DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Bridging the gap between millions of people in real time: fusing data of wearables in services

Kordian Bruck

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Bridging the gap between millions of people in real time: fusing data of wearables in services

Zusammenführung von Millionen Menschen in Echtzeit: Verwendung von Wearables Daten in Diensten

Author:	Kordian Bruck
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Nils Kannengießer, M.Sc. & ~~Dr. Ben Zoghi~~ Prof. Dr. Ben Zoghi (modified by NK, April 22nd 2016)
Submission Date:	15. February 2016

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15. February 2016 Kordian Bruck

Acknowledgments

First of all I want to thank Prof. Dr. Ben Zoghi from Texas A&M University in College Station, Texas - his invitation to the United States, collaboration effort and support throughout my stay made this project a success. The teamwork between A&M students and the TUM made this whole journey a really worthwhile and challenging experience.

I want to also thank professor Prof. Dr. Uwe Baumgarten and Nils Kannengießer for providing me with opportunity and contacts in the first place. Their commitment to help me and other students diversify their experience throughout their undergraduate studies is outstanding.

I also want to thank all the amazing students at A&M University and all the friends I made without whom this thesis would not have been possible. Transportation, accommodation and all those social activities you dragged me along to would have never happened without the continuous backing of Willam, Joseph, Ryan and many other Aggies.

Furthermore my family and friends in Europe supported me throughout my stay abroad. Supplying me with a steady stream of energy and good advice helped tremendously in completing this project.

Finally a big thank you to many friends who spend several hours of their free time for proofreading this thesis!

Abstract

A majority of today's global population possesses a smartphone enabling people to connect to billions of other users. Recently, a new trend has emerged as the next step to smartphones: wearable devices. Wearables provide the user with many more possibilities as they can track hand motions, vital signs and provide the user with an added benefit of being easily accessible.

Big sporting events such as the Superbowl or FIFA World Cup pull in people from around the world but they are not interactive in any way. The question is: can we connect these millions of people in real time and make sports something more interactive?

This thesis will explore one possible solution of how to connect those fans who are currently being left out using wearables. The central idea is to provide real time feedback between multiple crowds, detect their emotions and display them via various outlets while improving the experience of the visitors to the stadium. The challenge is to build a bi-directional communication chain using various technologies, which can not only transport information from wearables to the cloud, but also inform the user about relevant updates.

The presented approach will use the most current technologies and advancements to achieve a new user experience which hopefully can be then further developed into a marketable product.

Abstract

Heutzutage hat ein großteil der Bevölkerung ein Smartphone, dass es Ihnen ermöglicht mit Milliarden anderen Benutzern über das Internet zu kommunizieren. In den letzten Jahren ergibt sich ein neuer Trend: tragbare Geräte, besser bekannt als "Wearables". Diese heranwachsende Sparte an Geräten ermöglicht neue Anwendungskonzepte, da es mehr Sensoren gibt, welche etwa die Position der Hand und die Herzfrequenz messen können, sowie Informationen leichter zugänglich machen.

Sporteigeinisse werden mittlerweile auf der ganzen Welt verfolgt. Der Superbowl oder die FIFA Weltmeisterschaft begeistern Menschen auf allen Kontinenten. Sie sind allerdings nicht interaktiv. Die Frage stellt sich, ob wir das Erlebnis und die Stimmung aus dem Stadium vor Ort, auch an andere Standorte in Echtzeit übertragen können.

Diese Bachelorarbeit wird eine von mehreren Möglichkeiten genauer betrachten, wie man Zuschauer mithilfe von wearables untereinander besser vernetzen kann. Die Idee ist es, in Echtzeit Rückmeldungen über das Publikum weiterzugeben. Wir wollen die Emotionen erkennen und diese anschließend über verschiedene Kanäle wiedergeben, während wir den Fans zusätzlichen Komfort bei deren Besuch ermöglichen. Die Herausforderung ist, eine Möglichkeit zu finden, wie man einfach bi-direktional zwischen den verschiedenen Endgeräten kommunizieren kann.

Der gezeigte Ansatz wird die aktuellsten am Markt verfügbaren Technologien verwenden, um das Nutzererlebnis wesentlich zu verbessern. Das Ziel ist es, die Grundlage für ein Produkt zu finden, welches marktorientiert entwickelt werden kann.

Contents

1 Introduction

At the end of 2014 a new era has emerged for the world of web development with the publication of the official World Wide Web Consortium (W3C) recommendation of the Hypertext Markup Language (HTML) 5 standard[12]. Suddenly, a language previously only used for design purposes, could do a whole lot more like local storage, video & audio tags and the `<canvas>` element. Many browser developers at that point already had implemented many of the new features, as the process for finalizing the standard had been going on for over ten years.

Alongside HTML, advances in JavaScript (JS) - standardized under the term ECMAScript[14] - also enabled many new possibilities like web sockets and client side databases. With the release of NodeJS[1] in early 2009, JS switched from being a client-side only programming language to one equally useful as a server-side language. The event driven architecture of JavaScript and also the non-blocking Input Output (I/O) were new territory in server-side programming. Over the last few years larger companies have realised that using these new technologies and SCRUM not only significantly increase their productivity, but at the same time the job satisfaction of their developers. This interest has led to many contributions in open source projects as well as new developments.

Meteor, just like NodeJS, is a new approach to web development trying to revolutionize the way we think about creating websites once more. Meteor is built up on top of NodeJS and makes use of many capabilities that evolved in the past five years in new standardized technologies. These standards are important as the web inherently is a cross platform experience. One can view any website on the Internet on a variety of devices at different resolutions.

There has been a shift in the definition of what an application is. With the initial release of the iPhone, the abbreviation app changed to define an application designed specifically for mobile devices[6] but not a specific platform like Android, or iOS. In the beginning these apps often only had limited functionality, so often people thought of them as being a *light*, or smaller version of the desktop application. Today, websites provide similar capabilities as traditional desktop applications and as such web applications can be real competition. For instance the rise of Software as a Service (SaaS) is the clearest demonstration of this development[19].

[1] `https://nodejs.org/en/`

Meteor claims to be "the JS app platform" with which one can "build apps that are a delight to use, faster than you ever thought possible"[17]. In this thesis we test this premise and take a look at how the development process can be improved by using Meteor.

1.1 Project scope

The idea is to create a better fan experience during sport games using modern technologies. Wearables, smartphones and digital signage screens are the devices we want to ultimately reach and display our collected information on. Besides displaying ordinary game statistics to fans, we want to give people in remote locations a better impression of the mood in the stadium during the game. To do so, we collect motions on wearable devices and make use of other body sensors in order to analyze these. Some wearables already have heart beat sensors which could be used to measure excitement of a given individual. We then transmit analytics to our server, where we can further distribute statistics about the crowd's mood to various display outlets, as well as any user with our app.

We achieve this interconnectedness by implementing a full stack web application using Meteor. Every component is directly implemented using Meteor functionality. We create a server which processes and coordinates all requests, a website enabling users to view the data and finally an Android app that connects to the wearable device, for example a smartwatch.

Meteor brings a new aspect to web development, as one develops server- and client-side code at the same time. This code reuse makes rapid prototyping easier, while greatly reducing the need of immediately having to design good application interfaces at a early project stage. This is ideal for the development in an agile SCRUM development process.

Unfortunately, mobile browsers are not yet fully capable of interacting with Android Wear, the operating system that runs on wearable devices. Due to this obstacle we resort to implementing a normal Android app that connects to the server and the smartwatch, and forwards any data between those two endpoints.

Figure 1.1 shows the structure on how the data will flow between the different end points. (A) represents our server running meteor in some distant data center while transmitting data through the Internet (D) to our clients (B) and (C). (C) can be any device that can access the corresponding website in a browser directly like a television, laptop or computer. (B) represents an Android smartphone which then can be connected to a smartwatch over Bluetooth. All these connections are bi-directional as any endpoint can also send data back to the server to relay any sensor information.

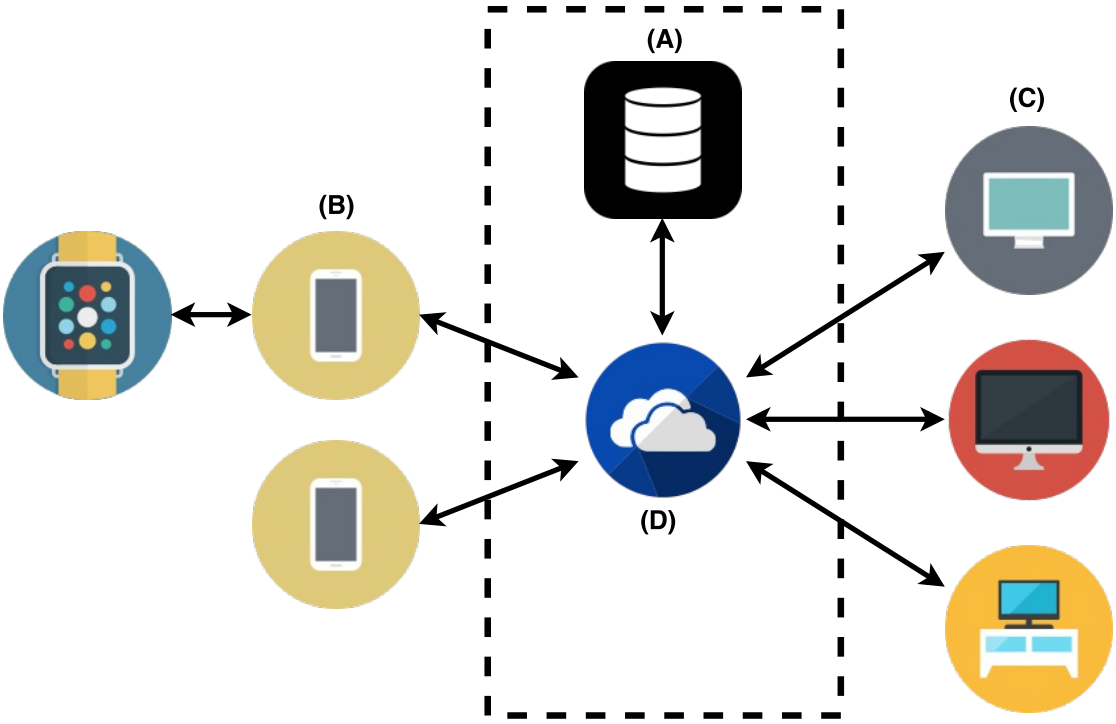

Figure 1.1: Basic structure of the product

1.1.1 Fanmode's approach to real time interaction

The London based company Fanmode has developed a similar system over the last couple years. Their idea is to connect sport fans and crowds worldwide in order to improve the game day experience a fan has, by giving feedback how other people feel about current events in the game. For example, the user can applaud a referee decision if he agrees with it or he could also show his disagreement while commenting in a Twitter-like discussion feed.

For this purpose Fanmode has developed an app for smartphones which enables fans to interact during the game. They can cheer, chat with friends and look at the activity stream which shows events of the current game.

Also feedback from the users can be implemented in many other places for example directly in the television broadcast, or on so-called "Vibeboards". Vibeboards are websites which update their content without reloading the page by using websockets, displaying most of the information gathered in the app. Boards can be shown on a variety of different displays because the only requirement is to have a somewhat current

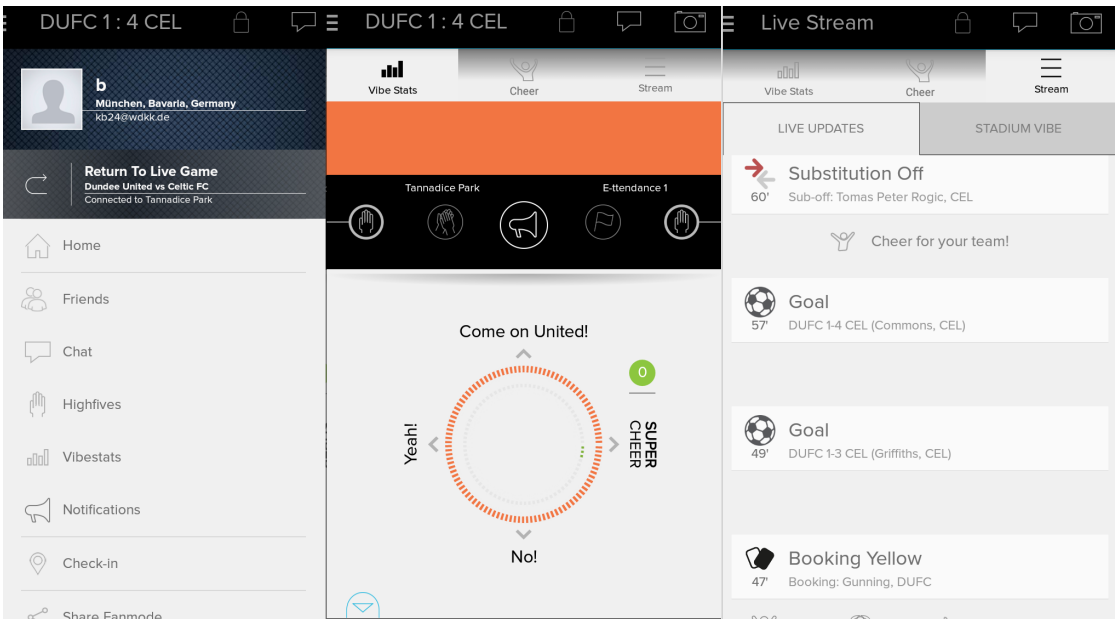

Figure 1.2: The Fanmode app

version of any web browser. So Vibeboards can be used in the stadium, billboards or at public viewing events without much setup effort. Also any user on the Internet can access the boards and follow the game on their tablet, computer or laptop.

1.1.2 Better user experience

Currently, the user has to open the mobile app and do all interactions explicitly in order to trigger them. This thesis will try to improve this interaction by utilizing the abilities of a smartwatch. Ultimately, we want to be able to use voice commands, motion detection and the display in order to simplify the feedback users can give and receive. Ideally, all motion gestures are automatically sent to the server as soon as they happen and in turn enable other users to see these interactions in close to real time.

Inherently the term real time in computer sciences describes a highly specialized field of computing where systems can calculate an operation by a specific deadline which would usually be within milli- or even microseconds. The computation by the deadline is guaranteed and requires a higher investment into the used hardware. In our web application context real time refers to deadlines within seconds as it is a distributed system. The delivery might not be guaranteed by all involved components, but we are working within the limitation of what is being perceived by a human as instant - a couple seconds at most.

The vision for the final product would be a smartwatch which can be produced at low costs and possibly used as a entry ticket for season ticket holders. This would simplify the procedure who regularly go to see sport games at the stadium. Instead of having to pick up tickets from the box office with their season pass, as it currently is at Texas A&M football games, people could simply walk up with their smartwatch, authenticate via the built-in NFC chip and enter without hassle.

Another possible enhancement would be to enable fans to pair their credit card with their season ticket and pay for concessions at the game without the need to carry along their wallet. With a simple scan of the smartwatch the expense would be billed directly to the linked account. This might even increase sales, as using credit cards with chip instead of a mag-stripe, can take some customers up to a minute to complete. Existing wearable device like the Apple Watch already support payments with the built-in chip[4]. Android Wear devices will probably follow in the years to come, but currently they don't support this feature[23]. Offering a common solution for Apple Pay, Android Pay and any other NFC enabled device will be in the interest of concession operators, if the market of wearable devices expands further.

1.1.3 Vibeband: cheap and easy-to-use wearable device

The team at Texas A&M university in College Station is designing and building a smartwatch as part of their Capstone project. The goal is to have a product that can be cheaply manufactured while improving the existing Fanmode experience.

This so called Vibeband will come with motion detection and eventually can be connected to the Fanmode app. It will have reduced overall functionality, compared to current models from large manufacturers that are available with Android Wear, but will also have a lower production cost.

Figure 1.3: Similar product from Microsoft recently featured on The Verge

Off-the-shelf smartwatches, which have been used by Fanmode until now, come at a much higher price and may lack the processing power required for motion detection. Vibeband's benefit is to have a tailored solution, which for example can perform motion detection calculations on the device itself, without being susceptible to possible erroneous Bluetooth connections.

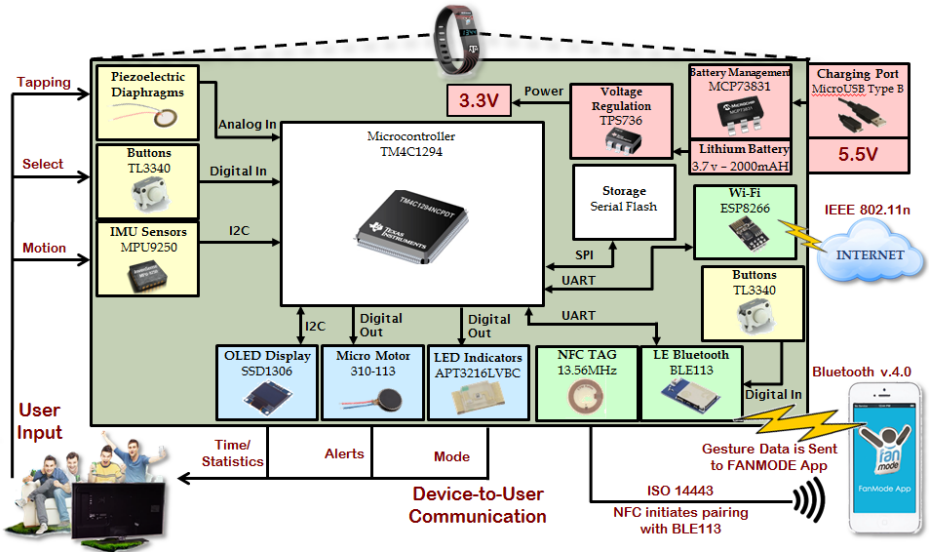

Figure 1.4: Conceptual block diagram showing all the major features of the vibeband

The Vibeband has its own processor, battery and comes with a variety of sensors while being able to connect to smartphones with Bluetooth, as well as directly accessing the internet via Wi-Fi hotpots. Some people may not own a smartphone and therefore would not be able to connect to their phone via Bluetooth. Directly accessing the internet using Wi-Fi also removes any need to set up the Vibeband, which might encourage non-digital natives to make use of it. Near Field Communication (NFC) will be primarily used to connect to the correct device over Bluetooth Low Energy (BLE) and prevent the user from having to go through a tedious setup process.

The micro-motor is able to vibrate in case the score changes. A game's current score is shown on the Organic Light-Emitting Diode (OLED) display. The solution transmits all scores to the device using a BLE write characteristic, but when connected to the internet directly over Wi-Fi it resorts to polling a Representational State Transfer (REST) Application Programming Interface (API) endpoint. As it only needs to transmit two values between the Vibeband and the server it seems like overkill to implement a full Distributed Data Protocol (DDP) client on the embedded micro controller.

1.2 Motivation for Meteor

As a web developer with several years of experience, the standardized approach to developing applications on the internet became very tedious. To offer users real time updates with Symfony, a commonly used PHP: Hypertext Preprocessor (PHP) framework,

one has to manually implement every component of the system yourself. Create the websocket server, negotiate a protocol between front and back end developers and finally actually tie this feature into the normal application on the website and server. This presents quite a challenge even for experienced developers.

Meteor comes as a full stack solution that manages everything, from database to frontend development in one tool. One does not have to think about how database schema's need to be build, or how synchronization with User Interface (UI) will properly function. We download the framework to our developer machine and can get right into programming our application while receiving many features of a modern website from the included package management system.

Rapid prototyping has become one of core development traits of agile development teams. An idea is discussed in the morning at the daily scrum meeting and by the end of the day it might even be already deployed to the customer. Meteor makes this process of rapid development and deployment incredibly easy for developers at any experience level. We want to take a closer look at this framework and look deeper into any disadvantages that might come with this boost of functionality.

2 Approach

In this chapter we will go into more detail on how we want to approach the problem statement at hand and which possibilities to solve it there are.

As we want to reach a big audience we have to make the application easily accessible. This means either develop a solution of each of the biggest platforms like Windows and Mac as well as mobile devices, or create a web application that can be accessed from any of those platforms. A customized solution for each of these platforms is out of the scope of this thesis, so to reach as many people as possible we will try to accomplish our goal using a web application.

2.1 Real time web applications

We want to connect all the fans in real time. Feedback from one user should be transmitted and distributed to others within a few seconds. Users should still have the impression that the action is immediate and that their gesture was detected. If the latency between the detection and display on the Vibeboard is too big then the value of the information decreases dramatically. In ice hockey the mood of the crowd can swing quickly with fast paced games, so we need to be able to handle a lot of clients while not making the system too slow or unresponsive.

We have several possibilities how to get dynamic updates in a browser. Each of these approaches has various advantages and drawbacks which we will take a close look at. Figure 2.1 displays the various ways we can implement dynamic updates in a browser.

2.1.1 Polling as a solution for real time communication

HTTP was specified as a one way protocol where a client can easily download information from a different machine on the network. Originally nobody thought about the fact that it would be actually nice to let a user know if the underlying information changes or that full duplex communication would be needed in a web application.

In order to resolve this issue with the available tools before websockets were standardized, developers would request all new information after a certain fixed time interval passes. This technique is called *polling* and is not very efficient as the client sends

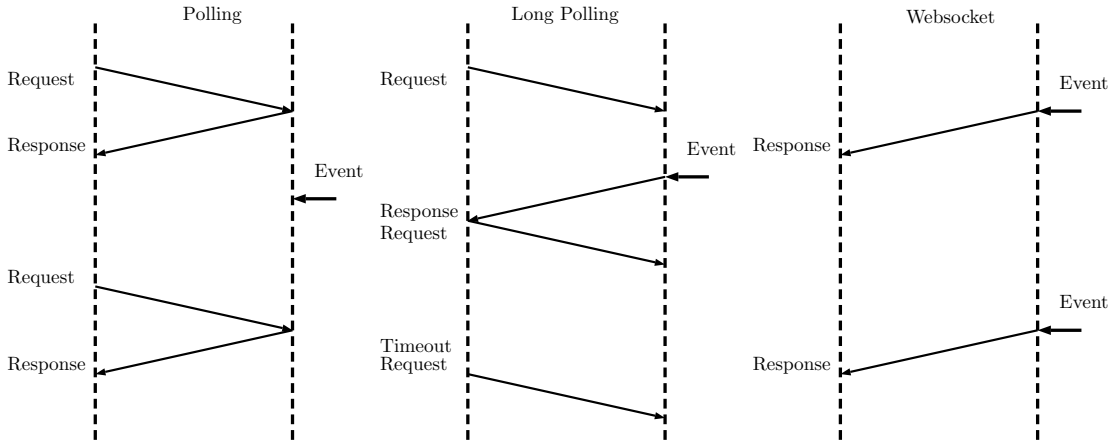

Figure 2.1: Comparison of the three possibilities on how to get updates with Hypertext Transfer Protocol Overview (HTTP)

requests to the server in any case, even if there isn't any new data to fetch. This generates a lot of useless request, traffic and does not scale with a growing user base. Aside from the overhead the fixed fetching interval might impact the user experience. If the time interval is set too high that might lead to big delays from when the data is updated on the server and when the user sees the change.

Another possibility similar to polling is *long polling*: a client sends a request to the server immediately after downloading the website, but instead of replying with an empty message if there is no new data, the server waits till he needs to send an update and only then replies back to the client. If a certain timeout is reached the server drops the Transmission Control Protocol (TCP) connection and the client sends in a new request.

Both polling and long polling use XMLHttpRequest (XHR)/Asynchronous JavaScript and XML (AJAX) functionality to send asynchronous HTTP requests to the webserver in the background after the initial website download. These requests act like a normal HTTP request but are initiated by JS code instead of the browser itself. Developers can choose to request additional data at any given point after the JS-scripts have been parsed and executed.

2.1.2 Websockets to the rescue

Websockets are a new approach in not only allowing clients to dynamically refresh content but also to allow for a full duplex communication between clients and servers. That means that the server as well as the client can send data at any given time, after the connection has been established and don't have to send out a new HTTP

request. Websockets are build on top of the HTTP protocol and use the same ports for communication. This enables a faster adoption of the protocol as no firewalls or proxies need to be reconfigured. There has been indications though, that some proxy servers might not forward websockets correctly, as they block some of the required HTTP headers for the connection to be upgraded[1].

Originally for every HTTP/1.0 connection a new TCP socket would be created and closed after the website was downloaded. With increased complexity of websites this approach was not practical anymore, as for every extra resource a complete new connection would be established. HTTP/1.1 enabled browsers to reuse a connection to download several resources from the same server, but still this connection would be closed after a couple of seconds making any subsequent AJAX calls create a new socket.

Websockets use the same TCP connection throughout their lifetime and the data frames send after the initial handshake have minimal overhead compared to HTTP headers that are send with every AJAX request. Websockets are a completely independent protocol but use HTTP headers to establish the initial connection. Listings 2.1 and 2.2 show this upgrade from HTTP to the Websocket protocol.

Listing 2.1: Upgrade request to establish a websocket

```
GET ws://localhost:3000/sockjs/872/0asvwl_m/websocket
Accept-Encoding:gzip, deflate, sdch
Accept-Language:de,en-US;q=0.8,en;q=0.6
Cache-Control:no-cache
Connection:Upgrade
Host:localhost:3000
Origin:http://localhost:3000
Pragma:no-cache
Sec-WebSocket-Extensions:permessage-deflate; client_max_window_bits
Sec-WebSocket-Key:914G8rbnwf0MmNZQpC7BwQ==
Sec-WebSocket-Version:13
Upgrade:websocket
User-Agent:Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
    Gecko) Chrome/48.0.2564.82 Safari/537.36
```

Listing 2.2: Upgrade response

```
HTTP/1.1 101 Switching Protocols
connection:Upgrade
sec-websocket-accept:DwPAFku+1BfAbYVrDMZwEDEGK2U=
sec-websocket-extensions:permessage-deflate
upgrade:websocket
```

After the initial handshake over HTTP the websocket communicates over a totally separate TCP connection enabling both parties to send frames at any given point in time. The performance of websockets compared to polling is significantly better[1][22] and can be used for our requirements of real time communication.

2.2 Meteor: A new approach to web development

There are several key Meteor technologies that help us reach our goal easier. In this chapter we will take a closer look at how the real time communication chain from the server to the client works.

2.2.1 Unified data access with DDP

DDP is a standard introduced by the Meteor developers to resolve on of the biggest problems encountered in web development: querying the server for data from a JS context and also enable the user to make changes to that dataset. Synchronisation between the permanent storage like a SQL server and the data the user sees in the browser has always been a big problem to tackle without any standardized solution.

With the emerge of jQuery, AJAX became a more accepted technology. AJAX enabled developers to design websites that dynamically fetch new data once the initial content has been downloaded using a XHR. The very first websites just downloaded one HTML file and maybe some additional resources. With AJAX we can trigger additional XHR requests fetching only the data that changed and eliminating the need for a full page download which can be multiple mega bytes on more complex websites. On a mobile connection we want to preserve as much bandwidth as possible to reduce costs and speed up the responsiveness of the website.

Although not an completely new idea, the first time the term AJAX was used in the context of web development was in 2005[15]. Back then Internet Explorer (IE) was dominating the browser market and was one of the first browser to implement this new technology. Initially the JS API for XHR that came with IE was complicated because there was no official standard released yet, but jQuery enabled developers to use a consistent, easy to understand and cross browser compatible API. This made the possibilities of AJAX available to a broader range of developers as the entry bar to use this functionality got lowered significantly. Listings 2.3 and 2.4 show the different implementations with and without jQuery. Although there might not be too big of a difference with this simple example, but there are more problems to watch out for with the normal XHR api.

Listing 2.3: AJAX request with jQuery

```
$.ajax('service/username', {data: {id: '1234'}});
```

Listing 2.4: AJAX request with the native browser API

```
var xhr = new XMLHttpRequest();
xhr.open('GET', encodeURI('service/username?id=1234'));
xhr.send();
```

Going back to the DDP protocol, the major difference is that AJAX relies on the HTTP protocol for transportation, while DDP is build upon websockets. The key distinction is that HTTP is considered to have a big overhead and also is not bi-directional. HTTP sends out specific headers with each request to the server which in some cases might be larger than the actual data transferred (Listing 2.5).

Listing 2.5: HTTP headers send along with a request to Wikipedia

```
:host:en.wikipedia.org
:method:GET
:path:/wiki/Duplex_(telecommunications)
:scheme:https
:version:HTTP/1.1
accept:text/html,application/xhtml+xml,application/xml;q=0.9,image/webp;q=0.8
accept-encoding:gzip, deflate, sdch
accept-language:de,en-US;q=0.8,en;q=0.6
cache-control:max-age=0
cookie:GeoIP=US:TX:College_Station:30.57:-96.28:v4; WMF-Last-Access=20-Jan-2016
dnt:1
if-modified-since:Tue, 29 Dec 2015 17:27:49 GMT
upgrade-insecure-requests:1
user-agent:Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
    Gecko) Chrome/47.0.2526.111 Safari/537.36
```

Also with HTTP the server cannot send updates to the client on their own account, but rather have to wait till the client comes around with the next request. Meaning if a client shows data at the time stamp t and the data gets updated on a second client at t+1 the first client will not automatically get the update until he refreshes the locally stored data sets. With websockets the users browser as well as the server can communicate at any given time with each other.

DDP basically is a standardization of the publish-subscribe pattern[9]. It defines specific messages[24] that can be send via the protocol and how to keep two distributed data sets in sync. The most important messages are:

- **Connect**: establish a connection to the server

- **Publish**: offer a specific data set to be subscribed
- **Subscribe**: consume a publication, receive the data and listen for changes
- **Ping/Pong**: heartbeat message to check if the client is still available
- **Added**: called when an item has been added to any subscribed collection
- **Changed**: an item in a subscribed publication has changed one of its values
- **Removed**: an item got removed

DDP is currently implemented by Meteor only, but could also be directly supported by MongoDB or any other storage system. MySQL could for example allow connections via DDP, by subscribing to the data directly from the client without having to go through a server application, that basically just interfaces database access with a REST API. This would allow for even better performance as the current database connectors only imitate some of the real time functionality.

DDP uses a specific subset of the JavaScript Object Notation (JSON) which is called Extended JSON (EJSON). EJSON[1] adds data types which JSON lacks, for example dates, binary data or even custom user defined ones. We want to really replicate the server side MongoDB to the client side minimongo database. Once a subscription to a specific set of data is made, all the data in that collection will be pushed to clients minimongo database over the websocket in the EJSON format. From there this data can be queried locally without the latency of a round trip to the server, making for a even more responsive UI. Once the collections underlying data changes these changes are automatically pushed to the client as soon as they happen. From there the UI can be updated and the user can immediately see the changes that were made from a different machine.

With the emerge of AngularJS and other front-end frameworks that enable developers to create single page apps, the term two way data binding was born. Single page apps are web applications that do not reload the page after it was initially downloaded, but solely rely on XHR for data acquisition and manipulation of the Document Object Model (DOM) tree. Two way data binding allows the MVC framework to bind data from the underlying model to the UI. Meaning that if the data changes in the DOM tree it would also change the bounded variable in the model. Lets take a form input on a sign up page for example: the user enters a password and while he types we can check the bound variable for minimum password requirements. Now this was already possible previously with jQuery and plain JS by listing to certain events that would be fired in case of a changing value but it was a pain to maintain this connection between multiple

[1] `https://github.com/primus/ejson`

fields and its models. In this scenario the data in the model would be lost as soon as the page would refresh and also could not be seen on by anyone other than the open browser on that computer.

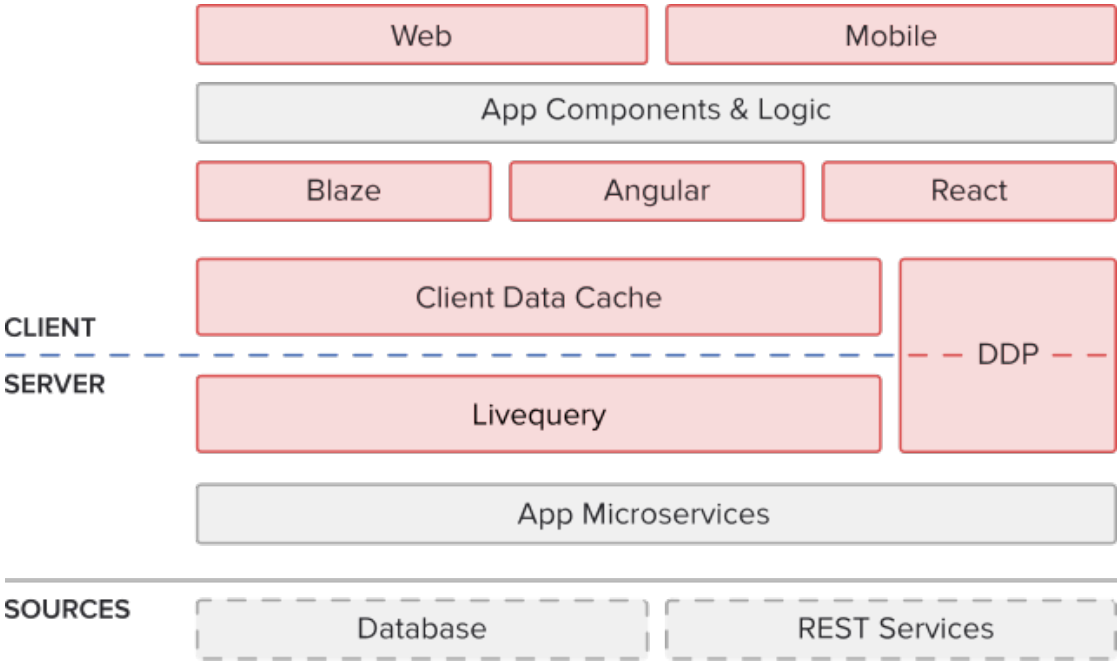

Figure 2.2: The Meteor stack[16]

The data chain from the server to what the user sees in his browser is called three way data binding and was a big challenge to accomplish previously. Meteor offers all the tools in one package out of the box making it easier then ever for novice developers to implement real time web applications. Now not only is the model bound between the UI and the client side model but the model will be also replicated to the permanent storage on the server. Figure 2.2 gives us a good overview on how all the different technologies are connected.

This also introduces many more challenges as now timing and latency comes into effect. We need to make sure that we do not encounter problems like the lost update dilemma. The problem arises when a client tries to update outdated data. Lets say we have two clients incrementing a simple integer value at the same time. If they both send out the request at the same time we might lose one of the increments as the update has not yet reached the other client and in turn it increments the old value. Currently Meteor and MongoDB do not try to resolve this problem, but rather MongoDB follows a

last write wins approach. This means that an update might be lost and integrity of the data is not guaranteed throughout the whole client network.

2.2.2 Real time database interaction using Livequery

Database connectors that support livequery return a normal cursor to iterate over the results of a query. In addition they return a stream of changes made to the objects in the result set, so one can track the modification or removal of items. This enables us to propagate the changes back through DDP to the client, where we can then update the UI. Meteor provides us with a proof of concept livequery database connector for MongoDB, but in the future this might be a feature directly implemented into database systems.

To accomplish this task the current implementation for MongoDB acts like a replication server, in order to receive notification about the changes made to collections. For Structured Query Language (SQL) based systems, triggers could be used or if that is not a possibility one would have to resort to polling for changes, which would be really inefficient.

Livequery is to be considered the main bottleneck when considering scalability with Meteor. Currently it does not support MongoDB sharding automatically, so one might have to think about implementing a message bus in the future, but to get started with a proof of concept for our product the current functionality will do.

2.2.3 Atmosphere: a online package repository

Atmosphere[2] is the official package repository for all Meteor packages. Almost all of the functionality of Meteor is distributed via packages from Atomosphere. The major benefit of using Atmosphere and Meteors packaging system is that it is build into the Meteor build chain. Once one starts a meteor server it automatically checks for any new versions and as the server runs, it also monitors changes to packages and restarts / reloads all connected clients & servers accordingly.

One can not only find packages for Meteors own components but also for many other front-end related frameworks as well as a few NodeJS packages commonly used. Without having to worry about dependencies one can simply install a package by running the `meteor install <packagename>` command in the project folder. To update packages one can simply run the server by typing `meteor` or to force an update run `meteor update` in the command line. This simplifies development even further for novice developers as one does not have to deal with NodeJS or its package manger NPM at any given moment.

[2] `https://atmospherejs.com/`

At the end of 2015 Atmosphere contained over 9000 packages[5] making it a good resource for any requirements one might have in a modern web application.

2.3 Easier front-end development

Up to now we have only talked about the server implementation and transmission of data changes to the client but have not taken a closer look at how to deal with the incoming DDP messages on the client and how to propagate changes from the model to the actual UI.

2.3.1 Blaze: Meteor's simple approach for an UI-framework

Blaze comes prepackaged with Meteor and is the recommended UI-framework. It renders templates with placeholders for data into HTML code but also takes into account any changes that are made to the underlying data and refreshes the template if needed.

Blaze claims to be simpler than other frameworks, but lacks significant functionality to really enable developers to create complex applications. It does not come with a router which would enable websites to have multiple sub pages or any possibility to create services to interact with systems other than the meteor server. Often you need to work with several different APIs and pull in data from other places than your own server.

For a basic project Blaze will work just fine, as it comes with the most basic of functionality. To build a real web application though, we will require more functionality. Plug-ins for Blaze are available but it cannot yet compete with other frameworks out there.

2.3.2 Using AngularJS to revolutionize front-end development

Angular was first released back in 2009 as a new approach to create single page applications. It is maintained by Google and the open source community and was quickly adapted by developers after the initial release, because Angular had a big company overseeing the development of the front-end framework.

Angular's main goal was to detach the manipulation of HTML code from the application logic. Classically with most other JS frameworks like jQuery one would get a reference to the element in the DOM tree with a selector and then alter its attributes from the application logic directly. Separating those two aspects from each other was not an easy task and often lead to novice developers not following best practices. Angular successfully achieves this separation by automatically binding the model to the

displayed template and refreshes the value automatically if the value in the underlying client side model changes.

Angular analyzes the HTML code when the page is first loaded and looks for special attributes in the HTML tags like `ng-app` and `ng-controller`. It then tries to load the matching controller module. That controller comes with its own template that it populates with data from an API or Angular service. The controller has its local `$scope` variable, which is actually an Angular service, that holds all values assigned from the controller to the template. In the template one can then access the variables with double curly brackets like so *{{user.name}}*.

Here is where the magic happens: as the scope object changes, Angular updates the value in the template accordingly. Likewise if the user changes the value in the DOM tree the value in the model will also be changed. Two-way data binding describes exactly this behaviour of bound variables between the model and the UI and it makes a separation of application logic from UI code much simpler, compared to the usual approach with jQuery.

For Meteor there is a specially adapted version of Angular that supports collections with Angular's two-way data binding out of the box. This enables us in the next step to actually use normal Meteor collections in any Angular controller and bind data from there directly to the object on the server. Any time the object on the server changes it is propagated to the clients minimongo via DDP, from where it then gets shown on the UI.

In Atmosphere we can find many more packages for Angular, which enables us to use Meteors package management system to keep these up to date. Sometimes those packages make small adaptions, but mostly they come directly from the representative open source repository.

3 Implementation

To show that the approach we chose actually holds up to the promise, we will implement a proof of concept application showing the full chain from the wearable device to the server. To demonstrate the capability of the full chain at A&M and at the TUM in Germany we will have similar setups at both locations.

3.1 Test environment

In order to be able to test our application at both locations we will use specific hardware and software.

As a server we will use a Debian based Linux system that comes with all the necessary requirements to run NodeJS and MongoDB. Debian is also supported by Meteor out of the box making it headache free to setup on. For development Jet Brains offers a specialized web development Integrated Development Environment (IDE) that supports Meteor.

For mobile development we will use Android as its development tools are free, documentation is easily accessible and its currently the most used operating system[13]. Android comes with a mature IDE of its own called Android Studio which enables us to easily debug any applications. As for the smartphone we are using an inexpensive device called the LG Power. It comes with a fairly recent version of Android called Lollipop, that brought many improvements especially in the area of BLE APIs[3].

In terms of the wearable device we have to support two options. First we will be using a LG G watch for testing with Android Wear. Wear comes with an build in APIs for communicating back to the Android app on the smartphone which makes it a lot easier to develop for it. Second we also want to support the Vibeband developed by the team at A&M university, that uses BLE to communicate efficiently and conserve energy. We will want to implement similar functionality with both devices but the Wear device has more possibilities, as it supports almost everything an Android smartphone could do, while the Vibeband will probably only have a embedded processor and a two color display. If we also support Android Wear customers will be able to bring their own devices and use that to make use of any functionality that our app provides.

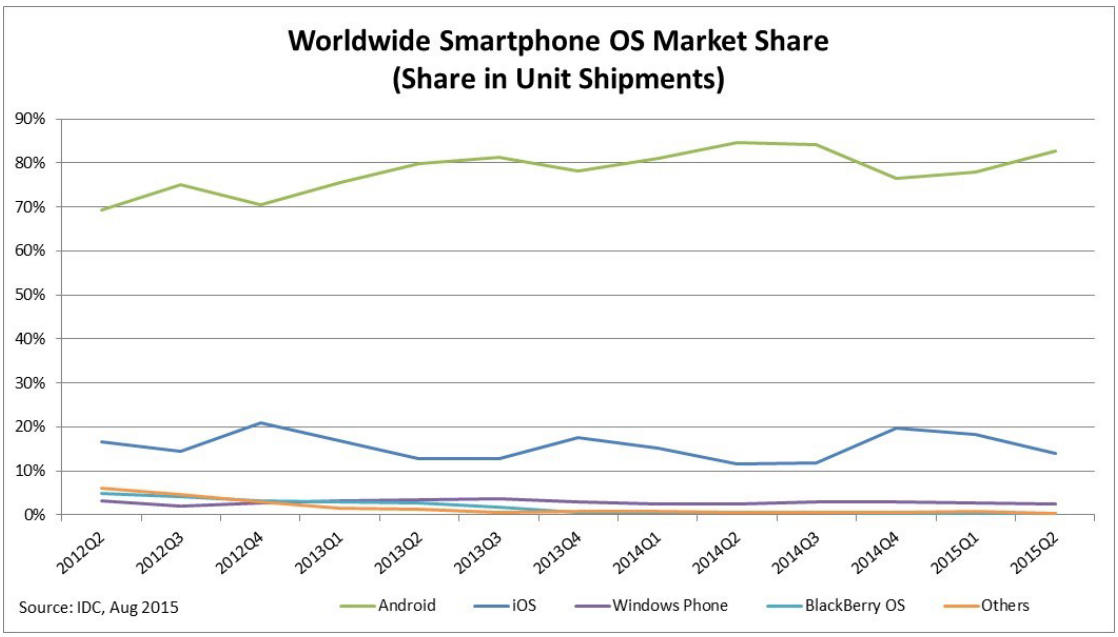

Figure 3.1: Current market shares for smartphone operating systems[13]

3.2 Server-side programming with Meteor

In this chapter we will take a closer look at the technologies on the server side and how to setup our project.

3.2.1 Introduction to websockets

To see sockets in action and get familiar of the concept of websockets we want to setup a simple application using PHP and a library named Ratchet[1] that enables us to implement a basic chat server with websockets.

With Ratchet we can setup a simple webserver within PHP that will handle any incoming requests and relay them to any connected client. Listing 3.1 shows how simple it is to create a websocket server

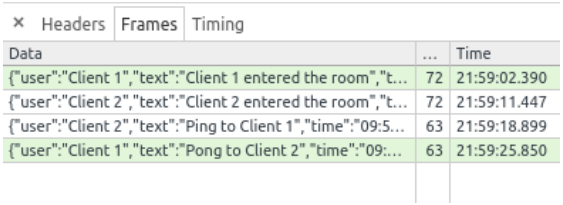

Figure 3.2: Frames send and received via the websocket in our sample chat client

[1] http://socketo.me/

with Ratchet. There is a lot going on in listings 3.1, 3.2 and 3.3 so let us take a closer look at the code.

In order to use websockets we also need to handle any incoming HTTP requests to upgrade the connection to a websocket. That is why we first use a basic `IoServer` to open the socket, second pass it a `HttpServer` to allow websocket upgrade connections and finally we create our `WsServer` that will handle all websocket requests. The `WsServer` uses the callbacks within the `Chat` class to handle any incoming connection requests and also it remembers any incoming clients in order to send out any messages that come in.

Listing 3.1: Simple websocket server with Ratchet

```
<?php
$server = IoServer::factory(new HttpServer(new WsServer(new Chat())),8080);
$server->run();
```

Listing 3.2: Connect to the server and send a message in JS

```
var con = new WebSocket('ws://localhost:8080/');
con.onmessage = function(e) {
    var msg = JSON.parse(e.data);
    alert(msg.user + "␣" + msg.text);
};
con.send(JSON.stringify({
    'user': 'Client␣1',
    'text': 'Ping/Pong'
}));
```

Listing 3.3: Ratchet server callbacks

```
class Chat implements MessageComponentInterface {
    protected $clients;

    public function onOpen(ConnectionInterface $conn) { ... }
    public function onMessage(ConnectionInterface $from, $msg) { ... }
    public function onClose(ConnectionInterface $conn) { ... }
    public function onError(ConnectionInterface $conn, \Exception $e) { ... }
}
```

To connect to our sample server we use a simple HTML file that contains some JS code. In listing 3.2 we can see how to connect to the websocket. The API for this is help simple thanks to the standardization effort of the W3C.

In figure 3.2 we can see the individual frames exchanged between two clients. Websockets provide no additional structure to data send over them but can transfer data

in binary or text form. The simplest way to structure data is to use JSON to serialize objects then send them and deserialize them after they arrive. One can inspect any websocket connection in Google Chrome inside the developer console and see all the individual frames. Meteor also utilizes JSON in their DDP protocol to transmit messages.

This short example showed us that websockets are a brilliant and simple way to bring real time updates to any web application. Without much work we were able to connect to our server and build a lightweight application.

3.2.2 Project setup

Our server will be based upon Meteor which uses NodeJS and MongoDB. To get started with our application we first need to install and configure Meteor on our local machine. We do that by running the command `curl https://install.meteor.com/ | sh` in the shell of our choosing which will download and install any dependencies as well as the Meteor command line tool. This works only on Unix based systems but there is an installer for windows as well.

Afterwards we can simple initialize a new Meteor project using the `meteor` command line tool that we just installed. Everything related to the Meteor project is managed using that command line tool. To create a new project in the current folder execute `meteor create <project-name>`. That will create a new sub folder with the project structure and some files to get started. Inside that folder you can just run `meteor` without any options to start the server and access the website at `http://localhost:3000`. As we can see using the `meteor` tool is really simple and easy to learn for beginners.

There is no need to restart the server each time one edits the file. Meteor takes care of automatically refreshing the client and server when files change. This makes development easier as one does not have to think about restarting. Meteor sets up watchers for all the files in the current project and also installs new packages as they are added to the list on the fly. Alongside with the fluent structure of MongoDBs object oriented database this improves the initial speed of development significantly. Meteors philosophy of pushing less important tasks into the future and simplifying rapid development of new applications allows us to rapidly prototype ideas.

3.2.3 Collections: abstracting the database away

To store any data permanently in a structured matter we will utilize MongoDB, Meteors connector and the API that Meteor supplies. One of Meteors core concepts are full stack database drivers that let you use the same API anywhere in the whole system. This mean that we can use the same code on the server as well as on the client side in

the browser. Meteor automatically bundles the correct library in the background and decides if we are working on the full MongoDB or minimongo database. This cuts down on the amount of code we have to write which in turn reduces the amount of bugs and debugging we have to do. Once the query runs on the server we can copy and paste it to the client side or event better use a common JS file that holds any queries we run.

Listing 3.4: Define a new collection

```
Soccerseason = new Mongo.Collection("soccerseason");
```

As we can see in listing 3.4 it only takes a single line to create a new collection on the server in our MongoDB. The exact same line can be used in on our client to create a collection in its minimongo. The current problem is though that now everyone can perform Create, Read, Update and Delete (CRUD) operations on the client and the server. To limit the operations the client can commit to the servers MongoDB we can specify policies.

Listing 3.5: Allow insertions to be made to the collection

```
Soccerseason.allow({
    insert: function (userId, post) {
        if(userId == post.userId){
            return true;
        }
        return false;
    },
});
```

Listing 3.5 shows us how to allow insertions on that specific collections based if the condition is true or not. The function can check against a variety of conditions so that complex right management should not be an issue.

To actually get any data from the collection we need to run a query against it. In contrast to relational database systems one does not use SQL to access any data but rather the api of the Meteor collection. These collections come with a set of functions to perform CRUD operations on the collection. Listing 3.6 shows how to find one specific record with its unique identification.

Listing 3.6: Find one record in the collection

```
Soccerseason.find({_id: season}, {
    limit: 1
});
```

`Find` runs the livequery against the database and returns a cursor with the current dataset but it also streams any further changes to the data set from any CRUD operation.

With the special Angular packages that comes with Meteor we don't have to worry about manually updating any changes in the UI but that is all done for use. We can essentially create a three-way bound interface with just a few lines of code. No need to think about synchronisation issues, transport or storage on the website.

Again this code can be used on the server as well as the client - Meteor automatically knows from the the environment what back-end we are working on. If we insert an object into the client's minimongo it propagates the change to the servers MongoDB via the DDP protocol.

The major benefit is, that the client's browser does not have to wait for the round trip to the server to be completed but rather the UI updates immediately as we basically work on the local browser based database. This makes the interface seem more responsive instead of having to wait till the server responds with a confirmation message. DDP allows for synchronization of every single operation that has been executed on the database instead of comparing every object with the server side.

3.2.4 Folder structure

When building the application Meteor pays close attention to what the folder structure is. As we are developing client and server at the same time we have some common code but also static files that we want to deliver to the client. Structuring the project correctly is essential so that we reduce the total size that is sent to the browser as well not leak any security sensitive code that might access internal APIs.

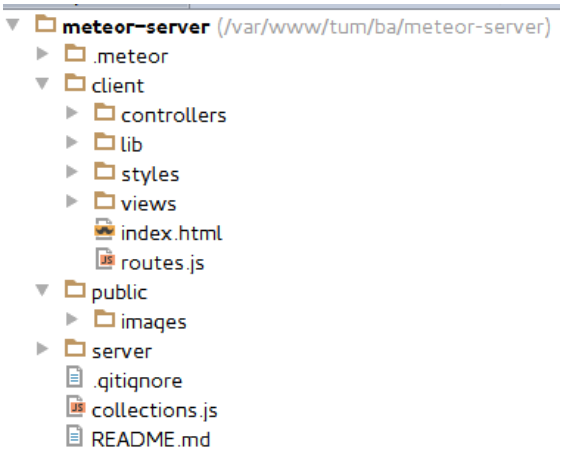

Figure 3.3: Meteor project structure

We have two possibilities to tell Meteor that code should either be run on the client or server. Listing 3.7 shows how to check in what context the current code is running in. We can use that to separate our Angular code from code that executes on the server.

Listing 3.7: Execute code on the server or client only

```
if (Meteor.isClient) { ... }
if (Meteor.isServer) { ... }
```

Working with conditionals might seem simpler at first but quickly will turn into spaghetti code as the application get more complex. Any javascript files that are in the `root` of the project will be executed on both sides. In contrast we have the `client` folder that will only be included in the browser and the `server` folder that will only run on the server side and never be deployed to the browser.

For any assets like pictures and other static content there are `public` and `private` folders. All files inside the `public` can be access directly via HTTP as for the `private` folder these files can only be read using Meteors `Assets` API. Files inside `client` get aggregated and optimized and also cannot be directly access from the web, so using the correct directory for the right purpose is crucial.

3.2.5 Publications

One of Meteors core concepts is the publish/subscribe paradigm that allows us to only request a specific subset of data be synchronized to our clients minimongo database. This is crucial for performance and security aspects as we cannot send several megabytes worth of data over the internet nor do we want to disclose all saved that. Currently with the `autopublish` package enabled all data that is stored inside our MongoDB is send to the client but we rather just want to get the information that is relevant for the current page the user is on.

Listing 3.8: Create a publication

```
Meteor.publish('soccerseason', function () {
    return Soccerseason.find({}, {
        fields: {'_id':1,'caption':1,'lastUpdated':1}
    });
});
```

Creating a publication is pretty easy: we just call the `publish` function of the global Meteor object, pass it a name for that publication and a function that will supply us with a livequery result cursor. To get that cursor we simply use the Meteors MongoDB api and pass any options to the `find` function. In listing 3.8 we return all leagues that we have stored but only a limited set of values from each object.

Once a client subscribes to this publication he will receive the full result set from our `find` operation and also any updates that happen to occur as long as he is subscribed. We can also filter out results based upon if the user is logged in or not.

3.2.6 Optimistic UI with Meteor methods

Publications do not offer to update any data but rather one has to either use a collections CRUD operations. This might not be ideal as we want specify which records a user

can alter. In Listing 3.5 we could see how to configure collections to allow/disallow updating of collections and how to check if a user has the right to alter specific records.

Although this might work for simple collections it might not work for more complex operations where we have to check multiple collections for the right to execute a given CRUD operation. This is where methods come in. Methods are run on server and client side simultaneously enabling us to add a layer of security before executing an update or insert statement on our collection.

A nice side benefit of using methods to perform any operation is that we can make use of a feature called *optimistic UI* that runs a simulation of that operation on our local minimongo while also sending out a AJAX request to the server at the same time, to execute the method on the real MongoDB. This allows the UI to show the performed action, before the result from the server is returned, saving us the requirement to wait for a full round trip. [25]

In listing 3.9 we can see how to define a method and then call it passing a parameter. In the method we then check if the user is logged in and add the detected gesture to an fixture's array of claps.

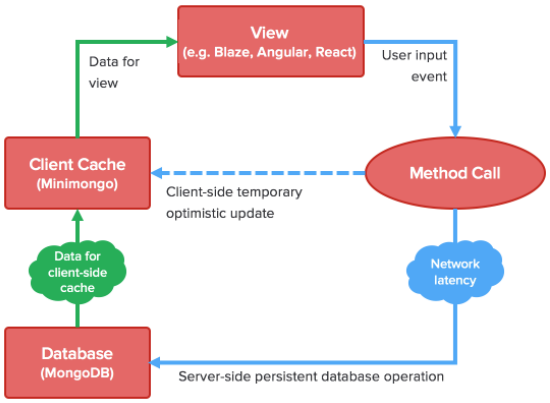

Figure 3.4: Optemistic UI latency compensation[25]

Listing 3.9: Define a meteor method and call it

```
Meteor.methods({
  addClap: function (fixture) {
    if (!Meteor.userId()) {
      throw new Meteor.Error("not-authorized");
    }
    Fixtures.update(fixture, { $push: { claps: Meteor.userId()} });
  }
});
Meteor.call("addClap", fixture);
```

3.2.7 Using cron jobs to automate recurring tasks

To synchronize our game data we currently use a cron manager from within Meteor. These is a package available on Atmosphere named `percolate:synced-cron` that sup-

plies an javascript API to set up recurring jobs.

Listing 3.10: Create a cron job that runs every 5 minutes

```
SyncedCron.add({
    name: 'Update␣Soccerseasons',
    schedule: function (parser) {
        return parser.text('every␣5␣minutes');
    },
    job: updateSoccerseaosons
});
```

We want to fetch the game data from an external REST API that supplies us with close to real time data for several leagues free of charge. As we want this to happen all the time and not when the user connects we need to use a cron job. Fetching the data on an request from the user would add unnecessary latency to the transaction.

Listing 3.10 shows how to set a cron job to run every five minutes. We pass it the function reference `updateSoccerseaosons` that will be called once the cron job should execute. Inside this function we can run any code that we need to fetch data and save it to our MongoDB.

3.3 Building the browser client

To achieve the best performance with a websocket on the website we want it to be a single page app so that we don't have to reconnect to the socket each time we want to display a different page. That means that we only download the page once and then each subsequent page will be handled by our JS UI frontend framework.

Angular solves challenges that come with building a single page app very elegantly and has matured to a product that is used by big companies today. Angular clearly separates application logic from code that is UI related following the well known MVC pattern. In MVC we distinguish between three parts: Model, View and controller. The view in Angular is represented in part by the template with the HTML code, the model is usually a service or in our case a Meteor collection and the controller is the intermediate that contains any logic to connect those two.

The big benefit of using a full on framework like Angular over jQuery is that we don't have to worry about anything in the DOM tree at all or when to update our view. That is all taken care of by Angular itself, giving us the time to focus on the actual application instead of worrying on when we need to refresh the view.

3.3.1 Solution structure

The structure of the initial HTML document that is delivered to the client is very important. The most important attribute to set in our HTML document is `ng-app="vibeboard"`. Usually this attribute is set on the `<body>` tag and declares what our application module is named.

Listing 3.11: Creating a new module with dependancies

```
var app = angular.module('vibeboard', [
    'angular-meteor', 'accounts.ui', 'ui.router', 'ngMaterial'
]);
```

We define a new Angular module in listing 3.11 with any dependencies on other modules passed as an array. The app variable then contains a reference to our newly created module which we can use to configure the router and define controllers.

In order to allow Angular to place another template inside our main HTML DOM tree we need to place the `ng-view` at a location that we want our content to appear. The router will according to the controller we are currently in load the linked template file into that tag.

3.3.2 Dependency Injection in Angular

Dependency Injection (DI) is one of Angular's core concepts on how to allow modules to interact easily with services. It basically is a plug-in system that allows modules to use other services. The injector itself does not know anything specific about the service or if any of them even exists.

After loading up the current module it checks the definition of the module for any dependencies. It then tries to supply the needed service once the module tries to access it. This means that any dependencies and sub dependencies are only lazily instantiated once needed.

3.3.3 Routing URLs to views

One of the big advantages that Angular has over Blaze is the router that comes with it. When working with single page apps that essentially means that we do not change the Uniform Resource Locator (URL) of the page we are currently on but rather just manipulate the HTML. If we go deeper into a pages structure we can't get back to this sub page directly without taking special precautions as the URL still links to the landing page. This method of providing the user with the ability to directly access sub pages is called deep linking. This is also important for bookmarks as well as back and forward navigation.

Originally the hash was used to link to a part of the website but with JS we can use that to our advantage and read anything that comes behind the hash. Anything after the hash, called the fragment, is not transmitted as a part of the HTTP request but only is accessible locally through JS. The Angular router puts the path to the current view as a part of the fragment in the URL and matches routes to that path.

Listing 3.12: Anatomy of an URL

```
scheme:[//[user:password@]host[:port]][/]path[?query][#fragment]
```

The standard router that comes with Angular is called `ngRoute`. It supports basic mapping between the path and a string but is limited in certain ways. Listing 3.13 shows how to define a controller and template for the path `/soccerseason`. Angular will automatically load the template and controller when that path is called up. One can easily chain multiple statements as any function returns an instance to the `$routeProvider`. At the end we use the `otherwise` function to redirect any requests that do not match any route to a default route.

Listing 3.13: Create a route with the ngRoute service

```
$routeProvider.when('/soccerseason', {
    templateUrl: 'client/views/soccerseason.html',
    controller: 'SoccerSeasonCtrl'
}).otherwise({redirectTo: '/soccerseason'})
```

ngRoute is limited in that way that we cannot define multiple views side by side on one page. We can always only load one template and controller but run into problems when we want to show two views side by side. We would need to run another controller from within one controller and that turns really complex quickly as there is not a simple way to tell the router that we are displaying a sub page in our current view.

As ngRoute is basically only a service within the Angular framework we can switch it out for a third party plug-in called `ui-router` that enables us to create more complex view constellations. Instead of using the `ng-view` attribute, ui-router uses a custom HTML tag called `<ui-view></ui-view>`. Also we work with so called states instead of routes that manifest the ability to load multiple controllers at once.

Listing 3.14: Define our application states

```
$stateProvider.state('soccerseason', {
    url: '/soccerseason',
    templateUrl: 'client/views/soccerseason.html',
    controller: 'SoccerSeasonCtrl',
}).state('soccerseason.league', {
    url: "/league/:leagueId",
```

```
    templateUrl: "client/views/league.html",
    controller: "LeagueCtrl"
});
$urlRouterProvider.otherwise('soccerseason');
```

As we can see in listing 3.14 we use a different service called `\$stateProvider` and `\$urlRouterProvider`. The first state is our landing page that will simply show a list of all leagues that we currently follow. The second state is a nested state that is based upon the first soccerseason state. We define this sub state by appending a dot and the sub states name. The sub state will be loaded into the `ui-view` which is located inside the first states template which in turn is loaded into the main `ui-view` tag in our starting HTML document. The URL is also appended to the path of the first state and the controllers are called in sequence.

This provides us a very simple approach to load multiple controllers in sequence without adding too much complexity inside the controllers themselves as well as being able to access any state directly using a URL which would not be possible using the `ngRoute` service.

3.3.4 Subscriptions

We need to subscribe to publications on the client side in order to make use of them. Luckily the `angular-meteor` package makes this very easy for use by providing a dedicated service that takes care of most of the heavy lifting. Listing 3.15 shows how to enable the reactive context, that is that we refresh any parametrized subscription once that parameter changes, and subscribe to a publication.

Listing 3.15: Attach the reactive service to the current scope

```
$reactive(this).attach($scope);
this.subscribe('soccerseason');
```

The really important part is that we only subscribe to the published data but we do not pass any data to the `\$scope`/template. Its crucial to remember that the subscription and the time the data actually arrives at the clients minimongo database do not execute. Once we ran the `subscribe` command we would need to wait a couple seconds till the data becomes available. Luckily the `subscribe` method offers to pass a callback once the actual data is present.

3.4 Android

In order to access the low level APIs for BLE and Android Wear we need to create an Android application. Although we can open the client website in the browser on

the phone, we cannot access any of the sensor data of the phone or the smartwatch. We want to be able to run motion gesture detection algorithms on the sensor data eventually, plus for accessing the Vibeband we would need access to BLE in any case.

3.4.1 DDP in mobile applications

First step to get an Android application working is to check if we can access the data from the Meteor server from the application context without having to change any of the server code. Luckily Meteor does not restrict access to its websocket in any way. It will happily connect to any client that speak DDP correctly and manage any incoming requests as it would handle requests from the browser. The server is completely platform-agnostic of the client and really does not care who connects to its websocket.

As we are accessing the websocket directly we need to implement our own UI in the Android application. For this purpose we will follow Android material design[2] guide lines to make the app look good. This comes with the downside that any changes in UI have to build separate from the website but gives user a better overall experience as he can quickly navigate the common elements used by material design while making use of significant better performance compared to a packaged web application.

Another downside is that we cannot make use of Meteors implementation of DDP but have to rely on a third party or write our own implementation. This means that we need to connect to the websocket and then handle any DDP messages according to the specification.

Luckily there are libraries for several languages out there that already implement both technologies on Android. Android-DDP[3] implements the full DDP protocol and uses TubeSock[4] for websocket handling. Unfortunately Android-DDP does not come with an equivalent of minimongo so we have to handle caching data locally ourselves. But the library gives a good point to start developing without having to deal with too basic protocols.

To cache subscriptions and any data locally we will use HashMaps that store the String identification of the object and the object itself. The JSON string is parsed into normal Java objects as they come in, giving us type safety when we access any values. Each collection will have its own HashMap representing the collection locally. This of course also means that we cannot use any of minimongos API but rather have to resort to Java functions for searching and sorting result sets.

Unfortunately Androids virtual machine does not fully support features that were

[2] `http://developer.android.com/design/material/index.html`
[3] `https://github.com/delight-im/Android-DDP`
[4] `https://github.com/firebase/TubeSock`

added in Java 8 like lambdas and streams which would make our life easier when having to deal with searching through an HashMap. There are ways to retrofit such functionality to be supported with the Android build tools but ultimately its up to Google to add support for that functionality.

To actually do anything with the DDP library all we have to do is create a class that implements the `MeteorCallback` interface. That interface supplies us with several callbacks that are executed as DDP messages come in. For example `onDataAdded` is called when any new data is added to any subscribed collection or `onDataChanged` when any of the subscribed data changes.

From there we need to remember all the values in local memory using HashMaps. In the Android adapters which are responsible to fill the UI with data we then access the data from the HashMaps. In order to receive a notification when the data in the HashMaps changes we need to register the adapter with the local DDP-callback class. The callback class will then notify the adapter in chase any of the requested data changes so that the adapter can update the UI/view.

3.4.2 Connecting Android Wear

We want to be able to transmit data between the app running on the phone and the smartwatch running its own app. Android Wear does not supply the ability to directly access the internet from the smartwatch. Rather one needs a dedicated companion app that then access any API over the internet. This makes sense as a user could have multiple wearable apps which access the same data that should be cached on the phone. Listing 3.5 shows how the data layer api is not limited to Bluetooth but can also work on wearables that directly connect to the internet over Wireless Local Area Network (WLAN).

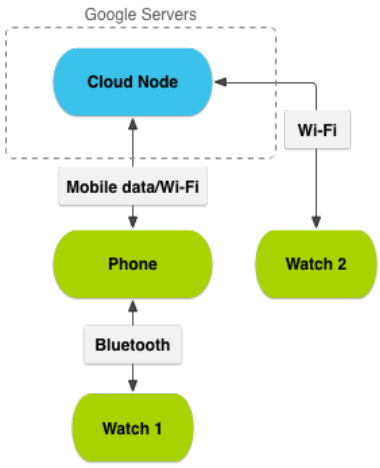

Figure 3.5: How data travels between the smartphone and smartwatch[10]

When transmitting data over the Data Layer API, one actually does not have to worry about any synchronisation issues, but unfortunately Google Play Services are required to be able to communicate with the watch. Similar to the DDP client one only has to implement some connection callbacks that get called once data arrives.

When you setup your project correctly you will use a library project that contains

common classes. Essentially we have to develop two separate applications as Android Wear does not have all of the same APIs of Android. In Android Studio we solve this by using two modules: one for the android smartphone application and one for the smartwatch application. Both modules include a common library module that contains all functionality that is shared between the two. The build tools then take care of merging resources, manifests and any classes.

3.4.3 Bluetooth low energy

In order to be able to communicate with the Vibeband we need to able to connect to a BLE Generic Attribute Profile (GATT) server. The GATT server defines in its configuration file what values can be read and written. These values are called characteristics and are bundled inside services. Listing 3.16 shows a portion of the current configuration file for the Vibeband BLE chip.

Listing 3.16: GATT configuration file

```
<service uuid="1ef88e1f-6745-4007-8742-77943cab8096" advertise="true">
    <description>Vibeband Communication</description>
    <characteristic uuid="dd6d5c32-d900-447d-ac56-5103d0523da6" id="gesture">
        <description>Gesture Data</description>
        <properties read="true" notify="true" />
        <value variable_length="true" length="1" type="user" />
    </characteristic>
    <characteristic uuid="502c1f71-c86e-4738-9882-972aa643e3f1" id="score">
        <description>Scores</description>
        <properties write="true" notify="true" />
        <value variable_length="true" length="11" type="user" />
    </characteristic>
</service>
```

We can see that we have two characteristics:

- **Gesture Data**: triggers if a new gesture was detected on the Vibeband
- **Scores**: allows us to send the current score to be displayed

This GATT file is consumed by the connecting client which in our case is the app on the Android smartphone. In contrast to normal Bluetooth connections there is no pairing process between the server and client. As soon as the chip on the Vibeband powers up it starts advertising its services and allows one client to connect without requiring any kind of authentication. In our application we first scan for any advertising BLE devices and let the user select the correct device.

In the next step we connect to the GATT server with the saved address in an Android service, so that the connection is persistent between Activities. Our service automatically subscribes to any characteristic that offer notifications. Notifications in BLE trigger a callback in our app once the value changes which is perfect for our use case of transmitting the detected gesture.

Anytime the game score changes we transmit that score to the Vibeband immediately. The Vibeband then displays that information and provides the user with a vibrating feedback.

The BLE APIs in Android are a very low level approach to transmit data. The API does not serialize requests on its own but it is the developers responsibility to make sure only one request is executed at any time. During development we encountered many instabilities as well as random disconnects if executing requests too quickly. The reliability of BLE will have to be improved before the Vibeband can be commercially sold on the market.

4 Evaluation

In retrospective we were able to implement a full size application within just a few months that is comparable to Fanmode's product. Using state of the art technology we could not only reduce the overall complexity of the full stack, but were able to maintain the real time aspect.

From the start it was clear that collaboration with the team in College Station and Fanmode would be decisive for the project to be a success. When I first met with the team, I got introduced to the scope of the project and how they are organized. Weekly meetings were planed in the beginning but as all of the team members saw each other in classes daily, there was no real need for meetings. We managed to order development hardware pretty quickly, which would match the chips used on the final board.

We assigned specific roles to each of the team members according to their area of expertise. Kyle has planned out complex schematics before, so it was only logical to make him the hardware engineer. Chelby gathered management experience in a previous professional job and volunteered to be the team leader. Embedded software was one of Joanas strong suits, while Michael was assigned the systems integration engineer, that works to merge software and hardware together and also being responsible for the final functional testing.

Communication inside the team was rough at times, as we had many cultures and characters clash with each other. We could overcome these differences though, in order to focus on what was important to finish the project. We used a group chat for communication, as well as a email distribution list to keep everybody in the loop about what is going on. Also we made use of Git and Google Drive in order to synchronize code and documents that we all worked on. We also made use of a shared calender to make sure everyone knows when and where we have meetings.

Support from the department of Electronic Systems Engineering Technology (ESET) and Prof. Dr. Ben Zoghi specifically was continuous and amazing. He helped the team to make the right decisions for the project while providing me with a great environment to work in.

5 Conclusion

In summary, we achieved our goal of connecting a wearable device to a real time network using the Meteor ecosystem. We reached comparable results to Fanmode's current implementation and in the future could develop the application further into a product for the US sports market.

We have proven that Meteor in conjunction with AngularJS is a mature enough solution for a modern and complex web application. We were able to show that websockets are able to provide real time functionality to any website that wants to improve their user experience significantly. For the application to be used on a large scale we need to refactor our code further, but are also dependent on Meteor to continue improving livequery and other components of their stack.

Other studies of Meteor have found similar results[2], that Meteor can be used as a full stack framework already and is fit for use in production. Formerly complex systems with many different components can now be replaced by a solution, that provides us with the essentials out of the box, reducing the development effort needed to achieve similar results. Problems encountered with performance can be put off and handled once encountered at a later development stage.

In implementing BLE, Android Wear and DDP communication in our Android application we have proven that Meteor is not only an closed off product, but can be interacted with from other platforms and programming languages. In collaboration with the students from A&M we have successfully established a method to integrate an embedded device into our full stack solution.

6 Future Work

In order to develop the proof of concept into a fully fledged product there are several things that need to be considered before this application can be use in production.

6.1 Real time sports data

For one, the information about the games statistics currently is not received in real time yet. For this we would need to hook into a commercial providers stream of data. Fanmode relies on Opta[1] and Sportradar[2] for this kind of data. The vendors push any updates to Fanmode's application server, which then takes care that all clients update accordingly. For better feedback to the user this data should be be synced more often than every couple hours.

6.2 Security considerations with Meteor

Currently the `autopublish` package is still used, in order to enable rapid development of the application. To make this application useful outside a lab environment we have to use the publish/subscribe pattern that Meteor ships with.

Initially to help newer developers understand the basic concepts in Meteor the server simply pushes all data from the server's MongoDB to the client's minimongo database. This works on local setups on a developers machine, but is not practical on a production server. Not only would we leak information about everything we save, but also this approach does not scale with a growing user base. The amount of information transferred via DDP to the clients would be several mega bytes each with the data we have stored in the current development environment.

In order to make the system secure and have good performance across the Internet we will need to setup publications for any data that we want to use on the client. We have explained in chapter 3.2.5 how these work, but as we are still in development we have not yet made use of them everywhere.

[1] `http://www.optasports.com/en.aspx`
[2] `https://www.sportradar.com/`

6.3 Deployment

Up to this point we have run the Meteor application only locally on the command line. If we want to use the application over the internet we need to deploy it to an application server. Meteor offers a simple command `meteor deploy <appname>.meteor.com` to deploy the current application to what they call Galaxy, which is basically a hosting service for Meteor application. This however might not be optimal, as we only have very little control on who has access to the data and it only supplies limited resources to the system. This service is free of charge, but also hibernates applications that are not being used, leading to a significant timeout to restart once accessed. There are paid plans[3] that enable more resources and bigger applications but we can also deploy the application to our own server.

There are several ways to deploy a Meteor application to your own server and several tools like Meteor Up[4] that make it a lot easier to go through this process. Basically one needs to first install all necessary software like NodeJS, MongoDB and any webserver. The webserver for example nginx acts as a local proxy that accepts all connections and passes them on to the NodeJS server. Although this not required, one might want to consider having a Secure Sockets Layer (SSL) connection instead of plain HTTP, in order to prevent eavesdropping on the delivered content.

After having setup all the necessary tools on the server one can build the application using the `meteor build .` command which will generate an archive, that can be unpacked on the server. Afterwards one should be able to normally start the application, how one would traditionally start any NodeJS application. Meteor Up simplifies this process and reduces it to a single command for deployment.

6.4 Scalability of the full stack

There are still several areas that might be a bottleneck when we talk about a million concurrent users. The way livequery currently is implemented will certainly impact the responsiveness of the overall application on a larger scale. Neither MongoDB nor NodeJS are known to have scalability issues, but with all the added functionality further research will be required to determine on just how scalable the overall Meteor platform really is.

[3] `https://www.meteor.com/why-meteor/pricing`
[4] `https://github.com/arunoda/meteor-up`

6.5 Vision for the Vibeband

To reach a truly large audience one would have to first reach out to season ticket holders. Incentivising customers to purchase a Vibeband and then use the system for one season to test-drive it would be straightforward. For instance, we could use the Vibeband with its NFC functionality as a ticket to access the stadium on game day, reducing queues in front of ticket offices. With on-board Wi-Fi Vibeband would connect to the stadium's Wi-Fi network as soon as fans enter the premises, so fans without a smartphone would be included in the enhanced viewing experience. One could even go as far as allowing customers to pay for snacks and drinks with the Vibeband and process the payment with a linked credit card.

If the Vibeband can be produced at low cost, we could even consider including it in the season ticket altogether. It is questionable whether customer paying several hundred dollars for a season ticket would pay extra for the Vibeband. They would be more inclined to use it if it comes as part of the ticket package. Rental would also be a possibility where the customer deposits a security which is credited back upon returning the device.

As this would be an all around solution providing fans with more comfort while collecting their feedback from gestures this would benefit both sides. A collaboration between A&M's Kyle field and Fanmode could be a first step to get this product started within the American Football league. A thorough market analysis would be needed though to estimate true market viability and whether customers are open to use this technology for the added convenience.

Glossary

app App is short for application, which is a synonym for a software program. While an app may refer to a program for any hardware platform, it is most often used to describe programs for mobile devices, such as smartphones and tablets[6] .

application An application, or application program, is a software program that runs on a computer[7].

Java A programming language that is strongly object-oriented and can be executed on many different platforms. .

jQuery jQuery is a comprehensive JS library helping developers with cross browser compatibility and manipulating the HTML DOM tree. It can take care of animating certain website elements and provides a excellent plug-in system to extend its functionality..

Meteor Meteor is a software stack implemented on top of NodeJS to simplify the development of modern web applications.

minimongo Minimongo is a reimplementation of (almost) the entire MongoDB API, against an in-memory JavaScript database. It is like a MongoDB emulator that runs inside your web browser. You can insert data into it and search, sort, and update that data[18] .

MongoDB MongoDB is an document oriented database that is designed for ease of development and scaling .

MVC MVC is an abbreviation for Model View Controller and is a concept often found in applications to divide classes into clear responsibilities. It helps separate application logic from code that displays the user interface. .

nginx Nginx is a web- as well as a proxy server that is focused on concurrency and performance .

NodeJS Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient[20] .

NPM Node Package Manager (NPM) is a command line tool to manage packages for NodeJS but also in any other project that uses JS packages .

SCRUM A new approach to rapidly develop software in an iterative work flow. Is considered to be a agile development method where the requirements change during the course of development.

spaghetti code Code that is highly complex because of a tangled control structure .

Unix Unix is a family of operating systems that are modular and share basic programs .

web application A web application or "web app" is a software program that runs on a web server. Unlike traditional desktop applications, which are launched by your operating system, web apps must be accessed through a web browser[8] .

Wi-Fi A technology that enables users to access a network over a wireless connection .

Acronyms

AJAX	Asynchronous JavaScript and XML.
API	Application Programming Interface.
BLE	Bluetooth Low Energy.
CRUD	Create, Read, Update and Delete.
DDP	Distributed Data Protocol.
DI	Dependency Injection.
DOM	Document Object Model.
EJSON	Extended JSON.
GATT	Generic Attribute Profile.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol Overview.
I/O	Input Output.
IDE	Integrated Development Environment.
IE	Internet Explorer.
JS	JavaScript.
JSON	JavaScript Object Notation.
NFC	Near Field Communication.
OLED	Organic Light-Emitting Diode.
PHP	PHP. Hypertext Preprocessor.

REST Representational State Transfer.

SaaS Software as a Service.
SQL Structured Query Language.
SSL Secure Sockets Layer.

TCP Transmission Control Protocol.
TUM Technische Universität München.

UI User Interface.
URL Uniform Resource Locator.

W3C World Wide Web Consortium.
WLAN Wireless Local Area Network.

XHR XMLHttpRequest.

List of Figures

Listings

Bibliography

[1] S. Agarwal. “Real-time web application roadblock: Performance penalty of HTML sockets.” In: *IEEE International Conference on Communications* (2012), pp. 1225–1229. ISSN: 15503607. DOI: 10.1109/ICC.2012.6364271.

[2] Y. Akkijyrkka. “Dynamic Web Applications with Meteor.js.” PhD thesis. 2015. URL: https://publications.theseus.fi/bitstream/handle/10024/102032/Dynamic%20web%20applications%20with%20meteor%20Yrkko%20Akkijyrkka.pdf.

[3] *Android 5.0 Lollipop brings BLE Improvements*. 2015. URL: http://www.argenox.com/blog/android-5-0-lollipop-brings-ble-improvements/ (visited on 02/05/2016).

[4] Apple. *Set up and use Apple Pay with your Apple Watch*. 2015. URL: https://support.apple.com/en-us/HT204506 (visited on 01/20/2016).

[5] *Atomosphere 9000 packages*. 2015. URL: https://twitter.com/atmospherejs/status/676881716565250048 (visited on 02/01/2016).

[6] P. Christensson. *Definition App*. 2012. URL: http://techterms.com/definition/app (visited on 01/14/2016).

[7] P. Christensson. *Definition Application*. 2008. URL: http://techterms.com/definition/application (visited on 01/14/2016).

[8] P. Christensson. *Definition Web application*. 2014. URL: http://techterms.com/definition/web%7B%5C_%7Dapplication (visited on 01/15/2016).

[9] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. “The many faces of publish/subscribe.” In: *ACM Computing Surveys* 35.2 (2003), pp. 114–131. ISSN: 03600300. DOI: 10.1145/857076.857078. URL: http://portal.acm.org/citation.cfm?doid=857076.857078.

[10] Google. *Android Data Layer*. URL: http://developer.android.com/training/wearables/data-layer/index.html (visited on 02/06/2016).

[11] H. Hämäläinen. “HTML5 : WebSockets.” In: *Communication* (2011), pp. 1–9.

[12] I. Hickson, Ian (Google. "HTML5: A vocabulary and associated APIs for HTML and XHTML." In: *W3C Working Draft* 2010 (2011), pp. 1–599. URL: http://www.w3.org/TR/html5/$%5Cbackslash$nhttps://github.com/w3c/html.

[13] IDC. *Smartphone OS Market Share, 2015 Q2*. 2015. URL: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (visited on 02/05/2016).

[14] E. International. "ECMA-262 ECMAScript Language Specification." In: *JavaScript Specification* 16.June (2009), pp. 1–252. ISSN: 09544879. URL: http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[15] Jesse James Garrett. *Ajax: A New Approach to Web Applications*. 2005. URL: https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php (visited on 01/20/2016).

[16] *Meteor platform technology*. 2016. URL: https://www.meteor.com/why-meteor/technology (visited on 02/07/2016).

[17] *Meteor Slogan*. 2015. URL: https://www.meteor.com/ (visited on 01/15/2016).

[18] *Mini Databases*. URL: https://www.meteor.com/mini-databases (visited on 01/28/2016).

[19] G. (n.d.) *Umsatz mit Software-as-a-Service (SaaS) weltweit von 2010 bis 2016 (in Milliarden US-Dollar)*. 2016. URL: http://de.statista.com/statistik/daten/studie/194117/umfrage/umsatz-mit-software-as-a-service-weltweit-seit-2010/.

[20] *Nodejs description*. 2016. URL: https://nodejs.org/en/ (visited on 02/04/2016).

[21] G. Palshikar. "Simple algorithms for peak detection in time-series." In: *Proc. 1st Int. Conf. Advanced Data Analysis, ...* January (2009).

[22] V. Pimentel and B. G. Nickerson. "Communicating and displaying real-time data with WebSocket." In: *IEEE Internet Computing* 16.4 (2012), pp. 45–53. ISSN: 10897801. DOI: 10.1109/MIC.2012.64.

[23] A. Russell. *Android Pay On Android Wear*. 2015. URL: https://wtvox.com/wearables/android-wear/android-pay-on-android-wear/ (visited on 01/20/2016).

[24] S. Stubailo. *DDP Specification*. 2014. URL: https://github.com/meteor/meteor/blob/master/packages/ddp/DDP.md (visited on 01/20/2016).

[25] S. Stubailo. *Optimistic UI with Meteor*. 2015. URL: http://info.meteor.com/blog/optimistic-ui-with-meteor-latency-compensation (visited on 02/07/2016).

[26] M. Ubl and K. Eiji. *Introducing WebSockets: Bringing Sockets to the Web - HTML5 Rocks*. 2010. URL: http://www.html5rocks.com/en/tutorials/websockets/basics/.