**Single Page Chat**

vorgelegt von

**Matthias GÖLZNER**

**Projektarbeit**

zur Erlangung der Berufsreifeprüfung

im Fachbereich

**Informationsmanagement und Medientechnik**



BFI-Salzburg

Salzburg, am 10. April 2019

vorgelegt bei

Manuel Ramsimmer, MSc BEd

Eidesstattliche Erklärung

*Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht. Die vorliegende Arbeit hat in dieser oder einer ähnlichen Form noch keiner anderen Prüfungsbehörde vorgelegen.*

*Salzburg, 10. April 2019 Unterschrift*

*Vorname Nachname*

Kurzbeschreibung

Die Ihnen vorliegende Projektarbeit befasst sich mit der Erstellung einer Website, deren Hauptaugenmerk auf eine Single-Page-Application (im Folgenden: „SPA“) gerichtet ist. Nach der Registrierung und dem Login auf der Landing Page, erhält der Nutzer Zugriff auf die eigentliche SPA, den Chat. Er landet zuerst in der Lobby und kann dann einen Raum erstellen oder einem bereits erstellten beitreten. Sobald sich mindestens 3 Nutzer in einem Raum befinden, können sie ein Spiel starten.

Das Spiel basiert auf dem Regelwerk von Gesellschaftsspielen die bereits unter den verschiedensten Namen veröffentlich wurden. „Werwölfe“, „Mafia“ oder „Town of Salem“ um nur ein paar zu nennen. Kerngedanke jedes dieser Spiele ist, dass jeder Spieler eine geheime - nur ihm bekannte - Rolle einnimmt. Dann unterhält man sich und versucht herauszufinden welcher Spieler welche Rolle verkörpert, woraufhin man versucht gegnerische Parteien durch Wahlen aus dem Spiel zu befördern. Implementiert wurden die 3 wichtigsten Rollen.

Durch den Chat, der eine möglichst niedrige Latenz aufweisen sollte und sehr oft im Verlauf des Besuchs sein Layout wechselt, entschied ich mich zu einer SPA auf Javascript -Basis. Die Website wird auf meinem Linux-Server mit NodeJS und dem Plugin socket.io gehostet, Registrierungsdaten werden in MySQL gesichert und das SSL-Zertifikat wurde mit OpenSSL erstellt.

Inhaltsverzeichnis

[Eidesstattliche Erklärung 2](#_Toc5545142)

[Kurzbeschreibung 3](#_Toc5545143)

[Inhaltsverzeichnis 4](#_Toc5545144)

[1 Einleitung 5](#_Toc5545145)

[2 Frontend 6](#_Toc5545146)

[2.1 Landing-Page 6](#_Toc5545147)

[2.1.1 HTML 6](#_Toc5545148)

[2.1.2 CSS 7](#_Toc5545149)

[2.1.3 SVG 9](#_Toc5545150)

[2.1.4 JS 10](#_Toc5545151)

[2.2 Single-Page-Application 12](#_Toc5545152)

[2.2.1 HTML 12](#_Toc5545153)

[2.2.2 CSS 14](#_Toc5545154)

[2.2.3 Spielregeln 15](#_Toc5545155)

[2.2.4 JS 16](#_Toc5545156)

[3 Backend 20](#_Toc5545157)

[3.1 Server-Struktur 20](#_Toc5545158)

[3.1.1 Ubuntu 18.10 20](#_Toc5545159)

[3.1.2 SSL-Zertifikat 20](#_Toc5545160)

[3.2 MySQL 21](#_Toc5545161)

[3.3 NodeJS 21](#_Toc5545162)

[3.3.1 File Management 23](#_Toc5545163)

[3.3.2 Socket Logic 24](#_Toc5545164)

[3.3.3 Game Logic 26](#_Toc5545165)

[Zusammenfassung 27](#_Toc5545166)

[Abbildungsverzeichnis 28](#_Toc5545167)

[Literaturverzeichnis 29](#_Toc5545168)

1. Einleitung

Bevor ich mich auf das Design meiner Website fixiert habe, recherchierte ich welche Möglichkeiten einem im Webdesign geboten werden. Das Plugin socket.io, welches eine permanent offene Verbindung zwischen Server und Client ermöglicht, sah für mich nach einer machbaren Herausforderung aus, der ich mich stellen wollte.

Ich möchte meine Website nach der Matura noch weiter verfeinern, das Spiel noch um einige Rollen erweitern und vor allem vor Hack-Angriffen sichern.

1. Frontend
   1. Landing-Page

Die Landing-Page ist sehr stilistisch gehalten. Das Logo, welches auch als Interface dient, wurde mit InkScape erstellt. Wenn man die Maus über das Logo navigiert, ändert sich die Farbe graduell auf Grün, damit man merkt, dass hier interagiert werden kann. Da das Spiel auf dem Cthulhu-Mythos des berühmten Horror-Buchautors H.P. Lovecraft basiert, wurde das Logo entsprechend im Stil eines Pentagramms designed.

Die Untermenüs sind als „modal-content“ gestaltet d.h. die Seite wird nicht verlassen, es wird lediglich ein bereits geladener aber nicht sichtbarer Bereich dargestellt, der sämtlichen vorherigen Content überlappt. Wenn man den nicht relevanten Randbereich anklickt, wird der „modal-content“ wieder unsichtbar.

* + 1. HTML

Die Landing-Page besteht aus äußerst wenig HTML-Code. Die Funktionen sind größtenteils Script-gesteuert, selbst das Registrations-Formular wird nicht über die herkömmliche Methode verschickt. Es umfasst 4 Elemente:

* SVG-Logo
* Login-Formular
* Registrierungsformular
* Impressum

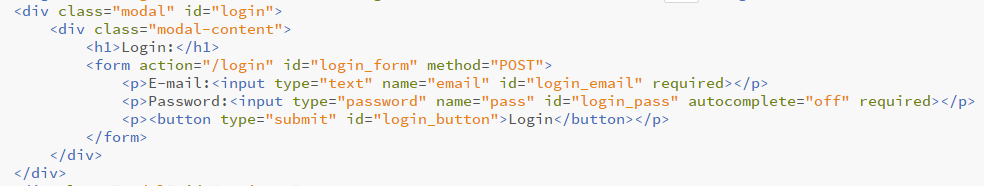


Abb. : Modal-Content

Die Verschachtelung zweier <div>-Elemente, die respektive als Klasse: „modal“ und „modal-content“ bezeichnet wurden, ermöglicht den restlichen Content abzudunkeln und nur den gewünschten Bereich anzuzeigen. In CSS wurde nicht nur das Layout sondern auch ein Teil der Funktion vordefiniert auf die mit Javascript anschliessend zugegriffen wird.

Die Formularfelder in Login und Registrierung sind allesamt Pflichtfelder. Serverseitig wird die Eingabe sicherheitshalber noch vor Weiterleitung an MySQL von speziellen Sonderzeichen befreit, um SQL-Injektionen abzuwehren.

Eine SQL-Injektion nutzt die Befehlsstruktur von SQL und Linux-Distributionen aus indem es einen Linux-Befehl als SQL „tarnt“ der dann auf dem Server ausgeführt wird. Dies kann man doppelt absichern, indem man das Serverprogramm NodeJS nicht mit Administratorrechten ausführt und zusätzlich SQL-typische Sonderzeichen entfernt um etwaige Fehlermeldungen zu unterdrücken.

* + 1. CSS

Der Hintergrund ist ein radialer Farbverlauf, der mit -webkit deklariert wird. Dies kann zu Kompatibilitätsproblemen führen, da -webkit nicht von allen Browsern unterstützt wird. Chrome, Safari und Opera unterstützen es nativ. Internet Explorer z.B. unterstützt es nicht.

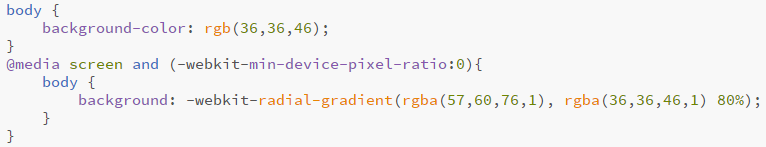


Abb. : Browser-Kompatibilität

Diese Problematik kann man umgehen, indem man zuerst eine Hintergrundfarbe definiert (dies wird von jedem Browser unterstützt) und dann mit der Regel @media screen (Die Seite wird auf einem Bildschirm angezeigt, sprich nicht wenn man es drucken würde) UND -webkit-min-device-pixel-ratio:0 (Eine Information die nur webkit unterstützende Browser liefern) überschreibt. Somit wird bei IE etc. ein einfarbiger Hintergrund dargestellt und bei Chrome, etc. der gewünschte radiale Farbverlauf.

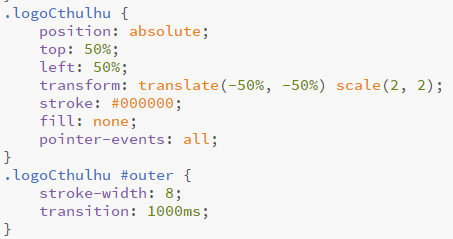


Abb. : SVG-CSS

Das Logo wurde absolut mittig zentriert, schwarz gefärbt und das „pointer-events“ Attribut auf „all“ gesetzt. Dies ermöglicht später Funktionen auszuführen, wenn man den Cursor über das Objekt bewegt. Mit transition und einer Zeitangabe kann man spätere Veränderungen durch Funktionen zeitlich ausdehnen. In diesem Fall werden alle Veränderungen in einer Sekunde durchgeführt.

Das Modale Interface wird in CSS vordefiniert:

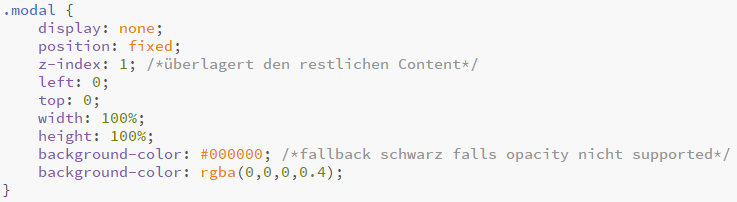


Abb. : Modal-CSS

Mit „display: none“ lädt das Fenster bereits, wird aber nicht angezeigt. Z-index: 1 überlagert den restlichen Content und für den Fall, dass Transparenz nicht unterstützt wird definiert man (ähnlich der Variante oben) die background-color zweimal. Die zweite Definition wird als letztes ausgeführt und überschreibt die erste sofern eine Darstellung möglich ist.

* + 1. SVG

Die „Scalable Vector Grafic“ wurde mit InkScape erstellt und als XML abgespeichert. Sämtliche Vektoren sind darin mit einem Editor nach Wahl lesbar.

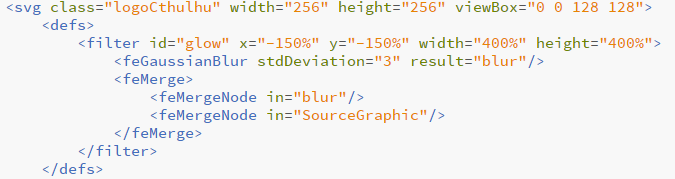


Abb. : SVG-Filter

Zuerst setzt man den Renderbereich mit width, height und viewBox fest. Der Filter funktioniert ähnlich einer CSS man kann nach dem Rendern nachträglich das Bild ähnlich einem Photoshop-Filter überarbeiten. In diesem Fall wird ein Gaußscher Weichzeichner auf die Punkte angewandt mit denen man interagieren kann.

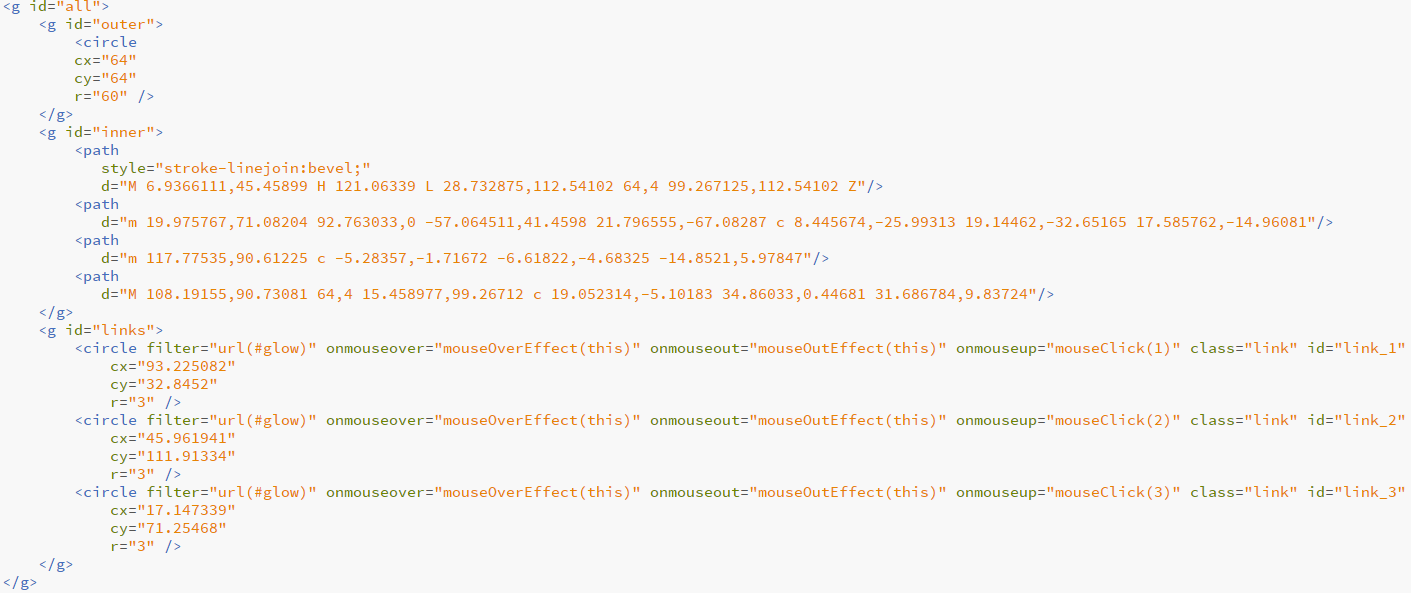


Abb. : SVG-Vektoren

* Outer: Vektoren des äußersten Kreises
* Inner: Vektoren des Pfades der inneren komplexen Struktur
* Links: Vektoren der Buttons für modalen Content

Onmouseover, onmouseout und onmouseup führen jeweils eine JS-Funktion aus.

Nach der Registrierung wird eine SVG gezeigt, die 3 blinkende Punkte darstellt, während man auf eine Antwort des Servers wartet. Da die Reaktionszeit zu schnell ist, sieht man diese „leider“ kaum.

* + 1. JS

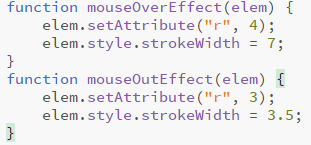
mouseOverEffect: Sobald man den Cursor über einen der Buttons bewegt, wird der Radius des Kreises größer, respektive wird mit mouseOutEffect dieser Vorgang bei Verlassen des Buttons wieder rückgängig gemacht.

Abb. : Mouse-Effects

mouseClick: Diese Funktion wird ausgeführt, sobald man einen der Buttons anklickt, Dabei wird eine Zahl von 1 – 3 übergeben und mit einem switch der entsprechende Modal-Container sichtbar gemacht.

window.onclick: Damit wird, wenn man den abgedunkelten Außenbereich des Modal-Containers anklickt, dieser wieder unsichtbar.

Abb. : Modal-Funktionen

Am Ende des HTML-Codes steht noch eine letzte Funktion mit der die normale Arbeitsweise des <form>-tags mit event.preventDefault() unterdrückt wird. Dies hat zwei Gründe: Zum einen soll die Seite nach der Registrierung nicht verlassen werden und zum anderen soll es die Anzeige eines Lade-Icons ermöglichen.



Abb. : Registrierungsformular

Zuerst wird das Formular versteckt und das Lade-Icon angezeigt. Dann wird mit der Ajax-Syntax der Post-Befehl verschickt und je nach Antwort das Lade-Icon wieder versteckt und erfolgreich/nicht erfolgreich angezeigt.

* 1. Single-Page-Application

Wenn der Login erfolgreich war, wird man auf den eigentlichen Kern der Website verwiesen, ohne Authentifizierung ist der Aufruf dieser Seite nicht möglich. Ein Chat sollte kurze Latenzen und eine schnelle Reaktionszeit aufweisen, ein PHP-gesteuerter Chat ist zwar möglich, aber mit einem enormen und unsachgemäßen Aufwand verbunden. Im Verlauf des Besuchs wird diese Seite dynamisch verändert. Der Client baut nach dem Login eine „socket“ Verbindung mit dem Server auf, die er erst beim Verlassen wieder auflöst. Der Client erwartet Befehle vom Server und kann ihm auch Befehle zusenden.

* + 1. HTML

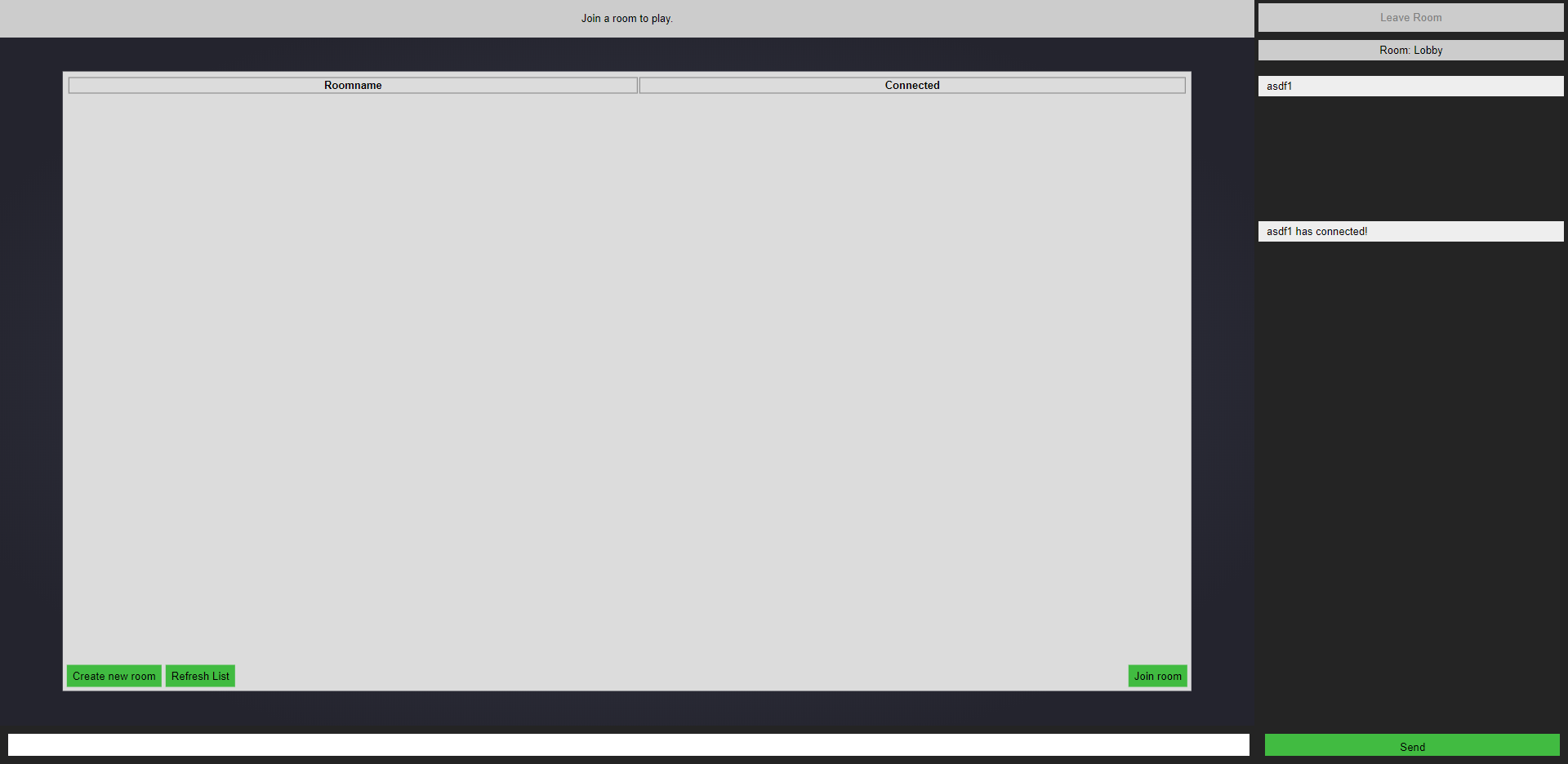


Abb. : Chat-Layout

Der Chat ist grob in 3 Teile aufgeteilt, links befindet sich ein Interface um Räume zu erstellen, betreten und später auch das Spiel anzuzeigen. Im rechten Bereich befindet sich ein Button um einen Raum wieder zu verlassen, der Name des Raums in dem man sich befindet, eine Liste der Clients die sich in selbigem Raum aufhalten und eine Liste der Chat Nachrichten. Unten ist die Eingabe für den Chat.

Der HTML-Code ist auch hier äußerst kurz ausgefallen, da man nur die Container und „unordered lists“ vordefinieren muss, damit Javascript darauf zugreifen und mit Content befüllen kann.

Auch hier wurde ein Modal-Container integriert um einen neuen Raum zu erstellen, der mit „Create new room“ aufgerufen wird.

Sobald sich mindestens 3 Clienten in einem Raum eingefunden haben, können sie Das Spiel mit dem „Vote“-Button starten. Der Linke Bereich wird neu generiert und mit je einem Button pro Spieler populiert. In der oberen Leiste, die als Infobox dient, wird die Spielphase angezeigt in der man sich gerade befindet.

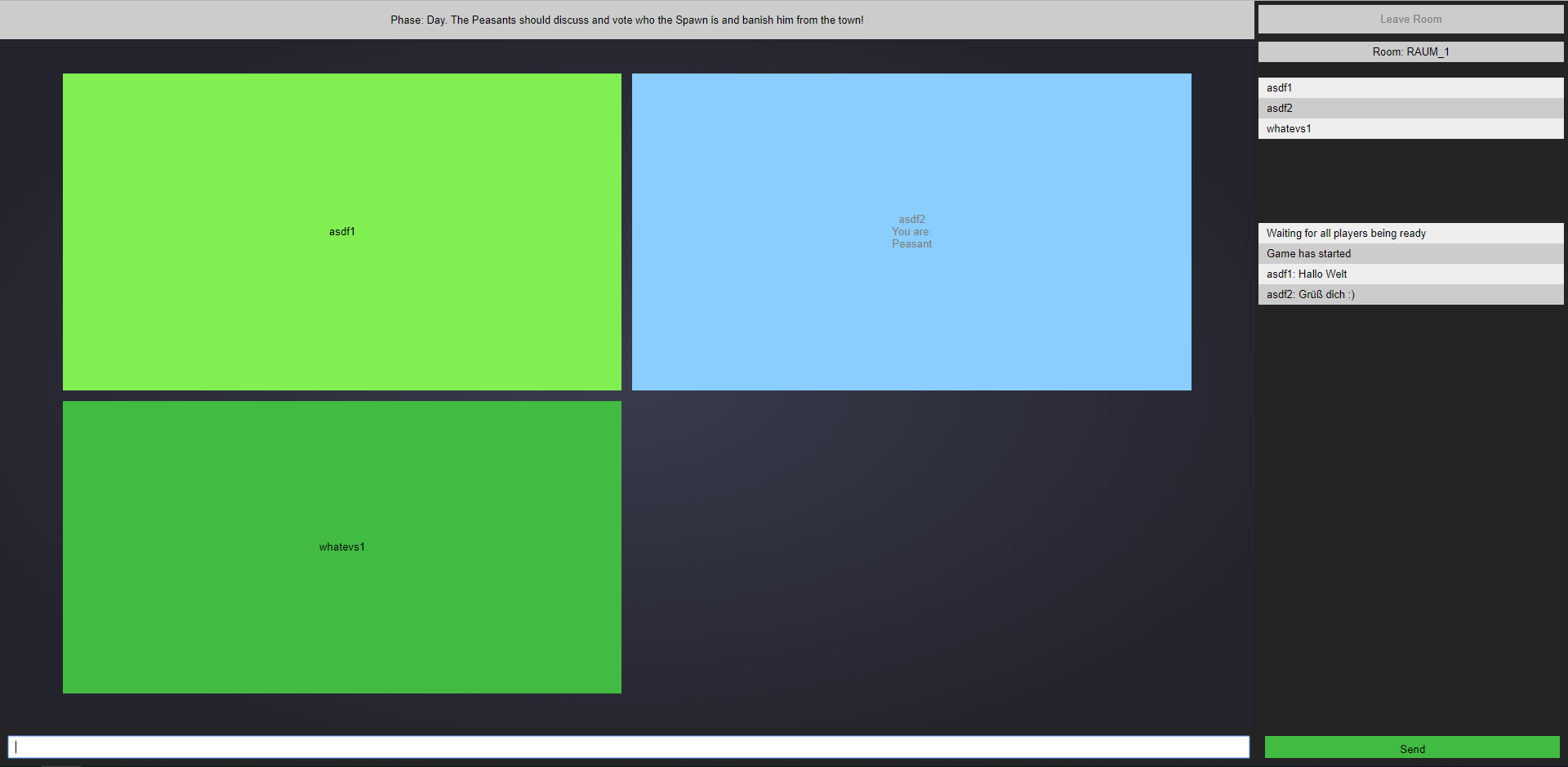


Abb. : Game-Layout

* + 1. CSS

Die drei Bereiche sind von der Bildschirmgröße abhängig. Bei besonders kleinen Geräten (Abbildung rechts), führt dies allerdings zu Leseschwierigkeiten. Auf horizontal gehaltenen Mobilgeräten verbessert sich die Lesbarkeit, allerdings überschreitet dann die Textgröße gelegentlich die Fläche der Buttons.

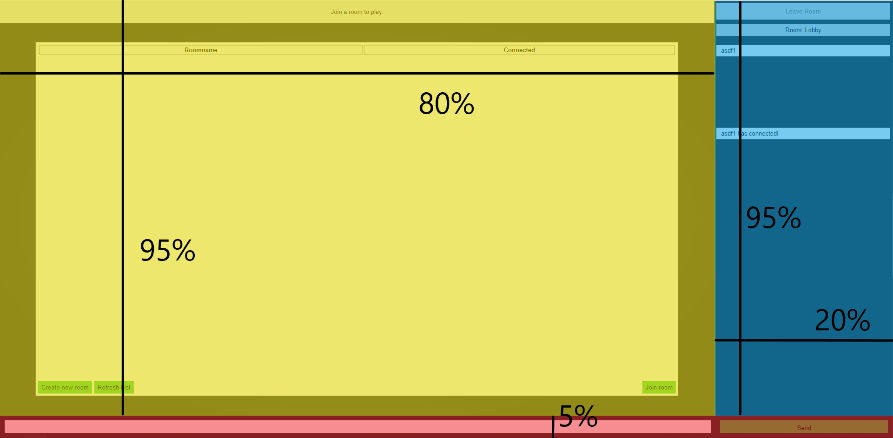
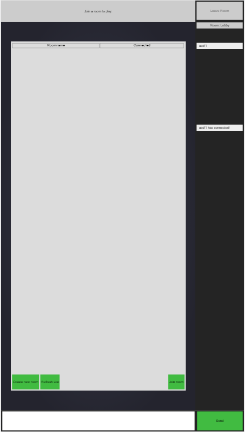


Abb. : Chat-Layout Mobiltelefon

Abb. : Chat-Layout prozentual

Im CSS sind auch noch nicht existente Objekte vordefiniert, wie zum Beispiel die Textnachrichten oder Buttons für das Spiel:



Abb. : Text-Child

Generell sind alle Child-Elemente einer Liste (i. d. Fall Userliste) in dem div mit der ID: userlist mit der Hintergrundfarbe #ccc eingestellt. Mit dem „:nth-child(odd)“ Selektor lassen sich aber alle ungeraden Child-Elemente auswählen und auf #eee umfärben. Mit der abwechselnden Hintergrundfarbe wird die Lesbarkeit enorm erhöht, vor allem da hier kein margin oder padding zwischen den List-Elementen existiert. Selbiges gilt für die Liste der Nachrichten.

Alle Buttons werden mit dem „:hover“ und „:disabled“ Selektor auf ähnliche Weise eingefärbt, ersteres wenn man mit dem Cursor darüber ist, zweiteres wenn der Button deaktiviert wurde.



Abb. : Button-CSS

* + 1. Spielregeln

Um den Programmcode besser nachvollziehen zu können, sollte man sich mit den Regeln des Spiels vertraut machen.

Jedem Spieler wird eine Rolle zugewiesen, die nur diesem bekannt ist. Das Verhältnis bei 8 Spielern ist:

* 5 Peasants
* 1 Inspector
* 2 Spawns

Die Peasants arbeiten zwar mit dem Inspector auf dasselbe Ziel hin, nämlich alle Spawns aus dem Dorf zu vertreiben. Eine Runde wird in 3 Phasen unterteilt:

Abb. : Game-Zyklus

Am Tag wählen alle noch lebenden Spieler, einen Spieler dem sie nicht vertrauen und aus dem Dorf verjagen wollen. Der Spieler mit den meisten Stimmen scheidet aus dem Spiel aus und darf nur noch beobachten aber nicht eingreifen. Bei Gleichstand scheidet niemand aus.

Am Abend wählt der Inspector einen Mitspieler aus, von dem er die Rolle erfährt.

In der Nacht halten die Spawns auch eine Wahl ab, um einen Nicht-Spawn zu beseitigen, welcher aus dem Spiel ausscheidet.

Gewonnen hat die Gruppe die am Ende übrig bleibt, Spawns oder Peasants (inkl. Inspector).

* + 1. JS

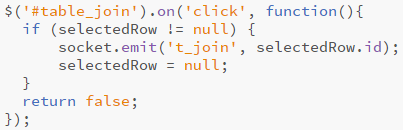
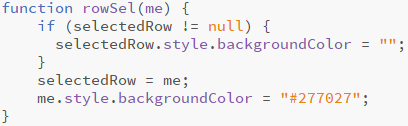
Das Javascript in diesem Bereich ist bereits um einiges aufwendiger, als auf der Login-Seite. Es umfasst rund 200 Zeilen Code und sorgt größtenteils für die Kommunikation zwischen Client und Server. Hier wurde wieder dieselbe Funktion für modal-content implementiert wie auf der Login-Seite. Als auch eine Funktion um die ausgewählte Tabellenzeile zu visualisieren.

Abb. : Room-Join

Abb. : Reihen-Selektor

Beim Anklicken einer Tabellenzeile wird, sofern vorher bereits eine Zeile ausgewählt wurde (if !=null), die Hintergrundfarbe wieder gelöscht und danach die neue Zeile eingefärbt. Die Variable selectedRow wird auch verwendet, wenn man auf den „JoinRoom“ Button klickt um den Namen des Raumes weiterzugeben. Hier sehen Sie bereits das erste Script, das socket.io verwendet. Der Befehl socket.emit(„STRING“, VARIABLE); sendet Daten an den Server. Damit dieser weiß wie er sie zuordnen muss, hat sie den Namen „STRING“ und der zu sendende Datensatz befindet sich in VARIABLE.

Das Gegenstück zu socket.emit bildet socket.on(„STRING“, function(VARIABLE)). Dieser Befehl wartet auf einen Datensatz mit dem Namen „STRING“, um dann eine Funktion auszuführen und dieser den Datensatz VARIABLE zu übergeben. VARIABLE kann alle möglichen Datentypen enthalten. Von Integer zu String über Arrays bis hin zu ganzen Objekten (Wobei Objekte und Arrays in Javascript dasselbe sind). Wenn die Objekte zu groß werden, kann dies allerdings zu einem Timeout und Datenverlust führen.

Um dem Browser diese Befehle verständlich zu machen muss man die Library mit folgendem Code integrieren:



Abb. : Library

Die erste Library „socket.io“ wird vom Server mitgeliefert, die zweite „JQuery“ kann man auch online referenzieren lassen, da die Entwickler diese auch permanent erweitern und verbessern. Somit entfällt die Mühe, diese Libraries zu updaten, wenn man neueren Code verwendet.

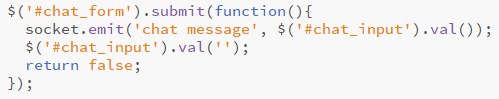


Abb. : Socket.emit

Hier zu sehen ist die grundlegendste aller Funktionen dieser Seite, sie bildet quasi den Backbone aller weiterer Funktionen. Mit dem JQuery Selector $(„#NAME“).submit() lässt man nach einem Element mit der ID: NAME suchen – in diesem Fall das Formular mit der man chattet – und auf dessen Event „submit“ warten, woraufhin eine Funktion ausgeführt wird.

Nach Versenden des Wertes der #chat\_form, den man sich mit #chat\_input holt, wird das Eingabefeld mit .val(„“) geleert. **Wichtig:** return false verhindert die normale Funktion eines Elements, im Falle des Absendens eines Formulars würde die Seite verlassen werden wollen, dies gilt es auf einer SPA unbedingt zu verhindern.

Wenn der Client einen Datensatz mit dem Namen „chat message“ erhält, wird in der Nachrichtenliste mit .append() ein neues Child hinzugefügt. „msg“ enthält Inhalt der Textnachricht, die der Client vom Server erhält.

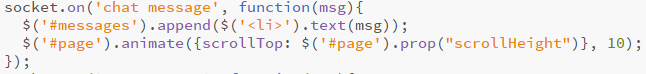


Abb. : Append-Message

Danach lässt man mit scrolltop() den scrollbaren Bereich ganz nach unten scrollen, diese Funktion kann man zusätzlich mit animate und einer Zeitangabe (hier 10 ms) flüssig übergehen lassen.

Ähnlich verhält es sich bei der Liste der User, nur deklariert man hier zur späteren Verwendung noch die ID des Objektes mit der socket.id des Users. Eine socket.id ist ein zufällig generierter String der eindeutig einem Client zugewiesen ist und zur Identifikation dient.

Die beiden socket.emit Funktionen senden die Interaktion des Spielers an den Server:

* Start\_button
* Grid\_container

„Start\_button“ sendet die Bereitschaft des Clients zum Spielstart an den Server und versteckt den Button, sollte ein weiter Spieler den Raum betreten bevor das Spiel startet (nach Spielstart nicht mehr möglich) wird der Button wieder angezeigt und der Bereitschaftsstatus Serverseitig zurückgesetzt.

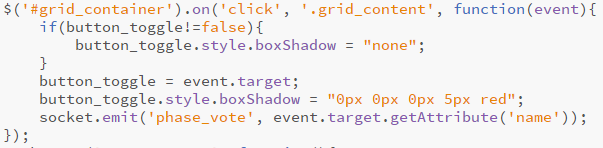


Abb. : grid\_container

„Grid\_container“ färbt den Rand der angeklickten Box rot und entfärbt die etwaige vorher angeklickte Box, danach wird der Name des Ziels an den Server gesendet. Je nach Phase in der man sich befindet wird dann vom Server eine Antwort gesendet. So wird der Klick in der Tagesphase als Votum gegen einen Mitspieler gewertet. In der Abendphase wird die Information dafür verwendet das Ziel des Inspectors aufzudecken.

Die folgenden 6 socket.on Funktionen sind allesamt für die Darstellung des Spiels verantwortlich:

* Game\_start
* Reveil\_self
* Reveil
* PhaseHandler
* Phase
* End

„Game\_start“ populiert den grid\_container mit Buttons für die teilnehmenden Spieler, färbt den Button, der für den Spieler steht, blau und deaktiviert ihn, damit man sich nicht unabsichtlich selbst aus dem Spiel wählt. Danach wird „get\_role“ an den Server gesendet, um seine eigene Rolle zu erfahren. Es ist unratsam von Anfang an ein array aller Rollen plus Spieler.IDs an den Client zu senden. Diese wäre äußerst einfach auszulesen und würde dem Spieler das Schummeln ermöglichen.

Als Antwort auf „get\_role“ schickt der Server „reveil\_self“. Diese Funktion manipuliert den eigenen Button und fügt die Rolle des Spielers sichtbar hinzu.

Eine ähnliche Funktion hat „reveil“, diese Nachricht schickt der Server an jenen Inspector der gerade die Rolle eines Mitspielers abfragt.

„phaseHandler“ blockiert das Interface bei allen ausgeschiedenen Spielern und die Buttons ausgeschiedener Spieler bei allen die noch im Spiel sind. Außerdem blockiert es das Interface bei Spielern, die noch im Spiel - aber nicht an der Reihe sind.

„phase“ ändert der Phase entsprechend die Infobox im oberen Bildschirmbereich.

Zu guter Letzt räumt die „end“ Funktion, nach Spielende, das Spielbrett auf und setzt alle Variablen wieder auf ihren Anfangswert. Zusätzlich zeigt es die gewinnende Partei an.

1. Backend

NodeJS und vor allem das Plugin socket.io benötigen einiges an Rechenleistung und müssen auf dem PC so eingerichtet werden, dass ein Arbeiten und die Abgabge dieser Arbeit auf einem USB-Stick nicht in Frage kamen. Also wurde ein Server gemietet.

* 1. Server-Struktur
     1. Ubuntu 18.10

Bei dem Server wurde auf eine grafische Oberfläche verzichtet, damit das gesamte Rechenpotential für die Website genutzt werden kann. Alles wird von der Kommandozeile aus gesteuert. Mit der Unix-Library „authbind“ werden die Ports 80 für http und 443 für https freigegeben. Diese Ports können normalerweise nur von Programmen benutzt werden, die mit „-sudo root“ gestartet wurden. „Sudo“ führt ein Programm mit den Rechten eines anderen Benutzers aus in diesem Fall „root“, sprich dem Admin. „Authbind“ umgeht diese Sicherung und ermöglicht NodeJS ohne root auf den Ports 80 und 443 zu arbeiten. NodeJS „hört“ auf diese beiden Ports und verweist alle http Zugriffe automatisch an den Port 443 um den User sofort an die authentifizierte Adresse umzuleiten. Würde man NodeJS mit Admin-Rechten starten und würde es gehackt werden, hat der Hacker Zugriff auf den gesamten Server.

* + 1. SSL-Zertifikat

Um die Website unter https:// erreichbar zu machen und vor Allem um den Zugang im bfi zu ermöglichen, welches unsichere Seiten blockiert, wurde ein SSL-Zertifikat ausgestellt. Das Zertifikat befindet sich auf einer anderen Hierarchie-Ebene als NodeJS, damit ein potentieller Hacker darauf keinen Zugriff hat. Der Ordner ist zusätzlich nur dem root, also dem Admin zugänglich. Das Zertifikat lässt sich mit der Unix-Library „Certbot“ erstellen und muss alle 90 Tage erneuert werden.

* 1. MySQL

Im Hintergrund läuft eine SQL-Datenbank mit einer einzigen Tabelle der Anmeldedaten mit 3 Feldern.

* Username: Deklariert als unique-Key, jeder Username darf nur ein einziges Mal vergeben sein.
* Email-Adresse: Auch ein unique-Key.
* Passwort: Bevor das Passwort in die Datenbank eingespeist wird, wird es mit der Funktion crypto.MD5 verschlüsselt.
  1. NodeJS

Mit rund 500 Zeilen Code ist dies der größte Teil dieser Arbeit. Um den Inhalt dieser Arbeit nicht künstlich aufzublasen, wird vorausgesetzt, dass der Leser die gängigsten Routinen des Programmierens beherrscht. Ansonsten ist das Buch „Node.js in Action“ (Cantelon, 2013) und die Website „W3Schools“ (unbekannt, 2016) eine gute Einstiegslektüre.

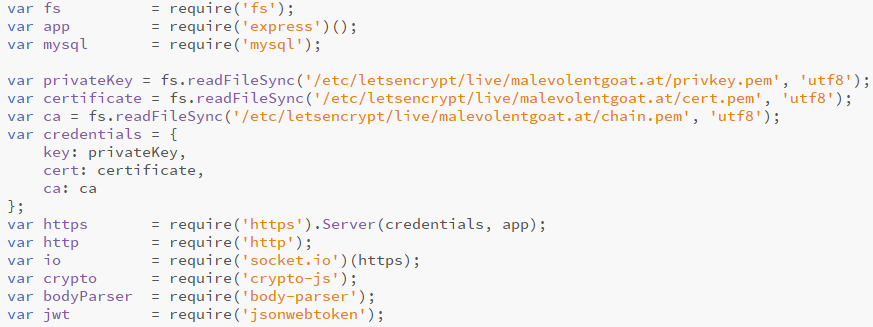


Abb. : Libraries

Am Anfang werden die notwendigen Libraries geladen.

* fs – wird benötigt, um die Daten der Zertifikate auszulesen und an das Objekt credentials weiterzugeben.
* Express – beinhaltet Befehle, um mit Get und Post Anfragen umzugehen.
* Mysql – ermöglicht den Zugriff auf die Datenbank
* http und https – ermöglichen jeweils auf jene Protokollaufrufe zu reagieren. Https hat zusätzlich als variable die Zertifikate erhalten.
* Socket.io – enthält Befehle, um eine permanente Verbindung mit dem Clienten zu ermöglichen, abhängig von https.
* Crypto-js – Sammlung an Funktionen für Kryptografische Operationen
* bodyParser – Wandelt die Post Übergabewerte in leicht verarbeitbare Variablen um.
* Jsonwebtoken – Lässt den Clienten einen Cookie erstellen der mit einem geheimen Schlüssel – hier „superSecretPassphrase“ – und dem Usernamen die Validität des Logins überprüft.
  + 1. File Management



Abb. : Server-Instanzierung

http.createServer(req, res).listen(80); lässt einen Server erstellen, der auf Port 80 zuhört, und je nach req-uest eine res-ponse verschickt. In diesem Fall einen Code 301 – eine Weiterleitung – an das https-Protokoll, also Port 443. https.listen(443) definiert den Port 443 als https-Port. Diese beiden Ports sind nach W3C Standards so vorgegeben.

App.get(„/“) wartet bereits auf einen Aufruf der Website und schickt die Landing-Page „index.html“. Es reagiert nur auf Get-Aufrufe.



App.post(„/register“) wartet auf einen Post Befehl durch das Registrierungsformular und verschlüsselt das Passwort, entfernt ungültige Zeichen und schickt eine Anfrage an die Datenbank. Mit try(), sorgt man dafür, dass bei Erfolg der Code 201 verschickt wird, also Erfolg. Bei einer Fehlermeldung „fängt“ Catch() diese, der Client erhält einen Code 400 – Bad Request und in der Serverkonsole wird der Fehler aufgezeichnet. Würde SQL ohne try/catch einen Fehler auswerfen, würde der Server abstürzen.

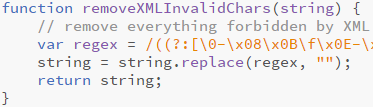
In der Variable regex sind alle Zeichen enthalten die ersetzt werden müssen, dies kann man sehr einfach mit der .replace Funktion realisieren.

Abb. : Remove-Characters



Abb. Login

App.post(„/login“) sendet auch eine Anfrage an mySQL und überprüft, ob der User bereits in diesem Moment angemeldet ist. Eine zweimalige Anmeldung führt zu groben Problemen, wenn sie sich im selben Raum aufhalten und sollte tunlichst unterbunden werden. In diesem Fall wird der User wieder auf die Landing Page zurückgeschickt.

Ist der Login erfolgreich wird dem Client ein Token gesendet, der aus seinem Usernamen mit dem Schlüssel „superSecretPassphrase“ generiert wird. Der Server kann dies mit dem Schlüssel wieder dechiffrieren und die Richtigkeit überprüfen. Der User sieht nur zufällige Zeichen.

* + 1. Socket Logic

Socket.io ist Eventbasierend, das bedeutet, sobald der Client ein Event meldet, reagiert der Server darauf, vice versa. Die realisiert man mit socket.on(„EVENT“, „REAKTION“), letzteres wird meist mit einem Funktionsaufruf programmiert. Da viele Befehle in socket asynchron abgearbeitet werden, muss man oft eine Funktion als zweiten Parameter angeben, um das gewünschte Ergebnis zu erreichen. Zum Beispiel: socket.join fügt den Clienten in einen Raum hinzu. Würde man dann mit „.emit“ sofort eine Nachricht an den Raum schicken, ist der Befehl „.join“ noch nicht vollständig bearbeitet, in Folge der Client noch nicht in dem Raum eingegliedert und er erhält die Nachricht nicht.

Abb. : Asynchronität

Das Event „chat message“ sorgt für die korrekte Zustellung jeder Nachricht. Zuerst wird kontrolliert, ob das erste Zeichen ein „Slash“ ist, dies ist ein Kennzeichen für Befehle, die zu Debugging-Zwecken an der Server-Konsole etwas ausgeben. Dann wird kontrolliert, ob der Client sich gerade in einem laufenden Spiel befindet. In diesem Fall wird die Nachricht nur zugestellt, solange er gerade am Zug ist und auch nur an Clients in seinem Team. Ansonsten wird die Nachricht an jeden im Raum zugestellt.

„t\_create“ und „t\_join“ sind beides Events zum Betreten von Räumen, sie sind fast ident. Create erstellt nur dann einen Raum, wenn er nicht bereits existiert und Join funktioniert nur wenn in diesem Raum gerade kein Spiel am Laufen ist. „t\_refresh“ liefert dem Client eine Liste der offenen Räume.

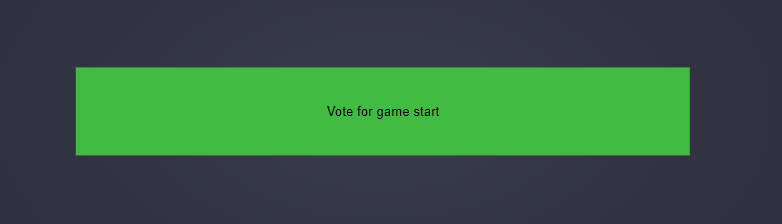


Abb. : Game\_start

Dieser Button löst das Event „vote“ aus, welches den Bereitschaftsstatus dieses Clients auf true setzt und die Funktion checkForStart auslöst.



Abb. : checkForStart

In der for-Schleife wird durch alle Clients in diesem Raum iteriert und bei jedem bereiten Spieler x um eins erhöht. Wenn am Ende x so hoch ist wie die Anzahl der Clients, also alle ja gevotet haben, und x größer ist als 3, das absolute Minimum an Spielern, dann wird „assignRoles“ ausgeführt und das Spiel startet. „assignRoles“ verteilt zufällig die Rollen entsprechend der Anzahl der Clients.

* + 1. Game Logic

Das Letzte und wichtigste Event ist das „phase\_vote“. Wenn ein Spieler gegen einen anderen Spieler eine Stimme abgibt, indem er auf einen Button klickt, wird dieses Event ausgelöst. Es bildet die gesamte Logik des Spiels mithilfe einiger Unterfunktionen.

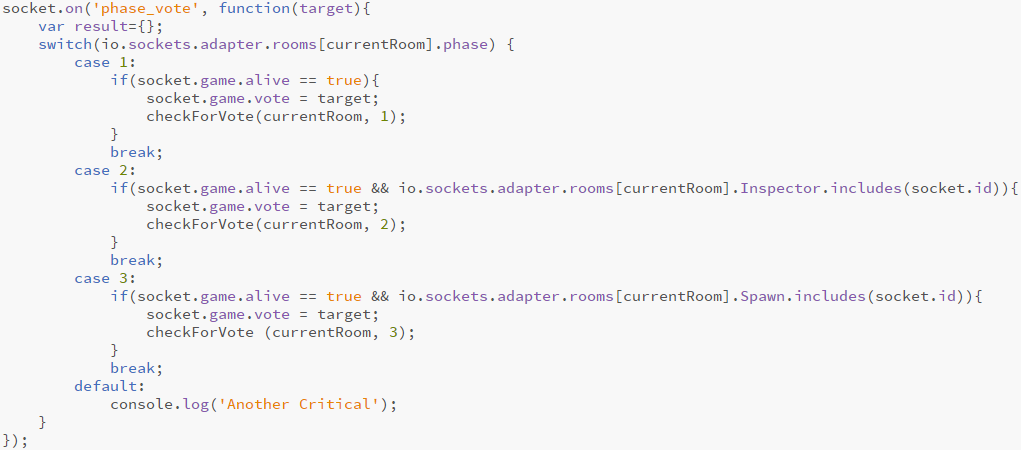


Abb. : phase\_vote

Zuerst wird mit einem Switch festgestellt in welcher phase der Raum sich befindet und entsprechend kontrolliert, ob der wählende Spieler überhaupt dazu im Moment berechtigt ist. Wenn dies der Fall ist, wird die Funktion „checkForVote“ ausgelöst.

Diese iteriert durch alle Spieler, die in dieser Phase berechtigt sind zu voten und übergibt, sobald alle gevotet haben, das Ergebnis an die Funktion „phaseHandler“ und entfernt Spieler entsprechend aus dem Spiel. Für den Fall, dass bereits jetzt, durch einen Disconnect ein Team ausgeschaltet wurde, wird stattdessen „cleanUp“ ausgeführt.

Der „phaseHandler“ kontrolliert, ob die Siegbedingung erreicht wurde, woraufhin es „cleanUp“ ausführt, wenn dies nicht der Fall ist, setzt es die Phase des Raums, entsprechend Abb.: 16, um eins weiter. Wenn kein Inspector mehr im Spiel ist, wird seine Phase übersprungen.

Die „cleanUp“-Funktion räumt den Raum auf, setzt ihn auf Anfangsstatus zurück und sendet an alle Clients, wer diese Runde gewonnen hat.

Zusammenfassung

Die Arbeit an diesem Projekt war eine Herausforderung, der ich mich unbedingt stellen wollte, um zu sehen ob ich den Ehrgeiz habe der notwendig ist, um mich an der Fachhochschule für den Studiengang Multimedia Technology einzuschreiben. Aus meiner Sicht fühle ich mich bereit dazu. Am Anfang hatte ich bereits ein gutes Konzept bereitgelegt, nach dem ich gearbeitet habe. Einen eigenen Server zu mieten war eigentlich nicht geplant, hat aber die Arbeit ungemein erleichtert, aber auch etwas schwieriger gemacht. Alles in allem war die Arbeit immer mit Spaß verbunden und selten etwas, dem ich ungern nachgegangen bin.

Besonders erfreulich finde ich die nahezu fehlerfreie Abwicklung des Spiels, also z.B. die Tatsache das egal zu welchem Zeitpunkt ein Spieler sich entscheidet das Fenster zu schließen, das Spiel immer noch ohne Probleme weiterläuft und den Server nicht zum Absturz bringt, wie es am Anfang der Fall war. Manchmal stoße ich auf noch unerklärliche Server-Abstürze. Dies könnte aber auch daran liegen, dass dieser nur über 2GB Arbeitsspeicher verfügt.

Dieses Projekt werde ich nach Abgabe noch etwas verbessern, indem ich noch weitere Rollen hinzufüge, wie zum Beispiel das Orakel, das einmal pro Spiel einen Mitspieler vor dem Ausscheiden aus dem Spiel bewahren kann.

Abbildungsverzeichnis

[Abb. 1: Modal-Content 6](#_Toc5545169)

[Abb. 2: Browser-Kompatibilität 7](#_Toc5545170)

[Abb. 3: SVG-CSS 8](#_Toc5545171)

[Abb. 4: Modal-CSS 8](#_Toc5545172)

[Abb. 5: SVG-Filter 9](#_Toc5545173)

[Abb. 6: SVG-Vektoren 9](#_Toc5545174)

[Abb. 7: Mouse-Effects 10](#_Toc5545175)

[Abb. 8: Modal-Funktionen 10](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545176)

[Abb. 9: Registrierungsformular 11](#_Toc5545177)

[Abb. 10: Chat-Layout 12](#_Toc5545178)

[Abb. 11: Game-Layout 13](#_Toc5545179)

[Abb. 12: Chat-Layout Mobiltelefon 14](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545180)

[Abb. 13: Chat-Layout prozentual 14](#_Toc5545181)

[Abb. 14: Text-Child 14](#_Toc5545182)

[Abb. 15: Button-CSS 14](#_Toc5545183)

[Abb. 16: Game-Zyklus 15](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545184)

[Abb. 17: Room-Join 16](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545185)

[Abb. 18: Reihen-Selektor 16](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545186)

[Abb. 19: Library 16](#_Toc5545187)

[Abb. 20: Socket.emit 17](#_Toc5545188)

[Abb. 21: Append-Message 17](#_Toc5545189)

[Abb. 22: grid\_container 18](#_Toc5545190)

[Abb. 23: Libraries 21](#_Toc5545191)

[Abb. 24: Server-Instanzierung 23](#_Toc5545192)

[Abb. 25: Remove-Characters 23](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545193)

[Abb. 26 Login 24](#_Toc5545194)

[Abb. 27: Asynchronität 24](file:///C:\Users\Destati\Documents\GitHub\nodejschat\Single%20Page%20Chat.docx#_Toc5545195)

[Abb. 28: Game\_start 25](#_Toc5545196)

[Abb. 29: checkForStart 25](#_Toc5545197)

[Abb. 30: phase\_vote 26](#_Toc5545198)

Literaturverzeichnis

Cantelon, M. (2013). *Node.js in Action.* Manning Publications.

OpenCollective. (kein Datum). Von https://socket.io/docs/. abgerufen

unbekannt. (17. 2 2016). *W3Schools*. Von http://www.w3schools.com/html/html\_css.asp abgerufen