Technische Konzeption für eine komponentenreiche Produktpräsentation

Bachelorarbeit zur Erlangung des akademischen Grades

„Bachelor of Engineering“

an der Hochschule der Medien Stuttgart
Fakultät Electronic Media
Studiengang Audiovisuelle Medien Bachelor (AMB)

ICT Innovative Communication Technologies AG Kohlberg

Bearbeitungszeitraum: 6. Januar 2015 – 17. Mai 2015

vorgelegt von

Jan Fiess
In den Schneckengärten 6
74343 Sachsenheim

Tel.: 0176 708 656 41
Mail: ff023@hdm-stuttgart.de
Matrikel-Nr.: 23957

Prüfer und Betreuer

Prof. Dr. Simon Wiest (HdM Stuttgart)
Prof. Ursula Drees (HdM Stuttgart)
Manfred Dolde (ICT AG)

Technische Konzeption für eine komponentenreiche Produktpräsentation

Abstract

Digitale Medien haben Messen und Events revolutioniert. Displays und Videotechnik sind inzwischen Standardausstattung von Messepräsenzen. Mit der Intention, den Besuchern aufzufallen, folgen viele Unternehmen dem Trend, mittels Opulenz und teilweise aggressiver audiovisueller Reizüberflutung die Aufmerksamkeit der Besucher auf sich zu lenken.
Diese Arbeit beschäftigt sich mit der Realisierung einer erlebnisreichen Messepräsenz, die sich Ansätze der Wahrnehmungspsychologie zunutze macht, bei denen Messebesucher effektiv in ein Rundumerlebnis integriert werden: Sie können mit Messeexponaten individuell interagieren, erhalten Feedback, erleben Emotionen und behalten das beworbene Produkt im Gedächtnis.

In dieser Thesis wird auf inhaltlicher und technischer Ebene eine erlebnisreiche Produktpräsentation skizziert, in der das beworbene Produkt und die präsentierende Medientechnik optimal miteinander verzahnt werden.
Die technische Realisierung orientiert sich am inzwischen allgegenwärtigen *Internet der Dinge*, bei dem physische Gegenstände aus der Umwelt mit Computersystemen vernetzt werden.

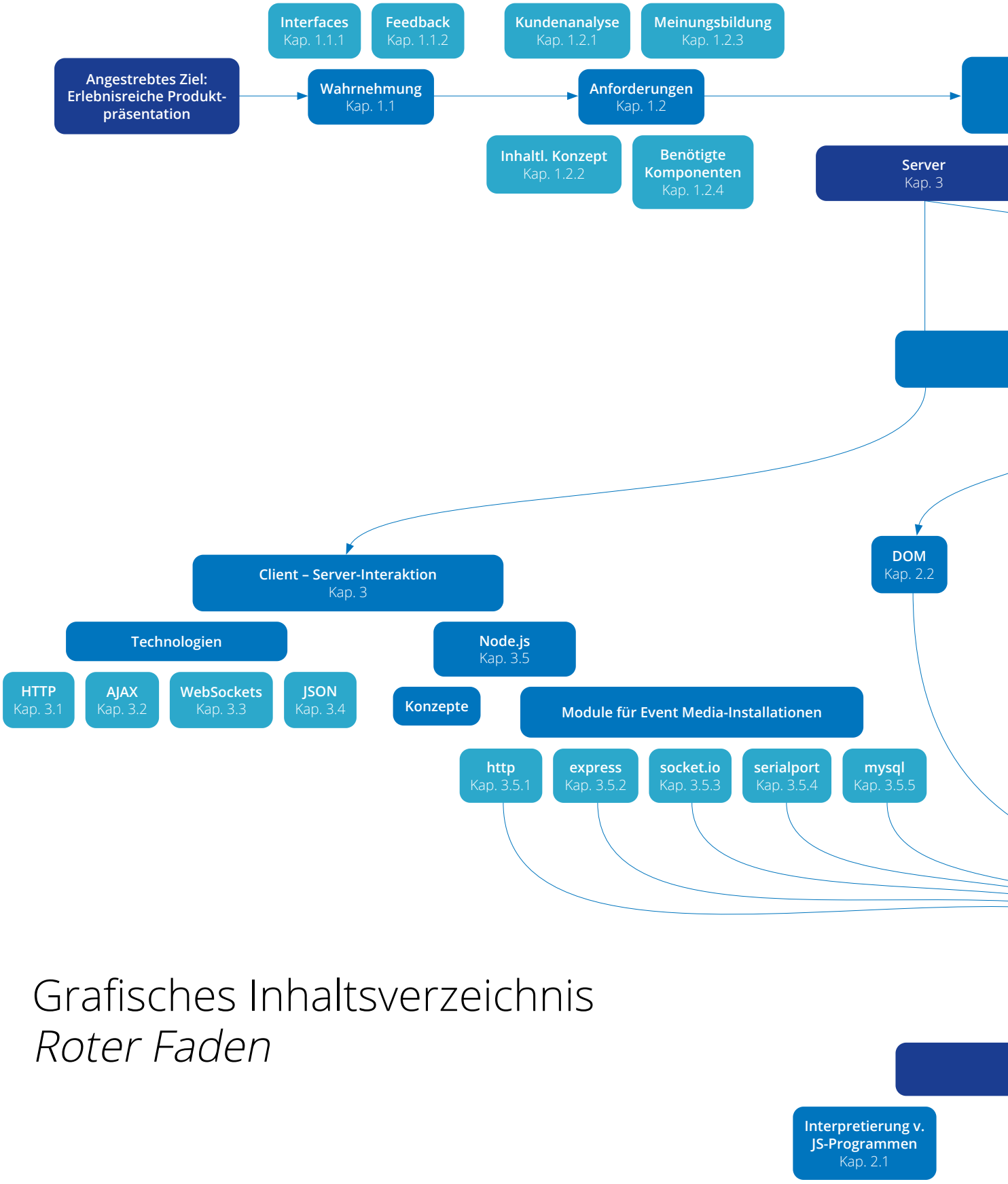
Die Umsetzung erfolgt einheitlich mit der Programmiersprache JavaScript, die, im Hinblick auf die Programmierung der einzelnen Elemente einer komponentenrechen Medieninstallation, expermentell nach Eignung untersucht und mit anderen bewährten Technologien verglichen wird.
Es entsteht ein Standardaufbau für eine komponentenreiche Produktpräsentation, der flexibel an unterschiedliche Kundenbedürfnisse angepasst werden kann.

English Abstract

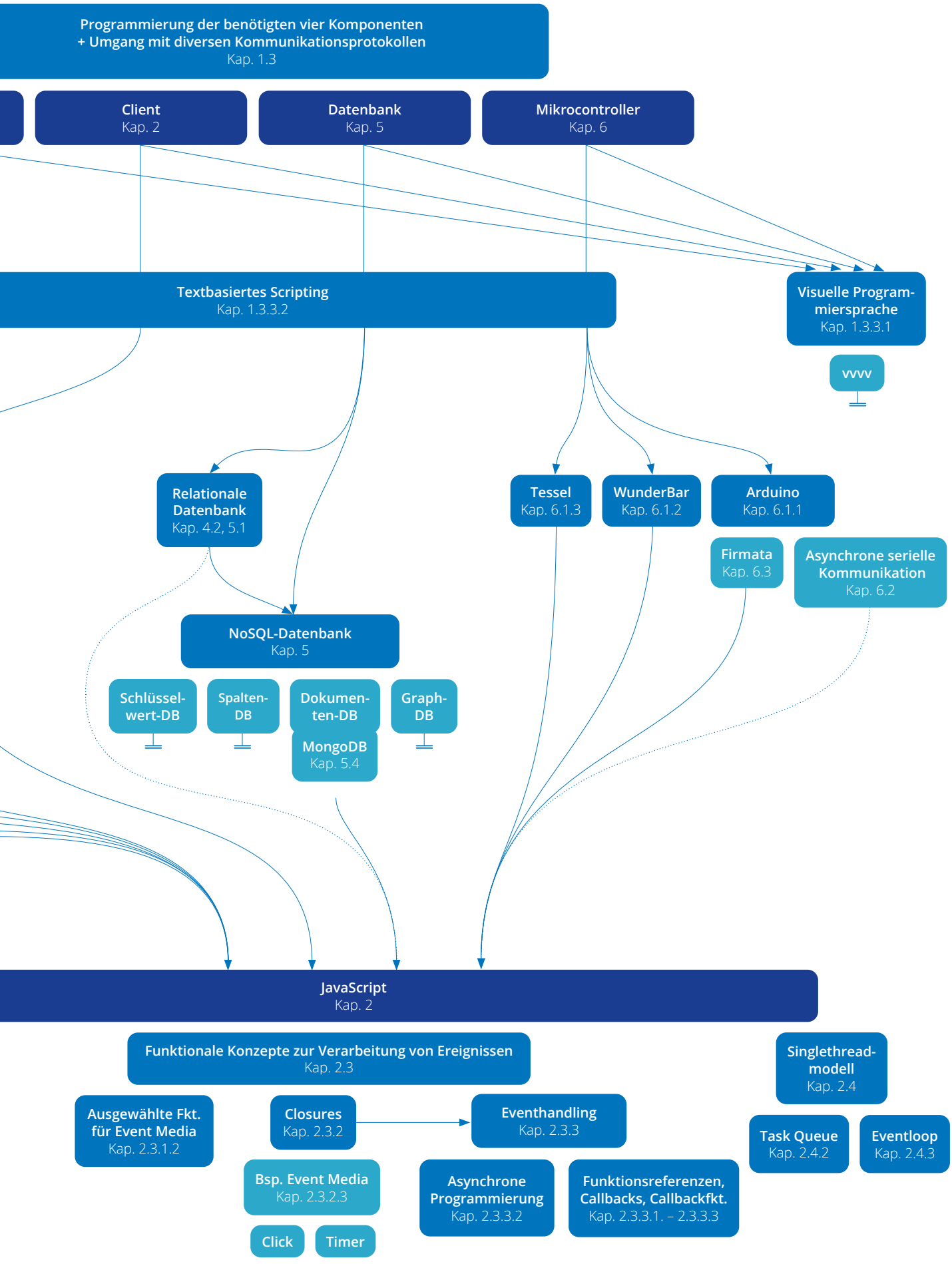
Digital media has revolutionized trade fairs, exhibitions and events. Displays and video technology have become standard equipment to highlight trade show presence. With the intention to catch the visitors' eye many companies follow the trend to draw the visitors' attention to their exhibition booth with an opulent and partly aggressive audiovisual overstimulation.

This thesis is about the implementation of a striking (eventful) trade show appearance which takes advantage of (od. benefits from) the approaches of psychology of perception whereby trade fair visitors are most effectively and immediately integrated in an outright experience: they can interact individually with show exhibits, get immediate feedback, can experience many emotions and bear in mind the advertised product.
Regarding content and technology this thesis outlines an eventful product presentation which interlocks optimally the advertised product and the presenting media technology. The technical realization deals with the by now omnipresent „internet of things" which links physical everyday objects with virtual computer systems.

This can be realized with the programming language JavaScript, which, in this thesis, is examined experimentally depending on suitability and compared with other approved technologies, as far as the programming of the single elements of a multi-component media installation is concerned.
The result is a standard structure for a multi-component product presentation which can be adapted flexibly to customer needs and requirements.



Grafisches Inhaltsverzeichnis
Roter Faden


Programmierung der benötigten vier Komponenten
+ Umgang mit diversen Kommunikationsprotokollen
Kap. 1.3
Client
Kap. 2
Datenbank
Kap. 5
Mikrocontroller
Kap. 6
Textbasiertes Scripting
Kap. 1.3.3.2
Visuelle Program-
miersprache
Kap. 1.3.3.1
vvvv
Relationale
Datenbank
Kap. 4.2, 5.1
Tessel
Kap. 6.1.3
WunderBar
Kap. 6.1.2
Arduino
Kap. 6.1.1
Firmata
Kap. 6.3
Asynchrone serielle
Kommunikation
Kap. 6.2
NoSQL-Datenbank
Kap. 5
Schlüssel-
wert-DB
Spalten-
DB
Dokumen-
ten-DB
Graph-
DB
MongoDB
Kap. 5.4
JavaScript
Kap. 2
Funktionale Konzepte zur Verarbeitung von Ereignissen
Kap. 2.3
Singlethread-
modell
Kap. 2.4
Ausgewählte Fkt.
für Event Media
Kap. 2.3.1.2
Closures
Kap. 2.3.2
Eventhandling
Kap. 2.3.3
Task Queue
Kap. 2.4.2
Eventloop
Kap. 2.4.3
Bsp. Event Media
Kap. 2.3.2.3
Asynchrone
Programmierung
Kap. 2.3.3.2
Funktionsreferenzen,
Callbacks, Callbackfkt.
Kap. 2.3.3.1. – 2.3.3.3
Click
Timer

Inhaltsverzeichnis

Rolle der ICT AG

Diese Arbeit wird in Kooperation mit dem Medientechnikunternehmen ICT Innovative Communications Technologies durchgeführt.
ICT ist ein Dienstleiter für die technische Realisierung von Medieninstallationen aller Art.
Typische Arbeitsbereiche bei ICT sind:

- Verleih von Veranstaltungs- und Medientechnik
- Technisches Consulting
- Support vor Ort
- Medienprogrammierung und Steuerung
- Projektmanagement
- Softwareentwicklung

ICT ist u. a. Spezialist für die technische Umsetzung von Messepräsenzen.
Bei konzeptionell aufwändigen Projekten, beispielsweise zum Anlass der Erstpräsentation eines Automobils, wird der kreative Teil von einer Kreativagentur übernommen, die den Auftrag zur technischen Realisierung an ICT vergibt.
Für kleine und mittelständische Unternehmen möchte ICT kostengünstigere Möglichkeiten anbieten, für die eine Kreativagentur nicht unbedingt nötig ist.
ICT bemüht sich, technologisch am Puls der Zeit zu bleiben und sucht mittels Innovationen ständig nach Differenzierungsmöglichkeiten zu anderen Wettbewerbern.

Kap. 1: Allgem. konzeptionelle Entwicklung einer erlebnisreichen Produktpräsentation

Dieses Kapitel beschäftigt sich mit der inhaltlichen Konzeption von erlebnisreichen Messexponaten für mittelständische Unternehmen mit begrenztem Budget. Dabei soll das Publikum mit Hilfe von intelligenten Messeapplikationen ins Geschehen einbezogen werden. Es soll mit mehreren Sinnen erreicht und mit einem Rundumerlebnis emotional angesprochen werden. Besucher sollen auf trickreiche Weise u. a. dazu aufgefordert werden, Messeexponate ggf. interaktiv zu beeinflussen. Diese antworten mit einem *Feedback*. Solche Messeexponate fallen nicht *ausschließlich* durch ihre Größe auf, sondern eher durch ihre *Vielseitigkeit* und *Innovation*.

Zunächst wird die Wahrnehmung eines Menschen im Umgang mit modernen Computersystemen untersucht. Dazu werde ich auf aktuelle Erkenntnisse aus der Wahrnehmungsforschung in Verbindung mit modernen User Interfaces eingehen.
Mit einer Kundenanalyse werden konkrete Anforderungen bei der Realisierung einer erlebnisreichen Produktpräsentation definiert, die der inhaltlichen Konzeption von Projektvorschlägen an diese Kunden dienen.
Eine Umfrage liefert ein allgemeines Meinungsbild hierfür.
Im Anschluss folgt die technische Konzeption, in der die Komponenten eines Standardaufbaus und eine geeignete Programmiermethode festgelegt werden.

1.1. Wahrnehmung auf Messen

Im Folgenden wird untersucht, welche Gestalt ein Messeexponat annehmen sollte, um das Publikum möglichst wirkungsvoll zu erreichen.
Das Messeexponat sollte sich von den Exponaten benachbarter Mitbewerber auf einer Messe abheben und vom Publikum bewusst wahrgenommen werden. Dabei soll Begeisterung ausgelöst werden und eine emotionale Bindung entstehen.
Die Individuelle Interaktion des Publikums mit Exponat über ein *Interface* sowie das gezielte *Feedback* spielen für die menschliche Wahrnehmung eine große Rolle.

1.1.1. Interfaces[1]

Eine wirkungsvolle Möglichkeit, die Aufmerksamkeit des Publikums zu gewinnen, besteht darin, es am Geschehen interaktiv teilhaben zu lassen.
Es soll dazu motiviert werden, sich mit einem Exponat *möglichst lange* zu beschäftigen. In individuell dosierten Häppchen sollen Informationen an das Publikum übermittelt werden. Dazu muss das Publikum über ein *Interface* Einfluss auf einen Computer nehmen.

WIMP-Interfaces

WIMP[2]-Interfaces stellen das seit der Einführung des Macintosh im Jahr 1984 „derzeit dominierende Grundkonzept moderner grafischer Benutzerschnittstellen (GUIs)" dar [Wikipedia, *WIMP*]. Ein klassisches Beispiel bei der Interaktion mit einem WIMP-Interface ist das alltägliche Arbeiten an einem Computer, indem sich der Nutzer an visuellen Elementen auf einem Display orientiert und diese vorwiegend mit einer Maus interaktiv beeinflusst. Dabei werden hinter der grafischen Benutzeroberfläche Aktionen ausgelöst.

Post-WIMP-Interfaces

Aus der Weiterentwicklung der herkömmlichen *WIMP*-Benutzerschnittstellen gehen *Post-WIMP*-Interfaces hervor. Post-WIMP-Interfaces sollen herkömmliche Interfaces durch realitätsbezogenere Interfaces ersetzen. Sie orientieren sich an gewohnten alltäglichen Interaktionen aus der realen, nicht-digitalen Welt. Für die Kommunikation mit einem virtuellen Computersystem sollen zunehmend natürliche, physische Gesten eingesetzt werden.
„Post-WIMP-interfaces [...] draw strength by building on users' pre-existing knowledge of the everyday, non-digital world to a much greater extent than before. They employ themes of reality such as users' understanding of naive physics, their own bodies, the surrounding environment and other people." [Jacob *et al.*]
Beispiele für Post-WIMP-Interfaces sind berührungsempfindliche mobile Endgeräte, digitale Whiteboards, Spracheingabe und Gesten, die mithilfe von Geräten wie Kinect- oder Leap Motion- Tiefenkamera erfasst werden.

Die Arbeitsgruppe Mensch-Computer-Interaktion (MCI) der Universität Konstanz beschäftigt sich mit der Entwicklung solcher realitätsbezogeneren Interfaces. Sie untersucht den Zusammenhang zwischen verschiedenartigen Interaktionsmöglichkeiten und der Wahrnehmung des Menschen.
Dieser Ansatz dient der inhaltlichen Konzeption einer effektiven Messepräsenz.
Die Arbeitsgruppe MCI bemüht sich, „möglichst umfassend alle Sinnesorgane sowie unsere körperlichen Fähigkeiten [...] bei der Benutzung von Computersystemen" zu nutzen
Die Entwicklung realitätsbezogenerer User Interfaces beruht auf der Erkenntnis, dass „unsere kognitive Entwicklung [...] maßgeblich durch unsere

1 Interfaces: Benutzerschnittstellen zwischen Mensch und Maschine
2 WIMP: Windows, Icons, Menus, Pointer

körperliche und soziale interaktion mit Objekten bzw. Lebewesen der Umwelt beeinflusst [wird]“ [Reiterer].
Die MCI versucht unter dem Stichwort „Blended Interaction“, die reale und die digitale Welt zunehmend miteinander zu vermischen. „The designer´s goal should be to allow the user to perform realistic tasks realistically, to provide additional non real-world functionality, and to use analogies for these commands whenever possible“ [Jacob et al.].
Ein aktuelles Ziel der Interaction Designer der Arbeitsgruppe MCI ist die einface „1:1-Übertragung [von] Realwelt-Metaphern“ [Reiterer] auf ein generatives Computersystem.
Die Einfussmöglichkeiten eines Nutzers auf einen Computer wird somit natürlicher: Der Besucher kann ziehen, schieben, berühren, wischen, biegen, malen, schreien, pusten, stoßen, ziehen, kicken, stampfen, drehen, näher kommen, laufen, klettern, musizieren, spielen, fangen, Gesten anwenden, Gewicht verlagern, fahren, fliegen, usw.

1.1.2. Feedback

Bei der effektiven Wahrnehmung von Exponaten spielen neben Augen und Ohren weitere Sinnesorgane eine immer größere Rolle; zum Teil unbewusst. Besucher können erschaudern, überrascht werden, Düfte riechen, Luft- und Wasserbrisen spüren, Vibrationen wahrnehmen, in Bewegung versetzt werden, die besondere Haptik eines Interfaces fühlen, usw.

Wie ein Messeexponat dem Publikum gegenüber wirken soll, hängt vom jeweiligen Kunden ab. Je nach Intention und Budget des werbenden Unternehmens können Messeexponate durch Interaktionsmöglichkeiten und Feedbacks ergänzt werden. Der Kunde kann beliebig den Informationsgrad oder den Spaßfaktor in den Vordergrund stellen oder beides kombinieren.

Recherchen nach Messeeponaten für kleine und mittelständische Unternehmen ergaben, dass das Messeexponat allgemein unterschiedliche Gestalten annehmen kann.

Die Präsentation kann die Form eines Spiels annehmen, bei dem das Publikum *immersiv*[1] eingebunden wird und mit dem beworbenen Produkt spielt. Der Kunde kann das Spiel allerdings auch durch einen interaktiven *Showcase* ersetzen, bei dem das Pulikum das Produkt auch ohne umgebende Story interaktiv testen kann. Dabei wird es mit Informationen versorgt.
Die Präsentation kann auch komplett auf Interaktionsmöglichkeiten verzichten und verlässt sich ausschließlich auf Feedbacks.
Hinzu kommen einige Variationen mit/ohne Interaktionen mit dem beworbenen Produkt bzw. mit hohem/niedrigem Informationsgrad.
Alle Wirkungsformen sind in der Projektedatenbank (vgl. Kap. 4) detailliert abrufbar und werden an dieser Stelle nicht weiter ausgeführt.

1 Immersion ist das Phänomen im Bewusstsein eines Menschen, sich zunehmend mit einer virtuellen Welt zu identifizieren und voller Ehrgeiz zu agieren, wie er es auch in der physischen Realwelt tun würde. Dieses Phänomen tritt häufig bei Videospielen auf, wenn sich Spieler geistig und emotional in virtuelle Charaktere hineinversetzen und die Realität ausblenden.

1.2. Konkrete Anforderungen

In diesem Abschnitt werden konkrete Anforderungen an eine erlebnisreiche Produktpräsentation entwickelt. Nach einer Kundenanalyse wird ein inhaltliches Schema erstellt, an dem sich die Kunden je nach Intention und Budget orientieren können. Im Anschluss wird die Meinung des Publikumseingeholt. Schließlich werden die technischen Komponenten grob zusammengestellt.

1.2.1. Kundenanalyse

Potenzielle Interessenten an einer erlebnisreichen Produktpräsentation von ICT sind überwiegend Unternehmen im Bereich Maschinenbau und Elektrotechnik.
Sie haben gemeinsam, dass sie greifbare und funktionale Produkte vermarkten.

- ZF Friedrichshafen: U. a. Antriebstechnik (z. B. Getriebe, Lenksysteme, Achsen, Kupplungen, Windkraft, ...)
- SEW Eurodrive: Antriebsautomatisierung (z. B. elektronisch geregelte Antriebe, Getriebe, Motoren, Servotechnik, ...)
- Krones: Getränkeabfüllanlagen von der Flaschenproduktion über Abfüllung bis zum Recycling
- Festo: U. a. pneumatische[1] Antriebe (z. B. Ventile, elektrische, Vakuumtechnik, Greifsysteme, ...), Bildverarbeitungssysteme
- ABB: Elektrotechnik-Produkte und Leistungselektronik (z. B. zur Strom- und Wärmeerzeugung, Stromübertragung, Wasserwirtschaft, ...)
- Phönix Contact: U. a. Interface- und Automatisierungstechnik, Datenübertragungstechnik, ...
- Bosch Thermotechnik (mit den Marken Junkers, Buderus): Heizungssysteme (z. B. Wärmepumpen, Solarheizungen, Wohnraumlüftungen, ...)
- ABP Inductions: U. a. Induktionshochöfen, Ansaugungstechnik
- Sprimag: U. a. Beschichtungs- und Lackieranlagen
- Grohe: Sanitärprodukte (z. B. Waschbecken, Armaturen, Duschköpfe, ...)
- Trelleborg: Gummispezialist (z. B. robuste Reifen für u. a. Traktoren, medizinische Anwendungen, ...)
- A. Lange & Söhne: Luxusuhren
- Adidas: Sportartikel

Analysekriterien

Diese Unternehmen wurden nach bestimmten Kriterie tabebellarisch untersucht, u. a. nach genutzten Werbeforen (z. B. Messen), Marketing, Prioritäten, Offenheit für Innovationen, Sinn für Ästhetik, Budget, Betreuung durch Agenturen, soziale Interessen und Chancen für ICT.
Besonders zielführend war die Begutachtung früherer Messepräsenzen.

1 Pneumatik: Verrichtung von mechanischer Arbeit mithilfe von Druckluft

Über ggf. beteiligte Kreativagenturen bei früheren Messepräsenzen verraten die meisten werbenden Unternehmen selbst wenig. Kreativagenturen hingegen schmücken ihr Profil mit den Projekten ihrer Kunden. Es musste bei den Kreativagenturen *direkt* gesucht werden. Um diese ausfindig zu machen, suchte ich erfolgreich in den Archiven renommierter Kreativwettbewerbe[1]. Schließlich sammelten sich Informationen über frühere Präsentationen der werbenden Unternehmen und beteiligten Kreativagenturen aus unterschiedlichen Quellen an: Internetpräsenzen einiger Kreativagenturen, Projektportfolio von ICT, Blogs[2], Literatur[3], usw.

Fazit der Kundenanalyse

Es fällt auf, dass das Potenzial digitaler Medientechnik auf den Messepräsenzen der oben genannten Unternehmen auf Messen und Ausstellungen nicht optimal ausgeschöpft wird: Die präsentierende Medientechnik und das beworbene Produkt bilden in allen Fällen separate Einheiten – ohne Bezug zueinander. Beispielsweise dient eine Displayfassade als passive Videokulisse oder zur Darstellung von Informationen, ohne einen tatsächlichen Bezug auf das beworbene Produkt als physisch vorhandenes Objekt zu nehmen.

In den wenigen Beispielen, in denen das Publikum die Möglichkeit hat, interaktiv am Geschehen teilzuhaben, ist die Interaktion auf den Bereich der Medientechnik begrenzt, während das beworbene Produkt eher die Rolle eines Acessoires einnimmt.
Das Publikum interagiert nicht mit dem Produkt selbst. Wenn eine Interaktion vorhanden ist, interagiert das Pubikum mit der Medientechnik und erhält auch das Feedback von der Medientechnik. Die Grenze zwischen dem beworbenen Produkt und der präsentierenden Medientechnik bleibt erhalten. Das kann dazu führen, dass die Aufmerksamkeit eher auf die Medienfassade als auf das tatsächlich beworbene Produkt gelenkt wird.
Es ist zu beachten, dass eine rundum gelungene Produktpräsentation nicht allgemein durch den Einsatz von Medientechnik gegeben ist. Das Publikum sollte mit dem Produkt selbst interagieren, unter Zuhilfenahme der Medientechnik. Produkt und Medientechnik sollen sich in einer Symbiose gegenseitig unterstützen und nicht ausschließen. Sie sollten näher im Dialog zueinander stehen.

Daher folgende Beobachtung:

> *Eine Produktpräsentation ist besonders wirkungsvoll, wenn das beworbene Produkt und die präsentierende Medientechnik grenzenlos ineinander verzahnt werden.*

Im Folgenden werden vier Szenarien untersucht, in denen das beworbene Produkt und die präsentierende Medientechnik eine Einheit bilden.
Der Vollständigkeit zuliebe werden vier weitere Szenarien ergänzt, in denen das beworbene Produkt und die Medientechnik zwar keine Einheit bilden, aber dennoch eine erlebnisreiche Produktpräsentation ergeben.

1 Kreativwettbewerbe: Z. B. CommAward, ADC-Wettbewerb
2 Blogs über Medieninstallationen: z. B. Create Digital Motion, Plusinsight
3 Literatur zu Medieninstallationen: z. B. Prototyping Interfaces, Generative Gestaltung, Touch Of Code

1.2.2. Acht mögliche Präsentationsformen

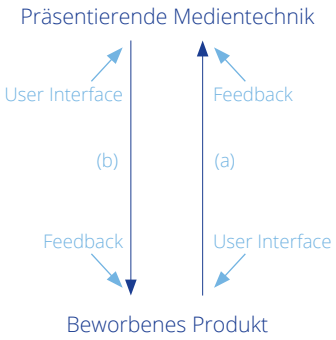


Abb. 1: Veranschaulichung zur Gegenseitigen Beeinflussung des umworbenen Produkts und der präsentierenden Medientechnik

In den ersten beiden Fällen reagieren das beworbene Produkt und die präsentierende Medientechnik aufeinander, wenn das Publikum mit ihnen interagiert (vgl Abb. 1).

- (a:) Das Publikum interagiert direkt mit dem eigentlichen beworbenen Produkt und erhält ein Feedback von den medientechnischen Komponenten der Installation. Das Steuerinterface *ist* das beworbene Produkt selbst, das mit Hilfe von Sensoren Einfluss auf die Medientechnik nehmen kann. Das Publikum wird motiviert, das Produkt zu berühren und zu erleben. Beispiel: Abb. 3.2
- (b:) Das Publikum interagiert mit einer medientechnischen Komponente und erhält ein Feedback vom beworbenen Produkt. Das Steuerinterface ist *nicht* das beworbene Produkt. Die Interaktion mit dem eigentlichen Produkt erfolgt indirekt über eine Medientechnikkomponente, z. B. über ein Tablet, mit Gesten, usw. (mögliche Interfaces: Vgl. Kap. 1.1.1). Beispiel: Abb. 3.1

In den beiden nächsten Fällen sind das beworbene Produkt und die präsentierende Medientechnik zwar ineinander zu einem gemeinsamen Ganzen verzahnt, allerdings beeinflussen sie sich nicht gegenseitig. Das Produkt fungiert als Kulisse, an der sich das multimediale Geschehen abspielt.

- (c:) mit Publikumsinteraktion. Beispiel: Abb. 3.3
- (d:) ohne Publikumsinteraktion. Beispiel: Abb. 3.4

Der Aufwand und die Kosten nehmen von a bis d ab, jedoch auch der Wirkungsgrad der Messepräsenz. Besonders für mittelständische Unternehmen aus den Bereichen Maschinenbau und Elektrotechnik, die sich überwiegend als innovativ darstellen, ist eine Messepräsenz dieser Art denkbar.

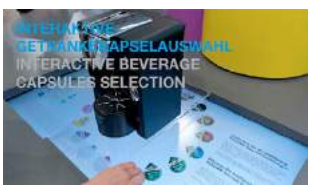


Abb. 3.1: Eine Kaffeemaschine steht auf einem Touchscreen, an dem das Publikum Kaffeekapseln per Touch auswählen kann. Danach wird der entsprechende Kaffee produziert.

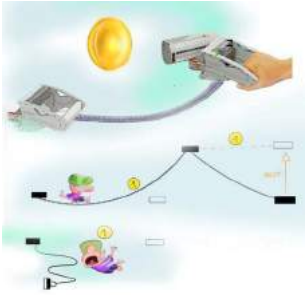


Abb. 3.2: In einem Videospiel werden an einer Videowand mithilfe eines ergonomisch gestalteten Steckers (das beworbene Produkt) Pfade abgesteckt, auf denen sich eine virtuelle Figur bewegt.

Abb. 3.3: Das Publikum kann die Fassade eines Produktes freipinseln und kann in tiefere Schichten navigieren, die an der jeweiligen Stelle an das Produkt projiziert werden.

Abb. 3.4: Das Produkt ist Teil einer Fassadenprojektion. Mit Hilfe einer audiovisuellen Simulation wird das in Wirklichkeit inaktive Produkt eindrucksvoll zum Leben erweckt.

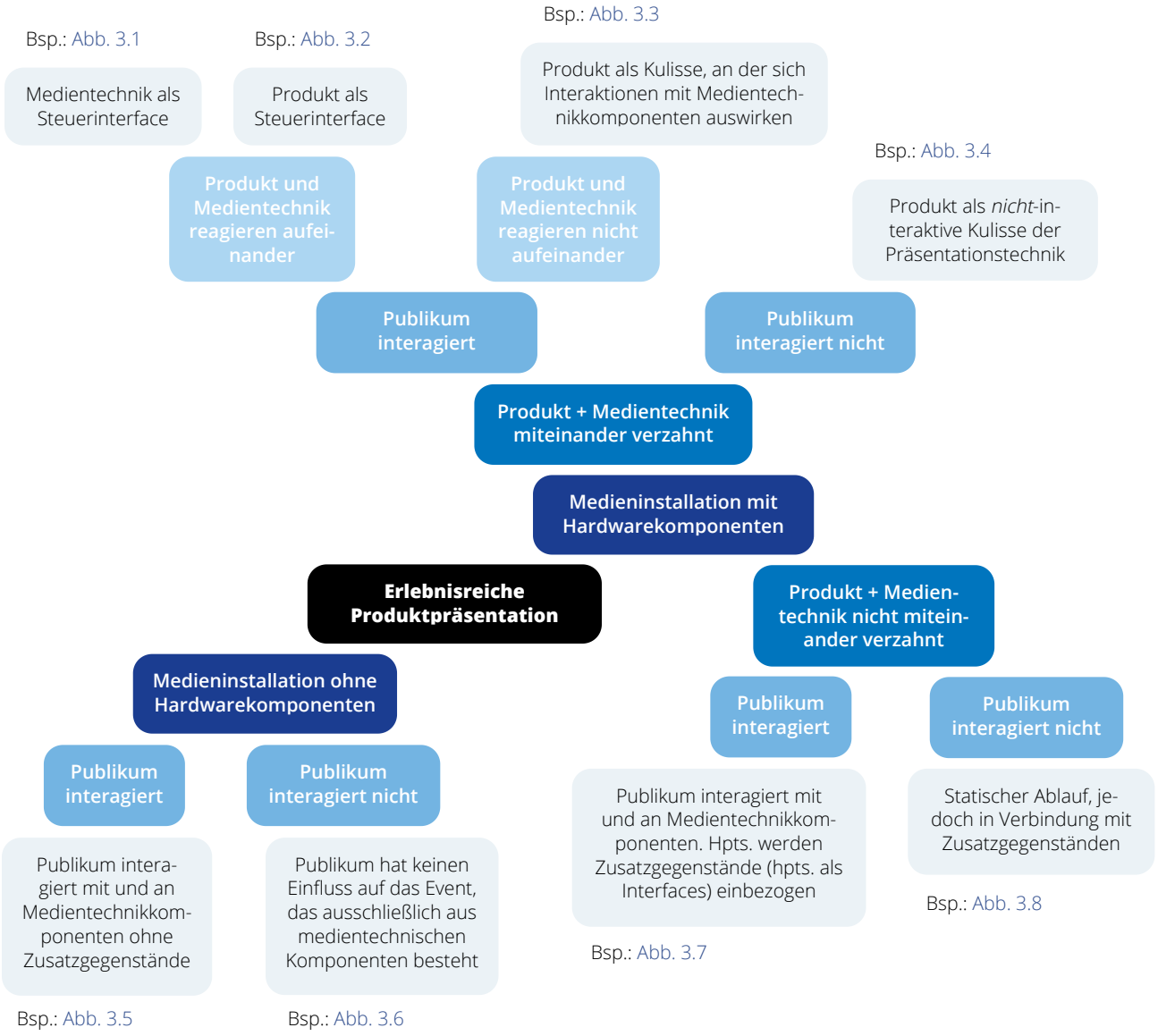


Abb. 2: Dieses Schaubild zeigt schematisch acht Möglichkeiten zur Realisierung einer erlebnisreichen Produktpräsentation

Abb. 3.5: Eine Interaktive LED-Kugel wird durch die lokale Lautstärke vom umgebenden Publikum angetrieben.

Abb. 3.6: Eine Matrix von motorisierten Edelstahlkugeln bildet visuell die legendären Modelle von BMW 3-dimensional ab

Abb. 3.7: Das Publikum kann auf trickreiche Art visuelle Effekte frei fegen. Der Besen ist nicht das umworbene Podukt, sondern ein medientechnisches Interface

Abb. 3.8: Mithilfe von audiovisuellen Effekten wird ein statisches Gebäude zur Kulisse einer eindrucksvollen Fassadenprojektion

Auf der Übersicht in Abb. 2 werden kategorisch acht Möglichkeiten einer erlebnisreichen Produktpräsentation vorgestellt.
Im oberen Abschnitt werden die vier Möglichkeiten a – d gezeigt, in denen das beworbene Produkt und die präsentierende Medientechnik ineinander zu einer untrennbaren Einheit verzahnt werden.
Im unteren Abschnitt werden vier weitere Möglichkeiten gezeigt, in denen die Medientechnik unabhängig vom beworbenen Produkt präsentiert wird – mit/ohne den Einsatz von zusätzlichen Hardwarekomponenten[1].

Für jeden der in Kap. 1.2.1 aufgeführten Kunden wurde anhand dieses Schaubildes zu jeder Kategorie mindestens ein individuelles Beispiel herausgearbeitet.
Die Kollegen vom Vertrieb wünschten eine Software, in der alle Projekte kategorisch archiviert werden und mit unterschiedlichen Suchkriterien abgerufen und grafisch aufbereitet werden können. Auf die Entwicklung dieser Software werde ich in Kap. 5 eingehen.

1.2.3. Meinungsbildung

Im vorherigen Abschnitt stellte ich folgende Behauptung auf:

> *Eine Produktpräsentation ist besonders wirkungsvoll, wenn das beworbene Produkt und die präsentierende Medientechnik grenzenlos ineinander verzahnt werden.*

Eine Umfrage (vgl. Fragebogen im Anhang A.4) mit rund 30 Teilnehmern im Februar/März 2015, online und auf einem Fragebogen, sollte ein Meinungsbild zu Messepräsentationen von Menschen aus völlig unterschiedlichen Altersgruppen und Fachrichtungen verschaffen. Teilgenommen haben Vertreter und Studenten aus den Fachrichtungen Medientechnik, Lehramt, Sprachen, Theologie, Wirtschaft, Elektronik, Bau und Informatik im Alter zwischen 21 und 67 Jahren.
Unter dem Leitsatz „Wenn ich auf eine Messe gehe …“ konnten die Teilnehmer beliebig 20 Optionen ankreuzen, womit sich ein Bild über die jeweilige Meinung zu Messenpräsentationen abzeichnete.
Die Teilnehmer sind sich einig, dass die Informationsbeschaffung auf Messen oberste Priorität hat. Bloßer Konsum ist sekundär. Die meisten Teilnehmer sind an einem Zusammenspiel von greifbaren, physischen Dingen und der Präsentationstechnik interessiert und sind Befürworter der digitalen Präsentationstechnik. Viele Teilnehmer geben an, individuell in Interaktion treten zu wollen; nur wenige möchten nicht zu sehr auffallen und ziehen ein kollektives Erlebnis mit anderen Leuten in der Menge dem individuellen Erlebnis vor. Allgemein sind Überraschungen willkommen.
Die meisten Teilnehmer möchten die präsentierten Gegenstände interaktiv ausprobieren und erleben. Jedoch wird teilweise eingeräumt, dass Publi-

1 mit diesen zusätzlichen Hardwarekomponenten ist nicht das beworbene Produkt gemeint, sondern Gegenstände, die als Post-WIMP-Interfaces Einfluss auf die Medientechnik nehmen können.

kumsinteraktionen möglicherweise nicht selbsterklärend genug sind. Während einige Teilnehmer gern des real vorhandene, beworbene Produkt in die Hand nehmen möchten, geben manche Teilnehmer an, mit komplizierten technischen Maschinen lieber über gewohnte Geräte interagieren zu wollen, z. B. über ein Tablet.

Insgesamt ist zu beobachten, dass sich die Teilnehmer der Umfrage auf diese Neuerung in der Präsentationstechnik generell einlassen. Jedoch ist zu beachten, dass es sinnvoll ist, dem Publikum gewohnte Benutzerschnittstellen anzubieten, um sich bei der Hektik auf einer Messe so intuitiv wie möglich orientieren zu können. Mobile Endgeräte, wie Tablets oder das eigene Smartphone, können als solche Interfaces agieren, vorausgesetzt das grafische User Interface ist übersichtlich genug gestaltet. Beim eigenen mobilen Endgerät handelt es sich um ein vertrautes Medium, mit dem der Endbenutzer in i. d. R. umgehen kann.

Das macht besonders Sinn bei großen, unhandlichen Produkten und wenn mehrere Benutzer gleichzeitig auf ein Exponat Einfluss nehmen möchten.

Im Folgenden werden geeignete, *allgemeine* Möglichkeiten zur Realisierung eines erlebnisreichen Messeexponats gemäß a – d untersucht.

1.2.4. Benötigte Komponenten

Ziel dieser Thesis ist die Entwicklung eines Standardsystems, das in der Lage ist, mit Hilfe von Sensoren und Aktoren mit der physischen Umwelt und mit mobilen Endgeräten zu kommunizieren.

Im Allgemeinen werden dafür folgende Komponenten benötigt:

- Ein ansprechendes **clientseitiges Screeninterface** für mobile Endgeräte[1]
- Ein **Webserver**, der die Anfragen der verbundenen Clients entgegen nimmt, mit der physischen Umwelt und mit einer Datenbank interagiert
- Eine **Datenbank**, die Informationen über das Publikum und statische Inhalte für die Medieninstallation enthält.
- Ein **Mikrocontrollerboard** als Schnittstelle zwischen dem Webserver und der physischen Umwelt. Es ist mit Sensoren und Aktoren ausgestattet.

In den folgenden Abschnitten wird untersucht, wie diese Komponenten miteinander verbunden werden.

1 Clientseitiges Screen-Interface: Website oder App für Smartphones und/oder Tablets

1.3. Grobes technisches Konzept

> *Any sufficiently advanced technology is indistinguishable from magic.*
> *– Arthur C. Clarke*

Der in Kap. 1.2.4 beschriebene Aufbau für eine komponentenreiche Medieninstallation fällt unter den Begriff *Internet der Dinge*. Zunächst wird dieser erläutert, dann soll geklärt werden, wie eine Kommunikation zwischen den einzelnen Komponenten im Allgemeinen zustande kommt.
Im Anschluss werden verschiedene Programmiermethoden untersucht, mit denen Entwickler auf die benötigten Kommunikationsprotokolle zugreifen können.

1.3.1. Das Internet der Dinge

Das *Internet der Dinge* beschreibt die Vernetzung der *physischen* und der *virtuellen* Welt. Real existierende, greifbare *Dinge* haben eine virtuelle Repräsentation in einem Netzwerk.
Diese eindeutig identifizierbaren *Dinge* können über das Internet untereinander kommunizieren und den Menschen verschiedene Aufgaben abnehmen. Beim Internet der Dinge handelt es sich somit um die Digitalisierung der Umwelt. Über geeignete *Sensoren* und *Aktoren* kommunizieren *physische* Gegenstände mit der *virtuellen* Welt und ermöglichen dem Internet, seine virtuellen Grenzen zu überschreiten.

> *Durch die Integration von einer Billionen Sensoren in die Umwelt – die alle über Computersysteme, Software und Services miteinander vernetzt sind – können wir den Herzschlag der Erde verfolgen. […].*
> – Peter Hartwell, Senior Researcher, HP Labs (vgl. Evans)

Die Einsatzmöglichkeiten der vernetzten und intelligenten Technik sind vielseitig. Sie revolutioniert u. a. Wirtschaft, Logistik, Industrie[1], Medizin, Notfall, Pflege, Bildung und Informationsvermittlung.
Es gibt unzählige Szenarien, in denen das Internet der Dinge, die die Menschen in ihrem Tun unterstützen und teilweise bereits allgegenwärtig sind:

- Paketverfolgung im Internet
- Veröffentlichung von sportlichen Leistungen bei Marathonläufen mit Hilfe von Transpondern im WWW
- Bestandaufnahme einer Bücherei im Internet
- automatische Aufforderung zum Nachkaufen von Druckerpatronen durch den Drucker auf der Herstellerwebseite

1 Die Digitalisierung der Technik wird auch als *Industrie 4.0* bezeichnet und fällt unter den Oberbegriff *Internet der Dinge*. Konkret beschreibt Industrie 4.0 eine „vernetzte Fabrik, in der Menschen und Werkstücke permanent Informationen austauschen (vgl. Spiegel15), indem neben dem Einsatz von Mechanik (*Industrie 1.0*), elektrischer Energie (*2.0*), Elektronik und Informationstechnologien (*3.0*) nun Produktionssysteme untereinander vernetzt werden, die somit intelligenter, kalkulierbarer und automatischer werden.

Derzeit werden unterschiedlichste weitere Einsatzszenarien umgesetzt, u. a. Waschmaschinen, die dann waschen, wenn der Strom am günstigsten ist[1], und kleine tragbare Geräte für pflegebedürftige Menschen, die bei Notfällen[2] ohne ihr aktives Zutun[3] das Pflegepersonal oder den Arzt informieren.

Das Internet der Dinge für Medieninstallationen

Während bei Smart Home-Diensten das Sicherheitskriterium häufig umstritten ist, bietet sich der Einsatz von intelligent vernetzten physischen Dingen durchaus zur Informationsvermittlung an. Bei Medieninstallationen ist das Datenschutzkriterium jedoch meist zweitrangig. Für diesen Zweck kann das Potenzial des Internets der *Dinge* optimal ausgeschöpft werden.

Die Kommunikation unter den Komponenten aus Kap. 1.2.4 wird mit unterschiedlichen Kommunikationsprotokollen umgesetzt, auf die ich nachfolgend eingehen möchte.

1.3.2. Kommunikationsprotokolle

Die Kommunikation zwischen zwei Komponenten einer Event Media-Installation ähnelt dem Gespräch zwischen zwei Menschen: Sie verstehen sich nur, wenn sie dieselbe Sprache sprechen. Wenn beide Seiten die Regeln der Sprache einhalten, kommt eine Kommunikation zustande. Die Sprache steht hier sinnbildlich für ein bestimmtes Kommunikationsprotokoll.
An einer Kommunikation sind mehrere Protokolle beteiligt, die aufeinander aufbauen. Das OSI-Modell[4] stellt die aufeinander aufbauenden Netzwerkprotokolle in übereinander liegenden Schichten dar (vgl. Abb. 4.1).
Interagiert ein Webentwickler beispielsweise mit dem HTTP-Protokoll an der Anwendungsschicht (Schicht 7), benutzt er indirekt auch die Protokolle der darunter liegenden Schichten 6 – 1, auf denen Schicht 7 basiert.
Sendet der Entwickler beispielsweise über ein Kabel oder per Funk ein Datenpaket an ein anderes Gerät, durchläuft es alle Schichten 7 – 1 bis zur physikalischen Schicht. Dort werden virtuelle Signale in elektrische Spannungspegel umgewandelt. Beim Empfänger wandert das Datenpaket von Schicht 1 bis 7 wieder nach oben bis zur Anwendungsschicht (vgl. Scholz).

HTML-Dokumente für Webseiten werden typischerweise über das HTTP-Protokoll[5] bereitgestellt, ebenso wie AJAX-Anfragen. Weitere wichtige Protokolle

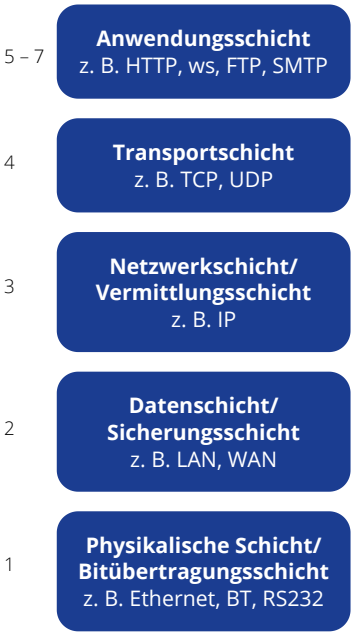


Abb. 4.1: OSI Schichtenmodell

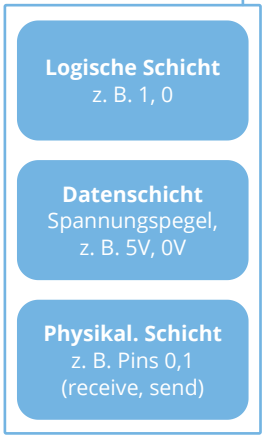


Abb. 4.2: Unterschichten der physikalischen Schicht aus dem OSI-Modell

1 Die intelligente Vernetzung von Geräten im Haus wird auch durch den Begriff *Smart Home* als Beispiel für das Internet der Dinge geprägt.
2 Über verschiedene Sensoren überwachen Kleinstcomputer z. B. Stürze oder die Überschreitung von Grenzwerten
3 Intelligente Systeme, die ohne das bewusste Zutun des Benutzers persönliche Daten über Gewohnheiten, Verhalten, Körperdaten und die Umwelt ihrer Benutzer sammeln und ihren Betrieb von diesen Informationen abhängig machen, nennt man auch *Permaaktive Systeme* (vgl. Permaactive).
4 OSI: Open Systems Interconnection Model
5 Bei TLS-verschlüsselten Verbindungen wird das verschlüsselte HTTPS-Protokoll verwendet.

dieser Schicht sind u. a. die Protokolle SMTP[1] und FTP[2]. Alle Protokolle basieren auf dem gemeinsamen verbindungssicheren Transportprotokoll TCP[3]. Streamingdienste beispielsweise verwenden als Transportprotokoll hingegen das verbindungslose UDP[4].
Der Webentwickler operiert meistens direkt mit den Protokollen der Anwendungsschicht und nutzt die tieferen Schichten des OSI-Modells nur indirekt und oft unbewusst.

Soll eine Webapplikation über Sensoren an Mikrocontrollerboards mit der physischen Umwelt interagieren können, kommt der Entwickler mit tieferen, hardwarenaheren Kommunikationsprotokollen in Verbindung, beispielsweise. wenn er die Übertragungsgeschwingigkeit (Baudrate) zwischen dem seriellen Port des Computers und einem Mikrocontrollerboard festlegen muss.[5]
Bei der Kommunikation zwischen Microcontrollerboard, in diesem Fall, einem Arduino Board, und dem Computer sind im Wesentlichen zwei Protokolle beteiligt: *TTL-seriell* und *USB* (vgl. Igoe, S. 42):

- Das Arduino Board versteht das asynchrone serielle Protokoll *TTL-seriell*. Das Arduino Board empfängt bzw. sendet Daten an den Pins 0 bzw. 1 (*Physikalische Schicht*[6]), verwendet die Spannungsimpulse 5V bzw. 0V (*Elektrische Schicht*), die die logischen Werte `1` und `0` repräsentieren (*Logische Schicht*). Die Spannungsimpulse werden z. B. mit der Geschwindigkeit 9.600 Bits/s übertragen (*Datenschicht*).
- Der Computer empfängt die Daten des Arduino Boards über das *USB*-Protokoll. Auf dem Weg vom Arduino Board zum Computer übersetzt ein USB-seriell-Wandler[7] die beiden Protokolle *TTL-seriell* und *USB* ineinander. USB sendet die Signale über zwei Drähte (– und +) (*Physikalische Schicht*) über ein symmetrisches Kabel[8] (*elektrische Schicht*) in Form von +/–5V- oder 0V-Impulsen, die die logischen Werte 1 oder 0 repräsentieren (*Logische Schicht*). Sie werden mit bis zu 480Mbit/s gesendet (*Datenschicht*). Über Treiber kann in Computersoftware auf das Arduino Board zugegriffen werden.

1.3.3. Programmierung

Es gibt viele Möglichkeiten, die in Kap. 1.2.4 genannten Komponenten zu konfigurieren und mit Hilfe von Kommunikationsprotokollen miteinander zu kombinieren.

---

1 SMTP: Simple Mail Transfer Protocol (Mailversand)
2 FTP: File Transfer Protocol (Dateiübertragung)
3 TCP: Transfer Control Protocol
4 UDP: User Datagram Protocol
5 Inzwischen wurde jedoch mit Firmata eine Bibliothek entwickelt, die dem Entwickler den Umgang mit der hardwareseitigen seriellen Kommunikation erübrigt, indem er alle Konfigurationen an der Anwendungsschicht vornehmen kann.
6 Die von Igoe beschriebenen Schichten (Logische Schicht, Datenschicht und Logische Schicht) sind alle in die Physikalische Schicht des OSI-Modells einzuordnen.
7 USB-seriell-Wandler: FTDI-Converter
8 Symmetrisches Kabel: Auf zwei Adern im Kabel werden jeweils gegenphasige Signale übertragen. Das Aufsummieren der gegenläufigen Spannungspegel, 5V oder 0V, sollte immer `0` ergeben. Abweichungen sind Störartefakte, die herausgefiltert werden können.

Um mit den benötigten Kommunikationsprotokollen komfortabel zu operieren, verwendet der Entwickler typischerweise eine anwendungsorientierte Programmiersprache, bei der der Quellcode für ihn lesbar ist. Von einem Compiler wird der Quellcode entlang seiner Syntax in eine für den Computer verständliche Sprache übersetzt.
Zu solchen anwendungsorientierten Programmiersprachen zählen

- *textbasierte* eindimensionale Skriptsprachen[1]
- *visuelle* Programmiersprachen[2]
- *vollgrafische* Generatorsysteme[3].

Kombination aus mehreren dieser Programmiermöglichkeiten werden *hybride* Generatorsysteme[4] genannt. Bei diesen Möglichkeiten nimmt die Benutzerfreundlichkeit der Reihe nach zu und die Flexibilität ab. Im Folgenden sollen textbasiertes Skripting und visuelle Programmierung miteinander verglichen werden.

Textskripting vs. visuelle Programmierung

Programme, die mit klassischen Programmiersprachen geschrieben werden, bestehen aus eindimensionalen Zeichenfolgen. „In Bezug auf die Richtigkeit der Semantik[5] wird lediglich die Abfolge der Zeichen überprüft. Für den Compiler [oder den Interpreter[6]] ist die grafische Anordnung[7] eines Programmcodes meist unsichtbar“ [Heinsch, S. 12].

In junger Vergangenheit wurden einige visuelle Programmiermöglichkeiten entwickelt. Ein bedeutender Grund dafür ist, dass visuell denkende Menschen immer häufiger generative Systeme als Gestaltungsmittel einsetzen.

In den meisten Fällen wird in visuellen Generatorsystemen Programmlogik aufgebaut, indem virtuelle *Boxen*[8] durch virtuelle *Kabel*[9] miteinander verbunden werden. Diese Boxen repräsentieren Objekte, Funktionen oder Werte (vgl. Abb. 5 auf der nächsten Seite). Der Vorteil visueller Programmiersprachen liegt hauptsächlich darin, in Echtzeit programmieren und das Ergebnis sofort sehen zu können. Außerdem lässt eine Programmiersprache keine Fehler zu: Nodes, die nicht miteinander harmonieren, können nicht miteinander verbunden werden. vvv, „a multi purpose kit“ [vvv.org], ist eine visuelle Programmiersprache. Im Folgenden wird erörtert, ob vvv für die Anforderungen in Kap. 1.2.4 geeignet ist und eine Skriptsprache ersetzen kann.

1 Eindimensionale Skriptsprachen: Z. B. Java, JavaScript, C, C++, C#, PHP, Ruby
2 Visuelle Programmiersprachen: Z. B. vvv, MSP/Max, PureData, Quartz Composer, Matlab Simulink
3 Vollgrafische Generatorsysteme: Z. B. Derivative TouchDesigner, Modul8, Milllumin
4 Hybride vollgrafische Generatorsysteme: Z. B. Unity, Ventuz, Adobe Flash
5 Semantik: Bedeutung der Zeichen
6 Interpreter (z. B. in JavaScript): Programm, das jeweils eine einzelne Zeile einer Anwendung übersetzt und ausführt.
7 Grafische Anordnung: Z. B. Code-Einrückung oder Syntax-Highlighting
8 Boxen: In vvv auch *Nodes* genannt
9 Kabel: In vvv auch *Links* genannt

1.3.3.1. Evaluierung der Eignung einer grafischen Programmiersprache für den Einsatz in der Messeprogrammierung am Beispiel von vvv

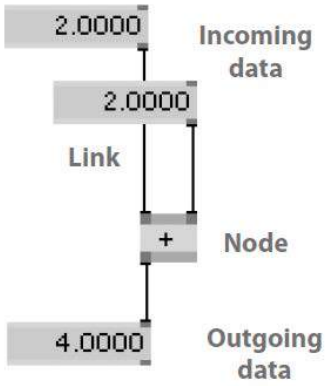


Abb. 5: Die Programmlogik entsteht durch Verbinden von Nodes durch Links (Beispiel aus vvv).

vvv wurde hauptsächlich zur Manipulation von Video-, Grafik- und Datenströme entwickelt, die bereits „*während* der grafischen Konstruktion kompiliert und ausgeführt" werden [Heinsch, S. 9]. Der logische Datenstrom fließt von Funktion zu Funktion von oben nach unten (vgl. Abb. 5). Der Skriptcode dazu wird im Hintergrund in einer XML-Datei abgelegt und spielt für den anwendenden Entwickler keine Rolle.

vvv wird typischerweise zur Manipulation vieler Bild-, Video- und Tonformate eingesetzt und bietet zahlreiche Schnittstellen zur physikalischen Umwelt über Input- und Output-Geräte, dem Internet und sogar Datenbanken an.
Jede Funktion in vvv ist in einer Node untergebracht, in die i. d. R. eine *weitere* Komposition verschachtelt wird. Nodes beherbergen u. a. logische Operationen[1] und Bindeglieder[2] für zusätzliche Interface- und Output-Geräte[3].
Zudem ist es möglich, eigene, in C# geschriebene Skripte, hinzuzufügen.
vvv ist eine Programmiersprache mit einem logischen Konzept und einem großen Funktionsumfang und ist somit ein Kandidat für die Programmierung einer erlebnisreichen Produktpräsentation.

Untersuchung von vvv nach Eignung

Der Leistungsumfang von vvv wird von einer großen Community an Entwicklern stetig weiterentwickelt. Mittlerweile ist es mit vvv möglich, mit Hilfe von diversen Kommunikationsprotokollen mit mobilen Endgeräten und dem Internet zu kommunizieren. Beispielsweise können mit der Node *venode* innerhalb von vvv die Funktionaliäten von *Node.js* genutzt werden und mit der Node *mysql* auf eine *MySQL*-Datenbank zugegriffen werden.
vvv erfüllt also theoretisch alle vier Anforderungen aus Kap. 1.2.4[4].

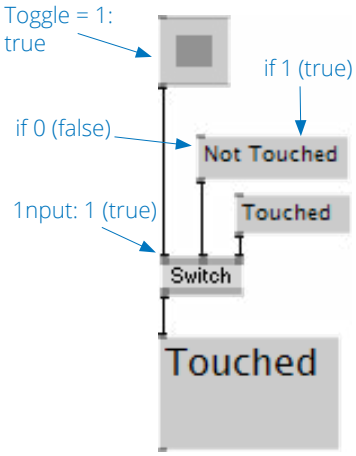
Dass vvv flexible Aufgaben wie in etablierten Skriptsprachen übernehmen kann, zeigen seine logischen Konzepte. Einige Beispiele hierfür sind (frei nach Boyarintsev, S. 29):

- *Toggle-Boxen* stehen für b**oolesche Werte**[5] (vgl. Abb. 6 oben).
- Mit `switch`- und `case`-Nodes können logische `if..else` bzw. `switch..case`-Abfragen realisiert werden. Für einfache Bedingungen können auch die Nodes `=`, `<` oder `>` verwendet werden.
- Die Verarbeitung von **Arrays**[6] in Echtzeit ist eine der Hauptstärken von vvv. Arrays werden dort *Spreads* genannt. Entsprechende Nodes sind z. B. `LinearSpread` und `CircularSpread`. Mit Hilfe der Node `Zip` können Werte in einem Array zusammengefügt werden (vgl. Abb. 6 Mitte).
- Mit den Nodes `OR`, `AND` und `NOT` können weitere **logische Beziehungen** geschaffen werden (vgl. Abb. 6 unten).

1 Logische Operationen in vvv: U. a. `OR`, `AND`, `SWITCH`, `CASE`, `=`, `<`, `>`, `TogEdge`, `FlipFlop`
2 Verbindungsprotokolle in vvv: U. a. TCP/IP, UDP/IO, MIDI, DMX (via Artnet),OSC, RS-232
3 In- und Outputgeräte in vvv: U. a. Arduino Board, Kinect, Leap Motion
4 Anforderungen: Server, clientseitiges GUI, Verbindung mit Arduino Boards, Datenbank
5 Boolescher Wert: Wahrheitswert (true, false)
6 Array: Sammlung von Daten

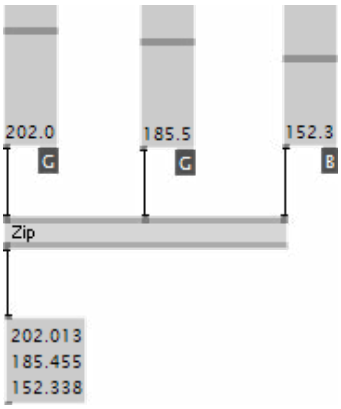
- Gesamte *Patches*[1] können als Sub-Patches in übergeordnete Patches eingefügt werden und wie **Funktionen** in etablierten Programmiersprachen an verschiedenen Stellen wiederverwendet werden. Mit den Nodes `R` und `S` können **Go-To-Verweise** erstellt werden.
- Es können **Funktionen** an andere Funktionen weitergereicht werden, indem entsprechende Nodes mit tieferen Nodes verbunden werden.

Für die Kommunikation mit der physikalischen Umwelt bewährt sich der Einsatz von Arduino Boards (vgl. Barth *et al.*). Das Arduino Board ermöglicht über analoge Input-Pins und digitalen In- und Output-Pins, sowohl Sensoren[2] als auch Aktoren[3] anzuschließen.
vvv bietet zwei Möglichkeiten, mit einem Arduino Board zu kommunizieren. Zum einen können mit der Node `RS232` über eine klassische serielle Kommunikation zwischen dem *Arduino Board* und dem *seriellen Port* eines Computers Zeichenketten ausgetauscht werden.
Zum anderen kann vvv über das *Firmata-Protokoll*, das hinter der Node `Arduino` arbeitet, die volle Kontrolle über das Arduino Board gewinnen. Während der Weg über das Firmata-Protokoll flüssig läuft, arbeitet die klassische zeichenkettenbasierte Kommunikation mit vvv oft problematisch. Firmata und die klassische serielle Kommunikation werden in Kap. 6 behandelt.



Bei intensiver Nutzung von vvv machen sich jedoch Schwächen bemerkbar:

- Komplexer werdende Patches werden zunehmend unübersichtlich[4].
- Die Performance leidet[5]: Programmabstürze sind keine Seltenheit. Regelmäßige Backups sind unabdinglich. Gelegentliche *Unstimmigkeiten* mancher Nodes sorgen für Kopfzerbrechen.[6]
- Schleifenbildung[7], wie aus etablierten Programmiersprachen bekannt, ist mit vvv nicht möglich. Das liegt daran, dass ein vvv-Patch pro Takteinheit *genau einmal* durchlaufen wird. Bildlich gesehen fließt der Datenstrom in einem vvv-Patch in jedem Takt von oben nach unten (vgl. Heinsch). Diese Zeit kann nicht weiter unterteilt werden. Deshalb müssen bei der Simulation von Schleifen Alternativen verwendet werden.[8]
- Aktuell (Stand Mai 2015) ist vvv Rechnern mit Windows-Betriebssystemen vorbehalten, da vvv auf Microsofts Softwareplattform *.NET* basiert.[9] Da viele Interaction Designer mit Mac-Rechnern arbeiten, stellt dieser Umstand ein starkes Defizit dar. Die Online-Entwicklungsumgebung vvvjs.com bietet nur eingeschränkte Möglichkeiten und bietet für die Entwicklung eines interaktiven Messeexponats keine Alternative zur .NET-basierten Software vvv[10].



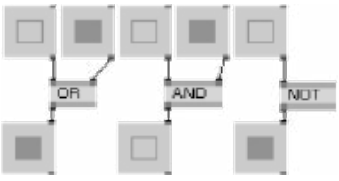


Abb. 6: Umsetzung von Logik mit vvv

1 Patch: vvv-eigene Bezeichnung für eine Komposition
2 Sensoren: Z. B. Taste, Drehpotenziometer, Touch-, Temperatur-, Bewegungssensor
3 Aktoren: Ausgabeelektronik, wie Z. B. Motor, LED, Transistor- und Relais-Schaltung
4 Die Komplexität eines logischen vvv-Patches beginnt bereits im Kleinen, wenn z. B. für eine einfache `if..else`-Abfrage
5 Das Konzept *Boygrouping*, das die Rechenlast einer vvv-Applikation auf mehrere Rechner verteilt, ist ein Lösungsansatz für das Performanceproblem,.
6 Mir ist beispielsweise nicht gelungen, den Sensorwert eines Arduino Boards in vvv zu bearbeiten und ihn für einen Aktoren am selben Arduino Board wiederzuverwenden.
7 Schleifen: Z. B. `for`-, `while`-, `do..while`- Schleife
8 Wiederkehrende Abläufe können z. B. mit der Node `LFO` realisiert werden.
9 Microsoft hat allerdings angekündigt, die .NET-Plattform ab der nächsten Version 5.0 auch für Mac OSX und Linux -Betriebssystemen verfügbar zu machen.
10 vvvjs.com basiert auf clientseitigem JavaScript und lässt keine Interaktion mit Hardware zu, die an den lokalen Computer angeschlossen ist. Somit kann mit vvvjs.com keine Kommuikation mit der physischen Umwelt hergestellt werden, etwa mit Hilfe von Arduino Boards und Kinect-Sensoren.

1.3.3.2. Skriptsprache statt visuelle Sprache

Bei wachsenden Projekten erweisen sich die textbasierten *Skriptsprachen* flexibler als *visuelle* Programmiersprachen, wie vvv.
Mit textgebundenen Skriptsprachen kann der Entwickler, u. a. mit Suchfunktionen Programmteile gezielt finden, Programmlogik bei Bedarf mithilfe von Zeichenketten zusammenstellen[1] und sich bei der Entwicklung an Mitteilungen in der Konsole orientieren. Mit u. a. Code-Einrückungen und Syntax-Highlighting bieten viele Skripteditoren weitere visuelle Orientierungspunkte an.

Ähnlich wie vvv bieten einige Webentwicklungs-Tools[2] Möglichkeiten zur Entwickung in Echtzeit an, bei der das Ergebnis unmittelbar nach der Texteingabe sichtbar wird.

Es ist von Vorteil, bei einer komponentenreichen Medieninstallation bewährte Standardtechnologien zu verwenden, mit denen viele Entwickler vertraut sind. Da eine solche Medieninstallation ein Beispiel für das *Internet der Dinge* ist, macht es Sinn, hierfür Webentwicklungs-Tools einzusetzen. Dabei werden häufig unterschiedliche Programmiersprachen eingesetzt. Beispiele sind:

- Client-seitiges GUI: Im Fall eines Webinterfaces im Browser: HTML, CSS, JavaScript, im Fall einer nativen App für mobile Endgeräte: Für Android z. B. Java, für iOS z. B. Objective C, Swift und für Windows Phone z. B. C#.
- Webserver: Apache (mit PHP), nginx, IIS
- Datenbank: MySQL (mit PHP)
- Schnittstelle zur physischen Umwelt mit einem Arduino Board[3]: PHP für die Kommunikation zwischen Computer und Microcontroller und eine C-artige Arduino-Sprache auf dem Microcontroller.

Zudem macht es Sinn, für möglichst viele dieser Komponenten dieselbe Programmiersprache zu verwenden. Hiermit werden die Grenzen der einzelnen Aufgabenbereiche aufgelöst.

JavaScript bietet den Funktionsumfang, der den Anforderungen gerecht wird: JavaScript ist eine mittlerweile universelle Programmiersprache, die klassischerweise dafür verwendet wird, statische Webseiten dynamisch zu machen. Im Zuge des Internets der Dinge ist JavaScript über die Grenzen des Browsers hinausgewachsen.

1 JavaScript bietet mit der Methode *eval* die Möglichkeit, Zeichenketten, die in Textform vorliegen, in z. B. Funktionsaufrufe oder Dateipfade umzuwandeln.
2 Editoren mit Echtzeitfunktionalität: Software z. B. WebStorm, Sublime Text, online z. B. codepen.io
3 Arduino: http://www.arduino.cc (Stand: 11. April 2015)

1.4. Zusammenfassung und Überleitung

In diesem Kapitel wurde auf konzeptionelle Weise ein Standardaufbau für eine komponentenreichen Medieninstallation zum Einsatz bei Produktpräsentationen skizziert. Dabei werden Erkenntnisse der Wahrnehmungspsychologie und Erwartungen von Kunden sowie dem Publikum berücksichtigt. Zudem wurde eine Schwachstelle bei vielen aktuellen Messepräsenzen aufgedeckt: Die Beobachtung, dass die beworbenen Produkte häufig nicht optimal in die präsentierende Medieninstallation integriert wird, führte zu dieser Behauptung:

> *Eine Produktpräsentation ist besonders wirkungsvoll, wenn das beworbene Produkt und die präsentierende Medientechnik grenzenlos ineinander verzahnt werden.*

Anschließend wurde die Kommunikation zwischen den beteiligten Komponenten untersucht. Um sie komfortabel zu konfigurieren, wurden visuelle und textbasierte Programmiersprachen miteinander verglichen – mit der Erkenntnis, dass sich die Skriptsprache JavaScript besonders für diese Zwecke eignet.
Ab Kap. 2 werde ich näher auf die Vorteile von JavaScript bei der Realisierung von komponentenreichen Medieninstallationen eingehen.

Kap. 2: JavaScript-Konzepte für Event Media-Installationen

Stand der Technik: One tool for all issues

> *Over the last several years [...], JS[1] has expanded beyond the browser into other environments, such as servers, via things like Node.js. In fact, JavaScript gets embedded into all kinds of devices these days, from robots to lightbulbs.* [Simpson (Async), S.6]

In Kap. 1.2.4 wurde ein Standardaufbau für eine komponentenreichen Event Media-Installation skizziert. Typischerweise werden für einen solchen Aufbau zahlreiche unterschiedliche Programmiersprachen verwendet.
Beispielsweise:

- **Client-seitiges GUI:** Im Fall eines Webinterfaces im Browser: HTML, CSS, JavaScript, im Fall einer nativen App für mobile Endgeräte[2]: Für Android z. B. Java, für iOS z. B. Objective C, Swift und für Windows Phone z. B. C#
- **Webserver:** Apache, nginx, IIS
- **Datenbank:** MySQL
- **Schnittstelle zur physischen Umwelt** mit einem Mikrocontrollerboard, z. B. mit einem Arduino Board[3]: PHP für die Kommunikation zwischen Computer und Microcontroller und die an die Skriptsprache C angelehnte Sprache für den Arduino-Controller

Unter Verwendung von JavaScript kann die Anzahl an unterschiedlichen Programmiersprachen deutlich reduziert werden. Prinzipiell können mit JavaScript unter Verwendung geeigneter Kommunikationsprotokolle *alle* erwähnten Komponenten programmiert und miteinander verknüpft werden.
Für einen Beispielaufbau mit Hilfe von JavaScript können folgende Tools verwendet werden:

- **Clientseitiges GUI:** Im Fall eines Webinterfaces im Browser: HTML, CSS, JavaScript, im Fall einer nativen App für mobile Endgeräte: Für Android, iOS und Windows Phone: PhoneGap[4]
- **Webserver:** Node.js[5] mit entsprechenden Modulen[6]
- **Datenbank:** MongoDB[7] (Zugriffssprache ist JavaScript)
- **Schnittstelle zur physischen Umwelt**: Tessel Board (mit eingebautem JavaScript-Interpreter)

1 JS: JavaScript
2 vgl. Dashewski (Stand: 1. Mai 2015)
3 Arduino: http://www.arduino.cc (Stand: 1. Mai 2015}
4 PhoneGap: http://phonegap.com (Stand: 1. Mai 2015)
5 Node.js: https://nodejs.org (Stand: 1. Mai 2015)
6 Module für Node.js für einen Webserver: http, express
7 MongoDB: https://www.mongodb.org (Stand: 1. Mai 2015)

Porträt JavaScript für den Einsatz in Event Media-Installationen

> *JavaScript has certain characteristics which makes it unique from the other dynamic languages. It has no concept of thread, but its model of concurrency is completely basing on events; i.e. it's an event driven programming language.* [Shan]

JavaScript ist eine *funktionale* (vgl. Kap. 2.3.1) und *ereignisbasierte* (vgl. Kap. 2.3) Skriptsprache, die sich zur Realisierung von leichtgewichtigen und dynamischen Webanwendungen eignet.
Es handelt sich um eine interpretierte und objektorientierte Programmiersprache, die ursprünglich zur Dynamisierung von statischen Webseiten entwickelt wurde. Die derzeit gängigsten Webbrowser[1] beinhalten JavaScript-Engines. Beispiele sind: Chrome, Opera: V8, Firefox: Spidermonkey, IE9, Spartan (ab Windows 10): Chakra, Safari: JavaScriptCore (Apples kommerzieller Name: Nitro).
Somit funktioniert JavaScript plattformunabhängig, so auch auf den meisten aktuellen mobilen Endgeräten. Auf die Interpretierung eines JavaScript-Programms werde ich in Kap. 2.1 näher eingehen.

JavaScript wird klassischerweise als clientseitige Programmiersprache eingesetzt, um HTML und CSS unaufdringlich[2] über das DOM[3] zu manipulieren. Darauf werde ich in Kap. 2.2 näher eingehen.

Dank seines ereignisbasierten Ansatzes hält JavaScript im Zusammenhang mit dem Internet der Dinge auch außerhalb des Webbrowsers Einzug auf u. a. Webserver, Mikrocontroller, Apps, Datenbanken und Game Engines.
JavaScript ist im Vergleich zu anderen Programmiersprachen wenig ressourcenintensiv[4] und für den Produktiveinsatz mit vielen mobilen Endgeräten im Bereich Event Media bestens geeignet. Für den ereignisbasierten Ansatz werden mehrere grundlegende JavaScript-Konzepte vorausgesetzt, die ab Kap. 2.3 behandelt werden. Der funktionale Charakter von JavaScript (vgl. Kap. 2.3.1) ermöglicht die asynchrone Programmierweise mit Callbacks und Callbackfunktionen (vgl. Kap. 2.3.3) im Singlethreadmodell (vgl. Kap. 2.4) in Verbindung mit der Task Queue (vgl. Kap. 2.4.2) und dem Eventloop (vgl. Kap. 2.4.3).

Auch dem Entwickler gegenüber präsentiert sich JavaScript als besonders komfortabel: Es ist flexibel, schwach typisiert[5], weitgehend fehlertolerant und akzeptiert unterschiedliche Lösungswege[6] zum gleichen Ziel.

---

1 Liste der JavaScript-fähigen Webbrowser: http://en.wikipedia.org/wiki/Comparison_of_web_browsers#JavaScript_support (Stand: 22. März 2015)

2 „unobstrusive" (zumindest clientseitig): Javascript ist wie CSS eine optionale Erweiterung von HTML, die die Grundfunktion von HTML, z. B. statischen Text und Bilder anzuzeigen, nicht beeinträchtigt. JavaScript ist keine Voraussetzung dafür, dass HTML funktioniert, aber der Zusatz von JavaScript (sozusagen das Verhalten) verleiht HTML Leben.

3 DOM: Document Object Model (vgl. Kap. 2.1)

4 wenig ressourcenintensiv, z. B. aufgrund der prototypenbasierten Vererbung und des Singlethreadmodells

5 Der Entwickler muss sich nicht bereits bei der Variablendeklaration darauf festlegen, welche Art von Daten später enthalten ist. Mögliche Datentypen in JavaScript sind `String`, `Number`, `Boolean`, `null`, `undefined`, `Object`, vgl. Simpson (this), S. 36.

6 Mehrere Wege zum selben Ziel, z. B. die Bildung von Objekten je nach Belieben in der konstruierten Form oder Literalschreibweise, vgl. Simpson (this), S.35.

Als etablierte Programmiersprache für die Webentwicklung erfreut sich JavaScript einer großen Community. Im Laufe der Zeit gingen aus ihr u. a. mehrere Erweiterungsbibliotheken[1], wie jQuery, und Online-Editoren hervor.

Vergleich zu Java

Besonders auffällig ist bei JavaScript die syntaktische Ähnlichkeit zu Java. Dass neben den nativen JavaScript-Möglichkeiten zunehmend Bordmittel aus Java adaptiert werden, z. B. der Operator `new`, obwohl die Vererbung bei JavaScript prototypen- statt klassenbasiert funktioniert, macht den Umgang mit JavaScript unübersichtlich, vor allem wenn mit neuen Versionen von ECMA Script ständig neue Methoden zum Standard erklärt werden, die zum selben Ziel führen. Die syntaktische Angleichung an Java ist sehr komfortabel für Quereinsteiger, verwirrt jedoch umso mehr[2], wenn dadurch die JavaScript-eigenen Konzepte immer mehr versteckt werden.

> *JavaScript und Java haben ebenso wenig gemein wie ein Hamburger und 'ham'. Beides ist lecker, ansonsten teilen sie aber nur die englische Silbe 'ham' .* [Resig & Bibeault, S. 63]

JavaScript verfügt über völlig eigene Konzepte, z. B. der funktionale Charakter von JavaScript, die prototypenbasierte anstatt klassenbasierte Vererbung und das Singlethread- statt Multithreadmodell (vgl. Kap. 2.4). Am Angleichungstrend der Syntax von Javascript an Java wird sich auch in der kommenden Version 6 von ECMA[3] Script nichts ändern.[4] Die Syntax ist an Java angelehnt, mit dem Hintergrund, möglichst vielen Entwicklern den Einstieg leicht zu machen.

Im folgendem Anschnitt wird der Vorgang bei der Ausführung eines JavaScript-Programms näher erläutert und dabei u. a. begründet, was JavaScript plattformunabhängig macht.

---

1 JavaScript-Bibliothek: Ein in JavaScript geschriebener Code, der eine fertige Lösung für komplexe Aufgaben beinhaltet.

2 Ähnlichkeit zu Java an der Oberfläche, z. B. die Vererbung: Neben der Vererbung mit `Object.create` oder mit einer einfachen Zuweisung eines Wertes an die Methode `prototype` ist wie auch in Java die Vererbung mit dem Operator `new` möglich. Allerdings handelt es sich auch hierbei um eine prototypenbasierte Vererbung und nicht um eine möglicherweise vermeintliche klassenbasierte Alternative.

3 ECMA Script ist ein Standard, der eine „dynamisch typisierte, objektorientiert, aber klassenlose Skriptsprache" [ecma, V. 5.1] beschreibt. ECMA Script ist der Sprachkern von u. a. JavaScript und ActionScript.

4 In ECMA Script 6 wird u. a. wie im klassenbasierten Java der Datentyp class eingeführt, obwohl JavaScript auch in Zukunft keinen klassenbasierten Ansatz annimmt.

2.1. Ausführung v. JS-Programmen

Damit ein Programmcode funktioniert, gilt (unabhängig von JavaScript):
Erst kompilieren[1], dann ausführen[2].
JavaScript-Programme werden, anders als z. B. in C oder C++ geschriebene Programme, nicht bereits in einem Build Step *vor* der Programmausführung kompiliert, sondern Zeile für Zeile, *unmittelbar vor* der Ausführung übersetzt. „[This] happens, in many cases, mere microseconds (or less!) before the code is executed". Demnach fällt JavaScript in die Kategorie der „‚dynamic' or ‚interpreted' languages" [Simpson (Scope), S. 3].

2.1.1. Unterschied zwischen kompilierten und interpretierten Programmiersprachen

Kompilierte Programmiersprachen müssen *vor* Ihrer Ausführung in Maschinencode übersetzt werden. Vorkompilierte Programme funktionieren schneller als interpretierte, jedoch sind sie nicht plattformübergreifend[3], was sie für den Einsatz als Webanwendung für unterschiedliche Endgeräte unflexibel macht[4].
Interpretierte Programmiersprachen werden *nicht* in Maschinencode kompiliert, bevor sie ausgeführt werden. Unter Verwendung von JavaScript erhält der ausführende Computer, der z. B. eine Webseite öffnet, von einem Server eine Kopie des Originalquellcodes in Textform, den der Entwickler geschrieben hat. Erst am Computer des Empfängers wird dieser Quelltext interpretiert (vgl. Abb. 1).

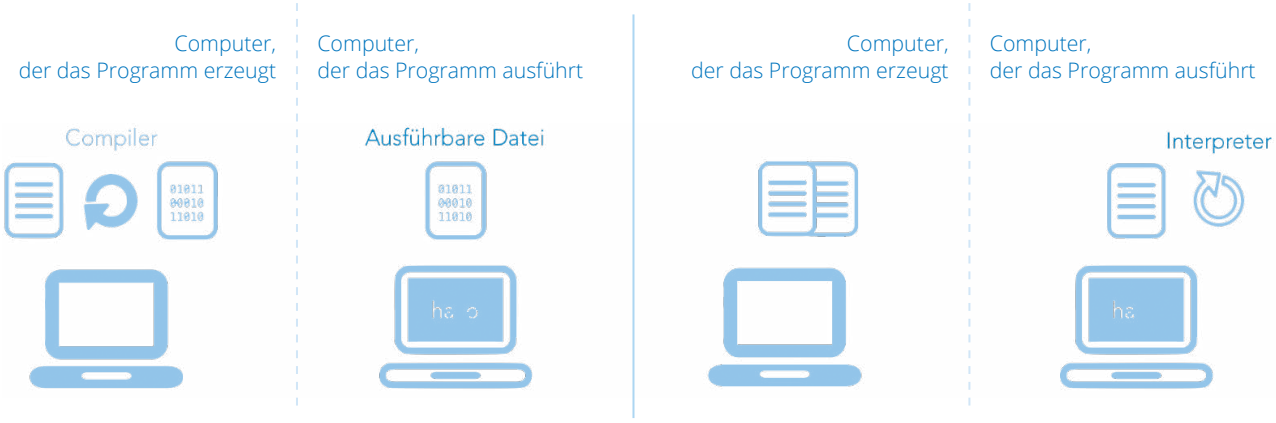


Abb. 1: Unterschied zwischen kompillierten (linker Teil) und interpretierten (rechter Teil) Programmiersprachen

1 Kompilieren: Programmcode wird in ein f. d. Computer ausführbares Format übersetzt.
2 Interpretieren: Das Ausführen eines JavaScript-Programms (engl. execution) durch eine JavaScript-Engine
3 Vorkompilierte Programme sind nicht plattformübergreifend, d. h. sie sind an ein bestimmtes Betriebssystem und manchmal auch an eine bestimmte CPU gebunden.
4 Mit der Java Runtime Environment (JRE) bietet Java eine Plattform, die auf unterschiedlichen Betriebssystemen installiert werden kann. Sie dient dazu, dass kompilierte Java-Programme auf jedem Betriebssystem funktionieren, vorausgesetzt, die Java Runtime Environment (JRE) ist installiert.

Wie *genau* die Interpretierung eines JavaScript-Programms umgesetzt wird, wird in den JavaScript-Engines der unterschiedlichen Browser unterschiedlich gelöst:
„The V8 'interpreter' compiles to native code internally, Rhino optionally compiles to Java bytecode internally, and various Mozilla interpreters (Trace-, Spider- und Jager Monkey) use a Just-in-Time[1] compiler". [Samuel]
Demnach sind interpretierte Programmiersprachen zwar langsamer als kompilierte Sprachen, dafür aber plattformübergreifend, einfacher zu debuggen und zu testen, da dem Empfänger i. d. R. das Programm in einem lesbaren Quellcode statt in einem bereits kompilierten Maschinencode vorliegt (vgl. Abb. 1, Allardice & Rose).

2.4.2. Ausführung einer interpretierten Programmiersprache

Die Kompilierung eines JavaScript-Programms besteht zusammengefasst aus drei Schritten: *Tokening, Parsing* und *Codegenerierung*.

- Beim **Tokening**[2] wird der JavaScript-Code in seine Einzelteile, in sogenannte *Tokens*, zerlegt. Beispielsweise wird die Zeile `var a = 2;` in die Tokens `var` , `a` , `=` , `2` und `;` zerlegt.
- Beim **Parsing** werden Tokens in eine verschachtelte key/value-Struktur überführt. In diesem Fall wird `a` zu einem *key* und `val` zu seinem *value*.
- Bei der **Codegenerierung** wird die key/value-Struktur für die Zeile `var a = 2;` schließlich in Bytecode[3] umgewandelt, „into a set of machine instructions" [Simpson (Scope), S.2]. Es wird Speicherplatz für den key `a` reserviert. Diesem Key wird später bei der Ausführung dieser Programmzeile durch die Engine ein *value* zugewiesen, nämlich `2`.

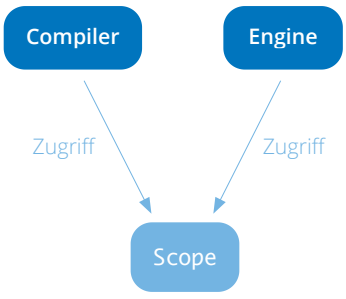


Abb. 2: Scope als gemeinsames Gedächtnis von Compiler und Engine

Bei der Interpretierung und Ausführung eines JavaScript-Programms sind drei Teile beteiligt: *Compiler*, *Engine* und *Scope*.

- Die **Engine** veranlasst die Kompilierung und die Ausführung eines JavaScript-Programms
- Der **Compiler** übernimmt die Aufgaben Tokening, Parsing und Code-Generierung (siehe oben).
- Der **Scope** ist das gemeinsame Gedächtnis von Compiler und Engine (vgl. Abb. 2).

1 Just-In-Time-Compiler (JIT): Teilprogramme werden während ihrer Laufzeit Stück für Stück in Maschinencode umgewandelt, z. B. beim dynamischen Rendern einer großen Website.
2 Tokening wird auch *Lexing* genannt
3 Bytecode: Kompakter numerischer Code in Form eines Arrays aus u. a. Konstanten und Referenzen, der aus dem Parsing hervorgeht und vom Interpreter genutzt wird. Beispiel: `31624 [214,1,0,4,0,59, ... ]`

Der Compiler veranlasst, dass während des Parsens eines JavaScript-Quellcodes die extrahierten key/value-Paare im Scope abgespeichert werden. Die Engine erhält vom Compiler einen Bytecode, den sie bei der Programmausführung interpretiert. Dieser Bytecode enthält Verweise auf key/value-Paare, die die Engine im Scope *nachschlagen* kann.

Abb. 3 fasst die Aufgabenschritte des Compilers (Tokening, Parsing, Code Generation) und der Engine (Ausführung) zusammen.

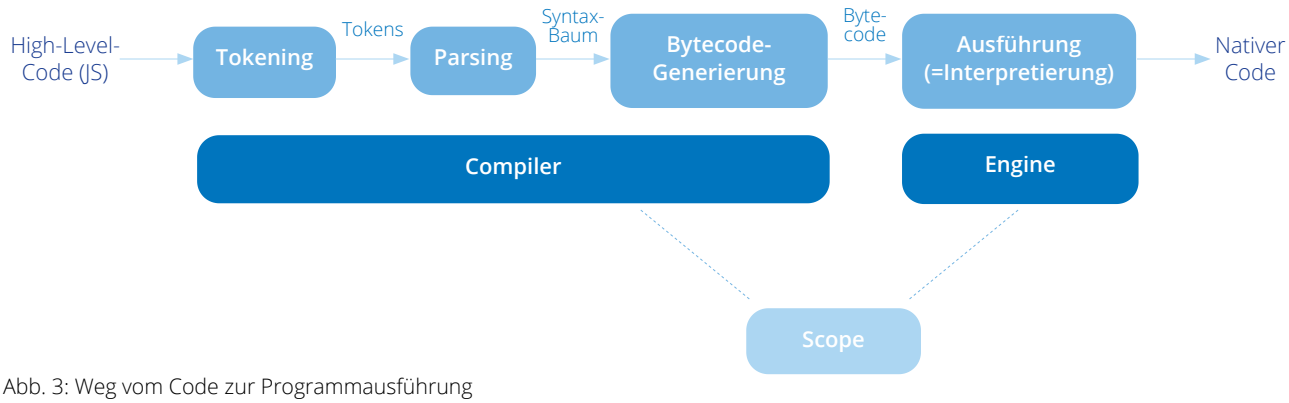


Abb. 3: Weg vom Code zur Programmausführung (kann je nach Browser abweichen)

Bei der Übersetzung eines vom Entwickler geschriebenen Quellcodes wird das Programm intern in Form von verschachtelten key/value-Paaren abgebildet. „First, Compiler declares a variable (if not previously declared) in the current Scope, and second, when executing, Engine looks up the variable in Scope and assigns to it, if found." [Simpson (Scope), S. 5]. Grob gesehen läuft dabei Folgendes ab:

- Der Compiler ist während der Kompilierungsphase zuständig für das Anlegen von *keys*, wenn sie noch nicht vorhanden sind; weder im aktuellen Scope noch in den Scopes der umgebenden Funktionen.
- In der anschließenden Ausführungsphase ist die Engine für die Zuweisung von *values* zu den dafür reservierten *keys* zuständig, vgl. Simpson (Scope), S. 42

Demnach wird die Programmzeile `var a = 2;` in zwei Programmschritte aufgeteilt: `var a;` und `a = 2;`

- `var a;` wird während der Kompilierungsphase
- und `a = 2;` während der Ausführungsphase verarbeitet.

Bei der Kompillierungsphase kreiert der Compiler anhand des Quellcodes im Scope eine Variable, die über den key `a` erreichbar ist. Dann erstellt er Bytecode für die Engine, der u. a. auf das neu angelegte Objekt hinweist.
Bei der Ausführungsphase sucht die Engine nach dem key `a` und schreibt den Wert `2` hinein.

Hoisting

```
getSweet(“Bonbon”);

function getSweet(type) {
    console.log(type);
}
// Bonbon
```

Listing 4.1: Demonstration des function hoistings mit einer Funktionsdeklaration

Function hoisting ist ein Nebenprodukt der Ausführung eines JavaScript-Programms, dessen man sich bei bei der ereignisbasierten Programmierung bewusst sein sollte.
In der ereignisbasierten Programmierung mit JavaScript werden vorwiegend Funktionen in Form von *Funktionsausdrücken* verwendet. Im Anhang A.5 befinden sich einige unterschiedliche Erscheinungsformen von Funktionsausdrücken. Häufig werden Funktionen in Variablen gespeichert (vgl. Listing 4.1) oder als Inlinefunktion an eine andere Funktion übergeben.

```
getSweet(„Bonbon“);

var getSweet=function(type){
    console.log(type);
}
// TypeError
```

Listing 4.2: Aufgrund der Aufgabentrennung von Compiler und Engine – anlegen und zuweisen – werden Funktionsausdrücke *nicht* g*ehoistet*.

Solche *Funktionsausdrücke* (vgl. Listing 4.2) werden erst dann geladen, wenn der Interpreter die jeweilige Zeile im Code erreicht. Dies liegt daran, dass die eigentliche Funktion auf der rechten Seite des `=` Operators deklariert wird (vgl. Simpson (Scope), S. 42). Demnach werden Funktionsausdrücke, ähnlich wie beim Beispiel `var a = 2;`, erst bei der Ausführungsphase verarbeitet.

Funktionsdeklarationen (vgl. Listing 4.1) hingegen werden bereits bei der Kompilierung geladen, bevor jeglicher Code ausgeführt wird, egal an welcher Stelle im Code sich ihr Funktionsaufruf befindet. Man spricht hierbei vom *function hoisting* (auf deutsch: *hochheben*). „Function declarations are parsed at a pre-excutive stage, when the browser prepares to execute the code“ [Kantor]. Funktionsdeklarationen sind Standalone-Funktionen ohne Bindung an ein Objekt.

Dass Funktionsdeklarationen zeitlich *vor* allen Wertzuweisungen bzw. Funktionsaufrufen erfasst werden, liegt daran, dass sie von keinem höherrangigen key oder value im Scope abhängig sind, die vom Compiler im Scope *vorher* angelegt werden müssten.

Infolgedessen muss beim Gebrauch von Funktionsausdrücken darauf geachtet werden, dass die Funktionen, anders als in Listing 4.2, erst an einer *späteren* Stelle im Code aufgerufen werden bzw. an ein bestimmtes Ereignis gekoppelt sind (vgl. Kap. 2.3.2.3).

Folgender Abschnitt behandelt, wie JavaScript-Programme im Browser zur Manipulation des DOM verwendet werden. Das ist der klassische Einsatzzweck für Webseiten im WWW[1]. Für Messepräsenzen können dieselben Webtechnologien eingesetzt werden, um Informationen dynamisch zu generieren und grafisch darzustellen. Eine grafische Weboberfläche kann sowohl als Feedback (vgl. Kap. 1.1.2) als auch als Interface (vgl. Kap. 1.1.1) dienen.
Es können auch mobile Endgeräte des Publikums eingebunden werden.

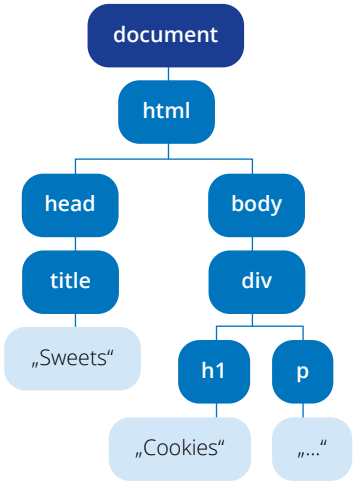
Ab Kap. 2.3 werden einige Konzepte vorgestellt, mit denen JavaScript zur optimalen, ereignisbasierten Programmiersprache wird, womit JavaScript auch außerhalb eines Webbrowsers sein Potenzial ausschöpfen kann.

1 WWW: World Wide Web

2.2. DOM-Manipulation

Das Document Object Model (DOM) beschreibt die Struktur eines HTML-Dokuments, das in einem Browser dargestellt wird.
Im HTML-Dokument wird festgelegt, wie verschiedene Objekte hierarchisch zueinander in Beziehung stehen. Diese Struktur wird von der Rendering-Engine des Browsers zu einer Baumstruktur, dem sogenannten DOM, transformiert.
Jeder Knoten dieses Baums repräsentiert ein HTML-Element.
„Ein Elementobjekt ist die browserinterne Darstellung dessen, was Sie in eine HTML-Datei eingeben." [Castledine]
`document` ist das Wurzelobjekt des DOM. Es steht für die Seite im Browser.

```
<html>
<head>
    <title>Sweets</title>
</head>

<body>
    <div>
        <h1>Cookies</h1>
        <p>...</p>
    </div>
</body>
</html>
```



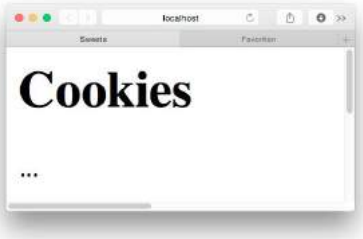


Abb. 5 1:
Oben: HTML-Dokument,
Mitte: Strukturierung im DOM
Unten: Darstellung im Browser

Rolle von JavaScript in Verbindung mit dem DOM

Eine auf JavaScript basierende API[1] kann auf das DOM zugreifen und es manipulieren. Das DOM ist die gemeinsame Anlaufstelle für JavaScript und auch CSS, um reine, durch HTML strukturierte Informationen funktional zu dynamisieren bzw. grafisch zu gestalten.

Mit `document.getElementById("myElement");` wird das Objekt `document` angewiesen, eine Referenz auf das Element-Objekt `myElement` zurückzugeben.
Die JavaScript-Bibliothek jQuery liefert in den meisten Fällen eine verkürzte Schreibweise für den Zugriff auf DOM-Elementobjekte.
`$("#element")` verkürzt beispielsweise den nativen JavaScript-Selektor `document.getElementByID("element")`.

Attribute als Datenspeicher

Elementobjekte können durch Attribute ergänzt werden. In ihnen können Informationen zum jeweiligen DOM-Elementobjekt abgelegt werden. Im folgenden Code-Snippet sind das beispielsweise `id`, `class` und `sweetType`.

```
<div id="uniqueIdentifier" class="allSimilarOnes"
sweetType="Cookie"> Prinzenrolle </div>
```

`id` und `class` sind bekannte Attribute, die einem Elementsobjekt eine einzigartige Identität bzw. die Zugehörigkeit zu einer Gruppe von Elementobjekten definieren. `sweetType` ist hingegen ein selbst definiertes Attribut, das als Speicher von Information dient. „Attribute werden bei der Übergabe an DOM-Methoden durch einen String repräsentiert und können fast nach Gutdünken benannt werden" [Resig & Bibeault, S. 339], z. B. `sweetType`.

Die Nutzung von Attributen als Speicherzellen im DOM bietet sich besonders in jenen Fällen an, wenn in JavaScript beispielsweise die Übergabe von Aktualparametern kompliziert ist. Das ist beispielsweise der Fall, wenn lokal

1 API: „Application Programming Interface"

deklarierte Objekte in anderen Funktionen verwendet werden, denn globale Variablen sollten generell vermieden werden.
Mit Hilfe der JavaScript-Methoden `setAttribute`, `getAttribute` und `removeAttribute` können mit Hilfe von JavaScript Attribute erstellt, ausgelesen und gelöscht werden.

2.3. Ereignisbasierter Ansatz

JavaScript verfolgt einen ereignisbasierten Ansatz und macht dabei von seiner funktionalen Art Gebrauch.
Dieser Abschnitt beschreibt grundlegende Konzepte für die flexible Verarbeitung von Ereignissen in JavaScript. Ein Einsatzbeispiel sind asynchrone Anfragen an einen Webserver durch eine große Anzahl an Clients in einer stark frequentierten Webanwendung oder auf Veranstaltungen.
Anschließend werden einige Formen von Funktionsausdrücken vorgestellt, die häufig in der ereignisbasierten Programmierung vorkommen.

Zunächst wird ein Einblick in den funktionalen Ansatz von JavaScript gegeben, insbesondere den *Objekten erster Klasse*.
Dann werde ich auf das funktionale Konzept der Closures eingehen (vgl. Kap. 2.3.2), das in der asynchronen, ereignisbasierten Programmierung mit Inlinefunktionen allgegenwärtig ist, z. B. im Zusammenhang mit Timern, jQuery-Animationen und Server-Client-Konversationen. Dieses Konzept ist fundamentell für weitere ereignisbasierte Konzepte.

Mit dieser Grundlage wird ab Kap. 2.3.3 die Verarbeitung von Ereignissen in JavaScript näher untersucht. In diesem Zusammenhang werde ich auf die asynchrone Verarbeitung von Ereignissen im Singlethreadmodell mit Hilfe von Callbacks und Callbackfunktionen eingehen.

2.3.1. JavaScript als funktionale, objektorientierte Programmiersprache

JavaScript ist wie z. B. Python, Haskell und CoffeeScript eine funktionale Programmiersprache, anders als z. B. C, C++ oder Java. In einer funktionalen Programmiersprache ist „sämtlicher Scriptcode in Funktionen enthalten“ [Resig & Bibeault, S. 64]. Funktionales Programmieren beschreibt eine Herangehensweise, bei der ein Problem in mehrere Funktionen zerlegt wird. Man verwendet Funktionen zur Abstraktion der Programmlogik. Funktionen enthalten die eigentliche Programmlogik und können z. B. in eine Bibliothek ausgelagert werden.
Im Hauptcode können Funktionen je nach Anforderung ineinander verschachtelt oder als Aktualparameter an andere Funktionen übergeben werden. Dieser Ansatz ist beim schnellen Prototyping, bei veränderlichen Projektanforderungen und besonders bei der Verabeitung von Ereignissen im

Vorteil gegenüber einem nichtfunktionalen Ansatz, wie z. B. in Java.
JavaScript unterstützt Objekte erster Klasse (vgl. Kap. 2.3.1.1) und ist somit eine funktionale Programmiersprache.
Indem bei funktionalen Programmiersprachen die komplette Programmlogik in Funktionen ausgelagert werden kann, müssen nicht, wie z. B. in Java, größere Extraklassen erzeugt und instanziiert werden. Der Vorteil beim Ansatz der Klassenhierarchie ist, dass bereits zu Beginn eines Softwareprojekts die spätere Funktionalität der Software bekannt ist. Beim funktionalen Ansatz hingegen bleibt eine große Flexibilität vorhanden (vgl. Nolte).

Der funktionale Ansatz begünstigt die Verschachtelung von Funktionen ineinander. Ein Beispiel für den funktionalen Ansatz bei der Verarbeitung von Arrays ist die Funktion `forEach`, die keine native Funktion von JavaScript ist, aber von einigen Bibliotheken[1] bereitgestellt wird (vgl. Listing 6). Sie ruft für jedes Element innerhalb eines Arrays eine Callbackfunktion auf. Im Innern der Funktion `forEach` operiert i. d. R. eine lange und somit unübersichtliche for-Schleife (vgl. Resig & Bibeault, S. 94). Dank des funktionalen Ansatzes bleibt der eigentliche Funktionsinhalt dem Entwickler gegenüber weitgehend verborgen.

```
function forEach(list,callback) {
    for (var n = 0; n < list.length; n++) {
        callback.call(list[n],n);
    }
}
```

Listing 6: Demonstration einer Verschachtelung von Funktionen ineinander am Beispiel der Methode `forEach`

2.3.1.1. Funktionen als Objekte erster Klasse

A function „is a subtype of Object (technically, a callable object)“ [Simpson (this), S. 36].

- Funktionen sind echte JavaScript-Objekte. Sie sind Instanzen[2] des Objekts `Function`.
- Funktionen können Variablen, also anderen Objekten, zugewiesen werden.
- Funktionen können als Aktualparameter[3] in Form von benannten oder anonymen Inlinefunktionen übergeben oder als Rückgabewert zurückgegeben werden.
- Funktionen können weitere Funktionen und Objekte beinhalten da sie selbst Objekte sind.

1 JavaScript-Bibliotheken, die eine Methode `forEach` bereitstellen, sind u. a. jQuery, underscore.js und d3.
2 Instanzen = Exemplare (vgl. Schäfer)
3 Unterschied zwischen Aktualparametern und Formalparametern:
Aktualparameter sind Daten, die einer Funktion beim Funktionsaufruf an der Senderseite übergeben werden. Da Funktionen zur Wiederverwendung zu unterschiedlichen Zwecken gedacht sind, können individuelle Spezifikationen mit Aktualparametern festgelegt werden, z. B. `tuWas("Ahoi");`. `Ahoi` ist ein Aktualparameter
Formalparameter nehmen Aktualparameter auf der Empfängerseite entgegen (vgl. Freeman & Robson, S. 123) und sind Platzhalter für diese Daten,
z. B. `function tuWas(param){..};`. `param` ist ein Formalparameter.

- Funktionen sind Objekte erster Klasse. Sie haben „die Fähigkeit, aufgerufen zu werden“ (vgl. Resig & Bibeault, S. 61 und 97).
- Funktionen können (wie alle Objekte) mittels Literalschreibweise[1] erstellt werden, z. B. `function myFunctionName();`. `function` ist ein reservierter Wert.
- Funktionen können (wie alle Objekte) zur Laufzeit dynamisch verändert werden und Eigenschaften besitzen, die dynamisch erstellt und zugewiesen werden können.

2.3.1.2. Ausgewählte Funktionsausdrücke für die ereignisbasierte Programmierung

Im Bereich Event Media sind Ereignisse allgegenwärtig. Bei JavaScript unterscheidet man u. a. zwischen *Browser Events*[2], *User Events*[3], *Server Events*[4] und weiteren *JavaScript Events*[5].
Im Folgenden wird eine Auswahl an Funktionsausdrücken näher beschrieben, die sich für die eventbasierte Programmierung besonders eignen. Eine umfassende Liste an Funktionsaudrücken mit Beispielen befindet sich im Anhang A.5.

Inlinefunktionen

Eine Inlinefunktion wird „üblicherweise für den späteren Gebrauch erzeugt“ [Resig & Bibeault, S. 100]. Häufig ist sie als Eventhandlerfunktion an eine Eventlistenerfunktion[6] gekoppelt. Sie wird der Eventlistenerfunktion als Aktualparameter übergeben. Somit ist die Inlinefunktion eine Callbackfunktion. Sie wird als innere Funktion innerhalb einer Eventlistenerfunktion notiert, die aufgerufen wird, sobald das jeweilige Ereignis eintritt. Bei Inlinefunktionen handelt es sich um *Closures*[7].
Bei Inlinefunktionen unterscheidet man zwischen *benannten* und *unbenannten (anonymen)* Funktionen.

Am häufigsten werden kurze und überschaubare *anonyme* Inlinefunktionen eingesetzt (vgl. Listing 2a im Anhang A.5). Da sie meistens direkt zu einem Eventlistener gehören, wird eine explizite Benennung in den meisten Fällen überflüssig.
Benannte Inlinefunktionen (siehe Listing 2b im Anhang A.5) eignen sich besonders für den Fall, dass sich eine Inlinefunktion mit ihrem Namen *selbst* rekursiv aufrufen muss. Diese Vorgehensweise bietet sich in komplexer werdenden Codepassagen an, bei denen ein nachträgliches Einarbeiten einer Schleife in den bestehenden Code vermieden werden sollte. Ein rekursiver Funktionsaufruf wird anlässlich der Projekteplattform (vgl. Kap. 4.3.2) erprobt.

---

1 Literalschreibweise: Abgekürzte Schreibweise, z. B. zur Erstellung von Objekten
2 Browser Events: Z. B. Ladeprozess einer Webseite (`onload`), Formular-Events
3 User Events: Z. B. Mouse-Events (`onclick`), Tastatur-Events, Audio/Video-Steuerung
4 Server Events: Z. B. AJAX-, WebSocket-Verbindungen
5 JavaScript Events: Z. B. Timer (`setTimeout, setInterval`)
6 In Listing 4.2 ist `$(document).ready(..)` eine Eventlistenerfunktion. Sie wartet darauf, bis eine Webseite geladen wurde, bevor die in ihr verschachtelte(n) Funktion(en) ausgeführt werden.
7 Closures: Referenzen innererer Funktionen auf den Funktionsscope ihrer umgebenden Funktion, vgl. Kap. 2.3.2.2.

Selbstaufrufende Funktionen

Selbstaufrufende Funktionen werden hauptsächlich zur Kapselung von Funktionen verwendet, z. B. wenn Code in Bibliotheken ausgelagert wird[1]. Indem sich die Funktionen selbst aufrufen, müssen diese Funktionen nicht mehr explizit aufgerufen werden. In einem HTML-Projekt genügt es beispielsweise, die in eine separate JavaScript-Datei ausgelagerte Bibliothek mit einem Einzeiler-Codesnippets im `<head>` in das HTML-Projekt zu laden.
Ein weiteres Einsatzgebiet für selbstaufrufende Funktionen ist die Simulation eines *Block Scopes*[2] in JavaScript, was vor ECMAScript 6 nur über Funktionsscopes möglich ist. Block Scopes setzen das Prinzip der Closures voraus und werden dort näher beschrieben.

2.3.1.3. Funktionen ausführen

Funktionsdeklarationen und Funktionsausdrücke können folgendermaßen aufgerufen werden:

- Über reguläre **Funktionsaufrufe** wie `myFunction();`
- Über **Funktionsreferenzen**, insbesonsere Callbackfunktionen, die eine Funktion abhängig von einem Ereignis zu einem späteren Zeitpunkt auslösen.
- Eine Ausnahme sind **selbstaufrufende Funktionen** in der Form `(function(){..})();`, die nicht explizit aufgeufen werden müssen.

Funktionsreferenzen können verschiedene Erscheinungsformen annehmen. Oft werden sie in Form von Inlinefunktionen als Callbackfunktion verwendet. Hierbei werden Funktionsreferenzen häufig in Verbindung mit Closures verwendet.
Auf das Konzept der Closures, mit dem viele Probleme elegant gelöst werden können, werde ich im folgenden Kapitel näher eingehen.

1 Viele moderne JavaScript-Libraries, wie auch jQuery, befinden sich innerhalb der Klammern einer sich selbst aufrufenden Funktion.
2 Scope: Funktionsweiter Verfügbarkeitsbereich von Objekten, vgl. Kap. 2.3.2.1

2.3.2. Closures

Closures sind ein Sprachkonstrukt funktionaler Programmiersprachen, in denen Funktionen ineinander verschachtelt werden können und die Gültigkeit von Variablen über Lexical Scoping verwaltet werden kann (vgl. Simpson (Scope), S.48, Freeman & Robson, S. 409).

> *Eine Closure ist die Referenz einer inneren Funktion auf den Scope der umgebenden Funktion.*

Für das Verständnis von Closures wird das Prinzip des Lexical Scoping vorausgesetzt.
Anders als mit Java kann mit JavaScript der Sichtbarkeitsstatus von Objekten im Programm *nicht* durch Schlüsselworte, wie `public`, `private` oder `privileged`, reguliert werden, sondern mittels *Lexical Scoping*.

2.3.2.1. Lexical Scoping

JavaScript-Variablen sind nur innerhalb jener Funktion sichtbar, in der sie erstellt werden. Dieser Bereich wird *Scope* genannt. Die Gültigkeit von Variablen beschränkt sich auf den Bereich der Funktion, in der sie angelegt werden. Der Funktionsscope beginnt und endet mit den Grenzen der Funktion. Aus diesem Grund ist ein Funktionsscope der Verfügbarkeitsbereich[1] von Variablen und Funktionen. Demnach werden Schlüsselworte, wie `public`, `private` oder `privileged` nicht benötigt.
Funktionsscopes sind ebenso verschachtelt wie die Funktionen, an die sie gebunden sind. Zusammengefasst: „Scope is the set of variables you have access to." [w3cSchools (Scope)]

```
function foo(a) {
    var b = a * 2;
    function bar(c) {
        console.log( a, b, c );
    }
    bar(b * 3);
}
foo( 2 ); // 2, 4, 12
```

Listing 7: Veranschaulichung von Funktionsscopes

Der Bereich *1* in Listing 7 ist der Scope der *globalen Funktion*[2] und wird *Global Scope* genannt. Bereich *2* ist der Scope der Funktion `foo`. *3* ist der Scope der Funktion `bar`.

Innerhalb einer Funktion sind nur jene Objekte verfügbar, die im Scope derselben Funktion oder in einem übergeordneten Funktionsscope deklariert wurden. In Listing 7 können innerhalb des Funktionsscopes *2* die Objekte aus Scope *1* und *2* gesehen werden.
Objekte aus anderen Funktionsscopes, einschließlich den Scopes jener Funktionen, die ihren Ursprung zwar in der aktuellen Funktion haben, jedoch noch tiefer verschachtelt sind, sind *nicht* sichtbar. In Listing 7 sind Objekte aus Scope *3* innerhalb des Funktionsscopes *2* nicht verfügbar.
Die Suche nach Objekten, zunächst im aktuellen, dann in den übergeordneten Funktionsscopes, ähnelt dem Nachschlagen in einem Lexikon und wird *Lexical Scoping*[3] genannt.

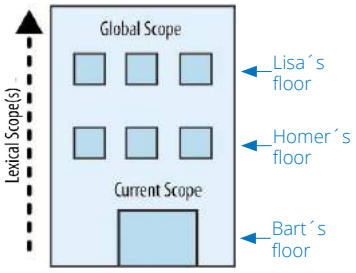
---

1 Der Verfügbarkeitsbereich (Scope) wird auch als *Gültigkeitsbereich* (vgl. Schäfer) oder *Geltungsbereich* (vgl. Flanagan) bezeichnet.
2 Die globale Funktion heißt im clientseitigen JavaScript `window` und bei Node.js `global`.
3 Lexical Scoping bezeichnet man auch als *Static Scoping* (vgl. Flanagan, S. 155).

„Funktionen [...] laufen in dem [Verfügbarkeitsbereich], in dem sie definiert wurden und nicht in dem sie ausgeführt werden." [Flanagan, S. 155]
Ein lexikaler Scope kann mithilfe von visuellen Orientierungspunkten, wie z. B. Codeeinrückungen, mit dem bloßen Auge nachvollzogen werden (vgl. Freeman & Robson, S. 488).
Das charakteristische Merkmal des *Lexikal Scoping* ist, dass der Scope bereits beim Erstellen des Codes festgelegt wird. Einen dynamischen Scope hingegen gibt es in JavaScript nicht. Eine Art dynamischer Scope wird mit dem Bezeichner `this` hergestellt, der für den Kontext einer Funktion steht. „*Dynamic Scope* doesn´t concern about itself with how and where functions and scopes are declared, but rather where they are called from. [...] The `this` mechanism [by contrast] is kind of the dynamic scope." [Simpson (Scope), S. 66]
Der Kontext und der Bezeichner `this` werden in Kurze näher untersucht.

Für eine JavaScript-Engine bedeutet Lexical Scoping, dass sie zur Auflösung von Variablennamen zunächst den aktuell ausgeführten Funktionsscope untersucht. Ist die Suche erfolgreich, wird die Suche beendet. Andernfalls sucht die Engine im übergeordneten Scope weiter.
Im Funktionsscope *3* in Listing 7 werden `a` und `b` zunächst nicht gefunden; erst nachdem die übergeordneten Funktionsscopes *2* und *1* nach `a` und `b` durchsucht wurden.

Abb. 8 stellt übergeordnete Funktionsscopes mit höher liegenden Stockwerken dar. Hat sich die Engine durch die verschachtelte Funktionsstruktur bis zum obersten Funktionsscope, dem *globalen Scope*, durchgearbeitet und findet das gesuchte Objekt auch dort nicht, erhält sie den Wert `undefined`. Diese rekursive Vorgehensweise erklärt, warum lokale Objekte im aktuellen Funktionsscope immer Priorität vor Objekten in weiter außen liegenden Funktionsscopes haben, wie auch das Listing zu Abb. 8 zeigt: Da bei der Ausführung von `alert(..)` für den Wert `name` zunächst `Bart` gefunden wird, wird *nicht* weitergesucht. Falls `Bart` nicht verfügbar ist, wird `Homer` ausgegeben, ansonsten `Lisa` (globaler Scope).




```
var name = "Lisa";
function secondFloor() {
    var name = "Homer";
    function firstFloor(){
        var name = "Bart";
        alert(name); //Bart
    }
    firstFloor();
}
secondFloor();
```

Abb. 8: Höher liegende Stockwerke stehen sinnbildlich für übergeordnete Scopes (frei nach Simpson (Scope), S. 10).

Simulation von lokalen, privaten und privilegierten Objekten

Mit dem Prinzip des Lexical Scoping kann der Verfügbarkeitsbereich von Objekten reguliert werden. Listing 9 (nächste Seite) zeigt die drei Szenarien:

Listing 9 (mittlerer Teil) zeigt mit `var name` ein lokales Objekt. Es ist nur innerhalb des Scopes der Funktion `sweet` verfügbar und ist von außen nicht aufrufbar. Es ist also `private`.

Listing 9 (linker Teil) zeigt mit `name` (es funktioniert auch `this.name`) ein globales Objekt. Variablen, die ohne das Schlüsselwort `var` deklariert werden, werden zu globalen Variablen. Sie sind also `public`.

```
// PUBLIC

function sweet() {
    name = 'Bonbon';
}

sweet();
console.log(name); // Bonbon
```

```
// PRIVATE

function sweet() {
    var name = 'Bonbon';
}

sweet();
console.log(name); // undefined
```

```
// PRIVILEGED

function sweet() {
    var name = "Bonbon";
    getName = function () {
        return name;
    }
    return getName;
}

sweet();
console.log(getName); // Bonbon
```

Listing 9: Simulation von `public`, `private` und `privileged` mit JavaScript

Listing 9 (rechter Teil) simuliert ein privilegiertes Objekt, indem die nach außen verfügbare getter-Methode[1] `getName` das lokale Objekt `name` nach außen verfügbar macht. Getter- und Setter-Methoden sind funktionale Beispiele für Scope Closures, einem besonders leistungsfähigen Konzept der funktionalen Programmierung für unterschiedlichste Einsatzbereiche (vgl. Kap. 2.3.2).

Scope vs. Kontext

Auf der vorigen Seite wurde angemerkt, dass der Scope und der Kontext zwei verschiedene Dinge sind. Beide Konzepte sind in der ereignisbasierten Programmierung sehr geläufig, doch man sollte sich des Unterschieds bewusst sein. Dieser soll in diesem Abschnitt erläutert werden.

Der Scope funktioniert *funktionsbasiert*, während der Kontext *objektbasiert* arbeitet. Jeder Funktionsaufruf bezieht sich sowohl auf einen Scope als auch auf einen Kontext.
Der Scope regelt den Zugriff auf alles Gleichwertige und Übergeordnete, während der Kontext genau *das* Objekt ist, auf das der Bezeichner `this` zeigt. Dies ist das besitzende Objekt, von dem aus eine Funktion *aufgerufen* wird (Call-Site) und nicht der Ort, wo die Funktion *deklariert* wurde.
"The object at the call site 'owns' or 'contains' the function reference at the time the function is called." [Simpson (this), S. 14]

```
function foo() {
    console.log( this.a );
}

var a= "außen";

var obj = {
    a: "innen",
    foo: foo
};
obj.foo(); // innen
foo();     // außen
```

Listing 10: Demonstration einer Funktion, die aus verschiedenen Kontexten heraus aufgerufen wird.

In Listing 10 wird die Funktion `foo` zweimal aufgerufen, aus zwei verschiedenen Kontexten heraus. Die rufenden Objekte sind verschieden.
Beim ersten Aufruf ist `foo` an das Objekt `obj` gebunden. Die Call-Site verwendet den Kontext des Objekts `obj`, um `foo` zu referenzieren. In diesem Fall zeigt `this` auf die Funktion `foo` innerhalb des Objektes `obj`.
Der zweite Aufruf der Funktion `foo` ist an kein spezielles Objekt gebunden und bezieht sich standardmäßig auf das globale Objekt.
Der Bezeichner `this` als Platzhalter für ein variables Objekt ist besonders nützlich bei der Wiederverwendung von gleichen Funktionen für unterschiedliche Objekte.

1 Getter und Setter:
Eine Getter-Methode innerhalb eines Funktionsobjekts ist eine Schnittstelle, über die von außen nicht zugängliche Eigenschaften abgerufen werden können.
Eine Setter-Methode kann ein verstecktes Objekt von außen verändern.

Die Verwendung des Bezeichners `this` ist allerdings auch gefährlich, da durch ihn Funktionen in bestimmten Fällen aus ihrem Kontext *gerissen* werden können, wenn sie mit Funktionen wie `eval`, `setTimeout` und `setInterval` verbunden werden: Indem globale Funktionen den eigentlichen Funktionen vorgeschaltet werden, wird das rufende Kontextobjekt gewechselt. Dieses ist nun das globale Objekt `window` (bei clientseitigem JavaScript) oder `global` (bei serverseitigem JavaScript mit Node.js).
Die `this`-Problematik kann auf verschiedenen Wegen umgangen werden:

- Indem Funktionsaufrufe an die Methoden `apply` oder `call` gebunden werden, kann als Kontext die Referenz auf ein beliebiges Objekt übergeben werden.
- Mit der Methode `bind` kann einer Funktion ein bestimmtes Objekt fest zugeordnet werden
- Durch den Einsatz von *Closures*

Der Einsatz von Closures ist eine elegante Methode, um auf den Bezeichner `this` zu verzichten, denn alle benötigten Variablen aus einer umgebenden Funktion werden mit Hilfe von Closures eingeschlossen (vgl. Schäfer).
Mit dem lexikalen Scope als Grundlage kann das in JavaScript allgegenwärtige Phänomen *Closures* im nachfolgenden Abschnitt näher erläutert werden.

2.3.2.2. Closures

Closures werden eingesetzt um „hauptsächlich die Komplexität des Codes zu reduzieren“ [Resig & Bibeault, S. 155]. Beispielsweise kann mit Closures die zuvor beschriebene `this`-Problematik umgangen werden.
Der Einsatzbereich von Closures ist allerdings vielseitig: Closures werden häufig in der asynchronen Programmierung eingesetzt, oft zur Verarbeitung von Ereignissen. „Closures are workhorses [for example] in Node.js' asynchronous, non-blocking architecture.” [Richard]

Im Schaubild in Abb. 11 (nächste Seite) habe ich eine Auswahl an typischen Einsatzmöglichkeiten von Closures zusammengestellt. Die Verarbeitung von Ereignissen ist *nur ein* Beispiel für den Einsatz von Closures.
Die Beispiele in Abb. 11 können zwar auch *ohne* Closures umgesetzt werden, allerdings erleichtert der Einsatz von Closures die Programmierarbeit erheblich.

2.3.2.2.1. Definition Closures

> *Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope.* [Simpson (Scope), S. 48]

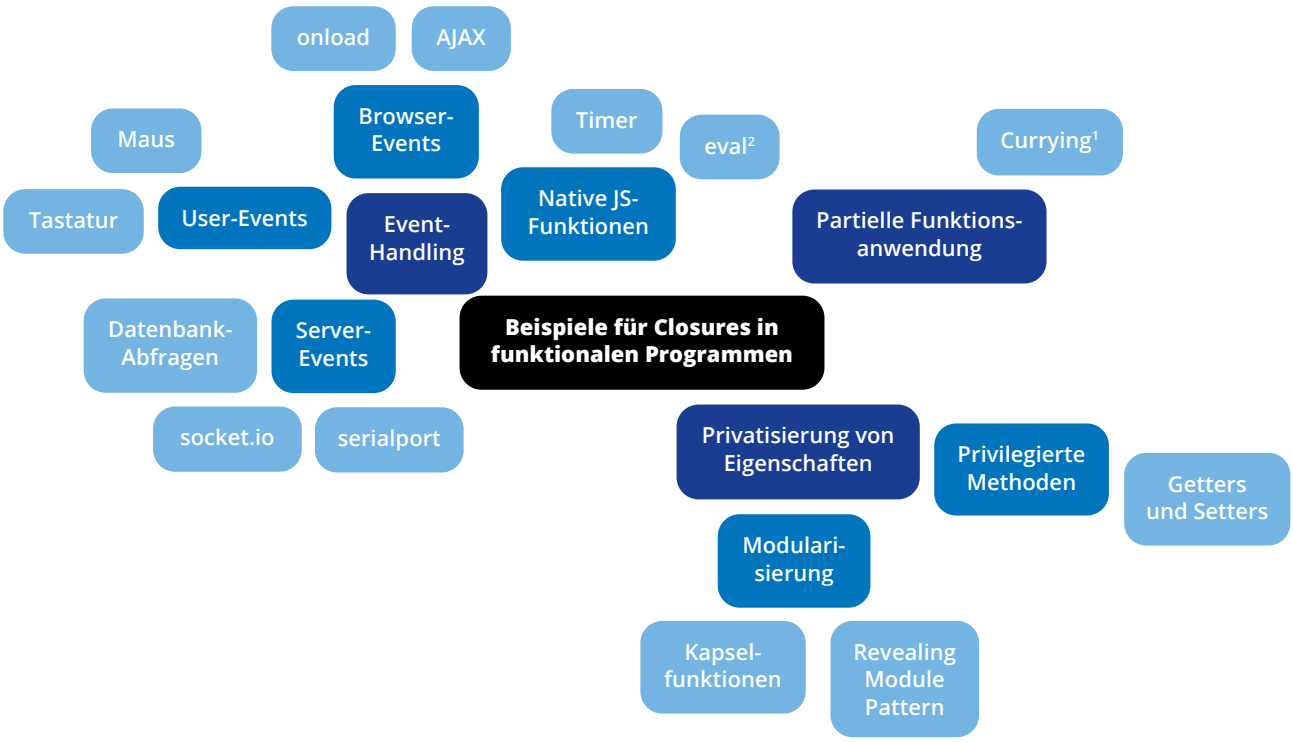


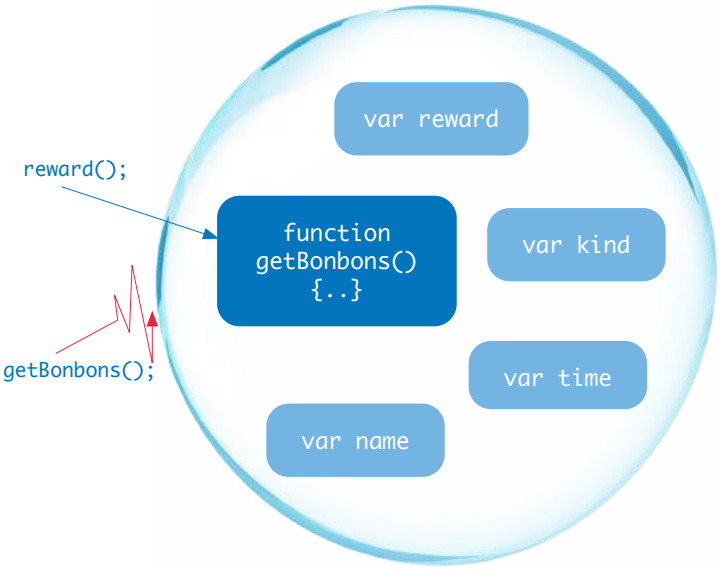
Abb. 11: Einige Einsatzmöglichkeiten von Closures

Eine Closure ist, anders als häufig[3] behauptet, kein anderer Name für eine innere Funktion, sondern ein Synonym für die Referenz einer inneren Funktion in einer verschachtelten Funktionsstruktur auf den Scope der umgebenden Funktion (vgl. Resig & Bibeault, S. 135). Das besondere Merkmal einer Closure ist, dass sie einer inneren Funktion den Zugriff auf alle Inhalte der umgebenden Funktion gestatten, auch dann noch, „wenn die umgebende Funktion und somit der Scope, in dem sie deklarierte wurde, nicht mehr vorhanden ist“ [Resig & Bibeault, S. 136].

„Eine Closure merkt sich die Umgebung in der sie [die innere Funktion] erzeugt wurde.“ [Schäfer] Dass innerhalb von Funktionen *nicht nur* auf die eigenen Eigenschaften zugegriffen werden kann, liegt in der Natur des Lexical Scoping (vgl. Stockwerke in Abb. 11). Die innere Funktion erhält eine Referenz auf den Scope der umgebenden Funktion, weiterhin aufrecht (Sie erhält eine *Closure weiterhin aufrecht*). „[...] it will maintain a scope reference to where it was originally declared, and wherever we execute him, that closure will be executed.“ [Simpson (Scope), S. 51]

Bildlich kann man sich eine Closure wie eine schützende Blase vorstellen, die die Variablen der inneren Funktion und die Variablen ihrer Funktionsumgebung konserviert und „solange vor der Speicherbereinigung bewahrt, wie die Funktion existiert“ [Resig & Bibeault, S. 138] (vgl. Abb. 12).

1 Currying ist eine Form der partiellen Funktionsanwendung, bei der eine Funktion, die mehrere Aktualparameter enthält, in eine andere Funktion eingesetzt wird, die nur ein einziges Aktualparameter entgegennimmt und eine andere Funktion zurückgibt, die die ausstehenden Aktualparameter entgegen nimmt (vgl. Richardson)
2 `eval`: Dynamische Codegenerierung mit Hilfe von Zeichenketten
3 vgl. MDN (Closures)




```
var time = “at the break”;

function getSweets() {
    var kind = ‘bonbons ‘;
    function getBonbons() {
        var number = 5;
        console.log(number + kind + time);
    }
    return getBonbons;
}
var reward = getSweets();

reward();      // 5 bonbons at the break
getBonbons(); // ReferenceError
```

Abb. 12: Die Blase stellt die Referenz (=*Closure*) auf alle Objekte der Funktionen getSweets und getBonbons sowie auf globale Objekte grafisch dar.

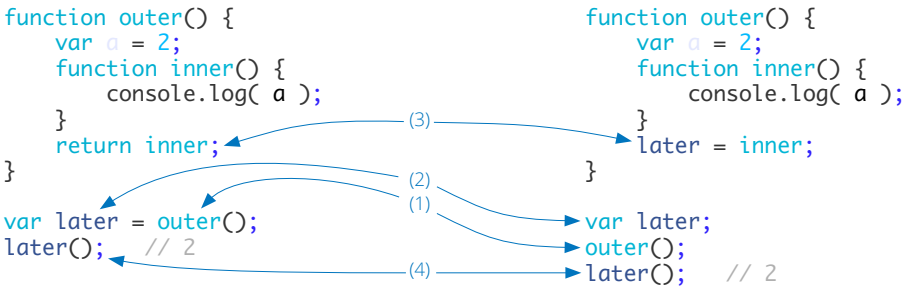
2.3.2.2.2. Begriff *Closure*

Closure wird häufig[1] als *Funktionsabschluss* ins Deutsche übersetzt. Der Begriff *Abschluss* ist mehrdeutig und bedarf einer Erläuterung: Während ein *Abschluss* für *Ende* oder für das *Einschließen* von Elementen stehen kann, trifft Letzteres zu (siehe Blase in Abb. 12).
Im Englischen lässt sich der Begriff *Closure* besser erklären:
„[`getBonbons`] (in Abb. 12) closes over the scope of [`getSweets`]“ (frei nach Simpson (Scope), S. 49).
Auf deutsch etwa: Die innere Funktion `getBonbons` schließt den Scope einer Funktion `getSweets`, die sie umgibt, mit ein. Er wird sozusagen konserviert. Sinnbildlich befinden sie sich gemeinsam in einer schützenden Blase.

2.3.2.2.3. Angewandte Funktionsweise von Closures

Das praktische Beispiel in Listing 13 (nächste Seite) soll zeigen, wie ein Zugriff von außen auf eine innere Funktion zustande kommt, auch wenn die besitzende, umgebende Funktion schon gar nicht mehr existiert bzw. wie die innere Funktion ihre umgebende Funktion referenzieren kann, selbst wenn jene nicht mehr existiert.

1 vgl. z. B. MDN (Closures)



Listing 13: Erstellung einer Scope Closure.
Links: Mithilfe eines return Statements (vgl. Simpson (Scope), S. 49, Freeman & Robson, S. 499)
Rechts: Ohne die Verwendung eines return Statements: (vgl. Simpson (Scope), S. 51, Resig & Bibeault, S. 137)

- Zunächst wird die umgebende Funktion `outer` aufgerufen (*1*).
- Dort wird die in ihr deklarierte innere Funktion `inner` im Bereich außerhalb von `outer`[1] in der Variablen `later` (*2*) abgespeichert.
  Listing 13 zeigt hierfür zwei Möglichkeiten:

  A: Im linken Teil von Listing 13 wird die innere Funktion `inner` von der umgebenden Funktion `outer` als Rückgabewert (*3*) zurückgegeben und wird mit dem Aufruf von `outer` (*1*) automatisch in der Variablen `later` (2) abgespeichert.

  B: Im rechten Teil von Listing 13 hingegen wird `inner` von innerhalb der umgebenden Funktion `outer` aus explizit in der globalen Variablen `later` abgespeichert (*3*).

  Listing 13 (linker Teil) fasst im Gegensatz zu Listing 13 (rechter Teil) die Schritte *1* und *2* zusammen.
- Die innere Funktion `inner` inklusive ihr Funktionsscope und der Funktionsscope der umgebenden Funktion `outer` bleiben somit „zu einem [beliebigen] späteren Zeitpunkt“ [Resig & Bibeault, S. 137] mit dem Aufruf des Funktionsausdrucks `later();` (*4*) erreichbar.

Eine Closure, also eine Referenz von `inner` auf den Scope von `outer`, wird geschaffen, indem die innere Funktion `inner` in der globalen Variable `later` agespeichert wird. Dadurch wird verhindert, dass der Scope von `outer` nicht direkt nach der Beendigung der Funktion `outer` durch den Garbage Collector entsorgt wird.

In diesem Abschnitt wurden der lexikale Scope und Closures eingeführt, die in der ereignisbasierten Programmierung mit JavaScript eine tragende Rolle spielen.
Im nächsten Abschnitt werden mit Click-Handlerfunktionen und Timer-Handlerfunktionen einige Einsatzszenarien von Closures, speziell zur Lösung von Problemen bei der Behandlung von Ereignissen, vorgestellt.

1 Der Scope außerhalb der umgebenden Funktion ist im Beispiel von `foo` der globale Scope.

2.3.2.3. Konkrete Anwendungsbeispiele für Closures in der ereignisbasierten Webentwicklung

Bei der Verarbeitung von Ereignissen in Webanwendungen werden Eventhandlerfunktionen[1] benötigt, die nach dem Eintreten eines bestimmten Ereignisses aufgerufen werden. Der Trend neigt dazu, Ereignisse in verschachtelten Eventhandlerfunktionen[2] unter dem Einsatz von Closures zu behandeln. In einigen Fällen, wie bei jQuery-Animationen und bei der Server – Client-Kommunikation mit Node.js, handelt es sich bei der inneren verschachtelten Funktion um eine *asynchrone* Eventhandlerfunktion.
In der Praxis werden Closures allerdings selten, wie es in Listing 13 der Fall ist, in globalen JavaScript-Variablen zwischengespeichert, sondern z. B. direkt im DOM. Das Prinzip der Closures bleibt dabei erhalten.

Im Folgenden werden typische Einsatzszenarien für Closures aus der ereignisbasierten Webentwicklung exemplarisch beschrieben (vgl. Übersicht auf Abb. 11).

2.3.2.3.1. Click-Handlerfunktion

Beim Click auf ein HTML-Element, wie z. B. auf den Button `#b`, wird der `click`-Funktion eine Eventhandlerfunktion zugewiesen, die im Falle eines Click-Events aufgerufen wird (vgl. Listing 14). Intern wird diese Funktion direkt im `onclick`-Attribut im DOM gespeichert. „Der `onclick`-Eigenschaft des Buttons wird somit eine Closure zugewiesen“ [Freeman & Robson, S. 504]. Der Browser erstellt eine Closure für die anonyme Click-Handlerfunktion, die dem `onclick`-Attribut des Buttons zugewiesen wurde.
Die Click-Handlerfunktion hat eine Closure über den Funktionsscope der umgebenden Funktion `ready` und kann zu jedem späteren Zeitpunkt auf die Objekte innerhalb dieser Umgebung zugreifen, ohne dass es zu Namenskonflikten mit den Objekten `nr` und `msg` kommt:
Im DOM wird quasi ein Event-Listener angelegt, der dort während der gesamten Lebenszeit des Buttons auf Click-Events auf diesen Button wartet.
In diesem Fall wird die entsprechende Eventhandlerfunktion veranlasst.
Dabei spielt es keine Rolle, ob für den Button ein JavaScript-Objekt angelegt wurde oder nicht.

```
$(document).ready(function(){
    var nr = 0;
    var msg = “Message “;
    $(‘#b’).click(function(){
        count++;
        console.log(msg+nr);
    });
});
```

Listing 14: Click-Handlerfunktion mit einer Closure über den Funktionsscope der umgebenden Funktion

2.3.2.3.2. Timer-Handlerfunktion

Im Zusammenhang mit Timerfunktionen können mit Closures zwei verschiedene Probleme gelöst werden: *Kontextprobleme* und *Modularisierungsprobleme*: Im Folgenden werde ich auf beide Probleme sowie deren Lösungen mit Hilfe von Closures eingehen.

1 Eventhandlerfunktionen und Callbackfunktionen sind dasselbe (vgl. Freeman & Robson, S. 250).
2 Bei verschachtelten Eventhandlerfunktionen handelt es sich entweder um benannte oder anonyme Inlinefunktionen.

```
function Sweet() {
    this.type = 'Bonbon';
    this.getSweet = function() {
        console.log(this.type);
    }
}

var mySweet = new Sweet();

setInterval(
    mySweet.getSweet
    ,3000);

// undefined (every 3s)
```

```
function Sweet() {
    this.type = 'Bonbon';
    this.getSweet = function() {
        console.log(this.type);
    }
}

var mySweet = new Sweet();

setInterval(function(){
    mySweet.getSweet()
},3000);

// Bonbon (every 3s)
```

```
function Sweet() {
    this.type = 'Bonbon';
    var self = this;
    this.getSweet = function() {
        console.log(self.type);
    }
}

var mySweet = new Sweet();

setInterval(
    mySweet.getSweet
    ,3000);

// Bonbon (every 3s)
```

Listing 15: Links: Timer *reißen* Funktionen aus ihrem Kontext, Mitte: Lösungsvorschlag mit einer Proxyfunktion, rechts: Lösungsvorschlag mithilfe einer Closure.

a) Lösung von Kontextproblemen

Wird eine Eventhandlerfunktion mit einer Timerfunktion, beispielsweise mit `setInterval`, verbunden, wird die Eventhandlerfunktion, die an die Funktion `setInterval` gekoppelt ist, als Aktualparameter überreicht. Somit ist die Eventhandlerfunktion eine innere, in die äußere Funktion `setInterval` verschachtelte Funktion. Demzufolge ist die Closure die Referenz der Eventhandlerfunktion auf den Scope der umgebenden Funktion `setInterval`. `setInterval` hält an der Eventhandlerfunktion fest. Die DOM-API des Webbrowsers ruft sie 1000 ms später auf (vgl. Listing 15, Freeman & Robson, S. 501).
Besonders bei Timer-Funktionen wie `setInterval` ist es wichtig, dass eine verschachtelte Funktionsstruktur und Closures angewendet werden. Da `setInterval` eine Funktion des globalen Objekts `window`[1] ist, geht bei jedem Aufruf der Eventhandlerfunktion der Kontext des aufrufenden Objekts verloren. Listing 14 veranschaulicht die Bindung von Funktionen an bestimmte Objekte. Bei jeder Funktionswiederholung mit `setInterval` wird als Funktionskontext das globale Objekt `window` statt das rufende Objekt verwendet. Die `this`-Problematik wurde bereits auf Seite 33 vorgestellt. Wenn der Zähler in der web API abläuft (vgl. Abb. 24 auf Seite 55), ruft die JavaScript-Engine eine native Funktion mit dem Kontextobjekt `window` auf. Diese wiederum ruft die Eventhandlerfunktion auf. Nun ist das rufende Objekt allerdings `window` und nicht mehr das Objekt, von dem aus die Funktion `setInterval` aufgerufen wurde. Die in `setInterval` erstellte Eventhandlerfunktion wurde aus ihrem ursprünglichen Kontext *gerissen*.

Abb. 15 (linker Teil) demonstriert diesen Umstand:
Die Methode `setInterval` ruft die Funktion `getSweet` alle drei Sekunden auf, allerdings zeigt `this.type` auf das Objekt `window` statt auf das Objekt `mySweet` und kann dort kein Objekt namens `type` finden.

In Abb. 15 (mittlerer Teil) vermittelt bei jedem Timerablauf eine jeweils neue anonyme Funktion zwischen der Timerfunktion und dem Aufruf der Funktion `getSweet`. Sie fungiert als Zwischenschicht, die vor jedem erneuten Funktionsaufruf den ursprünglichen Kontext wiederherstellt.

1 Timer-Funktionen sind Methoden des globalen Objets: Clientseitig: `window`, serverseitig: `global` (vgl. Schäfer).

In Abb. 15 (rechter Teil) wird das Beibehalten des ursprünglichen Kontexts mit Hilfe von Closures geregelt: Bereits bei der Erzeugung des Objekts `mySweet` mit der Konstruktorfunktion `Sweet` wird mittels `var self = this;` eine Referenz auf das besitzende Objekt in `self` gespeichert. Sobald die Funktion `getSweet` erstellt wird, erhält sie eine Referenz (=Closure) auf den Scope der umgebenden ehemaligen Konstruktorfunktion `Sweet`. Dank des Konzepts der Closures zeigt `self.type` bei späteren Aufrufen von `mySweet.getSweet` weiterhin auf das gewünschte Objekt `mySweet`. Das bedeutet: „The lexical scope reference is still intact." [Simpson (Scope), S. 51]

b) Lösung der fehlenden Modularisierung in JavaScript

Listing 17.1 (nächste Seite) zeigt, dass es z. B. bei wiederkehrenden Aufgaben, die durch iterierende Schleifen und `setInterval` geregelt werden, nicht sinnvoll ist, dass sich die Aufgaben innerhalb der Schleife und die besitzende Funktion dieselbe Funktionsumgebung (=Scope) teilen. In diesem Beispiel wird die Variable `i` für *zwei* Aufgaben verwendet: Einerseits wird `i` von der Schleife iteriert und andererseits im zeitlich festgelegten Intervall ausgegeben.
Die drei im Sekundentakt auslösenden Timer-Handlerfunktionen werden nicht nach *jeder* Schleifeniteration gerufen, sondern erst *nachdem* bereits der letzte Schleifendurchlauf vollständig abgeschlossen ist. Aus diesem Grund erhalten die drei anschließenden Callbackfunktionen nur den aktuellsten Wert von `i`, nämlich `3`, und das dreimal.
„All those function callbacks would still run strictly *after* the completion of the loop, and thus print `3` each time. [...] The way scope works, all [three] of those functions, though they are defined separately in each loop iteration, are closed over the same shared global scope, which has, in fact, only one `i` in it" [Simpson (Scope), S. 54].

Beide Aufgaben, Schleifenoperation und Timer-Handlerfunktion, teilen sich einen gemeinsamen Funktionsscope.
Die Schleifeniteration von `i` wartet *nicht* darauf, bis `i` jeweils sekündlich ausgegeben wurde. Das liegt daran, dass die Iteration eines Schleifenloops *nicht* asynchron arbeitet. Sie reiht sich nicht, wie die Timer-Handlerfunktion, in die Task Queue (vgl. Kap. 2.4.2) ein, sondern geht quasi an der Warteschlange vorbei direkt in den Ausführungsstack. Die Schleifeniteration kann direkt im Ausführungsstack ausgeführt werden.
Dieses Phänomen setzt Kenntnisse über den Unterschied zwischen synchroner und asynchroner Programmausführung voraus. In Kap. 2.3.3.2 werde ich dieses Thema näher beleuchten.

Dieses oft unerwünschte Verhalten von JavaScript-Programmen liegt an der fehlenden Modularisierung in JavaScript:
In JavaScript gibt es nur den funktionsweiten Scope, aber keinen blockweiten Scope, dessen Grenzen durch geschweifte Klammern wie `{..}` markiert wären.
Somit verwenden z. B. Schleifen oder verschachtelte Bedingungen denselben Scope wie die Funktion, in der sie sich befinden (vgl. Listing 16).
Diese fehlende Modularisierung führt besonders bei der Arbeit mit Timern

```
if (true) {
    var a = "Ahoi";
}
console.log(a); // Ahoi
```

Listing 16: In JavaScript gibt es keinen blockweiten Scope, sondern lediglich den funktionsweiten Scope.

```
for (var i = 0; i < 3; i++){
    setTimeout(function (){
        console.log(i);
    }, i * 1000);
}
// 3,3,3
```

Listing 17.1: Fehlende Modularisierung

```
for (var i = 0; i < 3; i++){
    (function () {
        var j = i;
        setTimeout(function (){
            console.log(j);
        }, i * 1000);
    })();
}
// 0,1,2
```

Listing 17.2: Zwischenpeichern von `i` in `j`

```
for (var i = 0; i < 3; i++){
    (function (j) {
        setTimeout(function (){
            console.log(j);
        }, i * 1000);
    })(i);
}
// 0,1,2
```

Listing 17.3: Übergabe von `i` als Aktualparameter

```
for (var i = 0; i < 3; i++){
    let j = i;
    setTimeout(function (){
        console.log(j);
    }, i * 1000);
}
// 0,1,2
```

Listing 17.4: Zwischenpeichern von `i` in `let j`

```
for (let i = 1; i < 3; i++){
    setTimeout(function (){
        console.log(i);
    }, i * 1000);
}
// 0,1,2
```

Listing 17.5: Übergabe von `let i` als Aktualparameter

häufig zu unerwarteten Ergebnissen.
Für die Lösung des Problem der kombinierten Schleifeniteration und Timerfunktion wären Block Scopes sehr nützlich. Da diese Möglichkeit in JavaScript nicht gegeben ist, müssen zusätzliche Funktionen gebildet werden, um zusätzliche Umgebungen zu erschaffen. Eine gängige Möglichkeit, um in JavaScript Block Scopes zu simulieren, ist die Verwendung von selbstaufrufenden Funktionen.

Listing 17.2 demonstriert diesen Fall und führt zum zentralen Prinzip der Closures.
„We can take any snippet of code and wrap a function around it“ [Simpson (Scope), S. 28]. Wird um die Timerfunktion eine selbstaufrufende Funktion gelegt, wird bei jedem Schleifendurchlauf ein neuer Funktionsscope erstellt. Dort wird bei jeder Schleifeniteration eine Kopie von `i` in einer Variablen `j` konserviert.
Der Zugriff auf die Werte von `j` bleibt auch bei der später im Sekundentakt erfolgenden Abarbeitung der folgenden drei asynchronen Timer-Handlerfunktionen erhalten.
Grund dafür ist, dass *jede* der drei späteren Timer-Handlerfunktionen eine Referenz auf den Scope der umgebenden Funktion – der selbstaufrufenden Funktion – zum Zeitpunkt der Erstellung der jeweiligen Callbackfunktion – aufrecht erhält. Diese Referenz wird *Closure* genannt.

Listing 17.3 zeigt eine Variante von Listing 17.2. Die Speicherung des jeweils aktuellen Werts für `i` erfolgt jeweils als Übergabe eines Aktualparameters. Hierbei wird automatisch eine Kopie von `i` im jeweils neuen Funktionsscope angelegt.

Listing 17.4 und 17.5 zeigen einen Ansatz, mit dem ab ECMA Script 6 unabhängig von Funktionen Block Scopes simuliert werden können: Für die Deklaration von Variablen in einem blockweiten Scope wird neben dem Bezeichner `var` zusätzlich `let` eingeführt. Mit `let` lassen sich Variablen dekla-

rieren, deren Gültigkeit auf einen Block Scope begrenzt ist. „The variable will be declared not just once for the loop, but each iteration“ [Simpson (Scope), S.56]. In diesem Fall wird bei jedem Schleifendurchlauf eine neue Variable `let j` (in Listing 17.4) bzw. `let i` (in Listing 17.5) angelegt, die beim Abarbeiten der Timer-Handlerfunktionen über je eine Closure erreichbar bleibt.

Im nächsten Abschnitt werde ich auf die Verarbeitung von Ereignissen näher eingehen. Tritt ein Ereignis ein, zeigt eine Funktionsreferenz auf die Funktion, die *danach* ausgeführt werden soll. Funktionsreferenzen können, außer mit ihrer Grundform (vgl. Listing 18 auf der nächsten Seite, oberer Teil), auch mit Hilfe von Inlinefunktionen elegant gelöst werden. In diesem Fall wird das Phänomen der Closures genutzt.

2.3.3. Verarbeitung von Ereignissen

Eventhandlerfunktionen sind Funktionen, die beim Eintreten eines Ereignisses ausgeführt werden. Ein anderer Name für eine solche Eventhandlerfunktion (oder auch *Eventhandler* oder *Handler*) ist *Callbackfunktion* (vgl. Freeman & Robson, S. 250).
In diesem Abschnitt werden Funktionsreferenzen eingeführt, mit denen solche Eventhandlerfunktionen gerufen werden. In der ereignisbasierten Programmierung handelt es sich dabei um asynchrone Funktionen. Die asynchrone Programmierung wird in Kap. 2.3.3.2 näher betrachtet. In diesem Zusammenhang wird in Kap. 2.3.3.3 auf Callbacks und Callbackfunktionen näher eingegangen. In Kap. 2.4 wird anhand des Singlethreadmodells erklärt, warum es in JavaScript so wichtig ist, dass Ereignisse asynchron verarbeitet werden.

2.3.3.1. Funktionsreferenzen

Funktionen können mit Funktionsaufrufen in der Form `myFunction();`, mit selbstaufrufenden Funktionen in der Form `(function(){..})();` oder über *Funktionsreferenzen* aufgerufen werden.

Funktionsreferenzen sind Zeiger auf eine bestimmte Funktion, die beim Auftreten eines bestimmten Ereignisses aktiv werden. Im Gegensatz zu einem Funktionsaufruf löst eine Funktionsreferenz eine Funktion *nicht* automatisch aus, sondern *erst* beim Eintreten eines bestimmten Ereignisses. Eine Funktionsreferenz ist an eine *Eventhandlerfunktion* gekoppelt. Sie ist ein Hinweis darauf, was passieren soll, sobald ein Ereignis eintritt. Diese Funktionsreferenz nennt man auch *Callback*. Es handelt sich um die Referenz auf die Eventhandlerfunktion, die im Anschluss ausgeführt werden soll. Diese anschließende Funktion ist die sogenannte *Callbackfunktion*.

In der Syntax unterscheidet sich eine herkömmliche Funktionsreferenz in der Form `myFunction` von einem Funktionsaufruf `myFunction()` durch die fehlenden Klammern (vgl. Listing 18 auf der nächsten Seite).

Listing 18: In allen drei Beispielen wird bei einem Click auf einen Button eine Funktion aufgerufen, die das Wort `Ahoi` auf der Konsole ausgibt.

Oben: Eine herkömmliche Funktionsreferenz ruft bei Eintreten des Click-Events die Eventhandlerfunktion `myFunction` auf.

Mitte: Die Eventhandlerfunktion wird in Form einer anonymen Funktion aufgerufen und ersetzt eine herkömmliche Funktionsreferenz. Die anonyme Funktion kann ausschließlich als Eventhandlerfunktion für dieses Click-Event verwendet werden.

Unten: Mit Hilfe der jQuery-Bibliothek wird die Eventhandlerfunktion in einer knappen Schreibweise innerhalb der Click-Funktion verschachtelt.

```
function myFunction(){
    console.log("Ahoi");
}
document.getElementById('myButton').onclick = myFunction;
```

```
document.getElementById('myButton').onclick = function(){
    console.log("Ahoi");
};
```

```
$('#myButton').click(function(){
    console.log("Ahoi");
});
```

Listing 18 (oberer Teil) zeigt eine klassische Funktionsreferenz, mit der eine beliebige Funktion, die im Programm erstellt wurde, aktiviert wird.

Wie Listing 18 (mittlerer und unterer Teil) zeigt, können Funktionsreferenzen auch die Form von *anonymen Funktionen* annehmen. Das bietet sich besonders bei Funktionen an, die ausschließlich für *ein bestimmtes* festgelegtes Event verwendet werden und nicht mit anderen Events geteilt werden können.

In Listing 18 (unterer Teil) befindet sich die Eventhandlerfunktion in einer verschachtelten Funktionsstruktur Es handelt sich um eine Callbackfunktion. Eventhandlerfunktionen und Callbackfunktionen sind dasselbe.
Die Verarbeitung von Ereignissen mit Callbackfunktionen wird in Kap. 2.3.3.3 näher betrachtet. In diesem Zusammenhang genügt es zu wissen, dass die `click`-Funktion der jQuery-Bibliothek eine Funktionsreferenz auf die nachfolgende anonyme Inlinefunktion enthält. Diese Funktionsreferenz wird auch Callback genannt.
Solche Konstrukte werden häufig zur Realisierung der *asynchronen* Verarbeitung von Ereignissen verwendet.
Mit beispielsweise Node.js und der jQuery-Bibliothek jQuery werden viele Ereignisse asynchron behandelt.
Folgender Abschnitt beschäftigt sich näher mit der asynchronen Ausführung von Programmen. In diesem Zusammenhang wird näher auf Callbacks und Callbackfunktionen eingagangen.

2.3.3.2. Asynchrone Programmausführung

Eine asynchrone Programmausführung ist sinnvoll, wenn zeitkritische Aufgaben[1] zu erledigen sind. Dies ist u. a. der Fall, wenn in einer Anwendung mehrere *Einheiten* beteiligt sind, z. B. ein serverseitiges- und clientseitiges Programm sowie möglicherweise eine Datenbank. Sobald eine komponentenreiche Applikation über mehr als eine Einheit verfügt, werden bestimmte Datenaustauschprotokolle zum Senden und Empfangen von Daten benötigt (vgl. Kap. 3: Client – Server-Interaktion).

Eine asynchrone Programmausführung erfolgt z. B. bei AJAX-Anfragen[2] (vgl Kap. 3.2), Datenbank-Abfragen, WebSocketverbindungen, Timer, Animationen und rechenintensive Aufgaben (z. B. Gesichtserkennung).

Funktionsweise

Eine asynchrone Kommunikation zwischen zwei Einheiten, z. B. Client und Server, beschreibt „einen Modus der Kommunikation, bei dem das Senden und Empfangen von Daten zeitlich versetzt und ohne Blockieren des Prozesses durch bspw. Warten auf die Antwort des Empfängers (wie bei synchroner Kommunikation der Fall) stattfindet" (vgl. Wikipedia, AsyncKomm).
Die Funktionsweise der asynchronen Kommunikation soll am Besispiel des Dialogs zwischen Client und Server anhand Abb. 19 veranschaulicht werden.

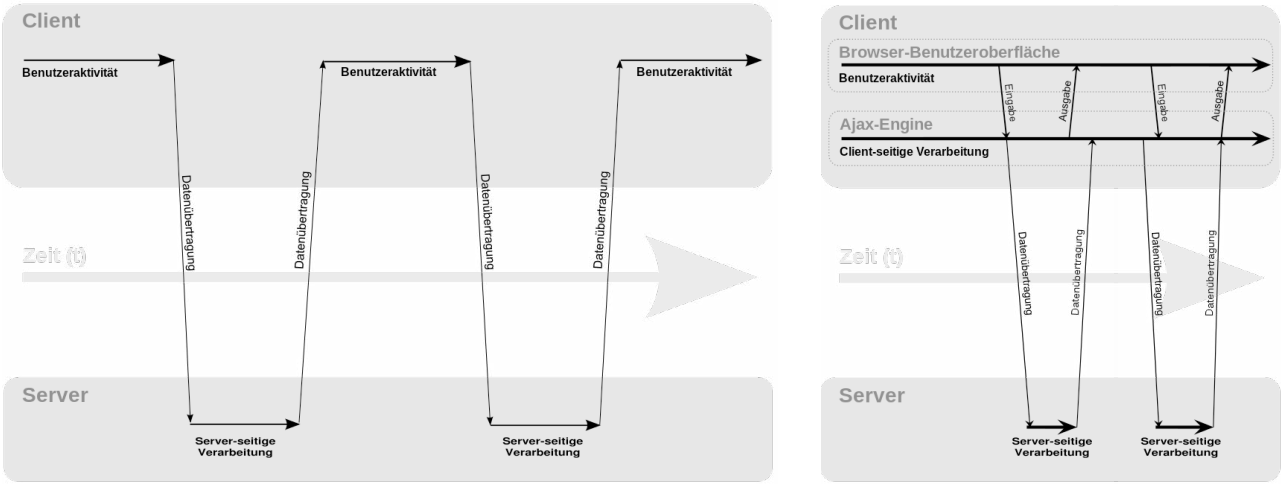


Abb. 19: Links: Synchrone-, rechts: Asynchrone Client-Server-Kommunikation am Beispiel von AJAX-Anfragen

- Bei einer synchronen Anfrage (vgl. Abb. 19 (linker Teil)) an den Server muss der Client solange warten, bis der Server fertig ist und dem Client antwortet. Alle Aufgaben laufen nacheinander ab. In dieser Zeit kann der Client nichts anderes tun. „The browser is blocked. It´s stuck. It

1 Zeitkritische Operationen sind z. B. Datenbank-Operationen, AJAX-Anfragen, jQuery-Animationen.
2 Beispiele für AJAX-Anfragen: Dynamischer Kartenaufbau in Google Maps, Suchvorschläge in Google, echtzeitfähiges Nachladen von Mitteilungen in Facebook ohne Seitenreload

can´t do anything until the request completes.“ [Roberts] Webseiten geraten ins Stocken. Beispielsweise werden nach einem Click auf einen Button, der eine intenive Berechnung auslöst, Animationen unterbrochen und die Website friert ein. Das clientseitige Programm kann erst mit Eintreffen der Antwort des Servers weiter ausgeführt werden. Grund für die Unterbrechung des *gesamten* Programms ist die Tatsache, dass JavaScript-Programme auf *nur einem Thread* (vgl. Kap. 2.4) arbeiten.

- Bei einer *asynchron* Anfrage (vgl. Abb. 19 (rechter Teil)) sendet der Client eine Anfrage an den Server und wartet dann, bis ihm der Server an einem unbestimmten späteren Zeitpunkt antwortet. In dieser Zeit werden andere clientseitige Aufgaben nicht blockert.

In JavaScript könnte ein asynchroner Programmablauf folgendermaßen erfolgen:
Ein Client stellt mit `function1` eine Anfrage an einen Server. Es folgt eine zeitintensive Operation auf dem Server. Sobald der Server fertig ist, ruft dieser mit seiner Antwort über einen sogenannten *Callback* `function2` (=Callbackfunktion) auf. `function2` befindet sich wieder beim Client. Während der Wartezeit auf den Server kann der Client andere Befehle bearbeiten.

Beispielszenarien für Event Media-Applikationen

Im diesem Abschnitt wird auf eine Auswahl an zeitintensiven Operationen eingegangen, die bei Event Media-Applikationen häufig vorkommen:

- eine Datenbankabfrage über Node.js
- eine jQuery-Animation.

Listing 20 zeigt je ein synchrones (linker Teil) und ein asynchrones (rechter Teil) serverseitiges Programm in Node.js, bei denen die zeitintensive Operation eine Datenbankabfrage ist.
Das synchrone Programm (linker Teil) benötigt mehr Zeit als das asynchrone Programm (rechter Teil), da es im Gegensatz zum asynchronen Programm nach dem Absenden einer Datenbankanfrage solange mit der weiteren Programmausführung wartet, bis eine Antwort von der Datenbank eintrifft.

Der Umgang von JavaScript mit asynchronem Code wird als *non-blocking*

```
console.log(“Received a request”); // 1
db.query(“SELECT * FROM myTable”); // 2
console.log(“Finished request”); // 3

// Received a request
// Database query finished
// Finished request
```

Listing 20.1: Synchrone Datenbankabfrage: Nachdem der Server eine Datenbankabfrage angewiesen hat, wartet er solange, bis er ein Ergebnis erhält, bevor er mit Ausführung des Programms fortfährt. Reihenfolge: Von oben nach unten

```
console.log(“Received a request”); // 1
db.query(“SELECT * FROM myTable”, function (err, rows) {
    if (!err) console.log(“Database query finished”); // 3
});
console.log(“Finished request”); // 2

// Received a request
// Finished request
// Database query finished
```

Listing 20.2: Asynchrone Datenbankabfrage: Nachdem der Server eine Datenbankabfrage angewiesen hat, fährt er mit der Programmausführung solange fort, bis er das Ergebnis der Datenbank erhält.

bezeichnet. Zeitkritisch ist ein JavaScript-Programm beispielsweise dann, wenn sich das Programm wie in einer, wie in Kap. 1.2.2 konzipierten komponentenreichen Medieninstallation auf mehrere interagierende Einheiten verteilt ist, z. B. auf einem Client, einem Server, einer Datenbank und auf einem Mikrocontroller.

Auch jQuery-Animationen verwenden im Hintergrund Timer. „So the animation code only runs in short bursts when it needs to change something. The timers causes events at set intervals, and the event handlers run when nothing else is running." [Guffa] Demnach sind auch zeitintensive jQuery-Animationen *non-blocking*, da sie die Animation lediglich initiieren und danach weitere Programmteile sofort ausgeführt werden können, ehe die Animation beendet ist. Der Timer selbst läuft in einer web API im Browser und nicht in der JavaScript-Engine selbst.

Zeitkritische Aufgaben werden in der Regel mit Hilfe von Callbacks und Callbackfunktionen realisiert (vgl. Kap. 2.3.3.3). Bei zeitunkritischen Aufgaben hingegen macht es keinen Unterschied, ob Funktionen synchroner oder asynchroner Natur sind. Zeitunkritische Aufgaben sind z. B. einfache Berechnungen, Bedingungen und Mitteilungen auf der Konsole. Sie finden ausschließlich auf *einer* Einheit statt, beispielsweise ausschließlich im serverseitigen oder ausschließlich im clientseitigen JavaScript-Code.
Zeitunkritische Aufgaben erstrecken sich *nicht* über mehrere Einheiten und beinhalten auch keine DOM-Funktionen, wie z. B. Timerfunktionen.

In Hinblick auf die asynchrone Verarbeitung von Ereignissen in einer komponentenreichen Event Media-Installation beschäftigt sich folgender Abschnitt näher mit Callbacks und Callbackfunktionen.

2.3.3.3. Callbacks und Callbackfunktionen

> *A callback function is a function called at the completion of a given task.* [Reed]

Eine *Callbackfunktion* (in den folgenden Beispielen wird sie zur Veranschaulichung `function2` genannt) wird für `function1` zur Ausführung zu einem bestimmten späteren Zeitpunkt bereitgelegt, sobald ein bestimmtes Ereignis einritt, nämlich das *Callback* (vgl. Listing 21 auf Seite 48). Die Callbackfunktion `function2` ist inaktiv. Sie wird erst ausgeführt, nachdem `function1` beendet wurde.

Funktionsweise

> *„Callbacks sind das neue goto."* [DotNetPro]

Hat `function1` alle ihre Aufgaben abgearbeitet, stellt sie als *Response* einen Antrag an den JavaScript-Interpreter, dass dieser an die Stelle im Code zurückkehrt, von der aus `function1` aufgerufen wurde. Dieser Antrag wird

Callback („Rückruf“) genannt. Da der Interpreter dadurch an die Stelle im Code vor `function2` springt, dient das Callback als auslösendes Ereignis für die Ausführung von `function2`, der sogenannten *Callbackfunktion*.
Im Englischen lässt sich eine Callbackfunktion einfacher beschreiben:
A callback function is the function which is being triggered after a previous function has *called* the interpreter *back* to an original position using a callback [kein Zitat].

Die Verwendung einer Callbackstruktur ist eine dominierende Praxis, um innerhalb des nativ synchron verlaufenden JavaScript-Programms eine asynchrone Chronologie von Aufgaben zu realisieren. Hierbei werden Funktionen aneinander gekettet. Jede Aufgabe wird bis zu ihrem Ende ausgeführt, bevor die jeweils nächste Aufgabe an der Reihe ist[1]. Diese wurde bereits im Vorfeld festgelegt.
Während `function2` wartet, bis `function1` abgearbeitet wurde und von `function1` ein Callback erhält, kann der Interpreter inzwischen andere Programmteile ausführen, ohne wartend Zeit zu verschwenden.
Da zeitintensive Aufgaben in JavaScript typischerweise *asynchron* gelöst werden, werden Callbacks und Callbackfunktionen eingesetzt, um den zeitlichen Ablauf der aufeinander folgenden Aufgaben zu regeln. Würde eine Aufgabe *synchron* aufgerufen werden, wäre das Programm bis zu ihrer Fertigstellung blockiert. Werden diese zeitintensiven Operationen *asynchron* aufgerufen, werden sie lediglich begonnen. In diesem Fall ist das Programm während zeitintensiver Operationen *nicht* blockiert und friert nicht ein, wie es bei einem *synchronen* Progammverlauf der Fall wäre – auch wenn JavaScript-Programme nur auf *einem einzigen* Thread laufen.
Details über das Singlethreadmodell sowie über die Task Queue und den Eventloop, die in diesem Zusammenhang bedeutende Rollen spielen, werden in Kap. 2.4 behandelt.
Im Anhang A.6 befindet sich ein alltägliches Beispiel im Rollenspiel-Charakter, das die asynchrone Programmierung mit Callbacks und Callbackfunktionen im Kontrast zur synchronen Programmierung veranschaulicht.

Vergleich mit anderen Programmiersprachen

Der Einsatz von Callbacks und Callbackfunktionen garantiert in anderen Programmiersprachen *nicht allgemein* einen asynchronen Programmverlauf. Havoc merkt an, dass in anderen Programmiersprachen neben asynchronenen Callbacks auch synchrone Callbacks auftreten können. „A synchronous callback is invoked *before* a function returns, [...] an asynchronous [...] callback is invoked *after* a function returns [...].“ In JavaScript jedoch stellen laut Microsoft „Rückruffunktionen eine andere Art der asynchronen Verarbeitung dar, da sie Rückrufe in den Code vornehmen, von dem der Prozess initiiert wurde.”

Vermeidung von Kontinuationsfehlern

Im Bereich Event Media ist eine explizit vorgegebene asynchrone Ablaufsteuerung meistens unverzichtbar.

1 Eine Serie von aneinander geketteten Funktionen nennt man *Callback Chain*

Im Folgenden einige geläufige und erprobte Beispiele:

- Im WWW oder auf Großveranstaltungen greifen oft zahlreiche Menschen gleichzeitig auf denselben Webserver zu. Dass JavaScript mehrere konkurrierende Aufgaben zur selben Zeit zulässt, nennt man auch *non-blocking*. Diese konkurrierenden Aufgaben können in Node.js z. B. mehrere gleichzeitige clientseitige Anfragen an einen Webserver sein. Mehr zu *non-blocking* in Verbindung mit Node.js werde ich auf Seite 65 näher erläutern.
- Socketverbindungen (vgl. Beispiel auf Seite 72).
- Datenbankverbindungen (vgl. Beispiel auf Seite 77)
- Kommunikation mit einem Mikrocontroller über socket.io (vgl. Beispiel auf Seite 74).

Diese Beispiele haben gemeinsam, dass ein Client zunächst eine Anfrage an einen Server stellen muss, der daraufhin die Anfrage des Clients bearbeitet und *erst im Anschluss* eine Antwort an den Client gibt. Da diese Aufgaben über mehrere Komponenten (vgl. Kap. 1.2.4) verteilt werden, werden sie zeitintensiv. Der Ablauf der aufeinanderfolgenden Funktionen muss asynchron über Callbackkonstrukte gelöst werden, damit das Programm nicht ins Stocken gerät.

Ein Beispiel möchte ich besonders hervorheben: Die Sicherung der referentiellen Integrität bei der Modellierung der MySQL-Datenbank für die Projekteplattform (vgl. Kap. 4.2.2.2 auf Seite 85): Zunächst müssen die Haupttabellen gefüllt werden, bevor sie über eine Zwischentabelle miteinander verbunden werden können. Erst nachdem beide Einfügeoperationen in die beiden Haupttabellen beendet sind, wird ein Callback ausgelöst, mit dem die Zwischentabelle befüllt wird.

Folgender Abschnitt beschreibt die Vorgehensweise, wie bestehende Funktionen in eigenen Projekten mit Callbackkontrukten nachgerüstet werden können, um bei ihnen ebenfalls eine Ablaufkontinuität festzulegen.

Bestehende Funktionen nachträglich asynchron machen mit Hilfe von Callbacks und Callbackfunktionen

Einige Bibliotheken, wie z. B. Node.js und jQuery, bieten bereits fertige asynchrone Funktionen an, die ohne zusätzliches Hinzutun verwendet werden können. Intern sind sie mit Callbackkonstrukten versehen. Ein bekanntes Beispiel ist die `click`-Methode aus der jQuery-Bibliothek:
`$('#myButton').click(function(){..});`.
Die `click`-Methode ist ein praktisches Beispiel für `function1` aus den vorangegangenen Beispielen. Im praktischen Gebrauch wird sie lediglich aufgerufen und mit einer Eventhandlerfunktion verbunden.

Die Beispiele in Listing 21 (nächste Seite) zeigen, wie auch bestehende Funktionen in eigenen Projekten mit Hilfe von Callbacks und Callbackfunktionen mit einer Ablaufreihenfolge versehen werden können. Dieser geregelte Programmfluss wird auch *Continuation Passing Style* (CPS) genannt.

```
function1(function2);

function function1(callback) {
    console.log('First Task');
    callback();
};

function function2() {
    console.log('Second Task');
};

//First Task
//Second Task
```

Listing 21.1 `function2` wird ausgeführt, sobald sie innerhalb von `function1` mit `callback()` aufgerufen wird.

```
function1(function2);

function function1(callback) {
    console.log('First Task');
    callback('Second Task');
};

function function2(data) {
    console.log(data);
};

//First Task
//Second Task
```

Listing 21.2: `function1` kann mit dem Callback-Parameter an `function2` mitgeben.

```
function1('First Task', function2);

function function1(data, callback) {
    console.log(data);
    callback();
};

function function2() {
    console.log('Second Task');
};

//First Task
//Second Task
```

Listing 21.3: Die Referenz auf `function2` in der Parameterliste von `function1` kann mit kann mit weiteren Parametern koexistieren. Die Referenz auf die Callbackfunktion nimmt konventionell den letzten Platz in der Parameterliste ein.

```
function1(function(){
    console.log('Second Task');
});

function function1(callback) {
    console.log('First Task');
    callback();
};

//First Task
//Second Task
```

Listing 21.4: `function2` wird als anonyme Inlinefunktion beim Aufruf von `function1` direkt in die zu überreichenden Parameterliste geschrieben

```
function1( );

function function1(callback) {
    console.log('First Task');
    if (callback)callback();
};

// First Task
```

Listing 21.5: Die Übergabe einer Callbackfunktion an `function1` ist optional, wenn innerhalb von `function1` mit einer Bedingung entsprechend darauf reagiert wird. Hier existiert kein Übergabeparameter.

Listing 21.1 zeigt die Grundstruktur, die nötig ist, um zwei Funktionen mit Hilfe eines Callbacks miteinander zu verketten: Zunächst wird an `function1` bei Ihrem Aufruf eine Referenz auf `function2` als Aktualparameter übergeben. Dieser Aktualparameter wird von `function1` mit dem Formalparameter callback[1] entgegengenommen.
Nachdem alle Aufgaben von `function1` abgearbeitet wurden, wird der Formalparameter `callback` wie eine Funktion aufgerufen: `callback();`. Somit wird die durch ihn referenzierte `function2` aufgerufen.
`function2` benötigt keine Modifikation.

Listing 21.2 zeigt, wie `function1` beim Aufruf von `function2` als Callbackfunktion in Form von Aktualparametern zusätzlich Daten mitgeben kann.

Listing 21.4 verändert die Grundform aus Listing 21.1 dahingehend, dass `function1` bei ihrem Aufruf nicht nur eine Referenz auf `function2` erhält, sondern die vollständige `function2` als anonyme Inlinefunktion.
Diese Vorgehensweise reduziert den Aufruf von `function1` und den Körper von `function2` auf nur noch *einen* Codeblock. `function1` hingegen kann der Übersichtlichkeit zuliebe in eine Bibliothek ausgelagert werden.

1 Der Formalparameter, der eine Referenz auf `function2` (Callbackfunktion) enthält, kann einen beliebigen Namen haben. Geläufig sind jedoch `callback` oder `next`.

```
taskchain();

function taskchain() {
    function1(function(){
        function2(function(){
            function3();
        });
    });
};

function function1(callback){
    console.log('First Task');
    callback();
}

function function2(callback){
    console.log('Second Task');
    callback();
}

function function3(callback){
    console.log('Third Task');
}

// First Task
// Second Task
// Third Task
```

Listing 22.1: Verkettung von beliebig vielen Funktionen, indem Funktionsaufrufe ineinander verschachtelt werden.

```
function1(function2(function3()))();

function function1(callback) {
    return function() {
        console.log('First Task');
        if (callback) callback();
    };
}

function function2(callback) {
    return function() {
        console.log('Second Task');
        if (callback) callback();
    };
}

function function3(callback) {
    return function() {
        console.log('Third Task');
    };
}

// First Task
// Second Task
// Third Task
```

Listing 22.2: Verkettung von beliebig vielen Funktionen indem alle Funktionsaufrufen in einem Einzeiler ineinander verschachtelt werden.

Konventionell wird die Eventhandlerfunktion `function2` (oder die Referenz auf sie) als *letztes* mögliches Glied der Parameterliste an `function1` übergeben[1]. Die Verwendung des letztmöglichen Aktualparameters ist besonders sinnvoll, da die Ankopplung von `function2` (Callbackfunktion) an `function1` *optional* ist. Dass der letzte Aktualparameter beim Aufruf von `function1` *nicht zwingend* ist und auch ausgelassen werden kann, liegt an der Tatsache, dass es in JavaScript möglich ist, weniger Aktualparameter an eine Funktion zu übergeben als dass diese Formalparameter entgegennehmen kann (vgl. Listing 21.5). Wäre bereits ein früherer Formalparameter für die Callbackfunktion `function2` reserviert, müsste `function1` bereits bei ihrem Aufruf mit dem Aktualparameter `null` an der entsprechenden Stelle in der Parameterliste mitgeteilt werden, dass `function1` keine Callbackfunktion enthält. Mit der Bedingung `if(callback)callback();` reagiert `function1` flexibel darauf, ob sie eine Callbackfunktion als Formalparameter erhalten hat oder nicht.

Beliebig viele Funktionen miteinander verketten

Um eine Funktionskette zu erstellen, können alle zu verkettenden Funktionen als Inlinefunktionen ineinander verschachtelt werden (vgl. Listing 22.1 (nächste Seite)). Listing 22.2 zeigt eine Variation, in der die Funktionsaufrufe recht übersichtlich mit einem Einzeiler im Code aneinandergereiht werden können.

1 Eine Ausnahme sind Timerfunktionen wie `setTimeout` und `setInterval`. Dort wird die Callbackfunktion `function2` (oder die Referenz auf sie) bereits als erster Aktualparameter übergeben: `setTimeout(function(){..},1000);`.

Geläufiger ist jedoch der Ansatz, vollständige Funktionskörper ineinander zu verschachteln (vgl. Listing 21.4). Auf diese Weise verschachteln sich bei komplexen Funktionsmodulen u. a. Funktionen, Bedingungen und Schleifen, usw. Der Code besteht aus einer Vielzahl von Codeeinrückungen und Go-To-Verweisen. Dadurch wird er sehr unübersichtlich und schwer nachvollziehbar. Ein derart strukturierter Programmcode wird auch *Callback-Hölle* genannt. „Callback hell is any code where the use of function callbacks in async code becomes obscure or difficult to follow." [Breck-McKye]

Ausblick

Weitere Möglichkeiten, mehrere Funktionen miteinander zu verketten, bieten u. a. diverse Hilfsbibliotheken, wie z. B. *async.js,* und das Konzept der *Promises* ab ECMA Script 6. Jedoch arbeiten auch im Hintergrund dieser Möglichkeiten weiterhin Callbacks und Callbackfunktionen.

In diesem Abschnitt wurde behandelt, wie in JavaScript mit Hilfe von Callbacks und Callbackfunktionen ein asynchroner Programmverlauf realisiert werden kann.
Im nächsten Abschnitt wird anhand des Singlethreadmodells erläutert, welch tragende Rolle die asynchrone Verarbeitung von Ereignissen mit Callbacks und Callbackfunktionen für flüssig laufende JavaScript-Programme spielen.

2.4. Singlethread-Modell

Zunächst muss geklärt werden, was ein Thread ist und was das Singlethreadmodell im Vergleich zum Multithreadmodell auszeichnet. Im Anschluss werden die Task Queue und der Eventloop beschrieben, die eine asynchrone Ausführung von JavaScript-Programmen im Singlethreadmodell ermöglichen.

Thread

Computerprogramme arbeiten entweder im *Singlethread-* oder im *Multithreadmodell*. Jedes Programm, das auf einem Betriebssystem läuft, ist ein Prozess. Jeder Prozess besteht aus einem oder mehreren Threads *(Spuren)*, die sich die Prozessorleistung untereinander aufteilen. Man spricht hierbei vom *Singlethreading* bzw. vom *Multithreading* (vgl. Khanvendal).
Beim *Multithreading* wird innerhalb eines Prozesses für jede Aufgabe ein neuer Thread geöffnet, der solange erhalten bleibt, bis diese Aufgabe erledigt ist. Das können langwierige Aufgaben wie eine Animation oder eine Datenbankabfrage sein, die mit dem Multithreadmodell parallel verlaufen können.
Auf Apache-Servern wird beispielsweise zur Verarbeitung einer Anfrage eines Clients ein neuer Thread erzeugt. Damit steigt mit steigender Anzahl an anfragenden Clients auch die Anzahl der benötigten Threads, woraufhin jedem einzelnen Thread immer weniger Prozessorzeit zur Verfügung steht.

Bei Webseiten, auf die viele Benutzer gleichzeitig zugreifen, wird dieses Problem gelöst, indem entweder der Server aufgerüstet (*Vertical Scaling*) oder ein asynchrones Zugriffssverfahren im Singlethread realisiert wird.

2.4.1. Singlethread

Beim Singlethreadmodell ist *nur ein* Thread vorhanden. Alle Aufgaben innerhalb des Prozesses eines Programms teilen sich diesen Thread.
Das Singlethreadmodell arbeitet nach dem Prinzip „One thread, one call stack, one thing at a time“ [Roberts].
Klassischerweise blockiert eine langwierige, synchrone Aufgabe eine andere solange, bis sie fertig ist. Ein generell unerwünschte Erscheinung ist hierbei, dass Programme einfrieren können[1], wenn eine synchron verlaufende Aufgabe unter Verwendung von nur einem Thread zu viel Zeit in Anspruch nimmt. „This results in system idle time and user frustration” [Khanvendal].
Im Fall einer Client-Server-Kommunikation ist bei einer Anfrage des Clients an den Server der einzige Thread so lange blockiert, bis der Client eine Antwort vom Server erhält.
JavaScript-Programme arbeiten im Singlethreadmodell. Deshalb ist es in JavaScript nicht möglich, mehrere Threads parallel auszuführen, wie es in einigen anderen Programmiersprachen[2] der Fall ist.

Asynchron im Singlethread ist quasi-parallel

Dass mit JavaScript trotz Singlethreadmodell mehrere Aufgaben gleichzeitig abgearbeitet werden können, liegt daran, dass langwierige Aufgaben *asynchron* abgearbeitet werden. Die asynchrone Arbeitsweise der JavaScript-Engine kann anschaulich mit der Arbeitsweise eines Menschen verglichen werden, der ebenso wenig multitaskingfähig ist:
Ein Mensch kann beispielsweise *nicht* gleichzeitig mit der einen Hand ein Kreuzworträtsel lösen und mit der anderen einen Brief schreiben. In kurzen Sequenzen hintereinander jedoch schon, wenn er zwischen den beiden Aufgaben wechselt. Ein möglicher Ablauf ist: *Aufgabe 1* beginnen, *Aufgabe 1* unterbrechen, mit *Aufgabe 2* beginnen, *Aufgabe 2* unterbrechen, mit *Aufgabe 1* weitermachen, usw. Die Interaktionen der JavaScript-Engine mit dem Singlethread verläuft ähnlich: JavaScript ist in der Lage, Anweisungen asynchron zu verarbeiten. Das heißt, die JavaScript-Engine arbeitet die einzelnen Aufgaben ebenfalls in schnellen Sequenzen *hintereinander* ab.
So kann mit einer asynchronen Ausführung des Programms unter Verwendung von *nur einem Thread* in vielen Fällen eine parallele Ausführung – also unter Verwendung von *mehreren Threads* (Multithread) – ziemlich realitätsgetreu nachgestellt werden. Einzelne Aufgaben hindern sich nicht gegenseitig. Dieses Phänomen nennt man auch *non-blocking*.

---

1 Beim Aufruf eines langwierigen Prozesses „wird die gesamte Ausführung blockiert, bis dieser Prozess abgeschlossen ist. UI-Elemente reagieren nicht, Animationen werden angehalten und es kann kein anderer Code in der App ausgeführt werden [...] denn JavaScript ist eine Singlethreadsprache” [Microsoft].

2 Programmiersprachen, die Multithreading ermöglichen: Z. B. C, C++, PHP, Java

Multithread vs. Singlethread

Dass in vielen Webapplikationen der singlethreadbasierte Ansatz häufig dem multithreadbasierten Ansatz vorgezogen wird, liegt daran, dass singlethreadbasierte Applikationen deutlich weniger Ressourcen verbrauchen als multithreadbasierte:
Beim Singlethreadmodell ehält jede Aufgabe die volle Prozessorleistung, die dem jeweiligen Programm zur Verfügung steht. Aus diesem Grund können singlethreadbasierte Applikationen auf sehr leistungsschwachen Geräten betrieben werden.
Beispielsweise muss eine interaktive Webseite auf leistungsschwachen mobilen Endgeräten funktionieren, oder ein stark frequentierter Webserver (mit Node.js) auch auf durchschnittlichen Computern. Beide Szenarien sind mit JavaScript realisierbar.

Damit die einzelnen Aufgaben im Singlethreadmodell asynchron ausgeführt werden können, werden eine *Task Queue* und ein *Eventloop* benötigt. Das Zusammenspiel dieser Einheiten wird im Folgenden beschrieben und in Abb. 24 auf Seite 55 grafisch dargestellt.

2.4.2. Task Queue

Zunächst wird ein Programm *von oben nach unten* abgearbeitet.
In Abb. 24, Schritt 1 wird das Programm gestartet.
In Schritt 2 wird der synchrone Befehl `console.log('Task1');` direkt in den Ausführungsstack geschrieben.
In Schritt 3 wird im Anschluss `Task1` auf der Konsole ausgegeben.
In Schritt 4 wird `setTimeout(..);` interpretiert.und ebenfalls in den Ausführungsstack geschrieben. Diesmal handelt es sich um eine *asynchrone* Funktion, da der Funktion `setTimeout` mit `function cb(..)` eine Callbackfunktion übergeben wurde. `setTimeout` ist eine API für JavaScript, die auf Timerfunktionen des Browsers zugreifen kann. Timer sind keine nativen JavaScript Funktionen. Demnach wird der Countdown vom Browser übernommen. Er ist unabhängig von der JavaScript-Engine.[1] Konkret bildet der Browser für den Timer einen neuen Thread (vgl. Roberts). Dort wird die Callbackfunktion `cb` zwischengespeichert. Der Timer beginnt nun im Hintergrund zu zählen.
In Schritt 4 wurde lediglich der Timer *angewiesen* und diesem eine Dauer bzw. eine Callbackfunktion übergeben.
In Schritt 5 führt der Interpreter danach sofort `console.log('Task3');` aus. Wie auch `Task1` wird sie *direkt* in den Ausführungsstack geschrieben und danach in der Konsole ausgegeben.
Das eigentliche Programm ist beendet und der Ausführungsstack wird bereinigt. `main()` verschwindet aus dem Ausführungsstack.
Sobald der Timer zwei Sekunden später abgelaufen ist, ist es Zeit, die Ausführung der Timer-Handlerfunktion vorzubereiten. „The web APIs pushes

1 „`setTimeout` is an API provided to us by the browser, it doesn't live in the V8 source, it's extra stuff we get in that we're running the JavaScript run time in." [Roberts].

the callback on to the task queue" [Roberts]. „Asynchrone Ereignisse werden zwangsläufig in eine Ausführungswarteschlange eingereiht." [Resig & Bibeault, S. 244] Bei mehreren asynchronen Ereignissen warten dort alle Eventhandlerfunktionen hintereinander auf ihre Ausführung.

2.4.3. Eventloop

In Schritt 7 (in Abb. 24) wird die Task Queue nach dem Prinzip *First In First Out* (FIFO) ausgelesen. Dafür ist der Eventloop zuständig. „Its task is to look at the stack and to look at the task queue. If the stack is empty, the event loop takes the first thing off the queue and pushes it on to the stack." [Roberts]
Wie der Pseudo-Code in Listing 23 veranschaulicht, verhält sich der Eventloop im Wesentlichen wie eine *unendliche Schleife*, die zwischen der Task Queue und dem Ausführungsstack vermittelt. „The event loop is a link between a program and the operation system." [Roberts]

```
var taskqueue = [];
var stack = [];
var myCallbackFunktion;
// keep going "forever"
while (true) {
    // perform a "tick"
    if (taskqueue.length > 0 && stack.length == 0) {
        // get the next event handler in the task queue
        myCallbackFunktion = taskqueue.shift();
        // now, execute the next event
        try {
            stack.push(myCallbackFunction);
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

Listing 23: Die `while`-Schleife im Pseudocode verdeutlicht die Arbeitsweise eines Eventloops.

In Schritt 8 (in Abb. 24) befindet sich der Befehl `console.log('Task2');` schließlich im Ausführungsstack, woraufhin nun auch `Task2` in der Konsole ausgegeben wird.

Mögliche Verzögerungen bei der Abarbeitung von Aufgaben im Singlethread-Ansatz

Dieses Beispiel zeigte, dass Timer-Handlerfunktionen in JavaScript asynchroner Natur sind und die Task Queue passieren müssen.
Zu gewissen Zeiten jedoch, wenn sich eine Vielzahl von Eventhandlerfunktionen in der Task Queue befinden, sind auch Timer „keine Garantie dafür, dass die vorgesehene Verzögerung für eine exakt getimte Ausführung der Eventhandlerfunktion tatsächlich eingehalten wird" [Resig & Bibeault, S.

243]. Grund dafür ist, dass Timer-Handlerfunktionen möglicherweise andere *asynchrone* Eventhandlerfunktionen in der Task Queue abwarten müssen und ggf. nicht sofort ausgeführt werden können. Aus diesem Grund eignen sich JavaScript-Programme nicht für u. a. Maschinenbauanwendungen, in denen kleine Verzögerungen nicht in Frage kommen.
Dieser Fall wird in Kap. 6.1.3 auf Seite 110 in Verbindung mit dem JavaScript-fähigen *Tessel Board* näher untersucht.

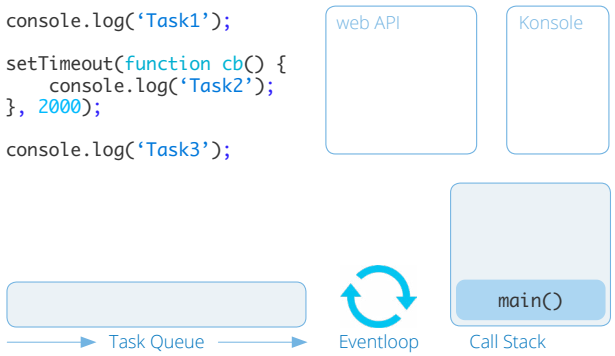
Das Prinzip der singlethreadbasierten Ausführung von Programmen in Verbindung mit asynchronen Callbackfunktionen, der Task Queue und dem Eventloop ist im clientseitigen und serverseitigen JavaScript weitgehend identisch. Der Hauptunterschied ist, dass beim serverseitigen JavaScript anstatt *web APIs* im Browser, *C/C++-APIs*[1] eingesetzt werden (vgl. grafische Darstellung zur Funktionsweise von Node.js auf Seite 67).

Ausblick

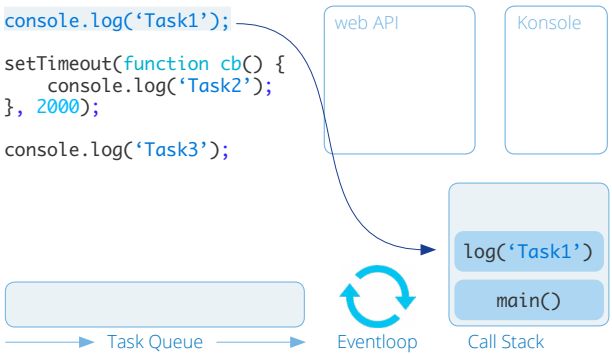
Timerfunktionen ermöglichen generell die Ausführung von asynchronem JavaScript-Code. Zunächst kann anstelle eines Callbackkontruktes, wie in Kap. 2.3.3.3 beschrieben, eine *Timer-Handlerfunktion* an eine *Timerfunktion* `setTimeout(..,0);` gekoppelt werden. Das führt dazu, dass die Timer-Handlerfunktion zunächst an eine *web API* des Browsers gesendet wird, von wo sie zur nächsten Gelegenheit in die Task Queue eingereiht wird.
Node.js liefert mit `process.nextTick(..);` zusätzlich einen mit `setTimeout(..,0);` vergleichbaren, jedoch performanteren Ansatz (vgl. Simpson (Async), S. 8).
Mit der nächsten Version 6 von ECMA Skript wird Asynchronität noch stärker in JavaScript integriert.
In Kap. 3. Client – Server-Interaktion werden mit AJAX und WebSockets zwei Technologien vorgestellt, die der Kommunikation zwischen Clients und Webservern dienen und auf dem Singlethreadansatz aufbauen.

1 Bei Node.js ermöglicht die libuv-Bibliothek das Anlegen mehrerer Ausführungsthreads (vgl. Roberts). Die JavaScript-Engine selbst verwendet dabei weiterhin nur einen einzigen Thread.

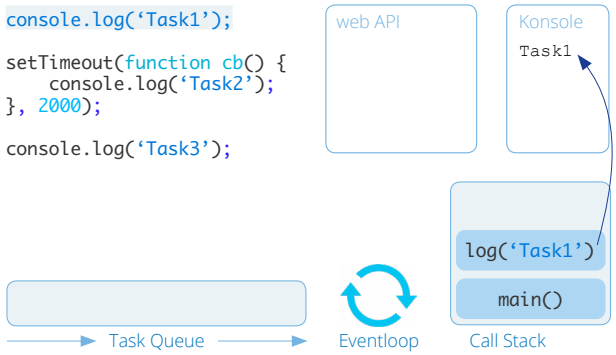
Schritt 1



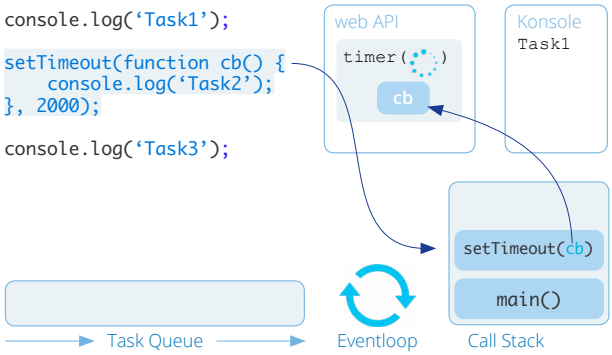
Schritt 2



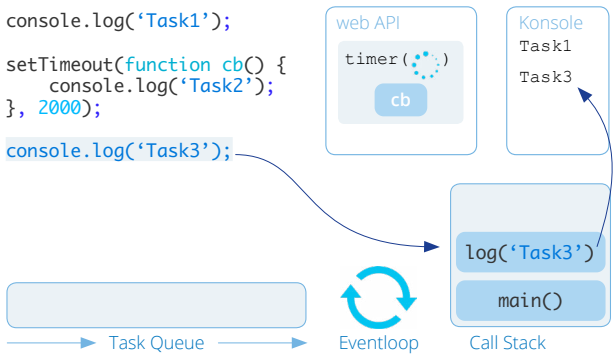
Schritt 3



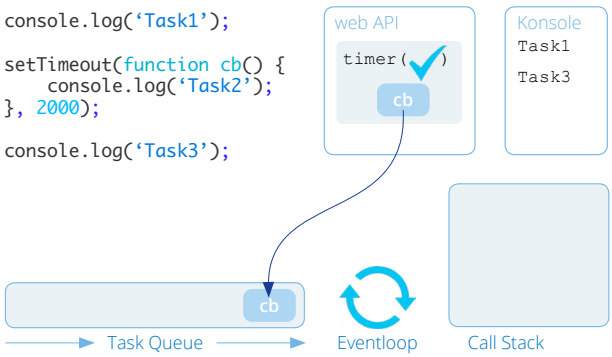
Schritt 4



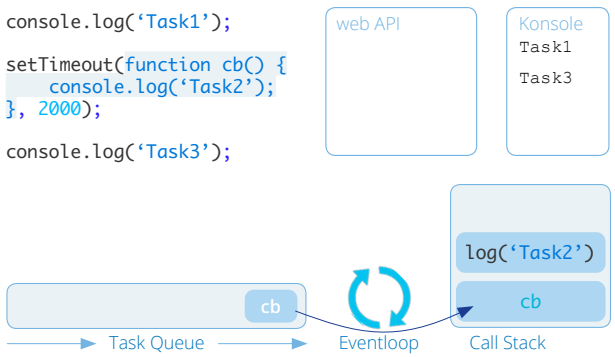
Schritt 5



Schritt 6



Schritt 7



Schritt 8

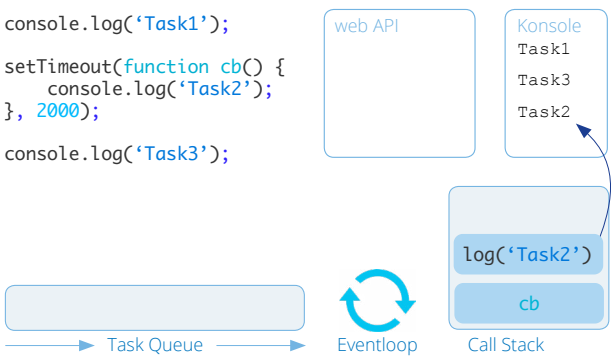


Abb. 24: Beispiel einer singlethreadbasierten Programmausführung mit synchronen und asynchronen Abschnitten

2.5. Zusammenfassung und Überleitung

In diesem Kapitel wurden einige grundlegende JavaScript-Konzepte erklärt, die zur Realisierung von Event Media-Applikationen essentiell sind. Viele Konzepte bauen aufeinander auf und lassen sich nicht strikt voneinander trennen. Bei den vorgestellten Konzepten handelt es sich um eine Auswahl an Konzepten, die ein angehender Entwickler von Event Media-Applikationen zwingend benötigt – auch wenn es darum geht, bestehenden Code, z. B. aus Bibliotheken, zu verstehen, zu nutzen und zu modifizieren. Die Adaption von Open Source Code in eigenen Projekten ist eine sehr wirtschaftliche Praxis und spricht für den Einsatz von JavaScript als etablierte Skriptsprache für das Internet „and beyond“ [Simpson (Async), S. 31].

JavaScript zeichnet sich durch seinen funktionalen Charakter aus, der besonders in der ereignisbasierten Programmierung nützlich ist. Mit dem Prinzip der Closures kann der funktionale Charakter optimal genutzt werden.
Um einige Eigenheiten bei der Programmausführung (z. B. Hoisting, Verfügbarkeit von Objekten, Closures, Plattformunabhängigkeit) zu verstehen, sind Kenntnisse über die Arbeitsweise einer JavaScript-Engine nötig.
Die asynchrone Ausführung von JavaScript-Programmen mit Callbacks und Callbackfunktionen im Singlethreadmodell in Verbindung mit der Task Queue und dem Eventloop sorgt dafür, dass JavaScript-Applikationen auch auf leistungsschwachen Endgeräten störungsfrei funktionieren.

Diese Konzepte sind *nicht* auf clientseitiges JavaScript beschränkt. Im Allgemeinen können sie auch für die Programmierung von u. a. Servern, Datenbanken und der physischen Umwelt verwendet werden. In Kap. 3. (Client – Server-Interaktion) werde ich einige Konzepte vorstellen, die die Grenze zwischen Client und Webserver auflösen.

3. Client – Server-Interaktion

Stand der Technik: Web 3.0

Client und Webserver sind die gegenseitigen Endpunkte einer klassischen Kommunikation über das WWW. Sie kommunizieren miteinander mit Hilfe von geeigneten Kommunikationsprotokollen, auf die ich in diesem Kapitel eingehen werde.
Die Fortschritte des WWW basieren im Wesentlichen auf der Fortentwicklung von Kommunikationstechnologien. Mit der wachsenden Nachfrage für Onlinedienste und der weiten Verbreitung von internetfähigen Endgeräten wächst auch der Bedarf an immer flexibleren, robusteren und intelligenteren Technologien. Der Fortschritt des Internets wird in drei Ären eingeteilt: *Internet 1.0*, *2.0* und *3.0*.

Das Web 1.0 diente vorwiegend der klassischen Informationsbeschaffung. Im Web 1.0 wurden Suchbegriffe in eine Suchmaschine eingegeben, woraufhin der Webserver ihr eine Auswahl an Webseiten mit diesem Schlagwort anbot. Die Kommunikation erfolgte meistens über das HTTP-Protokoll[1].

Mit dem Web 2.0 wurde der Nutzer tiefer in das WWW integriert:
Nutzer können Webinhalte in einem riesigen Netzwerk gemeinsam *selbst* erstellen, bewerten und teilen: Bloggig, tagging, social networking, usw. Prominente Beispiele sind Facebook, YouTube und viele weitere.
Internetanwendungen nahmen zunehmend die Gestalt von grafischen Desktop-Anwendungen mit intuitiv bedienbaren grafischen Oberflächen mit dynamischen Reaktionszeiten an. Als Leittechnologie gilt AJAX, das das HTTP-Protokoll um asynchrone Zugriffsverfahren erweitert. Am Beispiel der Suchmaschine können durch AJAX bereits *während* der Texteingabe Suchvorschläge angezeigt werden.

Das Web 3.0 ist eine Fortentwicklung existierender Technologien. Das Ziel von Web 3.0 ist, die „zurzeit *syntaktisch*[2] vorliegenden Informationen des World Wide Webs mit Hilfe von *semantischen*[3] Netzen *inhaltsorientiert* zu präsentieren“ [Neßelrath]. Man spricht auch vom *denkenden Internet*, das semantische Beziehungen aufzeigt, Informationen nach ihrer Bedeutung bewertet und in einen Kontext zu anderen Webinhalten stellt:
Web 3.0 = Semantisches Web + Web 2.0.
Suchmaschinen sollen in der Lage sein, zusätzlich zu bestimmten Zeichenkombinationen auch mit sinnbezogenen Beziehungen umgehen zu können.

1 Je nach Zweck kann die Kommunikation über das Internet auch mit anderen Protokollen zustande kommen. Z. B. bei Bedarf einer sicheren Verbindung, wie z. B. zu einer Bank, kann ein verschlüsseltes Protokoll (wie HTTPS) zum Einsatz kommen. Bei Streaming-Diensten kann beispielsweise das auf dem verbindungslosen Transportprotokoll UDP basierende Real Time Protocol (RTP) verwendet werden.
2 Syntax: Die Kombination einzelner Zeichen
3 Semantik: Sinnhafte Bedeutung von Zeichenfolgen. Das Wort `Haus` wird nicht nur als syntaktische Zeichenfolge von `H`, `a`, `u` und `s` erfasst, sondern auch als begehbares Gebäude verstanden.

Die Intelligenz für das Web 3.0 wird durch die Nutzer z. B. durch Bewertungen, wie in Amazon, mit den Technologien aus dem Web 2.0 selbst angelegt. Beim Web 3.0 geht es darum, existierende Technologien maximal zu optimieren und diese bei Bedarf[1] zu erweitern.

Rohles erklärt die *Suchmaschine 3.0* am Beispiel der Suche nach einer guten Pizzeria in der Nähe:
Im Web 1.0 ergab die Suche nach „gutes italienisches Restaurant in der Nähe" eine lange Liste von Websites mit diesen Schlagwörtern. Mit dem Web 2.0 ermöglichen technische Innovationen, Restaurants zu bewerten und die Position eines Restaurants zu suchen. Der Begriff „gutes" kann durch beliebige Bewerter im WWW definiert werden. Im Web 2.0 kann die Suchmaschine die Begriffe „gutes" und „in der Nähe" allerdings im semantischen Sinn derzeit *nicht* mit einem Restaurant verbinden.
Außerdem findet die Suchmaschine weiterhin keine Pizzeria, sondern lediglich Einträge zu „Italienisches Restaurant".
Mit dem Web 3.0 wird dieser semantische Bezug hergestellt.
Eine Web 3.0-Suchmaschine kann wie ein Freund befragt werden und findet auch eine Pizzeria.

Im Folgenden möchte ich in diesem Zusammenhang auf einige Technologien eingehen, die Clients und Server näher zusammenbringen und sich besonders zur Realisierung von komponentenreichen Event Media-Installationen eignen (wie in Kap. 1.2.4 beschrieben).

3.1. HTTP

Das HTTP-Protokoll dient zur Verbreitung von Dateien und anderen Ressourcen[2] über das Internet. Der HTTP-Transfer erfolgt über einen TCP/IP-Socket (vgl. OSI-Schichtenmodell auf Seite 11). Typischerweise wird HTTP zur Kommunikation zwischen Clients und Webservern eingesetzt. Sie läuft folgendermaßen ab:
Ein Client öffnet einen TCP Socket zum Webserver und sendet eine Anfrage an ihn. Im Optimalfall antwortet dieser und schließt im Anschluss die Verbindung wieder. Ein HTTP-Request wird *grundsätzlich* von einem Client initiiert. Bei diesen Anfragen handelt es sich in den meisten Fällen um *GET*- oder `POST`-Requests[3].

- Bei einem `GET`-Request (oben erläutertes Szenario) stellt der Client in einer *ersten* HTTP-Nachricht einen Antrag an einen Webserver, ihm eine Ressource bereitzustellen. Bei der Eingabe eines URL im Webbrowser verlangt er beispielsweise nach einer HTML-Datei. Die *zweite* HTTP-Nachricht ist die Antwort das Webservers an den Client. In ihr steckt die angefragte Ressource.

1 Ein veränderter Umgang mit Daten im Zeitalter *BigData* führt u. a. zur Notwendigkeit von speziellen Datenbanken (vgl. Kap. 5: Datenbank)
2 Ressource: Oberbegriff für jede Form von statischen Dateien und dynamisch generierten Ergebnissen, z. B. eines Serverprogramms oder von Datenbankabfragen.
3 andere HTTP Methoden sind u. a. `PUT`, `HEAD`, `CONNECT`, `TRACE`, `DELETE`

- Bei einem `POST`-Request stellt der Client einen Antrag an den Webserver, Daten entgegenzunehmen.[1] Das kann beispielsweise der Inhalt eines Formulars sein, der in eine Datenbank geschrieben werden soll. Mit *einer* HTTP-Nachricht werden die Daten an den Webserver übermittelt. Bei einem `POST`-Request erfolgt im Anschluss, anders als bei einem `GET`-Request, *keine* Antwort durch den Webserver an den Client.

Die Kommunikation mit dem HTTP-Protokoll ist grundsätzlich requestbasiert. Das bedeutet, dass die Kommunikation ausschließlich durch Anfragen eines jeweiligen Clients zustande kommt. Das HTTP-Protokoll ermöglicht dem Webserver *nicht*, auf Clients zuzugreifen ohne deren vorherigen Anfrage.[2]

Jede HTTP-Nachricht besteht aus einem Header und einem Body.
Im Body befinden sich die Daten[3], die ggf. übermittelt werden müssen.
Bei einem `GET`-Request eines Clients an einen Webserver ist der Body leer.
Im Header[4] befinden sich Pflichtinformationen über die Verbindung.

Der Zugriff auf einen Webserver mit dem herkömmlichen HTTP-Protokoll kann zu langen Ladezeiten führen. Ein Ausweg ist die Web 2.0-Technologie AJAX, die es ermöglicht, Teile von Webseiten asynchron zu befüllen.

3.2. AJAX

> *AJAX[5] ist eine Technologie, die den Browser mit dem Server interagieren lässt, ohne die angezeigte Seite zu stören.* [Castledine]

AJAX ist keine Programmiersprache und auch keine *eigenständige* Technologie, sondern ein asynchrones Kommunikationsverfahren zwischen Browser und Webserver, das sich mehrere bestehende Technologien zunutze macht (vgl. Garrett):

- HTML and CSS für die Entwickung der clientseitige Weboberfläche
- Das DOM zur dynamischen Anzeige von HTML-Elementen
- XML (oder auch bspw. JSON[6] oder Klartext) als Datenaustauschformat
- Ein `XMLHttpRequest`-Objekt, um asynchron Daten entgegenzunehmen
- JavaScript um alle Elemente miteinander zu verbinden

---

1 Für die Übermittlung eines Formulars kann auch die HTTP-Methode `GET` verwendet werden. Allerdings werden hier Daten in der URL übertragen und sind im Klartext lesbar. Passwörter oder längere Texte sollten über diesen Weg nicht übermittelt werden.

2 Die Kommunikation zwischen einem Webserver und Clients, bei der auch durch den Webserver Nachrichten an verbundene Clients initiiert werden können, werden mit dem Konzept der WebSockets hergestellt (vgl. Kap. 3.3)

3 Zu übermittelnde Daten: Z. B. Formulardaten, Dateiuploads

4 Beispielsweise wird bei einer Response im Erfolgsfall in der ersten Headerzeile mit `HTTP/1.0 200 OK` die verwendete Protokollversion, der Staturcode und die englische Bezeichnung für den Statuscode übertragen.

5 AJAX (Asynchronous JavaScript and XML) ist eine Abkürzung für eingesetzte Technologien bei der Realisierung von asynchronen Client – Server-Interaktionen. Der Namensgeber Garrett „needed something shorter than 'Asynchronous JavaScript+CSS+DOM+XMLHttpRequest' to use when discussing this approach with clients"

6 Heutzutage nimmt JSON weitgehend die Rolle von XML als Datenaustauschformat ein.

Der klassische Vorgang beim Laden einer Webseite ist Folgender:
Ein Client stellt einen Antrag an einen Webserver, ihm eine bestimmte Webseite auszuhändigen. Der Webserver stellt sie dem Client anschließend bereit. AJAX ermöglicht, clientseitig Anfragen für *nur bestimmte Teile* der aufgerufenen Webseite an den Webserver zu senden, um eine Webseite *dynamisch* mit neuen Inhalten zu füllen, ohne dass dafür die Seite komplett neu geladen werden muss. Während das Client-Programm im Hintergrund auf ausstehende Daten vom Webserver wartet, kann der User die Website ungestört weiterverwenden, ohne dass die Webseite während der Wartezeit blockiert wäre. AJAX ermöglicht, einzelne Inhalte für HTML-Elemente vom Webserver *während der Laufzeit* dynamisch nachzuladen. AJAX wird auch als der Schlüssel zu *Realtime* bezeichnet. Bei der Aktualisierung von einzelnen HTML-Elementen müssen Teile der Webseite, die nicht aktualisiert werden müssen, nicht erneut vom Webserver geladen werden. Somit eignet sich der Einsatz von AJAX besonders für echtzeitfähige Anwendungen, z. B. auf Großveranstaltungen.

Beispiele für AJAX-Anfragen sind der dynamische Kartenaufbau in Google Maps und Suchvorschläge in einer Suchmaschine

Funktionsweise

Sobald sich ein Event ereignet, bildet der Client ein `XMLHttpRequest`-Objekt, das im Anschluss konfiguriert wird und eine asynchrone Anfrage an den Webserver für neue Daten stellt (vgl. Abb. 1). Der Webserver stellt dem anfragenden Client diese Daten bereit, indem er sie beispielsweise in einem JSON-Objekt verpackt. Zurück beim Client ruft das `XMLHttpRequest`-Objekt eine zuvor definierte Callback-Funktion auf, in der die empfangenen Daten verarbeitet werden. Auf diese Weise kann AJAX zur Laufzeit Veränderungen im DOM vornehmen: So können Webseiten Stück für Stück aufgebaut oder bestimmte Bereiche von Webseiten aktualisiert werden. „So the user is never staring at a blank browser window and an hourglass icon [...]." [Garrett, S. 2]
Abb. 1 (linker Teil) stellt eine synchrone, und eine asynchrone Anfrage mit AJAX durch einen Client an einen Webserver gegenüber. In beiden Fällen läuft

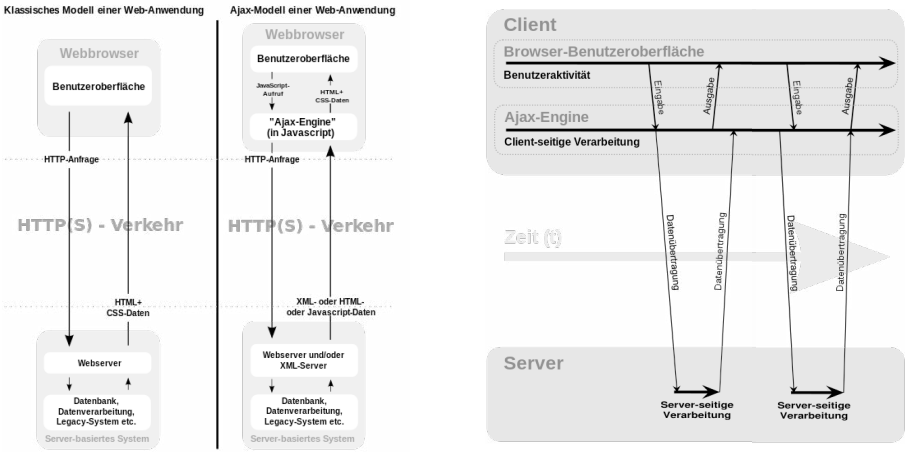


Abb. 1: Links: Übersicht über die beteiligten Kommunikationsprotokolle: Ohne und mit AJAX.
Rechts: Timeline-Darstellung einer Anfrage an den Server mit AJAX

die Anfrage an den Webserver über das HTTP-Protokoll[1] ab. Bei einer asynchronen Anfrage mit AJAX wird die Clientseite durch die AJAX-Engine ergänzt. Sie dient als Zwischenschicht zwischen Client und Webserver. Nun kommt bei einem eintretenden Ereignis die clientseitige Anfrage nicht mehr *direkt* beim Webserver an, sondern in der AJAX-Engine, wo für sie ein `XMLHttpRequest`-Objekt gebildet wird. Wie Abb. 1 (rechts) zeigt, bündelt die AJAX-Engine die zu beliebigen Zeiten eintreffenden Ereignisse und stellt in regelmäßigen zeitlichen Abständen mit Hilfe den HTTP-Protokolls Anfragen an den Webserver.
Neben den nativen Bordmitteln von JavaScript bietet die JQuery-Bibliothek dem Entwickler mit der Methode `$.ajax` und „several higher-level alternatives like `$.get` and `.load`" [jQuery, AJAX] komfortable Möglichkeiten, mit AJAX umzugehen.

Eine weitere, modernere Möglichkeit der Kommunikation zwischen Clients und Webservern bieten WebSockets an.

3.3. WebSockets

Wikipedia, *WebSocket* beschreibt WebSockets wie folgt:

> *Das WebSocket-Protokoll[2] ist ein auf TCP basierendes Netzwerkprotokoll, das entworfen wurde, um eine bidirektionale Verbindung zwischen einer Webanwendung und einem WebSocket-Server bzw. einem Webserver, der auch WebSockets unterstützt, herzustellen.*

Dieser Definition möchte ich auf den Grund gehen.
Walsh bezeichnet WebSockets als „The next generation method of asynchronous communication[..]". Während bei Verbindungen über das HTTP-Protokoll die Kommunikation zwischen einem Client und einem Webserver durch den Client initiiert werden muss, ermöglichen WebSockets auch das Senden von Nachrichten an Clients von einem Webserver aus (vgl. Abb. 2).
Dies wird durch das WebSocket-Protokoll ermöglicht: Anders als beim HTTP-Protokoll wird die Verbindung zwischen Client und Webserver nicht nach jeder Anfrage geschlossen, bevor sie bei der nächsten Anfrage erneut hergestellt werden muss.
Nachdem eine Websocket-Verbindung einmal aufgebaut wurde, bleibt sie solange erhalten, bis der Client die Verbindung selbst schließt, z. B. indem er eine bestimmte Webseite verlässt.
In dieser Zeit kann der Webserver den Client jederzeit frei mit aktuellen Daten versorgen, ohne dass diese zuerst vom Client angefragt werden müssen.

Ein klassischer Einsatzzweck für WebSockets ist die Realisierung eines Chatrooms, bei dem jeder Teilnehmer nicht nur Nachrichten *an* den Webserver senden kann, sondern auch *durch den* Webserver automatisch mit neuen

1 HTTP ist laut OSI-Modell (vgl. Seite 11) eine Anwendungsschicht, die auf der Transportschicht TCP basiert
2 Es existiert ein unsicheres (ws) und ein sicheres (wss) WebSocket-Protokoll

Nachrichten anderer Teilnehmer versorgt wird, ohne danach fragen zu müssen.
Das läuft wie folgt ab:
Ein Nutzer schreibt eine Nachricht in einen Chatroom. Zunächst sendet dessen Computer die Chatmitteilung an den Webserver. Dieser sendet die Mitteilung an alle Clients weiter, wo sie in einem angepassten Layout automatisch erscheinen – so auch am Computer des ursprünglichen Absenders. Der Webserver ist vergleichbar mit einem E-Mail-Verteiler.
WebSockets werden besonders für hochperformante, stark frequentierte Echtzeit-Webseiten entwickelt. Ein Beispiel ist das automatische Nachladen von Statusmeldungen in Facebook.

Einsatzszenario für Event-Installationen

Für den Einsatz von WebSockets im Bereich Event Media ist folgendes Szenario denkbar:
Bei Mach-mit-Veranstaltungen kann jede Person im Publikum mit einem internetfähigen mobilen Endgerät das Geschehen interaktiv beeinflussen. Beispielsweise kann das Publikum über Mehrheitsabstimmungen entscheiden, was als nächstes passieren soll. Dazu muss jeder Teilnehmer sein mobiles Endgerät mit dem Webserver der Veranstaltung verbinden, indem er im Browser eine bestimmte Webseite öffnet. Über eine grafische Weboberfläche kann er mitentscheiden, was als nächstes passieren soll.
Sobald das mehrheitlich abgestimmte Ereignis in der Show eintritt, ist der Inhalt dieser Webseite jedoch nicht mehr aktuell und muss durch einen neuen ersetzt werden. Dieser erscheint auf allen verbundenen Geräten *automatisch* und ohne das Zutun der Teilnehmer, also ohne clientseitige Anfragen.

Echtzeitfähigkeit

Wie das HTTP-Protokoll gehört auch das WebSocket-Protokoll zur Anwendungsschicht im OSI-Modell (vgl. Kap. 1.3.2: Kommunikationsprotokolle) und basiert ebenfalls auf dem verbindungssicheren Transportprotokoll TCP.
Beim WebSocket-Protokoll handelt es sich im Vergleich zum HTTP-Protokoll, das u. a. bei AJAX eingesetzt wird, um ein *leichtgewichtigeres* Protokoll mit geringerem Protokolloverhead[1] und eignet sich besonders für den Echtzeiteinsatz.
Wenn Webseiten regelmäßig mit neuen Inhalten befüllt werden, sind WebSocket-Verbindungen gegenüber AJAX-Verbindungen die geeignetere Wahl. Mit AJAX ist es dem Webserver nämlich nicht möglich, ohne die vorherige Aufforderung eines Clients, Nachrichten an Clients zu senden (vgl. Abb. 2).

In einem eigenen Versuchsaufbau (vgl. Abb. 3 (nächste Seite)) konnte mit einem mobilen Endgerät in einem Farbkreis eine Farbe gewählt werden. Mit einer drahtlosen WebSocket-Verbindung über WLAN wurde der ausgewählte Farbcode als Hexadezimalwert an den Webserver übermittelt, woraufhin

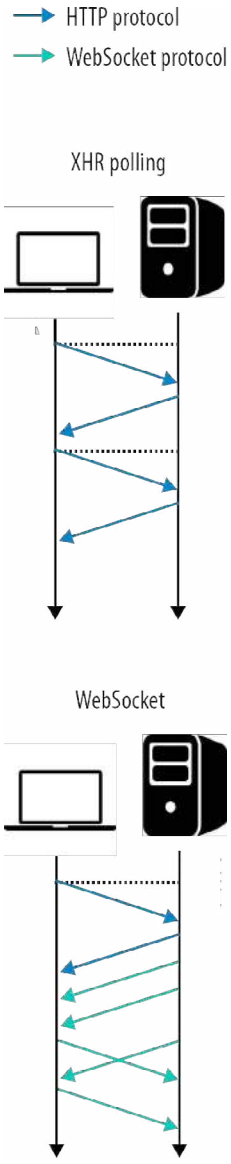


Abb. 2 (frei nach Grigorik): Bei AJAX (oberer Teil) muss vor jeder Client – Server-Anfrage erneut eine Verbindung aufgebaut werden. Bei WebSockets (unterer Teil) ist das nur zu Beginn nötig.

Nachrichten von Server an Client basieren bei AJAX immer auf einer Anfrage des Clients.
Eine Verbindung mit WebSockets funktioniert bidirektional, kann mit HTTP-Anfragen koexistieren und kann auch hergestellt werden, wenn eine vorherige Verbindung noch nicht abgeschlossen ist.

1 Protokolloverhead: Zusatzinformationen, wie z. B. Sender- und Empfängeradresse, die bei einem Datenaustausch mitgeliefert werden. Sie stehen im Head einer Nachricht, die aus einem *Head* und einem *Body* besteht.

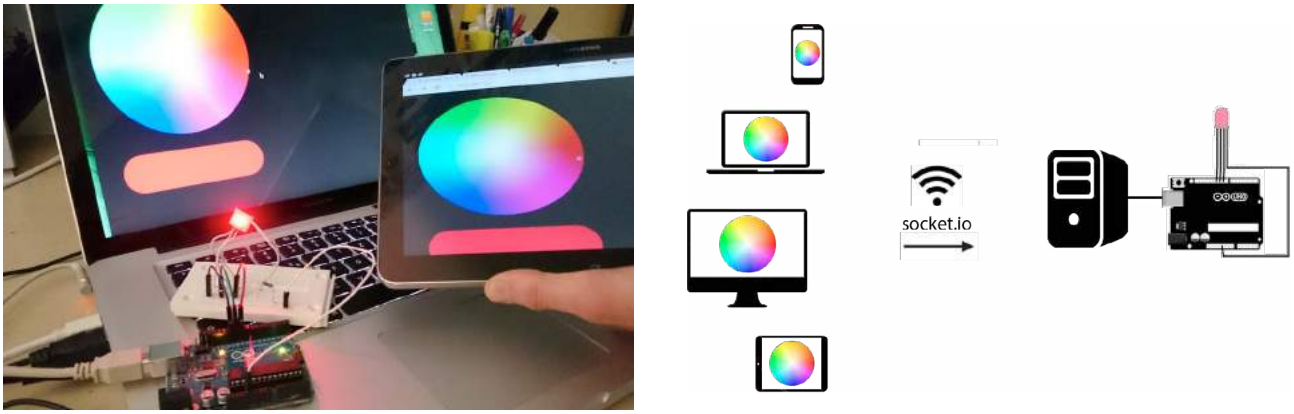


Abb. 3: Beliebig viele Geräte mit Webbrowser können sich gleichzeitig durch das Öffnen einer Webseite über WLAN mit dem Webserver verbinden (er läuft auf dem Laptop) und über einen Farbkreis auf der grafischen Weboberfläche die Farbe einer LED steuern, die über ein Arduino Board mit dem Server verbunden ist.

sich die LED auf einem Arduino Board, das mit dem Webserver verbunden war, änderte – mit einer geringfügigen Verzögerung von einer Drittelsekunde. Anschließend wurde der aktuelle Farbwert vom Webserver an alle verbundenen Clients gesendet wo die Farbe im unteren Farbfeld (vgl. Abb 3: Rot) aktualisiert wurde. Das Prinzip dieser Anwendung entspricht dem des Chat-Beispiels

.3.4. JSON

```
[
    {
        “sweets_id”: 1,
        “name”: “Choco”,
        “brand”: [“Ritter”,
                  “Lindt”],
        “img”: {
            “file”: “c.jpg”,
            “caption”: “B”
        }
    },
    {
        “sweets_id”: 2,
        “name”: “Bonbons”,
        “brand”: [“TicTac”,
                  “Nimm2”],
        “img”: {
            “file”: “b.jpg”,
            “caption”: “B”
        }
    }
]
```

Listing 4: Beispiel für eine JSON-Datei mit zwei Objekten, die Zahlenwerte (`sweets_id`), Strings (`name`), Arrays (`brand`) und Unterobjekte (`img`, Unterobjekte auch in Array möglich).

Beim Austausch von Daten zwischen einem Server und Client(s) werden diese häufig als Text versendet. JSON[1] ist ein beliebtes Datenaustauschformat, in dem Daten in key/value-Paaren strukturiert werden können. Sie können vom Sender als lesbare Textdatei auf dem Webserver abgespeichert werden. Eingesetzt wird das JSON-Format beispielsweise bei WebSocket-Verbindungen, Datenbankabfragen (über Node.js), und auch bei AJAX nimmt JSON zunehmend die Rolle von XML ein.

JSON kann u. a. Objekte, Arrays, Zahlen, Texte und Wahrheitswerte[2] in Textform speichern und sie wie in einem gewöhnlichen JavaScript-Objekt beliebig tief verschachteln (vgl. Listing 4).

Die Ähnlichkeit zu JavaScript-Objekten ist groß und somit auch die Verwechslungsgefahr.

Mit bloßem Auge unterscheiden sich JSON-Objekte und JavaScript-Objekte im JavaScript-Quelltext nur durch den Zusatz von Anführungszeichen bei den *key*-Werten und dass ohne manuelles Hinzutun, z. B. mit der `eval`-Funktion, keine ausführbaren Funktionen ausgetauscht werden können.

Das Austauschformat JSON ist von der verwendeten Programmiersprache unabhängig, jedoch ist in JavaScript die Umwandlung zwischen JavaScript-Objekt und JSON-Objekt mit `JSON.stringify(myJsObject);` bzw. `JSON.parse(myJsonObject);` sehr komfortabel.

1 JSON: JavaScript Object Notaion
2 Speicherbare Werte im JSON-Format: Vgl. json.org (Stand: 10. Mai 2015)

3.5. Node.js

Nodejs.org beschreibt sich selbst folgendermaßen:

> *Node.js is a platform built on Chrome's JavaScript runtime*[1] *for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.*

Prinzipiell ist Node.js „*JavaScript without a browser*" [Shan].
Programme, die in JavaScript geschrieben werden, können auch als Konsolenprogramm ausgeführt werden, ohne dass dafür ein Browser geöffnet werden muss, um eine JavaScript-Engine[2] nutzen zu können. Node.js-Programme werden in einem Terminal folgendermaßen ausgeführt:
`node myJavaScriptFile.js`. Ein grafisches UI liefert Node.js nicht.

Der Hauptunterschied von Node.js gegenüber JavaScript im Browser ist die Möglichkeit, in Node.js auf das native Dateisystem des Computers zuzugreifen, auf dem Node.js läuft. Aus diesem Grund wird Node.js hauptsächlich zur Realisierung von leichtgewichtigen Webservern eingesetzt. Durch externe Bibliotheken können Node.js-Applikationen beliebig erweitert werden.

Im Folgenden möchte ich auf einige Eigenschaften von Node.js aus der Beschreibung von nodejs.org eingehen: *„event-driven"*, *„non-blocking I/O model"*, *„across distributed devices"* und *„scalable"*:

Non-blocking

Mit der zunehmenden Nutzung von sozialen Netzwerken und der Allgegenwärtigkeit von mobilen Endgeräten leiden Server zunehmend unter Performanceproblemen, wenn sehr viele Clients gleichzeitig auf ihn zugreifen. Dieses Problem kann insbesondere bei Servern auftreten, die für jede Client-Verbindung einen neuen Thread öffnen[3]. Man spricht hierbei vom C10K-Problem[4]. Die anfragenden Clients stehen in Konkurrenz zueinander.
Node.js bewältigt das C10K-Problem, indem Anfragen von Clients mit *nur einem* Thread entgegengenommen werden.
„Node.js came up with a new idea of event driven single threaded server programming which is achieved with callback concept" [Shan].
Node.js liefert intern einkommende Events an eine bestimmten Eventhandlerfunktion aus. Zeitlich überlappende Aufgaben werden asynchron unter Verwendung von Callbacks und Callbackfunktionen bewältigt. Die strenge zeitliche Regulierung der Abläufe wird auch *Continuation Passing Style*[5] ge-

1 Node.js enthält eine Kopie von Google Chromes JavaScript-Engine V8, ist jedoch vom Chrome-Browser unabhängig.
2 Node.js beinhaltet eine Kopie der JavaScript-Engine V8 von Google (vgl. Abb. 6).
3 Ein Beispiel für einen Multithread-Server ist ein Apache Webserver
4 C10K: „Concurrent Ten Thousand Connections": Ein Server soll 10.000 Client-Verbindungen gleichzeitig verarbeiten können (vgl. Kegel)
5 Die Festlegung der Reihenfolge von einzelnen Aufgaben (Continuation Passing Style) wird in JavaScript mit Callbacks und Callbackfunktionen realisiert.

nannt.
Abb. 5 (linker Teil) veranschaulicht, dass z. B. in PHP und Java für jeden verbundenen Client ein separater Thread geöffnet wird. Bei 10.000 anfragenden Clients werden 10.000 Threads benötigt.
Blocking bedeutet, dass der gegebene Thread eines Clients keiner weiteren Anfrage dieses Clients nachgehen kann, während noch eine Antwort an diesen Client aussteht, denn der Kanal ist noch blockiert.
Abb. 5 (rechter Teil) zeigt den Ansatz in Node.js, bei dem von beliebig vielen Clients und Anfragen der selbe Thread verwendet wird, der aufgrund der asynchronen Arbeitsweise zu jeder Zeit Anfragen entgegennehmen kann.

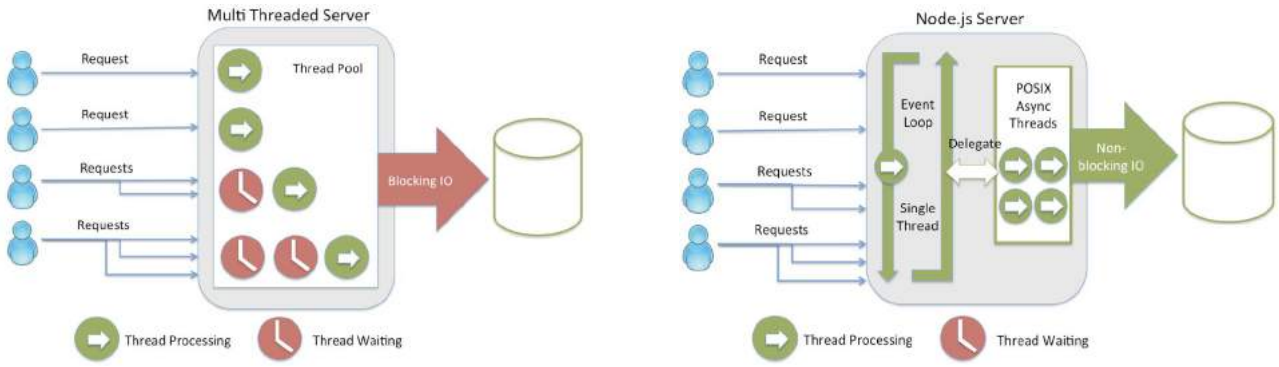


Abb. 5: Links: Blocking IO beim Multithread-Modell, rechts Non-Blocking IO beim Singlethread-Modell

Abb. 6 (rechter Teil) soll die prinzipielle Arbeitsweise eines Node.js-Servers verdeutlichen.
Zunächst stellen beliebig viele Clients Anfragen an einen Webserver. Sie übergeben ihm bereits bei der Anfrage eine Referenz auf die Funktion (Callbackfunktion), die später ausgeführt werden soll, sobald der Server antwortet. Der Webentwickler interagiert dabei mit der in JavaScript geschriebenen *Node.js Standard Library,* mit der der Zugriff auf alle tieferen Schichten ohne C-/ C++-Kenntnisse ermöglicht wird Intern werden die Anfragen über eine *Task Queue* (vgl. Kap. 2.4.2) in den in C geschriebenen *Eventloop* geschrieben (*libev*, vgl. Abb. 6, linker Teil).
Gemeinsam mit der in C++ geschriebenen V8-Engine wäre *libev* nun in der Lage, JavaScript auszuführen.
Noch ist es jedoch nicht möglich auf das Dateiverzeichnis des Rechners, auf dem der Webserver läuft, zuzugreifen.
Da in einer Serverapplikation der Zugriff auf das Dateiverzeichnis eine tragende Rolle spielt[1], wird die in C geschriebene asynchrone IO-Bibliothek *libeio* benötigt. „*libeio* arbeitet nahtlos mit *libev* zusammen und bildet mit dieser zusammen das Fundament für node. *libeio* bietet asynchrone Dateioperationen auf Basis eines Thread-Pools" [Schmidt], der intern vier Threads beinhaltet[2].
Nun wird die Anfrage eines Clients vom Eventloop in einen der vier Threads von *libeio* gesetzt.

1 Mit clientseitigem JavaScript (z. B. über das Internet) ist der Zugriff auf lokale Dateisysteme beliebiger Rechner, aus Sicherheitsgründen nicht möglich.
2 Die C-Bibliothek *libeio* arbeitet intern mit vier Threads. Vgl Protokoll auf Github (Zeile 32: `static uv_thread_t default_threads[4];`)

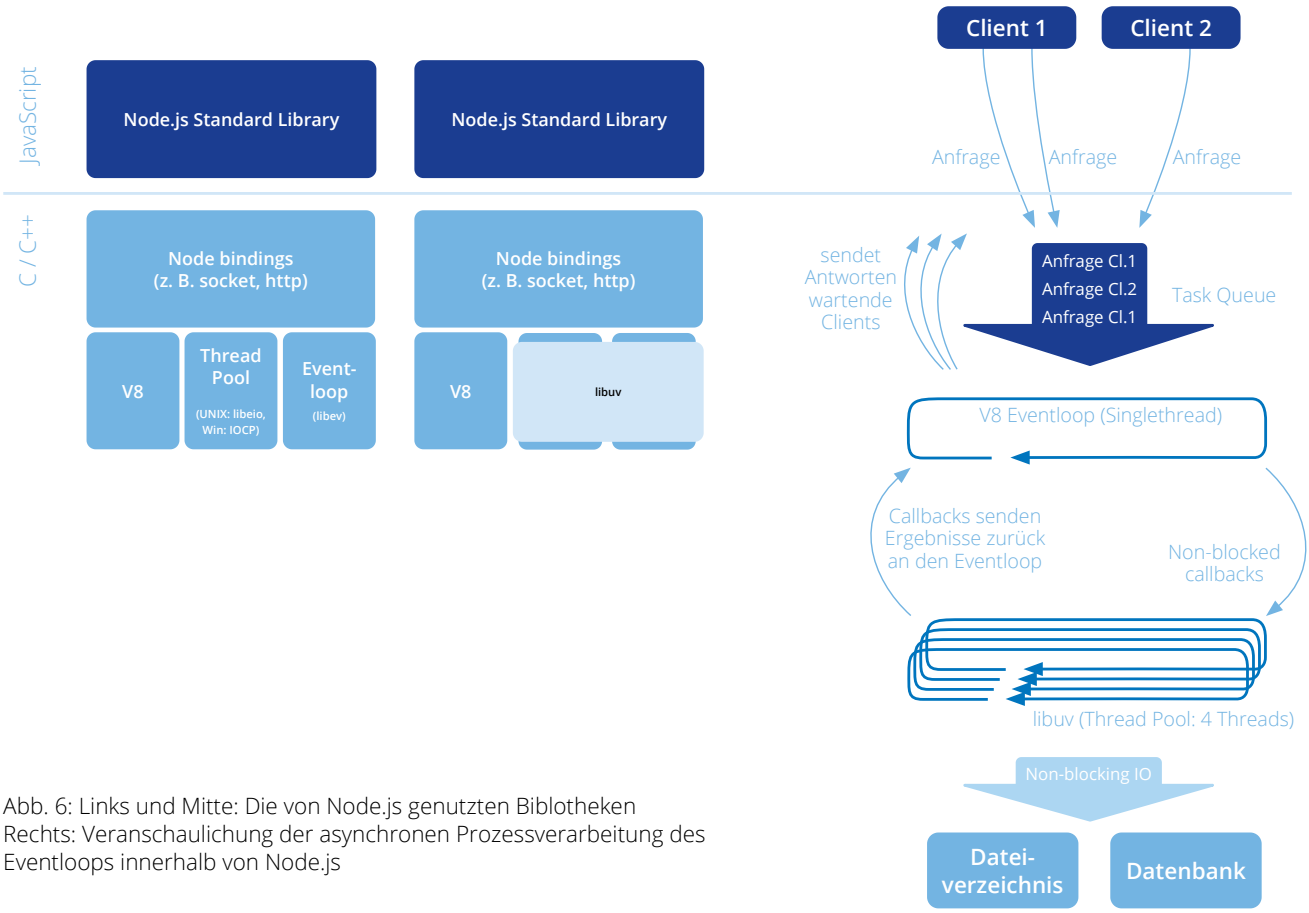


Abb. 6: Links und Mitte: Die von Node.js genutzten Bibliotheken
Rechts: Veranschaulichung der asynchronen Prozessverarbeitung des Eventloops innerhalb von Node.js

Nachdem die Anfrage bearbeitet wurde, wird das Ergebnis über ein Callback wieder in den Eventloop eingefügt. Dieser sendet die Antworten weiter an die entsprechenden anfragenden Clients.
Bei den jeweiligen Clients wird nun die jeweilige Callbackfunktion ausgeführt, die zuvor zurechtgelegt wurde.

event-driven

Node.js verfolgt einen ereignisbasierten Ansatz. Für den Entwickler bedeutet das, dass er, wie im obigen Beispiel beschrieben, Callbackfunktionen an bestimmte Ereignisse binden kann, die innerhalb der Applikation auftreten können (vgl. Springer, S. 16).

across distributed devices

Clientseite: Auf Node.js-Applikationen kann von unterschiedlichsten Computern, meistens über ein Webinterface, zugegriffen werden, insbesondere von mobilen Endgeräten. Zudem können Node.js-Applikationen ihre physische Umwelt über Mirococontroller beeinflussen bzw. von ihr über Sensoren beeinflusst werden.

Serverseite: Mit den Bibliotheken *libev* und *libeio*[1] funktioniert Node.js zunächst nur auf Linux-Systemen. Da die Programmierschnittstelle bei der Entwicklung mit Node.js vom Betriebssystem *unabhängig* sein soll, werden die Bibliotheken *libev* und *libeio* durch *libuv* ersetzt. *libuv* ist ein Wrapper um beide Bibliotheken, der die Unterschiede zwischen den unterschiedlichen Betriebssystemen abstrahiert (vgl. Abb. 6 Mitte).
Die Schicht *Node Bindings* dient als „Schnittstelle zwischen V8, Eventloop, Thread Pool und den anderen Beteiligten C-Bibliotheken[2] und der darüberliegenden, in JavaScript geschriebenen *Node Standard Library*" [Schmidt] und hält sie als eine Einheit zusammen.

scalable

Für besonders leistungsstarke Webanwendungen können Node.js-Anwendungen horizontal skaliert[3] werden. D. h. die Rechenlast des Webservers kann auf zusätzliche Rechner verteilt werden, indem die einzelnen Clientanfragen untereinander aufgeteilt werden (vgl. Abb. 7).

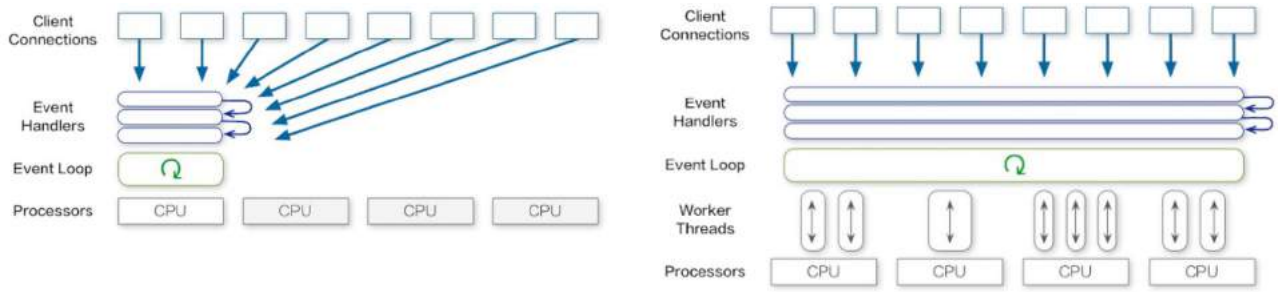


Abb. 7: Standardmäßige, nicht-skalierte (linker Teil) und horizontal skalierte (rechter Teil) Webapplikation mit Node.js

Node.js für Event Media-Installationen

Das volle Potential von Node.js-Applikationen kann durch das Hinzufügen von Zusatzmodulen nach dem Baukasten-Prinzip ausgeschöpft werden. Für eine typische Event Media Installation mit Interaktionen zwischen Client, Webserver, der physischen Umwelt und einer Datenbank können typischerweise folgende Zusatzmodule eingesetzt werden:

- **http** und/oder **express** z. B. zur Bereitstellung von Webinhalten und zur Einbindung von Middleware
- **socket.io** für den Austausch von Nachrichten zwischen Clients und dem Webserver
- **serialport** oder **firmata** zur Kommunikation mit der physischen Umwelt
- **mysql** für eine Datenbankverbindung

Nachfolgend werden einige dieser Module näher beschrieben:

1 Auf Windows-Systemen nimmt die Bibliothek *IOCP* die Rolle von *libev* ein.
2 andere Beteiligte Protokolle in Node.js: Z. B. DNS, Crypto
3 Das Gegenteil zur horizontalen Skalierung ist vertikale Skalierung, bei der der bestehende Rechner aufgerüstet wird (z. B. mehr RAM-Speicher).

3.5.1. http

Das Node.js-Modul *http* dient zum Umgang mit dem HTTP-Protokoll. Mit ihm können ein klassischer Webserver und ein HTTP-Client erstellt werden bzw. ein Webserver abgefragt werden (vgl. Springer, S. 82).
Das HTTP-Protokoll ist grundlegend für fast jede Form der Client–Server-Kommunikation in Webanwendungen.
Konkret beschreibt die Client–Server-Kommunikation einen ständig auftretenden Austausch von `request`- und `response`- Objekten.
Mit Hilfe des Moduls *http* in Node.js können Webserver realisiert werden (vgl. Listing 8):

```
var http = require('http');
var fs = require('fs');

var myServer = http.createServer(function cb(request, response) {
    response.writeHead(200, {'content-type': 'text/plain'});
    fs.readFile('client.html', function (err, data) {
        response.end(data);
    });
})
myServer.listen(3000);
```

Listing 8: Server sendet eine HTML-Datei an verbundene Clients

Mit `http.createServer(function(request, response) {..});` wird in Node.js eine Serverinstanz erstellt. Mit `myServer.listen(3000);` wird sie auf einen beliebigen TCP-Port, hier `3000`, gelegt.
Über den Formalparameter `request` erhält die Serverinstanz ein Objekt, das die Daten enthält, die der Client bei einem GET-Request an den Webserver übermittelt hat. In der Callbackfunktion `cb` befüllt die Serverinstanz das Objekt `response` mit den Daten, die zurück an den Client gesendet wird.
Das Objekt `response` kann niemals allein stehen, sondern muss *immer* mit einem Objekt `request` angefragt werden, denn HTTP ist ein requestbasiertes Protokoll.
Der HTTP-Server sendet dem Client eine statische HTML-Datei, die der Webbrowser des Clients über den zuvor festgelegten TCP-Port erreicht werden kann: `http://localhost:3000`. Beim Aufruf dieser URL im Webbrowser stellt der Client eine Anfrage an den Webserver `localhost`. `localhost` steht für die IP-Adresse des eigenen Computers, auf dem der Webserver läuft. Die Webseite liegt auf Port `3000` bereit. In diesem Fall befinden sich Client und Webserver auf *einer gemeinsamen* Maschine (*lokaler* Webserver).

Der mit Node.js und dem Modul `http` erstellte Webserver nimmt Anfragen von Clients entgegen. In der Callbackfunktion `cb` wurde im Voraus genau festgelegt, was passieren soll, wenn dieses Event, also eine Anfrage mit dem Objekt `request`, eintritt. In diesem Fall wird an das Objekt `response` eine bestimmte HTML-Datei angehängt und an den anfragenden Client gesendet. Ist die HTTP-Nachricht beim Client angekommen, wird sie dort analysiert. Der Client findet im Body ein HTML-Skript und stellt es im Fenster

seines Webbrowsers dar.
Andere Computer im Netzwerk können diese HTML-Seite ebenfalls aufrufen. In diesem Fall muss `localhost` durch die IP Adresse des Computers, auf dem der Webserver läuft, ersetzt werden, z. B. `http://10.0.1.33:3000`. Soll der Webserver auch von außerhalb des heimischen Netzwerks zugänglich sein, kann ein Port des Netzwerkrouters nach außen geöffnet werden, über den weltweit eine Verbindung auf den Webserver hergestellt werden kann. Dieses Verfahren wird *Port Forwarding* genannt. Mit Hilfe eines DNS-Dienstes kann dieser Verbindung ein eindeutiger Name im Stil `www.myWebsite.de` zugewiesen werden.

3.5.2. express

expressjs.com beschreibt sich folgendermaßen:

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

Das Modul express ist u. a. eine Erweiterung des http-Moduls. Intern enthält es einen Wrapper um das http-Modul.[1]
express wurde entwickelt, um für gewöhnliche Aufgaben bei Serverapplikationen vorgefertigte Lösungen anzubieten, die der Entwickler komfortabel nutzen kann.
„Like any abstraction, express hides difficult bits and says 'don't worry, you don't need to understand this part'. It does things for you so that you don't have to bother. In other words, it's magic." [Hahn]
express ist ein Framework, das den Entwickler bei der Umsetzung von Webapplikationen unterstützt und Arbeit reduziert.

Listing 9 zeigt, dass express, wie auch http, in der Lage ist, einen Webserver zu erstellen. Grund dafür ist, dass auch express mit dem HTTP-Protokoll umgehen kann.

Da express die Besonderheit bietet, *Middleware*[2] einzubinden, die die Umsetzung von typischen Aufgaben vereinfachen, wird http zunehmend durch express ersetzt.

Beispiele hierzu sind:

- Auslieferung von statischen Zusatzdateien, die zur HTML-Datei gehören (z. B. Stylesheets, Bilder und Schriften).
  Sie werden vom Webserver nicht ohne das Hinzutun von express an den anfragenden Client ausgeliefert.
  Mit `app.use(express.static(__dirname + '/public'));`
  kann express ein Verzeichnis auf dem Webserver, in diesem Fall

1 express enthält das http-Modul seit der Version 4.0 http-Modul und benötigt es seither nicht mehr (vgl. Springer, S. 387)
2 Eine Middleware ist ein anwendungsneutrales Programm, das zwischen Anwendungen vermittelt.

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
    res.sendfile('client.html');
});
app.listen(3000);
```

Listing 9: Beispiel-Server mit express (gleiches Ergebnis wie in Listing 8): HTML-Datei ausliefern

`/public` für die anfragenden Clients zugänglich machen.

- **Routing**: Für verschiedene Anfragen werden verschiedene Eventhandlerfunktionen gerufen. Beispielsweise behandelt der Webserver mit dem Befehl `app.get('/',..` eine Anfrage an `www.myWebsite.de`, während mit `app.get('/otherRoute',..` `www.myWebsite/otherRoute` gerufen wird.
- **Redirecting**: Mit `response.redirect('/otherRoute');` wird beispielsweise die Website `www.myWebsite/otherRoute` geladen.
- **Einbindung von externen Bibliotheken**, mit denen der Entwickler bei komplizierten Aufgaben vorgefertigte Lösungen komfortabel verwenden kann, wie z. B. multer[1]: `app.use(multer({..});`
- **Einbindung von Views** wie z. B. jade[2]:
  `app.set("views", __dirname + 'views');`

3.5.3. socket.io

socket.io beschreibt sich wie folgt:

> *Socket.IO enables real-time bidirectional event-based communication. It works on every platform, browser or device, focusing equally on reliability and speed.*

Wie das WebSocket-Protokoll grundsätzlich funktioniert, wurde bereits in Kap. 3.3 beschrieben. *socket.io* ist eine WebSocket-API, die für Node.js verfügbar ist. Um sie nutzen zu können, müssen der clientseitige und der serverseitige Code jeweils durch die entsprechende Bibliothek erweitert werden.

Funktionsweise

Listing 10 (nächste Seite) zeigt eine beispielhafte Chatanwendung.

1. Nachdem sich ein Client mit dem Webserver verbunden hat, baut der Webserver eine Socketverbindung zu ihm auf.

1 Mit der Middleware *multer* kann express einen Dateiupload realisieren. Konkret nimmt express ein Formular eines POST-Requests von einem Client entgegen. multer extrahiert die Dateien, die sich darin befinden, benennt sie ggf. um und speichert sie in einem vordefinierten Ordner auf dem Webserver ab.

2 *Jade* ermöglicht, wie HTML, mit reduzierter Syntax, ein statisches DOM aufzubauen.

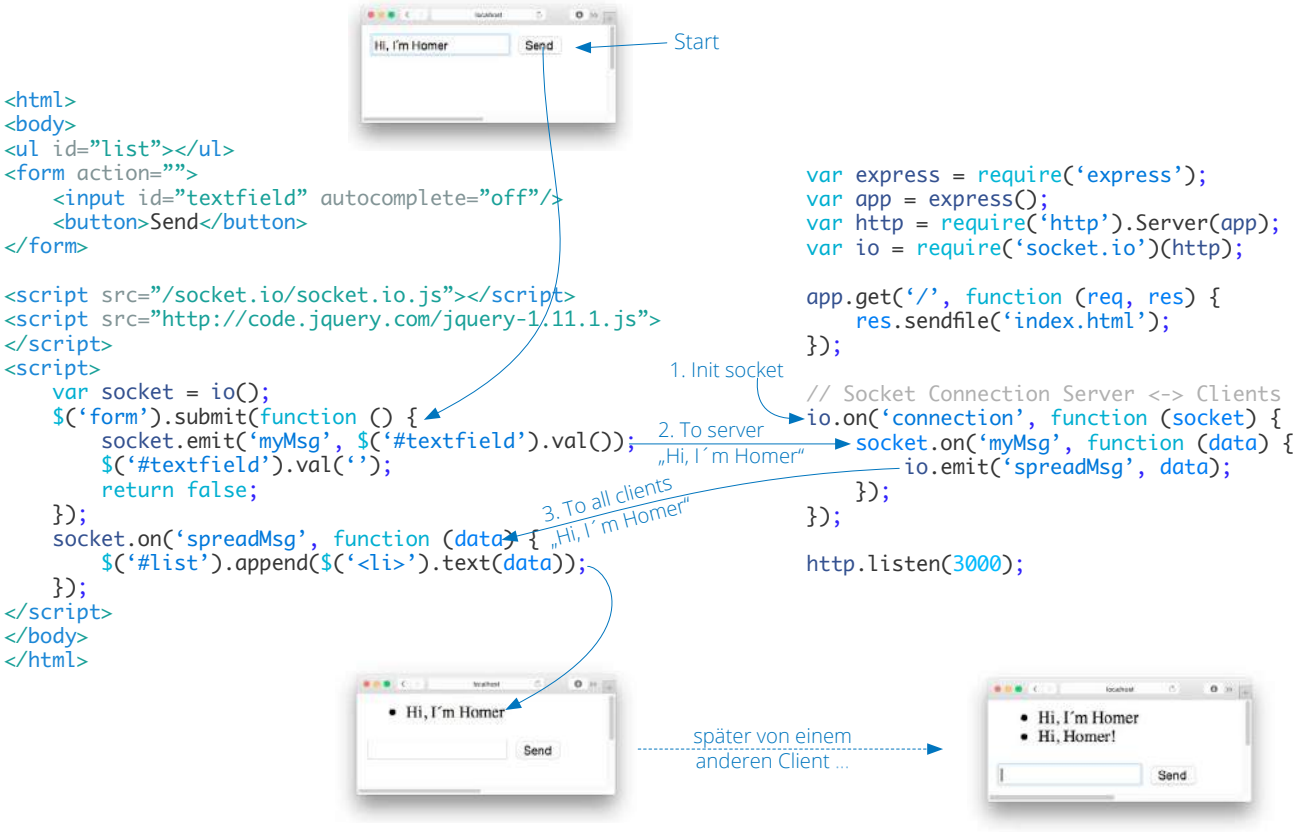


Abb. 10: Realisierung eines Chatrooms für mehrere Teilnehmer via socket.io
Links: Serverseitiger Code (JavaScript), rechts: clientseitiger Code (HTML)

2. Ein Client sendet die Nachricht `Hi, I´m Homer` an den Webserver in der Form `socket.emit("myIdentifier","myMessage");`.
3. Mit `socket.on("myIdentifier",callbackFunction);` nimmt der Webserver Nachrichten entgegen. Die beiden Kommunikationsendpunkte werden als zusammengehörig registriert, wenn ihre Identifier jeweils die gleiche Zeichenkette enthalten.
4. In der Eventhandlerfunktion sendet der Webserver die Nachricht an alle verbundenen Clients weiter. Schließlich wird die Nachrichtenliste in allen verbundenen Clients aktualisiert.

Während mit AJAX bei einem GET-Request (Client – Server *und* Server – Client) nur *eine* Anweisung benötigt wird, muss beim Umgang mit socket.io jede der beinen Kommunikationsrichtungen separat konfiguriert werden.
Für die Projekteplattform (vgl. Kap. 4: Entwicklung Projekteplattform) wird socket.io für Anfragen des Clients an eine Datenbank und zur Übertragung von Daten der Datenbank an den Client verwendet.

3.5.4. serialport

Damit der Webserver mit physischen Dingen in der realen Umwelt kommunizieren kann, benötigt er ein dafür geeignetes Kommunikationsprotokoll. Falls serielle Daten mit einem Arduino Board oder einem vergleichbaren Mi-

krocontrollerboard ausgetauscht werden, müssen serielle Daten an den seriellen Port des Computers geschrieben oder von dort aus gelesen werden (vgl. Kap. 6: Interaktion mit der physischen Umwelt).
Das Node.js-Modul serialport übernimmt Kommunikation mit dem seriellen Port des Webservers.
serialport kann in seiner Reinform oder auch über benutzerfreundlichere Zugriffsbibliotheken[1], z. B. *firmata*, verwendet werden (vgl. Kap. 6.3: Firmata).

Bei komponentenreichen Medieninstallationen können mobile Endgeräte mit physischen Gegenständen interagieren, die mit einem Webserver verbunden sind. Via Port Forwarding ist dies auch von außerhalb des lokalen Netzwerkes möglich.
An zwei praktischen Beispielen möchte ich die Interaktion zwischen Clients und einem physischen Gegenstand demonstrieren: Zunächst die Kommunikationsrichtung Client – Ding (vgl. Kap. 3.5.4.1), dann die Richtung Ding – Client (vgl. Kap. 3.5.4.2).

Für beide Kommunikationsrichtungen gilt:
Zunächst muss dem Webserver mitgeteilt werden, von welchem seriellen Port er serielle Daten abholen kann. Der serielle Port kann über die Arduino-Software oder über *Tools/Serieller Port* oder über die Konsole[2] herausgefunden werden.
Zudem muss die Baudrate (vgl. Seite 115) festgelegt werden, in diesem Fall 9.600 Bit/s.
Wenn nichts anderes angegeben wird, geht Node.js davon aus, dass jedes eintreffende binäre Datenpaket acht Zeichen lang ist und kein Paritätsbit enthält.

3.5.4.1. Client steuert Gegenstände

Abb 11 (nächste Seite) veranschaulicht den Kommunikationsweg Client – Arduino Board:
Mit einem virtuellen Button auf einer Webseite können Clients eine LED ein- oder ausschalten. Wird bei einem der verbundenen Clients dieser Button gedrückt, sendet er über eine socket.io-Verbindung eine bestimmte Zeichenkette an einen Webserver: In diesem Fall abwechselnd `1` oder `0`.
Der Webserver nimmt die Daten entgegen und sendet sie in einer Callbackfunktion sofort weiter an das Arduino Board. Dort wird die Zeichenkette, die seriell übertragen wurde, wieder in zusammenhängende Zeichenketten zusammengesetzt und analysiert. Stimmt sie mit einer auf dem Mikrocontroller hinterlegten Zeichenkette überein, in diesem Beispiel mit `1` oder `0`, wird am Pin 13 des Arduino Boards ein Spannungspegel von entweder 5V oder 0V erzeugt. Damit wird entweder die verbundene LED oder ein beliebiges anderes elektronisches Gerät ein- oder ausgeschaltet.

---

1 Eine vollständige Liste an Node.js-Modulen, die serialport verwenden, befindet sich auf https://github.com/voodootikigod/node-serialport (Stand: 15. April 2015).

2 Um den seriellen Port auf unterschiedlichen Betriebssystemen herauszufinden, gibt es keine plattformübergreifende Lösung. Bei Mac OSX lautet der Konsolenbefehl `ls /dev/tty.*` , bei Linux-Distributionen lautet er `dmesg | grep tty` und bei Windows muss mit dem Befehl `powershell` zunachst das Programm *Powershell* geöffnet werden, bevor folgender Befehl ausgeführt werden kann: `[System.IO.Ports.SerialPort]::getportnames()`

```html
<html>
<head>
    <script src="/socket.io/socket.io.js"></script>
    <script src="http://code.jquery.com/jquery-1.11.1.js">
    </script>
</head>
<body>
<button id="led">Switch LED </button>
<script>
    var socket, text, toggleVal = 0, button = $('#led');
    var socket = io();

    $('#led').click(function () {
        toggleVal += 1;
        toggleVal %= 2;

        if (toggleVal == 0) buttonState = '0';
        if (toggleVal == 1) buttonState = '1';

        socket.emit('toggleLED', buttonState);
        return false;
    });
</script>
</body>
</html>
```

Clientseitiger Code (HTML)

```javascript
var express = require('express');
var app = express();
var http = require('http').Server(app);

app.get('/', function (req, res) {
    res.sendfile('./index.html');
});
http.listen(3000);

// Receive Data from Client via socket.io
// and send it to Arduino via serialport
var socketio = require('socket.io')(http);
var spPackage = require("serialport");
var SerialPort = spPackage.SerialPort;
var portname = "/dev/tty.usbmodemfa141";
var sp = new SerialPort(portname,
    {baudrate: 9600});
sp.open();

socketio.on('connection',function (socket){
    socket.on('toggleLED',function (data){
        //send to Arduino
        sp.write(data);
    });
});
```

Serverseitiger Code (JavaScript)

1. socket.io

2. serialport

```
int ledPin = 13;
String readString;

void setup() {
    Serial.begin(9600);
    pinMode(ledPin, OUTPUT);
}

void loop() {
    while (Serial.available()) {
        delay(3);
        char
        c = Serial.read();
        readString += c;
    }

    if (readString.length() > 0) {
        Serial.println(readString);

        if (readString == "1") {
            digitalWrite(ledPin, HIGH);
        }
        if (readString == "0") {
            digitalWrite(ledPin, LOW);
        }
        readString = "";
    }
}
```

Arduino Code

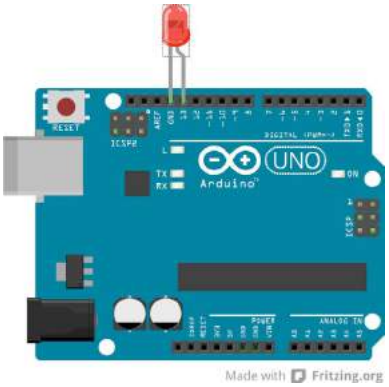


Abb. 11: Weg eines Clients zu einem Arduino Board über eine WebSocket-Verbindung (*1*) und eine serielle Verbindung über USB und TTL-seriell (*2*).

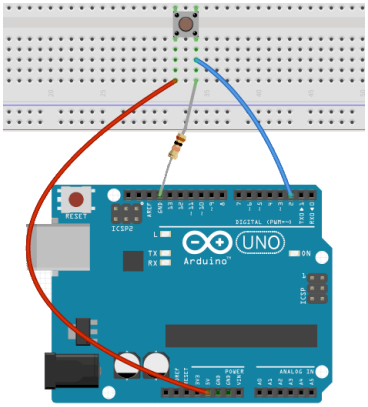
```
int btn = 2;

void setup() {
    Serial.begin(9600);
    pinMode(btn, INPUT);
}

void loop() {
    int btnState = digitalRead(btn);
    if(btnState == HIGH){
        Serial.println("1");
    } else Serial.println("0");
    delay(100);
}
```

Arduino Code

1. serialport

```
var express = require('express');
var app = express();
var http = require('http').Server(app);

app.get('/', function (req, res) {
    res.sendfile('./public/index.html');
});
http.listen(3000);

// Receive Data from Arduino via Serial Port
// and send it to Client via Socket.io
var socketio = require('socket.io')(http);
var spPackage = require("serialport");
var SerialPort = spPackage.SerialPort;
var portname = "/dev/tty.usbmodemfa141";
var sp = new SerialPort(portname, {
    baudrate: 9600,
    parser: spPackage.parsers.readline("\n")
});

sp.open(function () {
    sp.on('data', function (arduinoData) {
        socketio.emit('booleanState',arduinoData);
    });
});
```

Serverseitiger Code (JavaScript)



2. socket.io

```
<html>
<head>
    <script src="/socket.io/socket.io.js"></script>
    <script src="http://code.jquery.com/jquery-1.11.1.js">
    </script>
</head>
<body>
<script>
    var socket = io();
    socket.on('booleanState', function (msg) {
        var received = msg.trim();
        if (received == "1")
            $("body").css("background-color", "green");
        if (received == "0")
            $("body").css("background-color", "red");
    });
</script>
</body>
</html>
```

Clientseitiger Code (HTML)

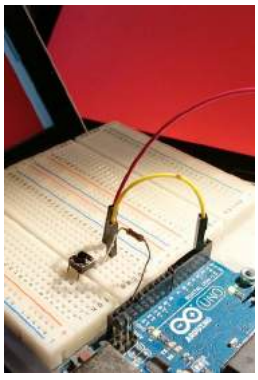
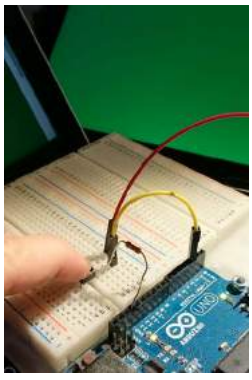
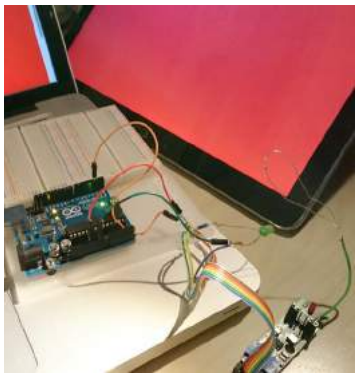
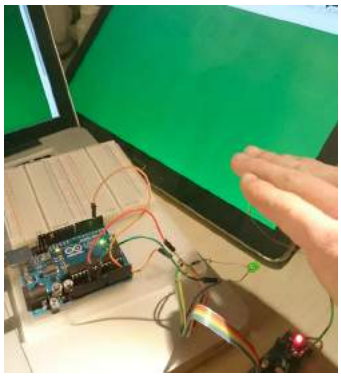





Abb. 12: Weg von einem Arduino Board zu den verbundenen Clients über eine serielle Verbindung über TTL-seriell und USB (*1*) und über einen WebSocket (*2*). Test mit einem Taster (Bilder 1 und 2 v. l.) und mit einem kapazitiven Touchsensor (Bilder 3 und 4 v. l.).

3.5.4.2. Gegenstand steuert Clients

Abb. 12 veranschaulicht den Kommunikationsweg Arduino Board – Clients: An einem Arduino Board ist ein beliebiger Sensor angeschlossen. In Abb. 12 werden der Einfachheit halber Sensoren gewählt, die digital nutzbare Spannungen[1] liefern: Ein physischer Button und ein kapazitiver Touchsensor. Folgende Schritte sind identisch für beliebige digitale Sensoren:
Das Arduino Board nimmt am digitalen Eingang *D2* Spannungen entgegen. Abhängig vom Spannungspegel `HIGH` oder `LOW`, also entweder 5V oder 0V, sendet es eine Zeichenkette, hier `1` oder `0`, an den Webserver. Dieser nimmt sie entgegen und sendet sie in einer Callbackfunktion über eine socket.io-Verbindung direkt weiter an alle verbundenen Clients. In den Browsern der Clients wird die Nachricht analysiert. Abhängig von der eintreffenden Nachricht, hier `1` oder `0`, färbt sich der `<body>` der Webseiten grün oder rot.

3.5.5. mysql

Mit einer Datenbankanbindung erhält eine komponentenreiche Medieninstallation ein *Gedächtnis*. In diesem Beispiel wird eine MySQL-Datenbank verwendet. Alternativen werden in Kap. 5.3. (Versch. NoSQL-Datenbanken) behandelt.
Mit dem Modul *mysql* ist es möglich, mit Node.js *asynchron* auf eine MySQL-Datenbank zuzugreifen.

Abb. 13 veranschaulicht die Kommunikation eines Clients über einen Node.js-Webserver mit einer MySQL-Datenbank:
Nachdem der Nutzer auf einem clientseitigen Webinterface den Button *Add & Show list* gedrückt hat (*1* und *2*), werden die Inhalte der beiden Textfelder *Type* und *Qty* über eine socket.io-Verbindung an den Webserver übermittelt (*3*). Dieser sendet das Datenobjekt an die Datenbankhandlerfunktion `add_data` (*4*). Dort wird das Datenobjekt mit Hilfe eines `INSERT`-Befehls in eine relationale Tabelle einer MySQL-Datenbank eingearbeitet (*5* und *6*). Nach Fertigstellung dieser Aufgabe (*7* und *8*) wird die nächste Datenbankhandlerfunktion `get_data` gerufen (*9*), die den aktuellen Bestand der Tabelle abfragt und an die verbundenen Clients sendet. Mit einem `SELECT`-Befehl sollen Daten aus einer relationalen Datenbank abgerufen und in die objektorientierte Struktur von JavaScript eingearbeitet werden.
Grob läuft das wie folgt ab:
Das Modul *mysql* wandelt das Ergebnis der Abfrage einer relationalen Datenbank in ein Array von verschachtelten JavaScript-Objekten um (vgl. Array zwischen *11* und *12*).
Bei relationalen Datenbanken werden Daten in zweidimensionalen Tabellen organisiert. Jedes Feld wird über einen Spaltennamen und i. d. R. über eine Zeilen-ID referenziert.
Bei einer Datenbankabfrage schreibt das Modul *mysql* jedes abgefragte

1 Digital nutzbare Spannungen an einem Arduino Board: Entweder 5V (auch möglich: 3,3V) für eine logische `1` oder 0V für eine logische `0`, keine Zwischenschritte, wie es bei analogen Sensoren der Fall ist

id	type	qty
1	Chocolate	2
2	Bonbons	18

Vorbefüllte Tabelle

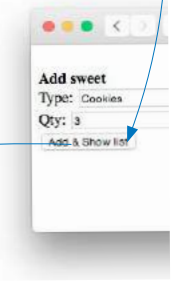
Client-Code (HTML)

```
<html>
<head>
    <script src="/socket.io/socket.io.js"></script>
    <script src="jquery-latest.min.js">
    </script>
</head>
<body>
<br> <b>Add sweet</b>
<div>Type: <input type='text' id='type' value=''></div>
<div>Qty: <input type='text' id='qty' value=''></div>
<button id='save'> Add & Show list</button>
<ul id='list'></ul>

<script>
    var socket = io();

    // Insert data
    $('#save').click(function () {
        var data = {
            type: $('#type').val(),
            qty: $('#qty').val()
        };
        socket.emit('toServer', data);

        $('#type').val('');
        $('#qty').val('');
    });

    // Pull data
    socket.on('toClient', function (data) {
        var list = "";
        $.each(data, function (i, obj) {
            list+="<li>"+obj.qty+"x "+obj.type+"</li>";
        });
        $('#list').html(list);
    });
</script>
</body>
</html>
```

Start

1

2



Server-Code (JavaScript)

```
var express = require('express');

var app = express();
var http = require('http').Server(app);
var io = require('socket.io')(http);
var mysql = require('mysql');

app.get('/', function (req, res) {
    res.sendfile('./index.html');
});

http.listen(3000);

// Communication with clients and database
io.on('connection', function (socket) {
    socket.on('toServer', function (data) {
        add_data(data, function () {
            get_data(function (param) {
                socket.emit('toClient', param);
            });
        });
    });
});

// Insert data from clients
function add_data(data, callback) {
    db.query("insert into sweetslist (type,qty)"+
        " values (?,?)", [data.type, data.qty],
        function (err, res){
            callback(res);
        });
}

// get data
function get_data(callback) {
    db.query("select * from sweetslist",
    function (err, rows) {
        // console.log(rows);
        callback(rows);
    });
}

// DB connection
var HOST = 'localhost';
var PORT = 8889;
var MYSQL_USERNAME = 'root';
var MYSQL_PASSWORD = 'root';

var db = mysql.createConnection({
    host: HOST,
    port: PORT,
    user: MYSQL_USERNAME,
    password: MYSQL_PASSWORD,
});

// DB Config Script goes here
```

3

4

5

```
{ id:3, type:'Cookie', qty:3 }
```

6

id	type	qty
1	Chocolate	2
2	Bonbons	18
3	Cookies	3

7

8

9

10

id	type	qty
1	Chocolate	2
2	Bonbons	18
3	Cookies	3

11

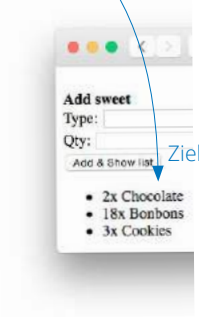
```
[ { id: 1, type: 'Chocolate', qty: 2 },
  { id: 2, type: 'Bonbons', qty: 18 },
  { id: 3, type: 'Cookies', qty: 3 }  ]
```

12

13

14

15



Ziel

Listing 13: Übersicht der Interaktion eines Nutzers mit einer MySQL-Datenbank über ein Webinterface mit Zwischenschritten: Click → Übermittlung des Formulars an Server → Einfügeoperation (+ Kommunikation mit Datenbankserver) → Abfrage (+ nochmal Kommunikation mit Datenbankserver) → Übermittlung der Daten an verbundeneClients → Ergebnis formatieren, anzeigen

Tabellenfeld in ein separates Objekt, bestehend aus einem *key* und einem *value*. *key* ist der jeweilige Spaltenname, *value* ist der eigentliche Inhalt des Feldes. Eine Tabellenzeile wird als übergeordnetes Objekt dargestellt, das alle Feldobjekte in der entsprechenden Zeile eines relationalen Datensatzes umfasst. Infolgedessen beinhaltet jedes Zeilenobjekt gleich viele Feldobjekte, bei denen die Namen für *key* übereinstimmen. Das Serverprogramm erhält das Objekte-Array (*12* und *13*). Über eine zweite socket.io-Nachricht wird es den verbundenen Clients übergeben (*14*). Jeder Client analysiert das ankommende JavaScript-Objekt und aktualisiert die Liste (*15*) im Webbrowser.

3.6. Zusammenfassung und Überleitung

In diesem Kapitel wurden einige Möglichkeiten geprüft, mit denen verschiedene Komponenten einer Applikation miteinander verbunden werden.
Mit dem HTTP-Protokoll sowie den 2.0-Technologien AJAX und WebSockets wurden einige Methoden zur Kommunikation zwischen Clients und einem Webserver betrachtet.
In diesem Zusammenhang wurden mit dem *Baukastensystem* Node.js einige Möglichkeiten von JavaScript außerhalb eines Webbrowsers geprüft, die in Event Media-Installatioen geläufig sind.
Mit den Erkenntnissen aus Kap. 2 wurden die konzeptionellen Anforderungen für eine komponentenreiche Medieninstallation aus Kap. 1.2.4 exemplarisch umgesetzt. Damit ist ein erster Grundaufbau entstanden, in dem mit Hilfe von JavaScript Computersysteme mit ihrer physischen Umwelt kommunizieren können, beispielsweise über eine beliebige Anzahl von mobilen Endgeräten. Mit diesen Mitteln kann ein einfaches *Internet der Dinge* umgesetzt werden.

In Kap. 4 (Entwicklung Projekteplattform) werden bisherige Erkenntnisse in die Praxis umgesetzt.

In Kap. 5 (Datenbank) und Kap. 6 (Interaktion mit der physischen Umwelt) werden die Komponenten Datenbank und Mikrocontrollerboard näher betrachtet.

4. Entwicklung Projekteplattform

Die Vertriebskollegen von ICT wünschten eine Software, in der interessante Projekte aus der Vergangenheit bzw. Projektvorschläge nach den konzeptionellen Ansätzen aus Kap. 1.2.2 (Acht mögliche Präsentationsformen) archiviert werden. Mit unterschiedlichen Suchkriterien sollen sie abgerufen und grafisch aufbereitet werden können. Die Software soll Präsentationszwecken dienen. Sie soll den Kunden von ICT einen Überblick über Trends in der Präsentationstechnik geben und ihnen näher bringen, welche Anforderungen an künftige Produktpräsentationen gestellt werden. Die Projekteplattform ist eine Übung für die spätere Umsetzung einer komponentenreichen Medieninstallation und nutzt dieselben Technologien.

4.1. Planung der Software

Anforderungen an die Software

- Sie soll **plattform**- und **geräteübergreifend** sein, ggf. auch offline.
- Die grafische Oberfläche soll **übersichtlich** und intuitiv bedienbar sein.
- Sie soll ebenso intuitiv durch Vertrieb und Marketing **erweiterbar** sein
- Suchergebnisse sollen nach **verschiedenen Kriterien** gefiltert werden können: z. B. Kategorie, Zeitraum, Branche, Messe, Ort, beteiligte Agenturen, ähnliche Projekte, verwendete Technik und verwendete Software. Die Abfragelogik für die Suche nach Projektkategorien, wie in Kap. 1.2.2 konzipiert, wird in Abb. 2 auf der nächsten Doppelseite dargestellt.

Für die Umsetzung der Projekteplattform werden bewährte Mittel eingesetzt, mit denen mehrere Mitarbeiter vertraut sind. Da mehrere Personen später am selben System operieren, bietet sich eine Webanwendung mit einem zentralen Server an.

Benötigte Komponenten und verwendete Mittel

- Client: HTML, CSS, JavaScript
- Server: Node.js, mit den Modulen mysql, socket.io, express
- Datenbank: MySQL

Konzeption der Software

- Modellierung einer MySQL-Datenbank, die Datenabfragen nach unterschiedlichen Kriterien zulässt
- Entwicklung einer grafischen Webanwendung, die mit einer Datenbank interagiert (in diesem Fall ein Client- und ein Serverprogramm)

4.2. Modellierung der Datenbank

Die Einträge der Datenbank sind aufgrund der Anforderungen in Kap. 4.1 (Planung der Software) stark untereinander verlinkt. Es entsteht ein umfangreiches Beziehungsgeflecht aus Tabellen, deren Einträge über *Join-Abfragen* miteinander verbunden werden können.

4.2.1. Ausgangssituation

Zu Beginn der inhaltlichen Konzeption (vgl. Kap. 1.2.2) wurden alle Informationen in zwei Google-Tabellen gehalten:

- In einer Tabelle *kunde* wurden während der Kundenanalyse alle Informationen über potenzielle Kunden von ICT gespeichert: U. a. Branche, Produkte, Messeteilnahmen, Einschätzungen zur Innovationsfreudigkeit, aktuelle Interessengebiete und Kontaktdaten.
- In einer Tabelle *projekte* wurden Projektvorschläge und interessante Projekte aus der Vergangenheit gesammelt, und in die in Kap. 1.2.2 konzipierten Kategorien eingeordnet.

4.2.2. Umsetzung

Die beiden eben genannten Google-Tabellen *kunde* und *projekte* werden für die Entwicklung der Projekteplattform in einer MySQL-Datenbank organisiert. Die Ausgangstabelle dieser Projektedatenbank ist `projekte`. Sie enthält alle Inhalte der beiden *kunde* und *projekte* (vgl. Abb. 1).
In MySQL wurde die recht umfangreiche Tabelle `projekte` (vgl. Abb. 3 auf Seite 84) derart organisiert, dass alle zusammenhängenden Tabellenspalten in Untertabellen ausgelagert wurden.

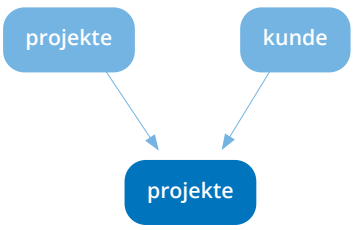


Abb. 1: Zwei Google-Tabellen werden in eine MySQL-Datenbank zusammengeführt.

4.2.2.1. Atomare Werte

In ihrer Ausgangsversion enthält die Tabelle `projekte` viele Tabellenspalten, in denen jeweils *mehr als ein* Wert vorkommen kann. Einzelne Tabellenfelder enthalten somit *Wiederholungsgru*ppen (vgl. Abb. 3 auf Seite 84).
Beispielsweise enthält die Tabelle `projekte` eine Spalte `visuals`, die zwei Visuals enthält: *Backstage.jpg* und *Showcase.mp4*. Da sich beide Visuals ein einziges Feld teilen, werden sie von der Datenbank als *eine Einheit* betrachtet. Zwar können beide Visuals später durch Zeichenanalyse voneinander separiert werden, allerdings werden sie bei Datenbankabfragen immer miteinander verbunden sein und können nicht als einzelne Entitäten behandelt werden.

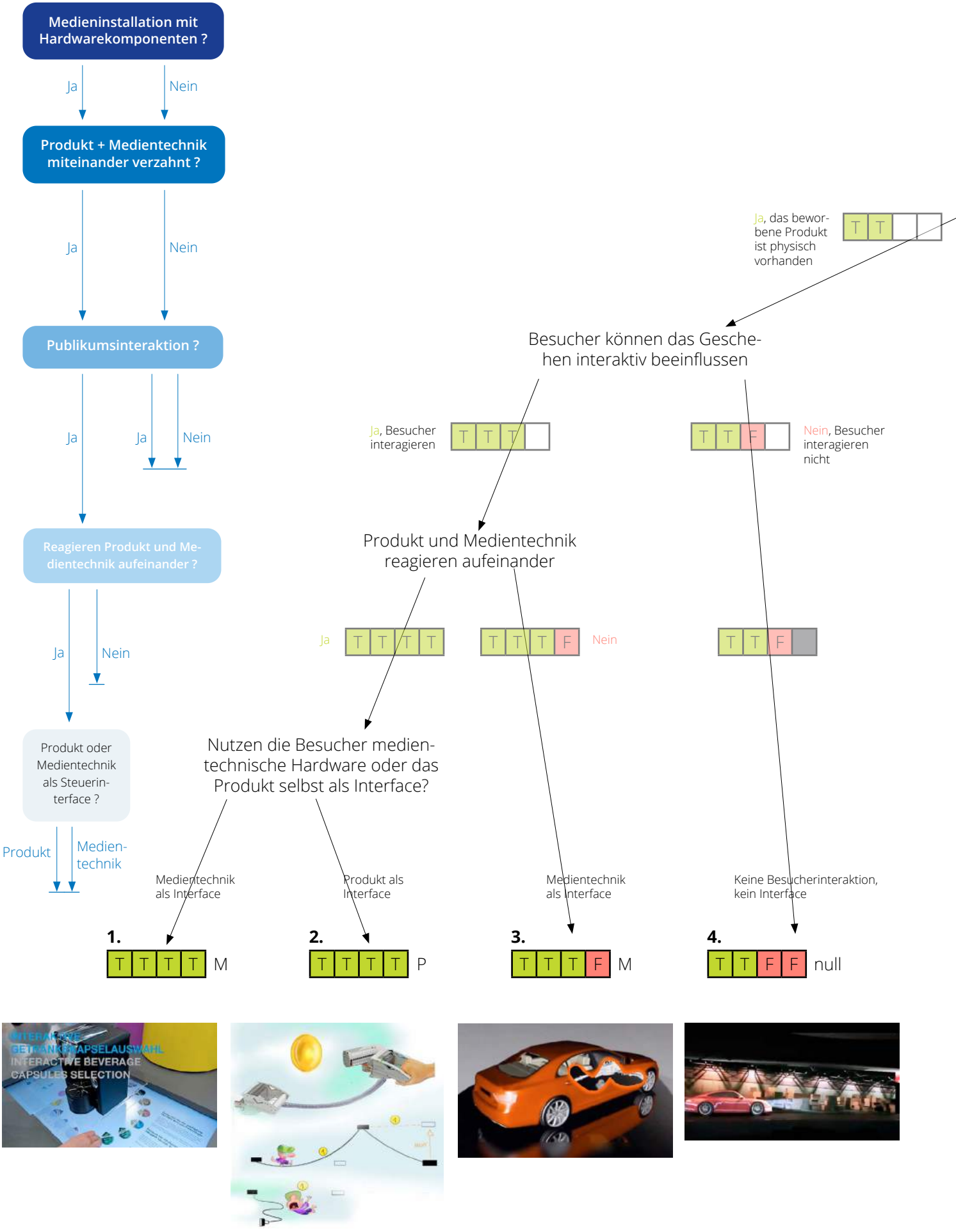


Abb. 2: Implementierung der Abfragelogik bei der Suche von Projekten nach den in Kap. (1.2.2. Acht mögl. Präsentationsformen) festgelegten Kategorien:
Linker Teil: Im User interface wählt der Nutzer mit mindestens drei, maximal 5 Abfragen eine Kategorie aus.
Rechter Teil: Im Programm wird die Selektion des Nutzers in Wahrheitswerte umgesetzt. Je nach Folge von Wahrheitswerten registriert das Programm eine Zahl zwischen 1 und 8, mit der der Datenbankbestand nach Kategorien gefiltert wird.

Medieninstallation besteht aus sowohl physischen als auch virtuellen Komponenten

Ja T

Nein, nur virtuelle Komponenten F

Produkt und Medientechnik sind miteinander verzahnt (stehen im Kontext zueinander)

T F

Nein, die physisch vorhandenen Gegenstände sind ebenfalls Medienkomponenten. Das Produkt selbst ist nur virturll präsent.

F

keine physischen Gegenstände vorhanden, also das Produkt selbst whrsch. auch nicht

Besucher können das Geschehen interaktiv beeinflussen

Ja, Besucher interagieren T F T

Nein, Besucher interagieren nicht T F F

Ja, Besucher interagieren F T

Nein, Besucher interagieren nicht F F

T F T

T F F

F T

F F

Medientechnik als Interface

Keine Besucherinteraktion, kein Interface

Medientechnik als Interface

Keine Besucherinteraktion, kein Interface

5. T F T F M

6. T F F F null

7. F F T F M

8. F F F F null

projekt_id (PK)	bezeichnung	insert_datum	interaktionsform	Kategorie	event	projekt_datum	projektvorschlag	moegl_kunde	agentur	umsetzer	projektvorschlag	quelle	wirkung	werbegeschenk	techDetails	software	beschreibung	visuals	aehnlicheProjekte
26	BMW@IAA2015	2014-12-29	klettern	mit Hardwarekomponenten, Produkt und Medientechnik verzahnt, Publikumsinteraktion, Produkt und Medientechnik reagieren aufeinander, Medientechnik als Steuerinterface	null	null	TRUE	ZF + Kontaktdaten, Festo + Kontaktdaten, ABB + Kontaktdaten, Phoenix + Kontaktdaten, Bosch Termotechnik + Kontaktdaten	NULL	ICT	TRUE	http://www.hdm-stuttgart.de/event-media/	Spiel	FALSE	Kap. Touchsensor, WIFI	Node.js-Applikation	Der Besucher interagiert im Ganzkörpereinsatz direkt an einer interaktiven Kletterwand, die als Videospiel funktioniert und in Wirklichkeit ein überdimensionales Display ist. Klettern wird in elektrische Energie umgesetzt, mit der ein Gerät angetrieben wird. Schnelleres Klettern -> schnellerer Betrieb. Denkbar ist auch: Lageenergie erklettern(Höhe zurücklegen).	Backstage.jpg, Showcase.mp4	Wasserfallklettern, Inspektorklettern

Abb. 3: Zusammenführung aller Daten aus der inhaltlichen Konzeption in eine denormalisierte Tabelle `projekte`: Sie enthält Wiederholungsgruppen (siehe Markierungen) und ist ohne Normalisierung nicht in einer relationalen Datenbank abbildbar.

Ist beispielsweise ein *Projekt A* mit beiden Visuals verlinkt, also sowohl mit `Backstage.jpg`, als auch mit `Showcase.mp4`, stellen Wiederholungsgruppen kein Problem dar. Ist allerdings ein *Projekt B* nur mit `Backstage.jpg`, nicht aber mit `Showcase.mp4`, verknüpft, müssen Wiederholungsgruppen in atomare Werte aufgelöst werden.
Jedes Feld sollte maximal einen atomaren[1] Wert enthalten.

Der Wunsch nach Atomarisierung führt zur **ersten Normalform**: Wiederholungsgruppen werden aufgelöst, indem für jeden ihrer Werte eine separate Tabellenzeile genutzt wird. Alle gemeinsamen Tabellenfelder werden redundant wiederholt (vgl. Abb. 4: Die Zeichenkette `BMW@IAA2015` wird redundant gehalten).

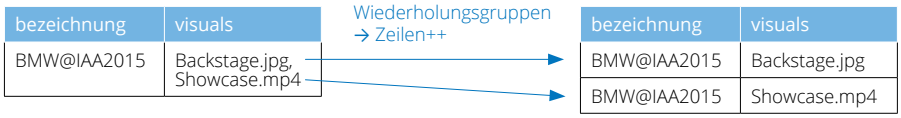


Abb. 4: Überführung einer denormalisierten Tabelle (links) in die erste Normalenform (rechts)

In Abb. 3 muss der Name des Projekts für jedes Visual wiederholt werden. Das ist speicherineffizient und kann zu *Einfüge-, Update- und Delete-* ***Anomalien*** führen: Wenn beispielsweise die Bezeichnung des Projekts `BMW@IAA2015` umbenannt werden soll, muss die Änderung bei jeder der beiden Zeilen separat vorgenommen werden. Es muss dabei darauf geachtet werden, dass die jeweiligen neuen Namen exakt gleich geschrieben werden. Andernfalls erkennt die Datenbank die beiden Datensätze nicht als zusammengehörig an. Anomalien können auch beim Einfügen neuer Datensätze auftreten. Dieses Problem führt zur Auflösung der Tabelle `projekte` in *mehrere* Entitätstypen.

Relationale Datenbanken können Daten nur in *zweidimensionalen Tabellen* halten. Anders als Dokumentendatenbanken (vgl. Kap. 5.3: Verschiedene

1 Atomar: Nicht weiter zerlegbar

NoSQL-Datenbanken) können sie keine verschachtelten Objekte speichern. In diesem Fall ist jedoch eine Verschachtelung von Daten nötig, beispielsweise wenn `Backstage.jpg` und `Showcase.mp4` mit demselben Projekt verbunden sind. Das wird mit relationalen Datenbanken umgesetzt, indem Datensätze auf mehrere Tabellen verteilt und über Fremdschlüsselbeziehungen miteinander verbunden werden. Die Abfrage der Daten erfolgt über **Inner Joins**[1]. Das wird in der **zweiten Normalform** beschrieben:
Sie fordert zusätzlich zu den Bedingungen der ersten Normalform, dass „alle Nichtschlüsselattribute vollfunktional von einem eindeutig identifizierbaren Schlüssel [candidate key] abhängig sind“ [Hinkelmann].
In Abb. 5 hängen alle Visuals vom Projekt `BMW@IAA2015` ab. Der eindeutig identifizierbare Schlüssel sollte kein informationstragender Wert sein. Um die Datensätze zu verbinden, die vor der zweiten Normalform einen *einzigen* Datensatz in einer *einzigen* Tabelle bildeten und nun auf *zwei* Tabellen verteilt werden, wird als Bindeglied die ID von `BMW@IAA2015` verwendet.

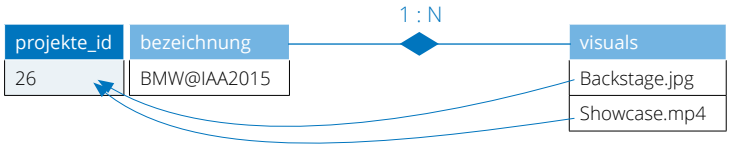


Abb. 5: Überführung in die zweite Normalform, indem die Daten auf zwei Tabellen (`projekte` und `visuals`) verteilt werden. Die beiden Tabellen sind über eine FK-Beziehung miteinander verbunden. Zusammenhängende Datensätze werden mit Hilfe einer *Inner Join*- Abfrage wieder miteinander verbunden.

Jede Tabelle im Tabellengeflecht der Datenbank für die Projekteplattform verfügt über einen Primärschlüssel (*Primary Key, kurz PK)* mit der Bezeichnung `<tabelle>_id`, die aus einer selbstinkrementierenden Zahl[2] besteht.

4.2.2.2. 1:N-Beziehungen

In Abb. 5 stehen die beiden Tabellen in einer 1:N-Beziehung zueinander: Zum Projekt `BMW@IAA2015` kann es *mehr als ein* Visual geben, aber jedes Visual kann zu *genau einem* Projekt zugeordnet werden, nämlich zu `BMW@IAA2015`.
„Eine 1:N-Beziehung ist der klassische Fall einer **PK/FK**[3]**-Beziehung**“ [Hinkelmann]. Da der Primärschlüssel der Tabelle `projekte` als Fremdschlüsselwert *(Foreign Key, kurz FK)* in der Tabelle `visuals` vorkommt, aber der Primärschlüsselwert der Tabelle `projekte` *nicht identisch* mit dem Primärschlüsselwert von `visuals` ist, handelt es sich um eine 1:N-Beziehung. Fremdschlüsselbeziehungen werden auch ***referentielle Integrität*** **(RI)**[4] genannt. Abb. 6 zeigt, wie `projekte` und `visuals` über eine 1:N-Beziehung miteinander verknüpft werden können.

1 Inner Join: Datensätze aus zwei Tabellen werden bei einer Abfrage miteinander verknüpft, wenn ein gemeinsames Feld die gleichen Werte enthält. Dazu gibt zwei mögliche SQL-Befehle:
SQL-Syntax ´89 (logischer):
`SELECT * FROM tab1, tab1 WHERE tab1.tab2_id = tab2.id`
SQL-Syntax ´92 (geläufiger):
`SELECT * FROM tab1 INNER JOIN tab2 ON tab1.tab2_id = tab2.id`
2 Selbstinkrementierende Zahl: `AUTO_INCREMENT`
3 PK: Primary Key, FK: Foreign Key
4 Referentielle Integrität: zu jedem Fremdschlüsselwert gibt es einen passenden Primärschlüsselwert (PK).

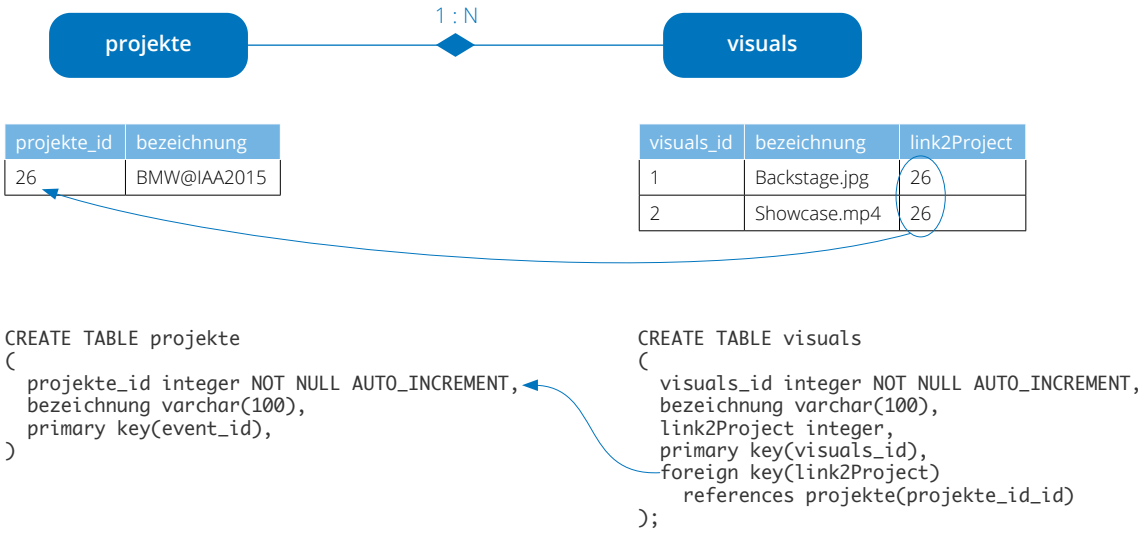


Abb. 6: Verbindung der Tabellen `projekte` und `visuals` über eine 1:N-Beziehung

4.2.2.3. N:M-Beziehungen

Das eben beschriebene Modell funktioniert nur, wenn die beiden Einträge in `visuals` nur für den Datensatz mit der Bezeichnung `BMW@IAA2015` in `projekte` verfügbar sein sollen. Nun soll allerdings beispielweise `Backstage.jpg` zusätzlich mit einem *anderen* Projekt als `BMW@IAA2015` verlinkt werden, z. B. mit `BMW@Autosalon`.
Gefordert wird: Zu einem Projekt kann es *mehr als ein* Visual geben und jedes Visual kann zu *mehr als einem* Projekt gehören. Es wird eine Many-to-Many-Beziehung (N:M) benötigt. Die Abbildung einer N:M-Beziehung im physischen ***Entity Relation Model (ERM)*** ist vorerst nicht möglich (vgl. Hinkelmann, S. 154) und muss in *zwei* 1:N-Beziehungen aufgelöst werden. Dazu muss zwischen `projekte` und `visuals` eine zusätzliche, schwache Entität eingefügt werden: `projekte_visuals`.
Abb. 7 zeigt die Tabellenstruktur die sich daraus ergibt.

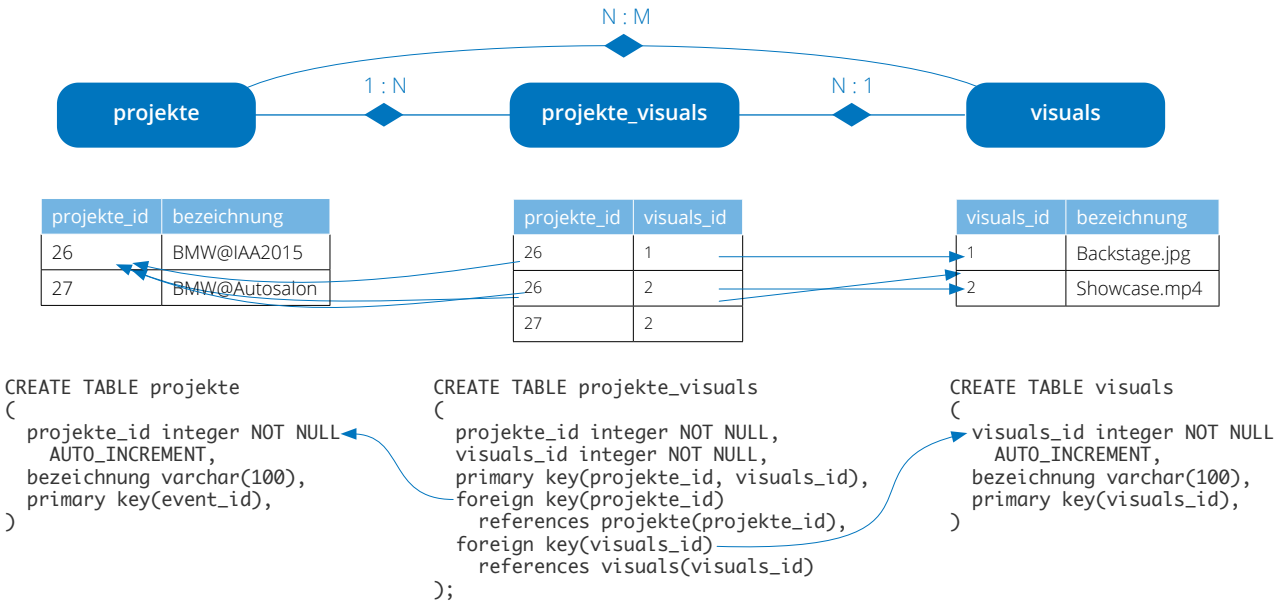


Abb. 7: Verbindung der Tabellen `projekte` und `sweets` über eine N:M-Beziehung

Bei der Erstellung der Tabellen ist zu beachten, dass `projekte` und `visuals` bereits bestehen müssen, wenn `projekte_visuals` erstellt wird.
Beim Anlegen eines Datensatzes über mehrere Tabellen (`projekte`, `visuals` und `projekte_visuals`), die über eine N:M-Beziehung in Verbindung miteinander stehen, müssen die Datensätze in `projekte` und `visuals` bereits bestehen, bevor ihr Bindeglied `projekte_visuals` mit Daten befüllt wird.
In der Tabellenübersicht zur fertigen Projektedatenbank in Abb. 9 auf der nächsten Doppelseite wird die Erstellungs- und Einfügereihenfolge in grauer Schrift dargestellt. Die Löschreihenfolge ist umgekehrt.

4.2.2.4. Dritte Normalform

In einzelnen Fällen wird in der Projektedatenbank auch die dritte Normalform eingesetzt. Sie besagt, dass ein „Nichtschlüsselattribut von anderen Nichtschlüsselattributen herleitbar ist." In diesem Fall ist es transitiv.
Abb. 8 veranschaulicht, dass sich der Ortsname durch die ID `10` herleiten lässt. So muss der Name `Frankfurt/M` nicht mehr redundant gehalten werden.

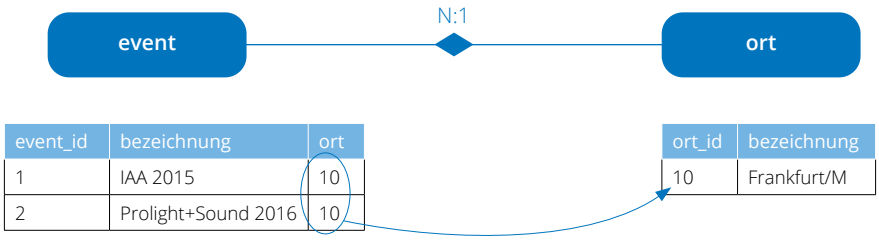


Abb. 8: Demonstration der dritten Normalform

4.2.2.5. Datentypen

Bei den Datentypen für u. a. Telefonnummern, Postleitzahlen habe ich mich für den Datentyp `varchar` (Zeichenkette) entschieden, da sie Buchstaben enthalten können und führende Nullen im Datentyp integer ignoriert werden.
Abb. 9 zeigt das Beziehungsgeflecht der vollständigen Projektedatenbank (Entity Relation Model).

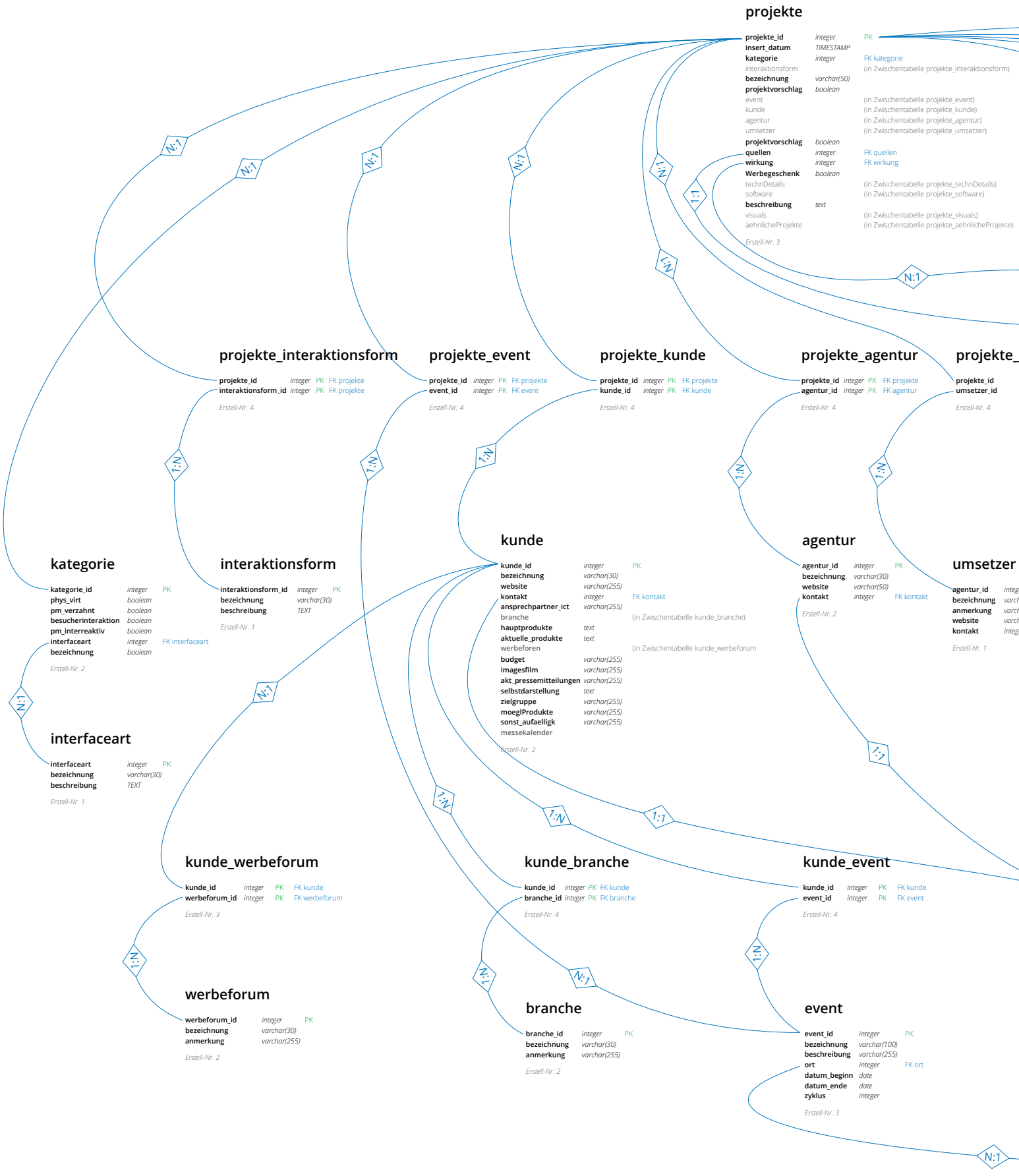
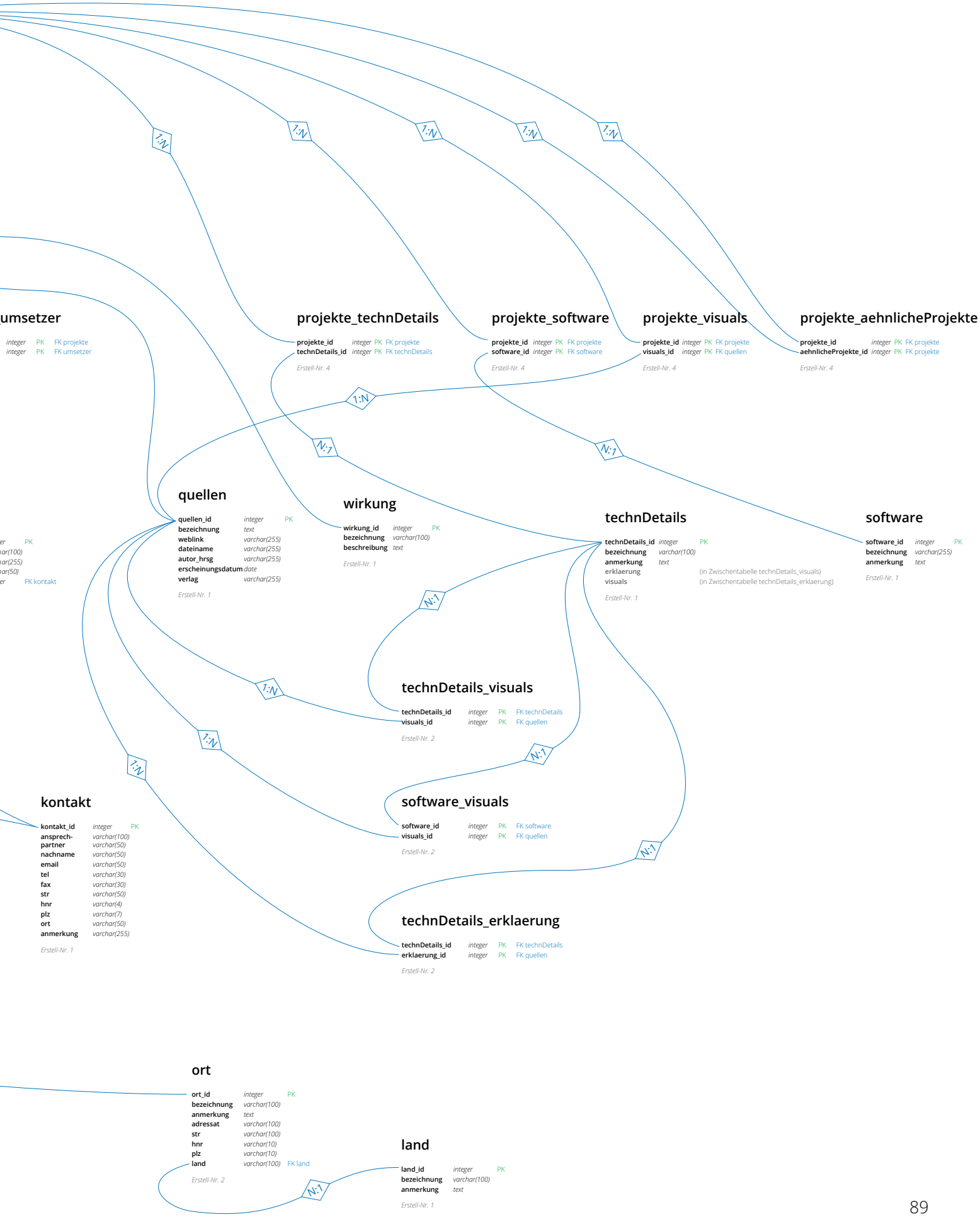


Abb. 9: Tabellenübersicht der MySQL-Datenbank für die Projekteplattform (Entity Relation Model)
SQL-Skript: https://github.com/fabifiess/sweets_app/blob/master/scripts_server/sql_db-bau.txt (Stand: 18. Januar 2015)


1:N
1:N
1:N
1:N
1:N
umsetzer
integer PK FK projekte
integer PK FK umsetzer
projekte_technDetails
projekte_id integer PK FK projekte
technDetails_id integer PK FK technDetails
Erstell-Nr. 4
projekte_software
projekte_id integer PK FK projekte
software_id integer PK FK software
Erstell-Nr. 4
projekte_visuals
projekte_id integer PK FK projekte
visuals_id integer PK FK quellen
Erstell-Nr. 4
projekte_aehnlicheProjekte
projekte_id integer PK FK projekte
aehnlicheProjekte_id integer PK FK projekte
Erstell-Nr. 4
1:N
N:1
N:1
quellen
quellen_id integer PK
bezeichnung text
weblink varchar(255)
dateiname varchar(255)
autor_hrsg varchar(255)
erscheinungsdatum date
verlag varchar(255)
Erstell-Nr. 1
wirkung
wirkung_id integer PK
bezeichnung varchar(100)
beschreibung text
Erstell-Nr. 1
technDetails
technDetails_id integer PK
bezeichnung varchar(100)
anmerkung text
erklaerung (in Zwischentabelle technDetails_visuals)
visuals (in Zwischentabelle technDetails_erklaerung)
Erstell-Nr. 1
software
software_id integer PK
bezeichnung varchar(255)
anmerkung text
Erstell-Nr. 1
er PK
har(100)
har(255)
har(50)
er FK kontakt
N:1
1:N
technDetails_visuals
technDetails_id integer PK FK technDetails
visuals_id integer PK FK quellen
Erstell-Nr. 2
1:N
1:N
N:1
kontakt
kontakt_id integer PK
ansprech-partner varchar(100)
nachname varchar(50)
email varchar(50)
tel varchar(30)
fax varchar(30)
str varchar(50)
hnr varchar(4)
plz varchar(7)
ort varchar(50)
anmerkung varchar(255)
Erstell-Nr. 1
software_visuals
software_id integer PK FK software
visuals_id integer PK FK quellen
Erstell-Nr. 2
N:1
technDetails_erklaerung
technDetails_id integer PK FK technDetails
erklaerung_id integer PK FK quellen
Erstell-Nr. 2
ort
ort_id integer PK
bezeichnung varchar(100)
anmerkung text
adressat varchar(100)
str varchar(100)
hnr varchar(10)
plz varchar(10)
land varchar(100) FK land
Erstell-Nr. 2
N:1
land
land_id integer PK
bezeichnung varchar(100)
anmerkung text
Erstell-Nr. 1

4.3. Entwicklung einer dynamischen Webanwendung

Die Applikation verfügt über zwei abgetrennte Bereiche:

- Einen Präsentationsbereich (für Kunden)
- Einen Wartungsbereich (für Administratoren: Daten ändern, hinzufügen, verlinken, löschen).

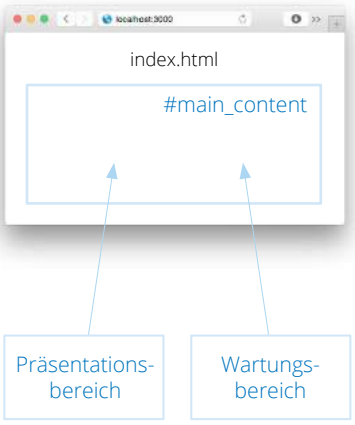


Abb. 10: index.html enthält einen `<div>`-Container `#main_content`, in den der Präsentationsbereich (für Kunden) oder der Wartungsbereich (für ICT-Mitarbeiter) geladen wird.

Mit der AJAX-Methode `$.load` aus der JQuery-Bibliothek kann der entsprechende Seiteninhalt in einen dafür vorgesehenen Bereich von index.html hineingeladen werden (vgl. Abb. 10).

Um Fehler besser erkennen zu können und um den Speicherplatz auf meinem Online-VCS[1] (Github) zu schonen, verwendete ich zur Entwicklung der Projekteplattform einen stark reduzierten Datenbestand. Zur Entwicklung der Programmlogik erstellte ich eine überschaubare Süßigkeitendatenbank mit drei Tabellen: `sweets`, `visuals` und `sweets_visuals`. Sie werden im späteren Verlauf durch die Projektedatenbank aus Abb. 9 ersetzt.

Die in dieser Arbeit beschriebene Version befindet sich auf https://github.com/fabifiess/sweets_app (im Terminal starten mit `node app.js`).

Funktionsprinzip

Ziel ist eine Website, die statische Ressourcen dynamisch zusammenstellt und grafisch aufbereitet. Bilder und Videos (=Visuals) werden in einem gemeinsamen Dateiverzeichnis `/img` ungeordnet archiviert. Textdaten befinden sich in einer MySQL-Datenbank.

4.3.1 Präsentationsbereich

Abb. 11 (linker Teil) zeigt einen Screenshot des Präsentationsbereichs.
Der Nutzer kann in einer Dropdownliste eine Süßigkeit wählen, z. B. „Chocolate“. Die Süßigkeiten stehen hier stellvertretend für die Projekte aus der Projektedatenbank, die später in die Applikation geladen werden, sobald sie stabil läuft.

Der Inhalt des Präsentationsbereichs ergibt sich aus dem Beziehungsgeflecht der Daten in der Datenbank, die zu der jeweiligen Süßigkeit gehören: Jede Süßigkeit ist beispielsweise mit unterschiedlich vielen Bildern und/oder Videos verlinkt. Diese werden dementsprechend vom Server geladen und zusammen mit Textinformationen aus der Datenbank dynamisch zu einer Seite zusammengestellt. Dabei kennt das Programm die Daten nur als *Varia-*

1 VCS: Version Control System

blen. Anhand der Beziehungen der Variablen untereinander wird bei gleichbleibender Programmlogik der statische Content dynamisch arrangiert.
Das Resultat eines solchen generativen Systems[1] ist variabel, da es an den Programmcode gekoppelt ist.
Das Programm sollte mit allen Süßigkeiten (oder später auch mit Projekten) auf die gleiche Art umgehen können: Unabhängig davon, wie viele Visuals (Bilder oder Videos) pro Süßigkeit vorhanden sind und unabhängig davon, wie sie heißen. Außerdem sollte das Programm auch mit Daten umgehen können, die *erst nach* der Erstellung der Software hinzukommen.
Somit wird ein Programm benötigt, das alle Aufgaben logisch mit Variablen löst. Das Programm kennt den semantischen Zusammenhang der Daten nicht und muss zusammengehörige Daten quasi *blind* zusammenstellen.

Zu Beginn lädt das Programm die Namen der Süßigkeiten aus einer JSON-Datei. Nachdem ein Nutzer eine Süßigkeit in der Dropdown-Liste gewählt hat, sucht das Programm nach dem Datensatz, der mit diesem Süßigkeitennamen in Zusammenhang steht. Mit der Referenz auf diesen Datensatz werden alle Texte und Visuals gefunden, die zur jeweiligen Süßigkeit gehören.
Ähnlich läuft auch der Zugriff auf die Daten im Wartungsbereich ab. Auch dort werden Änderungen (`insert`, `update`, `delete`) nicht *hardgecoded*, sondern mit Hilfe von Variablen umgesetzt.
Auf der grafischen Oberfläche verwendet der Nutzer statische Daten. Indirekt arbeitet das Programm jedoch mit Variablen. Die inneren Vorgänge bleiben den Nutzern verborgen. Sie nehmen nur das Resultat des zugrunde liegenden generativen Systems wahr: Eine grafische Aufbereitung von statischen Ressourcen.

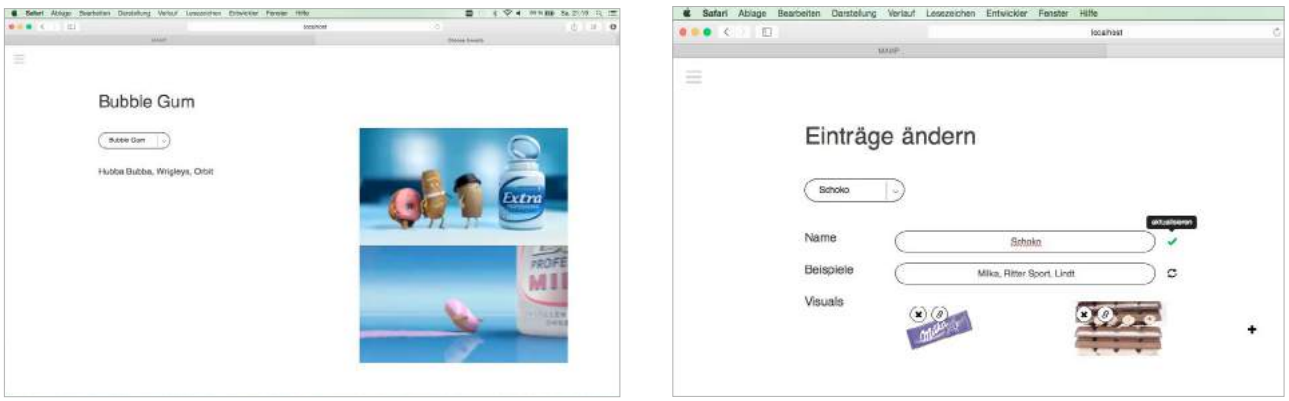


Abb. 11: Links: Präsentationsbereich, rechts: Update-Seite im Wartungsbereich

4.3.1.1. Ladeprozess von Informationsdaten

Abb. 12 (nächste Seite) veranschaulicht die elementaren Schritte bei der dynamischen Befüllung der Webseite mit statischem Content.

1 Generatives System: Ein unsichtbares, regelbasiertes System, das sein wahrnehmbares Feedback mit Hilfe eines vorher definierten Algorithmus, dem Regelwerk, automatisch generiert. Das Regelwerk implementiert die Logik der Applikation. Mit diesen Regeln werden statische Ressourcen dynamisch aufbereitet. Die Ressourcen können beispielsweise primitive Formen oder vorgefertigte Bilder auf dem Speicher sein.

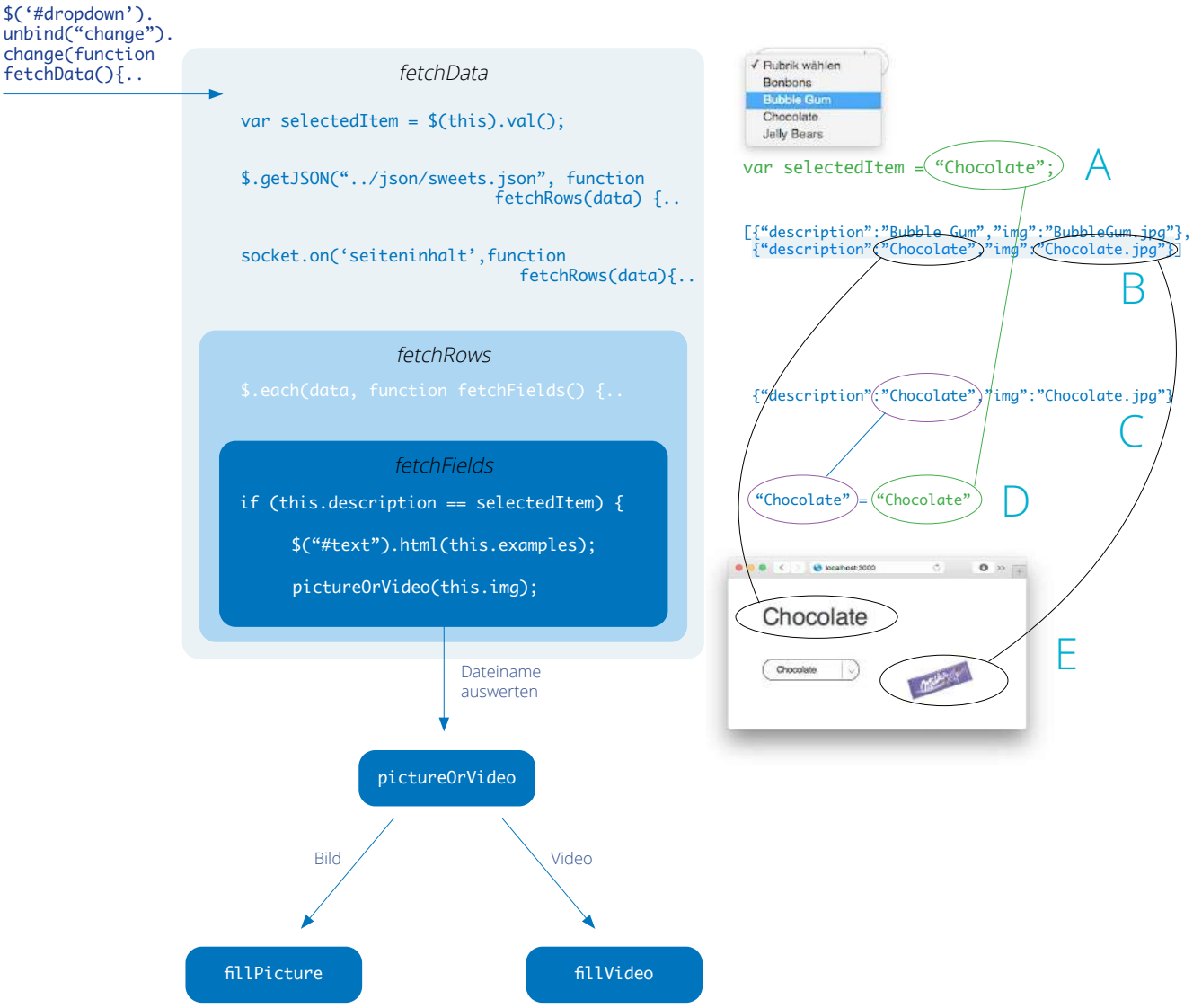


Abb. 12: Demonstration eines dynamischen Seitenaufbaus mit statischen Ressourcen,
vgl. https://github.com/fabifiess/sweets_app/blob/master/public/html/maincontent_search.html (Z. 35f).

Nachdem die Dropdownliste aus den alphabetisch sortierten Daten einer JSON-Datei zusammengestellt wurde, kann der Nutzer ein Dropdown-Item per Klick anwählen. Daraufhin baut sich die Webseite dynamisch mit statischen Daten auf.

Grob passiert dabei Folgendes:

Mit `$("dropdown").unbind(change(function(){..});` registriert das Programm, dass in der Dropdown-Liste (`<select>`-Liste im HTML-Code) eine Süßigkeit ausgewählt wurde.
Mit `unbind` werden frühere Eventhandlerfunktionen entfernt. Andernfalls werden nachfolgende Schritte ungewollt mehrmals ausgeführt, je öfter die Dropdown-Liste verändert wird.

Zunächst wird die Funktion `fetchData` gerufen.
Mit `var selectedItem = $(this).val();` wird der Wert der gewählten Dropdownoption in `selectedItem` gesichert (vgl. Abb. 11, Abschnitt A).

Dann erhält die clientseitige Applikation die Daten für die Seite (Abschnitt *B*): Entweder über die JQuery-Methode `$.getJSON` aus einer JSON-Datei, die auf dem Webserver liegt, oder über eine Websocket-Verbindung. Die Daten werden in beiden Fällen über eine JavaScript-Objektstruktur bereitgestellt (vgl. Listing 15 auf Seite 95). Diese enthält die Textinformationen, bzw. die Dateinamen der Visuals. Alle Daten einer Süßigkeit befinden sich in je einem verschachtelten Objekt. Diese Süßigkeitkeitenobjekte befinden sich mit anderen Süßigkeitenobjekten in einem gemeinsamen Array.

Dann wird mit der Funktion `fetchRows` jedes Datenobjekt aus dem Objekte-Array (aka alle Zeilen/Rows aus einer Tabelle einer relationalen MySQL-Datenbank) geladen (vgl. Abschnitt *C*). Jedes Objekt besteht aus weiteren Unterobjekten (aka Tabellenfelder). Das Unterobjekt `description` enthält den Namen der Süßigkeit, z. B. `Chocolate`.

In der Funktion `fetchFields` wird für jedes Hauptobjekt (vgl. Abschnitt *C*), das mit `fetchRows` vom Objekt-Array extrahiert wurde, Folgendes geprüft: Stimmt der Wert im Unterobjekt `description` mit dem Wert des gewählten Dropdown-Items (gespeichert in `selectedItem`) überein?
`if(this.description == selectedItem) {..};` (vgl. Abschnitt *D*).
Nun ist das gesuchte Datenobjekt – in diesem Fall die gewählte Süßigkeit – referenziert und die Seite kann mit Inhalten befüllt werden (vgl. Abschnitt *E*).

4.3.1.2. Seiteninhalte füllen

Die Dateinamen der Visuals liegen in einem Array im Unterobjekt `img` vor, das in einer for-Schleife iteriert wird. Für jedes einzelne Array-Element wird in der Methode `pictureOrVideo` anhand des Dateinamens geprüft, ob es sich um ein *Bild* oder ein *Video* handelt (vgl. https://github.com/fabifiess/sweets_app/blob/master/public/js/main.js, Z. 133f).
Handelt es sich um ein Bild, erstellt die Funktion `fillPicture` aus Variablen und Zeichenketten ein Stück dynamischen HTML-Code und speichert diesen in einem `<div>`-Tag im HTML-Code ab.
Ähnlich funktioniert die Methode `fillVideo` für Videodateien. Zum Abspielen der Videos wird die JavaScript-Bibliothek *video-js* verwendet. Mit *video-js* können u. a. der Videoplayer gestalterisch an das Corporate Design von ICT angepasst und Videos per Klick gestartet bzw. angehalten werden. Abhängig von der Länge des Arrays im Unterobjekt `img` können beliebig viele Bilder und / oder Videos geladen werden.

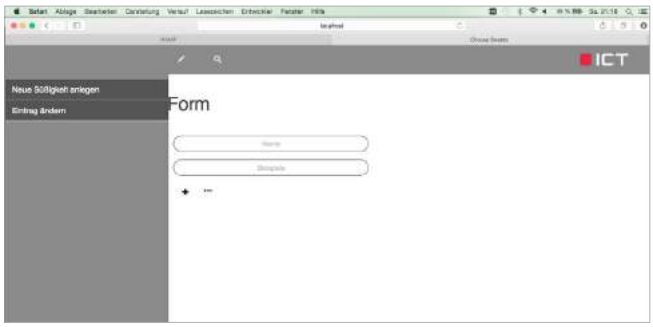


Abb. 13: Präsentations- und Wartungsbereich bzw. Untermenüs können in einblendbaren Seitenleisten ausgewählt werden

4.3.2. Wartungsbereich

Im Präsentationsbereich kann der Nutzer lediglich einer Datenbank *abfragen*, aber keine Datensätze in eine Datenbank *einfügen* oder *aktualisieren*. Der Wartungsbereich besteht aus zwei Teilbereichen (vgl. Abb. 13):

- **Add-Bereich:** Hier können neue Süßigkeiten (oder auch ICT-Projekte) angelegt werden (vgl. https://github.com/fabifiess/sweets_app/blob/master/public/html/maincontent_new.html).
- **Update-Bereich:** Hier können bestehende Datensätze modifiziert, Visuals ergänzt oder entfernt werden (vgl. https://github.com/fabifiess/sweets_app/blob/master/public/html/maincontent_new_update.html).

4.3.2.1. Upload

Über ein Uploadformular werden Textdaten und Bild-/ Videodateien mit einem POST-Request über das HTTP-Protokoll an den Node.js-Webserver übermittelt. Das HTTP-Protokoll wurde in Kap. 3.1 vorgestellt. Über das Node.js-Modul *express* wurde die Middleware *multer* eingebunden, die Daten entgegennimmt, ihnen einen eindeutigen Dateinamen (mit Datum und Uhrzeit) zuweist und in das Dateiverzeichnis `/public/img` abspeichert (vgl. Kap. 3.5.2: express).

4.3.2.2. Datenbankabfrage

Die Daten wurden in einer relationalen MySQL-Datenbank gespeichert. Werden für eine Süßigkeit mehrere Visuals angelegt, sollen diese Visuals später auch mit anderen Süßigkeiten verlinkt werden können. Die Daten müssen folglich in einer N:M-Tabellenstruktur organisiert werden. Many-to-many-Beziehungen wurden in Kap. 4.2.2.3 untersucht. Es entstanden drei Tabellen `sweets`, `visuals` und die schwache Zwischentabelle `sweets_visuals` (wird in Abb. 14 für nachfolgende Erklärungen nochmals gezeigt).

Abb. 14: Verknüpfung der Tabellen `sweets` und `visuals` über `sweets_visuals`

Im serverseitigen Skript (mit Node.js) wird für die Verbindung zu einer MySQL-Datenbank das Modul `mysql` eingesetzt. Bei einer Datenbankabfrage mit einem gewöhnlichen SQL-Statement werden die Resultate in eine JavaScript-Objektstruktur umgewandelt: Jeder Datensatz (*Zeile/Row*) wird als abgeschlossenes Objekt in ein Array geschrieben. Jedes Tabellenfeld dieser Zeile wird als einfaches *key/value*-Paar in jeweils ein Unterobjekt geschrieben. Das Modul `mysql` ist allerdings nicht in der Lage, zusammengehörige Daten in einem Array zu speichern, wie es beispielsweise mit der NoSQL-Datenbank MongoDB möglich ist.

Somit entstehen Redundanzen. Um diese zu vermeiden und um die zweite Normalform bei einer Übersetzung eines Datensatzes aus der Datenbank in eine JavaScript-Objektstruktur aufrecht zu erhalten, muss das resultierende JavaScript-Objet auch Arrays enthalten. Da bei der Übersetzung keine Konsistenzsicherung gegeben ist, habe ich ein solches Skript selbst erstellt. Listing 15 zeigt das Ergebnis.

```
[
{..},
{
  sweets_id: 2,
  "description: "Chocolate",
  examples: "Ritter Sport",
  visuals: [
    {
      visuals_id: 4,
      img: "Milka.jpg"
    },
    {
      visuals_id: 5,
      img: "Ritter.mp4"
    }
  ]
},
{..}
]
```

Listing 15: Ergebnis des Skripts von Kap. 4.3.2.2: Umwandlung einer MySQL-Abfrage in eine Objektstruktur mit einem Array und verschachtelten Objekten.

Es umfasst zwei Datenbankabfragen, die nacheinander ausgeführt werden. Die Ausführungsreihenfolge wird über eine Callbackstruktur (wie in Kap. 2.3.3.3 beschrieben) geregelt: Zuerst werden alle Dateinamen aus `visuals` abgefragt und in ein JavaScript-Objekt abgelegt. Anschließend, in der Callbackfunktion, erfolgt das Gleiche mit den Daten aus `sweets`.
Anschließend werden die Datenobjekte aus `sweets` der Reihe nach durchlaufen. Für jedes Datenobjekt aus `sweets` werden die Datenobjekte aus `visuals` komplett durchlaufen. Es handelt sich um zwei verschachtelte for-Schleifen. Passt ein Datenobjekt aus `visuals` zu `sweets`, wird es in ein Array `visuals` geschrieben und an das Datenobjekt mit den Einträgen aus `sweets` angehängt.
Dieses Skript wurde in eine Funktion gekapselt und mit Formalparametern abstrahiert, damit es für andere Tabellen mit der gleichen Beziehungsstruktur wiederverwendet werden kann.

4.3.2.3. Datenbankeintrag

Ein Eintrag über eine N:M-Tabellenstruktur wird ebenfalls über eine Callbackstruktur gelöst: Zunächst wird eine Funktion aufgerufen, die die beiden Haupttabellen `sweets` und `visuals` synchron befüllt. Jeweils im Anschluss der beiden Einfügeoperationen werden die IDs der neu eingefügten Datensätze abgefragt und in Variablen gespeichert. Im Anschluss wird eine Callbackfunktion ausgeführt, die die beiden IDs in die Zwischentabelle `sweets_visuals` einfügt.
Bei der Löschfunktion ist die Reihenfolge umgekehrt.
Werden für eine Süßigkeit *mehrere* Visuals auf einmal angelegt, kann der Einfügeoperation in `visuals` eine Schleife vorgeschaltet werden. Es ist auch möglich, dass sich die Einfügefunktion rekursiv selbst wieder aufruft.

4.3.2.3. Datenlöschung, Datenänderung

Weist ein Benutzer des grafischen clientseitigen Interfaces eine Lösch- oder Updateanweisung an, findet die Client – Server-Kommunikation in dieser Webanwendung über eine socket.io-Verbindung statt, wie sie in Kap. 3.5.3 experimentell untersucht wurde.
Sie erfolgt ebenfalls asynchron mit Hilfe von Callbackkonstrukten.
Bei der Anfrage eines Clients an den Webserver mit `Socket.emit` wird eine Eventhandlerfunktion bereitgelegt, die ausgeführt wird, sobald der Webserver antwortet. Abb. 11 (rechter Teil) auf Seite 91 zeigt, wie der Webbrowser einen grünen Haken darstellt, nachdem der Webserver meldet, dass ein Update-Vorgang erfolgreich war.

4.4. Zusammenfassung und Überleitung

In diesem Kapitel wurden einige Konzepte aus Kap. 2 (JavaScript-Konzepte für Event Media-Installationen) und Kap. 3. (Client – Server-Interaktion) praktisch angewandt, indem eine dynamische Webanwendung mit einem grafischen User Interface entwickelt wurde. Sie soll den Kunden von ICT die konzeptionellen Ansätze einer erlebnisreichen Produktpräsentation aus Kap. 1.2.2. (Acht mögliche Präsentationsformen) näher bringen. Die Applikation besteht aus einem clientseitigen Browserinterface und einem Webserver, die über das HTTP- und das WebSocket-Protokoll miteinander kommunizieren, sowie aus einer MySQL-Datenbank, auf die der Webserver Zugriff hat.
Die MySQL-Datebank ist in der Lage, komplexe Datenbankoperationen über komplexere relationale Tabellenbeziehungen durchzuführen.
Mit der Applikation kann der Benutzer auf intuitive Weise nach verschiedenen Kategorien suchen, Daten hinzufügen, verändern und löschen.

Das Tabellengeflecht (ERM) auf Seite 88 zeigt die hierfür benötigte Tabellen in einer relationalen MySQL-Datenbank und die Beziehungen zueinander.
Der Planungsaufwand für eine relationale Datenbank ist enorm und trotz präziser Dokumentation nicht komfortabel zu handhaben.
Im nächsten Kapitel werden alternative Datenbanksysteme nach Eignung für die Realisierung von komponentenreichen Event Media-Installationen untersucht.

Kap. 5: Datenbank

In diesem Kapitel wird eine Webanwendung, die aus einem clientseitigen und einem serverseitigen Skript besteht, durch eine geeignete Datenbank ergänzt.
Mit Hilfe eines *Gedächtnisses* können Webinterfaces dynamisch mit Inhalten gefüllt werden und die Nutzer Daten manipulieren.
In Kap. 4 (Entwicklung Projekteplattform) wurde eine relationale MySQL-Datenbank eingesetzt, was sich für Event Media-Installationen als nicht optimal herausstellte.

In wenigen Fällen genügt es, die Daten für eine interaktive Medieninstallation in einer Textdatei, wie JSON oder XML, abzulegen. Bei komplexeren Anforderungen empfiehlt Harvey allerdings den Einsatz eines Datenbankmanagementsystems.
In Bezug auf Medieninstallationen sind folgende seiner Punkte ausschlaggebend für die Verwendung eines Datenbanksystems zur Interaktion mit Anwendungsdaten:

- Komfortablere Abfragemöglichkeiten
- Beziehungen der Datensätze untereinander schaffen
- Skalierbarkeit
- Möglichkeit der gleichzeitigen Manipulation von Datensätzen durch mehrere Nutzer
- Fehlertoleranz
- Persistenz[1] [2]

In diesem Kapitel möchte ich verschiedene Datenbanksysteme nach Eignung für Medieninstallationen untersuchen.

Stand der Technik: Big Data

Im Zuge der Digitalisierung des Alltags in Verbindung mit „Internet der Dinge“ (vgl. Kap. 1.3.1) oder „Web 3.0“ (vgl. Kap. 3. (Client – Server-Interaktion)) fallen in den verschiedensten alltäglichen Bereichen gewaltige Datenmengen an, die mit Hilfe von unterschiedlichen Datenbanksystemen bewältigt werden können.
Mit immer spezielleren Anforderungen wurden zahlreiche spezialisierte Datenbanksysteme entwickelt, die unterschiedlichen Einsatzzwecken gerecht werden. Allgemein wird zwischen relationalen Datenbanken und NoSQL[3]-Datenbanken unterschieden.

1 Persistenz: Dauerhaft und über einen Programmabsturz hinweg verfügbar
2 Es ist nicht empfehlenswert, Nutzerdaten in einer JSON-Datei auf dem Webserver eines Onlinehosting-Dienstes zu speichern: Für die Umfrage in Verbindung mit dieser Arbeit wurden die Daten eines Formulars in einer JSON-Datei auf dem Server von Heroku abgelegt. Da sie regelmäßig auf den Ausgangszustand zurückgesetzt wurde, gingen vorhandene Daten verloren.
3 NoSQL: Not Only SQL

5.1. Relationale Datenbanken

Relationale Datenbanken haben sich in der Vergangenheit bewährt und dominieren noch heute: „Die populärste Open-Source-Datenbank der Welt“ [mysql] ist MySQL. MySQL wird häufig bei Content Management Systemen (CMS) für Blogs, wie z. B. Wordpress oder Joomla eingesetzt. Ein etwas spezielleres DBMS, das vowiegend für Finanzaktionen eingesetzt wird, ist Oracle SQL.
Im Folgenden möchte ich auf einige Eigenschaften von relationalen Datenbanken eingehen:

- Das relationale Datenmodell orientiert sich an der **mathematischen Mengenlehre**. Relationen sind Beziehungen zwischen Wertemengen[1]. Daten liegen ohne festgelegte Ordnung vor, weder horizontal noch vertikal.
- Daten werden in **zweidimensionalen Tabellen** strukturiert: Pro Tabellenzeile wird ein Datensatz gespeichert. Jedes Tabellenfeld kann über den jeweiligen Spaltenname und die Zeilen-ID referenziert werden.
- Datensätze aus verschiedenen Tabellen können über **Join-Abfragen** beliebig miteinander verbunden werden.
- Das Ergebnis einer Datenbankabfrage (auch über mehrere Tabellen) wird ebenfalls in einer Tabelle dargestellt.
- Alle relationalen Datenbanken (somit auch MySQL und Oracle SQL) verwenden **SQL** als normierte Zugriffssprache[2].

5.2. Vergleich zwischen SQL- und NoSQL-Datenbanken

Der Schwerpunkt beim Einsatz von NoSQL-Datenbanken liegt bei der Performance und ihrer Skalierbarkeit. Unterschiedliche NoSQL-Datenbanken lösen unterschiedliche Probleme des Big Data, frei nach dem Motto „Choose the right tool for your job“. Es gibt kein NoSQL-Datenbanksystem, das für alle Zwecke geeignet ist.
NoSQL-Datenbanken werden in vier Kategorien eingeteilt: Schlüsselwertdatenbanken (key/values-Stores), Spaltenorientierte Datenbanken (Column Stores), Dokumentendatenbanken und Graphendatenbanken.
Unabhängig von den Besonderheiten der verschiedenen Arten von NoSQL-Datenbanksystemen möchte ich mit folgender Gegenüberstellung auf die generellen positiven und negativen Eigenschaften von relationalen Datenbanken und NoSQL-Datenbanken eingehen, die für den Einsatz in Event Media-Installationen bedeutend sein können:

1 Wertemengen: Als Fachbegriff wird auch Domäne verwendet
2 SQL (Structured Query Language) ist eine Sprache für alle Aufgaben, die im Zusammenhang mit einer Datenbank stehen. SQL arbeitet mengenorientiert (nicht objektorientiert, wie NoSQL-Datenbanken), hat eine dreiwertige Logik (`true`, `false`, `null`) und kann über APIs in unterschiedlichste Programme einbezogen werden.

Relationale Datenbanken	NoSQL-Datenbanken
Die **referentielle Integrität** (RI) ist sichergestellt: zur Vermeidung von Einfüge-, Update-und Deleteanomalien werden Datensätze auf mehrere Tabellen verteilt.	I. d. R. **keine Integritätssicherung**
Performance ist für viele Anwendungen nicht ausreichend (Hauptgrund für NoSQL-Datenbanken).	Je nach Datenmodell und Einsatzzweck **sehr gute Performance**.
Modellierung von Beziehungen zwischen Datensätzen generell möglich, jedoch ressourcenintensiv.	**Modellierung von Beziehungen** zwischen Datensätzen ist nur bei Graphdatenbanken besonders performant und bei anderen Systemen z.T. überhaupt nicht gegeben.
Horizontale Skalierung (Verteilung von Datensätzen auf verschiedenen Datenbankservern) möglich, aber wenig performant.	Datenmodelle sind **auf horizontale Skalierung spezialisiert**.
Daten unterliegen einem **strikt strukturierten Schema**, das bereits im Voraus festgelegt werden muss (strenge Typisierung von Datentypen, festgelegte Anzahl von verfügbaren Spalten).	Daten können i. d. R. ohne aufwändige Planungsphase **schemalos** in die Datenbank eingefügt werden.
Datensätze liegen in **zweidimensionalen Tabellen** vor. Für Mehrdimensionalität müssen Tabellen über PK/FK-Beziehungen miteinander verknüpft werden.	Je nach Datenmodell ist eine **mehrdimensionale, objektorientierte Datenhaltung** möglich, z. B. verschachtelte Objekte in Dokumentendatenbanken.

Diese Gegenüberstellung zeigt, dass NoSQL-Datenbanken generell interessante Eigenschaften für die Realisierung von Event Media-Installationen besitzen. Die gravierendsten Schwächen zeigen NoSQL im Vergleich zu relationalen Datenbanken bei der Konsistenzsicherung. NoSQL-Datenbanken sind „eventually consistent". Das bedeutet, dass eine Konsistenz „zwar prinzipiell angestrebt wird, ihre Unverzüglichkeit aber nicht garantiert wird." [Pröll].
Da für mediale Anwendungen eine leichte Verzögerung in Kauf genommen werden kann, ist diese Schwäche von NoSQL-Datenbanken hier akzeptabel. Nicht so bei Banken-Datenbanken. Diese Schwäche von NoSQL-Datenbanken ist demnach *kein* Auswahlkriterium für Mediendatenbanken und wird deshalb in der obigen Tabelle nicht aufgeführt.
Die horizontale Skalierbarkeit, Objektorientierung und Schemalosigkeit sind bei der Entwicklung von Event Media-Installationen umso bedeutender.
Im Folgenden werden einige Anforderungen an ein Datenbanksystem für Eventinstallationen gestellt:

- Die Applikation soll für zahlreiche Nutzer zur selben Zeit nutzbar sein (**Concurrency**[1]).
- Die Applikation wird mit **JavaScript** erstellt.
- Die Datenbank muss von **mehr als einem Entwickler** mit **SQL-Hintergrund** instand gehalten werden können.

Nachfolgend möchte ich den vier verschiedenen Arten von NoSQL-Datenbanken auf den Grund gehen.

1 Concurrency: Gleichzeitiges Zugreifen und Modifizieren durch mehrere Nutzer

5.3. Überblick über verschiedene NoSQL-Datenbanken

Schlüsselwertdatenbanken

Beispiele: Einkaufswagen von Amazon (Amazon Dynamo), Linkedin (Voldemort), Posteingangssuche von Facebook (Cassandra).
Datenmodell: Jeder Datensatz besteht aus einem Schlüssel, der auf einen Wert zeigt (key/value). Die Werte können beispielsweise Zeichenketten oder Listen sein. Die Daten sind simpel strukturiert.
Vorteile: Hohe Geschwindigkeit, Hochverfügbarkeit und Skalierbarkeit, sehr geringe Reaktionszeit.
Nachteile: Komplexe Abfragen nicht möglich. Nicht geeignet zur Abbildung von komplexeren Beziehungen.

Spaltenorientierte Datenbanken

Beispiele: Google Maps, Google Documents: Big Table, analytische Informationssysteme, Amazon SimpleDB.
Datenmodell: Daten werden spaltenweise und nicht zeilenweise gruppiert. Jeder Eintrag besteht aus dem Namen der Spalte, den Daten und einem Zeitstempel. Zusammenhängende Spalten bilden eine Column-Family.
Vorteile: Hochskalierbar, sie eignen sich gut für große Datenvolumen aufgrund der Verteilung der spaltenorientierten Struktur auf mehrere Server, womit die Last auf einen einzelnen Server gering gehalten werden kann.
Nachteile: Großer Aufwand bei der Installation sowie bei Schreibprozessen, da sie mehrere Spalten umfassen.

Dokumentendatenbanken

Beispiele: MongoDB, CouchDB, Anwendungen: Synchronisation von mobilen Endgeräten. Komfortable Modellierung eines Beziehungsgeflechts über verschachtelte Objekte, wofür bei relationalen Datenbanken aufwändige PK/FK-Beziehungen nötig sind.
Datenmodell: Schemafrei, Daten werden, wie Schlüsselwertdatenbanken, in Form von Objekten mit key/value-Paaren gespeichert, die beliebig verschachtelt werden und auch Arrays beinhalten können.
Vorteile: Real existierende Objekte können ohne Abstrahierung in einen leichtgewichtigen Document Store geschrieben werden. Der Umgang mit Dokumentendatenbanken ist an relationale Datenbanken angelehnt.
Nachteile: Aufgrund der Schemafreiheit von Dokumentendatenbanken sind einige komfortable Funktionen von relationalen Datenbanksystemen, wie Trigger und selbstdefinierte Constraints, nicht möglich.

Graphdatenbanken

Beispiele: Neo4J, Anwendungen: „Alles, was sich in einem Graphen abbilden lässt“ [Ha], beispielsweise Empfehlungen bei z. B. Amazon, Beziehungen in sozialen Netzwerken, Routenplanung.
Datenmodell: Die Graphendatenbanken bestehen, wie echte Graphen, aus Knoten und Kanten. Ein Knoten repräsentiert eine Tabellenzeile einer relationalen Datenbank. Kanten repräsentieren die Beziehungen zwischen diesen Knoten.
Vorteile: Effiziente Speicherung und Verarbeitung von vernetzten Informationen. Datensätze können schneller verknüpft werden als mit relationalen Datenbanksystemen.
Nachteile: Nicht unendlich skalierbar. Die Partitionstoleranz ist gering. Es gibt viele Graphenmodelle, aber keine einheitliche Abfragesprache.

Für den medialen Einsatz zeichnen sich besonders Schlüsselwertdatenbanken und Dokumentendatenbanken wegen ihrer Einfachkeit bzw. wegen ihrer Nähe zur objektorientierten Programmierung, z. B. mit JavaScript, aus. Im Folgenden wird anhand der Anforderungen aus Kap. 4.2 die Dokumentendatenbank MongoDB nach Eignung für Event Media-Installationen untersucht.

5.4. Evaluierung von MongoDB für den Einsatz in Eventinstallationen

MongoDB wird allen Anforderungen aus Kap. 4.2 gerecht:
Concurreny wird durch die asynchrone, singlethreadbasierte Arbeitsweise von JavaScript ermöglicht. Details zu diesen Eigenschaften wurden ab Kap. 2.3.3 behandelt.
Im Folgenden möchte ich näher darauf eingehen, wie MongoDB die übrigen beiden Anforderungen erfüllt:

5.4.1. Verwandtschaft mit JavaScript

Zugriff auf Daten

MongoDB legt den Datenbestand in einer JSON-ähnlichen Form[1] ab. Greift ein Entwickler auf einen Datensatz einer MongoDB-Datenbank zu, arbeitet er direkt mit JavaScript-Objekten. Wie eine JSON-Datei organisiert MongoDB einzelne, ggf. verschachtelte Datensätze in einem Array. Die Datensätze einer MongoDB-Datenbank liegen beliebig verschachtelt, also mehrdimensional, vor. Daten in relationalen Datenbanken liegen hingegen in zweidimensionalen Tabellen vor. Um Datensätze einer relationalen Datenbank mehrdimensional zu organisieren, müssen mehrere Tabel-

1 Mongo DB speichert Daten im Format BSON ab, einer binären Form von JSON.

len über PK/FK-Beziehungen miteinander verknüpft und mit Joins abgefragt werden. PK/FK-Beziehungen bzw. Join-Abfragen wurden in 4.2.2.2. (1:N-Beziehungen) näher betrachtet. Die Erstellung der Beziehung von Tabellen zueinander muss bei relationalen Datenbanken *vor* der Befüllung detailliert geplant werden.
Join-Abfragen entfallen bei der Nutzung einer mehrdimensionalen MongoDB-Datenbank.

Einfügen von Daten

In relationalen Datenbanken müssen Daten oft auf mehrere Tabellen verlagert werden[1], was zahlreiche einzelne Einfügeoperationen erfordert. In MongoDB hingegen können ganze JavaScript-Objekte, die beliebig verschachtelt vorliegen können, in *einer einzigen* Einfügeoperation in die Datenbank geschrieben werden. Node.js hat mit dem Modul *mongodb* mit Hilfe von JavaScript nahezu nahtlos Zugriff auf eine MongoDB-Datenbank.

5.4.2. Nähe zu MySQL

Vorhandene SQL-Kenntnisse rechtfertigen den Einsatz von relationalen Datenbanksystemen. MongoDB überzeugt durch seine Orientierung an SQL-Befehlen. mongodb.org (Stand: 15. Mai 2015) beweist dies mit einer „SQL to MongoDB Mapping Chart". Diese Liste deckt die wichtigsten SQL-Befehle ab. Join-Abfragen unterstützt MongoDB nicht. Mit der Möglichkeit, Objekte ineinander zu verschachteln, erübrigt sich dieser Befehl weitgehend.

Wie MongoDB dennoch Beziehungen von Tabellen untereinander herstellen kann, wird nachfolgend untersucht:

- Abb. 1 veranschaulicht die Herstellung einer 1:N-Beziehung in einer relationalen Datenbank und einer MongoDB-Datenbank. Abb. 1 zeigt die Gegenüberstellung eines MongoDB-Dokuments und eines über mehrere Tabellen verteilten Datensatzes in relationalen Datenbanken.

```
[
{..},
{
  sweets_id: 2,
  "description: "Chocolate",
  examples: "Ritter Sport",
  visuals: [
    {
      visuals_id: 4,
      img: "Milka.jpg"
    },
    {
      visuals_id: 5,
      img: "Ritter.mp4"
    }
  ]
},
{..}
]
```

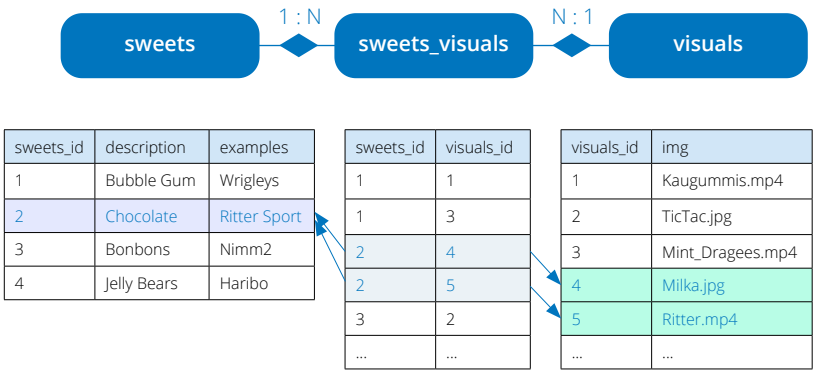


Abb. 1: Herstellung einer 1:N-Beziehung zwischen Daten in MongoDB (links) und MySQL (rechts)

1 Die Verteilung eines Datensatzes auf mehrere Tabellen einer rel. Datenbank dient zur Vermeidung v. `Insert`-, `Update`- und `Delete`-Anomalien (vgl. 2. Normalform, Kap. 4).

- Alle Daten, die in einer relationalen Datenbank über mehrere, streng strukturierte Tabellen verteilt werden, befinden sich unstrukturiert in *einer einzigen* Collection, wodurch viel Planungszeit bei der Modellierung einer Datenbank eingespart wird.
- Um N:M-Beziehungen herzustellen, können einzelne Dokumente innerhalb einer Collection über ihre IDs miteinander verknüpft werden. MongoDB liefert hierfür u. a. die Funktion `DB_Ref(..);`. Allerdings ist die referentielle Integrität aus relationalen Datenbanken (vgl. Kap. 4) in MongoDB nicht gegeben. Demnach können in MongoDB-Collections Dokumente gelöscht werden, auch wenn ein anderes Objekt noch eine Referenz auf sie enthält.
- Mit *MapReduce* bietet MongoDB ein Konzept zur Vermeidung von Redundanzen in unstrukturierten Collections an, das der zweiten Normalform in relationalen Datenbanken ähnlich ist.

5.5. Zusammenfassung und Überleitung

In diesem Kapitel wurden relationale Datenbanken und NoSQL-Datenbanken in Hinblick auf die Realisierung von Event Media-Installationen untersucht.
Es wurde ein Überblick über die möglichen Arten von NoSQL-Datenbanken gegeben. Danach wurde die Auswahl der in Frage kommenden Datenbanksysteme eingegrenzt. Schließlich wurde die Dokumentendatenbank *MongoDB* erfolgreich nach Eignung untersucht und deren Funktionsweise mit relationalen Datenbanken verglichen.
Mit MongoDB wurde ein objektorientiertes Datenbanksystem gefunden, das Nähe zu relationalen Datenbanken aufweist und mit JavaScript konfiguriert werden kann.

Bisher ist es nun möglich, das client- und serverseitige Skript sowie die Datenbank mit Hilfe von JavaScript zu programmieren.
In Kap. 6 wird die physische Komponente einer komponentenreichen Event Media-Installation näher untersucht. Sie soll eine klassische Webanwendung und die reale Umwelt miteinander verbinden.

Kap. 6: Interaktion mit der physischen Umwelt

Stand der Technik: Internet der Dinge

In Kap. 1.2.4. (Benötigte Komponenten) wurde ein Standardaufbau für eine komponentenreiche Medieninstallation skizziert. Ein Meinungsbild ergab, dass die Anreicherung klassischer Messepräsentationen durch interaktive, physische Gegenstände von den Menschen unserer Umgebung generell angenommen wird. Kap. 1.3.1 gibt eine Einführung in das Internet der Dinge.
Die zentrale Schnittstelle zwischen einem virtuellen Computersystem und der physischen Umwelt sind häufig Mikrocontrollerboards, mit denen Sensoren und Aktoren[1] ausgelesen bzw. gesteuert werden können.
Mit dem Internet der Dinge sind zahlreiche Mikrocontrollerboards entstanden, die den Einstieg in die Digitalisierung der Umwelt erleichtern sollen.

In diesem Kapitel werde ich eine Auswahl an Mikrocontrollerboards in Hinblick auf die Realisierung einer Medieninstallation für den Messeeinsatz nach Eignung untersuchen.
Im Anschluss werde ich die Kommunikation zwischen einem Mikrocontrollerboard und einem Computer analysieren. Dabei behalte ich komfortable Entwicklungsmöglichkeiten, insbesondere JavaScript, im Blickfeld.

Auswahlwahlkriterien für ein Mikrocontrollerboard

Ein Mikrocontrollerboard zur Realisierung von komponentenreichen Medieninstallationen sollte …

- **gut dokumentiert** sein und eine aktive Community haben
- **flexibel** einsetzbar und **erweiterbar** sein
- **preiswert** sein
- **einfach konfigurierbar** sein, möglichst mit einer bekannten Programmiersprache (z. B. mit JavaScript)

1 Aktoren wandeln elektrische Signale in mechanische Energie um.
Der geringe Spannungspegel eines Mikrocontrollerboards beträgt häufig 5V und kann sowohl zum Betrieb von Elektronikkomponenten, wie z. B. LEDs, als auch als Steuerstrom von strom- und spannungsintensiveren Geräten, wie z. B. Motoren über z. B. Transistoren und Relais, verwendet werden.

6.1. Auswahl eines Mikrocontrollerboards

Bei den eben genannten Kriterien kamen drei Mikrocontrollerboards in Frage: Ein *Arduino Board*, ein *WunderBar*-System und ein *Tessel Board*.

6.1.1. Arduino Board

> *Arduino is a tool for making computers that can sense and control more of the physical world than your desktop computer* [arduino.cc]

Arduino Boards dominieren den Markt der Mikrocontrollerboards.
Die Community ist riesig und sehr aktiv. Dementsprechend gut sind sie dokumentiert. Es existieren verschiedene Ausführungen von Arduino Boards, die sich in Größe und Leistung unterscheiden. Im Allgemeinen sind sie kostengünstig – besonders die technisch identischen Klone aus Fernost.
Der Entwickler erhält mit einem Arduino Board im Wesentlichen einen Satz an analogen Eingängen und digitalen Ein- und Ausgängen. Die Eingangs-Pins können mit Sensoren und die Ausgangs-Pins mit Aktoren verbunden werden. Zur Konfiguration eines Arduino Boards bietet arduino.cc eine eigene Software an, die eine eigene Skriptsprache mit vereinfachter C-ähnlicher Syntax liefert. Mit ihr können die Ein- und Ausgänge eines Arduino Boards konfiguriert und mit Logik versehen werden.
Mittlerweile sind weitere Hilfsprotokolle entstanden, vor allem Firmata. Firmata ermöglicht, mit beliebigen Programmiersprachen, z. B. mit JavaScript, direkt auf Arduino Boards auf komfortable Weise zugreifen zu können.

6.1.2. WunderBar-System

WunderBar ist ein Komplettpaket für das Internet der Dinge, bestehend aus einem „WLAN-Mastermodul sowie verschiedenen Sensoren. Sie kommunizieren über Bluetooth und messen Geräuschpegel, [Feuchtigkeit], Bewegung, Temperatur und Helligkeit oder senden Infrarotsignale wie eine Fernbedienung. Die einzelnen Sensoren [...] senden ihre Daten über ein Hauptmodul zu den Cloud-Servern des Herstellers Relayr, wo sie über Apps oder andere Geräte abgerufen werden können" [Neuhaus] (vgl. Abb. 1). Das Starterpaket besteht aus sechs Sensoren, die über Bluetooth Low Energy (BLE[1]) mit einem Master-Modul kommunizieren. Das Master-Modul ist über WLAN mit dem WWW verbunden und sendet die Sensordaten an die Cloud des Herstellers Relayr. Von dort kann auf die Sensordaten über verschiedene APIs zugegriffen werden: Aktuell sind Bibliotheken für Node.js, clientseitiges JavaScript mit HTML und CSS, Android SDK, iOs & OSX, Python[2] und C#/.NET verfügbar (vgl. Relayr).

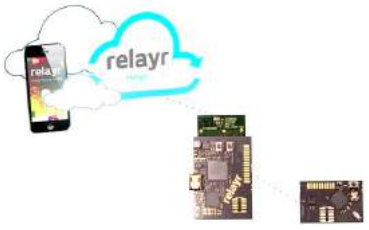


Abb. 1: V. r. n. l.: Sensormodul → BLE → Master-Modul → WLAN → Relayr-Cloud → App auf mobilen Endgeräten

1 Bluetooth Low Energy (BLE) ist die energiesparende Version 4.0 von Bluetooth
2 Die von Relayr angebotene Python-Bibliothek ist aktuell die einzige Bibliothek, die ohne den Umweg über die Relayr-Cloud direkt auf die Sensormodule des WunderBar-Systems zugreifen kann.

Vergleich mit einem Arduino Board

Das WunderBar-System richtet sich primär an App-Entwickler, die physische Objekte in Webanwendungen integrieren möchten, ohne sich dabei mit der Elektronik auseinander setzen zu müssen. Ein Arduino Board hingegen wird als pure Sammlung von Ein- und Ausgängen ausgeliefert – ohne Sensoren. Diese müssen mit Hilfe von elektronischen Schaltungen mit den Eingängen eines Arduino Boards verbunden werden. Der Umgang mit Sensoren über ein Arduino Board setzt im Gegensatz zum WunderBar-System, bei dem die Sensoren bereits fest verbaut sind, elektronische Grundkenntnisse voraus.

Lückenhafte Dokumentation

Da das WunderBar-System erst seit 2014 auf dem Markt ist, ist die Dokumentation entsprechend lückenhaft, doch sie verbessert sich zügig[1].

Flexibilität und Erweiterbarkeit mit Grenzen

Das WunderBar-System punktet durch seine Flexibilität und Erweiterbarkeit. Die Auswahl an APIs für den Zugriff auf Sensordaten ist groß. Zudem bietet das WunderBar-System neben den fünf Sensormodulen ein ebenfalls bluetooth-fähiges *Bridge-Modul* an, das mit weiteren Sensoren oder Ausgängen eines anderen elektonischen Geräts verbunden werden kann. Hierbei empfängt das Bridge-Modul über elektronische Pins elektrische Impulse von beispielsweise einem Zusatzsensor, einem Arduino Board oder einem Raspberry Pi-Computer (vgl. Abb. 1, ganz rechts).
Das WunderBar-System ist mit derzeit 179 Euro bei Conrad im Vergleich mit Arduino-Klons sehr teuer und für größere Projekte nicht rentabel, zumal die Sensorplatinen aktuell nicht einzeln nachgekauft werden können.

Begrenzte Erweiterbarkeit aufgrund von BLE

Eine Erweiterung des Mastermoduls durch weitere Bluetooth-fähige Sensorplatinen im Großen Stil ist in technischer Hinsicht[2] nicht ohne erheblichen Mehraufwand[3] möglich. Praxistests in Verbindung mit dem Studentenprojekt *Interaktive Kletterwand* belegen, dass ein Mastermodul unter der Verwendung von BLE nur begrenzt viele Sensoren bündeln kann.

Ferner ist anzumerken, dass die drahtlose Datenübertragung bzw. Stromversorgung bei Events Störfaktoren darstellen und vermieden werden sollten – somit auch BLE. Durch die Verbindung bestimmter elektrischer Kontakte kann der Einsatz von BLE und Batterien zwar umgangen werden, allerdings

1 Mit dem renommierten *code_n Award* anlässlich der CeBit im März 2015 sorgt das WunderBar-System zunehmend für Aufsehen. Durch die Unterstützung von Maker Faires, der Veranstaltung von Wettbewerben und die Präsenz auf Messen erhofft sich das Startup Relayr eine aktive Community.

2 Je nach Spezifikation können zwischen drei und acht BLE-fähige Geräte in einem Netzwerk zusammengefasst werden (vgl. Texas Instruments).

3 Indem mehrere Mastermodule in einem weiteren Master-Slave-Verbund gebündelt werden, in diesem sie jeweils die Rolle von Slaves einnehmen, kann die mögliche Anzahl der über BLE verbundenen Geräte potenziert werden.

erübrigen sich dadurch diese beiden Besonderheiten des WunderBar-Systems. Außerdem funktioniert die Verbindung des Mastermoduls mit einem Netzwerk mangels benötigter Anschlüsse der Platine nicht kabelgebunden, sondern ausschließlich über WLAN.

Nur Eingänge, aber keine Ausgänge

Ein weiterer Nachteil des WunderBar-Systems ist, dass die sechs Sensorplatinen nur als Sender verwendet werden können und nicht mit Aktoren verbunden werden können.
Was beim WunderBar-System schon vorgefertigt und frei nutzbar ist, ist mit entsprechendem Mehraufwand auch mit erheblich günstigeren Arduino Boards möglich.
Ein Kompromiss zwischen einem WunderBar-System (alles vorgefertigt, keine native Möglchkeit, Aktoren zu verbinden) und einem Arduino Board (alles möglich, alles selber machen) ist das Tessel Board.

6.1.3. Tessel Board

> *Tessel ist ein Internet-taugliches Entwicklerboard mit integriertem JavaScript-Interpreter.* [Make]

Die Hardware ist für die Programmierung mit JavaScript über die Node.js-Plattform ausgelegt. Das Herzstück der Prozessor / Koprozessor-Architektur ist Googles JavScript-Engine V8, die es ermöglicht, dass Node.js-Server nativ auf dem Tessel Board ohne zusätzlichen Computer funktionieren. Über WLAN oder auch kabelgebunden über Ethernet lassen sich Tessel Boards mit einem Heimnetzwerk verbinden, über das Sensoren am Tessel Board ausgelesen bzw. Aktoren gesteuert werden können.

Vergleich Tessel und WunderBar

Wie auch das WunderBar-System wird das Tessel Board als Entwicklerboard für das Internet der Dinge beworben. Anders als die meisten Arduino-Ausführungen können sie über WLAN direkt mit dem Internet verbunden werden. Beide Systeme haben gemeinsam, dass sie in vertrauten Webentwicklungsumgebungen konfiguriert werden können, u. a. mit JavaScript, ohne dass zur Übersetzung des Quellcodes in Machinencode für den Mikroconroller, wie es bei Arduino Boards der Fall ist, ein Kompilierprogramm benötigt wird.[1]
Das Tessel Board bietet die Besonderheit, dass es über das Netzwerk von einem gewöhnlichen Computer aus (um-)programmiert werden kann

1 arduino.cc bietet eine Software an, die die vom Entwickler geschriebenen Programme in Maschinencode verwandelt, bevor dieser an das Arduino Board gesendet wird. Dieser ist während seiner Laufzeit bis zu einer Neuprogrammierung nicht veränderlich. Mit der Firmata-Bibliothek wurde allerdings eine Schnittstelle geschaffen, mit der auch mit anderen Programmiersprachen, wie z. B. JavaScript, auf Arduino Boards zugegriffen und das Programm zur Laufzeit verändert werden kann.

und dabei nicht aus einer fest verbauten Stelle entfernt werden muss. Die Kompilierung des Programms erfolgt, wie es auch bei Webanwendungen üblich ist, über das HTTP-Protokoll direkt auf dem Tessel Board. Details über die Ausführung eines JavaScript-Programms wurden in Kap. 2.1 ergründet.

Hohe Flexibilität und Erweiterbarkeit

Während die Mikrocontrollerboards des WunderBar-Systems nicht frei programmierbar sind, sondern lediglich Sensoren eingelesen werden können, bietet das Tessel Board, wie auch Arduino Boards, pure Ein- und Ausgänge für Spannungsimpulse über Steckkontakte, die frei programmierbar sind. Über diese Kontakte können Sensoren und Aktoren nach Belieben ergänzt werden, auch über USB-Ports. Im Gegensatz zum WunderBar-System muss der Entwickler im Umgang mit dem Tessel Board über elektronische Grundkenntnisse verfügen.

Lückenhafte Dokumentation

Ähnlich wie beim WunderBar-System ist die Dokumentation lückenhaft und liefert als Ergänzung ebenfalls eine Schnittstelle zum Arduino Board, „Über ein spezielles Modul dürfen Bastler Arduino-Shields und die dazugehörigen Arduino-Bibliotheken und Sketches weiterverwenden." [Make]

Geringe Stromstärke an den Ausgängen

Im Vergleich zu Arduino Boards liefern die Prozessoren von Tessel Boards weniger Leistung an den Ausgängen, was sich u. a. an der Helligkeit von mehreren (ohne Transistorschaltung) verbundenen LEDs bemerkbar macht. Während ein Arduino Board pro Ausgang 40mA bereitstellt, sind es bei einem Tessel Board lediglich 25mA[1].

Ungenaue Timer

Zudem merkt die Community an, dass JavaScript für Mikrocontroller beim Umgang mit Timern ungenau arbeitet. Das liegt sicherlich an der singlethread-basierten Natur von JavaScript.
Der Singlethreadansatz wurde in Kap. 2.6 beschrieben.
In JavaScript „gibt es keine Garantie dafür, dass die vorgesehene Verzögerung für eine exakt getimte Ausführung der Eventhandlerfunktion genau eingehalten wird" [Resig, S. 243]. Bei getimten Funktionen handelt es sich um „asynchrone [...] Callbackfunktionen, die die zwangsläufig in eine Ausführungswarteschlange eingereiht werden" [Resig, S. 244]. Falls in dieser Warteschlange bereits andere Aufgaben auf ihre Ausführung warten, muss die getimte Timer-Handlerfunktion nach dem FIFO-Prinzip zunächst alle *vor* ihr anstehenden Aufgaben abwarten, wodurch es zu Verzögerungen oder komplett zu Ausfällen von Aufgaben kommen kann. Abb. 2 veranschaulicht diesen Umstand.

1 vgl. Arduino Board: http://www.arduino.cc/en/pmwiki.php?n=Main/ArduinoBoardUno, Tessel Board: https://tessel.io/docs/hardware#pins-and-ports (Stand: 4. April 2015)

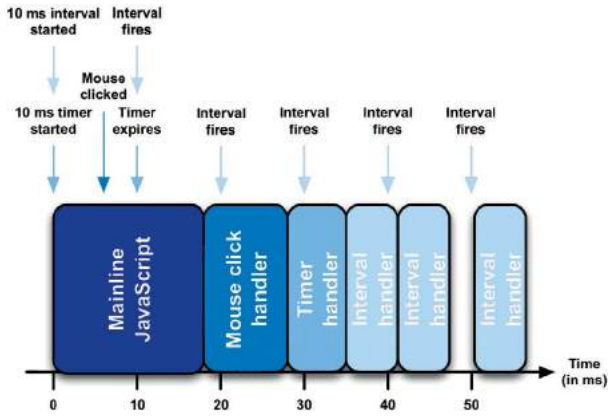


Abb. 2: Zeitlicher Ablauf bei der Ausführung des JavaScript-Kerncodes und der Eventhandler funktionen innerhalb eines Singlethreads

Läuft ein Timer ab, befindet sich der Timer-Handler, der nun ausgeführt werden sollte, *nicht* bereits in der Warteschlange, sondern wird zu diesem Zeitpunkt erst hinten in die Warteschlange eingefügt. Da im Singlethread-modell von JavaScript „immer nur *ein* Codeblock nach dem anderen ausgeführt wird, kann die Ausführung eines dieser Codeblöcke die Verarbeitung der anderen asynchronen Ereignisse blockieren" [Resig, S. 242] und es kann zu leicht verspäteten Aktionen kommen.

Bei präzisen Maschinenbauanwendungen sind ungenaue Timer nicht zu empfehlen. Bei medialen Anwendungen sind leichte Verzögerungen jedoch kaum realisierbar.
Allerdings begegnet das Tessel Board diesem Problem, indem es neben JavaScript sowie Python auch die Möglichkeit zu Programmierung anbietet: „A lower level alternative for folks who want the speed or memory safety benefit of a compiled language." [McKay] Z. B. mit C/C++.

Kosten

Im Vergleich zum WunderBar-System ist das Tessel Board ziemlich kostengünstig[1] und verzichtet auf Eigenschaften, die beim WunderBar-System im Messebetrieb ohnehin nicht genutzt werden würden.

Webserver mit direktem Anschluss von Elektronik

Während die meisten Arduino Boards nur über zusätzliche Computer mit dem Internet der Dinge verbunden werden können, benötigt ein Tessel Board weniger Hardware. Ein erfolgreich erprobter Ansatz ist die Kopplung eines Arduino UNO an einen Raspberry Pi-Einplatinencomputer, auf dem ein Node.js-Webserver läuft.
In speziellen Fällen, beispielsweise wenn Installationen über sehr wenig Platz verfügen, scheidet das recht sperrige Tessel Board aus und muss durch weitere Technologien ergänzt bzw. ersetzt werden.

1 Die im August 2015 erscheinende Version 2 des Tessel Boards kostet ca. 35US$

Fazit des Vergleichs

Alle drei Mikrocontrollerboards bzw. -systeme bieten eigene, durchaus geeignete Ansätze im Umgang mit dem *Internet der Dinge*. Alle drei Mikrocontrollerboards können mit JavaScript konfiguriert werden. Vom Arduino Board über das Tessel Board bis zum WunderBar-System nimmt der Komfort und der Preis zu und die jeweilige Dokumentation ab.
Da Arduino Boards flexibel, erschwinglich, am besten dokumentiert und vielen Entwicklern vertraut sind, werden sie nachfolgend für den Einsatz in komponentenreichen Event Media-Installationen näher untersucht.

6.2. Asynchrone serielle Kommunikation am Beispiel Arduino

Um mit anderen Geräten kommunizieren zu können, verwenden Mikrocontroller auf Arduino Boards eine *asynchrone serielle Kommunikation* (vgl. arduino.cc, SoftwareSerial).

Igoe, S. 423 und Sparkfun nennen dieses Protokoll *TTL-seriell*. Im Folgenden werden die Begriffe *seriell* und *parallel* bzw. *asynchron* und *synchron* voneinander abgewägt. Im Anschluss wird die Datenübertragungstechnik zwischen einem Arduino Board und einem Computer näher erläutert.

Seriell vs. parallel

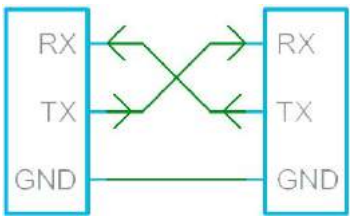


Abb. 3: Serielles Senden und Empfangen erfolgt über zwei separate Leiter

Eine Datenübertragung ist *seriell*, wenn Bit für Bit *nacheinander* in einem *einzelnen Leiter* übertragen wird. Für jedes Bit ist eine bestimmte Zeit vorgesehen. Ein serieller *Zugriff* bedeutet, dass das suchende Programm am Anfang einer Nachricht beginnt und sie Bit für Bit durchsucht, bis es die richtige Stelle findet. Bei der Kommunikation mit einem Arduino Board werden zwei Leitungen für jeweils eine Senderichtung (empfangen, senden) eingesetzt. Das ermöglicht eine gleichzeitige Verbindung in *beide* Richtungen und wird auch als *full-duplex* bezeichnet (vgl. Abb. 3).
Eine Datenübertragung ist *parallel*, wenn *mehrere Funktionen gleichzeitig* übertragen werden. Jede Funktion braucht einen eigenen Leiter. Bei *n* Funktionen werden *n* parallele Leiter benötigt. Ein Beispiel hierfür sind integrierte Computerbussysteme.

Asynchron vs. synchron

Bei einer *asynchronen* Datenübertragung können Bits *jederzeit* übertragen werden. Sie sind nicht an ein Taktsignal gebunden. Somit wird keine separate Taktleitung benötigt. Informationen zur Synchronisation von Sender und Empfänger sind mit Start- und Stoppbits in die Nachricht eingebettet.
Bei einer *synchronen* Datenübertragung hingegen sind Bits an ein Taktsignal gebunden, das in einer separaten Leitung übertragen wird.

Asynchron–serielle Protokolle

Asynchron-serielle Protokolle werden für einfache Ende-zu-Ende-Verbindungen über kurze Entfernungen verwendet. Beispiele sind RS232 und TTL-seriell.
Zum Vergleich: Asynchron-serielle Busprotokolle wie USB und RS485 werden für größere Netzwerke und größere Entfernungen verwendet.

Asynchron-serielle Protokolle werden heute hauptsächlich zur Kommunikation zwischen Computern und externer Hardware eingesetzt und werden dort ausgereifteren Bussystemen, wie z. B. USB vorgezogen, da die Entschlüsselung von asynchron-seriellen Protokollen einfacher ist.

Beispielcodierung einer seriellen Übertragung

Beispielsweise wird ein Zeichen an ein oder von einem Arduino Board mit 10 Bits pro Zeichen kodiert. Es handelt sich um je ein Start- und Stoppbit, das den Anfang und das Ende des kodierten Zeichens markiert sowie um das eigentliche Zeichen, *kurz: 8-N-1*. Laut Arduino Dokumentation werden für dieses Zeichen acht binäre Ziffern benötigt (vgl. Abb. 4). Dennoch werden bei der Kodierung eines Zeichens unter Verwendung der standardmäßigen ASCII[1]-Zeichencodierung[2] *nur sieben* Bits genutzt[3]. Das erste Bit der eigentlichen Nachricht nach dem Startbit hat standardmäßig den Wert `0` und wird in der Standardcodierung mit Zeichen aus der englischen Sprache nicht genutzt (vgl. graue `0` in Abb. 6). Dixon fasst das zu übertragende Datenwort folgendermaßen zusammen: „The Arduino default for the serial channel is one start bit (`0`), 7 data bits, 1 even parity bit, and one stop bit." Allerdings stellt er den Sachverhalt mit „1 even parity bit" vereinfacht dar: Ein Paritätsbit[4] resultiert aus der Summe der enthaltenen Bits eines Datenworts und kann demnach die Werte `0` *oder* `1` annehmen. Laut arduino.cc, *SerialBegin* und Arduino-Forum ist „The default [...] 8 data bits, no parity and 1 stop bit", kurz: *8-N-1*. Das Paritätsbit ist *nicht* vorhanden. Die ungenutzte `0` zu Beginn der Codierung ist demnach lediglich ein Auffüller des bisher siebenstelligen binären ASCII-Worts. Es dient zur Darstellung als Byte (mit acht binären Ziffern) und könnte mit Mehraufwand auch eingespart werden.
8-N-1 ist eine verbreitete Konvention bei der Codierung von Zeichen in Bytelänge. Arduino fügt sich ihr.

Abb. 4: Bitschema eines an/von an ein/em Arduino gesendeten Zeichens

1 ASCII: American Standard Code for Information Interchange

2 Zeichencodierungen sind im Wesentlichen Tabellen, die bestimmten Bit-Kombinationen konkrete Werte zuordnen. Zur seriellen Kommunikation mit einem Arduino Board wird die Zeichencodierung ASCII mit 7 Bit verwendet. In der Webentwicklung wird i. d. R. die Zeichencodierung UTF8 verwendet (vgl. `HTTP content-type im <head>` eines HTML-Dokuments).
Überblick: Ein *Zeichenvorrat* „beschreibt die [ungeordnete] Menge der Zeichen, die man [...] verwenden möchte [...]. Ordnet man den Zeichenvorrat [...], bekommt man einen *Zeichensatz* [...]. [Damit] Computer mit den Zeichen arbeiten können, müssen sie das Muster in Bits kennen. Dafür ist die *Zeichencodierung* [...] zuständig" [Wulftange].

3 vgl. ASCII-Tabelle, Arduino-Dokumentation, SparkFun

4 Ein Paritätsbit ist eine zusätzlich übertragene binäre Ziffer, die als einfache Fehlerkontrolle bei der Übertragung auf der Empfängerseite dient. Ist die Summe der binären Ziffer eines Datenworts gerade, wird ihm eine `0` vorangestellt. Ist sie ungerade, eine `1`.

Vergleich TTL-seriell mit RS232

TTL-seriell und RS232 unterscheiden sich im Wesentlichen im Spannungspegel und somit in der Reichweite.
Bei TTL-seriell wird eine logische `0` mit 0V repräsentiert und eine logische `1` mit 5V. Bei RS232 hingegen wird eine logische `0` mit einem positiven Spannungsegel (i. d. R. zwischen +3V und +15V) repräsentiert und eine logische `1` mit einem negativen (i. d. R. zwischen -3 und -15V). Die Reichweite beträgt bei TTL-seriell nur knapp einen Meter, während RS232 knapp 300m überbrücken kann.
Der Name TTL-seriell lässt sich von TTL-Bauteilen[1] ableiten, die ebenfalls mit 5V arbeiten.
Ansonsten funktionieren RS232 und TTL-seriell weitgehend gleich.

Parallel-seriell-Wandler UART

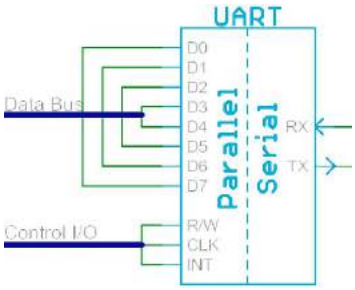


Abb. 5: Der digitale Schnittstellenbaustein UART wandelt parallel vorliegende Bits in serielle Bits um und umgekehrt.

Ein serieller Port kann von einem beliebigen asynchron-seriellen Punkt-zu-Punkt-Protokoll verwendet werden, also sowohl von RS232 als auch von TTL-seriell.
Beim seriellen Port des Mikrocontrollers auf einem Arduino Board handelt es sich um einen digitalen Schnittstellenbaustein, auch *UART*[2] genannt (vgl. Kopp, S. 8). Er wandelt die zu übertragenden Daten, die dem Prozessor parallel vorliegen, in ein serielles Signal um (vgl. Abb. 5). Zunächst wird ein Datenwort „aus dem Speicher in den Schnittstellenspeicher geschrieben" [Kopp, S. 8] (vgl. blau markiertes binär codiertes Zeichen in Abb. 6 (linker Teil)).
Danach wird es durch den UART-Baustein in einzelne Bits aufgeteilt, die nacheinander über dieselbe Leitung übertragen werden. Am Ziel werden die einzelnen Bits wieder zu zusammenhängenden Datenwörtern zusammengefügt. In Abb. 6 wird das codierte Wort entsprechend der ASCII-Tabelle als `K` interpretiert. Ist das ganze Datenwort übertragen worden, wird ein weiteres Datenwort aus dem Speicher ausgelesen.
Im Detail handelt es sich beim UART um einen integrierten Schaltkreis (IC), der die übertragenden Datenbits (bei Arduino 7 Bits) mit den Start-/Stopp- und ggf. Paritätsbit(s) *einrahmt* und in einer festgelegten Geschwindigkeit Bit für Bit ausgibt. Im Anschluss werden die einzelnen Bits von einem Pegelwandler in die gewünschten Spannungspegel umgewandelt, die über einen Leiter übertragen werden (vgl. arduino.cc, Serial).

Für ein Computerprogramm, das mit einem Arduino Board kommuniziert,- macht es keinen Unterschied, ob im Kabel die Daten mit den Spannungspegeln 0V bzw. 5V oder mit -12V bzw. +12V übertragen werden. Mit den eigentlichen Spannungspegeln kommt die Software nicht in Verbindung, da sie auf einer höheren Ebene im OSI-Modell operiert, in der die Spannungspegel bereits in logischen Werten umgewandelt vorliegen.
In vvv wird beispielsweise die Node `RS232` verwendet, um mit Arduino Boards zu kommunizieren, obwohl zur Kommunikation letztendlich TTL-seriell (und USB) verwendet werden.

1 TTL-Bauteile: Elektronische Elemente zur Realisierung von booleschen Funktionen, also Gatter, wie z. B. bipolare Transistoren
2 UART: Universal Asynchronous Receiver Transmitter. UART kann bis zu 115,2kBit/s übertragen.

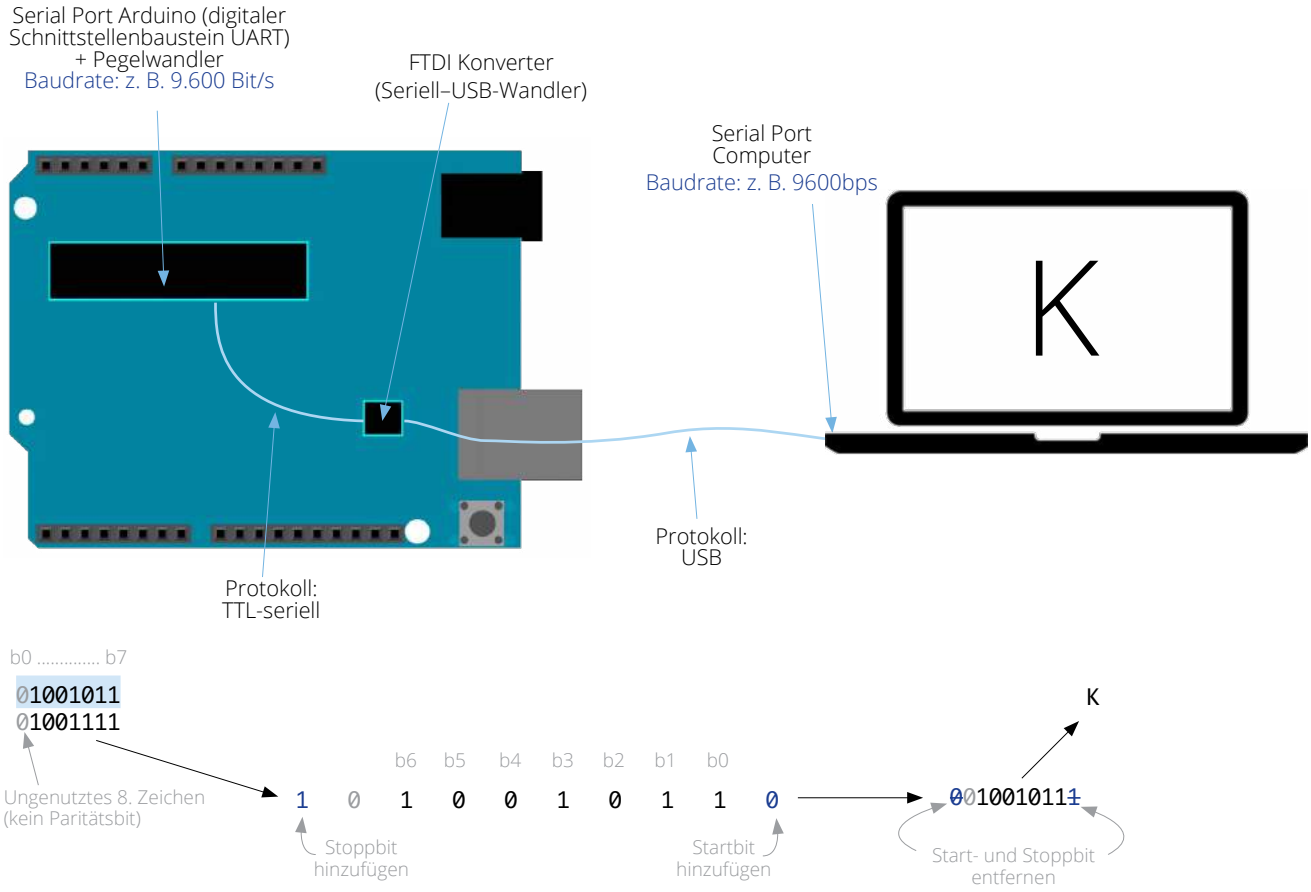


Abb. 6: Grafische Darstellung der Kommunikation zwischen einem Arduino Board und einem Computer

Kommunikation mit einem Computer

Kommuniziert ein Arduino Board mit einem Computer, wird auf der Computerseite für ein- und ausgehende Daten heute meistens das USB-Protokoll[1] verwendet.
Ein Arduino Board hingegen sendet und empfängt mit dem Protokoll TTL-seriell. Beide Seiten verfügen über einen seriellen Port (vgl. Igoe, S. 42).
Mit Hilfe eines USB-zu-seriell-Konverters (FTDI[2]-Chip) werden die beiden Protokolle ineinander übersetzt, damit ein Computer über USB mit einem Arduino Board kommunizieren kann.
Auf vielen Arduino Boards, wie z. B. dem Arduino UNO, ist ein USB-zu-seriell-Konverters bereits verbaut (vgl. Abb. 6).

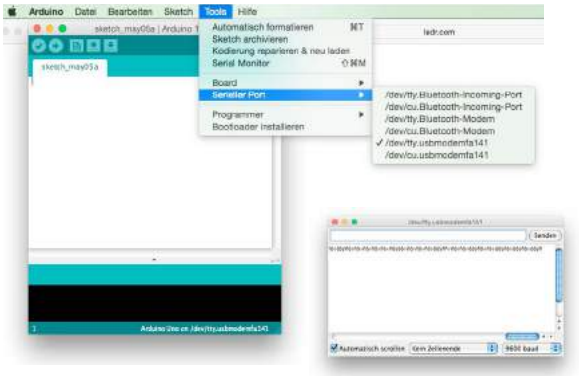
Übertragungsgeschwindigkeit: Baudrate

Damit serielle Daten des Senders beim Empfänger fehlerfrei empfangen werden, spielt neben den Start- und Endmarkierungen der zu übertragenden Zeichen und möglichst kurzen Kommunikationswegen die Übertra-

1 USB transportiert Daten asymmetrisch, d.h. Datenbits werden auf zwei separaten Leitern gegenphasig übertragen; bei einer logischen 1 jeweils 5V bzw. -5V. Beide Spannungspegel addiert sollten immer 0 ergeben. Bei Abweichungen handelt es sich um Störartefakte, die herausgefltert werden können (vgl. Igoe, S. 42). Folglich ist auch die mögliche Übertragungsdistanz von USB mit knapp 10m gegenüber TTL-seriell mit knapp 1m deutlich höher.

2 FTDI: Future Technology Devices International

```
var serialportpackage = require("serialport");
var SerialPort = serialportpackage.SerialPort;
var portname = "/dev/tty.usbmodemfa1441";
var serialport = new SerialPort(portname, {
  baudrate: 9600,
  parser: serialportpackage.parsers.readline("\n")
});
serialport.open(function(){
        console.log('Serial Port open');
});


serialport.write(..);    //send to Arduino
```



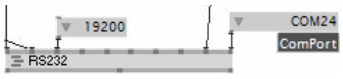


Abb. 7: Herstellung einer seriellen Verbindung: Links oben mit Node.js (mit dem Modul *serialport*), rechts oben mit der Arduino-Software, links unten mit vvv.

gungsgeschwindigkeit eine entscheidende Rolle. Der Empfänger muss mit derselben Geschwindigkeit lesen wie der Sender sendet.

Die Übertragungsgeschwindigkeit bei serieller Kommunikation wird mit der Größe Baud angegeben. Die Einheit ist *Bit/s* (auch mit *bps* abgekürzt). Die typische Baudrate 9.600 kann 9.600 Bits pro Sekunde übertragen. Pro codiertes Zeichen werden werden 10 Bits benötigt. Folglich werden mit der Baudrate 9.600 $\mathbf{9600\ Bit/s\ :\ 10\ Bit/Zeichen\ =\ 960\ Zeichen}$ pro Sekunde übertragen.

Um ein Arduino Board in ein Projekt zu integrieren, müssen dem Entwickler zumindest die Übertragungsgeschwindigkeit und der Name des seriellen Ports, mit dem es verbunden ist, bekannt sein (vgl. Abb. 7).
Abb. 9 zeigt ein Beispielprogramm zur zeichenkettenbasierten Kommunikation, je mit Processing, vvv und Node.js.

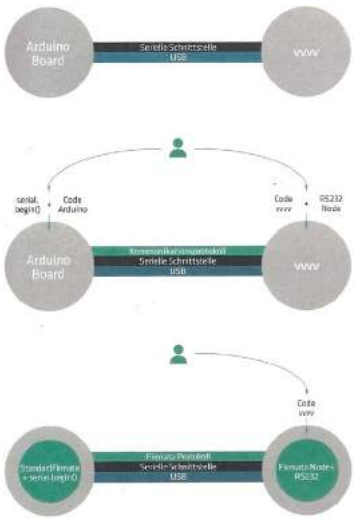


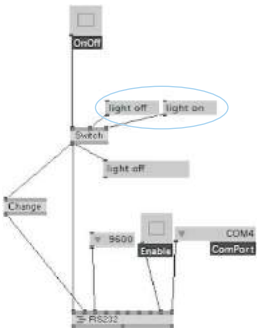
Abb. 8: Vergleich der seriellen Datenübertragung mit Firmata-Protokoll und ohne das Firmata-Protokoll

Asynchrone Kommunikation vs. asynchrone Datenübertragung

Asynchrone *Kommunikation* (vgl. Kap. 2.3.3.2) beschreibt das zeitlich versetzte Senden und Empfangen von Daten, ohne – wie es bei synchroner Kommunikation der Fall wäre – nachfolgende Prozesse bei Wartezeiten zu blockieren (vgl. HFT Berlin).
Asynchrone *Datenübertragung* hingegen beschreibt die Möglichkeit, Bits „zu beliebigen Zeiten" zu übertragen (vgl. Wikipedia, *AsDatenübertr*). Sie sind nicht an ein Taktsignal gebunden und werden demnach asynchron übertragen.
Zur Kommunikation zwischen Computer und einem Arduino Board können auf klassische Weise Zeichenketten ausgetauscht werden.
Um die Kommunikation mit einem Computer zu vereinfachen, wurde das Firmata-Protokoll entwickelt.
Da es in Verbindung mit Arduino Boards immer häufiger eingesetzt wird, möchte ich das Firmata-Protokoll an dieser Stelle näher beleuchten.

```
import processing.serial.*; Serial port;
void setup() {
   port=new Serial(this,Serial.list()[0],9600);
}
void draw() {
   textSize(15);
   text("Toggle LED", 10, 30);
}
boolean toggle = false;
void mousePressed() {
   if(toggle ==false){
      port.write("light on");
      toggle =true;
   }
   else{
      port.write("light off");
      toggle =false;
   }
}
```

Code für Processing



vvv-Patch

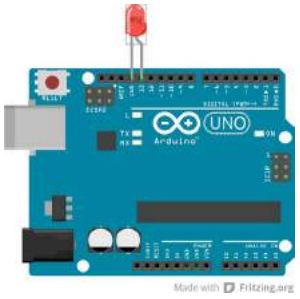
```
$('#led').click(function() {
   button = document.getElementById('led');
   toggleVal += 1;
   toggleVal %= 2;

   if(toggleVal == 0) {
      buttonState = 'light off';
      buttonBeschriftung(button);
   }
   if(toggleVal == 1) {
      buttonState = 'light on';
      buttonBeschriftung(button);
   }
   socket.emit('button_ClientToServer',buttonState);
   return false;
});
```

Ausschnitt aus dem clientseitigen JavaScript-Code einer einfachen Node.js-Applikation

Senderseite

Empfängerseite

```
int ledPin = 13;
String readString;
void setup() {
   Serial.begin(9600);
   pinMode(ledPin, OUTPUT);
}
void loop() {
   while (Serial.available()) {
      delay(3);
      char c =Serial.read(); readString += c;
   }
   if (readString.length() >0) {
      Serial.println(readString);
      if (readString  =="light on") {
         digitalWrite(ledPin, HIGH);
       }
       if (readString  =="light off") {
          digitalWrite(ledPin, LOW);
       }
       readString="";
    }
}
```



Gemeinsamer Arduino-Code und Aufbauschema

Abb. 9: Demonstration der zeichenkettenbasierten Kommunikation, jeweils zwischen Processing, vvv und Node.js mit der physischen Umwelt über ein Arduino Board. In diesem Fall wird jeweils über einen virtuellen Button eine LED an- und ausgeschaltet.

6.3. Firmata

Firmata ist ein asynchrones Datenübertragungsprotokoll, „das die Kommunikation zwischen Mikrocontrollern und einem Computer [von einer Software aus] ermöglicht" [Barth *et al.*, S. 111]. Bei der klassischen zeichenkettenbasierten Datenübertragung muss bereits bei der Programmierung des Microcontrollers festgelegt werden, welche Ein- und Ausgänge an einem Arduino Board verwendet werden sollen. „Außerdem muss das Datenpaket auf der Seite des Empfängers richtig dekodiert werden, um mit den Werten arbeiten zu können." [Barth *et al.*, S. 111] Der Entwickler muss – wie auf Abb. 8 (mittlerer Teil) gezeigt – gleichzeitig die Senderseite und die Empfängerseite im Blick behalten und aufeinander abstimmen.

Untersuchung der Stärken von Firmata im Vergleich zur zeichenkettenbasierten Kommunikation

Ein konkretes Beispiel für eine zeichenkettenbasierte Kommunikation zeigt Abb. 9, in dem mit drei verschiedenen Generatorsystemen (Processing, vvv, Node.js) nach einem Klick des Benutzers auf einen virtuellen UI-Button entweder die Zeichenketten `light on` oder `light off` seriell an ein Arduino Board gesendet werden. Das Arduino Board wartet auf genau *diese* Zeichenketten und schaltet bei deren Eintreffen eine LED entweder an oder aus.
Die zeichenkettenbasierte Form der Datenkommunikation kann bei komplexeren Beispielen allerdings recht kompliziert werden und rechtfertigt den Einsatz des Firmata-Protokolls. Ein Beispiel hierfür ist die Steuerung von RGB-LEDs in Echtzeit mit einem grafischen UI:

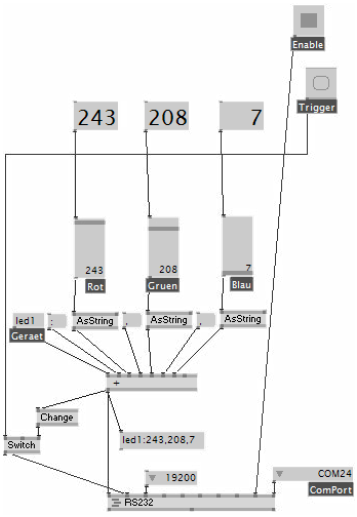


Abb. 10.1: Steuerung der Farbe einer RGB-LED mit der Zeichenkette `led1:243,208,7` in vvv.

In einem Versuchsaufbau mit vvv führt die Bestimmung des Farbtons der LED über Farbregler in Quasi-Echtzeit zu einem unschönen Flackern. Eine einmalige Wertzuweisung (vgl. Abb. 10) hingegen funktionierte tadellos.

Bei einer Echtzeitsteuerung müssen im geringen zeitlichen Abstand die Helligkeitswerte der RGB-LED in codierter Form übertragen werden.
Bei jeder Programmwiederholung wird je nach Stellung der Regler eine neue Zeichenkette generiert, die an den seriellen Port gesendet wird. Diese wird vom Mikrocontroller nach dem Empfang analysiert.

Zunächst wurden die Helligkeitswerte in Form von Dezimalzahlen übermittelt, z. B. `led1:243,208,7`.
Eine solche Datencodierung ist aufgrund der variablen Länge der zu übermittelnden Werte (zwischen einem und drei Zeichen für einen Helligkeitswert) und einer aufwändigen Analyse der Zeichenkette anhand der Zeichen `:` und `,` wenig performant.
Ein einzelner Helligkeitswert setzt sich dabei aus bis zu drei Zeichen zusammen. `255` wird nicht als ein einzelner Wert übertragen, sondern als drei Zeichenketten: `2`, `5` und `5`. Die Nachrichten werden vor dem Versand in Bytes umgewandelt. (vgl Igoe, S. 42).

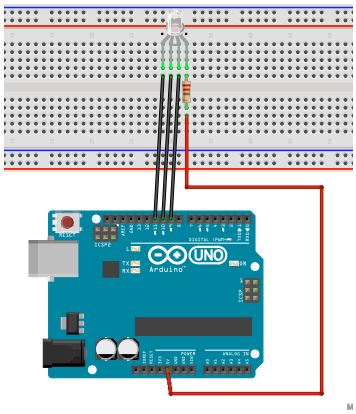
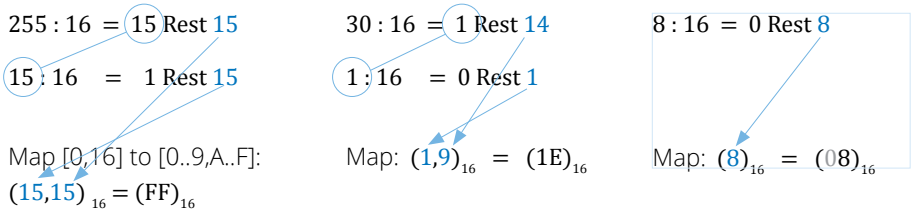


Abb. 10.2: Vereinfachtes Aufbauschema zur Ansteuerung einer RGB-LED mit einem Arduino Board (gemeinsamer Eingang, separate Ausgänge (hier ohne Widerstände zur Spannungs- und somit Helligkeitsregulierung der drei Farbchips der RGB-LED).

Die Länge der Zeichenkette wurde in weiteren Versuchen reduziert, indem *Dezimalwerte* durch *Hexadezimalwerte* ersetzt werden. Alle Farbwerte werden konstant durch zwei Zeichen repräsentiert, z. B. `DBD807`.
In den folgenden Beispielen werden je eine dreistellige, zweistellige und einstellige Dezimalzahl in eine Hexadezimalzahl umgerechnet.
Hexadezimalzahlen haben die Eigenschaft, dass alle dezimalen Helligkeitswerte (bis `255`) mit zwei Zeichen kodiert werden. Somit sind alle Helligkeitswerte einer LED abgedeckt. Die dezimalen Werte `0` - `15` bestehen im Dezimalzahlenraum aus nur einer Ziffer und werden durch eine führende `0` ergänzt (vgl. folgendes Umrechnungsbeispiel).



Die zu übertragende Zeichenkette beinhaltet nur noch sechs ASCII-Zeichen, wodurch die Übertragung flüssiger verläuft. Die Echtzeitsteuerung in vvv weist jedoch weiterhin Artefakte auf. Wird vvv an der Senderseite allerdings durch Node.js ersetzt, erfolgt die Echtzeitsteuerung der RGB-LED flüssig. Zu beachten ist, dass im JavaScript-Code der zeitliche Abstand der einzelnen Sendungen durch die Methode `setInterval` mit 100ms großzügig genug reguliert wird, während mir bei vvv keine entsprechende, native Methode bekannt ist. Demnach wird die zu sendende Zeichenkette bei jedem Programmdurchlauf an den seriellen Port geschrieben. Es werden in zu kurzer Zeit zu viele Zeichen in den Datenkanal geschrieben.[1]

Arbeitsweise des Firmata-Protokolls

Mit dem Firmata-Protokoll wird die Anzahl der zu übertragenden Bits noch deutlich reduziert. „As a comparison, the current Firmata protocol needs 3 bytes to set one digital pin or all digital pins." [Steiner] Das Firmata-Protokoll knüpft an den oben genannten Problemen mit der zeichenkettenbasierten Datenkommunikation an. Auch Steiner weist auf Artefakte bei der klassischen zeichenkettenbasierten Kommunikation ohne das Firmata-Protokoll hin: „ASCII protocols were ruled out since they would slow down the processing a lot [...] because of the latency and jitter."
Das Firmata-Protokoll basiert auf dem schlanken *MIDI message format*, einem Teil des MIDI-Protokolls.
Wie Abb. 8 (unterer Teil) veranschaulicht, ersetzt das Firmata-Protokoll eine selbst konfigurierte, zeichenkettenbasierte Kommunikationslösung (i. d. R. ein Austausch konkreter ASCII-Strings zwischen Empfänger und Sender, vgl. Abb. 9).
Abb. 8 (unterer Teil) zeigt auch, dass das Firmata-Protokoll auf dem seriellen Protokoll basiert. Das Firmata-Protokoll ist somit eine Zwischenschicht zwischen Benutzer und den seriellen Daten.

Einsatz von Firmata für komponentenreiche Medieninstallationen

Für Arduino Boards wird das Firmata-Protokoll als Bibliothek implementiert. Nachdem der Sketch[2] `Standard_Firmata` auf den Mikrocontroller des Arduino Boards geschrieben wurde[3], agiert das Arduino Board als "brainless slave" [vvv, Forum]. Über ein GUI kann uneingeschränkt auf das Arduino Board zugegriffen werden[4]. Von dort kann der Entwickler über das Arduino Board direkt mit der physischen Umwelt kommunizieren. Er hat vollen Zugriff auf alle Ein- und Ausgangs-Pins eines Arduino Boards.
Die Verwendung der Firmata-Bibliothek erübrigt die Programmierung eines separaten Programms für den Mikrocontroller.

---

1 vgl. http://vvv.org/forum/send-serial-data-fade-led (Stand: 22. April 2015)
2 Sketch ist der Arduino-eigene Name für ein Programm, das auf Arduino Boards läuft.
3 Mit dem Firmata-Protokoll bleibt die oft nötige Flexibilität vorhanden: Mit `#include <Firmata.h>` kann die Firmata-Bibliothek in bestehende Arduino-Sketches importiert werden. Mit der Methode `Firmata.sendAnalog` können beliebige Zahlenwerte an die Ausgänge des Arduino Boards geschrieben werden, die somit über das Firmata-Protokoll erreichbar sind.
4 z. B. in Verbindung mit vvv, Adobe Flash, Processing, Node.js, ...

Der Code für das Arduino Board entfällt. Im Beispiel auf Abb. 9 auf Seite 117 kann mit Firmata der Code für die *Empfängerseite* (unterer Teil) eingespart werden.
Mit Hilfe des Firmata-Protokolls kommunizieren Sender und Empfänger *nicht* über Zeichenketten, auf die beide Kommunikationsendpunkte abgestimmt sein müssten.

6.4. Zusammenfassung

In diesem Kapitel wurden mit dem Arduino Board, dem Wunderbar-System und dem Tessel Board drei Mikrocontrollersysteme für den Einsatz in komponentenreichen Event Media-Installationen miteinander verglichen und nach Eignung untersucht. In Anschluss wurde auf die klassische asynchrone serielle Kommunikation zwischen einem Arduino Board und einem Computer eingegangen. Dabei wurde das Problem der zeichenkettenbasierten Kommunikation untersucht und mit Hilfe des Firmata-Protokolls gelöst.
Schließlich wurde festgestellt, dass sich JavaScript mit allen untersuchten Mikrocontrollersystemen flexibel vereinbaren lässt.

7. Zusammenfassung und Ausblick

Messepräsenzen sind für viele Unternehmen von hoher Bedeutung. Mit der Allgegenwärtigkeit von Präsentationstechnik und deren stetiger Weiterentwicklung sind Unternehmen dazu angehalten, ihre Produktpräsentationen ihrer Mitbewerber zu differenzieren. Ein möglicher Ansatz ist, das Publikum über mehrere Sinne anzusprechen und es aktiv in die Präsentation einzubeziehen (vgl. Kap. 1.1). Hier kann das Publikum über intuitive Post-WIMP Interfaces direkt mit virtuellen Computersystemen interagieren und über die Grenzen virtueller Computersysteme hinaus mit Effekten überrascht werden. Die Einbeziehung der physischen Umwelt spielt dabei eine große Rolle.
Viele mittelständische Unternehmen verbinden ihre beworbenen Produkte und die präsentierende Medientechnik unzureichend bzw. überhaupt nicht miteinander (vgl. Kap. 1.2.1). Mit der Erkenntnis *Eine Produktpräsentation ist besonders wirkungsvoll, wenn das beworbene Produkt und die präsentierende Medientechnik grenzenlos ineinander verzahnt werden* wurde ein allgemeiner Aufbau für erlebnisreiche Produktpräsentationen konzipiert, der aus einer Server- und Clientapplikation, einer Datenbank und einem Mikrocontrollerboard für die Kommunikation mit der physischen Umwelt besteht (vgl. Kap. 1.2.4).

Im Anschluss wurden einige Möglichkeiten eruiert, diese Komponenten möglichst komfortabel miteinander zu verbinden. Mit vvvv (vgl. Kap. 1.3.3.1) wurde eine grafische Programmiersprache nach Eignung untersucht und schließlich ausgeschlossen. JavaScript ist eine funktionale Skriptsprache aus der Webentwicklung mit einem stetig wachsenden Funktionsumfang. Sie bietet dem Entwickler bei der Realisierung von Medieninstallationen nach einiger Eingewöhnungszeit einige komfortable Mittel an (vgl. Kap. 2).

Es wurden einige grundlegende Konzepte von JavaScript zur Realisierung von komponentenreichen Event Media-Installationen auf Messen und Events beleuchtet, die den genannten konzeptionellen Anforderungen gerecht werden. JavaScript verfolgt einen ereignisbasierten Ansatz (vgl. Kap. 2.3) und nutzt ein Singlethreadmodell (vgl. Kap. 2.4). Hiermit funktionieren beispielsweise flexible Webanwendungen auf teilweise leistungsschwachen mobilen Endgeräten. Mit dem Singlethreadmodell können zudem flexible Webserver auf durchschnittlichen Computern erstellt werden, die mit einer großen Anzahl an Anfragen problemlos umgehen können.
In diesem Zusammenhang wurden mit HTTP, AJAX, und WebSockets einige fortschrittliche Kommunikationstechnologien zwischen Clients und einem Webserver untersucht (vgl. Kap. 3.1f). Es wurde ein Node.js-Webserver eingesetzt (vgl. Kap. 3.5).
Um dieses System mit einem *Gedächtnis* auszustatten, wurden einige moderne NoSQL-Datenbanken im Vergleich zu relationalen Datenbanken untersucht und festgestellt, dass diese für den Einsatz in komponentenreichen Event Media-Installationen teilweise geeigneter sind (vgl. Kap. 5.2).

Mit den untersuchten technischen Mitteln wird in Kap. 4 eine Projekteplattform entwickelt, mit der mittelständische Kunden von ICT die Möglichkeit haben, sich über ein grafisches User Interface und mehreren Suchkriterien von geeigneten Projektvorschlägen inspirieren zu lassen.
Schließlich wurden die Mikrocontrollersysteme Arduino, WunderBar und Tessel erörtert (vgl. Kap. 6.1f). Mit Hilfe von Sensoren und Aktoren stellen sie die Verbindung eines virtuellen Computersystems zu der physischen Umwelt her. In Bezug auf das Arduino Board wurde das asynchrone serielle Datenaustauschprotokoll *TTL-seriell* experimentell durchleuchtet (vgl. Kap. 6.2).

Schließlich wurde bewiesen, dass sich JavaScript bestens dafür eignet, eine Event Media-Installation mit allen konzeptionell erforderlichen Komponenten zu erstellen und eine erlebnisreiche Produktpräsentation zu realisieren.
Grundsätzlich ist JavaScript in der Lage, alle Komponenten zu konfigurieren. Dabei werden andere Programmiersprachen allerdings nicht ausgeschlossen. Sie können allerdings mit JavaScript kombiniert werden. Letztendlich verwendet ein Entwickler das für ihn geeignete Werkzeug.

Das Internet der Dinge (vgl. Kap. 1.3) ist noch in einem jungen Stadium und entwickelt sich rasant fort. Somit sind Webtechnologien ein Zukunftsmarkt, die in den unterschiedlichsten Bereichen zunehmend unser alltägliches Leben beeinflussen werden.

In Kap 3.5.5 wurde die Kommunikation zwischen einem Node.js-Webserver und einem Arduino Board zeichenkettenbasiert gelöst.
Mit Hilfe des Firmata-Protokolls (vgl. Kap. 6.3) ist es nun möglich, Arduino Boards komfortabel mit verschiedenen Programmiersprachen zu verbinden. Somit kann eine Kommunikation mit der physischen Umwelt direkt in z. B. einer Webentwicklungsumgebung zustande kommen.

Somit kann eine komponentenreiche Event Media-Installation in Verbindung mit einem Arduino Board vollständig in *einer* Entwicklungsumgebung und mit *einer* Programmiersprache entwickelt werden – in diesem Fall mit JavaScript.

Am Ende dieser Arbeit steht ein technisches Grundgerüst, mit dem flexibel erlebnisreiche Produktpräsentationen realisiert werden können.

A. Anhang

A.1: Eidesstattliche Erklärung

A.2: Literaturverzeichnis

A.3: Abbildungs- und Listingverzeichnis

A.4: Fragebogen

A.5: Übersicht: Erscheinungsformen v. Funktionen

A.6: Asynchroner Programmverlauf (anschaulich)

A.7: Vortragsfolien

A.1: Eidesstattliche Erklärung

Ich erkläre ehrenwörtlich,

1. dass ich meine Bachelorarbeit selbstständig verfasst und ohne andere als die angegebenen Hilfsmittel verfasst habe.

2. dass ich die Übernahme wörtlicher Zitate aus der Literatur, sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

3. dass ich die Bachelorarbeit bei keiner anderen Prüfung vorgelegt habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum

Name

A.2: Literaturverzeichnis

Weblinks: Stand 16. Mai 2015

Kap. 1:

[Barth *et al.*]
Jan Barth [Hrsg.], Roman, Stefan, Grasy, Jochen Leinberger, Mark Lukas, Markus Lorenz Schilling:
„Prototyping Interfaces - Interaktives Skizzieren mit vvvv“, 1. Auflage
Mainz: Verlag Hermann Schmidt Mainz
Erscheinungsjahr: 2013

[Igoe]
Tom Igoe:
„Making Things Talk” (2. Auflage)
O'Reilly Media
Erscheinungsjahr: 2012

[Jacob *et al.*]
R. J. Jacob, A. Girouard, L. M. Hirshfield, M. S. Horn, O. Shaer, E. T. Solovey, J. Zigelbaum:
„Reality-based interaction: a framework for post-WIMP interfaces”
Weblink: http://hci.uni-konstanz.de/downloads/BlendedInteraction.pdf
Florenz, IT: 26. Annual SIGCHI Conference on Human Factors in Computing Systems
Vortragsdatum: 5. April – 10. April 2008

[Permaactive]
„PERMAACTIVE”
Weblink: http://permaactive.com
Erscheinungsdatum: April 2011

[Reiterer]
Harald Reiterer:
„Blended Interaction Ein neues Interaktionsparadigma”
Arbeitsgruppe MCI, Uni Konstanz
Weblink: http://hci.uni-konstanz.de/downloads/BlendedInteraction.pdf
Erscheinungsjahr: 2014

[Scholz]
Ronny Scholz:
„OSI-Schichten-Modell”
Weblink: http://www.netzwerke.com/OSI-Schichten-Modell.htm
Erscheinungsjahr: 2011

[vvvv.org]
„vvvv - a multipurpose toolkit”
Weblink: http://vvvv.org

[Wikipedia, *WIMP*]
„WIMP (Benutzerschnittstelle)”
Wikipedia
Weblink: http://de.wikipedia.org/wiki/WIMP_(Benutzerschnittstelle

Kap. 2:

[Allardice & Rose]
Simon Allardice, Thomas Rose:
„Kompilierte und interpretierte Sprachen”
Video2Brain
Weblink: https://www.video2brain.com/de/tutorial/kompilierte-und-interpretierte-sprachen
Erscheinungsdatum: 6. Dezember 2013

[Breck-McKye]
Jimmy Breck-McKye:
„What is “callback hell””
Stackoverflow
Weblink: http://stackoverflow.com/a/25098235/2426386
Erscheinungsdatum: 2. August 2014

[Castledine]
Earle Castledine, Craig Sharkie:
„jQuery - Vom Novizen zum Ninja“
Haar, D: Franzis
Erscheinungsjahr: 2012

[Dashewski]
Evan Dashevski:
„How to Create an App for iOS, Android, or Windows Phone”
PC Mag Online
Weblink: http://www.pcmag.com/article2/0%2C2817%2C2476480%2C00.asp
Erscheinungsdatum: 23. Februar 2015

[DotNetPro]
„Rettung aus der Callback-Hölle?”
Weblink: http://www.dotnetpro.de/service/link292.aspx
Erscheinungsdatum: 5. November 2012

[ecma, V. 5.1]
„Standard ECMA-262”
Weblink: http://www.ecma-international.org/publications/standards/Ecma-262.htm

[Freeman & Robson]
Eric Freeman, Elizabeth Robson:
„JavaScript Programmierung von Kopf bis Fuß”
Verlag: Köln, D: O'Reilly Media
Erscheinungsjahr: 2015

[Guffa]
Guffa:
„Are jQuery fadeIn(), animation() functions non-blocking?”
Stackoverflow
Weblink: http://stackoverflow.com/a/6880466/2426386
Erscheinungsdatum: 30. Juli 2011

[Flanagan]
David Flanagan:
„JavaScript: The Definitive Guide” (6. Auflage)
USA: O'Reilly & Associates
Erscheinungsjahr: 2011

[Havoc]
„Callbacks, synchronous and asynchronous"
Weblink: http://blog.ometer.com/2011/07/24/callbacks-synchronous-and-asynchronous/
Erscheinungsjahr: 2011

[HFT Berlin]
Thomas Sakschewski:
„Asynchrone - Synchrone Kommunikation"
Weblink: http://www.wissensstrukturplan.de/wissensstrukturplan/glossar/a_asynchrone_kom.php
Erscheinungsdatum: 29. April 2010

[Kangax]
„What's wrong with extending the DOM"
Perfection Kills
Weblink: http://perfectionkills.com/whats-wrong-with-extending-the-dom/
Erscheinungsdatum: 5. April 2010

[Kantor]
Ilya Kantor:
„Functions: declarations and expressions"
Weblink: http://javascript.info/tutorial/functions-declarations-and-expressions
Erscheinungsjahr: 2011

[MDN (Closures)]
MDN Mozilla Developer Network:
„Closures (Funktionsabschlüsse)"
Weblink: https://developer.mozilla.org/de/docs/Web/JavaScript/Closures
Seite noch in Arbeit

[Microsoft]
„Asynchrones Programmieren in JavaScript (HTML)"
Microsoft Dev Center
Weblink: https://msdn.microsoft.com/de-de/library/windows/apps/hh700330.aspx
Erscheinungsdatum: 5. November 2012

[Neßelrath]
Robert Neßelrath:
„Web 3.0"
Saarbrücken: Universität des Saarlandes, WS 2006/07
Microsoft Dev Center
Weblink: http://www.dfki.de/~kipp/seminar_ws0607/reports/RobertNesselrath.pdf
Erscheinungsjahr: 2007

[Nolte]
Karsten Nolte:
„Warum funktional Programmieren?"
Weblink: http://www.karsten-nolte.de/publikationen/funktionale-konzepte-in-javascript/warum-funktional-programmieren/
Erscheinungsdatum: 5. April 2011

[Pröll]
Stefan Pröll:
„MySQL 5.6: Das umfassende Handbuch", 2. Auflage
Bonn, D: Rheinwerk Verlag (Galileo Computing)
Erscheinungsdatum: 29. Juli 2013

[Reed]
Nico Reed:
„What are callbacks?"
Weblink: https://docs.nodejitsu.com/articles/getting-started/control-flow/what-are-callbacks
Erscheinungsdatum: 26. August 2011

[Resig & Bibeault]
John Resig, Bear Bibeault:
„Geheimnisse eines JavaScript Ninjas"
Zwickau: mitp
Erscheinungsjahr: 2014

[Richard]
Richard:
„Understand JavaScript Closures With Ease"
Weblink: http://javascriptissexy.com/understand-javascript-closures-with-ease/
Erscheinungsjahr: 2015

[Richardson]
Leland Richardson:
„Functional JavaScript, Part 4: Function Currying"
Weblink: http://tech.pro/tutorial/2011/functional-javascript-part-4-function-currying
Erscheinungsjahr: 2014

[Roberts]
Philip Roberts:
„What the heck is the event loop anyway?"
Vortrag auf der JSConf 2014
Weblink: http://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html
Vortragsdatum: 14. Oktober 2014

[Rohles]
Björn Rohles:
„Was ist das Web 3.0?"
Weblink: http://www.netzpiloten.de/begriffsklarung-was-ist-das-web-30/
Erscheinungsjahr: 2014

[Samuel]
Mike Samuel:
„Is JavaScript interpreted by design?"
StackExchange
Weblink: http://programmers.stackexchange.com/a/138541
Erscheinungsdatum: 6. März 2014

[Schäfer]
Mathias Schäfer:
„Organisation von JavaScripten: Objektverfügbarkeit und this-Kontext"
Weblink: http://molily.de/js/organisation-verfuegbarkeit.html
Erscheinungsjahr: 2009

[Simpson (Async)]
Kyle Simpson:
„You Don´t Know JS –Async & Performance"
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2015

[Simpson (this)]
Kyle Simpson:
„You Don´t Know JS – this & Object Prototypes"
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2014

[Simpson (Scope)]
Kyle Simpson:
„You Don´t Know JS – Scope & Closures"
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2014

[w3cSchools (Scope)]
„JavaScript Scope"
w3schools
Weblink: http://www.w3schools.com/js/js_scope.asp

[Wikibooks]
„Websiteentwicklung: JavaScript: Einleitung"
Wikibooks
Weblink: http://voidcanvas.com/describing-node-js/
Erscheinungsjahr: 2012

[Wikipedia, AsyncKomm]
„Asynchrone Kommunikation"
Wikipedia
Weblink: http://de.wikipedia.org/wiki/Asynchrone_Kommunikation

Kap. 3:

[Grigorik]
Ilya Grigorik:
„WebSocket"
Weblink: http://chimera.labs.oreilly.com/books/1230000000545/ch17.html
Erscheinungsjahr: 2014

[Garrett]
Jesse James Garrett:
„Ajax: A New Approach to Web Applications"
Weblink: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/
Erscheinungsdatum: 18. Februar 2005

[Hahn]
Evan Hahn:
„Understanding Express.js"
Weblink: http://evanhahn.com/understanding-express/
Erscheinungsdatum: 5. März 2014

[jQuery, AJAX]
„jQuery.ajax()"
Weblink: http://api.jquery.com/jquery.ajax/

[Kegel]
Dan Kegel:
„The C10K problem"
Weblink: http://www.kegel.com/c10k.html
Erscheinungsjahr: 2010

[Resig & Bibeault]
John Resig, Bear Bibeault:
„Geheimnisse eines JavaScript Ninjas"
Zwickau: mitp
Erscheinungsjahr: 2014

[Schmidt]
Prof. Dr. Uwe Schmidt:
„Node.js Architektur"
Wedel: FH Wedel
Weblink: http://www.fh-wedel.de/~si/seminare/ss11/Ausarbeitung/08.nodejs/architektur.html
Erscheinungsdatum: SS2011

[Shan]
Paul Shan:
„Node.js – reasons to use, pros and cons, best practices!"
Void Canvas
Weblink: http://voidcanvas.com/describing-node-js/
Erscheinungsdatum: 11. Oktober 2014

[Springer]
Sebastian Springer:
„Node.js – Das umfassende Handbuch"
Bonn, D: Galileo Press
Erscheinungsjahr: 2013

[Walsh]
David Walsh:
„WebSocket and Socket.IO"
Weblink: http://davidwalsh.name/websocketNAL.pdf
Erscheinungsdatum: 29. November 2010

[Wikipedia, ReactorPattern]
„Reactor Pattern"
Wikipedia
Weblink: http://en.wikipedia.org/wiki/Reactor_pattern

[Wikipedia, WebSocket]
„WebSocket"
Wikipedia
Weblink: http://de.wikipedia.org/wiki/WebSocket

Kap. 4:

[Hinkelmann & Jordine]
Prof. Dr. Mathias Hinkelmann, M.Sc. Tobias Jordine:
„Datenbanken 1"
Stuttgart: Vorlesung an der HdM Stuttgart
Erscheinungsdatum: WS 2013/14

Kap. 5:

[Ha]
Tran Ngoc Ha:
„Charakteristika und Vergleich von SQL- und NoSQL-Datenbanken"
Leipzig, D: Universität Leipzig, Fakultät Mathematik und Informatik
Weblink: http://dbs.uni-leipzig.de/file/seminar_1112_tran_ausarbeitung.pdf
Erscheinungsdatum: 12. Dezember 2012

[Harvey]
Robert Harvey:
„Why use a database instead of just saving your data to disk?"
StackExchange
Weblink: http://programmers.stackexchange.com/questions/190482/why-use-a-database-instead-of-just-saving-your-data-to-disk/190483#190483
Erscheinungsdatum: 14. März 2013

[Hinkelmann & Jordine]
Prof. Dr. Mathias Hinkelmann, M.Sc. Tobias Jordine:
„Datenbanken 1"
Stuttgart: Vorlesung an der HdM Stuttgart
Erscheinungsdatum: WS 2013/14

Kap. 6:

[arduino.cc, SoftwareSerial]
„Arduino Software Serial Interface"
arduino.cc
Weblink: http://www.arduino.cc/en/Tutorial/SoftwareSerial

[arduino.cc, *SerialBegin*]
„Serial.begin()"
arduino.cc
Weblink: http://www.arduino.cc/en/Serial/Begin

[Barth *et al.*]
Jan Barth [Hrsg.], Roman, Stefan, Grasy, Jochen Leinberger, Mark Lukas, Markus Lorenz Schilling:
„Prototyping Interfaces - Interaktives Skizzieren mit vvv", 1. Auflage
Mainz: Verlag Hermann Schmidt Mainz
Erscheinungsjahr: 2013

[Dixon]
Prof Dr. Alan Dixon:
„Serial Data Transfer"
Weblink: http://web.sunybroome.edu/~eet_dept/eet267/Arduino.htm
Erscheinungsjahr: 2014

[Evans]
Dave Evans:
„Das Internet der Dinge – So verändert die nächste Dimension des Internet die Welt"
Cisco Internet Business Solutions Group (IBSG)
Weblink: http://www.cisco.com/web/DE/assets/executives/pdf/Internet_of_Things_IoT_IBSG_0411FINAL.pdf
Erscheinungsdatum: April 2011

[Igoe]
Tom Igoe:
„Making Things Talk" (2. Auflage)
Bonn (D), O'Reilly Media
Erscheinungsjahr: 2012

[Kopp]
T. Kopp:
„Bericht Industrielles Netzwerk RS-232", S.9
Berner Fachhochschule, Hochschule für Technik und Informatik
Bern, CH
Erscheinungsjahr: 2009

[Liedel]
Bradford Liedel:
„What does 8-N-1 mean?"
Weblink: http://www.modemhelp.net/faqs/8n1.shtml
Erscheinungsjahr: 1999

[Make]
„Tessel: JavaScript-Entwicklerboard fürs'Internet der Dinge'"
Weblink: http://www.heise.de/make/meldung/Tessel-JavaScript-Entwicklerboard-fuers-Internet-der-Dinge-1936379.html
Erscheinungsdatum: 15. Juni 2013

[McKay]
Jon McKay:
„First Look: Tessel 2 Embeds Node.js in Your Project for 35 Bucks"
Make
Weblink: http://makezine.com/2015/03/06/first-look-tessel-2-embeds-node-js-in-your-project-for-35-bucks/
Erscheinungsdatum: 6. März 2015

[Neuhaus]
Elisabeth Neuhaus:
„2,3 Millionen US-Dollar für einen Internet-der-Dinge-Baukasten"
Gründerszene
Weblink: http://www.gruenderszene.de/allgemein/finanzierung-relayr-wunderbar
Erscheinungsdatum: 25. September 2014

[Relayr]
„WunderBar APIs"
Weblink: https://developer.relayr.io/dashboard/sdk
Erscheinungsjahr: 2015

[Resig & Bibeault]
John Resig, Bear Bibeault:
„Geheimnisse eines JavaScript Ninjas"
Zwickau: mitp
Erscheinungsjahr: 2014

[Sparkfun]
„Serial Communication"
SparkFun
Weblink: https://learn.sparkfun.com/tutorials/serial-communication

[Steiner]
Hans-Christoph Steiner:
„Firmata: Towards making microcontrollers act like extensions of the computer"
New York, USA: Interactive Telecommunications Program, New York University
Erscheinungsjahr: 2009

[Texas Instruments]
„About BLE connection"
Texas Instruments
Weblink: https://e2e.ti.com/support/wireless_connectivity/f/538/t/207823
Erscheinungsjahr: 2012

[Wikipedia, *AsDatenübertr*]
„Asynchrone Datenübertragung"
Wikipedia
Weblink: http://de.wikipedia.org/wiki/Asynchrone_Daten%25C3%25BCbertragung

[Wulftange]
Markus Wulftange:
„Zeichensätze und Zeichenkodierungen"
Weblink: http://webkrauts.de/artikel/2006/zeichensatze-und-zeichenkodierungen
Erscheinungsdatum: 6. Dezember 2006

Anhang:

[Simpson (Scope)]
Kyle Simpson:
„You Don´t Know JS – Scope & Closures"
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2014

A.3: Abbildungs- und Listingverzeichnis

Weblinks: Stand 16. Mai 2015

Folgende Grafiken wurden mit Hilfe dieser Quellen entworfen bzw. komplett entnommen:

Kap. 1

Abb. 3:
Grafik beinhaltet Bildmaterial folgender Quelle:
ICT AG:
„Success Box“
Portfolio an interaktiven Exponaten der ICT AG
Erscheinungsdatum: August 2014

Abb. 5:
Andrey Boyarintsev:
„Linear Spread - illustrated guide to vvv for newbies in computer arts“
http://vvvv.org/sites/default/files/uploads/vvvv%20illustrated%20beta_ENG.pdf

Kap. 2

Listing 2:
John Resig, Bear Bibeault:
„Geheimnisse eines JavaScript Ninjas“, S. 81
Zwickau: mitp
Erscheinungsjahr: 2014

Abb. 5:
Grafik frei nach folgender Quelle:
Kyle Simpson:
„You Don´t Know JS –Scope & Closures“, S. 14
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2014

Abb. 6:
Grafik frei nach folgender Quelle:
Kyle Simpson:
„You Don´t Know JS –Scope & Closures“, S. 10
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2014

Abb. 15:
Grafik frei nach folgender Quelle:
Simon Allardice, Thomas Rose:
„Kompilierte und interpretierte Sprachen”
Video2Brain
Weblink: https://www.video2brain.com/de/tutorial/kompilierte-und-interpretierte-sprachen
Erscheinungsdatum: 6. Dezember 2013

Abb. 21:
Jesse James Garrett:
„Ajax: A New Approach to Web Applications”
Weblink: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/
Erscheinungsdatum: 18. Februar 2005

Listing 26:
Grafik frei nach folgender Quelle:
Kyle Simpson:
„You Don´t Know JS –Scope & Closures“, S. 10
Sebastopol, USA: O'Reilly Media
Erscheinungsjahr: 2015

Abb. 27:
Grafik frei nach folgender Quelle:
Philip Roberts:
„What the heck is the event loop anyway?”
Vortrag auf der JSConf 2014
Weblink: http://2014.jsconf.eu/speakers/philip-roberts-what-the-heck-is-the-event-loop-anyway.html
Vortragsdatum: 14. Oktober 2014

Kap. 3

Abb. 1:
Jesse James Garrett:
„Ajax: A New Approach to Web Applications”
Weblink: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications/
Erscheinungsdatum: 18. Februar 2005

Abb. 2:
Ilya Grigorik:
„2,3 Millionen US-Dollar für einen Internet-der-Dinge-Baukasten”
Weblink: http://chimera.labs.oreilly.com/books/1230000000545/ch17.html
Erscheinungsjahr: 2014

Abb. 3:
„Serial Communication”
SparkFun
Weblink: https://learn.sparkfun.com/tutorials/serial-communication

Abb. 5:
Isaac Roth:
„What makes Node.js faster than Java?”
Weblink: https://strongloop.com/strongblog/node-js-is-faster-than-java/
Erscheinungsdatum: 30. Januar 2014

Abb. 7:
Maciej Lasyk:
„Scaling & securing Node.js apps”
Weblink: http://de.slideshare.net/d0cent/scaling-and-securing-nodejs-apps-35127655
Erscheinungsjahr: 2014

Kap. 5

Abb. 11:
Andrey Boyarintsev:
„Linear Spread - illustrated guide to vvv for newbies in computer arts“
Weblink: http://vvv.org/sites/default/files/uploads/vvv%20illustrated%20beta_ENG.pdf

Kap. 6

Abb. 1:
„WunderBar Schematics“:
Weblink: https://www.relayr.io
Erscheinungsjahr: 2014

Abb. 3 – 5:
„Serial Communication”
SparkFun
Weblink: https://learn.sparkfun.com/tutorials/serial-communication

Abb. 2:
Grafik frei nach folgender Quelle:
John Resig, Bear Bibeault:
„Geheimnisse eines JavaScript Ninjas“, S. 241
Zwickau: mitp
Erscheinungsjahr: 2014

Abb. 8:
Jan Barth, Roman, Stefan, Grasy, Jochen Leinberger, Mark Lukas, Markus Lorenz Schilling:
„Prototyping Interfaces - Interaktives Skizzieren mit vvv“,
1. Auflage
Mainz: Verlag Hermann Schmidt Mainz
Erscheinungsjahr: 2013

Arduino Layouts

Kap. 3, Abb 11, 12,
Kap. 6, Abb. 6 – 10:
generiert mit Fritzing
Weblink: http://fritzing.org/home/

A.4: Umfrage

Wenn ich auf eine Messe gehe …

- ☐ will ich einen Überblick bekommen, was im Trend ist
- ☐ will ich mich im Detail über etwas Bestimmtes informieren
- ☐ ist mir das Erlebnis wichtiger als detaillierte Information zu einem Produkt. Informieren kann ich mich im Internet
- ☐ bin ich eigentlich nur zum Konsum da
- ☐ finde ich ggf. das Zusammenspiel von greifbaren, physischen Komponenten (z. B. das umworbene Produkt) und Präsentationstechnik (Displays, Projektionen, …) klasse
- ☐ halte ich mich hauptsächlich dort auf, wo die Präsentation am interessantesten ist
  Lenkt die Präsentationstechnik (z. B. Displays und LED-Wände) zu sehr ab und stört eher
- ☐ ist es mir eigentlich egal, ob das umworbene Produkt vorhanden ist oder nicht. Solange der Informationsgrad und die Präsentation angemessen sind, bin ich zufrieden
- ☐ möchte ich Dinge interaktiv und individuell „entdecken“
- ☐ mag ich interaktive Exponate, die ich selbst berühren und erleben kann
- ☐ wünsche ich mir in Zukunft, dass die umworbenen Produkte und die Präsentationstechnik funktionell besser miteinander verzahnt werden und nicht ohne Zusammenhang nebeneinander stehen (z. B. ein Display sollte ein Produkt nicht nur anzeigen; beide Komponenten sollten sich gegenseitig beeinflussen können)
- ☐ möchte ich Produkte zwar gerne ausprobieren, aber ich bevorzuge eine Steuerung über gewohnte mediale Geräte wie Tablets. Einige Geräte sind ja nicht gerade selbsterklärend
- ☐ ist es mir egal, ob ich direkt oder indirekt (z. B. über ein Tablet) mit dem umworbenen Produkt interagiere
- ☐ bin ich auf alle Fälle zu haben, wenn das umworbene Produkt mit einem trickreichen Spiel verbunden ist
- ☐ vermeide ich interaktive Exponate. Ich möchte nicht zu sehr auffallen.
- ☐ lege ich keinen Wert auf ein individuelles Erlebnis, sondern erlebe es lieber zusammen mit anderen Leuten in der Menge
- ☐ mag ich nicht-interaktive Show-Performances lieber als interaktive Exponate (z. B. Vorstellung der C-Klasse mit Chereografie, medialen Effekten und einem Riesenbrimborium)
- ☐ bin ich für Überraschungserlebnisse offen (z. B. plötzliche kalte Brise, Vibration, …)
- ☐ vermeide ich grundsätzlich Orte, wo es laut und voll ist

Geschlecht: ☐ m ☐ w

Alter: _____

Beruf: _____

Vielen Dank für die Teilnahme!

A.5: Erscheinungsformen von Funktionen

Funktionen in JavaScript können verschiedene Erscheinungsformen haben. Man unterscheidet zwischen *Funktionsdeklarationen* und *Funktionsausdrücken*[1]. Beispiele für Funktionsdeklarationen werden in Listing 1.1 bzw. 1.2 gezeigt. Beispiele für Funktionsausdrücke befinden sich in der Übersicht in Listing 2.

A.5.1. Funktionsdeklarationen

```
function myFunction() {
    console.log("Ahoi");
}

myFunction();  // Ahoi
```

```
myFunction();  // Ahoi

function myFunction() {
    console.log("Ahoi");
}
```

Listing 1.1 und 1.2: Funktionsdeklarationen können von jeder Stelle im Code korrekt aufgerufen werden, egal ob der Funktionsaufruf *nach* oder *vor* ihr steht.

Funktionsdeklarationen (vgl. Listing 1.1 und 1.2) werden bei der Kompilierung geladen bevor jeglicher Code ausgeführt wird, unabhängig davon an welcher Stelle im Code sich ihr Funktionsaufruf befindet.[2] „Function declarations are parsed at a pre-excutive stage, when the browser prepares to execute the code" [Kantor]. *Funktionsdeklarationen* sind Standalone-Funktionen ohne Bindung an ein Objekt.
Die übrigen Erscheinungsformen von Funktionen sind Funktionsausdrücke. Der Unterschied zwischen Funktionsdeklarationen und Funktionsausdrücken bei der Interpretierung eines JavaScript-Programms wird in Kap. 2.1 beschrieben.

A.5.2. Funktionsausdrücke

Funktionsausdrücke (vgl. Listing 2) werden erst dann geladen, wenn der Interpreter die jeweilige Zeile im Code erreicht. Dies liegt daran, dass die eigentliche Funktion auf der rechten Seite des `=` Operators deklariert wird (vgl. Simpson (Scope), S. 42, mehr dazu in Kap. 2.1 (Ausführung v. JS-Programmen). Um sicher zu gehen, dass Funktionsausdrücke im gesamten Code verfügbar sind, sollten sie an dessen Anfang platziert werden, z. B. im `<head>`.

In Listing 2 stelle ich einige Funktionsausdrücke gegenüber. Besonders bei der Erweiterung eigener Projekte durch fremde Bibliotheken macht es Sinn, einen Überblick über die Besonderheiten der unterschiedlichen Ausführungen von Funktionsausdrücken zu haben. Aufgrund der besonderen Eignung spezieller Funktionsausdrücke für spezielle Anwendungen (vgl. Listing 4) macht es in verschiedenen Fällen Sinn, einen bestimmten Funktionsausdruck anstelle einer Funktionsdeklaration zu verwenden. Da sie im Gegensatz zu Funktionsdeklaration *erst zur Laufzeit* interpretiert werden, sind sie zwar etwas langsamer als Funktionsdeklarationen, jedoch wird dieser Unterschied beim Einsatz auf mittelgroßen Veranstaltungen kaum bemerkbar sein. Funktionsdeklarationen werden *gleich zu Beginn* des JavaScript-Programms interpretiert und nicht erst beim Aufruf bzw. beim Zugriff auf eine Variable.

1 Funktionsausdrücke: Engl. Function expressions
2 Dieses Phänomen des bevorzugten Ladens *vor* der eigentlichen Ausführung des Codes wird als *Hoisting* bezeichnet. Das Phänomen *Hoisting* wird im Zusammenhang mit der Arbeitsweise der JavaScript-Engine in Kap. 2.1. näher erläutert.

1. Grundform:

```
var myFunction = function(){
    return “Ahoi”;
}
console.log(myFunction()); // Ahoi
```

```
myFunction();  // TypeError

var myFunction = function (){
    console.log(“Ahoi”);
}
```

Ein Funktionsausdruck wird nur dann korrekt aufgerufen, wenn der Funktionsaufruf *nach* ihr steht.

2. Inlinefunktonen:
2a. Anonyme Inlinefunktionen:

```
$(document).ready(function(){
    console.log(“Ahoi”); // Ahoi (3x)
});
```

2b. Benannte Inlinefunktionen:

```
var i = 0;
$(document).ready(function myFunction(){
    console.log(“Ahoi”); // Ahoi (3x)
    i++;
    if(i < 3) myFunction();
});
```

3. Funktion als Konstruktor zur Bildung von Objekten:

```
function MyConstructor(){
    this.doSomething= “Ahoi”;
}
var myObject = new MyConstructor();
console.log(myObject.doSomething); // Ahoi;
```

4. Funktion als Methode:

```
var myObject={
    myFunction:function(){
        return “Ahoi”;
    }
}
console.log(myObject.myFunction()); // Ahoi
```

5. Konstruierte Form:

```
var myFunction = new Function(‘a’, ‘return a’);
console.log(myFunction(“Ahoi”)); // Ahoi
```

6. Selbstaufrufende Funktionen = Direkte Funktionen = Immediately Invoking Funktion Expressions (IIFE)

```
(function(){
    console.log(“Ahoi”)
})(); // Ahoi
```

7. Arrow-Funktionen (ECMA Script 6)

```
var myFunction = a => {
    console.log( a );
};
myFunction(“Ahoi”); // Ahoi
```

Listing 2: Unterschiedliche Erscheinungsformen von Funktionsausdrücken für unterschiedliche Einsatzzwecke

A.6: Asynchroner Programmverlauf anschaulich

Folgendes alltägliches Beispiel veranschaulicht die asynchrone Arbeitsweise eines Programms.

Synchron

Anna und Berta gehen spazieren.
Sie kommen an einem Imbissstand vorbei und bekommen Hunger. Anna möchte eine Schnitzelbrötchen, Berta eine Brezel.
Sie stellen sich in die Schlange vor dem Imbissstand, in dem nur ein Mensch arbeitet: Zuerst Anna, dann Berta. Hinter ihnen steht Clara. Alle Wartenden kommen der Reihe nach dran. Erst Anna, dann Berta, dann Clara.
Zunächst bestellt Anna ihr Schnitzelbrötchen. Daraufhin legt der Verkäufer ein Schnitzel in die Mikrowelle.
Solange die Mikrowelle beschäftigt ist, müssen Anna und alle Kinder hinter ihr warten.

Sobald das Schnitzelbrötchen fertig ist, überreicht der Verkäufer es Anna. Dann geht Anna aus der Schlange.
Dann ist Berta dran. Bei ihr geht es schneller, denn bei ihr muss der Verkäufer nach ihrer Bestellung nur in den Brezelkorb greifen und ihr die Brezel geben. Obwohl es bei Berta schneller ging als bei Anna, wurde zuerst Anna bedient, weil sie sich vor Berta angestellt hatte.
Erst danach kann Clara ein Brötchen kaufen.

// Ende

Zeit

Asynchron

Anna und Berta gehen mit Bertas Hund spazieren.
Sie kommen an einem Schnellrestaurant vorbei und bekommen Hunger. Anna möchte ein Schnitzelbrötzchen, Berta eine Brezel.
Da der Hund das Schnellrestaurant nicht betreten darf, wartet Berta draußen mit ihm, bis Anna zurückkommt. Anna geht rein und stellt sich in die Schlange. Hinter ihr steht Clara.

Nachdem Anna ihre Bestellung aufgegeben hat, legt der Verkäufer ein Schnitzel in die Mikrowelle.

Anna geht aus der Schlange und wartet im Hintergrund, bis ihr Essen fertig ist.
Während die Mikrowelle Annas Schnitzel aufwärmt, kann der Verkäufer Claras Bestellung aufnehmen. Clara möchte ein Brötchen. Dazu muss der Verkäufer lediglich in einen Korb greifen und Clara ein Brötchen überreichen. Clara hat ihr Essen schneller als Anna, obwohl sie es erst nach Anna bestellt hat.

Callback (Staffellauf-Prinzip):

Sobald Anna nun ihr Essen bekommen hat, kann sie den Laden verlassen und kehrt zu Berta zurück.

Jetzt passt Anna auf den Hund auf und Berta kann in das Schnellrestaurant gehen, um ihre Brezel zu kaufen.

// Ende

A.7: Vortragsfolien

ICT HOCHSCHULE DER MEDIEN

Technische Konzeption für eine komponentenreiche Produktpräsentation

Präsentation zur Bachelorarbeit von Fabian Fiess (ff023), Juni 2015

Folie 1

Hauptintention bei Messepräsenzen

Auffallen
und positiv im Gedächtnis bleiben → als Erlebnis

Umsetzung – Status Quo:

- Displays und Videotechnik sind Standard
- Trend: Agressive audiovisuelle **Reizüberflutung**, Größe, Opulenz
- **Differenzierungspotentiale** in der Präsentationstechnik müssen genutzt werden

Was tun mittelständische Unternehmen ?

Herangehensweise für mittelständische Unternehmen (hpts. Maschinenbau) (vgl. Kap. 1.1)

Mittelständische Unternehmen können den besonderen Erlebnisfaktor folgendermaßen herstellen:

- Mit Hilfe von Vielseitigkeit, Innovation, Individualität, Interaktion, Immersion
- Indem möglichst **viele Sinne angesprochen** werden

→ Nutzung von geeigneten **Interfaces** und **Feedbacks**

Gewicht verlagern
stoßen
ziehen
näher kommen
laufen
schieben
kicken
ziehen
berühren
pusten
klettern
biegen
Interfaces
fliegen
musizieren
stampfen
malen
spielen
wischen
drehen
fahren
schreien
fangen
Gesten anwenden

+

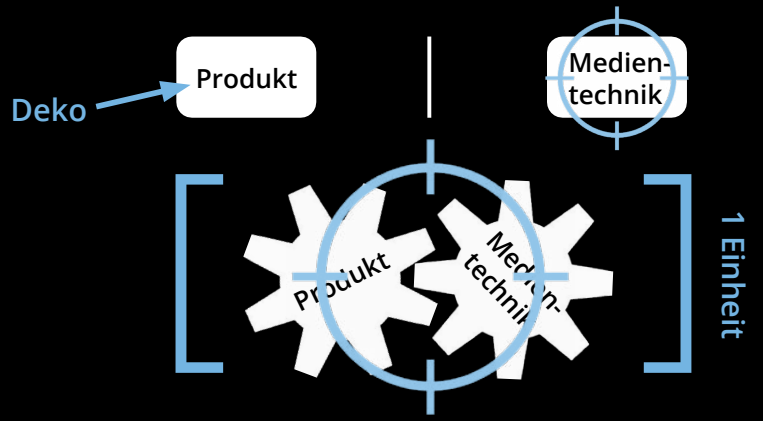
Vibrationen wahrnehmen
überrascht werden
erschaudern
in Bewegung versetzt werden
Feedback
Luft- und Wasser-brisen spüren
Düfte riechen
die besondere Haptik eines Interfaces fühlen

=

Interaktiver Showcase
Story und Informationen
Erlebnisreiche Produktpräsentation
Spiel
Informationen interessant vermittelt

Beobachtung (vgl. Kap. 1.2.1)

Das beworbene Produkt und die präsentierende Medientechnik müssen **besser miteinander verzahnt** werden.



N

Die Produktpräsentation wird komponentenreich (vgl. Kap. 1.2.4)

Technische Konzeption für eine **komponentenreiche** Produktpräsentation

Benötigte Komponenten für einen flexiblen typischen **Standardaufbau**:

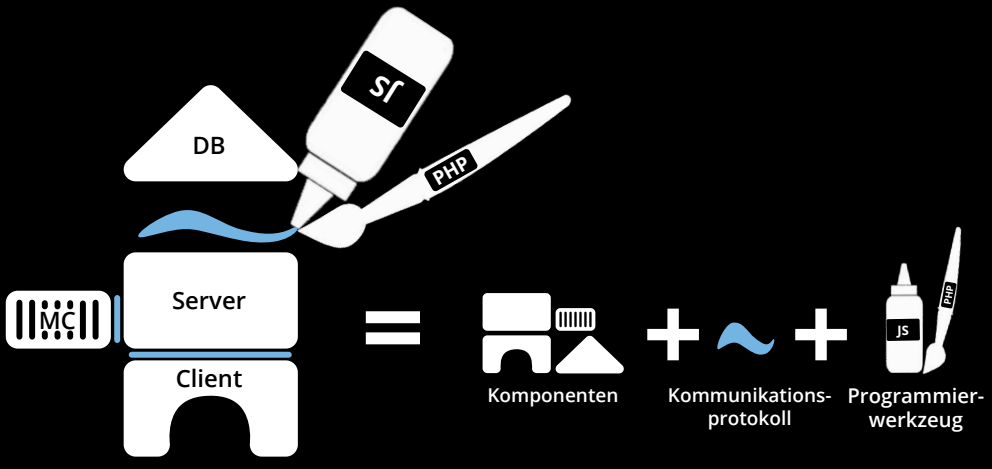
- Clientseitiges Screeninterface
- Webserver
- Datenbank
- Mikrocontrollerboard

Technische Konzeption (vgl. Kap. 1.3f)

Technische Konzeption für eine komponentenreiche Produktpräsentation

Wie kommunizieren technische Komponenten miteinander ?
Wie werden Gegenstände miteinander verbunden ?

Komponenten miteinander verbinden (vgl. Kap. 1.3f)



Choose the right tool for your job (vgl. Kap. 1.3f)

Bewährte Programmiersprachen *(Werkzeuge)* ...

HTML + CSS
Swift
PHP
Client
Java
JavaScript
C
C#
C++

nginx
Node.js
Webserver
IIS
Apache

SQL
Relationale Datenbanken
Datenbank
NoSQL-Datenbanken

Arduino Board
Wunderbar-System
Mikrocontroller
C
C++
Tessel Board
JavaScript

Choose the right tool for your job (vgl. Kap. 1.3f)

Bewährte Programmiersprachen *(Werkzeuge)* ...

Client

JS 4 ALL

Webserver

Datenbank

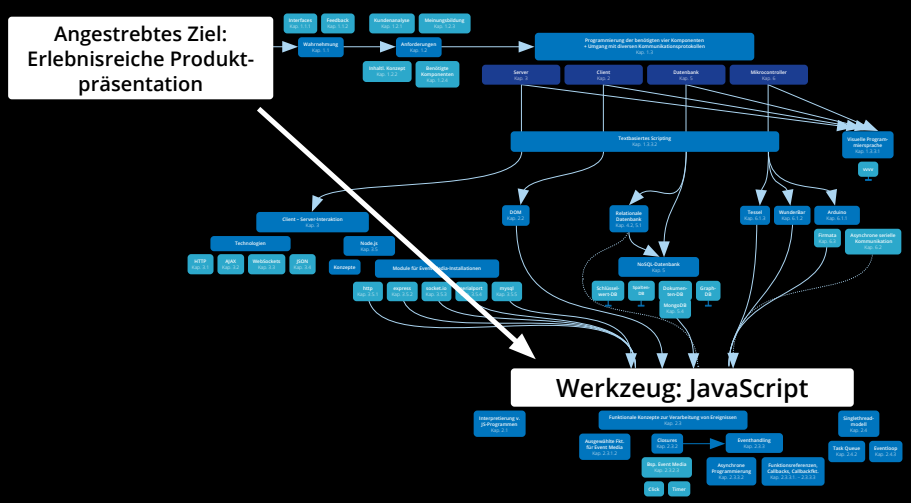
Mikrocontroller

Viele versch. Sprachen: Unkomfortabel → ~~Visuelle Programmiersprache (vvvv) statt Skriptsprache ?~~

Internet der Dinge

Übersicht

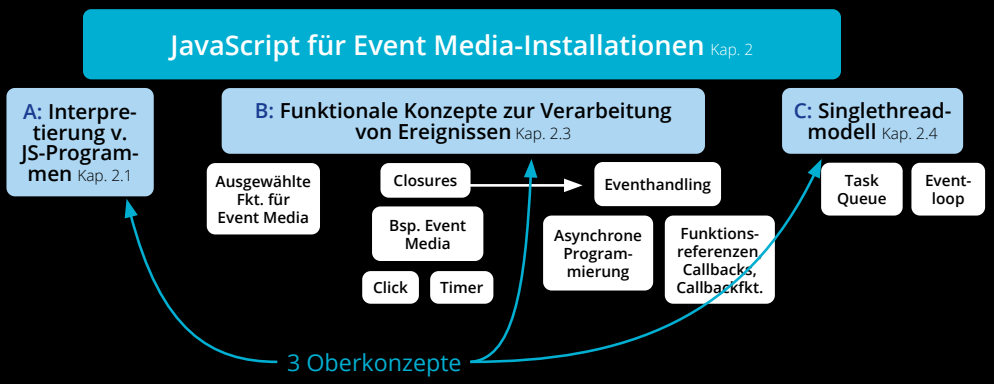
Angestrebtes Ziel: Erlebnisreiche Produkt-präsentation



P

Auswahl von JavaScript-Konzepten für Event Media-Installation (vgl. Kap. 2f)

mit dem Werkzeug umgehen



JavaScript f. Event Media-Installationen (vgl. Kap. 2f)

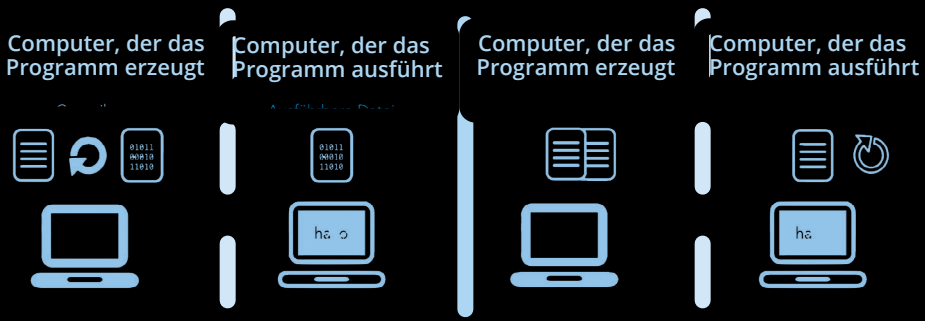
Die Programmiersprache JavaScript ist ...

- Interpretiert, plattformunabhängig → **A:** Interpretierung v. JavaScript-Programmen
- funktional, ereignisbasiert → **B:** Funktionale Konzepte
- leichtgewichtig, non-blocking → **C:** Singlethreadmodell

und wird eingesetzt ...

- für **leichtgewichtige** , **dynamische** Webanwendungen
- klassischerweise clientseitig
- nun auch für Webserver, Datenbanken, Mikrocontroller, ... → **Internet der Dinge**

A: Interpretierung von JavaScript-Programmen



Interpretierte Programmiersprachen, wie JavaScript, sind im Gegensatz zu kompilierten Sprachen **plattformübergreifend**.
Interpretierung findet **erst am Zielcomputer** statt.
Beteiligt: Compiler, Engine, Scope (vgl. S. 22f), Nebeneffekt: Hoisting (vgl. S. 24)

B: Funktionale Konzepte

- JavaScript ist eine funktionale, objektorientierte Programmiersprache mit **Objekten 1. Klasse** (vgl. S. 27f)
- ausgewählte **Funktionausdrücke für die ereignisbasierte Programmierung**:
  - **Inlinefunktionen** (häufig an Eventlistenern als Eventhandlerfunktionen gekoppelt)
  - **selbstaufrufende Funktionen** (Nutzung von Fremdbibliotheken, Bildung von Blockscopes in JavaScript)
- Konzept **Closures**: Referenz einer inneren Funktion in einer verschachtelten funktionalen Funktionsstruktur auf den Scope der umgebenden Funktion, z. B. zur Definition der Gültigkeit von Objekten (vgl. private, public, privileged in Java, vgl. S. 31f), Eventhandling (z. B. Click, Timer, vgl. S. 37f)
- Zum **Eventhandling**: Aufgaben in Event Media-Installation werden typischerweise
  - **asynchron** gelöst (zeitlich versetztes Senden und Empfangen, vgl. S. 43f)
  - unter Verwendung von **Callbacks und Callbackfunktionen** (Regulierung der Reihenfolge von Funktionen, vgl. S. 45)

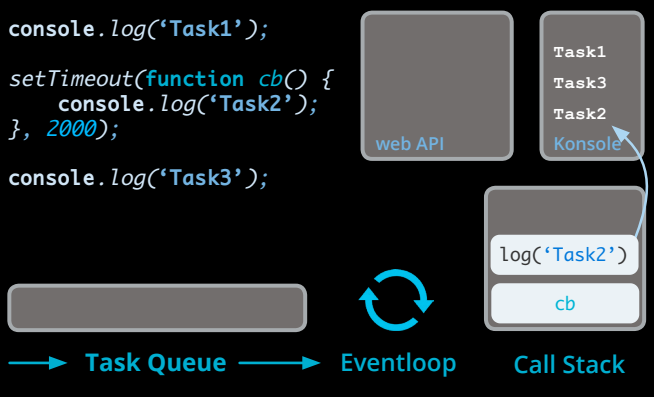
C: Singlethreadmodell

- Es kann **nur eine Aufgabe** zu einer bestimmten Zeit ausgeführt werden, also nicht mehrere Aufgaben paralell, wie es beim Multithreadmodell der Fall ist *(vgl. Einbahnstraße*, S. 50f) → leichtgewichtige Applikation
- Quasi-Parallelität kann realisiert werden mit ...
  - **asynchroner** Programmierung
  - **Task Queue** (Warteschlange für asynchron auszuführende Ereignisse vor dem Ausführungsstack, vgl. Baustellenampel, vgl. S. 52)
  - **Eventloop** (Vermittler zwischen Task Queue und Ausführungsstack, vgl. S. 53f)

C: Synchrone- u. asynchrone Funktionen, Task Queue und Eventloop im Singlethreadmodell

(vgl. S. 55, Schritt 8)

```
console.log('Task1');

setTimeout(function cb() {
    console.log('Task2');
}, 2000);

console.log('Task3');
```



R

Untersuchung von Kommunikationsprotokollen zwischen Webserver und Client (vgl. Kap. 3)

Web 2.0 (vgl. S. 58)

↓

- **HTTP** (klassisches Protokoll zur Verbreitung v. Ressourcen übers WWW, vgl. S. 59)
- **AJAX** (asynchrones Kommunikationsverf. zw. Webbrowser und Webserver, vgl. S. 60)
- **WebSockets** (bidirektionale Verbindung zw. Clients und einem Webserver, vgl. S. 62)
- **JSON** (Datenaustauschformat, das verschachtelte key/value-Paare enthält, vgl. S.63)

↓

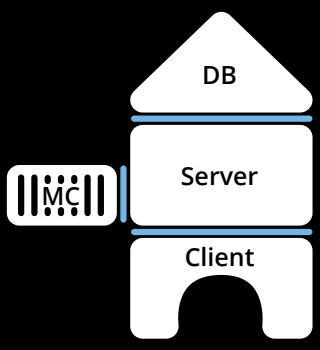
Verknüpfung mit JavaScript mit Hilfe von **Node.js**

Verbindung aller Komponenten einer konponentenreichen Event Media-Installation mit Node.js

Node.js ist JavaScript ohne Browser und wird typischerweise zur Erstellung von Webservern verwendet.
Node.js ...

- hat Zugriff auf das **Dateisystem** des Computers, auf dem der Server läuft
- kann aufgrund des Singlethreadmodells mit **vielen Clientverbindungen gleichzeitig** umgehen (non-blocking) und funktioniert auch auf durchschnittlichen Computern
- ist problemlos **horizontal skalierbar** (Verteilung der Rechenlast auf mehrere Computer, vgl. S. 68)
- ist **plattformunabhängig** (client- bzw. serverseitig nicht an ein bestimmtes Betriebssystem gebunden vgl. S. 67)

Verbindung aller Komponenten einer komponentenreichen Event Media-Installation mit Node.js



Die Funktionalität von Node.js befindet sich in unzähligen Modulen, die wie Bausteine an eine Applikation angebaut werden können ***(Baukastensystem)***.

Geeignete Node.js-Module zur Realisierung von komponentenreichen Produktpräsentationen (vgl. S. 68):

- **http** und/oder **express** z. B. zur Bereitstellung von Webinhalten und zur Einbindung von Middleware
- **socket.io** für den Austausch von Nachrichten zwischen Clients und dem Server
- **serialport** zur Kommunikation mit der physischen Umwelt
- **mysql** für eine Datenbankverbindung

Verbindung aller Komponenten einer komponentenreichen Event Media-Installation mit Node.js

Beispiele:

- **express**, **socket.io**, **serialport**:
  - **Webbrowser → Physische Umwelt**: Vgl. Beispiel S. 63 (RGB-LEDs über Farbkreis steuern mit Hilfe von mobilen Endgeräten)
  - **Physische Umwelt → Webbrowser**: vgl. Beispiel S. 75 (physische Umwelt steuert Webinhalte über Sensoren)
- **express**, **socket.io**, **mysql**: vgl. Beispiel S. 77 (Datenbankoperation über ein clientseitiges Webinterface

Anwendung der bisherigen Erkenntnisse anhand der Erstellung eine Projekteplattform (vgl. S. 80)

Sie soll den Kunden von ICT …

- einen **Überblick über Trends** in der Präsentationstechnik geben
- Anforderungen an **zukünftige** Produktpräsentationen näher bringen
- Projekte nach **unterschiedlichen Suchkriterien** anzeigen: Kategorie, Zeit, Branche, Messe, Ort, Agenturen, ähnliche Projekte, verwendete Technologien, Software

Verwendete Mittel:

- Client: HTML, CSS, JavaScript
- Server: Node.js, mit den Modulen mysql, socket.io, express
- Datenbank: MySQL

Anwendung der bisherigen Erkenntnisse anhand der Erstellung eine Projekteplattform (vgl. S. 80)

Aufgabenschritte:

- **Modellierung einer relationalen Datenbank** mit MySQL (vgl. S. 81f)
- **Entwicklung einer dynamischen Webanwendung** mit Präsentations- bzw. Wartungsbereich, mit HTML, CSS, JavaScript + Node.js (Module: express, http, socket.io, mysql, vgl. S. 90f)

T

Geeignete Datenbanken für Event Media-Installationen in Verbindung mit JavaScript (vgl. S. 98)

Vergleich **relationaler Datenbanksysteme** mit **NoSQL**-Datenbanksystemen (Schlüsselwertdatenbanken, spaltenorientierte Datenbanken, Dokumentendatenbanken, Graphdatenbanken)

→ Wahl der Dokumentendatenbank **MongoDB** aufgrund der Verwandtschaft mit JavaScript und der Nähe zu MySQL, jedoch mit komfortablerer Handhabung (vgl. S. 102)

Geeignetes Mikrocontrollerboard für Event Media-Installationen in Verbindung mit JavaScript

(vgl. Kap. 6)

Auswahlkriterien (horizontal) und geeignete **Kandidaten** (vertikal):

	Doku, Community	flexibel, erweiterbar	preiswert	intuitiv, konfigurierbar mit JavaScript
Arduino Board	optimal	optimal	günstig (+/–10 Euro)	Einarbeitungsaufwand, elektronische Grundkenntnisse
Wunderbar-System	nicht ausgereift	bedingt	teuer (180 Euro)	viele Konfigurationsmöglichkeiten, u. a. mit JavaScript
Tessel Board	nicht ausgereift	ziemlich	ziemlich (Tessel 2 ab 35 US $)	Tessel Boards verwenden nativ Node.js

Untersuchung der asynchronen seriellen Datenübertragung am Beispiel Arduino (vgl. S. 117)

- **Asynchrone Datenübertragung bedeutet**:
  - **Synchron** (vs. parallel): Bit für Bit werden in einem gemeinsamen Leiter übertragen
  - **Asynchron** (vs. synchron): Nicht an ein Taktsignal gebunden (*asynchrone Kommunikation* beschreibt zum Vergleich zeitlich versetztes Senden und Empfangen)
- **UART** (Parallel – seriell-Wandler, vgl. S. 114)
- **Baudrate** (Übertragungsgeschwindigkeit, vgl. S. 115)
- **Kommunikation mit Computern** (vgl. S. 116)

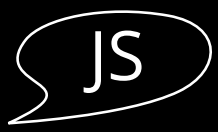
→ häufig auf komfortablere Weise lösbar mit **Firmata**

Firmata-Protokoll (Protokoll: TTL-seriell, vgl. S. 112)

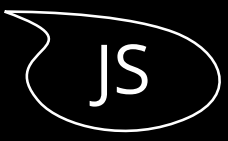
Firmata ist ein asynchrones Datenübertragungsprotokoll, das auf dem schlanken MIDI Message Format basiert und die **Kommunikation zwischen Mikrocontrollern und einem Computer** ermöglicht.

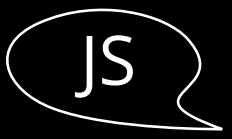
Uneingeschränkter Zugriff auf die Ein- und Ausgangs-Pins von Arduino Boards aus einem GUI aus zur direkten Kommunikation mit der physischen Umwelt

→ Lösung von Problemen mit Artefakten beim Fading einer RGB-LED in Echtzeit mit Hilfe einer klassischer zeichenkettenbasierten Datenübertragung (vgl. S. 118)



Technische Konzeption für eine komponentenreiche Produktpräsentation mit JavaScript





Danke für Ihre Aufmerksamkeit !