# Synchronisierte Darstellung hochinteraktiver Webanwendungen

BACHELORTHESIS

LARS KRAFFT
Mtr. Nr.: 2126500

Erstprüfer: Prof. Dr. Edmund Weitz
Zweitprüfer: Prof. Dr. Andreas Plaß

Hochschule für Angewandte Wissenschaften Hamburg

Fakultät: Design, Medien und Information

Department: Medientechnik

Studiengang: Media Systems

# 1 Kurzfassung

## 1.1 Titel der Arbeit

Synchronisierte Darstellung hochinteraktiver Webinhalte

## 1.2 Stichworte

HTML, JavaScript, NodeJS, Websocket, Software-Bibliotheken, SocketIO, MongoDB, HTML-Sync

## 1.3 Zusammenfassung

Im Zuge dieser Arbeit soll die JavaScript Library „HTML-Sync“ entwickelt werden, die das synchronisierte Darstellen hochinteraktiver Webinhalte ermöglicht. Dabei wird besonderer Wert darauf gelegt, vorhandene HTML-Strukturen beizubehalten und so die Layoutprinzipien auch auf dynamischen Websites nutzbar zu machen. Die Library stellt Mittel für die einfache Übertragung von HTML-Strukturen inklusive CSS-Styles und JavaScript Eventhandlern zur Verfügung. Diese Technologie ermöglicht es unter anderem, kollaborative Webanwendungen zu entwickeln.

Um die Funktionsweise der Library abseits der Theorie zu demonstrieren soll im Zuge dieser Arbeit auch ein Beispielprojekt realisiert werden. Hierfür wird der Spiele-Klassiker „TicTacToe“ als interaktive Webapplikation umgesetzt.

## 1.4 Title of this Paper

Synchronized presentation of highly interactive web content

## 1.5 Keywords

HTML, JavaScript, NodeJS, Websocket, Software-Libraries, SocketIO, MongoDB, HTML-Sync

## 1.6 Abstract

This paper describes the development of a JavaScript Library called “HTML-Sync” which allows the synchronous display of highly interactive web content over multiple clients. One of the key goals of this library is to keep existing HTML structures to allow the usage of proven layout principals in highly interactive web environments. The library will provide simple tools to transfer HTML structures including CSS styles and JavaScript event handlers. Amongst other things, this technology will enable the development of collaborative web applications.

To prove the concept in a practical use case, the board game classic “TicTacToe” will be developed as an interactive web application.

# Inhaltsverzeichnis

1 Kurzfassung ..... 1
1.1 Titel der Arbeit ..... 1
1.2 Stichworte ..... 1
1.3 Zusammenfassung ..... 1
1.4 Title of this Paper ..... 1
1.5 Keywords ..... 1
1.6 Abstract ..... 1
2 Einleitung ..... 4
3 Anforderungen an die Bibliothek ..... 5
3.1 Definition und Anforderung an eine Software-Library ..... 6
3.1.1 jQuery ..... 6
3.1.2 Package Manager ..... 6
4 Aufbau von Webservern ..... 7
5 Software Planung ..... 9
5.1 Server- und Clientcode ..... 9
5.2 UML-Diagramm ..... 10
5.3 Aufbau von Node-Modulen ..... 11
5.4 Generierung von IDs ..... 15
5.5 Design Patterns: Observer Pattern ..... 16
6 Verwendete Technologien ..... 18
6.1 NodeJS Server (Express) ..... 18
6.2 Socket.IO ..... 19
6.3 MongoDB ..... 20
6.4 TypeScript ..... 21
7 Aufbau der Bibliothek ..... 22
7.1 Add ..... 23
7.2 Updates ..... 23
7.3 Join ..... 27
7.4 Datenbankanbindung ..... 28
8 Dokumentation ..... 29
9 Beispielanwendung ..... 30
9.1 Beschreibung der Anwendung ..... 30
9.2 Installation ..... 30

9.3 Server ... 33
9.4 Client ... 37
9.5 Testen ... 39
10 Testanwendungen ... 40
10.1 Hello World ... 40
10.1.1 Aufbau der Anwendung ... 40
10.2 Thousand Stars ... 41
10.2.1 Aufbau der Anwendung ... 41
10.3 Dame ... 42
10.3.1 Aufbau der Anwendung ... 42
10.4 Tri ... 44
10.4.1 Aufbau der Anwendung ... 44
10.4.2 Auswertung des Performancetests ... 46
10.5 Yavango DESK ... 47
10.5.1 Aufbau der Anwendung ... 47
11 Fazit ... 50
11.1 Entwicklung ... 50
11.2 Praxis Einsatz ... 50
11.3 Ausblick ... 51
12 Abbildungsverzeichnis ... 52
13 Literaturverzeichnis ... 53

# 2 Einleitung

Seit Jahren werden Webtechnologien wie HTML oder CSS stetig weiterentwickelt. Webseiten werden als eine Folge daraus immer schöner, nutzerfreundlicher und interaktiver. Auch JavaScript wurde stetig erweitert und gibt Entwicklern somit neue Möglichkeiten. Gerade im Bereich der Browser-Spiele zeigt sich eindrucksvoll wozu Webtechnologien heutzutage fähig sind. Sie nutzen aber häufig klassische Herangehensweisen der Spieleentwicklung statt das breite Spektrum an Möglichkeiten von HTML auszuschöpfen.

HTML 5 und die damit einhergehenden Verbesserungen brachten eine Menge neuer Werkzeuge für Webentwickler. Das Canvas-Element zum Beispiel erlaubt das Zeichnen von Pixelgrafiken, wie Entwickler es von je her gewohnt sind, und konnte somit den Flash-Player weitestgehend ablösen.

CSS-Animationen brachten zusätzlich auch den reinen Webentwicklern eine einfache Möglichkeit zur Aufwertung ihrer Designs. Zu guter Letzt ermöglichte die Einführung von Websockets eine Zweiwegekommunikation im Web, die eine viel ausgereiftere Interaktion zwischen den Servern und den angezeigten Websites erlaubt.

All diese neuen Technologien werden dabei noch nicht im großen Stil kombiniert. Websockets werden hauptsächlich dazu genutzt, einige angezeigte Daten aktuell zu halten oder nachzuladen. Eine Ausnahme bieten dabei Online-Spiele, die das Potential der Sockets wesentlich intensiver nutzen, um Spielern einen stetig aktuellen Zustand der Spielwelt anzeigen zu können. Diese Spiele setzen dabei aber im Allgemeinen nicht mehr auf HTML und CSS, sondern verwenden das Canvas-Element oder Browser-Plugins.

Grundsätzlich spricht nichts gegen die Verwendung klassischer Rendertechniken im Web, häufig sind genutzte Elemente aber wesentlich einfacher mit HTML zu realisieren. Gerade bei komplexeren Komponenten wie Text-Input, Radiobuttons und anderen Standard HTML-Elementen ist die Synchronisierung der HTML-Elemente viel einfacher als das Entwickeln eines Objekts, welches die gleichen Eigenschaften besitzt, aber auf ein Canvas-Element gezeichnet werden kann. Diese Arbeit beschäftigt sich aufgrund dieser Vorteile damit, alle erwähnten Technologien zusammenzufassen.

Erreicht wird dieses Ziel durch eine Software-Bibliothek, die den Namen „HTML-Sync“ tragen wird. Die Bibliothek wird Methoden implementieren, welche es erlauben, HTML-Strukturen über das Netzwerk an entfernte Rechner zu übertragen und diese synchron auf einer beliebigen Anzahl von Endgeräten darzustellen.

# 3 Anforderungen an die Bibliothek

Das Synchronisieren von Daten mehrerer Clients erfordert einen Webserver. Dieser kann entweder als Mittelpunkt der Kommunikation agieren und in alle Vorgänge involviert sein, oder nur als Vermittler dienen, um eine direkte Verbindung zwischen Clients aufzubauen. Es ist erforderlich, Code sowohl für den Server als auch für die Clients zu schreiben und diesen in der Bibliothek zusammenzuführen.

Für die Bibliothek HTML-Sync steht Flexibilität stark im Vordergrund. Daher ist es wichtig, so viel Funktionalität wie möglich sowohl dem Server als auch dem Client zur Verfügung zu stellen. Um doppelten Code zu vermeiden, sollten möglichst große Teile des Codes ohne Änderungen sowohl auf Server- als auch auf Clientseite lauffähig sein.

Es soll möglich sein, zu synchronisierende Elemente sowohl beim Start der Anwendung als auch während der Laufzeit hinzuzufügen, zu bearbeiten oder zu löschen.

HTML-Strukturen inklusive CSS und JavaScript sollen einfach erstellt und synchronisiert werden können. Die Synchronisation von nicht an eine Aufgabe gebundenen Daten soll ebenfalls möglich sein. Dies bedeutet, dass die bestehende Websocket-Verbindung dem Entwickler wie gehabt zur Verfügung stehen soll und nicht durch HTML-Sync blockiert sein darf. Außerdem sollen implementierte Systeme auch auf Objekte anwendbar sein, die keine HTML-Strukturen sind.

Die Synchronisation der Elemente soll in weicher Echtzeit erfolgen. Verzögerungen dürfen dabei zwar durch die Netzwerkübertragung und die Serverhardware entstehen, aber nicht durch die Verwendung der Bibliothek. Das Ziel ist, dass ein Server ungefähr die gleiche Anzahl gleichzeitiger Nutzer bedienen kann, als würde er die HTML-Sync Bibliothek nicht verwenden.

Der Webserver soll bei der Interaktion mit Objekten die gleichen Möglichkeiten haben wie die Clients. Diese Anforderung wird durch die umgebungsspezifischen Eigenschaften eingegrenzt. Der Server wird in der Regel keinen Bildschirm haben und muss daher nicht in der Lage sein, Objekte grafisch darzustellen. Diese Einschränkung bedeutet aber nicht, dass der Server nicht in der Lage sein wird, die grafische Darstellung auf Clientseite zu verändern.

Der Server soll Systeme implementieren, welche den Zugriff auf die Elemente regeln und sich widersprechende Anweisungen auswerten und zu einer sinnvollen Lösung finden. Er dient als zentrale Kontrollstelle und stellt einen problemlosen Ablauf der Übertragungen sicher.

Alle vom Server oder von den Clients eingeführten Berechtigungen und Schutzmaßnahmen müssen vom Server überschrieben werden können. Die Flexibilität der Bibliothek ist höher zu bewerten als ein absolut sicheres System. Es soll Entwicklern möglich sein die integrierten Schutzmechanismen zu überschreiben, sofern ihre Anwendung dies erfordert.

## 3.1 Definition und Anforderung an eine Software-Library

Als eine Library oder Bibliothek bezeichnet man bei der Softwareerstellung eine Sammlung von Code, die einen bestimmten Zweck erfüllt und einfach in den eigenen Code integriert werden kann. Ein Beispiel wäre eine Sammlung von mathematischen Formeln. Diese werden in sehr vielen, aber nicht allen Programmen gebraucht. Es ist also sinnvoll, sie als optionalen Bestandteil zur Verfügung zu stellen.

Für den Erfolg einer Library sind drei Faktoren besonders wichtig:

1. Wie umfangreich sind die Funktionen der Library?
2. Wie einfach sind die bereitgestellten Funktionen zu verwenden?
3. Wie einfach kann die Library in den eigenen Code integriert werden?

### 3.1.1 jQuery

Ein Beispiel für eine besonders erfolgreiche Library ist jQuery, denn sie wird auf nahezu jeder Website verwendet. Die Bibliothek lässt sich, wie bei der Programmiersprache JavaScript üblich, mit einer einzelnen Zeile Code einfügen und überschreibt keinen anderen Code, da auf oberster Ebene nur die Funktion „$“ bereitgestellt wird. Diese Funktion erfordert die Verwendung von Selektoren, wie sie Webentwicklern aus der Sprache CSS bereits bekannt sind. Hat man ein Element selektiert, bietet jQuery eine große Vielfalt an Funktionen, die in den meisten Fällen umfangreicher und gleichzeitig einfacher zu verwenden sind als normale JavaScript-Methoden.

jQuery erfüllt also alle drei Anforderungen an eine Library und ist eine gute Orientierungshilfe, sollte man eine Software-Bibliothek entwickeln.

### 3.1.2 Package Manager

Package Manager sind Programme, die Bibliotheken installieren und verwalten und Entwicklern auf diese Weise eine Menge Arbeit ersparen. Hält sich das eigene Projekt an einige grundlegende Strukturen kann man mit Hilfe eines Package Managers wie „npm“ (Nondeterministic Postrequisite Metaprotocol) oder „RubyGems“ Bibliotheken mit einem Konsolenbefehl installieren und sie dann in den eigenen Code einbinden. Neben der erwähnten Zeitersparnis bietet sich dem Verwender der Bibliothek auch der Vorteil, dass Bibliotheken auf den aktuellen Stand gebracht werden können und in einem Umfeld existieren, das dem Entwickler der Bibliothek bekannt ist. Bibliotheken definieren dabei meist selbst, auf welchen anderen Libraries sie basieren, und binden diese bei der Installation automatisch ein.

Der Entwickler der Bibliothek muss sich beim Erstellen des Codes an bestehende Konventionen halten. Außerdem muss er einige zusätzliche Dateien anlegen, die die Art seiner Bibliothek beschreiben, hat aber den Vorteil seinen Code schnell, unkompliziert und sicher an Interessierte ausliefern zu können. Häufig lohnt es sich sogar, innerhalb der eigenen Firma einen Package Manager zu verwenden.

Nutzt man einen Package Manager wie npm und hält man sich an die Konventionen, kann man davon ausgehen das kein Entwickler Probleme haben wird die entwickelte Library in den eigenen Code einzubinden. Allein durch die Verwendung von z.B. npm ist also eine der drei Anforderungen an eine Bibliothek erfüllt.

# 4 Aufbau von Webservern

Wie bereits erwähnt ist für die Kommunikation mehrerer Clients zwingend ein Webserver erforderlich, der entweder im Mittelpunkt der Übertragung steht, oder eine direkte (Peer to Peer) Verbindung vermittelt. Das folgende Kapitel beschäftigt sich deshalb mit der grundlegenden Funktionsweise von Servern.

Als Webserver wird häufig ein Rechner oder ein Rack in einer Serverfarm bezeichnet. In dieser Arbeit soll allerdings nicht die Hardware, sondern lediglich die Software näher untersucht werden. Zu diesem Zweck werde ich die Bezeichnung Webserver für Programme wie den „Apache HTTP Server", die Library „Express" für NodeJS oder „Rails" bzw. „Sinatra" für Ruby verwenden.

Alle genannten Server verwenden das HTTP (Hypertext Transfer Protocol) welches die folgenden zwei Begriffe definiert:

> *„An HTTP "client" is a program that establishes a connection to a server for the purpose of sending one or more HTTP requests. An HTTP "server" is a program that accepts connections in order to service HTTP requests by sending HTTP responses."*
>
> \- *(Reschke 2014)*

Ein Client sendet eine Anfrage an einen Server und erhält eine Antwort. Für den Aufruf einer Website ist dieses Protokoll ausreichend. Der Client fordert zum Beispiel die Seite http://google.de an und erhält den HTML-Code als Antwort. Dieser Code enthält den Pfad zum Logo von Google, das auf der Seite dargestellt werden soll. Der Client schickt nun eine zweite Anfrage an den Server und bekommt das Bild als Antwort.

Das Protokoll erlaubt außerdem verspätete Antworten oder Antworten, die aus mehreren Teilen bestehen. Auf diese Weise können Server programmiert werden, die dem Client jede Sekunde die aktuelle Uhrzeit als Antwort senden. Diese Funktionalität ist dabei aber nie wirklich interaktiv und auch nicht für diese Zwecke vorgesehen. Vielmehr dient sie dazu, große Dateien wie Videos sinnvoll zu übertragen.

Es wird deutlich, dass mit dem klassischen HTTP eine Kommunikation von mehreren Clients nur bedingt und mit viel Aufwand möglich ist. Zwar können beliebig viele Clients zeitgleich auf den Server zugreifen, dieser hat aber in der Regel keine Möglichkeit, einem Client die Daten eines anderen zuzusenden ohne vorher eine erneute Datenanfrage zu erhalten.

Um diese Probleme und unsauberen Lösungen mit dem HTTP zu lösen wurde das Websocket-Protokoll entwickelt. Dieses Protokoll ist dem HTTP in vielen Punkten sehr ähnlich: Es nutzt ebenfalls URLs (Internetadressen), um eine Verbindung aufzubauen, und kann Proxies (Umleitungen) verwenden. Es benötigt keine neue Hardware und bereits installierte HTTP Server müssen lediglich mit einer Softwarebibliothek erweitert werden, um das Protokoll nutzen zu können.

Wie die meisten Netzwerkprotokolle beginnt auch das Socket-Protokoll mit einem Handshake. Dabei teilen sich der Client und der Server gegenseitig mit, dass sie eine Verbindung aufbauen wollen und wie diese Verbindung beschaffen sein soll. Beim HTTP würde nun eine einzelne Antwort gesendet und die Verbindung wieder geschlossen werden. Das Socket-Protokoll hingegen lässt die Verbindung wie einen Tunnel bestehen und erlaubt es sowohl dem Client als auch dem Server, jederzeit Daten zu senden und zu empfangen. Weil ein erneuter Handshake sowie der HTTP-Header (Informationen über die aufzubauende Verbindung) beim Senden der Daten entfallen, ist diese Art der Verbindung sehr schnell und effektiv.

# 5 Software Planung

Nachdem nun die funktionalen Anforderungen an die Bibliothek definiert und die technischen Anforderungen ermittelt wurden, soll in diesem Kapitel die Planung der zu entwickelnden Software erfolgen.

## 5.1 Server- und Clientcode

In den meisten Umgebungen ist es nur bedingt möglich, den gleichen Code sowohl auf dem Server als auch auf dem Client auszuführen, da server- und clientseitig unterschiedliche Programmiersprachen verwendet werden. In einem solchen Fall kann zwar die Signatur einer Klasse zu übernehmen, also Methoden und Variablen mit gleichem Namen zu implementieren, der Code muss in diesem Fall trotzdem zweimal, an die entsprechende Syntax angepasst, geschrieben werden.

Abhilfe schafft in diesem Fall NodeJS *(siehe 6.1 NodeJS Server (Express))*. Mit der Verwendung des gleichen Codes auf Server- und Clientseite ist natürlich auch die Signatur der Klassen gleich. Dieser Vorteil erlaubt es Entwicklern, welche die Bibliothek nutzen möchten, effektiv und intuitiv mit dem bereitgestelltem Code zu arbeiten.

Die Verwendung desselben Codes hat natürlich auch Grenzen wie die Verfügbarkeit der von NodeJS gelieferten Funktionen, die auf der Clientseite fehlen. Bei der Entwicklung von Klassen, die sowohl server- als auch clientseitig verwendet werden sollen muss also darauf geachtet werden, dass keine Abhängigkeiten erzeugt werden, die in einer der beiden Umgebungen nicht vorhanden sind.

Außerdem sind in den unterschiedlichen Umgebungen natürlich nicht alle Funktionen sinnvoll. Eine Klasse, die ein HTML-Element repräsentiert, muss sowohl auf dem Server als auch dem Client vorhanden sein, aber nur der Client kann diese Klasse auch rendern und auf der Website anzeigen. Bei manchen Funktionen ist eine glatte Trennung dabei jedoch nicht möglich. Auf Serverseite wird zum Beispiel nie ein Click-Event ausgeführt, da kein Interface vorhanden ist. Events im Allgemeinen können allerdings durchaus auftreten und müssen deshalb implementiert sein.

Die Verwaltung der verschiedenen Objekte erfolgt auf Server und Clientseite ebenfalls sehr unterschiedlich. Während der Client meist nur in einem „Raum“ tätig ist und eine Anzahl von Objekten mit anderen Clients teilt, muss der Server verschiedene „Räume“ managen und jedem Client nur die für ihn relevanten Objekte senden. Diese Unterschiede führen dazu, dass eine auf den ersten Blick gleiche Funktionalität unterschiedlich umgesetzt werden muss.

Aus Gründen der Übersicht sollte bei der Entwicklung darauf geachtet werden, dass die gleiche Funktionalität auf Client- und Serverseite unter dem gleichem Namen verfügbar ist, auch wenn die Umsetzung der Funktionalität eine andere ist. Gerade wenn Methoden eine Funktionalität aufweisen, die einer anderen nur stark ähnelt, aber nicht identisch ist, kann es auch sinnvoll sein, einen anderen Namen zu wählen, um auf die Unterschiede hinzuweisen und die Erwartungen der Entwickler nicht zu enttäuschen.

## 5.2 UML-Diagramm

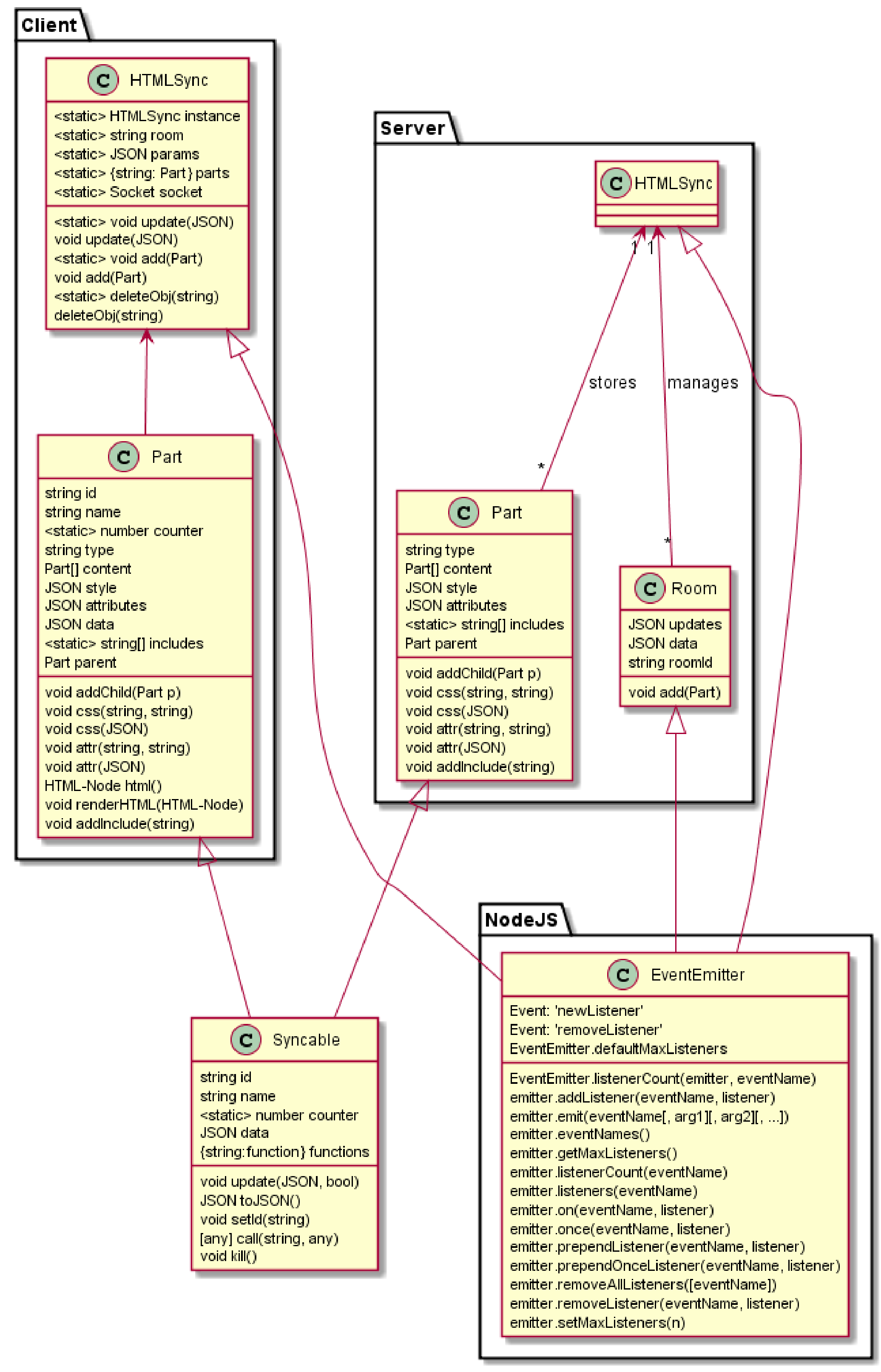


*Abbildung 1: UML-Diagramm*

*Der Bereich NodeJS dieses Diagramms wurde von (NodeJS API 2016) übernommen*

Das UML-Diagramm in Abbildung 1 zeigt den Aufbau der Bibliothek. Es besteht aus einem Server-, einem Client-, einem NodeJS- sowie einem freien Bereich. Im Kern der Bibliothek steht die Syncable-Klasse. Sie stellt ein Objekt dar, welches über eine Socket-Verbindung synchron gehalten werden kann. Das Syncable Objekt ist dabei nicht an eine HTML-Struktur gebunden, sondern implementiert nur Möglichkeiten, Attribute und Funktionen zu synchronisieren. Es ist damit der ideale Ausgangspunkt für Entwickler, welche die Bibliothek nicht ausschließlich zum Synchronisieren von HTML nutzen, sondern auch eigene synchrone Klassen implementieren wollen. Die Syncable-Klasse ist auf dem Server und dem Client vollkommen identisch.

Die Part-Klasse erbt von Syncable und fügt die Funktionalität zum Erzeugen von HTML hinzu. CSS und HTML-Attributparameter werden implementiert, um alle Eigenschaften eines HTML-Objekts abbilden zu können. Zusätzlich bietet der „Parent“-Parameter die Möglichkeit, Part-Objekte genau wie HTML ineinander zu verschachteln. Die Part-Klasse ist auf Server und Client über weite Teile identisch, auf dem Server fehlen jedoch Funktionen zur Übersetzung der Klasse von JavaScript nach HTML, da diese dort nicht benötigt werden.

Die Room-Klasse dient zur Kapselung von Part- und Syncable-Objekten, welche nicht für alle Nutzer der Website relevant sind. Sie implementiert Funktionen zum Speichern und Laden von Objekten und verwaltet diese. Die Room-Klasse wird nur auf dem Server verwendet, da sich Clients in der Regel nur mit einem einzigen „Room“ verbinden und eine Trennung von Elementen daher nicht erforderlich ist.

Die HTMLSync-Klasse ist der Dreh- und Angelpunkt der Bibliothek. Sie stellt die Verbindung zwischen Server und Client her und implementiert die Funktionen zum Synchronisieren von Syncable- und Part-Objekten. Obwohl viele der Funktionen der Klasse auf dem Server und dem Client jeweils den gleichen Namen tragen, ist die Funktionsweise doch sehr unterschiedlich. Die Klasse HTMLSync verwendet daher auf dem Server und dem Client nahezu komplett anderen Code.

Der von NodeJS implementierte EventEmitter ist ebenfalls im UML-Diagramm verzeichnet und wird von den HTMLSync-Klassen sowie der Room-Klasse geerbt. Die Verwendung von Events ist in Bibliotheken sehr wichtig (siehe 5.5 Design Patterns: Observer Pattern). Da auch Eventhandler von Part- und Sync-Objekten übertragen werden müssen, ist es leider nicht möglich, den EventEmitter auch an diese Klassen zu vererben und es muss stattdessen ein eigenes System entwickelt werden (siehe 7.1 Add).

## 5.3 Aufbau von Node-Modulen

Als Serverumgebung für die Bibliothek wurde NodeJS ausgewählt *(siehe Kapitel 6.1)*. Es soll an dieser Stelle also weiter auf die Anforderungen an eine Bibliothek in dieser Umgebung und die Vorteile die sich daraus ergeben eingegangen werden.

Softwarebibliotheken oder Libraries werden bei NodeJS als Module bezeichnet. Ursprünglich ist JavaScript für sehr kleine Programme auf Webseiten entwickelt worden und bietet daher keine direkte Möglichkeit modularisiert zu arbeiten. Pedro Teixeira beschreibt dieses Problem in seinem Buch „Module Patterns – Understanding and using Node module system" mit den folgenden Worten:

*„JavaScript started life without modules being part of the language, being designed for small browser scripts. Historically the lack of modularity led to shared global variables being used to integrate different pieces of code."*

*(Teixeira 2015)*

Anstatt einer sinnvollen Kapselung von Code wurden Werte und Funktionen früher in Form von globalen Variablen gespeichert. Dieses Vorgehen hat den Nachteil, dass zwei Module sich gegenseitig überschreiben können. Gerade bei der Arbeit mit Bibliotheken anderer Entwickler kann dies schwere Folgen haben. So muss der Entwickler entweder die Namen seiner eigenen Variablen ändern, die Bibliothek bearbeiten oder ganz auf die Verwendung der Bibliothek verzichten.

Die Lösung des Problems ist die Nutzung von Scopes. Ein Scope ist ein eingeschränkter Bereich in dem Variablen gültig sind und kann in JavaScript mittels Funktionen erreicht werden.

```
(function(){
    var a = 1;
    var b = 2;

    console.log("In the scope:", a + b);
})();

console.log("Out of scope:", a + b);
```

```
In the scope: 3
Out of scope: <<undefined>>
```

Im obigen Code wird eine anonyme Funktion deklariert und sofort aufgerufen. Die Variablen „a" und „b" sind dadurch vom restlichen Code getrennt. Sie können nicht gelesen werden, überschreiben aber auch keine bereits vorhandenen Variablen mit den gleichen Namen.

Eine Bibliothek, auf deren Werte und Funktionen man nicht zugreifen kann, ist nur in den wenigsten Fällen sinnvoll. Über den return-Befehl kann ein Objekt zurückgegeben werden, dass die implementierten Funktionen und Variablen bereitstellt.

```
var Circle = (function(){
    var PI = 3.141592653589793;

    return {
        area: function(radius){
            return PI * radius * radius
        }
    }
})();

console.log("Area:",Circle.area(2));
```

```
Area: 12.566370614359172
```

Auch außerhalb der anonymen Funktion kann nun auf die Methode „area" zugegriffen werden. Die Variable „PI" ist nur innerhalb des Moduls verfügbar. Diese Herangehensweise hat einen entscheidenden Vorteil gegenüber einem normalen Objekt. Der Entwickler kann selbst festlegen, wie die Variable heißen soll, die ihm die exportierten Methoden zur Verfügung stellt.

Nehmen wir einmal folgendes Beispiel an:

**Circle.js**

```
(function(){
    var PI = 3.141592653589793;

    return {
        area: function(radius){
            return PI * radius * radius
        }
    }
})();
```

**Script.js**

```
var Circle = {
    radius: 2,
    color: "blue"
};

var CircleUtil = require('./circle.js');

console.log("Area:", CircleUtil.area(Circle.area));
```

```
Area: 12.566370614359172
```

In diesem Beispiel soll die „require"-Methode eine Datei einlesen und den JavaScript-Code ausführen. Dieses Beispiel zeigt sehr schön, dass obwohl die Variable „Circle" bereits vergeben ist, das Modul „Circle.js" dennoch problemlos eingebunden werden kann. Der Entwickler ist vollkommen frei in seiner Namensgebung und braucht sich keine Sorgen darüber zu machen, dass seine Variablen durch ein Modul überschrieben werden.

An dieser Stelle sollte auch erwähnt werden, dass die „require“-Methode in JavaScript nicht definiert ist. Die Bibliothek RequireJS (früher CommonJS) implementiert sie allerdings und macht es so möglich, auch mit JavaScript modularisiert zu arbeiten.

Auch das Modulsystem in NodeJS basiert auf CommonJS. Bei der Verwendung dieses Systems müsssen einige Feinheiten beachtet werden, da es neben dem einfachen Einlesen und Ausführen von Quellcode auch noch andere Aufgaben übernimmt. So stellt NodeJS zum Beispiel sicher, dass Code nicht mehrfach geladen wird und Zirkelbezüge nicht zu Endlosschleifen führen (siehe Abbildung 2: Darstellung eines Zirkelbezugs).

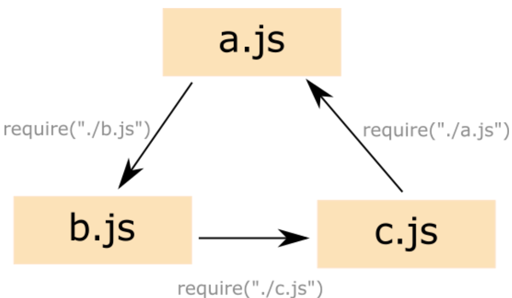


*Abbildung 2: Darstellung eines Zirkelbezugs*

## 5.4 Generierung von IDs

IDs sind Zahlen oder andere Zeichenketten, die es ermöglichen ein Objekt in einer Sammlung von Objekten zu finden. Die wichtigste Anforderung an eine ID ist dabei, dass sie eindeutig ist, d.h. dass keine zwei Objekte der gleichen Zeichenkette zugeordnet sind.

IDs kommt in Systemen, welche über mehrere Clients verteilt sind, eine besondere Bedeutung zu. Jeder Client sowie der Server müssen eine eigene Version eines synchronisierten Objekts erstellen, um es anzeigen zu können. Wenn also beispielsweise fünf Clients mit einer Website mit nur einem synchronisierten Objekt verbunden sind, liegen tatsächlich sechs Objekte vor: Fünf bei den Clients und eines auf dem Server. Um diese Objekte miteinander zu verbinden, müssen sie alle die gleiche ID erhalten. Wie oben erwähnt können nicht zwei Objekte mit derselben ID existieren. Haben zwei Objekte also die gleiche ID, muss es sich dabei - zumindest aus logischer Sicht - um das gleiche Objekt handeln.

Eine einfache Möglichkeit Objekten eine ID zuzuweisen, ist ein Zähler, der sich mit jedem neuen Objekt erhöht. Bei Elementen unterschiedlicher Art klönnen zusätzlich noch ein Buchstabe oder eine zusätzliche Zahl den Objekttyp angeben. Programmiert werden solche Zähler mit einer statischen Integer-Variablen, die im Konstruktor der Klasse erhöht wird.

Für Programme mit mehreren Clients, die getrennt voneinander Objekte erstellen, ist diese Herangehensweise allerdings nicht möglich, da der Zähler bei jedem Client wieder bei null anfangen würde und doppelte IDs entstünden.

Es ist auch möglich, einen Zeitstempel als ID zu verwenden. Jeder Zeitpunkt ist eindeutig und die Zeit schreitet gleichmäßig voran. Zeitstempel können also im Prinzip nicht doppelt auftreten. Ein Problem hierbei stellt die Genauigkeit da. Falls die ID nach dem Muster 28.04.2016 11:01:35 aufgebaut ist, könnte innerhalb einer Sekunde eine weitere ID mit dem gleichem Stempel erzeugt werden. Der Stempel 11:01:35:200 wäre zwar auf die Millisekunde genau, würde aber jeden Tag aufs Neue auftreten. Zwar ist die Wahrscheinlichkeit gering, dass die Zeit auf die Millisekunde genau am folgenden Tag wieder abgefragt wird, aber sie besteht und kann zu Fehlern im Code führen.

Die meisten Programmiersprachen implementieren einen Zeitstempel, der die Millisekunden zählt, die seit dem 01.01.1970 vergangen sind. Dieses Datum ist mehr oder weniger willkürlich gewählt und hat für die Funktionsweise des Vorgangs keine Bedeutung. Die Methode liefert nun also einen Integer, der sich pro Millisekunde um eins erhöht. Für menschliche Eingaben ist diese Genauigkeit in den meisten Fällen ausreichend, um doppelte IDs zu vermeiden, aber häufig werden Objekte in Schleifen innerhalb des Programms erstellt was erneut zu doppelten IDs führen kann.

Die Lösung des Problems besteht darin, beide Verfahren zu kombinieren. So kann man an den Zeitstempel einfach die Zahl des Objektzählers anhängen und somit die Vorteile beider Vorgehensweisen ausnutzen. Wichtig ist dabei zu beachten, dass die Zahlen nicht addiert werden, da der Zeitstempel dadurch ein Stück weit in der Zukunft liegen würde und somit eine Überlagerung von IDs theoretisch möglich wäre. Besser ist es, die Zahlen als Zeichenketten zu verbinden, da auf diese Weise zwei Systeme parallel doppelte IDs vermeiden können.

Mit dem beschriebenen System ist es also möglich, IDs zu generieren, die für jeden Client unterschiedlich sind und eine hohe Wahrscheinlichkeit bieten auch über mehrere Clients eindeutig zu sein. Da in der Regel aber mehrere Nutzer gleichzeitig arbeiten werden, ist

anzunehmen, dass die Zeitstempel nahe beieinanderliegen. Sollten zwei Clients die gleiche Anzahl an Objekten erstellt haben, könnte es nach wie vor zu überlagerten IDs kommen. Ohne eine Synchronisation der IDs über den Server lässt sich dies auch nicht vermeiden, es ist aber möglich, die Wahrscheinlichkeit weiter zu senken.

Für die IDs der HTML-Sync Library wird ein zusätzlicher, zufälliger Buchstabe der ID vorangestellt. Die Wahrscheinlichkeit einer doppelten ID wird damit nochmal für jeden Buchstaben des Alphabets reduziert. Zusätzlich macht es ein Buchstabe den Entwicklern einfacher, ihre Anwendungen zu testen, da die Zeichen eine viel angenehmere Möglichkeit bieten Objekte zu finden als reine Zahlenketten.

## 5.5 Design Patterns: Observer Pattern

*„The Observer is a design pattern in which an object (known as a subject) maintains a list of objects depending on it (observers), automatically notifying them of any changes to state."*

*(Osmani n.d.)*

Design Patterns sind Lösungen für immer wieder auftretende Probleme der Softwareentwicklung. Das Observer Pattern findet heute in JavaScript vor allem in einer leicht abgewandelten Form Anwendung: Bekannt unter dem Namen Publish/Subscribe Pattern wird es zum Beispiel bei Events im DOM (Document Object Model) verwendet und ist somit die Hauptschnittstelle zwischen HTML und JavaScript. Für Software-Bibliotheken spielt es eine wichtige Rolle, da es Entwicklern erlaubt, auf Ereignisse innerhalb der Bibliothek zu reagieren, ohne den Code der Bibliothek anpassen zu müssen.

Das Pattern erlaubt es, beliebig viele „Subscriber" (Abonnenten) an einen „Publisher" (Herausgeber) zu binden und diese auf spezielle Events warten zu lassen. Das wohl bekannteste Beispiel für dieses Pattern ist ein Button, der als Publisher fungiert und seine Subscriber über ein Klick-Event informiert.

```
<button id="btn">Klick mich</button>

<script>
    document.getElementById("btn").addEventListener("click",
function(){
        console.log("Der Button wurde geklickt.");
    });
</script>
```

Ein Grund für die Verbreitung des Pattern in JavaScript, ist die Möglichkeit Funktionen als Argumente an Methoden zu übergeben. Dies erleichtert eine Implementierung des Design Patterns deutlich.

Für Softwarebibliotheken hat das Pattern eine besondere Relevanz, weil es Nutzern der Library ermöglicht auf Vorgänge innerhalb der Bibliothek zu reagieren. In der Bibliothek „Socket.IO" gibt es zum Beispiel das „Join"-Event, dass immer dann aufgerufen wird, wenn sich ein Client mit dem Server verbindet. Beim Aufbau dieser Verbindung wird innerhalb der Library alles für den reibungslosen Ablauf der Socket-Verbindung vorbereitet.

Jeder Entwickler, der die Library verwendet, hat unterschiedliche Wünsche bezüglich dessen, was beim Verbindungsaufbau außerdem noch geschehen soll. Der Client könnte zum Beispiel begrüßt werden, oder die Verbindung könnte sofort unterbrochen werden, falls ein Passwort nicht übermittelt wird.

Das Publish/Subscribe Pattern erlaubt es, allen Anforderungen zu genügen, ohne dass Entwickler in den Code der Library eingreifen müssen. Sie „subscriben" einfach dem „Join"-Event und lassen ihren eigenen Code ausführen sobald sich ein Client verbindet.

Für eine Library wie HTML-Sync, die zu großen Teilen auf asynchronen Abläufen basiert, sind Events ebenso wichtig. Ohne sie wäre es nicht möglich, auf ein gerade hinzugefügtes Objekt zu reagieren, ohne in den Code der Library einzugreifen.

Das Publish/Subscribe Pattern ist mit seinen drei Grundfunktionen „subscribe", „unsubscribe" und „publish" einfach zu implementieren. Es gibt allerdings auch Funktionen innerhalb von NodeJS und jQuery, die das Pattern umsetzen und dabei eine Anzahl nützlicher Zusatzfunktionen bieten. Für die HTML-Sync Library werden diese externen Bibliotheken verwendet, da sie im Falle von NodeJS ohnehin vorhanden und im Falle von jQuery sehr weit verbreitet sind. Es sollten deshalb keine Kompatibilitätsprobleme auftreten.

# 6 Verwendete Technologien

Als Basis für die Library dient ein NodeJS Server, der eine Socket.io Verbindung mit dem Client herstellt. Optional kann eine MongoDB Datenbank an den Server angebunden werden. Der Client nutzt TypeScript für die Bereitstellung einer wohldefinierten Klassenstruktur.

## 6.1 NodeJS Server (Express)

*"Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications."*

\- *(Expressjs.com 2016)*

Anders als die am meisten verbreitete Programmiersprache für Webserver, PHP, arbeitet NodeJS komplett eventbasiert. Diese Herangehensweise erlaubt das parallele Abarbeiten von Aufgaben ohne auf zusätzliche Threads zurückgreifen zu müssen. Weil PHP als Scriptsprache darauf angewiesen ist von einer weiteren Softwarelösung wie einem Apache Server interpretiert zu werden, kann es zu Problemen bei vielen gleichzeitigen Anfragen kommen. Für jede Anfrage muss die PHP-Datei von der Festplatte geladen und in ein HTML-Dokument umgewandelt werden. Diese Aufgaben benötigen kostbare Zeit und sind in PHP nur über mehrere Threads parallelisierbar. Durch Limitierungen der Hardware ist die Anzahl gleichzeitiger Threads allerdings begrenzt und so kann es bei großer Serverlast zu erheblichen Wartezeiten kommen.

NodeJS ist keine Scriptsprache sondern selber ein Webserver. Im Allgemeinen wird dieser Server in nur einem Thread ausgeführt, kann aber durch die Verwendung von Events trotzdem parallel arbeiten. Wird eine Datei von der Festplatte angefragt, kann NodeJS sich ohne Wartezeit mit den nächsten Anfragen beschäftigen, erhält einen Callback sobald die Datei bereit ist und kann die Anfrage zu Ende abarbeiten. Durch dieses Vorgehen wird die CPU des Servers wesentlich gleichmäßiger ausgenutzt. Natürlich kommt es auch bei NodeJS unter hoher Last zu erhöhten Wartezeiten. Diese blockieren aber niemals den kompletten Server und alle Aufgaben werden ordnungsgemäß, wenn auch langsamer ausgeführt.

Weil NodeJS ein Webserver ist, kann es außerdem den Arbeitsspeicher sinnvoll ausnutzen. In PHP kann eine PHP-Datei nie auf die Variablen einer anderen zugreifen, da alle Daten nach Auslieferung der fertigen Website wieder verworfen werden. Jede Anfrage ist letztlich ein eigenes Programm. In NodeJS hingegen werden alle Anfragen von demselben Programm bearbeitet, Variablen können somit geteilt werden. Dies ist vor allem bei kleinen Aufgaben interessant für die sich das Anlegen einer Datenbank nicht lohnt und um häufig benötigte Werte schneller verfügbar zu machen. Wird zum Beispiel in jeder Abfrage ein Wert aus einer bestimmten Datei benötigt kann man diese Datei in NodeJS beim Start des Servers laden und den Inhalt im Arbeitsspeicher vorhalten, um ständigen Zugriff auf die Festplatte zu vermeiden.

Ein entscheidender Nachteil von NodeJS ist die Fehleranfälligkeit. Während in PHP im schlimmsten Fall eine Seite nicht ausgeliefert wird, kann bei NodeJS ein Fehler in der Programmierung zum Absturz des gesamten Servers führen. Es können zwar Vorkehrungen getroffen werden um solche kritischen Fehler zu vermeiden, jedoch müssen diese immer vom Entwickler bedacht werden. Gerade bei großen Projekten, in denen auch viele externe Module verwendet werden, ist dies nicht immer möglich.

Für die Library HTML-Sync hat NodeJS den Vorteil, mit nur einer Programmiersprache sowohl server- als auch clientseitig arbeiten zu können. JavaScript definiert seine Objekte in einem Format, dass JSON sehr ähnlich ist und welches ausgenommen von Methoden problemlos per Websocket übertragen werden kann. Die Verwendung von JavaScript auf der Serverseite erlaubt es daher, die vom Client erstellten Objekte ohne Probleme auch auf dem Server zu nutzen.

Die Library Express ist eine der am häufigsten verwendeten Libraries zum Erstellen eines Webservers mit NodeJS. Viele Bibliotheken arbeiten Hand in Hand mit ihr zusammen und sie bietet eine gute Mischung aus Kontrolle über die Abläufe auf dem Server und Vereinfachung der komplexen Anforderungen an einen HTTP-Server.

## 6.2 Socket.IO

*"Socket.IO enables real-time bidirectional event-based communication.*

*It works on every platform, browser or device, focusing equally on reliability and speed."*

- *(Socket.io 2016)*

Socket.IO ist eine Library, die das Versenden von Daten über das Websocket Protokoll erlaubt. Die Bibliothek setzt dabei auf ein Eventsystem, sodass dem Entwickler das Anlegen eines eigenen Protokolls für den Datenaustausch weitestgehend abgenommen wird. Socket.IO versendet seine Daten im JSON-Format, welches von JavaScript nativ unterstützt wird. Eine komplexe Konvertierung von Daten ist daher nicht erforderlich.

Der wohl größte Vorteil von Socket.IO dürfte die Abwärtskompatibilität sein. Nicht nur das Websocket-Protokoll wird von Socket.IO implementiert, sondern auch *„WebSockets, Adobe Flash Socket, AJAX long polling, AJAX multipart streaming, Forever Iframe and JSONP Polling“*- (Höfler 2013). Dabei wird automatisch das effektivste verfügbare Protokoll ausgewählt, ohne dass sich für den Entwickler Änderungen ergeben.

Durch die hervorragende Kompatibilität, die vor allem in der Anfangszeit der Websockets große Bedeutung hatte, konnte sich Socket.IO als Standard für Websocket-Aufgaben durchsetzen. Obwohl die Unterstützung von Sockets heute in allen Browsern gegeben ist, greifen noch immer viele Entwickler auf die Bibliothek zurück. Das npm Modul von Socket.IO wurde laut der offiziellen Website[1] im Monat Mai ca. 3,3 Millionen Mal herunter geladen. Zudem zählt es zu den am häufigsten für Module verwendeten Bibliotheken im npm.

Trotz dieser großen Verbreitung ist es schwer, kritische Quellen und Analysen zu Socket.IO zu finden. Während NodeJS oder auch npm zum Teil harter Kritik ausgesetzt sind, scheint die Entwicklerwelt mit Socket.IO durch und durch zufrieden zu sein.

[1] https://www.npmjs.com/package/socket.io-client

## 6.3 MongoDB

*„ MongoDB stores data using a flexible document data model that is similar to JSON. Documents contain one or more fields, including arrays, binary data and sub-documents. Fields can vary from document to document."*

- *(MongoDB, Inc 2016)*

Die MongoDB ist eine sogenannte NoSQL-Datenbank und speichert Daten nicht, wie klassische SQL-Systeme, in festen Tabellen, sondern in einem JSON-kompatiblen Format. Die variablen Dokumente der MongoDB erlauben es, ganz unterschiedliche Elemente in derselben Datenbank zu speichern. Zusammen mit der Möglichkeit Sub-Dokumente in Dokumenten zu speichern ist MongoDB die perfekte Lösung um die vom Client gesendeten JSON-Daten langfristig zu speichern.

Ein Nachteil von NoSQL-Datenbanken ist, dass sie mehr Speicherplatz benötigen um Datensätze zu speichern. In seiner Thesis „MongoDB An introduction and performance analysis" erklärt Rico Suter diesen Nachteil wie folgt:

*„Another drawback is the fact that it uses much more storage space for the same data then for example PostgreSQL. Because – as opposed to relational databases – every document can have different keys the whole document has to be stored, not only the values."* - (Suter 2012)

Durch den flexiblen Aufbau der Datenbank ist es nicht länger ausreichend die Werte der einzelnen Felder zu speichern. Auch der Aufbau des Dokuments und die Schlüssel, unter denen die Werte gespeichert sind, müssen in der Datenbank für jedes Dokument vorhanden sein.

Um den benötigten Speicherplatz so gering wie möglich zu halten, wird empfohlen möglichst kurze Bezeichnungen für die Schlüssel zu verwenden. Gerade in größeren Projekten kann dies aber zu unleserlichem Code und Übersichtsproblemen führen. Bibliotheken wie „MongoMinify"[2] können als zusätzlichen Schritt beim Schreiben und Lesen der Datenbank die Schlüssel in möglichst kurze Zeichenketten konvertieren und so sowohl Speicherplatz sparen als auch einen lesbaren Code ermöglichen.

Transaktionen sind in MongoDB nicht im eigentlichen Sinne möglich. Als Transaktion bezeichnet man eine Kette von Befehlen, die die Datenbank in einem konsistenten Zustand zurücklässt. Konsistent bedeutet hierbei, dass es unter anderem keine Referenzen zu gelöschten Elemente gibt. In MongoDB gibt es nur einige wenige „atomic operations", die in ihrer Funktionsweise einer Transaktion entsprechen. Auch wenn Transaktionen in jeder Datenbank von Bedeutung sein können, werden sie in NoSQL-Datenbank wesentlich seltener gebraucht. Insbesondere in einer MongoDB, welche verschachtelte Dokumente erlaubt sind Referenzen auf andere Tabellen und Dokumente weit weniger wichtig als in den klassischen SQL-Datenbanken.

[2] https://github.com/marcqualie/mongominify

## 6.4 TypeScript

*„TypeScript is a typed superset od JavaScript that compiles to plain JavaScript."*

– *(Microsoft 2016)*

TypeScript ist eine Entwicklung von Microsoft und führt Klassen, wie man sie aus gängigen typisierten Sprachen kennt, in JavaScript ein. Der geschriebene Code wird dann in reines JavaScript transpiliert und kann wie jede andere Library in Webseiten integriert werden.

Für die HTML-Sync Bibliothek ist eine Typisierung besonders interessant, da sie nicht nur Funktionen sondern auch Klassen zur Verfügung stellt. Die Part- und die Syncable-Klasse profitieren dabei immens von der einfachen Möglichkeit der Vererbung. Die Syncable-Klasse ist schon im Konzept auf Vererbung ausgelegt. Sie implementiert nur die nötigsten Funktionen um die Synchronisierung eines Objekts zu ermöglichen und kann so an beliebige andere Klassen vererbt werden um auch für diese eine Synchronisation zu erlauben. Die Part-Klasse ist zwar weit weniger flexibel ausgelegt, durch Vererbung kann hier allerdings sehr gut lesbarer Code erstellt werden. Die Möglichkeit einem Part-Objekt weitere Part-Objekte als Kind-Elemente hinzuzufügen, ermöglicht es, komplexe Part-Strukturen in einer Klasse zusammenzufassen.

Da JavaScript im Grundsatz nicht typisiert ist, sind alle in TypeScript definierten Regeln optional und dem Entwickler werden praktische Werkzeuge gegeben, optionale Parameter oder verschiedene Eingabetypen zu definieren ohne dabei eine Methode überladen zu müssen.

Die Verwendung von TypeScript ist durch diese Vorteile aber noch keine Selbstverständlichkeit. Der neue JavaScript-Standard ES6 bietet viele der Funktionen die in TypeScript implementiert sind. ES6 wird zurzeit (Jahr 2016) noch nicht von Browsern unterstützt und erfordert daher, genau wie TypeScript, einen Transpiler um es in den aktuellen Standard ES5 zu konvertieren. In Zukunft wird dieser Schritt allerdings entfallen und einen Transpiler für ES6 überflüssig machen.

TypeScript ist in der Lage alle bestehenden JavaScript Bibliotheken zu nutzen. Um während der Entwicklung hilfreiche Hinweise von der IDE zu erhalten sind allerdings „Definition-Files" erforderlich, welche die Signaturen der Library in TypeScript definieren. Eine lange Liste solcher Dateien kann zwar auf [http://definitelytyped.org/](http://definitelytyped.org/) heruntergeladen werden, diese Dateien sind aber von der Community und nicht von den eigentlichen Entwicklern der Bibliothek erstellt worden und können somit Fehler aufweisen.

Auch TypeScript selber hat einige bekannte Fehler bezüglich seiner Sourcemaps. Diese Dateien erlauben es Entwicklern TypeScript Code zu debuggen, indem sie Codestellen des generierten JavaScript-Codes auf den programmierten TypeScript Code übertragen. Werden die Sourcemaps nicht korrekt erstellt werden Fehlermeldungen falsch ausgegeben, was zur längeren Suche von Bugs führen kann. Diese Fehler wiegen nicht sehr schwer, da auch der eigentliche, von TypeScript erstellte JavaScript-Code für Entwickler gut lesbar ist und Sourcemaps somit nicht zwangsläufig für das Debugging erforderlich sind.

# 7 Aufbau der Bibliothek

Der NodeJS-Server stellt HTML, CSS und JavaScript Dateien bereit, welche über einen Webbrowser abgerufen werden können. Außerdem startet er einen Socket.io Server mit dem sich die Clients beim Laden der Website verbinden. Der Server implementiert zusätzlich Funktionen um eine Trennung von Räumen zu ermöglichen. Räume sind ein gängiger Weg Gruppen von Nutzern zu trennen, obwohl sie den gleichen Server verwenden. Ein häufiges Beispiel sind Chaträume. Der Beispielserver „http://tollechats.de" könnte die Räum „Hobbygärtner" und „Hobbyprogrammierer" anbieten. Beide Gruppen von Nutzern würden zwar auf denselben Server zugreifen, können und müssen durch die Trennung der Räume aber nicht miteinander kommunizieren. Außerdem bietet der Server Möglichkeiten synchronisierte Elemente in eine Datenbank zu speichern. Dabei ist zu beachten, dass es ausreichend ist die Elemente mit dem Server zu synchronisieren, ein zweiter Client ist dafür nicht erforderlich. Die hier beschriebene Bibliothek ist damit nicht nur dann interessant wenn mehrere Clients kooperativ an etwas arbeiten sollen, sondern auch wenn komplexe Strukturen wie zum Beispiel eine Mindmap für einen Nutzer auf dem Server gespeichert werden soll.

Clients verbinden sich mit dem Socket.io Server und betreten einen virtuellen Raum. Der Name des Raumes kann dabei immer derselbe sein, vom Nutzer gewählt werden oder über einen Algorithmus automatisch bestimmt werden. Nur Nutzer, die sich im gleichen Raum befinden können miteinander kommunizieren und ihre Elemente synchronisieren. Es ist also möglich alle Besucher einer Website zusammen arbeiten zu lassen, kleinere Gruppen von Nutzern zu bilden oder einzelne User allein mit dem Server kommunizieren zu lassen.

Die Bibliothek implementiert unter anderem die drei grundlegenden Methoden, die zur Übermittlung von HTML-Elementen benötigt werden.

**add:** Versendet ein komplettes HTML-Konstrukt mit Styles, Event-Handlern und Kind-Elementen.

**update:** Versendet einzelne Bestandteile eines HTML-Konstrukts, die sich bei allen Clients ändern sollen.

**delete:** Löscht ein HTML-Konstrukt bei allen Clients.

Die Daten werden dabei im JSON-Format übertragen in dem auch komplizierte Schachtelungen, wie sie bei HTML üblich sind, problemlos abgebildet werden können. Erforderlich in jedem übertragenen JSON-Paket sind die „id" (Identifikationsnummer) des Objekts, das hinzugefügt, geändert oder gelöscht werden soll, sowie die „roomId" (Identifikationsnummer des Raumes) für den das Paket relevant ist.

Um die HTML-Strukturen der Website für die Übertragung vorzubereiten implementieren die Clients außerdem die Part-Klasse. Die Part-Klasse ist ein Abbild einer HTML-Struktur als JavaScript Klasse. Sie bietet alle benötigten Komfort-Funktionen um Strukturen angenehm definieren und übertragen zu können und orientiert sich dabei stark an jQuery-Objekten, die den meisten Webentwicklern bekannt sein dürften.

Über eine Funktion kann die Part-Klasse in ein HTML-Konstrukt umgesetzt und dem Dokument hinzugefügt werden. Werden Änderungen an dem Konstrukt vorgenommen, die synchronisiert werden sollen, müssen diese Änderungen auch an der Part-Klasse vorgenommen werden. Über die IDs ist es einfach möglich eine Verbindung zwischen HTML-Element und Part herzustellen.

Die Update-Funktion der Part-Klasse sendet ein JSON-Objekt zum Server und dieser verteilt es an alle Clients im gleichen Raum.

## 7.1 Add

Die Add-Methode wird verwendet um Objekte dem HTML-Dokument hinzuzufügen und diese zu synchronisieren. Dafür wird das Objekt zuerst in ein JSON-Format konvertiert. Die meisten Attribute der für die Synchronisation erforderlichen Klassen lassen sich Problemlos in JSON speichern. Funktionen werden von JSON allerdings nicht unterstützt aus diesem Grund müssen sie in Strings konvertiert und beim Empfänger mit dem „eval“-Befehl erneut interpretiert werden.

Funktionen nicht mit JSON zu übertragen, sondern lokal Klassen zu initialisieren, welche diese Funktionen implementieren, ist keine gute Lösung, da dies voraussetzt, dass sowohl für alle Clients als auch den Server eine Definition für die entsprechende Klasse zur Verfügung steht. Dies ist Grundsätzlich dank NodeJS zwar möglich, erfordert aber umständliche Codestrukturen um sowohl als JavaScript-Klasse als auch als NodeJS-Modul funktionieren zu können. Zudem wäre es nicht möglich die implementierten Funktionen zur Laufzeit synchron zu ändern. Dies ist zwar keine Kernfunktion kann aber je nach Anwendungsfall durchaus relevant sein.

Erhält der Server ein JSON-Paket mit einem Add-Befehl leitet er dieses an alle Clients des Raums weiter. Um auch Clients eine synchrone Darstellung der Website zu ermöglichen, welche die Seite später besuchen, müssen alle hinzugefügten Objekte auch auf dem Server gespeichert werden. Werden keine weiteren Schritte unternommen, geschieht dies mit einer Hashmap im entsprechenden Raum. Dies hat zur Folge, dass der Zustand der Website nur „flüchtig“ gespeichert wird. Sollte der Server neugestartet werden, werden alle Variablen aus dem Arbeitsspeicher gelöscht und der Zustand aller Räume geht verloren. In vielen Fällen ist dies kein Problem, da der Raum nur kurze Zeit genutzt wird. Sollte der Zustand des aktuellen Raums auch nach längerer Zeit wiederherstellbar sein, muss er in einer Datenbank gespeichert werden.

## 7.2 Updates

Von den drei oben angesprochenen Grundmethoden „add“, „update“ und „delete“ wird die „update“-Methode wohl am häufigsten verwendet werden. Sie ist zudem die Methode, welche die höchsten Ansprüche an eine schnelle Verarbeitung hat.

Eine häufig benötigte Funktion ist das synchrone Bewegen von Elementen. Klickt ein Nutzer ein Element der Website und zieht es, treten je nach Geschwindigkeit der Bewegung in einer Sekunde zwischen 50 und 130 „Drag“-Events im Browser auf, die es erlauben eine fließende Bewegung darzustellen. Erste Tests haben gezeigt, dass diese Anzahl von Events kein Problem für NodeJS und Socket.IO darstellen.

Probleme gibt es stattdessen an anderer Stelle. Sendet man einen Update Befehl an den Server und wartet auf die Antwort ohne die Änderungen direkt auszuführen wirken Bewegungen schwammig und haben eine Verzögerung. Nimmt man hingegen Änderungen sofort vor und sendet dann den Update-Befehl darf der Server diesen nicht einfach zurück

senden wie es zum Beispiel bei der „Add“-Funktion der Fall ist. In einem solchen Fall würde das bewegte Objekt anfangen zu ruckeln, da die lokalen Änderungen nach einer kurzen Verzögerung mit den Änderungen des Servers überschrieben werden würden.

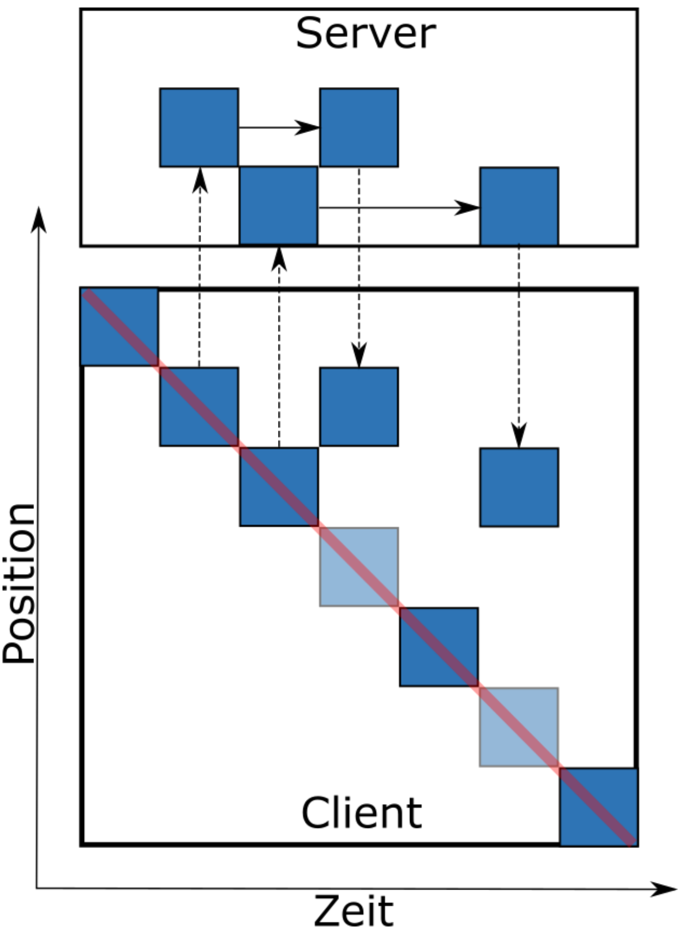


*Abbildung 3: Ablauf eines Drag-Events bei fehlerhaftem Update-Verhalten*

Dies wird in Abbildung 3 deutlich. In der Grafik wird das Quadrat über einen Zeitraum (horizontale Achse) nach unten geschoben. Die Änderungen werden dem Server mitgeteilt und dieser sendet sie an alle Clients im Raum. Je nach Dauer der Übertragung wird die gewünschte Bewegung (rote Line) unterschiedlich stark beeinflusst. Je nach Programmierung kann es sogar passieren, dass die vom Server übermittelte Position als neue Ausgangsposition herangezogen wird und sich die Fehler somit addieren.

Die Lösung für das beschriebene Problem ist den Server die eintreffenden Update-Anweisungen nicht wieder an denselben Client zurück senden zu lassen, sondern diese nur an alle anderen Clients zu verteilen. Dieses Vorgehen sorgt für einen etwas größeren Unterschied zwischen den einzelnen Clients, da der Client der die Änderung vorgibt nicht von der Latenz zwischen Client und Server betroffen ist.

Bei allen Anwendungen für kooperatives Arbeiten stellt sich die Frage: „Was passiert wenn zwei Nutzer den gleichen Bereich verändern?“. In komplexen und spezialisierten Systemen gibt es hierfür angepasste Lösungen. Ein Dokument wird zum Beispiel vollständig gesperrt solange es bearbeitet wird und in einem Online Spiel sind die Clients der Reihe nach in der

Lage mit dem Spiel zu interagieren. Diese Lösungsansätze sind für HTML-Sync nicht praktikabel, da sie die Library zu stark eingrenzen würden.

Eine Lösung ist aber erforderlich. Am Beispiel der Bewegung eines Elements könnte es sonst, wie bereits oben erwähnt zu flackern kommen, weil das Element zwischen zwei oder mehr unterschiedlichen Inputs hin und her springen würde.

In HTML-Sync wurde um dieses Problem zu lösen serverseitig ein Schloss eingebaut, welches Updates nur zulässt, sollte der passende Schlüssel geliefert werden. Dieses Schloss wird wenige Millisekunden nach dem letzten Update automatisch entfernt.

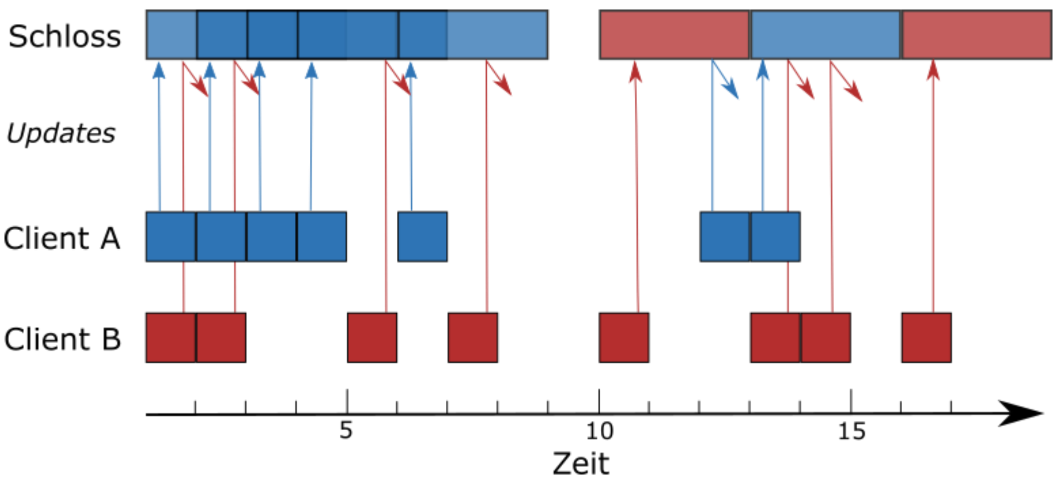


*Abbildung 4: Zeitlicher Ablauf mehrerer Updates desselben Objekts*

Abbildung 4 zeigt das Verhalten des Servers wenn mehrere Clients versuchen dasselbe Objekt zu verändern. Obwohl im ersten Schritt Client A und Client B quasi zeitgleich ein Update senden, muss eines der beiden zuerst abgearbeitet werden. Dies geschieht automatisch, da JavaScript in einem einzigen Thread arbeitet und parallele Ausführung von Code nicht möglich ist. In diesem Fall wird das Update von Client A zuerst verarbeitet und ein Schloss vor dem Objekt erstellt. Der Schlüssel für ist die ID des Sockets von Client A somit ist nur noch dieser Client berechtigt Änderungen an dem Objekt vorzunehmen. Das Update von Client B wird verworfen.

Das Schloss ist für drei Zeiteinheiten gültig, wird in Abbildung 4 aber im zweiten, dritten, vierten und sechsten Schritt erneuert, da ein weiteres Update von Client A gesendet wurde. Dies führt dazu, dass auch in Schritt fünf und sieben das Update von Client B abgelehnt wird.

In Schritt neun ist das Schloss abgelaufen, es wird aber kein Update gesendet, sodass das Objekt unverschlossen bleibt. Dies bietet Client B in Schritt zehn die Möglichkeit ein Update durchzuführen.

Diese Art sich widersprechende Updates zu unterdrücken schützt effektiv in Situationen in denen die Latenz zum Server das Implementieren einer eigenen Schutzmaßnahme verhindert. Sollten Entwickler ein Objekt für längere Zeit sperren wollen, können sie ein Update senden, welches die Funktion des Objekts für andere Clients einschränkt und dies auch optisch darstellen.

Gibt es auf der Website zum Beispiel ein Textfeld, welches nur von einem Client zurzeit bearbeitet werden darf, kann ein Update gesendet werden, welches das Input-Feld deaktiviert sobald ein Nutzer auf das Feld klickt. Das Textfeld wird damit ausgegraut und Texteingaben sind nicht mehr möglich. Durch die oben beschriebene Absicherung der Objekte ist zudem sichergestellt, dass im Fall wenn zwei Nutzer gleichzeitig in das Feld klicken nur einer den Fokus bekommt.

Für diese Art der langfristigen Reservierung von Objekten gibt es eine eigene Methode, die alle Aufrufe der Update-Methode auch auf der Clientseite unterbindet. Um Entwicklern eine Anpassung des Verhaltens in diesem Fall zu erlauben, wurden die „Locked" und „Unlocked" Events implementiert, die es zum Beispiel erlauben CSS-Styles zu setzen. Zudem wird automatisch die „sync-locked" CSS-Klasse zu reservierten Objekten hinzugefügt, sodass Entwickler die Gestaltung von Elementen planen können, auch ohne sie mit JavaScript programmieren zu müssen. Auch bei dieser Art des Reservierens von Objekten wird wieder die ID des Websockets als Schlüssel verwendet um Updates vom aktuellen Besitzer des Objekts weiterhin zu erlauben.

## 7.3 Join

Der „Join“-Befehl ist immer der Erste, der von einem Client gesendet wird. Er gibt an welchem Raum der Client beitreten möchte und veranlasst den Server dazu alle bisher hinzugefügten Objekte in ihrem aktuellen Zustand an den Client zu senden.

Der „Join“-Befehl erlaubt außerdem das nachträgliche Ändern des Raums. Dies kann nützlich sein, wenn einige Standard-Objekte geladen werden sollen. Der Client verbindet sich dann mit einem Raum und erhält alle Standard-Objekte. Dann wechselt er in den eigentlichen Raum und erhält auch alle Elemente dieses Raumes.

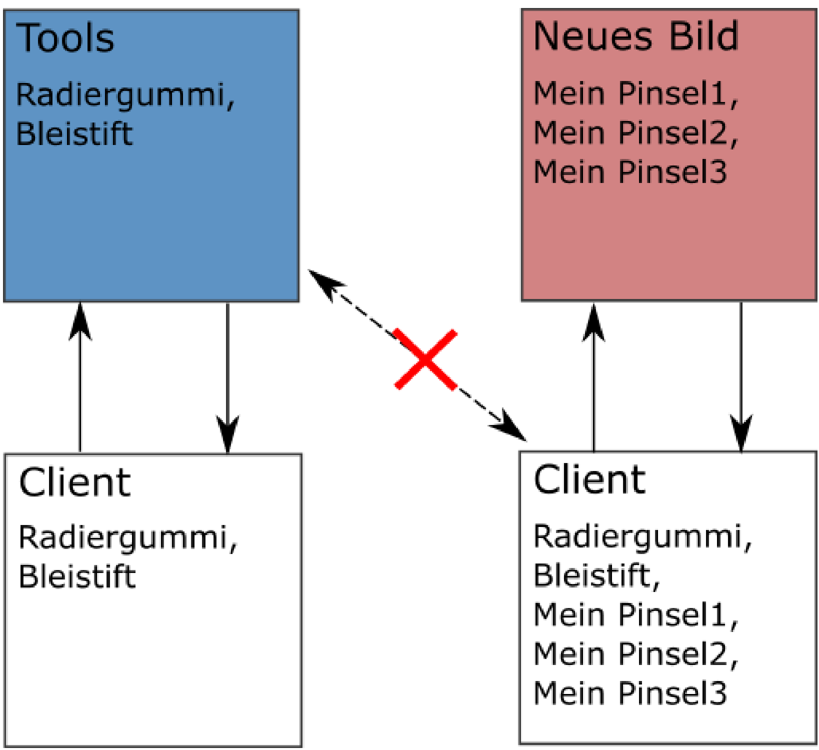


*Abbildung 5: Ablauf bei der Verbindung mit mehreren Räumen*

In Abbildung 5 verbindet sich der Client mit einer Malanwendung. Die Werkzeuge Radiergummi und Bleistift sollen dabei in jedem Raum verfügbar sein. Zuerst verbindet sich der Client mit dem Raum „Tools“ und bekommt die beiden Werkzeuge gesendet. Nachdem die Werkzeuge übertragen sind wechselt der Client in den Raum „Neues Bild“ in dem er bereits drei eigene Pinsel-Werkzeuge angelegt hat. Wichtig ist zu beachten, dass die Verbindung zum Raum „Tools“ komplett unterbrochen ist. Nimmt der Client Änderungen am Radiergummi oder dem Bleistift vor, gelten diese nur im Raum „Neues Bild“, obwohl die Werkzeuge ursprünglich aus einem anderen Raum stammen.

## 7.4 Datenbankanbindung

Räume lassen sich in ein JSON-Objekt umwandeln. Dieses Raum-Objekt enthält alle hinzugefügten Objekte in ihrem aktuellem Zustand sowie einige Meta-Informationen. Ist bei der Initialisierung von HTML-Sync eine Datenbankverbindung angegeben worden, wird der Raum automatisch in der Datenbank abgelegt. Der Aufbau von MongoDB ermöglicht dies ohne weitere Zwischenschritte.

Beim Laden von Räumen ist zu beachten, dass die Abfrage von Daten Asynchron abläuft. Die „getRoom"-Methode von HTML-Sync gibt daher kein Room- sondern ein Promise-Objekt zurück.

```
HTMLSync.getRoom("/").then(function success(room){
    console.log("Raum gefunden", room);
},
function failed(){
    console.log("Unter dem angegebenen Namen wurde kein Raum
gefunden");
```

Wenn der Raum vorhanden ist:

```
Raum gefunden <Room#XXXXXX>
```

Wenn kein Raum vorhanden ist:

```
Unter dem angegebenen Namen wurde kein Raum gefunden
```

Die Methode „getRoom" macht keinen Unterschied zwischen dem Laden aus dem Arbeitsspeicher und dem Laden aus der Datenbank. Entwickler können die Methode bedenkenlos nutzen und bekommen den Raum falls vorhanden sofort aus dem Arbeitsspeicher oder nachdem er geladen wurde aus der Datenbank.

# 8 Dokumentation

Einer der wichtigsten Bestandteile einer Softwarebibliothek ist die Dokumentation. Sie soll es anderen Entwicklern ermöglichen auf einen Blick festzustellen, ob die Library für ihre Anwendung eine Bereicherung bietet. In diesem Teil sind vor allem kurze, relevante Beispiele wichtig, die die Stärken der Bibliothek aufzeigen und deren Grenzen verdeutlichen.

Noch wichtiger als die Übersicht ist die eigentliche Dokumentation des Codes. Sie sollte alle Klassen der Library umfassen und auf alle öffentlichen Methoden eingehen und diese erläutern. Gerade bei komplexeren Vorgängen können auch hier kurze Beispiele viel bewirken. Beispiele sollten immer nur ergänzend zu einem Text verwendet werden.

Es gibt noch eine dritte Art der Dokumentation: Das Schritt-für-Schritt-Tutorial. In dieser Dokumentation wird der Nutzer an eine klar definierte Aufgabe herangeführt. Nach der groben Übersicht ist ein Tutorial meist der erste wirkliche Berührungspunkt von Entwicklern mit einer Software-Bibliothek. Es ist daher wichtig mit seinem Tutorial einen guten ersten Eindruck zu schaffen. Das heißt allerdings nicht, dass man Dinge verschweigen und die Dokumentation schönen sollte. Ein Entwickler der nach 15 Minuten merkt, dass die Bibliothek nicht für ihn geeignet ist, wird damit kein Problem haben. Ein Entwickler, der ein 30 Minütiges Tutorial durchgearbeitet hat und die Lösung all seiner Probleme durch die Bibliothek erwartet, wird sehr enttäuscht und vermutlich auch verärgert sein, wenn der Umgang mit der Bibliothek ganz anders funktioniert als im Tutorial dargestellt und somit für sein Projekt nicht zu verwenden ist.

# 9 Beispielanwendung

In diesem Kapitel soll die Entwicklung der Beispielanwendung beschreiben werden. Dabei werden die einzelnen Entwicklungsschritte ähnlich einem Tutorial erklärt um einen guten Eindruck der HTML-Sync Bibliothek im Einsatz zu erhalten.

## 9.1 Beschreibung der Anwendung

Es soll ein TicTacToe-Spiel entwickelt werden, welches über das Internet mit anderen gespielt werden kann. Die Anwendung soll dabei nicht die Logik des Spiels implementieren sondern den Spielern lediglich ein Spielfeld und die Möglichkeit Markierungen zu setzen bieten.

Es wird ein Spielfeld aus drei mal drei Feldern geben. Klickt ein Spieler auf eines der Felder wird die Markierung des Spielers in dieses Feld gesetzt. Ein Button kann bei einem Klick das Spielfeld zurücksetzen.

## 9.2 Installation

Für die Anwendung wird ein NodeJS Express Server benötigt. Die erforderlichen Dateien können mit dem NodeJS-Modul „Express application generator“[3] automatisch erstellt werden. Ist der NPM-Packetmanager bereits installiert, kann der Generator mit folgendem Befehl installiert werden.

```
$ npm install express-generator -g
```

[3] http://expressjs.com/en/starter/generator.html

Nach dem Abschluss der Installation kann mit dem Befehl „express <App-Name>" eine neue Express-Anwendung initialisiert werden.

```
$ express TicTacToe

   create : TicTacToe
   create : TicTacToe/package.json
   create : TicTacToe/app.js
   create : TicTacToe/public/javascripts
   create : TicTacToe/public
   create : TicTacToe/routes
   create : TicTacToe/routes/index.js
   create : TicTacToe/routes/users.js
   create : TicTacToe/public/stylesheets
   create : TicTacToe/public/stylesheets/style.css
   create : TicTacToe/public/images
   create : TicTacToe/views
   create : TicTacToe/views/index.jade
   create : TicTacToe/views/layout.jade
   create : TicTacToe/views/error.jade
   create : TicTacToe/bin
   create : TicTacToe/bin/www

   install dependencies:
     $ cd TicTacToe && npm install

   run the app:
     $ DEBUG=TicTacToe:* npm start
```

Durch diesen Befehl wurde ein neues Verzeichnis erstellt in dem sich neben einigen Unterverzeichnissen und Dateien auch die „package.json"-Datei befindet. Diese Datei enthält Informationen über alle von der Anwendung benötigten NodeJS-Module. Es ist erforderlich diese Module zu installieren.

```
$ cd TicTacToe
$ npm install
```

Nach der Installation der NodeJS-Module sollte das Verzeichnis wie in Abbildung 6 dargestellt aussehen.

| Name | Änderungsdatum | Typ | Größe |
|---|---|---|---|
| bin | 15.06.2016 10:53 | Dateiordner | |
| node_modules | 15.06.2016 11:08 | Dateiordner | |
| public | 15.06.2016 10:53 | Dateiordner | |
| routes | 15.06.2016 10:53 | Dateiordner | |
| views | 15.06.2016 10:53 | Dateiordner | |
| app.js | 15.06.2016 10:53 | JS-Datei | 2 KB |
| package.json | 15.06.2016 10:53 | JSON-Datei | 1 KB |

*Abbildung 6: Inhalt des TicTacToe-Verzeichnisses nach der Installation der Module.*

Sollte Git oder andere Versionsverwaltungssysteme verwendet werden, ist es gängige Praxis den Ordner „node_modules" nicht in die Versionsverwaltung aufzunehmen. In diesem Ordner befinden sich im Regelfall nur Bibliotheken anderer Entwickler die nicht lokal geändert werden sollten. Es ist jedem Entwickler außerdem möglich die Module selbst zu installieren.

Nachdem nun die Basisinstallation abgeschlossen ist kann das HTML-Sync Modul installiert werden.

```
$ npm install html-sync --save
html-sync@0.0.51 node_modules\html-sync
└── socket.io@1.4.6 (has-binary@0.1.7, socket.io-parser@2.2.6,
socket.io-adapter@0.4.0, engine.io@1.6.9, socket.io-client@1.4.6)
```

Das Modul „Socket.IO" ist in HTML-Sync als Abhängigkeit angegeben und wird dadurch automatisch mit installiert. Der Zusatz „--save" sorgt zudem dafür, dass das HTML-Sync Modul mit in die „package.json"-Datei aufgenommen und somit in Zukunft durch den Befehl „npm install" mit installiert wird.

## 9.3 Server

Die Datei „app.js“ stellt den Einstiegspunkt für den Server da und wird beim Start der Anwendung als erstes ausgeführt. Gleich zu Beginn der Datei werden einige Standard Bibliotheken und Routen geladen.

**app.js**

```
1   var express = require('express');
2   var path = require('path');
3   var logger = require('morgan');
4   var bodyParser = require('body-parser');
5   var HTMLSync = require('html-sync');
6
7   var app = express();
8   var server = app.listen(process.env.PORT || 3000);
9
10  var io = require('socket.io').listen(server);
11  var hs = new HTMLSync(io, {debug:false});
12
13  io.on('connection', function(socket){
14    console.log('a user connected');
15
16    HTMLSync.setSocket(socket);
17  });
18
19  var routes = require('./routes/index');
```

In Zeile Zehn und Elf werden die Bibliotheken Socket.IO und HTML-Sync initialisiert. HTML-Sync nimmt dabei im Konstruktor die Instanz von Socket.IO entgegen. Über Konfigurationsobjekt können zusätzlich einige Parameter der Library eingestellt werden.

Der Codeblock von Zeile 13 bis 17 reagiert auf das „connection“-Event der Socket.IO Bibliothek. Dabei wird zum einen eine Nachricht auf der Konsole ausgegeben und zum anderen dem Websocket HTML-Sync spezifische Eventhandler hinzugefügt.

Der darauf folgende Code regelt vor allem grundlegende Einstellungen des Express-Servers und kann wie vom Generator erstellt beibehalten werden.

Im nächsten Schritt soll die Datei „routes/index.js“ näher betrachtet werden.

**routes/index.js**

```
1   var express = require('express');
2   var router = express.Router();
3
4   /* GET home page. */
5   router.get('/', function(req, res, next) {
6       res.render('index', { title: 'Express' });
7   });
8
9   module.exports = router;
```

Es ist gängige Praxis die Routen, also die verschiedenen Pfade die der Server den Clients zur Verfügung stellt, in eigenen Modulen zu programmieren. Durch die Trennung des Moduls ist ein erneuter Import der HTML-Sync Library erforderlich. NodeJS wird dabei erkennen, dass die Bibliothek bereits geladen ist und die vorhandenen Daten bereitstellen.

**routes/index.js**

```
1   var express = require('express');
2   var router = express.Router();
3   var HTMLSync = require('html-sync');
4
5   /* GET home page. */
6   router.get('/', function(req, res, next) {
7       HTMLSync.getRoom("game", true).then(function(room) {
8           if (room.hasParts() == false) {
9               createBoard();
10          }
11          res.render('index', { title: 'TicTacToe' });
12      });
13  });
```

In den Zeilen Sechs bis 13 wird das Verhalten des Servers definiert, sobald die Startseite aufgerufen wird. Für die TicTacToe-Anwendung ist ein Spielfeld erforderlich. Dieses soll auf dem Server erstellt werden, da es für alle Clients nur ein einziges Mal erstellt werden soll. Um dies sicherzustellen wird zuerst der Raum „game" geladen. Der zweite Parameter der „getRoom"-Funktion gibt dabei an, dass ein neuer Raum erstellt werden soll sollte „game" noch nicht vorhanden sein.

Die „getRoom"-Funktion ist eine asynchrone Funktion um das Laden von Räumen auch aus Datenbanken zu erlauben. Die in die „then"-Methode übergebene Funktion beschreibt wie der Server vorgehen soll, sobald ein Raum gefunden wurde. In diesem Beispiel wird immer ein Raum zurückgegeben werden. Entweder aus dem Arbeitsspeicher des Servers, aus einer Datenbank oder es wird ein neuer Raum erstellt. Es ist auch möglich eine zweite Funktion zu übergeben, welche den Fall eines Fehlers beim Laden des Raums behandelt.

Es wird überprüft, ob der geladene Raum bereits Parts enthält. Beim ersten Aufruf des Pfades wird dies nicht der Fall sein. In diesem Fall wird die Methode „createBoard" aufgerufen, welche das Spielfeld erstellt und zur Synchronisierung hinzufügt. Das Spielfeld muss nur ein einziges Mal erstellt werden, da sich alle Clients ein Spielfeld teilen. Dieses Beispiel geht davon aus, dass nie mehr als zwei Clients den Server verwenden.

**routes/index.js**

```
16  function createBoard(){
17      console.log("Creating Board");
18      for(var x = 0; x < 9; x++){
19          var field = new HTMLSync.Part("div");
20          field.attr("className", "field");
21          field.on("click", function(){
22              this.update({
23                  style:{
24                      backgroundImage: "url(" + playerSymbol + ")",
25                  }
26              });
27          });
28          field.parent = "board-wrapper";
29          field.room = "game";
30
31          HTMLSync.add(field);
32      }
```

In diesem Codeabschnitt werden die neun Felder des Spielfelds erstellt. Das Spielfeld soll aus Div-Elementen bestehen deshalb wird dem Part-Konstruktor hier der String „Div“ übergeben. Um den erstellten Elementen einen CSS-Style geben zu können wird ihnen ein CSS-Klassenname hinzugefügt. Hier wird die Bezeichnung „className“ statt „class“ verwendet, da die Attribute beim Client mit JavaScript gesetzt werden müssen und das „class“-Attribut in JavaScript über das „className“-Attribut gesetzt werden muss.

In Zeile 21 wird ein Click-Eventhandler hinzugefügt. Sobald das Feld geklickt wird, soll das Hintergrundbild des Felds zum Symbol des klickenden Spielers geändert werden. Das „this“-Keyword bezieht sich innerhalb dieses Handlers auf das Part-Objekt des Clients und nicht auf das DOM-Element welches geklickt wurde. Es wird die „update“-Methode aufgerufen um den Style des Felds zu verändern. Auch in diesem Schritt muss wieder „backgroundImage“ anstatt des CSS-Befehls „background-image“ verwendet werden um den Style mit JavaScript hinzufügen zu können. Die Variable „playerSymbol“ ist eine globale Variable die auf dem Client gesetzt wird.

Als „parent“ also Elternelement aller Felder wird ein DOM-Element mit der ID „board-wrapper“ angegeben. Dieses muss nicht synchronisiert werden und kann daher normal mit HTML erstellt werden.

In Zeile 29 wird den erstellten Feldern der Raum „game“ zugewiesen, damit sie an die Clients ausgeliefert werden, welche sich mit diesem Raum verbinden. In Zeile 31 werden die Felder dann zur Synchronisierung hinzugefügt.

**routes/index.js**

```
34      function PlayerSymbol(path){
35          var part = new HTMLSync.Part("img");
36          part.attr({
37              className: "symbol",
38              src: path,
39          });
40          part.on("click", function(e){
41              if(!this.locked){
42                  playerSymbol = e.target.src;
43                  this.lock();
44                  $("#symbol-wrapper").hide();
45              }
46          });
47          part.parent = "symbol-wrapper";
48          part.room = "game";
49          return part;
50      }
51
52      var player1 = PlayerSymbol("/images/Player1.png");
53      HTMLSync.add(player1);
54
55      var player2 = PlayerSymbol("/images/Player2.png");
56      HTMLSync.add(player2);
57  }
```

In diesem Abschnitt sollen zwei Symbole erstellt werden, die es den Spielern erlauben ihr bevorzugtes Symbol auszuwählen. Da die Symbole bis auf den Pfad zu ihrem Bild genau gleich sind wurde hier eine Funktion erstellt um sie zu generieren. In TypeScript wäre auch eine einfache Vererbung der Part-Klasse möglich gewesen.

Die zu erstellenden Symbole sind Bilder werden also mit dem „img“-Tag erstellt. Ihnen werden eine CSS-Klasse und der Pfad zu dem jeweiligen Bild zugewiesen.

Bei einem Klick auf eines der Symbole wird zuerst überprüft ob dieses bereits gewählt wurde also „gelocked“ ist. Ist dies nicht der Fall wird die globale Client-Variable „playerSymbol“ gesetzt und das Symbol „gelocked“. Zusätzlich werden alle Symbole versteckt um eine Mehrfachauswahl zu unterbinden.

Die Symbole werden dem DOM-Element mit der ID „symbol-wrapper“ hinzugefügt um sie einfach mit CSS positionieren zu können.

## 9.4 Client

Für dieses Beispiel wurde die Template-Engine Jade verwendet. Die Dateien „layout.jade" und „index.jade" wurde vom Express-Generator im Verzeichnis „views" automatisch erstellt. Das Layout ist dabei eine Art Rahmen, der auf mehreren Seiten verwendet werden kann und eignet sich daher sehr gut um Menüs zu erstellen und Bibliotheken zu laden, welche auf mehreren Seiten verwendet werden.

**views/layout.jade**

```
1  doctype html
2  html
3    head
4      title= title
5      link(rel='stylesheet', href='/stylesheets/style.css')
6      script(src='/javascripts/jquery.min.js')
7      script(src='/javascripts/html-sync.min.js')
8    body
9      block content
```

Hier werden die Bibliotheken jQuery sowie HTML-Sync geladen. Außerdem wird eine CSS-Datei eingebunden. Der Code in Zeile neun gibt an, dass an dieser Stelle der eigentliche Content der Seite eingefügt wird.

**views/index.jade**

```
1  extends layout
2  
3  block content
4    h1= title
5    p Welcome to #{title}
6    div#symbol-wrapper
7    div#board-wrapper
8    button#reset
9      | Reset
10 
11   script(src="/javascripts/game.js")
```

Dies ist der Code zum Erstellen des Contents der Seite. Es wird der Titel als Überschrift ausgegeben und der Spieler begrüßt. Der Titel wird dabei vom Server als Variable übergeben und ist in diesem Beispiel „TicTacToe".

Es folgen der Symbol- und Board-Wrapper die beide keinen Inhalt benötigen, da dieser durch synchronisierte Elemente befüllt wird.

In Zeile elf wird das Script „game.js" eingebunden.

**public/javascripts/game.js**

```
1   var htmlSync = new HTMLSync({
2       debug:true,
3       room:"game"
4   });
5
6   var playerSymbol = false;
7
8
9   $("#reset").click(function(){
10
11      $(".field").each(function(){
12          var partId = $(this).attr("id");
13
14          HTMLSync.parts[partId].update({
15              style:{
16                  backgroundImage: ""
17              }
18          }, true);
19      });
20  });
```

In Zeile eins bis vier wird die Verbindung mit dem Server aufgebaut. Wichtig ist hierbei zu beachten, dass der Parameter „room“ auf den Raum gesetzt ist, welcher auf dem Server erstellt wurde.

Die Variable „playerSymbol“ wird initialisiert damit sie von den „Field“-Elementen verwendet werden kann, die vom Server übertragen werden.

In den Zeilen neun bis 20 wird ein Eventhandler für den Reset-Button definiert. Alle DOM-Elemente mit der CSS-Klasse „field“ werden ausgewählt und ihre ID wird ausgelesen. Die ID der DOM-Elemente ist gleichzeitig die ID der Part-Objekte, die vom Server erstellt wurden. Alle Part-Objekte, die auf dem Client verfügbar sind werden in der Hashmap „HTMLSync.parts“ gespeichert. Mithilfe der IDs kann also die „update“-Methode aller Felder aufgerufen, und ihr Hintergrundbild zurückgesetzt werden.

Damit das Spielfeld korrekt angezeigt wird sind noch einige CSS-Anweisungen erforderlich.

**public/stylesheets/style.css**

```
10  .field{
11      width:100px;
12      height:100px;
13      border:solid;
14      float:left;
15      background-size: 100% 100%;
16  }
17
18  #board-wrapper{
19      width:325px;
20  }
```

Die wichtigen Styles für die Darstellung des Spielfelds sind hier aufgeführt. Die Felder bekommen eine feste Höhe und Breite sowie einen Rahmen. Durch die Anweisung „float:left“ verhalten sie sich zudem wie Text. Sie ordnen sich so lange an der rechten Seite des vorangehenden Elements an bis das Umgebene Element nicht mehr ausreichend Platz bietet. In diesem Fall beginnen sie eine neue Zeile. Durch die Begrenzung der Breite des „board-wrapper“-Elements passen nicht mehr als drei Elemente inklusive Rahmen in eine Zeile es ergibt sich also automatisch ein Spielfeld mit drei mal drei Feldern.

Eine Eigenschaft des „lock"-Befehls, welche bei den Symbol-Objekten verwendet wurde ist es, dass er gesperrten Elementen eine „sync-locked" CSS-Klasse hinzufügt.

**public/stylesheets/style.css**

```
27  .symbol.sync-locked{
28      opacity:0.5;
29  }
```

Diese Anweisung sorgt dafür, dass das bereits gewählte Symbol für den anderen Spieler halbtransparent erscheint.

## 9.5 Testen

Der Server kann mit dem folgenden Befehl gestartet werden.

```
node app.js
```

Nach einer kurzen Wartezeit kann mit einem Webbrowser zur URL „127.0.0.1:3000" navigiert werden. Es sollte die folgende Seite angezeigt werden.

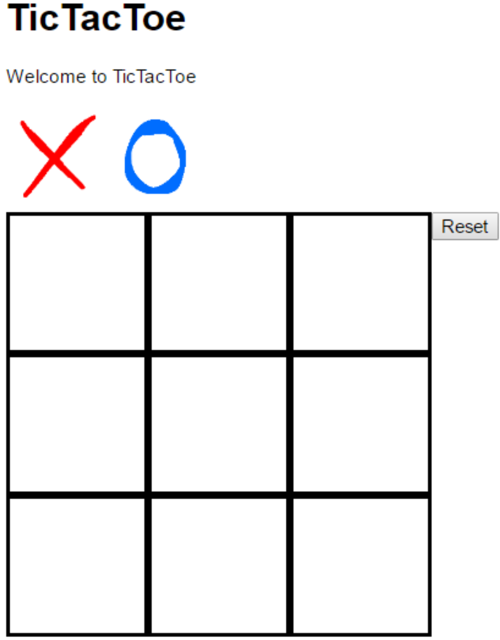


*Abbildung 7: Die fertige TicTacToe-Website*

Um die Funktion der Applikation vollständig zu testen kann die Seite in einem zweiten Tab erneut geöffnet werden. Spieler sollten ein Symbol wählen und dieses dann auf den Feldern platzieren können. Der Reset-Button wird die Felder wieder leeren. Alle Änderungen an der Seite mit Ausnahme des Ausblendens der beiden Symbole nachdem eine Wahl getroffen wurde sollten synchron in beiden Tabs erfolgen.

# 10 Testanwendungen

Um die Funktionsweise der Library zu testen und zukünftigen Entwicklern einige Codebeispiele zu liefern wurden gleich mehrere kleine Testanwendungen erstellt, die sich auf verschiedene Bereiche der Library spezialisieren.

## 10.1 Hello World

Hello World ist eine möglichst einfache Anwendung, die die grundlegenden Funktionen der Library implementiert. Es wird eine Überschrift für jeden Client erstellt, der sich mit dem Server verbindet und mit allen anderen Clients synchronisiert. Diese Überschrift lässt sich durch einen Klick rot einfärben.

Die Anwendung zeigt, wie einfache Elemente erstellt und synchronisiert werden können, wie Attribute und Styles von Elementen verändert werden können und wie Event-Handler eingesetzt werden können um zum Beispiel auf Klicks zu reagieren.

### 10.1.1 Aufbau der Anwendung

Der Server übernimmt in dieser Anwendung überhaupt keine gesonderten Aufgaben. Er stellt lediglich die Socketverbindung und die Website zur Verfügung.

Der Client verbindet sich mit dem Server sobald die Seite geladen ist und erstellt ein HelloWorld-Objekt, das synchronisiert wird.

```
var io = require('socket.io')(http);
var HTMLSync = require('html-sync');
var hs = new HTMLSync(io, {debug:true});

io.on('connection', function(socket){
  console.log('a user connected');

  HTMLSync.setSocket(socket);
});
```

*Auszug aus dem Code des Webservers.*

```
var htmlSync = new HTMLSync({debug:true});
var hello = new Part("h1");
hello.setAttributes({
    innerHTML:"Hello World"
});
hello.on("click", function(e){
    console.log(this);
    this.update({
        style:{
            color:"red"
        }
    });
});
htmlSync.add(hello);
```

*Code der Website ohne den Import der JS-Dateien*

## 10.2 Thousand Stars

Ähnlich wie das Hello World Beispiel ist auch Thousand Stars sehr einfach gehalten. Die Website besteht aus einem dunklen Nachthimmel auf den die User mit ihrer Maus Sterne platzieren können.

### 10.2.1 Aufbau der Anwendung

Der Servercode bedarf keiner Änderung zum Hello World Beispiel. Auch hier stellt der Server nur die Verbindung bereit.

Die Website besteht in diesem Beispiel aus zwei Div-Elementen, die sich über den kompletten Bildschirm erstrecken und mit einer Grafik und CSS einen Nachthimmel repräsentieren. Mit Hilfe von jQuery wird dem Himmel ein Click-Eventhandler zugewiesen, der einen neuen Stern erzeugt und synchronisiert.

```
<body>
    <div class="sky"></div>
    <div class="trees"></div>
</body>

<script>
    var htmlSync = new HTMLSync({debug:false});

    $(".sky").click(function(e){
        var star = new Part("div");
        star.attr("className", "star");
        star.setStyles({
            top: e.pageY + "px",
            left: e.pageX + "px"
        });
        htmlSync.add(star);
    });
</script>
```

*Code der Website ohne den Import der CSS und JS-Dateien*

*Abbildung 8: Thousand Stars Website*

## 10.3 Dame

In der Applikation „Dame“ wurde das gleichnamige Brettspiel nachgebaut. Diese Anwendung diente vor Allem zum Testen der Performance der HTML-Sync Library. Die Spielsteine können per Drag and Drop bewegt werden und jede Bewegung wird sofort synchronisiert. In den zum Testen verwendeten Browsern waren bei einer Bewegung zwischen 80 und 130 Drag-Events pro Sekunde zu verzeichnen, die der Server problemlos an andere Clients übertragen konnte.

Ein weiteres Feature der Library, welches hier getestet werden konnte war das Sperren von Elementen. Ohne dieses Feature war es möglich, dass zwei Clients dasselbe Objekt in unterschiedliche Richtungen bewegen konnten was zu erheblichen Flackern und anderen Störungen führte. Durch die in 7.2 Updates beschriebenen Systeme konnte dieses Fehlverhalten unterbunden werden.

### 10.3.1 Aufbau der Anwendung

Der Server erstellt in dieser Anwendung die Spielsteine für beide Spieler sobald ein erster Client die Website der Anwendung besucht. Das Spielbrett ist statisch und wird daher als normales HTML vom Server ausgeliefert. Für die Drag and Drop Events wurde die JavaScript Library Interact.js verwendet.

```
function dragMoveListener (event) {
    var target = event.target,
    // keep the dragged position in the data-x/data-y
attributes
        x = (parseFloat(target.getAttribute('data-x')) || 0) +
event.dx,
        y = (parseFloat(target.getAttribute('data-y')) || 0) +
event.dy;

    // update the posiion attributes
    target.setAttribute('data-x', x);
    target.setAttribute('data-y', y);

    HTMLSync.parts[target.id].update({
        style:{
            transform: "translate(" + x + "px, " + y + "px)",
        },
        data:{
            x: x,
            y: y,
        }
    }, true);
}
```

*Auszug aus dem Client-Code*

Die Anwendung verändert dabei nicht das CSS-Attribut „position“ bzw. „top“ und „left“, sondern nutzt die mit CSS3 eingeführte „transform“-Anweisung. Diese hat den Vorteil den Fluss des HTML-Dokuments nicht zu beeinflussen. Ein Element nimmt also einen gewissen Platz im Dokument in Anspruch, wird aber an anderer Stelle angezeigt. Dies führt dazu, dass nicht bei jeder Änderung der Position das gesamte Dokument neu berechnet werden muss, da sich nur die Position des bewegten Objekts ändert und andere Elemente nicht länger von diesen Änderungen betroffen sind.

Ein Problem, das sich bei der Entwicklung der Anwendung ergab, war die Berücksichtigung unterschiedlicher Bildschirmauflösungen. Ursprünglich wurde das Spielbrett zentriert auf dem Bildschirm angezeigt, war also relativ zur Displayauflösung. Durch die absolute Positionierung der Spielsteine führte dies dazu, dass Spielsteine beim zweiten Client nicht mehr genau auf den Feldern platziert wurden, sobald dieser eine andere Auflösung verwendete. Die simple Lösung war es auch das Spielbrett absolut zu positionieren. Für die zukünftige Entwicklung der HTML-Sync Library wäre es aber denkbar ein System zu entwickeln, welches Objekte relativ zur Bildschirmauflösung positioniert.

Zurücksetzen

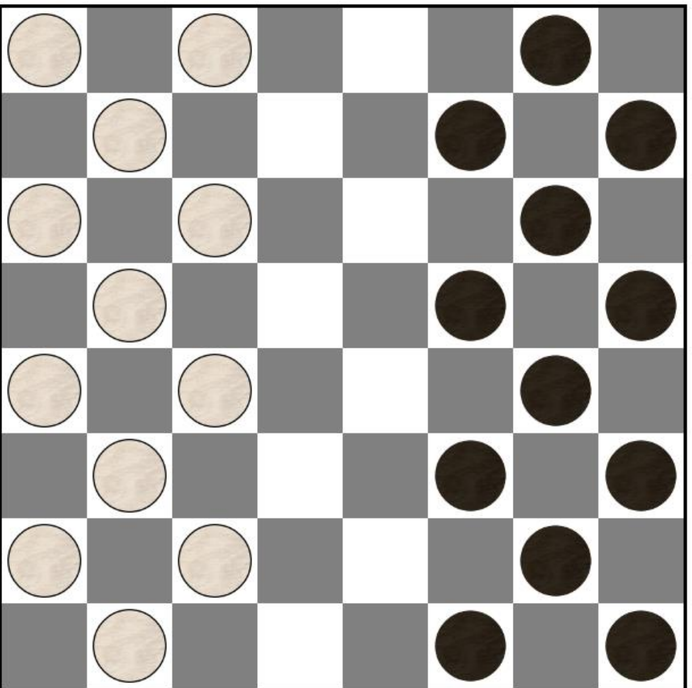

*Abbildung 9: Das fertige Dame-Spiel*

Obwohl es ursprünglich nur als Testanwendung gedacht war, wurde das Spiel an einigen Stellen optimiert um angenehm spielbar zu sein. Die Felder bieten eine Snap-Funktion, welche die Spielsteine einrasten lässt, sodass das Spielfeld immer ordentlich und aufgeräumt wirkt und keine zu präzise Eingabe mit der Maus erforderlich ist. Zudem wurden die schwarzen Felder des Spielfelds als Grau dargestellt um einen besseren Kontrast zu den schwarzen Spielsteinen zu bieten. Die weißen Spielsteine haben zu dem einen schwarzen Rand und umgekehrt um die Abgrenzung zu den Feldern zu erhöhen. Die Spielsteine haben außerdem eine leichte Holztextur, was sie weiter vom Hintergrund abhebt und einen angenehmeren Eindruck hinterlässt als schlichte schwarze und weiße Steine.

## 10.4 Tri

*Abbildung 10: Die Tri Website mit dem Seed „2345636452“*

Die Anwendung „Tri“ wurde als Leistungstest entwickelt. Es wird ein zufälliger Farbverlauf erzeugt und in Dreiecke unterteilt. Durch einen Klick auf eines dieser Dreiecke wird ein Ton abgespielt. Neben dem reinen Performancetest sollte auch das Aufrufen von Methoden an entfernten Clients und die Zusammenarbeit mit externen Libraries getestet werden. Zum Erzeugen der Töne wurde die Library „timbre.js“[4] verwendet.

### 10.4.1 Aufbau der Anwendung

Der Server stellt bei dieser Anwendung nicht nur einen Pfad zur Verfügung, sondern erlaubt das Übergeben eines Parameters in der URL. Dieser Parameter wird als Seed, also als Ausgangswert, für die Pseudozufallsfunktionen verwendet. Um den Verlauf zu erstellen wurde das NodeJS Modul „proc-noise“[5] verwendet. Dieses Modul implementiert den sogenannten „Perlin-Noise“, der 1982 entwickelt wurde um Lichteffekte im Film „Tron“ zu erstellen. Über den Seed ist es möglich denselben Farbverlauf wieder und wieder zu generieren.

Wird die Website mit einem bestimmten Seed angefordert, überprüft der Server zuerst ob der entsprechende Raum existiert, falls nicht wird ein neuer Raum erstellt und der Farbverlauf generiert. Der Perlin-Noise liefert einen pseudozufälligen Wert zwischen Null und Eins sorgt dabei aber dafür, dass benachbarte Zahlen sich nicht zu stark unterscheiden. Auf diese Weise werden Verläufe erstellt. Um von Werten zwischen null und eins zu einer Farbe zu kommen, werden gleich drei verschiedene Seeds verwendet um die RGB-Komponenten der Farben abzubilden.

[4] http://mohayonao.github.io/timbre.js/
[5] https://www.npmjs.com/package/proc-noise

```
function generateField(seed){
  console.log("Generating...");
  var rPerlin = new PerlinGenerator(seed);
  var gPerlin = new PerlinGenerator(seed + 1);
  var bPerlin = new PerlinGenerator(seed + 2);

  for (var y = 0; y < 15; y++) {
    for (var x = 0; x < 41; x++) {
      var p = new Part("div");
      p.room = seed;
      var r = parseInt(rPerlin.noise(x/25,y/25) * 255);
      var g = parseInt(gPerlin.noise(x/25,y/25) * 255);
      var b = parseInt(bPerlin.noise(x/25,y/25) * 255);
      var color = "rgb(" + r + "," + g + "," + b + ")";
      p.data.note = calcFrequency(rPerlin.noise(x/25,y/25));
      p.data.color = color;
      p.on("click", function(){
        this.update({
        calls:[
          {name:"note"}
        ]
        },true);
      });
      p.on("note", function(){
        var sine = T("sin", {
          freq:this.data.note,
          mul:0.5
        });

        T("perc", {r:500}, sine).on("ended", function() {
          this.pause();
        }).bang().play();

      });

      p.parent = "wrapper";
      HTMLSync.add(p);
    }
  }
}
```

*Auszug des Codes zur Generierung des Farbverlaufs*

Um die Tonhöhe eines Dreiecks zu bestimmen, wird nur der Rot-Wert der entsprechenden Farbe herangezogen. Je größer der Wert desto höher wird der Ton. Dabei sorgt die Funktion „calcFrequency“ dafür, dass alle Töne auf der normalen Tonleiter liegen. Dies führt zu einem angenehmen Klang selbst wenn mehrere Dreiecke gleichzeitig oder kurz nacheinander geklickt werden.

### 10.4.2 Auswertung des Performancetests

Um einen Bildschirm mit 1920x1080 Pixeln vollständig mit Dreiecken zu füllen werden 41x15 Dreiecke benötigt. Alle Dreiecke müssen nach der Erstellung an den Client über die Websocket-Verbindung an den Client übertragen werden. Die Übertragung der 615 Dreiecke funktioniert dabei in einer Zeit die kaum von dem normalen Laden einer Website zu unterscheiden ist. Kritischer ist der Bedarf an Arbeitsspeicher. Jeder weitere Raum von Dreiecken erfordert ungefähr zehn Megabyte Arbeitsspeicher. Auf einem schwachen Webserver wie zum Beispiel der Gratisvariante von Heroku[6] wären somit ungefähr 50 gleichzeitige Räume möglich bevor der Arbeitsspeicher des Servers ausgelastet ist. Hierbei ist zu bedenken, dass das programmierte Beispiel bewusst Grenzen auskundschaften sollte. Nach diesen Tests ist es selbst mit einem leistungsschwachen Webserver möglich ungefähr 30.000 Part-Objekte zu synchronisieren.

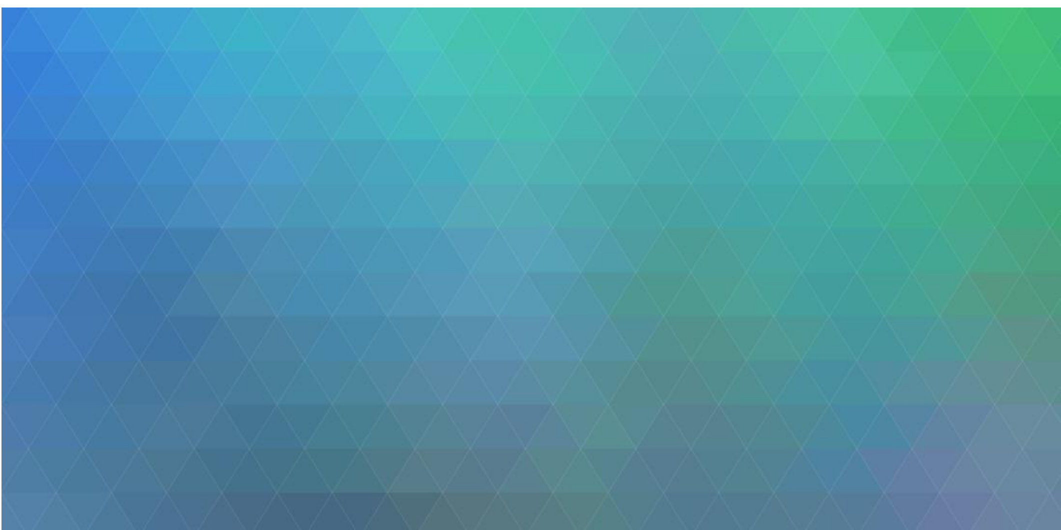

*Abbildung 11: Tri Beispiel mit dem Seed „bachelorarbeit"*

[6] www. heroku.com

## 10.5 Yavango DESK

Der Yavango DESK ist ein Onlineberatungstool, dass für das Startup Unternehmen Yavango entwickelt wurde. Mit dem DESK ist es möglich einen Reiseberater in einem Reisebüro „anzurufen“ und einen Webcamchat mit ihm zu starten. Zusätzlich zu den Videofenstern gibt es eine großzügige Arbeitsfläche auf der Bilder, PDFs und andere Dateien geteilt werden können. Der DESK ist dabei die wohl umfangreichste Anwendung in dieser Liste. Sie ist allerdings nicht mit der Bibliothek umgesetzt, sondern wurde schon vorher entwickelt. Die Anwendung ist somit der Ausgangspunkt für die Entwicklung der Library.

### 10.5.1 Aufbau der Anwendung

Der Server übernimmt in dieser Anwendung eine ganze Reihe von Aufgaben. Er erlaubt den Upload von Dateien auf einen AWS-Bucket (Amazon Web Service Angebot zum Speichern von Daten), regelt die Zugangsberechtigungen zu diesen Dateien, stellt eine API zur Verfügung über die Reiseberater ihre Profile auf der eigenen Website einbinden können und verwaltet die Beratungsräume und speichert sie in einer Datenbank ab.

Der Client dient in dieser Anwendung hauptsächlich zur Abfrage der bereitgestellten Daten. Im Beratungsraum ist es Clients möglich neue Formulare zu öffnen und diese zu synchron zu bewegen und auszufüllen.

Eine genaue Dokumentation des Codes würde den Rahmen dieser Arbeit überschreiten, es sei hier aber erwähnt, dass die Anwendung erheblichen Nutzen aus der von TypeScript ermöglichten Vererbung zieht.

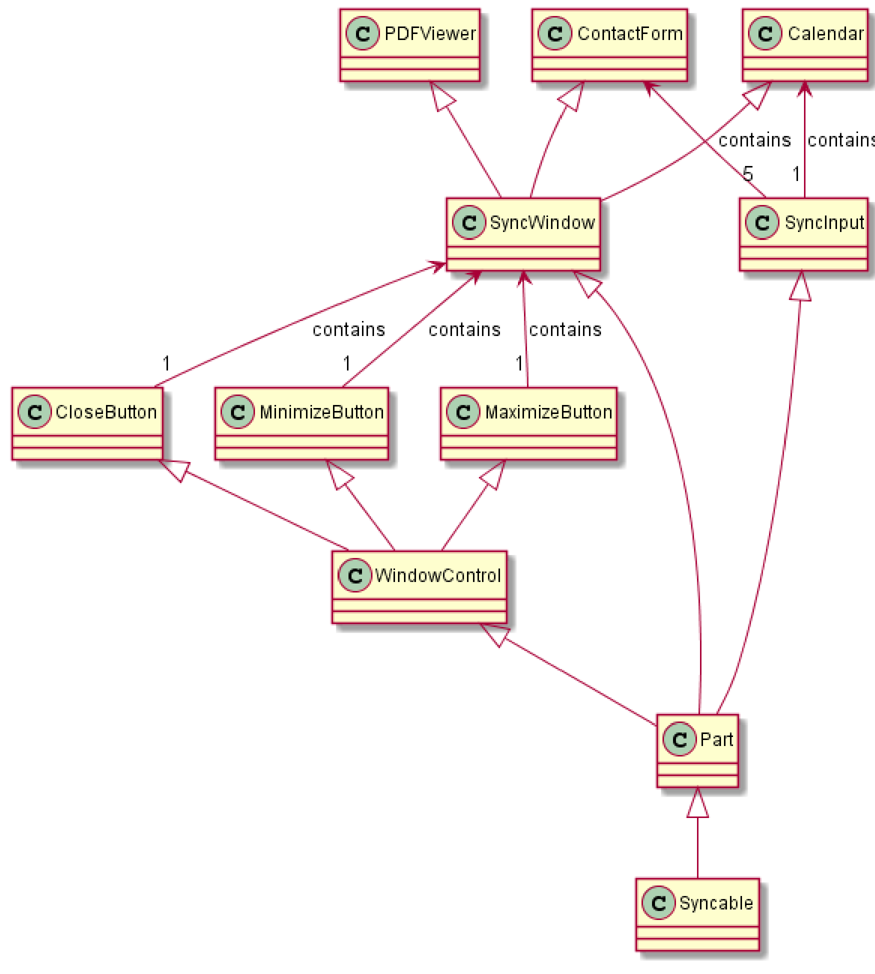


*Abbildung 12: Auschnitt der Klassenhirachie des Yavango DESK*

Das UML-Diagramm zeigt, dass die meisten Objekte im DESK von SyncWindow erben. Das SyncWindow stellt dabei ein leeres Fenster da, wie man es von modernen Desktop PCs gewohnt ist, dass nach Belieben mit Inhalt gefüllt werden kann. In den meisten Fällen ergänzen die Subklassen die Funktionsweise ihres Elternelements nicht, sondern rufen nur den Konstruktor mit bestimmten Parametern auf. Die Klassen in diesem Ausschnitt können also eher wie CSS-Klassen verstanden werden, die einem Objekt ein Aussehen verleihen ohne die Natur des Objekts zu veränden.

Es wird außerdem deutlich, dass Objekte der Part-Klasse andere Part-Objekte enthalten können. Genau wie auch HTML lassen sich Part-Objekte verschachteln, was die Nutzung von Vorlagen für bestimmte Bestandteile weiter begünstigt. Als Beispiel wäre hier die WindowControl-Klasse zu nennen. Sie erstellt einen kleinen Button, der sich rechtsbündig orientiert mit einem definierbaren Bild und Hintergrundfarbe. Die drei Klassen „CloseButton“, „MinimizeButton“ und „MaximizeButton“ setzen jeweils das Bild und die Farbe und implementieren die jeweiligen Klick-Events.

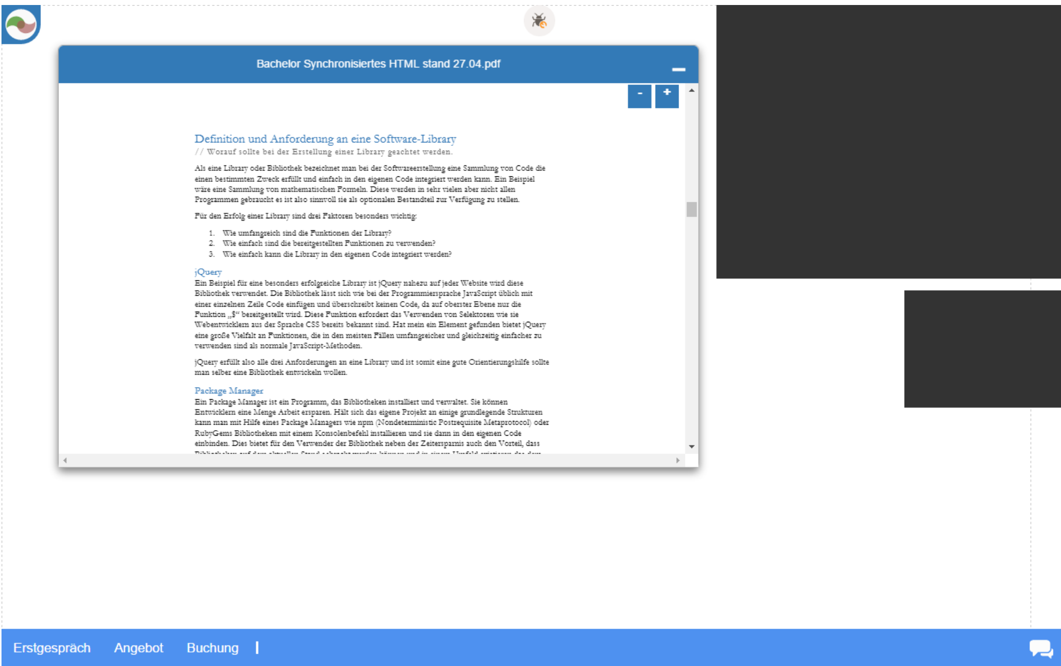


*Abbildung 13: Yavango DESK mit PDF*

Die Abbildung zeigt den Berater DESK. Die beiden schwarzen Rechtecke an der rechten Seite sind Platzhalter für die Videoübertragung per Webcam. Es ist ein PDF im DESK geladen welches von allen Clients synchron gelesen werden kann. Dies geschieht indem sowohl der Zoom über die Buttons oben rechts als auch die Scrollposition des Dokuments synchronisiert werden. Alle Clients sehen so immer den gleichen Abschnitt des Dokuments. Entwickelt wurde dieses System um es Reiseberatern zu ermöglichen Kunden ihre Kataloge im PDF-Format auch online zeigen zu können.

# 11 Fazit

## 11.1 Entwicklung

Für das Startup Unternehmen „Yavango" wurde bereits vor dem Start dieser Arbeit ein System zum Synchronisieren von HTML-Inhalten entwickelt. Dieses war stark im Code verwurzelt, deshalb konnten nur die bewährten Prinzipien aus diesem System in die HTML-Sync-Bibliothek übernommen werden.

Nachdem die Kernfunktionalität implementiert wurde, wurden zusätzlich weitere Features in die Bibliothek übernommen. Dabei handelte es sich vor allem um kleine Verbesserungen, welche den reibungslosen Ablauf der Anzeige bei sich widersprechenden Eingaben gewährleisten. Auch der Komfort während der Entwicklung wurde weiter verbessert, indem neue Methoden hinzugefügt wurden, um sich weiter an gängige Standards anzunähern.

Während die Auskopplung der Bibliothek aus dem wesentlich umfangreicheren Code von „Yavango" die Entwicklung dieser neuen Features begünstigte brachte sie jedoch auch Probleme mit sich.

Die Bibliothek ist für sich genommen nicht lauffähig und durch den starken Fokus auf asynchrone Abläufe zudem schwer mittels Unit Tests zu testen. Im Laufe der Entwicklung mussten daher immer wieder Beispielanwendungen zu Testzwecken geschrieben werden, welche die gerade in Entwicklung befindlichen Features nutzen. Die Library musste dabei nach jeder Änderung in das Beispielprojekt kopiert werden. Zu diesem Zweck wäre ein lokaler Package-Manager wie npm eine große Hilfe gewesen. Es wurde mit „Grunt" zwar ein Build-Tool verwendet dieses ist aber nicht auf die Verteilung von Code ausgelegt und konnte dieses Problem daher nicht hinreichend lösen.

## 11.2 Praxis Einsatz

Der Einsatz der Bibliothek in den erwähnten Testprojekten, sowie in der in dieser Arbeit beschriebenen TicTacToe-Anwendung hat gezeigt, dass die Bibliothek funktioniert und die Synchronisierung komplexer Strukturen auf einfache Art ermöglicht. Allerdings zeigte sie auch, dass vor allem bei der Installation und Einbindung des NodeJS Moduls noch Verbesserungsbedarf besteht.

Sobald die Bibliothek läuft, erledigt sie ihre Arbeit vorbildlich. Eine Testanwendung, in der das Spiel „Dame" umgesetzt wurde[7], erlaubt beispielsweise das synchronisierte Bewegen von Spielsteinen per Drag&Drop in Echtzeit und beweist somit, dass die Anforderung der weichen Echtzeit auch bei komplexen Aufgaben gegeben ist.

Die Arbeit mit der Bibliothek zeigte außerdem, dass in vielen Fällen nicht die Umsetzung, sondern die Planung einer Lösung den anspruchsvolleren Teil der Aufgabe darstellt. Insbesondere Aufgaben, die nicht synchron sind, wie zum Beispiel die Auswahl des Spielsteins in der TicTacToe-Anwendung, können mit der Bibliothek bisher nur schwer umgesetzt werden.

Trotz dieser Probleme wurden alle definierten Entwicklungsziele erreicht. HTML-Sync ist eine überaus flexible Bibliothek zum Synchronisieren von HTML-Elementen geworden und stellt somit, obgleich sie noch nicht als Lösung für Unternehmen zu empfehlen ist, zumindest ein sehr gelungenes Proof of Concept dar.

[7] http://html-sync.herokuapp.com/dame/

## 11.3 Ausblick

Schon früh während der Entwicklung wurde die Library im npm Packet-Manager veröffentlicht. Die Downloadzahlen[8] lassen darauf schließen, dass in jedem Fall Interesse für ein System wie HTML-Sync besteht. Zudem soll auch der ursprüngliche Code des Startups „Yavango“ zukünftig durch die entwickelte Bibliothek ersetzt werden. Somit ist eine Motivation für Weiterentwicklung des Projekts mehr als gegeben.

Um die Bibliothek für eine breite Masse von Entwicklern interessant zu machen, müssen vor allem die Integration des Moduls sowie die Dokumentation verbessert werden. Zudem sollen neue Komfortfunktionen die teilweise etwas umständliche Arbeit mit der Bibliothek weiter verbessern.

Im aktuellen Zustand kommt es zu Problemen mit bereits belegten Ports und Kernelemente der Library sind in anderen Bereichen nicht immer verfügbar, sollten einige Schritte beim Einbinden der Bibliothek nicht ganz genau eingehalten werden. Dies und die Tatsache, dass eben jene Schritte nicht ausreichend dokumentiert sind bilden den Hauptgrund dafür, dass die Bibliothek noch nicht von einer breiten Masse genutzt werden kann.

Die Update-Methode, die eines der wichtigsten Features der Library ist, ist ebenfalls noch zu umständlich zu bedienen. Das Übergeben eines verschachtelten Objekts erfordert, dass sich Entwickler alle möglichen Attribute merken müssen. Eine Lösung mit einer „UpdateData“-Klasse könnte hier in Zukunft Abhilfe schaffen.

Auch im Bereich Performance gibt es noch Verbesserungsmöglichkeiten. Die ersten Tests haben erstaunlich gute Ergebnisse gezeigt, doch unter realen Bedingungen könnte es trotzdem noch zu Problemen im Hinblick auf die Serverleistung kommen.

Insbesondere beim Speichern der Part-Objekte und Räume gibt es zum Beispiel noch viele Felder mit leeren Arrays. Dies ist zwar während der Entwicklung angenehmer, da nicht bei jeder Abfrage überprüft werden muss ob der Wert eines Schlüssels undefiniert ist, führt aber zu mehr Speicherbedarf sowohl im Arbeitsspeicher als auch in der Datenbank. Eine Funktion, die beim Speichern und Laden solche leeren Elemente entfernt könnte die Performance der Bibliothek weiter steigern.

[8] https://www.npmjs.com/package/html-sync

# 12 Abbildungsverzeichnis

Abbildung 1: UML-Diagramm ..... 10
Abbildung 2: Darstellung eines Zirkelbezugs ..... 14
Abbildung 3: Ablauf eines Drag-Events bei fehlerhaftem Update-Verhalten ..... 24
Abbildung 4: Zeitlicher Ablauf mehrerer Updates desselben Objekts ..... 25
Abbildung 5: Ablauf bei der Verbindung mit mehreren Räumen ..... 27
Abbildung 6: Inhalt des TicTacToe-Verzeichnisses nach der Installation der Module. ..... 32
Abbildung 7: Die fertige TicTacToe-Website ..... 39
Abbildung 8: Thousand Stars Website ..... 41
Abbildung 9: Das fertige Dame-Spiel ..... 43
Abbildung 10: Die Tri Website mit dem Seed „2345636452“ ..... 44
Abbildung 11: Tri Beispiel mit dem Seed „bachelorarbeit“ ..... 46
Abbildung 12: Auschnitt der Klassenhirachie des Yavango DESK ..... 48
Abbildung 13: Yavango DESK mit PDF ..... 49

# 13 Literaturverzeichnis

*Expressjs.com.* 31. 03 2016. http://expressjs.com (Zugriff am 31. 03 2016).

Höfler, Tobias. „sebis.“ *sebis.* 15. 05 2013. https://wwwmatthes.in.tum.de/pages/12k18c188un3w/Master-Thesis-Tobias-Hoefler (Zugriff am 09. 06 2016).

Microsoft. *TypeScript.* 31. 03 2016. http://www.typescriptlang.org/ (Zugriff am 31. 03 2016).

MongoDB, Inc. *How does MongoDB wokr?* 31. 03 2016. https://www.mongodb.com/what-is-mognodb (Zugriff am 31. 03 2016).

*NodeJS API.* 11. 05 2016. https://nodejs.org/api/events.html (Zugriff am 11. 05 2016).

Osmani, Addy. *Learning JavaScript Design Patterns.* O'Reilly Media, Inc., n.d.

Reschke, Fiedling &. „RFC 7230 HTTP/1.1 Message Syntax and Routing.“ *RFC 7230 HTTP/1.1 Message Syntax and Routing.* 06 2014. https://tools.ietf.org/html/rfc7230 (Zugriff am 30. 03 2016).

*Socket.io.* 31. 03 2016. http://socket.io (Zugriff am 31. 03 2016).

Suter, Rico. „MongoDB An Introduction and performance analysis.“ *MongoDB An Introduction and performance analysis.* 01. 01 2012. http://wiki.hsr.ch/Datenbanken/files/MongoDB.pdf (Zugriff am 29. 05 2016).

Teixeira, Pedro. *Module Patterns.* YLD publishing, 2015.

**Eigenständigkeitserklärung**

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich oder dem Sinn nach entnommenen Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Hamburg den, 17.06.2016 (Lars Krafft)