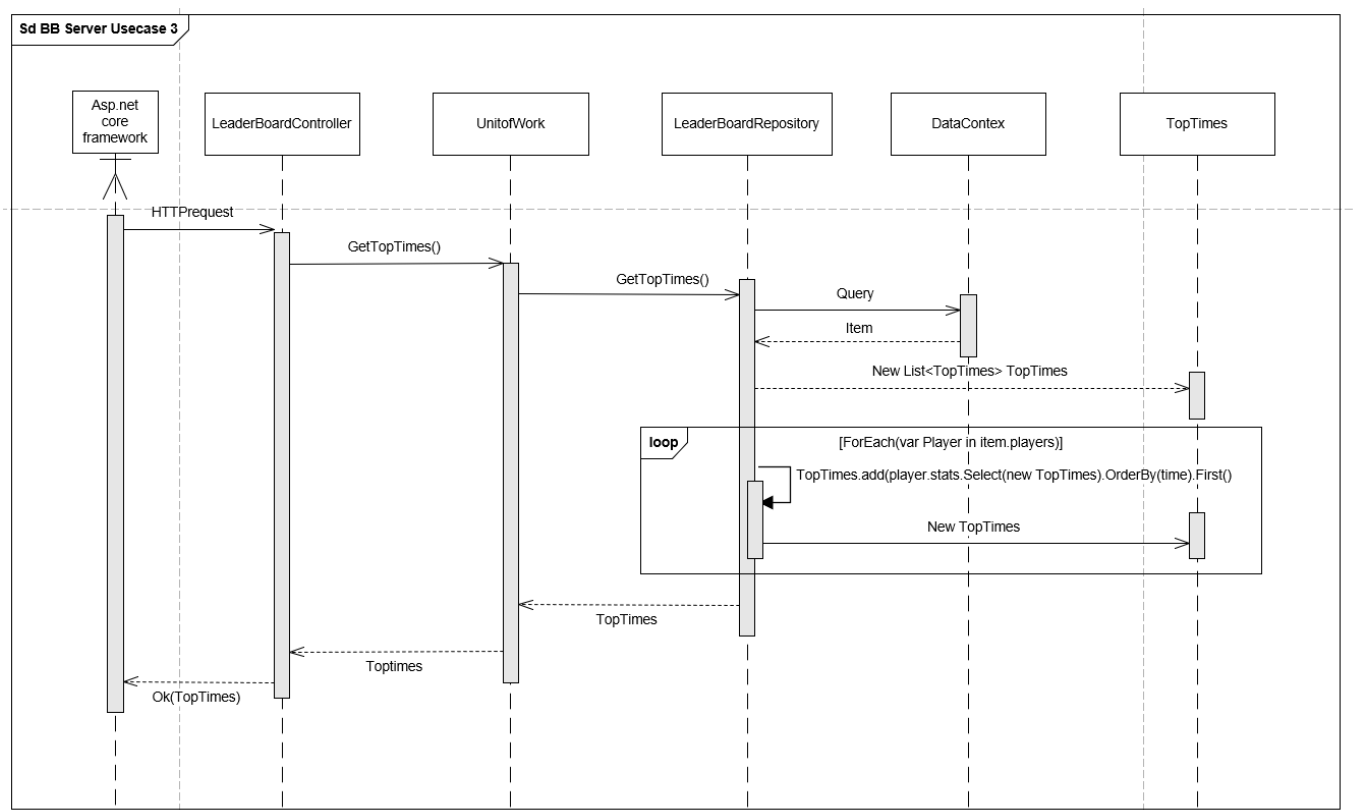
Zommes der ind på UnitofWork viser nedenstående figur 6.15 Hvordan Unitofwork holder på de forskellige repositories samt datacontext klassen.



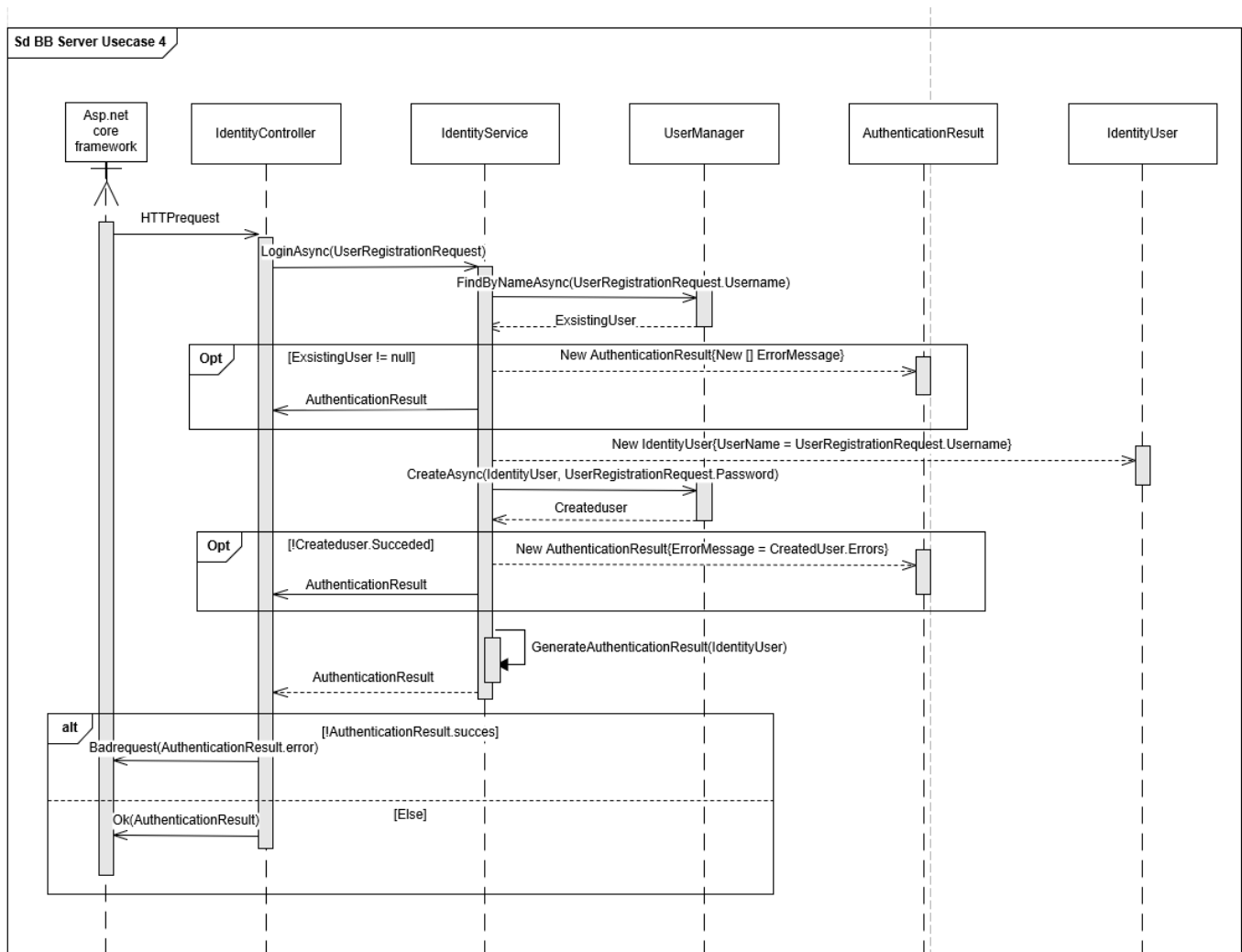
Figur 6.15: Diagram over unitofwork

6.4.2 Sekvensdiagram

Nedenstående figur 6.16 og figur 6.17 fremgår sekvensdiagrammerne, som er udarbejdet ud fra systemets usecases. Sekvensdiagrammerne viser klassernes interaktion med hinanden.



Figur 6.16: Sekvensdiagram af UC3 API



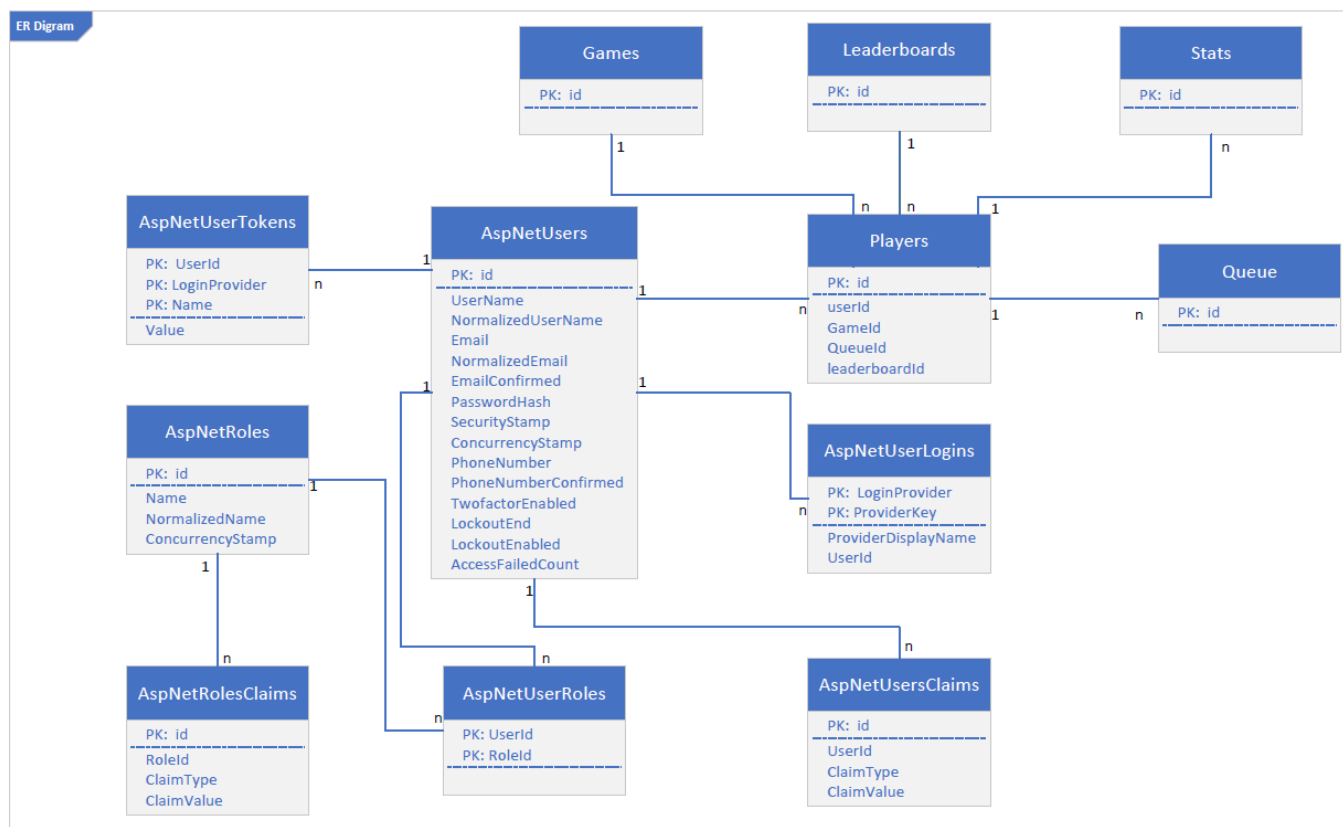
Figur 6.17: Sekvensdiagram af UC4 API

6.5 BeerBong Database

6.5.1 ER Diagram

BeerBong Database er blevet udviklet vha. SQL. Derfor er der blevet udviklet et ER-diagram til at beskrive databasen bedst muligt. Databasen kører på Azure, hvilket gør at databasen altid er aktiv og er klar til at en spiller vil spille eller se leaderboard. Derudover kan det ses på figur 6.18 hvordan det er blevet vlagt at opbygge databasen. Relationerne mellem de forskellige entities kan ses og om det er 'en til mange' eller 'en til en'. Databasen er lavet vha. Identity-biblioteket, som gør det muligt for brugere at logge ind på systemet og derefter gemme login.

Til håndtering af spillere er der blevet lavet en entity kaldet 'Players', som har relationer til henholdsvis 'Games', 'Leaderboards', 'Stats', 'Queue' og AspNetUsers, som indeholder alle gemte brugere der er registreret i systemet.



Figur 6.18: ER diagram for database struktur.

7.1 BeerBong Controller

7.1.1 Indledning

Der vil i følgende afsnit blive gennemgået valg af design og en mere dybdegående gennemgang af valg af komponenter end hvad der blev forklaret i afsnit 3.1.2 ”.

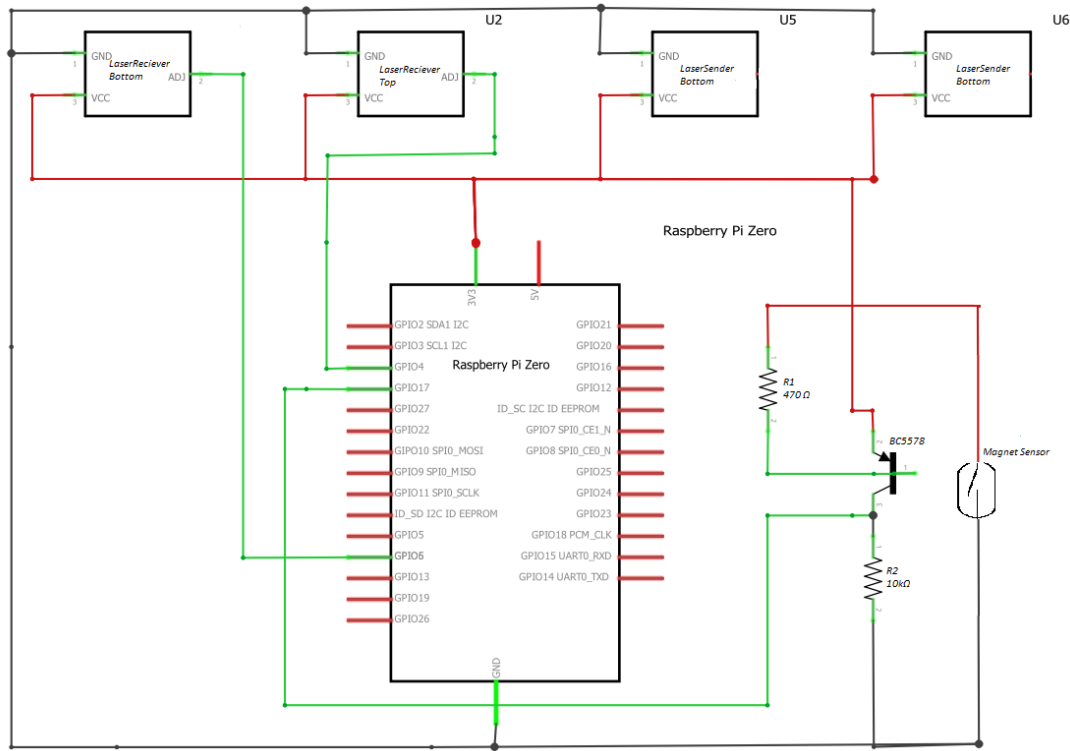
7.1.2 Design

Efter der var blevet besluttet hvilke komponenter der skulle anvendes til hardware for BeerBong Controlleren kunne det hele blive samlet. Det viste sig dog, at Magnet sensoren ikke

På figur 7.1 ses kredsløbsdiagrammet for BeerBong Controlleren. Controlleren består af en Raspberry Pi Zero Wireless, som er forsynet af en powerbank. Raspberry Pi'en forsyner 5 komponenter, 2 LaserReceivers, 2 LaserSendere og en Magnet Sensor. LaserReceiver Bottom er forbundet til Raspberry Pi'ens GPIO 6 port, og LaserReceiver Top er forbundet til GPIO port 4. Hvis laseren registreres i disse receive sendes et højt signal til den givne GPIO port. Den sidste komponent som er forbundet til denne Raspberry Pi er det mindre kredsløb til magnet sensoren, som er en magnetisk reed switch. Denne del af kredsløbet består af reed switchen, to resistorer og en transistor.

Når magneten kommer tæt på, bliver de to elementer inde i reed switchen trukket sammen, og det medfører at transistoren skifter til at være 'on'. Når de så er trukket sammen sker det at R1 trækker basen af transistoren til lavt, så transistoren tænder. Så løber der spænding fra emitteren til collectoren som er forbundet til GPIO14, og det kan læses som et højt signal på Raspberry Pi.

<https://www.gadgetronicx.com/thief-door-alarm-circuit-transistor/>



Figur 7.1: Kredsløb over komponenter tilsluttet Raspberry Pi Zero W

7.1.3 Delkonklusion

Det er en laser og receiver på hver sin side af røret i bunden, og et stykke længere op som passer med 33 cl. Hvis modtageren kan se sensoren sendes der et højt signal ud. Hvis ikke sendes et lavt. Denne mulighed blev valgt fordi det kunne dække begge behov. Laserne tjekker om der er væske i selv røret i ølbongen, og magneterne tjekker om håndtaget er drejet.

I dette afsnit vil systemets software design blive beskrevet. Den software som dette afsnit omfatter, er de komponenter som er beskrevet i software arkitekturen afsnit 6. Den interne funktionalitet i de enkelte containers for systemet som ses på figur 4.2, vil blive beskrevet, der vil blive redegjort for valg af metoder, kommunikation og kode. Derudover vil der være stort fokus på valg af design mæssige valg ifht. til koden i den enkelte moduler/containers.

8.1 Softwaredesign og implementering

8.1.1 BeerBong App

8.1.1.1 Indledning

I dette afsnit vil de design mæssige valg for BeerBong app blive beskrevet. De anvendte design mønstre vil blive redegjort for, og deres betydning for den overordnet funktionalitet beskrevet. Applikationens opbygning og interne funktionalitet vil der blive gået i dybden med, så der opnås en fuld forståelse for den interne logik i de mest væsentlige klasser/metoder, som er essentielle for at applikationen fungerer som en del af hele systemet BeerBongBaster. I de kommende afsnit vil systemet for app'en blive delt op i dets respektive komponenter, og hvert komponent vil så blive gennemgået.

8.1.1.2 Design og implementering

MVP og MVVM pattern

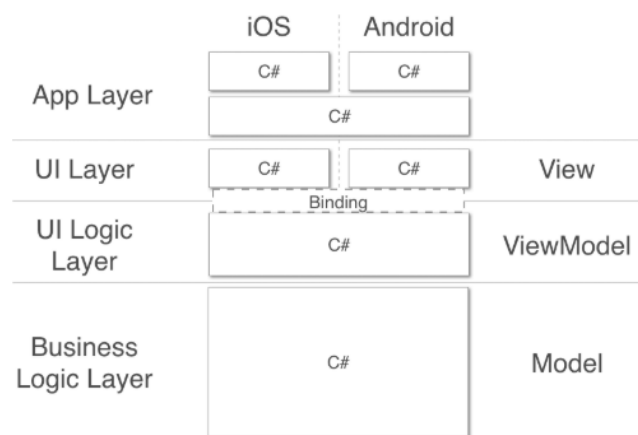
Det overordnet design for BeerBong App består af fire komponenter som hver især indeholder en række forskellige klasser. De fire komponenter kan se på figur 6.2. Komponenterne udgør tilsammen design mønstret MVVM(Model-View-ViewModel), som er et af de design mønstre applikationen er bygget på. Udover MVVM arkitekturen er der også anvendt MVP (Model-View-Presenter) arkitektur/design til udførsel af applikationen. MVP modellen er nødvendig at anvende når man udvikler applikationer i Xamarin til dedikeret Android eller IOS systemer.

BeerBong applikationen er udviklet med MVP mønstret, og først senere i udviklings processen er MVVM implementeret. Når der udvikles efter MVP modellen, anvender man tre komponenter: Model, View og Presenter til at opnå en funktionel applikation. De tre komponenter har hvert deres ansvar, model har til ansvar at definere den data der skal vises på bruger grænsefladen, view har til ansvar at præsentere data'en fra model komponentet og binde kommandoer og begivenheder til presenter komponentet. Presenter komponentet håndterer logikken og funktionaliteten bagved brugerens grænseflade. En dybere forståelse for dette vil blive gennemgået for hvert komponent, i de kommende afsnit. Anvendelse af MVP modellen er god til applikationer der ikke er store komplekse systemer, men hvis applikationen bliver for stor vil MVP modellen istedet have flere negative sider. Disse negative ting kommer fordi hvert view/side indeholder en code-behind fil som har flere referencer til de elementer der fremgår på brugergrænsefladen. Det gør at denne view-kode bruger meget tid på at læse og skrive til og fra brugergrænsefladen, hvilket gør applikationen langsom, da alle begivenheder der sker på grænsefladen er tæt koblet til view koden. Denne tætte kobling skaber to store

problemer. Det første problem opstår når der sker ændringer på grænsefladen, da vil applikationen løbe igennem view koden der er tæt koblet til grænsefladen, og grundet ændringer på grænsefladen, kræver det også ændringer i koden. Det andet problem opstår når der skal laves automatiseret tests til applikationen, da koden kan være meget svær at skrive automatiseret tests til når den er så tæt bundet til grænsefladen og den platform som applikationen køres på.

Løsningen til de problemer som MVP bringer, er at have mindre kode i hvert view, ved at flytte koden fra view komponentet til et andet komponent, og på den måde fjerne den tætte kobling imellem grænsefladen og view koden bagved. Her kommer Model-View-ViewModel mønstret så i brug, ved at flytte koden fra view til viewmodel komponentet. Dette fjerner den tætte kobling imellem view og kode, ved at lade grænsefladen referere til viemodel koden, men uden viewmodel indeholder referencer tilbage til grænsefladen. Istedet vil ændringer som sker på grænsefladen kræve at der bliver justeret i evt. navne og egenskaber defineret i viewmodel klassen.

Dog kan man ikke helt undgå MVP mønstret når man anvender xamarin, men man kan mindske den kode der tilhøre hvert view, og nøjes med enkelte linjer der referere til en viewmodel istedet. Den generelle opbygning af MVVM arkitekturen kan ses på nedenstående figur 8.1.



Figur 8.1: De forskellige komponenter i MVVM, og deres interaktion med hinanden

Singleton

BeerBong Applikationen anvender utrolig meget data der enten skal skrives eller læses fra BeerBong API'et. Data'en der bliver hentet skal oftes også deles imellem flere af siderne på applikationen, og det kan give problemer. Det kan løses ved at give data med i parametrene når man opretter en ny side, fra en anden side, dog er det ikke alle sider der er forbundet på den måde. En anden måde at løse det på er med singleton design mønstret, som er anvendt i BeerBong applikationen. Dette mønster er implementeret ved at have en klasse App der indeholder en række statiske variabler, eks. brugernavn og token. Dette tillader så de forskellige sider at initiere de statiske variabler, og anvende på tværs af siderne. Eks. når en bruger logger ind, så vil brugernavnet blive gemt i variabelen BrugernavnOnLogin, dette tillader så toolbaren på applikationen at vise brugerens brugernavn på alle sider, da toolbaren læser fra samme variable, som blev skrevet til da brugeren loggede ind. For en dybere forståelse for denne klasse henvises der til bilag. ¹.

Komponent Views

Komponentet views, indeholder som beskrevet i arkitekturen afsnit 6 layout filerne for grænsefladen og koden bagved layout filerne. Koden bagved XAML filen, eks. LoginPage.cs står for at initiere layout XAML filen, så den bliver synlig for brugeren. Alt logisk funktionalitet for siden er ifht. MVVM modellen implementeret i LoginPageViewModel som er en del af et andet komponent. XAML filen indeholder data bindinger/referencer til den respektives sides viewmodel fil. Dette gør

¹Sourcecode/BeerBongApp/BeerBong/App.cs

at vores LoginPage.cs fil næsten ikke indeholder noget kode, men istedet er det viewmodellen der styre funktionaliteten.

Komponent ViewModel

Komponentet viewmodel, implementere funktionaliteten og den interne logik for et view, som view så er istand til at binde sig til igennem dets xaml fil via databinding.

Tages der igen udgangspunkt i login usecasen, vil der når brugeren indtaster et brugernavn og password igennem databinding fra view komponentet sendt den information som brugeren har indtastet til viewmodellen. Trykkes der så på login, vil viewmodellen være bundet på en kommando der reagere når login knappen trykkes på, og logikken i viewmodellen vil så blive udført, og brugeren logges ind.

Komponent Model

Komponentet model, er det sidste lag i MVVM modellens opbygning som det ses på figur 8.1. Model klassen håndtere de ikke visuelle klasser, som indkapsulere applikationen data. Model komponenter samarbejder med viewmodellerne og service komponentet, hvor viewmodellen opretter en model af en modelklasse, og service komponenter henter data fra api'et og fylder den respektive model med data. Viewmodellen kan så gemme denne data på applikationen og vise den til brugeren. Med udgangspunkt i login usecasen som før, bliver der oprettet en model af LoginUser når brugeren trykker på login. Modellens indhold som består af to strings, vil så blive sat til hhv. brugers navn og password. Modellen vil så blive givet med som parameter til et service kald, som igennem api'et verificere modellen, altså brugernavn og password, og service komponentet vil så returnere et svar til viewmodellen.

Komponent Services

Services komponentet indeholder de service kald som sker til og fra BeerBong api'et. Et objekt kan oprettes af RestService klassen i service komponentet, op man kan på dette objekt så kalde de forskellige services som komponentet tilbyder.

8.1.2 BeerBong Server

8.1.2.1 Indledning

Nedenstående afsnit gennemgår de design mæssige valg for BeerBong serveren. Dette indebærer arkitektur og valgte design mønstre, og deres betydning for en vedligeholdig, skalerbar og testbar kode, som opretholder SOLID principperne for god kode. Herudover vil alternativer til designet også gennemgås, samt de fordele og ulemper de valgte design løsninger har givet.

8.1.2.2 Design og implementering

Det overordnede design for BeerBong API'et fremgår af 6.14. På figuren fremgår det at der er valgt at implementere et repositorie pattern sammen med unitofwork.

8.1.2.3 Generelt

Med Asp.net core platform kommer en lang række af nyttig værktøjer. En af disse værktøjer er egenskaben at anvende denpendency injection. Dette betyder at vi kan tilføje klasserne til IoC containeren, hvorefter vores afhængigheder automatisk vil blive inject i f.eks. controllernes constructor. Dette betyder at Interface segregation princippet fra SOLID bliver nemmere at anvende i den forstand klassernes afhængighederne afhænger af interfaces, og disse bliver injected i klassernes controllers. Således afhænger controlleren kun af de metoder som bliver defineret i de enkle repositories interfaces.

8.1.2.4 Repository pattern

Funktionaliteten for de forskellige repositories er at medierer mellem domæne- og data-laget. Data-lags mapping bliver håndteret i objekt-relational mapper (O/RM)) af Entityframework core iform af datacontext klassen, således der kan mappes mellem SQL-databasen og de implementerede .Net klasser. I denne sammenhæng fungerer de implementerede repositories som en in-memory kollektion af domæne objekter. Dette betyder at det muliggøres at arbejde på kollektion som var det en simpel kollektion af objektet, hvor man kan slette, redigere og tilføje data.

I sin essens virker et repository som abstraktion oven på Entityframework core, hvor man kan encapsulere sine queries, og genanvende disse. Dette betyder også at BeerBong API'et er uafhængigt af objekt-relational mapperen, og hvis det ønskes at anvende en anden (O/RM) end entityframework core vil dette være muligt. Der implementeres et repository per domæne. Dog er der blevet implementeret et generisk repository som implementerer CRUD operationerne for API, herunder GET, POST, PUT, DELETE og PATCH. De enkle repositories arver således de basale CRUD-operationer fra det generiske repository, og den mere usecase specifikke forretnings logik, bliver implementeret i de enkle repositories. Det skal pointeres at alle repositories er implementeret med et interface.

Et Design med repository pattern bidrage med at implementere SRP og OCP i SOLID principperne. single responsibility princippet ses ved hvert repository har et ansvar til at midere et enkelt domæne mellem data-laget og domæne-. Open close princippet ses ved at de enkle repositories er "open" for at der kan tilføjes funktionalitet igennem deres interfaces, samt at de enkle repositories er "closed" i form af det vel defineret interface som er blevet defineret, som controllernes anvender.

8.1.2.5 Unitofwork

UnitofWorks funktionalitet er at holde styr på de in-memory updates der forekommer når vi anvender de enkle repositories, og sender disse updates til databasen i én transaktion. Dette vil sige at når en bruger specifik aktion som f.eks. at kunne oprette sig om bruger på systemet, som er specificeret i usecase 4. Her bliver alle transaktioner såsom crudoperationerne POST, GET Eller DELETE i denne aktion færdig i én transaktion istedet for flere database transaktioner.

På denne måde bliver Unitofwork lag mellem vores controllers og repositories i systemet. Unitofwork står for at holde på alle de forskellige repositories, som skal modtage datacontext klassen fra entityframework core. Dette forsikre at en transaktion som strækker over flere repositories. Bliver alle enititeter gemt i databasen ellers fejler alle, da det er den samme instans af datacontext klassen fra entityframework core.

8.1.2.6 Delkonklusion

Ved at avende repository pattern og unitofwork får man al funktionaliteten forklaret i de tidligere afsnit, samt endnu en abstraktion til sit object-relational mapper (O/RM) hvilket i dette projekt har været Entityframework core. Ved denne putte en abstraktion mere på mellem data og domæne laget, er API'et gjort uafhængigt af O/RM'en hvilket har gjort designet mere testbart. Dette betyder at ved unit tests kan der anvendes en Mock af repostory og unitofwork til at kunne teste de enkle klasser.

8.1.3 BeerBong Database

8.1.3.1 Indledning

Der vil i dette afsnit argumenteres for valg af komponenter til databasen og i mindre gra komme med alternativer til de løsninger som er blevet taget. Forskellige designmønstre vil blive gennemgået og hvilke fordele og ulemper de havde.

8.1.3.2 Design og implementering

Til udvikling af databasen blev der lavet en context klasse, der arver fra biblioteket 'IdentityDbContext', som gør det muligt for en bruger at logge ind på API'et. I denne klasse bliver der også oprettet de entities, som bliver mappet til databasen hosted på Azure.

Der er både blevet anvendt fluent api og data annotation til udvikling af entities. Alt hvad der kan laves i fluent api kan laves som data annotation, men i gruppen var der enighed om, at en kombination af de to gav en god udnyttelse af de værktøjer og gjorde koden mere læsbar. Desuden var det ikke muligt at lave samtlige relationer som data annotation, derfor var det nødvendigt at lave fluent api til dele af database udviklingen.

8.1.4 BeerBong Controller

8.1.4.1 Indledning

Dette afsnit omhandler designet og implementering af BeerBong Controlleren. Valgene i forhold til designet bliver beskrevet, hvilke design principper og designmønstre der er blevet valgt samt hvorfor. Derudover tages fravalg af design principper også op, samt fordele og ulemper ved det valgte. Det overordnede design af BeerBong Controller fremgår på figur 6.2.

8.1.4.2 Design og implementering

Designet for BeerBong Controlleren er overordnet indelt i to moduler. Det ene modul er RaspberryPi.exe som kører på Raspberry Pi, og det andet er den Websocket som gør det muligt for BeerBong App af få informationer om det state RaspberryPi.exe er i. Denne websocket bliver også hosted fra Raspberry Pi.

State Pattern

Programmet der modtager data fra alle tilsuttet sensorer (Raspberry Pi.exe), dette program tjekker hvilken status der er på ølbongen, hvilket er opbygget som et State Pattern. Det betyder at afhængigt af sensorene sættes programmet i et specifikt state. Som det kan ses i afsnit 6.2 'BeerBong Controller' figur 6.4 er der 3 implementeringer af RaspberryPiStates, som er de states programmet kan være i. Alle 3 implementeringer implementerer den samme funktion IsFull. Funktionen er implementeret anderledes i hvert state, da det er forskellige sensorere den skal kigge på i de forskellige states. Fordelen ved at designe koden efter et state pattern er at det gør det muligt at ændre funktionaliten uden af ændre selve klassen. Main programmet står konstant og kigger på IsFull funktionen for dens Context.

'Context' bliver ved opstart sat til EmptyState, og derefter står Main programmet og kigger på IsFull funktionen. Hvis den øverste sensor sender 'False' tilbage inde i IsFull, sættes 'Context' state til at være 'FullState'. Main'en tjekker stadig kun på 'Context.IsFull', states ændres kun inde i de forskellige states. I 'FullState' tjekkes der på den øverste sensor og en magnet sensor i håndtaget, hvis den øverste laser sensor og magnetsenoren er 'True' startes timeren, og 'Context's state sættes til 'NotDoneState'. Igen tjekker Main på 'Context.IsFull', som nu er sat til at være 'NotDoneState'. Heri kigges på om magnet senoren er 'False' og om den nederste laser sensor er 'True', hvis dette er opfyldt stoppes timeren, og 'Context' state sættes til empty. Undervejs skrives state til en json fil, hver gang der skiftes til et nyt state, og når der skiftes fra NotDoneState skrives tiden også til denne fil.

Andre fordele ved at følge state pattern, er at der følges S og O i SOLID principperne. Det følger Single Responsibility, da koden bliver organiseret således at hver stykke kode til det specifikke klasse bliver enkapsuleret i sin egen klasse, og kun rørt hvis at programmet er i dette state. Open/Close princippet da det er muligt at introducere et nyt state uden ændringer i Context eller de andre states. Dette er meget positivt i forhold skal scalability, da hvis nye features blev lavet til Controlleren, ville det være simpelt at implementere i et nyt state.

Der er overordnet fulgt disse to design principper i designet af Controlleren. Alle 3 sensorere har hver

deres klasse, som implementerer et interface, dette følger også Single Responsibility og Open/Close princippet, da hver klasse kun har et ansvar, og der kan sagtens tilføjes flere sensorer uden at ændre i andre klasser.

Forbindelse til Applikation

Det første valg til at forbinde Controlleren til applikationen, var at forbinde over bluetooth. Dette var designet så Controlleren stod og advertivsedede sig selv hele tiden, og lige så snart at en forbindelse til et andet device blev lavet, blev det andet device trusted og RaspberryPi.exe begyndte at køre. Så var det meningen af Applikationen skulle forbinde direkte til Controlleren, men dette blev ikke ført til ende efter gentagende fejl, som opstod op applikations siden af forbindelsen. Koden for Bluetooth oprettelsen kan stadig findes i source koden, dog udkommenteret i de 3 states.

Eftersom bluetooth ikke længere var en mulig løsning til kommunikation, skulle der nu implementeres en ny løsning til kommunikation mellem BeerBong App og BeerBong Controller. Her var der to løsninger som kunne implementeres. Enten kunne der laves et WebAPI eller en Websocket. Eftersom der ønskes en hurtig kommunikation hvor BeerBong Controller automatisk fik sendt data, blev det valgt at implementere en Websocket, som var et af de valg overvejet i afsnit 3.1.2 ”.

Den valgte Websocket er designet som en node.js Websocket server. Den er implementeret efter en push model, så alle clienter automatisk modtager data fra server. Serveren læser på package.json, som fysisk ligger på Raspberry Pi. Det er denne fil, som RaspberryPi.exe skriver til. Denne JSON fil indeholder fire forskellige keys - name (BeerBong), state, time og comment. State er programmets nuværende state, time er tiden som brugeren har taget om at drikke og comment er eventuelle fejl der er sket i systemet, som kan håndteres af BeerBong App. Comment bliver kun fyldt ud hvis programmet smider en exception. Webserveren sender json filen som en utf8, og den modtages som dette på applikationen.

Production Process Manager (PM2)

Brugervenlighed havde stor betydning igennem hele udviklingen af projektet. Det var bl.a. en af grundende til, at anvende bluetooth i begyndelsen fremfor en websocket. Efter denne metode blev udelukket, ville vi gøre det muligt, at anvende BeerBong Controlleren uden brug af en computer. Derfor skulle alle nødvendige programmer startes ved opstart. Til dette blev der anvendt production process manager (PM2), som kan bruges til Node.js applikationer med en indbygget load balancer. Det gør det muligt at holde et program konstant kørende og give dem admin tilgængeligheder.

Kommunikation 9

9.1 BeerBong Controller

9.1.1 Bluetooth

Der blev forsøgt med at bruge Bluetooth som kommunikationsform imellem Raspberry og Smartphone applikationen. Mere specifikt blev der forsøgt med Bluetooth Low Energy. Bluetooth LE er en 'mindre' version af bluetooth. Der kan overføres med mindre hastigheder, men kræver også mindre energi. Det er optimalt hvis der kun skal overføres mindre mængder data, bluetooth bliver gennemgået mere dybdegående i 9.2.1 'Bluetooth'. I controlleren blev bluetooth forbindelsen implementeret som en seriellport. Hver gang Controller programmet skiftede state ville dette blive skrevet til den port so bluetooth var forbundet til, samt en tid hvis dette var aktuelt. Der blev ikke fulgt nogen specifik protokol, men det specifikke state samt tid blev sendt som en string. I sidste ende blev bluetooth dog ikke implementeret, efter for mange problemer fra BeerBong App siden.

9.1.2 Websocket

En websocket protokol gør det muligt at lave en to vejs kommunikation mellem klient(er) der er godkendt af server. Dvs. at klienterne kører ikke godkendt kode, i et kontrolleret miljø over http. Protokollen starter med et 'Opening Handshake'. Det er klienten som skal initiere forbindelsen til server, ved at kontakte den og herefter spørge om tilladelse til at abonnere på websocket serveren. Når serveren modtager en forespørgelse på en forbindelse, giver den et svar tilbage hvori at protokollen ændres fra HTTP til Websocket.

Websocket serveren holder styr på hvilke klienter der er lavet et 'Handshake' med, så den ikke laver et handshake når der bliver sendt data mellem de to.

Formålet med denne type kommunikation er give en sikker og pålidelig kommunikationsform, hvori klienter får skubbet (push) alt data fra server til klienter istedet for en pull funktion ligesom på WebApi.

9.2 BeerBong App

9.2.1 Bluetooth

Bluetooth er en kommunikationsform der tillader enheder at sende data til og fra hinanden. Bluetooth netværker anvender en master/slave model til at kontrollere hvornår og hvor enheder kan sende data hen. I BeerBongBattle systemet har app'en fungeret som slave til BeerBong controlleren, hvor dens opgave har været at læse data der bliver sendt fra controlleren.

Dette viste sig at give nogle problemer undervejs i implementeringen, da xamarin frameworket ikke officelt understøtter bluetooth, og det er ikke muligt at hente nogle officielle biblioteker der gør det muligt. Der blev prøvet at anvende bruger bygget biblioteker til at løse problemet, men dokumentation for disse var dårlig, og det gjorde det svært at få til at virke. Der blev opnået at skabe en forbindelse

over bluetooth imellem applikationen og controlleren, men det var ikke muligt at sende data imellem de to enheder. Grundet tidspres blev der istedet gået over til en websocket forbindelse som beskrives senere i dette kapitel 9.1.2.

9.2.2 Websocket

Applikationen anvender en websocket forbindelse til BeerBong controlleren, som tillader kommunikation imellem de to enheder, når de er på samme netværk. Applikationen fungerer som websocket klient til controlleren, ved at oprette et objekt af `ClientWebSocket`, som findes i namespace `.Net.WebSocket`, der findes i `.NET` frameworket, og tilbydes af microsoft. Klient objektet gives en URI der forbinder til den specifikke adresse for controller websocket. Hvis forbindelsen til uri'et er åbent, læser klienten fra controlleren og gemmer den modtagne data i en model der svare til den data der modtages. Ønskes der en større forståelse for hvordan klienten på applikationen fungerer se source koden for klassen `Websocket` klient i komponentet `Services` [REF](#) [REF](#) [REF](#).

9.2.3 HTTP

Applikationen fungerer som en `httpclient` til api'et. `HttpClient` på applikationen er en base klasse defineret i `Service` komponentet, som sørger for at sende og modtage HTTP svar fra api'et som defineres i form af en URI. Applikationen kalder asynkrone kald i form af `GET` og `POST` til en specifik uri, som er koblet sammen med api'et. Kaldende fra applikationen anvender en "await" operation på selve kaldet, denne `await` operation suspendere applikationen i at forsætte dens arbejde indtil den asynkrone metode er færdiggjort og `await` operationen har retuneret dets svar. Dette er en super effektiv måde at lave kald til api'et på, da applikationen ofte skal bruge den information som kommer som svar fra kaldet inden den kan gå videre i dens eksekvering af kode. For en dybere forståelse for denne opbygning af asynkrone kald til api'et henvises der til source koden [REF](#) `SOURCE` `KODE`

På applikationen er `.Net.Http` namespace der findes i `.NET` frameworket anvendt, som tilbydes af microsoft. Dette namespace tilbyder et interface som tillader kommunikation over HTTP, og det er dette interface som er anvendt på applikationen.

9.3 BeerBong Server

9.3.1 HTTP/REST

HTTP er en protokol som bruges til kommunikation på WWW, protokollen fungerer ved at bede en server på en TCP port om nogle specifikke resourcer. Serveren svare så med en HTTP kode, som fortæller om resultatet af anmodningen, og derefter sender den selve resourcen, som kan være alt fra et billede, HTML dokument eller et `JSON` objekt som er tilfældet i dette system. HTTP tilbyder otte forskellige handlinger, som en klient kan anmode om. De mest anvendte i projektet er `GET` og `POST`. `GET` bruges til at læse et svar fra api'et, og `POST` bruges til at sende information fra applikationen til api'et.

REST definere en række arkitektoniske principper, som man kan designe sine web services efter. Dette betyder man bygger sine services op efter HTTP methods eller CRUD operationer såsom `POST`, `GET`, `PUT` og `DELETE`. Et Restful API er stateless og passiv. Dette gøres ved at serveren reagere på de endpoints serveren tilbyder, som karakteriseret ved REST arkitekturen. Når en request kommer til serveren via de tilbudte endpoints, så loader server resourcen fra databasen og sender en repræsentation tilbage til klienten. Dette betyder også at "body" på HTTP requestene bruges til at overføre ressourcens state. Serveren kommer aldrig til at gemme noget information om klienten, og er derfor stateless. Derfor bruges der heller ikke sessions i API'et til at gemme information om klienten, men denne information kan ses via en token, klienten giver med i request headeren. REST arkitekturen er derfor en måde at strukturere API'et i forhold til HTTP protokollen i forhold til

HTTP verberne. I tilfældet med BeerBong Serveren retuneres der fra tilbudte endpoints resultater i JSON format. Dette kunne dog også gøres i andre formate såsom XAML eller SOAP.

Modultest 10

10.1 Indledning

Denne sektion gennemgår modultest for de individuelle moduler i dette system. Der vil ved hvert afsnit blive forklaret hvad formålet med modultesten er, hvad det forventede resultat er og det faktiske resultat. Derudover vil der også blive gennemgået Unit test for software modulerne, hvad der er testet, hvad der ikke er testet og begrundelse for dette.

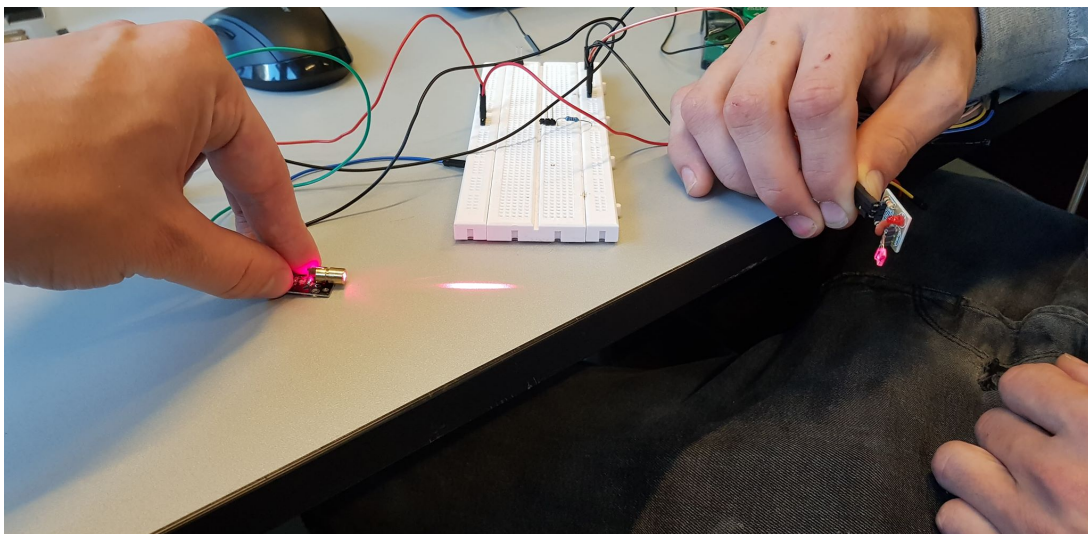
10.2 BeerBong Controller

10.2.1 Modultest af hardware

Til at teste laser recievere, laser sendere og magnet sender, er der overvejende blevet gjort brug af visuelle test. Til at teste om sensorne giver den ønsket funktionalitet er der blevet anvendt forskellige værktøjer. Der er bl.a. blevet anvendt Analog Discovery til at teste output fra begge typer sensor.

10.2.2 Laser Sender

På figur 10.1 ses den visuelle test af laser senderen. Den fik en spænding på 3,3 V fra Analog Discovery, og blev visuelt bedømt om der kom lys eller ej. Når spændingsniveauet opnåede det ønskede niveau, lyste laseren, eller var den slukket.



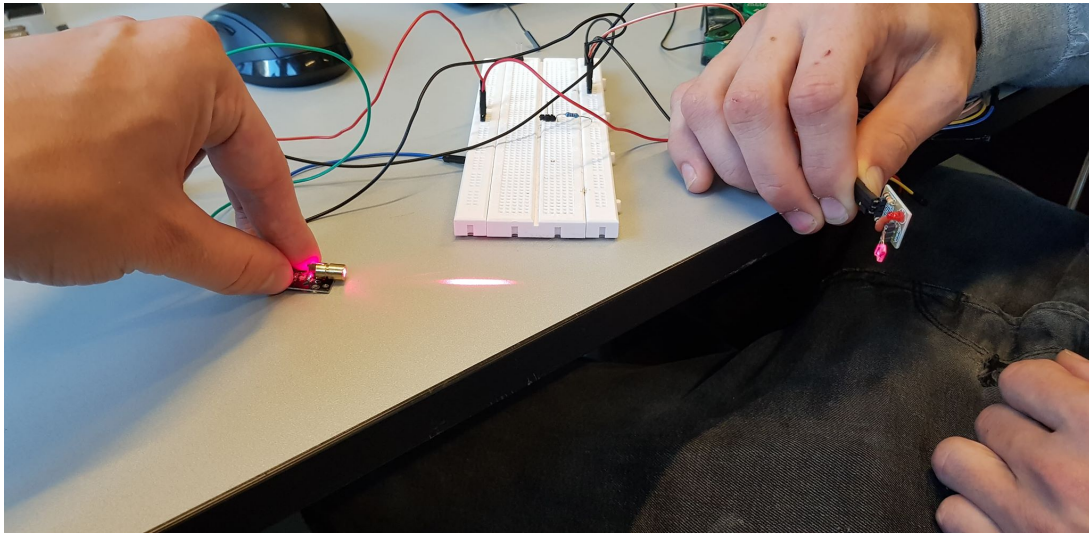
Figur 10.1: Kredsløbsopstilling af Laser Sender til test af output.

10.2.3 Delkonklusion

Efter denne test blev lavet, kunne det bekræftes, at laser senderen fungerede som ønsket, og kunne operere med en forsyningsspænding på 3.3 V.

10.2.4 Laser Reciever

På figur 10.2 ses et billede af forsøgsopstilling til at teste laser receiveren. Både receiver og laser sender er forbundet til Analog Discovery med 3.3 V.. Der blev lyst på laser receiveren med en laser sender, og tjekket på Waveforms om der kom et højt signal. Det ses på figur 10.3 at laser receiveren at der bliver sendt et højt signal på 3.3 V, når der bliver registreret et kraftigt nok lys.



Figur 10.2: Kredsløbsopstilling af Laser Reciever til test af output.



Figur 10.3: Output signal fra Laser Reiever.

10.2.5 Delkonklusion

Efter denne test blev det bekræftet, at laser receiveren sendte det ønskede output, når den registrerede et lys og at den kunne fungere med en forsyningsspænding på 3.3 V.

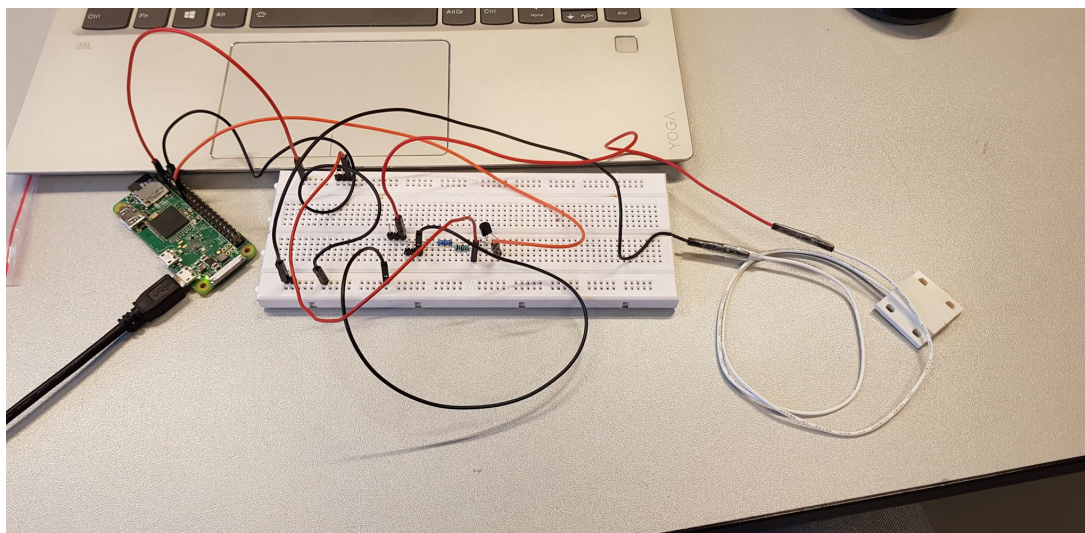
10.2.6 Magnet Sensor

Magnet sensoren blev først testet på et fumlebræt forsynet med Analog Discovery og Raspberry Pi. Denne del af testen kan ses på figur 10.4. Testen blev først lavet på et fumlebræt for at sikre, at den Magnetiske reed switch sensor fungerede efter hensigten. Forsøgsopstillingen blev med følgende

komponenter:

- Transistor: BC5578¹
- 470 Ohm modstand
- 10.000 Ohm modstand
- Analog Discovery
- Fumlebræt
- Raspberry Pi

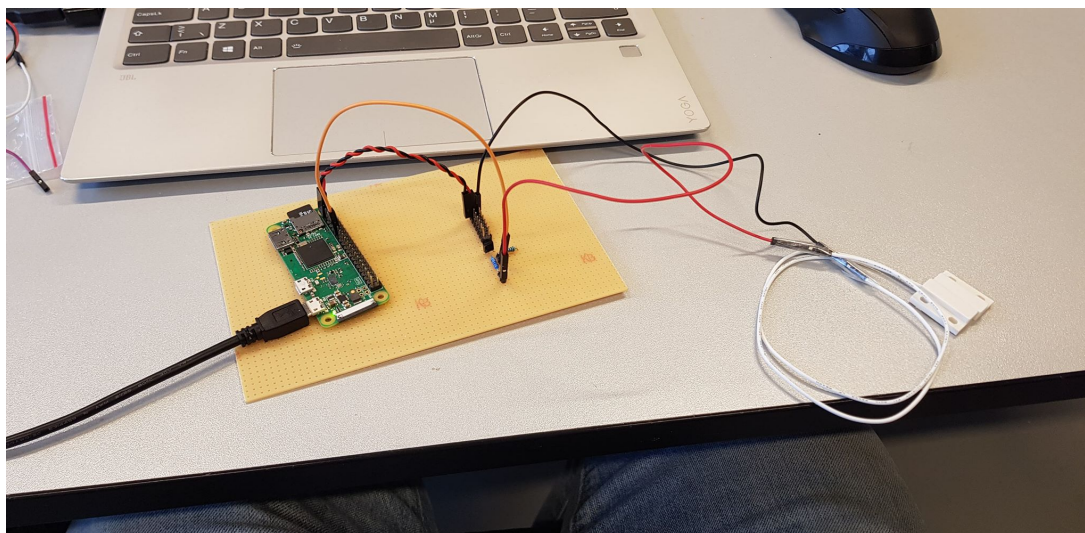
Testen viste Analog Discovery viste et højt signal når sensoren registrerede en magnet i en maksimum afstand på 10mm(Figur 10.6. Det viste sig at hvis spændingen blev øget fra 3.3 V til 5.0 V ville afstanden øges med yderligere 5mm. Dette var imidlertid ikke ønsket, da der ellers skulle laves en levelconverter mellem Raspberry Pi og Magnet Sensoren.



Figur 10.4: Test af Magnet Sensor på fumlebræt.

Efter testen blev fuldstændt på fumlebræt blev der lavet et mere permanent løsning der kunne monteres på BeerBong Controlleren. Der blev lavet et verobard, hvor testen ellers blev udført på samme måde som ved test på fumlebræt. Testen kan ses på figur 10.5 og output på Analog Discovery kan ses på figur 10.6.

¹Datablad: BC5578



Figur 10.5: Test af Magnet Sensor på Veroboard.



Figur 10.6: Output signal fra Magnet Sensor.

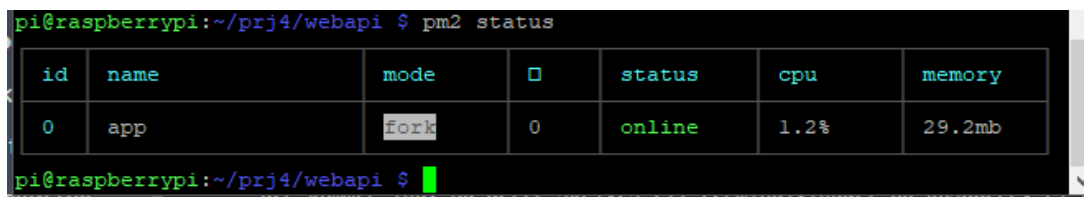
10.2.7 Delkonklusion

Efter denne test blev udført kan det konkluderes, at Magnet Sensoren virker efter hensigten når det lille kredsløb er blevet bygget. Den sender et output signal ud der svarer til den spænding som den bliver forsynet med. Derudover kan det konkluderes, at en højere spænding øger rækkevidden på sensoren. Eftersom det ikke har af stor betydning for projektet, er det blevet besluttet at prioritere at alle sensorer opererer med 3.3 v.

10.2.8 Modultest af software

Til at modulteste BeerBong controlleren på den softwaremæssige del, er der blevet lavet en række unittests, for at teste den interne funktionalitet for programmet. Disse test ligger som et ekstra projekt under den solution hvor programmet er implementeret. Hele controlleret er opbygget som et state pattern, for mere om dette se afsnit 3.1 'BeerBong Controller'. Controlleren er designet, med intention om at opnå et testbart system. Dette betyder at for at alle klasser er implementeringener af interfaces, for at det er muligt at lave fakes af klasserne. Det er vigtigt at der kan laves fakes på

Efter testen blev udført skulle det sikres, at serveren altid blev kørt når Raspberry Pi var forsynet med strøm. Det blev gjort ved hjælp af PM2. På figur 10.8 kan det ses, at programmet kører i baggrunden og hele tiden sender data til alle klienter som abbonnerer på serveren.



```
pi@raspberrypi:~/prj4/webapi $ pm2 status
```

id	name	mode		status	cpu	memory
0	app	fork	0	online	1.2%	29.2mb

```
pi@raspberrypi:~/prj4/webapi $
```

Figur 10.8: PM2 status på Websocket Server.

10.2.8.2 Bluetooth

Raspberry Pi advertiser sig selv med bluetooth LE. Selvom det ikke kom i med i det endelige produkt, var bluetooth med i system så længe hen af vejen at det stadig blev testet. Bluetooth blev testet gennem en bluetooth terminal på en android telefon. Bluetooth var indstillet på RaspberryPi, så hver gang der blev forbundet til et device blev det automatisk trusted, og Controller programmet på RaspberryPi startede op. Til at starte med blev der testet ved at en device fil blev sat til at lytte på bluetooth porten. Når denne devicefil lyttede på bluetooth blev det muligt at sende og modtage i terminalen, ved at kalde echo og cat. Efter at det var bekræftet at der kunne sendes og modtages til terminalen på telefonen, blev det testet med at controller programmet kørte. Der blev tjekket om det korrekte states blev sendt over bluetooth, samt om tiden blev sendt når der var taget en øl. Alle states blev modtaget korrekt på terminalen på telefonen.

Det var ikke muligt at unit teste bluetooth funktioner, da de interagerede direkte med det device som skulle lyttes på, hvilket betød at det skulle være på raspberry pi, og et bluetooth device skulle være connected.

10.2.8.3 Delkonklusion

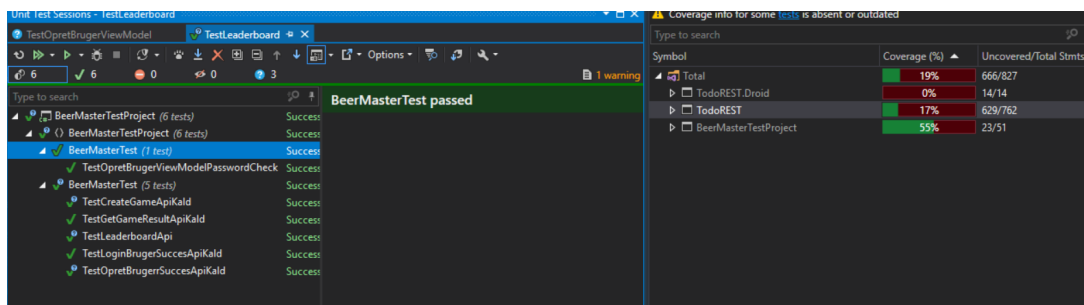
10.3 BeerBong Applikation

10.3.0.1 Unit test af software

Til test af software på BeerBong applikationen er der blevet anvendt unit tests til at teste en række af de funktioner og metoder som udgør applikationen funktionalitet. Mere specifikt er der lavet tests til fem af de mest væsentlige api kald, og på en af viewmodellerne.

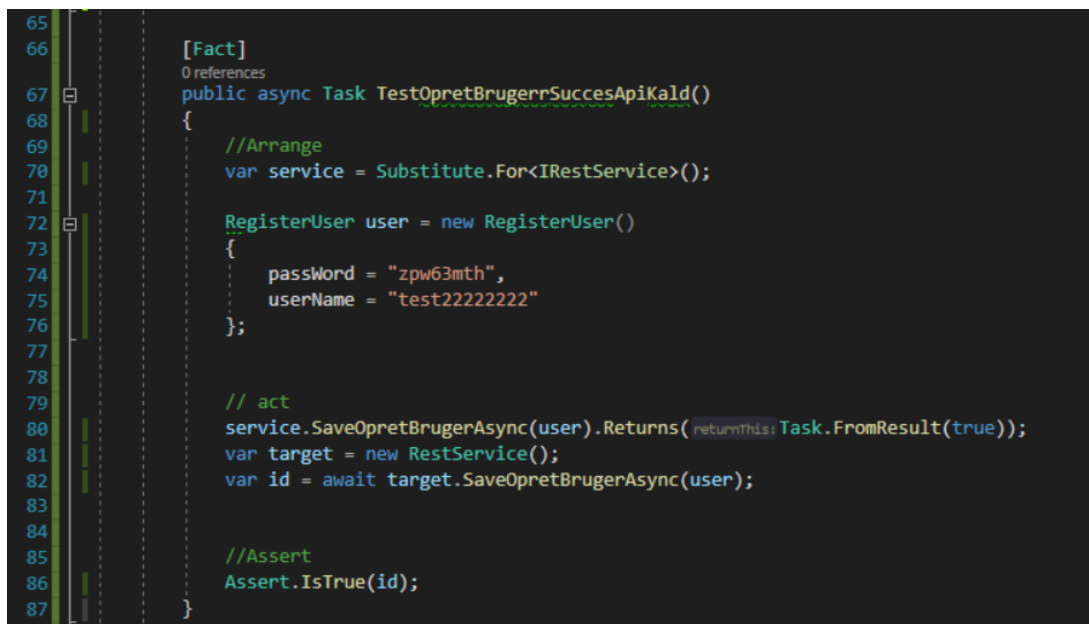
Unit test af api kaldende er lavet med Nsubstitute og XUnit frameworket, og har sikret at det altid var klart om forbindelsen til og fra api'et var intakt, og blev kaldt korrekt fra applikationen. Test af viewmodellen er lavet for at sikre at den korrekte kommando bliver kaldt, når brugeren interagerer med applikationen.

Testprojektet som kører som et parallelt projekt med selve applikationen, har opnået en code coverage på 19% for hele projektet, som det ses på figur 10.9. Dette er ikke optimalt, men af flere grunde har der ikke været tid til at fokusere på denne del af projektet.



Figur 10.9: Testcoverage af BeerBong applikationen

Udover at code coverage ikke er optimal, så har det vist sig at det heller ikke optimalt at unit teste på kald til api'et. Kigges der på figur 10.10 som er en unit test der sørger for at teste kaldet til api'et der opretter en bruger, ses det at der skal gives en bruger med i kaldet til api'et, linje 82. Hvis denne bruger allerede findes i databasen for api'et vil testen fejle, og det vil give et forkert billede af testen. For den gør egentlig det den skal, men api'et returnere bare en fejl, da brugeren allerede findes. Derfor kræves det af programmøren at der ændres i den RegisterUser model som gives med i kaldet, hver gang at testen skal køre, for ikke at fejle. Dette er gældende for flere af kaldene til api'et, og er en af grundene til der ikke er skrevet flere tests, da det fjerner den automatiseret del af tests som man gerne vil opnå via. unit testing. Udover dette, så foregår meget af selve funktionaliteten på api delen af projektet, og selve applikationen udførere minimalt med hensyn til beregninger osv. Istedet lader applikationen api'et gøre arbejdet ved at sende modeller med data, også modtage svar fra api'et. Ønskes der en dybere forståelse for de forskellige unit tests henvises der til source koden for test projektet REF HERXXX.



```

65
66 [Fact]
67 0 references
68 public async Task TestOpretBrugerrSuccesApiKald()
69 {
70     //Arrange
71     var service = Substitute.For<IRestService>();
72
73     RegisterUser user = new RegisterUser()
74     {
75         passWord = "zpw63mth",
76         userName = "test2222222"
77     };
78
79     // act
80     service.SaveOpretBrugerAsync(user).Returns(returnThis: Task.FromResult(true));
81     var target = new RestService();
82     var id = await target.SaveOpretBrugerAsync(user);
83
84
85     //Assert
86     Assert.IsTrue(id);
87 }

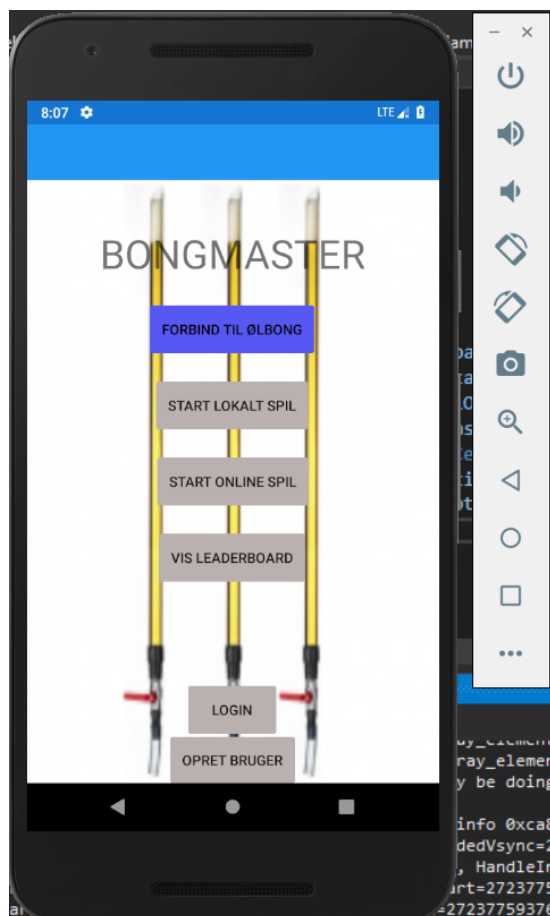
```

Figur 10.10: Unit test af api kaldet der opretter en bruger.

10.3.0.2 Test af brugergrænseflade

BeerBong applikationen indeholder en række views som udgør den grænseflade brugeren præsenteres for. Disse views og brugerens interaktion med dem er utrolig svære at teste vha. unit testing, er derfor er der anvendt en anden metode til at teste disse.

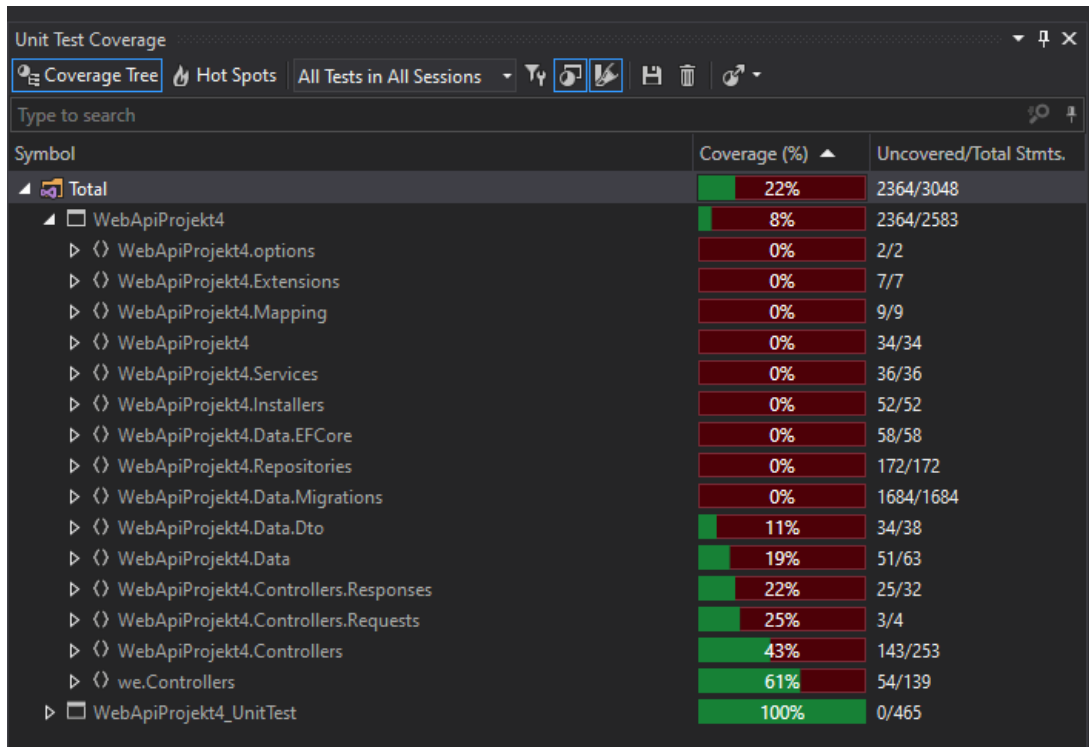
Visual studio som er IDE'en applikationen er udviklet i, tilbyder en android emulator, der faker en mobil telefon og køre applikationen der på, som det ses på figur 10.11. Igennem denne emulator er de forskellige sider på applikationen blevet testet, på ved at sørge for de præsenterer det korrekte for brugeren, men også hver sides funktionalitet i form af at navigere rundt i applikationen. Emulatoren tilbyder nemlig også at man interagere med applikationen, og på den måde er knapper osv. blevet testet. Dette har været utrolig nemt at anvende, og samtidigt været meget essentielt for test af brugergrænsefladen. Det har derfor ikke været nødvendigt at bruge tid på at skrive automatiseret unit tests til denne del af applikationen, da det har været vurderet at emulatoren har tilbydet tilstrækkelig test af brugergrænsefladen.



Figur 10.11: Android emulator til test af BeerBong applikationen

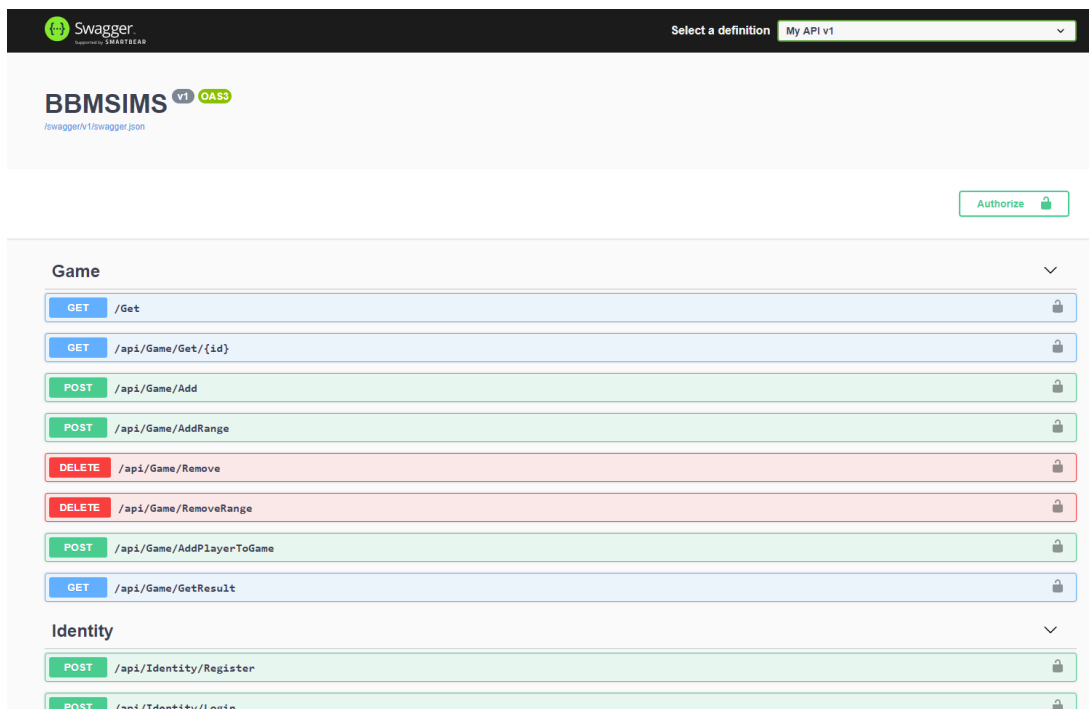
10.4 BeerBong Server

Til at teste de enkelte klassers funktionalitet BeerBong serveren består, er det blevet lavet en række unit tests. Disse test ligger i et separat unit test projektet, hvor der er blevet lavet unit test af controllerne på BeerBong Serveren, hovedsagligt af CRUD operationerne POST, DELETE Og GET. Til at teste API'et er der blevet anvendt Moq og Xunit frameworket til at teste controllernes basale CRUD operationerne, for at se om controllerne anvendte unitofwork og de implementeret repositories korrekt. Unit Testene er blevet lavet i et separat projekt, og der opnået et code coverage på 22% for hele projektet. Dette fremgår af figur 10.12. Dette er ikke optimalt og code coverage burde være 100%, men af flere forskellige grunde har der ikke været tid til at fokusere på denne del, og der har istedet været fokus på BeerBong serverens funktionalitet. Da der i starten af projektet har været taget højde for et testbart design, hvor klasserne er abstraheret med interfaces, og klasserne afhænger af interfaces, som bliver injectet med dependency injecton. Har dette betydet at det har været muligt at anvende moq frameworket til frit at kunne lave mocks af controllernes afhængigheder, og derfor unit teste de enkelte controllere.



Figur 10.12: Test Coverage af BeerBong serveren

I stedet for at afhænge af unit test til at bekræfte de enkle klassers funktionalitet. Er der i store træk blevet anvendt swagger til at dokumentere API'ets funktionalitet. Dette er gjort ved at anvende swashbuckle til at implementere swagger, og udfra den auto genererede UI med swagger, teste controllernes funktionalitet. På figur 10.13 fremgår et snapshot af swagger UI'et. Så i stedet for at unit teste de enkle klassers funktionalitet, er der blevet integrationstestet ved at anvende swagger. Dette har været med til at holde styr på de overordnet billedet af API'ets funktionalitet, og hvordan controllernes har anvendt unitofwork og de implementerede repositories korrekt. Dette er dog ikke en optimal løsning, da de enkles klassers detaljer ikke bliver testet, men det har været nødvendigt i forhold til prioritering af tiden igennem projektet. I swagger er der også blevet taget højde for at kunne teste Identity controller, og se om denne har genere de korrekte tokens ved Register endpointet.



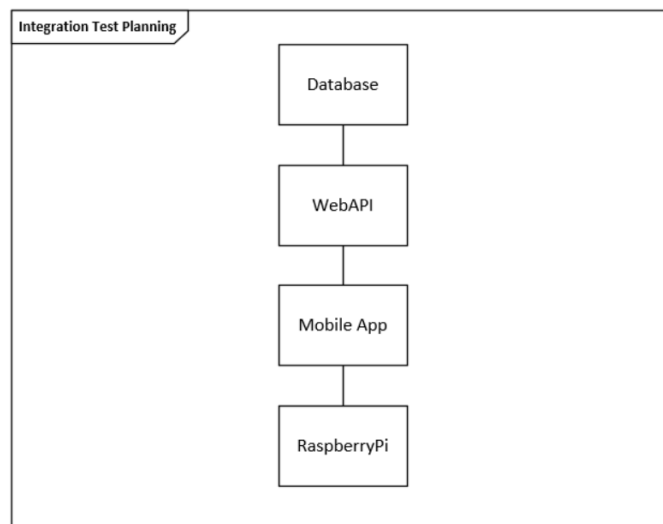
Figur 10.13: Swagger dokumentations UI

Integrationstest

11

11.1 Indledning

I dette afsnit testes systemet ved at teste de enkelte moduler sammen på skift. Der anvendes topdown metode til at teste modulerne sammen. Figur 11.1, viser hvordan systemet vil blive testet ifht. anvendelse af top down metoden, hvor der altså startes fra toppen med at teste database og api, og på den måde arbejde sig længere ned igennem, indtil hele systemet er testet sammen modul for modul.



Figur 11.1: Testplan for integrationstest, hvor top-down metoden anvendes

11.2 Scenarie 1

Enheder under test: BeerBong Database, BeerBong API

11.2.1 Beskrivelse af test

Formålet ved denne test er at skrive noget specifikt data til databasen fra serveren, og se den specifikke data kan findes i databasen. Dette er blevet gjort med samplige endpoints serveren tilbyder. Dog viser nedenstående resultater kun et mindre afsnit af resultaterne.

Der forbindes til databasen igennem Microsoft SQL Management Studio. Databasen "" vælges, og Collection "" vælges, her kan det ses at der er 1 entry og dette er MathiasTP.

11.2.2 Resultater

Test	Forventet resultater	Resultat
POST player	Player kan findes i Databasen	Efter at have anvendt endpoint api/Players/Add kan den givne Player findes i databasen med det korrekt id.
Post Game	Game kan findes i Databasen	Efter at have anvendt endpoint api/Game/Add kan den givne Game findes i databasen
Get LeaderboardTopTimes	Leaderboard med alle scores hentes	Efter at have anvendt endpoint api/LeaderBoard/GetTopTimes returneres alle de bedste tider for players der kan findes i databasen

11.3 Scenarie 2

Enheder under test: BeerBong Database, BeerBong API, BeerBong Applikation

11.3.1 Beskrivelse af test

Formålet med denne test er at sikre at api'et og mobil applikationen fungerer sammen. Testen vil blive udført ved at sende en post metode fra applikationen til api'et. Denne post opretter en ny bruger, som fra api'et bliver gemt i databasen. Dette er blot en af de metoder som testes fra applikationen og til api'et, men udførelsen af denne test er gældende for resten af de metoder der laver kald til api'et fra applikationen, dog findes det ikke relevant at inddrage test af alle metoder her, da den overordnet funktionalitet imellem applikation og api fungerer, hvis blot en af kaldende fra applikationen til apiet fungerer.

11.3.2 Resultater

Test	Forventet resultater	Resultat
Der indtastes brugernavn og password på applikation og efterfølgende trykkes der opret	Den specifikke bruger kan findes i database	Brugeren findes i databasen

11.4 Scenarie 3

Enheder under test: BeerBong Applikation, BeerBong Controller

11.4.1 Beskrivelse af test

BeerBong applikationens forbindelse til BeerBong controlleren testes ved at sende data over en websocket forbindelse, som er skabt imellem de to moduler. Data'en vil blive modtaget på applikationen, og igennem VS debugger, vil der blive tjekket om dataen indeholder det eksakte som controlleren sendte. Controlleren vil sende en tekststreng indeholdende "test, data" over dens websocket forbindelse, og applikationen vil igennem dens websocket forbindelse til controlleren modtage tekststrengen.

11.4.2 Resultater

Test	Forventet resultater	Resultat
Controller sender tekststreng over websocket	Applikationen modtager en tekststreng med det eksakte samme indhold som den sendte fra controlleren	Tekststrengen på applikationen modtages og indeholder det samme som det controlleren sendte

Accepttest

12

12.1 Accepttest for funktionelle krav

Use Case 1

Use case under test	Lokalt spil			
Scenarie	Hovedscenarie			
Prækondition	Applikationen på smartphonen er startet, og startside vises.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på lokalt spil	Bruger præsenteres for spillervalg.	Bruger præsenteres for spillervalg.	OK
2	Bruger vælger 1 spiller og trykker start	Bruger præsenteres for spillerside, og bliver bedt om at fylde ølbong.	Fejl da lokalt spil ikke færdigimplementeret	FAIL

Tabel 12.1: Accepttest af Use Case 1 - Hovedscenarie

Use Case under test	Lokalt spil			
Scenarie	Extension 1			
Prækondition	Applikationen på smartphonen er startet, og startside vises			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (OK/FAIL)
1	Der tilføjes flere spillere	De nye spillere kan ses på listen af spillere	De nye spillere kan ses på listen af spillere	OK
2	Der trykkes start spil	Spil startes	Lokalt spil ikke implementeret yderligere	FAIL

Tabel 12.2: Accepttest af Use Case - Extension 1

Use case under test	Start online spil			
Scenarie	Extension 2 - Flere brugere			
Prækondition	Applikationen på smartphonen er startet, og startside vises.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på lokalt spil	Bruger præsenteres for spillervalg.	Bruger præsenteres for spillervalg.	OK
2	Bruger vælger 1 spiller og trykker start	Bruger præsenteres for spillerside, og bliver bedt om at fylde ølbong.	Lokalt spil ikke implementeret yderligere	FAIL
3	Bruger fylder øl i bongen	Applikationen meddeler at der er nok øl i bongen.	Lokalt spil ikke implementeret yderligere	FAIL
4	Bruger drikker al øl i bongen.	Applikationen viser næste knap.	Lokalt spil ikke implementeret yderligere	FAIL
5	Næste bruger fylder øl i bongen.	Applikationen meddeler at der er nok øl i bongen.	Lokalt spil ikke implementeret yderligere	FAIL
6	Bruger drikker al øl i bongen.	Applikationen viser alle tider.	Lokalt spil ikke implementeret yderligere	FAIL

Tabel 12.3: Accepttest af Use Case 1 - Extension 2

Use Case under test	Lokalt spil			
Scenarie	Extension 3 - Countdown udløber			
Prækondition	Applikationen på smartphonen er startet, og startside vises			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (OK/FAIL)
1	Tester lader countdown udløbe	Der tildeles en ugyldig tid	Lokalt spil ikke implementeret yderligere	FAIL

Tabel 12.4: Accepttest af Use Case 1 - Extension 3

Use Case 2

Use case under test	Online			
Scenarie	Hovedscenarie			
Prækondition	Applikationen på smartphonen er startet, og startside vises.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Start Online Spil'.	Applikationen viser 'Find modstander' knap.	Applikationen viser en 'Find modstander' knap	OK
2	Bruger trykker 'Find modstander'.	Applikationen finder en modstander, og viser en ny side med modstandernavn.	Applikationen finder en modstander, og viser en ny side med modstandernavn.	OK
3	Bruger fylder øl i bongen	Applikationen viser 'Klar' knap efter 1 minut.	Applikationen viser 'Klar' knap efter 1 minut.	
4	Bruger trykker på 'Klar'.	Applikationen starter en tre minutter lang nedtælling	Applikationen starter en tre minutter lang nedtælling	OK
5	Bruger drikker ølbong.	Efter tre minutter viser applikationen tid for spiller, modstander og en vinder.	Efter tre minutter viser applikationen tid for spiller, modstander og en vinder.	OK

Tabel 12.5: Accepttest af Use Case 2 - Hovedscenarie

Use case under test	Online			
Scenarie	Extension 1			
Prækondition	Applikationen på smartphonen er startet, og startside vises.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Start Online Spil'.	Applikationen viser 'Find modstander' knap.	Applikationen viser 'Find modstander' knap.	OK
2	Bruger trykker 'Find modstander'.	Applikationen finder en modstander, og viser en ny side med modstandernavn.	Applikationen finder en modstander, og viser en ny side med modstandernavn.	OK
3	Bruger fylder øl i bongen	Applikationen meddeler at der ikke er nok øl i bongen.	Applikationen meddeler at der ikke er nok øl i bongen.	OK

Tabel 12.6: Accepttest af Use Case 2 - Extension 1

Use case under test	Online			
Scenarie	Extension 2			
Prækondition	Applikationen på smarthponen er startet, og startside vises.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Start Online Spil'.	Applikationen viser 'Find modstander' knap.	Applikationen viser 'Find modstander' knap.	OK
2	Bruger trykker 'Find modstander'.	Applikationen finder en modstander, og viser en ny side med modstandernavn.	Applikationen finder en modstander, og viser en ny side med modstandernavn.	OK
3	Bruger fylder øl i bongen	Applikationen meddeler at der er nok øl i bongen og starter nedtælling.	Applikationen meddeler at der er nok øl i bongen og starter nedtælling.	OK
4	Bruger når ikke at færdiggøre øl indenfor tidsgrænse.	Bruger får får en ugyldig tid.	Bruger får får en ugyldig tid.	OK

Tabel 12.7: Accepttest af Use Case 2 - Extension 2

Use Case 3

Use case under test	Vis Online Leaderboard			
Scenarie	Hovedscenarie			
Prækondition	Applikationen er startet, bruger er logget ind og startskærm vises.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Vis Leaderboard'.	Applikationen viser online leaderboard.	Applikationen viser online leaderboard.	OK

Tabel 12.8: Accepttest af Use Case 3 - Hovedscenarie

Use Case 4

Use case under test	Opret bruger			
Scenarie	Hovedscenarie			
Prækondition	Brugeren har trykket 'Online spil' eller er på startskærmen.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Opret Bruger'.	Applikationen viser opret side.	Applikationen viser opret side.	
2	Bruger indtaser brugernavn og kodeord, og trykker derefter på 'Opret'.	Applikation validerer brugernavn og kode, og sender bruger tilbage til startskærm.	Applikation validerer brugernavn og kode, og sender bruger tilbage til startskærm.	

Tabel 12.9: Accepttest af Use Case 4 - Hovedscenarie

Use case under test	Opret bruger			
Scenarie	Extension 1			
Prækondition	Brugeren har trykket 'Online spil' eller er på startskærmen.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Opret Bruger'.	Applikationen viser opret side.	Applikationen viser opret side.	OK
2	Bruger indtaser brugernavn og kodeord, og trykker derefter på 'Opret'.	Applikation meddeler at brugernavn er brugt.	Applikation meddeler at brugernavn er brugt.	OK

Tabel 12.10: Accepttest af Use Case 4 - Extension 1

Use case under test	Opret bruger			
Scenarie	Extension 2			
Prækondition	Brugeren har trykket 'Online spil' eller er på startskærmen.			
No.	Handling	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	Bruger trykker på 'Opret Bruger'.	Applikationen viser opret side.	Applikationen viser opret side.	OK
2	Bruger indtaser brugernavn og en invalid kode.	Appen meddeler at password er forkert.	Appen meddeler at password er forkert	OK

Tabel 12.11: Accepttest af Use Case 4 - Extension 2

12.2 Accepttest for ikke funktionelle krav

Ikke funktionelle krav		Test			
No.	Krav	Test/Udførelse	Forventet resultat	Faktisk resultat	Vurdering (+/-)
1	BeerBong Server skal have et leaderboard med plads til minimum 20 tider.	Der bliver vha. Swagger indsat 20 forskellige tider til BeerBong server.	BeerBong server indeholder 20 tider i leaderboard.	BeerBong server indeholder 20 tider i leaderboard.	OK
2	Beerbong server skal give mulighed for at oprette et personligt login.	Der bliver vha. Swagger oprettet en bruger med password og brugernavn.	BeerBong server indeholder det indtastet login.	BeerBong server indeholder det indtastet login.	OK
3	BeerBong Controller skal have en kapacitet på 0,67 liter (+/- 0.05 liter).	Bruger åbner en dåseøl og fylder den i ølbongen.	BeerBong controller registrer væske og sender besked derom.	BeerBong controller registrer væske og sender besked derom.	OK
4	BeerBong App skal have et leaderboard med plads til minimum 20 tider.	Bruger åbner app og viser leaderboard.	Leaderboard viser minimum 20 tider.	Leaderboard viser minimum 20 tider.	OK
5	BeerBong App skal kunne oprette et personligt login.	Bruger opretter en ny bruger via BeerBong App.	Der er oprettet en ny bruger, som kan ses vha. Swagger.	Der er oprettet en ny bruger, som kan ses vha. Swagger.	OK
6	BeerBong App skal give mulighed for burgeren at logge ind med sit personlige login.	Bruger forsøger at logge ind med brugernavn og adgangskode.	Bruger er logget ind.	Bruger er logget ind.	OK
7	. BeerBong Controller skal kunne registrere tider i tidsrummet 0,5-10 sek [+/- 0,1 sek].	Den angivne tid fra BeerBong Controller bliver samlet med et stopur.	Præcisionen er indenfor et acceptabelt område.	Præcisionen er indenfor et acceptabelt område.	OK
8	. BeerBong Controller skal kunne forbinde til et internet netværk..	Der bliver delt internet fra en smartphone.	BeerBong Controller opretter forbindelse til det delte netværk.	BeerBong Controller opretter forbindelse til det delte netværk.	OK
9	. BeerBong Controller skal kunne registrere væskenniveau i ølbong.	Bruger fylder ølbong med væske.	BeerBong sender vha. websocket, en streng der viser registrering af væskenniveau.	BeerBong sender vha. websocket, en streng der viser registrering af væskenniveau.	OK

Tabel 12.12: Accepttest af ikke funktionelle krav