Bachelorarbeit
im Studiengang Medieninformatik

Möglichkeiten und Grenzen von Webtechnologien bei der Entwicklung von Kiosksoftware anhand der Implementierung einer Sharing-Station für Museen und Ausstellungen

vorgelegt von
Marcus Schreiter
an der Hochschule der Medien Stuttgart
am 06.04.2020

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

Erstprüfer: Prof. Uwe Schulz
Zweitprüfer: Joakim Repomaa

Hiermit versichere ich, Marcus Schreiter, ehrenwörtlich, dass ich die vorliegende Bachelorarbeit mit dem Titel: “Möglichkeiten und Grenzen von Webtechnologien bei der Entwicklung von Kiosksoftware anhand der Implementierung einer Sharing-Station für Museen und Ausstellungen” selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Stuttgart, 06.04.2020 Marcus Schreiter

Zusammenfassung/Abstract

Diese Bacherloarbeit untersucht das Entwickeln von Kiosksoftware mit Webtechnologien anhand eines realen Praxisbeispiels. Dabei werden im ersten Schritt die Anforderungen an Kiosksoftware analysiert. Das geschieht nach einer typischen Methode der Systemanalyse und der Unterteilung in funktionale und nicht-funktionale Anforderungen. Anhand dieser wird der Softwarestack strukturiert betrachtet und Lösungen mit Hilfe von Pattern und Software-Bibliotheken entwickelt. Dabei wird im letzten Schritt das Problem der Hardwareanbindung genau betrachtet und eine standardisierte Konvention auf Basis des MQTT/Websocket Protokolls und der Homie Konvention entworfen.
Die Arbeit richtet sich dabei an Softwareentwickler – ein Grundverständnis für typische Software- und Webtechnologien sollte gegeben sein. Nach Lesen der Arbeit soll klar sein, wie bei der Entwicklung von Kiosksoftware mit Webtechnologien vorgegangen werden muss, welche Vor- und Nachteile sie bieten und wie ein typisches Kiosksystem mit Webtechnologien funktionieren kann.

To do (1)

Inhaltsverzeichnis

1. Einleitung

Diese Thesis mit dem Titel *Möglichkeiten und Grenzen von Webtechnologien bei der Entwicklung von Kiosksoftware anhand der Implementierung einer Sharing-Station für Museen und Ausstellungen* untersucht die Softwareentwicklung eines Kiosksystems mit Webtechnologien. Im Rahmen dieser Thesis wurde dafür das Kiosksystem *Sharing-Station* entwickelt. Dieses ging aus einem Projektauftrag der Firma MESO Digital Interiors GmbH (MESO) für das Dialogmuseum Frankfurt hervor. Ebenso ist diese Thesis in Zusammenarbeit mit der Firma MESO entstanden. Die Praxisbetreuung und die Rolle des Zweitprüfers übernahm dabei Joakim Repomaa, welcher Backend-Entwickler bei der Firma MESO ist.

In diesem Kapitel werden die zwei Projektpartner, sowie das entwickelte Produkt kurz vorgestellt.

1.1. MESO

MESO Digital Interiors GmbH ist eine Digitalagentur mit Sitz in Frankfurt am Main. Gegründet wurde sie 1997 und begann damals als Bürogemeinschaft von Programmierern und Designern. Von Beginn an lag der Fokus auf Grafik, Musik, Elektronik, 3D-Visualisierungen und Softwareentwicklung. Kommerzielle Projekte und eigene Projekte wurden gemeinschaftlich umgesetzt.

Über die Jahre entstanden aus dieser Bürogemeinschaft mehrer Firmen: Aspekt1, MESO Web Scapes, MESO Digital Interiors, MESO Image Spaces und MESO Digital Services. Bis zuletzt arbeiteten MESO Digital Interiors, MESO Digital Services und MESO Image Spaces als Firmenkollektiv zusammen, ehe sie im Jahr 2018 schließlich durch die drei Geschäftsführer zu der MESO Digital Interiors GmbH zusammengelegt wurden.
Heute hat MESO rund 30 feste und einige freie Mitarbeiter*innen. Ein Großteil sind Designer oder Informatiker, andere haben einen handwerklichen Hintergrund, wie Schreiner oder Goldschmied. Geführt wird das Unternehmen von Sebastian Oschatz, Max Wolf und Mathias Wollin.
So transdisziplinär wie die Mitarbeiter sind auch die Projekte. MESO konzipiert, gestaltet und entwickelt Ausstellungen, Messeauftritte, Showrooms, Webapplikationen

und Apps. Die Projekte bewegen sich dabei meist an der Schnittstelle zwischen Raum, Kommunikation und Technik. Kunden sind dabei oft Firmen aus der Automobilindustrie (BMW, Merceds-Benz, Yanfeng oder Moovel), aus der Technologiebranche (HERE Technologies, Siemens, Keyence) oder Institutionen öffentlicher Träger wie Museen, Hochschulen oder Städte (Hochschule Mainz, Senckenberg Museum, Stadthalle Karlsruhe).

Bei Medieninstallationen und Echtzeit 3D-Grafik arbeitet MESO weitestgehend mit dem eigenentwickelten Tool VVVV [1] und mit der Spiel-Engine Unreal [2]. Für Backend-Systeme, Apps, Interfaces und Applikationen werden meist Webtechnologien eingesetzt.

1.2. Dialogmuseum

Das Dialogmuseum Frankfurt existiert seit 2005 und ist ein privates, soziales Unternehmen. Es hat sich zum Ziel gesetzt informativ, integrativ und wirtschaftlich mit den Themen Blindheit und Sehbehinderung umzugehen. Dabei ist es ein Ort für sehende, blinde und sehbehinderte Menschen gleichermaßen [3]. Allein im Jahr 2017 zählte das Museum 53.000 Besucher*innen [4].

Das Ausstellungskonzept beruht auf dem 1988 von Prof. Dr. Andreas Heinecke entwickelten *Dialog im Dunkeln*. Dabei werden Besucher*innen von blinden Menschen durch eine völlig im Dunkeln gehaltene Ausstellung geführt. Dabei erleben sie Alltagssituationen wie Szenen im Park, der Stadt oder einer Bar – müssen sich dabei jedoch einzig auf den Guide und ihre übrigen Sinne verlassen [3].
Das Ausstellungskonzept ist mittlerweile ein weltweites Franchisesystem, welches vom Dachverband Dialogue Social Enterprise GmbH (DSE) [5] mit Sitz in Hamburg verwaltet wird. Zu Dialog im Dunkeln sind weitere Konzepte wie *Dialog im Stillen* und Business Workshops hinzugekommen. Diese schaffen weltweit Begegnungen integrativer Art und geben dabei Menschen mit Behinderung einen Arbeitsplatz.
Dialog im Dunkeln zählt seit 1989 weltweit rund 6,5 Millionen Besucher und Besucherinnen [6].

Das Museum in Frankfurt ist seit Ende 2018 geschlossen und soll Anfang Mai 2020 in neuen Räumen, in der B-Ebene der Haltestelle Frankfurter Hauptwache, wiedereröffnet werden. Während das Ausstellungskonzept weitestgehend gleich bleibt, soll der Foyerbereich, sowie der Raum zu dem die Besucher zum Abschluss der Führung gelangen, neu gestaltet werden. Dabei bekam MESO den Projektauftrag für zwei digitale Exponate. Zum einen soll ein digitales, barrierefreies Gästebuch entstehen, sowie die Sharing-Station, welche Gegenstand dieser Thesis ist. Das Gästebuch soll sich dabei später in dem genannten

Reflexionsraum am Ende der Ausstellung und die Sharing-Station im Foyer des neuen Museums befinden.

1.3. Sharing-Station

Das im Rahmen dieser Thesis entwickelte Produkt trägt den Namen Sharing-Station. Der Name ist eine interne Bezeichnung und soll die zwei Haupteigenschaften des Produkts zusammenbringen. *Sharing*, da das Produkt in der Hauptaufgabe Informationen mit Benutzer*innen teilen soll. Und *Station* da es sich um ein Kiosksystem handelt.

Das Produkt besteht dabei aus der Software und mehreren Hardwarekomponenten. In der Hauptsache wird sich diese Arbeit mit der Softwareentwicklung des Produkts beschäftigen. In Kapitel 6 wird zwar auf die Hardwareanbindung eingegangen, jedoch liegt auch hier der Fokus auf der Softwarekommunikation.
Zu erwähnen sei auch, dass der Autor sich in der Entwicklung des Produkts hauptsächlich um die Softwareentwicklung gekümmert hat. Produkt-, Screen- und Hardwaredesign sowie das Projektmanagement wurden von anderen Mitarbeiter*innen der Firma MESO übernommen.

Ausgangspunkt für die Idee der Sharing-Station war der Umzug und die damit geplante Neueröffnung des Dialogmuseums. Es entstand der Wunsch, im Foyer des Museums den Besucher*innen die Möglichkeit zu geben, sich zum einen über das Museum und den Dachverband, sowie über andere ähnliche Projekte zu informieren. Auch sollte es die Möglichkeit geben sich aktiv zu engagieren und zu beteiligen, beispielsweise durch Spenden oder Eintragen in einen Newsletter. Weiter gab es den Wunsch, den Besucher*innen die Möglichkeit zu geben ein öffentliches soziales Commitment zu hinterlassen.
Schon früh war die Idee da, diese Anforderungen nicht beispielsweise durch Drucken und Auslegen von Broschüren und Flyern, sowie Aufstellen einer Spendenbox zu erfüllen, sondern das Ganze in einer digitalen Form anzubieten. So entstand die Idee der Sharing-Station: Ein Terminal mit großem Touchscreen, an dem die Besucher*innen all die zuvor beschriebenen Dinge tun können und welches durch seine digitale Form in seinem Funktionsumfang beliebig erweiterbar ist.

1.3.1. Sharing-Station als Produkt

Von Anfang an wurde die Software der Sharing-Station unter den Gesichtspunkten der Adaptier- und Wiederverwendbarkeit entwickelt. Schon früh wurde von MESO erkannt, dass die Anforderungen, welche das Dialogmuseum an die Sharing-Station hat, Anforderungen sind, die in der Praxis häufig von Museen und Ausstellungsmachern

gestellt werden. Die Forderung nach Alternativen zu klassischen Ausstellungsheften, Flyern und Newsletter-Listen, wird gerade im Zuge von Digitalisierung im Museums- und Ausstellungsbetrieb sehr häufig gestellt. So entstand die Idee, die Sharing-Station für das Dialogmuseum so zu entwickeln, dass sie später als Produkt auch anderen Kunden angeboten werden kann.
Schon während der Entwicklungszeit gab es erstes Interesse innerhalb des Dialogue Social Enterprise GmbH Verbands, die Sharing-Station in anderen Museen einzusetzen.

2. Grundlagen

In diesem Kapitel werden die zwei zentralen Begriffe dieser Thesis, *Kiosksystem* und *Webtechnologien*, erklärt und eingeordnet. Da der Begriff Kiosksystem, auch unter Softwareentwicklern, nicht unbedingt geläufig ist oder zumindest nich ausreichend bekannt, wird er an dieser Stelle in seiner Wortherkunft und Wortbedeutung ausführlich erläutert. Der Begriff der Webtechnologien wiederum, sollte für die meisten Leser*innen bekannt sein. Trotzdem wird er an dieser Stelle kurz erklärt, sowie die Bedeutung in der Entwicklung über die Zeit und im Zusammenhang mit dieser Arbeit analysiert.

2.1. Kiosksysteme

Wenn man in der Informationstechnologie von Kiosksystemen spricht, sind damit meist zugängliche Computersysteme gemeint, die im öffentlichen oder halböffentlichen Raum platziert sind. Zudem besitzen sie eine Benutzerschnittstelle – sehr oft in Form eines Touchscreens. Sie bieten dabei in der Regel Zugang zu Informationen oder elektronischen Transaktionen [7].

Der Begriff *Kiosk* hat seinen Ursprung im Persischen und steht dort für ein zeltartiges Gartenhaus oder eine Art Erker an orientalischen Palästen [8]. Allgemeiner steht er in der islamischen Baukunst für einen pavillonähnlichen Bau [9]. In der heutigen, allgemeinsprachlichen Definition, versteht man unter einem Kiosk eine Verkaufsstelle für Zeitungen und Zeitschriften [9, 8].
In der Bedeutung hat unsere heutige Definition dabei die äußere Form, das Pavillonähnliche, des ursprünglichen Kiosks übernommen. Oft sind Kioske alleinstehende Häuschen, die in ihrer Form an einen Pavillon erinnern. Hinzugekommen zur Bedeutung ist das Öffentlichzugängliche**To do** (2). Während der Kiosk in seiner ursprünglichen Wortbedeutung für etwas steht was meist an einen Palast angegliedert ist – und somit vermutlich nur beschränkt zugänglich – verstehen wir heute unter dem Begriff einen Ort, dessen Zugang für jeden Menschen gedacht ist. Darüber hinaus ist er in der Regel an belebten und gut zugänglichen Orten wie Straßen, Plätze und Sehenswürdigkeiten platziert [10].
Dieser Teil der Wortbedeutung hat sich auf das informationstechnische Kiosksystem übertragen. Mit dem pavillonähnlichen Bau hat es nichts mehr zu tun, dafür aber mit der Tatsache, dass es etwas Zugängliches im öffentlichen Raum darstellt [10].

Holfelder [10] beschreibt weitere Parallelen: die Art der Kunden und die Verweildauer. Zum einen gibt es die Laufkundschaft, die durch optische oder akustische Reize zum Herantreten animiert werden, sowie die Kunden die gezielt und mit einer bestimmten Absicht an den Kiosk oder das Kiosksystem herantreten. Die Verweildauer des Kunden ist in beiden Szenarien kurz, vergleicht man den Besuch in einem Kiosk mit dem in einem Kaufhaus, oder das Benutzen eines Kiosksystems mit dem Benutzen der eigenen elektronischen Geräte [10].
Kiosksysteme sind bekannte Systeme. Fasst man den Begriff weit, so ist beispielsweise auch der Geldautomat ein Kiosksystem. Aber auch Ticketautomaten oder der Self-Ordering Kiosk, wie ihn McDonalds 2011 in Europa eingeführt hat [11], sind bekannte Systeme und zum Teil nicht mehr wegzudenken. Oft trifft man in Geschäften oder öffentlichen Gebäuden auf Kiosksysteme, die spezifische Informationen und Transaktionen bereitstellen. Denkbar wäre ein Ausleih-Kiosk in einer Bibliothek oder ein Kiosk in einem Kaufhaus, welcher einen Lageplan und Informationen über die Geschäfte bereithält.
Dabei zeigt sich auch: Kiosksysteme können sehr unterschiedliche Zwecke haben. Borchers, Deussen und Knörzer [12] klassifizieren daher Kiosksysteme in vier Kategorien:

1. Informations-Kioske
2. Werbe-Kioske
3. Service-Kioske
4. Entertainment-Kioske

Mischformen sind denkbar und üblich. Tatsächlich kommen Kiosksysteme die nur in eine der genannten Kategorien fallen eher selten vor.
Der Informations-Kiosk hat die Aufgabe kontextbezogene Informationen bereitzustellen. Benutzer*innen sind motiviert und gehen eher zielgerichtet vor. Der Werbe-Kiosk hat die Aufgabe eine Firma oder ein Produkt in der Öffentlichkeit zu bewerben. Benutzer*innen müssen animiert werden das System zu nutzen und werden beispielsweise durch ein ansprechendes Design motiviert. Der Service-Kiosk ist ähnlich dem Informations-Kiosk. Zusätzlich können Benutzer*innen Transaktionen über eine Input-Schnittstelle tätigen. Beispielsweise ein Ticket kaufen. Der Entertainment-Kiosk hat außer der Unterhaltung der Benutzer*innen keine weitere Aufgabe. Denkbar wäre ein solches System zum Beispiel in einem Wartebereich.

2.2. Webtechnologien

Mit der Erfindung des World Wide Web durch Tim Berners-Lee und Robert Cailliau und damit auch der Erfindung von HTML [13], entstand auch der Begriff der Webtechnologien.

Der Begriff umfasst also all die Technologien, die beteiligt sind um eine Webseite von einem Server über ein Netz an einen Client zu übertragen und anzuzeigen.

Im Rahmen dieser Arbeit soll der Begriff sich weiter nur auf die Softwaretechnologien beschränken. Dabei lassen sich die Softwaretechnologien klassischerweise in clientseitige und serverseitige Technologien einteilen.
Während sich serverseitig allerhand Programmiersprachen und Technologien finden, sind seit jeher die Softwaretechnologien auf der Clientseite im Kern auf HTML, CSS und JavaScript beschränkt. Dabei markiert JavaScript als neueste dieser drei Technologien [14] den Beginn der Rich-Client Applikationen und bietet die Möglichkeit im Browser vollständige, interaktive Applikationen statt nur statische Seiten anzuzeigen.
Spätestens mit der Entwicklung und Quellcode-Veröffentlichung der JavaScript-Engine V8 [15] beschränkt sich die Nutzung von JavaScript auch nicht mehr nur auf den Browser. V8 ist ein in C++ geschriebener JavaScript-Interpreter, **To do** (3) welcher Standalone oder eingebunden in einem C++ Programm genutzt werden kann.
Auf der V8-Engine beruht beispielsweise die asynchrone JavaScript Laufzeitumgebung Node.js, mit der in JavaScript geschriebene Programm direkt ausgeführt werden können [16]. Genutzt wird Node.js so meist für Serverapplikationen oder serverseitige Microservices.
Auf Node.js und Chromium wiederum basiert das Framework Electron [17]. Mit diesem können Desktop-Anwendungen entwickelt werden. Dabei werden Browsertechnologien, sowie serverseitiger JavaScript-Code gleichermaßen genutzt um vollständige Anwendungen zu implementieren.

Es zeigt sich, dass der Begriff *Webtechnologien* nicht mehr nur in der ursprünglichen Definition gesehen werden kann, sondern weiter fasst. Vielmehr sind Webtechnologien heute ein Toolkit, welche in vielen Situationen eingesetzt werden können. Um die Webtechnologien weiter einzuschränken, soll daher der Begriff der *Browsertechnologien* eingeführt werden. Er umfasst nur die clientseitigen Technologien, also HTML, CSS und clientseitigen JavaScript-Code, welcher in erster Linie zur DOM-Manipulation eingesetzt wird.
To do (4)

3. Anforderungsanalyse

Nun sind wir am Ausgangspunkt der Arbeit angelangt: Es soll die Software für ein Kiosksystem entwickelt werden. Im Rahmen dieser Arbeit und am Beispiel der Sharing-Station geschieht dies alleinig mit Webtechnologien. Um die Möglichkeiten und Grenzen dieser Technologien strukturiert und genau prüfen zu können, werden in diesem Kapitel die Anforderungen an das System und an die Software des Systems analysiert. Dies ist auch nötig, da die Webtechnologien nun in einem neuen Kontext stehen. Anforderungen an diese Technologien in einem klassischen Webentwicklungskontext sind klarer und bekannter. Im Kontext des Kiosksystems bedarf es einer neuen und veränderten Betrachtung.

Nach einer üblichen Methode der Systemanalyse, werden die Anforderungen in funktionale und nicht-funktionale Anforderungen unterteilt. Dabei definieren funktionale Anforderungen die reine Funktionalität und geben Antworten auf die Frage "Was soll das System machen?". Nicht-funktionale Anforderungen bilden die restlichen Anforderungen ab, und definieren dabei Anforderungen an die Eigenschaften und Qualitäten der Funktionen [18]. Sie ergänzen dabei die funktionalen Anforderungen [19]. Nicht-funktionale Anforderungen werden oft noch weiter unterschieden. Beispielsweise in Qualitätsanforderungen und Randbedingungen [18, 19]. Da sich der Anforderungsumfang in dieser Arbeit in Maßen hält, wird diese Unterscheidung im Folgenden nicht vorgenommen.
Funktionale Anforderungen sind immer sehr spezifisch, da sie die konkreten Features und Funktionalitäten definieren. So beschreiben auch hier die funktionalen Anforderungen die konkreten Eigenschaften der Sharing-Station, und geben dabei einen guten Überblick über den geplanten Funktionsumfang. Nicht-funktionale Anforderungen hingegen sind generischer und lassen sich leichter auf andere Projekte übertragen. So sind die im Folgenden beschriebenen nicht-funktionalen Anforderungen, Anforderungen, die auch typisch für Kiosksoftware im Allgemeinen sind. Sie sind für die Zielsetzung dieser Arbeit also von größerer Bedeutung.
Beim Definieren der Anforderungen beschränkt sich diese Arbeit auf die Anforderungen an die Software. Dabei stellen manche Anforderungen auch Ansprüche an andere Teile des Systems. Diese Überschneidung lässt sich nicht vermeiden, denn gerade die funktionalen Anforderungen stellen meist auch Anforderungen an das Gesamtsystem.

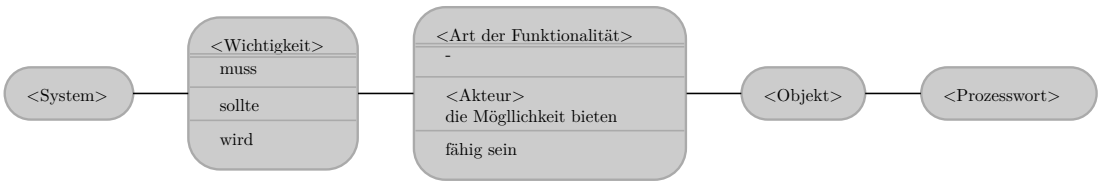

Abbildung 3.1.: Schablone für funktionale Anforderungen nach Rupp und SOPHISTen [20]

3.1. Funktionale Anforderungen

Alle Anforderungen sind in natürlicher Sprache verfasst und folgen dabei einer bestimmten Syntax. Diese Syntax folgt den Schablonenregeln von Rupp und SOPHISTen [20]. Abbildung 3.1 zeigt die Schablone für die Formulierung einer funktionalen Anforderung. *System* entspricht dem Subjekt**To do** (5), z.B. die Sharing-Station oder die Clientsoftware. Anschließend wird die *Wichtigkeit* festgelegt. Das Schlüsselwort *muss* bedeutet, eine Anforderung ist im rechtlichen Sinne verpflichtend. *Sollte* beschreibt eine Anforderung, welche nicht verpflichtend ist aber die Zufriedenheit erhöht. Und *wird* wird verwendet, um Anforderungen zu definieren, welche in der Zukunft erst integriert werden.
Die *Art der Funktionalität* beschreibt die unterschiedlichen Systemaktivitäten. Wird hier kein Schlüsselwort eingesetzt, so beschreibt die Anforderung eine selbständige Aktivität. Der Satzteil *die Möglichkeit bieten* setzt immer eine Benutzerinteraktion voraus. *Fähig sein* definiert eine Schnittstellenanforderung.
Schließlich folgt das *Objekt*, für welches die Funktionalität gefordert wird und das so genannte *Prozesswort*. Das Prozesswort ist das Verb des Satzes, welches die Funktionalität identifiziert.
Gilt eine Anforderung nur unter einer bestimmten Bedingung, wird diese dem Wichtigkeit-Schlüsselwort vorausgestellt und das System-Schlüsselwort wird vor der Art der Funktionalität platziert. Ein Beispiel hierfür ist **FA8**.

Den folgenden Anforderungen folgt jeweils eine kurze Erklärung, welche nicht der eben beschriebenen Syntax folgt, sondern lediglich die Anforderung weiter ausführen soll.

FA1 Die Sharing-Station muss den Besucher*innen die Möglichkeit bieten, Informationen über das Museum zu erhalten.
Diese Informationen sollen dabei in Artikelform und zu unterschiedlichen Themen abrufbar sein. Ein Menü soll Überblick über die verschiedenen Artikel geben.

FA2 Die Sharing-Station muss den Besucher*innen die Möglichkeit bieten, Informationen über andere Projekte zu erhalten.
Analog zu den Informationen über das Museum, sollen Informationen über andere, ähnliche Projekte abrufbar sein. Diese sollen dabei in einem eigenen Menü organisiert sein.

FA3 Die Sharing-Station muss den Besucher*innen die Möglichkeit bieten, eine Spende zu hinterlassen.
Das Spenden soll dabei kontaktlos per Kreditkarte, EC-Karte und Mobile Payment oder mit Bargeld, in Form eines Münzeinwurfs, möglich sein.

FA4 Die Sharing-Station muss den Besucher*innen die Möglichkeit bieten, ein Foto-Commitment zu hinterlassen.
Diese Funktion soll ähnlich wie eine Fotobox funktionieren. Eine an das System angeschlossene Webcam, soll die Möglichkeit bieten ein Foto zu machen und dieses in einer Galerie zu speichern oder sich selbst per Mail zu schicken. Vor Ort sollen außerdem kleine Tafeln ausgelegt werden auf denen Besucher*innen ein soziales Commitment schreiben und dann auf dem Foto präsentieren können.

FA5 Die Sharing-Station sollte den Besucher*innen die Möglichkeit bieten, sich für einen Newsletter einzutragen.
Das Eintragen soll über ein Formular und eine Bildschirmtastatur möglich sein. Der Request soll dabei an das Wordpress-basierte Newslettersystem des Museums erfolgen.

FA6 Die Sharing-Station sollte den Besucher*innen die Möglichkeit bieten, Informationen mitzunehmen.
Am Ende von Artikeln sollen weiterführende Informationen und Verweise auf Webseiten per QR-Code verlinkt sein. Besucher*innen können diese Informationen so auf dem eigenen Geräten lesen und speichern.

FA7 Die Sharing-Station sollte den Mitarbeiter*innen des Museums die Möglichkeit bieten, die Inhalte über ein Content Management System (CMS) zu verwalten.
Die Inhalte, gerade die Artikel über das Museum und andere Projekte, sollen redaktionell verwaltbar sein. Dies soll über ein intuitiv zu bedienendes CMS möglich sein.

FA8 Nach einer gewissen Zeit ohne Benutzerinteraktion, sollte die Oberfläche der Clientsoftware in einen Idle-Modus wechseln.
Dieser Modus soll wie ein Bildschirmschoner wirken. Eine Animation soll dabei

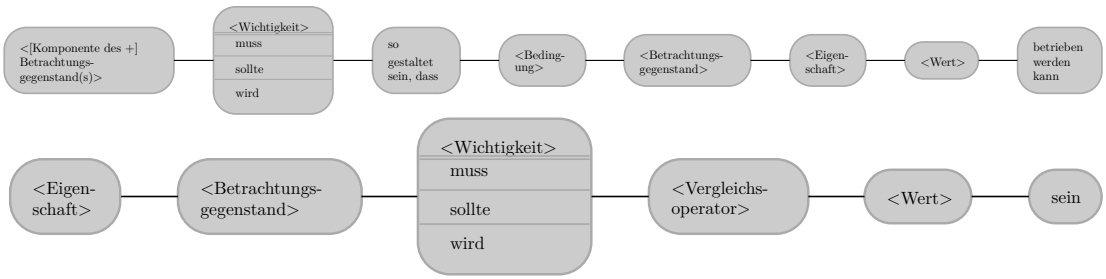

Abbildung 3.2.: Umgebungs- und Eigenschaftsschablone für nicht-funktionale Anforderungen nach Rupp und SOPHISTen [20]

die Oberfläche überdecken und mit Schlagwörtern und bewegten Formen die Besucher*innen zur Interaktion animieren.

3.2. Nicht-funktionale Anforderungen

Die Formulierung nicht-funktionaler Anforderungen anhand einer Schablone fällt schwer. Nicht-funktionale Anforderungen sind in der Literatur nicht einheitlich definiert und in ihrer Art unschärfer als funktionale Anforderungen. Trotzdem schlagen Rupp und SOPHISTen [20] drei verschiedene Schablonen für die Formulierung vor: Die Umgebungs-, Eigenschafts- und Prozessschablone. Abbildung 3.2 zeigt die ersten beiden dieser Schablonen. Sie ähneln dabei der Schablone in Abbildung 3.1. Wieder spielt das Wichtigkeit-Schlüsselwort eine große Rolle. Anders als vorher, ist das Subjekt nun nicht mehr *System* sondern *Betrachtungsgegenstand*, welches im Falle der Umgebungsschablone im Hauptsatz auf einen Teilbestandteil beschränkt werden kann (*[Komponente des +]*). Die Umgebungsschablone bildet Anforderungen ab, die von der Umgebung des Betrachtungsgegenstands abhängig sind. **NFA1** und **NFA2** sind nach dieser Schablone gebildet. **NFA2** beispielsweise fordert, dass die Software auch dann funktioniert, wenn das Netz ausfällt in dem sich diese befindet. Das Netz ist dabei Teil der Umgebung und nicht Teil des Betrachtungsgegenstands.
Alle weiteren Anforderungen folgen grob der Struktur der Eigenschaftsschablone. Eigentlich fordert die Schablone immer einen definierten Wert für die beschriebene Eigenschaft, allerdings fordern nicht-funktionale Anforderungen auch oft nur die reine Existenz einer Eigenschaft, wie z.B. in **NFA3**. In solchen Fällen schiebt sich die Eigenschaft hinter das Wichtigkeit-Schlüsselwort und der Vergleichsoperator und der Wert werden weggelassen.
Die dritte Schablone nach Rupp und SOPHISTen [20], die Prozessschablone, ist in Abbildung 3.2 nicht abgebildet. Sie stellt Anforderungen an Akteure, z.B. an den

Auftragnehmer, und nicht an das System. Da im Falle dieser Arbeit nur Anforderungen an die Softwaretechnologien betrachtet werden, findet diese Schablone keine Anwendung und wird nicht weiter erläutert.

Wie schon bei den funktionalen Anforderungen, folgt auch hier jeder Anforderung eine frei formulierte Erklärung. Diese bezieht sich immer zuerst auf Kiosksoftware und Kiosksysteme im Allgemeinen und erklärt dann die Bedeutung der Anforderung für die Sharing-Station.

NFA1 Die Clientsoftware muss so gestaltet sein, dass sie zusammen mit Hardwarekomponenten betrieben werden kann.
Für viele Kiosksysteme sind angeschlossene Hardwarekomponenten, als Nutzerschnittstelle oder Ausgabegeräte, essenziell. Gerade Servie-Kioske sind meist ohne ihre Hardwarekomponenten nicht zu benutzen. Das wohl eindrücklichste Beispiel hierfür ist der Ticketautomat: Ohne Kreditkartenterminal und Ticketdrucker wäre dieses Geräte nicht denkbar.
Auch die Sharing-Station soll Hardwarekomponenten beinhalten. Für die Foto-Commitment-Funktion (**FA4**) wird eine Kamera und für die Spenden-Funktion (**FA3**) ein Kreditkartenterminal und ein Münzeinwurf benötigt. Alle Geräte müssen mit der Clientsoftware kommunizieren können um eine Steuerung und ein Feedback über die Oberfläche zu gewährleisten.

NFA2 Die Clientsoftware sollte so gestaltet sein, dass sie auch offline betrieben werden kann.
Die Bedienbarkeit der Clientsoftware sollte auch dann gewährleistet werden, wenn das Netz zwischenzeitlich ausfällt oder hoch belastet ist. Gerade Werbe- und Informations-Kioske sind sehr darauf bedacht, ihre Nutzer möglichst lange am Gerät zu halten [12]. Ein zu langer Ladevorgang, durch welchen die Oberfläche zwischenzeitlich nicht bedienbar ist, würde dazu führen, dass viele Nutzer*innen sich vom Gerät entfernen. Dies soll verhindert werden.
Bei der Sharing-Station soll das Navigieren durch die Oberfläche und das Ausführen möglichst aller Funktionen auch offline möglich sein. Lediglich Transaktionen, bei denen die Nutzer*innen Daten abschicken, wie Speichern eines Fotos oder Eintragen für den Newsletter, sollen auch nur dann positives Feedback geben, wenn diese erfolgreich über das Netz erfolgt sind. Bei Nichterreichen eines Servers oder Services, soll eine entsprechende Fehlermeldung angezeigt werden, welche die Nutzer*innnen darauf hinweist, es später noch einmal zu versuchen.

NFA3 Die Oberfläche der Clientsoftware muss multilingual sein.
Durch die meist öffentliche oder halböffentliche Platzierung von Kiosksystemen, ste-

hen diese immer einer großen Anzahl unterschiedlicher Nutzer*innen zur Verfügung. Um möglichst vielen ein optimales Erlebnis zu gewährleisten, ist die Möglichkeit einer Spracheinstellung von großer Wichtigkeit.
Auf Grund der typischen Besucher*innen und den finanziellen Möglichkeiten des Museums, werden die Spracheinstellungen der Sharing-Station auf Deutsch und Englisch beschränkt.

NFA4 Die Oberfläche der Clientsoftware muss in sich geschlossen sein.
Diese Anforderung ist typisch für Kiosksoftware. Sie bedeutet, dass die Oberfläche der Anwendung von Benutzer*innen nicht verlassen werden kann. Sie soll den Eindruck eines geschlossenen Systems vermitteln. Dies ist zum eine aus der User-Experience Sicht wünschenswert aber auch aus einer sicherheitstechnischen Sicht relevant. Den Benutzer*innen soll keine Möglichkeit geboten werden Schaden auf dem System anzurichten oder andere Webseiten im Netz zu erreichen.
Diese Anforderung gilt genauso für die Sharing-Station. Den Nutzer*innen soll keine Möglichkeit geboten werden, die Oberfläche der Clientsoftware zu verlassen. Auch soll nicht der Eindruck einer Webseite entstehen. Typische Browser-Funktionalitäten müssen also deaktiviert werden, so dass der Eindruck einer in sich geschlossenen Anwendung entsteht.

NFA5 Die Clientsoftware sollte plattformunabhängig sein.
Diese Anforderung ist nicht unbedingt typisch für Kiosksoftware im Allgemeinen – bei Kiosksoftware im Ausstellungs- und Messebetrieb wird sie jedoch häufig gestellt. Beispielsweise soll eine Software bei einer Messe auf einem großen Touchscreen und bei der nächsten auf mehreren iPads laufen. Oder eine Archivsoftware, welche als Kiosksystem in einer Ausstellung betrieben wird, soll nach der Ausstellung online zur Verfügung gestellt werden. Die Software muss also so entwickelt werden, dass sie leicht auf ein anderes System übertragen werden kann.
Wie in Unterabschnitt 1.3.1 bereits erwähnt, soll die Sharing-Station als Produkt gedacht und unter den Gesichtspunkten der Adaptier- und Wiederverwendbarkeit entwickelt werden. Dazu zählt auch, dass die Software auf verschiedenen Plattformen lauffähig ist, da zukünftige Kunden die Anforderung nach einer anderen Plattform haben könnten.

NFA6 Das Deployment der Clientsoftware sollte möglichst einfach und von außerhalb möglich sein.
Bei Kiosksystemen ist es meist von Vorteil, wenn Softwareupdates nicht vor Ort am Gerät eingespielt werden müssen, sondern von außerhalb erfolgen können.
Bei der Sharing-Station ist dies keine vom Kunden geforderte Anforderung. Allerdings zeigt die Erfahrung, dass nachträgliche Anpassung und Fehlerbehebung an der

Software in den meisten Fällen nötig ist. Eine entsprechende Deployment-Strategie ist also von großem Vorteil.

NFA7 Die Software sollte modular und erweiterbar gehalten sein.
Auch diese Anforderung ist nicht unbedingt typisch für Kiosksoftware aber dennoch ist es denkbar, dass sie häufig gestellt wird. Eine modulare Software ist oft von Vorteil, auch wenn sie möglicherweise längere Zeit in der Entwicklung benötigt. So können Teile der Software leichter gegen andere ersetzt oder neue zum System hinzugefügt werden.
Im Falle der Sharing-Station ist dies eine gewünschte Anforderung. Wie in Unterabschnitt 1.3.1 beschrieben, soll die Sharing-Station zukünftig als Produkt angeboten werden und muss so in Teilen ersetzbar und leicht um weitere Funktionalitäten erweiterbar sein.

NFA8 Das CMS muss über das Internet aufrufbar sein.
Ebenso wie bei dem Deployment ist es auch bei der Inhaltserstellung und -pflege von Vorteil wenn diese bei Kiosksystemen nicht vor Ort am Gerät oder im lokalen Netz erfolgen muss, sondern von außerhalb gemacht werden kann. Das CMS ist dafür am besten ein über das Internet erreichbarer Dienst.
Diese Anforderung gilt genauso für die Sharing-Station. Alleine schon da die Büroräume des Kunden sich an einem anderen Ort wie das Museum befinden, ist dies eine verpflichtende Anforderung.

4. Systemarchitektur

Für die weitere Betrachtung der Softwaretechnologien ist es erst nötig einen Überblick über das Gesamtsystem der Sharing-Station zu erhalten, um zu verstehen, wie die einzelnen Komponenten zusammenarbeiten.
Dafür zeigt das Diagram in Abbildung 4.1 die Topologie des Gesamtsystems. Es soll dabei Überblick über alle im System enthaltenen Hardwarekomponenten und Applikationen geben. Zusätzlich werden die Netzstruktur und die Kommunikationswege zwischen den Komponenten aufgezeigt.

Bei dem Diagram handelt es sich um ein UML Deployment Diagram [21]. Es ist dafür gedacht, um die Systemarchitektur und die Verteilung von Hardware- und Softwarekomponenten in der Umgebung des Systems zu visualisieren [22]. Hardwarekomponenten werden dabei als Geräteknoten (*device*) dargestellt. Auf ihnen können Softwarekomponenten verteilt werden (*service, application, database*) [23, 22]. Die Verbindungen zwischen Knoten und Komponenten sind eine spezielle Art der Assoziation und stellen in diesem Fall Kommunikationspfade dar [22]. Die Beschriftung an den Wegen nennt dabei das genutzte Kommunikationsprotokoll. Die umschließenden Rahmen zeigen welche Komponenten sich in welchem Netz befinden.
Im Mittelpunkt der Architektur steht die Kiosk-Workstation – der Ausstellungsrechner. Auf ihm läuft die Clientapplikation (*Sharing Station App*). An die Workstation direkt angeschlossen, und mit der Applikation über die USB-Schnittstelle kommunizierend, ist die Webcam, welche für die Foto-Commitment Funktion (**FA4**) benötigt wird. Die zwei weiteren Hardwarekomponenten *Card Terminal* und *Coin Slot* (**FA3**) sind nicht an die Kiosk-Workstation, sondern über serielle Kommunikation an einen Mikrocontroller (*Embedded-Client*) angeschlossen. Dieser kommuniziert über den *MQTT Broker* mit der Clientapplikation. Die Hintergründe zu dieser Architektur werden in Kapitel 6 erläutert.
Alle eben beschriebenen Komponenten befinden sich im lokalen Netzwerk des Museums. Darüber hinaus gibt es noch die Dienste, welche über das Internet erreichbar sind. Im Zentrum steht hier der Server. Bei diesem handelt es sich um einen Cloud-Server mit Root-Zugriff. Auf diesem ist das CMS zusammen mit der Datenbank gehostet. So ist das CMS über eine Weboberfläche im Internet erreichbar und erfüllt damit die Anforderung **NFA8**. Neben dem CMS und der Datenbank ist auf dem Server noch ein Webserver gehostet, welcher die Applikationsfiles der *Sharing Station App* ausliefert. Die Hintergründe

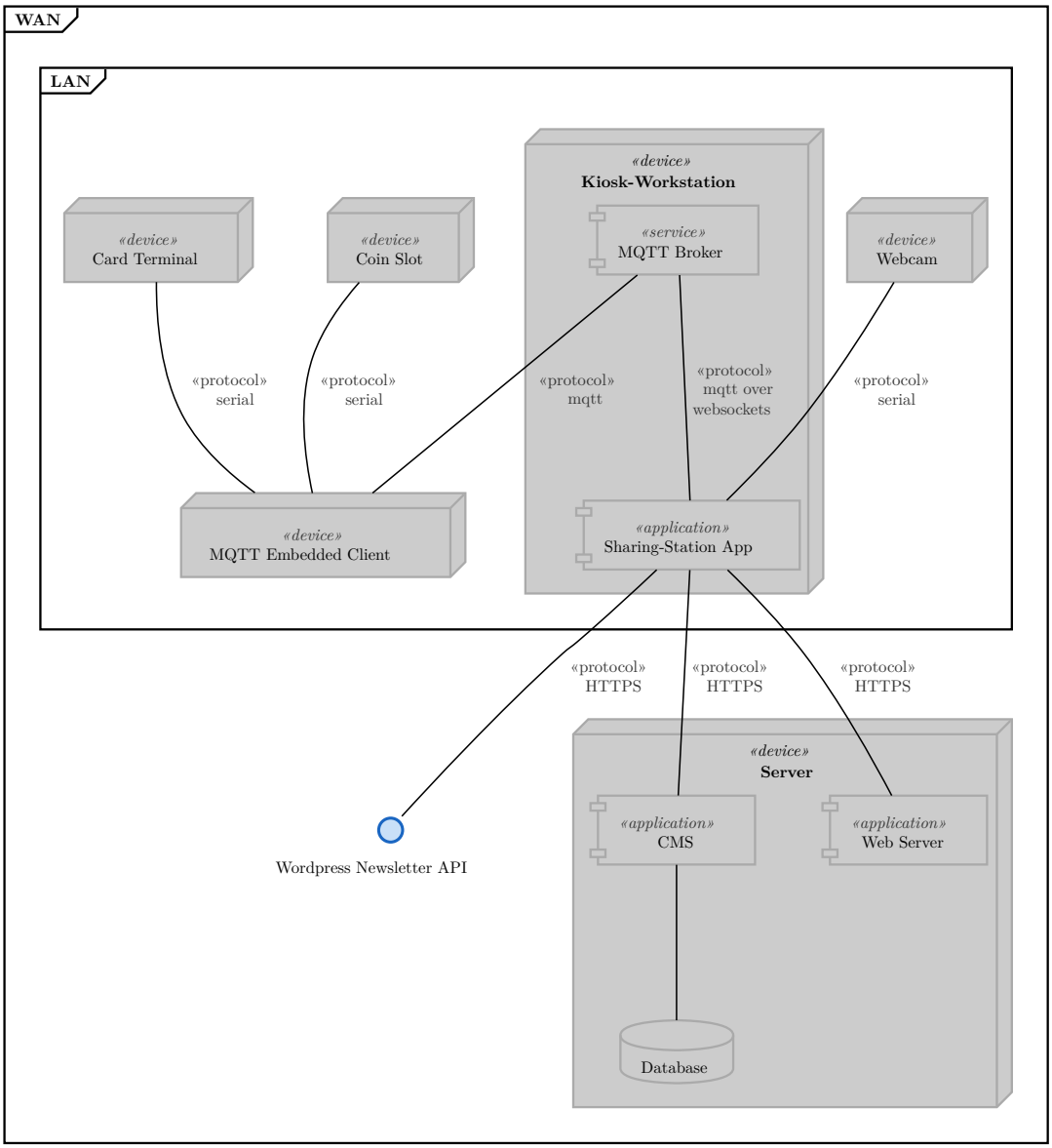

Abbildung 4.1.: Deployment Diagram der Sharing-Station

hierzu werden in Unterabschnitt 5.2.3 beschrieben. CMS sowie Webserver kommunizieren über das HTTPS-Protokoll mit der Clientapplikation.
Und schließlich gibt es noch den Wordpress-Server des Dialogmuseums. Dessen Architektur ist nicht bekannt, aber auch nicht weiter von Bedeutung. Lediglich die Schnittstelle zum Eintragen einer E-Mail Adresse in die Newsletterliste ist relevant. Bei dieser handelt es sich um einen einzelnen Endpoint, welcher ebenfalls über das HTTPS-Protokoll angesprochen werden kann.

5. Softwarestack

In diesem Kapitel wird der webtechnologiebasierte Softwarestack genau betrachtet. Ausgehend von den in Abschnitt 3.2 definierten nicht-funktionalen Anforderungen, werden Softwaretechnologien und Architektur-Pattern besprochen, welche diesen Anforderungen gerecht werden sollen. Dies geschieht beispielhaft an den verwendeten Technologien der Sharing-Station. Ingesamt stehen hierbei die Client-Technologien im Fokus – die Themen CMS und Datenbank werden nur kurz behandelt.

5.1. CMS und Datenbank

Die Anforderungen an CMS und Datenbanken für Kiosksysteme können sehr unterschiedlich sein. Darüberhinaus gibt es für CMS eine große Auswahl an kommerziellen und freien Softwareprodukten. Nach der Anforderungsanalyse aus Kapitel 3 muss das CMS den Anforderungen **NFA8**, **NFA7** und **NFA3** gerecht werden.

NFA3 stellt die Forderung der Multilingualität an die Oberfläche der Clientsoftware. Um dies zu erfüllen, müssen Inhalte mehrsprachig verfügbar sein, also muss auch das CMS die Möglichkeit bieten Datenobjekte mehrsprachig anzulegen.
NFA7 stellt die Forderung der Modularität an die Software. Dabei spielt auch die Kopplung zwischen CMS und Clientapplikation eine Rolle. Diese sollte möglichst gering sein. Eine geringe Kopplung wird erreicht, wenn das CMS nicht für die Darstellung der Daten verantwortlich ist, sondern diese nur über eine Schnittstelle zur Verfügung stellt. CMS dieser Art besitzen also keine Präsentationsschicht und werden deshalb als *headless* bezeichnet [24]. Die Darstellung der Daten findet alleinig in der Clientapplikation statt. Und schließlich fordert **NFA8**, dass das CMS über das Internet erreichbar ist. Das CMS sollte also selbst eine Webanwendungen sein, welche auf einem über das Internet erreichbarem Server gehostet ist.

All diese Anforderungen werden von vielen auf dem Markt verfügbaren CMS-Systemen erfüllt und sind daher weniger außergewöhnlich. Beispielsweise besitzt auch das sehr verbreitete CMS *Wordpress* standardmäßig eine REST-API, über die Daten abgefragt werden können [25]. Zusammen mit einem Plugin wie *WP Headless* [26], welches die Präsentationsschicht von Wordpress entfernt, kann es so ebenfalls headless betrieben

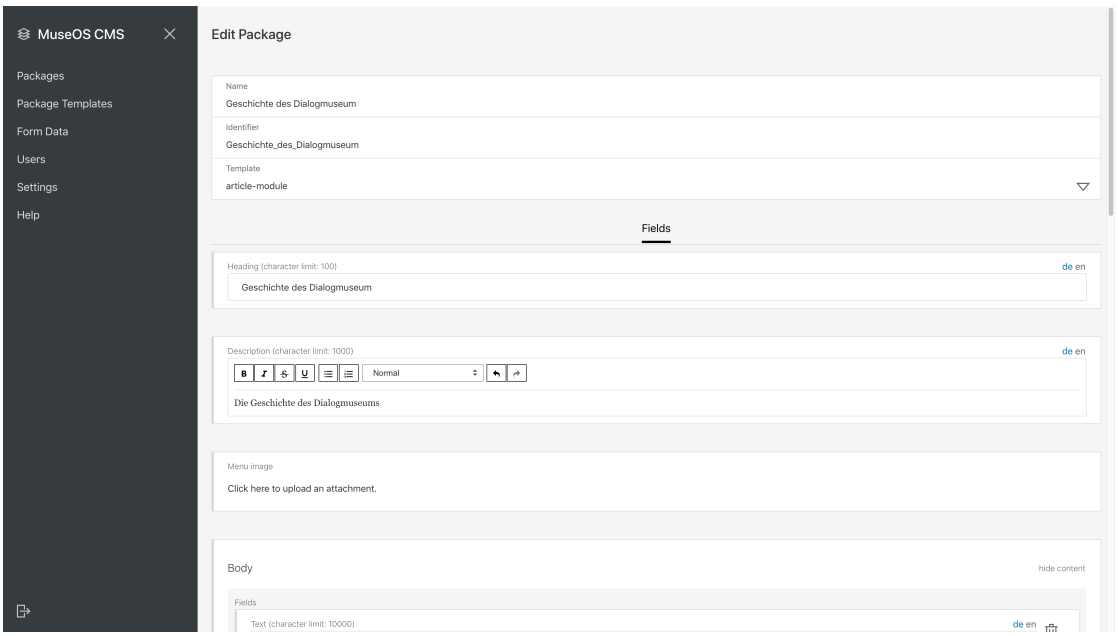

Abbildung 5.1.: Weboberfläche des CMS

werden.**To do** (6)

Für die Sharing-Station wurde auf ein bereits vorhandenes, von MESO entwickeltes, CMS zurückgegriffen. Es wurde jedoch vom Autor dieser Arbeit an manchen Stellen angepasst und um einige Funktionen erweitert. Es ist in Node.js [16] geschrieben und speichert die Daten in einer MongoDB [27].
Es handelt sich dabei um ein reines headless CMS. Es besitzt neben einer JSON-API um die Daten auszuliefern noch eine Upload-Schnittstelle sowie eine Schnittstelle um Formulardaten abzuspeichern.
Die Struktur des CMS ist Template-basiert **To do** (7). Administratoren können Templates anlegen, welche eine Datenstruktur definieren. Redakteure können dann anhand dieser Templates konkrete Content-Objekte erstellen, welche baumartig miteinander verknüpft werden. Templates werden dabei mithilfe von verschiedenen Datenfeldern gebildet. Datenfelder sind zum Beispiel Zahlenfeld, Textfeld oder Rich-Text-Feld. Datenfelder können dabei als übersetzbare Felder angelegt werden. Beim Erstellen eines Content-Objekts können später so verschiedene Sprachversionen hinterlegt werden.
Das CMS ist dauerhaft bei einem Cloud-Anbieter gehostet und die Oberfläche somit über das Internet aufrufbar. Abbildung 5.1 zeigt diese Oberfläche.

5.2. Clientsoftware

Im Folgenden wird die Clientsoftware des Kiosksystems betrachtet. Damit ist der Softwareteil des Systems gemeint, welcher auf der Kiosk-Workstation vor Ort läuft. Diese wird in diesem Kapitel zur weiteren Betrachtung weiter unterteilt. Zum ersten in die *Applikation*, welche die eigentliche, den Nutzer*innen zugängliche Kioskapplikation darstellt. Zum zweiten in die *Plattform* der Clientsoftware. Also die Umgebung, in welcher die Applikation läuft. Und schließlich wird das *Deployment* besprochen. Damit ist die Art gemeint, wie die Applikation und Updates dieser auf die Kiosk-Workstation gelangen.

5.2.1. Applikation

Anforderungen aus Kapitel 3, welche die eigentliche Applikation der Clientsoftware betreffen, sind die Anforderungen **NFA2**, **NFA3** und **NFA7**.

NFA7 stellt die Anforderung der Modularität an das gesamte Softwaresystem. Diese betrifft also auch die Clientapplikation. Wie in Abschnitt 5.1 bereits erläutert, ist für eine große Modularität eine lose Kopplung von Daten- und Präsentationsschicht ein wichtiger Faktor. Umgekehrt bedeutet dies, dass die Clientsoftware eine in sich geschlossene eigene Anwendung sein muss. Sie besitzt lediglich die Datenschnittstelle zum CMS. Das leistet im Falle der Webtechnologien eine Single-Page Applikation (SPA). Eine SPA wird einmalig von einem Server geladen. Laden von weiteren Seiten beim Navigieren durch die Oberfläche findet nicht statt [28]. Um Transaktionen und neue Zustände möglich zu machen, können jedoch im Hintergrund asynchrone Datenanfragen gestellt werden (AJAX). Die SPA gibt so den Nutzer*innen nicht mehr das Gefühl einer Webseite, sondern einer geschlossenen Anwendung, was im Falle der Kiosksoftware ein gewünschter Effekt ist und damit auch zur Erfüllung der Anforderung **NFA4** beiträgt.
Das Laden einer einzigen Seite vom Server bedeutet jedoch nicht, dass keine Navigationsstruktur mit Vor- und Zurück-Bewegungen implementieren werden kann. Die Navigation findet jedoch alleinig in der Clientapplikation statt und ist nicht mit Laden von neuen Seiten verbunden. Es werden lediglich neue Zustände aufgerufen [29].

NFA2 fordert Offline-Verfügbarkeit. Auch hier trägt das Konzept der SPA zur Erbringung der Anforderung bei. Dadurch, dass die Anwendung nur einmal initial geladen werden muss und danach vollständig im Client zur Verfügung steht, ist eine Netzwerkverbindung nach dem initialen Laden nicht mehr nötig. Das umfasst allerdings nur die eigentliche Applikation und nicht die AJAX-Anfragen, die auch zu späteren

Zeitpunkten erfolgen können. Auch ist bei einem Neuladen der Applikation immer eine Netzwerkverbindung zwingend nötig. Das Konzept der SPA erfüllt die Anforderung der Offline-Verfügung also nur zu einem gewissen Teil. Allerdings bildet die Geschlossenheit einer SPA eine wichtige Voraussetzung, um die Applikation auch vollständig offline zur Verfügung zu stellen. *Vollständig offline* meint hier also, dass auch Daten, die über einen AJAX-Aufruf aus dem CMS geladen werden, lokal gecached werden können. Zwar wären Content-Daten bei Offlinestatus nicht mehr durch das CMS aktualisierbar, aber zumindest die letzte Version solange verfügbar, bis ein Netzwerkzugriff wieder möglich ist. Um solch einer Anforderung gerecht zu werden, bieten moderne Browser die Service-Worker API [30].
Bei Service-Workern handelt es sich um Proxy-Server, welche sich zwischen der Webanwendung und dem Netzwerk befinden. Ein Service-Worker wird in einem Skript definiert, welches unabhängig vom Prozess der Webseite ausgeführt wird [31]. Dieses speichert Netzwerkzugriffe in einem lokalen Speicher und kann Anfragen auf diesen verweisen, sollte das Netzwerk bei dem nächsten Zugriff nicht erreichbar sein [30]. Service-Worker werden bei initialem Aufruf heruntergeladen und bleiben auch nach Schließen des Browsers gespeichert. Das gilt auch für die Daten, welche sie speichern. Diese sind also auch bei Neustart ohne Netzwerkverbindung noch vorhanden und können angezeigt werden.
Eine vollständige Offline-Verfügbarkeit wäre somit fast erreicht. Einzig Transaktionen, bei welchen Eingaben und Daten von Nutzer*innen gespeichert werden sollen, können noch nicht offline verfügbar gemacht werden. Auch hier kommt die Technik der Service-Worker an ihre Grenzen [32].
Dennoch gibt es auch hier Lösungen für dieses Problem. Beispielsweise bietet die JavaScript Datenbank pouchDB [33] die Möglichkeit POST-Anfragen und Dateien in einer lokalen Datenbank solange zu speichern, bis eine Netzwerkverbindung wiederhergestellt ist, um sie dann mit einer Backend-Datenbank zu synchronisieren. Dieser Ansatz wird jedoch im Rahmen dieser Arbeit, aus Gründen des Umfangs, nicht weiter verfolgt.

NFA7 fordert schließlich die Multilingualität der Oberfläche. Wie in Abschnitt 5.1 bereits erläutert, können Datensätze im CMS mehrsprachig angelegt werden. Diese werden der Clientapplikation in einem gemeinsamen Datensatz zur Verfügung gestellt und sind so zu jedem Zeitpunkt verfügbar. Nutzer*innen sollen später in der Oberfläche der Clientapplikation die Möglichkeit haben, die Sprache umzustellen. Erfolgt dies, muss keine neue Seite oder neuer Datensatz vom Server angefordert werden, sondern lediglich die angezeigten Daten durch die entsprechende Sprachversion ausgetauscht werden.
To do (8)

Um dieser beschriebenen Architektur gerecht zu werden, wurde für die Clientapplikation

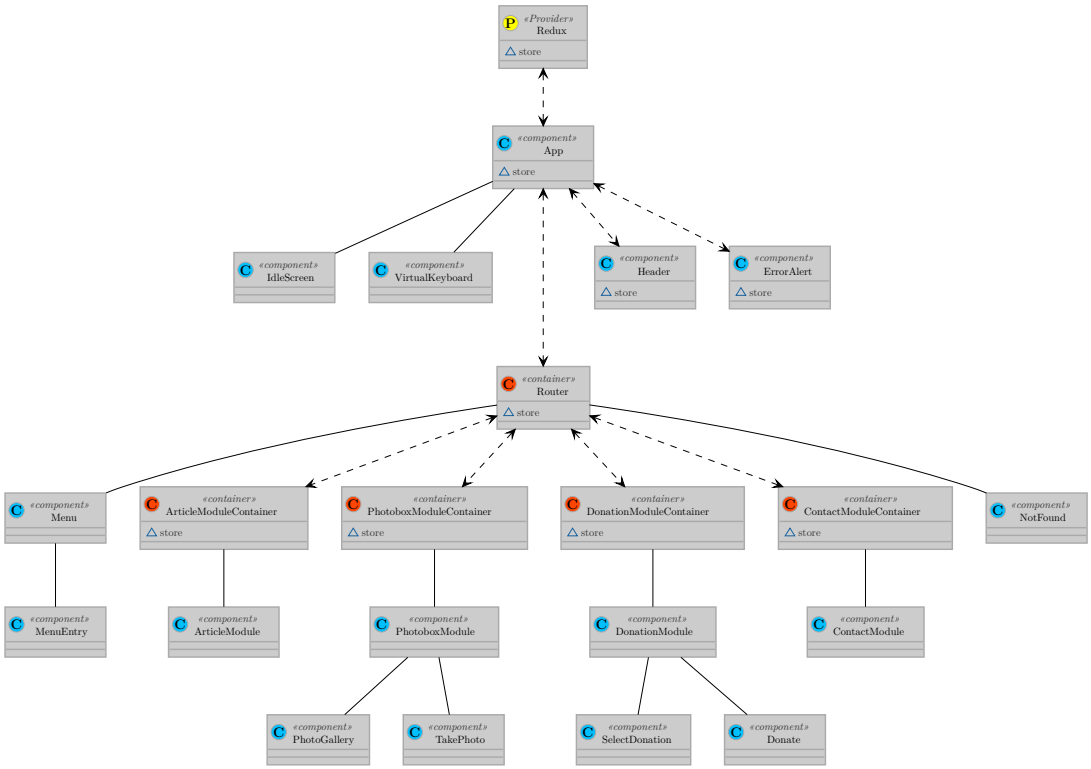

Abbildung 5.2.: Komponenten Diagram der Sharing-Station App

der Sharing-Station unter Anderen die JavaScript Bibliotheken React [34] und Redux [35] zusammen mit dem Toolkit Workbox [36] verwendet. React bildet dabei den Rahmen für die SPA-Architektur; Workbox bietet eine Bibliothek, welche es erleichtert Service-Worker zu implementieren und Redux ist eine Bibliothek, welche ein Pattern für die Verwaltung von Applikationszuständen bereitstellt. Redux hilft bei steigender Komplexität von Zuständen in Anwendungen – ist im Rahmen dieser Thesis aber von untergeordneter Bedeutung und wird daher nicht weiter erläutert.

React bietet die Möglichkeit, die meisten der zuvor genannten Anforderungen zu erfüllen. Das ist neben der Implementierung der SPA-Architektur, auch die Möglichkeit die Anwendung zu modularisieren. Denn: Neben der losen Kopplung von CMS und Clientapplikation ist auch eine Modularität innerhalb der Clientapplikation nötig, um die Anforderung **NFA7** vollständig zu erfüllen. React erlaubt dies durch sein Komponenten-basiertes System. Einzelne Interface-Elemente oder auch einzelne Unterseiten können als Komponenten gekapselt werden. Wie fein diese Kapselung sein soll, kann von den Entwickler*innen entschieden werden. Richtig implementiert, können Komponenten so wiederverwendet, ausgetauscht oder das System leicht um neue erweitert werden.
Abbildung 5.2 zeigt die Komponentenstruktur der Clientapplikation der Sharing-Station. Die Abbildung ist vereinfacht und zeigt nur die wichtigsten Komponenten. Die Art der Darstellung orientiert sich dabei an der von Kitson [37] vorgeschlagenen Weise um React Anwendungen zu visualisieren. Diese wiederum basiert auf dem System des UML-Klassendiagramms [21], jedoch sind die Klassen in diesem Falle keine Klassen, sondern die Komponenten [1]. Rot markierte Komponenten sind Container-Komponenten und haben keine eigene Darstellung – sie implementieren lediglich Geschäftslogik. Blau markierte Komponenten sind Interface-Komponenten, die eine direkte Darstellung in der Oberfläche haben. Die einzelne gelb markierte Komponente ist der Redux-Store. Er ist ebenfalls eine besonderer Art der Container-Komponente und beinhaltet den Applikationszustand. **To do** (9)
Im Zentrum der Abbildung befindet sich die Router-Komponente. Sie leitet zwischen den einzelnen Unterseiten zu denen die Nutzer*innen navigieren können. Teilbäume, abgehend von der Router-Komponente, können als die Hauptmodule der Applikation gesehen werden. Sie können dabei Untermenüs oder Module sein, welche die konkreten Funktionalitäten wie die Spenden-Funktion (**FA3**) oder die Newsletter-Funktion (**FA5**) implementieren. Diese klarer Abgrenzung der einzelnen Module bietet so die Möglichkeit, diese leicht durch andere zu ersetzen oder um neue zu erweitern.

[1] React-Komponenten können Klassen sein - müssen aber nicht. Seit React 16.8.0 und der Einführung der Hook-API [38] kann sogar gänzlich auf Klassen verzichtet werden und Komponenten durchgängig als Funktionen implementiert werden. Das wurde in diesem Falle so umgesetzt.

```
1  const path = require('path');
2  const webpack = require('webpack');
3  const merge = require('webpack-merge');
4  const WorkboxPlugin = require('workbox-webpack-plugin');
5  const WebpackPwaManifest = require('webpack-pwa-manifest');
6  const baseConfig = require('./webpack.base.config');
7
8  module.exports = merge(baseConfig, {
9    mode: 'production',
10   target: 'web',
11   plugins: [
12     new webpack.DefinePlugin({
13       'process.env.NODE_ENV': JSON.stringify('production'),
14     }),
15     new WebpackPwaManifest({
16       name: 'Sharing-Station',
17       short_name: 'ShSt',
18       description: 'Sharing-Station Dialogmuseum Frankfurt',
19       background_color: "#fafafa",
20       theme_color: "#d33139",
21       display: 'fullscreen',
22       start_url: '/',
23       scope: '/',
24       icons: [
25         {
26           src: path.resolve('src/assets/icons/logo-fb.png'),
27           sizes: [96, 128, 192, 256, 384, 512]
28         }
29       ]
30     }),
31     new WorkboxPlugin.GenerateSW({
32       swDest: 'sw.js',
33       skipWaiting: true,
34       maximumFileSizeToCacheInBytes: 60000000,
35       runtimeCaching: [{
36         urlPattern: new RegExp('.*\/api\/.*'),
37         handler: 'NetworkFirst'
38       }]
39     }),
40   ],
41 });
```

Abbildung 5.3.: Ausschnitt der Webpack Konfiguration der Sharing-Station App

Workbox hilft bei der Implementierung von Service-Worker Skripten. Darüber hinaus gibt es ein Workbox-Plugin [39] für den JavaScript-Bundler Webpack [40]. Beide Technologien wurden im Falle der Sharing-Station eingesetzt. Mit Hilfe dieses Plugins, müssen Service-Worker Skripte nicht selbst implementiert werden, sondern werden automatisiert erstellt. Im Konfigurationsfile von Webpack, kann durch setzen von Parametern bestimmt werden, was der Service-Worker leisten soll. Abbildung 5.3 zeigt einen Teil der Webpack-Konfiguration der Sharing-Station. In Zeile 31-39 wird das Objekt zur Generierung des Service-Workers erstellt. Als Parameter wird ein Objekt mit der Konfiguration übergeben. Dieses Objekt ist optional – auch ohne Konfiguration würde ein Service-Worker erstellt

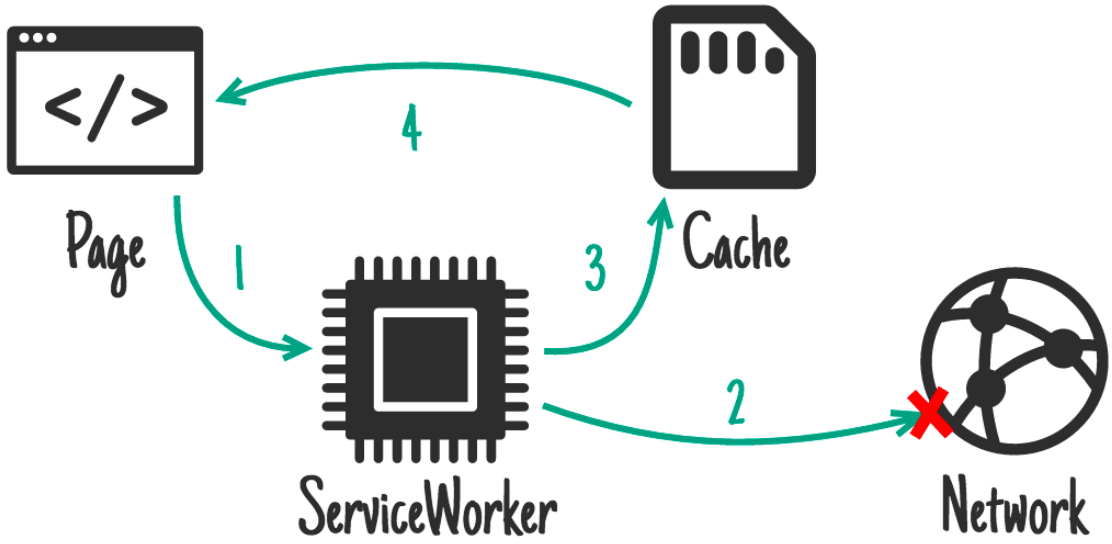

Abbildung 5.4.: NetworkFirst-Strategie. Quelle: [41]

werden, welcher dafür sorgt, dass alle von Webpack erstellten Files beim Nutzen der Applikation gecached und offline verfügbar gemacht werden. Weitere Funktionalitäten können durch Setzen der Konfigurationsparameter aktiviert werden.
In Zeile 34 wird die maximale Dateigröße gesetzt. Dateien größer als dieser Wert, werden nicht offline verfügbar gemacht. Dieser Wert ist standardmäßig niedriger und wurde an dieser Stelle erhöht um auch das Video zu cachen, welches für den Idle-Modus (**FA8**) benötigt wird.
In Zeile 35 wird definiert welche Dateien, neben den von Webpack erstellten Dateien, gecached werden sollen. Das sind in diesem Fall alle Daten und Dateien, die durch Netzwerkanfragen an das CMS geladen werden. Diese Anfragen haben den gemeinsamen Namespace `/api/` und können dadurch mit dem regulären Ausdruck `/.*\/api\/.*/` erfasst werden. Mit der Option `handler` wird eine Caching-Strategie gewählt. Workbox bietet hier die Möglichkeit zwischen fünf verschiedenen, vordefinierten Handler-Klassen zu wählen [42]. Die Strategie der Klasse `NetworkFirst` beruht darauf, dass die Anfrage zuerst an das Netzwerk gestellt wird. Ist dieses nicht erreichbar, wird auf die letzte Version im Cache zurückgegriffen. Ist ein Netzwerkzugriff erfolgreich, wird auch immer die Version im Cache durch die neue Version ersetzt. Abbildung 5.4 visualisiert diese Strategie. **To do** (10)

5.2.2. Plattform

Nach dem Anforderungskatalog aus Kapitel 3, betreffen die Anforderungen **NFA4** und **NFA5** die Plattform der Clientapplikation.

NFA5 fordert die Plattformunabhängigkeit. Dies meint, dass die entwickelte Applikation später, wenn gewünscht, in unterschiedlichen Umgebungen genutzt werden kann. Also beispielsweise neben auf einer herkömmlichen Windows-Workstation auch auf einem Tablet oder einer Linux-Workstation. Webtechnologien bieten hierfür die beste Vorraussetzung. Browser oder Umgebungen die Webanwendungen ausführen können, gibt es für praktisch jede Plattform. Wichtig ist jedoch trotzdem, dass bei der Entwicklung auf Plattformunabhängigkeit geachtet wird. Beispielsweise bietet das Framework Electron.js die Möglichkeit, mit Webtechnologien native Desktop-Apps zu entwickeln. Es wird dabei aber nicht nur mit Browsertechnologien gearbeitet – ein parallel laufender Node.js-Prozess bietet die Möglichkeit auf Betriebssystem-Funktionalitäten zuzugreifen und über einen Inter-Process-Communication (IPC) Kanal mit der Oberfläche der Anwendung zu kommunizieren [43]. Dies bietet zwar allerhand Vorteile, beispielsweise bei der Kommunikation mit Hardwarekomponenten, bindet die Anwendung aber an eine Desktop-Umgebung. Bei der Entwicklung der Applikation wird im besten Fall also darauf geachtet, dass nur mit Browsertechnologien gearbeitet wird, da eine Bindung an eine spezifische Plattform so nicht stattfindet. Bei einem solchen Vorgehen wäre es später immer noch möglich, die Anwendung in einem Electron.js-Framework zu verpacken und als Desktop-Anwendung auszuliefern. Genauso könnte die Applikation aber auch als klassische Webanwendung im Internet verfügbar gemacht oder als mobile App, mit ein Framework wie Apache Cordova [44], ausgeliefert werden.

NFA4 fordert die Geschlossenheit der Clientapplikation. Im Folgenden wird davon ausgegangen, dass die Software auf einer Workstation und nicht beispielsweise auf einem mobilen Endgerät präsentiert werden soll – dies würde eine andere Herangehensweise erfordern.
Für die Geschlossenheit ist es im ersten Schritt zwingend notwendig, dass die Applikation in einem Vollbildmodus gestartet wird. Darüberhinaus sollten jegliche Betriebssystemfunktionen, die das Verlassen der Applikation möglich machen würden, deaktiviert werden. Electron.js bietet dafür die *kiosk* Option. Diese sorgt unter Anderem dafür, dass die Applikation im Vollbildmodus gestartet, die Menüleiste ausgeblendet und das Verlassen des Vollbildmodus deaktiviert wird.
Da sich jedoch zuvor entschieden wurde eine reine Browseranwendung zu implementieren, ist es nun naheliegend diese auch in einem Browser und nicht als native Applikation zu präsentieren. Der Browser Google Chrome bietet dafür ebenfalls eine *kiosk* Option.

Diese kann durch Übergeben des Startparameters `--kiosk` aktiviert werden. Zusätzlich kann eine URL angegeben werden, die beim Öffnen direkt aufgerufen werden soll. Der Browser startet dann ebenfalls in einem nicht verlassbaren Vollbildmodus und blendet jegliche Menüleisten aus. Zusätzlich wird bei Touch-Input der Mauszeiger vollständig ausgeblendet.

Für die Clientapplikation der Sharing-Station wurde, wie zuvor beschrieben, eine reine Browseranwendung implementiert, um die Plattformunabhängigkeit zu gewährleisten. Für die Auslieferung der Software wurde auf ein Framework wie Electron.js gänzlich verzichtet. Stattdessen wurde sich entschieden, durch Definieren eines so genannten Webmanifests, eine installierbare Progressive Web App (PWA) zu erstellen [45]. Dies bedeutet, dass die Applikation zum einem in einem Browser aufgerufen, aber auch durch Installieren dem Desktop der Workstation hinzugefügt werden kann. Diese installierte Applikation entspricht nichts weiter als einem Chrome-Browser, welcher bei Öffnen die gewünschte Seite direkt, bzw. durch den zuvor definierten Service-Worker die Applikation-Files aus dem lokalen Speicher, aufruft. Auch können der Applikation die selben Startparameter wie dem Chrome Browser übergeben werden. Das ist wichtig, um hier ebenfalls die `--kiosk` Option nutzen zu können. Zusätzlich kann für die installierte Version der Applikation ein Desktop-Icon definiert werden [45]. Dieses trägt dazu bei, den Eindruck einer nativen Applikation zu erwecken.
Wie schon beim Erstellen der Service-Worker, wurde auch für das Erstellen des Webmanifests auf ein Webpack-Plugin [46] zurückgegriffen. Zeile 15-30 in Abbildung 5.3 zeigt die Einbindung dieser Plugins, welches während dem Build-Vorgang, mit den übergebenen Parametern, das Manifest erstellt.

5.2.3. Deployment

Die einzige Anforderung, welches das Deployment direkt betrifft, ist die Anforderung **NFA6**. Diese besagt lediglich, dass dies möglichst einfach und von außerhalb möglich sein soll.
Bei nativen Applikation wäre hier ein Updater denkbar. Also ein Service, welcher regelmäßig die Verfügbarkeit neuer Versionen auf einem Server prüft und einen Download dieser anbietet. Electron.js bietet für diesen Fall einige Möglichkeiten, diesen Prozess möglichst simpel zu gestalten. Beispielsweise mit dem eigenen `autoUpdater`-Modul [47] oder Plugins wie *electron-builder* [48]. Auch sind mit beiden Möglichkeiten automatische Updates umsetzbar.
Im Falle einer Applikation, die einzig mit Browsertechnologien implementiert wurde, sind noch simplere Strategien denkbar. Hier ist ein normales Webhosting der Applikationsfiles bereits eine ausreichende Deployment-Strategie. Wurde keine Offline-Verfügbarkeit durch

Service-Worker implementiert, können die statischen Files einer Browseranwendung auch einfach auf dem Filesystem der Workstation hinterlegt und in einem Browser aufgerufen werden. Eventuelle Updates müssten in diesem Falle aber vor Ort oder durch einen Fernzugriff auf den Rechner eingespielt werden. Im Fall des Webhostings genügt das Überspielen der neuen Files auf einen Webserver.

Für die Sharing-Station wurde auf dem Server neben dem CMS auch ein Webserver installiert, welcher die Applikationsfiles ausliefert (Abbildung 4.1). Zusätzlich wurde über den firmeninternen GitLab-Server [49] eine Build- und Deployment-Pipeline eingerichtet, welche die Applikationsfiles automatisch generiert und vom Master-Branch des Repositorys auf den Webserver überträgt. Für ein Updaten der Clientapplikation, auf der Workstation in der Ausstellung, ist also nichts weiter nötig als die Änderungen auf den Master-Branch des Repositorys zu übertragen und die Applikation vor Ort neu zu laden. Das Webhosting der Applikation bietet außerdem den Vorteil, dass diese jeder Zeit in einem Browser über das Internet aufgerufen werden kann. Beim Anlegen von Daten im CMS kann dies so als Preview genutzt werden. **To do** (11)

6. Hardwareschnittstellen

In diesem Kapitel wird die Kommunikation mit Hardwarekomponenten betrachet. **NFA1** fordert, dass die Clientsoftware zusammen mit Hardwarekomponenten funktionieren muss. Wie in der Anforderung bereits erläutert, sind Hardwarekomponenten meist ein essenzieller Bestandteil von Kiosksystemen und die Steuerung dieser, durch die Clientsoftware, muss gewährleistet werden. Im Falle der webbasierten Softwaretechnologien stößt man hier auf einige Hürden.
Während Serveranwendungen, native Anwendungen und auch der Node-Prozess einer Electron.js-Anwendung Zugriff auf eine Vielzahl von Betriebssystemschnittstellen haben **To do** (12) und somit leicht mit Hardwarekomponenten kommunizieren können, sind die Möglichkeiten einer reinen Browseranwendung wesentlich limitierter. Browseranwendungen und Webseiten werden typischerweise, aus sicherheitstechnischen Gründen, kein direkten Zugriff auf Betriebssystem-APIs gegeben. Nur einige ausgewählte Funktionalitäten werden durch die Web APIs zur Verfügung gestellt [50]. Für Anwendungsfälle, die über die Möglichkeiten der Web APIs hinausgehen, müssen eigene Lösungen entwickelt werden. Hierzu wurde im Rahmen dieser Arbeit eine Lösung über das MQTT/Websocket Protokoll entwickelt.

6.1. Web APIs

Die Web APIs bilden Schnittstellen, mit denen eine Browserandwendung, typischerweise über ein JavaScript-Interface, kommunizieren kann [50]. Manche APIs bieten dabei auch Zugriff auf ausgewählte Betriebssystemschnittstellen und Hardware. Beispielsweise kann mit der *Web Audio API* auf angeschlossene Aufnahme- und Ausgabequellen für Audio zugegriffen werden [51].
Für die in **FA4** beschriebene Foto-Commitment-Funktion wird das Bild einer angeschlossenen Webcam benötigt. Das ist mit dem *Media Capture and Streams API* möglich [52]. Abbildung 6.1 zeigt den Zugriff über das API. In Zeile 16 wird über die `getUserMedia()` Funktion ein `MediaStream` Objekt angefordert. Über das Konfigurationsobjekt kann genau spezifiziert werden, welche Art von Stream vom Betriebssystem angefordert werden soll. In diesem Fall wird ein reiner Video-Stream mit einer 4k-Auflösung im Seitenverhältnis 1:1 gefordert. Der Zugriff muss im Browser über ein Dialogfenster bestätigt werden, anschließend kann das `MediaStream` Objekt als Videoquelle in der Anwendung

```
let webcamStream;

const config = {
  video: {
    aspectRatio: { exact: 1 },
    width: { ideal: 4096 },
    height: { ideal: 2160 },
  },
  audio: false,
};

const initWebcam = async () => {
  try {
    webcamStream = await navigator.mediaDevices.getUserMedia(config);
  } catch (error) {
    Logger.error(`Error initialising webcam: ${error}`);
    throw error;
  }
};

const insertWebcamStream = async (webcamElement) => {
  if (!webcamStream) {
    await initWebcam();
  }
  webcamElement.srcObject = webcamStream; // eslint-disable-line

  try {
    await webcamElement.play();
  } catch (error) {
    Logger.error(`Error playing webcam stream ${error}`);
  }
};
```

Abbildung 6.1.: Zugriff auf die Webcam über die *Media Capture and Streams API*

genutzt werden. Der `insertWebcamStream` Funktion in Zeile 23 kann dafür ein HTML Video-Element als Parameter übergeben werden. In Zeile 27 wird das Stream-Objekt als Quelle des Video-Elements gesetzt und anschließend in Zeile 30 gestartet.

6.2. meso-connect

Um nicht von den Möglichkeiten der Web APIs abhängig zu sein und die Kommunikation zwischen einer Browseranwendung und weiteren Geräten zu ermöglichen, wurde im Rahmen dieser Arbeit eine generische Lösung auf Basis des MQTT und Websocket Protokolls entwickelt. Diese entwickelte Konvention trägt den Namen *meso-connect*. Im Falle der Sharing-Station, soll damit die Kommunikation zu Kartenlesegerät und Münzschlitz (**FA3**) ermöglicht werden. Darüber hinaus soll sie aber auch eine Standardlösung für zukünftige Projekte sein.
Sie basiert dabei auf der Kommunikation über das Websocket und MQTT Protokoll und orientiert sich stark an der Homie Konvention [53].

6.2.1. Websocket und MQTT

Für eine sinnvolle Kommunikation zwischen einem Gerät und einer Anwendung ist meist eine bidirektionale Verbindung nötig. Ein Protokoll, welches bei Webanwendungen die bidirektionale Verbindung erlaubt, ist das Websocket Protokoll [54]. Über dieses kann eine Browseranwendung zum einen Nachrichten an einer Server übermitteln, zum anderen aber auch zu jedem Zeitpunkt Nachrichten vom Server empfangen und auf diese reagieren. Im TCP/IP-Modell ist Websocket dabei ein Protokoll, welches auf der Anwendungsschicht arbeitet. Dennoch kann es selbst auch eine Transportschicht für weitere Protokolle sein [55]. Um in Browseranwendungen mit dem Websocket Protokoll zu arbeiten, stellen die meisten Browser die Websocket API zur Verfügung [56].

MQTT ist ein auf dem publish/subscribe-Pattern basierendes Nachrichten-Protokoll [57]. Es wird häufig im den Bereichen IoT und Home-Automation eingesetzt. Es folgt einer klassischen Server-Client-Architektur, wobei der Server in der MQTT Umgebung *Broker* genannt wird. Clients können über den Broker Nachrichten-Topics abonnieren oder Nachrichten über den Broker auf Topics versenden. Der Broker leitet versendete Nachrichten an alle Clients weiter, die das entsprechende Topic abonniert haben. Topics sind Namespaces **To do** (13), welche hierarchisch aufgebaut sind. Das Slash-Zeichen wird dabei als Trennzeichen genutzt. Ein Topic kann also zum Beispiel

```
home/living-room/temperature
```

```
homie / device123 / $homie : 3.0
homie / device123 / $name : My device
homie / device123 / $state : ready
homie / device123 / $nodes : mythermostat

homie / device123 / mythermostat / $name : My thermostat
homie / device123 / mythermostat / $properties : temperature

homie / device123 / mythermostat / temperature : 22
homie / device123 / mythermostat / temperature / $name : Temperature
homie / device123 / mythermostat / temperature / $unit : C
homie / device123 / mythermostat / temperature / $datatype : integer
homie / device123 / mythermostat / temperature / $settable : true
```

Abbildung 6.2.: Homie Topic Struktur. Quelle: [53]

lauten. Mit Wildcards (+ und #) können auch ganze Namespaces, anstelle einzelner Topics, abonniert werden [58].
Über MQTT können Nachrichten in beliebiger Encodierung oder als binäre Daten versendet werden [59].

Wie eingangs erwähnt, kann das Websocket Protokoll Transportschicht für höhere Protokolle sein. So kann es auch die Transportschicht für MQTT sein und eine Browseranwendung kann zum MQTT-Client werden. Da das Websocket Protokoll alle nötigen Eigenschaften besitzt, ist es möglich MQTT Pakete über Websocket Pakete zu versenden. Die einzige Voraussetzung hierfür ist, dass der Broker die Option der Kommunikation über Websockets unterstützt. Der beliebte Open Source Broker Mosquitto [60] besitzt diese Option.

6.2.2. Homie

Ein Designprinzip von MQTT lautet Einfachheit [61]. Das Protokoll gibt nicht vor wie Topics aufgebaut werden und Daten benötigen keine spezielle Encodierung und Typisierung. Eine Kommunikation muss also immer selbst und für jedes Projekt neu konfiguriert werden. Das ist aufwendig und führt in der Praxis häufig zu Fehlern.

Homie [53] bietet dafür eine Konvention, welche diese Probleme lösen soll. Dabei ist Homie ein Satz an Regeln, welcher in einigen Softwarebibliotheken, für verschiedene Sprachen und Plattformen, implementiert wurde. Die Konvention schreibt dabei vor, wie Topics gebildet werden und führt eine Typisierung für die Payload-Daten ein. Um Topics zu bilden, unterscheidet Homie zwischen Devices, Nodes und Properties. Ein Device ist ein physisches Gerät, wie beispielsweise ein Mikrocontroller. Ein Device kann mehrere Funktionen besitzen – diese werden als Nodes bezeichnet. Im Falle des Mikrocontrollers

können das einzelne angeschlossene Sensoren oder Geräte sein. Nodes wiederum können mehrere Properties haben. Properties sind Eigenschaften, welche eine konkreten Wert besitzen können oder deren Wert von anderen Clients gesetzt werden kann. Ist am Mikrocontroller ein Thermometer angeschlossen, könnte ein Property *Temperatur* heißen. Aus dieser Unterscheidung heraus werden die Topics nach der Struktur

```
homie/device/node/property
```

gebildet. Abbildung 6.2 zeigt ein vollständiges Beispiel. Dort ist auch zu sehen, dass Metadaten zu Devices, Nodes und Properties über spezielle Topics, mit dem Prefix `$`, veröffentlich werden. Zu den Metadaten auf Property-Ebene zählt unter Anderem die Angabe eines Datentyps. Homie gibt dafür die Typen String, Integer, Float, Boolean, Enum und Color vor. Weitere Metadaten sind das Attribut `$settable`, mit dem bestimmt werden kann ob ein anderer Client den Wert setzten darf oder das Attribut `$retained`, über welches gesteuert werden kann, ob es sich bei dem Property um ein Event handelt oder einen Wert, welcher dauerhaft auf dem Topic ausgelesen werden kann. Es existieren noch einige weitere Attribute, welche aber aus Gründen des Umfangs nicht weiter erläutert werden.
Geräte, welche die Homie Konvention implementieren, sind praktisch konfigurationslos mit anderen Homie Clients verwendbar. Denn: Durch die konsistente Veröffentlichung der Metadaten unter den speziellen Topics ergibt sich eine Autodiscovery-Eigenschaft. Clients können durch Abonnieren von Wildcard-Topics andere Clients finden und deren Eigenschaften erkennen. Zusätzlich ist durch die konsistente Typisierung mit vorgegebenen Datentypen eine standardisierte Kommunikation möglich.

6.2.3. Konvention

Basierend auf der Kommunikation über das MQTT/Websocket Protokoll und der Homie Konvention, wurde im Rahmen dieser Arbeit eine eigene Konvention entwickelt. Diese soll die Kommunikation zwischen Hardwaregeräten und Browseranwendungen standardisieren. Der Einsatz beschränkt sich dabei nicht nur auf Kiosksysteme, sondern ist auch in anderen Szenarien denkbar.
Im Folgenden werden die wichtigsten Eigenschaften der Konvention erläutert – die vollständige Konvention befindet sich in Abschnitt A.1. Auch sei an dieser Stelle erwähnt, dass die Konvention in Absprache und teilweise zusammen mit anderen Mitarbeitern der Firma MESO entwickelt wurde.

Die Konvention setzt dabei im ersten Schritt die Kommunikation über das MQTT Protokoll voraus. Um Hardwaregeräte über das MQTT Protokoll ansprechen zu können

sind unterschiedliche Lösungen denkbar. Diese sind vom jeweiligen Gerät und Anwendungsfall abhängig. Bei einfachen Sensoren oder Geräten mit serieller Schnittstelle ist der Anschluss an einen Mikrocontroller denkbar. Dieser muss entsprechend programmiert werden, um den Zugriff auf die angeschlossenen Geräte oder Sensoren über eine MQTT Schnittstelle zu ermöglichen.
Um eine Browseranwendung in die Kommunikation einzubinden, wird die Kommunikation über das MQTT/Websocket Protokoll vorausgesetzt. Der Broker muss dafür die Kommunikation auf zwei verschiedenen Netzwerkports ermöglichen. Zum Beispiel Port 1883 für reguläre MQTT-Nachrichten und Port 9001 für Nachrichten über das Websocket/MQTT Protokoll.
Nach diesem Setup ist eine bidirektionale, flexible Datenverbindung zwischen Browseranwendung und Broker hergestellt.

Um die Kommunikation zu vereinheitlichen, werden einige Grundsätze der Homie Konvention übernommen. Das ist zum ersten die Unterscheidung in Devices, Nodes und Properties und die sich daraus ergebende Topic-Struktur. Weiter wird das Typsystem für die Payload-Daten übernommen. Dieses wird allerdings um den *JSON*-Typ erweitert. Ein JSON-Objekt könnte auch als String übertragen werden – dann wäre allerdings das Überprüfen der Wohlgeformtheit des JSON-Objekts nicht Teil der Konvention. Da in der Praxis, und gerade bei Projekten der Firma MESO, oft mit JSON-Objekten in der MQTT-Kommunikation gearbeitet wird, ist das standardmäßige Prüfen von JSON-Objekten eine sinnvolle Erweiterung.
Die Hauptsache in der sich meso-connect von Homie unterscheidet, ist die Konfiguration. meso-connect sieht dafür ein File vor. Dieses, im JSON-Format gehaltene File, ist Grundlage jeder Kommunikation mit der meso-connect Konvention. Jeder Client wird dabei mit dem selben File konfiguriert – es muss deshalb nur einmalig erstellt werden und kann in einer Versionsverwaltung gehalten werden. Abbildung 6.3 zeigt die Vorlage für ein solches Konfigurationsfile. Gut zu erkennen ist die Unterscheidung von Devices, Nodes und Properties. Diese sind jeweils Arrays – es können also jeweils beliebig viele eingetragen werden. Wichtig ist, dass jeder MQTT-Client als ein Device im Konfigurationsfile aufgelistet werden muss. Auch dann, wenn der Client selber keine Nodes und Properties besitzt. Das Konfigurationsfile gibt so immer einen Überblick über die gesamte MQTT Kommunikation eines Systems und kann für die Konfiguration jedes Clients genutzt werden.
Die im Konfigurationsfile angegebenen Attribute sind von der Homie Konvention übernommen. Einzig *id* heißt bei Homie *name*. Diese Attribute werden bei Homie auf eigenen Topics veröffentlicht, um die Auto-Discovery Funktion möglich zu machen. Darauf verzichtet meso-connect. Da jeder Client das gemeinsame Konfigurationsfile zur Grundlage hat, wäre das Veröffentlichen der Attribute redundant. Ein Auto-Discovery Feature ist im

```
{
  "broker": {
    "address": "<brokerAddress>",
    "port": "<brokerPort>",
    "username": "<username>",
    "password": "<password>"
  },
  "devices": [
    {
      "id": "<deviceId>",
      "nodes": [
        {
          "id": "<nodeId>",
          "properties": [
            {
              "id": "<propertyId>",
              "datatype": "<propertyDatatype>",
              "unit": "<propertyUnit>",
              "format": "<rangeOfPayloads>",
              "settable": "<isSettable>",
              "retained": "<isRetained>"
            }
          ]
        }
      ]
    }
  ]
}
```

Abbildung 6.3.: meso-connect Konfigurationsfile

Kontext von Kiosk- bzw. Installationssystemen für Museen und Ausstellungen ohnehin weniger wichtig. Solche Systeme sind, mit all ihren Clients, meist im Voraus ausreichend bekannt. Das Konfigurationsfile bietet dafür in diesem Kontext größere Vorteile, da es die Konfiguration der einzelnen Clients stark vereinfacht **To do** (14). Die einzigen Attribute, welche auf Topics zur Laufzeit veröffentlicht werden, sind

`meso-connect/devicename/$state`

und

`meso-connect/devicename/nodename/propertyname/$error`

Das Attribut `$state` ist aus der Homie Konvention übernommen und gibt Information über den aktuellen Zustand des Geräts. Dieser Wert ist dynamisch und kann sich zur Laufzeit verändern. `$error` ist nicht Teil der Homie Konvention. Aber auch dieser Wert kann nur dynamisch gesetzt werden. Auf diesem Topic sollen eventuell auftretende Fehler in der Kommunikation veröffentlicht werden. Das könnten zum Beispiel Fälle sein, in denen Daten verschickt werden, die nicht dem Typ entsprechen. Oder wenn ein Client versucht ein Property-Wert zu ändern, welcher nicht setzbar ist. Beispielsweise können

so, durch abonnieren des Topics

```
meso-connect/+/+/$error
```

alle auftretenden Fehler in einem System erfasst werden.

Die meso-connect Konvention wurde bei der Sharing-Station angewandt. Wie Abbildung 4.1 zeigt, sind dafür Kartenterminal und der Sensor des Münzschlitz an einem Mikrocontroller angeschlossen. Dieser kommuniziert über Netzwerk und das MQTT Protokoll mit dem Broker. Bei dem Broker handelt es sich um eine Mosquitto-Instanz [60], welche auf der Kiosk-Workstation installiert ist. **To do** (15) Der Broker wiederum kommuniziert über das Websocket/MQTT Protokoll mit der Clientapplikation.
Für die eigentliche Kommunikation über das Protokoll wurde im Rahmen dieser Arbeit und für die Clientapplikation der Sharing-Station eine JavaScript Bibliothek implementiert, welche in einer Browser- oder Node.js-Anwendung nutzbar ist. Diese implementiert die gesamte meso-connect Konvention und ist für den universellen Einsatz konzipiert. Die vollständige API-Dokumentation dieser Bibliothek befindet sich in Abschnitt A.2.
Abbildung 6.4 zeigt das bei der Sharing-Station genutzte Konfigurationsfile für die meso-connect Kommunikation.

```
{
  "broker": {
    "protocol": "ws",
    "address": "localhost",
    "port": "9001"
  },
  "devices": [
    {
      "id": "donation-controller",
      "nodes": [
        {
          "id": "123",
          "properties": [
            {
              "id": "start-transaction",
              "unit": "#",
              "datatype": "float",
              "format": "0:20",
              "settable": "true",
              "retained": "false"
            },
            {
              "id": "abort-transaction",
              "datatype": "string",
              "settable": "true",
              "retained": "false"
            },
            {
              "id": "transaction-success",
              "datatype": "string",
              "settable": "false",
              "retained": "false"
            },
            {
              "id": "transaction-error",
              "datatype": "string",
              "settable": "false",
              "retained": "false"
            }
          ]
        },
        {
          "id": "coin-slot",
          "properties": [
            {
              "id": "coin-inserted",
              "datatype": "string",
              "settable": "false",
              "retained": "false"
            }
          ]
        }
      ]
    },
    {
      "id": "sharing-station-app",
      "nodes": []
    }
  ],
  "broadcasts": [
    "alert",
    "error"
  ]
}
```

Abbildung 6.4.: meso-connect Konfigurationsfile der Sharing-Station

7. Fazit

To do...

- ☐ 1 (p. iii): Englisches Abstract
- ☐ 2 (p. 5): Schreibweise prüfen
- ☐ 3 (p. 7): check
- ☐ 4 (p. 7): Überarbeiten und besser ausführen
- ☐ 5 (p. 9): Wirklich subjekt?
- ☐ 6 (p. 19): mehr Beispiele aufzählen
- ☐ 7 (p. 19): Schreibweise checken
- ☐ 8 (p. 21): Abschnitt allgemeiner halten
- ☐ 9 (p. 23): Assoziationen erklären
- ☐ 10 (p. 25): Strategie vielleicht nochmal überarbeiten bzw. Nachteile behandeln
- ☐ 11 (p. 28): Screenshot der Browser Preview einfügen
- ☐ 12 (p. 29): Node API
- ☐ 13 (p. 31): wirklich namespaces?
- ☐ 14 (p. 35): Vorteile ausführen
- ☐ 15 (p. 36): wirklich mosquitto?

Abbildungsverzeichnis

Literatur- und Quellenverzeichnis

[1] *vvvv - a multipurpose toolkit*. https://vvvv.org/. [Accessed 31.01.2020].

[2] *Unreal Game Engine*. https://www.unrealengine.com. [Accessed 31.01.2020].

[3] *Dialogmuseum Frankfurt*. https://dialogmuseum.de/mehr-dialog/museumsgeschichte/. [Accessed 31.01.2020].

[4] *Dialogmuseum Frankfurt – Zahlen und Fakten 2017*. https://dialogmuseum.de/wp-content/uploads/2018/04/Zahlen-und-Fakten-2017.pdf. [Accessed 31.01.2020].

[5] *Dialogue Social Enterprise*. https://www.dialogue-se.com/. [Accessed 23.01.2020].

[6] *Dialog im Dunkeln weltweit*. https://dialogmuseum.de/mehr-dialog/dialog-im-dunkeln-weltweit/. [Accessed 31.01.2020].

[7] J. Rowley und F. Slack. „Kiosks in retailing: the quiet revolution“. In: *International journal of retail and distribution management, 31 (6), 329-339.* (2003).

[8] *Meyers Großes Konversationslexikon*. http://www.woerterbuchnetz.de/Meyers?lemma=kiosk. [Accessed 31.01.2020].

[9] A. Zwahr. *Meyers großes Taschenlexikon*. Hrsg. von M. Lexikonredaktion. Bd. 11. Meyers Lexikonredaktion, 2006.

[10] W. Holfelder. *Multimediale Kiosksysteme*. 1. Aufl. Braunschweig Vieweg, 1995.

[11] A. Sawall. *McDonald's-Kunden sollen per Touchscreen bezahlen*. https://www.golem.de/1105/83508.html. [Accessed 31.01.2020]. Mai 2011.

[12] J. Borchers, O. Deussen und C. Knörzer. „Getting It Across: Layout Issues for Kiosk Systems“. In: *ACM SIGCHI Bulletin ; 27, 4. - S. 68-74* (1995).

[13] T. Berners-Lee und R. Cailliau. *WorldWideWeb: Proposal for a HyperText Project*. https://www.w3.org/Proposal.html. [Accessed 23.01.2020]. Nov. 1990.

[14] *JavaScript Press Release*. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html. [Accessed 31.01.2020]. Dez. 1995.

[15] *V8 – A high-performance JavaScript engine*. https://opensource.google/projects/v8. [Accessed 31.01.2020].

[16] *Node.js*. https://nodejs.org. [Accessed 31.01.2020].

[17] *Electron – Build cross-platform desktop apps with JavaScript, HTML, and CSS*. https://www.electronjs.org/. [Accessed 31.01.2020].

[18] C. Rupp und die SOPHISTen. *Systemanalyse kompakt*. 3. Aufl. Springer Vieweg, 2013.

[19] C. Ebert. *Systematisches Requirements Engineering*. 6. Aufl. dpunkt.verlag, 2019.

[20] C. Rupp und die SOPHISTen. *Requirements-Engineering und -Management*. 6. Aufl. München Hanser, 2014.

[21] *Unified Modeling Language*. Spezifikation 2.5.1. https://www.omg.org/spec/UML/2.5.1: Object Management Group (OMG), Dez. 2017.

[22] C. Kecher, A. Salvanos und R. Hoffmann-Elbern. *UML 2.5 - Das umfassende Handbuch*. 6. Aufl. Bonn Rheinwerk Verlag, 2018.

[23] K. Fakhroutdinov. *Deployment Diagrams Overview*. https://www.uml-diagrams.org/deployment-diagrams-overview.html. [Accessed 14.02.2020].

[24] D. Barker. *The State of the Headless CMS Market*. https://gadgetopia.com/post/9926. [Accessed 24.02.2020]. Feb. 2017.

[25] *REST API Resources*. https://developer.wordpress.com/docs/api/. [Accessed 24.02.2020].

[26] J. Bailey-Roberts. *WP Headless*. https://de.wordpress.org/plugins/wp-headless/. [Accessed 27.02.2020].

[27] *MongoDB - Die beliebteste Datenbank für moderne Apps*. https://www.mongodb.com. [Accessed 17.02.2020].

[28] D. Flanagan. *JavaScript: The Definitive Guide*. 5. Aufl. O'Reilly Media, Inc., Aug. 2006.

[29] J. M. A. Santamaria. *The Single Page Interface Manifesto*. http://itsnat.sourceforge.net/php/spim/spi_manifesto_en.php. [Accessed 28.02.2020]. Sep. 2015.

[30] *MDN web docs - Service Worker API*. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API. [Accessed 28.02.2020].

[31] M. Gaunt. *Service Workers: an Introduction*. https://developers.google.com/web/fundamentals/primers/service-workers. [Accessed 01.03.2020]. Aug. 2019.

[32] karolkip. *Replaying POST requests - Open issue on github.com*. https://github.com/w3c/ServiceWorker/issues/693. [Accessed 28.02.2020]. Mai 2015.

[33] *pouchdb - The Database that Syncs!* https://pouchdb.com/. [Accessed 28.02.2020].

[34] *React - A JavaScript library for building user interfaces*. https://reactjs.org/. [Accessed 28.02.2020].

[35] *Redux - A Predictable State Container for JS Apps*. https://redux.js.org/. [Accessed 28.02.2020].

[36] *Workbox - JavaScript Libraries for adding offline support to web apps*. https://developers.google.com/web/tools/workbox. [Accessed 28.02.2020].

[37] A. Kitson. *ZF4 Blog - Visualizing React*. https://zf4.biz/blog/visualizing-react. [Accessed 09.03.2020].

[38] *Introducing Hooks*. https://reactjs.org/docs/hooks-intro.html. [Accessed 28.02.2020].

[39] *Workbox webpack Plugins*. https://developers.google.com/web/tools/workbox/modules/workbox-webpack-plugin. [Accessed 01.03.2020].

[40] *Webpack*. https://webpack.js.org/. [Accessed 01.03.2020].

[41] J. Archibald. *The Offline Cookbook*. https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook. [Accessed 01.03.2020].

[42] *Workbox Docs - workbox-strategies*. https://developers.google.com/web/tools/workbox/reference-docs/latest/module-workbox-strategies. [Accessed 01.03.2020].

[43] *Electron Application Architecture*. https://www.electronjs.org/docs/tutorial/application-architecture. [Accessed 05.03.2020].

[44] *Apache Cordova - Mobile apps with HTML, CSS and JS*. https://cordova.apache.org/. [Accessed 01.03.2020].

[45] *MDN web docs - Web App Manifest*. https://developer.mozilla.org/en-US/docs/Web/Manifest. [Accessed 06.03.2020].

[46] *npm - webpack-pwa-manifest-plugin*. https://www.npmjs.com/package/webpack-pwa-manifest. [Accessed 06.03.2020].

[47] *Electron Documentation autoUpdate*. https://www.electronjs.org/docs/api/auto-updater. [Accessed 05.03.2020].

[48] *electron-builder*. https://www.electron.build/. [Accessed 05.03.2020].

[49] *GitLab*. https://about.gitlab.com/. [Accessed 06.03.2020].

[50] *MDN web docs - Web APIs*. https://developer.mozilla.org/en-US/docs/Web/API. [Accessed 06.03.2020].

[51] *MDN web docs - Web Audio API*. https://developer.mozilla.org/de/docs/Web/API/Web_Audio_API. [Accessed 10.03.2020].

[52] *MDN web docs - Media Capture and Streams API (Media Stream)*. https://developer.mozilla.org/en-US/docs/Web/API/Media_Streams_API. [Accessed 09.03.2020].

[53] *HOMIE - An MQTT Convention for IoT/M2M*. https://homieiot.github.io/. [Accessed 10.03.2020].

[54] I. Fette und A. Melnikov. *The WebSocket Protocol*. RFC 6455. https://www.rfc-editor.org/rfc/rfc6455.txt: Internet Engineering Task Force (IETF), Dez. 2011.

[55] V. Wang, F. Salim und P. Moskovitas. *The Definitive Guide to HTML5 WebSocket*. 1. Aufl. Apress, Feb. 2013.

[56] *MDN web docs - The WebSocket API (WebSockets)*. https://developer.mozilla.org/en-US/docs/Web/API/Websockets_API. [Accessed 09.03.2020].

[57] A. Banks, E. Briggs, K. Borgendale und R. Gupta. *MQTT Version 5.0*. Techn. Ber. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html: OASIS, März 2019.

[58] *MQTT man page*. https://mosquitto.org/man/mqtt-7.html. [Accessed 10.03.2020].

[59] T. H. Team. *MQTT Publish, Subscribe and Unsubscribe - MQTT Essentials: Part 4*. https://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe/. [Accessed 10.03.2020]. Feb. 2015.

[60] *Eclipse Mosquitto - An open source MQTT broker*. https://mosquitto.org/. [Accessed 10.03.2020].

[61] *mqtt wiki - Design principles*. https://github.com/mqtt/mqtt.github.io/wiki/Design-Principles. [Accessed 10.03.2020].

A. Anhang

A.1. meso-connect specification

A.1.1. Intro

`meso-connect` is a convention for MQTT communication which is heavily based on the Homie convention for IoT-devices.
The base of every meso-connect communication is a common configuration file.
Analog to Homie, the topic layout follows the pattern

```
meso-connect / device / node / property
```

where:

- `meso-connect` is the root namespace
- A `device` is a participator of the MQTT communication, like a micro controller or a exhibition workstation.
- A `node` is a functionality of the device, like for example a LED Panel.
- A `property` is one property of the node, like brightness.

A configuration file for that would look like this:

```
{
  "broker": {
    "address": "mqtt.meso.design",
    "port": "1883",
    "username": "meso",
    "password": "vvvv"
  },
  "devices": [
    {
      "id": "arduino",
      "nodes": [
        {
          "id": "led-panel",
          "properties": [
            {
              "id": "brightness",
              "datatype": "float",
              "unit": "%",
              "format": "0:100",
              "settable": "true",
              "retained": "true"
            }
          ]
        }
      ]
    }
  ]
}
```

with the resulting topics:

```
meso-connect / arduino / $state → "ready"

meso-connect / arduino / led-panel / brightness → "22"
meso-connect / arduino / led-panel / brightness / $error → "error message"
```

The main difference to the Homie convention is that we are letting go of the auto-discovery feature in favor of the common configuration file.

- For more information see Configuration
- Or see a full example at Example

[[*TOC*]]

A.1.2. Configuration

The base of every meso-connect communication is the configuration file. It is of the JSON format and it is the same file for every device involved in the communication.
The blueprint of the config file looks like this:

```
{
  "broker": {
    "address": "<brokeraddress>",
    "port": "<brokerport>",
    "username": "<username>",
    "password": "<password>"
  },
  "devices": [
    {
      "id": "<deviceid>",
      "nodes": [
        {
          "id": "<nodeid>",
          "properties": [
            {
              "id": "<propertyid>",
              "datatype": "<propertydatatype>",
              "unit": "<propertyunit>",
              "format": "<rangeOfPayloads>",
              "settable": "<isSettable>",
              "retained": "<isRetained>"
            }
          ]
        }
      ]
    }
  ]
}
```

Which results in topics of the shape

`meso-connect / device / node / property`

Note: every value in the config file must be of type `string`.

Topic IDs

Analog to Homie. See https://homieiot.github.io/specification/#topic-ids.
`$` is reserved for attributes.
The base topic has the id `meso-connect` by default. If needed, it should be changeable.

Payload

Mostly analog to Homie. Important is that every payload MUST be send as a UTF-8 encoded string. The value published MUST be valid for the respective property/attribute type. For the list of types see: https://homieiot.github.io/specification/#payload
To that list we are adding:

JSON *todo: add JSON type*

QoS and retained messages

Analog to Homie. See https://homieiot.github.io/specification/#qos-and-retained-messages

Last will

Analog to Homie. See https://homieiot.github.io/specification/#last-will

Topology

Mostly analog to Homie. See https://homieiot.github.io/specification/#topology.

The topology is all about

- devices
- nodes
- properties
- attributes

IMPORTANT: In contrast to Homie we are letting go of most of the attributes being published on topics because they are readable from the config file.
The only one we are keeping is the device attribute `$state` because it is dynamically set:

```
meso-connect / devicename / $state -> e.g "init"
```

On top we are adding the property attribute `$error`:

```
meso-connect / devicename / nodename / propertyname / $error
-> e.g. "wrong type. 'String' expected."
```

on which error messages are published per property. For example when one participant is trying to set a value of wrong type.

Broadcasts

todo: think about broadcasts

A.1.3. Example

For better understanding let's have a look at a full example. The scenario includes a kiosk terminal application to which a credit card terminal as well as a coin slot should be attached, so visitors can make donations and get visual feedback in the application.

Consider following configuration file:

```
{
  "broker": {
    "address": "mqtt.meso.net",
    "port": "1883",
    "userid": "testuser",
    "password": "password123"
  },
  "devices": [
    {
      "id": "donation-controller",
      "nodes": [
        {
          "id": "card-terminal",
          "properties": [
            {
              "id": "donation-amount",
              "unit": "#",
              "datatype": "float",
              "format": "0:20",
              "settable": "true",
              "retained": "true"
            },
            {
              "id": "donate",
              "datatype": "string",
              "settable": "true",
              "retained": "false"
            },
            {
              "id": "donate-success",
              "datatype": "string",
              "settable": "false",
              "retained": "false"
            },
            {
              "id": "donate-error",
              "datatype": "string",
              "settable": "false",
              "retained": "false"
            }
          ]
        },
        {
          "id": "coin-slot",
          "properties": [
            {
              "id": "coin-inserted",
              "datatype": "boolean",
              "settable": "false",
              "retained": "false"
            }
          ]
        }
      ]
    },
    {
      "id": "sharing-station-app",
      "nodes": []
    }
  ],
  "broadcasts": [
    "alert",
    "error"
  ]
}
```

So after defining our connection to the MQTT broker we are defining our two devices. One being the application (which is a computer) with the id `sharing-statinon-app` and the other being a micro controller with the id `donation-controller`.
Note that even though the application has no nodes itself it MUST be defined in the configuration file in order to participate in the communication.
Further, we are defining two nodes for the donation-controller being the credit card terminal `card-terminal` and the coin slot `coin-slot`.
The coin slot only has one property, which is a event property, on which the coin slot publishes a message every time a coin is inserted. The event characteristic is defined trough the `retained` attribute being `false`. `settable` being `false` means that the event is only emitable by the coin-slot.
The card-terminal defines multiple properties. The first one being the `donation-amount`, which is settable and retained. So this is the topic on which the application sets the selected donation amount on the credit card terminal. The donation process can get started trough the `donate` property. When the donation was successful the card-terminal emits a message on the `donate-success` property and when there was an error on the `donate-error` property.
todo: explain broadcasts

A.2. meso-connect-js specification

meso-connect-js is the JavaScript implementation of meso-connect. It can be used both in the browser and in Node.js.
For using it as a npm package in your project add it with

```
npm install git+ssh://git@git.meso.design:meso-connect/meso-connect-js.git
```

A.2.1. API

Class: Connection

new Connection(configFile, deviceId, [brokerConfig])
Create main connection object

- params:
  - `configFile`: the json file containing the meso-connect configuration
  - `deviceId`: own id of devicewhich is being set up
  - optional: `brokerConfig`: a json object containing a broker configuration which will override the configuration from the configFile

- returns:
  - a Connection object on successful setup
- throws error:
  - when config file is malformed
  - when deviceId is not present in the configFile

Connection.connect()
Connect to mqtt broker

- returns: Promise
  - resolves when connection has been successfully initialized
  - rejects when an error occurs while connecting

Connection.end()
Close the connection to mqtt broker

- returns: Promise
  - resolves when connection has been successfully closed
  - rejects when error occurs on closing the connection

Connection.publish(propertyId, value)
Publish a value for specified property

- params:
  - `propertyId`: a property id from the config file for which a value is to be published
  - `value`: the value to be published
- returns: Promise
  - resolves when successfully published
  - rejects when
    - propertyId is not a string
    - propertyId is not present in config file
    - propertyId is no own property and is not settable
    - value is not formatted as string
    - value is not of specified type
    - value is of wrong format

        * there is a problem with the connection

Connection.subscribe(propertyId)
Subscribe to specified property or array of properties

- params:
  - `propertyId`: a property id or an array of property ids from the config file which should be subscribed to
- returns: Promise
  - resolves when successfully subscribed
  - rejects when
    * propertyId is not a string or array
    * propertyId or one of propertyIds is not present in the config file
    * already subscribed to propertyId or one of propertyIds
    * there is a problem with the connection

Connection.unsubscribe(propertyId)
Unsubscribe specified property or array of properties

- params:
  - `propertyId`: a property id or an array of property ids from the config file which should be unsubscribed
- returns: Promise
  - resolves when successfully unsubscribed
  - rejects when
    * propertyId is not a string or array
    * propertyId or one of propertyIds is not present in the config file
    * propertyId or one of propertyIds has not been subscribed to
    * there is a problem with the connection

Connection.on(eventType, callback)
Initialize event handlers for different events

- params:
  - `eventType`: type of event that has been emitted
  - `callback`: the event handler callback function

- events:
  - an event with the propertyId as the eventType will be emitted for every incoming message on a property which has previously been subscribed to
  - `connect`: when connection to broker has been established
  - `reconnect`: on every attempt to reconnect to the broker
  - `error`: when there has been an connection error
  - `close`: when connection to broker has been closed
  - `offline`: when the client is offline