Hochschule Esslingen
University of Applied Sciences

Bachelorarbeit

Entwicklung von Echtzeit-Chatkomponenten mit den Web-Technologien SignalR und Blazor am Beispiel eines Azure-Cloud basierten 2-Personen Spieles

im Studiengang Wirtschaftsinformatik
der Fakultät Informationstechnik

Frank Fuchs

Matrikelnummer 763085

Zeitraum:	01.03.2023 bis 31.07.2023
Erstprüfer:	Prof. Dr. rer. nat. Mirko Sonntag
Zweitprüfer:	Prof. Dr.-Ing. Kai Warendorf

Firma:	SYSTECS Informationssysteme GmbH
Betreuer:	Dr. Thomas Zurawka

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 31. Juli 2023 _____

Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

1 Einleitung

Echtzeit-Komponenten sind in der heutigen Webentwicklung nicht mehr wegzudenken. Sie sorgen dafür, dass Daten zwischen Servern und Clients in Echtzeit ausgetauscht werden können. Dies kann beispielweise bei Anwendungen zur Überwachung einer Produktionslinie eine große Rolle spielen, aber auch bei unkritischen Chat- oder Spieleanwendungen Bedeutung finden. Die Softwarebibliothek SignalR bietet die Möglichkeit, diese Echtzeit-Komponenten auf objektorientierter Ebene zu entwickeln, ohne Hintergrundwissen über Transporttechnologien oder Protokolle zu benötigen. Die Firma SYSTECS Informationssysteme GmbH hat eine Webanwendung entwickelt, auf der das Spiel Oware Abapa, abgekürzt Oware, gespielt werden kann. Oware ist ein abstraktes 2-Personen Spiel, bei dem das Ziel ist, möglichst viele Punkte in Form von Steinen oder Bohnen zu gewinnen. Man kann sowohl gegen eine Künstliche Intelligenz als auch gegen Menschen spielen. Die Webanwendung dient bis zum aktuellen Stand als Demo für die KI-Kompetenz des Unternehmens. Diese Bachelorarbeit widmet sich der Erstellung eines Konzepts und der Implementierung von Echtzeit-Komponenten für die Oware Webanwendung mit der Microsoft Softwarebibliothek SignalR. Diese Komponenten umfassen eine Chatfunktion, die Spielerstellung sowie die Echtzeit Übertragung von Spielzügen zwischen zwei Benutzern. Der Zweck der Arbeit besteht darin die Möglichkeiten der SignalR Bibliothek für SYSTECS zu erarbeiten und zu dokumentieren. Da der Microsoft Technologiestapel zu den Hauptkompetenzen des Unternehmens zählt, wird nur SignalR untersucht und keine alternativen Bibliotheken oder Technologien. Zu Beginn der Arbeit werden die Grundlagen zu den verwendeten Technologien sowie dem Spiel Oware erläutert. Dazu zählen SignalR, ASP.NET, Blazor und Azure. Im anschließenden Kapitel werden die Anforderungen sowie das Konzept erklärt. Es werden Konzepte für eine Chatfunktion, ein Match Making, eine Verbindungsverwaltung der Clients sowie dem Spielablauf erarbeitet. Zur Veranschaulichung der Konzepte werden UML-Diagramme, vor allem Sequenzdiagramme der Kommunikation zwischen Server und Client verwendet. Außerdem wird die Oberfläche der Spielansicht mit einem Chatfenster mit dem webbasierten Tool Balsamiq modelliert. Als Nächstes wird die Implementierung des Konzepts dokumentiert. Dabei wird mit Codebeispielen und Erklärungen die Umsetzung demonstriert. Der Proof of Concept für das Match Making wird anhand einer reinen Chat Webanwendung gezeigt, die während des Verlaufs der Bachelorarbeit zum Einarbeiten in SignalR programmiert wurde. Im letzten Kapitel werden die wichtigsten Erkenntnisse zusammengetragen und eingeordnet.

2 Technologien und Grundlagen

In diesem Teil der Arbeit werden die verwendeten Technologien SignalR, ASP.NET Core, Blazor sowie Azure Cloud genauer erklärt. Außerdem werden die Grundlagen für die Arbeit beschrieben, wie zum Beispiel die Regeln für das Spiel Oware. Alle verwendeten Technologien stammen von Microsoft, da diese zu den Hauptkompetenzen des Unternehmens zählen, bei welchem diese Arbeit erstellt wurde. Dadurch, dass alle verwendeten Technologien vom selben Hersteller kommen, ist die Verwendung des Technologiestapels sehr angenehm und reibungslos umsetzbar. Zudem sind das Aufsetzen, Installieren, Einrichten und Veröffentlichen sehr gut auf der Microsoft Webseite dokumentiert.

2.1 ASP.NET Core

ASP.NET Core ist ein Framework zum Erstellen von Web-Anwendungen und Web-APIs. Es ist plattformübergreifend und kann sowohl in Windows als auch in macOS und Linux verwendet werden. Dabei bietet es verschiedene Bibliotheken und integrierte Frameworks, welche es ermöglichen einfach Anwendungen zu entwickeln und zu veröffentlichen. Außerdem hat ASP.NET Core eine integrierte Unterstützung für das Dependency Injection Entwurfsmuster, was hilfreich ist um Abhängigkeiten im Code so gering wie möglich zu halten. In dieser Arbeit wird vor allem die Bibliothek SignalR und deren Funktionsweise für Echtzeit-Webfunktionen dargestellt.[1] Die verwendete Version ist ASP.NET Core 7.0, welche zum Erstellungszeitpunkt der Arbeit die aktuelle veröffentlichte Version ist.

2.2 SignalR

SignalR ist eine Bibliothek von Microsoft zur Entwicklung von Echtzeit-Webfunktionen in ASP.NET Core Anwendungen. Microsoft selbst empfiehlt SignalR für verschiedenste Webanwendungen, zum Beispiel Überwachungsanwendungen, die sofort und automatisch Updates vom Server an das Frontend schicken müssen oder Anwendungen, die regelmäßig Benachrichtigungen an Clients schicken möchten. Auch Spiele- und Chatanwendungen gehören dazu, wozu die in dieser Arbeit erstellte Anwendung zählt. Die Echtzeitfunktionalität wird bei SignalR durch Remote-Procedure-Calls umgesetzt, die es ermöglichen, dass der Server bei bestimmten Clients Methoden aufruft, ohne dass der Client eine Anfrage an den Server geschickt haben muss. SignalR bietet dafür eine API, mit der es einfach möglich ist, solche Remote Procedure Calls zu erstellen. Die Methoden, die beim Client aufgerufen

[1] Vgl. Microsoft Documentation – Overview of ASP.NET Core.

werden, können in verschiedenen Programmiersprachen geschrieben sein, zum Beispiel JavaScript. SignalR selbst ist in C# geschrieben. In dieser Arbeit wird für das Frontend Blazor verwendet, dadurch sind auch die clientseitigen Methoden mit C# programmiert.[2]

2.2.1 Remote Procedure Call (RPC)

SignalR verwendet, wie in Kapitel 2.2 erwähnt, Remote Procedure Calls, um Nachrichten und Methodenaufrufe von dem Server an die Clients zu schicken. Ursprünglich sind Remote Procedure Calls für Anfragen von Clients an Server gedacht. Der Client schickt eine Anfrage an den Server mit der Methode, die aufgerufen werden soll und den benötigten Parametern für die Methode. Der Server führt diese Methode mit den entsprechenden Parametern dann aus und schickt das Ergebnis zurück an den Client.[3] Bei SignalR wird dieses Ablaufmodell in beide Richtungen verwendet, das heißt dass, auch der Server Anfragen an Clients schicken kann, und dann clientseitig die entsprechende Methode aufgerufen wird.

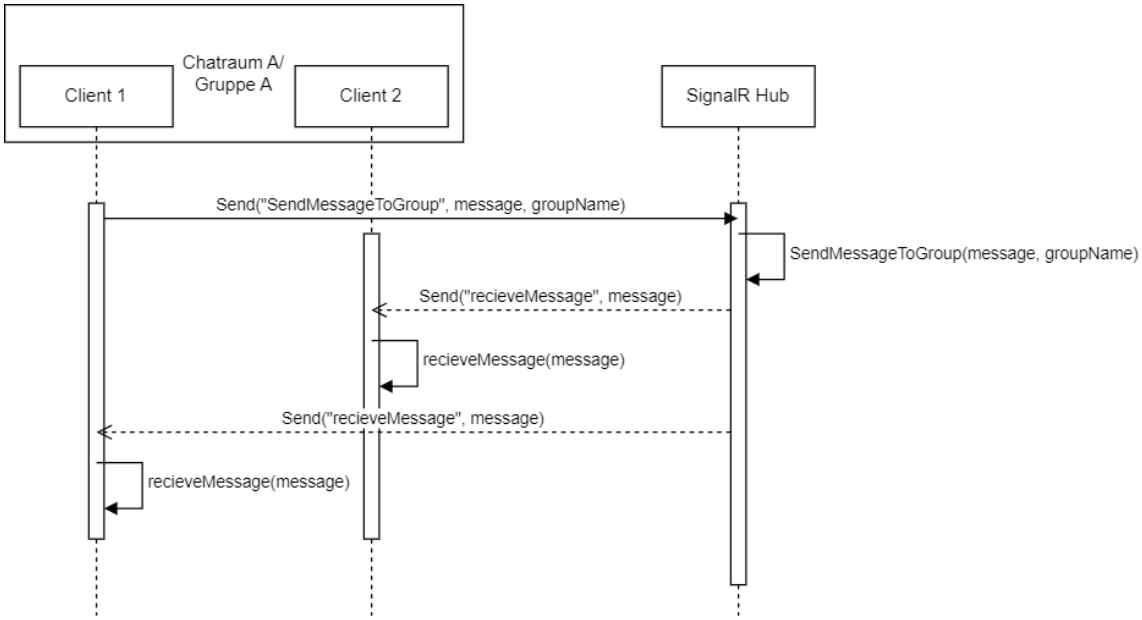

Abbildung 1: Sequenzdiagramm für das Verschicken einer Nachricht mit SignalR[4]

In Abbildung 1 sind sequenziell die Abläufe eines einfachen Remote Procedure Calls mit SignalR dargestellt. Zuerst schickt *Client 1* eine Anfrage an den *SignalR Hub,* dieser stellt den Server dar. Die Anfrage möchte beim Server die Methode *SendMessageToGroup*() aufrufen, die eine Nachricht an eine Gruppe von Clients schickt. Der Parameter *message* soll vom *SignalR Hub* an die gewünschten

[2] Vgl. Microsoft Documentation – Overview of ASP.NET Core SignalR
[3] Vgl. Thurlow, R., Seite 4
[4] Eigene Darstellung

Clients weitergeleitet werden. Der Parameter *groupName* gibt an, bei welchen Clients der Server die Methode aufrufen soll. Das *SignalR Hub* ruft bei sich die Methode *SendMessageToGroup*() auf, die wiederrum eine Anfrage an die spezifizierten Clients, in diesem Fall *Client 1* und *Client 2* schickt. Diese Anfrage enthält den Methodennamen „recieveMessage“ als Parameter, sowie die *message* selbst. Bei den Clients wird jetzt jeweils die Methode *recieveMessage()* aufgerufen, welche den Clients die Nachricht, die in *message* enthalten ist, in der Oberfläche anzeigt. Dieser Ablauf kann abstrahiert als Beispiel für die Grundlage für jede Kommunikation zwischen Clients und SignalR verwendet werden.

2.2.2 SignalR - Verbindungsverwaltung

SignalR bietet einige Funktionen, die eine Lösung für die grundlegenden Abläufe in einer Echtzeitanwendung anbieten. So gibt es eine automatische Verbindungsverwaltung, die verbundenen Clients eine sogenannte Connection-ID zuweist. Diese ID ist ein 22 Zeichen langer String, über den jeder Client identifiziert werden kann. Sobald ein Client die Verbindung verliert oder unterbricht, wird die Connection-ID gelöscht. Bei einer Wiederherstellung der Verbindung wird wieder eine neue Connection-ID vergeben. Connection-IDs können als Gruppen zusammengefasst werden, die einfach angelegt werden können und mit einem, bei der Erstellung festgelegten String abfragbar, beziehungsweise erreichbar sind. Außer der Zuweisung von IDs an Clients und den Gruppen gibt es Methoden, um bestimmten verbundenen Clients Aufrufe zu schicken.[5] Die folgenden Abbildungen zeigen Beispiel Methoden, die jeweils die als Parameter angegebene Funktion bei den spezifizierten Clients aufrufen.

```
Clients.All.SendAsync("UserDisconnected", _connections.GetConnections());
```

Codebeispiel 1: SignalR Serverseitiger Aufruf an alle verbundenen Clients[6]

```
Clients.Caller.SendAsync("RecieveWarning");
```

Codebeispiel 2: SignalR Serverseitiger Aufruf an den Client Aufrufer[7]

[5] Vgl. Microsoft Documentation – Overview of ASP.NET Core SignalR
[6] Eigener Code
[7] Eigener Code

```
Clients.Group(groupName).SendAsync("RecieveMessage", user, message,
connectionId);
```

Codebeispiel 3: SignalR serverseitiger Aufruf an eine Gruppe von Clients[8]

In Codebeispiel 1 wird bei allen Verbundenen Clients die Methode „UserDisconnected“ aufgerufen. Codebeispiel 2 zeigt die Methode, die nur bei dem Client, der die Server Methode aufgerufen hat, eine Methode aufruft und in Codebeispiel 3 ist eine Methode zu sehen, welche allen Clients in der Gruppe, die mit dem Parameter „groupName“ angesprochen wird, einen Methodenaufruf schickt. Der „SendAsync“ Methode können auch Parameter übergeben werden, die dann den clientseitigen Methoden übergeben werden.

2.2.3 SignalR – Transporttechnologien

Für den Transport der Remote Procedure Calls wird von SignalR automatisch die beste zur Verfügung stehende Technologie ausgewählt. Im Optimalfall werden WebSockets verwendet, falls dies nicht möglich ist, wird automatisch auf Server-Sent Events zurückgegriffen und als letzte Möglichkeit wird Long Polling verwendet.[9] Dies ist eine weitere Erleichterung zu der bereits erwähnten Verbindungsverwaltung bei der Verwendung von SignalR im Vergleich zu einer eigenen Implementierung von Remote Procedure Calls, da man mehrere Technologien zur Verfügung hat und diese je nach Situation ausgewählt werden, ohne dass man dies bei der Erstellung einer SignalR Anwendung berücksichtigen muss.[10]

2.3 Blazor

Blazor ist ein Framework zur Erstellung von Webanwendungen. Es stammt wie SignalR auch von Microsoft und ist speziell für ASP.NET Core Anwendungen gedacht. Ein großer Unterschied von Blazor zu anderen Webframeworks ist, dass kein JavaScript für den clientseitigen Code benötigt wird, sondern alles in C# geschrieben wird. Eine Blazor Anwendung besteht aus Komponenten. Jeder Teil oder Bereich der Oberfläche einer Blazor Anwendung kann als Komponente abgebildet werden. Komponenten sind im Code C# Klassen. Eine Komponente besteht mindestens aus einer sogenannten Razor Page, die bereits HTML, CSS und bereits Funktionalität in Form von C# Code enthalten kann. Zusätzlich kann für jede Razor Page eine eigene C#-Datei erstellt werden, in der die Funktionalität ausgelagert werden kann, um eine übersichtlichere Dateienstruktur und

[8] Eigener Code

[9] Vgl. Microsoft Documentation – Overview of ASP.NET Core SignalR

[10] Vgl. Microsoft Documentation – WebSockets support in ASP.NET Core

übersichtlicheren Code zu bekommen. Es gibt mit Blazor mehrere Möglichkeiten, Webanwendungen zu erstellen. Die zwei wichtigsten sind Blazor Server und Blazor WebAssembly. Mit Blazor WebAssembly ist es möglich, Singe-Page-Anwendungen auf der Basis des .NET Frameworks zu erstellen.

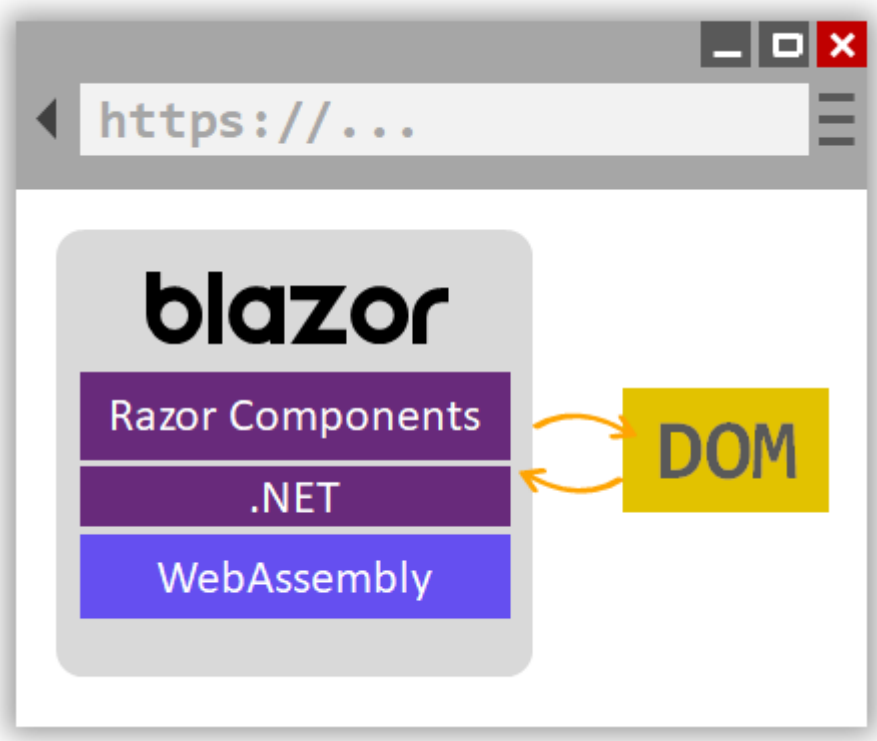

Abbildung 2: Aufbau einer Blazor WebAssembly Anwendung[11]

Dabei wird eine clientseitige .NET-Anwendung entwickelt, die dann mithilfe von WebAssembly im Browser ausgeführt werden kann. In Abbildung 5 ist zu sehen, dass die komplette Anwendung im Browser der Clients läuft und dort direkt die HTML-Komponenten aktualisiert. Die zweite Möglichkeit, die auch für diese Arbeit verwendet wurde, ist Blazor Server. Bei dieser Variante wird die Anwendung serverseitig ausgeführt, das heißt die gesamte Funktionalität liegt auf dem Server und schickt Updates an die clientseitige Benutzeroberfläche.

[11] https://learn.microsoft.com/en-us/aspnet/core/blazor/index/_static/blazor-webassembly.png?view=aspnetcore-7.0 [abgerufen am 04.07.2023]

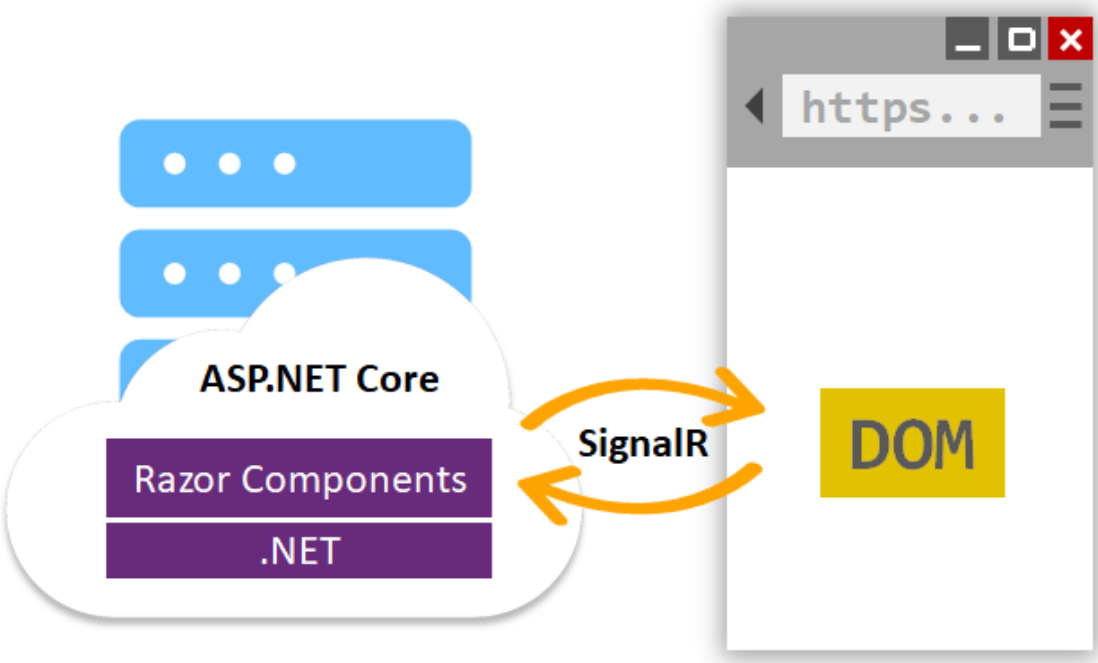

Abbildung 3: Aufbau eine Blazor Server Anwendung[12]

In Abbildung 6 ist der Unterschied zur WebAssembly Variante verdeutlicht. Man erkennt, dass die Anwendung außerhalb des Browsers läuft und über Remote Procedure Calls die Oberfläche aktualisiert. Die Updates werden dabei automatisch mit SignalR an den Client-Browser gesendet. Diese Variante verwendet statt dem reinen .NET-Framework das ASP.NET Core Framework.[13]

2.4 Azure

Microsoft Azure ist eine Cloudplattform, die eine Vielzahl von verschiedenen Produkten für die Softwareentwicklung anbietet. In dieser Arbeit wird nur der App Service verwendet, der es ermöglicht, eine Webanwendung in der Azure Cloud zu hosten ohne sich um die Infrastruktur kümmern zu müssen.

2.5 Oware

Oware ist ein strategisches Spiel für zwei Spieler. Es geht darum, eine bestimmte Anzahl von Punkten, oft Bohnen oder Steine genannt, da diese ursprünglich als Spielmittel dienten, zu gewinnen. Das Spielfeld besteht aus zwölf Feldern, die in zwei Reihen mit je sechs Feldern unterteilt sind, wie in Abbildung 4 zu sehen.

[12] https://learn.microsoft.com/en-us/aspnet/core/blazor/index/_static/blazor-server.png?view=aspnetcore-7.0 [abgerufen am 04.07.2023]

[13] Vgl. Microsoft Documentation – ASP.NET Core Blazor

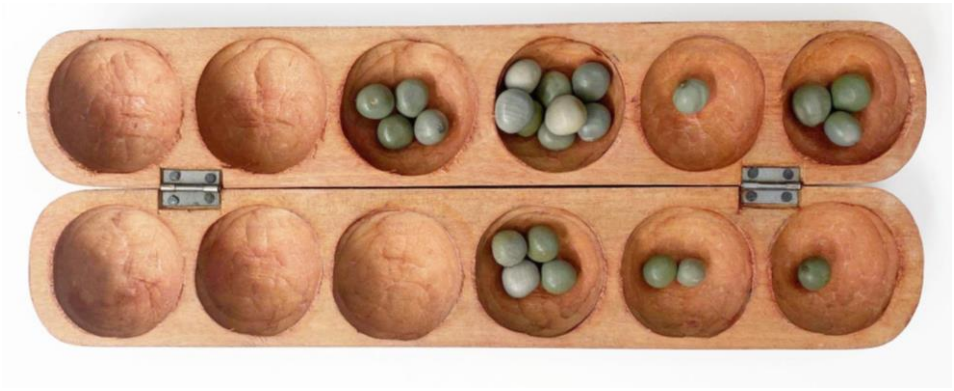

Abbildung 4: Oware Abapa Spielbrett[14]

Jedem Spieler gehört eine Seite mit sechs Feldern. Zu Beginn des Spiels liegen in jedem Feld drei Steine, also insgesamt 36 Steine. Jeder Spieler ist abwechselnd an der Reihe und nimmt alle Steine aus einem seiner Felder und verteilt diese gegen den Uhrzeigersinn auf die nächsten Felder. Dabei wird in jedes Feld ein Stein gelegt. Das heißt, wenn in dem ursprünglichen Feld drei Steine waren, haben nach dem Zug die folgenden drei Felder einen Stein mehr. Sobald nach einem Zug der letzte Stein in ein gegnerisches Feld gelegt wird und sich mit dem neuen Stein insgesamt zwei oder drei Steine in dem Feld befinden gewinnt der Spieler am Zug alle Steine aus dem Feld. Wenn dann das vorletzte Feld auch ein gegnerisches Feld ist, in dem sich nach dem Zug zwei oder drei Steine befinden, gewinnt der Spieler am Zug diese ebenfalls. Dies geht so weiter bis entweder die Anzahl kleiner zwei oder größer drei ist, oder das Feld kein gegnerisches Feld ist. Sobald ein Spieler mehr als die Hälfte der Steine gewonnen hat, also 19, gewinnt er das Spiel. Wenn ein Spieler an der Reihe ist, aber keine Steine in seinen Feldern sind, bekommt der andere Spieler alle Steine auf seiner Seite und das Spiel ist zu Ende. Es kann außerdem zu einer Endlosschleife kommen, bei der sich die Brettkonstellation nach jedem zweiten Zug wiederholt. Hier wird das Spiel ebenfalls beendet und jeder Spieler bekommt die Steine aus seinen Feldern.

[14] https://img.welt.de/img/reise/mobile230530433/8491620167-ci23x11-w1920/WS-Reise-Souvenir-Ghana-Oware-Spiel-Kat-3.jpg [abgerufen am 10.07.2023]

3 Konzept

In diesem Kapitel werden die Anforderungen sowie das Konzept für die Echtzeitkomponenten vorgestellt. Es soll eine Blazor Webanwendung sein, die Benutzern ermöglicht, das Spiel Oware gegen andere Spieler in Echtzeit zu spielen. Während des Spiels soll ein Austausch der beiden Benutzer über einen Chat möglich sein. Zuerst werden die Anforderungen an die Webanwendung genau beschrieben. Außerdem werden die benötigten Abläufe und Komponenten für eine Chatfunktion, sowie das Spielen und das Matchmaking, also das Zusammenbringen der Benutzer für ein Spiel dokumentiert.

3.1 Funktionale Anforderungen

Die folgenden Stichpunkte zählen die funktionalen Anforderungen auf:

- Jeder Benutzer kann in Echtzeit gegen einen anderen Benutzer über eine Internetverbindung das Spiel Oware spielen.
- Jeder Benutzer kann ich Echtzeit, während eines laufenden Spiels mit dem Gegnerischen Benutzer chatten.
- Jeder Benutzer kann ohne Registrierung oder Anmeldung alle Funktionen der Anwendung nutzen.
    a. Ein Benutzer soll trotzdem, bei einem kurzzeitigen Verbindungsabbruch wieder zugeordnet und identifiziert werden können.
- Jeder Benutzer kann ein Spiel erstellen oder einem bereits erstellten Spiel beitreten.
- Es müssen genau 2 Spieler an einem Spiel teilnehmen, bevor gespielt werden kann.
    a. Es darf nicht möglich sein, dass einem Spiel mehr als 2 Spieler beitreten oder Zugriff auf den Chat oder das Spielfeld haben.
- Es soll eine automatische aktualisierende Übersicht über alle erstellten und nicht beendeten Spiele geben.
    a. Es soll daraus ersichtlich sein wie viele Benutzer im Spiel sind.
    b. Wenn alle Benutzer ein Spiel verlassen, soll das Spiel automatisch geschlossen werden.

3.2 Modellierung

Um die in Kapitel 3.1 aufgezählten Anforderungen umzusetzen und sicherzustellen, wurden zunächst verschiedene Abläufe und notwendige Komponenten modelliert. Der Fokus wird dabei auf die Echtzeitfunktionalität mit der SignalR Bibliothek gelegt. Die gesamte Anwendung beinhaltet weitere Komponenten.

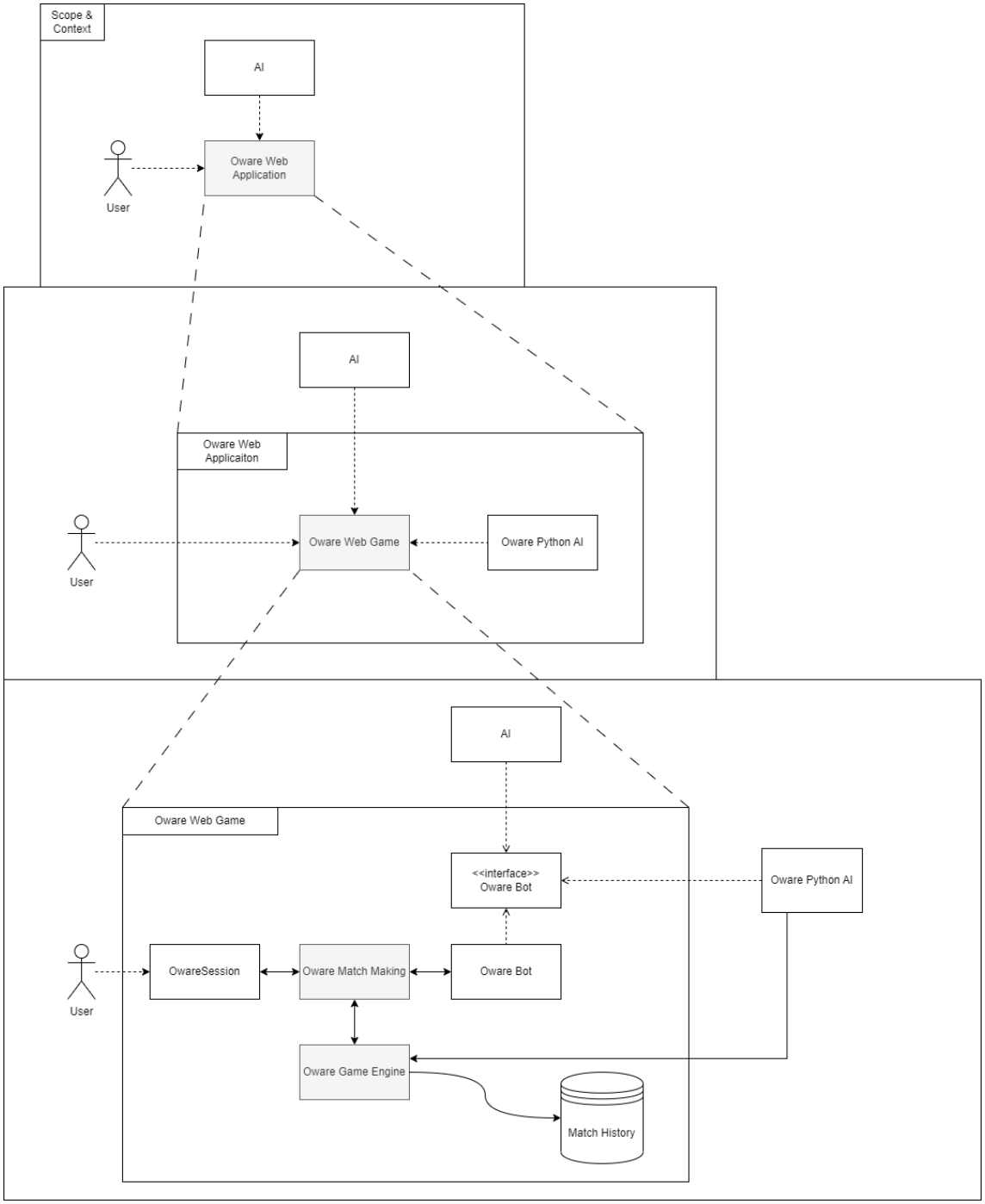

Abbildung 5: Umfang der gesamten Webanwendung[15]

[15] Eigene Darstellung

In Abbildung 8 ist der Umfang der gesamten Webanwendung und den Akteuren, die mit ihr interagieren oder mit denen die Anwendung interagiert, dargestellt. Es soll Benutzern auch möglich sein, gegen eine Künstliche Intelligenz zu spielen oder eine eigene Künstliche Intelligenz antreten zu lassen. Diese Funktionen werden aber nicht Teil dieser Arbeit sein, sondern es wird das Matchmaking und der Spielablauf sowie die Chatfunktion modelliert und später versucht durch eine Umsetzung das Konzept zu realisieren.

3.2.1 Verbindungsverwaltung

Als Grundlage soll die SignalR Verbindungsverwaltung verwendet werden, die den Benutzern zur Laufzeit der Anwendung und während des Bestehens der Verbindung des Clients zum Server eine eindeutige Connection-ID zuweist. Um Clients für ein Spiel zusammenbringen zu können, sollen die SignalR Gruppen verwendet werden. So können dann, wie in Abbildung 1 in Kapitel 2.2.1 bereits dargestellt, Remote Procedure Calls an die Gruppenmitglieder geschickt werden, um Chatnachrichten oder auch Updates für das Spielfeld nach einem Spielzug zu übermitteln. Diese Gruppen sollen maximal zwei Mitglieder haben dürfen und es sollen die Connection-IDs der Mitglieder auslesbar sein. Da SignalR dafür keine Funktion anbietet, das heißt es können weder Gruppengröße noch die Mitglieder IDs einer Gruppe abgerufen werden ist eine Erweiterung der, in SignalR integrierten Verbindungsverwaltung notwendig. Diese soll sicherstellen, dass die Gruppengröße kontrolliert und, gegebenenfalls die Mitglieder Clients einzeln identifiziert werden können.

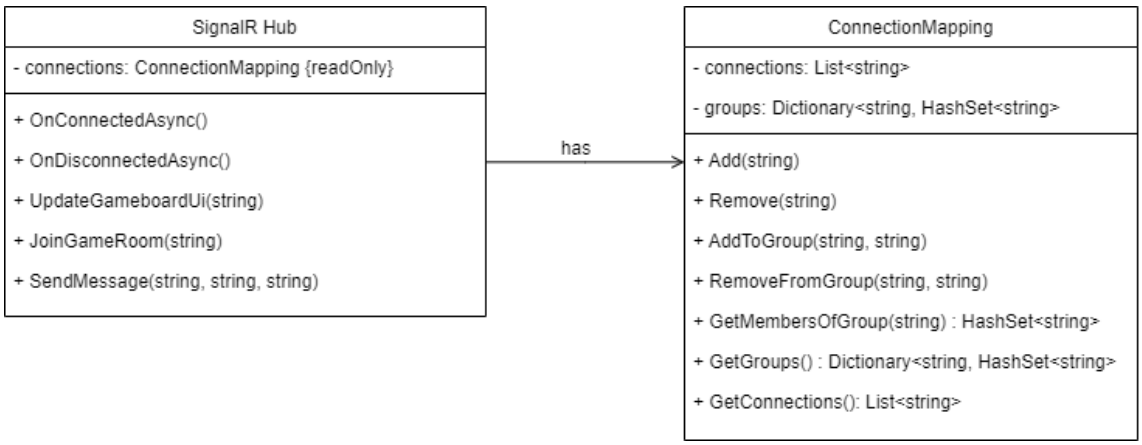

Abbildung 6: SignalR Hub und erweiterte Verbindungsverwaltung[16]

In Abbildung 9 ist die Erweiterung der SignalR Hub Klasse zu sehen. Die neue Klasse ConnectionMapping übernimmt das Speichern der Verbindungen, Gruppen und Gruppenzugehörigkeiten während der Laufzeit der Anwendung. Das Konzept für die erweiterte Verbindungsverwaltung basiert größtenteils auf

[16] Eigene Darstellung

einer von Microsoft selbst vorgeschlagenen Lösung für das Problem. In der Dokumentation für SignalR wird eine ConnectionMapping Klasse beschrieben und beispielhaft implementiert, welche für Benutzer, die sich bei einer Anwendung über Microsoft Identity anmelden, den Identity Benutzernamen und die zugewiesene SignalR Connection-ID speichert.[17] Diese Logik kann angepasst werden und soll statt Benutzername und ID, die Gruppenzugehörigkeit und ID speichern. Die SignalR Hub Klasse kann über das connections Objekt vom Typ ConnectionMapping die Daten verändern oder abrufen.

3.2.2 Match Making

Um die Clients zusammenzuführen, die an einem Spiel teilnehmen wollen, ist ein Match Making notwendig. Wie bereits erwähnt sollen Clients, die gegeneinander spielen wollen, der gleichen SignalR Gruppe hinzugefügt werden. Die Idee ist es eine Übersicht zu erstellen in der alle existierenden Spiele, beziehungsweise SignalR Gruppen aufgelistet sind. Es soll dort die Möglichkeit geben ein neues Spiel zu erstellen oder einem Spiel beizutreten. Um sicherzustellen, dass jeder Client immer eine aktuelle Übersicht hat, muss bei jeder Änderung der Gruppen, also bei Neuerstellung, Änderung der Mitgliederzahl oder Schließung einer Gruppe, ein Update an alle verbundenen Clients geschickt werden.

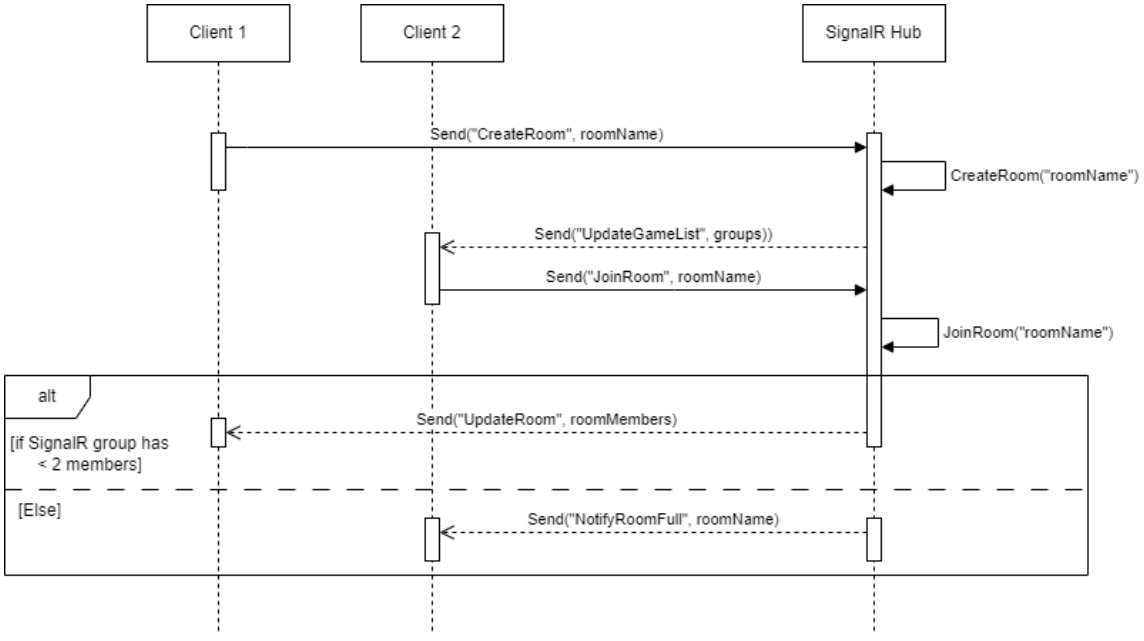

Abbildung 7: Sequenzdiagramm Match Making[18]

In Abbildung 7 ist das Match Making für zwei Clients abgebildet. Der Aufruf des SignalR Hubs *Send(„UpdateGameList“, groups)* geht an alle verbundenen Clients,

[17] Vgl. Microsoft Documentation – Mapping SignalR Users to Connections
[18] Eigene Darstellung

außer den Client, der den Raum erstellt hat. In diesem Diagramm ist dies nur Client 2, theoretisch würde aber jeder andere Client ebenfalls ein Aufruf zum Updaten der Übersicht über die Spiele erhalten. Beim Erstellen eines Raumes soll außerdem eine Überprüfung stattfinden, ob der Raumname bereits verwendet wird. Wenn dies der Fall ist, muss der Client darauf hingewiesen werden und der Raum, sowie die SignalR Gruppe darf nicht erstellt werden. Dies ist notwendig, da der Client im SignalR Hub sonst zu der bestehenden Gruppe hinzugefügt werden würde, anstatt eine neue Gruppe zu erstellen.

3.2.3 Chatfunktion

Die Chatfunktion soll den Clients, die sich in einem Raum befinden, ermöglichen, miteinander Nachrichten auszutauschen. Dabei soll sichergestellt werden, dass nur Clients, die Mitglieder des Raumes sowie der entsprechenden SignalR Gruppe sind, Nachrichten an die anderen Mitglieder schicken können. Da der Chat erst zur Verfügung steht, nachdem Spieler bereits einem Raum und dementsprechend einer SignalR Gruppe beigetreten sind, ist es einfach umsetzbar, dass nur die Gruppenmitglieder die Nachrichten empfangen können. Die SignalR eigene Verbindungsverwaltung kann jedoch nicht überprüfen, ob ein Aufruf an eine SignalR Gruppe von einem Mitglied der Gruppe stammt. Um Nachrichten von Nicht-Mitgliedern abzufangen, kann die erweiterte Verbindungsverwaltung, die in Kapitel 3.2.1 beschrieben ist, genutzt werden.

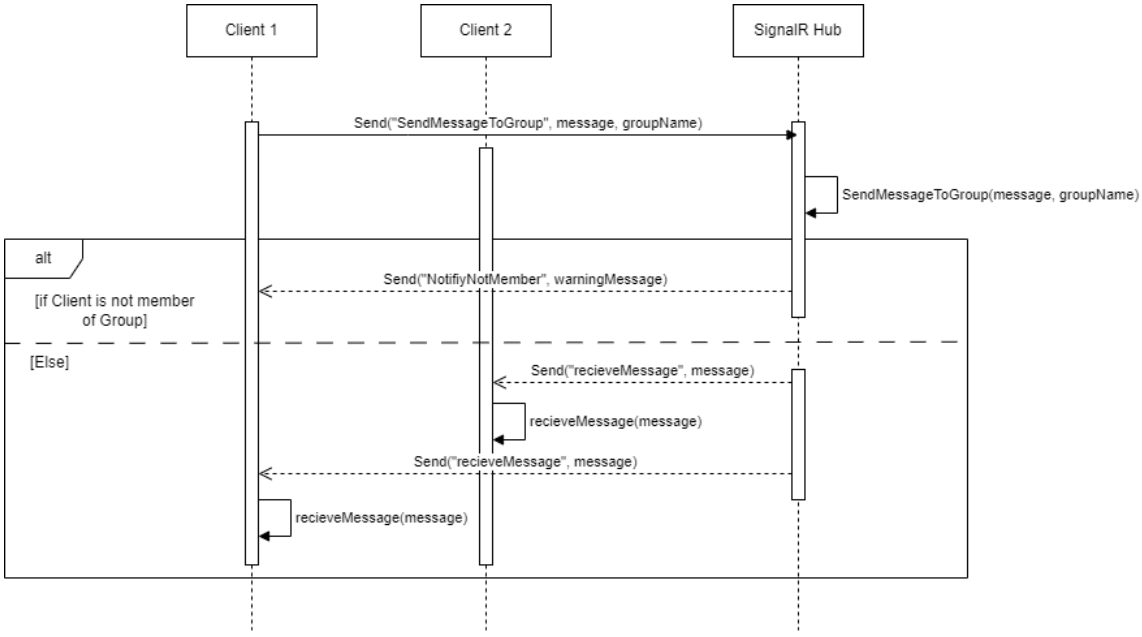

Abbildung 8: Sequenzdiagramm Chat mit Zugehörigkeitsüberprüfung[19]

Abbildung 8 ist eine Anpassung, beziehungsweise Erweiterung des in Abbildung 1 zu sehenden Sequenzdiagramms. Es wurde eine Überprüfung eingefügt, die

[19] Eigene Darstellung

Nachrichten nur von Gruppenmitgliedern zulässt. Wenn ein Client eine Nachricht an eine Gruppe schicken möchte, also bei den Mitgliederclients einen Funktionsaufruf auslösen möchte, wird serverseitig im SignalR Hub, mithilfe der Klasse ConnectionMapping aus Abbildung 6 überprüft, ob der Client selbst Gruppenmitglied ist. Nur wenn das der Fall ist, wird bei den Gruppenclients tatsächlich die Funktion aufgerufen, die Nachricht anzuzeigen. Wenn der Client kein Mitglied ist, wird nur eine Warnung an den Aufrufer selbst geschickt, dass nur Gruppenmitglieder die Rechte haben Nachrichten an die Gruppe zu schicken.

3.2.4 Spielzüge

In diesem Abschnitt wird das Konzept für den Spielablauf und die Spielzüge ausgearbeitet. Nachdem die Spieler einem Raum beigetreten sind und das Spiel gestartet ist, sollen neben der Möglichkeit zu chatten auch die Spielzüge in Echtzeit für beide spielenden Clients ausgeführt und dem jeweils gegnerischen Client angezeigt werden. Dabei kann grundsätzliche die Idee der Chatfunktion genutzt werden, die ja bereits Updates an der Oberfläche vornimmt und sicherstellt, dass diese nur von Gruppenmitgliedern an andere Gruppenmitglieder geschickt werden können. Allerdings muss zusätzlich gewährleistet werden, dass die Züge nur abwechselnd ausgeführt werden können. Das bedeutet, dass auch nur der Spieler, der am Zug ist, einen Funktionsaufruf bei dem anderen Client auslösen darf. Dies soll durch das Match Objekt sichergestellt werden. Dieses hat den aktuellen Stand des Spiels und merkt sich nach jedem Zug, welcher Spieler als Nächstes seinen Spielzug ausführen muss.

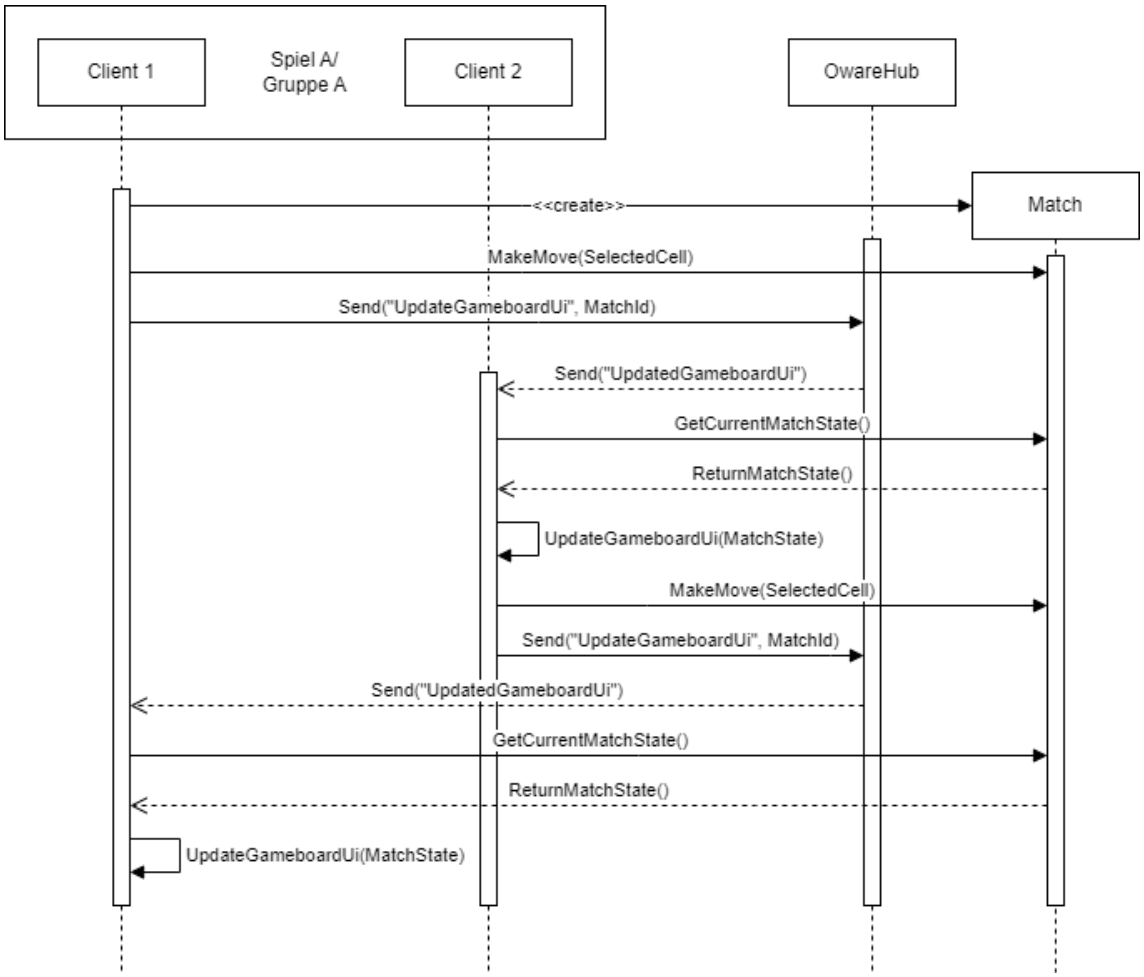

Abbildung 9: Sequenzdiagramm Spielerstellung und Spielzüge[20]

In Abbildung 9 ist die Erstellung eines Match Objekts beim Spielstart sowie die Abfolge der Methodenaufrufe bei zwei Spielzügen zu sehen. Der Client, der das Spiel erstellt, legt auch fest, welcher Client startet. In diesem Fall startet Client 1. Das Match Objekt nimmt nur Züge von Clients entgegen, die am Zug sind, alle anderen Aufrufe werden ignoriert. Sobald ein Client eine Zelle auf dem Spielfeld auswählt, wird das Match Objekt entsprechend geupdatet. Es wird für jede Zelle die neue Anzahl an Steinen berechnet. Nachdem der Zug ausgeführt wurde, wird außerdem ein Methodenaufruf an das SignalR Hub geschickt, um das Spielfeld des gegnerischen Clients zu updaten. Dieser fragt dann anhand der MatchId das Match Objekt an und lädt die Updates in seine Oberfläche. Nun ist Client 2 am Zug und die Abläufe wiederholen sich, bis das Spiel einen Endzustand erreicht. Diese sind entweder, ein Spieler hat gewonnen oder es gibt keine legalen Züge mehr. Wenn beide Clients, das Spiel vorzeitig verlassen, wird das Match Objekt verworfen.

[20] Eigene Darstellung

3.2.5 Identifizierung eines Benutzers nach Reconnect

Da keine Anmeldung oder Registrierung stattfinden soll und SignalR bei jedem Neuverbinden eines Clients eine neue Connection-ID vergibt, kann ein Client der kurzzeitig seine Verbindung zum SignalR Hub verliert, nicht zugeordnet werden. Das heißt, wenn der Client sich in einem Spiel befindet, muss das Spiel geschlossen werden, da nicht festgestellt werden kann, welcher Client im Spiel war. Da dies bei einem Neu laden des Browserfensters passiert oder bei kurzen Internetabbrüchen, und dies für die Benutzererfahrung störend ist, soll dieses Konzept eine Lösung dafür bieten. Dabei soll jedem Client, der sich zum ersten Mal mit dem SignalR Hub verbindet, nicht nur eine Connection-ID zugewiesen werden, sondern zusätzlich eine ID, welche lokal im Browser des Clients hinterlegt wird. Auf dem Server soll dann die lokale ID und die Connection-ID verknüpft abgelegt werden. Wenn ein Client dann beim Verbinden eine lokale ID mitschickt, kann überprüft werden, welche Connection-ID der Client vorher hatte und es zugeordnet werden, in welchem Spiel, beziehungsweise welcher Gruppe er Mitglied war. Da Spieleräume als SignalR Gruppen abgebildet werden, kann der Client dann direkt der entsprechenden SignalR Gruppe mit seiner neuen Connection-ID hinzugefügt werden und das Spiel weiterspielen. In Abbildung 10 ist der Ablauf für einen Verbindungsaufbau zwischen Client und Server mit einer lokalen ID zu sehen.

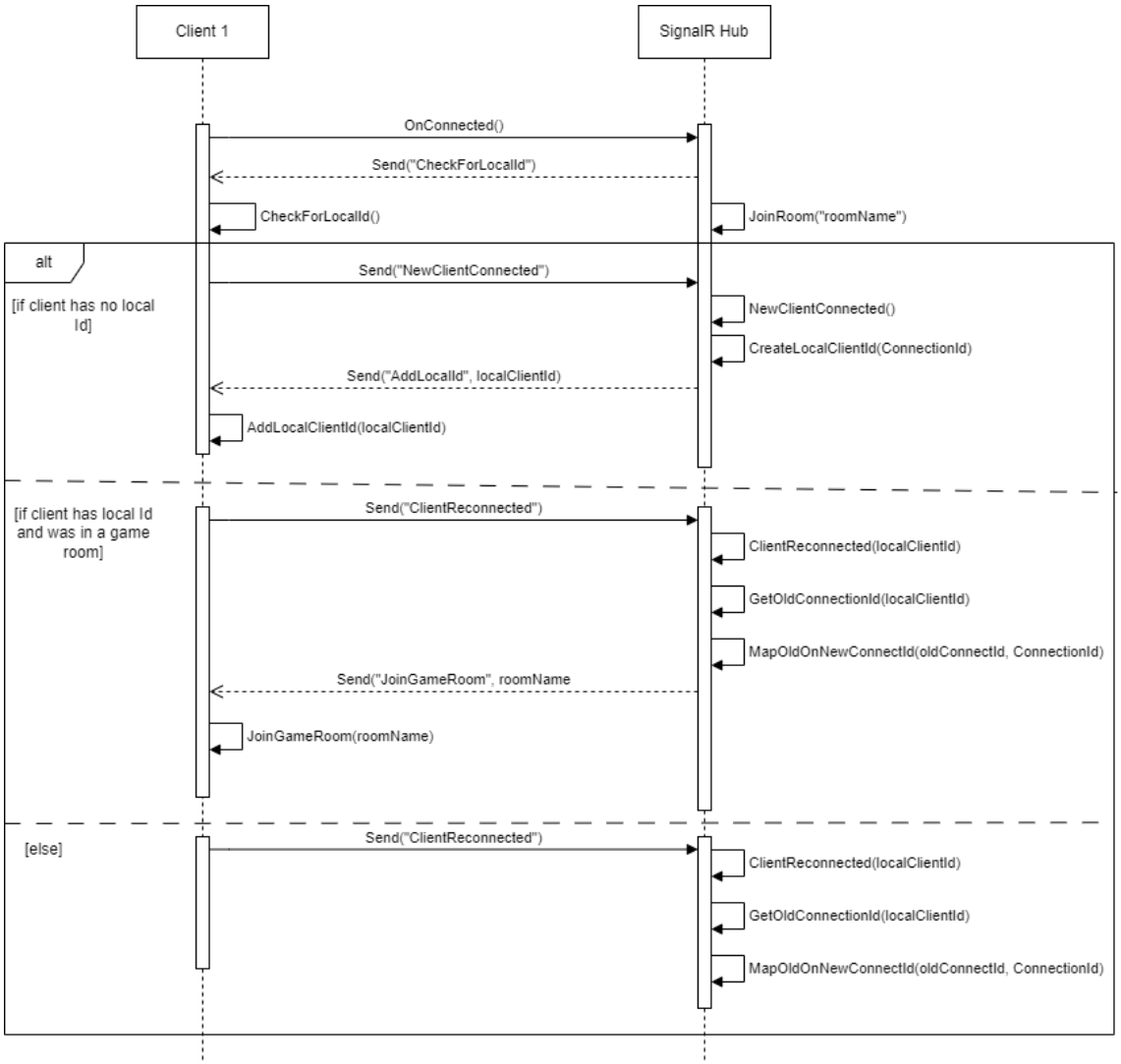

Abbildung 10: Sequenzdiagramm für Verbindungsaufbau mit lokaler ID[21]

Man kann dafür die Verbindungsverwaltung aus Kapitel 3.2.1 verwenden. Gruppenzugehörigkeiten sollen bei einem Verbindungsverlust eines Clients nicht einfach gelöscht werden, sondern in einem neuen Dictionary, zum Beispiel *Dictionary<string, HashSet<string>> oldGroups* gehalten werden. Außerdem soll dort die lokale ID vergeben werden und direkt der Connection-ID zugeordnet werden. Die Erweiterung, für die in 3.2.1 gezeigte Klassenstruktur, ist in Abbildung 11 zu sehen.

[21] Eigene Darstellung

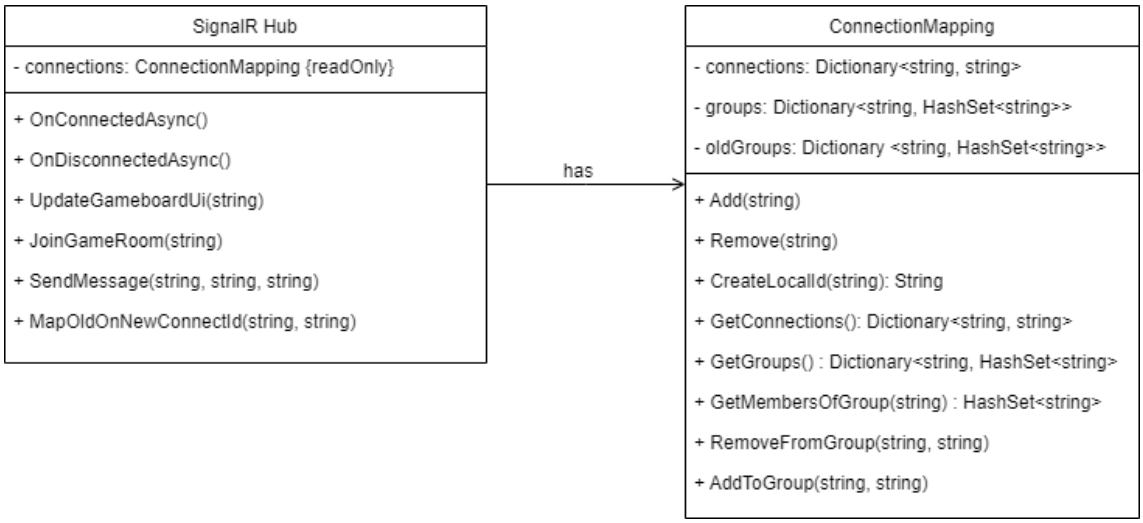

Abbildung 11: Neue Erweiterte Verbindungsverwaltung[22]

Die Methode *CreateLocalId()* generiert zum Beispiel eine GUID als localId oder beginnt mit 1 und zählt dann hoch für jede neu vergebene Id. Die Methode *MapOldOnNewConnectId()* soll mithilfe des ConnectionMapping Objekts und dem Dictionary *oldGroups*, welches ehemalige Gruppenmitgliedschaften von Clients anhand deren lokalen IDs und dem Gruppennamen hält, Clients wieder ihren Gruppen zuordnen. Allerdings sollen nur Gruppenmitgliedschaften über einen Verbindungsabbruch hinaus gehalten werden, wenn die Gruppe ein Spiel abbildet, das nicht regelkonform beendet wurde, und die Gruppe mindestens noch ein anderes verbundenes Mitglied hat. Dies soll vermeiden, dass die Datenmenge zu groß wird. Wenn beide Clients im Spiel die Verbindung verlieren, kann davon ausgegangen werden, dass kein Interesse mehr besteht, das Spiel zu beenden, und es ist kein Merken dieser Gruppenmitgliedschaften notwendig.

[22] Eigene Darstellung

3.3 Modellierung der Oberfläche

Zusätzlich zu dem Aufbau und den Abläufen der Software, soll die Oberfläche modelliert werden. Es werden die, von den in Kapitel 3.2 beschriebenen Abläufe betroffenen Bereiche der Oberfläche als Mockup designt und kurz erklärt. Zur Erstellung des Mockups wird das webbasierte Werkzeug Balsamiq verwendet. Die erste Aktion, die Benutzer bei dem Besuchen der Webseite vornehmen können sollen, ist die Auswahl, ob sie gegen einen Bot oder gegen einen anderen Benutzer spielen möchten.

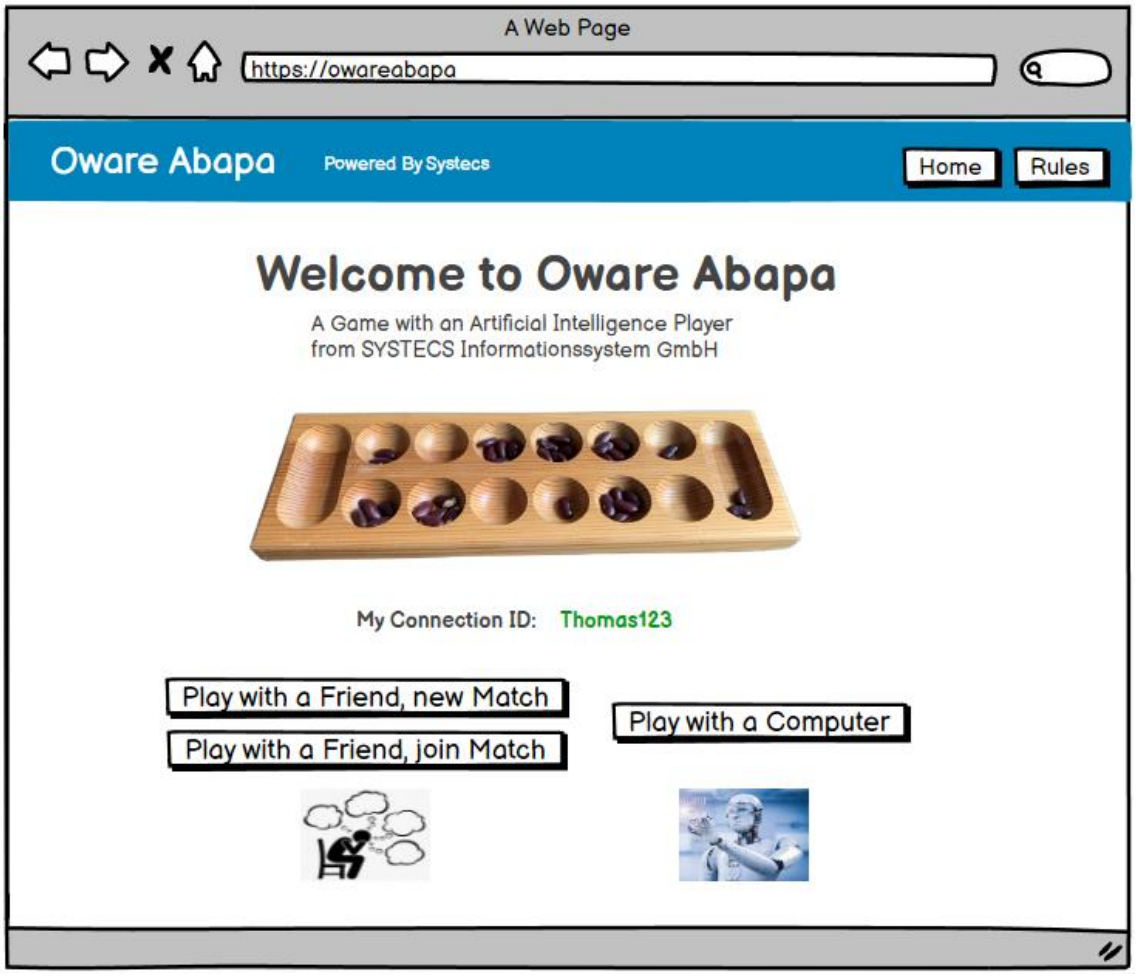

Abbildung 12: Startseite der Oware Webanwendung[23]

In Abbildung 12 ist das Mockup für die Startseite zu sehen. Hier stehen dem Benutzer die Möglichkeiten zur Auswahl für ein Spiel gegen einen Freund, beziehungsweise einen anderen menschlichen Benutzer oder für ein Spiel gegen einen Computer, beziehungsweise einen Bot zur Verfügung. Da das Spiel gegen den Bot nicht Teil der Arbeit ist, wird der Ablauf, der bei der Auswahl dieser Möglichkeit folgt, nicht genauer betrachtet, sondern es wird der Fokus auf den Ablauf für ein Spiel zwischen zwei Menschen gelegt. Bei dem Spiel gegen einen

[23] Eigene Darstellung

Menschen kann der Benutzer außerdem auswählen, ob er ein neues Spiel erstellen oder einem bereits erstellten Spiel beitreten möchte.

3.3.1 Match Making

Wenn ein Benutzer auf der Startseite auswählt, dass er ein Spiel gegen einen Menschen erstellen möchte, wird er zunächst auf eine Oberfläche weitergeleitet, in der er grundlegende Einstellungen für das zu erstellende Spiel festlegen kann.

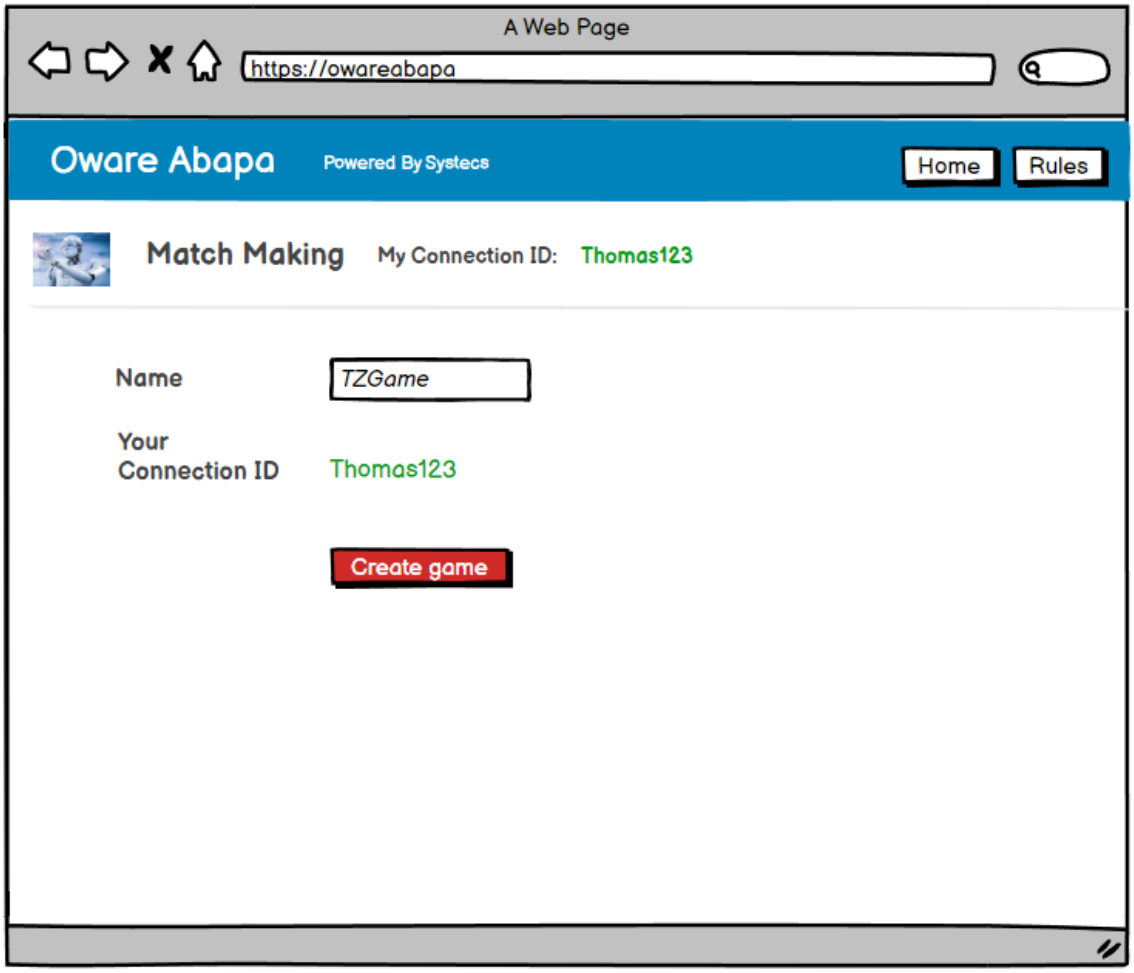

Abbildung 13: Erstellung eines Spiels[24]

Diese Ansicht ist in Abbildung 13 zu sehen. Der Benutzer wird aufgefordert einen Namen für das Spiel einzugeben, der dann auch im Backend der Anwendung für die Erstellung der SignalR Gruppe verwendet wird. Sobald der Button „Create game“ geklickt wird, wird die SignalR Gruppe erstellt und das Spiel taucht in einer Übersicht, wie in Abbildung 14 zu sehen, auf.

[24] Eigene Darstellung

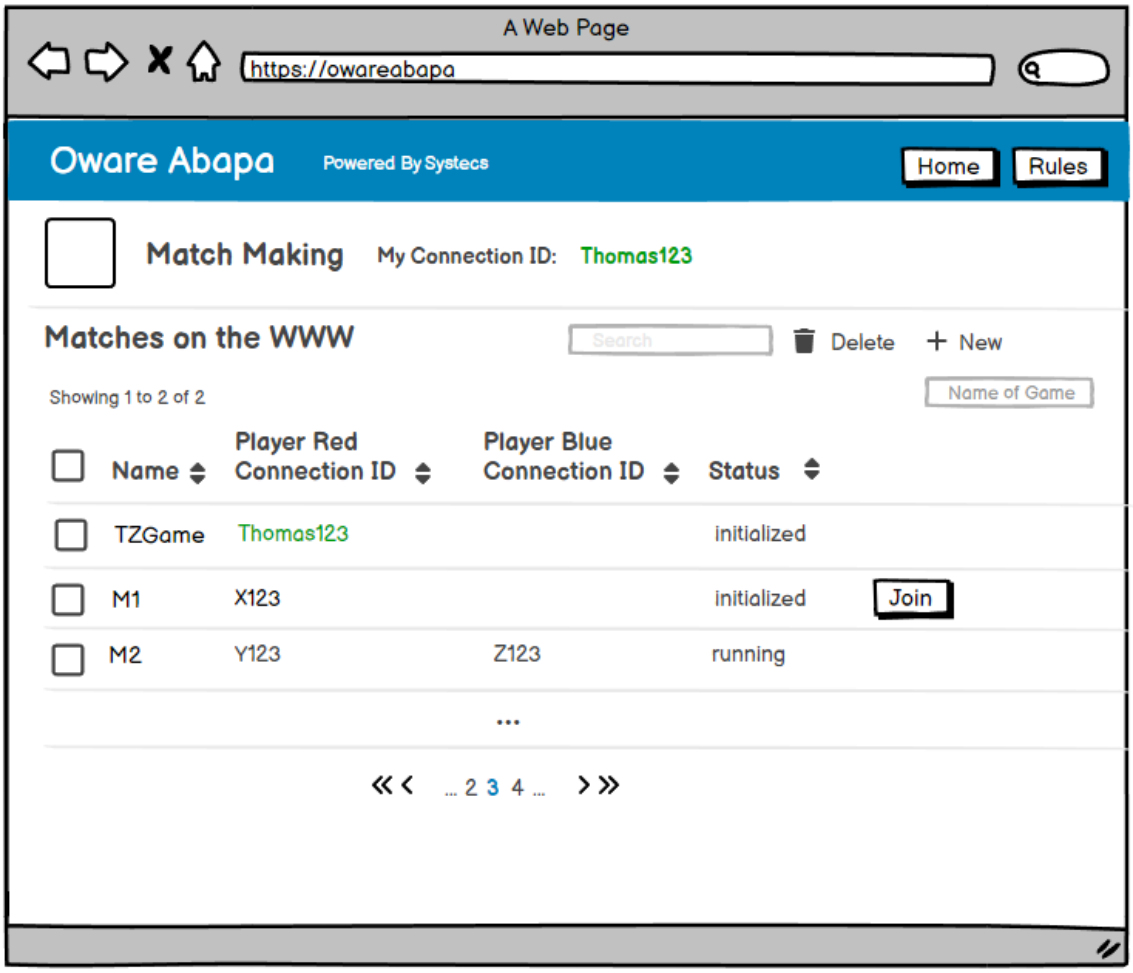

Abbildung 14: Übersicht über alle erstellten und laufenden Spiele[25]

Benutzer, die kein Spiel erstellen, sondern einem bereits erstellen Spiel beitreten möchten, werden direkt auf die in Abbildung 14 gezeigte Oberfläche weitergeleitet. Diese Übersicht wird für alle verbundenen Benutzer in Echtzeit aktualisiert, sobald ein Spiel erstellt, geschlossen oder dessen Status geändert wird. Zum Beispiel, wenn ein Spiel gestartet wird oder ein Benutzer beitritt. Sobald einem Spielraum zwei Spieler beigetreten sind, kann das Spiel gestartet werden und beide Spieler werden zu einer neuen Ansicht weitergeleitet, auf der das Spielbrett zu sehen ist.

3.3.2 Spielbrett

Das Spielbrett ist in Abbildung 15 abgebildet. Zusätzlich zu dem Spielbrett mit den einzelnen Zellen und der sich darin befindlichen Anzahl von Bohnen ist rechts das Chatfenster zu sehen, über das die beiden Spieler sich austauschen können. Links und rechts des Spielbretts werden die bereits erspielten Punkte der Spieler angezeigt. Außerdem gibt es einen visuellen Hinweis, welcher Spieler

[25] Eigene Darstellung

am Zug ist. In dem Mockup aus Abbildung 15 wird dies durch den Text „to play“ auf der Seite des sich am Zug befindlichen Spielers umgesetzt.

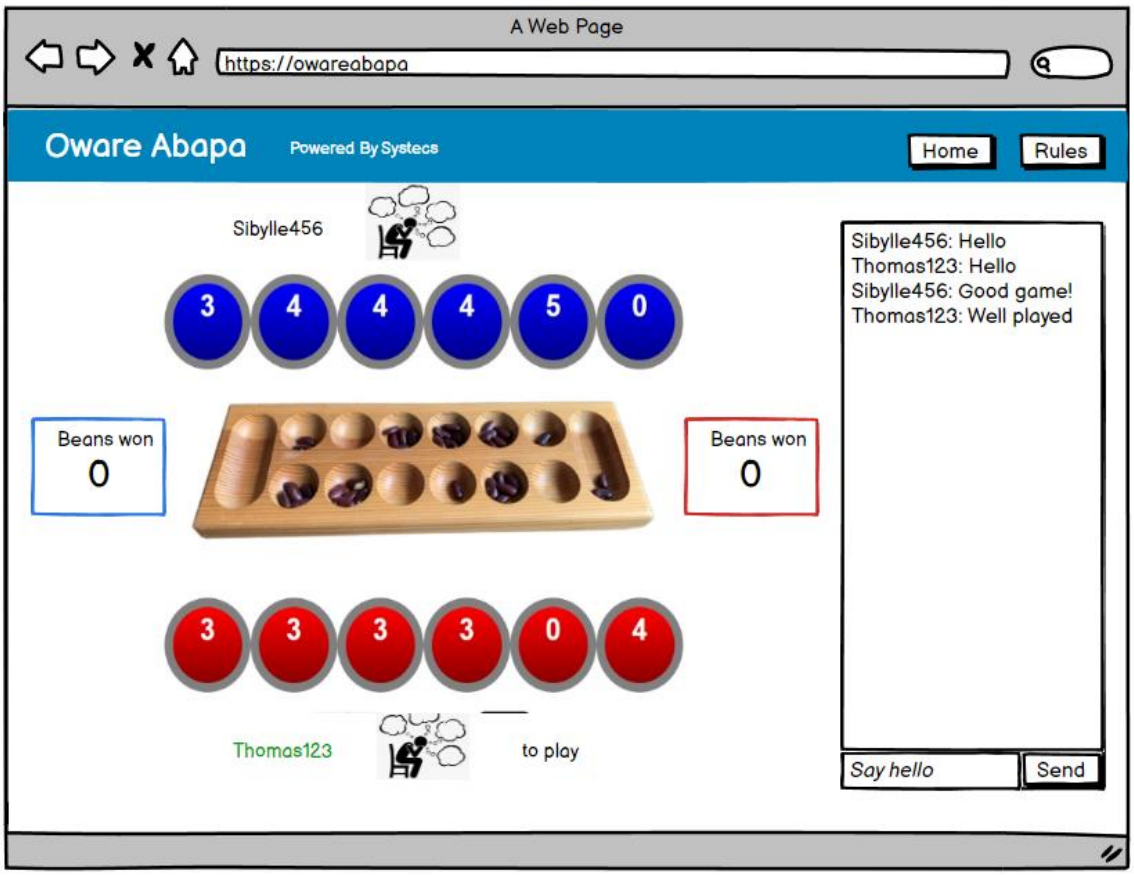

Abbildung 15: Spielbrett mit Chatbox[26]

Für ein Spiel gegen einen Computer sind die Oberflächen leicht angepasst. Bei der Spielerstellung muss noch ein Schwierigkeitsgrad eingestellt werden. Außerdem taucht das Spiel nicht in der Oberfläche auf, da es keinen zweiten Spieler benötigt, der dem Spiel beitritt. Der letzte Unterschied ist die fehlende Chatbox auf der Ansicht mit dem Spielbrett.

[26] Eigene Darstellung

4 Implementierung

In diesem Kapitel wird die Implementierung des Konzepts dokumentiert und beschrieben. Da zuerst ein Teil des Konzeptes in einer Demo Chat Anwendung programmiert wurde und später der Code für das Spiel wiederverwendet, angepasst oder erweitert wurde, gibt es zwei Implementierungen. In der ersten Implementierung der Echtzeitfunktionalitäten in einer Chatanwendung wurde das komplette ConnectionMapping entwickelt, welches später genauso für das Spiel übernommen werden konnte. Außerdem wurde die Chatfunktion sowie das Match Making implementiert. Es fehlt der Spielablauf. Außer dem Match Making wurden alle Konzepte in der Implementierung des Spiels umgesetzt und anhand dieser auch im Folgenden beschrieben. Um dennoch einen Proof of Concept für das Match Making Konzept vorzulegen, wird dessen Implementierung anhand der Demo Chat Anwendung gezeigt. Außerdem wird zu Beginn kurz beschrieben, wie die SignalR Bibliothek in eine Blazor Anwendung eingebunden wird.

4.1 Einbindung der SignalR Bibliothek

Um die SignalR Bibliothek in ein Projekt einzubinden, muss über den NuGet Package Manager die SignalR Client Bibliothek hinzugefügt werden. Die Referenzen tauchen dann in der csproj-Datei des Projekts auf, dort kann auch die Version eingesehen und verwaltet werden. Die Referenzen sind in Codebeispiel 4 zu sehen.

```
<PackageReference Include="Microsoft.AspNetCore.SignalR.Client" Version="7.0.8"
/>

    <PackageReference Include="Microsoft.Azure.SignalR" Version="1.21.4" />
```

Codebeispiel 4: SignalR Packages in der csproj-Datei[27]

Zusätzlich muss eine SignalR Hub Klasse angelegt werden, welche die serverseitige Funktionalität enthält. Sie implementiert die Basisklasse Hub.

```
public class OwareHub : Hub
```

Codebeispiel 5: Klassenkopf der OwareHub Klasse[28]

In dieser Klasse, deren Kopf in Codebeispiel 5 zu sehen ist, sind alle Methoden zu finden, die von Clients aufgerufen werden können und dann Remote Procedure Calls an bestimmte Clients schicken. Für diese Anwendung heißt die Klasse OwareHub, zur besseren Verständlichkeit wird in diesem Kapitel aber

[27] Eigener Code
[28] Eigener Code

trotzdem von der SignalR Hub Klasse gesprochen. Zuletzt müssen in der Program.cs Datei der SignalR Service hinzugefügt, sowie Endpunkte definiert werden, damit der Client weiß, an welchen Hub er seine Methodenaufrufe schicken muss. In Codebeispiel 6 sind die beiden Zeilen abgebildet, die dies sicherstellen. Sie befinden sich beide an verschiedenen Stellen innerhalb der Program.cs Datei.

```
services.AddSignalR();


endpoints.MapHub<OwareHub>("/owareHub");
```

Codebeispiel 6: Hinzufügen des SignalR Service sowie Definition der Endpoints[29]

4.2 Verbindungsverwaltung

Im Konzept, Kapitel 3.2.1 über die Verbindungsverwaltung wird beschrieben, dass die SignalR Verbindungsverwaltung erweitert werden muss, um benötigte Funktionen zu ermöglichen. Zum Beispiel Mitglieder einer SignalR Gruppe zu verwalten, eine maximale Mitgliederanzahl zu garantieren oder einzelne Mitglieder identifizieren zu können. Die Implementierung dieser Erweiterung wird in den folgenden Unterkapiteln dokumentiert.

4.2.1 Klasse ConnectionMapping

```
public class ConnectionMapping<T>
        {
            private readonly List<string> _connections = new ();

            private readonly Dictionary<string, HashSet<string>> _groups = new();
```

Codebeispiel 7: Klasse ConnectionMapping mit Attributen[30]

Es wurde zuerst die Klasse ConnectionMapping implementiert, die mit den Attributen *_connections* und *_groups* in Codebeispiel 7 zu sehen ist. Das Attribut *_connections* soll die SignalR Connection-IDs halten und *_groups* die SignalR Gruppennamen sowie die Connection-IDs der Mitglieder. Die Grundlage und Grundidee dieser Klasse werden, wie im Konzept bereits beschrieben, aus der Microsoft Dokumentation übernommen und angepasst. Um eine Connection-ID hinzuzufügen, gibt es die Methode *Add(string connectionId)*.

[29] Eigener Code
[30] Eigener Code

```
public void Add(string connectionId)
            {
                lock (_connections)
                {
                    if (!_connections.Contains(connectionId))
                    {
                        _connections.Add(connectionId);
                    }

                }

            }
```

Codebeispiel 8: *Add()*-Methode zum Speichern von Connection-IDs[31]

In Codebeispiel 8 ist diese Methode zu sehen. Um sicherzustellen, dass nur ein Client gleichzeitig auf das *_connections* Attribut zugreift, wird dieses durch das *lock*-Statement gesperrt. Es wird zusätzlich überprüft, ob sich die Connection-ID bereits in der Connection Liste befindet. Wenn dies nicht der Fall ist, kann die Connection-ID hinzugefügt werden. Um eine Connection-ID wieder zu entfernen, wenn ein Client seine Verbindung trennt, gibt es die *Remove(string connectionId)*-Methode.

```
public void Remove(string connectionId)
            {
                foreach (var group in _groups.Keys)
                {
                    RemoveGroup(group, connectionId);
                }
                lock (_connections)
                {
                    if (!_connections.Contains(connectionId))
                    {
                        return;
                    }

                    _connections.Remove(connectionId);

                }

            }
```

Codebeispiel 9: *Remove()*-Methode zum Entfernen von Connection-IDs[32]

Diese Methode, die in Codebeispiel 9 abgebildet ist, entfernt die Connection-ID zuerst aus dem *_groups* Attribut, falls der Client Mitglied einer oder mehrerer Gruppen war. Danach wird wie bei der *Add()*-Methode das *_connections* Attribut gesperrt und dann die ID, falls es diese überhaupt gibt, entfernt. Die weitere Implementierung ist der Hauptzweck der Klasse ConnectionMapping, nämlich das Verwalten der Gruppen und den Mitgliedern. Dafür gibt es wieder eine Methode zum Hinzufügen und eine zum Entfernen einer Connection-ID zu einer Gruppe.

[31] Eigener Code
[32] Eigener Code

```
public void AddGroup(string groupName, string connectionId)
            {
                lock (_groups)
                {
                    HashSet<string> members;
                    if (!_groups.TryGetValue(groupName, out members))
                    {
                        members = new HashSet<string>();
                        _groups.Add(groupName, members);
                    }

                    lock (members)
                    {
                        members.Add(connectionId);
                    }
                }

            }
```

Codebeispiel 10: *AddGroup()*-Methode zum Zuordnen von Connection-ID und Gruppenname[33]

In Codebeispiel 10 ist die Methode zum Hinzufügen einer ID und dem Gruppenamen zu dem *_groups* Attribut zu sehen. Zuerst wird wieder das Attribut, das verändert wird, gesperrt, um den gleichzeitigen Zugriff von mehreren Clients zu verhindern. Danach wird in dem Dictionary *_groups* nach dem Schlüssel, also dem Gruppennamen gesucht. Falls es für den Schlüssel *groupName* keinen Eintrag gibt, wird einer neuer Eintrag mit dem Gruppenname erstellt. Dies ist der Fall, wenn der Client, der erste ist, der zu der Gruppe hinzugefügt werden soll, da eine Gruppe erst mit dem ersten Mitglied erstellt wird und genauso mit dem Entfernen des letzten Mitglieds gelöscht wird. Wenn bereits ein Eintrag im Dictionary für die Gruppe existiert, wird dieser Schritt übersprungen. In jedem Fall wird dann dem entsprechenden HashSet welches der Value Teil für einen Eintrag im Dictionary ist und die Connection-IDs der Gruppenmitglieder hält, die neue Connection-ID hinzugefügt. Das heißt, die Methode fügt entweder nur die Connection-ID zu einer bestehenden Gruppe hinzu oder erstellt zuerst die Gruppe, falls diese nicht existiert hat, und fügt dann direkt die Connection-ID hinzu.

[33] Eigener Code

```
public void RemoveGroup(string groupName, string connectionId)
            {
                lock (_groups)
                {
                    HashSet<string> members;
                    if (!_groups.TryGetValue(groupName, out members))
                    {
                        return;
                    }

                    lock (members)
                    {
                        members.Remove(connectionId);

                        if (members.Count == 0)
                        {
                            _groups.Remove(groupName);
                        }
                    }
                }

            }
```

Codebeispiel 11: *RemoveGroup()*-Methode zum Entfernen von Connection-IDs aus einer Gruppe[34]

Das Gegenstück zur *AddGroup()* – Methode ist die *RemoveGroup()* – Methode, mit der eine Connection-ID aus dem entsprechenden Key-Value Paar entfernt werden kann. Diese ist in Codebeispiel 11 abgebildet. Es wird wieder zuerst überprüft, ob es überhaupt einen Eintrag für den Key *groupName* gibt. Wenn es einen Eintrag gibt, wird der Value Teil, also die *members* geladen und die Connection-ID, die der Methode als Parameter übergeben wurde, entfernt. Wenn es keinen Eintrag wird, wird der Methodenaufruf direkt beendet. Falls die ID jedoch entfernt wurde, wird außerdem geprüft, ob dies die letzte Mitglieder-ID war. Dann wird der komplette Eintrag aus dem Dictionary entfernt, da auch die SignalR Gruppe automatisch gelöscht wird, sobald der letzte Client die Gruppe verlässt. Die letzte Methode, die implementiert wurde, ist die, zum Abrufen aller Mitglieder einer Gruppe, beziehungsweise deren Connection-IDs.

[34] Eigener Code

```
public HashSet<string> GetMembersOfGroup(string group)
            {
                HashSet<string> members;
                if (_groups.TryGetValue(group, out members))
                {
                    return members;
                }
                return members;

            }
```

Codebeispiel 12: Methode zum Abrufen aller Connection-IDs einer Gruppe[35]

Diese Methode ist in Codebeispiel 12 zu sehen. Wenn es für den Schlüssel *group*, also dem Gruppennamen einen Eintrag im *_groups* Dictionary gibt, wird der entsprechende Value Eintrag mit den Mitglieder IDs zurückgegeben. Wenn kein Eintrag für den Key gefunden wird, wird ein leeres HashSet zurückgegeben.

4.2.2 ConnectionMapping im SignalR Hub

Die Methoden der Klasse ConnectionMapping werden im SignalR Hub verwendet, um dort Teile der im Konzept aufgezählten Anforderungen sicherzustellen. Dafür gibt es in der SignalR Hub Klasse ein Objekt der Klasse ConnectionMapping, siehe Codebeispiel 13.

```
private static readonly ConnectionMapping<string> _connections = new();
```

Codebeispiel 13: ConnectionMapping Objekt in der SignalR Hub Klasse[36]

Bei jeder neuen Verbindung eines Clients mit dem SignalR Hub beziehungsweise dem Server wird das ConnectionMapping Objekt verwendet, um die von SignalR vergebene Connection-ID zu hinterlegen. Dafür wird die in Codebeispiel 8 gezeigte Methode innerhalb der *OnConnectedAsync()*-Methode aufgerufen.

```
public override async Task OnConnectedAsync()
        {
            _connections.Add(Context.ConnectionId);
            await Clients.Caller.SendAsync("SetConnectionId",
Context.ConnectionId);
            await base.OnConnectedAsync();

        }
```

Codebeispiel 14: Hinzufügen eines neuen Clients zum ConnectionMapping Objekt[37]

[35] Eigener Code
[36] Eigener Code
[37] Eigener Code

Im Codebeispiel 14 ist die *OnConnectedAsync()*-Methode aus dem SignalR Hub abgebildet. Zuerst wird die bereits erwähnte *Add()*-Methode mit der Connection-ID als Parameter aufgerufen. Außerdem wird dem neu verbundenen Client seine Connection-ID zurückgeschickt. Genauso wird die Connection-ID auch wieder entfernt, wenn ein Client seine Verbindung trennt.

```
public override async Task OnDisconnectedAsync(Exception? exception)
        {
            _connections.Remove(Context.ConnectionId);
            await base.OnDisconnectedAsync(exception);

        }
```

Codebeispiel 15: Entfernen einer Connection-ID bei Trennung der Verbindung[38]

In Codebeispiel 15 ist die *OnDisconnected()*-Methode dargestellt, die immer aufgerufen wird, wenn eine Clientverbindung getrennt wird. Innerhalb der Methode wird die *Remove()*-Methode aus Codebeispiel 9 aufgerufen, welche die Connection-ID wieder aus der erweiterten Verbindungsverwaltung entfernt. Des Weiteren ist die Verbindungsverwaltung, beziehungsweise das ConnectionMapping Objekt noch dafür zuständig sicherzustellen, dass eine Gruppe maximal zwei Mitglieder hat.

```
public async Task JoinGameRoom(string gameId)
        {
            var groups = _connections.GetGroups();

            if (groups.ContainsKey(gameId))
            {
                if (groups[gameId].Count >= 2)
                {
                    return;
                }
            }
            _connections.AddGroup(gameId, Context.ConnectionId);
            await Groups.AddToGroupAsync(Context.ConnectionId, gameId);


        }
```

Codebeispiel 16: Spielbeitritt mit Überprüfung, ob bereits 2 Mitglieder in der Gruppe sind[39]

In Codebeispiel 16 ist die Methode *JoinGameRoom()* abgebildet, die aufgerufen wird, wenn ein Client einem Spiel beitreten möchte oder ein neues Spiel erstellt. Dabei werden zuerst alle Key-Value Paare aus dem ConnectionMapping Objekt geholt, welche jeweils einen Gruppennamen, sowie die zugehörigen Mitglieder Connection-IDs enthalten. Dann wird überprüft ob, für den als Parameter

[38] Eigener Code

[39] Eigener Code

übergebenen Gruppenname, zwei oder mehr Values, also Mitglieder vorliegen. Nur wenn dies nicht der Fall ist, wird die Connection-ID den Values hinzugefügt.

4.3 Match Making

Wie bereits in der Einleitung zu diesem Kapitel erwähnt, wird die Implementierung des Match Making anhand einer anderen Anwendung als dem Spiel erklärt, da es nicht in das Spiel übernommen wurde. Dass das Match Making Konzept dennoch umsetzbar ist, soll in diesem Kapitel aufgezeigt werden. Im SignalR Kontext werden Gruppen verwendet, um die Chaträume abzubilden. In der Oberfläche wird die Bezeichnung Raum verwendet. Die Begriffe Raum und Gruppe sind bedeutungsgleich. Verständnis halber wird in diesem Kapitel immer von einem Raum oder Räumen gesprochen, auch wenn an manchen Stellen eine SignalR Gruppe gemeint ist.

4.3.1 Übersicht aller Räume

Zuerst muss dafür die *OnConnectedAsync()*-Methode erweitert werden, um jedem neu verbundenen Client eine aktuelle Übersicht der Räume zu schicken, die in der Verbindungsverwaltung hinterlegt sind.

```
await Clients.Caller.SendAsync("GroupsUpdated", _connections.GetGroups());
```

Codebeispiel 17: Aufruf an Client seine Raumübersicht upzudaten[40]

In Codebeispiel 17 ist die Codezeile abgebildet, mit der dies sichergestellt wird. Der Code schickt dem Aufrufer der *OnConnectedAsync()*-Methode, also dem gerade neu verbundenen Client, einen Aufruf, die *GroupsUpdated()*-Methode bei sich aufzurufen und gibt als Parameter die Räume aus dem ConnectionMapping Objekt *_connections* mit. Im Frontend, also clientseitig gibt es einen Event Handler, der dann eine Übersicht in der Benutzeroberfläche aktualisiert.

```
hubConnection.On<Dictionary<string, HashSet<string>>>("GroupsUpdated",
(updatedGroupList) =>
        {
            rooms = updatedGroupList;
            InvokeAsync(StateHasChanged);

        });
```

Codebeispiel 18: Clientseitiger Event Handler zum Updaten der Gruppenübersicht[41]

[40] Eigener Code

[41] Eigener Code

Der Event Handler aus Codebeispiel 18 wird dann aufgerufen und weist der Variablen *rooms* den Wert des Parameters aus Codebeispiel 17 zu. Außerdem wird mit der Methode *InvokeAsync(StateHasChanged)* die Blazor Komponente, welche die Übersicht für die Räume enthält, neu gerendert, um die Änderungen zu übernehmen. Der CSHTML Code für die Übersicht ist in Codebeispiel 19 zu sehen, dort wird eine Auflistung erstellt, die für jedes Listenelement der *rooms* Variablen einen neuen Tabelleneintrag erstellt, der den Raumnamen sowie die aktuelle Anzahl der Mitglieder enthält. Außerdem wird ein Button erstellt, über welchen dem entsprechenden Raum beigetreten werden kann.

```
<tbody>
        @foreach(var room in rooms)
    {
        <tr>
            <td>@room.Key</td>
            <td>(@room.Value.Count / 2)</td>
            <td><button class="@room.Key" @onclick="() => Join(room.Key)"
disabled="@(IsConnected == false || room.Value.Count >= 2)">Join
Room</button></td>
        </tr>
    }

    </tbody>
```

Codebeispiel 19: CSHTML für die Raumübersicht[42]

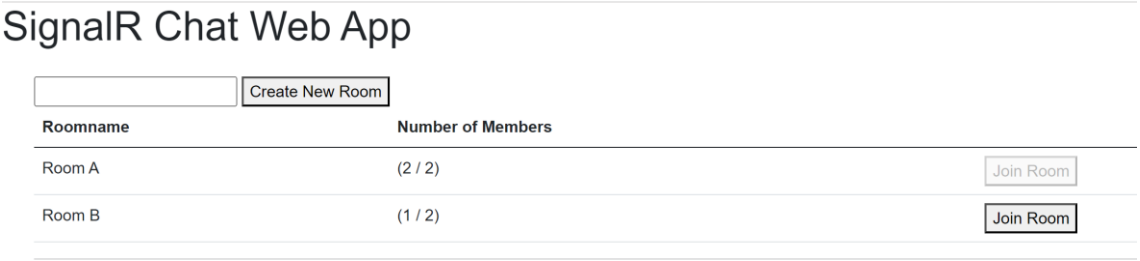

Abbildung 16: Raumübersicht in der Demo Chat Anwendung[43]

In Abbildung 16 ist diese Übersicht in der Anwendung zu sehen. Es wurden beispielhaft zwei Räume erstellt. Der Button für den Room A ist ausgegraut, da hier bereits zwei Benutzer Mitglied sind. Die Überprüfung, ob ein Benutzer beitreten darf, erfolgt, wie in Codebeispiel 16 zu sehen, auch im Backend.

4.3.2 Räumen beitreten

Wenn ein Raum mit einem Mitglied in der Übersicht aufgelistet ist, kann über den Button „Join Room“ dem Raum beigetreten werden. Wenn ein Client diesen Button klickt, wird wie, in Codebeispiel 19 zu sehen, die Methode *Join()* aufgerufen.

[42] Eigener Code
[43] Eigene Abbildung

```
private async Task Join(string room)
    {
        if(hubConnection is not null)
        {
            await hubConnection.SendAsync("RemoveFromGroup", currentRoom);
            await hubConnection.SendAsync("AddToGroup", room);

            currentRoom = room;
        }

        _navigationManager.NavigateTo($"/chatroom/{room}");

    }
```

Codebeispiel 20: *Join()*-Methode, um einem Raum beizutreten[44]

Die Methode aus Codebeispiel 20 schickt zwei Remote Procedure Calls an den Server, beziehungsweise das SignalR Hub. Zuerst wird der Client aus seinem bisherigen Raum entfernt, falls er noch Mitglied eines Raumes war. Danach wird er der neuen Gruppe hinzugefügt. Die beiden Methoden *RemoveFromGroup()* und *AddToGroup()*, die dabei serverseitig aufgerufen werden, sind fast die gleichen aus den Codebeispielen 10 und 11 mit einer Erweiterung. Zusätzlich zu dem, was in diesen Codebeispielen zu sehen ist, werden sie durch eine Zeile, zu sehen in Codebeispiel 21, erweitert. Diese Codezeile schickt ein Update an alle verbundenen Clients mit den Räumen und deren neuen Mitgliederzahlen. So kann sichergestellt werden, dass jeder Client immer eine aktuelle Raumübersicht hat.

```
await Clients.All.SendAsync("GroupsUpdated", _connections.GetGroups());
```

Codebeispiel 21: Aufruf an alle Clients die Raumübersicht upzudaten[45]

4.3.3 Räume erstellen

Über das Textfeld in Abbildung 16 und den Button „Create New Room“ lassen sich neue Räume erstellen. Das Erstellen und Beitreten eines Raumes wird im SignalR Hub mit denselben Methoden umgesetzt. Nur im Frontend gibt es eine weitere Überprüfung. Wenn ein Client den Button „Create New Room“ klickt, wird die Methode *CreateRoom(string roomName)* aufgerufen, wobei der String *roomName* den Wert des Textfelds neben dem Button zum Zeitpunkt des Klicks enthält. Wie in Codebeispiel 19 zu sehen, wird dabei zuerst überprüft, ob bereits ein Raum mit dem eingegebenen Namen existiert. Nur wenn dies nicht der Fall ist, wird beim Beitreten zu einem Raum die *Join()*-Methode aufgerufen und der restliche Ablauf ist derselbe wie in Kapitel 4.3.2 beschrieben.

[44] Eigener Code
[45] Eigener Code

```
private async Task CreateRoom(string roomName)
    {
        if (!rooms.ContainsKey(roomName))
        {
            await Join(roomName);
        }
        else
        {
            roomExists = true;
            await InvokeAsync(StateHasChanged);
        }

    }
```

Codebeispiel 22: Methode zum Erstellen eines neuen Raumes[46]

4.4 Chatfunktion

Die Chatfunktion wurde wie geplant und konzipiert in der Spielanwendung implementiert. Die verschiedenen Methodenaufrufe, die im Konzeptteil in Abbildung 8 als Sequenzdiagramm modelliert sind, werden von der Logik genauso übernommen, jedoch gibt es zusätzlich zu Client und Server, beziehungsweise dem SignalR Hub zwei Objekte, die dazwischen liegen und die Kommunikation weiterleiten, ohne größeren Einfluss zu nehmen. Es werden lediglich Daten mitgegeben, die nur in diesen Objekten vorliegen. Diese Objekte sind zum einen eine OwareSession, in der die Daten für das Spiel, sowie über den Spielerclient gehalten werden, zum Beispiel die Spielerfarbe. Das andere Objekt ist ein OwareSessionHub, welches die clientseitige Schnittstelle zum Server, beziehungsweise SignalR Hub darstellt. Hier wird die Verbindung zum Hub initialisiert und es werden die Remote Procedure Calls an den SignalR Hub geschickt. Sowohl bei dem Empfangen als auch bei dem Senden von Nachrichten wird die Nachricht über Methoden dieser Objekte vom Front- bis in das Backend, beziehungsweise vom Back- in das Frontend geschickt. Die Implementierung im Frontend ist CSHTML Code, der ähnlich wie bei der Raumübersicht aus Kapitel 4.3.1 eine Liste von allen Nachrichten, die ausgetauscht wurden, in eine HTML-Liste schreibt. Dieser Code, in Codebeispiel 23 zu sehen, wird nur gerendert, wenn das Spiel ein Spiel zwischen zwei Menschen ist.

[46] Eigener Code

```
@if (Session.IsMatchHuman2Human()){
    <div class="container col-md-auto">
        <div class="row justify-content-md-center" style="border: black; border-
style: solid; margin: 50px; padding: 10px">
            <ul style="list-style: none">
                @foreach (string message in messages)
                {
                    <li>
                        @message
                    </li>
                }
            </ul>
            <div class="row gx-1">
                <div class="col p-1">
                    <input @bind="message" type="text" placeholder="Enter a chat
message..."/>
                </div>
                <div class="col p-1">
                    <button @onclick="SendMessage"> Send Message</button>
                </div>
            </div>
        </div>
    </div>

}
```

Codebeispiel 23: CSHTML Code für ein Chatfenster[47]

Es gibt ein Input Feld für den Textinput und einen Button über den die eingegebene Nachricht an die, sich im Raum befindlichen Clients geschickt werden kann. Der Button löst im C#-Teil der Razor Page die *SendMessage()*-Methode aus, die in Codebeispiel 24 abgebildet ist.

```
public async Task SendMessage()
        {
            if (message != null)
            {
                await Session.SendMessageToSessionHub(message);
            }
            message = null;

        }
```

Codebeispiel 24: *SendMessage()*-Methode in der Razor Page[48]

In dem Beispiel sieht man, dass nicht direkt ein Aufruf an den SignalR Hub geschickt wird, sondern wie am Anfang erwähnt, eine Methode des Session-Objekts aufgerufen wird, welche die Nachricht weiterleitet.

[47] Eigener Code
[48] Eigener Code

```
internal async Task SendMessageToSessionHub(string message)
        {
            if (MatchDto != null)
            {
                await _sessionHub.SendMessageToSignalRHub(message,
_myColor.ToString(), MatchDto.MatchDtoId.ToString());
            }

        }
```

Codebeispiel 25: Methode in der OwareSession Klasse zur Weiterleitung der Nachricht[49]

Die Methode des Session-Objekts in Codebeispiel 25 ruft wiederrum eine Methode des OwareSessionHub Objekts auf, wenn es ein Spielobjekt gibt. Diese Überprüfung ist wichtig, da das Spielobjekt, in dem Codebeispiel *MatchDto* genannt, als Attribut unter anderem die Spiel-ID hat, die zur Identifizierung der SignalR Gruppe dient, an welche die Nachricht geschickt werden soll. Wenn es ein Spielobjekt gibt, wird mit der Nachricht und der Spiel-ID, sowie der Farbe des Spielers, welcher die Nachricht geschickt hat, die Methode des OwareSessionHub Objekts aufgerufen.

```
public async Task SendMessageToSignalRHub(string message, string playerClr,
string gameId)
        {
            await _hubConnection.SendAsync("SendMessage", message, playerClr,
gameId);

        }
```

Codebeispiel 26:Methode im OwareSessionHub für RPC an SignalR Hub[50]

Die aufgerufene Methode *SendMessageToSignalR()*, abgebildet in Codebeispiel 26, löst dann den Remote Procedure Call an den Server aus.

```
public async Task SendMessage(string message, string playerClr, string gameId)
        {
            string connectionId = Context.ConnectionId;
            Dictionary<string, HashSet<string>> groups =
_connections.GetGroups();
            HashSet<string> members = groups[gameId];

            if (members.Contains(connectionId))
            {
                await Clients.Groups(gameId).SendAsync("RecieveMessage", message,
playerClr);
            }

        }
```

Codebeispiel 27: SignalR Hub Methode zum Verschicken von Textnachrichten[51]

[49] Eigener Code

[50] Eigener Code

[51] Eigener Code

Im SignalR Hub wird dann die in Codebeispiel 26 als Parameter übergebene Methode *SendMessage()* aufgerufen, die zuerst mithilfe der erweiterten Verbindungsverwaltung überprüft, ob der Absender Client Mitglied der SignalR Gruppe ist, welche den Spielraum abbildet. Wenn der Client Mitglied ist, also eine Nachricht an die Gruppe verschicken darf, wird, wie in Codebeispiel 27 zu sehen, eine Remote Procedure Call an alle Clients, die Mitglieder der Gruppe sind geschickt. Bis die Nachricht dann in der Oberfläche der Clients angezeigt wird, wird sie wieder, dieses Mal rückwärts, durch das OwareSessionHub Objekt und dann durch das Session-Objekt. Der Remote Procedure Call, wird über einen Event Handler, siehe Codebeispiel 28, verarbeitet. Dadurch wird bei den Clients, die *RecieveMessage()*-Methode aufgerufen, die zum OwareSessionHub Objekt gehört.

```
_hubConnection.On<string, string>("RecieveMessage", RecieveMessage);
```

Codebeispiel 28: EventHandler in der OwareSessionHub Klasse[52]

```
private async Task RecieveMessage(string message, string playerClr)
        {
            await _session.SendChatMessageToView(message, playerClr);
        }
```

Codebeispiel 29: *RecieveMessage*()-Methode der OwareSessionHub Klasse[53]

```
internal async Task SendChatMessageToView(string message, string playerClr)
        {
            await _owareView.UpdateChatWindow(message, playerClr);
        }
```

Codebeispiel 30: *SendChatMessageToView()*-Methode der OwareSession Klasse[54]

Diese Methode ist in Codebeispiel 29 abgebildet. Sie ruft wiederum eine Methode *SendChatMessageToView()* des Session-Objekts auf, die in Codebeispiel 30 dargestellt ist. Es werden dabei keine Änderungen an den Daten vorgenommen, die als Parameter weitergereicht werden, jedoch ist für eine saubere Softwarestruktur dieses wiederholte Aufrufen von Methoden mit der einzigen Aufgabe Daten weiterzuleiten, notwendig. Der letzte Aufruf ist dann eine Methode der OwareView Klasse, welche zur Razor Page gehört, auf der die Nachricht dann gerendert wird.

[52] Eigener Code
[53] Eigener Code
[54] Eigener Code

```
public async Task UpdateChatWindow(string message, string playerClr)
        {
            var formattedMessage = $"({DateTime.Now.ToLongTimeString()})
{playerClr}: {message}";
            messages.Add(formattedMessage);
            await InvokeAsync(StateHasChanged);

        }
```

Codebeispiel 31: *UpdateChatWindow()*-Methode der Razor Page OwareView[55]

Diese Methode, die in Codebeispiel 31 zu sehen ist, formatiert die Chatnachricht dann und fügt die aktuelle Uhrzeit, sowie die Farbe des Spielers, der diese abgeschickt hat als Text hinzu. Daraufhin wird die Nachricht der Liste aller Nachrichten, die während dieses Spiels innerhalb des Spielraumes verschickt wurden, angehängt und die Razor Page neu gerendert. Nun wird die Nachricht in der Oberfläche angezeigt, wie in Abbildung 17 zu sehen. Die Einbindung der Chatbox in die komplette Oberfläche ist in Abbildung 18 zu sehen.

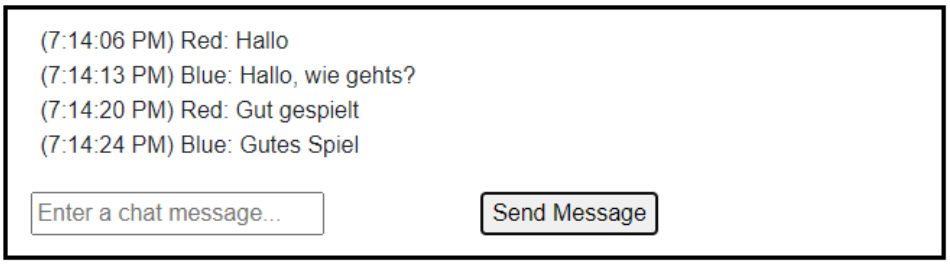

Abbildung 17: Chatbox während eines Spiels[56]

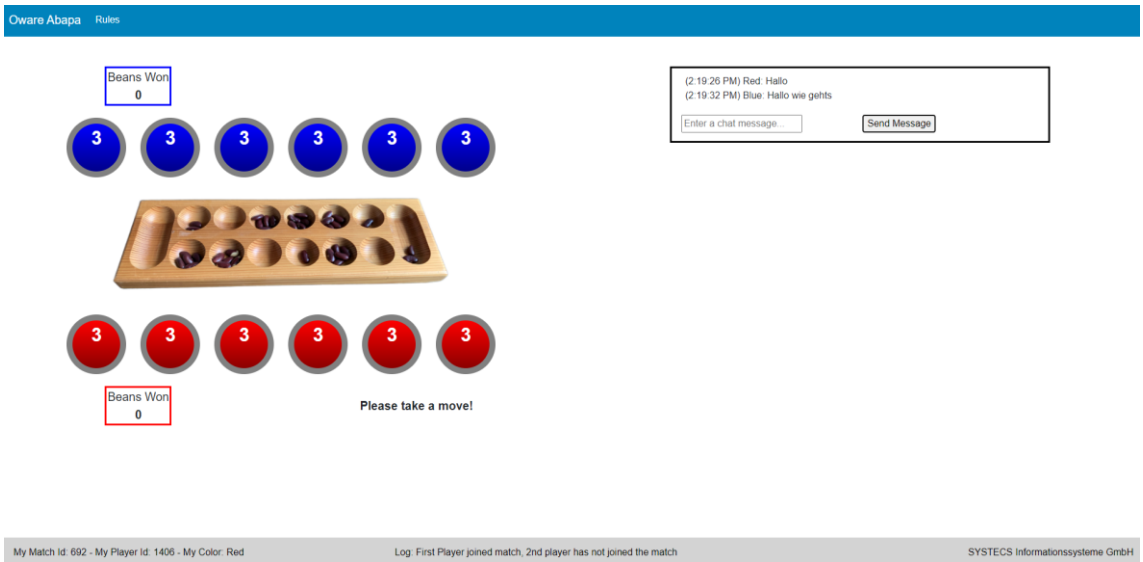

Abbildung 18: Oberfläche eines Spiels zwischen zwei Menschen[57]

[55] Eigener Code
[56] Eigene Abbildung
[57] Eigene Abbildung

4.5 Spielzüge

Das Update der Oberfläche nach einem Spielzug funktioniert ähnlich wie die Chatfunktion. Es werden aber nicht bei jedem Remote Procedure Call zwischen Client und Server oder andersherum alle benötigten Daten als Parameter mitgegeben. Stattdessen fordert der Client, der den Spielzug ausgeführt hat, den Server dazu auf, den anderen Client im Spiel aufzufordern, sich den aktuellen Stand von einer Datenbank zu holen und sein Spielbrett zu updaten. Der Auslöser für die Sequenz dieser Aufrufe liegt in den Zellen des Spielbretts. Sobald ein Client auf eine Zelle klickt und die Maus wieder loslässt, wird der Spielzug ausgeführt, angenommen der Zug ist legal. Die Oberfläche des Spielbretts ist in Abbildung 19 abgebildet.

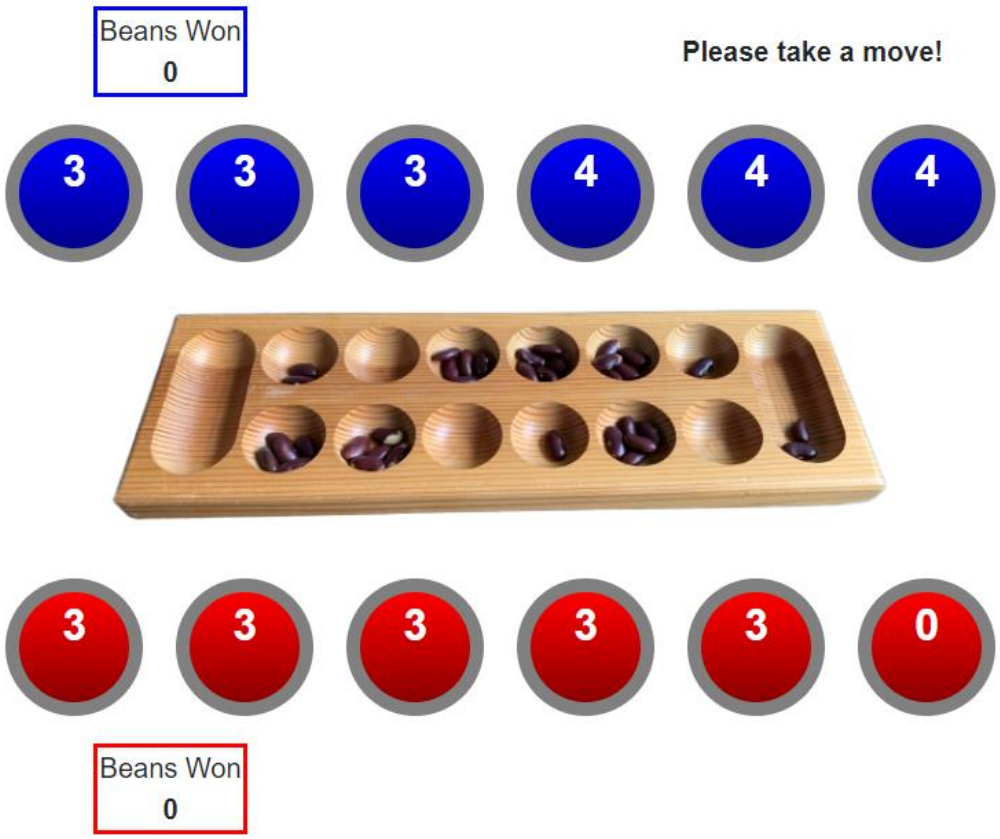

Abbildung 19: Oware Spielbrett in der Webanwendung[58]

[58] Eigene Abbildung

```
<div class="owCell @_owCellVariant"
     @onmousedown="@(e => SelectCell())"
     @onmouseout="@(e => ShowCurrentState())"
     @onmouseup="@(e => EmptyCellAndDistributeBeans())">
    @_noOfBeans

</div>
```

Codebeispiel 32: CSHTML Code für eine Zelle[59]

Der CSHTML Code für eine der, in Abbildung 19 zu sehenden Zellen ist in Codebeispiel 32 abgebildet. Das Event „mouseup“ löst die *EmptyCellAndDistributeBeans()*-Methode aus.

```
private async Task EmptyCellAndDistributeBeans() // called by a human player only
by mouse up
        {
            Session.HumanMoveStarted = true;
            await OnUpdateAllCells.InvokeAsync(new List<int> { MyCellId, 1 });
            await Session.HumanDistributeBeans(PlayerClr, MyCellId);
            Session.HumanMoveStarted = false;
            await OnUpdateAllCells.InvokeAsync(new List<int> { MyCellId, 1 });
            if (Session.IsMatchHuman2Human())
            {
                await Session.CallUpdateGameBoardUi();
            }

        }
```

Codebeispiel 33: *EmptyCellAndDistributeBeans()*-Methode zum Ausführen eines Zuges[60]

Die Methode ist in Codebeispiel 33 dargestellt. Zuerst wird der neue Stand des Spielbretts berechnet und in anderen Methoden an die Datenbank geschickt. Außerdem wird das Spielbrett des Spielers, der gezogen hat, upgedatet. Die letzte if-Anweisung in Codebeispiel 33 ist für das Echtzeitupdate des anderen Spielers relevant. Da ein automatisches Update nach einem Spielzug nur in Spielen zwischen zwei Menschen und nicht zwischen einem Menschen und Computer passieren soll, ist diese if-Anweisung notwendig. Die aufgerufene Methode *CallUpdateGameBoardUi()* liegt in der OwareSession Klasse. Wie bei der Chatfunktion geht der Aufruf erst durch ein OwareSession und dann ein OwareSessionHub Objekt.

[59] Eigener Code

[60] Eigener Code

```
internal async Task CallUpdateGameBoardUi()
        {
            if (MatchDto != null)
            {
                await
_sessionHub.CallUpdateGameBoardUi(MatchDto.MatchDtoId.ToString());
            }

        }
```

Codebeispiel 34: *CallUpdateGameBoardUi*()-Methode der OwareSession Klasse[61]

Diese Methode ist in Codebeispiel 34 zu sehen. Es wird zuerst überprüft, ob ein MatchDto für das Session-Objekt existiert. Die ID des Objekts wird dann der *CallUpdateGameboardUi()* Methode des OwareSessionHub Objekts übergeben, um später die SignalR Gruppe identifizieren zu können.

```
public async Task CallUpdateGameBoardUi(string gameId)
        {
            await _hubConnection.SendAsync("UpdateGameboardUi", gameId);

        }
```

Codebeispiel 35: *CallUpdateGameboardUi()*-Methode der OwareSessionHub Klasse[62]

In Codebeispiel 35 ist die Methode des OwareSessionHub Objekts zu sehen. Hier wird dann der Remote Procedure Call an den SignalR Hub ausgelöst. Im SignalR Hub wird die *UpdateGameboardUi*()-Methode, welche in Codebeispiel 36 abgebildet ist, aufgerufen. Diese Methode ermittelt zuerst den Client, welcher Mitglied der Gruppe ist und nicht der Aufrufer der Methode ist, da nur bei jenem ein Update durchgeführt werden muss.

```
public async Task UpdateGameboardUi(string gameId)
        {
            string playerToUpdate = null;
            HashSet<string> members = _connections.GetMembersOfGroup(gameId);
            foreach (var member in members)
            {
                if (member != Context.ConnectionId)
                {
                    playerToUpdate = member;
                await Clients.Client(playerToUpdate).SendAsync("UpdateGameBoardUi");
                    return;
                }
            }

        }
```

Codebeispiel 36: *UpdateGameboardUi()*-Methode im SignalR Hub[63]

[61] Eigener Code
[62] Eigener Code
[63] Eigener Code

Dafür ist die erweiterte Verbindungsverwaltung notwendig, welche Gruppennamen die entsprechenden Mitglieder Connection-IDs zuordnen kann. Wenn der Client gefunden wurde, wird an diesen ein Remote Procedure Call geschickt, mit der Anweisung, die *UpdateGameBoardUi()*-Methode, siehe Codebeispiel 38, aufzurufen. Dieser Aufruf wird durch einen EventHandler in der OwareSessionHub Klasse angenommen und die Methode aufgerufen. Der Event Handler ist in Codebeispiel 37 abgebildet.

```
_hubConnection.On("UpdateGameBoardUi", UpdateGameBoardUi);
```

Codebeispiel 37: Event Handler für UpdateGameBoardUi Aufruf[64]

```
private async Task UpdateGameBoardUi() // called by the Hub
        {
            await _session.UpdateGameBoardUi();
            _logger.LogWarning("(OwareSessionHub.UpdateGameBoardUI) request to
update UI received");

        }
```

Codebeispiel 38: *UpdateGameBoardUi()*-Methode der OwareSessionHub Klasse[65]

```
internal async Task UpdateGameBoardUi()
        {
            await _owareView.RefreshWithSignalR();
            _logger.LogWarning("(OwareSession.UpdateGameBoardUi) UI has been
updated");

        }
```

Codebeispiel 39: *UpdateGameBoardUi()*-Methode der OwareSession Klasse

Mit dem Methodenaufruf von *UpdateGameBoardUi()* geht die Aufrufsequenz wieder zurück über das OwareSession Objekt zur Oberfläche. Die Methode des OwareSession Objekts ist in Codebeispiel 39 zu sehen. Diese ruft Methode *RefreshWithSignalR()* der Razor Page OwareView auf, welch zuerst den aktuellen Stand des Spiels, also des *MatchDtos* lädt und dann die Razor Page neu rendert.

```
public async Task RefreshWithSignalR()
        {
            await Session.ReloadMatch();
            await InvokeAsync(StateHasChanged);

        }
```

Codebeispiel 40: *RefreshWithSignalR()*-Methode der Razor Page OwareView[66]

[64] Eigener Code
[65] Eigener Code
[66] Eigener Code

In Codebeispiel 40 ist die *RefreshWithSignalR()*-Methode zu sehen und in Codebeispiel 41 die Methode zum aktualisieren des *MatchDto* Objekts.

```
internal async Task ReloadMatch()
        {
            // Reload the current match with complete info from the Db
            MatchDto = (await
OwareMatchServiceApi.TryGetMatchAsync(MatchDto!.MatchDtoId))!;

        }
```

Codebeispiel 41: *ReloadMatch()*-Methode zum Aktualisieren des MatchDto Objekts[67]

Sobald die Razor Page dann neu gerendert ist, hat der Client, der nicht am Zug war, das aktuelle Spielbrett vor sich und kann selbst den nächsten Zug ausführen.

[67] Eigener Code

5 Zusammenfassung

Die Anforderungen an die Echtzeit-Komponenten für die Oware Webanwendung konnten konzeptionell komplett erfüllt werden. Die Implementierung weist allerdings noch Lücken bezüglich des Konzepts auf. Grundlegende Funktionen wie der Chat sowie die Echtzeitübertragung der Spielzüge wurden vollständig in die Anwendung implementiert. Auch die erweiterte Verbindungsverwaltung wurde komplett umgesetzt. Jedoch fehlt das Match Making in der Spielanwendung. Das Problem, das ein Benutzer bei kurzzeitigem Verbindungsabbruch nicht mehr identifizierbar ist, wurde erst im Verlauf der Arbeit erkannt. Daher wurde dieses Problem nur konzeptionell gelöst und ist ebenfalls nicht in der Oware Webanwendung implementiert. Das Konzept, das dafür im Rahmen dieser Arbeit erstellt wurde, kann aber in Zukunft noch umgesetzt werden, oder als Grundlage für eine Lösung des Problems dienen. Die Verwendung von SignalR stellte sich im Laufe der Arbeit als sehr einfach heraus. Sowohl eine ausführliche Dokumentation auf der Microsoft Webseite, sowie SignalR selbst ermöglichen eine unkomplizierte Erstellung von Echtzeit-Komponenten für ASP.NET Webanwendungen. Dadurch, dass die komplette Logik für Transporttechnologien und Protokolle sowie die optimale Auswahl derer von SignalR übernommen wird, reichen grundlegende Kenntnisse der objektorientierten Programmierung, um mit SignalR Webanwendungen zu entwickeln. Die, in dieser Arbeit erstellten Konzepte reizen die Möglichkeiten von SignalR nicht aus. Das Potenzial für verschiedenste Arten von Webanwendungen ist durch die Einfachheit der Verwendung hoch. Die erweiterte Verbindungsverwaltung zeigt, dass die Grundlage von SignalR durch eigenen Code ausgebaut werden kann und neue Anforderungen erfüllt werden können. Auch das Konzept für die Identifizierung eines Benutzers ist eine interessante Idee, die in anderen Anwendungen mit SignalR ebenfalls Verwendung finden kann. Abschließend lässt sich sagen, dass die Entwicklung der Echtzeit-Komponenten mit SignalR sehr gut funktioniert hat. Probleme, die auftraten, können durch eigene Lösungsbausteine behoben werden. Die Einbindung von SignalR in ASP.NET Blazor Projekte ist einfach und gut dokumentiert, was SignalR im Microsoft Umfeld zu einer empfehlenswerten Bibliothek für Echtzeit-Komponenten macht.

Literaturverzeichnis

Microsoft Documentation – Overview of ASP.NET Core (15.11.2022): [online] https://learn.microsoft.com/de-de/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0 [abgerufen am 03.07.2023].

Microsoft Documentation – Overview of ASP.NET Core SignalR (14.02.2023): [online] https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-7.0 [abgerufen am 03.07.2023]

Microsoft Documentation – WebSockets support in ASP.NET Core (02.12.2022): [online] https://learn.microsoft.com/en-us/aspnet/core/fundamentals/websockets?view=aspnetcore-7.0 [abgerufen am 03.07.2023]

Thurlow, R. (05.2009): Request for Comments 5531 - Remote Procedure Call Protocol Specification Version 2, [online] https://datatracker.ietf.org/doc/html/rfc5531 [abgerufen am 31.07.2023]

Microsoft Documentation – ASP.NET Core Blazor (24.02.2023): [online] https://learn.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-7.0&WT.mc_id=dotnet-35129-website [abgerufen am 04.07.2023]

Microsoft Documentation – Mapping SignalR Users to Connections (10.05.2022): [online] https://learn.microsoft.com/en-us/aspnet/signalr/overview/guide-to-the-api/mapping-users-to-connections [abgerufen am 25.07.2023]