Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Masterarbeit

# Persistenz für Smart Clients in 2-Tier-Architekturen

**Hannes Kuhlmann**

7kuhlman@informatik.uni-hamburg.de
Studiengang Informatik (MSc)
Matr.-Nr. 5944922

Erstgutachter: Professor Dr. -Ing Norbert Ritter
Zweitgutachter: Professor Dr. Winfried Lamersdorf

# Inhaltsverzeichnis

**1 Einleitung** **5**
1.1 Motivation und Zielsetzung . . . . . 8
1.2 Use-Case . . . . . 8
1.3 Aufbau der Arbeit . . . . . 9

**2 Orestes** **11**
2.1 Grundlagen . . . . . 11
2.2 Web-Caching . . . . . 14
2.3 Transaktionen . . . . . 17
2.4 Fazit . . . . . 20

**3 Smart Clients** **21**
3.1 Grundlagen . . . . . 21
3.1.1 Long-Polling . . . . . 22
3.1.2 Server-Sent Events . . . . . 23
3.1.3 WebSockets . . . . . 25
3.1.4 socket.io . . . . . 26
3.1.5 Event-Server . . . . . 27
3.1.6 Optimierung . . . . . 29
3.2 AngularJS . . . . . 32
3.2.1 Grundlagen . . . . . 32
3.2.2 *Hello-World*-Beispiel . . . . . 34
3.3 Backbone.js . . . . . 36
3.3.1 Grundlagen . . . . . 36
3.3.2 Hello-World-Beispiel . . . . . 36
3.4 Ember.js . . . . . 38
3.4.1 Grundlagen . . . . . 38
3.4.2 Hello-World-Anwendung . . . . . 39
3.5 Vergleich der Frameworks . . . . . 41
3.6 Fazit . . . . . 41

**4 Sicherheit** **43**
4.1 Grundlagen . . . . . 43
4.2 Cross-Origin Resource Sharing . . . . . 44

4.3 Grundlagen der Authentifizierung . . . . . 47
4.3.1 Basic-Access-Authentication . . . . . 47
4.3.2 Cookie-Authentication . . . . . 49
4.3.3 OpenID . . . . . 53
4.4 Authentifizierung im Kontext von Orestes . . . . . 56
4.5 Autorisierung . . . . . 61
4.6 Fazit . . . . . 63

**5 Suchmaschinenoptimierung 65**
5.1 Grundlagen . . . . . 65
5.1.1 URL . . . . . 65
5.1.2 Node.js . . . . . 68
5.1.3 PhantomJS . . . . . 69
5.2 Seoxy . . . . . 71
5.2.1 Grundlagen . . . . . 71
5.2.2 Implementierung . . . . . 75
5.2.3 Optimierung . . . . . 82
5.2.4 Evaluation . . . . . 85
5.3 Rendr . . . . . 87
5.4 Fazit . . . . . 88

**6 Prototyp 91**
6.1 Funktionen . . . . . 91
6.2 Implementierung . . . . . 92
6.3 CouchDB-Konfiguration . . . . . 93
6.4 Prototyp aufbauend auf Orestes . . . . . 95

**7 Fazit und Ausblick 97**

**Literaturverzeichnis 103**

**Eidesstattliche Erklärung 109**

# Abstract

Durch aktuelle Trends wie Software-as-a-Serivce, Cloud-Computing und NoSQL sowie einer gestiegenen Browserperformance entstehen neue Chancen sowie Herausforderungen für die Entwicklung von Webanwendungen.

Datenbanken, die eine REST-API zur Verfügung stellen, erlauben die Entwicklung von Webanwendungen als Smart Clients aufbauend auf einer 2-Tier-Architektur. Dabei stellen sich im Besonderen Fragen zu den Themen Sicherheit, Suchmaschinenoptimierung und Clientarchitektur. Diese Masterarbeit diskutiert diese Fragen und stellt Lösungen vor.

Des Weiteren wird auf ein Protokoll mit dem Namen Orestes eingegangen. Dieses hat es sich zur Aufgabe gemacht, die Entwicklung von Smart Clients zu vereinfachen und sowohl die Latenz als auch die Auslieferungszeit von Daten zu verringern.

Neben der Betrachtung der genannten Themen wird eine prototypische Implementierung auf einer NoSQL-Datenbank sowie auf dem Orestes-Protokoll gezeigt.

# Abstract

Due to recent trends like Software-as-a-Serivce, Cloud-Computing, NoSQL and a rising browser performance, there are a lot of new chances and challenges for the development of web applications.

Databases, which provide a REST-API, enable the development of web applications as Smart Clients, based on a 2-Tier-Architecture. This throws up many questions especially regarding safety, search engine optimization and client architecture. The following master's thesis discusses about these questions and offers some answers.

Furthermore it introduces a protocol, named Orestes. Its functions are the simplification of development of Smart Clients and the reduction of the latency. Besides these above mentioned subjects, the master's thesis contains two prototypical implementations of Smart Clients. The first one is based on a NoSQL-Database and the last one is based on Orestes.

# 1 Einleitung

Das Internet und im Besonderen das *World Wide Web* ist heutzutage omnipräsent und aus unserer Gesellschaft nicht mehr wegzudenken. Die Scriptsprache des Webs ist *JavaScript*. Andere Ansätze wie zum Beispiel *Java-Applets* haben sich nicht durchgesetzt. Auf diese wird daher in dieser Arbeit nicht eingegangen.

JavaScript wurde bereits 1995 von *Brenden Eich* entwickelt. Der Sprachkern ist unter der Bezeichnung *ECMAScript* standardisiert [40]. Er befindet sich zurzeit in der Version 5, wobei die 6. Version bereits entwickelt wird und Browser wie *Chrome* und *Firefox* diese schon teilweise unterstützen [1].

Zwischen den unterschiedlichen Browsern herrscht ein starker Konkurrenzkampf um die schnellste JavaScript-Engine. Die verbreitetsten sind *V8* (Chrome), *JägerMonkey* (Firefox) und *Chakra* (Internet Explorer) [77]. Durch diesen Wettkampf wurde die Geschwindigkeit der Engines in den letzten Jahren erheblich gesteigert [78][79][47]. Nicht nur auf der Seite des Clients befindet sich JavaScript im Aufschwung. *Node.js* implementiert die V8-Engine und dient zur Erstellung von Serveranwendungen. Es wird im Besonderen für Webserver verwendet [63]. Im weiteren Verlauf dieser Arbeit wird auf Node.js noch tiefergehend eingegangen.

Klassische Webseiten werden als *Thin Client* aufbauend auf einer 3-Tier-Architektur realisiert. Diese beinhaltet eine Präsentationsschicht (Client/Browser), eine Logikschicht (Applikationsserver) und eine Persistenzschicht (Datenbankserver) [53]. Abbildung 1.1 zeigt seine solche Architektur. Der Client stellt eine Anfrage an den Applikationsserver, welcher anhand seiner programmierten Logik und den Daten, die er aus dem Datenbankserver lädt, die gewünschte Seite und liefert diese dem Thin Client, welcher sie nur noch darstellen muss. Ein Rechenaufwand ist somit hauptsächlich auf dem Applikationsserver vorhanden. Über den Fileserver werden statische Dateien wie zum Beispiel Bilder oder Dokumente ausgeliefert. Diese können auch von dem Applikationsserver bereitgestellt werden, in der Regel wird dafür aber ein extra Fileserver verwendet, um die Last des Applikationsserver zu verringern.

Durch aktuelle JavaScript-Engines und deren Performanzsteigerung bietet sich die Entwicklung von *Smart Clients* oder auch *Fat Clients* genannt an. Im weiteren Verlauf dieser Arbeit wird der Begriff Smart Client verwendet. Bei dieser Variante wird die Logik, welche in einer 3-Tier-Architektur auf dem Applikationsserver ausgeführt wird, in den Client und somit in den Browser verlegt.

Eine 2-Tier-Architektur bietet daher erhebliches Potential zur Kosteneinsparung, da keine Applikationsserver mehr benötigt werden. Ein weiterer Vorteil, welcher durch das

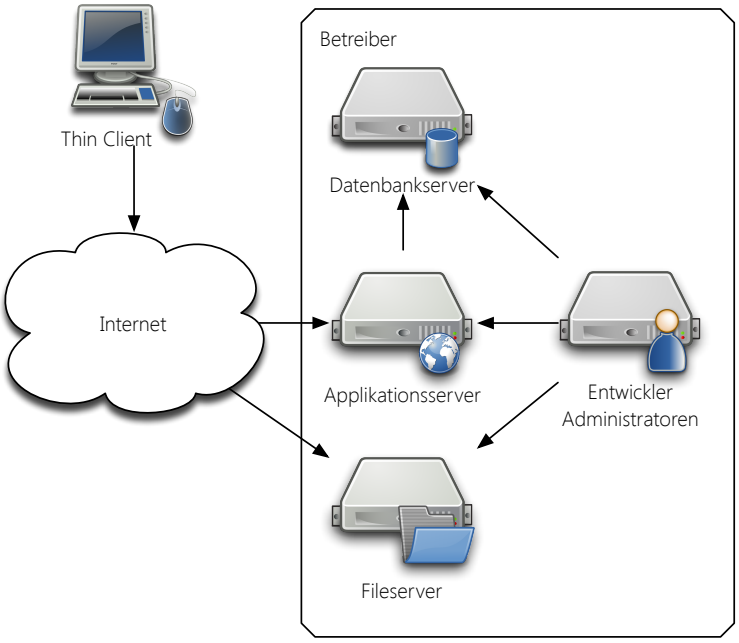


Abbildung 1.1: Aufbau einer 3-Tier-Architektur

*Orestes*-Protokoll, das ebenfalls in dieser Arbeit behandelt wird, noch gesteigert werden kann, ist die Reduzierung der Reaktionszeit der Anwendung. Die Antwortzeiten spielen für den Benutzer eine entscheidende Rolle. *Amazon* hat gezeigt, dass eine Erhöhung der Reaktionszeit um 100*ms* einen Umsatzverlust von 1% bedeutet [41].

Abbildung 1.2 zeigt den groben Aufbau einer 2-Tier-Architektur. Der Browser greift in diesem Fall direkt auf die Datenbank zu. Die Logik wird in den Browser verlagert, der sich somit als Smart Client verhält. Er verwendet die Daten der zugrundeliegenden Datenbank und stellt diese entsprechend seiner Logik dar. Der Fileserver übernimmt auch in diesem Fall die Auslieferung von statischen Daten wie Bilder und Dokumente. In einer 2-Tier-Architektur muss aber auch die Applikation selber an den Browser ausgeliefert werden. Dieses gehört ebenfalls zu den Aufgaben des Fileservers in einer 2-Tier-Architektur.

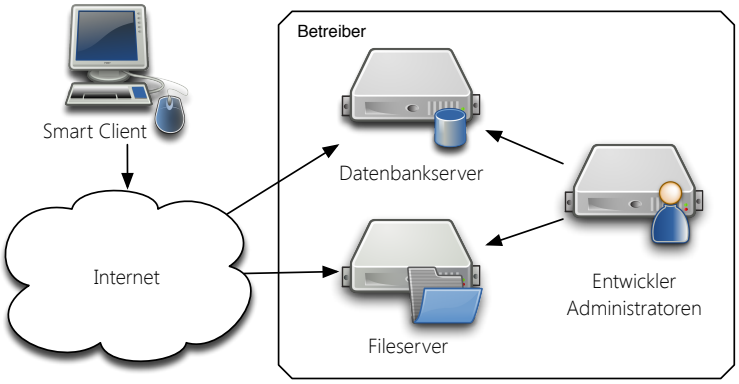


Abbildung 1.2: Aufbau einer 2-Tier-Architektur

Das Betreiben einer IT-Infrastruktur zur Bereitstellung einer stark frequentierten Web-

seite verursacht einen enormen Kostenapparat. Für eine internationale Webseite wird unter anderem Serverhardware in verschiedenen Regionen der Welt benötigt, damit die Latenz und damit die Reaktionszeit der Applikation für alle Benutzer möglichst gering ist, ebenso werden neben den eigentlichen Entwicklern auch Administratoren zur Verwaltung dieser Hardware benötigt. Diese Kosten sind für viele Unternehmen ein hohes Risiko. Daher bietet es sich an, die Infrastruktur in die *Cloud* zu verlagern und einen *Infrastructure-as-a-Service (IaaS)* Anbieter wie zum Beispiel *Amazon AWS* zu verwenden [9].

Abbildung 1.3 zeigt den Aufbau einer 2-Tier-Architektur unter Verwendung der AWS-Cloud. Es wird die NoSQL-Datenbank *DynamoDB* und der Fileserver *Simple Storage Service (S3)* verwendet. Diese Produkte stellen beide *REST-APIs* zur Verfügung und können somit direkt per *HTTP* vom Browser angesprochen werden [65][66]. Der Betreiber der Webseite muss somit nur noch die eigentliche Applikation entwickeln und nicht mehr die zugrundeliegende IT-Infrastruktur verwalten. Die komplette Wartung dieser übernimmt der Provider [21].

Ein weiterer Vorteil ist das *Pay-as-you-go* Preismodell, dieses wird häufig von Cloud-Anbietern verwendet wird [21]. Der Betreiber einer Webapplikation muss daher nur für den Speicherplatz und Datenverbrauch bezahlen, den er wirklich benötigt. Was im Besonderen für *Start-Ups* und mittelständische Unternehmen von Vorteil ist, da die Kosten überschaubar bleiben und keine teure Hardware angeschafft werden muss. Das Unternehmen kann sich somit voll auf die Entwicklung konzentrieren.

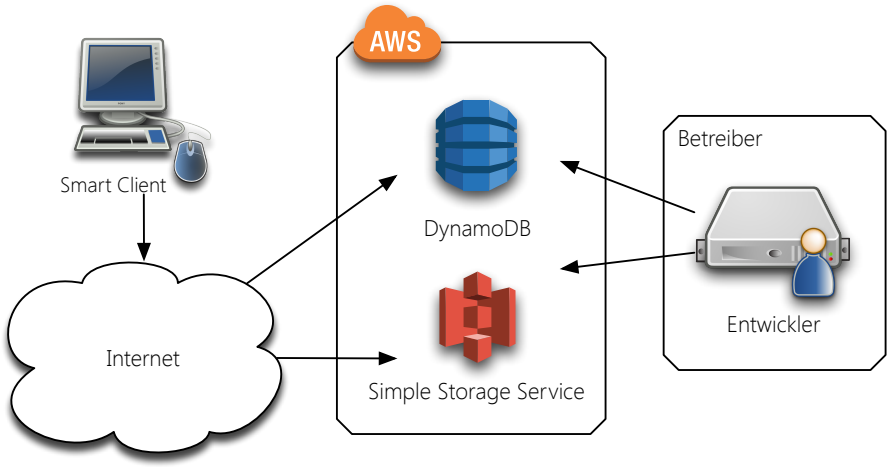


Abbildung 1.3: Aufbau einer 2-Tier-Architektur auf Basis der Amazon-Cloud

## 1.1 Motivation und Zielsetzung

Durch die in der Einleitung aufgezeigten Vorteile eines Smart Clients im Besonderen aufbauend auf einer Cloud-Infrastruktur bietet sich die tiefergehende Betrachtung des Themas an.

Da die auszuführende Logik einer Webapplikation vom klassischen Applikationsserver in den Client verlagert wird, muss die Datenbank Aufgaben zur Kontrolle der übermittelten Daten übernehmen und einen für die Cloud-Infrastruktur passenden Authentifizierungsmechanismus unterstützen.

Für eine kommerzielle Webapplikation ist es unabdingbar, dass sie durch Suchmaschinen gefunden werden kann. Ein Smart Client, wie er in dieser Arbeit betrachtet wird, ist in JavaScript geschrieben. Suchmaschinen führen beim Erfassen von Webseiten kein JavaScript aus [17]. Daher müssen Daten für diese bereits vorverarbeitet ausgeliefert werden. Ein Ansatz ist, den Smart Client auf dem Server auszuführen und die Suchmaschine erhält anschließend die jeweiligen relevanten Informationen.

Des Weiteren stellen sich Fragen zum allgemeinen Aufbau des Clients und der Persistenzschicht, sowie die Kommunikation mit dieser. So lässt sich zum Beispiel die Kommunikation über *Long Polling* oder eventbasiert per *WebSockets* realisieren. Ebenso werden aktuelle Lösungen zur Erstellung von Smart Clients betrachtet und verglichen. Dazu zählen unter anderen die Frameworks *AngularJS* und *Ember.js*. Das Ziel dieser Arbeit ist die Diskussion der genannten Probleme. Es werden aktuelle Lösungen betrachtet und eine prototypische Implementierung eines Smart Clients auf einer 2-Tier-Architektur realisiert. Im nachfolgenden Unterkapitel wird der Use-Case für diesen Prototypen beschrieben.

## 1.2 Use-Case

Der Use-Case für den Prototypen und die in dieser Arbeit vorkommenden Beispiele ist ein Kurznachrichtendienst ähnlich zu *Twitter*. Er ermöglicht den Benutzern das Schreiben von Nachrichten, welche möglichst in Echtzeit zu seinen Abonnenten übertragen werden. Es wird ebenfalls eine Benutzerverwaltung benötigt und implementiert.

Dieser Anwendungsfall ist optimal, um die Vorteile eines Smart Clients in einer 2-Tier-Architektur zeigen:

- Große Datenmengen, die sich optimal in einer NoSQL Datenbank unterbringen lassen.
- Geschriebenen Nachrichten sollen in Echtzeit zu den Abonnenten übertragen werden. Dafür kann entweder das Long-Polling (Kapitel 3.1.1) oder ein eventbasiertes Verfahren auf Basis von Web-Sockets (Kapitel 3.1.3) verwendet werden.
- Es finden größtenteils Lesezugriffe auf die Datenbasis statt, wodurch ein erheblicher Geschwindigkeitsvorteil durch Orestes entsteht. (Kapitel 2)

- Benutzer sind international verteilt womit sich ein weiterer Vorteil Orestes zeigen lässt. (Kapitel 2)
- Durch die benötigte Benutzerverwaltung wird ein geeignetes Authentifizierungsverfahren benötigt. (Kapitel 4)
- Die Nachrichten müssen von Suchmaschinen durchsuchbar sein. (Kapitel 5)

## 1.3 Aufbau der Arbeit

Im 2. Kapitel wird zunächst tiefergehend auf das Orestes-Protokoll eingegangen. Es wird das System im Ganzen und die Architektur beschrieben. Anschließend wird das Web-Caching erörtert, welches ein elementarer Bestandteil ist und einen Performanzgewinn mit sich bringt. Mittels eines sogenannten *Bloomfilters* kann trotz der Verwendung von Web-Caching-Technologien die Konsistenz der auszuliefernden Daten sichergestellt werden. Auf diesen wird im darauffolgenden Unterkapitel eingegangen.

Das 3. Kapitel beschäftigt sich mit den Grundlagen eines Smart Clients auf Basis von JavaScript. Dafür werden zunächst im Unterkapitel die Grundlagen aller benötigten Technologien erläutert. Anschließend werden unterschiedliche Frameworks zur Entwicklung eines Smart Clients diskutiert. Diese sind *Backbone.js*, *AngularJS* und *Ember.js*. Daraufhin folgt ein Fazit mit einem Vergleich zu den vorgestellten Frameworks.

In dem nachfolgenden 4. Kapitel werden sicherheitsrelevante Themen besprochen. Im Besonderen werden unterschiedliche Authentifizierungsverfahren betrachtet und ein für das Orestes-Protokoll passende Verfahren vorgestellt.

Kapitel 5 beschäftigt sich mit verschiedenen Möglichkeiten zur Vorgenerierung des Smart Clients, damit auch Suchmaschinen diesen indizieren können. Es wird dafür tiefergehend auf *Node.js* sowie auf das Tool *PhantomJS* eingegangen. Dieses ermöglicht die serverseitige automatisierte Erstellung von HTML-Seiten. Des Weiteren wird eine Lösung für das Framework *Backbone.js* Namens *Rendr* beschrieben und beurteilt.

Im nachfolgenden Kapitel 6 wird ein Prototyp implementiert, der auf den zuvor vorgestellten Techniken basiert, um die Machbarkeit eines SEO-Optimierten Smart Clients in einer 2-Tier-Architektur zu zeigen.

Die Arbeit endet mit einem Fazit im 7. Kapitel und gibt einen Ausblick zu den besprochenen Themen.

# 2 Orestes

Bei Orestes, was ein Akronym für *Objects RESTfully encapsulated in standard formats* ist, handelt es sich um ein Zugriffsprotokoll für Datenbanken. Es basiert auf dem *REST*-Paradigma und kann direkt über HTTP angesprochen werden. Das Protokoll wird seit 2010 von Felix Gessert und Florian Bücklers entwickelt.

In diesem Kapitel werden zunächst die Grundlagen von Orestes erörtert. Anschließend wird das *Web-Caching*, welches ein essentieller Teil des Protokolls ist, näher betrachtet. Im darauffolgenden Unterkapitel wird näher auf die Sicherstellung der Konsistenz sowie die Behandlung von Transaktionen im Orestes-Protokoll und auf den so genannten *Bloomfilter* eingegangen. Das Kapitel endet mit einem Fazit zu dem Orestes-Protokoll.

Das Kapitel basiert weitestgehend auf [24] und [25].

## 2.1 Grundlagen

In der klassischen Entwicklung einer Anwendung, die zur Datenhaltung eine Datenbank verwendet, greift diese, wie in Abbildung 2.1 gezeigt, über ein Netzwerk auf das Datenbanksystem (DBS) zu.

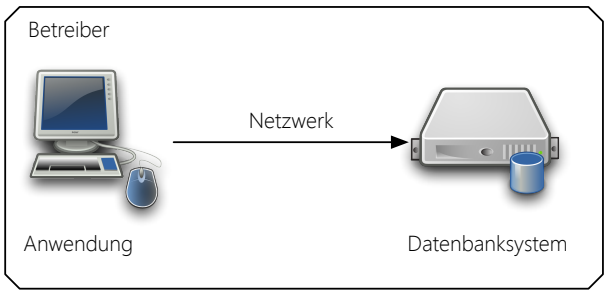


Abbildung 2.1: Architektur einer klassischen Anwendung

Für die Kommunikation zwischen Anwendung und Datenbanksystem wird in der Regel ein proprietäres Protokoll des DBS-Herstellers verwendet. Die Anwendung muss somit dieses Protokoll beherrschen. Dafür wird eine von dem Hersteller zur Verfügung gestellte Bibliothek verwendet. Dies hat den Nachteil, dass beim Austausch des Datenbanksystems die Persistenzschicht der Anwendung in großen Teilen neugeschrieben werden muss. Im Hinblick auf den aktuellen Trend der *Polyglot Persistence*[1] ergibt sich ein weiterer erheblicher Nachteil. Für jedes verwendete Datenbanksystem muss eine separate

[1] *Polyglot Persistence* beschreibt die Verwendung von unterschiedlichen Datenbanktypen innerhalb eines Datenmodells [20].

Bibliothek verwendet werden, damit das jeweilige Kommunikationsprotokoll der Datenbank unterstützt wird.

Orestes löst diese Probleme, indem es die proprietären Schnittstellen der verwendeten Datenbanken durch eine eigene, basierend auf REST/HTTP, kapselt. HTTP steht für *Hypertext Transfer Protocol* und ist ein Anwendungsprotokoll, welches auf das *Transmission Control Protocol (TCP)* aufsetzt. Die Verwendung von HTTP hat den Vorteil, dass es im Unterschied zu nativen TCP-Verbindungen, wie sie von properitären Datenbankschnittstellen verwendet werden, von Proxys und Firewalls nicht blockiert wird. Des Weiteren handelt es sich bei HTTP um ein Protokoll, was von gängigen Programmiersprache unterstützt wird. Insbesonders im Hinblick auf einen im Webbrowser laufenden Smart Client, ist diese Wahl optimal, da es sich bei HTTP um das Protokoll des Webs handelt und somit alle Webbrowser dieses unterstützen. Es müssen daher für den in JavaScript geschriebenen Smart Client keine zusätzlichen Bibliotheken verwendet werden [60].

Abbildung 2.2 zeigt den architektonischen Aufbau einer Anwendung unter der Verwendung von Polyglot-Persistence und dem Orestes-Protokoll. Die Anwendung muss lediglich den Orestes-Server kennen und kann über diesen alle zur Verfügung stehenden Datenbanken über ein einheitliches *Application Programming Interface (API)* ansprechen. Da nicht alle Funktionen der unterschiedlichen Datenbanksysteme über die Orestes-API abgebildet werden können, bietet es die Möglichkeit, Anfragen direkt an eine Datenbank weiterzuleiten.

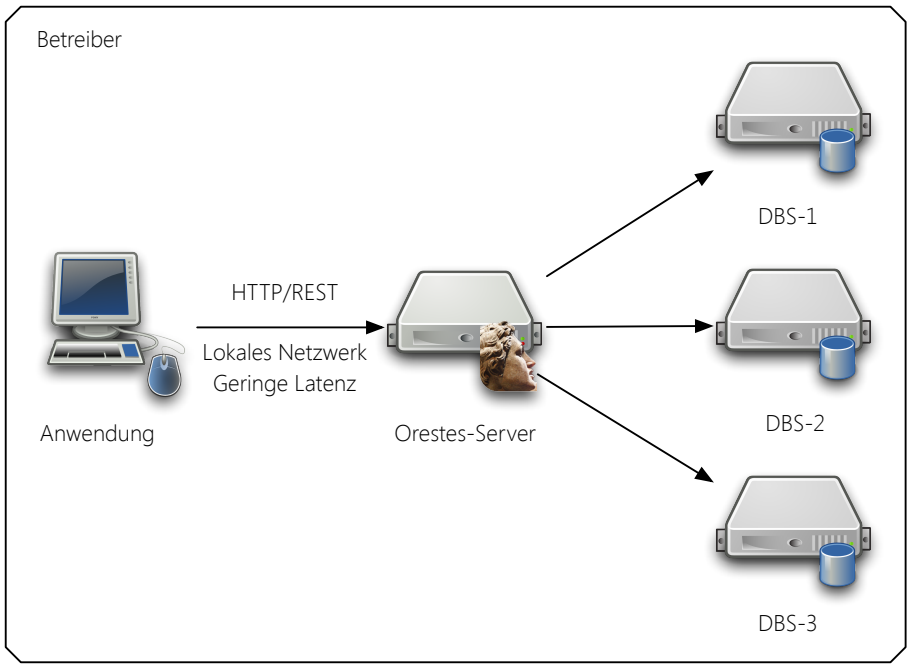


Abbildung 2.2: Verwendung von mehreren Datenbanken mittels Orestes-Protokoll

Wie bereits in der Einleitung beschrieben, kann für viele Anwendungsgebiete die Verwendung eines Database-As-A-Service (DBaaS) Anbieter von Vorteil sein. Abbildung 2.3 zeigt einen möglichen Aufbau einer Infrastruktur unter der Verwendung des Orestes-

Protokolls sowie der AWS-Cloud. Diese haben allerdings den Nachteil, dass im Vergleich zu einer Anwendung, die eine lokale Datenbank verwendet, die Latenz erheblich höher ist und somit zu einem *Bottleneck* der Anwendung werden kann.

Das Orestes-Protokoll adressiert dieses Problem indem es Leseanfragen mittels des *HTTP-Caching-Modells* beschleunigt. Die verwendeten Technologien und Möglichkeiten werden in Kapitel 2.2 näher betrachtet.

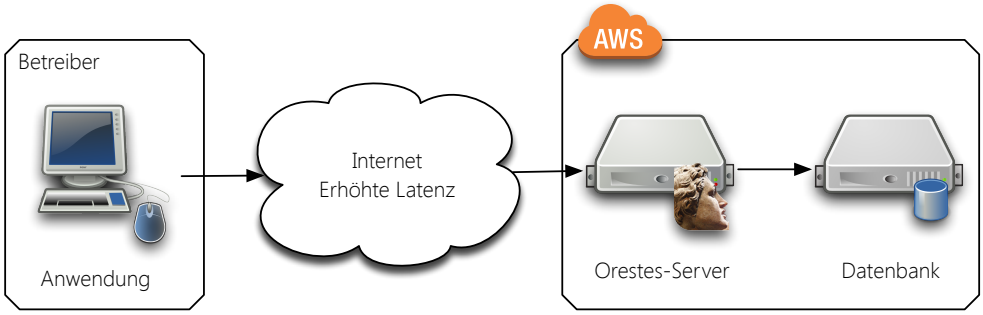


Abbildung 2.3: Auslagerung der Datenbank mittels das Orestes-Protokoll in die Cloud

Durch die Verwendung von Web-Caches kann die Konsistenz der Daten nicht mehr garantiert werden, da durch Schreiboperationen auf der Datenbank Veränderungen durchgeführt werden können, von denen im Cache gehaltene Daten betroffen sind. Dadurch entsteht die Möglichkeit, dass eine Anfrage von einem Cache mit veralteten Daten beantwortet wird. Abbildung 2.4 zeigt ein solches Szenario. *Client 2* möchte das Datum *a* verändern und schickt eine entsprechende Anfrage an den Orestes-Server, welcher diese an die entsprechende Datenbank weiterleitet. Anschließend wird dieses von *Client 1* gelesen. Da das Datum bereits in einem Web-Cache vorgehalten wird, erhält *Client 1* eine veraltete Version. Es kann somit lediglich garantiert werden, dass die Daten zu einem zukünftigen Zeitpunkt (spätestens nach Ablauf der Vorhaltezeit), konsistent sind. Dies wird auch als *eventual consistency* bezeichnet.

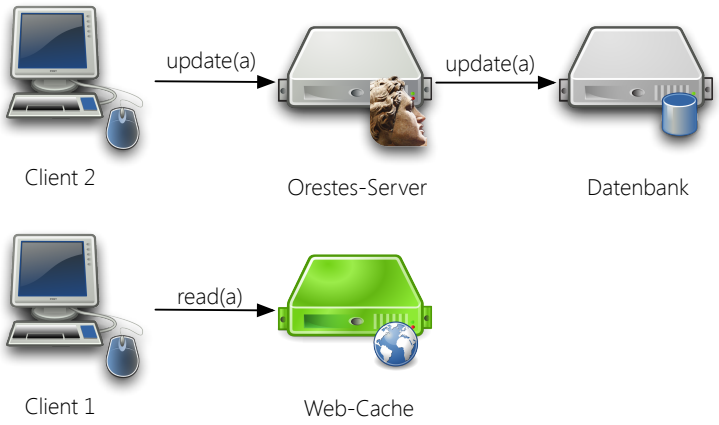


Abbildung 2.4: Konsistenzverletzung durch Web-Caches

Viele Vertreter der NoSQL-Datenbanken garantieren zur Performanzsteigerung lediglich eine eventual consistency. Das Orestes-Protokoll bietet hingegen die Möglichkeit, die Einhaltung einer starken Konsistenz zu gewährleisten. Es werden dafür Bloomfilter verwendet. Auf diese wird in Kapitel 2.3 näher eingegangen.

Im nachfolgenden Unterkapitel wird das Thema *Web-Caching* betrachtet.

## 2.2 Web-Caching

Wie im vorherigen Unterkapitel beschrieben, gehört das Web-Caching zu einem essentiellen Bestandteil des Orestes-Protokolls. Mit dessen Hilfe wird versucht, den Nachteil der erhöhten Latenz des Cloud-Computings auszugleichen. Ebenso wird mittels Web-Caching eine horizontale Skalierung realisiert.

Es wird zwischen sechs unterschiedlichen Arten von Caches unterschieden [54]:

***Client Cache:*** Daten werden lokal im Client zwischengespeichert. Der bekannteste Vertreter dieser Art ist der Browser-Cache.

***Forward Proxy Cache:*** Wird im Netzwerk des Clients, welcher über diesen seine Anfragen stellt, die gegebenenfalls direkt vom Forward Proxy Cache beantwortet werden, installiert.

***Web Proxy Cache:*** Befindet sich außerhalb des Client- und Server-Netzwerks. In der Regel bei den jeweiligen *Internet Service Providern (ISP)*.

***Content Delivery Network (CDN):*** CDN-Provider wie zum Beispiel *Akamai, Limelight* oder *AWS* stellen weltweit Server zur Auslieferung von Daten zur Verfügung. Der Client wird bei einer Anfrage an ein CDN nach Möglichkeit an den Knoten mit der für ihn niedrigsten Latenz geleitet.

***Reverse Proxy Cache:*** Wird im Netzwerk des Servers installiert. Anfragen an diesen passieren den Proxy und werden, falls das benötigte Datum im Cache vorhanden ist, von diesem beantwortet.

***Server Cache:*** Der Server speichert ebenfalls Daten, um Anfragen schneller bearbeiten zu können. Die Software *memcached* speichert zum Beispiel Ergebnisse häufig durchgeführter Datenbankanfragen im Arbeitsspeicher, um langsamere Zugriffe auf die Festplatte zu vermeiden.

Abbildung 2.5 zeigt eine Möglichkeit die verschiedenen Caches zu verwenden.

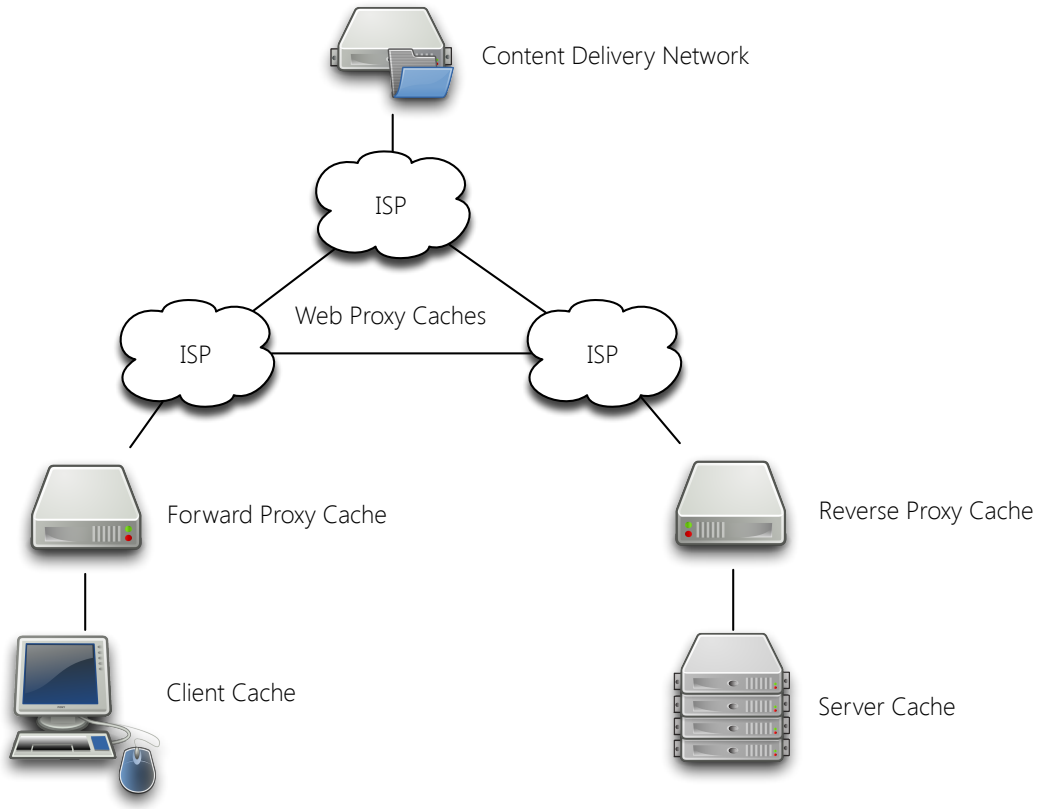


Abbildung 2.5: Verteilung der unterschiedlichen Caches

HTTP bietet Mechanismen, um die Datenhaltung der Caches zu steuern. Im HTTP-Header-Feld *Cache-Control* kann das Attribut *max-age* definiert werden. Dieses gibt in Sekunden an, wie lange eine Antwort des Servers als gültig betrachtet werden kann. Des Weiteren definiert die HTTP-Spezifikation ab Version 1.1 das Feld *ETag*. Mittels diesem kann eine Versionsnummer angegeben werden [30].

Stellt ein Client eine Anfrage an den Server, wird diese zunächst von dem Reverse Proxy bearbeitet. Dieser prüft das Attribut *max-age*. Ist die Zeit noch nicht überschritten, wird die Anfrage vom Proxy beantwortet und der Server wird nicht belastet. Diese Szenario ist in Abbildung 2.6 gezeigt.


Abbildung 2.6: Auslieferung einer Ressource durch den Reverse Proxy Cache

Ist hingegen das maximale Alter bereits überschritten, wie in Abbildung 2.7 zu sehen, wird die vorhandene Ressource als *stale* gekennzeichnet. Der Proxy muss in diesem Fall

vor der Auslieferung eine Anfrage an den Server stellen. Dies bedeutet nicht zwangsläufig, dass der Server die Ressource erneut ausliefert. Der Proxy übermittelt bei seiner Anfrage im HTTP-Header-Feld *If-None-Match* den ETag-Wert. Der Server prüft, ob sich die Versionsnummer der Ressource geändert, falls die nicht der Fall ist, antwortet er mit dem HTTP-Status-Code 304 und übermittelt nur den aktuellen Header. Die Ressource wird also nicht erneut ausgeliefert. Die erneute Auslieferung wird nur bei einer Differenz der ETag-Werte durchgeführt.

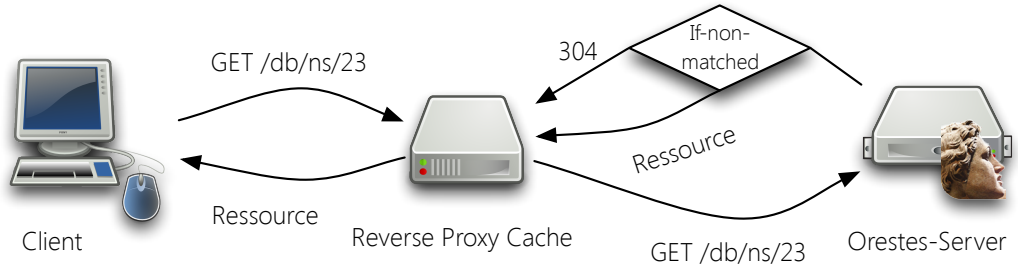


Abbildung 2.7: Auslieferung einer *stale* Ressource durch den Reverse Proxy Cache

HTTP definiert *Request-Methoden*. Folgende Aufzählung zeigt die für Orestes relevanten Methoden:

**GET** Dieser Parameter dient zum Abrufen einer Ressource, welche über einen *Uniform Resource Identifier (URI)* bestimmt wird.

**POST** Dient zum Hinzufügen einer Ressource. Daten werden an einen Pfad, wie zum Beispiel */db/filme*, gesendet und der Server entscheidet, wie die Daten verarbeitet werden. Häufig wird der Ressource eine ID zugewiesen und kann anschließend per GET, zum Beispiel unter */db/filme/5*, abgerufen werden.

**PUT** Dient ebenfalls zum Übertragen von Daten an den Server. Im Unterschied zu POST wird mittels PUT explizit eine Ressource per URI angegeben. PUT wird daher im Normalfall für das Verändern einer Ressource verwendet.

**DELETE** Diese Methode wird zum Löschen einer über die URI identifizierten Ressource verwendet.

Auf die restlichen HTTP Methoden wird nicht weiter eingegangen, da diese für ein grundlegendes Verständnis des Orestes-Protokolls nicht benötigt werden.

Da PUT und DELETE Anfragen ebenfalls den Reverse Proxy Cache passieren, kann dieser direkt die gegebenen Ressourcen als stale markieren und beim Nächsten Request erneut vom Orestes-Server anfordern. Eine direkte Übernahme der mittels PUT geänderten Daten in den Cache ist nicht möglich, da der Orestes-Server diese gegebenenfalls ablehnen kann.

Im nachfolgenden Unterkapitel wird genauer auf das Thema Transaktionen im Orestes-Kontext eingegangen.

## 2.3 Transaktionen

Das *CAP*-Theorem besagt, dass es in einem verteilten System nicht möglich ist, die drei Eigenschaften *Konsistenz (**C**onsistency)*, *Verfügbarkeit (**A**vailability)* und *Partitionstoleranz (**P**artition Tolerance)* zur selben Zeit einzuhalten.

Das Orestes-Protokoll erlaubt dem Client zu entscheiden, ob es durch die Verwendung von Web-Caching den Verlust einer garantierten Konsistenz im Sinne des CAP-Theorems hinnehmen kann oder ob die Eigenschaften des zugrundeliegenden Datenbanksystems Verwendung finden. Orestes stellt dafür einen Bloomfilter zur Verfügung.

Bei einem Bloomfilter handelt es sich um eine sehr kompakte und schnelle Datenstruktur zur Speicherung eines *Fingerabdruckes* eines Eingabewertes. Der Filter besteht aus einem Bit-Array mit der Länge $m$, welches mit Nullen initialisiert wird. Um die Eingabewerte auf dieses Abzubilden werden $k$ unterschiedliche Hashwerte mit einer Hashfunktion gebildet. Die gebildeten Hashwerte liegen im Wertebereich 0 bis $m - 1$ und bestimmen, an welchen Punkten im Bit-Array eine 1 eingefügt wird.

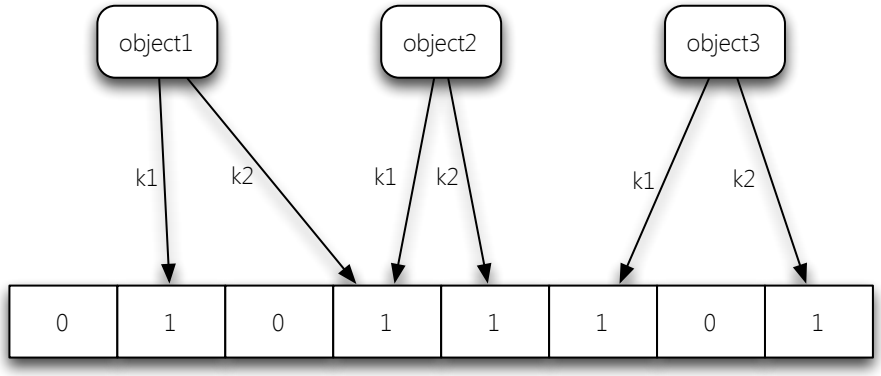


Abbildung 2.8: Beispiel eines Bloomfilters für $m = 8$ und $k = 2$

Abbildung 2.8 zeigt ein Beispiel eines Bloomfilters mit $m = 8$ und $k = 2$. Für die Eingabewerte *object1*, *object2* und *object3* werden die jeweiligen zwei Hashwerte gebildet. Durch *object1* wird Position zwei und vier im Bloomfilter gesetzt. *object2* Position vier und fünf. *object3* Position sechs und acht. Um zu prüfen, ob sich ein Objekt im Bloomfilter befindet, werden die Hashwerte von diesem erstellt. Befindet sich an beiden Positionen eine 1, enthält der Bloomfilter das Objekt.

Wie durch *object1* und *object2*, die beide Position vier belegen, angedeutet, kann es auch zu Kollisionen kommen. Diese verhindern, falls sie beim Erstellen des Bloomfilters auftreten, nicht die Funktionsfähigkeit. Allerdings kann bei der Prüfung, ob ein Eingabewert sich im Bloomfilter befindet, eine Kollision zu einem *False-Positive* führen, da die Felder des zu prüfenden Objektes bereits durch andere belegt sind.

So kann zum Beispiel der Eingabewert eines weiteren Objektes die Positionen zwei und acht belegen. Bei der Überprüfung, ob sich dieses Objekt im Bloomfilter befindet, wird daher mit ja beantwortet, obwohl dieses nicht eingefügt wurde.

Die Wahrscheinlichkeit für einen False-Positive ist von $m$, $k$ und $n$ abhängig, wobei $n$ die Anzahl der bereits vorhandenen Einträge ist. Diese lässt sich wie folgt berechnen:

$$FP = (1 - (1 - \frac{1}{m})^{kn})^k$$

Es muss also beim Festlegen der Werte $m$ und $k$ beachtet werden, wie viele Einträge für den besagten Bloomfilter zu erwarten sind. Des Weiteren haben die Werte Einfluss auf die Größe sowie Laufzeit. Die Größe ist äquivalent zu $m$. Die Laufzeit zur Überprüfung oder das Einfügen eines Wertes berechnet sich aus den Laufzeiten der gewählten Hashfunktion. Ebenso muss abgewogen werden, welche Auswirkung die Entstehung eines False-Positives hat.

Das Orestes-Protokoll stellt einen Bloomfilter zur Verfügung, um den Client überprüfen zu lassen, ob eine Ressource, die von einem Web-Cache ausgeliefert wird, stale ist. Wird ein Objekt in der Orestes zugrundeliegenden Datenbank geändert, wird dieses im Bloomfilter festgehalten. Ruft ein Client dieses Objekt aus einem Web-Cache ab, kann er vom Orestes-Server den aktuellen Filter erhalten und überprüfen, ob vor Ablauf der Vorhaltezeit das Objekt geändert wurde und gegebenenfalls das aktuelle direkt vom Orestes-Server abrufen. Somit ist es dem Client überlassen, ob er mit einem möglichen veralteten Objekt arbeiten möchte oder nicht.

False-Positives des Bloomfilters führen im Orestes-Protokoll dazu, dass der Client eine erneute und somit unnötige Anfrage zur Beschaffung der Ressource an den Orestes-Server stellt. Ein False-Positive verhindert also nicht die Funktionsfähigkeit des Protokolls. Es entsteht lediglich eine erhöhte Last für den Server und eine länger Laufzeit der auszuführenden Transaktion. Diese Fälle sind zu akzeptieren, so lange sie mit einer möglichst geringen Wahrscheinlichkeit eintreten.

Da nach Ablauf der Vorhaltezeit eines geänderten Objektes davon ausgegangen wird, dass dieses nicht mehr verwendet wird, kann es aus dem Bloomfilter wieder entfernt werden. Der Filter aus Abbildung 2.8 unterstützt das Löschen von Einträgen nicht, da eine Position von mehreren Eingabewerten belegt sein kann. Wird zum Beispiel *object1* gelöscht, werden die Positionen zwei und vier auf null gesetzt. Dieses Vorgehen hat zur Folge, dass auch *object2* gelöscht wird. Aus diesem Grund erweitert der so genannte *Counting Bloomfilter*, welcher in Abbildung 2.9 dargestellt wird, die gezeigte Version um die benötigte Löschfunktion.

Für die Löschfunktionalität wird ein Array bestehend aus Integer-Werten ebenfalls mit der Länge $m$ hinzugefügt. In jedem Array-Feld wird gespeichert, von wie vielen Eingabewerten das zugehörige Feld im Bloomfilter verwendet wird. Position vier wird von *object1* und *object2* belegt. Der Zähler steht für dieses Feld somit auf zwei. Beim Löschen von *object1* wird der Zähler und das Feld des Filters von Position zwei null gesetzt. Position vier wird nicht auf null gesetzt, da der zugehörige Counter einen Wert $> 0$ besitzt. Die Möglichkeit zur Entfernung von Einträge geht somit zulasten des Speicherbedarfs.

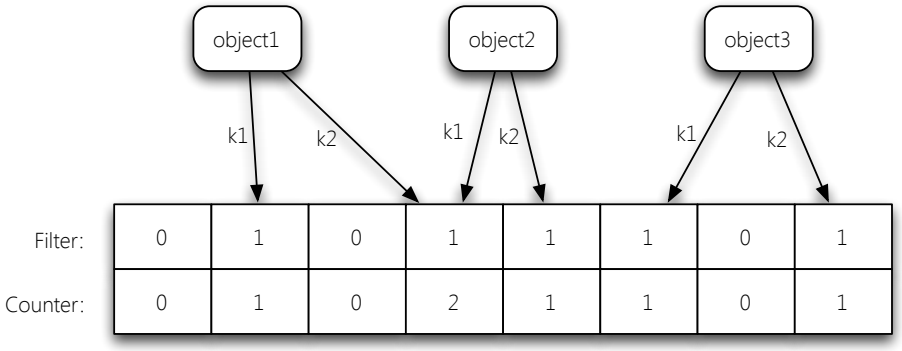


Abbildung 2.9: Der *Counting Bloomfilter* enthält eine Löschfunktion

Für das Orestes-Protokoll spielt dies nur eine nebensächliche Rolle, da die Übertragung des Filters ohne Counter an den Client ausreichend ist. Der erhöhte Platzbedarf muss somit nicht beachtet werden.

Die Verwendung des Bloomfilters macht für den Client im Besonderen beim Ausführen von schreibenden Transaktionen Sinn, da durch ihn verhindert werden kann, dass sie aufgrund eines stale Objektes abgebrochen wird. Eine hundertprozentige Verhinderung ist dennoch nicht möglich, da zwischen Abruf des Bloomfilters und Start der Transaktion das Objekt von einem anderen Client verändert werden kann. In diesem Fall muss die Transaktion abgebrochen werden.

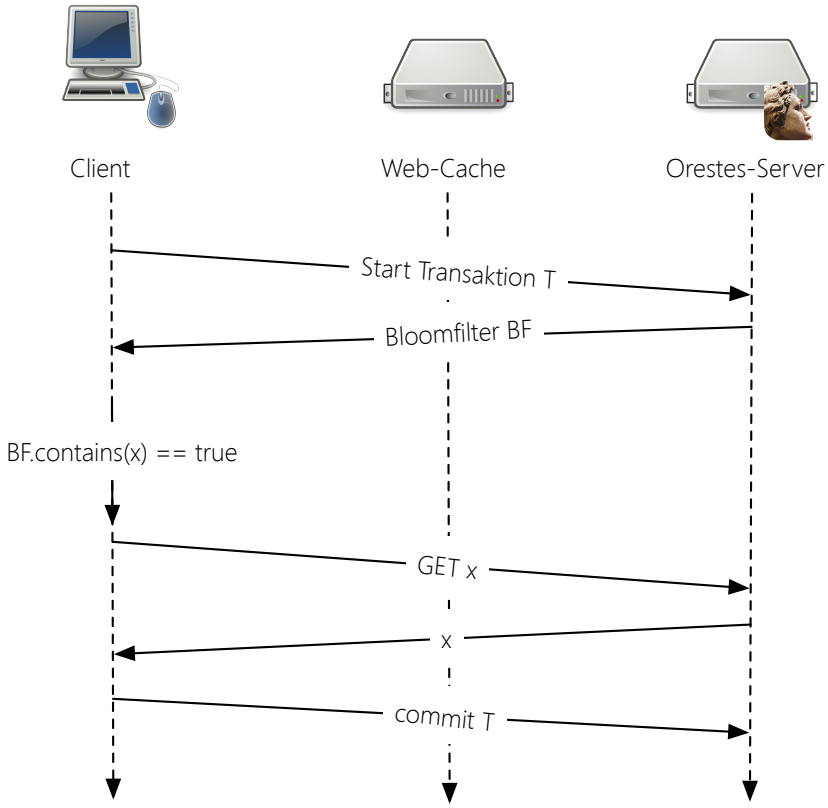


Abbildung 2.10: Ablauf einer Transaktion

Abbildung 2.10 zeigt den Ablauf einer Transaktion unter Verwendung des Bloomfilters. Der Client startet seine Transaktion am Orestes-Server und erhält den aktuellen Bloomfilter. Er prüft, ob die abzufragende Ressource im Bloomfilter vorhanden ist. Orestes verwendet zur Bildung der Hashwerte eine eindeutige Objekt-ID. Die Prüfung ist somit vor der Abfrage des eigentlichen Objektes möglich. In dem gezeigten Beispiel beinhaltet der Bloomfilter das Objekt. Dies bedeutet, dass die Ressource stale ist. Der Client ruft sie daher direkt vom Orestes-Server ab. Abschließend kann der Client somit seine Transaktion beenden. Wenn die Ressource nicht im Bloomfilter vorhanden ist, wird die Ressource, falls zwischengespeichert, aus einem Web-Cache geladen.

Im nachfolgenden Unterkapitel findet ein kurzes Fazit zu Orestes statt.

## 2.4 Fazit

Das Orestes-Protokoll bietet erhebliches Potential zur Steigerung der Geschwindigkeit von lesenden Transaktion. Im Besonderen für Webanwendungen macht der Einsatz Sinn, da die Verwendung von REST auf HTTP diesen erheblich erleichtert und der Großteil von Datenbankanfragen lesend sind.

Eine Einsparung von Kosten durch die Verwendung ist ebenso zu erwarten, da Web-Caches zur horizontalen Skalierung verwendet werden. Diese Caches sind zum Beispiel als Reverse Proxy Cache beim Internet Service Provider vorhanden und die Kosten dafür müssen somit nicht vom Betreiber der Datenbank getragen werden.

Um eine große Anzahl von Requests auf eine Datenbank verarbeiten zu können, wird häufig ein Datenbankcluster verwendet. Ein verbreitet Lizenzmodell ist die Bezahlung pro verwendeten Server. Eine Verkleinerung des Clusters ist somit wünschenswert. Diese Möglichkeit bietet das Orestes-Protokoll.

Im nachfolgenden Kapitel wird auf die Entwicklung von Smart Clients aufbauend auf einer 2-Tier-Architektur eingegangen.

# 3 Smart Clients

Bei der Entwicklung eines Smart Clients aufbauend auf einer 2-Tier-Architektur in JavaScript stellen sich im Besonderen Fragen zur Kommunikation mit der Datenbank sowie zur Architektur des Clients.

In diesem Kapitel werden zunächst die Grundlagen geschaffen. Anschließend folgt die Betrachtung mehrerer JavaScript Frameworks, welche die Entwicklung von Smart Clients vereinfachen sollen.

## 3.1 Grundlagen

Ein in der klassischen Webentwicklung aufbauend auf einer 3-Tier-Architektur häufig verwendetes Architekturmuster ist das *Model-View-Controller*-Pattern. Bei diesem wird die Software in Datenmodell (*engl. Model*), Präsentation (*engl. View*) und Programmsteuerung (*engl. Controller*) aufgeteilt [43].

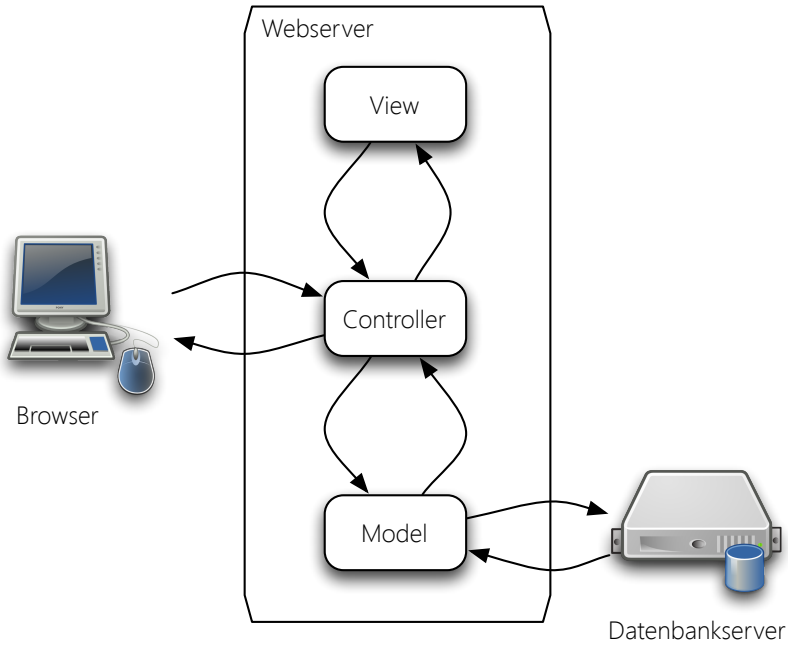


Abbildung 3.1: Model-View-Controller-Pattern in einer 3-Tier-Architektur

Die Abbildung 3.1 zeigt den Ablauf des MVC-Musters. Der Client (Browser) stellt eine Anfrage an den Controller. Dieser verwendet das Model, um die benötigten Daten für die View aufzubereiten und zur Verfügung zu stellen. Das Model lädt im Normalfall die

Daten aus einer Datenbank und speichert gegebenenfalls Änderungen in dieser. Aus der View wird eine HTML-Seite generiert, die anschließend vom Controller an den Browser ausgeliefert wird.

Durch die Verwendung des Musters wird der Programmentwurf flexibel gehalten. Dieses vereinfacht die Möglichkeit, neue Komponenten hinzuzufügen oder auszutauschen, um geänderten Anforderungen gerecht zu werden.

Durch den Wegfall des Webservers stellt sich die Frage nach einem geeigneten Architekturmuster für die Programmierung eines Smart Clients. Die naheliegendste Lösung ist daher die Übertragung des MVC-Patterns in den Smart Client (Abbildung 3.2). Die in diesem Kapitel vorgestellten Frameworks gehen genau diesen Weg oder verwenden das MVC-Muster in einer abgewandelten Form (*Model-View-Whatever (MVW)*).

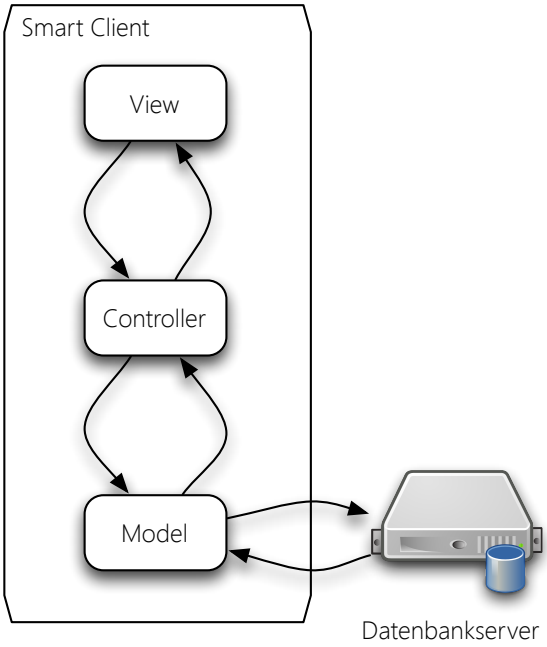


Abbildung 3.2: Model-View-Controller-Pattern auf einer 2-Tier-Architektur aufbauend

In der klassischen Oberflächenentwicklung, zum Beispiel in Java, wird häufig aufbauend auf dem MVC-Pattern das so genannte *Beobachter* Entwurfsmuster (*engl. Observer Pattern*) verwendet [73]. Dies kann unter anderem dazu benutzt werden, die Oberfläche zu aktualisieren, wenn sich etwas an dem Model geändert hat. Um dieses Pattern in eine Webapplikation zu übertragen, stehen unterschiedliche Technologien zur Verfügung: *Long Polling*, *Server-Sent Events* und *WebSockets* [71][10].

### 3.1.1 Long-Polling

Die einfachste Methode ist das so genannte *Polling*. Bei diesem Verfahren sendet der Client in einem definierten Abstand Requests an den Server, um zu überprüfen, ob Daten aktualisiert wurden.

```
setInterval(
    $.getJSON('/db/tweets/subscribe', function (tweet) {
        tweets.push(tweet);
    }), 30000
);
```

Listing 3.1: Polling

Abbildung 3.1 zeigt die Implementierung einer simplen Polling Methode. Der Funktion *setInterval* werden zwei Parameter übergeben. Der erste beinhaltet die Funktion, die ausgeführt wird. Der zweite Parameter gibt in Millisekunden an, wie lange zwischen der Ausführung der übergebenen Funktion gewartet wird. Die Funktion *$.getJson(URI, function(){})* stammt aus dem JavaScript-Framework *jQuery* und wird verwendet, um eine *JSON*[1]-Ressource von einem Webserver abzurufen. Die übergebene Funktion wird nach einem erfolgreichen Laden der Ressource aufgerufen und das JSON wird dieser als Parameter übergeben. Das gezeigte Beispiel ruft somit alle 30 Sekunden die URI */db/tweets/subscribe* auf und fügt das geladene JSON einem Array hinzu.

Da bei der gezeigten Methode lediglich die Ressource in einem festgesetzten Interval neu geladen wird, kann der Server nur in diesem Abstand mit dem Client kommunizieren. Das Long Polling versucht dies zu umgehen, indem der Request, der vom Client beim Polling an den Server gesendet wird, vom Server erst beantwortet wird, wenn neue Daten zur Verfügung stehen.

```
(function poll(){
  $.getJSON('/db/tweets/subscribe',function(tweet){
    tweets.push(tweet);
  }).complete(poll);
})();
```

Listing 3.2: Long-Polling

In Abbildung 3.2 ist eine Clientseitige Implementierung des Long-Polling-Verfahren zu sehen. Es wird eine Funktion *poll* erstellt, die mittels *$.getJSON* eine Ressource lädt. Der Request wird vom Server erst beantwortet, wenn neue Daten zur Verfügung stehen. Wird auf diesen geantwortet, wird die Funktion über die *complete*-Methode wiederholt ausgeführt.

Bei dieser Methode handelt es sich lediglich um einen Workaround. Ihre Funktionalität ist stark vom Browser abhängig. Ist zum Beispiel die Zeit für einen Timeout des Clients zu gering, stellt dieser in einem zu kurzem Interval Anfragen an den Server stellen.

### 3.1.2 Server-Sent Events

*Server-Sent Events (SSE)* ist ein vom *Word Wide Web Consortium (W3C)* standardisiertes Verfahren. Der Browser registriert sich am Server und dieser kann, sobald neue Informationen zur Verfügung stehen, den Client informieren. Es handelt sich dabei um eine

[1] *JSON* steht für *JavaScript Object Notation* und ist ein textbasiertes Format zum Datenaustausch.

unidirektionale Verbindung. Es kann somit lediglich der Server mit dem Client kommunizieren [33].

```
1  var source = new EventSource('/db/tweets/subscribe');
2
3  source.addEventListiner('message', function(event) {
4    var tweet = JSON.parse(event.data);
5    tweets.push(tweet)
6  }, false);
7
8  source.addEventListener('open', function(event) {
9    console.log('Verbindung wurde hergestellt');
10 }, false);
11
12 source.addEventListener('error', function(event) {
13   console.log('Fehler: ' + event.readyState);
14 }, false);
```

Listing 3.3: Server-Sent Events

Abbildung 3.3 zeigt eine mögliche Implementierung auf der Seite des Clients. Es wird zunächst ein neues Objekt vom Typ *EventSource* erstellt. Diesem wird die URI der anzusprechenden Ressource übergeben. Anschließend können mittels der Methode *addEventListener* sogenannte *Event-Listener* übergeben werden, die beim Auftreten eines bestimmten Typs von Event aufgerufen werden. Der Eventtyp wird als erster Parameter übergeben. Der zweite definiert die Funktion, die beim Auftreten eines Events ausgeführt wird. Dieser wird das Event als Parameter übergeben. Die Daten, die vom Server übertragen werden, lassen sich mittels *event.data* abrufen.

Die in der Abbildung 3.3 gezeigten Typen sind vordefinierte Events:

***open:*** Wird nach einem erfolgreichen Verbindungsaufbau ausgeführt.

***error:*** Findet ein Fehlverhalten wie zum Beispiel ein Verbindungsabbruch statt, wird dieses Event ausgeführt.

***message:*** Wird beim Auftreten einer neuen Servernachricht aufgerufen.

```
1 id: 5\n
2 retry: 10000\n
3 event: newtweet\n
4 data: {\n
5 data: "tweet": "NoSQL ist die Zukunft!", \n
6 data: "user": "felix", \n
7 data: }\n\n
```

Listing 3.4: Datenformat

Abbildung 3.4 zeigt das Format in dem der Server mit dem Client kommuniziert. Die Attribute *id*, *retry* und *event* sind optional und *data* enthält die zu übertragenden Daten. Sie können in mehrere Zeilen aufgeteilt werden. Dafür muss jede Zeile mit einem $\backslash n$ enden. Wichtig ist dabei zu beachten, dass die letzte Zeile des Events mit $\backslash n \backslash n$ endet.

Über das Feld *id* lässt sich eine ID definieren. Falls der Browser die Verbindung zum Server verliert, kann er somit diese ID beim erneuten Verbinden über das HTTP-Header-Attribut *Last-Event-ID* angeben und der Server weiß, welche Nachrichten der Browser verpasst hat und kann diese übertragen.

Mittels *retry* kann der Server in Millisekunden definieren, wie lange der Browser nach einem Verbindungsabbruch wartet, um sich erneut zu verbinden.

Über das Attribut *event* können eigene Eventtypen definiert werden. Somit kann der Client auf die für ihn passenden Events reagieren und gegebenenfalls andere ignorieren. In dem gezeigten Beispiel wird der Eventtyp *newtweet* übergeben. Abbildung 3.5 zeigt die clientseitige Implementierung, um eine Aktion für diesen Typen auszuführen.

```
1 var source = new EventSource('/db/tweets/subscribe');
2
3 source.addEventListiner('newtweet', function(event) {
4   var tweet = JSON.parse(event.data);
5   tweets.push(tweet)
6 }, false);
```

Listing 3.5: Selbstersteller Eventtyp

### 3.1.3 WebSockets

Bei *WebSockets* handelt es sich um eine Browser-API, die es ermöglicht *Socket*-Verbindung zu erstellen. Es kann somit eine bidirektionale Verbindung zum Server hergestellt werden. Dies hat den Vorteil gegenüber der Server-Sent-Event-Methode, dass der Browser über diesen Kanal ebenfalls mit dem Server kommunizieren kann [34].

```
1  var wsConnection =
2    new WebSocket('ws://orestes.info/db/tweets/subscribe');
3
4  wsConnection.onopen = function() {
5    console.log('Verbindung wurde hergestellt!');
6    wsConnection.send('Hallo Server');
7  };
8
9  wsConnection.onerror = function(error) {
10   console.log('Es ist ein Fehler aufgetreten: ' + error);
11 };
12
13 wsConnection.onmessage = function(event) {
14   var tweet = JSON.parse(event.data);
```

```
15   tweets.push(tweet);
16 };
```

Listing 3.6: Erstellung einer WebSockets-Verbindung

Es wird in Abbildung 3.6 zunächst eine Verbindung zum Server hergestellt. Dafür muss ein Objekt vom Typ *WebSocket* erstellt werden. Dem Konstruktor wird dafür einen URI übergeben. Das URI-Schema ist analog zum HTTP-Protokoll mit *ws* und *wss* für eine verschlüsselte Verbindung anzugeben.

Der Event-Listener *onopen* wird ausgeführt sobald eine Verbindung zum Server hergestellt wurde. Mittels der Methode *send* können Nachrichten an den Server gesendet werden. Im gezeigten Beispiel sendet der Client nach einem erfolgreichen Aufbau einer Verbindung die Nachricht *Hallo Server*.

*onerror* wird in einem Fehlerfall ausgeführt. Der Funktion wird der ausgegebene Fehler übergeben.

Die Funktion *onmessage* wird verwendet, wenn der Browser eine Nachricht vom Server erhält. Es wird das Event als Parameter übergeben. Mittels *Event.data* kann auf die übertragenen Daten zugegriffen werden. Im gezeigten Beispiel wird ein Tweet als JSON gesendet. Anschließend wird dieser einem Array hinzugefügt.

### 3.1.4 socket.io

Die gezeigten Techniken bieten alle Möglichkeiten, um eine Realtime Browser-Server-Kommunikation zu realisieren. Dabei ist jedoch zu beachten, dass das Long-Polling als Workaround gedacht ist und nur als Fallback für ältere Browser zu verwenden ist.

Das Long-Polling wird von allen aktuell verwendeten Browsern unterstützt. WebSockets und Server-Sent-Events stehen hingegen nicht allen Browsern zur Verfügung. WebSockets wird ab der *Internet Explorer* Version 10, *Firefox* Version 6 und *Chrome* Version 14 unterstützt. Server-Sent-Events wird von keiner veröffentlichen Internet Explorer Version unterstützt. In Firefox ab der Version 6 und in Chrome ebenfalls ab Version 6 [59].

Da es beim Programmieren des Smart Clients sehr aufwändig ist, auf die Kompatibilität der Verschiedenen Browser zu achten, bietet es sich an, eine JavaScript-Bibliothek, die von der verwendeten Methode abstrahiert, zu verwenden. Ein Beispiel für eine solche ist *socket.io*.

*Socket.io* ist mit dem Großteil der aktuell verwendeten Browser kompatibel. Es ist somit unter anderem möglich, eine Verbindung, die sich wie ein WebSocket verhält, mit dem Internet Explorer ab Version 5.5 herzustellen. Dafür muss sowohl der Smart Client als auch der Server die Bibliothek verwenden. Auf Serverseite ist *socket.io* mit *Node.js* kompatibel [61]. Auf *Node.js* wird im weiteren Verlauf dieser Arbeit noch eingegangen.

```
1 var io = require('socket.io').listen(80);
2 io.sockets.on('connection', function (socket) {
3   console.log('Neuer Client verbunden');
```

```
4 });
```

Listing 3.7: socket.io in Node.js

```
1 var socket = io.connect('http://subpub.orestes.info');
2 socket.on('newtweet', function (data) {
3   tweets.push(data);
4 });
```

Listing 3.8: socket.io im Smart Client

Abbildung 3.7 zeigt eine mögliche Implementierung innerhalb eines Node.js-Servers. Mittels *require* wird die Socket.io-Bibliothek geladen. Über *listen* wird anschließend der Port, auf dem der Server auf ankommende Verbindungen hören soll, definiert. Im gezeigten Beispiel ist es somit Port 80. Mittels *io.sockets.on('connection', function() )* wird definiert, was nach einem erfolgreichen Verbindungsaufbau ausgeführt wird. Im Beispiel wird ein Hinweis in der Konsole ausgegeben.

Auf dem Client (Abbildung 3.8) wird zunächst mittels *io.connect* eine Verbindung zum Server hergestellt. Über *socket.on* wird anschließend definiert, dass der Client auf das Event mit dem Namen *newtweet* reagieren soll. Im gezeigten Beispiel werden die vom Server übermittelten Daten einem Array hinzugefügt. Der Server kann über *socket.emit('newtweet', { tweet: 'Orestes ist klasse', user: 'florian'})* Daten an alle Browser senden, die das Event *newtweet* abonniert haben.

Es ist somit möglich, die Funktionalitäten von WebSockets zu verwenden und zwar ohne auf die Kompatibilität der zu unterstützenden Browser achten zu müssen. Ob eine WebSocket-Verbindung notwendig oder die Verwendung von Server-Sent Events ausreichend ist, muss abhängig von dem zu implementierenden Use-Case entschieden werden. Für den in dieser Arbeit vorgestellten Prototypen ist zum Beispiel eine unidirektionale Verbindung vom Server ausgehend ausreichend. Es wird daher die Server-Sent Events Methode verwendet.

### 3.1.5 Event-Server

Da für jeden Client eine Verbindung aufrecht gehalten werden muss und dieses abhängig von den Besucherzahlen der jeweiligen Seite zu einer starken Belastung für den Datenbankserver werden kann, ist es sinnvoll einen separaten Server zu verwenden, der sich nur um die Verwaltung der angemeldeten Clients und das Verschicken von Events kümmert. Dafür kann zum Beispiel *Node.js* mit der *socket.io* Bibliothek verwendet werden.

Orestes muss dafür einen Mechanismus implementieren, der dem Event-Server mitteilt, wenn sich Datensätze geändert haben. Es ist nicht jede Änderung oder jedes neues Objekt für alle Clients interessant. So muss zum Beispiel eine Änderung am Benutzerkonto nicht allen Browsern mitgeteilt werden. Daher sollte die Konfigurationsseite von Orestes eine Möglichkeit bieten, Events zu definieren. Ein solches Event besteht aus einem Namen, einem Datenbankquery und aus einer Option, die angibt, ob das neue bzw.

geänderte Objekt oder nur eine Referenz gesendet wird. Durch das Query wird bestimmt, welche Änderungen oder neue Objekte beachtet werden. Tritt eine Änderung an einem definierten Event auf, teilt Orestes dies dem Event-Server mit. Dieser schickt abhängig von dem Event entweder das geänderte bzw. neue Objekt an alle Clients, die sich auf das Event registriert haben, oder lediglich eine Referenz. Eine Referenz ist sinnvoll, wenn es sich um größere Datenmengen handelt. Der Client kann somit selbst entscheiden, ob er diese laden möchte oder nicht. Des Weiteren ist die Angabe einer fortlaufenden Sequenznummer sinnvoll. Somit kann der Client dem Server mitteilen, welche Nachricht er als letztes erhalten hat. Diese Funktionalität ist im Besonderen bei einem Verbindungsabbruch vom großen Nutzen, da der Browser die Sequenznummer vor dem Abbruch gespeichert hat und beim Wiederaufbau der Verbindung diese übermitteln kann. Der Server kann anschließend alle Events übermitteln, die seit dem Verbindungsabbruch ausgeführt wurden.

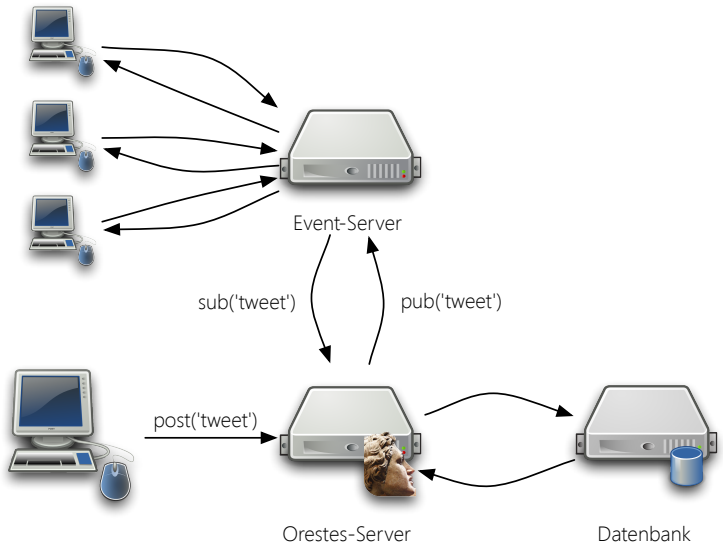


Abbildung 3.3: Erweiterung der Orestes-Infrastruktur um einen Event-Server

Abbildung 3.3 zeigt den Aufbau der Orestes-Infrastruktur mit einem Event-Server. In dem gezeigten Beispiel wurde ein Event mit dem Namen *tweet* konfiguriert. Dieses beinhaltet ein Query, das auf neue Einträge in der *tweets*-Tabelle reagiert. Es registrieren (*subscribe (sub)*) sich an dem Event-Server drei Clients für dieses Event. Der Event-Server muss sich dafür am Orestes-Server für das Event *tweet* registrieren. Ein weiterer Client schreibt anschließend einen neuen Eintrag mittels *post('tweet')* in die zugehörige Tabelle. Der Orestes-Server schreibt den Eintrag in die Datenbank und setzt anschließend dem Event-Server darüber in Kenntnis *(publish (pub))*. Dieser teilt das Ereignis wiederum den registrieren Browsern mit. Der zugrundeliegende Mechanismus wird auch als *pubsub* bezeichnet.

### 3.1.6 Optimierung

Wie bereits beschrieben, ist es wichtig, dass ein Smart Client möglichst schnell lädt und reagiert [41]. In der Entwicklung eines Smart Client wird der Code im Normalfall über mehrere JavaScript-Dateien aufgeteilt, um ihn übersichtlicher und verständlicher zu gestalten. Dadurch kann, abhängig von der Komplexität, eine große Anzahl von Dateien entstehen. Des Weiteren werden in der Regel verschiedenste JavaScript-Bibliotheken wie zum Beispiel *jQuery* und *Underscore.js* verwendet. Es ist möglich, *jQuery* durch Plugins, welche ebenfalls als eigenständige Datei geliefert werden, beliebig zu erweitern.

Ebenso müssen zur Gestaltung des Smart Clients Stylesheet-Dateien (*CSS*-Dateien) geladen werden.

Diese Dateien müssen alle, bevor der Smart Client funktionsfähig ist, vom Browser geladen und ausgeführt werden. Der Browser lädt die Dateien per HTTP aufbauend auf *TCP*. Um eine TCP-Verbindung aufzubauen, muss zunächst ein *3-Way-Handshake* durchgeführt werden [70]. Da dieser, abhängig von der Verbindungslatenz, welche im Besonderen bei der Verwendung von mobilen Geräten innerhalb von Mobilnetzen sehr hoch sein kann [31], eine gewisse Zeit in Anspruch nimmt, muss die Anzahl der benötigten Anfragen, um eine Seite zu laden, möglichst klein gehalten werden.

Um diesen Problemen entgegen zu wirken, werden JavaScript- und Stylesheet-Dateien zusammengefügt. Es wird anschließend nur noch jeweils eine Datei geladen. Da es abhängig von der Komplexität des Projekts beliebig aufwendig sein kann, alle Dateien zusammen zu führen, gibt es Programme, die diese Aufgabe automatisiert übernehmen. Dazu gehören unter anderem die JavaScript-Bibliothek *RequireJS*, der in Java geschriebene und von Google entwickelte *Closure Compiler* sowie das Automatisierungstool *Grunt*, welches per Plugins für verschiedenste Aufgaben erweitert werden kann [62][27]. Im weiteren Verlauf dieses Kapitel wird auf das Werkzeug Grunt eingegangen, da es sich um das flexibelste handelt und auch für weitere Aufgaben verwendet werden kann. Zur Entwicklung des Prototypens wird ebenfalls Grunt verwendet.

Grunt ist in JavaScript geschrieben und verwendet Node.js. Es werden bereits von den Grunt-Entwicklern Plugins zur Verfügung gestellt, die für die beschriebene Aufgabe verwendet werden können [3].

Grunt wird über die *Gruntfile.js*, welches sich im Hauptverzeichnis des Projektes befinden muss, gesteuert. In dieser können Aufgaben definiert und Plugins konfiguriert werden. Des Weiteren muss eine *package.json* erstellt werden. Diese enthält ein JavaScript-Objekt mit weiteren Konfigurationsparametern. Dazu gehören zum Beispiel der Namen des Projekts und die Abhängigkeiten. Abbildung 3.9 zeigt eine solche. Es wird definiert, dass das Projekt den Namen *twt* hat. Des Weiteren wird bestimmt, dass die Module *grunt*, *grunt-contrib-concat*, *grunt-contrib-uglify* und *grunt-contrib-cssmin* benötigt werden.

```
1 {
2   "name": "twt",
3   "version": "0.0.1",
```

```
  "dependencies": {},
  "devDependencies": {
    "grunt": "~0.4.1",
    "grunt-contrib-concat": "~0.3.0",
    "grunt-contrib-uglify": "~0.2.0",
    "grunt-contrib-cssmin": "~0.6.0"
  }
}
```

Listing 3.9: Beispiel einer *package.json*

Das Module *grunt-contrib-concat* wird verwendet, um JavaScript- und CSS-Dateien zusammen zu führen. Abbildung 3.10 zeigt die Konfiguration. Es wird definiert, dass sowohl CSS- als auch JavaScript-Dateien konkateniert werden. Grunt verwendet alle JavaScript-Dateien, die sich in dem Ordner *src/js/* befinden und erstellt in dem Ordner *dest/js/* das Ergebnis der Konkatenation. Im gezeigten Beispiel hat die Datei den Namen *twt.js*, da durch die Angabe von <%= *pkg.name* %> der in der *package.json* konfigurierte Name verwendet wird. Die Konfiguration der CSS-Dateien verläuft analog dazu. Es wird lediglich der Ordner *css* anstelle von *js* verwendet.

```
grunt.initConfig({
  concat: {
    js: {
      src: 'src/js/*.js',
      dest: 'dest/js/<%= pkg.name %>.js'
    },
    css: {
      src: 'src/css/*.css',
      dest: 'dest/css/<%= pkg.name %>.css'
    }
  }
});
```

Listing 3.10: Konfiguration des *concat*-Plugins

Neben der Anzahl der Requests spielt die Größe der zu ladenden Dateien eine entscheidende Rolle. Je größer eine Datei, desto länger ist die Ladezeit. Da es sich bei JavaScript um eine Scriptsprache handelt, wird sie nicht kompiliert. Stattdessen wird der Quellcode zum Browser übertragen, damit dieser das Programm anschließend interpretieren und ausführen kann [40].

Um Quellcode lesbar und verständlich zu halten, werden Konstrukte und Formatierungen verwendet, die durch kürzere und somit platzsparende Methoden ersetzt werden können. So genannte *Minifizierer* machen sich dies zu nutzen, indem sie den Quellcode analysieren und auf möglichst große Platzersparnis hin optimieren [69].

```
angular.module('twitterApp')
  .directive('stopPropagation', function () {
```

```
3     var directiveDefinitionObject = {
4       link: function postLink(scope, iE) {
5         iE.bind('click', function(e) {
6           e.stopPropagation();
7         });
8       }
9     };
10    return directiveDefinitionObject;
11  });
```

Listing 3.11: JavaScript vor der Minifizierung

```
1 angular.module("twitterApp").directive("stopPropagation",function(){var
    n={link:function(n,t){t.bind("click",function(n){n.stopPropagation
    ()})}};return n});
```

Listing 3.12: JavaScript nach der Minifizierung

Abbildung 3.11 zeigt eine Funktion, die in dem Prototyp, der in dieser Arbeit ausgearbeitet wird, verwendet wird. Auf die genauere Funktion wird nicht eingegangen. Abbildung 3.12 zeigt die minifizierte Version des Programmauszuges. Diese wurde mit dem Werkzeug *UglifyJS2* erstellt [7]. Das Grunt Plugin *grunt-contrib-uglify* verwendet ebenfalls dieses. Das gezeigte Beispiel konnte von 306 auf 156 Zeichen komprimiert werden. Es wurde somit fast die Hälfte der Größe gespart. *UglifyJS2* hat dafür alle unnötigen Leerzeichen entfernt. Des Weiteren wurden Variablen umbenannt. Der Name der Variablen *directiveDefinitionObject* wurde unter anderen in *n* geändert. Dieses Beispiel zeigt nur einen kleinen Auszug der Optimierungsmöglichkeiten, die durchgeführt werden. Es gibt unter anderem auch die Möglichkeit, eine *unsichere* Methode zu verwenden [7]. Wird der zugehörige Parameter gesetzt, werden Optimierungen durchgeführt, die in bestimmten Fällen das Verhalten des Programms ändern können. So wird zum Beispiel *new Array(1, 2, 3)* zu *[1, 2, 3]* geändert. Dieses kann abhängig vom Kontext, zu Problemen führen. Daher sollte dieser Parameter nicht verwendet werden, da der Größengewinn nicht im Verhältnis zu den Problemen steht, die dieser Parameter auslösen kann.

Ein weiteres Beispiel ist das JavaScript-Framework *jQuery*. Dieses wird von den Entwicklern in einer komprimierten sowie unkomprimierten Version zur Verfügung gestellt. Die unkomprimierte ist *237 KB* groß. Die komprimierte benötigt hingegen lediglich *82 KB* Speicherplatz [38]. Dies wirkt sich nicht nur auf die Ladezeit, sondern auch auf den gesamten Datenverkehr, der im Normalfall bezahlt wird, aus. So hatte zum Beispiel *ard.de* im Mai 2013 80 Millionen Besucher [5]. Wird davon ausgegangen, dass alle Benutzer die JavaScript-Datei nicht zwischengespeichert haben und herunterladen, ergibt sich für die unkomprimierte Version ein Datenverbrauch von 18515,6 Gigabyte. Die komprimierte Version benötigt hingegen nur 6406,3 Gigabyte. Daraus ergibt sich eine Ersparnis von 12109,3 Gigabyte. Dieses Beispiel zeigt die Wichtigkeit der Verwendung eines Minifizierers.

Im nachfolgenden Kapitel wird auf das JavaScript-Framework *AngularJS* eingegangen.

## 3.2 AngularJS

*AngularJS* ist ein freiverfügbares und quelloffenes MVC-Frameworks, welches von *Google* entwickelt wird. Es soll durch verschiedene Mechanismen die Entwicklung von Smart Clients vereinfachen. In diesem Kapitel werden diese erörtert. Folgendes Kapitel basiert weitestgehend auf [26].

### 3.2.1 Grundlagen

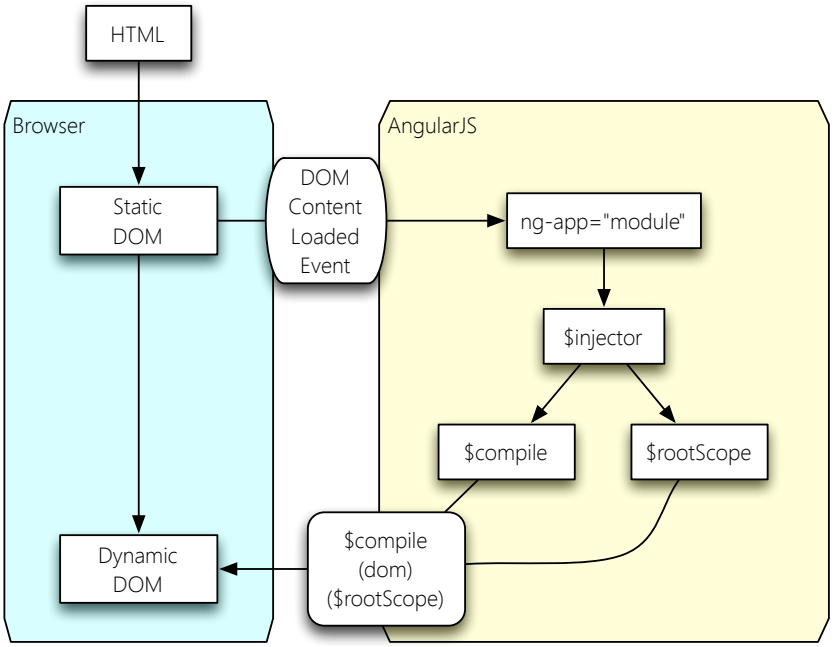


Abbildung 3.4: Ablauf des Starts von AngularJS [26]

Abbildung 3.4 zeigt den Startprozess von AngularJS. Anhand diesem werden im Folgenden die Grundkonzepte erläutert. Der Browser lädt zunächst die HTML-Seite in der das AngularJS-Script eingebunden ist und erstellt das *Document Object Model (DOM)*[2]. Während der Erstellung des DOMs wird bereits JavaScript geladen, interpretiert und ausgeführt. Da AngularJS das DOM benötigt, wird das Framework erst gestartet, wenn der Browser das Event *DOMContentLoaded* ausgibt. Das passiert sobald das HTML-Dokument geladen und eingelesen wurde. Auf das Laden von Bildern und Stylesheets wird somit nicht gewartet.

AngularJS erweitert das HTML Vokabular mittels so genannten *Directives*. Ein Directive kann bestehende HTML-Elemente wie zum Beispiel <*div*> und <*span*> erweitern oder neue Elemente erstellen. Die vordefinierten Directives haben das Präfix *ng-*. Die

[2]Das DOM stellt die HTML-Seite als Baumstruktur dar.

Verwendung von selbstdefinierten HTML-Elementen und Attributen wird häufig kritisiert, da der HTML-Code anschließend nicht mehr konform mit dem W3C-Standard ist. AngularJS bietet daher die Möglichkeit, *data-* als Präfix für Directives zu verwenden, da *data-**-Attribute dem Standard entsprechen [75].

```
1 <html ng-app="twtApp">
```

Listing 3.13: ng-app Directive

Mittels des Directives *ng-app* wird der Namensraum sowie die Grenzen der Applikation festgelegt. AngularJS beachtet nur die DOM-Knoten innerhalb des *np-app* Directives. Im Normalfall wird dieses, wie in Abbildung 3.13 zu sehen, dem *html*-Element hinzugefügt. Pro HTML-Seite kann nur ein *ng-app* verwendet werden. Sind mehrere definiert, wird lediglich das erste benutzt. Im gezeigten Beispiel wird eine Applikation Namens *twtApp* definiert.

```
1  angular.module('twtApp', [])
2    .config(function ($routeProvider, $locationProvider) {
3      $locationProvider.html5Mode(true);
4      $routeProvider
5        .when('/', {
6          templateUrl: '/views/main.html',
7          controller: 'MainCtrl'
8        })
9        .when('/tweet/', {
10         templateUrl: '/views/main.html',
11         controller: 'MainCtrl'
12       })
13       .otherwise({
14         redirectTo: '/'
15       });
16   });
```

Listing 3.14: Konfiguration der Applikation

Abbildung 3.14 zeigt die Erstellung sowie Konfiguration der Applikation. Mittels *angular.module('twtApp', [])* wird die Applikation geladen. Der erste Parameter gibt den Namen an. Dem zweiten Parameter wird ein Array übergeben. Mittels diesem können AngularJS-Erweiterungen installiert werden. Im Beispiel werden somit keine verwendet. Mittels der Methode *config* kann anschließend die Applikation konfiguriert werden. Dazu gehört unter anderem die Definition von Routen.

Der Funktion, die der config-Methode übergeben wird, kann verschiedene Provider enthalten. Für die Instanziierung von Providern ist der *$injector* verantwortlich. Dafür wird an dem zu erstellenden JavaScript-Objekt die *.toString()*-Methode aufgerufen. Der $injector analysiert den String und entscheidet anhand der Parameter des Objekts, welche Provider geladen werden müssen.

Von großer Bedeutung ist der *$routeProvider*. An diesem werden die Routen definiert. Die Methode *when* werden zwei Parameter übergeben. Der erste definiert den Namen der *URI*. Beim zweiten handelt es sich um ein *JSON*. In diesem wird festgehalten, welcher Controller und welches Template geladen werden, wenn die zugehörige Route aufgerufen wird.

Mittels *otherwise* kann definiert werden, was ausgeführt wird, wenn keine der definierten Routen zutrifft.

Wird die Applikation auf den Server mit dem Hostname *www.orestes.info* geladen. Würden die *www.orestes.info* und *www.orestes.info/tweet/* jeweils den Controller *MainCtrl* und das Template in *main.html* in dem Ordner *views* aufrufen. Wird eine nicht definierte URI aufgerufen, wird auf *www.orestes.info* weitergeleitet.

Über den *$locationProvider* können weiter Einstellungen durchgeführt werden. Mittels *html5Mode(true)* wird definiert, dass HTML5-URLs und keine *Hashbangs* verwendet werden. Auf die unterschiedlichen URL-Arten wird im Kapitel 5 näher eingegangen.

Neben den Providern zur Konfiguration der Applikation erstellt der $injector noch den $rootScope sowie den *$compile*-Service.

Der $compile-Service analysiert den DOM und führt anschließend gefundenen Directives aus und bindet Daten des Models mittels des so genannten *scopes* an die View.

### 3.2.2 *Hello-World*-Beispiel

Wie in Abbildung 3.14 gezeigt, wird mittels Routen definiert, welche Controller und Views geladen werden. Mittels des Directives *ng-view* wird festgelegt, an welche Stelle des DOMs die Views geladen werden.

```
1  <html ng-app="hello">
2    <head>
3      <link rel="stylesheet" href="/styles/main.css">
4      <title>Hello World</title>
5    </head>
6    <body>
7      <div class="container" ng-view></div>
8
9      <script src="/scripts/angular.js"></script>
10     <script src="/scripts/app.js"></script>
11     <script src="/scripts/controllers/main.js"></script>
12   </body>
13 </html>
```

Listing 3.15: ng-app Directive

Abbildung 3.15 zeigt die *index.html* der Beispielapplikation. Im gezeigten Beispiel werden somit alle Views innerhalb des *div*-Elements mit der Klasse *container* eingefügt.

Über die Script-Elemente werden die benötigten JavaScript-Dateien geladen. Die Datei *app.js* enthält eine Konfiguration wie in Abbildung 3.14 gezeigt. Die verwendeten Con-

troller müssen ebenfalls geladen werden.

```
angular.module('hello')
  .controller('MainCtrl', function ($scope) {
    $scope.hello = 'Hello World!';
  });
```

Listing 3.16: Beispiel des Controllers *MainCtrl*

Abbildung 3.16 zeigt die Definition eins Controllers. Mittels *angular.module('hello')* wird die Referenz auf die jeweilige AngularJS-Applikation geladen. An ihr wird mittels der Methode *controller* ein solcher erstellt. Der erste Parameter gibt dabei den Namen an. Dem zweiten wird eine Funktion übergeben. Diese wird beim Aufruf des Controllers ausgeführt. Mittels Funktionsparametern kann definiert werden, welche Module der $injector der Funktion zur Verfügung stellt. Im gezeigten Beispiel wird lediglich der *$scope* benötigt. An diesem wird in der Funktion das Attribut *hello* mit dem Inhalt *Hello World!* definiert.

```
<div class="row">
  <strong>{{ hello }}</strong>
</div>
```

Listing 3.17: Beispiel der View *main.html*

Listing 3.17 zeigt die zugehörige *main.html*. Mittels *{{ hello }}* wird das Attribut *hello* des $scope-Objektes ausgegeben.

```
$routeProvider
  .when('/', {
    templateUrl: '/views/main.html',
    controller: 'MainCtrl'
  });
```

Listing 3.18: Route zum Aufruf der *main.html* mittels *MainCtrl*

Im Codeausschnit 3.18 ist die Route definiert, die verwendet wird, um die View *main.html* mittels Controller *MainCtrl* aufzurufen.

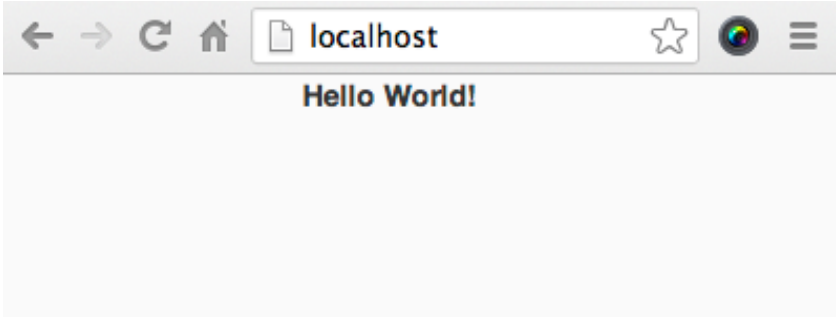


Abbildung 3.5: *Hello World*-Beispiel in AngularJS

Abbildung 3.5 zeigt die Ausgabe des *Hello World*-Beispiels. Im Controller wird am *$scope* das Attribut *hello* auf den String *Hello World!* gesetzt. Dieses Attribut wird mittels *{{ hello }}* ausgegeben.

## 3.3 Backbone.js

Bei Backbone.js handelt es sich um ein quelloffenes Framework zur Erstellung von interaktiven Webapplikationen. Es bezeichnet sich als *MV*-Framework.

### 3.3.1 Grundlagen

Backbone.js besteht aus fünf Arten von Elementen [46][6]:

**Models:** Bei einer Model-Definition handelt es sich um eine Klasse, in der Attribute, Getter, Setter und beliebige weitere Funktionen definiert werden.

**Views:** Eine View entspricht nicht der View in einem klassischen MVC-Framework. Eine View in Backbone.js beinhaltet kein Markup. Es wird der Einstiegspunkt der View im HTML und Events definiert. Des Weiteren können beliebige Funktionen erstellt werden.

**Events:** In Views können beliebige Events definiert werden. Diese beinhalten den Namen des Events (z.B. *keypress*), das HTML-Element, an dem das Event registriert wird, sowie den Namen der Funktion, die ausgeführt wird.

**Collections:** Eine Collection ist ein geordnetes Set, das Objekte eines definierten Models enthält. Es können Objekte hinzugefügt und entfernt werden. Des Weiteren ist die Angabe einer URL zu einer *REST-API* möglich. Die Collection übernimmt damit das Speichern und Laden der Objekte aus der Datenbank.

**Routers:** In Backbone.js können ebenfalls Routen definiert werden. Dies ist optional und nicht zwingend erforderlich. Werden Routen verwendet, erzeugt Backbone.js in den Standardeinstellungen Hashbang-URLs. Die Umstellung auf HTML5-URLs ist ebenso möglich.

Eine Templateengine wird nicht zur Verfügung gestellt. Die Bibliothek *Underscore.js* wird von Backebone.js vorausgesetzt. Diese beinhaltet eine Templateengine. Es bietet sich daher an, diese zu verwenden. Es ist ebenfalls möglich, sie gegen eine beliebige andere wie zum Beispiel *Moustache* auszutauschen.

### 3.3.2 Hello-World-Beispiel

Im Codeauszug 3.19 wird die *index.html* der Hello-World-Anwendung in Backbone.js gezeigt. Es wird ein *div*-Element mit der *ID view* erstellt. In dieses wird der von Backbone.js erstellte Inhalt eingefügt.

```
<html>
<head>
  <link rel="stylesheet" href="/styles/main.css">
  <title>Hello World</title>
</head>
<body>
  <div class="container" id="view">
    <strong id="hello"></strong>
  </div>
  <script src="/app.js" type="text/javascript"></script>
</body>
</html>
```

Listing 3.19: Die *index.html* der Hello-World-Anwendung in Backbone.js

In Abbildung 3.20 wird die Definition einer View gezeigt. Über das Attribut *el* wird der Einstiegspunkt der View innerhalb des DOMs festgelegt. Die Funktion *initialize* wird beim Erstellen des *AppView*-Objektes ausgeführt. Diese ruft wiederum die Funktion *render* auf. In dieser wird ein Model vom Typ *Hello* erstellt. Die Definition des Models wird im weiteren Verlauf beschrieben. Dem Model wird ein JSON mit dem Attribut *txt* und dem Wert *Hello World* mitgegeben. Über das Attribut *el* wird die *ID* des HTML-Elementes, welchen den Einstiegspunkt festsetzt, definiert. Im gezeigten Beispiel hat das Element die *ID view*. Die # dient als Selektor für *IDs*. Backbone.js kapselt das DOM-Element in ein jQuery-Element. Mittels der Funktion *find* wird das Element mit der *ID hello* geladen und der Inhalt des *hello*-Objektes eingefügt.

```
var AppView = Backbone.View.extend({
  el: '#view',
  initialize: function(){
    this.render();
  },
  render: function(){
    var hello = new Hello({txt: 'Hello World'});
    this.$el.find('#hello').append(hello.toJSON().txt);
  }
});
var appView = new AppView();
```

Listing 3.20: Definition einer View in Backbone.js

In Listing 3.21 wird das Model *Hello* definiert. Es handelt sich dabei um ein Objekt, das von der Klasse *Backbone.Model* erbt. Mittels dem JSON *defaults* können Standardwerte für Attribute definiert werden.

```
var Hello = Backbone.Model.extend({
  defaults: {
    txt: 'Hello World'
  }
```

```
5 });
```

Listing 3.21: Definition eines Models in Backbone.js

Im Folgenden wird die Definition und Verwendung einer Collection gezeigt. Diese ist nicht Bestandteil der Hello-World-Anwendung. Es werden drei Objekte vom Typ *Hello* erstellt. Neben dem vordefinierten Attribut *txt* wird noch eine *id* festgelegt. Mittels dieser kann ein Objekt aus der Collection geladen werden. Die erstellten Objekte werden der Collection hinzugefügt. In der Definition der Collection wird mittels *model* definiert, welche Objekttyp erlaubt ist. Über das Attribut *url* wird die zugehörige Ressource der REST-API der Datenbank festgelegt.

```
1 var HelloList = Backbone.Collection.extend({
2   model: helloApp.Hello,
3   url: '/db/hellos'
4 });
5 var hello1 = new Hello({txt: 'Hello World 1', id: 1});
6 var hello2 = new Hello({txt: 'Hello World 2', id: 2});
7 var hello3 = new Hello({txt: 'Hello World 3', id: 3});
8 var helloList = new HelloList([hello1, hello2, hello3]);
```

Listing 3.22: Definition und Verwendung einer Collection in Backbone.js

In Listing 3.23 wird dargestellt, wie in Backbone.js eine Route definiert wird. Es wird ein Objekt, welches vom Typ *Backebone.Router* erbt, erstellt. In dessen Definition werden die Routen festgelegt. Die URL *www.orestes.info/hellos/2* trifft auf die im Beispiel gezeigte Route zu. Wird diese geladen, wird die Funktion (Zeile 7) aufgerufen. Diese lädt das Objekt aus der Collection.

```
1 var Router = Backbone.Router.extend({
2   routes: {
3       "hellos/:id": "getHello",
4   }
5 });
6 var helloRouter = new Router();
7 helloRouter.on('route:getHello', function (id) {
8   console.log(helloList.get(id));
9 });
```

Listing 3.23: Definition einer Route in Backbone.js

## 3.4 Ember.js

Ember.js ist ebenfalls ein quelloffenes Framework zur Entwicklung von Smart Clients.

### 3.4.1 Grundlagen

Das Grundkonzept von Ember.js besteht aus den Folgenden Elementen [16].

**Templates:** Ember.js verwendet als Templateengine *Handlebars*. Die Templates können als Views betrachtet werden. Auf diese wird in der *Hello-World*-Anwendung eingegangen.

**Router:** Mittels Router wird definiert, welche Kombination aus Route, Template und Controller beim Aufruf einer URL verwendet wird.

**Components:** Sind ähnlich zu den in AngularJS definierbaren *Directives*. Es handelt sich um selbsterstellte HTML-Elemente. Diese können an beliebigen Stellen der Applikation wiederverwendet werden.

**Models** Die Definition eines Models in Ember.js ist analog zu der in Backbone.js. Es werden Klassen mit freidefinierbaren Eigenschaften erstellt.

**Route:** Innerhalb einer Route werden Models erstellt, bearbeitet und verwaltet. Eine Route hält somit den Zustand des jeweiligen Models.

**Controller:** In einem Controller wird der momentane Zustand der gesamten Applikation gehalten.

Der Unterschied zwischen Route und Controller ist auf Konzeptebene. In einer Route werden alle Persistenzdaten, also die Objekte, die aus der Datenbank geladen werden, gehalten und verwaltet. Ein Controller hingegen verwaltet den momentanen Zustand der Applikation. Mit der Beispielanwendung eines Blogs lässt sich der Unterschied zeigen. Dieser stellt die Anmeldung für Administratoren zur Verfügung. Administratoren können Blogeinträge bearbeiten. Die Route verwaltet dafür die darzustellenden Blogeinträge. Der Controller hingegen ist dafür zuständig festzustellen, ob der Benutzer angemeldet ist und ausreichend Rechte für die Bearbeitung besitzt. Sowohl Controller als auch Route nehmen Einfluss die aktuelle View.

### 3.4.2 Hello-World-Anwendung

Abbildung 3.24 zeigt die *index.html* zur Erstellung der Hello-World-Anwendung in Ember.js. Dabei ist im Besonderen die Angabe der Templates interessant. Diese werden innerhalb eines *script*-Elementes definiert. Der Typ des Elements wird auf *text/x-handlebars* gesetzt. Daran erkennt Ember.js, dass es sich beim Inhalt um ein Template handelt.

Die erste Angabe eines Templates (Zeile 8 bis 12) wird bei jedem Aufruf geladen. Es handelt sich dabei um den Container, in dem alle weiteren Templates eingefügt werden. Dies ist daran zu erkennen, dass das Attribut *data-template-name* nicht definiert ist. Des Weiteren ist mittels *{{ outlet }}* die Position zum Einfügen angegeben.

Die zweite Templatedefinition hat den Namen *hello*. Dieses wird beim Ausführen der zugehörigen Route geladen. Über die Variable *model* werden die Models zwischen View und Route verknüpft.

```
<html>
  <head>
    <meta charset="utf-8">
    <title>Hello World!</title>
    <link rel="stylesheet" href="css/style.css">
  </head>
  <body>
    <script type="text/x-handlebars">
      <div class="container">
        {{outlet}}
      </div>
    </script>
    <script type="text/x-handlebars" data-template-name="hello">
      <div class="row">
        <strong>{{ model.txt }}</strong>
      </div>
    </script>
    <script src="js/libs/jquery-1.9.1.js"></script>
    <script src="js/libs/handlebars-1.0.0.js"></script>
    <script src="js/libs/ember-1.0.0-rc.7.js"></script>
    <script src="js/app.js"></script>
  </body>
</html>
```

Listing 3.24: Die *index.html* der Hello-World-Anwedung in Ember.js

In Listing 3.25 ist der restliche Code zur Erstellung der Hello-World-Application zu sehen. Es wird zunächst eine Anwendung erstellt. An dieser wird über den Router definiert, welche Pfade aufgerufen werden können. Dafür wird die Methode *this.route* verwendet. Dieser können zwei Parameter übergeben werden. Der erste ist der Name des Pfades und bestimmt den Namen der Route, des Controllers sowie des Templates. Mittels dem zweiten Parameter kann ein Pfad angegeben werden. Wird dieser nicht definiert, wird impliziert, dass der aus Pfad */hello* besteht.

Anschließend wird die zugehörige Route definiert. An dieser wird ein Model erstellt, welches das Attribut *txt* beinhaltet. Dieses wird im Template *hello* ausgegeben.

```
App = Ember.Application.create();

App.Router.map(function() {
  this.route('hello')
});

App.HelloRoute = Ember.Route.extend({
  model: function() {
    return {
      txt: 'Hello World!'
    };
```

```
12    }
13 });
```

Listing 3.25: Hello-World-Application in Ember.js

## 3.5 Vergleich der Frameworks

An einem simplen Beispiel wie die *Hello-World*-Anwendung lassen sich bereits große Unterschiede erkennen. Backebone.js stellt keine automatisierte Verknüpfung zwischen View (Controller) und Template (View) zur Verfügung. Es muss entweder eine separate Templateengine verwendet werden oder die zu ändernden Elemente müssen per JavaScript geladen und bearbeitet werden. Des Weiteren implementiert Backbone.js kein bidirektionales *Data-Binding*[3]. Dieses wird von Ember.js und AngularJS zur Verfügung gestellt. Es hat sich gezeigt, dass Backbone.js mehr als Bibliothek zu sehen ist, die innerhalb einer bestehenden Webseite verwendet wird, um die Implementierung von Benutzerinteraktionen zu vereinfachen. Als Framework für einen Smart Client ist es daher nicht optimal geeignet.

Sowohl Ember.js als auch AngularJS eigenen sich für Verwendung als Framework. In Ember.js erben Models von einer vordefinierten Model-Klasse [16]. In AngularJS werden keine Vorgaben gemacht [26]. Da Orestes bereits eine JavaScript-Bibliothek zur Erstellung von Objekten zur Verfügung stellt, muss diese bei der Verwendung von Ember.js dementsprechend angepasst werden. Dies ist bei AngularJS nicht der Fall. Daher wird AngularJS für die Verwendung mit Orestes empfohlen und im weiteren Verlauf dieser Arbeit verwendet.

## 3.6 Fazit

In diesem Kapitel wurde gezeigt, dass bereits Technologien zur Verfügung stehen, um Smart Clients aufbauend auf einer 2-Tier-Architektur für die Verwendung im Browser zu Entwickeln. Im Besonderen kann durch die Verwendung eines Frameworks die Entwicklung erheblich vereinfacht und beschleunigt werden.

Im nachfolgenden Kapiteln werden Sicherheitsfragen diskutiert, die sich bei der Verwendung einer 2-Tier-Architektur stellen.

[3] Das bidirektionale *Data-Binding* sorgt dafür, dass das eine Änderung am Model in der View oder im Controller der jeweiligen anderen Instanz automatisiert übermittelt wird.

# 4 Sicherheit

Das Thema Sicherheit hat in einer 2-Tier-Architektur eine besondere Bedeutung, da die Schnittstelle zur Datenbank öffentlich verfügbar ist. In einer klassischen 3-Tier-Architektur hat lediglich der Applikationsserver Zugriff auf diese und kann somit kontrollieren, wie weit ein Benutzer mit der Datenbank kommunizieren kann.

Es müssen daher besondere Anforderungen an die verwendete Datenbank gestellt werden. Im Folgenden werden diese Anforderungen erörtert.

## 4.1 Grundlagen

In der klassischen Webentwicklung übernimmt der Applikationsserver den größten Teil der Sicherheitsvorkehrungen. Dazu gehören die Validierung von Benutzereingaben. Beispiele dafür sind die Verwendung eines Kontaktformulars und die Aufgabe einer Bestellung in einem Online-Shop. Des Weiteren übernimmt der Applikationsserver die Zugangskontrolle zu Bereichen, die nur von bestimmten Benutzern betreten werden dürfen, wie zum Beispiel ein privates Benutzerkonto, welches aktuelle Bestellungen enthält. Es werden anhand von privaten Daten, die nicht öffentlich zur Verfügung gestellt werden dürfen, Elemente der Webseite generiert. Vorstellbar ist eine Logik, die anhand von Einkaufspreisen Produktvorschläge generiert. Diese Logik kann nicht im Browser ausgeführt werden, da ansonsten die Einkaufspreise öffentlich zugänglich wären. Es ist somit nicht komplett möglich, alle Berechnung vom Client übernehmen zu lassen.

Die für eine 2-Tier-Architektur verwendete Datenbank muss Lösungen zur Verfügung stellen, die diese Probleme behebt. Die NoSQL Datenbank *Apache CouchDB*, welche seit 2005 als freie Software entwickelt wird und zur Kategorie der dokumentenbasierten Datenbanken zählt, bietet einige Möglichkeiten, um diese Probleme zu lösen [15][37]. Innerhalb einer CouchDB-Instanz lassen sich multiple Datenbanken erstellen. CouchDB stellt eine REST-API zur Verfügung. Benutzer können sich über diese registrieren und authentifizieren. Des Weiteren kann der CouchDB-Administrator Rechte für Datenbanken vergeben. Es ist möglich, Administratoren (*Admin*) sowie Mitglieder (*Member*) für einzelne Datenbanken zu definieren. Wird kein Mitglied definiert, ist die Datenbank öffentlich verfügbar. Ansonsten können nur die jeweiligen Mitglieder und Administratoren auf diese zugreifen. Eine Lesekontrolle ist somit nur auf Datenbankebene möglich. Schreibzugriffe können vor dem Speichern überprüft werden. Innerhalb von so genannten *design*-Dokumenten können Funktionen, die auf den Daten der jeweiligen Datenbank ausgeführt werden, definiert werden. So wird zum Beispiel die Funktion *validate_doc_update*

vor jedem Speichervorgang ausgeführt. Mittels dieser ist es möglich, das zu speichernde Dokument zu prüfen und die Speicherung gegebenenfalls abzulehnen. So kann zum Beispiel geprüft werden, ob eine Nachricht eine bestimmte Länge nicht überschreitet. Es ist nicht möglich, Leseanfragen auf diese Art zu validieren.

Durch die gezeigten Funktionen geht CouchDB bereits erste Schritte, um Smart Clients auf einer 2-Tier-Architektur zu realisieren. Für den in dieser Arbeit vorgestellten Use-Case fehlen allerdings bereits Funktionalitäten. Es ist unter anderem nicht möglich, dass ein Benutzer seine *Tweets* nur für bestimmte Personen freigibt, da eine Lesekontrolle lediglich über die Verwaltung von Administratoren und Mitglieder der einzelnen Datenbank möglich ist. Die naheliegende Lösung, dass jeder Benutzer des Smart Clients eine Datenbank erhält und Administrator dieser ist, damit er beliebige Mitglieder hinzufügen kann, ist möglich, aber nicht produktiv einsetzbar, da ein Administrator auch *design*-Dokumente ändern kann und somit die Validierung von Schreiboperationen entfernen kann.

Im Nachfolgenden werden Funktionalitäten vorgestellt, die eine Datenbank zur Verfügung stellen muss, damit ein Smart Client aufbauend auf ihrer möglich ist.

## 4.2 Cross-Origin Resource Sharing

Browser erlauben es aufgrund der so genannten *Same-Origin-Policy (SOP)* nicht, dass *XMLHttpRequests*[1] zwischen unterschiedlichen Servern durchgeführt werden [48][76]. Es ist somit zum Beispiel nicht möglich, dass das JavaScript der Web-Seite *www.uni-hamburg.de* Daten von dem Server *www.hamburg.de* lädt. Folgende Tabelle (4.1) zeigt, wann die SOP erfüllt bzw. verletzt wird. Als Ausgangsdomain (Origin) wird dabei *http://uni-hamburg.de* verwendet.

| Zieldomain | Erlaubt | Grund |
|---|---|---|
| http://uni-hamburg.de/s/i.json | Ja | Gleiche Domain, anderer Pfad |
| http://inf.uni-hamburg.de/i.json | Nein | Unterschiedliche Domain |
| http://uni-hamburg.de:8080/s/i.json | Nein | Anderer Port |
| https://uni-hamburg.de/s/i.json | Nein | Unterschiede in Protokoll und Port[2] |

Tabelle 4.1: Was erlaubt die *Same-Origin-Policy* [50]?

Es sind somit nur Anfragen erlaubt, welche die gleiche Domain, den gleichen Port sowie das gleiche Protokoll verwenden. Als Ausgang *Origin* wird dabei der Server betrachtet, von dem die HTML-Seite, welche das auszuführende Script enthält, geladen wird.

Bei der *Same Origin Policy* handelt es sich um ein Sicherheitskonzept, welches eingeführt wurde, damit innerhalb eines Kontexts Daten nicht in einen anderen Kontext trans-

[1] *XMLHttpRequest (XHR)* ist eine API, die es erlaubt, Daten über HTTP aus JavaScript-Programmen heraus zu laden [51].

[2] HTTPS verwendet als Standardport den Port 443 und HTTP Port 80 [30].

feriert werden können. So ist es zum Beispiel erwünscht, dass Online-Banking-Informationen nur der Seite, auf der die Daten angegeben werden, zur Verfügung stehen.

Die Anforderungen haben sich in den letzten Jahren geändert. Es werden immer häufiger Daten asynchron per JavaScript nachgeladen (*AJAX*). Dabei ist gegebenenfalls gewünscht, dass Daten von Webservices geladen werden, die sich nicht auf dem Origin-Server befinden. Um diesen Anforderungen gerecht zu werden, wurde das *Cross-Origin Resource Sharing (CORS)* eingeführt.

CORS ermöglicht einem externen Server zu bestimmen, welche Origin-Servern der Zugriff gestattet ist. Der Mechanismus wird vollständig ab Firefox 3.5, Chrome 3 sowie Internet Explorer 10 unterstützt.

Folgenden Beispiele gehen davon aus, dass die HTML-Seite, welche das Script, was für das Laden der Daten zuständig ist, beinhaltet, auf dem Server *http://www.orestes.info* (Origin) zur Verfügung steht. Die Daten werden von dem Remote-Server *http://db.orestes.info* geladen.

Browser, die CORS unterstützen, senden bei einer Anfrage, welche die SOP verletzt, im HTTP-Header das Feld *Origin* mit, welches den Origin-Server enthält (Abbildung 4.1).

```
1 Origin: http://www.orestes.info
```

Listing 4.1: Origin-Feld im HTTP-Header

Der externe Server (*http://db.orestes.info*) fügt in seiner Antwort das Feld *Access-Control-Origin* hinzu. Dieses muss den Origin-Server beinhalten (Abbildung 4.2).

```
1 Access-Control-Allow-Origin: http://www.orestes.info
```

Listing 4.2: Access-Control-Allow-Origi-Feld im HTTP-Header

Dem Browser wird somit übermittelt, auf welchem Origin-Server er sich befinden muss, damit die Daten geladen werden dürfen. Für die Werte der Felder *Access-Control-Allow-Origin* und *Origin* muss die SOP gelten. Ist dies nicht der Fall, werden die Daten vom Browser nicht geladen und es wird eine Fehlermeldung ausgegeben (Abbildung 4.3).

```
1 XMLHttpRequest cannot load http://data.orestes.info/. Origin https://
     www.orestes.info is not allowed by Access-Control-Allow-Origin.
```

Listing 4.3: Fehlermeldung bei Verletzung der *Same-Origin-Policy (SOP)*

Die Verwendung des Platzhalters * ist ebenfalls möglich. Sendet der externe Server das HTTP-Header-Feld wie in Abbildung 4.4 gezeigt, wird, unabhängig von dem Origin-Server, die Anfrage ausgeführt. Diese Methode sollte mit Bedacht gewählt werden, weil anschließend von jeder Webseite die Daten von dem Server geladen werden können.

```
1 Access-Control-Allow-Origin: *
```

Listing 4.4: Erlaubt jeden Origin-Server

Ein weitere Konfigurationsmöglichkeit bietet das HTTP-Header-Feld *Access-Control-Allow-Credentials*. Mittels diesem ist es möglich, zu erlauben, dass der Browser *Cookies*[3] für den Remote-Server speichert und diese bei einer Anfrage an diesen übermittelt. Für einen Smart Client aufbauend auf einer 2-Tier-Architektur ist es wichtig, dass dieses Feld den Wert *true* enthält (Abbildung 4.5), da die Datenbank die Authentifizierung übernimmt. Diese zugehörigen Daten werden in einem Cookie gespeichert. Damit die Datenbank den Benutzer identifizieren kann, muss somit dieser Cookie übertragen werden.

```
1 Access-Control-Allow-Credentials: true
```

Listing 4.5: Access-Control-Allow-Credentials-Feld im HTTP-Header

Ein Request mittels JavaScript wird durch ein Objekt der Klasse *XMLHttpRequest* durchgeführt. Damit dieses die zugehörigen Cookies mitsendet, muss das Attribut *withCredentials* den Wert *true* erhalten.

```
1 var xhr = new XMLHttpRequest();
2 xhr.withCredentials = true;
3 xhr.open('get', 'http://data.orestes.info/products.json', true);
4 xhr.onreadystatechange = function() {
5   if(xhr.readyState === 4) {
6     alert("Daten empfangen: " + xhr.responseText);
7   }
8 };
```

Listing 4.6: Verwendung des XMLHttpRequest-Objekts

Abbildung 4.6 zeigt die Funktionsweise des *XMLHttpRequest*-Objekts. Es wird zunächst ein neues Objekt erstellt, das *withCredentials* Attribut wird auf *true* gesetzt, damit Cookies übertragen werden, anschließend wird mittels der *open* Methode die URI geladen. Der erste Parameter gibt die HTTP-Request-Methode an. Der zweite die URI und über den dritten Parameter wird angegeben, ob die Funktion asynchron ausgeführt wird. Per *onreadystatechange* kann eine Funktion angegeben werden, die ausgeführt wird sobald sich der Status der Verbindung geändert hat. Hat das Attribut *readyState* den Wert 4, wurden alle Daten empfangen. Anschließend können mittels des Attributes *responseText* die empfangenen Daten ausgegeben werden.

In Abbildung 4.1 ist der Ablauf des *Cross-Origin-Resource-Sharings* dargestellt. Der Browser lädt alle benötigten Dateien vom Fileserver. Anschließend wird der Smart Client gestartet. Dieser lädt alle weiteren benötigten Daten vom Datenbankserver. Da beide Server auf unterschiedlichen Domains laufen, wird das HTTP-Header-Feld *Origin* hinzugefügt. Der Datenbankserver sendet anschließend die Daten zum Client. Im zugehörigen HTTP-Response-Header werden die Felder *Access-Control-Allow-Credentials* und *Access-Control-Allow-Oigin* hinzugefügt.

[3] Bei einem Cookie handelt es sich um eine Textdatei, mittels der eine Web-Seite in einem Key-Value-Paar Daten innerhalb des Browser speichern kann.

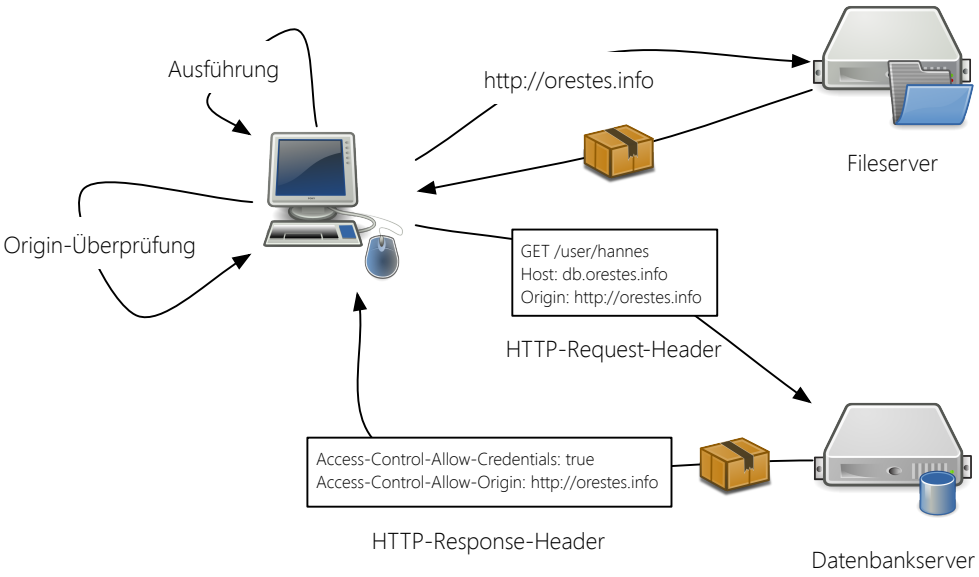


Abbildung 4.1: Ablauf des *Cross-Origin-Resource-Sharings*

## 4.3 Grundlagen der Authentifizierung

Da in einer 2-Tier-Architektur kein Applikationsserver vorhanden ist, der die Aufgaben der Authentifizierung und Autorisierung übernimmt, muss die Datenbank Mechanismen dafür zur Verfügung stellen. Im Folgenden werden Methoden vorgestellt, die aufbauend auf HTTP Möglichkeiten zur Authentifizierung und Autorisierung bieten.

### 4.3.1 Basic-Access-Authentication

Bei der so genannten *Basic-Access-Authentication* handelt es sich um die einfachste Methode zur Authentifizierung. Diese wird direkt durch das Protokoll HTTP realisiert [30]. Mittels dieser Methode kann eine Ressource durch eine Kombination aus Benutzername und Passwort geschützt werden. Handelt es sich bei einer Anfrage um eine solche, antwortet der Server mit dem Statuscode *401 - Not Authorized* und fügt dem HTTP-Header das Feld *WWW-Authenticate* hinzu. Abbildung 4.7 zeigt dieses Feld mit dem zugehörigen Wert. Mittels *realm* wird angegeben, welcher Text der Browser im Dialog anzeigt.

```
1 WWW-Authenticate:Basic realm="Gesicherte Ressource"
```

Listing 4.7: Verwendung des *WWW-Authenticate*-Feld im HTTP-Header

Nach Erhalt des *WWW-Authenticate*-Felds im HTTP-Header öffnet der Browser ein modalen Dialog (Abbildung 4.2). In diesem muss der Benutzer anschließend seine Daten eingeben.

Wurden die Benutzerdaten angegeben, encodiert der Browser diese als Base64[4] und antwortet dem Server anschließend mit dem Feld *Authorization* im HTTP-Header. Ab-

[4]Base64 ist ein Verfahren, um 8-Bit-Binärdaten in eine Codepage-unabhängige ASCII-Zeichenkette zu konvertieren.

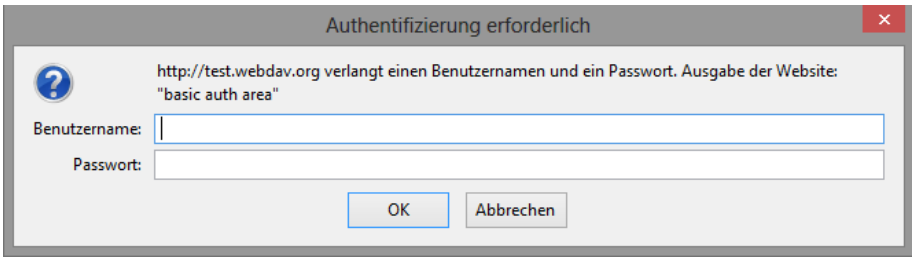


Abbildung 4.2: Modaler Dialog zur Eingabe der Benutzerdaten unter Verwendung der HTTP-Basic-Access-Authentication

bildung 4.8 zeigt dieses. Im Beispiel wird der Benutzername *hannes* und das Passwort *informatik* verwendet. Der Browser konkateniert beide Angaben zu *hannes:informatik* und encodiert diesen String anschließend in das Base64-Format.

```
1 Authorization: Basic aGFubmVzOmluZm9ybWF0aWs=
```

Listing 4.8: Verwendung des *WWW-Authenticate*-Feld im HTTP-Header

Bei der Base64-Encodierung handelt es sich nicht um ein Verschlüsselungsverfahren. Der Base64-String kann ohne Aufwand zurück in einen ASCII-String konvertiert werden und das Passwort und der Benutzername sind somit bekannt. Die Verwendung ist daher aus Sicherheitsgründen nur in Kombination mit einer verschlüsselten Verbindung wie zum Beispiel *HTTPS* zu verwenden. Des Weiteren speichern Browser den Benutzernamen und Passwort zwischen, damit die Daten nicht bei jedem Request wiederholt eingegeben müssen. Dies stellt ein weiteres Sicherheitsrisiko dar, da die Möglichkeit besteht, dass diese Daten mittels einer Schadsoftware ausgelesen werden.

Um diesen Schwachstellen entgegen zu wirken, wurde das *Digest-Access-Authentication*-Verfahren standardisiert. Diese Methode verschlüsselt die Benutzerdaten vor der Übertragung. Der Server sendet zu Beginn des Verfahrens eine zufällige Zeichenfolge (*Nonce*). Der Browser bildet einen MD5-Hash aus Nonce, Benutzername, Passwort, HTTP-Methode und URI. Anschließend wird der erstellte Hash, Benutzername und Nonce zurück an den Server gesendet. Dieser erstellt aus der gleichen Kombination einen MD5-Hash. Stimmen beide überein, wurde der Benutzer erfolgreich authentifiziert. Mittels dieser Methode wird das Passwort nicht im Klartext übertragen.

Durch den modalen Dialog, der bei beiden Verfahren Verwendung findet, wird verhindert, dass weitere Interaktionen mit der Webseite, bevor dieser beendet wurde, durchgeführt werden können. Das schränkt die Benutzerfreundlichkeit ein. Ein weiterer Nachteil ist, dass die Zeit, in dem der Benutzer als authentifiziert gilt, nicht festgelegt werden kann. Häufig ist es gewünscht, dass ein Benutzer sich erst nach einem vom Webseitenbetreiber definierten Zeitpunkt erneut authentifizieren muss. Dies ist mittels der HTTP-Authentifizierung nicht möglich.

Aus den genannten Gründen eignen sich die *Basic-Access-Authentication* und *Digest-Access-Authentication* nicht für die Verwendung innerhalb eines Smart Clients.

### 4.3.2 Cookie-Authentication

Eine verbreitete Methode ist die Verwendung einer auf Cookies basierenden Authentifizierung. Bei dieser werden die Daten, die zur Authentifizierung benötigt werden, in einem Cookie gespeichert. Somit ist es möglich, zu bestimmen, wie lange eine Authentifizierung Gültigkeit besitzt.

Die einfachste Methode ist die Übertragung der *Basic-Access-Authentication*-Methode auf einen Cookie. Diese würde zwar ermöglich, eine Authentifizierung über einen definierten Zeitpunkt aufrecht zu erhalten, hat allerdings die gleichen sicherheitsrelevanten Nachteile. Die Verwendung der *Digest-Access-Authentication*-Methode ist ebenfalls möglich. Der Nachteil dabei ist, dass Browser aktuelle keine kryptographischen Funktionen zur Verfügung stellen[5]. Das verwendete Hash-Verfahren muss somit eigenständig in JavaScript implementiert werden.

Ein weiterer Nachteil ist, dass für jeden Benutzer die Verwendeten Nonce für die Dauer der Authentifizierung gespeichert werden muss. Häufig verwenden Anwender unterschiedliche Geräte (Notebook, Desktop-PC, Handy, etc.). Auf jedem Gerät wird die Authentifizierung separat ausgeführt. Dies bedeutet, dass für jedes eine Nonce gespeichert werden muss. In einem verteilten System, wie es bei Orestes der Fall ist, müssen diese Daten jedem Server zur Verfügung stehen.

#### Cookie-Authentication nach Fu et al.

Fu et al. schlagen daher in ihrer Ausarbeitung *Dos and Don'ts of Client Authentication on the Web* eine zustandslose Methode vor [22]. Der Benutzer authentifiziert sich mit seinem Benutzernamen und Passwort am Server. Die Übertragung dieser Daten muss, um *Man-In-The-Middle*-Angriffe zu verhindern, verschlüsselt durchgeführt werden. Dafür wird HTTPS verwendet. Wurden die Daten vom Server positiv überprüft, bildet er einen Authentifizierungstoken. Dabei handelt es sich um eine Zeichenfolge, mittels der geprüft werden kann, ob der Benutzer bereits erfolgreich authentifiziert wurde. Der verwendete String besteht aus Benutzername, Verfallsdatum sowie einem *Keyed-Hash Message Authentication Code (HMAC)*[6], der aus Benutzername und Verfallsdatum gebildet wird:

$$exp = t \,\&\, user = s \,\&\, digest = HMAC_k(exp = t \,\&\, user = s)$$

Abbildung 4.3: Authentifizierungstoken nach Fu et al [22]

Durch die Verwendung des HMAC ist sichergestellt, dass der Benutzer vom Server authentifiziert wurde, da nur diesem der geheime Schlüssel, mit dem der HMAC erstellt

[5] Das W3C befindet sich zurzeit in der Phase der Standardisierung einer API, die kryptographische Verfahren zur Verfügung stellt. Diese wird von bisher keinem Browser unterstützt.

[6] Bei einem *HMAC* handelt es sich um einen Mechanismus, der unter Verwendung eines geheimen Schlüssels einen Hash erstellt. Er wird verwendet, um Datenintegrität sicherzustellen.

wurde, bekannt ist. Der Browser speichert den Authentifizierungstoken in einem Cookie und sendet ihn bei allen weiteren Anfragen an den Server mit. Erhält der Server einen Token, überprüft er den Hash, indem er aus den Daten *exp* und *user* den HMAC bildet und diesen mit dem Wert von *digest* vergleicht. Abbildung 4.4 zeigt diesen Ablauf.

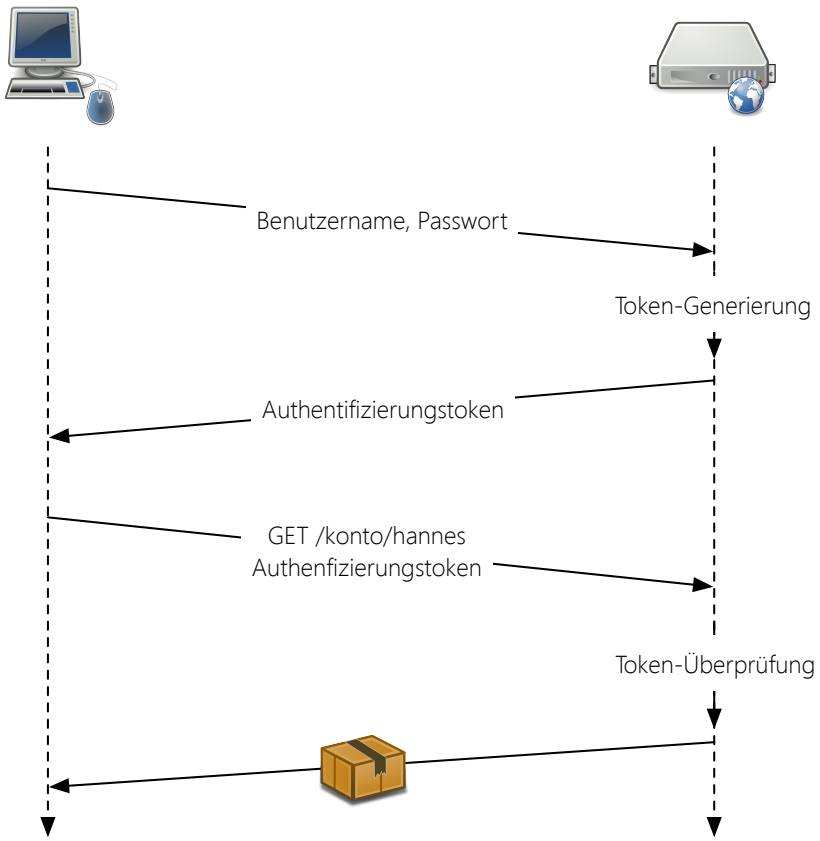


Abbildung 4.4: Cookie-Authenfizierung nach Fu et al [22]

**Erweiterung der Fu et al. Methode nach Murdoch**

Eine Cookie-Authenfizierung ist nicht gegen *Man-In-The-Middle*-Angriffe gesichert. Kann der Cookie kopiert werden, ist es möglich, sich mit diesem zu authentifizieren. Ein besonderes Risiko für diese Art der Angriffe stellen öffentliche WLAN-Netzwerke dar, weil die Kommunikation komplett mitgelesen werden kann. Da das Passwort nicht anhand des Cookies ausgelesen werden kann, ist die Gefahr, die von einem solchen Angriff ausgeht, von der Verfallszeit des Cookies abhängig. Der Angreifer kann ihn nur innerhalb dieser Zeit nutzen. Eine Möglichkeit gegen solche Angriffe ist die Verwendung von HTTPS, da die Kommunikation verschlüsselt übertragen wird und die Authentizität des Kommunikationspartners sichergestellt werden kann. Es ist daher zu empfehlen, dass der Cookie nur über HTTPS übertragen wird. Dies kann mittels der *SECURE*-Option sichergestellt werden. Ist diese im Cookie aktiviert, wird dieser vom Browser nur per HTTPS übertragen.

Ein weiteres Sicherheitsrisiko stellt der verwendete Schlüssel zur Generierung des HMACs dar. Ist dieser bekannt, kann ein Authentifizierungstoken für jeden beliebigen Benutzer erstellt werden. Die Sicherheit des Verfahrens ist somit von der Sicherheit des verwendeten Schlüssels abhängig. Durch das Sammeln von erstellten HMACs, die alle denselben Sicherheitscode verwenden, können Angriffe durchgeführt werden, um den Schlüssel zu berechnen [45].

Eine Erweiterung des vorgestellten Verfahrens verhindert, dass Authentifizierungstoken generiert werden können, wenn der HMAC-Schlüssel öffentlich wird [52]. Es wird dafür bei der Registrierung des Benutzers ein *Salt*[7] in einer Länge erstellt, damit dieser nicht mittels eines *BruteForce*[8]-Angriffes generiert werden kann (z.B. 128 Bits). Anschließend wird mittels folgender Methode aus Passwort und Salt rekursiv ein Hash erstellt:

$$a_0(Salt, Passwort) = H(Salt||Passwort)$$
$$a_x(Salt, Passwort) = H(a_{x-1}(Salt, Passwort)||Passwort)$$

Abbildung 4.5: Rekursive Erstellung des Hashes [52]

In der Datenbank wird neben Salt auch der erstellte Hash $v = H(a_n(Salt, Passwort))$ gespeichert. Der Wert $n$ gibt die Anzahl der Iterationen an. Ein praktikable und sichere Anzahl ist 256.

Nachdem alle benötigten Daten des Benutzers gespeichert wurden, kann dieser sich anmelden. Bei der Anmeldung wird mittels gespeichertem Salt und angegebenem Passwort $c = a_n(Salt, Passwort)$ berechnet. Anschließend wird geprüft $H(c) = v$ geprüft. Gilt dies nicht, ist die Kombination aus Benutzername und Passwort falsch und die Authentifizierung wird negativ beantwortet. Gilt $H(c) = v$, wird ein Cookie in Anlehnung zu Fu et al. mit folgendem Inhalt erstellt.

$$exp = t \,\&\, user = s \,\&\, auth = c \,\&\, digest = HMAC_k(exp = t \,\&\, user = s \,\&\, auth = c)$$

Abbildung 4.6: Erweiterter Authentifizierungstoken [52]

Im Vergleich zu Fu et al. enthält der Authentifizierungstoken zusätzlich den Parameter *auth*. Diese beinhaltet $a_n(Salt, Passwort)$.

Ruft der Benutzer eine geschützte Ressource auf, muss der Authentifizierungstoken geprüft werden. Dafür wird $v$, welches sich in der Datenbank befindet, benötigt.

Wird bei einem Angriff die Benutzerdatenbank kompromittiert und der HMAC-Schlüssel ausgelesen, kann der Angreifer anhand seiner gewonnen Daten keinen Cookie erstel-

[7] Bei einem *Salt* handelt es sich um zufällige Zeichenfolge, die verwendet wird, um die Entropie der Eingabe von Hashfunktionen zu erhöhen.

[8] Mittels eines *BruteForce*-Angriffs wird versucht, eine Zeichenkette durch die zufällige Erstellung von weiteren Zeichenketten zu erraten.

len, da für die Erstellung des Parameters *auth* das Passwort benötigt wird und es nicht möglich ist, dieses Anhand der gespeicherten Daten zu generieren. Durch den gewonnen HMAC kann allerdings das Verfallsdatum eines Cookies geändert werden. Daher hat der Angreifer der Datenbank nur Vorteile durch seinen Angriff, wenn er weitere Attacken auf die Benutzer durchführt.

Diese Sicherheit geht auf Kosten der Performanz, da bei jeder Überprüfung eines Authentifizierungstoken eine Datenbankanfrage durchgeführt wird. Des Weiteren muss neben der Bildung des HMAC ein weiterer Hash erstellt werden, um $H(c) = v$ überprüfen zu können. Dieses bedeutet ein erhöhter Rechenaufwand. Die Datenbankanfrage ist im Besonderen bei verteilten Systemen von Nachteil, da jeder Server, der geschützte Daten zur Verfügung stellt, auf die Benutzerdatenbank Zugriff haben muss.

**Erweiterung der Fu et al. Methode nach Liu et al.**

Liu et al. stellen eine Erweiterung vor, die weiterhin von der Sicherheit des HMAC-Schlüssels abhängig ist, aber Angriffe, die durch die Generierung einer großen Anzahl von unterschiedlichen HMACs mit gleichem Schlüssel möglich sind, verhindert [45]. Des Weiteren wird der Parameter *user* verschlüsselt, damit durch das Abgreifen eines Cookies keine Rückschlüsse auf den Benutzer gezogen werden können. Es ist ebenso möglich, in dem *user*-Parameter weitere Daten wie zum Beispiel eine Rolle (Administrator, Benutzer, Autor, etc.) zu speichern.

Es wird zunächst ein Schlüssel analog zur Fu et al. Methode erstellt. Die Variable *sk* entspricht dabei dem HMAC-Schlüssel der vorherigen Methoden.

$$k = HMAC_{sk}(exp = t \,\&\, user = s)$$

Abbildung 4.7: Erstellung des Schlüssels [45]

Mittels $k$ wird anschließend der Authentifizierungstoken gebildet.

$$exp = t \,\&\, user = (s)_k \,\&\, digest = HMAC_k(exp = t \,\&\, user = s)$$

Abbildung 4.8: Authentifizierungstoken [45]

Der HMAC, der im Cookie Verwendung findet, wird auf diese Methode nicht mit dem geheimen Schlüssel des Servers erstellt. Der Schlüssel, der zur Erstellung des Authentifizierungstoken verwendet wird, ändert sich stets, da es selbst ein HMAC ist und die Verfallszeit beinhaltet. Dabei ist zu beachten, dass $t$ nicht die Länge der Gültigkeit enthält sondern die Zeit in Sekunden seit dem 01.01.1970 bis zum Ablaufdatum.

Da sich die verwendeten Schlüssel der sich in Cookies befindlichen HMACs stets ändert, kann kein Rückschluss mehr auf den geheimen Schlüssel des Servers gezogen wer-

den.

Im Entwurf von [45] wird der HTTPS-Session-Schlüssel mit in den Authentifizierungstoken einbezogen, um *Replay*-Attacken zu verhindern. Dies ist für die Verwendung im Zusammenhang mit einem Smart Client, wie er in dieser Arbeit betrachtet wird, nicht geeignet, weil dadurch die Möglichkeit verfällt, dass die Gültigkeitsdauer bestimmt werden kann. Der Cookie ist dadurch maximal bis zum Ablauf der HTTPS-Session gültig. Ein weitere Nachteil ist, dass jeder Web-Cache im Besitz des HTTPS-Schlüssels sein muss.

Bei den Methoden von Fu et al. und Liu et al. muss jeder Server, der den Authentifizierungstoken überprüft, den geheimen Schlüssel zur Erstellung des HMACs speichern. Dies ist Problematisch in einer Architektur, wie sie von Orestes verwendet wird, da eine Vielzahl von Servern für die Auslieferung von Daten verwendet wird. Die Wahrscheinlichkeit, dass ein Schlüssel kompromittiert wird, steigt mit der Anzahl seiner Vervielfältigung.

### 4.3.3 OpenID

Bei *OpenID* handelt es sich um webbasierten Authentifizierungsprotokoll, das sich aktuell in der Version 2.0 befindet [19]. Es ermöglicht die Verwendung eines Benutzerkontos auf unterschiedlichen Seiten. Der Benutzer muss sich einmalig bei einem OpenID-Provider registrieren. Anschließend kann er sich mittels diesem bei allen Web-Seiten, welche das OpenID-Protokoll unterstützen, anmelden. Internetseiten, die eine Anmeldung per OpenID unterstützen, werden *Relying Parties* genannt.

Bei OpenID handelt es sich um ein dezentrales Protokoll. Es gibt keinen zentralen Server, der alle OpenID-Konten beherbergt. Stattdessen kann es beliebig vielen OpenID-Provider geben. Da es sich um ein offenes Protokoll handelt, sind bereits viele Implementierungen in unterschiedlichen Programmiersprachen vorhanden. Es bedarf somit nur einen geringen Aufwand, um einen OpenID-Provider zu installieren. Im Folgenden wird auf diese nicht weiter eingegangen. Es wird die Anmeldung mittels eines OpenID-Providers betrachtet.

Bei einem Anmeldeprozess mittels OpenID sind drei Parteien involviert: der Browser, die Seite, auf der die Anmeldung durchgeführt wird, (RP) sowie der OpenID-Provider (OP).

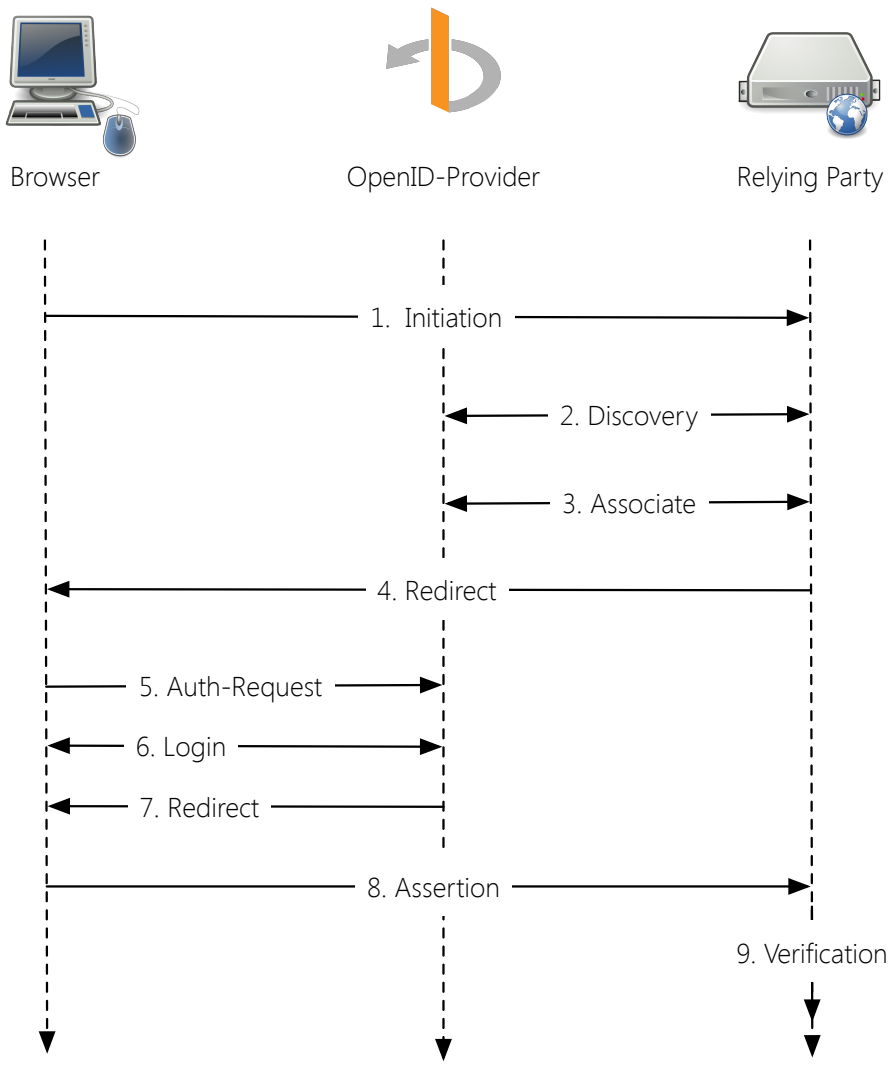


Abbildung 4.9: Ablauf der Authentifizierung des OpenID-Protokolls [32]

Abbildung 4.9 zeigt den Ablauf der Authentifizierung innerhalb des OpenID-Protokolls in der Version 2.0. Im Folgenden wird dieser näher beschrieben [56].

1. **Initiation:** Zunächst wählt der Anwender aus, welchen OpenID-Provider er verwenden möchte. Häufig werden von der Relying Party einige vorgegeben. Es kann aber auch der favorisierte Provider per URL angegeben werden.

2. **Discovery:** Wurde ein OpenID-Provider angegeben, werden Informationen zwischen der Relying Party und dem Provider ausgetauscht. Dazu gehören unter anderen die Protokollversion sowie der OpenID-Provider-Endpoint.

3. **Associate:** Sind alle Daten des Providers bekannt, wird ein *Shared Secret*[9] mittels dem *Diffie-Hellman*-Verfahren[10] ausgetauscht. Dieser dient zur Verifizierung der

[9] Ein *Shared Secret* ist ein geheimer Schlüssel, der allen Kommunikationspartnern bekannt ist. Mittels diesen können Daten symmetrisch verschlüsselt werden.

[10] Das *Diffie-Hellman*-Verfahren ermöglicht das sichere Austauschen von einem geheimen Schlüssel *(Shared Secret)* zwischen zwei Kommunikationspartnern.

weiteren Daten, die im Zuge der Authentifizierung ausgetauscht werden.

4. **Redirect:** Der Benutzer wird zum OpenID-Provider umgeleitet.

5. **Auth-Request:** Der Browser sendet nach dem Redirect eine Authentifizierungsanfrage an den OpenID-Provider.

6. **Login:** Der Anwender gibt seine Benutzerdaten an, um sich gegenüber dem OpenID-Provider zu authentifizieren. Es ist möglich, bei einem Provider mehrere Identitäten zu registrieren. Er kann somit nach der Anmeldung eine beliebige auswählen.

7. **Redirect:** Hat der Benutzer sich authentifiziert, wird er zurück an die Relying Party geleitet. Die benötigte Redirect-URL wurde beim *Disovery*-Schritt ausgetauscht.

8. **Assertion:** Der Browser leitet Daten, die ihm vom OpenID-Provider zur Verfügung gestellt wurden, an die Relying Party weiter. Dazu gehören unter anderen ein Benutzername, den Status der Authentifizierung (erfolgreich, fehlgeschlagen) sowie die Signatur, um die Integrität der Daten zu prüfen.

9. **Verification:** Die Relying Party prüft mittels Shared Secret, ob die übertragenen Daten vom OpenID-Provider ausgestellt und nicht geändert wurden.

Wurde ein Benutzer mittels OpenID authentifiziert, empfiehlt es sich, bei der erst Anmeldung anschließend ein Profil in der Datenbank der Relying Party anzulegen. Dies ist möglich, da es dieser überlassen ist, wie sie mit den erhaltenen Daten umgeht. Der OpenID-Provider ist lediglich dafür zuständig, um die Echtheit des angegebenen Benutzernamens zu bestätigen. Dies hat den Vorteil, dass neben der Verwendung von OpenID auch weitere Authentifizierungsverfahren wie zum Beispiel ein klassisches basierend auf Benutzername und Passwort benutzt werden können.

Es gibt bereits eine Vielzahl von OpenID-Providern. Einige der bekanntesten sind *Google*, *Yahoo*, *Wordpress.com* und *AOL* [19]. Dies zeigt, dass das OpenID-Protokoll sich etabliert hat und die Verwendung zu empfehlen ist, wenn der Authentifizierungsprozess möglichst einfach gehalten werden soll. Anderen Webseitenbetreiber wie zum Beispiel *Facebook* stellen selbstentwickelte Protokolle zur Verfügung [18]. Auf diese wird nicht weiter eingegangen.

Ein weiteres Protokoll, welches häufig im Zusammenhang mit OpenID genannte wird, ist das so genannte *OAuth*-Protokoll. Dabei handelt es sich nicht, wie der Name es vermuten lässt, um ein Authentifizierungsprotokoll, sondern um ein Protokoll zur Autorisierung [55]. Mittels OAuth kann der Benutzer zustimmen, dass eine Partei auf die Daten einer 2. Partei, bei der der Benutzer bereits Daten hinterlegt hat, zugreifen kann. Das Protokoll wird häufig verwendet, um bei der Registrierung Daten von zum Beispiel *Facebook* abzufragen, damit diese nicht erneut angegeben werden müssen. Der Benutzer kann dabei festlegen, welche Daten er zur Verfügung stellt.

Im nachfolgenden wird ein Authentifizierungsmechanismus, der im Rahmen dieser Arbeit entwickelt wird, vorgestellt.

## 4.4 Authentifizierung im Kontext von Orestes

Die Authentifizierungsmethoden, die im Protokoll HTTP integriert sind, eigenen sich nicht für die Verwendung, da nicht kontrolliert werden kann, wie lange die Authentifizierung gültig ist und sie nicht ausreichend flexibel sind. Da durch die Cache-Hierarchie diverse Server an der Auslieferung beteiligt sind und diese kontrollieren müssen, ob ein Client Zugriff auf die angeforderten Daten hat, bietet sich die Verwendung eines zustandslosen Authentifizierungssystems, wie das von Fu et al. vorgestellte, an, da ansonsten alle Server den aktuellen Zustand zu halten haben. Die Methode von Fu et al. sowie die Erweiterung durch Liu et al. eigenen sich dafür. Sie haben aber den großen Nachteil, dass alle beteiligten Server den geheimen Schlüssel speichern müssen. Dies stellt ein Sicherheitsrisiko dar. Die Abwandlung durch Murdoch verhindert zwar, dass beim Bekanntwerden des Schlüssels Cookies erstellt werden. Sie hat allerdings den Nachteil, dass bei jeder Überprüfung eine Datenbankanfrage durchgeführt werden muss. Sie ist daher ungeeignet für das Orestes-Protokoll, da ansonsten bei jeder Anfrage der Orestes-Server kontaktiert werden muss und damit die Vorteile der Caches verloren gehen.

Diese Arbeit schlägt aus den genannten Gründen eine Abwandlung der Fu et al. Methode vor. Der HMAC wird dabei durch ein asymmetrisches Signierungsverfahren ersetzt. Bei asymmetrischen Kryptographieverfahren erstellt jeder Kommunikationspartner zwei Schlüssel. Einen geheimen und einen öffentlichen. Wird eine Nachricht mittels des öffentlichen Schlüssels verschlüsselt, kann diese Nachricht nur mittels des privaten entschlüsselt werden (Abbildung 4.10). Dies hat den Vorteil, dass kein Shared Secret auf sicherem Wege ausgetauscht werden muss. Der öffentliche Schlüssel kann frei zugänglich sein.

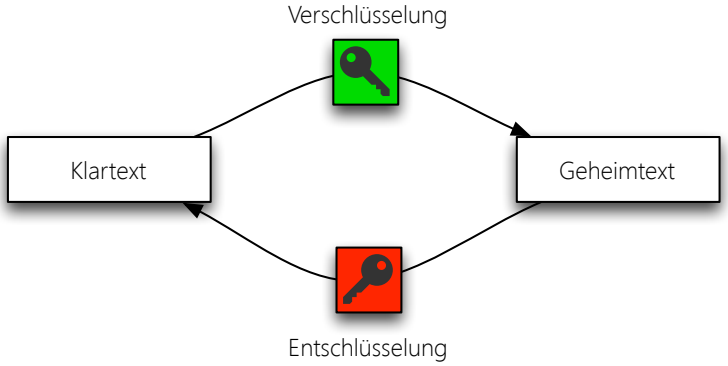


Abbildung 4.10: Asymmetrische Verschlüsselung

Bei der Erstellung einer Signatur, welche zur Sicherung der Datenintegrität sowie Herkunft verwendet wird, wird ein privater Schlüssel eingesetzt. Anschließend kann mittels

dem passenden öffentlichen Schlüssel geprüft werden, ob die Nachricht verändert wurde und die Signatur korrekt ist (Abbilundg 4.11).

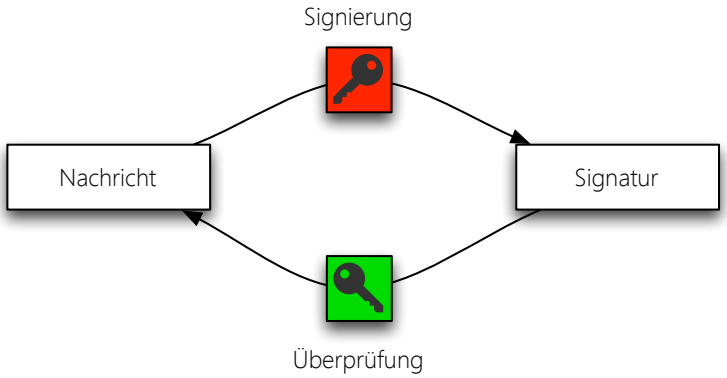


Abbildung 4.11: Asymmetrische Signierung

Die im weiteren Verlauf dieser Arbeit vorgestellte Methode, macht sich die asymmetrische Verschlüsselung zu nutzen. Es gibt unterschiedliche asymmetrische Verfahren zur Signierung von Nachrichten. Für das Orestes-Protokoll muss das richtige Verfahren gewählt werden. Die dabei entscheidenden Faktoren sind neben der Sicherheit die Länge der Signatur sowie der Aufwand der Berechnung. Im Folgenden werden die Verfahren *RSA (Rivest, Shamir und Adleman)*, *Digital Signature Algorithm (DSA)* und *Elliptic Curve Digital Signature Algorithm (ECDSA)* betrachtet.

RSA mit einer Schlüssellänge von 2048 Bits entspricht der Sicherheit eines symmetrischen Verschlüsselungsverfahrens von 112 Bits [68]. Es wird momentan angenommen, dass ein symmetrisches Verfahren mit einer Schlüssellänge von 112 Bits bis zum Jahre 2030 als sicher gilt [44]. Bei dem ECDSA-Verfahren entspricht eine Schlüssellänge von 224 Bits einem symmetrischen Verfahren mit 112 Bits [68]. Bei dem DSA-Verfahren entspricht dies einer Schlüssellänge von 2048 Bits [68]. Es werden im Weiteren daher ECDSA-224, DSA-2048 sowie RSA-2048 betrachtet .

*OpenSSL* ist ein quelloffenes und freiverfügbares Programm, welches unterschiedliche kryptographische Verfahren zur Verfügung stellt [57]. Folgende Werte wurden auf einem *Intel Core i7 2.6GHz* aus dem Jahre 2012 mittels OpenSSL berechnet. Dabei ist zu beachten, dass nur ein CPU-Kern verwendet wurde. Aktuelle Prozessoren haben im Normalfall mehrere. Die Geschwindigkeit kann also durch die parallele Durchführung von der Erstellung von Signaturen sowie der Verifizierung gesteigert werden.

| Verfahren | Signieren | Verifizierung | Signierungen/s | Verifizierungen/s |
|---|---|---|---|---|
| ECDSA-224 | 0,0001s | 0,0004s | 9698 | 2615,6 |
| RSA-2048 | 0,001189s | 0,000038s | 840,8 | 26643,6 |
| DSA-2048 | 0,000364s | 0,000438 | 2745,9 | 2284,7 |

Tabelle 4.2: Geschwindigkeitsvergleich der unterschiedlichen Signierungsverfahren

Das Ergebnis aus Tabelle 4.2 zeigt, dass sich der Berechnungsaufwand der unterschied-

lichen Verfahren stark unterscheidet. RSA ist bei der Verifizierung der schnellste, bei der Erstellung der Signatur hingegen der langsamste. DSA liegt bei beiden Aufgaben im Mittelfeld. Die Verifizierung sowie Signierung beansprucht in etwa gleich viel Zeit. ECDSA ist die richtige Wahl, wenn häufig Signierungen durchgeführt werden.

Im Folgenden werden die unterschiedlichen Längen der Signaturen gezeigt (Tablle 4.3). Dafür wurde der Text *user=hannes|exp=1387839600* mittels OpenSSL signiert. Die Angabe *exp=1387839600* entspricht dem Datum *24.12.2013 0:00:00*. Die Signatur wird anschließend in das Base64-Format umgewandelt, damit sie Problemlos per HTTP übertragen werden kann. Dabei ist zu beachten, dass die Länge des zu signierenden Dokumentes keinen Einfluss auf die Länge der Signatur hat, da zunächst ein Hash gebildet wird. Anschließend wird dieser Hash signiert. Dies bringt einen Geschwindigkeitsvorteil bei großen Dokumenten, da der Prozess der Hashbildung schneller als die Erstellung einer Signatur durchgeführt werden kann. Die Signatur muss anschließend nur für eine deutlich kleinere Zeichenkette erstellt werden. Des Weiteren wird die Sicherheit durch die so genannte *Hash-Then-Sign*-Methode gesteigert [39].

| Verfahren | Signaturlänge im Base64-Format |
|---|---|
| ECDSA-224 | 88 Bytes |
| RSA-2048 | 344 Bytes |
| DSA-2048 | 96 Bytes |

Tabelle 4.3: Signaturlänge der unterschiedlichen Verfahren

Anhand der gezeigten Daten muss entschieden werden, welche das optimale Verfahren für die Benutzung eines Authentifizierungsmechanismus im Orestes-Kontext ist. DSA empfiehlt sich nicht, da der Algorithmus in den Bereichen Signaturlänge, Signierungs- und Verifizierungsgeschwindigkeit schlechter als ECDSA abschneidet. RSA ist um den Faktor $\sim 10$ schneller bei der Verifizierung, aber ebenfalls um den Faktor $\sim 10$ langsamer in der Erstellung einer Signatur. Des Weiteren ist die Signatur knapp viermal größer.

Der Größenunterschied von 256 Bytes hat durch die fortgeschrittene Verbreitung von Breitbandanschlüssen (Deutschland 82% [11]) keinen Einfluss auf die Geschwindigkeit und muss somit nicht beachtet werden. Ein maximale Größe für den HTTP-Header ist nicht spezifiziert. Webserver definieren aber ein Limit: *Apache* 8000 Bytes, *nginx* $4000 - 8000$ Bytes (Versionsabhängig), *IIS* $8000 - 16000$ Bytes (Versionsabhängig) und *Varnish* 32000 Bytes (Standardwert, konfigurierbar). Die 344 Bytes sind somit auch für den HTTP-Header nicht zu groß.

Da das Signieren der Authentifizierungsdaten nur nach einem Login durchgeführt wird und bei jeder Anfrage auf eine geschützte Ressource die Daten überprüft werden, wird die Verifizierung um ein vielfaches öfter durchgeführt. Aus den genannten Gründen wird daher in dieser Arbeit die Verwendung des RSA-Algorithmus empfohlen.

Die Erweiterung des in Java geschriebenen Orestes-Servers, damit dieser das Verifizie-

ren unterstützt, ist problemlos möglich, da der benötigte Algorithmus in Java zur Verfügung steht.

Die Erweiterung der Reverse-Proxy-Caches ist ebenfalls möglich, da diese innerhalb der Orestes-Infrastruktur betrieben werden. So unterstützt zum Beispiel *Varnish* die Erstellung von Erweiterungen, welche als *VMODs (Varnish Modifications)* bezeichnet werden [72]. Diese VMODs werden in der Programmiersprache C geschrieben. OpenSSL steht als C-Biltiothek (*libssl*) zur Verfügung [57]. Es kann somit ein VMOD programmiert werden, welche dem Varnish Reverse-Proxy-Cache die OpenSSL-Funktionalitäten zur Verfügung stellt. Der erstellte VMOD kann anschließend per *Varnish Configuation Language (VCL)* verwendet werden. Varnish scheint hier ein geeigneter Kandidat zu sein, um als Reverse Proxy Cache verwendet zu werden.

Die Verwendung eines CDN-Providers für geschützte Inhalte ist nicht möglich, da diese im Normfall keinen vollen Zugriff auf die Server erlauben und somit nicht die benötigten Funktionalitäten implementiert werden können. Die Verwendung eines CDN-Providers für geschützte Inhalte ist ohnehin nicht zu empfehlen, da die Kontrolle über die zu sichernden Daten abgegeben wird.

Der Orestes-Server muss Ressourcen, dessen Zugriff nur für bestimmte Benutzer autorisiert ist, markieren. Dies ist nötig, da der Varnish entscheiden muss, ob eine Anfrage legitim ist. Innerhalb der VCL ist kein Zugriff auf die auszuliefernden Daten möglich. Der Zugriff auf den HTTP-Request-Header sowie den HTTP-Respone-Header steht hingegen zur Verfügung. Des Weiteren ist im Normfall die Ressource erheblich größer als der HTTP-Header und das einlesen dieser bedeutet somit einen erhöhten Rechenaufwand. Es ist daher notwendig, dass der Orestes-Server die erlaubten Rollen sowie Benutzernamen in einem HTTP-Header-Feld überträgt. Diese Arbeit schlägt dafür zwei HTTP-Header-Felder vor: *Orestes-Check-Users* sowie *Orestes-Check-Roles*. Das Präfix *X-* für selbsterstellte HTTP-Header-Felder wird nicht mehr verwendet [36].

Der Browser überträgt ebenfalls die Authentifizierungsdaten des Benutzers im HTTP-Header. Dafür werden die Felder *Orestes-Auth-Data*, welches die Daten wie Benutzername, Rollen und Verfallsdatum enthält, sowie *Orestes-Auth-Signature*, welches die Signatur der Daten enthält, verwendet. Sowie Signatur als auch Authentifizierungsdaten werden im Base64-Format übertragen, damit keine Probleme durch unterschiedliche Zeichensatztabellen entstehen.

Aus HTTP-Request-Header sowie HTTP-Respone-Header kann anschließend entschieden werden, ob die Ressource ausgeliefert wird.

Da die Erstellung der Signatur einen erhöhten Rechenaufwand benötigt, wird empfohlen, die Aufgaben der Authentifizierung auf einen separaten Login-Server zu verlegen. Dieser erstellt nach einer Registrierung einen Benutzer, signiert bei der Anmeldung Authentifizierungsdaten und stellt das OpenID-Protokoll zur Verfügung. Abbildung 4.12 zeigt diese Infrastruktur. Der Benutzer registriert sich einmalig beim Login-Server. Anschließend kann er sich mit seinem erstellten Benutzernamen und Passwort

an diesem anmelden. Die Kommunikation wird dabei mittels HTTPS verschlüsselt. Der Login-Server erstellt die Signatur und liefert die Authentifizierungsdaten im HTTP-Response-Header. Der Smart Client speichert diese Daten (z.B. mittels HTML5 Web-Storage-API) und sendet diese bei geschützten Inhalten mit. Die Verwendung von Cookies für die Übertragung der Authentifizierungsdaten wird nicht empfohlen, da im Cookie spezifiziert wird, wann der Browser diesen übersendet. Dies ist nicht flexibel genug, da lediglich eine Domain oder ein Pfad angegeben werden kann. Werden die Daten mittels Cookie übertragen, muss somit genau festgelegt sein, unter welchem Pfad geschützte Inhalte abgelegt sind. Eine Übertragung bei jeder Anfrage ist ebenfalls nicht zu empfehlen, da einige Caches Anfragen, die einen Cookie enthalten, als dynamisch ansehen und nicht zwischenspeichern. Bei der Abfrage von Daten, die eine Authentifizierung benötigen, muss der Smart Client daher per JavaScript die entsprechenden HTTP-Header-Felder hinzufügen. Die Reverse-Proxy-Caches überprüfen die Signatur und das Verfallsdatum und liefern gegebenenfalls die angeforderten Daten aus. In der Abbildung ist zu erkennen, dass bei einer kleinen Infrastruktur mit lediglich drei Reverse-Proxy-Caches bereits fünf Server an der Verifizierung und Signierung beteiligt sind. Bei der Verwendung eines HMAC bedeutet dies, dass es fünf Kopien des geheimen Schlüssels gibt. Mittels der asymmetrischen Kryptographie muss lediglich der Schlüssel des Login-Servers geheim gehalten werden.

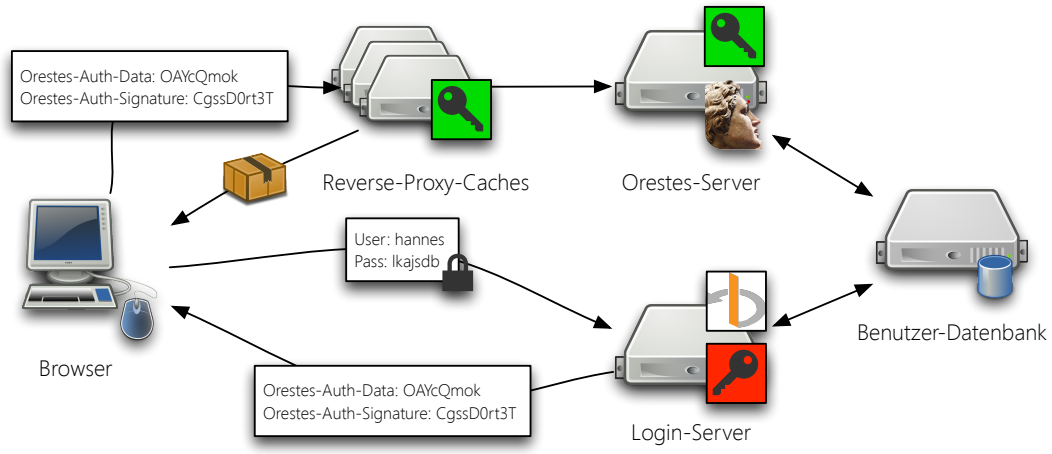


Abbildung 4.12: Authentifizierungsarchitektur aufbauend auf Orestes

Ein Problem bei einem zustandsloses Authentifizierungsverfahren ist die Deaktivierung von Benutzerkonten. Wird ein Account in der Benutzerdatenbank deaktiviert bzw. gelöscht, bleibt die Authentifizierung bis zum Erreichen des Verfallsdatums bestehen. Dies ist im Besonderen problematisch, wenn die Anwendung eine Authentifizierung über einen längeren Zeitraum wie z.B. mehrere Tage erlaubt. Im folgenden werden zwei Lösungsvorschläge vorgestellt.

Eine Möglichkeit ist die Einführung eines Prüfdatums. Dieses wird den Authentifizierungsdaten hinzugefügt. Wurde der Zeitpunkt des Prüfdatums überschritten, gilt der

Benutzer für die Reverse-Proxy-Caches und dem Orestes-Server nicht mehr als authentifiziert. Der Unterschied zum Verfallsdatum ist dabei, dass eine erneute Anmeldung mittels Benutzername und Passwort, so lange das Verfallsdatum nicht überschritten wurde, nicht durchgeführt wird. Der Smart Client kann mit gültigem Verfallsdatum und abgelaufenem Prüfdatum neue Authentifizierungsdaten beim Login-Server abrufen. Dieser kann dabei prüfen, ob das Benutzerkonto noch aktiviert bzw. vorhanden ist. Dies geschieht transparent für den Anwender. Somit ist es zum Beispiel möglich, dass eine Anmeldung für den Anwender nur alle 30 Tage nötig ist, die Gültigkeit seines Kontos aber im Abstand von 15 Minuten geprüft wird. Da der Orestes-Server direkten Zugriff auf die Benutzerdatenbank hat, ist es denkbar, dass Ressourcen, bei denen immer sichergestellt sein soll, dass das Benutzerkonto aktiviert ist, also nicht zu Cachen markiert werden. Der Orestes-Server kann somit stets überprüfen, ob die Zugriffsberechtigung noch gültig ist.

Eine weitere Möglichkeit ist die Pflege einer Sperrliste. Die Datenbank speichert dafür für jeden Benutzer das ausgestellte Verfallsdatum mit der längsten Gültigkeit. Wird ein Konto deaktiviert, wird der Benutzer der Sperrliste hinzugefügt. Er kann, sobald das gespeicherte Verfallsdatum überschritten wurde, wieder entfernt werden, da alle ausgestellten Authentifizierungstoken ihre Gültigkeit verloren haben. Die Sperrliste muss dafür auf den Reverse-Proxy-Caches stets aktuell gehalten werden. Der Orestes-Server kann direkt auf die Benutzerdatenbank zugreifen und benötigt daher keinen Zugriff auf die Sperrliste. Bei jeder Anfrage prüfen die Reverse Proxy Caches, ob sich der Benutzer auf der Sperrliste befindet und weisen gegebenenfalls die Anfrage zurück.

Der Vorteil des Prüfdatums ist, dass lediglich der Login-Server ein erhöhtes Anfrageaufkommen bewältigen muss und die restliche Infrastruktur davon unberührt bleibt. Ein Nachteil hingegen ist, dass eine sofortige Deaktivierung für zwischengespeicherte Ressourcen nicht möglich ist. Diese Funktionalität stellt die Sperrliste zur Verfügung. In der Verwendung dieser werden Reverse Proxy Caches stärker belastet, da bei jeder Anfrage von geschützten Inhalten geprüft wird, ob der Benutzer sich auf der Liste befindet. Des Weiteren bedeutet die Pflege der Liste einen erhöhten Aufwand, da in der Datenbank periodisch geprüft wird, ob Benutzer von der Sperrliste entfernt werden dürfen (z.B. nach Ablauf des Verfallsdatums). Da beide Vor- und Nachteile bieten, muss bei der Entwicklung der Anwendung entschieden werden, welche Methode verwendet wird.

## 4.5 Autorisierung

Eine entscheidende Rolle für die Möglichkeit einer Verwendung einer Datenbank in einer 2-Tier-Architektur spielt die Granularität des Autorisierungsmechanismus. Ist diese nicht fein genug, können gegebenenfalls Funktionalitäten nicht ohne Sicherheitsbedenken implementiert werden.

Wird eine Orestes-Datenbank neu angelegt, wird zunächst ein Schema erstellt [23]. In diesem werden Klassen, welche typisierte Attribute enthalten, definiert. Diese Klassen

sind wiederum in Namensräume organisiert. Abbildung 4.13 zeigt diese Hierarchie. Dabei ist zu beachten, dass die Datenbank mehrere Namensräume enthalten kann. Diese können wiederum mehrere Klasse enthalten und eine Klasse enthält in der Regel mehr als ein Attribut.

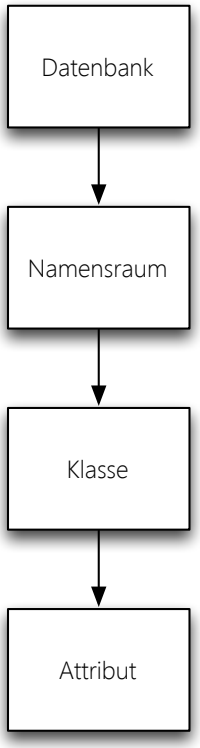


Abbildung 4.13: Aufbau eines Datenbank-Schemas in Orestes

Diese Arbeit schlägt eine rollen- und benutzerorientierte Autorisierung auf Klassen-, Objekt- und Namensraumebene vor. Beim Erstellen eines Schemas kann festgelegt werden, welche Rollen bzw. Benutzer auf die erstellten Objekte Zugriff haben.

Ein Beispiel für eine Anwendung, die Rechte auf Namensraumbene benötigt, ist eine, dessen Inhalt nur für registrierte Benutzer zur Verfügung steht. Dafür erhält jeder Nutzer bei der Registrierung die Rolle *user*. Der geschützte Inhalt befindet sich in einem Namensraum, der nur Zugriff für Benutzer mit der Rolle *user* zulässt. Dieses wird über das Schema festgelegt.

Benutzerrechte auf Klassenebene werden für Anwendungen benötigt, die nur Teile eines Namensraums schützen. Zum Beispiel ein Onlinezeitungsverlag, der einen Teil seiner Artikel nur für zahlende Kunden zur Verfügung stellt.

Die Zugriffsverwaltung auf Objektebene wird für dynamisch vergebene Rechte benötigt. Registriert sich ein Benutzer, werden im Normalfall personenbezogene Daten angegeben, diese dürfen nur von dem Benutzer selber verwendet werden. Die Rechte werden daher bei der Erstellung des Objekts vergeben.

Ein weiterer notwendiger Schritt ist die Einführung einer Art *Stored Procedures*. Dabei

handelt es sich um Scripte, die beim Aufruf von der Datenbank ausgeführt werden. Mittels dieser Scripte ist es zum Beispiel möglich, Daten zu verwenden, die nicht öffentlich zugänglich sein dürfen. Des Weiteren werden diese benötigt, um Benutzereingaben zu kontrollieren. Ein weiterer Vorteil ist die Steigerung der Geschwindigkeit in bestimmten Anfragesituation. Es kann zum Beispiel schneller sein, Daten zu aggregieren und dann auszuliefern als alle benötigten Daten an den Smart Client zu senden, damit dieser den Prozess ausführt. Seit Version 6 beinhaltet das *Java Development Kit (JDK)* die Mozilla JavaScript-Egine *Rhino* [58]. Es bietet sich somit an, dass als Scriptsprache JavaScript verwendet wird.

Stored Procedures werden als Ressource innerhalb eines Namensraums definiert. Sie werden wie Objekte abgerufen und verhalten sich dabei wie solche. Innerhalb einer Stored Procedure können zwei Methoden definiert werden. Dabei wird unterschieden, ob es sich um eine schreibende oder lesende Anfrage handelt. Des Weiteren muss das Schema die Funktionalität zur Verfügung stellen, Klassen zu definieren, die ausschließlich per Stored Procedure abgerufen werden können. Somit lassen sich in eine Berechnung Daten integrieren, die nicht öffentlich werden dürfen. Eine weitere Möglichkeit ist die Integration einer Stored Procedure innerhalb einer Klasse. Dieses bewirkt, dass bei jeder Objekterstellung (POST-Request) das definiert Skript ausgeführt wird. Ebenso bei jeder lesenden Anfrage (GET-Request).

## 4.6 Fazit

Die Authentifizierungsmethode mittels zustandslosem Cookie nach Fu et al. hat sich als geeignet für die Verwendung des Orestes-Protokolls gezeigt. Da kein Zustand gehalten wird, können geschützte Inhalte mittels Reverse-Proxy-Caches ausgeliefert werden. Das Orestes-Protokoll kann somit auch für diese Daten skalieren. Dies ist wichtig, da häufig ein Benutzerkonto für die Verwendung einer Anwendung vorausgesetzt wird. Die Sicherheitsbedenken der Fu et al. Methode, die diese Arbeit gezeigt hat, können durch die Verwendung eines asymmetrischen Kryptographieverfahren reduziert werden.

Durch die Autorisierung auf den drei unterschiedlichen Ebenen entsteht große Flexibilität. Dies ermöglicht die Erstellung von unterschiedlichsten Anwendungen. Mittels *Stored Procedures* wird eine weitere Option zur Benutzerkontrolle zur Verfügung gestellt. Des Weiteren ermöglichen sie die Verwendung von geheimen Daten innerhalb einer Anwendung.

Im nachfolgenden Kapitel wird eine Lösung Vorgestellt, welche es ermöglicht, Smart Clients für Suchmaschinen zu optimieren.

# 5 Suchmaschinenoptimierung

Für eine kommerziell erfolgreiche Webseite ist es unabdingbar, dass die Indizierung durch Suchmaschinen möglich ist [42]. Dies ist bei einem Smart Client nicht ohne weiteren Aufwand möglich, da *Crawler*[1] kein JavaScript ausführen [17]. Es muss somit eine Möglichkeit gefunden werden, den Inhalt des Smart Clients ohne, dass der Client JavaScript interpretieren und ausführen muss, auszuliefern.

In diesem Kapitel wird eine bereits bestehende mit dem Titel *Rendr* sowie im Rahmen dieser Masterarbeit entwickelte Lösung namens *Seoxy (Seo Proxy)* vorgestellt und diskutiert. Dafür werden zunächst die Grundlagen geschaffen, auf dem der *Seoxy* basiert.

Es wird dabei ausschließlich das JavaScript-Framework AngularJS betrachtet, da es sich bei diesem, wie in Kapitel 3 erörtert, um das am bestgeeignetste für einen Smart Client in einer 2-Tier-Architektur auf Basis von Orestes handelt. Ebenso wird nur auf die Funktionsweise des Suchmaschinen-Crawlers von *Google* eingegangen. Die Konkurrenz wie zum Beispiel *Bing* wird nicht betrachtet, da es sich bei *Google* um die mit Abstand verbreitetste Suchmaschine handelt [64].

## 5.1 Grundlagen

Da der Crawler kein JavaScript ausführen kann und es somit für ihn nicht möglich ist, den Inhalt des Smart Clients zu laden und zu erstellen, muss diese Aufgaben eine dritte Instanz übernehmen. Es handelt sich somit um eine 3-Tier-Architektur. Diese wird allerdings nur verwendet, wenn der Client bzw. Browser kein JavaScript ausführt.

### 5.1.1 URL

Ändert man in einem Browser die URL, wird die Seite neu geladen. Dies findet ebenfalls statt, wenn ein Link, der auf eine Unterseite weiterleitet, aufgerufen wird. Da das Laden von Inhalten der Smart Client per JavaScript übernimmt, darf der Browser die Seite nicht neu laden, wenn auf dieser navigiert wird.

Die einfachste Methode um dies zu umgehen ist, dass der Inhalt geladen wird und die URL unverändert bleibt. Diese hat allerdings den erheblichen Nachteil, dass der aktuelle Zustand des Smart Clients nicht über die URL rekonstruiert werden kann. Es müsste somit dieselbe Navigationsabfolge stattfinden, um den gleichen Inhalt anzuzeigen. Es wäre mittels dieser Methode nicht möglich, die URL zu speichern, um den Inhalt zu einem

[1] Als Crawler wird die Software von Suchmaschinen bezeichnet, die Internetseiten durchsucht.

späteren Zeitpunkt neu anzeigen zu lassen. Ein weiterer Nachteil ist, dass keine URL, die den erwünschten Inhalt referenziert, weitergegeben werden kann. Dies hat im Besonderen im Hinblick auf die Suchmaschinenoptimierung den Nachteil, dass der Smart Client nicht verlinkt werden kann und somit vom Crawler schlechter bewertet wird [42]. Ebenso funktioniert der Historiefunktion des Browsers nicht, da keine Änderung an der URL stattfindet.

Eine Lösung für diese Probleme sind so genannte *Hash-Bangs*. Ein Hash-Bang besteht aus den Zeichen *#!* und wird der URL angehängt [29]. Abbildung 5.1 zeigt eine URL mit einem *Hash-Bang*. Im gezeigten Beispiel wird zunächst die URL *http://orestes.info/twt* aufgerufen. Anschließend wird der Benutzer mit der Identifikationsbezeichnung *hannes* geladen. Der Smart Client lädt die dafür benötigten Daten aus der Datenbank und fügt zur URL *#!user/hannes* hinzu. Einem Doppelkreuz (#) in der URL wird nach Spezifikation verwendet, um ein HTML-Element zu referenzieren und zu diesem innerhalb der geladenen Seite zu springen ohne diese nochmalig vom Server zu laden. Das Doppelkreuz inklusive Referenz wird als *Fragmentbezeichner* (engl. *Fragment Identifier*) bezeichnet [8]. Das HTML-Element wird mittels des Attributes *id* identifiziert. In dem Wert dieses Attributs ist kein Ausrufezeichen erlaubt [74]. Die Methode der Verwendung eines Hash-Bangs macht sich diese beiden Eigenschaften zu nutzen.

```
1 http://orestes.info/twt#!user/hannes
```

Listing 5.1: Hash-Bang in einer URL

Im Browser lässt er sich der Hash-Bang mittels des in 5.2 gezeigten Befehls auslesen. Bei dem Beispiel aus 5.1 würde die Ausführung *#!user/hannes* ergeben. Der Client kann somit den Hash-Bang interpretieren und die benötigten Daten vom Server Laden.

```
1 window.location.hash
```

Listing 5.2: Hash-Bang per JavaScript auslesen

Da es sich bei dieser Methode um einen Workaround handelt und die Verwendung einer Raute in der URL nicht für diesen Zweck angedacht ist, wird der Inhalt des Hash-Bangs vom Browser nicht an den Server übermittelt. Es ist somit nicht möglich, die gewünschte Seite auf dem Server vorzugenerieren.

Der Crawler von Google wandelt daher Links, die einen Hash-Bang enthalten, um. Der Hash-Bang wird durch einen GET-Parameter[2] ersetzt [29]. Der Inhalt wird somit auch an den Server übertragen. Abbildung 5.3 zeigt die URL, die Google im verwendeten Beispiel aufruft.

```
1 http://orestes.info/twt?_escaped_fragment_=user/hannes
```

Listing 5.3: Googles Umwandlung des Hash-Bangs in einen GET-Parameter [29]

[2]Bei einem HTTP-GET-Request können Parameter übertragen werden. Diese bestehen aus einem Key-Value-Paar.

Der Server kann somit erkennen, wenn eine Unterseite aufgerufen wird und die entsprechende laden und ausgeben.

Die HTML-Spezifikation in der Version 5 stellt die Funktion *pushState* zur Verfügung. Diese ist bereits in den aktuellen Versionen von den Browsern Internet Explorer, Chrome, Firefox und Opera implementiert [49].

Mittels dieser Funktion ist es in JavaScript möglich, die URL, die der Browser anzeigt, zu ändern ohne, dass dieser die Seite neu geladen wird. Es kann somit der aktuelle Zustand der Seite über die URL ohne die Verwendung eines Workarounds abgebildet werden.

```
1 window.history.pushState({ user: 'hannes' }, 'Benutzer Hannes', '/twt/
    user/hannes/');
```

Listing 5.4: Verwendung der pushState-Funktion [49]

Abbildung 5.4 zeigt die Benutzung der Funktion *pushState*. Es wird ein State-Objekt, der Titel sowie eine URL übergeben [49]:

**State-Objekt:** Diesem Parameter wird ein JavaScript-Objekt übergeben [49]. Für die einfache Änderung der URL spielt dieser Parameter keine Rolle. Beim Ausführen der Funktion *pushState* wird das Event *window.onpopstate* ausgeführt. Diesem Event wird das State-Objekt übergeben. Des Weiteren lässt es sich über *history.state* abrufen. Es dient somit dazu, weitere Informationen, die gegebenenfalls an anderer Stelle benötigt werden, zu speichern und weiterzugeben.

**Titel:** Es kann ein String übergeben werden, der den Titel des Dokumentes ändert. Dieser Parameter wird allerdings von allen aktuellen Browsers ignoriert. Um den Titel zu ändern, muss das Attribut *document.title* gesetzt werden. Da es gegebenenfalls sein kann, dass der Parameter in Zukunft Verwendung findet, bietet es sich an, *document.title* zu übergeben.

**URL:** Dieser Parameter ist der entscheidende. Es wird als String angegeben, wie die URL des neuen Zustandes aussieht. Dabei ist zu beachten, dass sie entweder relativ oder absolut angegeben werden kann. Das Ändern der Domain ist aus Sicherheitsgründen nicht zugelassen und führt zu einem *SecurityError*.

Befindet sich der Client auf der URL *www.orestes.info/twt/* und es wird die Funktion aus Abbildung 5.4 ausgeführt, befindet sich anschließend im Adressfeld des Browsers die URL *www.orestes.info/twt/user/hannes*. Wird der beginnende / des URL-Parameters entfernt, handelt es sich um einen relativen Pfad, beim vorherigen Beispiel handelt es sich um einen absoluten, und würde in diesem Kontext die URL zu einer fehlerhaften und zwar zu *www.orestes.info/twt/twt/user/hannes/* ändern. Es muss somit stets darauf geachtet werden, welche URL der Funktion *pushState* übergeben wird.

HTML5-URLs haben den Nachteil, dass sie gesondert behandelt werden müssen, wenn sie direkt durch die Adressleiste und nicht per Navigation aufgerufen werden. Es handelt sich bei den URLs nur um virtuelle URLs, die Ressourcen referenzieren, welche erst

durch den Smart Client erstellt werden. Im Normalfall versucht der Webserver die Ressource zu laden, die durch die URI aufgerufen wird. Das ist in diesem Fall nicht möglich, da sie noch nicht vorhanden ist. Der Webserver muss somit so konfiguriert werden, dass er alle URLs bei denen es sich nicht um statische Dateien wie zum Beispiel Bilder oder Stylesheets handelt auf die *index.html* weiterleitet, damit über diese AngularJS geladen werden kann und anschließend die benötigte Ressource erstellt wird.

### 5.1.2 Node.js

Durch die immer größer werdende Anzahl von Clients im World Wide Web und die geänderten Anforderungen dieser, bringen klassische Webserver wie *Apache* an ihre Grenzen. Immer mehr Desktopanwendungen werden als Webapps realisiert. Sie sollen sich aber weiterhin wie Desktopanwendungen verhalten. Das bedeutet, dass sie in Echtzeit über neue Informationen benachrichtigt werden müssen. Es müssen somit Verbindungen zu mehreren tausend Clients aufrecht gehalten werden. Klassische Webserver erstellen für jede Verbindung einen eigenen Thread. Es ist nicht unüblich, dass ein Thread zwei Megabyte benötigt. Somit werden bereits für 10.000 Anfragen 20 Gigabyte an Speicher verwendet. Dieses Beispiel zeigt, dass Webserver wie der Apache nicht ausreichend genug skalieren [63].

Bei Node.js handelt es sich um eine Server-Plattform, die es sich zur Aufgabe gemacht hat, diese Probleme zu Lösen. Es wurde ursprünglich von *Ryan Dahl* entwickelt und die erste Version im Jahre 2009 veröffentlicht. Mittlerweile wurde die Leitung des Projekts an *Isaac Schlueter* übergeben. Es handelt sich somit um eine noch sehr junge Plattform. Anwendungen für Node.js werden in JavaScript geschrieben. Es verwendet zur Ausführung den von Google entwickelten Interpreter *V8*. Dieser wird ebenfalls von dem Browser Chrome verwendet [63].

Node.js geht einen anderen Weg und erstellt nicht für jede Anfrage einen eigenen Thread. Alle Anfragen werden von einem beantwortet. Es macht sich zu Nutzen, dass bei einer Webanwendung der Großteil der Ausführungszeit auf *I/O-Operationen* zurückzuführen ist. Es verwendet daher das so genannte *Non-Blocking I/O*. Dies bedeutet, dass der Thread weitere Anfragen beantworten kann, während I/O-Operationen vom System ausgeführt werden, da diese den Thread nicht blockieren. Node.js kann auf diese Weise eine erhebliche höhere Zahl von Anfragen beantworten. Da Node.js nur einen Thread verwendet, stellt sich die Frage, wie die Leistung eines Multiprozessorsystems ausgenutzt werden kann. Node.js beantwortet diese nicht. Die einfachste Lösung ist das Starten von mehreren Node.js Instanzen. Diese können anschließend per Interprozesskommunikation wie zum Beispiel *TCP/IP* kommunizieren. Was wiederum den Vorteil hat, dass es für die Entwicklung der Anwendung keinen Unterschied macht, auf wie vielen dedizierten Rechnern Node.js läuft [63].

Des Weiteren stellt Node.js Module zur Verfügung, die es besonders einfach machen, HTTP-Server zu entwickeln.

Abbildung 5.5 zeigt die Implementierung eines einfachen HTTP-Servers. Es wird zunächst das *http*-Modul von Node.js geladen. Mittels diesem wird über die Funktion *createServer* ein HTTP-Server erstellt. Anschließend muss noch ein Port über die Funktion *listen* angegeben werden. Im gezeigten Beispiel reagiert der Server auf Anfragen auf Port 80. Die Referenz auf die Instanz des Servers wird in der Variablen *server* gehalten. An dieser wird über die Methode *on* eine Funktion übergeben, die bei jeder Anfrage ausgeführt wird. Diese enthält die Parameter *req* für Request (Anfrage) und *res* für Response (Antwort). An dem Request-Objekt können Informationen der Anfrage ausgelesen werden. Wie zum Beispiel der Request-Header. Über die Request-Variable lässt sich somit auch bestimmen, welche URI abgefragt wird. Mittels der Response-Variable wird die Antwort, die der Server an den Browser schickt, zusammengestellt. Über *writeHead* wird zunächst der HTTP-Response-Header definiert. Der erste Parameter enthält den Statuscode. Im Beispiel wird der Code *200* für *OK* übergeben. Der Code bedeutet, dass die Anfrage erfolgreich bearbeitet wurde. Dem zweiten Parameter wird ein JavaScript-Objekt übergeben. Über dieses Objekt lassen sich die weiteren Felder des Response-Headers definieren. Im Beispiel wird das Attribut *Content-Type* mit dem Inhalt *text/plain* definiert. Mittels der Funktion *write* wird der eigentliche Inhalt der Antwort bestimmt. Im Beispiel wird lediglich der Text *Hello World!* ausgegeben. Es ist ebenso möglich, eine Datei einzulesen und auszugeben. In den Beispielen des Kapitels 5.2 wird dieses gezeigt. Um die Antwort zu beenden, wird die Funktion *end* an dem Response-Objekt aufgerufen.

```
1 var http = require('http');
2 var server = http.createServer().listen(80);
3 server.on('request', function(req, res) {
4   res.writeHead(200, {'Content-Type': 'text/plain'});
5   res.write('Hello World!');
6   res.end();
7 });
```

Listing 5.5: HTTP-Server in Node.js

Ein weiterer Vorteil ist die Unterstützung von JSON und WebSockets. Es stellt somit eine optimale Serverbasis für Smart Clients dar.

Ebenso ist eine Bibliothek eines Drittanbieters verfügbar, welche die Kommunikation zwischen Node.js und PhantomJS verwaltet. Aus den in diesem Kapitel geschilderten Gründen, ist die in dieser Arbeit entwickelten Software Seoxy für Suchmaschinenoptimierung für Smart Clients in Node.js geschrieben.

### 5.1.3 PhantomJS

Bei *PhantomJS* handelt es sich um einen so genannten *Headless-Browser*. Er basiert auf der HTML-Rendering-Engine *WebKit*. Diese wird ebenfalls von dem Browser *Safari* eingesetzt. Chrome und Opera verwenden in den aktuellen Versionen einen WebKit-Fork

mit dem Namen *Blink*. Des Weiteren verwendet PhantomJS ebenfalls die V8 JavaScript-Engine [35].

Bei einem Headless-Browser handelt es sich um einen Browser, der keine grafische Benutzeroberfläche besitzt. Dies klingt im ersten Moment weniger sinnvoll, da er somit nicht die erstellte Webseite anzeigen kann. Es gibt allerdings unterschiedliche Anwendungsfälle, die keine grafische Benutzeroberfläche benötigen. Dazu zählt im Besonderen das automatisierte Testen von Webseiten. So lassen sich unter Anderem Scripte erstellen, die nach dem Laden der Webseite durch PhantomJS diese testen. Ebenso lässt sich das Ladeverhalten von Webseiten analysieren. Es kann protokolliert werden, welcher Aufruf wie viel Zeit in Anspruch nimmt. Mit diesen Informationen ist es möglich, die Seite anschließend zu optimieren. Es ist ebenso denkbar, dass nach jeder Änderung automatisiert geprüft wird, wie sich die Ladezeiten verändert haben und beim Überschreiten eines Schwellwertes diese Änderung rückgängig gemacht werden. PhantomJS kann ebenfalls Bilder der generierten Webseiten erstellen und abspeichern.

Die für diese Arbeit essentielle Funktion ist die Speicherung der erstellten Webseite als HTML-Datei. PhantomJS kann somit den Smart Client ausführen, den Inhalt erstellen und anschließend als HTML-Datei speichern, damit diese an den Google Crawler ausgeliefert werden kann.

Folgendes Beispiel ist in Node.js geschrieben und verwenden das Modul *phantom*. Dieses verwaltet die Kommunikation zwischen Node.js und PhantomJS [2].

```
1  var phantom = require('phantom');
2  phantom.create(function(ph) {
3    ph.createPage(function(page) {
4      page.open('http://www.orestes.info/twt/user/hannes',
5        function(status) {
6          page.evaluate(function() {
7            $.getJSON('/user/hannes/tweets', function(data) {
8              tweets.push(data);
9            });
10         }, function(result) {
11           sendToClient(result);
12         });
13     });
14   });
15 });
```

Listing 5.6: Erstellung einer Seite in PhantomJS

In Abbildung 5.6 wird zunächst das *phantom*-Modul über *require('phantom')* geladen. An diesem wird anschließend über die Funktion *create* der PhantomJS-Prozess gestartet. Über die Variable *ph* kann der PhantomJS-Prozess gesteuert werden. Um eine Seite zu laden, muss zunächst ein *page*-Objekt erstellt werden. Dies wird mittels der Funktion *createPage* realisiert. Anschließend kann über die Methode *open* an dem *page*-Objekt das

Laden der Seite gestartet werden. Diese Methode kann eine Funktion übergeben werden, die ausgeführt wird, sobald die Seite geladen wurde. Über die Variable *success* lässt sich prüfen, ob das Laden erfolgreich war oder Fehler aufgetreten sind, damit diese gegebenenfalls abgefangen und entsprechend behandelt werden können. Nachdem die Seite erfolgreich geladen wurde, kann sie mittels *evaluate* manipuliert werden. Der Funktion *evaluate* können zwei Parameter übergeben werden. Der erste beinhaltet gekapselt in einem JavaScript-Objekt den Code, der auf der Seite ausgeführt wird. Im gezeigten Beispiel wird ein JSON geladen, das anschließend einem Array hinzugefügt wird. Für den zweiten Parameter der Methode *evaluate* kann eine Funktion definiert werden, welche das Ergebnis der manipulierten Seite erhält. Im Beispiel aus Abbildung 5.6 wird eine nicht näher definierte Methode *sendToClient* aufgerufen. Dieser wird das Ergebnis der Seite übergeben.

Auf die weiteren Möglichkeiten von PhantomJS wird nicht näher eingegangen, da sie für diese Masterarbeit nicht relevant sind. Im nachfolgenden Kapitel wird eine Lösung auf Basis von Node.js und PhantomJS vorgestellt, um Smart Clients aufbauend auf dem JavaScript-Framework AngularJS für den Google Crawler zur Verfügung zu stellen.

## 5.2 Seoxy

Im Rahmen dieser Arbeit wird ein Programm auf Basis von Node.js und PhantomJS namens *Seoxy (Seo Proxy)* entwickelt, welches es ermöglicht, Smart Clients ohne die Verwendung von JavaScript anzuzeigen, was den großen Vorteil bringt, dass die Webapplikation von Suchmaschinen indiziert werden kann. Des Weiteren hat ein geringer Teil der Internetnutzer in der Regel aus Sicherheitsgründen JavaScript deaktiviert. Diese können unter der Verwendung von Seoxy ebenfalls den Inhalt des Smart Clients betrachten.

Abbildung 5.1 zeigt den groben Aufbau der Infrastruktur unter Verwendung der Applikation Seoxy. Der Browser, in der Abbildung als *Web-App* beschrieben, lädt seine Daten direkt von dem Orestes-Server und generiert anschließend den anzuzeigenden Inhalt. Da der Google Crawler diese nicht laden kann, weil dafür JavaScript benötigt wird, muss dieser bereits eine vorgenerierte HTML-Datei erhalten. Diese Aufgabe übernimmt der Seoxy. Er agiert wie ein eigenständiger Browser und arbeitet für den Orestes-Server vollkommen transparent. Er lädt die Daten, erstellt die HTML-Seite und übermittelt sie anschließend an den Google Crawler.

### 5.2.1 Grundlagen

Damit die Applikation gegebenenfalls von dem Seoxy erstellt werden kann, muss entschieden werden, ob es sich bei der Anfrage um eine reguläre oder von einem Crawler erstellte handelt. Dies ist auf unterschiedliche Arten möglich und abhängig von der verwendeten URL-Methode.

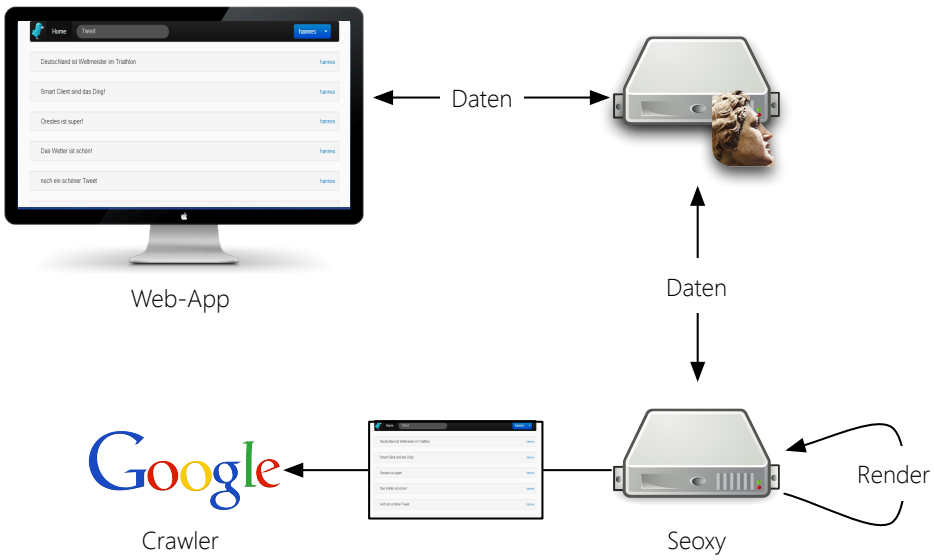


Abbildung 5.1: Aufgabenverteilung unter Verwendung von Seoxy

Eine Möglichkeit ist, den *User Agent*[3] auszulesen und anhand diesem zu unterscheiden, ob die Seite bereits vorberechnet werden muss oder nicht. Diese ist umständlich und führt häufig zu Problemen, da eine Liste von Regulären Ausdrücken gepflegt werden muss. Anhand dieser werden die unterschiedlichen Crawler erkannt. Ändert ein Crawler den User Agent, wird dieser nicht mehr erkannt und die Liste muss geändert werden. Des Weiteren ist diese Methode anfällig für False-Positives, da ein Fehler in der Liste der Regulären Ausdrücke dazu führen kann, dass Clients, die JavaScript interpretieren können, auf den Seoxy geleitet werden. Dieses würde dazu führen, dass alle Vorteile eines Smart Clients auf Basis einer 2-Tier-Architektur verloren gehen.

Wie bereits in Kapitel 5.1.1 beschrieben, gibt es unterschiedliche Arten von URLs, die AngularJS verwenden kann. Werden Hash-Bang-URLs verwendet, wie in es den Standardeinstellungen der Fall ist, übergibt der Google Crawler den Inhalt des Hash-Bang als Wert des GET-Parameters *_escaped_fragment_*. Es müssen somit an der Web-Applikation keine weiteren Veränderungen vorgenommen werden.

Für HTML5-URLs müssen noch weiteren Details beachtet werden. Da im HTML5-Modus alle URLs auf die *index.html* umgeleitet werden und die eigentliche Ressource erst durch AngularJS erstellt wird, wird dem Google Crawler lediglich der Inhalt der *index.html* angezeigt. Da kein Hash-Bang verwendet wird, ist es für den Crawler nicht ersichtlich, dass es sich um eine Ressource handelt, welche durch ein JavaScript-Framework erstellt wird. Über einen Meta-Tag im HTML-Header lässt sich dieses Problem umgehen. Abbildung 5.7 zeigt dieses.

[3]Bei dem *User Agent* handelt es sich um ein HTTP-Header-Attribut, in welchem Angaben zur Client-Software übermittelt werden.

```
1 <meta name="fragment" content="!">
```

Listing 5.7: Googles Umwandlung des Hash-Bangs in einen GET-Parameter

Der Crawler erkennt somit, dass es sich um eine durch ein JavaScript-Framework generierte Seite handelt. Er ruft anschließend die URI erneut auf und fügt dabei den GET-Parameter *_escaped_fragment_* hinzu. Dabei ist zu beachten, dass der Inhalt des Parameters leer ist, da kein Hash oder Hash-Bang in der URI vorhanden ist. Der Pfad der URI bleibt aber weiterhin bestehen. Es ist somit möglich, die auszuliefernde Ressource anhand der Anfrage zu erkennen.

Für die Erkennung wird in den gezeigten Beispielen ein Reverse-Proxy verwendet. Da der Seoxy in JavaScript auf Basis von Node.js geschrieben ist, wird im Rahmen dieser Arbeit ebenfalls ein auf Node.js aufsetzender Reverse-Proxy entwickelt. Es ist nicht zwingend erforderlichen, diesen zu verwenden. Es ist ebenfalls möglich, eine bereits bestehende Software wie zum Beispiel *nginx* oder *Varnish* einzusetzen. Der Reverse-Proxy muss lediglich so konfiguriert sein, dass er Anfragen, die den besagten GET-Parameter enthalten, an den Seoxy weiterleitet und die restlichen Anfragen an den Orestes-Server. In der Orestesinfrastruktur wird die Funktion des Reverse Proxys durch den Reverse Proxy Cache übernommen. Da die *index.html* durch die Cache-Hierachie ebenfalls zwischengespeichert wird, liegt die Vermutung nahe, dass beim Abrufen einer HTML5-URL der Crawler nicht den Reverse Proxy bzw. Reverse Proxy Cache erreicht. Dies ist aber nicht der Fall, da die verwendeten Caches Anfragen, die einen GET-Parameter enthalten, ignorieren und weiterleiten.

Abbildung 5.2 zeigt die Infrastruktur unter der Verwendung des Seoxys und einen Reverse-Proxy. Des Weiteren wird der Anfragenverlauf des Google Crawlers dargestellt. Im gezeigten Beispiel versucht der Crawler, die URI */usr/hannes* zu indizieren. Es wird die *index.html* aus der Orestesinfrastruktur ausgeliefert. Der Ablauf dieses Vorgangs wird durch die roten Pfeile dargestellt. Der Crawler erkennt den in Abbildung 5.7 gezeigten Meta-Tag und ruft die URI mit dem *_escaped_fragment_*-Parameter erneut auf. Der Reverse-Proxy erkennt den Parameter und leitet die Anfrage an den Seoxy weiter. Dieser ruft die URI auf, erhält die *index.html*, führt das JavaScript-Framework aus und sendet die generierte Seite an den Google Crawler. Dieser kann anschließend die Seite indizieren und die Webapplikation wird in den Suchmaschinenindex aufnehmen. Dieser Verlauf wird durch die grünen Pfeile gezeigt.

Browser, die nicht fähig sind, JavaScript zu interpretieren, müssen die Seite mit dem *_escaped_fragment_* Parameter aufrufen, damit sie ebenfalls an den Seoxy geleitet werden. Da dieser so genannte Redirect automatisiert durchgeführt werden muss, werden die HTML-Elemente <*noscript*> und <*meta*> innerhalb der *index.html* verwendet.

**noscript:** Elemente, die sich innerhalb dieses befinden, werden nur ausgeführt, wenn der Browser kein JavaScript unterstützt bzw. JavaScript deaktiviert wurde.

**meta:** Das Element beinhaltet Metainformationen der Webseite. Es lassen sich aber auch

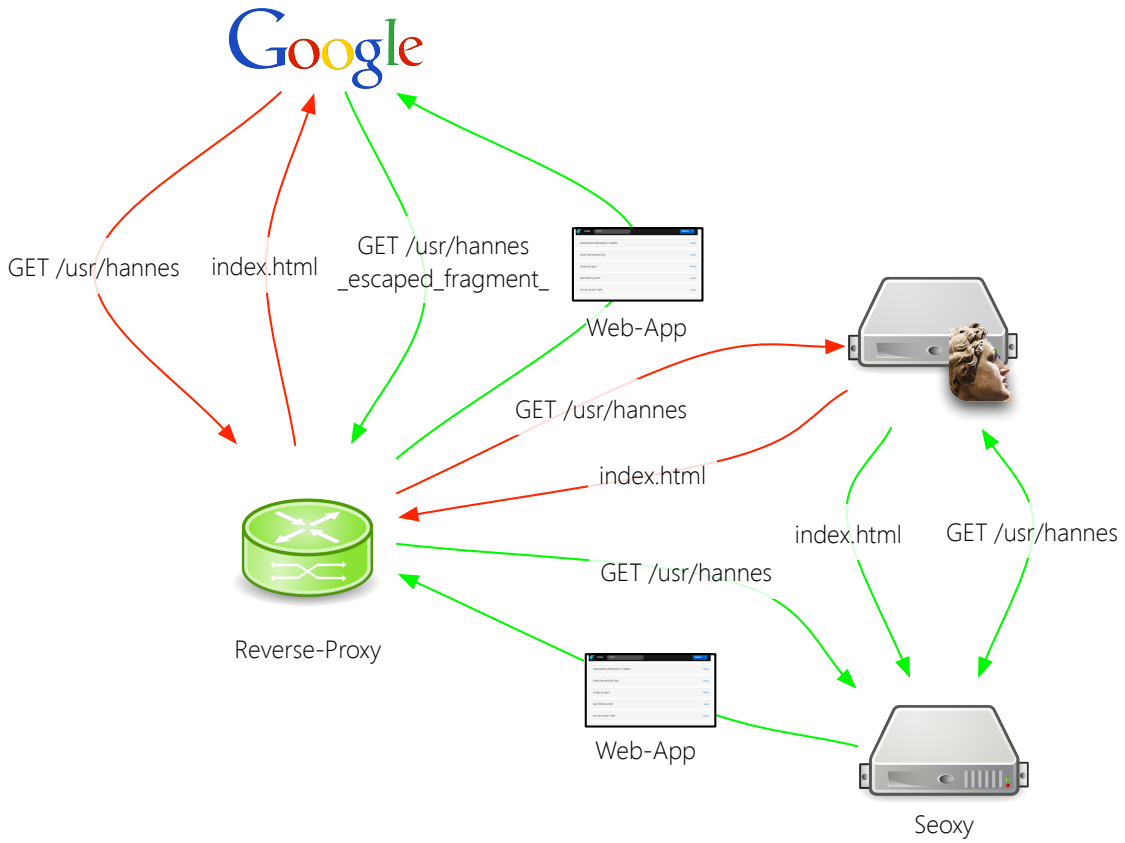


Abbildung 5.2: Ablauf der Anfragen des Google Crawlers

Anweisungen für Clients definieren. Ein weit verbreiteter Anwendungsfall sind Anweisungen für Suchmaschinencrawler. Damit können Seiten unter anderem vom Suchmaschinenindex ausgeschlossen werden. Des Weiteren kann der Browser aufgefordert werden, einen Redirect durchzuführen.

```
1 <noscript data-type="seoxy">
2   <meta http-equiv="refresh" content="0;url=?_escaped_fragment_" />
3 </noscript>
```

Listing 5.8: Redirect für Browser mit deaktiviertem JavaScript

Die in Abbildung 5.8 gezeigten HTML-Elemente werden zu dem HTML-Header der *index.html* hinzugefügt. Das Attribut *data-type* mit dem Wert *seoxy* hat auf die Funktionsweise des Redirects keine Auswirkung. Es wird im späteren Verlauf dieses Kapitels näher darauf eingegangen. Das Meta-Element wird nur ausgeführt, wenn der Browser kein JavaScript interpretieren kann. Durch den Wert *refresh* des Attributs *http-equiv* wird der Client angewiesen, die Seite erneut zu laden. Im Attribut *content* wird durch die Angabe der 0 definiert, dass die Seite ohne Wartezeit erneut aufgerufen wird. Des Weiteren wird die aufzurufende URL angegeben. Da es sich bei der Angabe im Beispiel um keine absolute URL handelt, wird die aufgerufenen und mit der Angabe des Meta-Elements konkateniert.

Dabei ist zu beachten, dass diese Methode nur funktionsfähig ist, wenn die URL nicht bereits GET-Parameter enthält. Wäre dies der Fall, müsste der *_escaped_fragment_*-Parameter mit einem & und nicht mit einem ? beginnen, weil mehrere GET-Parameter mittels & zu trennen sind. Da kein JavaScript aktiviert ist, ist es nicht möglich, das Symbol dynamisch und abhängig von der Anzahl der Parameter zu ändern. Eine Lösung für dieses Problem ist der Redirect auf die Startseite. Des Weiteren muss der *_escaped_fragment_*-Parameter und ein weiterer wie zum Beispiel *nojs* angegeben werden. Der zweite Parameter wird benötigt, damit der Seoxy unterscheiden kann, ob es sich um einen Crawler oder einen umgeleitet Browser ohne JavaScript-Unterstützung handelt. Tritt der zweite Fall ein, kann der Referrer verwendet werden, um die zu generieren Ressource zu definieren. Bei dem Referrer handelt es sich um ein HTTP-Header-Feld, das die URL des vorherigen Seitenaufrus angibt. Dabei ist zu beachten, dass dieser nur übermittelt wird, wenn zum Beispiel ein Link angeklickt oder ein Redirect per Meta-Element durchgeführt wurde. Da AngularJS im Normalfall keine GET-Parameter verwendet, wird auf diese Methode im weiteren Verlauf nicht eingegangen.

### 5.2.2 Implementierung

```
1  var url = require('url');
2  var httpProxy = require('http-proxy');
3  var reverse = http.createServer().listen(80);
4  var proxy = new httpProxy.RoutingProxy();
5  reverse.on('request', function(req, res) {
6    var urlParts = url.parse(req.url, true);
7    var query = urlParts.query;
8    if(query.hasOwnProperty('_escaped_fragment_')) {
9      proxy.proxyRequest(req, res, {
10       host: 'localhost',
11       port: 8000
12     });
13   } else {
14     proxy.proxyRequest(req, res, {
15       host: 'localhost',
16       port: 9000
17     });
18   }
19 });
```

Listing 5.9: Reverse-Proxy in Node.js

Abbildung 5.9 zeigt den auf Basis von Node.js geschrieben Reverse-Proxy. Es wird die Drittanbieterbibliothek *http-proxy* verwendet. Diese stellt einen *RoutingProxy* zur Verfügung, welcher dafür verantwortlich ist, Anfragen an einen definierten Server weiterzuleiten. Der Reverse-Proxy wird auf dem HTTP-Port *80* gestartet. In der Funktion, die

bei einer Anfrage an den Proxy ausgeführt wird, wird zunächst die URL des Requests ausgelesen. An dem *url*-Objekt ist es anschließend möglich, zu überprüfen, ob der GET-Parameter *_escaped_fragment_* enthalten ist. Tritt dieser Fall ein, wird die Anfrage an den Seoxy, welche im gezeigten Beispiel auf Port *8000* gestartet ist, weitergeleitet. Ist der Parameter nicht gesetzt, wird die Anfrage an den Orestes-Server auf Port *9000* geleitet.

```
1  var seoxy = http.createServer().listen(8000);
2  var proxy = new httpProxy.RoutingProxy();
3  seoxy.on('request', function(req, res) {
4    var urlParts = url.parse(req.url, true);
5    var path = urlParts.path;
6    var query = urlParts.query;
7    if(query.hasOwnProperty('_escaped_fragment_')) {
8      path = path.replace(/(&|\?){1}_escaped_fragment_=/, '');
9      res.writeHead(200, { 'Content-Type': 'text/html' });
10     toHtml(path, stdTTL, function(data) {
11       res.write(data);
12       res.end();
13     });
14   } else {
15     proxy.proxyRequest(req, res, {
16       host: 'localhost',
17       port: 9000
18     });
19   }
20 });
```

Listing 5.10: Seoxys Einstiegspunkt

Abbildung 5.10 zeigt den Einstiegspunkt des Seoxys. Es wird zunächst ein HTTP-Server auf Port *8000* gestartet (Zeile 1). Findet ein Request statt, wird das URL-Objekt erstellt, um anschließend die URI auszulesen (Zeile 4). Daraufhin wird erneut geprüft, ob der Parameter *_escaped_fragment_* gesetzt ist (Zeile 7). Ist dies der Fall, wird die Funktion *toHtml* aufgerufen (Zeile 10). Mittels eines regulären Ausdrucks wird der Parameter *_escaped_fragment_* entfernt, da dieser kein Teil der URI ist (Zeile 8). Ist der Parameter nicht gesetzt, wird die Anfrage an den Orestes-Server weitergeleitet (Zeile 15). Diese Prüfung erfolgt, damit der Seoxy auch ohne einen Reverse-Proxy verwendet werden kann und Anfragen, die statischen Inhalt wie Bilder oder Stylesheets anfragen, dürfen nicht an den PhantomJS geleitet werden.

```
1  var seoxyCache = new cache({stdTTL: stdTTL, checkperiod: 120});
2  var cache = require('node-cache');
3  var domain = 'http://localhost';
4  var toHtml = function(path, ttl, success) {
5    ttl = ttl == undefined ? stdTTL : ttl;
6    seoxyCache.get(path, function(err, value) {
7      if(!value[path] || err) {
8        var localUrl = domain + path;
9        phRun(localUrl, function(html) {
10         seoxyCache.set(path, html, ttl);
11         success(html);
12       });
13     } else {
14       success(value[path])
15     }
16   });
17 };
```

Listing 5.11: Methode zur Generierung der HTML-Datei

Der Seoxy speichert, die durch PhantomJS generierten Seiten, in einem *In-Memory Key-Value-Store* zwischen. Dafür wird die Drittanbieterbibliothek *node-cache* verwendet. Diese ermöglicht das Erstellen und Auslesen von Cache-Einträgen. Das Zwischenspeichern von generierten HTML-Seiten ist notwendig, da der PhantomJS abhängig von der Komplexität Zeit zum Erstellen der Webseite benötigt. Google zieht in seine Bewertung die Auslieferungsgeschwindigkeit mit ein [42]. Daher ist es unabdingbar, dass die Anfrage schnellstmöglich mit der gewünschten Ressource beantwortet werden kann. Für die Identifizierung eines Cacheeintrages wird die *URI* verwendet.

Der *toHtml*-Funktion (Abbildung 5.11) werden URI, die Speicherungszeit für den Cache sowie eine Funktion, die nach Beendigung ausgeführt wird (*Callback*), übergeben. Ist keine Vorhaltezeit gesetzt, wird die Standardzeit, die in der Variablen *stdTTL* definiert ist, verwendet (Zeile 5). Mittels der Methode *seoxyCache.get* kann ein Cacheeintrag ausgelesen werden (Zeile 6). Durch den ersten Parameter wird der *Key* gesetzt. Dem zweiten Parameter wird ein Callback übergeben. Dieser wird ausgeführt, sobald die *Get*-Methode beendet wurde. Ihr werden Wert sowie im Fehlerfalle die zugehörige Beschreibung mitgegeben. Dabei ist zu beachten, dass der Callback auch ausgeführt wird, wenn kein Wert gefunden wurde. Ist der Wert nicht gesetzt oder es wurde eine Fehlermeldung ausgegeben (Zeile 9), wird die *phRun*-Methode aufgerufen. Dieser kann ebenfalls ein Callback übergeben werden. Wurde die HTML-Seite generiert, wird diese im Cache zwischengespeichert (Zeile 10). Dafür wird die Methode *seoxyCache.set* verwendet. Dieser wird *Key, Value* sowie die Zeit bis zur Invalidierung übergeben. Wird hingegen die passende HTML-Datei gefunden, wird diese direkt ausgegeben (Zeile 14).

```
1  var phantom = require('phantom');
2  var phRun = function(url, cb) {
3    var portscanner = require('portscanner');
4    portscanner.findAPortNotInUse(40000, 60000, 'localhost', function(err
         , freeport) {
5      phantom.create({ port: freeport }, function(ph) {
6        ph.createPage(function(page) {
7          page.open(url, function () {
8            phPageEvaluate(page, cb, ph);
9          });
10       });
11     });
12   });
13 };
```

Listing 5.12: Kapselung der Erstellung des PhantomJS-Prozesses

Der Funktion *phRun* kann die URL sowie einen Callback übergeben werden (Zeile 2). Der PhantomJS-Prozess wird auf einem definierbaren Port gestartet. Über diesen findet die Kommunikation zwischen Node.js und PhantomJS statt. Für jede Seitengenerierung wird ein neuer Prozess gestartet. Da ein Port nur von einem zurzeit verwendet werden kann, muss vor der Ausführung von PhantomJS geprüft werden, welche Ports zur Verfügung stehen. Dafür wird das Node.js Modul *portscanner* verwendet. Dieses stellt die Funktion *findAPortNotInUse* zur Verfügung. Dieser wird ein Start- und Endport sowie eine Funktion übergeben, die nach Ende der Suche ausgeführt wird. Das Modul prüft alles Ports zwischen Start- und Endport, diese sind im gezeigten Beispiele alle Ports zwischen 40000 und 60000, und beendet die Suche sobald ein freier Port gefunden wurde (Zeile 4). Diese Methode ist nicht optimal, da die Anzahl von gleichzeitigen Seitengenerierungen begrenzt wird. Für einen produktiven Einsatz muss daher gegebenenfalls eine andere Methode gewählt werden. Zum Beispiel die Wiederverwendung eines PhantomJS-Prozesses, dafür muss allerdings geprüft werden, ob dieser die parallele Generierung von Seiten unterstützt.

Wurde ein freier Port gefunden, wird zunächst der Prozess über *phantom.create* erstellt (Zeile 5). Diesem werden ein Callback sowie der Port, auf dem der Prozess gestartet wird, übergeben. Im Callback wird das *page*-Objekt erstellt (Zeile 6). Mittels diesem wird die Seite geladen. Die Evaluierung ist in der Funktion *phPageEvaluate* ausgelagert (Abbildung 5.13). Diese erhält das *page*-Objekt, das PhantomJS-Objekt (*ph*) sowie den Callback (Zeile 8). Bei diesem handelt es sich um den der Funktion *toHtml*, der bis zur *phPageEvaluate*-Funktion durchgereicht wird.

```
1  var phPageEvaluate = function(page, ph, cb, timeout, count) {
2    count = count || 0;
3    if(count >= 10) {
4      return;
5    }
6    setTimeout(function() {
7      page.evaluate(function () {
8        var $head = $('head');
9        $head.find('noscript[data-type="seoxy"]').remove();
10       if(!$('link[rel="canonical"]').length) {
11         $head.append(
12           '<link rel="canonical" href="'+ window.location.href +'" />'
13         );
14       }
15       $('a').attr('href', function(i, h) {
16         return h + (~h.indexOf('?')
17           ? '&_escaped_fragment_=' : '?_escaped_fragment_=');
18       });
19       $head.find('meta[name="fragment"][content="!"]').remove();
20       if($('body').attr('data-status') === 'ready') {
21         return $('html').html();
22       } else {
23         return null;
24       }
25     }, function (result) {
26       if(result) {
27         ph.exit();
28         cb(result);
29       } else {
30         phPageEvaluate(page, ph, cb, 1000, ++count);
31       }
32     });
33   }, timeout || 0);
34 };
```

Listing 5.13: Kapselung der Erstellung des PhantomJS-Prozesses

Da die benötigten Daten der zu generierenden Ressource asynchron aus einer Datenbank geladen werden, muss vor der Rückgabe der HTML-Datei geprüft werden, ob sie bereits vollständig erstellt wurde. Dafür muss der Smart Client so geschrieben sein, dass an dem *body*-Element das Data-Attribut *status* auf den Wert *ready* gesetzt wird, nachdem die Seite vollständig geladen wurde. Dieses Attribut wird von dem Seoxy geprüft und die Seite wird erst ausgeliefert, wenn das Attribut gesetzt ist. Die Funktion ruft sich rekursiv auf, bis das Attribut gesetzt ist oder die Anzahl der Versuche überschritten ist. Im gezeigten Beispiel ist die Anzahl der Iterationen auf zehn begrenzt (Zeile 3).

Der Funktion werden beim Aufruf aus der *phRun*-Methode, die Parameter URI, Port und Callback übergeben (Zeile 1). Die restlichen Parameter dienen für den rekursiven Aufruf. Wenn der Variable *count* keinen Wert übergeben wurde, handelt es sich um den initialen Aufruf der Funktion. Die Variable wird somit auf 0 gesetzt (Zeile 2). Sie definiert die Anzahl der rekursiven Aufrufe.

Da das Laden der Daten eine unbestimmte Zeit in Anspruch nehmen kann, ist es sinnvoll, zwischen den rekursiven Aufrufen zu warten. Ansonsten besteht die Möglichkeit, dass die Daten noch geladen werden und die maximale Anzahl der Versuche bereits durchlaufen wurde. Um eine Zeit zwischen den Aufrufen zu warten, wird die JavaScript-Methode *setTimeout* verwendet (Zeile 6). Dieser Methode kann eine Funktion und eine Wartezeit in *ms* übergeben werden. Sie ruft die angegeben Funktion erst nach Ablauf der definierten Wartezeit auf. Im Beispiel wird die Wartezeit über den Parameter *timouet* definiert. Dieser ist beim initialen Aufruf nicht gesetzt, was zur Folge hat, dass die erste Iteration instantan durchgeführt wird (Zeile 33). Erst beim rekursiven Aufruf, wird eine Wartezeit gesetzt. Im Beispiel beträgt diese *1000ms* (Zeile 30).

Der Funktion *evaluate* des *page*-Objektes können zwei Funktionen übergeben werden. Die erste beinhaltet JavaScript, welches nach dem Laden des Smart Clients ausgeführt wird. Bei der zweiten handelt es sich um einen Callback, der nach Beendigung der ersten übergebenen Funktion gestartet wird (Zeile 7).

Wie im vorherigen Verlauf dieses Kapitel erörtert, wird der Redirect für Browser ohne JavaScript-Funktionalität innerhalb eines *noscript*-Elementes durchgeführt. Dieses muss vor Auslieferung der generierten HTML-Seite entfernt werden. Ansonsten würde ein Redirect-Schleife entstehen und die Seite kann nicht angezeigt werden. Zur Entfernung des Elementes, welches sich im HTML-Header befindet, wird zunächst das *head*-Element mithilfe der JavaScript-Bilbiothek *jQuery* geladen (Zeile 8). Innerhalb dieses *head*-Elementes wird das *noscript*-Element mit dem Attribut *data-type* und dem Wert *seoxy* gesucht und anschließend entfernt, damit kein erneuter Redirect durchgeführt wird (Zeile 9).

Mittels der Angabe eines kanonischen Links (engl. *Canonical Link*) kann definiert werden, unter welcher URL der Google Crawler den geladenen Inhalt zuordnet. Dies ist im Besonderen wichtig, wenn Inhalt über unterschiedliche URLs erreichbar ist, da der Crawler ansonsten willkürlich entscheidet, welche URL zu welchem Inhalt zugeordnet wird. Des Weiteren zieht der Crawler in seiner Bewertung negativ mit ein, ob gleiche Daten unter unterschiedlichen URLs vorhanden sind. Dies kann durch die Angabe eines Canonical Links verhindert werden. Abbildung 5.14 zeigt die Definition eines solchen Links. Über das Attribut *href* wird der Link angegeben. Der Seoxy prüft, ob bereits ein Canonical Link definiert wurde, damit keine doppelte Definition des Canonical Links stattfindet (Zeile 10). Ist dies nicht der Fall, generiert der Seoxy diesen. Er beinhaltet die URL der aufgerufenen Seite. Diese wird mittels der JavaScript-Funktion *window.location.href* ausgelesen. Das HTML-Element wird anschließend zum HTML-Header hinzugefügt (Zeilen 11 bis 13).

```
1 <link rel="canonical" href="http://www.orestes.info/about" />
```

Listing 5.14: Definition eines Canonical Links

In den Zeilen 15 bis 18 werden alle Links, die sich auf der Seite befinden, editiert. Es wird der Parameter *_escaped_fragment_* hinzugefügt, damit beim Aufruf einer Unterseite kein erneuter Redirect durch den Crawler durchgeführt werden muss. Dafür wird ebenfalls das Meta-Element mit dem Namen *fragment*, welches Google auffordert, die Seite erneut mit dem besagten Parameter zu laden, entfernt (Zeile 19). Ein weiterer Grund ist, dass Browser ohne JavaScript-Unterstützung auch auf nachfolgenden Unterseiten erkannt werden müsse.

In Zeile 20 wird geprüft, ob das *data-status*-Attribut auf den Wert *ready* gesetzt ist. Für das Ändern des Status ist der Smart Client verantwortlich. Ist dies der Fall, wird der Inhalt der HTML-Datei als String zurückgegeben (Zeile 21). Ansonsten wird *null* ausgegeben.

Der zweiten Funktion, die der Methode *page.evaluate* übergeben wird, erhält das Ergebnis der ersten als Parameter. Im Beispiel wird der Inhalt der HTML-Seite in der Variablen *result* gespeichert. Wenn diese nicht leer bzw. *null* enthält, wird der PhantomJS-Prozess beendet (*pg.exit()*), anschließend wird der Callback, der in der *toHtml*-Funktion übergeben wurde, aufgerufen. Dieser erhält die generierte HTML-Seite.

Ist die Variable *result* nicht gesetzt, wird die *phPageEvaluate*-Funktion erneut aufgerufen. Der Iterationszähler wird erhöht und der Wert, welcher der *setTimeout*-Methode übergeben wird, auf *1000ms* gesetzt. Der Prozess der Evaluierung startet somit eine Sekunde später erneut.

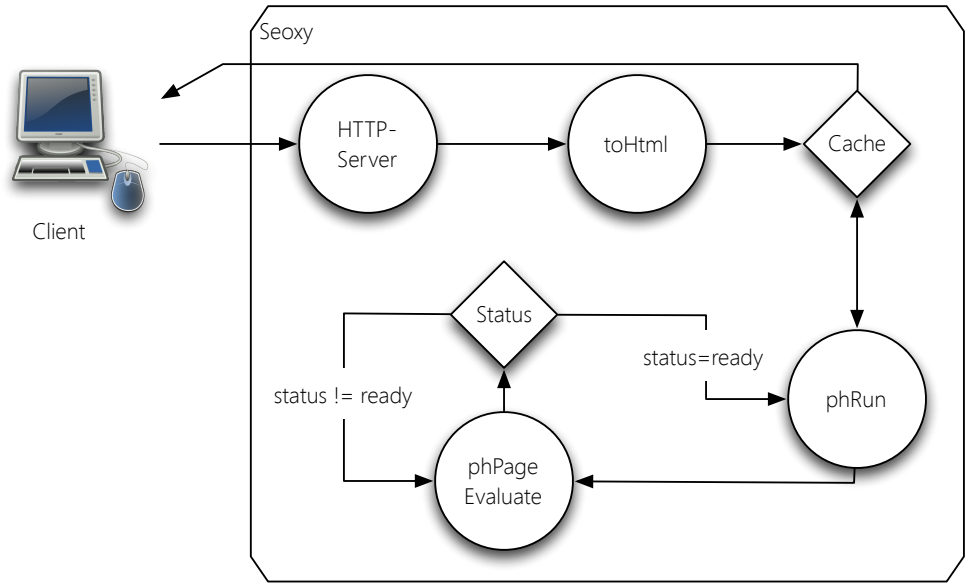


Abbildung 5.3: Interner Ablauf des Seoxys

Abbildung 5.3 zeigt zusammenfassend den Funktionsablauf des Seoxys. Es wird eine Anfrage an den HTTP-Server gestellt. Dieser ruft die *toHtml*-Funktion auf. Diese Prüft,

ob die URI sich bereits im Cache befindet. Ist dies der Fall, wird der Request mit dem Inhalt des Caches beantwortet. Wurde die Seite nicht zwischengespeichert, wird die *phRun*, welche den PhantomJS startet, aufgerufen. Die Funktion *phPageEvaluate* wird rekursiv ausgeführt bis der *Status* den Wert *ready* hat. Ist die Seite generiert, wird sie mittels des durchgereichten Callbacks zurückgegeben.

### 5.2.3 Optimierung

Da die Erstellung der HTML-Seite abhängig von der Komplexität eine gewissen Zeit in Anspruch nimmt und Google die Ladegeschwindigkeit mit in seine Bewertung aufnimmt, muss ein Weg gefunden werden, um die Generierung zu beschleunigen. Der Seoxy speichert bereits, wie beschrieben, generierte Daten nach einem Request zwischen. Es kann aber auch der Fall eintreten, dass der Google Crawler eine URI aufruft, die bisher noch nicht erstellt wurde oder aufgrund des Alters der Erstellung wieder aus dem Cache entfernt wurde. In diesen Fällen würde die Generierung der Seite erst nach dem Request des Crawlers ausgeführt werden. Die Antwort zu diesem würde erheblich länger dauern und Google würde die Bewertung der Seite entsprechend herabsetzen.

Daher wird der Seoxy periodische aufgerufen, um URIs zu generieren und im Cache zu speichern. Um dies durchzuführen, muss dieser automatisiert alle Seiten, die im Suchmaschinenindex aufgenommen werden sollen, ausführen. Eine Möglichkeit ist, dass der Seoxy die *index.html* aufruft und rekursiv allen Links folgt, die zu finden sind. Dies kann aber zu einer unvollständigen Generierung führen, da gegebenenfalls Unterseiten nicht direkt verlinkt sind. Zum Beispiel bei einem Gewinnspiel, welches nur über *Facebook* beworben wird. Da auch ein Suchmaschinenbot keine Seiten finden kann, die nicht verlinkt sind, wurde am 16. November 2006 das *Sitemaps-Protokoll* von *Google, Microsoft* und *Yahoo!* veröffentlicht [28]. Dieses definiert eine Möglichkeit, um den Crawlern auf Links hinzuweisen. Für die Verwendung muss eine eine Datei erstellt werden, die zu indizieren Verlinkungen enthält. Es kann sich dabei um Textdatei handeln, die pro Zeile eine URL enthält, oder eine *XML*-Datei[4]. Im nachfolgenden wird auf die XML-Datei eingegangen, da diese mehr Optionen bietet.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.sitemaps.org/schemas/sitemap/0.9
5   http://www.sitemaps.org/schemas/sitemap/0.9/sitemap.xsd">
6   <url>
7    <loc>http://orestes.info/</loc>
8    <lastmod>2013-07-30</lastmod>
9    <changefreq>daily</changefreq>
10   <priority>0.8</priority>
11  </url>
```

[4] *Extensible Markup Language* ist eine Auszeichnungssprache zur Erstellung von strukturierten Dateien.

```
12  <url>
13   <loc>http://orestes.info/login/</loc>
14   <lastmod>2013-07-30</lastmod>
15   <changefreq>daily</changefreq>
16   <priority>0.8</priority>
17  </url>
18 </urlset>
```

Listing 5.15: Aufbau einer *sitemap.xml* [67]

Abbildung 5.15 zeigt den Aufbau einer Seitemap im XML-Format [67]. Es wird ein *urlset* erstellt. Bei diesem handelt es sich um ein Array, welches mehrere *url*-Elemente enthält. Ein *url*-Element besteht aus dem Attribut *loc*, welches die URL angibt. Folgende Elemente und Attribute können definiert werden, wobei die als optional markierten Eigenschaften dem Crawler lediglich Metainformationen der definierten URIs liefern.

***urlset*:** Bei diesem Element handelt es sich um ein Array, welches mehrere *url*-Elemente enthält.

***url*:** Ein *url*-Element wird aus folgenden Attributen zusammengesetzt:

- ***loc*:** Gibt die zu durchsuchende URL an.
- ***lastmod*:** Gibt an, wann die letzte Änderung an der Ressource durchgeführt wurde. Der Crawler kann somit entscheiden, ob er bereits die aktuelle Version verwendet.
- ***changefreq*:** Mittels dieses Attributes lässt sich definieren, wie oft der Crawler eine Ressource aufruft. Die Werte *always, hourly, daily, weekly, monthly, yearly* und *never* sind möglich. Wobei *never* nicht bedeutet, dass die URI gar nicht betrachtet wird, sondern nur nach der ersten Aufnahme nicht weiter aktualisiert wird. *always* gibt an, dass die Seite bei jedem Besuch des Crawler aktualisiert wird,
- ***priority*:** Es kann angegeben werden, wie *wichtig* URIs im vergleich zu anderen sind. Der Wertebereich geht dabei von 0 bis 1, wobei 1 die höchste Priorität bedeutet. Der Standardwert liegt bei $0,5$.

Da in der Regel eine *Sitemap* Verwendung findet, wenn es sich um eine für Suchmaschinen optimierte Seite handelt und sich der Seoxy an diese Zielgruppe wendet, bietet es sich an, dass dieser zur Generierung der Ressourcen ebenfalls die *Sitemap* verwendet.

```
1  var generate = function() {
2    var parser = require('xml2json');
3    fs.readFile('sitemap.xml', function(err, data) {
4      var jsonString = parser.toJson(data.toString());
5      var json = JSON.parse(jsonString);
6      var urlSet = json.urlset;
7      var urls = urlSet.url;
8      for(i in urls) {
9        var urlSitemap = urls[i];
10       var loc = urlParser.parse(urlSitemap.loc);
11       toHtml(loc.path);
12     }
13   });
14 };
```

Listing 5.16: Erzeugung von HTML anhand der *Sitemap*

Abbildung 5.16 zeigt die Funktion, die für das Auslesen der *sitemap.xml* verantwortlich ist. Es wird das Modul *xml2json* verwendet. Dies generiert aus einem String, welcher Daten im XML-Format beinhaltet, ein JSON. Da *xml2json* das JSON als String ausgibt, wird anschließend aus diesem mittels *JSON.parse* ein JavaScript-Objekt erstellt. In diesem das *urlset* ausgelesen werden. Dieses beinhaltet die in der Sitemap definierten URLs. Für jede URL wird die *toHTML*-Funktion aufgerufen. Somit generiert der PhantomJS die HTML-Dateien. Anschließend werden sie im Cache zwischengespeichert.

Abbildung 5.4 zeigt den Ablauf Seoxys unter Verwendung der Generierung von HTML-Seiten anhand der *sitemap.xml*.

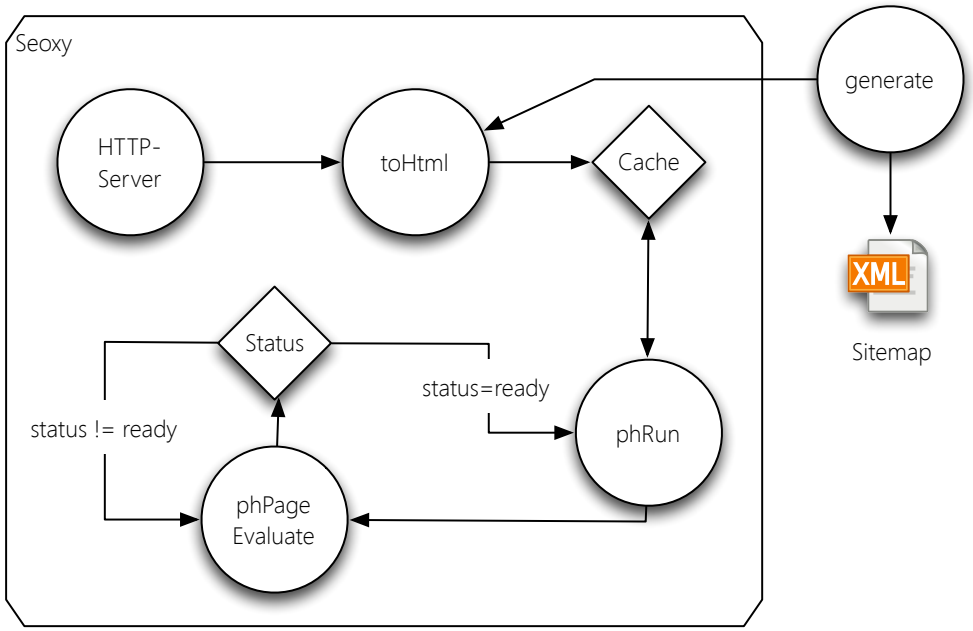


Abbildung 5.4: Interner Ablauf des Seoxys unter Einbezug der Sitemap

In einer späteren Version, die in dieser Arbeit nicht betrachtet wird, ist es denkbar, dass über die *sitemap.xml* Metainformationen für den Seoxy angegeben werden können. Eine Option ist die Vorhaltezeit des Caches. Dies ist zum Beispiel für Newsseiten, die stets den aktuellsten Inhalt an die Crawler übermitteln möchten, nötig. Dabei ist jedoch zu beachten, dass innerhalb des Seoxys eine Liste aller URIs mit den zugehörigen Metainformationen gehalten werden muss, da zum Beispiel auch ein Cacheeintrag nach einer Anfrage erstellt wird. Diese Funktion benötigt somit die Informationen, ob die HTML-Seite zwischengespeichert werden darf. Es ist somit notwendig, dass nach jeder Änderung der Sitemap diese erneut geladen wird.

### 5.2.4 Evaluation

Folgende Evaluation wird anhand des Prototypens, der im weiteren Verlauf dieser Arbeit vorgestellt wird, durchgeführt.

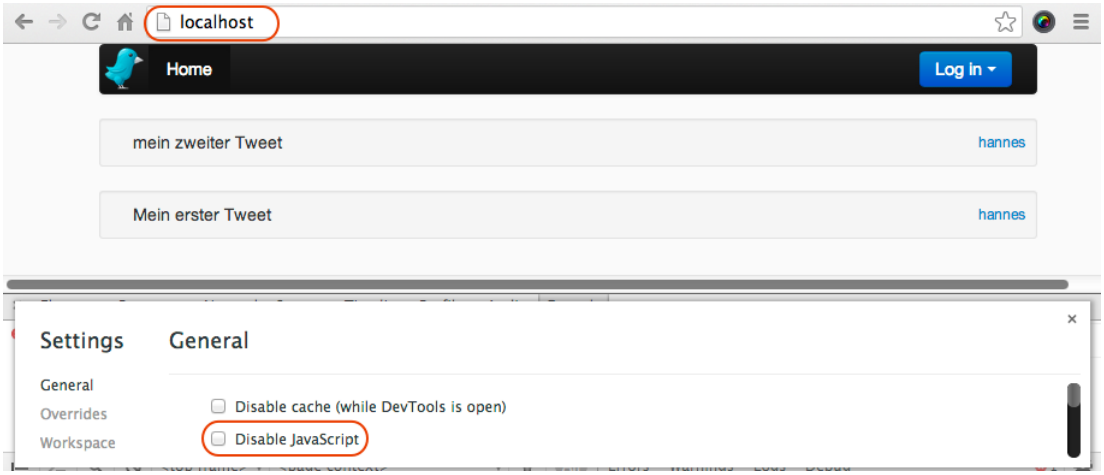


Abbildung 5.5: Aufbau von Rendr (Quelle rendr blog post)

Abbildung 5.5 zeigt den Aufruf des Smart Clients mit aktiviertem JavaScript. An den rot markierten Stellen ist zu erkennen, dass er ohne *_escaped_fragment_*-Parameter sowie mit aktiviertem JavaScript aufgerufen wird. Die Abbildung zeigt somit die Referenzdarstellung des Smart Clients.

```
1  <!doctype html>
2  <html>
3    <head>
4      <noscript>
5          <meta http-equiv="Refresh" content="1;url=http://localhost/?
               _escaped_fragment_">
6      </noscript>
7      <meta charset="utf-8">
8      <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
9      <title></title>
10     <meta name="description" content="">
```

```
11      <meta name="viewport" content="width=device-width">
12      <link rel="stylesheet" href="/styles/4711fa63.main.css">
13    </head>
14    <body ng-app="twitterApp" data-status="{{ status }}">
15      <div class="container" ng-view=""></div>
16      <script src="/scripts/5c992a3f.scripts.js"></script>
17    </body>
18  </html>
```

Listing 5.17: Übertragener HTML-Sourcecode bei Aufruf des Smart Clients

Listing 5.17 zeigt die zugehörige HTML-Datei, die beim initialen Aufruf geladen wird, zu Abbildung 5.5. Es ist kein Inhalt vorhanden, da dieser erst durch die Ausführung der *script.js* aus der Datenbank geladen und in den DOM eingefügt wird.


Abbildung 5.6: Aufbau von Rendr (Quelle rendr blog post)

Abbildung 5.6 zeigt den Aufruf mit deaktiviertem JavaScript. Das Symbol oben rechts gibt an, dass die Seite ohne JavaScript aufgerufen wurde. Wie zu sehen ist, wird die Seite identisch zu Abbildung 5.5 dargestellt. Der PhantomJS hat den Smart Client somit erfolgreich generiert und ausgeliefert. Dabei ist zu beachten, dass JavaScript, welches durch eine Interaktion vom Benutzer ausgeführt wird (z.B. öffnen des Login-Menus), nicht funktionsfähig ist.

Befindet sich die HTML-Seite nicht im Cache des Seoxys, benötigt das Laden des HTML-Dokumentes bei zehn ausgeführten Messungen im Durchschnitt 962 Millisekunden. Der PhantomJS läuft dabei auf einem *Intel Core i7 2.6GHz* aus dem Jahre 2012. Die Netzwerklatenz hat auf die Messung keinen Einfluss, da Browser und Seoxy auf demselben Rechner ausgeführt wurden.

Ist die Ressource bereits im Cache vorhanden, benötigt das Laden bei 10 Messungen im Durchschnitt 7 Millisekunden. Dies zeigt, dass die Generierung der Seite durch PhantomJS im Schnitt ca. 955 Millisekunden beansprucht. Diese Messungen zeigen, dass die Vorgenerierung und Zwischenspeicherung im Cache ein wichtige Bestandteile des Soexys sind.

Im nachfolgenden Unterkapitel wird eine bereits bestehende Lösung diskutiert.

## 5.3 Rendr

Bei *Rendr* handelt es sich um eine JavaScript-Bibliothek, die von *airbnb* entwickelt wird. *airbnb* stellt einen webbasierten Marktplatz für weltweite Buchung und Vermietung von Unterkünften zur Verfügung [13][4].

Die mobile Webseite von airbnb ist als Smart Client realisiert. Das verwendete JavaScript-Framework ist *Backbone.js*. Mit Rendr versucht airbnb, die Probleme, die durch die Verwendung von Smart Clients entstehen und im vorherigen Kapitel 5.2 erörtert werden, zu lösen. Da der airbnb Smart Client auf Basis von *Backbone.js* entwickelt wird, setzt Rendr die Verwendung von diesem voraus. Dies hat den Nachteil, dass es sich somit um keine allgemeingültige Lösung für Smart Clients handelt. Sie kann damit nicht für Smart Clients, die zum Beispiel auf Angular.js aufbauen, verwendet werden.

Rendr basiert, wie der *Seoxy*, auf Node.js. Es verwendet zur Erstellung der HTML-Seite nicht PhantomJS. Es basiert auf dem *Web-Application-Framework Express*.

Abbildung 5.7 zeigt den Aufbau. Der Server, im weiteren Verlauf als Applikationsserver bezeichnet, sowie der Client teilen sich einen Bereich der Logik des Smart Clients. Diese Schnittmenge lässt sich sowohl auf dem Server als auch auf dem Client ausführen. Der Server stellt zusätzlich Funktionalitäten zum Logging sowie zur Auslieferung von statischen Dateien zur Verfügung. Der Client beinhaltet Elemente zum Zwischenspeichern von Benutzerdaten (*localStorage*[5]) und Logiken zur Behandlung von Benutzereingaben (*user events*).

Server sowie Client greifen auf dieselbe API zu. Die API ist im Kontext der Masterarbeit der Orestes-Server.

Ruft ein Browser eine URL auf, wird in jedem Fall die Seite zunächst auf dem Applikationsserver erstellt. Es handelt sich somit beim initialen Aufruf um eine klassische 3-Tier-Architektur. Erst beim weiteren Navigieren kommuniziert der Smart Client direkt mit der API und verhält sich wie in einer 2-Tier-Architektur. Dies hat den Vorteil, dass der Smart Client beim initialen Laden schneller angezeigt wird, da das Document-Object-Model bereits vom Server erstellt wurde und keine weiteren Requests an die API durchgeführt werden müssen. Dies wirkt sich im Besonderen auf mobilen Geräten aus, da der Verbindungsaufbau durch die Verwendung von Mobilfunkstandards wie zum Beispiel *UMTS* langsamer ist und durch die Vorgenerierung weniger Anfragen benötigt werden. Der Smart Client ist so aufgebaut, dass bei der Navigation mit aktiviertem JavaScript die URLs per *pushState* erzeugt und die Daten direkt aus der API geladen werden. Ist hingegen kein JavaScript aktiv, handelt es sich beim Aufruf eines Links um eine URL, die von dem Applikationsserver beantwortet wird. Dies hat den Vorteil, dass keine Logik

[5] Bei *localStorage* handelt es sich um eine HTML5-API, welche das Speichern von Daten im Browser erlaubt. Die Verweildauer ist dabei unabhängig von der Session.

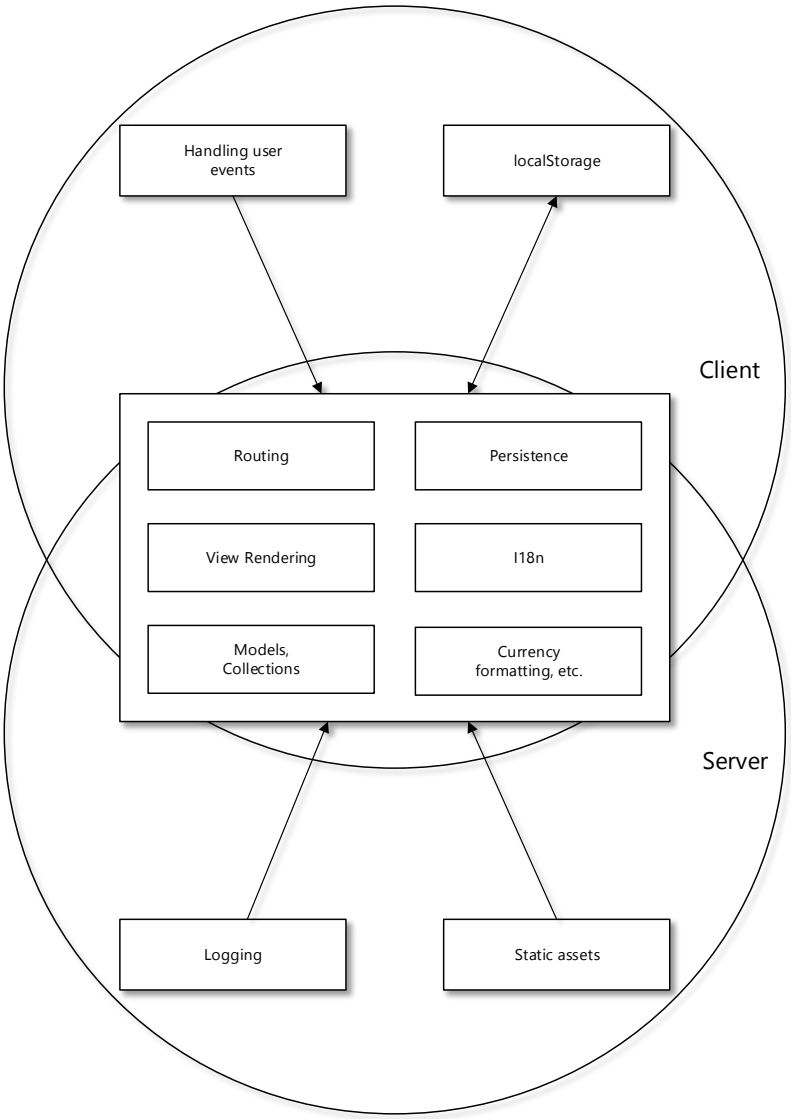


Abbildung 5.7: Aufbau von Rendr [12]

vorhanden sein muss, die entscheidet, ob die Seite generiert werden muss oder nicht. Dies passiert implizit durch das Verhalten des Browsers. Ein Nachteil ist hingegen, dass jeder Benutzer, der die Seite anfragt zunächst vom Applikationsserver bedient wird. Es müssen somit Serverkapazitäten geschaffen werden, um diese Anfragen beantworten zu können.

## 5.4 Fazit

Der in dieser Arbeit entwickelte Seoxy ermöglicht die Erstellung von suchmaschinenoptimierten Smart Clients. Er arbeitet dabei unabhängig des verwendeten JavaScript-Frameworks. Durch die Verwendung eines Caches und die periodische Generierung der

HTML-Seiten und die damit einhergehende Verbesserung der Antwortzeit entsteht ein weiterer Vorteil, der sich positiv auf die Bewertung von Suchmaschinen gegenüber des Smart Clients äußert.

Aktuelle Lösungen wie zum Beispiel *Rendr* gehen einen anderen Weg und bauen weiterhin auf einer 3-Tier-Architektur auf, was höhere Kosten gegenüber einer 2-Tier-Architektur bedeutet und somit ein großer Vorteil von Smart Clients verloren geht. Gegenüber dem Seoxy, die diese Lösung bietet ist die gesteigerte Ladegeschwindigkeit der Startseite, insbesondere auf mobilen Geräten.

Im nachfolgenden Kapitel wird der in dieser Arbeit entwickelte Prototyp erläutert.

# 6 Prototyp

Im Folgenden wird ein Prototyp auf Basis des Use-Cases aus Kapitel 1.2 vorgestellt. Da Orestes noch nicht alle nötigen Funktionen (Authentifizierung, Autorisierung, Event-Server), die für eine Anwendung, wie er im Use-Case beschrieben wird, zur Verfügung stellt, wird dieser auf Basis von *CouchDB* entwickelt. Eine featurereduzierte Version der Anwendung wird anschließend auf Basis von Orestes entwickelt. Der Smart Client verwendet dabei das Framework AngularJS.

## 6.1 Funktionen

Auf dem in diesem Kapitel entwickelten Smart Client ist es möglich, sich zu registrieren, anzumelden und Kurznachrichten (*Tweets*) zu schreiben. Auf der Startseite werden die Kurznachrichten aller Benutzer gezeigt. Jeder Benutzer hat des Weiteren noch eine eigene Seite, die per Link aufrufbar ist. Diese enthält nur seine Nachrichten. Ein Tweet kann direkt über einen Link referenziert werden. Schreibt ein Benutzer eine neue Nachricht, werden alle weiteren aktiven Applikation mittels *Server-Sent Events* benachrichtigt, damit diese die entsprechende Laden können.


Abbildung 6.1: Startseite des Prototypens

Abbildung 6.1 zeigt die Startseite. Es sind momentan drei Nachrichten vorhanden. Sie wurden von den Benutzern *hannes* und *felix* erstellt. Aktuell ist der Benutzer *hannes* angemeldet. Der Symbol neben einer Nachricht verlinkt auf diese. Mittels diesem ist es möglich, den Tweet direkt zu öffnen. Der Anwendername rechts verlinkt auf die jeweilige Benutzerseite. Der blaue Button ermöglicht neben der Registrierung auch die Anmeldung. Das Eingabefeld (*Tweet*) dient zum Schreiben von Nachrichten.


Abbildung 6.2: Benutzerseite

In Abbildung 6.2 ist eine Benutzerseite dargestellt. Es werden nur noch die Nachrichten von *hannes* gezeigt. Des Weiteren ist in der Adresszeile zu erkennen, dass HTML5-URLs verwendet werden.

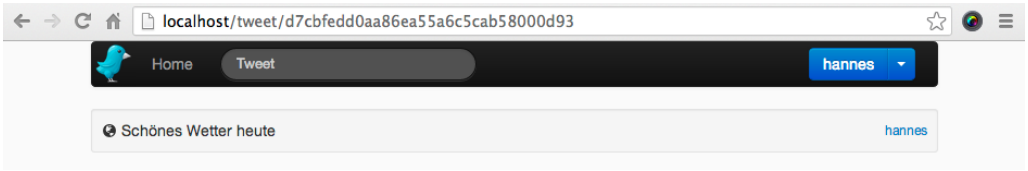


Abbildung 6.3: Link zu einem Tweet

Wird ein Dokument in der CouchDB erstellt, erhält dieses eine ID, welche pro Datenbank einzigartig ist. Diese ID wird verwendet, um den Tweet zu referenzieren und ist in der Addresszeile von Abbildung 6.3 zu erkennen.

## 6.2 Implementierung

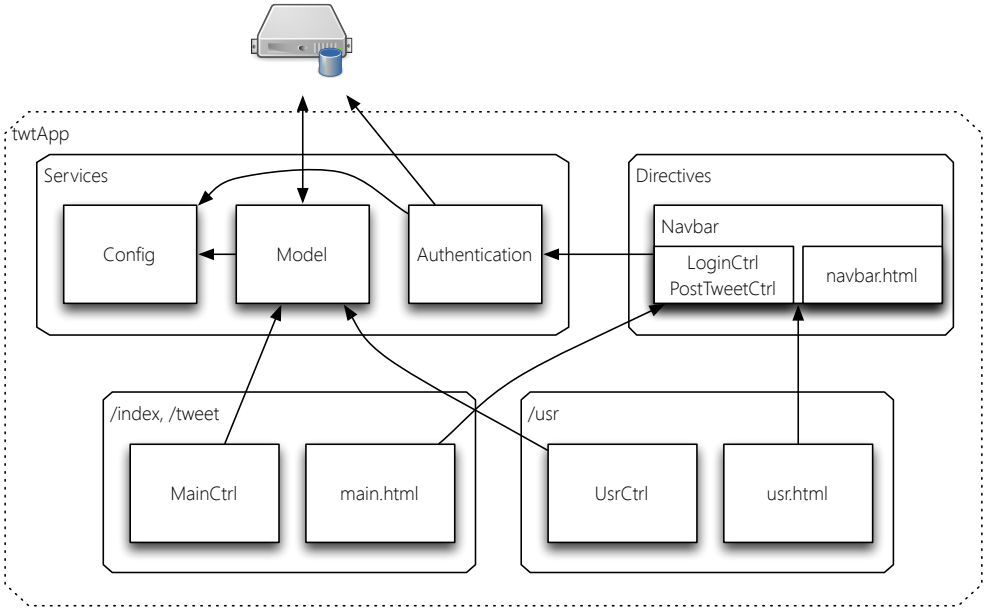


Abbildung 6.4: Aufbau des Smart Clients

In Abbildung 6.4 wird der Aufbau der Applikation gezeigt. Es werden Services, Directives sowie Controller und Views zur Erstellung von Ressourcen zur Verfügung gestellt. Wird der Smart Client aufgerufen, wird der Controller *MainCtrl* verwendet. Dieser lädt das Model aus der Datenbank. Dafür wird der *Model-Service* verwendet. Dieser kapselt den Zugriff auf die zugrundeliegende Datenbank. Ein Austausch dieser wird somit erleichtert. Neben dem Laden von Objekten bietet der Model-Service die Möglichkeit, auf einen Event-Server zu verbinden. Bei Serviceobjekten handelt es sich um *Singletons*[1]. Die geladenen Daten werden der View *main.html* übergeben. *UsrCtrl* und *usr.html* verhalten sich ähnlich. Es wird zusätzlich eine Benutzername benötigt und es werden nur die Dokumente für diesen geladen. Neben dem Model zeigen die Views *usr.html* und *main.html* auch die Navigationsleiste an, welche den Login-Button und das Eingabefeld zum Erstellen von Nachrichten enthält. Diese ist als Directive realisiert. Dieses Directive enthält einen Controller und die View *navbar.html*. Die Controller *LoginCtrl* und *PostTweetCtrl* werden innerhalb *navbar*-View verwendet. Einem Controller muss keine HTML-Datei zugewiesen sein. Es ist möglich, einen Controller einem HTML-Element mittels des Attributes *ng-controller* zu zuweisen. Ein Beispiel dafür ist in Listing 6.1 gegeben. Durch das Attribut wird ein neuer Scope erstellt. Dieser ist an de *PostTweetCtrl*-Controller gebunden. Die Funktion *postTweet* wird daher zunächst in diesem gesucht.

```
1 <form class="navbar-search" ng-hide="!$root.user.loggedIn" ng-
    controller="PostTweetCtrl" ng-submit="postTweet()">
2   <input type="text" class="search-query transall" ng-model="tweetBar"
      placeholder="Tweet">
3 </form>
```

Listing 6.1: Zuweisung eines Controllers zu einem HTML-Element

Der *PostTweetCtrl*-Controller wird zur Erstellung einer Kurznachricht verwendet. Der *LoginCtrl*-Controller übernimmt die Aufgaben zur An- und Abmeldung. Das *Navbar*-Directive greift dafür auf den *Authentication*-Service zu. Der Authentication-Service wird verwendet, um zu prüfen, ob der Benutzer an der Datenbank angemeldet ist. Der Mechanismus ist abhängig vom Authentifizierungsverfahren. Durch die Kapselung in einen Service kann das Verfahren unabhängig vom Rest der Anwendung geändert werden. Sowohl der Model-Service als auch der Authentication-Service greift auf den Config-Service zu. In diesem werden datenbankspezifische Einstellungen wie zum Beispiel der Port und Host gespeichert.

## 6.3 CouchDB-Konfiguration

CouchDB muss für die Verwendung entsprechend konfiguriert werden. In den Standardeinstellungen ist die Unterstützung für CORS deaktiviert [14]. Mittels folgender Erwei-

[1] Bei einem *Singleton* handelt es sich um ein Entwurfsmuster. Es wird damit sichergestellt, dass nur ein Objekt einer Klasse erstellt wird [73].

terung der Konfigurationsdatei, wird es aktiviert und festgelegt, welche Origin-Server unterstützt werden. Des Weiteren wird aktiviert, dass Authentifizierungsdaten übertragen werden.

```
[httpd]
enable_cors = true
[cors]
credentials = true
origins = http://localhost
```

Listing 6.2: Aktivierung von CORS

Für die Speicherung der Kurznachrichten wird innerhalb von CouchDB eine Datenbank mit dem Namen *tweets* verwendet. Dieser werden keine Benutzer oder Rollen hinzugefügt. Sie ist somit öffentlich verfügbar. Mittels *Design*-Dokumenten können Datenbanken in CouchDB weiter konfiguriert werden. Somit lässt sich eine Funktion in JavaScript erstellen, die für POST-, PUT- und DELETE-Request ausgeführt wird. Da es ansonsten möglich ist, beliebige Daten zu speichern. Für die Definition dieser wird in dem *Design*-Dokument ein Attribut mit dem Namen *validate_doc_update* erstellt. Der Wert des Attributes beinhaltet das auszuführende Script. Diesem werden das neue Dokument (POST- und PUT-Request), das vorherige Dokument (PUT- und DELETE-Request) und der authentifizierte Benutzer übergeben.

```
function (newDoc, oldDoc, userCtx) {
  if (oldDoc || !userCtx.name) {
    throw ({
      forbidden: 'not allowed'
    });
  }
  require('tweet');
  require('user');
  for (var key in newDoc) {
    validate(key, newDoc[key]);
  }
}
```

Listing 6.3: Definition einer Validierungsfunktion in CouchDB

Listing 6.3 zeigt einen Ausschnitt des erstellten und verwendeten Scripts. Die Funktionen *require*, welche prüft, ob das Dokument das angegebene Attribut enthält, und *validate*, welche die übermittelten Attribute auf bestimmte Eigenschaften, wie zum Beispiel die Länge, prüft, sind nicht mit angegeben. Das Bearbeiten und Löschen von Dokumenten ist nicht gestattet. Daher wird geprüft, ob die Variable *oldDoc* gesetzt ist. Ist dies der Fall, wurde eins gelöscht bzw. geändert. Da dies nicht gestattet ist, wird mittels *throw* die Anfrage zurückgewiesen. Des Weiteren wird geprüft, ob der Benutzer authentifiziert ist. Auf die Benutzerdaten kann mittels Parameter *userCtx* zugegriffen werden. Ist kein

Name gesetzt, ist der Benutzer nicht angemeldet und ihm ist es nicht erlaubt, Nachrichten zu erstellen. Die Attribute *tweet* und *user* werden vorausgesetzt. Weitere sind nicht erlaubt, dies sichert die Funktion *validate* zu.

## 6.4 Prototyp aufbauend auf Orestes

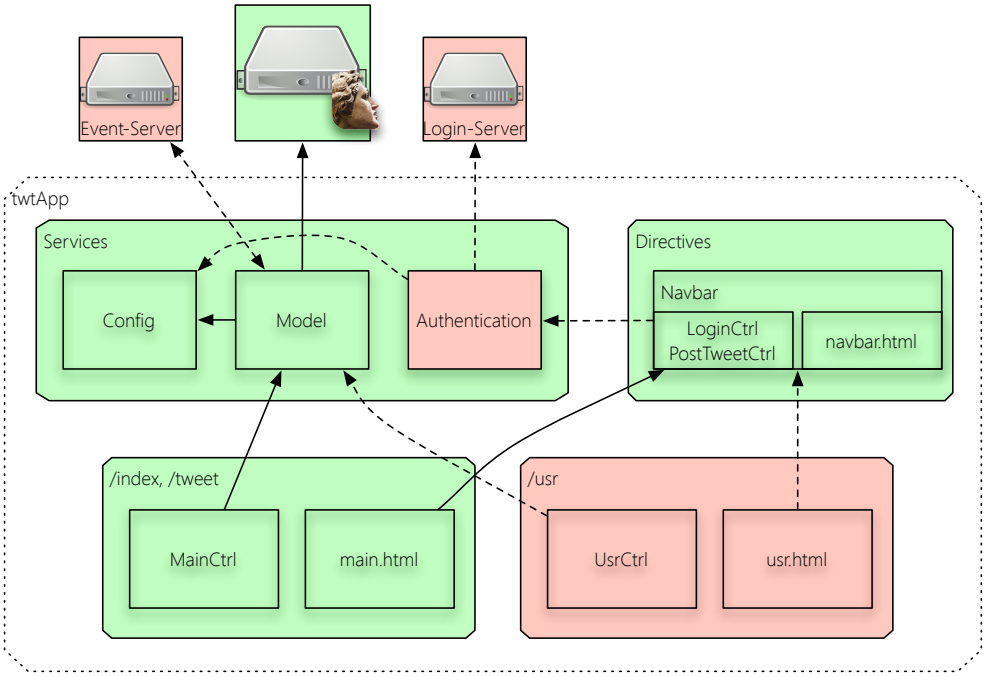


Abbildung 6.5: Aufbau des Smart Clients mit Orestes als Backend

Im Rahmen dieser Arbeit wird der vorgestellte Prototyp in einer featurereduzierten Version auf Basis von Orestes mit der objektorientierten Datenbank *db4o* implementiert. Abbildung 6.5 zeigt den Aufbau. In diesem markieren die grünen Elemente die Programmteile, die implementiert werden. Die roten stehen hingegen nicht zur Verfügung.

Eine Benachrichtigung kann aufgrund des fehlenden Event-Servers nicht implementiert werden. Diese Funktion fehlt somit. Des Weiteren ist es nicht möglich, sich am Orestes-Server zu authentifizieren. Diese Funktion bleibt in der Oberfläche des Smart Clients bestehen, da ein Benutzername für das Speichern einer Nachricht benötigt wird. Ein Benutzer kann sich daher mit einer beliebigen Kombination aus Name und Passwort anmelden. Ebenso ist es nicht möglich, eine Benutzerseite anzuzeigen.

```
1 {
2   "class":"/db/test.persistent.TwtClass",
3     "fields":{
4       "user":"/db/_native.String",
5       "tweet":"/db/_native.String"
6     }
7 }
```

Listing 6.4: Verwendetes Datenbankschema in Orestes

Abbildung 6.4 zeigt das verwendete Datenbankschema. Es wird eine Klasse mit den Attributen *user* und *tweet* erstellt. Beide sind vom Typ *String*.

Im nachfolgenden Kapitel wird ein kurzes Fazit sowie einen Ausblick zu dem Thema der Arbeit gegeben.

# 7 Fazit und Ausblick

In dieser Arbeit wird gezeigt, dass die Entwicklung von Smart Clients in einem Browser aufbauend auf einer 2-Tier-Architektur möglich und zu empfehlen ist, da sie gegenüber klassischen Webseiten erhebliche Vorteile, wie ein gesteigertes Benutzererlebnis, vereinfachte Entwicklung durch Frameworks wie AngularJS und eine günstigeren Betrieb der Applikation, mit sich bringt.

Die Inkompatibilität Suchmaschinencrawler gegenüber, welche einen professionellen Einsatz von Smart Clients ausschließt, wird durch den in dieser Arbeit entwickelten Seoxy behoben. Durch den verwendeten Cache kann die Bewertung der Suchmaschinen gegebenenfalls sogar verbessert werden, da dieser eine beschleunigte Auslieferungszeit ermöglicht. Für einen produktiven Einsatz muss dieser noch um Funktionen erweitert werden. Dazu gehört eine Konfigurationsmöglichkeit durch die Sitemap, die Unterstützung von Datenbanken zur Haltung des Caches und den Support für weitere Suchmaschinen, wie zum Beispiel Bing.

Durch die Wahl eines kompatiblen Authentifizierungsverfahren und einer Autorisierungsarchitektur, die einen möglichst flexiblen Einsatz erlaubt, können Smart Clients in einer 2-Tier-Architektur auch für Anwendungen verwendet werden, welche geschützte Inhalte ausliefern. Die Verwendung eines zustandslosen Authentifizierungsverfahrens ermöglicht zudem die Skalierung durch Reverse-Proxy-Caches für die zu sichernden Daten. Dabei ist zu beachten, dass stets geprüft wird, ob die verwendeten kryptographischen Verfahren noch als sicher gelten.

Gezeigt wird ebenso, dass die komplette Verlagerung der Logik in den Browser nicht möglich und sinnvoll ist, da nicht alle Daten öffentlich zugänglich sein dürfen. Eine Datenbank, die sich für den Betrieb eignet, muss daher die Ausführung von Stored Procedures unterstützen.

Orestes bietet in seiner aktuellen Version noch nicht alle Features, die für einen Smart Client benötigt werden. Es stellt aber durch die Verwendung einer REST-API und der gezeigten Cachehierarchie bereits Lösungen zur Verfügung, die für einen optimalen Betrieb nötig sind. Da durch die Bloomfilter Konsistenz von Objekten sichergestellt wird, kann Orestes auch für Anwendungen, die diese benötigen, wie zum Beispiel Onlineshops, verwendet werden.

Abbildung 7.1 zeigt die Orestesinfrastruktur inklusive den benötigten Erweiterungen, die in dieser Arbeit erörtert werden. Dazu gehören Login-Server, Event-Server und Seoxy.

Abschließend bleibt festzuhalten, dass Orestes bei einer kontinuierlichen Weiterentwicklung zu der optimalen Infrastruktur für die Verwendung von Smart Clients werden

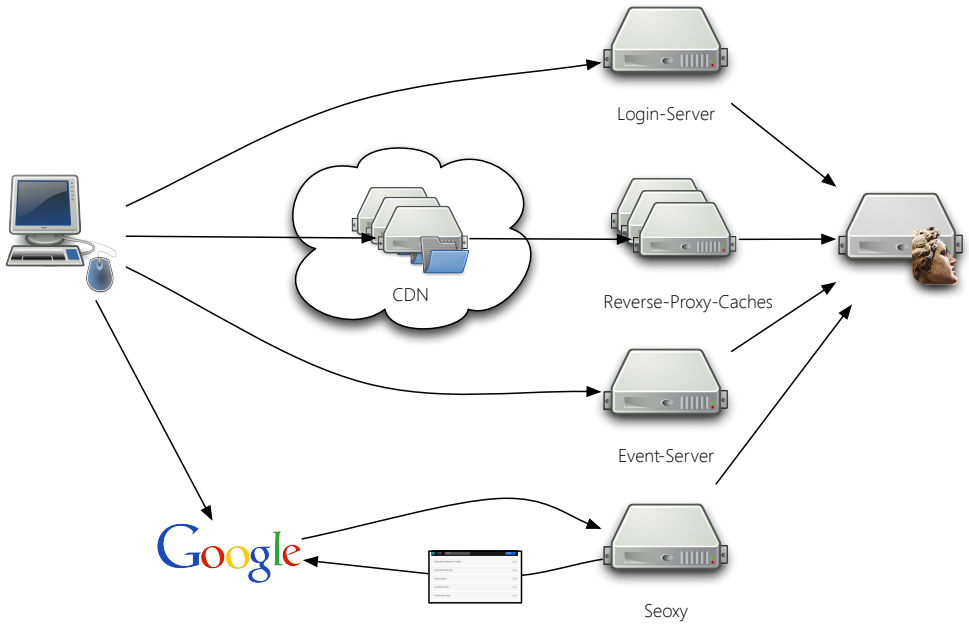


Abbildung 7.1: Orestes inklusive benötigte Erweiterungen für den Betrieb eines Smart Clients

kann.

# Abbildungsverzeichnis

1.1 Aufbau einer 3-Tier-Architektur . . . . . 6
1.2 Aufbau einer 2-Tier-Architektur . . . . . 6
1.3 Aufbau einer 2-Tier-Architektur auf Basis der Amazon-Cloud . . . . . 7

2.1 Architektur einer klassischen Anwendung . . . . . 11
2.2 Verwendung von mehreren Datenbanken mittels Orestes-Protokoll . . . . 12
2.3 Auslagerung der Datenbank mittels das Orestes-Protokoll in die Cloud . . 13
2.4 Konsistenzverletzung durch Web-Caches . . . . . 13
2.5 Verteilung der unterschiedlichen Caches . . . . . 15
2.6 Auslieferung einer Ressource durch den Reverse Proxy Cache . . . . . 15
2.7 Auslieferung einer *stale* Ressource durch den Reverse Proxy Cache . . . . 16
2.8 Beispiel eines Bloomfilters für $m = 8$ und $k = 2$ . . . . . 17
2.9 Der *Counting Bloomfilter* enthält eine Löschfunktion . . . . . 19
2.10 Ablauf einer Transaktion . . . . . 19

3.1 Model-View-Controller-Pattern in einer 3-Tier-Architektur . . . . . 21
3.2 Model-View-Controller-Pattern auf einer 2-Tier-Architektur aufbauend . . 22
3.3 Erweiterung der Orestes-Infrastruktur um einen Event-Server . . . . . 28
3.4 Ablauf des Starts von AngularJS [26] . . . . . 32
3.5 *Hello World*-Beispiel in AngularJS . . . . . 35

4.1 Ablauf des *Cross-Origin-Resource-Sharings* . . . . . 47
4.2 Modaler Dialog zur Eingabe der Benutzerdaten unter Verwendung der HTTP-Basic-Access-Authentication . . . . . 48
4.3 Authentifizierungstoken nach Fu et al [22] . . . . . 49
4.4 Cookie-Authenfizierung nach Fu et al [22] . . . . . 50
4.5 Rekursive Erstellung des Hashes [52] . . . . . 51
4.6 Erweiterter Authentifizierungstoken [52] . . . . . 51
4.7 Erstellung des Schlüssels [45] . . . . . 52
4.8 Authentifizierungstoken [45] . . . . . 52
4.9 Ablauf der Authentifizierung des OpenID-Protokolls [32] . . . . . 54
4.10 Asymmetrische Verschlüsselung . . . . . 56
4.11 Asymmetrische Signierung . . . . . 57
4.12 Authentifizierungsarchitektur aufbauend auf Orestes . . . . . 60

4.13 Aufbau eines Datenbank-Schemas in Orestes . . . . . 62

5.1 Aufgabenverteilung unter Verwendung von Seoxy . . . . . 72
5.2 Ablauf der Anfragen des Google Crawlers . . . . . 74
5.3 Interner Ablauf des Seoxys . . . . . 81
5.4 Interner Ablauf des Seoxys unter Einbezug der Sitemap . . . . . 84
5.5 Aufbau von Rendr (Quelle rendr blog post) . . . . . 85
5.6 Aufbau von Rendr (Quelle rendr blog post) . . . . . 86
5.7 Aufbau von Rendr [12] . . . . . 88

6.1 Startseite des Prototypens . . . . . 91
6.2 Benutzerseite . . . . . 92
6.3 Link zu einem Tweet . . . . . 92
6.4 Aufbau des Smart Clients . . . . . 92
6.5 Aufbau des Smart Clients mit Orestes als Backend . . . . . 95

7.1 Orestes inklusive benötigte Erweiterungen für den Betrieb eines Smart Clients . . . . . 98

# Listings

3.1 Polling . . . 22
3.2 Long-Polling . . . 23
3.3 Server-Sent Events . . . 24
3.4 Datenformat . . . 24
3.5 Selbstersteller Eventtyp . . . 25
3.6 Erstellung einer WebSockets-Verbindung . . . 25
3.7 socket.io in Node.js . . . 26
3.8 socket.io im Smart Client . . . 27
3.9 Beispiel einer *package.json* . . . 29
3.10 Konfiguration des *concat*-Plugins . . . 30
3.11 JavaScript vor der Minifizierung . . . 30
3.12 JavaScript nach der Minifizierung . . . 31
3.13 ng-app Directive . . . 33
3.14 Konfiguration der Applikation . . . 33
3.15 ng-app Directive . . . 34
3.16 Beispiel des Controllers *MainCtrl* . . . 35
3.17 Beispiel der View *main.html* . . . 35
3.18 Route zum Aufruf der *main.html* mittels *MainCtrl* . . . 35
3.19 Die *index.html* der Hello-World-Anwendung in Backbone.js . . . 37
3.20 Definition einer View in Backbone.js . . . 37
3.21 Definition eines Models in Backbone.js . . . 37
3.22 Definition und Verwendung einer Collection in Backbone.js . . . 38
3.23 Definition einer Route in Backbone.js . . . 38
3.24 Die *index.html* der Hello-World-Anwedung in Ember.js . . . 40
3.25 Hello-World-Application in Ember.js . . . 40

4.1 Origin-Feld im HTTP-Header . . . 45
4.2 Access-Control-Allow-Origi-Feld im HTTP-Header . . . 45
4.3 Fehlermeldung bei Verletzung der *Same-Origin-Policy (SOP)* . . . 45
4.4 Erlaubt jeden Origin-Server . . . 45
4.5 Access-Control-Allow-Credentials-Feld im HTTP-Header . . . 46
4.6 Verwendung des XMLHttpRequest-Objekts . . . 46
4.7 Verwendung des *WWW-Authenticate*-Feld im HTTP-Header . . . 47

4.8 Verwendung des *WWW-Authenticate*-Feld im HTTP-Header . . . . . 48

5.1 Hash-Bang in einer URL . . . . . 66
5.2 Hash-Bang per JavaScript auslesen . . . . . 66
5.3 Googles Umwandlung des Hash-Bangs in einen GET-Parameter [29] . . . 66
5.4 Verwendung der pushState-Funktion [49] . . . . . 67
5.5 HTTP-Server in Node.js . . . . . 69
5.6 Erstellung einer Seite in PhantomJS . . . . . 70
5.7 Googles Umwandlung des Hash-Bangs in einen GET-Parameter . . . . . 72
5.8 Redirect für Browser mit deaktiviertem JavaScript . . . . . 74
5.9 Reverse-Proxy in Node.js . . . . . 75
5.10 Seoxys Einstiegspunkt . . . . . 76
5.11 Methode zur Generierung der HTML-Datei . . . . . 77
5.12 Kapselung der Erstellung des PhantomJS-Prozesses . . . . . 78
5.13 Kapselung der Erstellung des PhantomJS-Prozesses . . . . . 79
5.14 Definition eines Canonical Links . . . . . 80
5.15 Aufbau einer *sitemap.xml* [67] . . . . . 82
5.16 Erzeugung von HTML anhand der *Sitemap* . . . . . 84
5.17 Übertragener HTML-Sourcecode bei Aufruf des Smart Clients . . . . . 85

6.1 Zuweisung eines Controllers zu einem HTML-Element . . . . . 93
6.2 Aktivierung von CORS . . . . . 94
6.3 Definition einer Validierungsfunktion in CouchDB . . . . . 94
6.4 Verwendetes Datenbankschema in Orestes . . . . . 95

# Literaturverzeichnis


[1] Ecmascript 6 compatibility table: http://kangax.github.io/es5-compat-table/es6/.

[2] Phantomjs wrapper for node.

[3] Grunt - the javascript task runner, 2013.

[4] airbnb. About: https://www.airbnb.de/about.

[5] ARD.de. Online-nutzungszahlen: http://www.ard.de/intern/medienbasisdaten/onlinenutzung/re
/id=1198700/mi1adx/index.html.

[6] Backbone.js. http://backbonejs.org/.

[7] Mihai Bazon. Uglifyjs2: https://github.com/mishoo/uglifyjs2.

[8] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. RFC 3986 uniform resource identifier (URI): Generic syntax. Technical report, January 2005.

[9] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[10] Eric Bidelman. Stream updates with server-sent events. *HTML5ROCKS by Google*, 2010.

[11] Bitcom. Breitband: Starkes wachstum in deutschland. 2012.

[12] Spike Brehm. Building single-page apps: http://nerds.airbnb.com/slides-and-video-from-spike-brehms-tech-talk/.

[13] Spike Brehm. Our first node.js app: Backbone on the client and server: http://nerds.airbnb.com/weve-launched-our-first-nodejs-app-to-product.

[14] CouchDB. Couchdb wiki - cors: http://wiki.apache.org/couchdb/cors.

[15] Apache CouchDB. https://couchdb.apache.org/.

[16] Ember.js. Ember.js guides - http://emberjs.com/guides/.

[17] Sebastian Erlhofer. *Suchmaschinen-Optimierung: das umfassende Handbuch*. Galileo Press, 2011.

[18] Facebook. Facebook login: https://developers.facebook.com/docs/facebook-login/.

[19] OpenID Foundation. http://www.openid.net/get-an-openid/.

[20] Martin Fowler. Polyglotpersistence.

[21] Armando Fox, Rean Griffith, A Joseph, R Katz, A Konwinski, G Lee, D Patterson, A Rabkin, and I Stoica. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28, 2009.

[22] Kevin Fu, Emil Sit, Kendra Smith, and Nick Feamster. The dos and don'ts of client authentication on the web. In *USENIX Security Symposium*, pages 251–268, 2001.

[23] Felix Gessert and Florian Bücklers. Performanz- und reaktivitätssteigerung von oodbms vermittels der web-caching- hierarchie unter einsatz und adaption offener standards. Master's thesis, Universität Hamburg, 2010.

[24] Felix Gessert and Florian Bücklers. Kohärentes web-caching von datenbankobjekten im cloud computing. Master's thesis, Universität Hamburg, 2013.

[25] Felix Gessert, Florian Bücklers, and Norbert Ritter. Orestes: a rest protocol for horizontally scalable cloud database access. 2012.

[26] Google. Angularjs - developer guide: http://docs.angularjs.org/guide.

[27] Google. Closure compiler: https://developers.google.com/closure/compiler/.

[28] Google. Major search engines unite to support a common mechanism for website submission.

[29] Google. Making ajax applications crawlable: https://developers.google.com/webmasters/ajax-crawling/.

[30] David Gourley, Brian Totty, Marjorie Sayer, Anshu Aggarwal, and Sailu Reddy. *HTTP: the definitive guide*. O'Reilly Media, Inc., 2009.

[31] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly, 2013.

[32] Heine. The openid 2.0 compliance crusade. 2010.

[33] Ian Hickson. *Server-Sent Events*, Juli 2013.

[34] Ian Hickson. *The WebSocket API*, Juli 2013.

[35] Ariya Hidayat. Phantomjs.

[36] Internet Engineering Task Force (IETF). Deprecating the "x-prefix and similar constructs in application protocols. 2012.

[37] Noah Slater J. Chris Anderson, Jan Lehnardt. *CouchDB: The Definitive Guide*. O'Reilly, 2010.

[38] jQuery. jquery: http://jquery.com/.

[39] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC Press, 2008.

[40] Stefan Koch. *JavaScript: Einführung, Programmierung und Referenz-inklusive Ajax*. Dpunkt-Verlag, 2007.

[41] Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, 2007.

[42] Jerri L Ledford. *SEO Search Engine Optimization Bible*, volume 584. John Wiley & Sons, 2009.

[43] Avraham Leff and James T Rayfield. Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International*, pages 118–127. IEEE, 2001.

[44] Arjen K Lenstra. Key lengths. *Handbook of Information Security*, 2:617–635, 2004.

[45] Alex X Liu, Jason M Kovacs, C-T Huang, and Mohamed G Gouda. A secure cookie protocol. In *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*, pages 333–338. IEEE, 2005.

[46] Adrian Mejia. Backbone.js for absolute beginners. 2012.

[47] Mozilla. arewefastyet: http://arewefastyet.com.

[48] Mozilla. Http access control (cors): https://developer.mozilla.org/en-us/docs/http/access_control_cors.

[49] Mozilla. Manipulating the browser history: https://developer.mozilla.org/en-us/docs/web/guide/dom/manipulating_the_browser_history.

[50] Mozilla. Same-origin policy: https://developer.mozilla.org/en-us/docs/web/javascript/same_origin_policy_for_javascript.

[51] Mozilla. Xmlhttprequest: https://developer.mozilla.org/en-us/docs/web/api/xmlhttprequest.

[52] Steven J Murdoch. Hardened stateless session cookies. In *Security Protocols XVI*, pages 93–101. Springer, 2011.

[53] San Murugesan and Yogesh Deshpande. *Web engineering: Managing diversity and complexity of Web application development*, volume 2016. Springer, 2001.

[54] SV Nagaraj. *Web caching and its applications*. Springer, 2004.

[55] OAuth. http://oauth.net/.

[56] OpenID. Openid authentication 2.0 - final specification.

[57] OpenSSL. Openssl: The open source toolkit for ssl/tls: http://ww.openssl.org.

[58] Oracle. Java scripting programmer's guide.

[59] Masataka Yakura Paul Irish, Divya Manian, 2013.

[60] Larry L Peterson and Bruce S Davie. *Computer networks: a systems approach*. Elsevier, 2007.

[61] Guillermo Rauch. socket.io: http://www.socket.io.

[62] RequireJS. http://requirejs.org/.

[63] Golo Roden. *Node.js & Co.* Dpunkt-Verlag, 2012.

[64] seo united.de. Suchmaschinen im august 2013 http://www.seo-united.de/suchmaschinen.html.

[65] Amazon Web Services. Amazon dynamodb: https://aws.amazon.com/de/dynamodb/.

[66] Amazon Web Services. Amazon s3: https://aws.amazon.com/de/s3/.

[67] sitemaps.org. Sitemaps xml format: http://www.sitemaps.org/protocol.html.

[68] Nigel Smart et al. Ecrypt ii yearly report on algorithms and keysizes (2011-2012). *Framework*, 2011.

[69] Peter Smith. *Professional Website Performance: Optimizing the Front-End and Back-End*. Wrox, 2012.

[70] Stephen Thomas. *Http Essentials with Cdrom.* John Wiley & Sons, Inc., 2001.

[71] Malte Ubl and Eiji Kitamura. Introducing websockets: Bringing sockets to the web. *HTML5ROCKS by Google*, 2010.

[72] Varnish. Varnish: https://www.varnish-cache.org.

[73] John Vlissides, R Helm, R Johnson, and E Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49:120, 1995.

[74] W3C. Html workingdraft: http://www.w3.org/html/wg/drafts/html/master/dom.html.

[75] W3C. Html5 - a vocabulary and associated apis for html and xhtml.

[76] W3C. Same origin policy: http://www.w3.org/security/wiki/same_origin_policy.

[77] w3schools.com. Browser statistics.

[78] whyeye.org. History of javascript performance: Chrome: http://whyeye.org/blog/browsers/history-of-javascript-performance-chrome/.

[79] whyeye.org. History of javascript performance: Firefox: http://whyeye.org/blog/browsers/history-of-javascript-performance-chrome/.

# Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____