Integration von Web-Realtime in CMS-basierte Web Applikationen

Julian Raab

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im September 2017

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 25. September 2017

Julian Raab

Inhaltsverzeichnis

Kurzfassung

Web-Nutzer erwarten nicht mehr bloß statische Inhalte, sondern Webseiten entwickeln sich immer mehr zu dynamischen und kollaborativen Web-Anwendungen. Wenn Web Content Management Systeme diese Anforderungen bewältigen können sollen, ist es notwendig, Echtzeit-Benutzererfahrung (*Realtime User Experience*) darin zu etablieren und einige der vorhandenen Content Management Konzepte zu überdenken. Web-Realtime forcierende Technologien, wie HTML5 WebSockets, bieten Funktionalitäten zur Erstellung dieser Benutzererfahrung. Auf der LAMP-Technologieplattform basierende Server-Umgebungen sind jedoch nicht für bidirektonale Kommunikation, oder auch nur für das Senden von Daten von Server zu Client, ausgelegt. Dies führt dazu, dass die Implementierung jeglicher Art von Anwendung, die auf Benutzerinteraktion oder bidirektionaler Kommunikation zwischen CMS und Benutzer basiert, in solchen Umgebungen im Allgemeinen vermieden wird. Der Zweck dieser Arbeit ist es, den Prozess und die Voraussetzungen für die Integration von Technologien des Realtime Webs in ein existierendes CMS zu analysieren, mit dem Fokus auf CMS-Entwickler, CMS-Redakteuren und Benutzer der Endanwendungen. Folglich wird ein neues Kommunikationskonzept erstellt, auf welchem eine Programmbibliothek aufgebaut wird, welche als globale Schnittstelle dient, um die Web-Realtime Funktionalitäten innerhalb des Programmcodes des CM-Systems nutzen zu können und somit die gewünschte *Realtime User Experience* in die diversen Komponenten eines CM-Systems integrieren zu können.

Abstract

Web users do no longer expect to be served only static content, but websites are becoming highly responsive, collaborative web applications. If Web Content Management Systems should be able to handle those requirements, there is no way around integrating real-time user experience and rethinking content management concepts. Web Real-time supporting technologies like HTML5 WebSockets provide means for introducing this experience. LAMP-stack based server environments are typically not built for bidirectonal communication or even only pushing data from the server to the client. As a result the implementation of any kind of application which depends on user to user or CMS to user interaction gets generally avoided in such environments. The purpose of this thesis is to analyse the process and the requirements of joining an existing WCMS with technologies of the real-time web, targeting developers, content managing and content consuming users. This results in the creation of a new communication concept, on which a program library is set up, which serves as a global interface in order to use the web real-time functionalities within the program code of the CMS and thus introdurce the desired real-time user experience in various components of the CMS.

Kapitel 1

Einleitung

Webseiten entwickeln sich zu dynamischen und auch kollaborativen Web-Applikationen und Benutzer erwarten bereits stets aktuelle und personalisierte Inhalte basierend auf den neuesten Datenbeständen, wie auch soziale Interaktionmöglichkeiten. Grundvoraussetzungen, um diese Funktionen zu ermöglichen, sind Technologien des Realtime Webs und somit bidirektionale Datenübertragung zwischen Server und Client. Etablierte Technologien wie *XHR*[1] senden Server-Anfragen asynchron zur Benutzereingabe und ermöglichen somit eine dynamischere Erscheinung einer Web Applikation. Diese asynchronen Anfragen haben allerdings Einschränkungen hinsichtlich der Kommunikationsabfolge, so könnte der Server nicht Daten an den Client senden, ohne zuvor eine explizite Anfrage von diesem erhalten zu haben. Bidirektionale Kommunikation zwischen Client und Server wäre somit nicht möglich. Content Management Systeme bilden laut [36] zum Zeitpunkt des Schreibens dieser Arbeit die Basis für über 48% aller veröffentlichten Webseiten und der Trend ist heute, wie auch über die letzten Jahre hinweg stetig im Steigen begriffen (siehe Abb. 1.1). Dieser Entwicklung zum Trotz bieten allerdings nur wenige dieser Systeme Ansätze zur Integration von *Realtime User Experience*. Beim Versuch, diese beiden Technologien zusammenzuführen, ergeben sich mehrere Problemstellungen:

- Was sind die technologischen Voraussetzungen und welche Kommunikationskonzepte könnten Anwendung finden bei der Integration von Web-Realtime in ein bestehendes, auf der LAMP-Technologieplattform basierendes, Web Content Management System?
- Wie und an welchen Anknüpfungspunkten kann Web-Realtime Funktionalität in ein Web Content Management System integriert werden?
- Ergeben sich neue Möglichkeiten oder Vorteile bzw. Nachteile aus der Nutzung von Web-Realtime Technologien in Kombination mit Web Content Management Systemen?

[1] Abk.: XMLHttpRequest

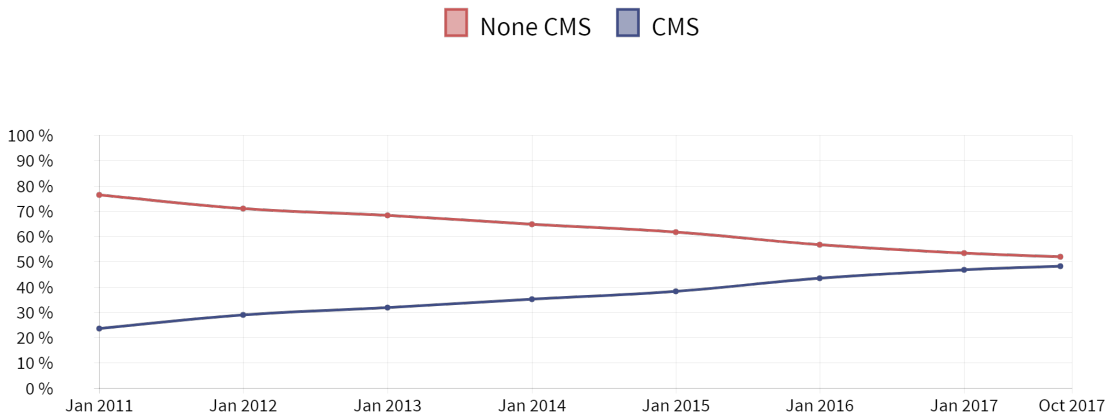


Abbildung 1.1: Anteil der Verwendung von CM-Systemen zum Betreiben von Webseiten im historischen Verlauf seit 2011 nach [36].

1.1 Lösungsansatz

Mit der Einführung von HTML5 wurden neue nun standardisierte Technologien zur Realisierung von Web-Realtime vorgestellt, von welchen WebSockets als gewählter Lösungsansatz in dieser Arbeit hervorgehen. Hierbei muss der Client keine explizite Anfrage an den Server stellen, um Daten zu erhalten, sondern lediglich eine Verbindung öffnen. Diese etablierte persistente Verbindung ermöglicht dann dem Server, Nachrichten zu jeder Zeit an den Client zu senden, ohne auf einen erneuten Verbindungsaufbau warten zu müssen. Veraltete *User Agents* oder Komponenten, wie Firewalls oder Proxies innerhalb des Netzwerks, unterstützen die WebSocket-Technologie teilweise noch nicht und so wäre es, für eine verlässliche Web-Realtime Verbindung, nötig Rückfalllösungen[2] zu berücksichtigen. Um diese nicht selbst implementieren zu müssen, gibt es bereits diverse Programmbibliotheken, welche die nötigen Rückfalllösungen bereitstellen. Für die Integration dieser Verbindungsvariante in auf der LAMP-Technologieplattform basierende CM-Systeme muss ein angepasstes Kommunikationskonzept entwickelt und eine dazugehörende Programmbibliothek implementiert werden. Diese Programmbibliothek soll nach dem Programmierparadigma *Publish-Subscribe* funktionieren und ermöglichen, über eine dritte Komponente, welche als Web-Realtime Dienst fungiert, Daten aus dem CMS zu allen verbundenen Clients senden zu können. Im Fallbeispiel ist diese Komponente als *Node.js*[3] Server abgebildet. Die CMS-spezifischen Logiken sollen in eine CMS-Erweiterung (*Plugin*) für das, als Fallbeispiel verwendete, CMS *Pimcore*[4] gekapselt werden.

[2] Fallback
[3] https://nodejs.org/
[4] https://www.pimcore.org/

1.2 Struktur der Arbeit

Das zweiten Kapitel dieser Arbeit ist der Vorstellung grundlegender Terminologien und fundamentalem Wissen, welches für das weitere Verständnis notwendig sind, gewidmet. Danach wird der aktuelle Stand der Technik im Realtime Web Bereich und einige dazugehörende Dienste vorgestellt. Außerdem werden Anwendungsfälle von Web-Realtime innerhalb von CM-Systemen und deren Endbenutzer Anwendungen, wie auch bestehenden Ansätzen zu deren Umsetzung dargelegt. Die Anforderungen für die Umsetzung der Web-Realtime Integration und das eigene Konzept werden im darauf folgenden Kapitel definiert und begründet. In Kapitel fünf wird die Architektur und die Umsetzung der Implementierung, der Schnittstelle und der Anwendungsfälle, im Detail erklärt. Das vorletzte Kapitel beschäftigt sich mit der Evaluierung der definierten Anforderungen und möglicher Verbesserungen bzw. Erweiterungen. Abschließend wird eine Zusammenfassung der Ergebnisse und ein kurzes Fazit bereitgestellt.

Kapitel 2

Technische Grundlagen und Terminologie

Beschäftigt man sich mit Web-Entwicklung, bemerkt man sehr bald, dass es sich hierbei um ein schnelllebiges Forschungsgebiet handelt. Begrifflichkeiten und Fakten die zum jetzigen Zeitpunkt noch als klar definiert gelten mögen, können sich sehr schnell verändern bzw. weiterentwickeln und somit womöglich zu Missverständnissen führen. In diesem Kapitel werden grundlegende Konzepte, die zum weiteren Verständnis der eigentlichen Thematik notwendig sind, aufgezeigt. Ein grundlegendes Verständnis für IT-Infrastrukturen und Web-Entwicklung wird dennoch vorausgesetzt.

2.1 Terminologie

Der folgende Abschnitt beschäftigt sich mit fachspezifischen Terminologien zum klareren Verständnis der weiteren Arbeit.

2.1.1 Web Applikation

Eine *Web Applikation* (oder auch Webanwendung, Web App, Online-Anwendung, usw.) ist eine Anwendung, welche, anders als lokal installierte Desktopanwendungen, nach dem *Client-Server-Paradigma* arbeitet. Hierbei stellt die Anwendung, die den Kommunikationswunsch hegt (*Client*), eine Anfrage an eine korrespondierende Anwendung auf einem entfernten Rechner (*Server*), in der sie ihren Kommunikationswunsch (*Request*) formuliert. Der Server antwortet daraufhin mit der angefragten Ressource (*Response bzw. Reply*) oder stellt den gewünschten Dienst bereit [13, S. 722]. Dieser Definition zufolge kann jede Webseite als Web Applikation bezeichnet werden, wenn auch der Begriff heute vor allem von Interaktions-basierten und somit JavaScript-basierten Webseiten geprägt wird.

2.1.2 Realtime Web

Als *Realtime Web* bezeichnet man das Web unter Verwendung von Technologien, welche es ermöglichen, neue Informationen zu empfangen, sobald sie verfügbar sind, anstatt periodische Anfragen an eine Quelle stellen zu müssen.

Web-Realtime

Die Möglichkeit der bidirektionaler Client-Server Kommunikation wird in dieser Arbeit auch als *Web-Realtime* bezeichnet. Im Wissenschaftsgebiet des *Real-time Computing* können Anwendungen bzw. Aufgaben grundsätzlich als *hard*, *firm* oder *soft* klassifiziert werden, wobei sich diese Einteilung auf die Auswirkung der Einhaltung einer definierten Antwort-Frist (*Deadline*) bezieht. So wäre ein Überschreiten der Deadline bei einer als *hard real-time* klassifizierten Applikation in ihrem Kontext katastrophal (z. B. System Crash). *Firm* würde bedeuten, dass das Ergebnis nach Ablauf der Deadline keine Relevanz mehr hat, allerdings sind wenige Missachtungen tolerierbar – wenn sie auch die Qualität des Dienstes mindern. Ist ein System weder als *hard* noch als *firm* definiert, bezeichnet man es als *soft*, was bedeutet, dass die Relevanz des Ergebnisses über Zeit abnimmt [19, 51]. Dieser Definition entsprechend können Web-Realtime Anwendungen als *soft*, oder *firm* klassifiziert werden, die übermittelten Daten könnten hier z. B. Status-Aktualisierungen, Chat-Nachrichten oder ein Gebote in einem Online-Trading-System sein. Die Deadline ist somit bei kollaborativen Anwendungen oft subjektiv je nach Anwendung und Benutzer.

Realtime User Experience

Wörtlich übersetzt bedeutet *Realtime User Experience* (oder auch Realtime UX) *Echtzeit Benutzererfahrung*. Der Term bezieht sich in dieser Arbeit auf die Zeit zwischen dem Auftreten eines Ereignisses und dessen Wahrnehmung durch einen Benutzer [11, S. 19]. Im Kontext von Web Applikationen bedeutet dies meist eine serverseitige Änderungen des Datenbestandes und der Benachrichtigung eines Benutzers über diese. Ist diese Zeitspanne so kurz, dass sie im jeweiligen Anwendungsfeld keine relevante Verzögerung aufweist, spricht man von *Realtime User Experience*.

2.1.3 User Agent

Der Begriff *User Agent* bezieht sich auf einen Teilaspekt von Web Clients, welche eine Anwendungsumgebung für Web Applikationen zur Verfügung stellen. In der Regel ist diese Umgebung ein Web Browser wie z. B. Google Chrome, Mozilla Firefox, Microsoft Edge, Safari usw. Es können aber auch andere Anwendungen als User Agent fungieren, wie z. B. die *Webcrawler* von Suchmaschinen.

2.2 Web-spezifische Grundlagen

In diesem Abschnitt wird grundlegendes Wissen zu CM-Systemen und Kommunikationsabläufen im Web vorgestellt.

2.2.1 Content Management

Content Management ist kein digitales Konzept, allerdings brachte der Aufstieg des World Wide Webs die Möglichkeit für beinahe jedermann, Inhalte problemlos zu erstellen und zu veröffentlichen und somit explodierte die Notwendigkeit von Content Management [2, Kapitel 1].

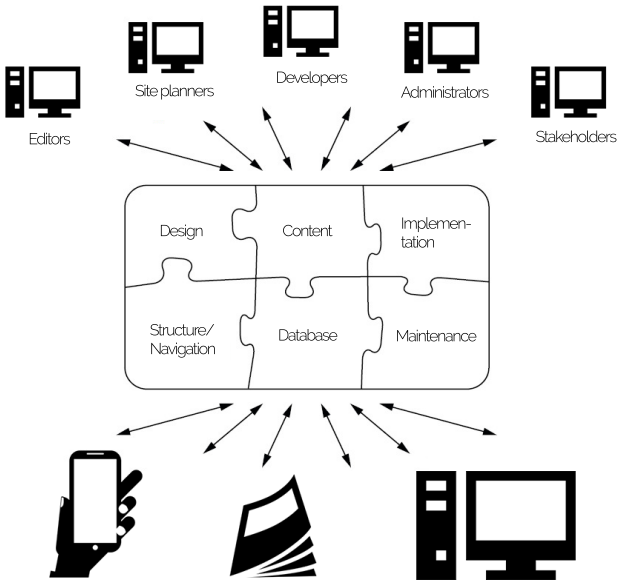


Abbildung 2.1: Einzelne Teile formen ein gemeinsames Ganzes, das *Content Managment System*. Verschiedene Anwender können das System verwenden, um Inhalte für diverse Ausgabekanäle zu verwalten. Bildquelle [4, S. 324].

Web Content Management Systeme

CM-Systeme bieten Redakteuren einer Anwendung einen vereinheitlichten Weg, Inhalte aus diversen Kompetenzbereichen zu verwalten. Bei WCM-Systemen[1] passiert dies meist über einen Management-Bereich (z. B. *www.mydomain.at/admin*), welcher eine vorherige Authentifizierung per Login erfordert. Ein WCMS besteht aus vielen Teilen, die alle innerhalb des Systems selbst autonom sein können, aber für einen nicht technischen Redakteur als Ganzes wahrgenommen werden (siehe Abb. 2.1). Diese Systeme sind für die Verwendung in einer Mehrbenutzerumgebung konzipiert. Ein WCMS ist eine Art *Groupware*, um gemeinsam Inhalte für verschiedene Ausgabemedien via einer Web-Benutzeroberfläche zu verwalten. Beispiele für solche Systeme sind Wordpress, Drupal, Typo3 oder auch Pimcore. Ein jedes dieser Systeme zielt gemeinhin auf einen speziellen Einsatzzweck, wie z. B. Blog-, Shop- oder DAM-Systeme[2], ab [4, S. 324–325].

2.2.2 Kommunikationsabläufe im Web

Das Web verbindet heute Computer, Smartphones, Fernseher und viele andere Geräte. Abstrahiert man die Kommunikationsabfolgen, sind aus technischer Sicht hierbei hauptsächlich zwei Akteure von Relevanz, und zwar Web Client und Web Server. Dem *Client-Server-Paradigma* folgend fragt gemeinhin der Client Daten vom Server an, wenn die Anfrage korrekt ist, sendet dieser die jeweiligen Informationen zurück. Um einen universellen und effizienten Datenaustausch zu gewährleisten, muss die Kommunikation nach

[1] Abk.: Web Content Management
[2] Abk.: Digital Asset Managment

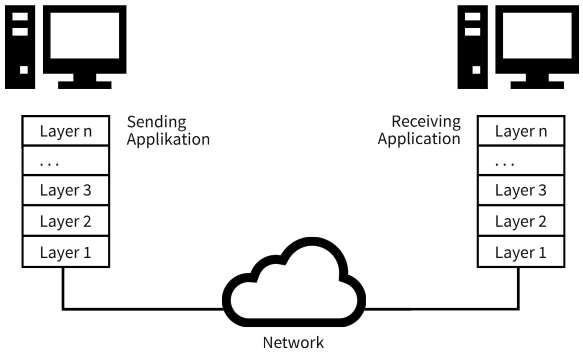


Abbildung 2.2: Die Datenübertragung erfolgt vertikal über den Protokollstapel des Schichtenmodells. Bildquelle [13, S. 34].

bestimmten Regeln ablaufen – sogenannten Kommunikationsprotokollen. Diese Protokolle sind, gegliedert nach Verantwortungsbereich im Rahmen der Netzwerkkommunikation, in verschiedene hierarchische Schichten unterteilt. Die Kommunikation erfolgt vertikal über den Protokollstabel (siehe Abb. 2.2). Bei den höher gelegenen Schichten (Anwendungsschicht) unterscheidet man, wie in [13, S. 39] beschrieben, grundsätzlich zwischen zwei Diensten:

Verbindungslose Dienste: Jede Nachricht wird als einzigartig und unabhängig gehandhabt, ein Beispiel dazu wäre das zustandslose Protokoll *HTTP* [7]. Eine Analogie hierfür wäre postaler Briefverkehr.

Verbindungsorientierte Dienste: Nach einem erfolgreichem Verbindungsaufbau können, analog zu einem Telefonat, entlang dieser Verbindungen beide Parteien Nachrichten versenden und empfangen, ein Beispiel ist das *WebSocket* Protokoll [6].

2.2.3 HTML5-Communication Spezifikation

Um die technologischen Voraussetzungen zu erfüllen, *Realtime User Experience* zu etablieren, ist ein Ausbrechen aus dem klassischen Request-Response Schema, auf welchem ein Großteil der heutigen Web Applikationen noch basiert, unumgänglich. In den letzten Jahrzehnten wurden diverse Möglichkeiten eingeführt, Web-Realtime unter Einbüßen von Übertragunsleistung[3] und Konventions-Legitimität zu ermöglichen, wie z. B. via XHR-Polling, XHR-Long-Polling oder HTTP-Streaming. Mit der Einführung von HTML5 wurden im Abschnitt *Communication* [37] JavaScript-basierte APIs spezifiziert, welche effiziente Methoden zur Umsetzung von *Web-Realtime* ermöglichen:

- Server-Sent-Events,
- WebSocket API,
- WebRTC.

[3] Performance

Programm 2.1: *Server Sent Events*: EventSource API Event-Handling.

```
1 var list = new EventSource('list.php');
2
3 var addHandler = function(event){};
4 list.addEventListener('add', addHandler, false);
```

Server-Sent Events

Um Servern das Senden von Daten zum Client via einer langlebigen HTTP Verbindung, ähnlich dem *HTTP-Streaming*, zu ermöglichen, führt diese Spezifikation das `EventSource Inferface` ein. Die API arbeitet Ereignis-basiert[4] aufbauend auf dem ebenfalls in der HTML5 Spezifikation enthaltenen `MessageEvent Interface`. Nachrichten werden mit dem Mime-Type `text/event-stream` und ausnahmslos *UTF8* kodiert übertragen. Wenn die Verbindung abbricht, wird der User Agent versuchen, sich automatisch wieder zu verbinden. Die Wiederverbindungsversuche können gestoppt werden, indem ein *HTTP 204 No Content response code* gesendet wird. Ein Event Objekt hat immer ein Attribut `data` und ein Attribut `type`. Das `type` Attribut hat standardmäßig den Wert `message`. Es können aber auch eigene Ereignis-Typen definiert und separat gehandhabt werden (siehe Prog. 2.1). Die Verwendung dieser API ist im Gegensatz zu herkömmlichen Lösungen wie *XHR Polling* darauf ausgelegt, Netzwerk- und Client-Ressourcen optimal auszunützen, was zu Verbesserungen in Aspekten wie der Übertragunsleistung und Akkulaufzeit führen soll.

WebSockets

WebSockets bieten eine standardisierte Möglichkeit, bidirektionale Kommunikation über eine einfache voll-duplexe TCP-Verbindung zu realisieren. Die vom *W3C*[5] entwickelte `WebSocket API` baut dabei auf der gleichnamigen Protokoll Spezifikation [6] der *IETF*[6] auf [20, S. 9].

WebSocket Protokoll: Eine WebSocket Verbindung etabliert eine bidirektionale Verbindung zwischen Client und Server für asychronen Austausch diskreter Nachrichten, welche wiederum aus einem oder mehreren sogenannten *WebSocket-Frames* bestehen. Das WebSocket Protokoll ist ein TCP-basiertes Protokoll. Seine einzige Beziehung zum HTTP-Protokoll ist, dass der Verbindungsaufbau vom Server als Upgrade-Request interpretiert wird – dem sogenannten *Opening Handshake* [6, S. 10]. Die Adressierung erfolgt dabei per URL (z. B. ws://www.example.com) und unidirektional, was bedeutet, dass der Verbindungsaufbau immer vom Client initialisiert werden muss. Sobald eine Verbindung besteht, können beide Parteien zu jeder Zeit Daten senden und empfangen (siehe Tab. 2.1). Die Daten werden als Frames transportiert, jeder dieser Frames enthält einen 2–14 Bytes langen Header, welcher aus dem Operationscode zur Verbindungserstellung, den Nutzerdaten und Angaben zu deren Länge besteht [1, S. 190].

[4] Event-based
[5] Abk.: World Wide Web Consortium
[6] Abk.: Internet Engineering Task Force

Tabelle 2.1: Netzwerkeigenschaften der Protokolle TCP, HTTP und WebSocket im Vergleich nach [20, S. 37].

	TCP	HTTP	WebSocket
Adressierung	IP Adresse und Port	URL	URL
Simultane Übertragung	Voll-duplex	Halb-duplex	Voll-duplex
Inhalt	Byte-Streams	MIME-Nachrichten	Text und binäre Nachrichten
Diskrete Nachrichten	Nein	Ja	Ja
Verbindungsorientiert	Ja	Nein	Ja

WebSocket API: Die API ist eine JavaScript-basierte Programmier-Schnittstelle, welche es Applikationen ermöglicht das WebSocket Protokoll einfach zu verwenden und alle Ereignisse, die während des Verbindungs-Zyklus ausgeführt werden, zu verwalten. Es werden demnach diverse Methoden zur Verfügung gestellt, um Event-basiert auf alle diese Ereignisse reagieren zu können:

`OnOpen` Ein Ereignis-Listener, der aufgerufen werden soll, wenn sich der `readyState` der WebSocket-Verbindung auf `OPEN` ändert. Dies bedeutet, dass die Verbindung zum Senden und Empfangen von Daten bereit ist.

`OnMessage` Das `Message`-Event wird immer dann ausgeführt, sobald eine Nachricht empfangen wurde. Die WebSocket API gibt hierbei Applikationen keine Informationen über die einzelnen WebSocket-Frames Preis, eine Nachricht kann aus einem oder mehreren Frames bestehen [20, S. 44]. Eine WebSocket Verbindung unterstützt neben Text auch noch binäre Daten, welche entweder als *BLOB*[7] oder *ArrayBuffer* übertragen werden können.

`OnError` Ein `Error`-Event tritt auf, wenn ein unerwarteter Fehler auftritt. Dieses Event löst auch ein Schließen der Verbindung aus, und somit wird kurz darauf auch ein `close` Event ausgeführt.

`OnClose` Das `Close`-Event wird bei einer Trennung der Verbindung ausgeführt, sowohl bei einem erfolgreichen *Closing-Handshake*, als auch bei einem unerwarteten Verbindungsfehler.

Die verschiedenen Events werden zu verschiedenen Verbindungszuständen ausgeführt. Diese Zustände können durch den Parameter `readyState` abgefragt werden und lauten *CONNECTING, OPEN, CLOSING* und *CLOSED*. Der Parameter `bufferedAmount` gibt an wie viele Bytes sich in der Event-Loop befinden, aber noch nicht über das Netzwerk übertragen wurden und dient somit zur Kontrolle der Verbindungsauslastung bzw. -überlastung.

[7] Abk.: Binary Large Object

WebRTC

WebRTC[8] ist ein gesamtes Paket von JavaScript APIs, welche Möglichkeiten bieten, um Multimedia-Inhalte aufzunehmen und diese live und direkt von einem Client zu einem anderen Client[9] zu übertragen [5, S. 155]. Diese APIs werden, wie in [9, S. 113] beschrieben, oft kombiniert um eine Web-Realtime Verbindung zwischen zwei Clients herzustellen:

> *WebRTC is actually a set of specifications that are often used together to create a real-time communication link.*

Die direkte Datenübertragung von Client zu Client ist ein vollkommen neuer Ansatz im Vergleich zum üblichen Client-Server Modell des World Wide Webs. Dieser unterscheidet sich dabei auch grundlegend von dem von Server-Sent-Events und einer WebSocket-Verbindung. *WebRTC* kümmert sich neben der Verbindung an sich auch noch um diverse andere Aspekte, welche für performantes Media-Streaming notwendig sind, wie zum Beispiel *Media Codecs*, Echokompensation[10], oder Verlust von Datenpaketen.[11] Die Peer-to-peer Übertragung kann signifikante Vorteile bezüglich der Skalierbarkeit einer Web-Anwendung bieten, da die Ressourcen des Servers geschont werden [5, S. 155]. WebRTC ist optimiert für maximale Geschwindigkeit der Datenübertragung, Anwendungsgebiete hierfür wären zum Beispiel Online-Games, Video-Chats oder Live-Übertragungen usw. In der API Spezifikation [39] wird allerdings noch ein weiterer Anwendungsfall definiert:

> *One core component is to enable real-time media like audio and video, a second is to enable data transfer directly between clients.*

Die API Spezifikation beinhaltet P2P Streaming von Audio-/Video-Daten, P2P Daten-Verbindungen und auch Media Daten Zugriff des User Agents und lässt sich wie in [9, S. 114] beschrieben gliedern:

Media-Stream API Ermöglicht den Zugriff auf Video und Audio-Inputgeräte des Endanwenders direkt über den jeweiligen Browser (sofern dieser die API unterstützt), die JavaScript Funktion hierfür lautet: `getUserMedia()`.

Peer-Connection API Dieses stellt eine *WebRTC*-Verbindung zwischen einem lokalen Computer und einer entfernten Partei dar und bietet Methoden, zur Wartung, Überwachung und Schließung der Verbindung, sobald sie nicht mehr benötigt wird. Es stellt Funktionalitäten zur Erstellung eines *Peer-to-Peer* Objekts bereit, welches die *ICE*-Verbindung verwaltet.

Data-Channel API Ermöglicht es, außer Video oder Audio Inhalten, auch andere beliebige Datenpakete zu übertragen, was somit eine direkte Daten-Verbindung zwischen zwei User Agents ermöglicht. Die Struktur der API ist dabei nach der der WebSocket API abgebildet [35].

[8] Abk.: Web Real-time Communication
[9] Peer-to-Peer (P2P)
[10] Acoustic Echo Cancellation (AEC)
[11] Packet Loss Concealment (PLC)

Auf Netzwerk-Protokoll Ebene baut *WebRTC* unter anderem vor allem auf die Protokolle *ICE* [17] und *RTP* [18]. *ICE* ist für die Erstellung der *Peer-to-Peer* Verbindung verantwortlich, ein Protokoll, welches dafür konzipiert wurde, den besten Weg zu finden um Daten von einem Client zu einem anderen zu übertragen, trotz anderer *NAT*[12] Geräten wie Routern, Gateways oder Firewalls, welche auf der Verbindungsstrecke passiert werden würden [5, S. 158]. Ist die *Signalisierung* abgeschlossen, steht somit ein Signalkanal zwischen den beiden Gegenstellen bereit, dann ist *RTP* bzw. *SRTP* für die eigentliche Datenübertragung verantwortlich [14, S. 325].

[12] Abk.: Network Address Translation

Kapitel 3

Stand der Technik

Ein *Content Managment System* ist, wie jede Web Anwendung, eingebettet in eine passende Software-Infrastruktur. Eines der bekanntesten Modelle ist die *LAMP*[1] Technologieplattform, welche als Basis für viele bekannte CM-Systeme dient, unter anderem auch für das in dieser Arbeit als Fallbeispiel verwendete System *Pimcore*. CM-Systeme haben, wie in Abschnitt 2.2.1 beschrieben, diverse Kompetenzbereiche, für die es bei der Integration von Web-Realtime programmatikalisch oft unterschiedlichste Herangehensweisen benötigt. Letztendlich wird allerdings immer ein Dienst benötigt, der die jeweiligen Nutzdaten[2] zum Client sendet.

3.1 CMS Anwendungen

Für einige Content Management Systeme, welche auch auf der LAMP Technologieplattform basieren, bestehen Plugins unterschiedlicher Komplexität, welche versuchen, *Realtime User Experience* für gewisse Teilbereiche des jeweiligen CM-Systems zu ermöglichen, wie z. B. Chat-Funktionen als Plugin für Wordpress[3], oder auch als Drupal-Modul.[4] Diese Erweiterungen arbeiten allerdings meist auf der Endanwender-Ebene (z. B. Chats, Notifikationen, Statistiken usw.) und stellen keine universelle Schnittstelle für das CMS an sich zur Verfügung.

3.1.1 Interne Anwendung

Anwendungsfälle innerhalb der CMS Adiministrationsoberfläche, welche prädestiniert für die Anwendung von Web-Realtime wären, werden bis heute durch verschiedene Methoden, wie zyklischem Abfragen (*Polling*), Versionierung, Logging[5], Markieren oder Sperren von Inhalten, realisiert:

[1] Abk.: Linux, Apache, MySQL, PHP
[2] Data payload
[3] https://wordpress.org/plugins/bowob/
[4] https://www.drupal.org/project/bowob
[5] Informationen in einer persistenten Form protokollieren, z. B. in einer Textdatei.

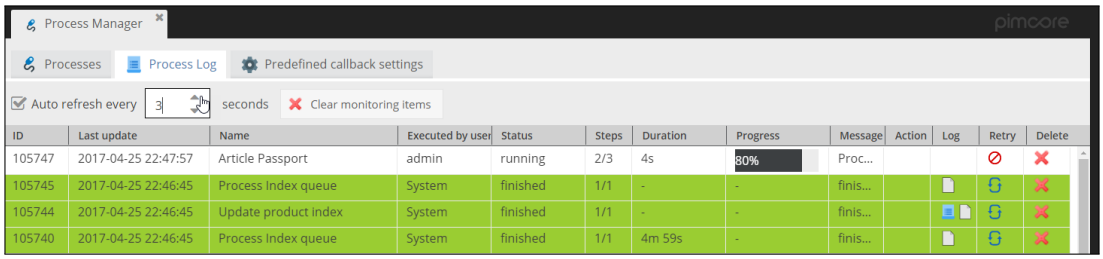


Abbildung 3.1: *Pimcore Process Manager Plugin*: Ein Neuladen des gesamten Inhalts ist notwendig, um den aktuellen Status der Prozesse abzurufen.

Konkurrierender Zugriff

Gleichzeitige Änderungen an Inhaltselementen in einem CMS resultieren in unterschiedlichsten Ergebnissen: Das Standardresultat von *Drupal* ist z. B. ein Fehler, durch welchen die durchgeführten Änderungen nicht gespeichert werden könnten [41]. Allerdings gibt es verschiedene Module, welche z. B. das Sperren von Inhalten (*record-locking*)[6], die Erstellung verschiedener Versionen (*revisions*)[7], ein Git-ähnliches Verhalten zum Auflösen von Änderungs-Konflikten[8], oder sogar Echtzeit-Änderungen[9] ermöglichen.

Serverseitige asynchrone Prozesse

Aufgaben, welche eine längere serverseitige Verarbeitung oder eine weitere Bestätigung durch andere Anwender benötigen (z. B. Import/Export für Produkt Daten), können in manchen Systemen parallel und asynchron zum eigentlichen CMS ausgeführt werden. Ob eine solche Aufgabe Fortschritte erzielt hat oder abgeschlossen ist, wird durch in regelmäßigen Abständen ausgeführte Requests (*Polling*) oder im schlechtesten Fall durch ein Client-seitiges manuelles Neuladen der Seite abgefragt. Ein Beispiel hierfür ist das *Pimcore Process Manager Plugin*, ein für Pimcore Partner zugängliches Plugin, welches es ermöglicht, komplexe *CLI-Skripte* manuell oder per Cron-Job auszuführen und deren Fortschritte und Status in der CMS Administrationsoberfläche zu überwachen (siehe Abb. 3.1).

System Benachrichtigungen

Bei der Verwendung eines WCMS gibt es viele Anlässe, den Anwender zu benachrichtigen. Da dies nicht möglich ist, werden die verfügbaren Informationen oft in eine Log-Datei geschrieben, ignoriert oder auch als E-Mail versendet. Beispiele hierfür sind das An- bzw. Abmelden von Benutzern, Systemänderungen durch einen andren Benutzer (z. B. installierte Plugins), Systemüberlastungen, verfügbare Software-Updates, usw.

[6] https://www.drupal.org/project/content_lock
[7] https://www.drupal.org/project/revision_fu
[8] https://www.drupal.org/project/conflict
[9] https://www.drupal.org/project/websockets

Statistiken und Analysen

Oft ist es von Interesse, Analysedaten über die Verwendung des CMS zu erhalten. Beispiele hierfür wären Elemente, die sich derzeit in Bearbeitung befinden, aktive Nutzer oder auch die Server-Ressourcenauslastung usw. Pimcore bietet hierfür standardmäßig auf der Übersichtsseite (*Dashboard*) eine statische Ansicht der letzten Änderungen der Kernkomponenten des CM-Systems (Objekte, Dokumente, Assets). Des Weiteren ist es möglich, Statistiken zu benutzerspezifischen Daten hinzuzufügen – sog. *Custom Reports*. Aber auch andere CM-Systeme bieten verschiedene Lösungen. Drupal 8 beinhaltet z. B. ebenfalls standardmäßig ein Statistik Modul, für Typo3 existiert die sog. *StatCounter Extension*[10] und auch für Wordpress gibt es ein bereits äußerst ausgereiftes Plugin[11], welches zahlungspflichtig sogar Web-Realtime Funktionen bereitstellt.

3.1.2 Endbenutzer Anwendungen

Über die Anwendungsgebiete von Web-Realtime Technologien innerhalb eines CMS hinaus ist die Ermöglichung von Realtime Benutzererfahrung auch für immer mehr Endbenutzer Anwendungen eine unverzichtbare Anforderung. Aus der Verwendungsstatistik der Web-Realtime Dienste und Frameworks laut [33] geht hervor, dass Web-Realtime vor allem in Web Applikationen mit folgenden thematischen Schwerpunkten Verwendung findet: E-Business und E-Commerce, Medien und Unterhaltung, E-Learning, Online-Spiele, News, Social-Media usw. Selbst wenn die konkrete Umsetzung der Web-Realtime Integration in den verschiedenen Bereichen sehr stark voneinander differenzieren mag, ist erkenntlich, dass alle diese Bereiche vorwiegend inhaltsbasierte Applikationen mit hoher Datenkomplexität beinhalten, was sie wiederum gleichermaßen prädestiniert für die Verwendung eines CMS.

Einige der wichtigsten Aspekte der Web-Realtime Integration sind soziale Komponenten und Interaktionsmöglichkeiten, welche seit dem Aufstreben der sozialen Medien und Netzwerken immer mehr Bedeutung in Web Applikationen, unabhängig des thematischen Fokus, bekommen. Ein Beispiel hierfür ist die Entwicklung im Bereich des E-Business und E-Commerce: *Salesforce*[12], eine der bekanntesten Online-Lösungen für Kundenbeziehungsmanagement[13], integrierte mit der Einführung von Chatter[14] soziale Funktionalitäten, wie Kommentare und Beitragsaktualisierungen in ihr System, welche live für das Verwaltungsteam ersichtlich sind [16, Kap. 1]. Auch im Bereich des Geschäftsprozess-Managements[15] wird Web-Realtime zur live Beobachtung von Prozessstatus eingesetzt. Unternehmen stellen Chat-Funktionen bereit, um ihre Kunden mit dem Support-Team in Kontakt treten lassen zu können und Online-Shops könnten Lagerstände, Reservierungen, variable Preise, Lieferstatus uvm. live und konsistent synchronisieren. Aber auch in anderen Sparten wird Web-Realtime zu einem integralen Teil der Online-Präsenz: In Echtzeit aktualisierte Aktivitätenlisten (*Live-Ticker*) werden vor allem im Nachrichten- und Sport-Bereich eingesetzt um Anwender stets mit dem

[10] https://extensions.typo3.org/extension/ns_statcounter/
[11] https://de.wordpress.org/plugins/wp-statistics/
[12] https://www.salesforce.com/de/
[13] Customer Relation Management (CRM)
[14] https://www.salesforce.com/products/chatter/overview/
[15] Business Process Management (BPM)

aktuellsten Datenbestand der jeweiligen Web Applikation zu versorgen. Mehrbenutzer-Online-Spiele, mit Integration von sozialen Kontakten aus sozialen Netzwerken wie *Zynga*[16], wurden erst ermöglicht durch Web-Realtime innerhalb des Web Browsers. Über APIs bereitgestellte Echtzeit-Datenströme aus diversen Quellen, wie sozialen Medien (z. B. Twitter [48] oder Facebook [32]), Verkehrsdaten oder beliebige Sensordaten von Geräten aus dem *Internet der Dinge*[17] eröffnen immer mehr neue Umsetzungs-Möglichkeiten für Web Applikation.

3.2 Externe Web-Realtime Dienste

Auf externen Systemen betriebene Dienste bieten eine Schnittstelle, um Daten von Server zu Client senden zu können, ohne sich selbst um die Implementierung der Verbindung kümmern zu müssen. Der Client verbindet sich dabei mit dem Dienst z. B. via einer WebSocket-Verbindung, je nach bereitgestellter Client-API stehen bei Bedarf auch Rückfalllösungen wie XHR-Polling zur Verfügung. Der Server schickt dabei HTTP-Requests zu diesem Dienst, welcher die Daten zum Client weiterleitet. Die unterschiedlichen Anbieter wie *Pusher*[18], *Firebase*[19] oder *PubNub*[20] usw. bieten für die Client-seitige Handhabung der empfangenen Ereignisse ihre eigenen JavaScript-Bibliotheken an.

Diese Dienste sind vor allem dann sehr hilfreich, wenn die Ressourcen des eigenen Servers limitiert sind oder man nur eingeschränkt Zugriff auf diesen besitzt, wie z. B. bei *Shared-Webhosting* Umgebungen. Allerdings sind extern betriebene Web-Realtime Dienste immer mit Kosten verbunden, sobald man ein vertragsmäßiges Limit an Nachrichten bzw. gleichzeitigen Verbindungen überschreitet.

3.2.1 Pusher

Pusher versendet nach eigenen Angaben derzeit monatlich 6,5 Mrd. Nachrichten, ist einer der bekanntesten Anbieter der Branche und wird derzeit unter anderem von *GitHub*[21] oder der *New York Times*[22] verwendet. Der Dienst bietet über 30 APIs in verschiedensten Programmiersprachen für Client und Server, dabei setzt er auf ein *Kanal-System* (*Channels*), welches es ermöglicht, Daten einfach zu filtern und die Zugriffsrechte je nach Kanal-Typ zu verwalten. Jeder Kanal kann über eine eindeutige Zeichenkette (Kanalname) identifiziert werden, unterschiedliche Präfixe ermöglichen eine einfache Unterscheidung des Kanal-Typs. Kanal-Abonnements erfolgen als *AJAX*-Requests[23] und werden via der bereitgestellten JavaScript-API ausgeführt:

```
var pusher = new Pusher('app_key', {cluster: 'eu'});
var channel = pusher.subscribe('channel_name');
```

[16] https://zyngagames.com/
[17] Internet of Things (IOT)
[18] https://pusher.com/
[19] https://firebase.google.com/
[20] https://www.pubnub.com/
[21] https://github.com/
[22] https://www.nytimes.com/
[23] Abk.: Asynchronous JavaScript and XML

Programm 3.1: *Pusher*: Presence-Channels ermöglichen die Speicherung von Informationen über verbundene Benutzer.

```
presenceChannel.members.each(function(member) {
    var userId = member.id;
    var userInfo = member.info;
});
```

Abonniert ein Anwender einen Kanal (*Subscription*), wird dieser erstellt, falls er innerhalb der Applikation noch nicht existiert. Das Senden Senden von Events zu bestimmten Kanälen erfolgt serverseitig via der PHP-API von *Pusher*:

```
$pusher = new Pusher('APP_KEY', 'APP_SECRET', 'APP_ID');

if($pusher->trigger($channels, $event, $data, $socket_id)){
    // success
}
```

Zum Senden eines Events an einen bestimmten Kanal müssen die Funktionsargumente `$channels` (ein oder mehrere Kanalname), `$event` (Name des Event-Typs), `$data` (Nutzdaten) und `$socket_id` (optionaler Parameter, um spezifische Sockets auszuschließen) übergeben werden. Derzeit gibt es laut Dokumentation [49] drei verschiedene Kategorien von *Channels*:

Public Channels Diese Kanäle bieten keine Authentifizierung und sollten somit nur für öffentlich zugängliche Daten verwendet werden.

Private Channels Private Kanäle finden Verwendung, wenn Abonnements eines Kanals eingeschränkt sein sollen, oder nur unter Berücksichtigung einer Authentifizierung ermöglicht werden. Die Autorisierung des Zugriffsrechts erfolgt über einen konfigurierbaren HTTP-Endpunkt beim Aufruf von `channel.subscribe()`. Private Kanäle müssen laut Konvention den Prefix `private-` besitzen.

Presence Channels Diese Art von Kanälen baut auf dem Konzept von privaten Kanälen auf und verfügt somit auch über eine Authentifizierungsschnittstelle, der konventionelle Prefix ist allerdings `presence-`. Der wesentliche Unterschied bzw. Mehrwert besteht darin, dass Informationen über Abonnenten des Kanals gespeichert werden können (siehe Prog. 3.1). Diese Art von Kanälen sind somit optimal für z. B. Chatrooms, oder Mehrbenutzer-Spiele. Der Nachteil daran ist allerdings die Limitierung auf maximal 100 verbundene Nutzer pro Kanal.

Es ist jederzeit möglich Kanäle zu abonnieren oder Abonnements zu kündigen und es besteht keine Notwendigkeit zu warten, bis *Pusher* eine Web-Realtime Verbindung hergestellt hat. *Pusher* bietet derzeit für Entwicklungszwecke einen gratis Sandbox-Zugang für 100 simultane Verbindungen und 200.000 Nachrichten pro Tag.

Programm 3.2: *Firebase*: Authentifizierung-Regel für Schreibrechte auf das Benutzer-Objekt des aktuell authentifizierten Benutzers.

```
{
  "rules": {
    "users": {
      "$uid": {
        ".write": "$uid === auth.uid"
      }
    }
  }
}
```

Programm 3.3: *Firebase*: Erstellen eines Event-Listeners für einen spezifischen Pfad in der Datenbank.

```
var stockReference = firebase.database().ref('products/' + id + '/stock');

stockReference.on('value', function(snapshot) {
    updateStock(product, snapshot.val());
});
```

3.2.2 Google Firebase

Firebase ist ein Set von Diensten, zur Verfügung gestellt von Google, welches unter anderem eine *Echtzeit-Datenbank*[24] mit *Authententifizierung* und Cloud-Funktionen[25] beinhaltet. Firebase spezialisiert sich somit nicht nur auf Realtime UX, sondern, wie es ein Marketing-Slogan besagt, auf die Erstellung Server-loser Applikationen [34]. Die Ermöglichung von *Realtime UX* ist hier durch eine *Cloud-basierte*[26] No-SQL Echtzeit-Datenbank gegeben. Das Datenbankschema ist JSON[27] und der Zugriff erfolgt über einen Pfad entlang der Objekt-Hierarchie. Die Authentifizierung erfolgt über eine bereitgestellte Benutzer-Authentifizierung API und Regeln, welche für die verschiedenen Felder innerhalb der Datenbank definiert werden können: `.read, .write, .validate, .indexOn` (siehe Prog. 3.2). Um Client-seitig auf Aktualisierungen der Datenbank reagieren zu können, bietet Firebase eine Ereignis-basierte API (siehe Prog. 3.3). Außerdem stellt Firebase andere serverseitige Dienste zur Verfügung, wie z. B. Datenbank- und Authentifizierungs-Trigger-Funktionen. Der große Unterschied zu einfachen Messaging- oder Pub/Sub-Diensten ist die Persistierung der Daten – *Firebase* ist also als Datensynchronisationsdienst einzuordnen. Wie *Pusher* bietet auch *Firebase* einen kostenlosen Einstieg für bis zu 100 simultanen Verbindungen.

[24] Real-time Database
[25] Cloud Functions
[26] Auf einem über das Web erreichbaren Server befindlich.
[27] Abk.: JavaScript Object Notation

3.3 Web-Realtime Frameworks

In diesem Kapitel werden einige *Frameworks* und *Libraries* besprochen, welche dazu dienen, einen eigenen WebSocket-Server zu verwenden. Der Vorteil liegt hierbei darin, dass es keine vertragsmäßigen Limitierungen für die Nachrichtenanzahl und für die maximalen gleichzeitigen Verbindungen gibt, sowie keine zusätzlichen direkten Kosten entstehen. Welche technischen Grenzen gelten, liegt an der jeweiligen Server-Umgebung. Man ist selbst voll verantwortlich für Skalierung und Wartung der Applikation.

3.3.1 Ratchet

Ratchet ist eine Komponenten-basierte Implementierung eines WebSocket-Servers für PHP, basierend auf der *Event-Loop* von *ReactPHP*[28]. Es bietet sowohl Komponenten zur Erstellung eines einfach WebSocket-Servers als auch komplexere Schnittstellen zur Implementierung von *WAMP*[29](*RPC*[30] und *Publish-Subcribe Prinzip*) und auch eine *Flash-Socket* Rückfalllösung. Ratchet ist nach eigenen Angaben *PSR-0*[31] kompatibel und ist installierbar über *Composer*.[32]

> *Ratchet is a loosely coupled PHP library providing developers with tools to create real time, bi-directional applications between clients and servers over WebSockets. This is not your Grandfather's Internet.* [42]

Auf Client-Seite empfiehlt *Ratchet* bei der Verwendung von *Flash-Socket Fallbacks web-socket-js*[33], obwohl es so scheint, als ob dieses Projekt aktuell nicht mehr aktiv weiterentwickelt wird. Bei der Verwendung von WAMP wird *Autobahn.js*[34] als Client-seitige Lösung empfohlen, allerdings gibt es hier noch keine Unterstützung für Rückfalllösungen [50]. Für die Kommunikation eines *Ratchet Servers* und einer bestehenden Website bietet *ReactPHP* eine *ZeroMQ*[35] Komponente, so kann mit geringem Aufwand *Realtime User Experience* geschaffen werden, ohne eine bestehende Website vollkommen neu implementieren zu müssen [43]. Für die Authentifizierung stellt Ratchet eine `SessionProvider` Komponente bereit, welche auf der *Symfony2 Session* basiert und somit einen geteilten Session-Zugriff ermöglicht.

3.3.2 Socket.io

Wie in Abschnitt 2.2.3 bereits erwähnt, ist die WebSocket-Technologie ein relativ junger Standard, so unterstützen laut [23] ältere User Agents (z. B. jede Version von Internet Explorer vor Version IE10 und Firefox 4.0) oder auch Firewalls und Proxies diesen oft nicht. Auch limitierte Server-Umgebungen können durch limitierten Server-Zugriff

[28] https://reactphp.org/
[29] Abk.: Web Application Messaging Protocol, http://wamp-proto.org/
[30] Abk.: Remote Procedure Call
[31] http://www.php-fig.org/psr/psr-0/
[32] https://getcomposer.org/
[33] https://github.com/gimite/web-socket-js
[34] https://github.com/crossbario/autobahn-js
[35] https://github.com/reactphp/zmq

Programm 3.4: *Ratchet*: Beispiel für eine serverseitige Implementierung des WAMP.

```
1 use Ratchet\ConnectionInterface as Conn;
2
3 class OpenPubSub implements Ratchet\Wamp\WampServerInterface {
4
5     public function onPublish(Conn $c, $topic, $event, array $excl, array $elig) {
6         $topic->broadcast($event);
7     }
8
9     public function onCall(Conn $c, $id, $topic, array $params) {
10        $conn->callError($id, $topic, 'RPC not supported.');
11    }
12
13    // onSubscribe(), onUnSubscribe(), onClose(), onError(), onOpen()
14
15 }
```

und Server-Ressourcen zu Problemen führen. Hier liegt eine Stärke von *Socket.io*, dessen Schnittstelle es ermöglicht, auch bei Umgebungen welche WebSockets nicht unterstützen, dieselbe Implementierung anwenden zu können. Dies wird realisiert mittels diverser hierarchisch aufgebauter Rückfalllösungen. Wird eine der Technologien nicht unterstützt, wird die nächstbeste Lösung angewendet und so weiter. Die standardmäßige Hierarchie der Verbindungsvarianten lautet wie folgt: *WebSocket*, *FlashSocket*, *XHR Long Polling*, *XHR multipart Streaming*, *XHR Polling*, *JSONP Polling* und *iframe* [16, Kap. 5]. Trotz dessen, dass es Kritiker zu *Socket.io* gibt, welche Alternativen, wie zum Beispiel *Sock.js*[36] bevorzugen, macht es dessen Funktionsumfang, Flexibilität und einfache API zu einem der beliebtesten Module von *npm*[37] und wird derzeit laut [46] von über 4000 Modulen als Abhängig angeführt [10, S. 213]. WebSockets werden von den Kernkomponenten von Node nicht unterstützt, allerdings gibt es zahlreiche Module, welche per *npm* verfügbar sind. Socket.io hat das *ws*[38] Modul in Verwendung [12, S. 209]. Es wird sowohl eine Client- als auch eine serverseitige Schnittstelle (*Node.js*) zur Verfügung gestellt. Diese implementieren eine weitere Ebene zur WebSocket API [37]. Dabei können eigene *Events*, *Namespaces* und *Rooms* definiert werden. Namespaces stellen Endpunkte dar, zu welchen sich ein Client mittels `var socket = io('/myNamespace')` verbinden kann. Ein Server kann dann auf diese spezielle Verbindung hören (siehe Prog. 3.5). Wird kein Namespace angegeben, verwendet *Socket.io* standardmäßig `'/'` als Namespace. Rooms sind Kanäle innerhalb eines Namespaces, zu denen sich ein Socket via `socket.join('room')` verbinden oder via `socket.leave('room')` trennen kann. Um Events an einen speziellen Room zu senden wird `.to()`, oder `.in()` verwendet (z. B. `io.to('room').emit('event')`). Gibt man keinen expliziten Room an, wird standardmäßig die Socket-Id (eine zufällige, einzigartige Zeichenkette) verwendet.

[36] https://www.npmjs.com/package/sockjs
[37] https://www.npmjs.com/
[38] https://www.npmjs.com/package/ws

Programm 3.5: *Socket.io*: Definition eines Namespaces und Senden eines Events an diesen.

```
1 var namespace = io.of('/my_namespace');
2
3 namespace.on('connection', function(socket){
4     // connection established
5 });
6
7 namespace.emit(event, data);
```

Programm 3.6: *Sails.js*: Abonnieren einer Produkt-Instanz.

```
1 subscribeAction(req, res) {
2     Product.subscribe(req, ids);
3 },
4
5 updateAction(req, res) {
6     Product.publishUpdate(id, changes);
7 }
```

Programm 3.7: *Sails.js*: Beobachten des gesamten Produkt-Modells.

```
1 watchAction(req, res) {
2     Product.watch(req);
3 },
4
5 createAction(req, res) {
6     Product.publishCreate(data);
7 }
```

3.3.3 Sails.js

Sails.js[39] ist nach eigenen Angaben das am weitesten verbreitete *MVC Framework* für *Node.js*. Aufbauend auf *Express.js*[40] und *Socket.io*, wirbt es speziell, für die Umsetzung von Echtzeitanwendungen, wie Chats oder Multiplayer-Spielen, geeignet zu sein. Die Web Realtime API *Sails.io.js* bietet wie *Socket.io* eine Client- und eine serverseitige API und stellt Fallbacks für Probleme bei WebSocket-Verbindungen zur Verfügung. *Sails.js* bietet für Modells ein Publish-Subscribe System, welches es ermöglicht, auf Veränderungen einzelner Modell-Instanzen zu hören (siehe Prog. 3.6), bzw. auch auf Erstellung neuer und Löschung bestehender Instanzen (siehe Prog. 3.7). Die möglichen Ereignisse lauten wie folgt: `create`, `destroy`, `update`, `add`, `remove` und `message`, wobei sich `add` und `remove` auf definierte Fremdschlüssel-Beziehungen beziehen. Das *Sails.js* Framework bietet einen virtuellen Request-Interpreter für WebSocket-Nachrichten und

[39] https://sailsjs.com/
[40] http://expressjs.com/de/

übersetzt diese automatisch, sodass sie mit den HTTP-Routes der Applikation kompatibel sind. Es kann somit auch über die WebSocket-Verbindung eine virtuelle `GET`-, `POST`-, `PUT`- oder `DELETE`-Anfrage gestellt werden. Dabei sendet *Sails* immer einen Request- und Response-Frame als HTTP-Pendant. Somit wird der serverseitige Code leichter wieder zu verwenden und es macht den Einstieg für Entwickler, die noch nicht mit WebSockets vertraut sind, unkomplizierter. Außerdem hat es den Vorteil, dass auch für *WebSocket*-Nachrichten die *Policy*-basierte Authentifizierung angewendet werden kann. Eine *Policy* ist dabei als *Middleware* im Sinne des *Connect*[41] Frameworks zu verstehen, welche vor der eigentlichen Controller-Action aufgerufen wird und den Request weiter- bzw. umleitet oder auch abbricht. Für den Verbindungsvorgang selbst wird *sails.io.js*[42] verwendet, welches im *Sails* Framework enthalten ist. Dabei handelt es sich um eine leichtgewichtige JavaScript Komponente, die auf *Socket.io* aufbaut und darüber hinaus noch andere nützliche Funktionalitäten bietet, wie z. B. die Auslagerung der Socket-Ids in einen externen Speicher, wie einer Redis Datenbank, für bessere horizontale Skalierbarkeit der Anwendung oder auch Methoden zur Verwendung in Kombination mit dem WebSocket Request-Interpreter, wie z. B. `io.socket.get(url, data, callback)`.

[41] https://github.com/senchalabs/connect
[42] https://github.com/balderdashy/sails.io.js

Kapitel 4

Konzept und Technisches Design

4.1 Anforderungen

Bei der Integration von *Realtime User Experience* in CM-Systeme ergeben sich verschiedene Anforderungen, welche beachtet werden sollen.

4.1.1 CMS-Autonomie

Eine essentielle Anforderung bei der Integration der Funktionalitäten ist die Autonomie der Web-Realtime Erweiterung zum CMS. Dieses muss somit auch mit der Erweiterung unabhängig davon funktionieren, es darf das System selbst nicht beeinflussen. Die Erweiterung darf die Stabilität und Robustheit des Systems nicht beeinträchtigen, z. B. dürfen Speichervorgänge, welche ein Web-Realtime Event auslösen, nicht scheitern, falls ein Problem mit der Web-Realtime spezifischen Verbindung vorliegt. Des Weiteren sollte das System, unter Berücksichtigung einer angepassten Serverleistung, keine bzw. kaum längere Verarbeitungszeiten aufweisen. Zusätzliche Anfragen, die durch die Erweiterung entstehen, wie z. B. Datenbank-Abfragen oder Anfragen zu anderen Servern, sollten wenn möglich asynchron erfolgen.

4.1.2 Verbindungsautonomie

Abhängig von den jeweiligen Server-Umgebungen besteht die Möglichkeit, dass manche Web-Realtime Dienste (siehe Abschnitte 3.2 und 3.3) nicht unterstützt werden. Um dennoch eine universell anwendbare Lösung bereitzustellen, ist es notwendig, die Erweiterung nicht nur autonom gegenüber dem CMS selbst zu gestalten, sondern auch gegenüber der verwendeten Verbindungsvariante zum Web-Realtime Dienst bzw. zum verwendeten Framework oder Dienst selbst, wie z. B. WebSocket, Polling, Pusher, Firebase usw.

4.1.3 Anwendbarkeit

Der Fokus dieser Arbeit liegt auf der Verwendung von CM-Systemen, welche auf der LAMP- Technologieplattform basieren. Somit ist PHP die einzige Programmiersprache, für die tiefgründige Kenntnisse des CMS-Entwicklers bei der Anwendung der Web-Realtime Erweiterung vorausgesetzt werden können. Es soll nach der Installation der

Erweiterung also möglich sein, einfache Pub/Sub-basierte Applikationen umzusetzen, ohne am Verbindungsdienst (z. B. Node.js Server) selbst Implementierungen vornehmen zu müssen.

Auch im Zusammenhang mit der Installation der Erweiterung ist die einfache Anwendbarkeit von Relevanz. Dies bezieht sich vor allem darauf, diese nicht manuell integrieren zu müssen, sondern dass sie unter Zuhilfenahme einer Anwendung zur Verwaltung von Abhängigkeiten[1], wie z. B. *Composer*[2] für PHP oder *NPM*[3] für Node.js, installierbar ist. Dieser Installationvorgang sollte wiederum alle benötigten Abhängigkeiten, wie z. B. externe Programmbibliotheken oder auch Datenbanktabellen, automatisch erstellen und über eine zentrale Konfigurationsmöglichkeit (z. B. Konfiguration im CMS) zugänglich machen.

4.1.4 Verlässlichkeit der Verbindung

Die Umsetzung des Aspekts der Verlässlichkeit der Verbindung betrifft vor allem die Wahl des verwendeten Web-Realtime Dienstes. Viele Technologien im Bereich des Realtime Webs sind noch sehr jung und finden somit bei veralteten *User Agents* teilweise noch keine Unterstützung. Die Implementierung der Erweiterung muss es somit erlauben, flexibel gegenüber der konkreten Verbindungsvariante zu sein, sodass es möglich ist, dynamisch Rückfalllösungen zu verwenden (z. B. XHR Polling).

Im Falle eines Ausfalls der Verbindung dürfen keine Informationen über die aktuelle Sitzung verloren gehen. Dies betrifft vor allem Kanal Abonnements (siehe Abschnitt 4.3.1), welche für den Wiederverbindungsvorgang (*Reconnect-Callback*) immer konsistent bleiben müssen. Wird z. B. ein Chat geöffnet, welcher bei der ursprünglichen Seitenanfrage noch nicht benötigt wurde, muss dies im *Reconnect-Callback* registriert werden, um bei einem Verbindungsproblem die Möglichkeit zu haben, automatischen wieder zu diesem zu verbinden.

4.1.5 Flexibilität der Implementierung

Da die Erweiterung viele dynamische und somit austauschbare Komponenten beinhaltet, ist diese Thematik umso wichtiger. Die wichtigsten dieser Komponenten sind:

- die Schnittstelle zwischen Apache Server und dem Web-Realtime Dienst,
- der Schnittstellen-spezifische generierte Programmcode für Interaktionen mit dem Web-Realtime Dienst (z. B. Kanal-Abonnements),
- die Ereignis-Daten (*Events*), welche über den Web-Realtime Dienst zum Client übertragen werden sollen,
- der Web-Realtime Dienst selbst.

Darüber hinaus ist es wichtig, dennoch einen einheitlichen Zugriff auf diese Bestandteile zu erhalten, ohne die Möglichkeit zu verlieren, die Funktionalitäten flexibel erweitern zu können.

[1] Dependency-Management-Tool
[2] https://getcomposer.org/
[3] https://www.npmjs.com/

4.1.6 Erweiterbarkeit

Da die Integration einer Web-Realtime Verbindung eine ganze Palette an neuen Möglichkeiten zur Umsetzung von Applikationen bietet, die ohne dieser nicht möglich wären, ist es umso wichtiger, die CMS-Erweiterung selbst auch erweiterbar zu gestalten. Jeder dieser Anwendungsfälle mag andere Anforderung an die Erweiterung stellen, wie z. B. eine Authentifizierung der Kanäle oder Sub-Kanäle, welche alle Ereignisse des übergeordneten Kanals erben.

Die Implementierung soll somit nicht auf einen spezifischen Anwendungsfall eingeschränkt sein, sondern eine Schnittstelle für verschiedensten Fälle zur Verfügung stellen.

4.2 Systemeigenschaften

Innerhalb einer CMS Applikation können viele verschiedene Ereignisse auftreten, die einen potentiellen Datenbestand zur Weiterleitung an einen verbundenen Benutzer aufweisen. Um ein Konzept zu erstellen, welches alle diese Fälle optimal abdeckt, ist es notwendig, möglichst alle diese Ereignisse zu identifizieren. Des Weiteren muss eine Kommunikationsmethodik zur Strukturierung der Web-Realtime Kommunikation zwischen Client und Server definiert werden.

4.2.1 Ereignisidentifikation

Für die Integration von *Realtime User Experience*, ist es vorerst notwendig, potentielle Stellen innerhalb von CM-Systemen zu identifizieren, welche Web-Realtime Ereignisse auslösen können und deren Kommunikationabläufe zu analysieren. Nach [52] können diese Interaktionen in drei Kategorien eingeteilt werden, die Grenzen verlaufen hierbei allerdings fließend:

System Interaktionen

System Interaktionen sind Ereignisse, welche mit einem assoziierbaren Datenbestand auftreten, dieser sollte teilweise sofort weitergeleitet bzw. für andere Systeme zur Verfügung gestellt werden. Derartige Interaktionen können viele Formen annehmen, prädestinierte Beispiele wären Web-Endpunkte, Logging, CRON Jobs oder E-Mails usw. Im Allgemeinen stellt eine jede Web Applikation mindestens einen Web Endpunkt zur Verfügung, sei es um HTML bereitzustellen oder als Web-API. Jeder Aufruf kann als ein Web-Realtime Ereignis mit verknüpften Datenbestand interpretiert werden, wobei jede Anfrage bzw. Antwort als eigenständiges Ereignis gesehen werden kann. *Logging* wird verwendet um Informationen persistent zu protokollieren, welche von weiterer Relevanz sein könnten, aber teilweise nicht direkt weiterverarbeitet werden können. Dies trifft nicht zu unter Verwendung von Web-Realtime Technologien, hier könnten diese Informationen direkt an die daran interessierten Nutzer weitergeleitet werden.

Daten Interaktionen

CMS-basierte Applikationen sind in der Regel Inhalts-zentriert, somit liegt der Fokus am Manipulieren der Daten. Inhalte werden erzeugt, dargestellt, aktualisiert oder gelöscht

– diese Aktionen werden auch als *CRUD*[4] bezeichnet. Diese Art von Web-Realtime Ereignis ist meist einfach zu identifizieren. Jede Art von Datenbank-Interaktion fällt in diese Kategorie und könnte zum Beispiel ein MySQL `INSERT`- oder `UPDATE`-Statement sein, oder auch, wie in den meisten Fällen innerhalb von CM-Systemen, eine ORM[5] Interaktion, wie z. B. `$object->save()`. Diese Kategorie kann als Abstraktion der System-Interaktionen gesehen werden, da eine Datenbank-Interaktion auch eine System-Interaktion ist.

Benutzer Interaktionen

Jede Form von Interaktion eines Benutzer mit der Web Applikation kann als Web-Realtime Ereignis eingestuft werden. Ein Benutzer, welcher z. B. die Navigation einer Website bedient, interagiert wiederum mit einem Web Endpunkt der Applikation. Allerdings mit der Erweiterung, dass hierbei zu den Request-Daten auch Daten über den Benutzer selbst enthalten sein können. Dies trifft vor allem für *Single-Page* Applikationen zu, zu welchen auch viele CMS Administrationsoberflächen gezählt werden können. Aber auch andere Benutzerinteraktionen, welche augenscheinlich keine Anfragen an das System stellen würden, wie das Bewegen der Maus (z. B. für *Usage Heatmaps*[6]) oder eine Eingabe in einem HTML Inputfeld, kann als Web-Realtime Ereignis gehandhabt und verarbeitet werden. Auch kollaborative Anwendungsfälle gehören dieser Kategorie an, wie z. B. Chat-Applikationen. Hierbei kann der Zusammenhang der Kategorien untereinander gut dargestellt werden: Wird ein Datensatz aktualisiert oder erstellt (*Daten Interaktion*), an welchem ein Benutzer interessiert ist, hinterlässt dieser womöglich einen Kommentar (*System Interaktion*), auf welchen hin andere Benutzer aktiv werden und sich somit eine Konversation entwickelt (*Benutzer Interaktion*). Wobei hierbei auch erkenntlich wird, dass die Grenzen dieser Kategorien fließend verlaufen.

4.2.2 Methoden der Kommunikation

Wie in [24] beschrieben, können Realtime Web Applikationen je nach ihrer Applikations- und Datenkomplexität zu vier verschiedenen Methodiken für den Nachrichtenaustausch zugeordnet werden: *onMessage*, *Publish-Subscribe*, *Data Synchronization* und *Remote Methode Invokation* (siehe Abb. 4.1).

OnMessage

OnMesage steht für ein simples Aktion-Reaktion Schema, welches sich optimal für einmalige Nachrichten ohne spezifischen Kontext eignet. Wenn eine Aktion auftritt (z. B. eine Nachricht vom Server wird am Client empfangen), wird darauf eine Reaktion ausgeführt, wie z. B. eine Benutzerbenachrichtigung. Anwendungsfälle für diese Art der Kommunikation bieten sich in der Regel bei Applikationen mit geringer Daten- und Anwendungskompläxität.

[4] Create, Read, Update, Delete
[5] Abk.: Object-Relational Mapping
[6] Visualisierzungen der Benutzungstatistik

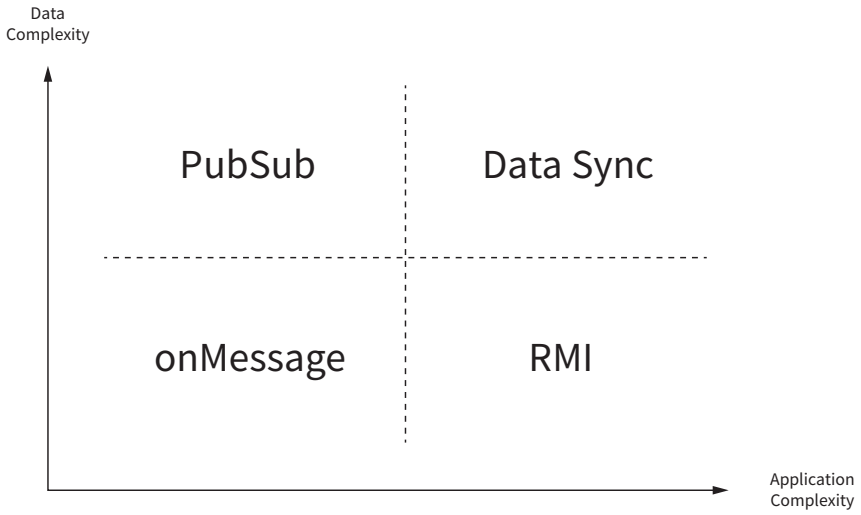


Abbildung 4.1: Kategorisierung der Kommunikationsmethodik einer Applikation nach Daten- und Applikationskomplexität. Bildquelle [24].

Publish-Subscribe

Pub/Sub ist ein Daten-zentrierter Ansatz, welcher daraus resultiert, dass mit höherer Datenkomplexität auch die Notwendigkeit steigt die empfangenen Daten zu kategorisieren. Somit besteht die Möglichkeit, nur Nachrichten zu empfangen, welche für den verbundenen Empfänger relevant sind. Bei Pub/Sub abonniert man diese spezifischen Daten dann in der Regel über eine eindeutige Kennung[7], welche z. B. als *Channel*, *Topic* oder *Subject* bezeichnet werden kann. Sobald es innerhalb der Applikation dazu kommt, gruppierte Teilmengen an Daten empfangen zu müssen, wie z. B. bei authentifizierungspflichtigen Daten, ist diese Variante empfehlenswert.

Data Synchronization

Diese Methode kann als eine Abstraktionsvariante aufbauend auf dem Pub/Sub-Prinzip gesehen werden. Der Fokus dieses Ansatzes liegt dabei darauf, Datenstrukturen zu manipulieren und daraufhin alle verbundenen Parteien stets mit dem aktuellen Datenbestand zu synchronisieren. In diesem Kontext ist es wichtig, auf sog. *Operational Transformations* (OT) zu achten, eine Technologie, welche sich mit dem Wartungsvorgang der Datenkonzistenz[8] beschäftigt, wie z. B. Kollisionen bei gleichzeitigen Änderungen in kollaborativen Applikationen. Datensynchronisation ist die empfohlene Lösung für Anwendungsfälle mit hoher Daten- und Anwendungskomplexität.

Remote Methode Invocation

Bei der *Remote Method Invocation* (RMI) ist ein Server-Objekt einem Client-seitigen Proxy-Objekt zugeordnet, ein Client-seitiger Aufruf einer Methode kann somit zu einer

[7] Unique string identifier
[8] Consistency Maintenance

Netzwerkwerkanfrage zu der zugeordneten Server-Objektmethode führen. Im Gegensatz zu Pub/Sub-Lösungen, bei welchen Netzwerk-Anfragen klarer ersichtlich sind, abstrahiert RMI diese bis zu einem Maß, in dem der Anwender sich dieser Anfragen gar nicht mehr bewusst ist. Diese Methodik ist vor allem für komplexe Applikationenstrukturen mit einfachem Datenaustausch optimal.

4.3 Architektur

In diesem Abschnitt werden das Konzept und die abstrakten Kommunikationsabläufe des eigenen Ansatzes vorgestellt.

4.3.1 Kanal-Konzept

Da die Verwendung eines CMS für eine Web-Applikation offenkundig in der Regel dadurch begründet wird, ihre Inhalte an einer zentralen Stelle verwalten zu können, liegt der Fokus solcher Applikationen meist auf Seiten des Datenbestandes. Somit bietet sich nach Abb. 4.1 bei der Wahl unter den in Abschnitt 4.2.2 erwähnten Methodiken, das *Publish-Subscribe Prinzip* als Lösung an. Dieses beinhaltet immer ein *PUB*-Element, auf welches Nachrichten publiziert werden können und zu welchem sich beliebig viele *SUB*-Elemente verbinden können. Wenn eine neue Nachricht an das *PUB*-Element gesendet wird, wird diese an alle verbundenen *SUB*-Elemente weitergeleitet [15, S. 196]. Ein solches *PUB*-Element wird innerhalb des verwendeten Ansatzes als Kanal (*Channel*) bezeichnet und ist dabei seitens PHP nur ein abstraktes Konzept. Die Verbindungskomponente selbst ist ein adäquates Element auf Seiten des Web-Realtime Dienstes, welches zum Beispiel als *Sails.io* WebSocket Room des *Node.js* Servers abgebildet wird. Dieser kann über eine eindeutige Zeichenkette adressiert werden. Ein Kanal bietet dem Anwender die Möglichkeit, diesen zu abonnieren (*subscribe*), ein solches Abonnement zu kündigen (*unsubscribe*) und neue Nachrichten zu veröffentlichen (*publish*). Der Zugriff auf alle Kanäle der Applikation soll einheitlich über eine einheitliche bereitgestellte Schnittstelle erfolgen können.

Abonnements

In Abb. 4.2 wird ein schematischer Ablauf einer *Channel Subscription* illustriert: Der erste Schritt (1) ist immer die initiale Anfrage (*Page Request*), im Beispiel dargestellt als Objekt mit der ID 5 der CMS Objektstruktur. Unter der Annahme, die Web-Realtime Erweiterung ist im CMS installiert und aktiviert, wird durch den Aufruf automatisch ein zugehöriger Kanal erstellt und dieser abonniert. Das Abonnement ist allerdings bisweilen nur eine Aufforderung für den Client, welches ausgeführt werden soll, sobald die Verbindung zum jeweiligen Web-Realtime Dienst besteht (2) und wird mit dem *HTTP Response* mit zurückgeschickt. Die Erstellung der Aufforderung an sich passiert über einen *Code-Generator*, welcher es ermöglicht, für verschiedene Aktionen Skripte an definierte Positionen im HTML Dokument anzufügen. Die in der Regel beste Position für Abonnement-Aufforderungen ist der *Reconnect-Callback*, eine Methode, welche aufgerufen wird, falls die Verbindung kurzzeitig getrennt worden war und dann wieder eingerichtet wurde. Die Abonnement-Aufforderung (3) kann somit bereits über die

Web-Realtime Verbindung ausgeführt werden. Im Web-Realtime Dienst selbst entscheidet nun eine sogenannte *Policy*, ob für die jeweilige *Route* (z. B. `/app/sub/my_channel`) eine Authentifizierung erforderlich ist. Falls ja, wird die Anfrage über eine Authentifizierungsschnittstelle des CMS legitimiert (4). Bei erfolgreicher Authentifizierung oder falls der Kanal nicht authentifizierungspflichtig ist, wird ein Kanal auf Seiten des Web-Realtime Services erstellt, die *Subscription* ausgeführt und der Erfolg bzw. Misserfolg der Aktion als Antwort zurückgeschickt (6).

Das Konzept der *Policy* ist ein Ansatz, welcher vom *Sails.js* Framework implementiert wird. Wie dieser Routing-Vorgang im Detail gehandhabt wird ist somit vom Schema der Programmschnittstelle des verwendeten Web-Realtime Dienstes selbst abhängig. Bei der Verwendung eines extern betriebenen Web-Realtime Dienstes (siehe Abschnitt 3.2) gibt es dieses Policy-Schema klarerweise nicht, der Vorgang der Legitimation der Anfragen über eine Authentifizierungsschnittstelle ist allerdings auch hier üblich (z. B. bei den privaten Kanälen des Anbieters *Pusher*).

Kündigung eines Abonnements

Das Kündigen eines Abonnements (*unsubscribe*) kann nach dem gleichen Schema wie das Senden von Ereignissen (*publish*) ablaufen. Da dies eine Aktion ist, bei welchem die Web-Realtime Verbindung bereits besteht, kann der verbundene User Agent direkt über ein gesendetes Ereignis getrennt werden. Unabhängig davon, welche Methode für die Trennung verwendet wird, ist es allerdings notwendig, die Signatur des nicht mehr abonnierten Kanals auch im *Reconnect-Callback* zu entfernen. Andernfalls konnte es im Fall eines Verbindungsfehlers zu Inkonsistenzen in den Kanal-Abonnements kommen.

Senden von Events

In Abb. 4.3 wird der abstrakte Ablauf des Veröffentlichen eines Ereignisses (*Publish*) beschrieben. Zuerst benötigt es immer einen Auslöser (siehe Abschnitt 4.2.1), welcher ein Publish-Ereignis impliziert (1). Dies könnte, wie in der genannten Abbildung dargestellt, zum Beispiel eine HTTP-Anfrage sein, welche eine *Update*-Operation eines Objektes ausführt. Die dazugehörenden Ereignisdaten werden dann an den jeweiligen Kanal weitergereicht und zur Übertragung an den Web-Realtime Dienst aufbereitet. Im erwähnten Beispiel würde dies bedeuten, nur die benötigten Felder aus dem Objekt auszulesen und eventuelle Transformationen darauf durchzuführen (z. B. die Generierung eines Thumbnails). Die finalen Daten werden dann im JSON-Format zum Web-Reatlime Dienst übertragen (2). Auf Seiten des Web-Realtime Dienstes wird zuerst über mögliche Policies überprüft ob der Request vom jeweiligen Nutzer ausgeführt werden darf (3, 4). Außerdem werden die übertragenen Parameter anhand eines gegeben Schemas validiert. Sind alle notwendigen Daten vorhanden, wird sofort ein Response an das CMS geschickt, dass der Request verarbeitet werden kann (5a). Somit bleibt die zusätzliche Verarbeitungszeit der ursprünglichen Anfrage an das CMS möglichst gering. Nach diesem Vorgang werden die Daten an alle zum jeweiligen Kanal verbundenen Benutzer gesendet (5b).

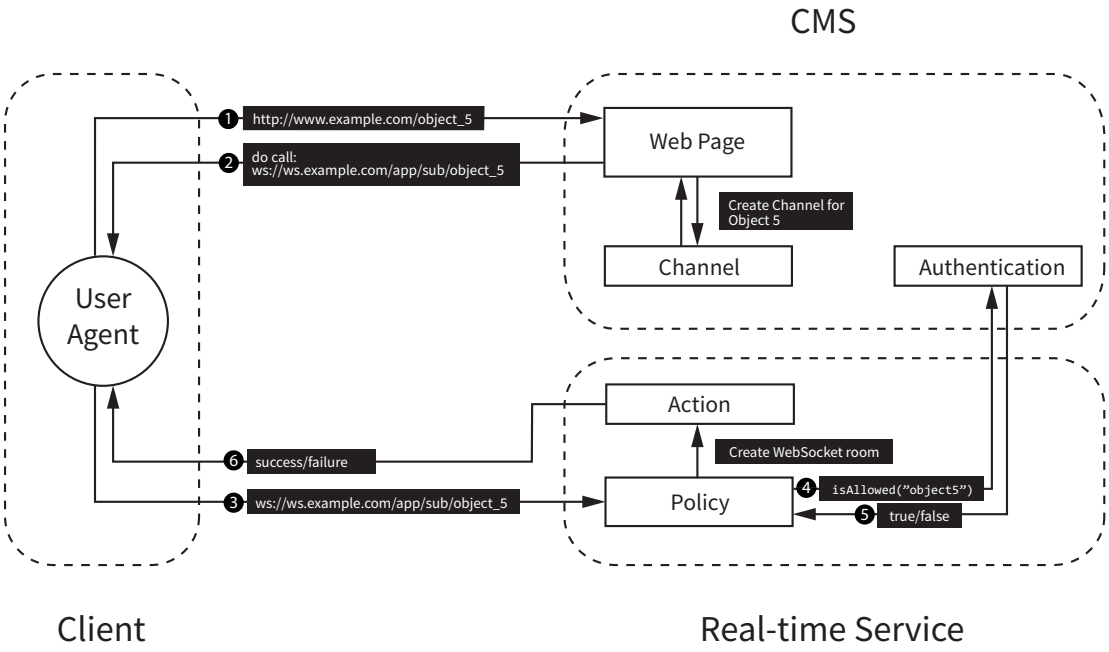


Abbildung 4.2: Schematischer Ablauf eines Abonnements zu einem Objekt-Kanal innerhalb des CMS, im Beispiel einer Objektinstanz.

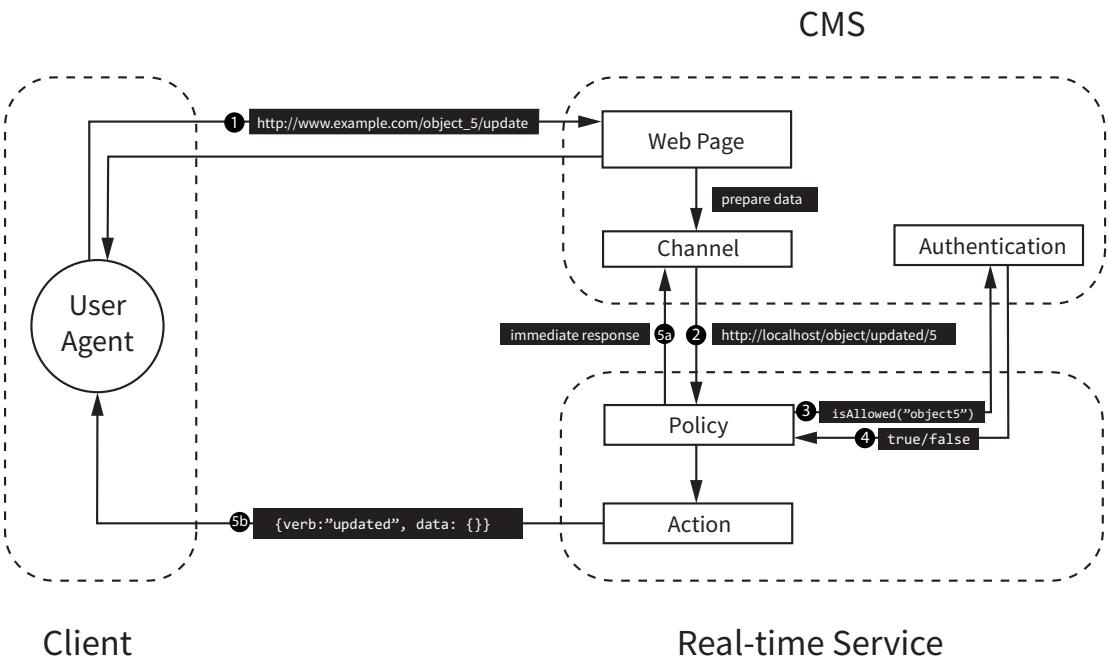


Abbildung 4.3: Schematischer Ablauf des Veröffentlichen eines Ereignisses zu einem Web-Realtime Kanal, im Beispiel eines Update-Ereignisses zu einem Objektinstanz-Kanal innerhalb des CMS.

4.3.2 Arten von Kanälen

Ein Kanal ist wie in Abschnitt 4.3.1 beschrieben immer ein PUB Element des *Publish-Subscribe Prinzips*, welches über eine eindeutige Zeichenkette adressiert werden kann. Bei der Anwendung dieses Konzeptes innerhalb des CMS wurden diverse Kanäle vorweg definiert, wie z. B. ein globaler Website-Kanal, Dokument-Kanäle, Objektinstanz-Kanäle oder Objektlisten-Kanäle usw. Diese Kanäle können wiederum in folgende zwei verschiedene Kategorien unterteilt werden:

Automatisierte Kanäle

Diese Art von Kanal charakterisiert sich dadurch, dass sie automatisiert Ereignisse von Seiten der CMS-internen Operationen empfängt. Der Anwender muss lediglich den gewünschten Kanal abonnieren und erhält dessen vordefinierte Ereignisse automatisch. Beispiele hierfür wären *C(R)UD*[9] Ereignisse für Objektlisten- bzw. Objektinstanz-Kanäle, bei welchen alle Benutzer, die ein Abonnement dieser Objektinstanz haben, automatisch Benachrichtigungen zu ausgeführten *Update*- und *Destroy*-Operationen empfangen. Zusätzlich zu den automatisch gesendeten Ereignissen können auch manuell beliebige Ereignisse an die jeweiligen Kanäle gesendet werden, sowohl programmatikalisch als auch über die Administrationsoberfläche des CM-Systems.

Benutzerspezifische Kanäle

Es können beliebige Kanäle für benutzer- und applikationsspezifische Logiken erstellt werden, um allen Anwendungsfällen, sei es innerhalb des CMS selbst oder in der Endbenutzer Anwendung, gerecht zu werden. Diese Art von Kanäle empfängt keine Ereignisse, welche nicht manuell gesendet wurden.

4.4 CMS Integration

Ein Enterprise-Level CMS bietet sowohl auf Seiten der CMS-Administration als auf Seiten der möglichen Endanwendungen diverse Einsatzzwecke für Web-Realtime, wie in Abschnitt 3.1 bereits beschrieben. Hiervon wurden in der Implementierung des Projekts nur ein Bruchteil umgesetzt.

Kollaborative Aspekte: Eine gravierende Verbesserung, welche durch die Integration von Web-Realtime möglich wird, ist für CM-Systeme neben der Fokussierung auf die Verwaltung von Inhalten, Aspekte des *CSCW*[10] intensiver und besser berücksichtigen zu können. Der Begriff *CSCW* – auch genannt *Rechnergestützte Gruppenarbeit* – beschreibt ein Forschungsgebiet der Informatik, Psychologie, Soziologie und weiteren Fachgebieten. Er beschäftigt sich damit, grundlegende Methoden zur Verbesserung der Zusammenarbeit zu finden und dadurch technische Systeme zu entwerfen, welche Gruppenarbeit unterstützen [26]. Das Ziel wäre somit, das CMS in eine kollaborative *Groupware* Applikation zu verwandeln. Anwendungsfälle würden sich hier in vielen Bereichen eines

[9] Create (Erstellung), Update (Aktualisierung), Delete (Löschung)
[10] Abk.: Computer Supported Cooperative Work

jeden CMS-basierten Projekts bieten, welches durch mehr als einen Benutzer verwaltet wird: von allgemeinen Komponenten, wie Online-Status, Verfügbarkeit und Tätigkeit anderer Benutzer oder Kommunikationsmöglichkeiten innerhalb des CMS z. B. via eines Chat-Systems, bis zu konkreten CMS-internen Anwendungen, wie der Integration von Arbeitsprozessmanagement-Funktionen, Web-Realtime Templating (siehe Abschnitt 5.4.2) oder auch Echtzeit-Datenmodellierung usw.

Automatisierung von CMS-internen Ereignissen Die meisten CM-Systeme bieten bereits ein System, via Einschubmethoden[11] oder anderen Möglichkeiten der Ereignisverwaltung[12], die in Abschnitt 4.2.1 beschriebenen Ereignisse zu behandeln und zusätzlichen Programmcodes an diesen Stellen auszuführen. Es würde sich somit anbietens diese Ereignisse, vorausgesetzt einer vorherigen Konfiguration, direkt in Web-Realtime Ereignisse zu verwandeln und somit diverse automatisierte Kanäle von Seiten des CMS zur Verfügung zu stellen, welche Client-seitig gehandhabt werden könnten.

[11] Hook
[12] Event-Management

Kapitel 5

Implementierung

Das implementierte System dient als Fallstudie der in Kapitel 4 beschriebenen Konzepte. Somit handelt es sich bei der Applikation um einen konzeptionellen Beweis und nicht um ein vollständig ausgereiftes System.

5.1 Systemumgebung

Wie in Abb. 5.1 schematisch dargestellt, besteht die Systemumgebung aus zwei Web-Servern mit gemeinsamer Datenbank, welche über verschiedene Verbindungsvarianten vom jeweiligen User Agent angesprochen werden können. In dieser Arbeit stehen auf der *LAMP* Technologieplattform basierende CM-Systeme im Fokus, demnach ist ein *Apache*[1] Server in Kombination mit *Pimcore*[2] als CMS in Verwendung. Als Web-Realtime Dienst dient ein *Node.js*[3] Server mit *Sails.js*[4] als verwendetes Framework (siehe Abschnitt 3.3.3). Der Client kann das CMS wie gehabt via HTTP ansprechen, die Web-Socket Verbindung besteht unabhängig davon nur zum *Node.js* Server. Beide Systeme teilen sich die selbe Datenbank, welche Pimcore-konform eine MySQL Datenbank ist. *Sails.js* ist hier durch *Waterline*[5] als Datenbankschnittstelle flexibel. Für die Kommunikation zwischen den Web-Servern stehen zwei Transportdienste zur Verfügung: eine TCP Implementierung von *ZeroMQ*[6] und HTTP Requests.

5.1.1 Pimcore

Pimcore ist eine Open-Source Software für PIM[7], CMS, DAM[8] und E-Commerce. Es ist eine Web-basierte Anwendung und baut auf PHP und MySQL/MariaDB als Persistierungsschicht. Die grundlegenden Komponenten der derzeit aktuellsten stabilen Ver-

[1] https://httpd.apache.org/
[2] https://www.pimcore.org/
[3] https://nodejs.org/
[4] https://sailsjs.com/
[5] https://github.com/balderdashy/waterline
[6] http://zeromq.org/bindings:php
[7] Abk.: Produktinformationsmanagment
[8] Abk.: Digital Asset Management

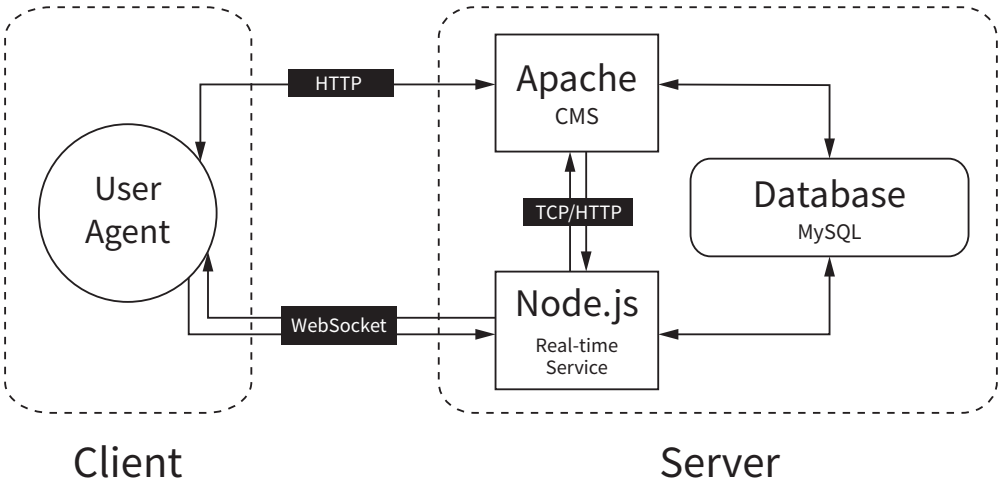


Abbildung 5.1: Schematische Darstellung der Architektur des Systems.

sion 4.6 kommen vor allem aus dem *Zend Framework 1*[9] und dem *Symfony*[10] Projekt. Für die Verwaltung der einzelnen internen und externen Abhängigkeiten wird *Composer*[11] verwendet. Die Architektur der Pimcore Anwendung ist strikt nach den Vorgaben der PSR-Spezifikationen[12] implementiert und folgt Konventionen der objektorientierten Programmierung mit Beachtung des MVC-Patterns.[13]

Optimale Erweiterbarkeit

Pimcore ist besonders geeignet für die Integration der Realtime UX, da es, wie in Dokumentation [28] genauer ausgeführt, diverse Möglichkeiten bietet, es beliebig zu erweitern. Einige dieser Möglichkeiten sind:

- Hinzufügen eigener Abhängigkeiten über *Composer*,
- *Dependency Injection* via *PHP DI*[14],
- *Event API*, um sich an essentielle Stellen des Applikationsablauf einhängen zu können und um eigene Logiken zu erweitern,
- Definition von *Parent Classes* für Pimcore Objekte im Admistrationsbereich,
- Anwenderspezifische *Persistence-Models*, um ORM-Objekte außerhalb der Pimcore Objekt-Struktur zu definieren,
- Implementierung komplexe installierbare Plugins, welche es z. B. ermöglichen die Administrationsoberfläche von Pimcore zu erweitern bzw. zu verändern oder neue editierbare Bereiche für Dokumente (*Document Editables*) mit benutzerspezifischen Logiken zu erstellen.

[9] https://framework.zend.com/
[10] https://symfony.com/
[11] https://getcomposer.org/
[12] Abk.: PHP Standard Recommendation
[13] Abk.: Model View Controller
[14] http://php-di.org/

5.1.2 Client-Server Verbindung

Die in Abschnitt 2.2.3 vorgestellten HTML 5 Verbindungsvarianten haben alle ihren spezifischen Einsatzbereich. Für die angestrebte *Realtime User Experience* in CM-Systemen wird eine Verbindung benötigt, welche es ermöglicht, dass sowohl Client als auch Server zu jeder Zeit Daten senden und empfangen können. Dieser Anforderung würden sowohl *Server-Sent-Events* als auch *WebSocket* Verbindungen gerecht werden, wobei SSE, wie der Name vermuten lässt, dafür entworfen wurden einzelne Events zu senden. Eine WebSocket Verbindung ist hingegen dafür prädestiniert, eine persistente Verbindung herzustellen, die es jeder Partei ermöglicht zu jeder Zeit Daten zu senden, sowohl in binärer als auch in textueller Form. Grundsätzlich kann jede Applikation, welche mit SSE implementiert werden kann, auch mittels einer WebSocket-Verbindung realisiert werden, unter anderem auch deshalb bekamen WebSockets in den letzten Jahren mehr Aufmerksamkeit von der Web-Gemeinschaft. Daraus folgend ist die Browser Unterstützung für WebSockets wesentlich besser als für SSE. SSE bieten allerdings den Vorteil, dass sie serverseitig weite Unterstützung finden und die Implementierung unkomplizierter sein mag. Allerdings kann das Offenhalten einer persistenten HTTP Verbindung auch bei SSE in manchen Server Umgebungen zu Problemen mit den Server-Ressourcen führen. Ein weiterer Grund dafür, dass *SSE* derzeit keine sehr weite Verbreitung finden ist, neben der nicht bidirektionalen Verbindung, die mangelnde Client-seitige Unterstützung [22] vor allem seitens der Browser *Microsoft Edge* und *Microsoft Internet Explorer*. Ein essentieller Aspekt, weshalb SSE für die Anwendung in einer CMS-basierten Applikation nicht geeignet sind, ist, dass derzeit die Anzahl der gleichzeitig geöffneten HTTP-Verbindungen stark limitiert ist – *Google Chrome* ermöglicht z. B. maximal sechs Verbindungen. Hier bestehen in den Browsern *Google Chrome* [40] und *Mozilla Firefox* [38] bereits offene Fehlerberichte, welche allerdings als *WONTFIX* deklariert wurden und somit für HTTP/1.1 nicht in absehbarer Zeit behoben werden werden. Dies mag auch aufgrund der neuen Konzepte, welche mit HTTP/2 eingeführt werden, so beschlossen worden sein (siehe Abschnitt 6.2).

Nach einer Studie der Queen's University Kingston [8] werden WebSockets als leistungsfähigste Standard-basierte Realtime Web Technologie bestätigt. Außerdem werden sie aufgrund ihrer einfachen Schnittstelle und dessen implementierte Konzepte, welche an im *Groupware Development* verbreitete Konzepte angelehnt sind, für Entwickler empfohlen. Obwohl 2011, zum Zeitpunkt der Studie, die Browser-Unterstützung noch nicht zufriedenstellend war, wurden WebSockets als künftiger Standard im *Groupware Networking* im Browser vorhergesagt. Bei der zu dieser Arbeit gehörenden Implementierung einer Web-Realtime CMS Erweiterung wurde deshalb und aufgrund des Vorhandenseins ausgereifter Programmbibliotheken, der Spezifikation [6], ihrer weiten Verbreitung und aktiven Web-Gemeinschaft, WebSockets als bevorzugter Transportdienst zwischen Client und Server verwendet.

5.2 Architektur

Die Architektur der Implementierung besteht aus vier eigenständigen *Repositories*.[15] Diese Programmkomponenten sind teilweise voneinander abhängig und spielen je nach ihren Aufgabengebieten zusammen.

5.2.1 Web-Realtime Package

Dieses Verzeichnis beinhaltet ein PHP-basiertes, via *Composer* installierbares Paket von PHP-Code, welches abstrakte Implementierungen für Web-Realtime Komponenten, wie Kanäle (*Channels*), *Radios*, Ereignisse (*Events*) oder Programmcode-Vorlagen (*Code Generateables*) zur Verfügung stellt. Diese Vorlagen beinhalten wiederum Standard Implementierungen für Pub/Sub Grundfunktionsweisen. Außerdem sind verschiedene Implementierungen für die Server-Server Kommunikation zwischen *Node.js* und *Apache* und die Logik der Code-Generierung für Client-seitige Kanal-Abonnements (siehe Abschnitt 4.3.1), wie auch deren *Sails.js*-spezifische Standard-Implementierung darin enthalten.

5.2.2 CMS Erweiterung

Diese Erweiterung ist ein Plugin für Pimcore, welches die gesamte Pimcore-spezifische Web-Realtime Logik enthält. Es ist implementiert unter Verwendung, der in Abschnitt 5.1.1 beschriebenen Möglichkeiten zur Erweiterung von Pimcore und installierbar via der CMS-Administrationsoberfläche. Das Plugin beinhaltet alle Standard-Kanäle innerhalb der Applikation, das Senden von automatisierten Events für Objekt-Kanäle, Konfigurationsdateien, jede Logik zur Adaption der CMS Administrationsoberfläche, Models zur Speicherung aktueller Objektlisten-Abonnements und auch die Authentifizierungsschnittstelle für WebSocket-Anfragen.

5.2.3 CMS Web Applikation

Dieses Verzeichnis beinhaltet eine auf Pimcore 4.6 basierende Web Applikation im Beispiel einer Produktgallerie. Diese bietet Filterfunktionen nach vordefinierten Kategorien und Hyperlinks zu Detailseiten der gelisteten Produkte. Außerdem gibt es die Option Produktrezensionen zu erstellen, welche auf den Detailseiten, wie auch in der Gallerie konditional aufgelistet werden.

5.2.4 Web-Realtime Dienst

Der Web-Realtime Dienst wurde in Form einer *Node.js* basierten Applikation umgesetzt und basiert auf dem *Sails.js* Framework. Der Dienst beinhaltet die Logik um Clients die Möglichkeit zu bieten, spezifische Kanälen innerhalb der Applikation zu abonnieren und auch die Funktionalitäten um Daten wiederum zu den verbundenen Clients zu senden. Außerdem umfasst der Dienst die Authentifizierungslogik, um CMS-interne Anfragen anhand der Pimcore Benutzer- und Rechteverwaltung zu authentifizieren,

[15] Projektverzeichnis

Programm 5.1: Einschubmethode, welche vor dem Objekt Abonnement ausgeführt wird. In diesem Beispiel würde das Abonnement nur ausgeführt werden, wenn der Aufruf nicht im Bearbeitungsmodus (*Editmode*) von Pimcore passiert, welcher aktiv ist, wenn ein Dokument im Administrationsmodus in der Pimcore Administrationsoberfläche geöffnet wird.

```
1 public function beforeSubscribe()
2 {
3     return !$this->view->editmode;
4 }
```

die TCP basierte *ZeroMQ* Implementierung für Server-Server Kommunikation, Code-Generatoren für Controller und Models neu erstellter Pimcore-Klassendefinitionen, welche als Web-Realtime definiert wurden und auch die Logik zur Übersetzung der Events für Objektlisten-Kanäle.

5.3 Umsetzung der Web-Realtime Kanäle

In diesem Abschnitt wird die Anwendung der in Abschnitt 4.3 vorgestellten Konzepte genauer dargelegt und die bereitgestellte Web-Realtime PHP-API im Detail erklärt. Außerdem werden die Verwendung und das Zusammenspiel der einzelnen Komponenten der Systemarchitektur auf Basis des implementierten Anwendungsbeispiels einer Produktgallerie dargelegt.

5.3.1 Klassenstruktur

Die Eigenschaften eines Kanals wurden in zwei PHP-Interfaces logisch aufgetrennt. Zum einen muss das PHP-Interface `PubSubable` implementiert werden, um die Methoden `publish()`, `subscribe()` und `unsubscribe()` zu definieren. Zum anderen muss auch `IRealtimeChannel` implementiert werden, da ein Kanal durch eine eindeutige Zeichenkette identifiziert wird und um es zu ermöglichen den Verbindungsdienst (z. B. HTTP/TCP) dynamisch zu definieren. Eine Standard Implementierung des `PubSubable` Interface wird als PHP Trait `PubSub` zur Verfügung gestellt, welcher es außerdem optional ermöglicht, `beforeSubscribe()` und `beforePublish()` zu implementieren, um benutzerspezifischen Code auszuführen oder die Aktionen unter Rückgabe eines Boolean Wertes `false` abzubrechen (siehe Prog. 5.1. Die Auslagerung der Implementierung in einen Trait hat den Grund, dass es möglich sein soll, dieselbe Implementierung für Benutzer- bzw. CMS-spezifische Kanäle, wie auch für Objekt-basierte Kanäle anwenden zu können.

5.3.2 Objektinstanz-Kanäle

Objekte sind Datenhalter innerhalb von Pimcore und ermöglichen es, Daten strukturiert zu verwalten. Sie basieren auf einer frei konfigurierbaren Klassendefinition, welche die Struktur und die Attribute aller Objekte dieser Klasse definiert. Es lässt sich

erahnen, dass Objekte innerhalb Pimcores auch Objekte nach dem Sinn der Objektorientierten Programmierung darstellen und über eine PHP-API, welche übliche ORM-Funktionalitäten zur Verfügung stellt, angesprochen werden kann. Die Klassendefinition ist über die Adminstrationsoberfläche von Pimcore konfigurierbar, hier ist es auch möglich, Eltern-Klassen[16] zu definieren und somit die Methoden und Parameter der Objekte zu erweitern. Diese Möglichkeit der Erweiterung wird nun genutzt um Objekte in Kanäle zu verwandeln, indem sie von einer Klasse `ObjectChannel` ableiten. Diese Klasse muss dann wiederum von der ursprünglichen Eltern-Klasse der Pimcore Objekte erben (`Pimcore\Model\Object\Concrete`), um die Funktionalitäten zur Datenbankschnittstelle und somit auch zur Pimcore Administrationsoberfläche beizubehalten. Somit wird eine Objektinstanz zu einem Web-Realtime Kanal und es wird möglich, Objektinstanzen zu abonnieren:

```
$products = new \Pimcore\Model\Object\Product\Listing();

foreach ($products as $product) {
    $product->subscribe();
}
```

Außerdem wird ermöglicht, neben den automatisierten Ereignissen auch beliebige benutzerspezifische Nachrichten zu einer spezifische Instanz zu veröffentlichen:

```
$product = \Pimcore\Model\Object\Product::getById(1234);
$product->publish(new Message(["foo" => "bar"]));
```

Eine abstrakte Modellierung der beschriebenen Klassenstruktur als UML-Diagramm[17] ist in Abb. 5.2 dargestellt. Der standardmäßige Verbindungsdienst für Objektinstanz-Kanäle verwendet HTTP, um den vollen Funktionsumfang des *Sails.js* Routings nutzen zu können (siehe Abschnitt 5.3.6).

Automatisierung der relevanten Events

Bei Objektinstanz-Kanälen handelt es sich um automatisierte Kanäle, dies bedeutet, sie erhalten Benachrichtigungen zu relevanten Ereignissen automatisch, für Objektinstanzen sind dies `Destroy` und `Update` Events. Bei der Realisierung dieser Art von Kanälen wurde die Event API von Pimcore verwendet, um via des Zend Event-Managers, welcher innerhalb von Pimcore für die Ereignisverwaltung angewendet wird, bei signifikanten Ereignissen innerhalb der CMS Applikation (siehe Prog. 5.2), Web-Realtime Events zum *Node.js* Server bzw. zum Client zu senden.

5.3.3 Objektlisten-Kanäle

So wie Objektinstanz-Kanäle das Gegenstück zu *Sails.js Model-Subscriptions* darstellen, ist das Verhalten von Objektlisten-Kanälen ähnlich zu der von *Sails.js Model-Watches*, welche es ermöglichen, den gesamten Klassenraum eines Modells zu abonnieren und somit Benachrichtigungen zur Erstellung bzw. Löschung jeglicher Instanzen einer Klasse zu erhalten. Objektlisten können in Pimcore allerdings selbstverständlich auch Bedingungen enthalten, sodass nur Objekte, die diese Bedingung erfüllen, zur Liste hinzugefügt werden [29]. In der Signatur zur Identifizierung von Objektlisten-Kanälen ist

[16] Parent classes
[17] Abk.: Unified Modeling Language

Programm 5.2: Event-Management von Pimcore via des Zend Event-Managers. Dieses wird verwendet für automatisierte Kanäle innerhalb des CMS.

```
1 $eventManager = \Pimcore::getEventManager();
2
3 $eventManager->attach("object.postAdd", function (\Zend_EventManager_Event $e) {
4     $object = $e->getTarget();
5
6     if ($object instanceof \Realtime\Channel\Object\ObjectChannel
7             && $object->getPublished())
8     {
9         $object->publish(new \RT\Event\Create());
10    }
11 });
```

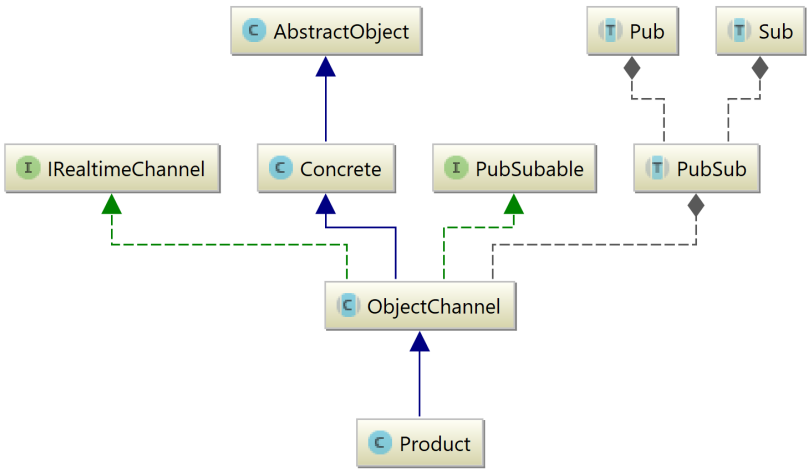


Abbildung 5.2: UML-Diagramm der Struktur für Objektinstanz-Kanäle im Beispiel eines Produkt Objekts.

die base64-kodierte Bedingung der Liste enthalten. Eine Client-seitige Abonnement-Aufforderung für die Bedingung `category = 'fashion'` würde sich demnach wie folgt zusammensetzen:

```
socket.get('/product/watch/Y2F0ZWdvcnkgPSAnZmFzaGlvbic=')
```

Objektlisten-Kanäle sind wie Objektinstanz-Kanäle ebenfalls automatisiert und erhalten somit `Create`- und `Destroy`-Events automatisch. Die Möglichkeit einer definierten Bedingung bedeutet im Bezug auf Web-Realtime Events allerdings auch, dass ein Pimcore internes `Update` Event eines Objekts möglicherweise zu einem Web-Realtime `Create`- oder `Destroy`-Event für einen Listen-Kanal übersetzt werden muss, falls die definierte Bedingung nach der Aktualisierung erfüllt bzw. nicht mehr erfüllt wird. Auch müssen die betroffenen Objekte für Pimcore `Create`- und `Delete`-Ereignisse gegen die gegebenen Bedingungen überprüft werden, ob sie für die jeweilige Liste von Relevanz sind. Zur Ermöglichung dieses Übersetzungsvorgangs sind zwei Voraussetzungen zu erfüllen: Zum

einen ist es notwendig, zu jedem Zeitpunkt alle Objektlisten mit verbundenen Client-Sockets zu kennen und zum anderen muss es möglich sein, die Parameter des aktualisierten Objekts vor und nach dem Update vergleichen zu können. Hierfür wurden zwei neue Datenbanktabellen – somit auch Pimcore und *Sails.js* Models – erstellt, um aktive Objektlisten und die zugehörigen verbundenen WebSockets zu persistieren. Sobald eine Objektliste, deren Klassendefinition `ObjectChannel` als Parent-Class definiert hat, abgefragt wird, wird ein neuer Objektlisten-Eintrag erstellt und ebenso die zugehörige WebSocket-Id wird mit Referenz zur Objektliste gespeichert. Die verbundenen Sockets werden am *Node.js* Server bei Aufruf eines Listen-Abonnements hinzugefügt und können im *Sails.js* Framework mittels eines Hooks `afterDisconnect()` wieder entfernt werden. Ein Skript welches mit dem Pimcore Maintenance Cron Job in regelmäßigen Abständen ausgeführt wird, entfernt alle Objektlisten-Einträge die aktuell keine verbundenen WebSocket Verbindungen referenzieren. Bei Objekt-Update Ereignissen wird immer eine aktuelle (`current`) und eine vorherige (`previous`) Version der Web-Realtime relevanten Parameter des Objekts zum *Node.js* Server übertragen. Am *Node.js* Server wird dieses Update Event dann übersetzt, indem alle gespeicherten Objektlisten-Kanäle der gegebenen Klasse iteriert werden und deren Bedingungen (z. B. `category = 'fashion'`) für die Current- und die Previous-Version überprüft werden. Falls eine zuvor zutreffende Bedingung nicht mehr gilt, wird ein `Destroy`-Event an die jeweilige Liste gesendet, falls eine Bedingung nun zutrifft, ein `Create`-Event.

Instanzierung: Da für Objektlisten keine Manipulation der Klassenhirachie wie für Objektinstanzen möglich ist, werden Objektlisten-Kanäle, wie auch alle anderen CMS internen Kanäle, über die PHP Klasse `\Realtime\Radio` instantiiert:

```
$products = new \Pimcore\Model\Object\Product\Listing();
$produts->setCondition("category = 'fashion'");

$fashionProductsChannel = \Realtime\Radio::getObjectListChannel($products);
```

5.3.4 Anwender- und CMS-spezifische Kanäle

Die bereitgestellte PHP-API soll es ermöglichen, beliebige Kanäle innerhalb der Applikation erstellen zu können.

Standard-Kanäle

Standard-Kanäle können via folgender Schnittstelle instantiiert werden, optional kann als zweiter Parameter der Verbindungsdienst verändert werden:

```
$channel = \Realtime\Radio::getChannel($identifier)
```

Dies erstellt einen Kanal mit Standard Pub/Sub Funktionalitäten, identifiziert durch eine eindeutige Zeichenkette und mit TCP als standardmäßigen Transportdienst zum Web-Realtime Dienst. Kanäle, welche über diese Schnittstelle erstellt werden, sind Instanzen der Klasse `\Realtime\Channel\DefaultChannel` und bieten alle Standard Pub/Sub Funktionen. Die Client-seitige Schnittstelle für Abonnements dieser Art von Kanälen setzt sich dann wie folgt zusammen:

```
socket.get('/app/sub/' + identifier);
```

Diese Art der Kanal-Instantiierung ist vor allem dann dienlich, wenn keinerlei spezifische Logik benötigt wird, lediglich ein eindeutig identifizierbarer Web-Realtime Kanal, wie z. B. für einen globalen News-Kanal.

Spezifische Kanäle

Um Kanäle zu erstellen, welche eine spezifische Schnittstelle zum Web-Realtime Dienst beinhalten oder nur unter bestimmten Bedingungen Abonnements oder Veröffentlichungen von Events erlauben sollen, kann eine neue Klasse definiert werden, welche `\RT\Channel\AbstractChannel` als Parent definiert. Dadurch werden die vollständigen Publish-Subscribe Funktionen bereits eingebunden. Zusätzlich können die Parameter `room` und `identifier`, aus denen sich die Pfad-Signatur zum Web Realtime Dienst zusammensetzt, definiert werden. Wie auch bei Instanzkanälen, ermöglichen die Einschubmethoden `beforeSubscribe()` und `beforePublish()`, das Ausführen der Aktionen unter Rückgabe eines Boolean-Wertes `false` an eine Bedingung zu knüpfen, oder zusätzliche Logiken auszuführen. Kanäle für Pimcore Dokumente oder auch ein globaler Website-Kanal wurden in dieser Weise implementiert und sind durch die Klasse `\Realtime\Radio` abrufbar. Ein Dokument-Kanal bietet zusätzlich eine Schnittstelle, um Sub-Kanäle für Dokument-Editables hinzuzufügen (siehe Abschnitt 5.4.2). Abonniert ein Benutzer einen Dokument-Kanal, abonniert er somit automatisch auch alle Editable-Kanäle innerhalb der Dokument-Template-Datei.

5.3.5 Interaktionen mit Kanälen

Jeder Kanal bietet Schnittstellen für Pub/Sub Funktionalitäten, der Ablauf dieser Methoden wird in diesem Abschnitt genauer beschrieben.

Abonnements

Die Abonnements erfolgen, wie in 4.3.1 beschrieben, Client-seitige, somit generiert jeder Aufruf eines `$channel->subscribe()` eine zusätzliche Abonnement-Aufforderung, welche an den JavaScript Code des Clients angefügt wird. Hierfür wird eine Klasse `Codebase` bereit gestellt, welche es ermöglicht an, definierten Stellen im HTML-Dokument JavaScript-Code anzufügen. Es werden Standard-Implementierungen des PHP-Interfaces `Generateable` für Pub/Sub-Skripte bereitgestellt. Ein solches Skript kann wie folgt angefügt werden, unter Verwendung des Standard `PubSub` Traits passiert dies für Abonnements automatisch:

```
$codebase = ServiceLocator::instance()->getCodebase();
$subscription = new Subscription($signature);

$codebase->add($subscription, Placement::RE_CONNECT_CALLBACK_URL());
```

Bei diesem Vorgang wird darauf geachtet, dass die Anfragen in der Rückruf-Funktion[18] des Wiederverbindungs-Ereignisses der Web-Realtime Verbindung passieren, um nach einem möglichen Verbindungsverlust automatisch wieder alle Abonnements initialisieren zu können. Außerdem werden die hinterlegten Abonnement-URLs außerhalb dieser

[18] Callback

Programm 5.3: Automatisch generierter JavaScript-Code unter Berücksichtigung des Wieder-Verbindungsvorgangs (*reconnect*) für Abonnement-Aufforderungen.

```
1 var reconnect = [];
2
3 reconnect.push("/product/sub/1");
4 reconnect.push("/product/sub/2");
5 reconnect.push("/productreview/watch/Y2F0ZWdvcnkgPSAnYm9va3MnIA==")
6 ...
7
8 var handleReconnect = function() {
9     reconnect.forEach(function(url) {
10         socket.get(url, function (response) {
11             console.log(response);
12         });
13     });
14 }
15
16 socket.on('reconnect', handleReconnect);
```

Funktion definiert. Dies ist essentiell, um die Abonnements dynamisch unter laufender Applikation anpassen zu können, ohne dazu die Website neu laden zu müssen. Der generierte Code für Kanal Abonnements stellt sich demzufolge wie in Prog. 5.3 zu sehen dar.

Senden von Events

Über die Methode `$channel->publish()` können beliebige Objekte die das Interface `IEvent` implementieren an den *Node.js* Server gesendet werden. Unter Verwendung einer HTTP-Verbindung zum Web-Realtime Dienst setzt sich der Pfad der gesendeten Events aus den zwei Parametern `room` und `identifier` innerhalb der Kanal-Signatur und dem `verb` des Event-Objekts selbst zusammen:

```
$url = "/{$channel->getRoom()}/{$event->getVerb()/{$channel->getIdentifier()}";
```

Wird TCP als Verbindungsdienst verwendet, wird die Signatur zur Identifizierung des WebSocket-Rooms an das übertragene JSON-Objekt angefügt. Alle Event-Objekte müssen eine Methode `getJsonData()` implementieren, um relevante Parameter in JSON kodierter Form für die Übertragung exportieren zu können. Das Schema der zum Client übertragenen Event-Objekte ist angelehnt an das des *Sails.js* Frameworks, ein Event-Objekt beinhaltet wie in der Dokumentation von *Sails.js* [44] beschrieben demnach folgende Parameter:

`verb` Beinhaltet die Bezeichnung der Aktion des Events. Die Aktionen der bereitgestellten Events lauten: *created*, *destroyed*, *updated*, *messaged*. Anders als in der *Sails.js* Konvention spezifiziert, können auch beliebige benutzerspezifische Events und Aktionen definiert werden.

`id` Enthält den Pimärschlüssel der Objektinstanz.

`data` Beinhaltet die relevanten Daten des jeweiligen Events und kann in Pimcore für automatisch gesendete Events via der Methode `getRealtimeData()` im Pimcore Objekt überschrieben werden. Bei Events des Typs *destroyed* ist dieser Parameter nicht vorhanden.

`previous` Dieser Parameter ist nur vorhanden für *destroyed* und *updated* Events und beinhalten die Daten vor dem Ausführen des Ereignisses.

Um auf Seiten des Clients auf die empfangenen Events reagieren zu können, bietet *Sails.io.js* eine JavaScript-API an. Der *Sails.js* Konvention folgend ist der Name des Client-seitigen Events der Name des Modells im Sinne des MVC-Prinzips und somit auch des Pimcore-Objekts:

```
socket.on('product', function (event) {
    Product.route(event);
});
```

Szenarien der Event-Entstehung: Wie in Abschnitt 4.2.1 bereits erwähnt, ist es essentiell die Entstehungsszenarien der möglichen Events zu identifizieren. Aus technischer Sicht sind hierbei, wie in Abb. 5.3 dargestellt, drei Fälle relevant:

1. Events ausgelöst durch jegliche Interaktionen zwischen Client und der Pimcore Applikation.
2. Aus serverseitigen Prozessen ausgelöste Events, wie z. B. durch Cron-Jobs oder anderen asynchron gestarteten serverseitigen Prozessen.
3. Interaktionen direkt von Client zu Client via des Web-Realtime Dienstes z. B. innerhalb eines WebSocket-Rooms, für Online-Spiele usw.

5.3.6 Web-Realtime Dienst

Wie in Absch. 4.3.1 beschrieben, gibt es zu jedem Kanal auf Seiten des *Node.js* Servers einen zugehörigen Socket Room, welcher über eine eindeutige Zeichenkette identifiziert werden kann und zu welchem sich beliebig viele Sockets verbinden können. Für nicht Objekt-basierte Kanäle wird eine generische Schnittstelle für Pub/Sub-Funktionalitäten wie Abonnements und Weiterleitung der Events zum Client zur Verfügung gestellt. Für Objekt-Kanäle wurden spezielle *Sails.js* Generatoren implementiert, diese ermöglichen es über ein Consolen-Kommando dynamisch Dateien innerhalb der *Sails.js* Applikation via definierter Template-Dateien zu generieren [45]. Sie werden ausgeführt, sobald eine Klassendefinition, die `ObjectChannel` als Parent-Klasse definiert hat (siehe Abschnitt 5.3.2), gespeichert wird. Diese Generatoren erstellen für die jeweilige Klasse einen zugehörigen Controller und ein Modell auf Seiten des *Node.js* Servers, welche bereits alle notwendigen Methoden und Attribute beinhalten. Die Ausführen der Generatoren passiert via der Event API von Pimcore und wird ausgelöst von den PHP-

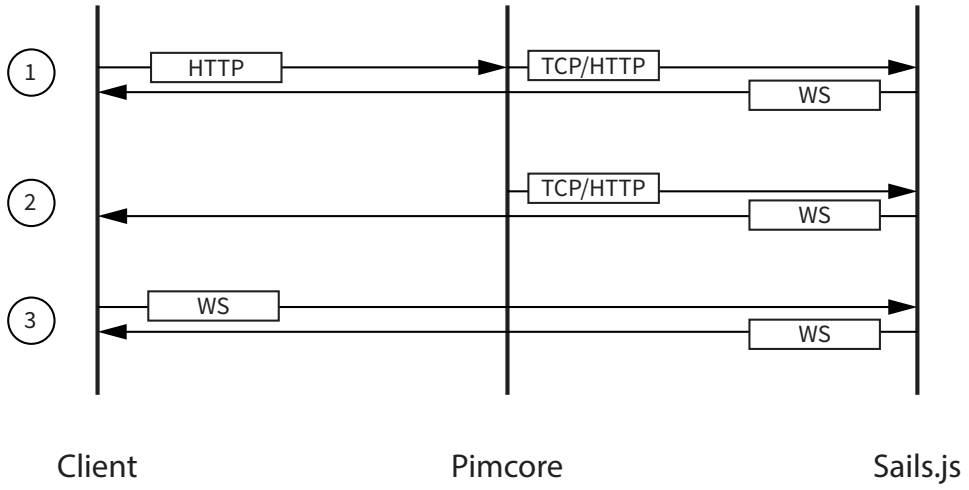


Abbildung 5.3: Szenarien der Web-Realtime Event-Entstehung.

Events `object.class.postAdd` und `object.class.postUpdate`. Die Consolen-Befehle zum Ausführen der Generatoren lauten wie folgt:

```
exec("sails generate rt-model {$classId} {$tablename}");
exec("sails generate rt-controller {$classId} {$tablename}");
```

Der *Sails.js* Autoreaload-Hook[19] ermöglicht die dynamische Einbindung dieser generierten Dateien, ohne den *Node.js* Server regelmäßig neu starten zu müssen. Auch das Routing aller generierten Controller-Actions wird durch das *Blueprint-Routing* des *Sails.js* Frameworks ohne explizite Definition ermöglicht. Des Weiteren wurde ein Authentifizierungsdienst zur Pimcore Applikation implementiert, welcher in Kombination mit diversen definierten *Policies* sicherstellt, dass das Rechte-Managements von Pimcore auch bei WebSocket Anfragen und empfangenen Events nicht umgangen werden kann (siehe Prog. 5.4). Als Verbindungsdienst zwischen *Node.js* und *Apache* wird neben der HTTP/WS Schnittstelle des *Sails.js* Frameworks, auch eine direkte TCP Verbindung via *ZeroMq* angeboten. Hierbei werden die Events allerdings lediglich zum jeweiligen Socket-Room durchgeschleust, bereits erwähnte Funktionen des *Sails.js* Frameworks wie Route-basierte Policies werden nicht unterstützt.

5.4 CMS Integration und Endanwendung

Aufbauend auf den in Abschnitt 5.3 beschriebenen PHP-API wurde das CMS mit einigen Web-Realtime Komponenten erweitert. Diese sollen neben der implementierten Endanwender Applikation der Produktgallerie als *Proof-Of-Concept*[20] dienen, welche neuen Möglichkeiten durch die Erweiterung nun umgesetzt werden können.

[19] https://github.com/sgress454/sails-hook-autoreload
[20] Machbarkeitsnachweis bzw. Machbarkeitsstudie

Programm 5.4: Die definierten Policies für Controller-Actions innerhalb der *Sails.js* Applikation in der Konfigurationsdatei `/config/policies.js` ermöglichen Request basierte Authentifizierung. Im diesem Beispiel würden die Policy-Dateien `websocket.js`, `auth.js` und `document.js` den aktuellen Request nur zur Controller-Action `subsribe()` weiterleiten, falls die Anfrage über eine WebSocket Verbindung erfolgt ist, eine aktive Pimcore Session besteht und dieser Benutzer die benötigten Rechten für das gegebene Pimcore Dokument besitzt.

```
1 module.exports.policies = {
2   DocumentController: {
3     "subscribe": ["websocket", "auth", "document"]
4   }
5 }
```

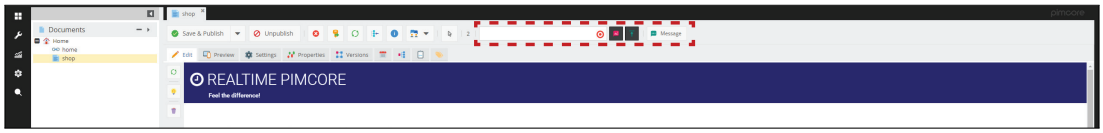


Abbildung 5.4: Die neuen Elemente der Pimcore Administrationsoberfläche ermöglichen das Senden von Textnachrichten, Objekte oder Assets zum Client für Dokumente und Objekte innerhalb des CMS.

5.4.1 Elemente der Administrationsoberfläche

Jedes Pimcore Dokument und jedes Objekt, welches durch die Definition der Klasse `ObjectChannel` als Parent-Class, als Web-Realtime Objekt deklariert wurde, wurde in der Administrationsoberfläche, wie in Abb. 5.4 ersichtlich, durch neue Elemente in der Haupt-Toolbar erweitert. Durch diese Elemente wird es möglich über ein Pop-Up Fenster einfache Textnachrichten, durch einen Direkt-Upload Bilddateien oder via Drag-and-Drop auch Pimcore Objekte und Assets an den Kanal des geöffneten Elements zu senden. Dies wurde über die JavaScript-Hooks `postOpenObject()` und `postOpenDocument()` der *startup.js* Datei des implementierten Pimcore Plugins umgesetzt. So ist es für Redakteure des CMS möglich, direkt über die Administrationsoberfläche Daten an den Client zu senden.

5.4.2 Dokument-Editables

Die editierbaren Bereiche innerhalb von Pimcore Dokumenten (*Document Editables*) sind eine essentielle Komponente des Templating-Konzepts von Pimcore [30]. Es handelt sich dabei um Platzhalter, die innerhalb eines Pimcore Dokuments definiert werden können und unterschiedliche Funktionen verschiedener Komplexität zur Verfügung stellen, wie z. B. einfache Textfelder, WYSIWYG-Bereiche[21], oder auch Platzhalten für Videos, Renderlets usw. Außerdem ist es möglich, benutzerspezifische Editables zu definieren.

[21] What you see is what you get.

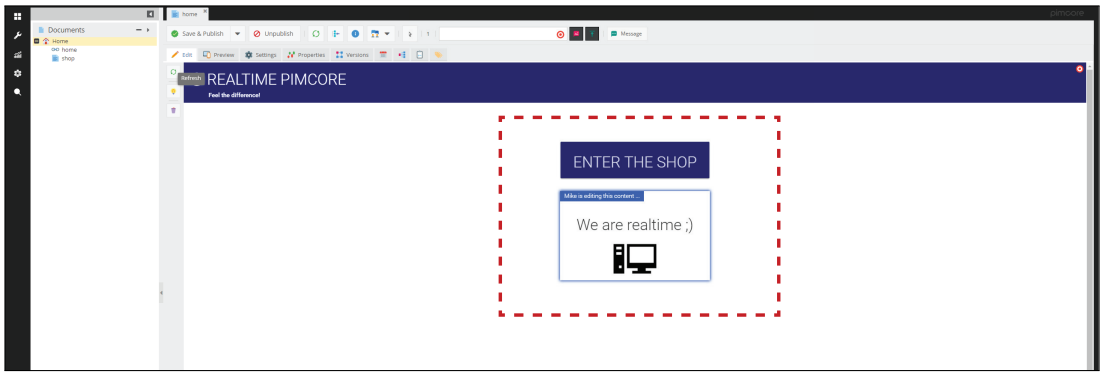


Abbildung 5.5: Kollaborativer Web-Realtime WYSIWYG-Editor als neues Pimcore Dokument Editable.

Web-Realtime WYSIWYG

Basierend auf dem bestehenden WYSIWYG-Editor von Pimcore, welcher auf dem bekannten Open-Source Projekt *CKEditor*[22] basiert, wurde ein neues Dokument-Editable erstellt, welches den Editor um kollaborative Funktionalitäten erweitert (siehe Abb. 5.5). Jedes Editable bekommt einen eigenen authentifizierungpflichtigen Web-Realtime Kanal zugewiesen, welcher als Sub-Kanal des zugehörigen Dokument-Kanals registriert wird. Somit abonniert man alle in einem Dokument enthaltenen Editable-Kanäle automatisch im Zuge eines Dokument-Abonnements. Die Authentifizierung erfolgt via der bereitsgestellten HTTP-Schnittstelle zu Pimcore und wird derzeit für jede gesendete WebSocket-Nachricht überprüft. Der Web-Realtime Editor kann dann innerhalb eines Dokument-Templates wie folgt instantiiert werden:

```
$this->rtwysiwyg("collaborative-content");
```

Funktionsweise: Fängt ein CMS Anwender damit an, den WYSIWYG-Bereich zu bearbeiten indem er ihn fokussiert, wird der Editor für alle anderen Anwender gesperrt und alle Änderungen werden in Echtzeit zu allen verbundenen Clients übertragen. Ein Info-Bereich oberhalb des Editors informiert andere Anwender welcher Anwender den Editor derzeit bearbeitet.

5.4.3 Endanwendung

In diesem Teil der Implementierung befindet sich vorrangig Anwendungs-spezifischer Programmcode zur Ereignisverwaltung[23] der empfangenen Nachrichten. Diese Implementierungen sind im Allgemeinen dem Entwickler der Applikation überlassen und werden hier als Beispiel der Herangehensweise und Proof-Of-Concept angeführt. Die als Beispiel implementierte Endbenutzer-Anwendung beinhaltet grundsätzlich drei verschiedene Webpage-Typen:

[22] https://ckeditor.com/
[23] Event handling

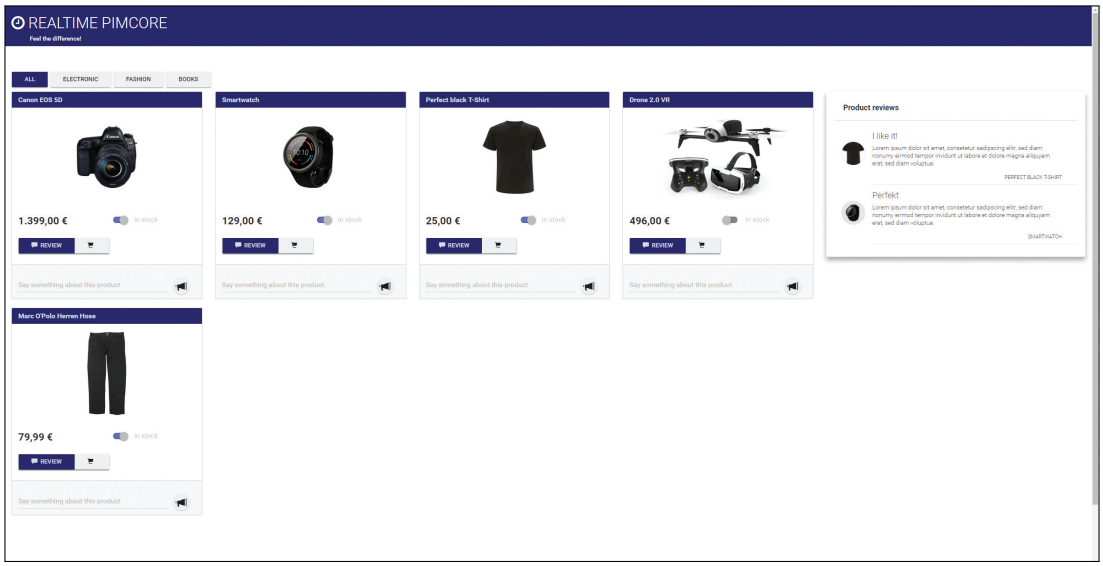


Abbildung 5.6: Screenshot der Produktgallerie der Beispiel-Applikation. Innerhalb der Applikation werden alle Parameter der gelisteten Objekte mit den serverseitigen Daten der Pimcore Objekte synchronisiert.

1. Landing-Page: Diese Webpage ist CMS-seitig die Wurzel der Pimcore Dokumentstruktur (*Root-Dokument*) und beinhaltet den in Abschnitt 5.4.2 beschriebenen Web-Realtime WYSIWYG-Editor. Ein Link zentral unterhalb des Editors führt weiter zur Übersichtsseite der Produktgallerie.
2. Produktgallerie: Hier befindet sich eine Produktauflistung in Form eines Rasters mit Filterfunktion nach Produktkategorie (siehe Abb. 5.6). Neben des Rasters ist eine Liste der zu den aufgelisteten Produkten gehörenden Produktrezesionen. Das zugehörige Dokument seitens des CM-Systems lautet `/shop`.
3. Produkt-Detailseiten: Jedes der gelisteten Produkte hat eine entsprechende Detailseite, mit erneuter Auflistung der zugehörigen Rezesionen (siehe Abb. 5.7). Diese Seite hat kein zugehöriges Dokument, sondern ist über eine statische Route zugänglich. Die dargestellten Daten stammen ausschließlich aus dem Produktobjekt.

Kanäle innerhalb der Anwendung

Eine jede Webpage innerhalb der CMS-Anwendung kann, wenn nicht explizit deaktiviert, über einen globalen Webseiten-Kanal angesprochen werden, egal ob sie ein zugehöriges Dokument innerhalb des CMS besitzt. Dokument-basierte Seiten haben zusätzlich einen expliziten Kanal, zu welchem via der PHP-API wie auch über die Administrationsoberfläche des CMS Daten via Web-Realtime gesendet werden können. Dasselbe gilt für Objekte innerhalb des CMS, für welche die Web-Realtime Funktionalität aktiviert wurde – auch sie besitzen einen expliziten Kanal. Dieser Kanal wird dafür angewendet automatisiert Ereignisse zu *Update*- und *Destroy*-Operationen zu den Objekten zu empfangen und ihre Daten somit synchronisieren zu können. In der Beispiel-Anwendung wurde dies für Produkt-Objekte realisiert. Die letzte Art der derzeit verwendeten Kanäle

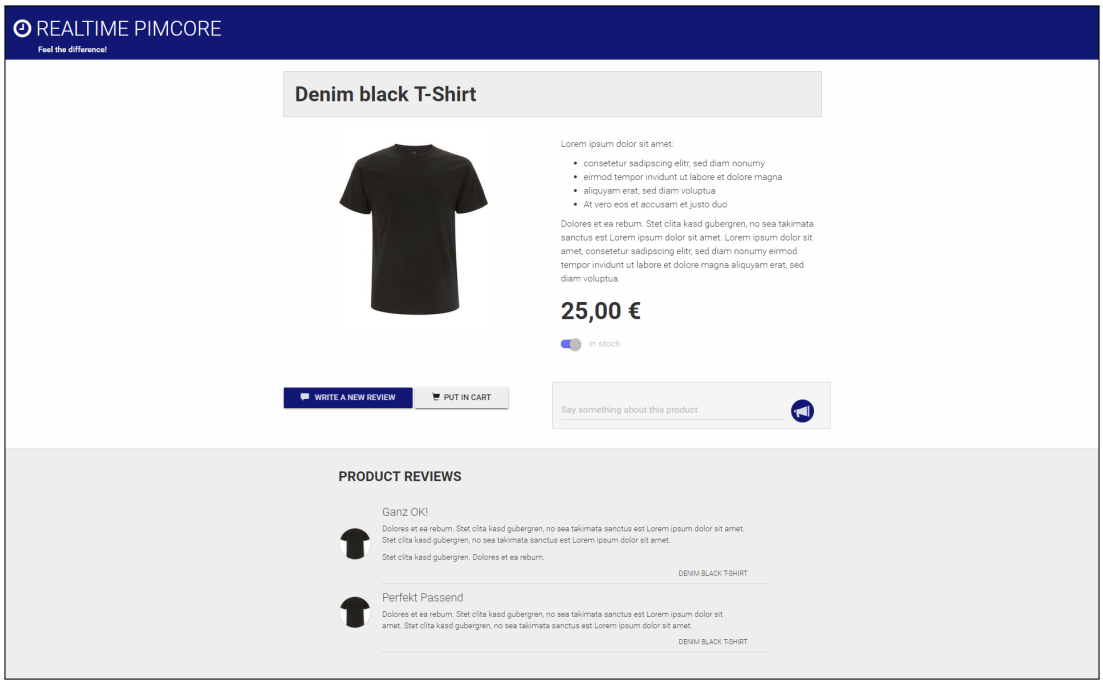


Abbildung 5.7: Screenshot einer Detailseite der Produkt-Objekte. Alle Parameter des Produkts und die Produktrezensionen sind stets synchron mit dem CMS.

sind Objektlisten-Kanäle, welche nicht Ereignisse zu den Parameter der Objekte erhalten, sondern wie in Abschnitt 5.3.3 beschrieben, anhand einer möglichen definierten Bedingung Benachrichtigungen zu neu hinzugekommenen Objekten bzw. von der Liste entfernte Objekte erhalten. Dies wurde innerhalb der Beispiel-Anwendung für produktspezifische Rezensionen angewendet.

Client-seitige Ereignisverwaltung

Alle Ereignis-Benachrichtigungen, welche via der Benutzeroberfläche des CMS empfangen werden, werden sichtbar als Datenobjekte innerhalb der Seite ausgegeben (siehe Abb. 5.8). Automatisierte Ereignisse zu *Create*-, *Update*- oder *Destroy*-Operationen von Datenobjekten werden dazu verwendet, um Daten live mit dem Server zu synchronisieren. Um dies zu ermöglichen, ist eine Client-seitige Ereignis-basierte Weiterleitung[24] notwendig. Hierfür könnte ein Ereignis-basiertes Rendering-Framework genutzt werden, wie z. B. *react.js*[25], *backbone.js*[26], *ember.js*[27] usw. Für die implementierte Beispiel-Applikation würde dies allerdings aufgrund der geringen Komplexität den Rahmen sprengen. Hier wurde das Client-seitige Rendering, basierend auf den Pimcore Datenobjekten und deren Parametern, selbst implementiert. Hierbei können Pimcore-Objekte, identifiziert anhand ihrer im CMS zugewiesenen eindeutigen Klassenbezeichnungen, de-

[24] Routing
[25] https://facebook.github.io/react/
[26] http://backbonejs.org/
[27] https://www.emberjs.com/

Programm 5.5: Client-seitige Ereignis-Weiterleitung zur Handhabung von Ereignissen basierend auf CMS-Objekten.

```
1  var objects = ["product", "productreview"];
2
3  var initEventRouting = function (object) {
4
5      socket.on(object, function (event) {
6          var module = app[object];
7
8          if(typeof module === "object" && typeof module[event.verb] === "function")
9              module[event.verb](event);
10         else
11             console.warn("Routing for module app." + object + " not possible.");
12     });
13
14 };
15
16 objects.forEach(initEventRouting);
```

Abbildung 5.8: Ausgabe einer über das CMS empfangenen Benachrichtigung. Im Beispiel wurde ein weiteres Produkt gesendet, welches z. B. für eine proaktive Werbeanzeige oder zum Cross-Selling verwendet werden könnte.

finiert werden, welche als Web-Realtime gehandhabt werden sollen. Zu jedem dieser definierten Objekte muss ein zugehöriges JavaScript-Modul existieren, welche eine Schnittstelle für die in Abschnitt 5.3.5 erwähnten möglichen *verb*-Parameter (*created, updated, destroyed, messaged*) bereitstellt, um das Ereignis zu behandeln (siehe Prog. 5.5).

Kapitel 6

Evaluierung

6.1 Anforderungsanalyse

Im folgenden Abschnitt wird das angewendete Konzept und die implementierte Lösung kritisch betrachtet und evaluiert. Die Evaluierung erfolgt anhand der in Abschnitt 4.1 definierten Anforderungen.

6.1.1 CMS-Autonomie

Bei der Implementierung der Erweiterung wurde darauf geachtet, dass mögliche Fehler, welche im Prozess des Senden der Web-Realtime Events auftreten können, die eigentliche Operation nicht beeinträchtigen. Dass zusätzlicher Programmcode zusätzliche potentielle Fehlerquellen mit sich bringt, ist nicht abwendbar.

Verarbeitungszeit

Die Anfragen, welche von Pimcore zum Web-Realtime Dienst gesendet werden, blockieren die weitere Verarbeitung und verzögern das Senden einer Antwort zum Client und somit die insgesamte Verarbeitungszeit der initialen Anfrage. Um diesen Mehrwert zu minimieren, werden Web-Realtime Events, welche am *Node.js* Server eingelangen, lediglich auf ihre benötigten Parameter validiert und gleich danach wird eine entsprechende Server Antwort an Pimcore zurückgeschickt. Die Verarbeitung der Anfrage des Web-Realtime Events auf Seiten des *Node.js* Servers und auch das Senden der Nachrichten zum Client via der Web-Realtime Verbindung passiert erst nach der Server Antwort zu Pimcore und somit asynchron zur ursprünglichen HTTP Anfrage an Pimcore. Wie in Abb. 6.1 ersichtlich, liegt die maximale durchschnittliche zusätzliche Verarbeitungszeit, welche durch gesendete Web-Realtime Events entsteht für Objekt Operationen bei durchschnittlich etwa sechs Millisekunden. Der Mehrwert[1] liegt somit zwischen etwa 13 Prozent für `Update`-Operationen und 20 Prozent für `Create`- und `Delete`-Operationen, was für die Anwendung innerhalb des CMS und für Standard Web-Anwendungen durchaus tolerierbar ist. Vor allem deshalb, da der absolute Wert der gesamten Anfragedauer für Operationen innerhalb des CMS (z. B. Speichern eines Objekts oder Dokuments) wesentlich höher ist, im Vergleich zu den genannten Werten der Objekt-Operationen an

[1] Overhead

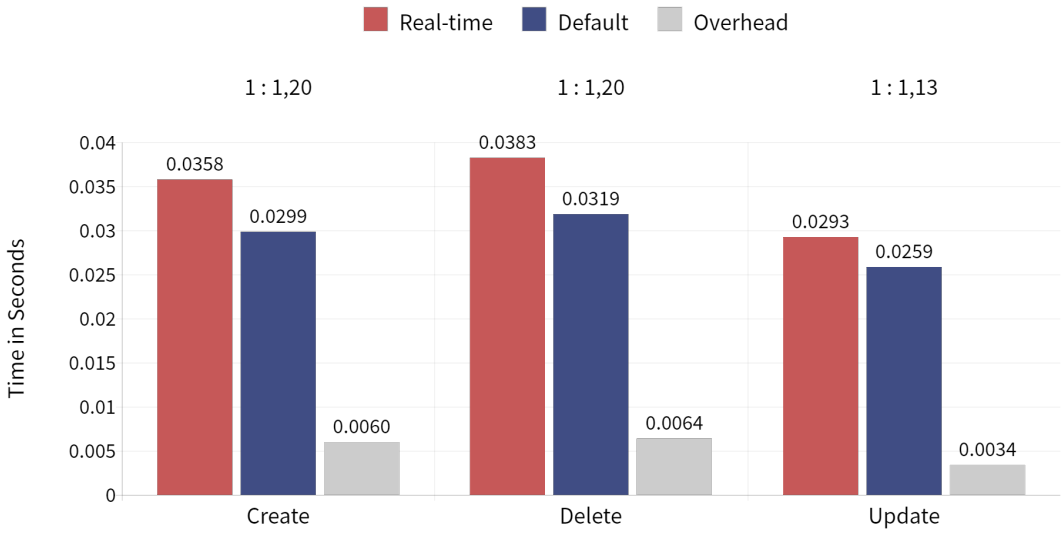


Abbildung 6.1: Analyse der Verarbeitungszeit der Anfragen für Operationen zu Objektinstanzen in Sekunden. Die Werte der vertikalen Achse entsprechen der Zeitspanne in Sekunden und resultieren aus einem Durchschnittswert von jeweils 2000 ausgeführten `Create`-, `Update`- und `Delete`-Operationen.

sich. Eine Anfrage im selben Server-Setup, welche zum Beispiel genaue eine `Create`-, `Update`- und `Delete`-Operation ausführt, dauert im Durchschnitt etwa 0,2 Sekunden. Daraus wird ersichtlich, dass der gemessene maximale durchschnittliche *Overhead* von etwa 0,0064 Sekunden erst wirklich relevant wird, sobald diverse Operationen in einer Anfrage ausgeführt werden (siehe Abb. 6.2).

Testumgebung: Als Testumgebung wurde ein virtueller Server mit folgenden Spezifikationen verwendet: *VPS Debian GNU/Linux 8 (jessie), 1GB RAM, 2CPUs 2.2 GHz.* Alle Grafiken zur Evaluierung der Verarbeitungszeit stammen aus einer implementierten Analyse-Schnittstelle[2], welche im Web-Realtime Plugin enthalten ist. Somit kann die exakte Leistung unter verschiedenen Umgebungen und Nutzungsprofilen in jedem System neu festgestellt werden. Letztendlich ist allerdings vor allem das Verhältnis zwischen der Dauer der Anfragen bzw. Operationen mit gesendeten Web-Realtime Events zu Anfragen ohne dieser von Relevanz. Deshalb und da die Eigenschaften der verwendeten Testumgebung als typisch für eine Web Applikation mit moderater Nutzung bezeichnet werden können, werden die Ergebnisse der gemessenen Analysen als Evaluierungsresultat angenommen.

6.1.2 Verbindungsautonomie

Die Autonomie der Erweiterung gegenüber der Verbindungsvariante wurde in so vielen Aspekten wie möglich berücksichtigt, wobei auch hier noch Verbesserungspotential besteht. Für die Verbindung zwischen dem Web-Realtime Dienst und dem Apache Server

[2] Profiling-Interface

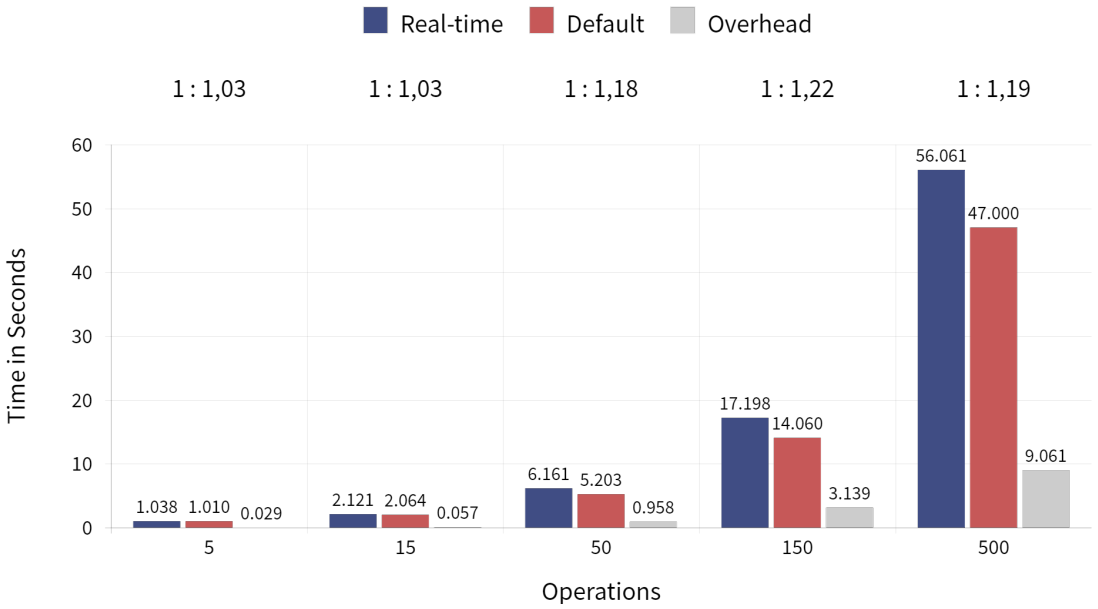


Abbildung 6.2: Verarbeitungszeit einer HTTP-Anfrage bei verschiedener Anzahl an durchgeführten Objekt-Operationen. Die zusätzliche Verarbeitungszeit durch das Senden von Ereignissen bleibt konstant bei mehreren ausgeführten Operationen. Die auf der vertikalen Achse dargestellte Zeit in Sekunden wurde gemessen für eine HTTP Anfrage, in welcher die auf der horizontalen Achse angegebene Anzahl an Operationen ausgeführt wurden. Die angegebenen Werte resultieren aus einem Durchschnitt aus jeweils 100 ausgeführten Anfragen.

wurden zwei Lösungen für TCP und HTTP implementiert. Weitere Verbindungsdienste wie z. B. *Redis Pub/Sub*[3] hinzuzufügen wäre mit geringem Aufwand möglich. Wie in Abb. 6.3 erkenntlich, ist die Dauer der Anfragen zum Senden eines Events von Apache zum *Node.js* Web-Server innerhalb der selben Unix-Serverinstanz nur minimal unterschiedlich. Somit ist die Wahl des Verbindungsdienstes leistungstechnisch nur von geringer Relevanz und die Vorteile des HTTP-Routings seitens des verwendeten Frameworks *Sails.js* des Web-Realtime Dienstes überwiegen. Da *Sails.js* für HTTP und WebSocket Verbindungen optimiert ist, ist HTTP somit die vorrangig verwendete Server-interne Verbindungsvariante.

Des Weiteren wurden Vorbereitungen getroffen, nicht nur die Verbindungsvariante, sondern auch den Web-Realtime Dienst selbst frei wählen zu können. Allerdings trifft dies noch nicht für alle Teile der Implementierung zu, da wie im Abschnitt 6.1.1 bereits erwähnt, spezifische Logiken zur Leistungsoptimierung in den Web-Realtime Dienst ausgelagert wurden. Dies macht die Verwendung eines gehosteten Web-Realtime Dienstes (siehe Abschnitt 3.2) in Kooperation mit der bereitgestellten API nur zum Teil möglich. Vor allem Objektlisten-Kanäle müssten seitens PHP neu implementiert werden. Der Web-Realtime Dienst dürfte nur noch als Verbindungsdienst dienen und für keine weiteren Aufgaben verantwortlich sein. Konzepte für Pub/Sub Verhalten oder einer einheitli-

[3] https://redis.io/topics/pubsub

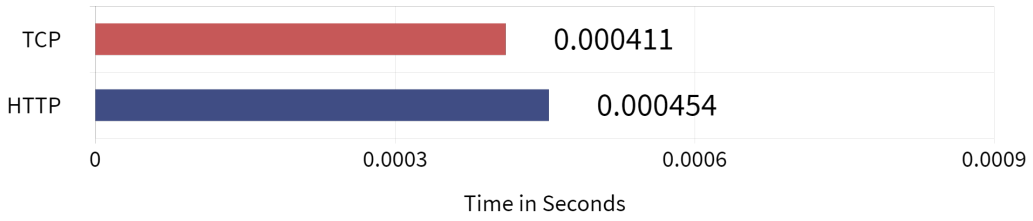


Abbildung 6.3: Durchschnittliche Dauer zum Senden eines Events von Apache zu *Node.js* Server. Die Werte ergeben sich aus einem Durchschnitt aus 2000 gesendeten Nachrichten angegeben in Sekunden.

chen Authentifizierungsschnittstelle sind auch bei gehosteten Web-Realtime Anbietern etablierter Standard und könnten weiterhin übernommen werden. Implementierungen des Web-Realtime Dienstes basierend auf anderen Frameworks (siehe Abschnitt 3.3) wären durchaus, auch mit der derzeitigen Implementierung mit vollständiger Funktionsweise, denkbar.

6.1.3 Anwendbarkeit

Die Anwendbarkeit ist schwer messbar und wäre lediglich einschätzbar am Aufwand der Einrichtung und der subjektiven Einfachheit der Verwendung für Entwickler und Anwender des Systems.

System-Setup

Der erste Schritt der Einrichtung sind die Einbindung und das Setup aller in Abschnitt 5.2 erwähnten Komponenten der Systemarchitektur. Via des Abhängigkeits-Verwaltungs-Systems *Composer* kann die CMS-Erweiterung als Pimcore Plugin installiert werden. Da diese als Abhängigkeiten des Plugins definiert wurden, werden hierbei das implementierte Web-Realtime Package und die ZeroMQ Programmbibliothek für PHP eingebunden. Sobald die benötigten Dateien bereitgestellt sind, ist es möglich das Plugin innerhalb der Administrationsoberfläche des CMS zu installieren und zu konfigurieren (siehe Abb. 6.4 und 6.5). Der Installationsvorgang wurde insofern automatisiert, als dass eine Beispiel-Konfigurationsdatei aus dem `Plugins/Realtime/install` Verzeichnis in das `website/var/config` Verzeichnis kopiert wird und alle weiteren benötigten Datenbanktabellen kreiert werden. Die letzte fehlende Komponente der Systemarchitektur ist der Web-Realtime Service. Dieser ist derzeit noch nicht frei wählbar und muss manuell eingerichtet werden. Dazu ist ein *Node.js* Server Voraussetzung. Die zugehörigen Projekt-Dateien der *Sails.js* Applikation müssen via *git*[4] oder manuell ein-

[4] https://git-scm.com/

Abbildung 6.4: Die Web-Realtime CMS-Erweiterung ist installierbar innerhalb der CMS Administrationsoberfläche.

Abbildung 6.5: Die Konfiguration der CMS-Erweiterung erfolgt innerhalb der CMS Administrationsoberfläche.

gebunden werden. Die benötigten weiteren Abhängigkeiten dieser *Node.js* Applikation können via *NPM* installiert werden. Nach Start beider Web-Server (Apache und *Node.js*) und des ZeroMQ-Skripts zur Verwaltung der Server-internen Socket Verbindungen, stehen alle Web-Realtime Funktionalitäten zur Verfügung.

Für die vollständige Einrichtung des Systems zur Integration der Realtime User Experience in das CMS wird somit Know-How im Bereich der Software-Entwicklung vorausgesetzt – ein Setup allein durch einen CMS-Anwender wird nicht ermöglicht. Der Vorgang selbst ist durch die Abhängigkeits-Verwaltungs-Systeme *NPM* für *Node.js* und *Composer* für PHP weitestgehend automatisiert und für einen Entwickler mit wenigen Arbeitsschritten auszuführen.

Verwendung

Die Verwendung der neuen CMS-Funktionen und der PHP-API, welche nach erfolgreicher Einrichtung der Erweiterung bereitgestellt werden, wurde in Kapitel 5 im Detail ausgeführt. Um deren Verwendung für den CMS-Entwickler so einfach wie möglich zu gestalten und eine steile Lernkurve zu ermöglichen, wurden die im Hintergrund ablaufenden Prozesse auf Seiten des *Node.js* Servers und des Clients so weit abstrahiert und automatisiert, dass der CMS-Entwickler damit nicht mehr in Kontakt kommen muss. Die wichtigsten hierfür implementierten Konzepte sind:

- Anknüpfung der Web-Realtime Kanäle an bestehende CMS-interne Pimcore Komponenten wie Daten-Objekte und Dokumente,
- Konfigurationsmöglichkeiten innerhalb der CMS Administrationsoberfläche,
- Standard-Implementierungen für alle wichtigen Publish-Subscribe Komponenten wie Kanäle, Ereignis-Objekte und Verbindungsdienste,
- Automatische Code Generierung auf Seiten des *Node.js* Servers,
- Automatisierte Kanal-Abonnements via Client-seitiger Code Generierung,
- Automatisierte Kanäle für Daten-Objekte innerhalb des CM-Systems.

Für den CMS-Editor werden die in Abschnitt 5.4 beschriebenen neue Funktionen innerhalb der Administrationsoberfläche zur Verfügung gestellt. Diese können ohne jegliches technische Know-How verwendet werden, sind im Design-Schema der Pimcore Oberfläche implementiert und kompatibel mit bestehenden Pimcore Elementen. Sowohl für CMS-Entwickler als auch für CMS-Anwender sollte somit eine einfache Verwendung der gebotenen Funktionen ohne größeren Aufwand möglich sein.

6.1.4 Verlässlichkeit der Verbindung

Wie in Abschnitt 5.1.2 erklärt, ist der zu bevorzugende Transportdienst zwischen Client und Server eine WebSocket-Verbindung. Die Zuverlässigkeit dieser Verbindung wird in der aktuellen Implementierung durch die JavaScript-Bibliothek *Sails.io.js*, welches auf *Socket.io* (siehe Abschnitt 3.3.2) basiert, gewährleistet. Denn auch wenn sich in den letztens Jahren die WebSocket-Unterstzüzung stetig verbessert hat, kann die Verbindung gestört werden. *Socket.io* bietet durch das Protokoll und der zugehörigen JavaScript-API die Garantie, dass auch in solchen Fällen *Realtime UX* etabliert werden kann. Dies wurde unter Verwendung von Firefox 4.0 unter Windows 7, wie in Abb. 6.6 ersichtlich, getestet. Des Weiteren gibt es die Möglichkeit der horizontalen Skalierung zu multiplen Server Instanzen via Auslagerung der Socket-Ids in einen externen Speicher, wie z. B. einer Redis Datenbank. Eine zuverlässige Verbindung bedeutet, neben der Verlässlichkeit des Transportdienstes an sich, auch die Korrektheit der Kanal-Abonnements nach einem Verbindungsverlust. Wie in Abschnitt 5.3.5 beschrieben, wird hierfür der Reconnect-Callback der *Sails.io.js* Programmbibliothek genutzt. Sollte es zukünftig ermöglicht werden, den Dienst zu wechseln und somit auch eine andere Client-seitige API als *Sails.io.js* zu verwenden, müssten diese Aspekte neu berücksichtigt werden.

Es kann demnach eine zuverlässige Verbindung, welche unabhängig vom verwendeten User Agent funktioniert, garantiert werden. Auch wird es ermöglicht, Abonnements dynamisch zu ändern und trotzdem bei möglichen Verbindungsverlusten diese immer konsistent zu halten. Außerdem kann realisiert werden, dass die Verbindungen, auch bei Skalierung der Applikation über mehrerer Server hinweg, konsistent bleiben.

6.1.5 Flexibilität der Implementierung

Für die in Abschnitt 4.1 erwähnten dynamischen Komponenten des Systems wurden Vorbereitungen getroffen:

- Die Server-Server Kommunikation zwischen Apache und *Node.js* ist auf Kanal-Ebene konfigurierbar. Derzeit stehen TCP und HTTP als Verbindungsdienste zur Verfügung. Die Implementierung ist darauf angepasst, es zu ermöglichen weitere Dienste hinzufügen zu können.
- Wie Kanal-Abonnements erfolgen ist von der API des Web-Realtime Dienstes abhängig. Auch wenn derzeit nur *Sails.js* unterstützt wird, ist es bereits möglich, den generierten Code anzupassen, sodass auch JavaScript Schnittstellen anderer Systeme unterstützt werden könnten, wie in Abschnitt 5.3.5 genauer erläutert.
- In der derzeitigen Implementierung wird ein Wechsel des Web-Realtime Dienstes nicht unterstützt. Das angewendete Kommunikationskonzept der Pub/Sub-Kanäle ist allerdings universell einsetzbar.

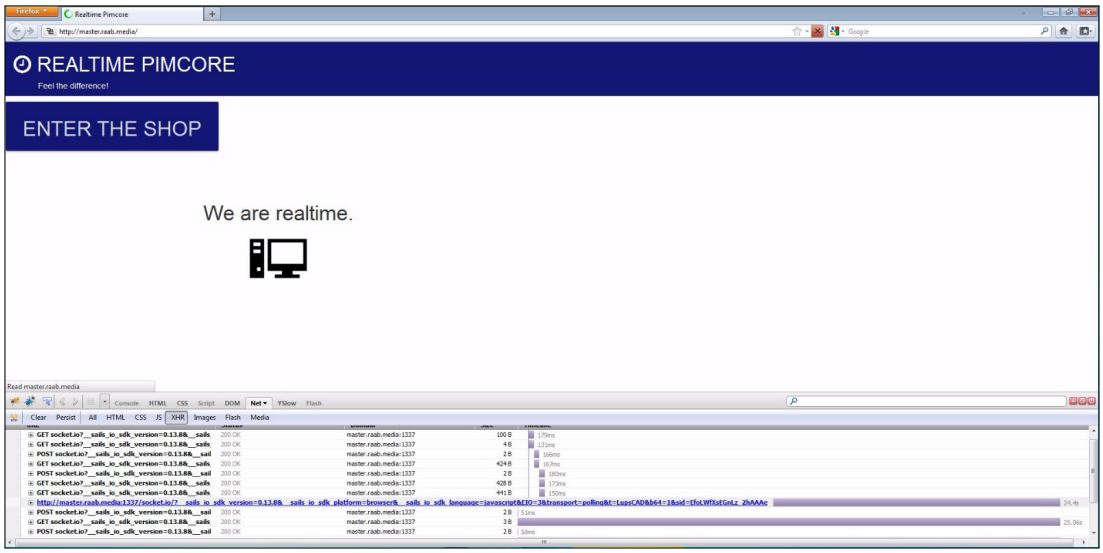


Abbildung 6.6: *Socket.io* ermöglicht trotz eines Web Clients welcher WebSocket-Verbindungen nicht unterstützt, durch einen Rückfall auf Long-Polling die gewünschte Realtime User Experience. In der Testumgebung diente Mozilla Firefox 4.0 als User Agent und Windows 7 als Betriebssystem.

Egal welche variablen Komponenten im Hintergrund in Verwendung sind, kann der Zugriff auf die Kanäle, wie in Abschnitt 4.3 erklärt, immer einheitlich über das Radio/-Kanal Konzept erfolgen.

6.1.6 Erweiterbarkeit

Die Erweiterbarkeit ist eine essentielle Anforderung an das System, da es wie in Abschnitt 3.1 beschrieben diverse Anwendungsfälle für Web-Realtime gibt und durch die rapide Entwicklung des World Wide Webs immer wieder neue entstehen können. Das als Kanal-System angewendete Publish-Subscribe Prinzip ist flexibel anwendbar und die Implementierung bietet viele mögliche Erweiterungen und Anwendungen. Im Abschnitt 5.4 werden die bereits implementierten Anwendungsszenarien, welche als Proof-Of-Concept[5] durchgeführt wurden, beschrieben.

6.2 Verbesserungsmöglichkeiten

In diesem Abschnitt werden mögliche Verbesserungen und Erweiterungen zum vorgestellten Konzept dargelegt werden.

6.2.1 Abstraktion des Web-Realtime Dienstes

Wie in Abschnitt 6.1 bereits erwähnt, ist die Implementierung derzeit nicht darauf ausgelegt, verschiedene Web-Realtime Dienste zu unterstützen, sondern auf die Verwendung von *Sails.js* unter *Node.js* optimiert. Ein möglicher Wechsel, vor allem zu

[5] Nachweis der Machbarkeit

gehosteten Web-Realtime Diensten wie etwa *Pusher, Firebase,* oder *PubNub* wäre allerdings für die Anwendung in einem Open-Source CMS wie Pimcore, welches diverse Server-Umgebungen unterstützen soll, durchaus wünschenswert. Um auch extern gehostete Web-Realtime Dienste unterstützen zu können, welche es im Allgemeinen nicht ermöglichen benutzerdefinierte Logiken auszuführen, müsste somit jegliche Logik, die derzeit in den *Node.js* Server ausgelagert wurde, davon entkoppelt werden, bzw. darauf verzichtet werden. Der Web-Realtime Dienst sollte nur für das Weiterleiten der Nachrichten verantwortlich sein. Da, neben den Überprüfungen der Bedingungen für Objektlisten-Kanäle, die Leistungsoptimierung im Allgemeinen lediglich darauf beruht, das Senden der Ereignisse asynchron zur initialen Anfrage durchzuführen, wäre es hier durchaus eine Möglichkeit, dies durch eine PHP-basierte Programm-Komponente zu realisieren. Beispiele von Systemen, welche für die Umsetzung verwendet werden könnten wären *Gearman*[6]*, ActiveMQ*[7]*, ZeroMQ*[8], oder einfache asynchron gestartet PHP-Programmskripte.

Authentifizierung Die Konfiguration der Authentifizierung der Kanäle sollte über eine globale Konfigurationsmöglichkeit innerhalb des CMS erfolgen können. Wie diese vom verwendeten Web-Realtime Dienst interpretiert wird, wäre weiterhin API-abhängig, im verwendeten Beispiel *Sails.js* müsste diese Konfiguration, wie in Abschnitt 5.3.6 beschrieben, in die `/config/policies.js` Datei übersetzt werden. Das Konzept der Authentifizierung der Kanäle über eine externe Schnittstelle ist angelehnt an das der Private-Channels von *Pusher* (siehe Abschnitt 3.2.1), so könnte auch für diesen Dienst die Konfiguration einfach angewandt werden.

6.2.2 Abstraktion des CM-Systems

Die Implementierung ist, wie in Abschnitt 5.2 erklärt, in vier Programmcode-Archive aufgeteilt. Wobei sich nur in der Komponente des *Web-Realtime Packages* nicht Systemspezifische Logiken befinden, die auch in anderen CM-Systemen angewendet werden könnten. Der Grund für diese Aufteilung ist die Abgrenzung der Logik der Schnittstelle für das Publizieren und Abonnieren (*Publish-Subscribe*) von CMS-spezifischen Implementierungen. Ein Beispiel hierfür wäre die Ereignisverwaltung innerhalb von CM-Systemen. So wäre das Pendant zum Ereignis-Management von Pimcore 4 im CMS *Drupal* 8 dessen sogenannte *Rules* [31]. Auch das auf Blog-Systeme spezialisierte CMS Wordpress bietet Einschubmethoden an vordefinierten Stellen wie z. B. nach dem Speichern eines Beitrag-Datenobjekts[9] [25]. Für die Implementierung der Ereignisverwaltung der Web-Realtime Events für automatisierten Kanäle analog zu Prog. 5.2, müsste somit eine neue CMS-spezifische Umsetzung bereitgestellt werden, im Beispiel Drupal könnte dies, wie in Prog. 6.1 ersichtlich, erfolgen. Außerdem könnte hier in Zukunft angedacht werden ein einheitliches CMS-unabhängiges PHP-Interface für CRUD-Operationen für Datenobjekte bereitzustellen, welches CMS-spezifische Logiken kapselt und eine einheitliche Schnittstelle zum Web-Realtime Dienst bereitstellt. Dieses Interface könnte für alle

[6] http://gearman.org/
[7] http://activemq.apache.org/
[8] http://zeromq.org/
[9] Post

Programm 6.1: Pseudo-Programmcode für eine mögliche Implementierung von automatisierten Kanäle für Drupal *Entities* unter Verwendung des implementierten Web-Realtime Packages.

```
1 function hook_entity_insert(Drupal\Core\Entity\EntityInterface $entity) {
2
3     if ($entity->supportsRealtime() && $entity->isPublished()) {
4         $channel = \Realtime\Radio::getEntityChannel($entity);
5         $channel->publish(new \RT\Event\Create($entity));
6     }
7
8 }
```

verallgemeinbaren Ereignisse innerhalb eines CMS angedacht und angewendet werden, wie z. B. Systemstart, Erstellung neuer Dokumente, Hinzufügen von Datei-Assets oder beim Login eines Benutzers usw.

6.2.3 Client-Server Verbindung

Im Zusammenhang der bidirektionalen Client-Server Verbindung wird oft HTTP/2 als mögliche Variante der Verbindungsrealisierung genannt, vor allem durch den *Server-Push* Aspekt der Spezifikation [3], welcher es erlaubt, Ressourcen ohne explizite Anfrage vom Server zum Client zu senden, und auch aufgrund der spezifizierten persistenten Verbindung zwischen Server und Client [3, S. 59-65]. HTTP/2 ist allerdings nicht als ein Kommunikationsprotokoll definiert, sondern dient als ein Transportprotokoll. Auch wenn es also möglich ist aktiv Daten Server-seitig zum Client zu senden, ist es nicht möglich diese innerhalb der Applikation selbst in Empfang zu nehmen, lediglich der Browser interpretiert die empfangenen Ressourcen. Eine Client-seitige API gibt es hierfür nicht. Was HTTP/2, neben vieler anderer bedeutsamer Verbesserungen in der Internetkommunikation, gegenüber HTTP/1.1 auch im Bereich der bidirektionalen Client-Server Kommunikation interessant macht, ist der Aspekt des *Multiplexing* der HTTP-Streams (siehe Abb. 6.7). Hierdurch wird es möglich, über eine einzige TCP Verbindung mehrere *Streams* zu übertragen, was Server-Sent-Events in Zukunft wieder neu Relevanz einräumen könnte. SSE sind, wie in Abschnitt 2.2.3 beschrieben, basierend auf HTTP-Streaming und somit auch kompatibel mit HTTP/2. Es würde somit möglich werden, ebenso via HTTP eine bidirektionale Verbindung herzustellen und dabei auch alle Vorteile von HTTP/2 nutzen zu können. Es besteht die Möglichkeit, dass diese Lösung in der Zukunft an Bedeutung gewinnen wird, derzeit scheitert es hierbei, sowohl für SSE [22] als auch für HTTP/2 [21], noch an der mangelnden Browser-Unterstützung [27].

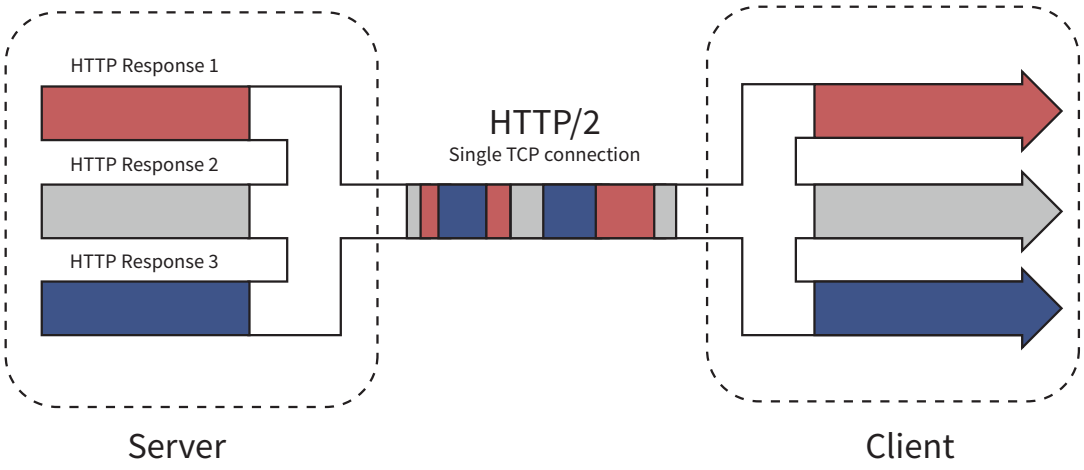


Abbildung 6.7: Mehrere HTTP-Streams können bei HTTP/2 per Stream-Multiplexing in einer TCP Verbindung übertragen werden [47].

Kapitel 7

Zusammenfassung und Fazit

Durch die Integration von Web-Realtime in CMS-basierte Web Applikationen ergibt sich eine ganze Palette an neuen Möglichkeiten, sowohl innerhalb der Administrationsoberfläche des CM-Systems, als auch in der jeweiligen Endbenutzer-Anwendung. Die Vorteile, die dadurch entstehen, wie die Ermöglichung der Berücksichtigung kollaborativer Aspekte oder Echtzeit-Datensynchronisation mit verbundenen Clients, überwiegen bereits erwähnte Nachteile, wie längere Verarbeitungszeiten der CMS-Operationen und eine mögliche komplexere Server-Architektur, bei Weitem. Die technologischen Voraussetzungen für eine bidirektionale Web-Realtime Verbindung zwischen Client und Server, können seit der Einführung von HTML5 Standard-konform erfüllt werden. Sind die Auslöser der Web-Realtime Events innerhalb einer Applikation erst identifiziert, sollte somit der Etablierung der gewünschten *Realtime User Experience* nichts mehr im Weg stehen. Diese Events können sehr spezifisch abhängig vom konkreten Anwendungsfall auftreten. Für die meisten CM-Systeme, wie auch für das als Fallbeispiel verwendete System Pimcore, bestehen meist allerdings auch schon interne Event-Schnittstellen, welche sich zur simplen Übersetzung in Web-Realtime Events anbieten – z. B. zur Synchronisation von Datenobjekten innerhalb des CM-Systems zum Client. Bestehende Software-Paradigmen wie *Publich-Subscribe* forcieren Abonnements zu kategorisierten Daten-Teilmengen und bieten sich somit zur Realisierung dieser benötigten Event-Kanäle an. Es liegt nun in der Verantwortung der Web-Entwickler, sich mit neuen Technologien, wie WebSockets und Server-Sent-Events, auseinanderzusetzen und sie auch anzuwenden.

Web-Realtime ist heute ein integraler Bestandteil vieler Web Applikationen und sollte meiner Meinung nach auch für CM-Systeme eine obligatorische Voraussetzung sein. Dies betrifft sowohl die System-Schnittstellen für jeweilige Endanwendungen, als auch die Benutzeroberfläche und die Benutzererfahrung bei der redaktionellen Verwendung des CM-Systems selbst. Durch die als Fallstudie implementierte CMS-Erweiterung wird beides ermöglicht und wenn auch die konkrete Implementierung für eine generalisierte Anwendung noch Verbesserungsmöglichkeiten offen lässt, werden für eine weitere CMS-Integration alle Bausteine zur Verfügung gestellt.

Anhang A

Inhalt der CD-ROM/DVD

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF-Dateien

Pfad: /

Raab_Julian_2017.pdf Masterarbeit mit Instruktionen (Gesamtdokument)

A.2 Bild-Dateien

Pfad: /online-references

caniuse_http2.png . . . [21]
caniuse_sse.png . . . . [22]
caniuse_ws.png . . . . [23]
realtime_stack.png . . . [24]
wp_save_post.png . . . [25]
cscw.png . . . . . [26]
websocket_http2.png . [27]
pc_extending.png . . . [28]
pc_o_list.png . . . . . [29]
pc_editables.png . . . . [30]
drupal_rule_et.png . . [31]
fb_webhooks.png . . . [32]
built_firebase.png . . . [33]
built_pubnub.png . . . [33]
built_pusher.png . . . . [33]
built_socketio.png . . . [33]
built_sockjs.png . . . . [33]
firebase.png . . . . . [34]

peer-to-peer.png . . . . [35]
trends_cms.png . . . . [36]
html5.png . . . . . [37]
bug_ff_sse.png . . . . . [38]
rtc_status_pages.png . [39]
bug_ch_sse.png . . . . [40]
simultaneous_edit.png . [41]
ratchet.png . . . . . [42]
ratchet_push.png . . . [43]
sails_pubsub.png . . . . [44]
sails_generator.png . . [45]
npm-socketio.png . . . [46]
http2_module.png . . . [47]
twitter_api.png . . . . . [48]
pusher.png . . . . . [49]
autobahnjs.png . . . . . [50]
real-time.png . . . . . [51]
realtime_data.png . . . [52]

Pfad: /images

*.eps . . . . . Original Adobe Illustrator-Dateien
*.png . . . . . Original Rasterbilder

A.3 Programmcode-Dateien

Pfad: /source

pubsub.zip . . . . . PHP Composer PubSub Package, siehe Abschnitt 5.2.1
cms_plugin.zip . . . . . CMS Web Real-time Erweiterung, siehe Abschnitt 5.2.2
cms.zip . . . . . CMS Beispiel-Applikation, siehe Abschnitt 5.2.3
realtime_service.zip . . Web Real-time Dienst, siehe Abschnitt 5.2.4
db.sql . . . . . Datenbankstruktur und Inhalte der Beispielapplikation

Quellenverzeichnis

Literatur

[1] Dietmar Abts. *Bidirektionale Kommunikation mit WebSocket*. Wiesbaden: Springer Fachmedien, 2015 (siehe S. 8).

[2] Deane Barker. *Web Content Management*. Sebastopol: O'Reilly Media, Inc., 2016 (siehe S. 5).

[3] M. Belshe. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. RFC Editor, Mai 2015. URL: https://tools.ietf.org/html/rfc7540#section-8.2 (besucht am 04. 08. 2017) (siehe S. 57).

[4] J. Böhringer, P. Bühler und D. Sinner P. Schlaich. *Kompendium der Mediengestaltung*. 6. Aufl. Springer Vieweg, 2014 (siehe S. 6).

[5] Deborah Dahl. *Multimodal Interaction with W3C Standards*. Springer International Publishing, 2017 (siehe S. 10, 11).

[6] I. Fette und A. Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, Dez. 2011. URL: http://www.rfc-editor.org/rfc/rfc6455.txt (siehe S. 7, 8, 34).

[7] Roy T. Fielding u. a. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. RFC Editor, Juni 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt (siehe S. 7).

[8] Carl A. Gutwin, Michael Lippold und T. C. Nicholas Graham. „Real-time Groupware in the Browser: Testing the Performance of Web-based Networking". In: *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*. CSCW '11. Hangzhou, China: ACM, 2011, S. 167–176 (siehe S. 34).

[9] Shane Hudson. *JavaScript Creativity. Exploring the Modern Capabilities of JavaScript and HTML5*. New York: Apress, 2014 (siehe S. 10).

[10] Colin J. Ihrig. *Pro Node.js for Developers*. New York: Apress, 2013 (siehe S. 19).

[11] Jason Lengstorf und Phil Leggetter. *Realtime Web Apps*. New York: Apress, 2013 (siehe S. 5).

[12] Azat Mardan. *Pro Express.js. Master Express.js for your development*. Berlin Heidelberg: Springer, 2014 (siehe S. 19).

[13] Christoph Meinel und Harald Sack. *Internetworking*. Berlin Heidelberg: Springer, 2012 (siehe S. 4, 7).

[14] Den Odell. *Pro JavaScript Development. Coding, Capabilities, and Tooling*. New York: Apress, 2014 (siehe S. 11).

[15] Jose Palala und Martin Helmich. *PHP 7 Programming Blueprints*. Birmingham: Packt Publishing, 2016 (siehe S. 27).

[16] Rohit Rai. *Socket.IO Real-time Web Application Development*. Birmingham: Packt Publishing, 2013 (siehe S. 14, 19).

[17] J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols*. RFC 5245. RFC Editor, Apr. 2010. URL: http://www.rfc-editor.org/rfc/rfc5245.txt (siehe S. 11).

[18] H. Schulzrinne u. a. *RTP: A Transport Protocol for Real-Time Applications*. STD 64. RFC Editor, Juli 2003. URL: http://www.rfc-editor.org/rfc/rfc3550.txt (siehe S. 11).

[19] K. G. Shin und P. Ramanathan. „Real-time computing: a new discipline of computer science and engineering“. *Proceedings of the IEEE* 82.1 (Jan. 1994), S. 6–24 (siehe S. 5).

[20] Vanessa Wang, Frank Salim und Peter Moskovits. *The Definitive Guide to HTML5 WebSocket*. New York: Apress, 2013 (siehe S. 8, 9).

Online-Quellen

[21] *Can I use HTTP 2 protocol*. 2017. URL: http://caniuse.com/#feat=http2 (besucht am 04. 08. 2017) (siehe S. 57, 60).

[22] *Can I use Server-sent events*. 2017. URL: http://caniuse.com/%5C#feat=eventsource (besucht am 28. 07. 2017) (siehe S. 34, 57, 60).

[23] *Can I use WebSockets*. 2017. URL: http://caniuse.com/%5C#feat=websockets (besucht am 30. 08. 2017) (siehe S. 18, 60).

[24] *Choosing your Realtime Web App Tech Stack - Phil Leggetter*. Youtube. Jan. 2014. URL: https://www.youtube.com/watch?v=VENVNimklWg (siehe S. 25, 26, 60).

[25] *Code Reference. Save Post*. 2017. URL: https://developer.wordpress.org/reference/hooks/save_post/ (besucht am 22. 09. 2017) (siehe S. 56, 60).

[26] *CSCW*. 2014. URL: https://www.psychologie.uni-freiburg.de/Members/rummel/alt/wisspsychwiki/wissenspsychologie/index.html/CSCW (besucht am 07. 09. 2017) (siehe S. 30, 60).

[27] Allan Denis. *Will WebSocket survive HTTP/2?* 2016. URL: https://www.infoq.com/articles/websocket-and-http2-coexist (besucht am 04. 08. 2017) (siehe S. 57, 60).

[28] *Development Documentation. Extending Pimcore*. 2017. URL: https://www.pimcore.org/docs/4.6.x/Extending_Pimcore/index.html (besucht am 22. 09. 2017) (siehe S. 33, 60).

[29] *Development Documentation. Working with PHP API Object Listings*. 2017. URL: https://www.pimcore.org/docs/4.6.x/Objects/Working_with_PHP_API.html#page_Object_Listings (besucht am 22. 09. 2017) (siehe S. 37, 60).

[30] *Development Documentation. Document Editables*. 2017. URL: https://www.pimcore.org/docs/4.6.x/Documents/Editables/index.html (besucht am 22. 09. 2017) (siehe S. 44, 60).

[31] *Documentation. Rule Event Type*. 2017. URL: https://www.drupal.org/docs/8/modules/business-rules/concepts/rule-event-type (besucht am 22. 09. 2017) (siehe S. 56, 60).

[32] *Facebook Webhooks*. 2017. URL: https://developers.facebook.com/docs/graph-api/webhooks (besucht am 22. 09. 2017) (siehe S. 15, 60).

[33] *Find out what websites are Built With*. URL: https://trends.builtwith.com (besucht am 03. 09. 2017) (siehe S. 14, 60).

[34] *Firebase Realtime Database*. 2017. URL: https://firebase.google.com/products/database/ (besucht am 15. 05. 2017) (siehe S. 17, 60).

[35] Ian Hickson. *WebRTC 1.0: Real-time Communication Between Browsers*. 2017. URL: https://www.w3.org/TR/webrtc/#peer-to-peer-data-api (besucht am 20. 04. 2017) (siehe S. 10, 61).

[36] *Historical yearly trends in the usage of content management systems for websites*. URL: https://w3techs.com/technologies/history_overview/content_management/all/y (besucht am 03. 09. 2017) (siehe S. 1, 2, 61).

[37] *HTML5*. Sep. 2017. URL: https://html.spec.whatwg.org/#comms (besucht am 14. 09. 2017) (siehe S. 7, 19, 61).

[38] *Increase number of permitted EventSource connections*. 2017. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=906896 (besucht am 22. 09. 2017) (siehe S. 34, 61).

[39] C. Jennings, S. Turner und T. Hardie. *Rtcweb Status Pages. Real-Time Communication in WEB-browsers*. 2017. URL: https://tools.ietf.org/wg/rtcweb/charters?item=charter-rtcweb-2017-03-30.txt (besucht am 19. 04. 2017) (siehe S. 10, 61).

[40] *Limit of 6 concurrent EventSource connections is too low*. 2017. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=275955 (besucht am 22. 09. 2017) (siehe S. 34, 61).

[41] Steve Mokris. *Handling Simultaneous Editing of Node Content*. 2011. URL: https://groups.drupal.org/node/50083 (besucht am 25. 04. 2017) (siehe S. 13, 61).

[42] *Ratchet. WebSockets for PHP*. 2017. URL: http://socketo.me/ (besucht am 22. 09. 2017) (siehe S. 18, 61).

[43] *Ratchet Documentation. Push to an Existing Site*. 2017. URL: http://socketo.me/docs/push (besucht am 22. 09. 2017) (siehe S. 18, 61).

[44] *Sails Documentation. Resourceful PubSub*. URL: http://sailsjs.com/documentation/reference/web-sockets/resourceful-pub-sub (besucht am 02. 08. 2017) (siehe S. 41, 61).

[45] *Sails Documentation. Custom Generators*. URL: http://sailsjs.com/documentation/concepts/extending-sails/generators/custom-generators (besucht am 22. 09. 2017) (siehe S. 42, 61).

[46] *Socket.io*. 2017. URL: https://www.npmjs.com/package/socket.io (besucht am 07. 09. 2017) (siehe S. 19, 61).

[47] *The HTTP/2 Module in NGINX*. Jan. 2016. URL: https://www.nginx.com/blog/http2-module-nginx/#multiplexing (besucht am 04. 08. 2017) (siehe S. 58, 61).

[48] *Twitter Streaming APIs*. 2017. URL: https://dev.twitter.com/streaming/overview (besucht am 22. 09. 2017) (siehe S. 15, 61).

[49] *Understanding Pusher*. 2017. URL: https://pusher.com/docs/ (besucht am 28. 04. 2017) (siehe S. 16, 61).

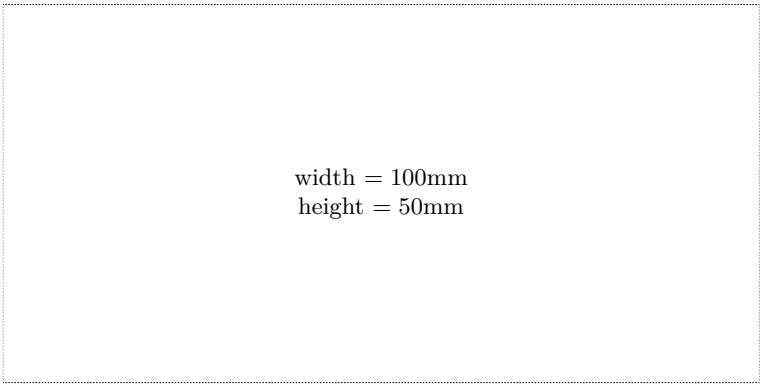
[50] *Use other transports as fallback*. 2017. URL: https://github.com/crossbario/autobahn-js/issues/191 (besucht am 22. 09. 2017) (siehe S. 18, 61).

[51] *What is Real-Time?* Apr. 2017. URL: https://www.leggetter.co.uk/2016/04/22/what-is-realtime.html (besucht am 14. 09. 2017) (siehe S. 5, 61).

[52] *You Have Real-Time Data. You Just Don't Know It!* 2017. URL: https://youtu.be/rk5Jm1lHxlI?list=PLWIRAL-4bCKKcrpxdKgIzz1ixvdCiLDfQ (besucht am 14. 07. 2017) (siehe S. 24, 61).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —