Verteiltes Veranstaltungsmanagement mit einer mobilen Webanwendung

Bachelorarbeit

von

Christian Meter

aus

Remscheid

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

September 2013

Betreuer:
Philipp Hagemeister M. Sc.

Danksagungen

Mit vielen Leuten konnte ich wichtige Gespräche führen, die mir thematisch und inhaltlich sehr weitergeholfen haben. Viele haben mich auch technisch unterstützt, was mir eine große Hilfe war. Daher danke ich allen, die mich bei dieser Arbeit unterstützt haben.

Danke auch an alle Korrektoren dieser Arbeit. Gerade die, die nicht vom Fach waren, haben sicherlich schwere Arbeit leisten müssen, um den Überblick zu behalten.

Ein besonderer Dank gilt meinem Betreuer Philipp Hagemeister, welcher immer ein offenes Ohr hatte und mir stets geholfen hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Jede große Veranstaltung muss mit viel Aufwand geplant und organisiert werden. Viele Hände sind dafür nötig, doch wie kann man diese Hände sinnvoll verwenden und ihre Arbeit koordinieren?

Seit vielen Jahren bin ich Teil der bündischen Jugendbewegung im Deutschen Pfadfinderbund. Traditionsbewusst erinnert man sich an wichtige Ereignisse, wie der wohl bedeutendsten Verkündung der Jugendlichen in Deutschland im Jahr 1913 auf dem Hohen Meißner, einem 753,6 m über Normalnull hohen Berg in Nordhessen. Bei diesem Treffen entstand eine Formel, die die Entstehung deutscher bündischer Pfadfinder ausgelöst hat - die Meißner Formel:

> *„Die Freideutsche Jugend will nach eigener Bestimmung, vor eigener Verantwortung, in innerer Wahrhaftigkeit ihr Leben gestalten. Für diese innere Freiheit tritt Sie unter allen Umständen geschlossen ein.“* [Kin63, S. 109]

Seitdem sind 100 Jahre vergangen. Die Formel behält ihre Aktualität und es wird im Oktober das große Meißnerlager mit über 4.000 Pfadfindern und 90 freiwilligen Helfern an eben jenem Ort von 1913 stattfinden.

1.1 Ziele der Arbeit

Als Ziel soll eine Webanwendung entwickelt werden, die auf nahezu allen modernen Endgeräten ungeachtet des benutzten Betriebssystems lauffähig ist. Sie muss dabei grundlegende Funktionen zur Erstellung und Verwaltung von Veranstaltungen und deren zugehörigen Teilnehmern besitzen. Diese Einträge sollen zentral in einer MySQL-Datenbank gespeichert werden und auf diese Datenbank sollen die zugeordneten Helfer entsprechend ihren Rechten Daten einsehen und verändern können.
Die Meißner App soll eine mobile Seite bereitstellen, die für touchfähige Geräte optimiert ist und somit die bequeme Nutzung von Smartphones und Tablets ermöglicht.
Das Meißnerlager findet unter freiem Himmel statt, sodass die App für den Außenbereich und die mobile Nutzung optimiert sein muss. Auf knapp 40.000 m^2 Lagerplatzfläche werden sich ca. 90 freiwillige Helfer zwischen den ca. 4.000 erwarteten Teilnehmern bewegen. Dabei werden sie eigene Smartphones mit zu der Veranstaltung nehmen und über ihren Zugang zu der Meißner App ihre Aufgaben organisieren können. Damit die App auch bei schlechtem Empfang auf diesen Smartphones funktioniert, muss die Anwendung kritische Dateien cachen und lokal auf dem Gerät speichern.

Die Organisation durch diese Anwendung wird so einfach gehalten, dass sie von ungelernten Helfern produktiv genutzt werden kann. Mit einer einfachen Implementierung wird die App auch auf einem leistungsschwachem Server mit ausreichender Geschwindigkeit ausführbar sein, sodass keine kostenintensiven Computer notwendig sind.

Diese App wird im Oktober 2013 bei der oben beschriebenen Veranstaltung getestet und verwendet. Die dafür nötige Infrastruktur wird durch ein großes Telekommunikationsunternehmen gefördert. Sollte es zu Empfangsproblemen über die interne Antenne kommen, können sich die Smartphones in die installierten LTE Router einwählen.

Über die Homepage `http://www.meissner-2013.de` kann man sich über ein hinterlegtes PDF Formular zu der Veranstaltung anmelden. Dort werden viele Details abgefragt, welche den Großteil der Daten in der Datenbank ausmachen. Eine PDF wird verwendet, da die Pfadfinder oft klassisch veranlagt sind und die Anmeldung lieber ausdrucken und von Hand ausfüllen.

1.2 Gliederung

Das zweite Kapitel befasst sich mit der Implementierung der Grundfunktionen einer Webanwendung, Kapitel drei und vier erläutern Erweiterungen, die über die Grundausstattung der App hinausgehen.
Kapitel fünf befasst sich mit Benchmarks des Servers, um eine Vorstellung der Skalierbarkeit zu erhalten. Im sechsten Kapitel wurden ähnliche Arbeiten gesucht und die Unterschiede kurz verdeutlicht. Das siebte und damit letzte Kapitel fasst die Ergebnisse dieser Arbeit kurz zusammen und gibt einen Ausblick für potentielle Erweiterungen.

Kapitel 2

Implementierung

Identifizierung des Problems Bei dem Gedanken eine App zu entwickeln, steht man oft vor der Entscheidung für welche Plattform man denn entwickeln möchte: Linux, Windows, OS X, iOS, Android, Windows Phone, Blackberry OS etc. Doch das würde voraussetzen, dass alle Benutzer dieser App das gleiche Betriebssystem benutzen. Für eine Veranstaltung, wie das Meißnerlager, eine sehr schlechte Annahme. Die ehrenamtlichen Helfer müssen mit ihren eigenen / privaten Smartphones und Tablets arbeiten. Es ist davon auszugehen, dass es sich dabei um unterschiedliche Geräte mit verschiedenen Betriebssystemen handelt.

Wieso also nicht für alle Plattformen gleichermaßen entwickeln? Dank dem Google Projekt „HTML5 Rocks“ [Goo13d] und vielen interessierten Entwicklern ist *HTML5* in aller Munde. Somit gibt es tatsächlich die Möglichkeit, jedes Betriebssystem mit einer einzigen Webanwendung abzudecken.

Definition Webanwendung Eine Webanwendung, auch *Web-App* genannt, ist eine Anwendung, die vollständig in einem Browser ausführbar ist. Da sie (fast) vollständig auf einem Webserver ausgeführt wird, ist das zugrunde liegende Übertragungsprotokoll *HTTP*.

Das Interessante an dieser Art der Anwendungen ist, dass sie auch auf Smartphones und Tablets ausgeführt werden können, ohne sie vollständig auf die jeweilige Plattform por-

tieren zu müssen. Das heißt, man entwickelt einmal eine Webanwendung und kann sie dann auf allen gängigen Endgeräten ausführen.
Damit wäre ein wichtiges Kriterium für die Nutzung beim Meißner erfüllt: die betriebssystemunabhängige Nutzung.

Auszeichnungssprache HTML5 *HTML5* ist der angehende Nachfolger des aktuell verwendeten HTML 4.0.1. Es ist immer noch im experimentellen Entwicklungsstand, wird aber durch große Unternehmen wie Google gefördert und erreichte damit auch eine große Beliebtheit. Alle Browser verstehen die gängigsten HTML5 Befehle, allerdings werden nie alle Funktionen unterstützt. Der Grundbefehlssatz ist in allen Browsern enthalten, weshalb es schon seit einiger Zeit möglich ist, mit dem neuen angehenden Webstandard seine Web-App zu gestalten.

Framework Mit der Wahl des Open Source Frameworks *cakePHP* [CSF13] stehen auch serverseitig *PHP5* und clientseitig *JavaScript* als weitere Programmiersprachen fest. Als Datenbank wird *MySQL* gewählt und für den Webserver kommt *Apache* zum Einsatz, beides ist durch das Entwicklerpaket *XAMPP* gegeben.

Außerdem wurde das Framework *socket.io* [Rau13] gewählt, da es mit wenig Aufwand einen stabilen WebSocket Server bereitstellt. Dieser beinhaltet wichtige Funktionen, wie Broadcast, Fallbacks und Socketverwaltung, welche von dieser Arbeit verwendet werden.

2.1 Struktur der Webanwendung

Für die Grundstruktur wurde das Open Source Framework cakePHP verwendet. Es verwendet für eine klare Trennung der Logik und dem Design das Entwurfsmuster *Model View Controller*.

2.1.1 Entwurfsmuster: Model View Controller

Befasst man sich mit dem Entwickeln von modernen Webseiten, Apps oder ähnlichen Projekten, so stößt man direkt auf das Muster zur Strukturierung als Model View Controller (*MVC*, deutsch: Modell-Präsentation-Steuerung). Dieses Muster kapselt drei Elemente voneinander und ermöglicht dadurch, dass Änderungen einfach implementiert werden können.

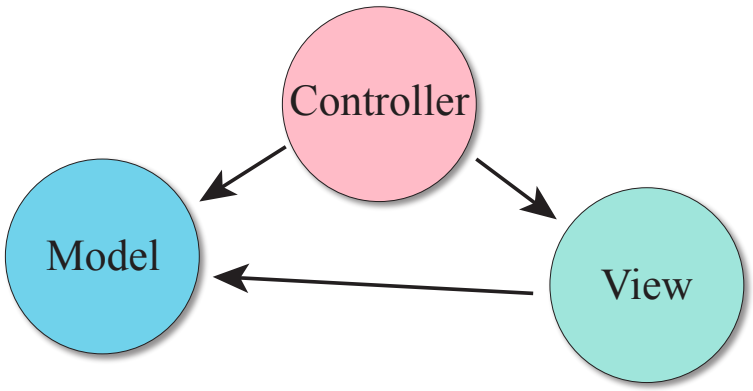

Abbildung 2.1: Direkte Assoziationen zwischen Model, View und Controller

Model: Verantwortlich für alles, was die Daten eines bestimmten Models angeht. In der Anwendung ist dieser Bereich für Interaktionen, Gültigkeit und Auswertung verantwortlich und repräsentiert hier zum Beispiel eine Veranstaltung, einen Benutzer, Statistiken oder Koordinaten der Benutzer.

View: Generiert das Aussehen einer bestimmten Seite. Hier wird vorwiegend mit gängigen Webformaten, wie HTML, PHP und JavaScript gearbeitet. Ein View kann auf die Daten zugreifen, die ihm der Controller liefert und diese dann anschaulich darstellen. Dadurch muss sich ein View nicht darum kümmern, woher er seine Informationen zusammensuchen muss, sondern bekommt garantiert, dass diese vom Controller bereitgestellt werden. Auf diese Weise können verschiedene Views für verschiedene Geräte erstellt werden und erlauben einfach eine mobile Seite zu generieren.

Controller: Hier findet die gesamte Logik und Aufbereitung der Daten statt, die später dem Benutzer im View angezeigt werden sollen. Sämtliche Datenbankabfragen, Berechnungen oder Ähnliches werden hier durchgeführt und dem Model / View bereitgestellt. Für jedes Objekt existiert ein eigener Controller. Diese werden *EventsController*, *Users-Controller* etc. genannt.
In diesem Framework ist jede Methode direkt mit den Views verknüpft, das heißt definiert man im EventsController eine Funktion namens *index()*, so stellt dies eine Seite der Webanwendung dar, die mit *http://localhost/Events/index* aufgerufen werden kann und den entsprechenden View in */View/Events/index.ctp* erwartet.

Die Vorteile bei diesem Entwurfsmuster liegen auf der Hand: Es muss nur einmal ein Objekt im Model definiert werden. Im Controller wird die komplette Logik verarbeitet und damit kann man mit entsprechenden Views für verschiedene Ansichten (z.B. eine mobile Anwendung der App) einfach auf die Daten des Controllers zugreifen. Der Code kann dadurch kompakter gehalten werden und muss nur im Controller angepasst werden, um auf allen Ansichten der Seite die Daten aktualisieren zu können.

2.2 Datenbanken

Um alle Daten schnell und einfach verarbeiten zu können, ist eine Datenbank nötig. Da es sich hier um eine Webanwendung handelt, bietet sich eine SQL Datenbank an, weil diese in der Regel zum Komplettpaket eines Webservers gehört.
Folgende Tabellen wurden angelegt:

users: Ein Benutzer besteht aus einer eindeutigen Identifizierungsnummer (*ID*), einem Benutzernamen, einem Passwort und einer zugewiesenen Rolle (z.B. Administrator, Benutzer usw.). Dazu werden von der App automatisch zwei Zeitstempel mit Erstellungs- und letztem Änderungsdatum sowie einem Boolean hinzugefügt, welcher angibt, ob der Benutzer sich im System der Webanwendung einloggen kann und damit Daten einsehen kann.

events: Verpflichtende Felder sind ein Titel und eine Beschreibung. Dazu wird von der App eine eindeutige ID zugewiesen und die Benutzer-ID des Benutzers hinzugefügt, welcher diese Veranstaltung erstellt. Dazu noch zwei Zeitstempel: Erstellungsdatum und Zeitpunkt der letzten Änderung.

Dies sind die beiden grundlegenden Tabellen, die für diese Anwendung nötig sind. Jedoch besteht noch keine Verknüpfung untereinander, abgesehen von der Benutzer-ID in der *events*-Tabelle.

2.2.1 Verknüpfung von Benutzer und Veranstaltung

Um einer Veranstaltung beliebig viele Benutzer zuordnen zu können, wurde die Tabelle *events_users* erstellt. Diese beinhaltet nur die Fremdschlüssel ID aus den *events*- und *users*-Tabellen.

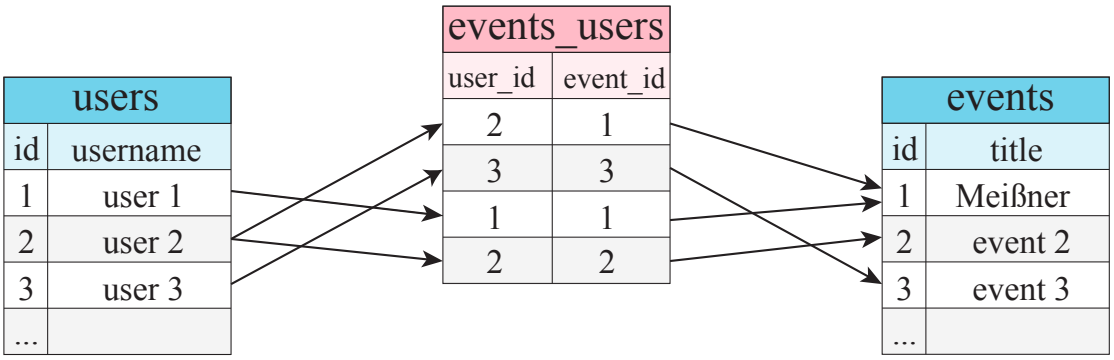

Abbildung 2.2: Zuweisung eines Benutzers zu einer Veranstaltung (vereinfacht)

Diese *Many-to-many*-Verknüpfung (in cakePHP auch *hasAndBelongsToMany, HABTM* genannt) kann der EventsController mit einer einzigen SQL-Abfrage erfragen und erhält damit alle dem Event zugeordneten Benutzer. Diese Daten können dann in einer Variable gespeichert und dem View bereitgestellt werden.

2.2.2 Veranstaltungsspezifische Eigenschaften

Die ersten Schritte zu einer Webanwendung, die Veranstaltungen und Benutzer verwalten kann, sind nun erledigt. Bisher ist es allerdings nicht möglich in Veranstaltungen Felder zu definieren, die für eben jene Veranstaltung von Bedeutung sind. Das können im konkreten Fall Felder sein wie „Art des Transportmittels“, „Wird ein Parkplatz benötigt?“, „Welcher Holzstangenbedarf besteht?“ oder Ähnliches.

Key-Value-Store In einem Key-Value-Store wird die Datenbank mit den Werten (*values*) über die Schlüssel (*keys*) indexiert. Der Vorteil ist, dass vorher nicht bekannt sein muss, welche Werte in die Datenbank eingetragen werden sollen.
In der Meißner App wurde diese Methode implementiert, um eventspezifisch ein beliebiges Feld zu definieren, welches Werte als Strings annimmt. So ist gewährleistet, dass die Benutzer dieser Anwendung die größtmögliche Freiheit besitzen, was die Inhalte der Veranstaltungen angeht. Jede kann individuell angelegt und personalisiert werden, sodass sie den gewünschten Anforderungen genügt.
Um die häufigsten Nutzungsszenarien abzudecken, wurden Strings als Hauptformat für die Values gewählt. Diese lassen sich einfach auswerten und erhalten die einfache Bedienung der Formulare.
Für die automatischen Statistiken ist das Format irrelevant, da nur auf Gleichheit der Strings geprüft wird und anhand dessen die Grafiken erstellt werden.

Implementierung Für die Implementierung sind zwei weitere Tabellen notwendig:

1. *event_columns*: Dient als Maske, um Feldnamen und -typen zu definieren. Kann zudem innerhalb einer speziellen Veranstaltung genutzt werden, um diese Felder den zugeordneten Benutzern verfügbar zu machen.

2. *event_properties*: Im View wird ein Formular generiert, welches die Feldnamen aus *event_columns* anzeigt und die Möglichkeit gibt, Werte einzutragen. Diese Werte werden dann über den Controller in *event_properties* gespeichert.

Mit den Tabellen steht nun die Datenstruktur zur Verfügung, die es ermöglicht veranstaltungsspezifische Eigenschaften zu erstellen und die so erzeugten Felder mit den entsprechenden Werten der Teilnehmer dieser Veranstaltung zu befüllen.

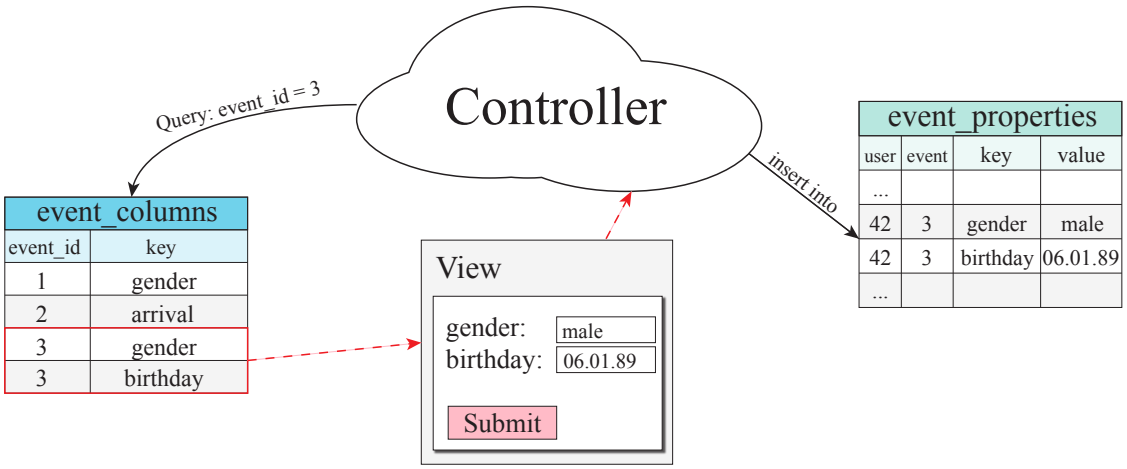

Abbildung 2.3: Controller erfragt Felder aus event_columns, stellt sie dem View zur Verfügung, speichert danach die Werte in event_properties

Obige Abbildung zeigt kurz das Zusammenspiel von Controller und View: Datenbankzugriffe werden über den View an den Controller gerichtet und dort ausgeführt. So können Daten abgefragt oder neue Daten geschrieben werden.

Als einzigartige primäre Schlüssel werden *user_id* und *key* gewählt, womit automatisch Duplikate ausgeschlossen werden. Somit kann immer nur genau ein Wert in einem benutzerspezifischen Feld des Benutzers stehen.
Beispiel: In der Abbildung existiert die Zeile *Benutzer: 42, Event: 3, Key: gender, Value: male*. Wenn der Benutzer nun das Geschlecht in dem Formular ändert und das abschickt, dann sieht die Nachricht so aus: *Benutzer: 42, Event: 3, Key: gender, Value: female*. Weil das Feld *key* als Primärschlüssel definiert wurde, wird dieser Eintrag überschrieben und kein neuer Eintrag in der Datenbank angelegt.

2.3 Verhinderung von Angriffen

Gängige Angriffsmethoden, wie Cross-Site-Scripting oder SQL-Injection, werden in dieser Arbeit weitgehend verhindert.

2.3.1 SQL-Injection

Bei der *SQL-Injection* versucht ein Angreifer durch geschicktes Manipulieren von Eingabefeldern eines Formulars, einen String zu erzeugen, mit dem er einen ungewollten Zugang zur Datenbank erhält. Schafft er das, so kann er Datensätze aus der SQL Tabelle ausgeben lassen, verändern oder sogar komplett löschen.

Um diese Art der Angriffe zu verhindern, haben die Entwickler von cakePHP einige Sicherheitsvorkehrungen implementiert. Zusätzlich gibt es das Modul *Sanitize*, welches die Eingabestrings des Benutzers überprüft und alle Sonderzeichen, wie Hochkommata und Semikolons, aus dem String entfernt. Damit lässt sich keine zusätzliche Anweisung in den SQL-Query mit einbauen und die Anwendung ist somit weitgehend vor diesen Angriffen geschützt.

2.3.2 Cross-Site-Scripting XSS

Mit *XSS* beschreibt man eine Angriffsmethode, die webseitenübergreifend Schadcode über normale Hyperlinks ausführt. Dabei können bei Clients, die JavaScript unterstützen, Cookies an den Angreifer übertragen werden.

Auch hier übernimmt cakePHP das Abwehren dieser Angriffe. Damit diese Mechanismen auch greifen, werden in dieser Webanwendung die von den Entwicklern vorgeschlagenen Paradigmen beachtet.

2.4 Views entwickeln

Nun wurde mit modernen Webtechnologien und möglichst geringem Aufwand direkt eine mobile Seite der Anwendung erstellt, welche für Smartphone und Tablets optimiert ist. Schließlich soll und muss diese Web-App auch auf dem Gelände mobil genutzt werden können.

Dabei gibt es mehrere Ansätze, wobei in dieser Arbeit das sogenannte *Responsive Webdesign* (deutsch: bedarfsgesteuertes Webdesign) für die mobile Version der Anwendung benutzt wird.

2.4.1 Responsive Webdesign mit jQuery Mobile

Hauptaugenmerk wird hier auf die Auflösung des Endgeräts gelegt: Handelt es sich hierbei um ein günstiges Smartphone mit schlechtem Display, um ein Google Nexus 10 mit 2560x1600 Pixeln oder um einen SmartTV mit FullHD?
Diese Frage ist entscheidend, denn sie bestimmt wie viele und wie groß bestimmte Objekte im Sichtbereich des Geräts sein können, sodass der Benutzer nicht von der schlechten Bedienung genervt ist, sondern weiterhin die Informationen aus der App holen möchte. Beim Responsive Webdesign werden über *Media Queries* (Abfrage der Eigenschaften des Geräts, Beispiel: Auflösung, Orientierung des Endgeräts usw.) Informationen des Endgeräts eingeholt und darauf abgestimmte Stylesheets geladen.

Um aber eine Webanwendung zu entwickeln, die aussieht und sich „anfühlt“ wie eine echte App wurde in dieser Arbeit das Framework *jQuery Mobile* verwendet, welches auch auf Responsive Webdesign setzt und dabei viele Steuerelemente zur Verfügung stellt, die dem Benutzer schon von nativen Apps her bekannt sind, wie typische Slider, Buttons, Dropdown-Menüs und vielem mehr.

Mobile View Damit jQuery Mobile aus normalen Links touchoptimierte Bedienelemente erzeugt, konnte nahezu der komplette Code von der Desktop-Version genommen werden. Lediglich kleine Ergänzungen um das *data-role* Attribut sind notwendig, damit jQuery Mobile diese anpassen kann, das bedeutet ein normaler Hyperlink wird um *data-role="button"* erweitert.

```
<a href="#">normal link</a>
<a href="#" data-role="button">touchoptimized link</a>
```

Listing 2.1: Eine kleine Ergänzung erzeugt aus einem Link einen touchfreundlichen Button

Auf ähnliche Art und Weise können auch viele andere Elemente umgewandelt werden, damit diese für Smartphones optimiert sind. Oft erfolgt dies automatisch, da jQuery Mobile Bedienelemente erkennt und diese dann sofort anpasst.

Kapitel 3

Echtzeitaktualisierung mit WebSockets

Mit der Echtzeitaktualisierung wird ein zeitlich direkter Datenaustausch zwischen Endgeräten über einen separaten WebSocket Server (kurz: WS Server) ermöglicht.

3.1 Einführung in WebSockets

Mit der HTML5-Spezifikation wurden *WebSockets*, ein auf *TCP* basierendes Protokoll der Anwendungsschicht, eingeführt, welches eine Vollduplex, bidirektionale, Single-Socket Verbindung ermöglicht [WSM12, S. 7]. Eine Anfrage öffnet die Verbindung zum WebSocket Server und kann beliebig lange offen gehalten werden, wobei zu jeder Zeit Daten zwischen Client und Server ausgetauscht werden. Dieser Datenaustausch geschieht sehr schnell und mit einem sehr kleinen Overhead, da die Verbindung nicht neu aufgebaut werden muss und die Daten somit sofort verschickt werden können.
Ähnlich wie „normale“ Sockets werden WebSockets von einer Anwendung erzeugt und stellen dann die Schnittstelle zwischen der Webanwendung und dem Transportprotokoll TCP dar.
Im direkten Vergleich zu den generischen Sockets, erfüllen WebSockets eine ähnliche Funktion, sind allerdings auf Webanwendungen und TCP beschränkt.

Handshake Der Handshake erfolgt über HTTP/1.1 und ähnelt dem zum Aufruf einer Homepage:

```
GET / HTTP/1.1
Host: server.example.com
Origin: http://www.example.com
Sec-WebSocket-Key: 7+C600xYyb0v2zmJ69RQsw==
Sec-WebSocket-Version: 13
Upgrade: WebSocket
```

Listing 3.1: HTTP Request des Clients [IFG+11, S. 6]

Mit *Upgrade: WebSocket* wird signalisiert, dass der Client eine WebSocket-Verbindung zum Server aufbauen möchte. Der entsprechende WS Server reagiert darauf mit dem HTTP-Statuscode *101 Switching Protocols*, womit er bestätigt, dass er mit dem Wechsel des Protokolls einverstanden ist.

```
101 Switching Protocols
Connection: Upgrade
Sec-WebSocket-Accept: fYoqiH14DgI+5y1EMwM2sOLzOi0=
Upgrade: WebSocket
```

Listing 3.2: HTTP Response des Servers [IFG+11, S. 8]

Der kryptische Schlüssel *Sec-WebSocket-Accept* muss vom Server berechnet und zurückgegeben werden und zeigt damit, dass er das WebSocket Protokoll versteht. Ab diesem Zeitpunkt ist der Handshake abgeschlossen und die Verbindung wurde aufgebaut.

Nun kommen die Vorteile von WebSockets zur Geltung: nach erstmaligem Aufbau der Verbindung bleibt diese geöffnet und zu jedem Zeitpunkt können Client und Server die Vollduplex Verbindung nutzen, um Daten miteinander auszutauschen. So kann ein fast volles TCP innerhalb einer Webanwendung verwendet werden.

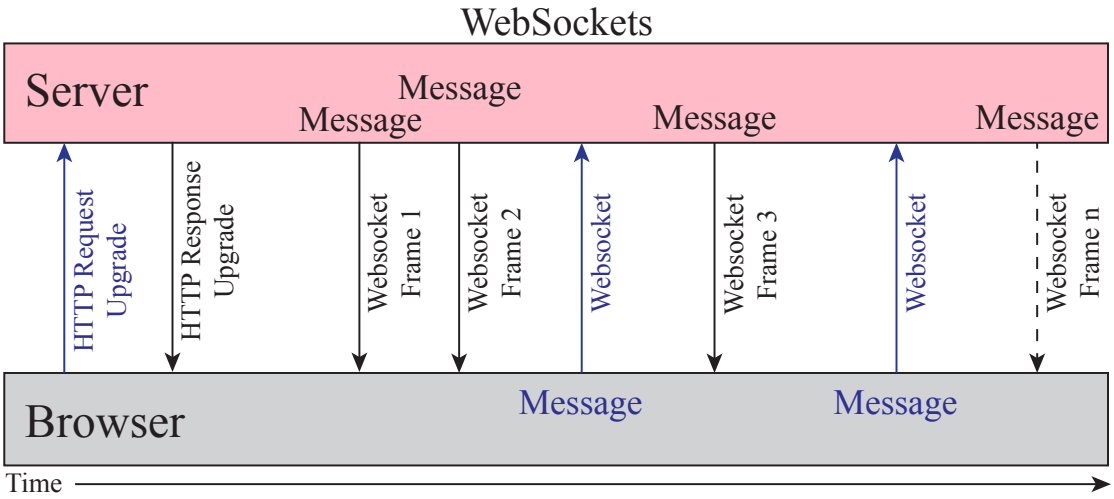

Abbildung 3.1: Aufbau einer WebSocket Verbindung. Danach können beliebig Nachrichten in beide Richtungen (gleichzeitig) übermittelt werden [WSM12, S. 7]

Es verringern sich außerdem die Latenzzeiten, der Traffic und die CPU-Leistung der Server. Das liegt daran, dass die Pakete direkt zugestellt werden können, der Overhead dank einmaligem Handshake nur 2 Byte beträgt und der Austausch der Daten sehr einfach gestaltet ist.

3.2 Implementierung der Echtzeitaktualisierung

Im ersten Ansatz dieser Arbeit wurde der WebSocket-Server auf Basis von *node.js* [joy13], einem serverseitigem JavaScript Framework für Server, implementiert. Das verlief aufgrund der einfachen Handhabung von WebSockets und der guten Unterstützung von HTML5 problemlos. Jedoch wurden zu diesem Zeitpunkt weder verschlüsselte Verbindungen noch Broadcasting unterstützt. Daher war dies ungeeignet, weil sensible Daten, wie die Standorte der Clients, nicht im Klartext verschickt werden sollten.
Außerdem ist eine Broadcast-Funktion notwendig, die den bestehenden Sockets bei Aktualisierung einer Position sofort die neuen Koordinaten übermittelt.

Die Implementierung beider Funktionen hätte den Rahmen dieser Arbeit überschritten, weshalb serverseitig das Open Source Framework *socket.io* für node.js verwendet wird. Es stellt sämtliche benötigten Dienste sowie einen Fallback bereit, wodurch auch älteren Browsern ohne HTML5 Unterstützung eine websocketähnliche Verbindung ermöglicht wird.

Clientseitig wurde ein Skript in JavaScript entwickelt, welche im Hintergrund der Web-App ausgeführt wird. Es verbindet sich mit dem WebSocket Server, schickt initial die aktuellen Koordinaten unabhängig davon, welcher View gerade angezeigt wird und wartet auf eingehende Nachrichten oder auf eine Änderung der eigenen Position. Bewegt sich der Client, werden automatisch die Koordinaten GoogleMaps-kompatibel in Form von Längen- und Breitengrad an den WebSocket Server übermittelt.
Erhält der Server so eine Nachricht, aktualisiert er seine Datenbank von Standorten der Clients und schickt per Broadcast eine Nachricht an alle verbundenen Clients. So erhalten die Endgeräte in Echtzeit eine Aktualisierung aller Positionen.

Fallback Nicht jedes Endgerät unterstützt die Verwendung von WebSockets. Zwar gab es keine Komplikationen mit den getesteten Geräten, da WebSockets mit allen gängigen Desktop Browsern und fast allen mobilen Browsern außer Opera Mini lauffähig ist. Bei veralteter Software ist eine eine Unterstützung jedoch nicht immer gegeben. Daher wurde socket.io so eingerichtet, dass ein Fallback über *flashsockets* hin zum *polling* ermöglicht wird. So kann der größte Teil aller Browser die Echtzeitaktualisierung verwenden.

Entwicklung eines eigenen Protokolls zur Kommunikation über WebSockets Um bestimmte Funktionen auf dem Server anzusprechen, wurde ein einfaches Protokoll entwickelt, welches angibt, um welchen Inhalt es sich bei der Nachricht handelt. Alle Nachrichten, die zwischen Client und Server ausgetauscht werden, werden vorher intern als JSON Objekte benutzt und für die Übertragung über die WebSockets in Strings konvertiert. Dabei enthalten alle Nachrichten ein Feld *type*, welches folgende Werte annehmen kann:

location: enthält Latitude und Longitude des Clients

syn: enthält eine Signatur für die Authentifizierung

subscribe: abonniert bestimmte Events, dazu später mehr

publish: signalisiert eine Änderung an der Datenbank

message: eine neue Chat Nachricht hat den Server erreicht und wird per Broadcast an die Clients verschickt

history: Anfrage eines Clients nach der aktuellen Historie des Chats

Der socket.io Server wertet diese Fälle über ein *Switch-Case-Statement* aus und ruft entsprechende Methoden zur weiteren Verarbeitung auf.

3.3 Kommunikation zwischen Apache und WebSocket Server

Bei dem Webserver Apache und dem WebSocket Server mit dem Framework socket.io handelt es sich um zwei Anwendungen, die selbstständig und unabhängig voneinander aktiv sind. Beide verwenden HTTP und werden mit HTTP Requests angesprochen. Diese Implementierung sieht vor, dass beide parallel auf einem Server ausgeführt und mit verschiedenen Ports angesprochen werden.
Um die Berechtigungen des WebSocket Servers möglichst gering zu halten und diesen nur für die Echtzeitaktualisierung zu nutzen, hat socket.io keinerlei Rechte, um auf die Datenbank oder andere Elemente der Webanwendung zugreifen zu können. Dadurch bleibt die Kapselung beider Anwendungen erhalten, aber ein weiterer Schritt zur Authentifizierung eines Benutzers ist erforderlich.

WebSockets über PHP mit ElephantIO Damit die Meißner App selbst eine Verbindung zum WebSocket Server aufbauen kann, ist die Open Source Bibliothek *ElephantIO*

[Lud13] erforderlich. Sie ermöglicht die Kommunikation mit Apache und socket.io, da diese sonst nicht ohne Weiteres möglich ist. Dadurch ist die Anwendung in der Lage selbst Nachrichten über WebSockets zu verschicken.
Die Kommunikation in die andere Richtung ist nicht notwendig, da die Web-App keine Informationen benötigt welche Endgeräte sich über WebSockets angemeldet haben.

3.4 Authentifizierung eines Clients beim WebSocket Server

In der Webanwendung kann mit einem normalen Formular der Benutzername und das Passwort eingegeben werden, allerdings hat der WS Server bekanntlich keinen Zugang zur Datenbank. Es muss aber eine Authentifizierung am WebSocket Server stattfinden, damit nur Benutzer aus der Webanwendung eine Verbindung erstellen können. Dafür wird das *RSA Kryptosystem* verwendet.

Erzeugung der Schlüssel Zur ersten Initialisierung der Webanwendung erstellt die Webanwendung sich selbst per RSA-Verfahren einen 2048-Bit langen privaten und einen öffentlichen Schlüssel. Sie gehören ausschließlich der Meißner App selbst und werden lokal gespeichert. Der öffentliche Schlüssel wird dem WebSocket Server bereitgestellt, welcher sich diesen direkt aus dem entsprechenden Ordner in der Webanwendung laden kann. Das ist sinnvoll, weil WS Server und Webanwendung sich vertrauen können, da sie auf der gleichen Maschine laufen.

Signieren der Nachricht und anschließende Authentifizierung Wenn sich nun ein Client in der Webanwendung mit Benutzernamen und Passwort erfolgreich einloggt, nimmt die App den Benutzernamen *user* und erstellt mit dem privaten Schlüssel eine Signatur $sig = sign(user)$, welche dem Client zur Verfügung gestellt wird. Damit kann dieser nun einen syn-Request mit Benutzernamen und Signatur an den WebSocket Server schicken. Dieser überprüft nun die Signatur mit dem öffentlichen Schlüssel $verify(sig)$ und wenn diese der ursprünglichen Nachricht entspricht, ist der Client authentifiziert.

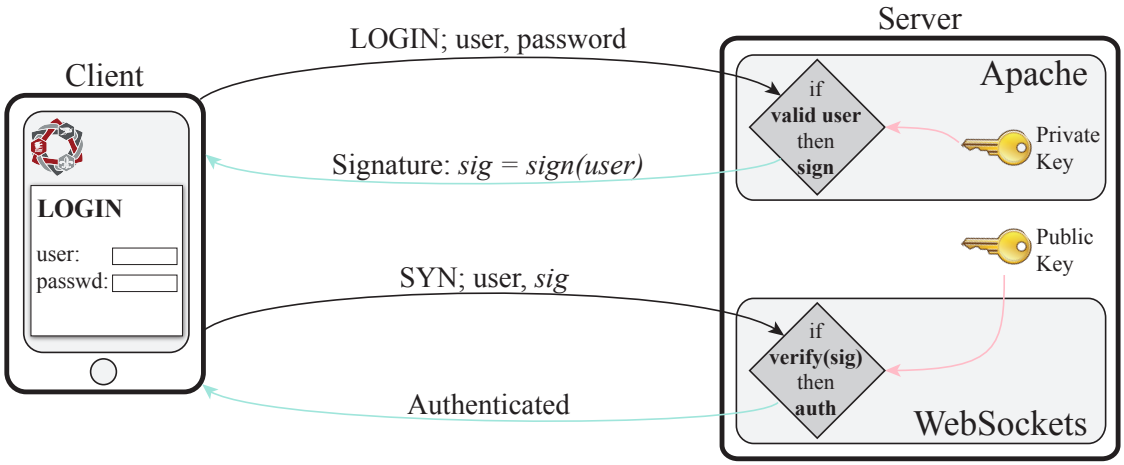

Abbildung 3.2: Authentifizierung eines Clients beim Server

Vorteile dieser Implementierung Durch diese Authentifizierung kann socket.io alle eingehenden Verbindungen ablehnen, die keine korrekte Signatur erhalten haben. Ein Angreifer muss somit über den Benutzernamen und das Passwort eines Benutzers verfügen, um sowohl zu der Webanwendung als auch zum WS Server einen Zugang zu erhalten. Er kann nun aber nicht ohne diese Daten eine Verbindung aufbauen und Daten abgreifen.

Dieser Weg der Authentifizierung wurde gewählt, da zum Zeitpunkt des Logins der WebSocket des Endgeräts noch nicht bekannt ist. Also kann die Webanwendung dem WS Server nicht mitteilen, welcher Benutzer zu welchem WebSocket gehört. Ein zweiter Schritt zur endgültigen Authentifizierung über eine Signatur ist also notwendig für die korrekte Zuordnung vom offenen WebSocket zum Benutzer.

3.5 Entwurfsmuster: Publish / Subscribe

Mit Publish / Subscribe (deutsch: veröffentlichen / abonnieren) wird ein Muster beschrieben, mit dem ein Client eine Benachrichtigung erhält, sofern ein Ereignis verändert wurde, welches von ihm abonniert wurde. Bei diesem Entwurfsmuster wissen weder Publisher noch Subscriber voneinander und es läuft vollkommen unabhängig, asynchron

miteinander. Sie müssen auch nichts voneinander wissen, da sämtlicher Datenaustausch über den socket.io Server läuft [Tav13].

Konkret können in dieser Anwendung Veranstaltungen abonniert werden. Das geschieht automatisch über die Webanwendung selbst, welche dem WebSocket Server mit Einloggen des Benutzers eine *subscribe*-Nachricht schickt. Ausgewählt werden die Events, die der Benutzer selbst erstellt hat.

Gab es eine Änderung an einer Veranstaltung, so meldet sich die Web-App beim WebSocket Server mit einer *publish*-Nachricht. Dieser schaut dann in seiner Liste der authentifizierten Clients nach, welcher die Veranstaltung abonniert hat und schickt an alle verbundenen Endgeräte eine Mitteilung, dass die Veranstaltung bearbeitet wurde.

Implementierung Im ersten Entwurf der Technik in dieser Arbeit wurden alle Nachrichten, die über WebSockets verschickt wurden, über die Clients selbst versendet, da die Kommunikation nur über JavaScript nativ unterstützt wird. Daher wurden auf Basis von Elephant.io Methoden entwickelt, die die publish-Funktion der Webanwendung übernehmen.
Der EventsController hat nun noch die zusätzliche Aufgabe bei jedem Datenbankzugriff eine Verbindung zum WS Server zu öffnen, eine publish-Nachricht abzuschicken, und dann die Verbindung wieder zu schließen.
Dadurch funktioniert Publish/Subscribe auch mit Endgeräten, die JavaScript verbieten. Die Clients, die Nachrichten erhalten möchten, bekommen diese nun ungehindert zugestellt.

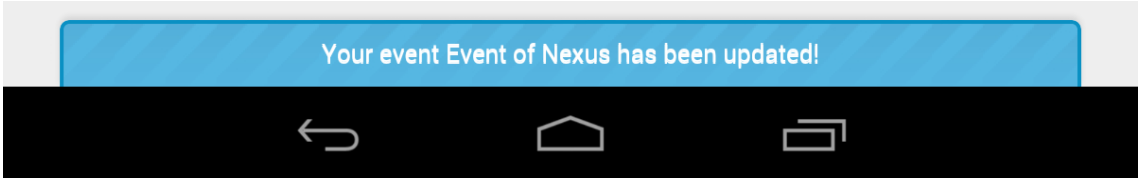

Abbildung 3.3: publish-Benachrichtigung am unteren Bildschirmrand mit Android 4.3

So ist es nun möglich über WebSockets eine Benachrichtigung an die Abonnenten zu schicken. Dadurch wird man aktiv informiert, wenn Änderungen an den Veranstaltungen vorgenommen werden.

3.6 Vor HTML5: Benutzung von Polling

Für den Datenverkehr von Internetseiten wird bekanntlich HTTP benutzt, welches die wechselseitige Datenübermittlung *Halbduplex* verwendet. So erfolgt der Datenverkehr nur in eine Richtung zur gleichen Zeit: der Client schickt eine Anfrage an den Server und dieser übermittelt danach die Antwort [WSM12, S. 5].

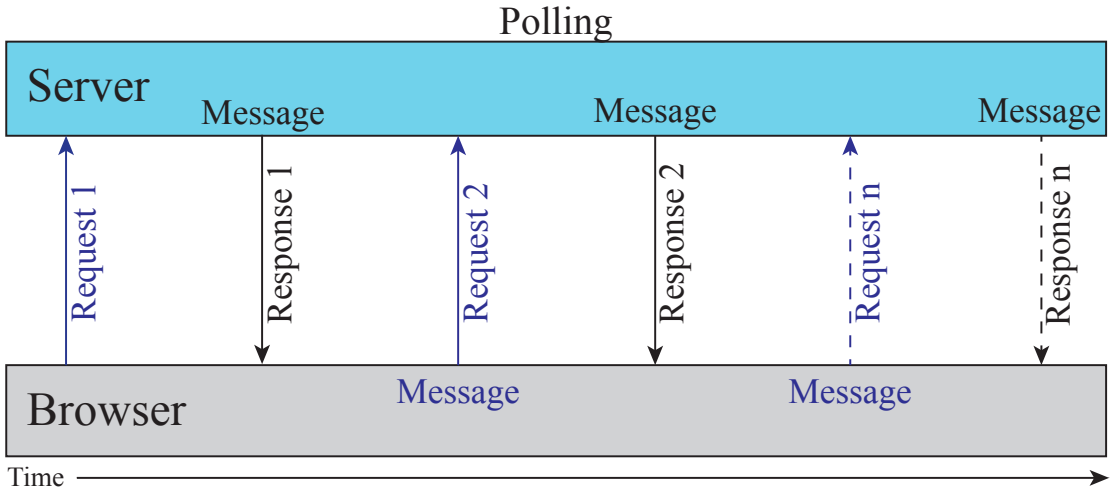

Abbildung 3.4: Datenaustausch mittels HTTP Request / Response beim Polling [WSM12, S. 7]

Vor diesem technischen Hintergrund wurde *Polling* entwickelt, bei dem in einem zeitlich bekannten Intervall eine Anfrage an den Server geschickt wird mit der Bitte um Aktualisierung. Diese Technik ist sehr attraktiv, wenn die zeitlichen Abstände der Aktualisierung der Daten bekannt ist, allerdings sind Echtzeitdaten schlecht vorhersehbar. Dadurch ist Polling nicht die richtige Wahl für dieses Projekt, weil es auf eine wirkliche Echtzeitaktualisierung ankommt.

Da bei sämtlichen Techniken, die dem echtzeitähnlichen Austausch von Daten dienen, höhere Latenzen, mehr Rechenleistung und ein komplizierterer Aufbau zu erwarten sind, ist HTML5 mit WebSockets auf dem aktuellen Stand der Dinge und kann vieles besser machen als seine Vorgänger, weshalb für diese Arbeit auch auf diese junge Technik zurückgegriffen wird. Pollingähnliche Verbindungen werden bei dieser Webanwendung nur angewendet, wenn HTML5 oder die WebSockets von dem verwendeten Browser nicht

unterstützt werden. Tritt so ein Fallback ein, so ist zwar immer noch die Kommunikation mit dem WebSocket Server möglich, aber es sind höhere Latenzen und erhöhter Traffic zu erwarten.

Kapitel 4

Erweiterte Funktionalitäten

In diesem Kapitel werden nun weitere Funktionen beschrieben, die über die Grundausstattung der Web-App und die WebSockets hinausgehen, um die Arbeit zu erweitern und produktiver zu gestalten.

4.1 Lokalisierung von Clients

Mit HTML5 wurde die *navigator.geolocation* Klasse in die JavaScript API eingebaut. Diese Klasse ermöglicht es einer Webanwendung die Position des Anwenders zu erhalten sofern dieser es erlaubt. Das geschieht bei aktuellen Endgeräten unter Anderem mit in der Umgebung befindlichen WLAN Netzwerken, deren Standorte in einer Datenbank gespeichert sind. Verfügt das Gerät über GPS, so werden diese Daten auch berücksichtigt und ermöglichen eine genauere Ortung [9el13, 1. Absatz].

Koordinieren von Helfern Um die eingetragenen Helfer in dieser Webanwendung besser koordinieren zu können, wurde daher ein Skript implementiert, welches im Hintergrund der Web-App läuft und die aktuelle Position des Endgeräts über eine SSL verschlüsselte Verbindung an den Server übermittelt. Unter der Voraussetzung, dass der Client diesem Vorgang zustimmt sind dem Server damit die Positionen der eingeloggten

Benutzer bekannt. Diese Positionsdaten können dann von der Anwendung ausgewertet und in einer Karte von *Google Maps* [Goo13b] visualisiert werden.

Abbildung 4.1: Getestet unter iOS 6.1.4, iPhone 5, Safari: Grün markiert eigene Position. Durch Tippen auf andere Marker erscheint der Name des Benutzers

Jeder Client kann auf diese Weise die Positionen der anderen Helfer sehen. Der Vorteil liegt klar auf der Hand: Eine zentral eingerichtete Verwaltung kann mit einem Blick sehen, wer sich an welcher Stelle auf dem Gelände befindet. So können Wege optimiert und gezielt Aufgaben an sogenannte Springer verteilt werden, da ortsnahe Helfer die entsprechenden Aufgaben übernehmen können. Die kurzen Wege sorgen dann dafür, dass eine höhere Nutzung der Ressourcen (hier: die Helfer) möglich ist.

Anforderungen Eine wichtige Anforderung an diese Funktion ist, dass die Daten schnell ausgetauscht werden. Wenn zwischen den Updates der Geolocations zu viel Zeit vergeht, ist die Position nicht mehr aktuell und damit nicht mehr relevant. Deswegen findet der Austausch von Längen- und Breitengrad über WebSockets statt.
Damit ein Client die Positionen der anderen Teilnehmer erhält, muss er seine eigene Position an den WS Server schicken oder auf eine Aktualisierung der anderen Benutzer warten. Mit jedem *locations*-Paket, welches an den Server geschickt wird, werden

die aktualisierten Positionen an alle verbundenen Geräte verschickt. Das bedeutet auch, dass ein Client ohne WebSockets-Unterstützung keine Informationen über die Standorte erhält.

Kein Speichern der Positionsdaten Beendet ein Client die Verbindung oder erhält der Server keine Aktualisierungen mehr, so wird sein letzter Standort noch 15 Minuten gespeichert und dann auf allen Endgeräten gelöscht. Zu keiner Zeit werden Daten protokolliert oder länger als nötig aufbewahrt.

Automatische Berechnung der Ansicht der Karte Die Zoomstufe und Zentrierung der Google Maps Karte wird automatisch berechnet, wenn der Benutzer das möchte. Dabei werden alle Koordinaten in eine Variable geschrieben und der Mittelwert gebildet, um so einen Punkt für die Zentrierung zu finden. Über die Methode *fitbounds(LatLngBounds)*, die von der Google API bereitgestellt wird, wird die optimale Zoomstufe berechnet. So wird die Karte immer so positioniert, dass sie alle Marker angepasst anzeigt.

4.1.1 Umgebungsbedingte Einschränkungen

Bei diesem Projekt handelt es sich um eine Webanwendung, die aufgrund von Sicherheitsvorkehrungen einigen Einschränkungen unterliegt. So erlauben die mobilen Betriebssysteme keine Geopositionierung durch den Browser, wenn dieser minimiert ist oder das Smartphone / Tablet nicht aktiv genutzt wird. Das bedeutet, dass die Meißner App geöffnet sein muss, damit alle Funktionen genutzt und aktualisiert werden können.

Das ist für die Lokalisierung problematisch, da so die Position des Clients nicht aktuell bleibt. Diese Sicherheitsmaßnahmen sind jedoch sehr sinnvoll, da es sehr bedenklich wäre, wenn die Position des Besuchers im Hintergrund überwacht werden könnte. Zudem würde bei einem dauerhaften GPS Signal ein sehr hoher Stromverbrauch anfallen. Allerdings genügt es, wenn die Webanwendung in einem Tab geöffnet ist, welcher nicht aktiv sein muss, um die Position des Endgeräts zu erfahren.

4.2 Offline-Funktionalität

Mit der HTML5-Spezifikation wurde die *Application Cache API* eingeführt, welche es ermöglicht, bestimmte Dateien einer Webanwendung in einen lokalen Cache zu speichern [Fri13, S. 189f]. Dafür ist eine *Manifest*-Datei notwendig, die eine Liste mit den zu cachenden Dateien erstellt. Findet ein Browser ein Manifest, so lädt er initial die Webseite und speichert dabei alle Dateien in seinem Cache. Dieser ist bei normalen Anwendungen auf 5 MB beschränkt, welche unter normalen Bedingungen vollkommen ausreichen.

Dynamische Erstellung Die Meißner App generiert dynamisch mit jedem Aufruf der Anwendung ein Manifest, in der alle relevanten JavaScript Files, Bilder oder PHP-Seiten aufgeführt sind [Gia10]. Das Manifest geht dabei in jedem View davon aus im gleichen Verzeichnis zu sein, weshalb es stets das gleiche Manifest generiert, sofern sich keine Datei verändert hat. Jedes Verzeichnis, welches nicht ausgeschlossen wurde, wird dabei kurz besucht und die entsprechenden Dateien aufgelistet.
Dadurch kann die App erweitert werden, ohne die Offlinefunktion überarbeiten zu müssen.

Update des Caches Wurde die Anwendung einmal geladen und im Cache gespeichert, so zeigt sie immer diesen Stand der Seite an. Um ein Neuladen der Seite im Browser hervorzurufen, muss eine Änderung am Manifest vorliegen. Das geschieht über eine Versionsnummer innerhalb dieser Datei: werden Änderungen an der Datenbank vorgenommen, dann inkrementiert die Webanwendung diese Nummer. Erreicht sie die Zahl 1000, so beginnt die Versionsnummer wieder bei 0. So wird ein möglicher Überlauf verhindert und eine potentielle Sicherheitslücke nicht ermöglicht.
Beim Update des Caches durchläuft der Browser alle zu cachenden Dateien und prüft diese auf Änderungen. Gibt es keine, so wird der letzte Stand aus dem Cache beibehalten. Das gilt auch für einzelne Dateien, weshalb bei einer Modifikation einer Datei nicht der gesamte Cache neu geladen werden muss. Wurden allerdings Dateien verändert, so werden diese nachgeladen und die Seite anschließend bei erfolgreichem Update über ein JavaScript neu geladen, wonach der aktuelle Stand der Seite zu sehen ist.

Das geschieht in der Regel so schnell, dass dieser Vorgang ungeachtet vom Benutzer geschieht. Den Prozess des Updates kann man in der JavaScript Konsole des Browsers nachverfolgen.

Mögliche Konflikte Sollten bei aktiver Benutzung der App zwei Helfer gleichzeitig einen Eintrag ändern und die Webanwendung dadurch anregen die Versionsnummer zu erhöhen, so wäre es möglich, dass beide zweimal die gleiche Zahl inkrementieren und von der Änderung des jeweils anderen nichts erfahren. Das ist allerdings nicht relevant, da die Versionsnummer nur angibt, *dass* sich etwas geändert hat und nicht *was* sich geändert hat. Letztlich soll damit nur bewirkt werden, dass die Endgeräte ihren Cache aktualisieren. Durch welchen Benutzer das Update ausgelöst wurde oder wie viele Änderungen es gab, ist von keiner Bedeutung.

Dadurch ist die Anwendung nun großteils offline verfügbar. Funktionen wie die Geolocations oder der Chat, können allerdings nicht einwandfrei genutzt werden ohne Internetanbindung. Das liegt zum einen daran, dass Google das Cachen von Google Maps im Allgemeinen für Webanwendungen verbietet [Goo13c] und daher die Offlineverwendung der Karte nicht möglich ist. Zum anderen ist für den Chat und Publish / Subscribe eine WebSocket-Verbindung notwendig, da sonst keine Nachrichten ausgetauscht werden können.
Mit gewissen Einschränkungen kann die Meißner App auch dort verwendet werden, wo nur eine schlechte Verbindung zur Verfügung steht. In jedem Fall lädt die App aber nun schneller, da sie nicht mehr alle Daten vom Webserver beziehen muss.

4.3 Installation auf einem Debian-basiertem System

Um diese Webanwendung mit sämtlichen Funktionen benutzen zu können, wurde eine einfache Installationsdatei in Form eines Shell-Scripts entwickelt. Dieses Skript installiert und lädt als Webserver Apache2, MySQL und PHP5 herunter, danach wird die aktuellste Version von node.js heruntergeladen und kompiliert. Zuletzt wird die Meißner App in das Verzeichnis */var/www/meissner* verschoben und die entsprechenden Rechte gesetzt, damit Apache2 darauf zugreifen kann.

Diese Installation wurde für ein unverändertes Debian 7 entwickelt und alle Abhängigkeiten, die für die Installation eines Webservers und WebSocket Servers notwendig sind, werden beachtet. Unter anderen debianbasierten Distributionen, wie Linux Mint, Ubuntu, etc., wurde das Skript auch erfolgreich getestet. Notwendig ist lediglich eine funktionierende Internetverbindung, da sich das Betriebssystem vorher aktualisiert und die aktuellste Version von node.js heruntergeladen wird.

Abschließend erfolgt die Einrichtung der MySQL Datenbank über das Webinterface und kann unter *http://localhost/meissner/setup* erreicht werden. Die Zugangsdaten zur gewünschten Datenbank müssen dort eingegeben werden und daraus fertigt die Anwendung eine Konfigurationsdatei an, die sämtliche Informationen zur Verbindung mit der Datenbank enthält.

Alle Änderungen an der *apache2.conf* und *php.ini* werden automatisch eingefügt, sollten sie nicht schon standardmäßig installiert sein. Dabei handelt es sich in erster Linie um das Modul *rewrite*, welches in den genannten Dateien geladen werden muss. Es ermöglicht die Verwendung von kurzen URIs, die in dieser Anwendung genutzt werden. Das heißt der Benutzer kann in der Adressleiste des Browsers Links eingeben wie *http://localhost/meissner/events* und wird sofort in die richtige Sektion weitergeleitet.

ReadMe Eine Installationsanleitung liegt separat zu der Meißner App bei. Da während der Installation der notwendigen Pakete einige Benutzereingaben notwendig sind, werden diese in Form von Bildschirmfotos erklärt. So ist eine verständliche Einrichtung der Anwendung möglich.

4.4 Paketierung für App-Stores

Auch bei Web-Apps besteht die Möglichkeit diese in den gängigen App-Stores wie dem von Apple oder Googles Play Store anzubieten. Dafür gibt es einige Frameworks, welche die eigentliche Webanwendung in eine Browserumgebung verpacken und diese dann wie eine native App aussehen lassen.

Für diese Arbeit wurde bewusst nicht darauf eingegangen und auch keine Paketierung für die Stores eingeplant. Das gesamte Projekt kann als Webanwendung binnen kurzer Zeit aktualisiert werden und ohne Grenzen der Stores für alle Geräte zur freien Verfügung stehen. Der Aufruf einer Webseite stellt für diesen Zweck die einfachste Möglichkeit dar, diese Anwendung plattformübergreifend nutzen zu können.

Vorteile So können Änderungen an der Meißner App vorgenommen werden ohne, dass man kompliziert über die Stores die App neu verteilen muss. Außerdem ist absolut keine Installation notwendig, da sie wie eine normale Homepage geladen wird. Der Benutzer ist zwar nicht zwangsläufig gewohnt Apps über einen anderen Weg als den bekannten Store zu beziehen, aber da die Web-App genau so einfach geöffnet wird wie der Aufruf einer Homepage, kann man davon absehen.

4.4.1 Add to Homescreen

Bei Geräten mit dem Betriebssystem iOS ist die Funktion *Add to Homescreen* verfügbar, welche eine Verknüpfung zur Webanwendung auf dem Homescreen erstellt. So kann die Anwendung wie eine normale App gestartet werden. Durch eine einfache Ergänzung im Header der mobilen HTML Seite wird so eine appähnliche Installation ermöglicht.

```
<meta name="apple-mobile-web-app-capable" content="yes">
<link rel="apple-touch-icon" href="img/icon.png">
```

Listing 4.1: Ergänzung im Header der mobilen Seite

Mit diesen zwei Zeilen wird die Option *Add to Homescreen* erlaubt und der Pfad zu einem Icon angegeben, welches dann auf dem Homescreen erscheinen soll.

4.5 Chatfunktion

Zur Kommunikation der Clients untereinander wurde der Bereich *Chats* hinzugefügt. Der Datenaustausch findet über WebSockets statt und es ist keine weitere Konfiguration notwendig. Es muss nur der entsprechende View geöffnet werden, die Webanwendung erfragt die Historie beim WS Server und zeigt sie an. Ein einfaches Eingabeformular ermöglicht dann die Kommunikation mit allen eingeloggten Benutzern.

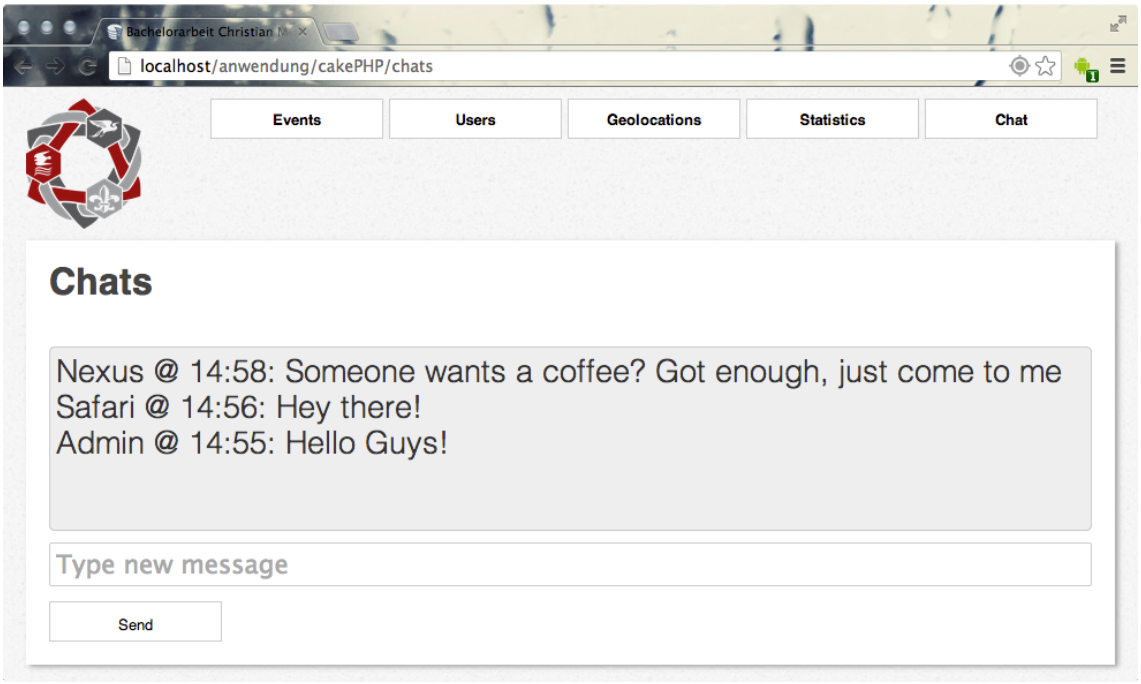

Abbildung 4.2: Getestet unter OS X 10.8.5., Chrome 29: Chatverlauf von drei verschiedenen Benutzern

Hierbei wird wieder Publish / Subscribe verwendet: Befindet sich ein Endgerät gerade im View des Chats, so abonniert die Webanwendung automatisch den Chat beim WebSocket Server und erhält dadurch alle eingehenden Nachrichten.

4.6 PUSH-Benachrichtigungen

Bei Webanwendungen gibt es noch weitere Einschränkungen. So konnte für diese Arbeit nicht auf die systemeigenen Benachrichtigungsmechanismen zurückgegriffen werden, wie man sie aus nativen Applikationen her kennt. In den aktuellen Versionen aller mobilen Betriebssystemen sind Bereiche für Benachrichtigungen aus den jeweiligen Anwendungen implementiert um dem Benutzer zu signalisieren, dass neue Nachrichten vorliegen. Allerdings darf eine Web-App darauf nicht zugreifen, daher wurden für diese Anwendung eigene Methoden auf Basis von *noty* [Ara13], einem jQuery Plugin für Benachrichtigungen, implementiert. Mit dieser Bibliothek wurde eine Benachrichtigungsleiste entwickelt, die am unteren Bildschirmrand Informationen anzeigen kann, wie zum Beispiel die oben angesprochene publish-Benachrichtigung vom WebSocket Server.

Auf diese Weise können nun auf allen Plattformen, die JavaScript aktiviert haben, Push-Benachrichtigungen angezeigt werden, wenn relevante Informationen über die WebSockets an das Endgerät gelangen.

4.7 Statistiken

Da eine Auswertung der eingegebenen Daten für Veranstaltungen unabdinglich ist, wurde ein weiterer Controller implementiert, welcher sämtliche speziellen Felder der Events aus der Datenbank abfragt und diesen dann die Werte der einzelnen Benutzer zuweist.

Im View wird dann eine grafische Auswertung gestartet, die mit Hilfe von *Google Charts* [Goo13a] zu den Feldern ansehnliche Graphen generiert, bei denen vergleichbare Werte von den Benutzern hinterlegt wurden. Außerdem gibt es allgemeine Statistiken, die die Veranstaltungen untereinander vergleichen und man so einen schnellen Überblick über die angelegten Events erhält.

Kapitel 5

Analyse und Auswertung

Um zu analysieren, wie sich die Webanwendung unter Last verhält, werden verschiedene Aspekte, wie die Netzwerkauslastung und der notwendige Arbeitsspeicher, betrachtet.

5.1 Geolocations: zeitliches vs. konditionelles Update

Bei der Aktualisierung der eigenen Position gibt es zwei Möglichkeiten diese abzufragen:

1. Zeitliches Intervall definieren, in dem die Position aktualisiert wird
2. Überwachung des Standortes über HTML5-eigene Module

Intervall Zuerst wurde ein Intervall von 10 Sekunden benutzt, um die Positionsinformationen zu erfragen, verarbeiten und an den Server zu schicken. Das ist eine einfache Lösung, die zum gewünschten Ergebnis führt, allerdings nicht wirklich in Echtzeit abläuft.

watchPosition() Die zweite Methode ist der Abruf einer Funktion, die von der navigator-Klasse bereitgestellt wird. Mit *watchPosition()* wird die Position des Gerätes überwacht und bei Veränderung eine *successCallback*-Funktion aufgerufen [Pop12]. Diese Callback-Funktion verarbeitet die aktuelle Position in einem JSON-Objekt, wandelt es in einen String um und schickt anschließend das Update an den WS Server.
Da watchPosition() nur auf Änderung des Standortes reagiert, können Zugriffe auf das GPS Modul des Endgeräts und damit auch der Stromverbrauch verringert werden. Daher ist diese Methode vorzuziehen und wird in der Meißner App verwendet.

5.2 Webserver mit Apache2

Hier werden die Zugriffszeiten der Anwendung gemessen. Dabei werden eine leere cakePHP Anwendung, die Loginseite der Meißner App sowie 3 Veranstaltungen mit unterschiedlichen Anzahlen von Einträgen miteinander verglichen.
Die Loginseite wird verwendet, da sie eine Seite mit minimalen Zugriffen auf die Datenbank darstellt und sich somit besser mit der Startseite einer leeren cakePHP Anwendung vergleichen lässt.
Es wurden drei Veranstaltungen für diesen Benchmark erstellt. Diese haben 0, 10 und 100 verschiedene eventspezifische Felder, in die die Daten der Benutzer eingetragen werden können. So entstehen Vergleichswerte, die verschiedenen Anzahlen von Zugriffen auf die Datenbank simulieren.

5.2.1 Testsystem

Als Testsystem wurde für den Server ein MacBook Pro (Frühjahr 2011) mit Mac OS X 10.8.5, Intel Core i5-2415M CPU @ 2.30GHz und 8 GB DDR3 1333 MHz Arbeitsspeicher verwendet. Auf diesem wurden Apache 2.4.4, PHP 5.5.3 und MySQL 5.6.12 installiert.
Für den Benchmark wurde ein weiterer Computer verwendet, damit der Server nicht durch den Test auf sich selbst zu viele Ressourcen verbraucht. Dafür kam ein PC mit

Linux Mint 15, Intel Core i5-2500K CPU @ 4,5 GHz und 8GB DDR3 1333MHz Arbeitsspeicher zum Einsatz. Auf diesem Computer wurde *Apache Bench 2.3* mit folgenden Parametern aufgerufen:

-k Verwendet eine persistente Verbindung

-c 10 Erzeugt 10 Verbindungen gleichzeitig

-n 1000 Insgesamt werden 1000 Verbindungen zum Server aufgebaut

Daraus ergibt sich der Befehl: *ab -k -c 10 -n 1000 http://localhost/meissner/events/edit/1/*, um beispielsweise das Event mit der ID 1 zu testen.

Der Server und der PC wurden über eine Gigabit-Leitung miteinander verbunden, um eine größtmögliche Geschwindigkeit zu erreichen.

Erwartungen Zu erwarten ist hier, dass der Webserver mit dem unmodifizierte Framework die meisten Anfragen pro Sekunde verarbeiten kann. Da für die weiteren Tests immer komplexer werdende Views aufgerufen werden, sollten diese langsamer geladen werden als einfache Views.

5.2.2 Ausführung des Benchmarks

Bei diesem Test interessiert vor allem, wie viele Anfragen der Webserver pro Sekunde verarbeiten kann. Um eine Vorstellung davon zu bekommen, was ein normaler Wert für eine unmodifizierte cakePHP Anwendung ist, wurde eine Testseite mit cakePHP 2.4.1 eingerichtet, welche als Richtwert dient.
Um möglichst repräsentative Ergebnisse zu erhalten, wurde ein Skript entwickelt, welches den Arbeitsspeicher leert und dann Apache Bench startet. Danach wird fünf Sekunden gewartet und der nächste Test gestartet. Der gesamte Vorgang wird fünfmal wiederholt und dann die Ergebnisse ausgewertet. Dabei wurden die bearbeiteten Anfragen pro Sekunde grafisch dargestellt:

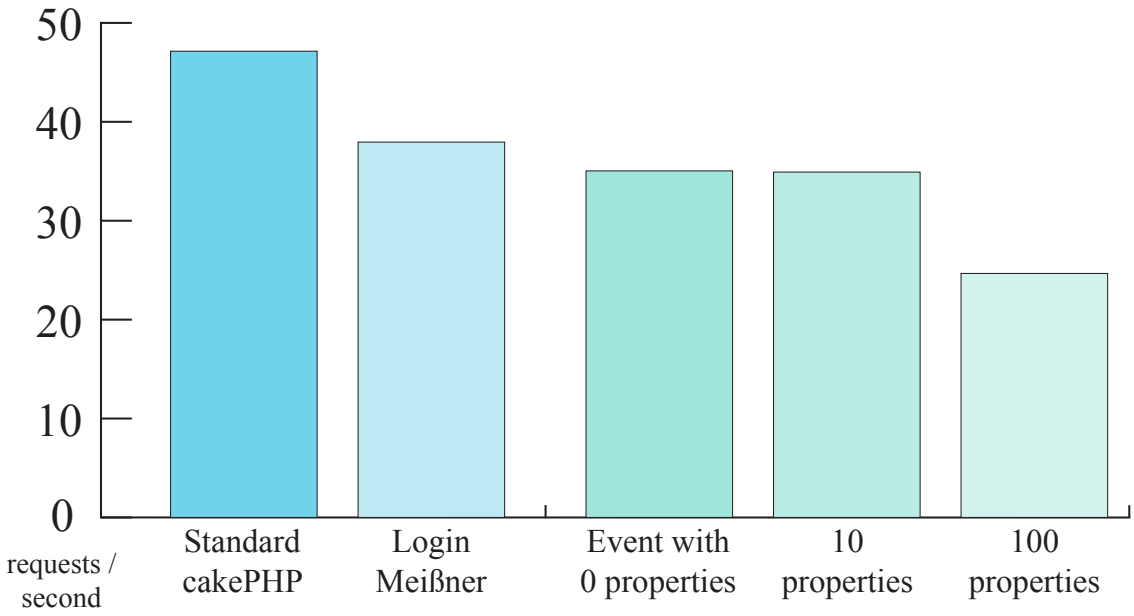

Abbildung 5.1: Benchmark des Webservers mit Apache Bench

Die unmodifizierte cakePHP Webanwendung erreicht den Höchstwert 47,14 Anfragen pro Sekunde. Dass hier der höchste Wert vorliegt, war zu erwarten, da das gleiche Framework für die Meißner Anwendung verwendet wurde, allerdings müssen keine Bilder, Skripte, etc. geladen werden.

Das Laden der oben genannten Elemente erklärt auch das schlechtere Ergebnis der Meißner App bei der „Login Meißner“-Säule. Im Hintergrund werden hier außerdem Skripte von socket.io angefragt, um eine Verbindung zum WebSocket Server aufzubauen. Dadurch ist die App im Allgemeinen komplexer aufgebaut und erreicht so nur einen Wert von 37,95 Anfragen pro Sekunde. Dieses Ergebnis ist für das flüssige Bedienen der Anwendung vollkommen ausreichend und wird in der Praxis durch den Offline Cache weiter beschleunigt, da nicht mehr so viele Requests an den Server gestellt werden müssen.

In grün gehalten wird deutlich, wie ein Event mit mehr Details deutlich länger zum Laden benötigt. Dabei sind für die Übersichtsseite zur Bearbeitung einer Veranstaltung widerrum mehr Anfragen notwendig als für die Startseite der Meißner App und erreicht somit einen Wert von 35,04 bearbeiteten Anfragen pro Sekunde.

Wenige Eigenschaften sorgen für keinen signifikant schlechteren Wert. So erreicht eine Veranstaltung mit nur 10 Eigenschaften mit 34,92 Anfragen pro Sekunden nahezu den gleichen Wert wie ein leeres Event. Nur bei einer hohen Anzahl wird der Aufruf der Seite komplexer und verlangsamt sich auf 24,67 Anfragen pro Sekunde.

5.2.3 Fazit

Mit erhöhter Komplexität eines Views verlängert sich die Zeit, die der Webserver zum bereitstellen benötigt. Das trifft die Erwartungen und ist auch wenig verwunderlich, da mehr Daten aus der Datenbank erfragt und aufbereitet werden müssen. Alle Ergebnisse sind allerdings gut für einen Server, der auf einem Notebook läuft.

Der Server erfährt in der Praxis aber eine geringere Belastung, da der Offline-Cache viele Requests unnötig macht. Denn nur veränderte Dateien werden neu vom Server angefragt, abgesehen von den dynamischen Bibliotheken für socket.io oder Google Maps.

5.3 WebSockets mit node.js

Bei den WebSockets wird analysiert, wie groß die Pakete, die verschickt werden, tatsächlich sind und wie viel Arbeitsspeicher dafür von node benötigt wird.

5.3.1 Speicherauslastung

Zunächst wird versucht zu erfassen, wie viel Arbeitsspeicher pro geöffnetem WebSocket benutzt wird. Das ist interessant, weil es eine messbare Größe ist, die angibt, wie viele WebSockets bei festem Arbeitsspeicher verwaltet werden können.

Als Testsystem kommt das gleiche MacBook Pro wie beim Apache2 zum Einsatz, da sowohl Apache als auch node.js auf dem gleichen Server installiert sind.

Für den Versuch wird mit dem Tool *Activity Monitor* (integriert in OS X) die Speicherauslastung von node gemessen und so die durchschnittlichen Anforderungen für jede aktive Verbindung ermittelt.

5.3.2 Messung der Speicherauslastung

Es wurden drei Messreihen durchgeführt in denen über den Browser Google Chrome je 50 Sockets geöffnet wurden. Dabei stieg die Speicherauslastung im Schnitt von 75,50 MB auf 78,63 MB an. Die CPU Auslastung betrug dabei ca. 2-3%. Demnach benötigt eine aktive Verbindung zu node.js ca. 63 KBytes.

Somit berechnet sich die zusätzliche Speicherauslastung für 1000 Clients wie folgt:

$$1.000 \times 63 \text{ KBytes} = 63.000 \text{ KBytes} \sim 61,5 \text{ MB}$$

Die Messung gestaltete sich relativ schwierig, da die Garbage Collection von node.js ab und zu Speicher freigab, was einige Messreihen unbrauchbar machte. Es finden also noch weitere eigene Optimierungsvorgänge statt, um den Speicherverbrauch möglichst gering zu halten. Diese werden automatisch von node.js ausgeführt.

Fazit Die Tests haben gezeigt, dass die Speicherauslastung durch WebSockets geringer ist, als erwartet wurde. Dadurch können sehr viele Clients bei geringem Speicheranspruch verwaltet werden.

5.3.3 Netzwerkauslastung

In dem Kapitel über die WebSockets wurde beschrieben, dass die Netzwerkauslastung bei eben diesen besonders gering sein soll. Das liegt daran, dass der Header eines WebSockets mit gerade 2 Bytes pro Nachricht auskommt, während bei normalen HTTP Anfragen ein Header von ca. 700-800 Bytes benötigt wird [Ver13].

Beim vorgestelltem *Polling* sind diese 700-800 Bytes beim HTTP-Request und HTTP-Response notwendig. Das würde bedeuten, dass bei 1.000 Clients nur für den Header ohne weiteren Inhalt und bei einem Intervall von einer Sekunde folgende Rechnung aufgestellt werden kann:

$$1.000 \times 800 \frac{\text{Bytes}}{\text{Sekunde}} = 800.000 \frac{\text{Bytes}}{\text{Sekunde}} = 6.400.000 \frac{\text{Bits}}{\text{Sekunde}} \sim 6,104 \text{ Mbps}$$

Im Vergleich dazu macht sich der deutlich schmalere Header der WebSockets bemerkbar. Auch hier wird zur Vereinfachung von einer sekündlichen Übermittlung einer Nachricht mit 1.000 Clients ausgegangen:

$$1.000 \times 2 \frac{\text{Bytes}}{\text{Sekunde}} = 2.000 \frac{\text{Bytes}}{\text{Sekunde}} = 16.000 \frac{\text{Bits}}{\text{Sekunde}} \sim 0,015 \text{ Mbps}$$

Das bedeutet, dass über 400% Overhead durch WebSockets eingespart werden kann. Da die Meißner App vor allem mobil genutzt werden soll, ist dieses Ersparnis von großer Bedeutung: Umso weniger Daten für die Echtzeitaktualisierung benötigt werden, desto schneller und zuverlässiger können publish-Nachrichten die Endgeräte erreichen.

Komplexität Mit jeder Nachricht, die an den WebSocket Server geschickt wird, gibt es eine Antwort, die fast immer eine Broadcast-Nachricht darstellt. Das hat zur Folge, dass auf eine einzige Nachricht n Nachrichten an verbundene Clients folgen. Wenn nun alle n Clients eine Nachricht verschicken, werden $n \times n = n^2$ Antworten vom WS Server versendet. Dadurch ergibt sich eine quadratische Komplexität für diese Implementierung.

Für die Anwendung stellt dies ein Problem dar, wenn hinreichend viele Endgeräte gleichzeitig Nachrichten verschicken. Das Verschicken einer neuen Geoposition findet mit jeder Veränderung der Position statt. Auf diese Weise können schnell sehr viele Updates zusammenkommen. Die quadratische Anzahl von Antworten könnte an dieser Stelle zu einer Überlastung des Servers führen.

Optimierung: Puffermanagement Mit einem intelligenten Puffermanagement könnte das verhindert werden. Es müsste Nachrichten erst puffern und dann den Inhalt nach Aktualität untersuchen. Wenn zum Beispiel in einem Puffer zwei Nachrichten mit neuen Positionen eines Clients sind, so könnte die Nachricht, die zuerst an den WS Server geschickt wurde, verworfen werden, da schon ein weiteres Update auf diese Position folgt.

Optimierung: Intervalle einrichten Bei zu hoher Belastung könnten gewisse Intervalle eingerichtet werden, bei denen die Endgeräte die Updates vom Server erhalten. Die Echtzeitaktualisierung wäre etwas eingeschränkt, aber die Belastung des Netzwerkes und des Servers könnten hierdurch deutlich verringert werden.

Skalierung Die Anwendung wurde in erster Linie für circa 90 Clients entwickelt. Bei dieser geringen Anzahl werden die Server nicht überfordert sein. Bis zu 50 Clients gleichzeitig wurden erfolgreich getestet und haben node.js nicht besonders gefordert. Durch die quadratische Komplexität ist dies allerdings nicht für beliebige Anzahlen von Endgeräten garantiert. Das kann durch das oben angesprochene intelligente Puffermanagement gelöst werden, wodurch die Anzahl der Nachrichten reduziert wird und somit auch deutlich mehr Clients gleichzeitig Updates oder Chat-Nachrichten verschicken können.

Kapitel 6

Verwandte Arbeiten

6.1 Verschiedene WordPress Plugins

EventsEspresso Es handelt sich um ein WordPress Plugin, welches ein einfaches Veranstaltungsmanagement ermöglicht [Eve13]. Hierbei können in der Grundausstattung auch verschiedene Veranstaltungen, Teilnehmer und Helfer erstellt und verwaltet werden. Weitere Features wie die Verwaltung von Zahlungen, Erstellung von Eintrittskarten und verschiedene Sprachen gehören zum Basispaket.
An dieser Stelle endet auch schon die kostenfreie Version dieses Plugins und der Hersteller verlangt für weitere Premium Features hohe Preise: Für $89.95 wird lediglich eine JSON API, eine Anbindung an soziale Netzwerke und ein Kalender integriert. Um den vollen Funktionsumfang zu haben und selbst Änderungen am Plugin vornehmen zu können, sind $499.95 notwendig.

Features wie Echtzeitaktualisierung, Chats oder die Optimierung für den Außenbereich wie in der Meißner App gibt es nicht. Der Funktionsumfang der kostenlosen Version ist sehr spärlich und lässt sich nicht ohne Weiteres erweitern. Außerdem ist man an WordPress gebunden, da es nicht eigenständig lauffähig ist und eine angepasste Version für mobile Endgeräte gibt es auch nicht.

EventsManager Diese Software bewirbt sich selbst als das beliebteste WordPress Plugin für dieses Anwendungsgebiet [Net13]. Der Funktionsumfang ist ähnlich wie bei der oben beschriebenen Software. Möchte man auch hier weitere Addons nutzen, so müssen $75 bezahlt werden.

Eine Unterstützung für großflächige Veranstaltungen im Außenbereich ist auch hier nicht vorgesehen, das Google Maps Modul dient lediglich zur Lokalisierung des eigenen Events und nicht zur Ortung der Helfer. Auch ist hier keine mobile Seite für Smartphone oder Tablets vorgesehen, es wird lediglich die gleiche Seite angezeigt wie an einem Desktop und ist nicht touchoptimiert. Bei einem Testaufruf dieses Plugins mit einem 7-Zoll Tablet, wurde die schlechte Anpassung an mobile Geräts dadurch deutlich, dass manche Container über anderen liegen und damit deren Inhalte verdecken.

6.2 Übersicht

Bei der Recherche konnte keine Anwendung gefunden werden, welche als Plugin oder eigenständiges System die Organisation von Veranstaltungen unterstützt und dabei noch moderne Featues wie die Echtzeitaktualisierung, Publish / Subscribe oder Lokalisierung von Endgeräten bietet. Die Meißner App ist zudem dafür ausgelegt über eine große Fläche (auch unter freiem Himmel) und mit touchfähigen Geräten zu funktionieren, was auch bei keiner änlichen Arbeit gefunden werden konnte.

Kapitel 7

Zusammenfassung

Mit dieser Bachelorarbeit ist die *Meißner App* entstanden, welche das Erstellen von beliebig vielen Veranstaltungen und Benutzern ermöglicht. Sie bietet die Möglichkeit eigene Felder und Formulare für spezifische Events zu definieren und diese frei mit Werten der Benutzer zu füllen. Diese Werte stammen hauptsächlich aus den Anmeldungen der Teilnehmer.

Die Anpassung an touchfähige Geräte ermöglicht eine einfache Bedienung mit Smartphone und Tablets, wobei hauptsächlich mobile Endgeräte in Verwendung sein werden. Hierbei wurde auf die Unterstützung von Android 2.3 - 4.3 und iOS 5 - 6 sowie die Browser Safari 6, Firefox 24 und Google Chrome 29 besonderen Wert gelegt.
Durch die einfache Bedienung können ungeschulte Personen die Webanwendung benutzen, verwalten und anpassen.

Moderne Webtechnologien aus dem Web 2.0 kommen hier zum Einsatz. Davon sind die bedeutensten das experimentelle HTML5 und die damit definierten WebSockets.

Besonderheiten Für den produktiveren Einsatz der Helfer wurde eine Echtzeitaktualisierung mit Hilfe von HTML5 und WebSockets implementiert. Dadurch sind Erweiterungen wie die Geolokalisierung von Benutzern, PUSH-Benachrichtigungen innerhalb der Webanwendung und Chats für interessierte Benutzer möglich.
Optimiert ist die Anwendung für die Verwendung unter freiem Himmel, denn nur dort

kann die Lokalisierung mit Smartphones genau erfolgen. Eine automatische statistische Auswertung der einzelnen Veranstaltungen ermöglichen eine grafisch ansprechende und verständliche Aufbereitung der eingegebenen Daten.

Mit diesen sinnvollen Erweiterungen erreicht die Anwendung einen Funktionsumfang, der bei keiner ähnlichen Anwendung gefunden werden konnte. Professionelle Entwickler verlangen viel Geld für ihre Produkte, während hier sowohl notwendige als auch erweiternde Funktionen für ein produktives Veranstaltungsmanagement-System gratis und Open Source zur Verfügung gestellt werden.

Zuletzt bleibt noch zu erwähnen, dass die Meißner App ein eigenständiges Programm ist, welches auf einem einfachen Webserver installiert werden kann. Sie benötigt kein besonderes Backend, sondern eine Standardumgebung, wie sie für heutige Webseiten üblich ist. Das heißt eine Apache Installation, MySQL Datenbank sowie die Unterstützung von HTML5 und PHP5.
Für die erweiterten Funktionen muss serverseitig node.js installiert sein, aber diese Erweiterung ist optional und nicht notwendig für die Grundfunktionen.

7.1 Ausblick

Die folgenden Erweiterungen könnten noch implementiert werden, um den Funktionsumfang sinnvoll zu erweitern. Aus Zeitgründen konnten diese Funktionen nicht mit in die Bachelorarbeit aufgenommen und implementiert werden:

Geolocations Eine Unterstützung für mehrere Endgeräten, die mit dem gleichen Account eingeloggt sind. Zur Zeit wird immer nur ein Endgerät pro Benutzeraccount auf der Karte angezeigt.

Chat Im Moment ist nur ein Kanal zum chatten verfügbar. Das könnte man ändern und für jede Veranstaltung einen eigenen kleinen Channel erstellen.

Erweiterung von Publish / Subscribe Beliebige Veranstaltungen sollten abonniert werden können, um damit mehr Informationen zu Veranstaltungen zu erhalten, die den Benutzer interessieren.

Kompression der übertragenen Daten Um den Datentransfer möglichst gering zu halten, könnten die für die Webanwendung notwendigen Daten weiter komprimiert werden. Dadurch werden Ladezeiten optimiert und das benötigte Datenvolumen minimiert.

Voice Chat Um direkten Kontakt zu anderen Helfern aufnehmen zu können, könnte man aus der App heraus einen Sprachanruf starten. So muss man nicht umständlich die Telefonnummer desjenigen Helfers suchen, sondern kann per Klick / Tipp auf den Namen einen Anruf starten.

Puffermanagement Ein intelligentes Puffermanagement für den WebSocket Server könnte eine bessere Skalierung der Webanwendung liefern. Aktuell ist die quadratische Komplexität des Nachrichtenaustauschs nicht geeignet für beliebig viele Clients.

Literaturverzeichnis

[9el13] 9ELEMENTS, GmbH: *HTML5 Geolocation abfragen*. http://9elements.com/html5demos/geolocation/. Version: September 2013.

[Ara13] ARABACI, Nedim: *noty v2.1.0 - A jQuery Notification Plugin*. Webseite, August 2013. Open Source Software, online frei verfügbar unter http://needim.github.io/noty/

[CSF13] CAKE SOFTWARE FOUNDATION, Inc.: *CakePHP - the rapid development php framework*. Webseite, August 2013. Open Source Software, online frei verfügbar unter http://cakephp.de/

[Eve13] EVENTESPRESSO: *Event Espresso – WordPress Event Registration and Ticketing Manager Plugin*. http://eventespresso.com/. Version: September 2013.

[Fri13] FRIBERG, P.: *Web-Apps mit jQuery Mobile: Mobile Multiplattform-Entwicklung mit HTML5 und JavaScript*. Dpunkt.Verlag GmbH, 2013 http://books.google.de/books?id=PcwqlQEACAAJ. ISBN 9783864900563

[Gia10] GIACORNELLI, Nial: *Using the HTML5 cache manifest with dynamic files · Nial Giacomelli*. http://nial.me/2010/01/using-the-html5-cache-manifest-with-dynamic-files/. Version: 25. Januar 2010.

[Goo13a] GOOGLE, Inc.: *Google Charts — Google Developers*. `https://developers.google.com/chart/`. Version: September 2013.

[Goo13b] GOOGLE, Inc.: *Google Maps API — Google Developers*. `https://developers.google.com/maps/?hl=de`. Version: September 2013.

[Goo13c] GOOGLE, Inc.: *Google Maps Google Earth APIs Terms of Service - Google Maps API — Google Developers*. `https://developers.google.com/maps/terms#section_10_1_3`. Version: September 2013.

[Goo13d] GOOGLE, project: *HTML5 Rocks - A ressource for open web HTML5 developers*. `http://www.html5rocks.com`. Version: August 2013.

[IFG+11] IETF; FETTE, I.; GOOGLE, Inc.; MELNIKOV, A.; ISODE, Ltd.: *RFC 6455 - The WebSocket Protocol*. `http://tools.ietf.org/html/rfc6455#section-1.3`. Version: Dezember 2011.

[joy13] JOYENT, Inc.: *node.js - platform for fast, scalable network applications*. Webseite, August 2013. Open Source Software, online frei verfügbar unter `http://nodejs.org/`

[Kin63] KINDT, W.: *Grundschriften der deutschen Jugendbewegung*. Eugen Diederich, 1963 (Dokumentation der Jugendbewegung). `http://books.google.de/books?id=uAMcAAAAIAAJ`

[Lud13] LUDOVIC, Barreca: *Elephant.io - A socket.io client in PHP*. Webseite, August 2013. Open Source Software, online frei verfügbar unter `http://elephant.io/`

[Net13] NETWEBLOGIC: *Events Manager Wordpress Event Registration, Calendars, Bookings, Tickets and more!* `http://wp-events-plugin.com/`. Version: September 2013.

[Pop12] POPESCU, Andrei: *Geolocation API Specification*. `http://www.`

w3.org/TR/geolocation-API/#geolocation_interface. Version: Mai 2012.

[Rau13] RAUCH, Guillermo: *Socket.IO - The cross-browser WebSocket for realtime apps*. Webseite, August 2013. Open Source Software, online frei verfügbar unter http://socket.io/

[Tav13] TAVENDO, GmbH: *Publish and Subscribe with AutobahnJS*. http://autobahn.ws/js/tutorials/pubsub. Version: August 2013.

[Ver13] VERSCHIEDENE: *The Chromium Projects - An experimental protocol for a faster web*. http://dev.chromium.org/spdy/spdy-whitepaper. Version: September 2013.

[WSM12] WANG, V.; SALIM, F.; MOSKOVITS, P.: *The Definitive Guide to HTML5 WebSocket*. Apress, 2012 (Apressus Series). http://books.google.de/books?id=f7Gyuv-tZE0C. ISBN 9781430247401

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 20. September 2013 Christian Meter

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Software *Meißner App* unter *anwendung/meissner*
- den zur Auswertung verwendeten Datensatz im Ordner *anwendung/benchmark*
- die Websites der verwendeten Internetquellen im Ordner *arbeit/Quellen*