
FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

## Seminararbeit

# Architekturen und Verfahren zur Absicherung des Informationsaustauschs im Client-Server Modell

Vorgelegt an:
**Fachhochschule Aachen, Campus Jülich**
Fachbereich Medizintechnik und Technomathematik
Studiengang Scientific Programming

Vorgelegt von: Sven Bergmann
Matrikelnummer: 3231105
Ausbildungsbetrieb: CAE GmbH
1. Betreuer: Prof. Dr. rer. nat. Hans-Joachim Pflug
2. Betreuer: Dipl. Ing. Christoph Hennecke

Stolberg, den 23. November 2021

## Abstract

Diese Seminararbeit soll verschiedene Verfahren zur Absicherung des Informationsaustauschs im Client-Server Model vorstellen und vergleichen. Dafür werden zuerst alle relevanten Grundlagen geschaffen, indem auf die Funktionsweise des Informationsaustauschs über das Internet kurz, jedoch hinreichend, eingegangen wird. Auf Basis dessen werden Web Services definiert und bekannte Architekturen, wie SOAP und REST, vorgestellt, näher erläutert und verglichen. Danach wird die Verschlüsselung im Allgemeinen behandelt und die Brücke zum TLS-Protokoll geschlagen, was wiederum zur Definition des Begriffes Web-Security führt.
Der darauf folgende Abschnitt wendet sich den eigentlichen Verfahren zu, wie der Zugriff über Authentifizierung und Autorisierung tatsächlich eingeschränkt oder gewährt werden kann. Das letzte Kapitel fügt alles zusammen und gibt einen Ausblick darüber, welches Verfahren für die Implementierung eines Web-Services tatsächlich genutzt werden könnte. Dies wird in Hinblick auf die Architektur, die persönliche Meinung, jedoch vor allem auch auf die gestellten Anforderungen, betrachtet.

# Inhaltsverzeichnis

## Akronyme

| | |
|---|---|
| 7z | 7-Zip *Glossar:* 7z, 10 |
| AES | Advanced Encryption Standard 10 |
| API | Application Programming Interface *Glossar:* API, 4, 6, 7, 8, 9, 14, 19 |
| CORBA | Common Object Request Broker Architecture *Glossar:* CORBA, 4 |
| DCOM | Distributed Component Object Model *Glossar:* DCOM, 4 |
| DES | Data Encryption Standard 10 |
| FLIP | Flight Information Publication |
| FTP | File Transfer Protocol 3 |
| HATEOAS | Hypermedia as the Engine of Application State 8 |
| HTML | Hypertext Markup Language *Glossar:* HTML, 7 |
| HTTP | Hypertext Transfer Protocol 1, 2, 3, 5, 8, 9, 13, 19 |
| HTTPS | Hypertext Transfer Protocol Secure 12, 13, 14, 15, 19, X |
| IETF | Internet Engineering Task Force *Glossar:* IETF, 11, 14 |
| JSON | JavaScript Object Notation *Glossar:* JSON, 4, 7 |
| LFID | Luftfahrt Informations Datenbank |
| REST | Representational State Transfer 6, 7, 8 |
| SMTP | Simple Mail Transfer Protocol X, XII, |
| SOAP | Simple Object Access Protocol 4, 5, 6 |
| SSH | Secure Shell *Glossar:* SSH, 10 |
| SSL | Secure Sockets Layer 11 |
| TCP | Transmission Control Protocol 2, 3, 11, X |
| TLS | Transport Layer Security 11, 12 |

| | |
|---|---|
| URI | Uniform Resource Identifier *Glossar:* URI, V, 3, 4, 5, 6, 8, 9, 15, 16, XII |
| URL | Uniform Resource Locator *Glossar:* URL, V, 7, 8, 16 |
| URN | Uniform Resource Name *Glossar:* URN, V, |
| W3C | World Wide Web Consortium *Glossar:* W3C, 1, 3, 4, XII |
| WPA2 | Wi-Fi Protected Access 2 *Glossar:* WPA2, 10 |
| WS-Security | Web Services Security *Glossar:* WS-Security, 4 |
| WS-ReliableMessaging | Web Services Reliable Messaging *Glossar:* WS-ReliableMessaging, 4 |
| WS-Addressing | Web Services Addressing *Glossar:* WS-Addressing, 4 |
| WSDL | Web Services Description Language *Glossar:* WSDL, 4, 5, XII |
| WWW | World Wide Web *Glossar:* WWW, V, 1, 6, 12 |
| XML | Extensible Markup Language *Glossar:* XML, 4, 5, 7 |

## Glossar

**7-Zip (7z)** Freies Dateiformat mit der Endung .7z, was zur Komprimierung und Archivierung von Daten verwendet wird

**Application Programming Interface (API)** Satz von Befehlen, Protokollen und Funktionen, welche Entwickler für die Erstellung von Software zur Ausführung allgemeiner Operationen verwenden können. Diese Programmierschnittstelle stellt somit die Grundlage für die Kommunikation zwischen Anwendungen bereit.

**Authentifizierung** Stellt die Prüfung der behaupteten Authentisierung dar (Verifizierung der Identität)

**Authentisierung** Ein Nutzer legt Nachweise vor, welche dessen Identität bestätigen sollen (Behauptung einer Identität)

**Autorisierung** Nach erfolgreicher Authentifizierung werden spezielle Rechte an den Nutzer vergeben (Vergabe oder Verweigerung von Rechten)

**Base64** Kodierungsverfahren, welches den Text zuerst in einen Bytestrom umwandelt, aus diesem jeweils immer drei Bytes hernimmt, die 24 Bits wiederum in 4 Blöcke à 6 Bit aufteilt und letztendlich die Binärwerte wieder in ein Zeichen umwandelt. So entsteht bei $n$ zu kodierenden Zeichen ein Bedarf von $z = 4 \cdot \lceil \frac{n}{3} \rceil$ Zeichen.

**Common Object Request Broker Architecture (CORBA)** Objektorientiertes Kommunikationsprotokoll, welches plattformübergreifend und unabhängig von Programmiersprachen ist

**Distributed Component Object Model (DCOM)** Von Microsoft entwickeltes Protokoll um auf Basis des Component Object Models (COM) über das Netzwerk kommunizieren zu können

**Eulersche $\phi$-Funktion** Gibt für jedes $n \in \mathbb{N}$ an, wie viele zu $n$ teilerfremde $m \in \mathbb{N}$ mit $m < n$ existieren

**Extensible Markup Language (XML)** Auszeichnungssprache zur Darstellung von Objekten oder Dateien in einem menschen- und maschinenlesbaren Format

**Follow-up-Link** Link, der nach einer Aktion auf dem Server an den Clienten zurückgeschickt wird, um verfügbare Aktionen/Ressourcen zu signalisieren

**Framework** Zu deutsch „Rahmenwerk“, was eine wiederverwendbare Struktur bereitstellt, welche für die Entwicklung verschiedener Programme genutzt werden kann

**Hashfunktion** Surjektive Abbildung $h : K \to S$ mit $|K| \geq |S|$, welche eine Basismenge auf eine kleinere Zielmenge abbildet

**Head-of-Line-Blocking** Problem, das entsteht, wenn mehrere zu versendende Pakete durch das erste Paket aufgehalten werden

**Host** Computer, welcher eine eindeutige IP-Adresse besitzt und darüber erreicht werden kann

**Hypertext Markup Language (HTML)** Sprache zur Darstellung von Inhalten über einen Web Browser

**Internet Engineering Task Force (IETF)** Organisation zur Verbesserung und Weiterentwicklung der Funktionsweise des Internets

**ISO/OSI-Referenzmodell** Modell für die Netzwerkprotokolle als Darstellung über eine Schichtenarchitektur, siehe auch Schichtenmodell

**JavaScript** Skriptsprache, welche ursprünglich für das dynamische Erstellen von Internetseiten entwickelt wurde

**JavaScript Object Notation (JSON)** Maschinenlesbare Sprache zur Darstellung von Objekten. Bei JavaScript kann dies unter anderem für eine Instanziierung neuer Objekte genutzt werden.

**Overhead** Daten, welche nicht primär zu den Nutzdaten zählen, aber trotzdem zusätzlich als Information beispielsweise zur Weiterverarbeitung oder Speicherung benötigt werden

**Pipelining** Falls ein Client persistente Verbindungen unterstützt, können die Anfragen hintereinander geschickt werden, ohne auf die jeweilige Antwort zu warten. Der Server muss dann in genau dieser Reihenfolge auf die Anfragen antworten.

**Secure Shell (SSH)** Netzwerkprotokoll, welches dazu befähigt, über ein ungesichertes Netzwerk, sicher auf einen Computer zuzugreifen

**Single Sign-On** Auch als „Einmalanmeldung“ bekannt, berechtigt den Clienten sich nur einmal zu authentifizieren und danach alle bereitgestellten Dienste ohne erneute Anmeldung zu nutzen.

**String** Zeichenkette, welche in manchen Programmiersprachen einen eigenen Datentypen darstellt

**Uniform Resource Name (URN)** Adressierung von Objekten ohne ein Protokoll festzulegen (Eindeutige und gleichbleibende Referenz - Name der Ressource)

**Uniform Resource Locator (URL)** Adressierung von Informationsobjekten mit Festlegung des Zugangs-Protokolls (Ort der Ressource)

**Uniform Resource Identifier (URI)** Eindeutige Adressierung von abstrakten und physikalischen Ressourcen im Internet
URI: URLs ∪ URNs

**Web Services Addressing (WS-Addressing)** Mechanismus für Web Services um Adressinformationen austauschen zu können

**Web Services Reliable Messaging (WS-ReliableMessaging)** Standard, welcher garantiert, dass Nachrichten auf jeden Fall beim Empfänger ankommen

**Web Services Security (WS-Security)** Kommunikationsprotokoll, welches ermöglicht, Sicherheitsaspekte bei Web Services einzubeziehen

**Web Services Description Language (WSDL)** Auszeichnungssprache um Web Services zu charakterisieren

**Wi-Fi Protected Access 2 (WPA2)** Standard für die Authentifizierung und Verschlüsselung von WLANs, basierend auf dem IEEE 802.11 Standard

**World Wide Web Consortium (W3C)** Konsortium, welches aus verschiedenen Mitgliedsorganisationen besteht, deren Ziel es ist, Protokolle und Richtlinien des WWW zu standardisieren

**World Wide Web (WWW)** Dezentrales Kommunikationsnetz, welches dazu dient, Informationen zu übertragen und nutzbar zu machen

# Abbildungsverzeichnis

# 1 Einleitung

„Einfach einloggen und herunterladen.“ — Ein Satz, der im alltäglichen Leben immer mehr an Bekanntheit gewinnt. Doch wie funktioniert dieses „Einloggen und Herunterladen“ eigentlich? Wie kann gewährleistet werden, dass die zu übermittelnden Daten sicher übertragen werden, oder die Identität des Benutzers nicht gestohlen wird? Wie werden überhaupt Daten übertragen? Was sind verteilte Systeme? All diese Fragen sollen im Folgenden zumindest ansatzweise beantwortet werden. Die Ziele dieser Arbeit werden demnach sein, die Datenübertragung über das Internet im Ansatz zu verstehen, verschiedene Architekturen zur Realisierung dieser kennenzulernen und darüber hinaus die Wichtigkeit und Notwendigkeit der Absicherung dieser aufzufassen.
Da das Netzwerkprotokoll World Wide Web (WWW) und damit auch das Hypertext Transfer Protocol (HTTP) schon seit 1989 existieren, eignet sich eine Literaturarbeit zur Darlegung des Themas am besten. Im ersten Kapitel werden anfangs die grundlegenden Funktionsweisen und programmiertechnischen Hintergründe definiert und erläutert, wie ein Datentransfer tatsächlich funktionieren kann. Danach werden auf Basis dessen ausgewählte Architekturstile aufgezeigt und verglichen. Da für eine sichere Kommunikation eine Verschlüsselung unerlässlich ist, folgen Erklärungen und Definitionen über Kryptographie. Dieser eher theoretische Einschub führt einerseits zu einem Verfahren, welches einen sicheren Informationsaustausch gewährleisten kann, und andererseits zur Definition des Begriffes „Web-Security“.
Das Kapitel Absicherung des Informationsaustauschs behandelt verschiedene Frameworks, die für die Authentifizierung, unter Benutzung der vorhergehenden Protokolle und Grundlagen, genutzt werden können. Hierbei werden auch die Einsatzgebiete dieser erläutert. Die Arbeit wird mit einer Zusammenfassung abgerundet und es wird ein Ausblick darauf gegeben, wie diese zu der Entscheidung beigetragen hat, welche Verfahren und welche Architekturen in die engere Auswahl für die Implementierung eines Web-Services kommen.
Die Literatur dieser Arbeit besteht ausnahmslos aus Online-Quellen, da das Thema mit der Entstehung des Internets verwurzelt ist und daher fast ausschließlich alles online zu finden ist. Hierbei spielt die Reihe „Request for Comments“ eine große Rolle, in welcher viele Protokolle definiert sind. Ebenso stellt das World Wide Web Consortium (W3C) auch einige Definitionen bereit. Nahezu alle Quellen haben gemeinsam, dass bei der Erklärung die Sprachkonventionen laut RFC 2119 [Bra97] genutzt werden, was im Folgenden durch eine *kursive* Schrift angezeigt wird.

# 2 Begriffsdefinitionen und Abgrenzungen

## 2.1 Netzwerkprotokolle

Im Folgenden wird kurz auf relevante Netzwerkprotokolle eingegangen, welche sich in der Anwendungsschicht (engl. Application Layer) des Transmission Control Protocol (TCP)/IP-Modells[1] befinden.

### 2.1.1 HTTP

Das HTTP existiert bereits seit 1996 in der Version „HTTP/1.0“ [Vgl. NFB96] und definiert ein leichtgewichtiges, schnelles und einfach zu implementierendes Protokoll, welches für viele Aufgaben eingesetzt werden kann. Es ist zudem zustandslos und objektorientiert, woraus folgt, dass jede Operation über eine separate TCP-Verbindung realisiert wird.
1999 wurde „HTTP/1.1“ [Vgl. FR14b] standardisiert. Diese Version stellte eine Verbesserung bezüglich der Ladezeiten dar, indem ein Header Feld namens „Connection: keep-alive“ hinzugefügt wurde. Dieses Feld kann vom Clienten optional gesetzt werden, um zu signalisieren, dass dieser die TCP-Verbindung offen halten möchte. Des Weiteren wurden „Non-IP-based Virtual Hosts“ durch das Header Feld „Host“ als Lösung für virtuelle Hosts hinzugefügt, wodurch sich nun beliebig viele Domänen und Dienste die gleiche IP-Adresse teilen, aber trotzdem auf einem Rechner liegen können.
Eine weitere Beschleunigung der Datenübertragung stellte „HTTP/2.0“ [Vgl. BPT15] dar. Das 2015 standardisierte Protokoll ist abwärtskompatibel zur Version 1.1. Es kann aber eine Anfrage vom Clienten über das Header Feld „Connection: Upgrade“ gestartet werden, ob der Server die neuere Version unterstützt. Falls diese Konstellation gegeben ist, wird nach der Antwort des Servers mit der Version 2.0 weitergearbeitet. Da das Pipelining in Version 1.1 zwar erlaubt ist, jedoch von Browser zu Browser entweder unterschiedlich oder gar nicht implementiert ist, wurde 2015 das „Multiplexing“ hinzugefügt. Die Verbesserung hierbei stellen die Streams dar, welche mehrere Anfragen oder Antworten gleichzeitig übertragen können. Des Weiteren muss der Server nicht mehr die Reihenfolge der Antworten beachten, sondern weist den Streams eine eindeutige ID zu, was das sogenannte Head-of-Line-Blocking löst.
Der grundsätzliche Ablauf ist wie folgt:

1. Der Client baut eine TCP/IP-Verbindung zum Server auf.
2. Der Client sendet eine HTTP-Anfrage (s. HTTP-Methoden), in der das geforderte Dokument / der geforderte Dienst genau spezifiziert ist.
3. Der Server antwortet mit einem Statuscode und je nach Erfolg der Anfrage auch mit dem angeforderten Dokument.
4. Im Falle von „HTTP/1.0“ wird die TCP/IP-Verbindung geschlossen. Wie oben schon erwähnt, kann durch das Setzen des „Connection: keep-alive“ Feldes die Verbindung aufrechterhalten werden und so entfallen die Schritte 4 und 1 bei weiteren Anfragen.

[1] s. Abbildung 3

#### HTTP-Methoden

Eine HTTP-Anfrage benutzt Methoden zur Spezifikation der gewünschten Aktion. Davon gibt es seit „HTTP/1.1“ folgende acht Methoden, von welchen alle Server die Methoden **GET** und **HEAD** unterstützen *müssen*. Alle anderen Methoden sind optional.

- **GET** fordert eine im Request-URI definierte Resource an.
- **HEAD** ist ähnlich wie GET, übermittelt aber nur den Header ohne den Body.
- **POST** überträgt Daten an den Server.
- **PUT** möchte alle Daten, welche momentan auf dem Server liegen, mit den eigenen Daten überschreiben.
- **DELETE** löscht Dokumente auf dem Server (sofern die Rechte dazu vorhanden sind).
- **CONNECT** kann einen Tunnel zum Server für die bidirektionale Kommunikation öffnen.
- **OPTIONS** fordert den Zugriffspfad einer Resource an und die weitere Kommunikation, welche darüber möglich ist.
- **TRACE** ist zum Testen. Die Anfrage wird vom Server zurückgeschickt.

### 2.1.2 FTP

Im Oktober 1985 wurde das File Transfer Protocol (FTP) spezifiziert [Vgl. PR85]. Seitdem ist dies der Internet-Standard für die Übertragung von Dateien und wird benutzt, um eine komplette Datei von einem auf den anderen Rechner zu kopieren oder interaktiven Zugriff zu ermöglichen. Hierbei werden zwei Ports genutzt, wobei Port 20 und Port 21 der Standard sind. Über den Control Port des Servers (Port 21) wird eine TCP-Verbindung aufgebaut, mit der die Befehle gesendet werden. Falls dies erfolgreich verlaufen ist, wird über einen anderen Port (standardmäßig Port 20) die Dateiübertragung abgewickelt. Normalerweise ist die Vorgehensweise etwas komplizierter, funktioniert je nach aktivem oder passivem Modus etwas anders und besitzt seinerseits auch Methoden. Das ist hier aber nicht von Belang und würde den Rahmen der Arbeit sprengen.

## 2.2 Web Services

Die Definition eines Web Services ist laut dem W3C folgende [Boo+04]:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Unter einem Web Service versteht man also ein Softwaresystem, welches dafür da ist, die Maschine-zu-Maschine Kommunikation über ein Netzwerk zu realisieren. Dieses System

stellt eine Schnittstelle bereit, die in einem maschinenlesbaren Format, genauer Web Services Description Language (WSDL), beschrieben ist. Andere Systeme können dann über sogenannte Simple Object Access Protocol (SOAP) Nachrichten mit dem Service kommunizieren. Typischerweise werden diese Botschaften über HTTP Methoden, unter Benutzung einer Extensible Markup Language (XML) Serialisierung, in Verbindung mit anderen web-bezogenen Standards, übermittelt.
Aus dieser Definition folgt unter anderem, dass ein Web Service ein oder mehrere Dienste bereitstellt, welche über das Web in Anspruch genommen werden können. Jeder Web Service besitzt auch immer einen Uniform Resource Identifier (URI), mit dem dieser eindeutig identifiziert werden kann. Wie oben schon genannt, erfolgt die Kommunikation mit Web Services ausschließlich zwischen Maschinen, womit diese zu Webanwendungen stark abgegrenzt werden, bei denen wiederum eine menschliche Interaktion erfolgt. Außerdem gibt es zwei Eigenschaften, die für die Popularität der Web Services entscheidend sind.

#### Plattformunabhängigkeit

Client und Server (in diesem Fall der Web Service) können unabhängig von ihrer jeweiligen Konfiguration miteinander kommunizieren. So kann ein Client, welcher unter Linux läuft, beispielsweise problemlos mit einem Windows Server interagieren und umgekehrt. Daraus folgt auch, dass es eine Schnittstelle zwischen beiden Maschinen geben muss, die einheitlich ist. Meistens übertragen die Kommunikationspartner die Daten über XML, JavaScript Object Notation (JSON) oder ähnliche Strukturen.

#### Verteilt

Ein Server stellt seinen Dienst nicht nur für genau einen Clienten bereit, sondern muss auch mehrere Clienten bedienen können.

## 2.3 SOAP

Das SOAP, zu Deutsch „einfaches Objektzugriffsprotokoll“, ist ein Protokoll, welches in einer dezentralen Umgebung für den Austausch von Informationen zwischen verschiedenen Systemen genutzt werden kann. Es wurde ursprünglich von Microsoft entwickelt, um alte Übertragungstechnologien, wie beispielsweise das Distributed Component Object Model (DCOM) oder die Common Object Request Broker Architecture (CORBA), zu ersetzen. SOAP wurde als sehr flexibles, erweiterbares, aber auch leichtgewichtiges Protokoll entwickelt, weshalb auch häufig Abkürzungen, wie zum Beispiel Web Services Security (WS-Security), Web Services Reliable Messaging (WS-ReliableMessaging) oder Web Services Addressing (WS-Addressing) damit assoziiert werden. Es umfasst gewisse Regeln, welche die zu verschickenden Informationen komplexer machen und Overhead produzieren, jedoch für einen gewissen Standard sorgen, welchen jede SOAP Application Programming Interface (API) versteht. Dieses Verfahren befindet sich seit 2007 in der Version 1.2 und wurde als offizielles Protokoll von der W3C definiert [Vgl. Box+00].

Damit ein SOAP Web Service funktionieren kann, muss dieser ein WSDL Dokument bereitstellen. In diesem Dokument befinden sich alle Informationen über die angebotenen Methoden, die Parameter und die Rückgabewerte dieser und wie der Kontakt hergestellt werden kann. Dieses auf XML basierende Dokument ist für jeden (zugelassenen) Nutzer einsehbar, womit alle Funktionen genutzt werden können.
Die Zustellung einer SOAP Nachricht kann mit dem Prinzip einer Hauspost verglichen werden. Der abgeschickte Brief kommt zuerst im Sekretariat (SOAP Dispatcher) an, wo dieser dann geöffnet wird, um den richtigen Empfänger zu bestimmen. Die Nachricht wird dann an den eigentlichen Empfänger weitergeleitet, womit dieser dann darauf antworten kann. Der Client kann also nicht direkt mit dem Server kommunizieren, da alle Nachrichten immer vom Dispatcher gefiltert werden. Bei SOAP in Kombination mit HTTP weist diese Nachricht HTTP POST als Methode auf und stellt diesen Request immer an dieselbe Adresse, was in Abbildung 1 gezeigt ist.
Listing 1 stellt ein Beispiel einer SOAP Nachricht dar. Man sieht, dass hier drei Elemente definiert sind, welche im Folgenden näher erläutert werden.

### 2.3.1 Envelope

Der Envelope funktioniert quasi als „Briefumschlag“, welcher zwingend in dem zu übermittelnden Dokument angegeben sein muss. Das Element *kann*, wie in Listing 1 gezeigt, ein Attribut zur Definition des Namensraums enthalten. Des Weiteren *darf* es zusätzliche Attribute, wie zum Beispiel das Encoding, beinhalten, welche aber durch den vorhergehenden Namensraum qualifiziert sein müssen. Der Body *muss* als Kind Element vorhanden sein, jedoch *kann* davor noch ein Header stehen. Alle weiteren Attribute *müssen* nach dem Body eingefügt werden.
Da SOAP keine traditionelle Versionierung bereitstellt, *muss* ein Envelope den Namensraum „`http://schemas.xmlsoap.org/soap/envelope/`“ referenzieren. Falls eine Nachricht mit einem anderen Namensraum, als dem des Programmes empfangen wird, muss die Applikation dies als „Version Error“ behandeln und die Nachricht verwerfen. Wenn die Nachricht aber mit einem Frage/Antwort Protokoll (wie beispielsweise HTTP) geschickt wurde, muss auf die Nachricht mit einem „SOAP VersionMismatch“ Fehlercode geantwortet werden.

### 2.3.2 Header

Falls der SOAP Header existiert, *muss* dieser das erste Kind Element des Envelopes sein. Alle Kinder des Header Elements werden „Header Entries“ genannt. Jedes dieser Elemente muss mit seinem Namensraum URI und dem lokalen Namen voll identifizierbar sein, wobei der Namensraum immer gesetzt sein *muss*. Ansonsten existieren noch das EncodingStyle, MustUnderstand und Actor Attribut, welche alle genutzt werden *können*.

#### EncodingStyle

Dieses Attribut definiert, mit welchen Regeln die Nachricht serialisiert wurde. Es *kann* in jedem Element präsent sein und ist dann für alle Elemente innerhalb dieses Baumes verantwortlich, außer diese haben selbst jenes Attribut definiert.
Der Wert stellt sich als eine geordnete Liste von einem oder mehreren URIs dar, welche Regeln zur Deserialisierung beinhalten. Diese Liste ist nach absteigender Genauigkeit sortiert.

#### Actor

Der SOAP actor *kann* genutzt werden, um den Empfänger eines Header Elements zu kennzeichnen. Dies ist zwingend notwendig damit die Nachricht effektiv sein kann, da die Nachrichten durch eine große Menge an SOAP Vermittlern geleitet wird, welche diese entweder empfangen oder weiterleiten können. Daher sind nicht alle Teile sinnvoll oder bestimmt für das endgültige Ziel dieser Nachricht, sondern können weitergeleitet und auch verändert werden. Die Applikation, welche Attribute verändert hat, ist dann der neue „Partner“ des finalen Empfängers.

#### MustUnderstand

Dieses Attribut *kann* genutzt werden, um dem Leser zu signalisieren, ob dieser den Eintrag verpflichtend oder optional verarbeiten muss. Falls der Wert auf 1 steht, *muss* der Empfänger entweder das Element korrekt nach den Richtlinien verarbeiten oder bei der Weiterverarbeitung der Nachricht fehlschlagen. Um die Effektivität der Nachrichtenverarbeitung gewährleisten zu können, muss dieses Attribut ebenfalls gesetzt sein.

### 2.3.3 Body

Der Body einer SOAP Nachricht *muss* direkt nach dem Header folgen, falls dieser präsent ist, andernfalls *muss* er das erste Element sein. Dieses Element stellt einen simplen Mechanismus zur Informationsübertragung für den finalen Empfänger der Nachricht bereit. Äquivalent zum Header nennt man auch hier die Kind-Elemente „Body Entries“, welche mit deren Namensraum URIs und den lokalen Namen voll identifizierbar sein *müssen*. Des Weiteren *kann* der EncodingStyle definiert werden.

## 2.4 REST

Anders als SOAP ist die Representational State Transfer (REST) API kein offizielles Protokoll, sondern beschreibt eine Schnittstelle, welche sich an den Prinzipien und Denkweisen des WWW, demzufolge auch an den HTTP-Methoden, orientiert. Das bedeutet unter anderem, dass keine strikten Regeln festgelegt, sondern eher Empfehlungen zur Implementierung ausgegeben werden. Somit liegt der Schwerpunkt einer RESTful API darin, die Maschine-zu-Maschine Kommunikation über mehrere unterschiedliche Programmiersprachen zu ermöglichen. Dabei hat REST sechs Architekturprinzipien, welche im

Folgenden dargestellt werden.

### 2.4.1 Client-Server

Das Client-Server-Modell, auch Client-Server-Prinzip genannt, fordert eine strikte Aufgabenteilung. Der Server stellt Dienste bereit, die von (meistens mehreren) Clienten, unter Umständen auch gleichzeitig, genutzt werden können. Dadurch entsteht eine Architektur, in der die Server passiv agieren. Diese warten auf eingehende Verbindungswünsche und Anfragen und die Clienten fordern aktiv einen Dienst an. Ein Dienst definiert in diesem Fall wie und welche Daten ausgetauscht werden sollen.

### 2.4.2 Zustandslosigkeit

Clienten und Server müssen zustandslos kommunizieren, was bedeutet, dass es (streng genommen) keine „Sitzungsverwaltung“ gibt. Dadurch folgt, dass jede REST Nachricht immer alle Informationen bezüglich des Clienten und Server beinhalten muss. Dies begünstigt dann unter anderem die „Skalierbarkeit“ eines Web Services, sprich die Aufgabenverteilung auf beliebig viele Maschinen wird erleichtert.

### 2.4.3 Caching

Das Caching wird genutzt, um die Effizienz und Schnelligkeit der Anwendung zu verbessern. Die Informationen müssen dann entsprechend als „cacheable“ oder eben „non-cacheable“ gekennzeichnet werden. Wenn also eine Nachricht „cacheable“ beinhaltet, so werden die Nutzdaten clientseitig im „Cache“ gespeichert und bei jeder erneuten Anfrage an den Server abgerufen. Dies birgt natürlich das Risiko, veraltete Daten aus dem Cache zu bekommen, weshalb die Kennzeichnung so wichtig ist.

### 2.4.4 Einheitliche Schnittstelle

REST setzt auf eine einheitliche Schnittstelle, damit alle Dienste gleich angesprochen werden können. Dieses Prinzip unterscheidet sich nochmals in vier Aspekte.

#### Adressierbarkeit von Ressourcen

Jede RESTful API hat eine eindeutige Adresse, den Uniform Resource Locator (URL), welche nur durch die Methoden des Netzwerk Protokolls HTTP manipuliert werden können. Abbildung 2 zeigt ein Beispiel für die Adressierung von Objekten. Zu sehen ist, dass die Anfrage direkt auf eine Methode des Web Services routet, welche den Request dann bearbeiten und die Antwort zurückschicken kann.

#### Repräsentationen zur Veränderung von Ressourcen

Jede so angefragte Ressource kann die Antwort in unterschiedlichen Darstellungsformen, beispielsweise in Hypertext Markup Language (HTML), JSON oder XML kodieren. Alle

unter dieser Adresse zugänglichen Dienste können dabei voneinander verschiedene Repräsentationen haben, die aber dringend für eine RESTful API gleich sein sollten.

#### Selbstbeschreibende Nachrichten

Nachrichten, welche über den REST Dienst verschickt werden, sollen selbst beschreibend sein. Jede Anfrage soll also genau das darstellen, wofür die deklarierte Methode gedacht ist. Eine „DELETE“ Nachricht soll zum Beispiel nur das Löschen anfordern und keine anderen Anfragen.

#### HATEOAS

Das Prinzip der Hypermedia as the Engine of Application State (HATEOAS) ist laut Roy Fielding[2] die wichtigste Eigenschaft der REST Architektur. Wichtig ist hierbei, dass der Client als Antwort vom Server immer alle aktuell verfügbaren URLs bekommt und damit mit den sogenannten „Follow-up-Links“ weiter mit dem Server kommunizieren kann.

### 2.4.5 Mehrschichtige Systeme

Ganz nach dem Vorbild des Schichtensystems des Internets[3] setzt auch REST auf die Mehrschichtigkeit der Systeme. Somit kann ein Client die oberste Schicht sehen und ansprechen, was aber intern eigentlich passiert, bleibt dem Server vorbehalten und verborgen. Dadurch kann eine bessere Übersichtlichkeit und Skalierbarkeit erreicht werden. Der Nachteil daran ist natürlich der Overhead durch zusätzliche Funktionsaufrufe.

### 2.4.6 Code on Demand (optional)

Diese letzte und optionale Bedingung beschreibt, dass der Client, zur besseren Bedienung des Servers, Code, zum Beispiel in Form von Skripten oder ähnlichem, nachladen und clientseitig ausführen kann.

## 2.5 Vergleich: REST vs. SOAP

Tabelle 1 zeigt im Groben die Gemeinsamkeiten und Unterschiede der beiden Architekturen. Auffallend ist, dass beide ein URI Adressmodell besitzen und über das HTTP Protokoll gesteuert bzw. angesprochen werden können. Das vereinfacht den Vergleich bezüglich der Performance enorm, welcher bewusst in der Tabelle fehlt, da verschiedene Gesichtspunkte unter Performance beleuchtet werden können. Wenn man aber nur den reinen Datenverkehr und damit zusammenhängend auch die Zeit nimmt, die eine Nachricht vom Clienten zum eigentlichen Endpunkt braucht, so ist REST um einiges schneller. Dies liegt vor allem an dem Overhead, welcher durch den SOAP Dispatcher und die Anforderungen an eine SOAP Nachricht entsteht.

[2] Roy Thomas Fielding ist ein US-amerikanischer Informatiker, welcher maßgeblich zur Entwicklung des HTTP und der REST Architektur beitrug.
[3] s. Abbildung 3

Des Weiteren wird die Schnittstellenbeschreibung nur vom SOAP Standard gefordert, was sowohl positiv als auch negativ gesehen werden kann. Diese gibt nämlich Aufschluss darüber, welche Methoden und Services zur Nutzung bereitstehen. Jedoch wird auch wieder eine zusätzliche Anforderung an die Entwickler gestellt, selbst wenn das eventuell gar nicht benötigt wird. Auf der anderen Seite bietet eine RESTful API eine generische Beschreibung, optimalerweise als Antwort auf jede Nachricht an den Server, welche durch das Prinzip der HATEOAS als verfügbare URIs zurückgegeben wird. Die Schnittstellen sind daher auch unterschiedlich, da sich REST strikt an die HTTP-Methoden hält, und SOAP eben die Freiheit besitzt, die Schnittstellen anwendungsspezifisch zu definieren und in der Beschreibung des Web Services festhalten zu können. Eben hierbei zeichnet sich wieder die Leichtgewichtigkeit einer REST API dadurch ab, dass die vordefinierten Methoden genau so benutzt werden können und nichts zusätzlich implementiert werden muss.
Zusammenfassend kann gesagt werden, dass es immer noch auf die persönlichen Vorlieben der Entwickler oder Firmen ankommt, welche Architektur tatsächlich genutzt werden soll. Für SOAP gibt es zum Beispiel unzählige Bibliotheken, die die Implementierung eines solchen Services erheblich vereinfachen. Der Architekturstil REST wird allerdings immer häufiger genutzt, da die Kopplung an die Funktionsweise des HTTP für viele einen großen Vorteil darstellt und dadurch auch mehr Frameworks auf den Markt kommen.

## 2.6 Verschlüsselung

Die Kryptographie, also Ver- und Entschlüsselungen von Informationen, hat eine sehr lange Geschichte und lässt sich bis ins alte Ägypten zurückführen. Konkret geht es mathematisch gesehen hierbei um eine Funktion oder auch Matrix, welche den „Klartext“ in einen „Geheimtext“ umwandelt. Dabei kann man die Zeichen des zu verschlüsselnden Textes entweder transponieren oder mit anderen Zeichen substituieren. Im besten Fall kann der „Geheimtext“ dann nur mit dem Wissen über den Schlüssel wieder in den „Klartext“ umgewandelt werden. Heutzutage unterscheidet man zwischen symmetrischen und asymmetrischen Verschlüsselungsverfahren, welche aber häufig zusammen eingesetzt werden, um die Rechenzeit zu verringern. Da asymmetrische Verschlüsselungsverfahren sicherer sind, aber viel mehr Zeit und Leistung benötigen, werden diese häufig nur zur Schlüsselübergabe verwendet und die Nutzdaten dann mit einem symmetrischen Verfahren verschlüsselt.

### 2.6.1 Symmetrische Verschlüsselung

Wie der Name schon sagt, erfolgen hierbei Chiffrierung und Dechiffrierung mit dem gleichen Schlüssel. Dieses Verfahren wird auch „Private-Key-Verfahren“ genannt, da die Kommunikationspartner sicherstellen müssen, dass der gemeinsame Schlüssel privat, also geschützt vor öffentlichen Zugriffen, bleibt. Das erste bekannte Verfahren hierzu war die sogenannte „Caesar-Chiffre“, bei welcher das Alphabet gegeneinander verschoben wird. Mathematisch ist die Verschlüsselung als $C_K(P) = (P+K) \bmod 26$ mit $P$ als Buchstabe des „Klartextes“, $K$ als Anzahl der Verschiebung der Zeichen und $C$ als chiffrierter Buchstabe

darstellbar. Die Entschlüsselung ist dann dementsprechend $P_K(C) = (C - K) \bmod 26$. Hierbei erkennt man schon, dass sich ein solches Verfahren relativ einfach durchschauen lässt. Der nächste Schritt war also, nicht mehr nur eine Funktion zu verwenden, sondern ganze Algorithmen, welche mehrfach transponieren und substituieren. Der aktuelle symmetrische Verschlüsselungsstandard ist der Advanced Encryption Standard (AES), welcher im Oktober 2000 zertifiziert wurde und aus dem Data Encryption Standard (DES) entstanden ist. Die Technologie beruht auf der Blockverschlüsselung, wobei einzelne Blöcke von 128, 192 oder 256 Bit verschlüsselt werden. Durch die Schlüssellänge wird AES als sehr sicher bezeichnet und wird z. B. bei Wi-Fi Protected Access 2 (WPA2), Secure Shell (SSH) oder aber auch bei Komprimierungsverfahren wie 7-Zip (7z) verwendet.

### 2.6.2 Asymmetrische Verschlüsselung

Die asymmetrische Verschlüsselung, auch „Public-Key-Verfahren“ genannt, beruht auf der Verwendung von zwei statt nur einem Schlüssel. Dabei ist einer der beiden Schlüssel öffentlich, welcher zum Chiffrieren der Nachricht genutzt wird, der andere ist privat und wird zum Dechiffrieren der Nachricht genutzt. Dieses Verfahren funktioniert, da sogenannte Einwegfunktionen existieren, welche einfach zu berechnen aber schwierig umzukehren sind. Beispiele für solche Funktionen sind Modulo Operationen, Multiplizieren von Primzahlen oder Potenzieren. Für alle diese Funktionen gilt, dass $y = f(x)$ „einfach“, also in akzeptabler Zeit zu berechnen ist, aber die Umkehrfunktion, also $f^{-1}(y) = x$ schwierig und nicht in akzeptabler Zeit ohne Vorkenntnis herausgefunden werden kann. Von diesem Verfahren existieren nicht so viele, wie bei den symmetrischen Chiffrierungen. Die bekanntesten sind der Diffie-Hellman-Merkle Schlüsselaustausch und die RSA - Rivest, Shamir und Adleman Verschlüsselung. Beide Verfahren profitieren davon, dass die oben erwähnten Einwegfunktionen auch mit „Falltür“ existieren, was bedeutet, dass man die Umkehrfunktion durch Kenntnis einer Variablen leichter berechnen kann.

#### Diffie-Hellman-Merkle Schlüsselaustausch

Beim Diffie - Hellman - Merkle Schlüsselaustausch, häufig auch nur Diffie - Hellman genannt, wird angenommen, dass das Potenzieren einer Zahl einfach, jedoch das Logarithmieren schwierig ist. Angenommen es gibt zwei Partner, **A** und **B**, welche sich auf zwei Zahlen verständigen, eine Primzahl $p$ und eine Basis $g$. Des Weiteren generieren beide einen (zufälligen) privaten Schlüssel, $a$ für Partner **A** und $b$ für Partner **B**. Diese Schlüssel werden dann genutzt, um den öffentlichen Schlüssel von **A** mit $\alpha = g^a \bmod p$ und analog von **B** mit $\beta = g^b \bmod p$ zu berechnen, welche dann ausgetauscht werden. Durch diesen Austausch und der Rechnung $\Psi = \beta^a \bmod p$, oder für **B** $\Psi = \alpha^b \bmod p$ haben beide Partner den gemeinsamen Schlüssel $\Psi$.

#### RSA - Rivest, Shamir und Adleman

Die RSA Verschlüsselung beruht darauf, dass die Primfaktorzerlegung großer Zahlen zu rechenaufwändig ist. Seien nun wieder zwei Partner **A** und **B** beteiligt, wobei dieses Mal **A**

als Empfänger fungiert. **A** wählt die (möglichst großen) Primzahlen $p$ und $q$, berechnet daraus $n$ mit $n = p \cdot q$ und $\phi(n) = (p-1) \cdot (q-1)$ mit der Eulerschen $\phi$-Funktion. Danach sucht **A** $e$ mit $ggT(e, \phi(n)) = 1$ und berechnet im Anschluss $d$ mit $d \cdot e \bmod \phi(n) = 1$. Die beiden Zahlen $n$ und $e$ werden nun veröffentlicht und der Sender der Nachricht, in diesem Falle **B**, kann diese jetzt mit $c = m^e \bmod n$ verschlüsseln. Die Daten werden von **A** wieder mit $m = c^d \bmod n$ entschlüsselt.

## 2.7 TLS/SSL

Bereits neun Monate nach der Verbreitung des ersten Webbrowsers „Mosaic“ 1994 stellte Netscape Communications die erste Version des Secure Sockets Layer (SSL) Protokolls (1.0) vor. Nur fünf Monate später kam die Version 2.0, zusammen mit dem neuen Browser „Netscape Navigator“, auf den Markt. Die Firma übergab letztendlich im Sommer 1996 die Versionsverwaltung über die Version 3.0 an die Internet Engineering Task Force (IETF), um einen einheitlichen Internet-Standard zu entwickeln. 1999 wurde dann SSL in Transport Layer Security (TLS) umbenannt und unter „The TLS Protocol Version 1.0“ [Vgl. AD99] veröffentlicht. Heutzutage werden die Versionen 1.2 [Vgl. RD08] und 1.3 [Vgl. Res18] genutzt, da frühere Standards seit März 2021 unzulässig sind.
Das TLS Protokoll wird genutzt, damit sensible Daten bei der Übertragung über das Internet nicht gelesen oder manipuliert werden können. Im Protokollstapel des ISO/OSI-Referenzmodells befindet sich TLS auf der Ebene 5, der Sitzungsschicht, auch „Session-Layer“ genannt. Diese Schicht ist damit genau zwischen Transport- und Darstellungsschicht, also zwischen Protokollen, wie TCP und HTTP. Aufgabe hiervon ist es also, Daten aus den höheren Ebenen zu verschlüsseln und an die Transportschicht weiterzugeben. Genauer besteht das Protokoll aus zwei Schichten, auf welche im Folgenden näher eingegangen wird.

### 2.7.1 TLS Handshake Protocol

Das Handshake Protokoll ermittelt den stärksten, gemeinsam unterstützten Algorithmus, authentifiziert die Kommunikationspartner und bestimmt optionaler Weise einen Sitzungsschlüssel für die symmetrische Verschlüsselung. In vereinfachter Darstellung passieren dabei folgende Schritte. Der Client schickt eine „Hello“ Nachricht an den Server, in welcher dieser alle kryptografischen Informationen, wie unterstützte Algorithmen, eine Zufallsnummer des Clienten und eine Session ID mitteilt. Der Server antwortet seinerseits mit einem „Hello“, in welchem der stärkste gemeinsame Algorithmus, eine Zufallsnummer des Servers und die Session ID übertragen wird.

### 2.7.2 TLS Record Protocol

Das Record Protocol (engl. Record Layer) ist vollständig getrennt vom TLS Handshake Protocol. Es verschickt die Daten symmetrisch mit den vorher ausgehandelten Verschlüsselungsalgorithmen und Sitzungsschlüsseln. Das Effiziente hierbei ist, dass die Sicherheit der asymmetrischen Verschlüsselung nur beim Aushandeln des Schlüssels genutzt wird,

während die eigentlichen Daten dann mit der symmetrische Verschlüsselung verschickt werden. Damit geht man dem Problem der Performance-Einbuße aus dem Weg.

## 2.8 Web-Security

Unter dem Begriff Web-Security versteht man im Allgemeinen die Absicherung der Übertragung von Dateien über das WWW und damit auch in Verbindung mit dem HTTP.
Da das HTTP grundsätzlich unverschlüsselt ist und daher jeder Mittelsmann die Kommunikation mithören und insbesondere auch verändern kann, muss Hypertext Transfer Protocol Secure (HTTPS) genutzt werden. Das Protokoll wird auch „HTTP over TLS“ genannt und liegt in der Schichtenarchitektur[4] auf der Anwendungsebene. Die Übertragung der Daten beim HTTPS ist identisch zur Übertragung der Daten mit dem HTTP. Der Unterschied beider Protokolle liegt nur darin, dass bei HTTPS die Nutzdaten durch TLS/SSL verschlüsselt wurden, jedoch IP-Adresse und Port weiterhin sichtbar sind. Ein wichtiger Begriff hierbei ist das sogenannte „X.509“ [Vgl. Boe+08] Zertifikat. Dieses folgt einer gewissen Struktur und beinhaltet unter anderem die Version, die Algorithmen-ID, den Aussteller, die Gültigkeit und den Inhaber. Mit solch einem Zertifikat kann ein Browser sicherstellen, dass tatsächlich mit der Internetseite/ dem Web Service kommuniziert wird. Die Zertifikate werden von einer „Certificate-Authority (CA)“ ausgestellt, welche die Signatur übernimmt und diese in einen Verzeichnisdienst, welcher bereits ausgestellte Zertifikate enthält, einträgt.

[4] s. Abbildung 3

# 3 Absicherung des Informationsaustauschs

Bisher wurde also behandelt, wie die zur Übertragung freigegebenen Daten abgesichert werden können und wie sichergestellt werden kann, dass tatsächlich mit dem erwarteten Partner kommuniziert wird. Im Folgenden werden nun auf Basis dessen bekannte und häufig benutzte Sicherheitskonzepte in Verbindung mit dem HTTP vorgestellt.

## 3.1 HTTP Authentication

Die wohl einfachste Methode, eine Authentifizierung über das HTTP durchzuführen, ist es, ein schon vorhandenes Verfahren zu nutzen. Dieses ist in RFC 7235 [Vgl. FR14a] spezifiziert und basiert auf einem einfachen „challenge-response“ Modell, wobei die „challenge“ die Frage nach dem korrekten Passwort und die „response“ die Antwort mit dem korrekten Passwort darstellt. Der Client bekommt für jede Anfrage an einen Bereich, welcher eine Autorisierung benötigt, den Status-Code 401 (Unauthorized) vom Server zugeschickt. Zusätzlich muss der Server noch einige Header-Felder füllen und mitsenden, um dem Clienten unter anderem die Authentifizierungsmethode mitzuteilen. Das Feld „WWW-Authenticate“ *muss* zuerst die Methode und danach mindestens eine „challenge“ enthalten, in welcher der Bereich spezifiziert ist. Um nicht immer wieder die Nutzerdaten authentifizieren zu müssen, gibt es noch das Feld „Authorization“, welches in der Form „Authorization = credentials“ die Informationen beinhaltet. Damit kann der Server dem Clienten Zugang zu weiteren Bereichen mit der gleichen Authentifizierungsmethode gewähren. Theoretisch gibt es noch zwei Proxy Felder, welche hier jedoch vernachlässigt werden.
Im Folgenden werden ausgewählte Schemata aufgezeigt, die bei dieser Art der Authentifizierung genutzt werden können.

### 3.1.1 HTTP Basic Authentication

Die sogenannte „HTTP Basic Authentication“ [Vgl. Res15], setzt den Benutzernamen und das übergebene Passwort in der Form „BENUTZER:PW“ zusammen, kodiert alles mit Base64 und schickt es an den Server. Als „challenges“ werden hierbei der Bereichsname, mit „realm=<Bereichsname>“, und ein optionaler Parameter „charset“, welcher das Kodierungsschema angibt, erwartet. Alle anderen Parameter müssen ignoriert werden.
Diese Methode wird nur in Verbindung mit HTTPS als sicher gewertet, da die Passwörter ansonsten quasi im Klartext mitgelesen werden können.

### 3.1.2 HTTP Digest Access Authentication

Die Antwort des Servers beinhaltet bei diesem Schema im „WWW-Authenticate“ Header, neben Bereich und Namen, auch „qop“, „nonce“ und „opaque“. Der „Quality of Protection (qop)“ Parameter *muss* gesetzt sein und beschreibt die Sicherheitsqualität des Servers. Werte hierfür können „auth“ für Authentifizierung und/oder „auth-int“ für Authentifizierung mit Integritätsschutz sein. Ein einzigartiger, vom Server generierter String wird „nonce“

zugewiesen, der bei jeder „401“ Response neu generiert wird. Die Inhalte dieser Zeichenfolge sind abhängig von der Implementierung, könnten aber beispielsweise etwas wie „timestamp : ETag : secret-data“ [Vgl. SAB15] sein, was mit Base64 kodiert wird. Der „opaque“ Wert ist ebenfalls ein vom Server generierter String (Base64 oder hexadezimal), welcher einfach unverändert vom Clienten für den gleichen Bereich zurückgeschickt wird. Zur besseren Übersicht werden alle Werte, mit denen der Client antwortet, in Tabelle 2 dargestellt und erklärt.
Falls der Server nun einen erfolgreichen Datenabgleich durchgeführt hat und der Client so authentifiziert werden konnte, wird ein „200 OK“ Code mit der gewünschten Seite/ dem gewünschten Inhalt zurückgeschickt.

### 3.1.3 NTLM HTTP Authentication

Das ursprünglich von Microsoft entwickelte NTLM Schema bietet eine weitere Anmeldemöglichkeit auf Servern, unter anderem mit dem Single Sign-On in Kombination mit einer Windows-Benutzeranmeldung. Bei diesem Schema wird als Antwort auf die Anfrage einer Resource der „WWW-Authenticate“ Header mit dem Namen der „challenge“ zurückgeschickt. Der Client schickt daraufhin im „Authorization“ Header seinen Benutzernamen, welcher mit dem SPNEGO GSSAPI Mechanismus [Vgl. Lea+05] generiert wurde. Diese Anfrage wird dann vom Server seinerseits dekodiert, mit einer Sicherheitsfunktion innerhalb des Protokolls überprüft, und falls der Kontext vollständig ist, wird die geforderte Resource mit dem erneut generierten Hash des Mechanismus an den Clienten geschickt. Falls der Kontext nicht vollständig war, wiederholt sich dieser Prozess zwischen Client und Server so lange, bis der Server von der oben genannten Sicherheitsfunktion einen Wert zurückbekommt, der anzeigt, dass der Prozess erfolgreich war.

## 3.2 API Keys

Als API Key wird ein einmalig generierter Schlüssel bezeichnet, welcher bei der ersten Verbindung des Clienten mit dem Server erstellt wird. Dieser Schlüssel kann entweder in einem POST-Request als Request-Query, oder in einem GET-Request als Header, oder Cookie, geschickt werden. Dabei kann der Schlüssel für die Authentisierung genutzt werden, aber nicht für die Authentifizierung, was bedeutet, dass der Server dann weiß, mit wem er kommuniziert, den Clienten aber nicht auf seine Rechte überprüft. Dieses Verfahren ist ebenfalls nur im Kontext von HTTPS sicher, da ansonsten theoretisch sehr einfach der übermittelte Schlüssel abgegriffen werden könnte.

## 3.3 OAuth

Die Bezeichnung OAuth bezieht sich auf zwei unterschiedliche Protokolle, welche eine standardisierte, sichere Autorisierung erlauben. Die Version 1.0 des Protokolls wurde ab 2006 entwickelt und 2010 von der IETF veröffentlicht [Vgl. Ham10]. Im Oktober 2012 wurde die Version 2.0 eingeführt [Vgl. Har12], welche heutzutage häufiger in Implementierungen

zu finden ist. Wichtig dabei ist, dass die Versionen nicht kompatibel zueinander sind, da sie beispielsweise andere Rollen spezifizieren, auf welche im Folgenden näher eingegangen wird. Des Weiteren ist das OAuth Protokoll 1.0 unabhängig von dem darunterliegenden Transportprotokoll, während sich Version 2.0 vollkommen auf TLS/SSL stützt. Wer Version 1.0 nutzt, muss sich also selbst um die Transportsicherheit kümmern, kann allerdings jedes Transportprotokoll nutzen, während 2.0 nur in Kombination mit HTTPS genutzt werden kann. Dies führt auch zu einem weiteren Aspekt, dem der Benutzererfahrung. Vor allem bezüglich dieses Unterschieds der Sicherheit ist Version 2.0 leichter zu implementieren und benutzerfreundlicher, daher wird es auch im Folgenden näher betrachtet.

### 3.3.1 Beispiel

Zur besseren Übersicht wird der Protokollfluss zuerst anhand eines Beispiels erläutert. Es wird angenommen, dass eine Applikation namens „Witz des Tages“ existiert, welche täglich einen Witz an registrierte E-Mail-Adressen verschickt. Ein Benutzer möchte nun seine Bekannten an den Witzen teilhaben lassen und die E-Mail-Adressen dieser auch dort eintragen. Hierfür möchte er nicht alle Adressen eingeben, da diese schon in seinem Kontaktbuch stehen. Er hat sich über ein externes Programm an dieser Applikation angemeldet, welches im Folgenden als **Authorization Server** dargestellt ist.
Der **Resource Owner** (Benutzer) schickt also eine initiale Anfrage an die „Witz des Tages“ App. Daraufhin fragt der **Client** (Applikation) beim **Authorization Server** zusammen mit einigen Parametern (**Client ID**, **Redirect URI**, **Response Type** und **Scope**) den **Authorization Code** für die Weitergabe der E-Mail-Adressen an. Der **Resource Owner** (Benutzer) wird jetzt gefragt, ob er dem **Clienten** (Applikation) die angefragte Freigabe erteilen möchte, worauf dieser zustimmt. Im nächsten Schritt wird der **Resource Owner** (Benutzer) an den **Clienten** (Applikation) über die **Redirect URI**, zusammen mit dem generierten **Authorization Code**, zurückgeschickt. Dieser kann dann damit beim **Authorization Server**, in Kombination mit **Client ID** und **Client Secret**, ein **Access Token** anfragen. Hat der **Client** (Applikation) das **Access Token** erhalten, schickt dieser dies nun an den **Resource Server** und erhält im Austausch dafür die Kontakte.

### 3.3.2 Terminologien

In OAuth 2.0 existieren also einige Terminologien [Vgl. Har12], welche schon in das vorhergehende Beispiel eingearbeitet wurden. Diese sind für das Verständnis des Protokolls erforderlich und werden im Folgenden aufgeführt und näher erläutert.

**Resource Owner:** Stellt eine Entität (auch „end-user“ genannt, falls es eine Person ist) dar, welche Zugriff zu einer geschützten Resource gewährleisten kann.

**Client:** Eine Anwendung, welche die geschützte Resource im Namen des **Resource Owners** mit dessen Autorisierung anfragt. Dieser Terminus impliziert nicht, auf welchem System die Anfrage ausgeführt wird.

**Authorization Server:** Dieser Server authentifiziert den **Resource Owner** und stellt dem **Clienten** Token aus.

**Resource Server:** Ein Server, auf welchem diese geschützten Ressourcen liegen. Zudem ist dieser dazu in der Lage, auf Access Token zu antworten und diese zu akzeptieren.

**Redirect URI:** Eine URL, auf die der **Resource Owner** zurückgeschickt wird, nachdem dem **Clienten** die Erlaubnis erteilt wurde.

**Response Type:** Der vom **Clienten** erwartete Typ der Information. Meistens ist dies ein **Authorization Code**.

**Scope:** Beschreibt die vom **Clienten** angefragten Informationen. (Bsp.: Die E-Mail-Adressen der Bekannten auslesen und weitergeben.)

**Consent:** Der **Authorization Server** fragt den **Resource Owner** nach einer Zustimmung, ob der **Client** die angefragten Informationen benutzen darf. (Bsp.: „Die Applikation möchte die E-Mail-Adressen aus dem Kontaktbuch auslesen. Zulassen / Verweigern“)

**Client ID:** Eine ID für die Identifizierung des **Clienten** beim **Authorization Server**.

**Client Secret:** Stellt ein geheimes Passwort dar, welches nur **Client** und **Authorization Server** kennen. Dies wird genutzt, um private Informationen zu teilen.

**Authorization Code:** Ein temporärer Code, welcher vom **Clienten** an den **Authorization Server** gegeben wird. Der **Client** erhält dafür ein **Access Token**.

**Access Token:** Wie der Name schon andeutet, beinhaltet dieses Token spezifische Bereiche und die Dauer des gewährten Zugangs. Um auf eine Resource zuzugreifen, muss dieses Token vom Clienten an den **Resource Server** als Autorisierung übermittelt werden.

**Refresh Token:** Das Refresh Token wird verwendet, um beim „Authorization Server“ ein neues Access Token anzufragen, falls das vorherige abgelaufen ist. Dadurch kann man die Lebensdauer des Access Tokens kürzer gestalten, was die Sicherheit dieses Protokolls erheblich erhöht. Ein Angreifer kann sich selbst mit Kenntnis des Access Tokens nur in einem gewissen Zeitraum als der eigentliche autorisierte Nutzer ausgeben, da das Token seine Gültigkeit verliert.

### 3.3.3 Ablauf der Autorisierung

Der abstrakte Ablauf einer Autorisierung stellt sich dabei, wie in Abbildung 4 zu sehen ist, dar:

(A) Der **Client** beantragt Autorisierung vom **Resource Owner**. Diese Anfrage kann entweder direkt an den **Resource Owner** gestellt werden, oder besser indirekt an den **Authorization Server**.

(B) Der **Resource Owner** antwortet mit einer Autorisierungsgenehmigung, welche eine aus vier zugelassenen Typen sein kann, die jeweils abhängig von den unterstützten Typen des **Authorization Servers** sind.

(C) Der **Client** fordert ein Access Token beim **Authorization Server** an und schickt die vorher erhaltene Autorisierungsgenehmigung mit.

(D) Der **Authorization Server** authentisiert den **Clienten**, validiert die Autorisierungsgenehmigung und erstellt ein Access Token.

(E) Der **Client** fragt die Resource beim **Resource Server** an und authentifiziert sich mit dem Access Token.

(F) Der **Resource Server** prüft das Token und gibt Zugriff auf die Resource.

Der Ablauf der Autorisierung ist also meistens nicht genau so, wie die Abbildung OAuth 2.0 Abstrakter Protokoll Fluss zeigt. Bei den meisten Implementierungen interagiert der **Resource Owner** tatsächlich nur zu Anfang mit dem Clienten, danach folgt eine Interaktion zwischen **Authorization Server** und **Resource Owner**, wonach der **Authorization Code** an den **Clienten** geschickt wird.

## 3.4 OpenID Connect

OpenID Connect ist eine einfache Protokollschicht, angesiedelt oberhalb von OAuth 2.0. Während OAuth 2.0 als Autorisierungsprotokoll konzipiert ist, stellt OpenID Connect auch Authentisierung bereit. Es befähigt Clienten dazu, Identitäten auf Basis einer Authentifizierung durch den Authorization Server zu verifizieren, sowie grundlegende Informationen über den Benutzer auf eine REST-ähnliche Weise zu erhalten. Die Nachfrage nach authentifizierten Sitzungen kann dabei von web-basierten, mobilen oder JavaScript Clienten kommen [Vgl. Sak+14].

### 3.4.1 Terminologien

Für OpenID Connect gelten die gleichen Terminologien wie für OAuth, jedoch werden diese noch durch Folgende ergänzt:

**Claim:** Stellt ein Teil einer Information über eine Entität dar.

**OpenID Provider:** Der **OP** ist ein **Authorization Server**, welcher in der Lage ist, den **End-User** zu authentifizieren und der **RP Claims** zur Verfügung zu stellen.

**Relying Party:** Die **RP** ist eine OAuth 2.0 **Client** Applikation, welche **Claims** und Authentifizierung von einem **Open ID Provider** beantragen kann.

**ID Token:** Ein JSON Web Token (JWT) (s. Abbildung 2), was **Claims** über die Authentifizierung beinhaltet. Diese Token *kann* auch noch andere **Claims** beinhalten.

### 3.4.2 Ablauf der Autorisierung

Der abstrakte Ablauf stellt sich hierbei, wie in Abbildung 5 gezeigt, dar:

(1) Der **RP** sendet eine Anfrage an den **OP**.

(2) Der **OP** authentifiziert den „End-User“ und erhält Autorisierung.

(3) Der **OP** antwortet mit einem **ID-Token** und meistens auch mit einem **Access Token**.

(4) Der **RP** kann nun eine Anfrage mit dem erhaltenen Token an den „UserInfo Endpoint“ schicken.

(5) Der „UserInfo Endpoint“ gibt die **Claims** über den „End-User“ zurück.

Grundsätzlich unterscheidet sich der Ablauf auf den ersten Blick kaum von dem des OAuth 2.0 Protokolls. Bei OpenID Connect ist es allerdings so, dass anfangs bei der Anfrage an den **OP** der Wert von **Scope** auf „openid“ gesetzt wird, um dem **Authorization Server** mitzuteilen, dass eben dieses Protokoll genutzt wird. Der eigentliche Mehrwert oder besser die Erweiterung zu OAuth 2.0 besteht nun vor allem in den sogenannten **ID-Token**. Während OAuth 2.0 die Funktionalität bereitstellt, einem **Clienten** Zugang zu den eigenen Daten ohne Übermittlung des Passwortes durch ein **Access Token** zu gewähren, kann OpenID Connect zusätzlich die Identität des Benutzers prüfen. Dadurch kann jeder **Client** sicher sein, dass der so authentifizierte **End-User** tatsächlich derjenige ist, der er vorgibt zu sein. Das eben erwähnte **ID Token** enthält **Claims** über die Authentifizierung eines „End-Users“ von einem **Authorization Server** und andere potenziell angefragte Parameter. Es stellt sich als JWT dar und besitzt einige Schlüssel-Wert Paare, welche zum Beispiel auch in Abbildung 2 dargestellt sind. Hierbei findet sich unter anderem auch das „nonce“ aus dem darunterliegenden Protokoll wieder. Dieses Token kann nun vom **Clienten** genutzt werden, um am **UserInfo Endpoint** weitere Informationen über den Benutzer abzufragen und über ein neues **ID Token** zurückzubekommen.

# 4 Zusammenfassung und Ausblick

„Einfach Einloggen und Herunterladen.“ Für den einen oder anderen Leser der Seminararbeit mag das Wort „einfach“ in diesem Satz mittlerweile ziemlich beschönigend aussehen. Sicherlich bekommen Anwender eher weniger davon mit, aber Anwendungsentwickler und Sicherheitsexperten müssen sich mit dem gar nicht so einfachen Thema dann doch auseinandersetzen. Zusammenfassend kann also gesagt werden, dass die Seminararbeit die theoretischen Grundlagen — wenn auch nur oberflächlich — einer Datenübertragung über das Internet aufzeigt und Basiswissen über Kryptographie bereitstellt. Damit wurden die beiden ersten Ziele, die Datenübertragung über das Internet im Ansatz zu verstehen und verschiedene Architekturen zur Realisierung dieser kennenzulernen, abgedeckt.
Das zweite Kapitel handelte von einigen Verfahren zur Absicherung des Vorhergehenden und erfüllte damit das Ziel, die Wichtigkeit und Notwendigkeit der Absicherung eines Informationsaustauschs aufzufassen. Zuerst wurde die in HTTP integrierte Authentifizierung exemplarisch mit drei Varianten vorgestellt. Danach kamen kurz die API Keys, wobei bei beiden Verfahren darauf zu achten war, dass zwingend HTTPS verwendet werden muss, um diese als sicher einstufen zu können. Das OAuth, als relativ neues Protokoll, und OpenID Connect als zusätzliche Schicht dazu, rundete das Kapitel ab.
Für eine tatsächliche Implementierung eines Web Services mit beschränkten Zugriffsrechten und nur einer einmaligen Anmeldung eines Clienten würde sich eine RESTful API anbieten, da diese um einiges leichtgewichtiger ist und momentan die bevorzugte Wahl großer Anbieter darstellt.
Die Absicherung könnte dann entweder über eine Form der HTTP Authentication, über die ähnlichen API Keys oder eben über OpenID Connect funktionieren. Welches Verfahren sich aber tatsächlich dazu eignet und vor allem den vorgegebenen Richtlinien entspricht, muss vor der Umsetzung noch einmal ausführlich diskutiert werden.

## Literaturverzeichnis


[AD99] Christopher Allen und Tim Dierks. *The TLS Protocol Version 1.0*. RFC 2246. Jan. 1999. DOI: 10.17487/RFC2246. URL: https://rfc-editor.org/rfc/rfc2246.txt.

[Boe+08] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell und David Cooper. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. Mai 2008. DOI: 10.17487/RFC5280. URL: https://rfc-editor.org/rfc/rfc5280.txt.

[Boo+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris und David Orchard. *Web Services Architecture*. 2004. URL: https://www.w3.org/TR/ws-arch/ (besucht am 20. 10. 2021).

[Box+00] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte und Dave Winer. *Simple Object Access Protocol (SOAP) 1.1*. 2000. URL: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383512 (besucht am 20. 10. 2010).

[BPT15] Mike Belshe, Roberto Peon und Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. Mai 2015. DOI: 10.17487/RFC7540. URL: https://rfc-editor.org/rfc/rfc7540.txt.

[Bra97] Scott O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. März 1997. DOI: 10.17487/RFC2119. URL: https://rfc-editor.org/rfc/rfc2119.txt.

[FR14a] Roy T. Fielding und Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. RFC 7235. Juni 2014. DOI: 10.17487/RFC7235. URL: https://rfc-editor.org/rfc/rfc7235.txt.

[FR14b] Roy T. Fielding und Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. Juni 2014. DOI: 10.17487/RFC7230. URL: https://rfc-editor.org/rfc/rfc7230.txt.

[Ham10] Eran Hammer-Lahav. *The OAuth 1.0 Protocol*. RFC 5849. Apr. 2010. DOI: 10.17487/RFC5849. URL: https://rfc-editor.org/rfc/rfc5849.txt.

[Har12] Dick Hardt. *The OAuth 2.0 Authorization Framework*. RFC 6749. Okt. 2012. DOI: 10.17487/RFC6749. URL: https://rfc-editor.org/rfc/rfc6749.txt.

[Lea+05] Paul J. Leach, Karthik Jaganathan, Larry Zhu und Wyllys Ingersoll. *The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism*. RFC 4178. Okt. 2005. DOI: 10.17487/RFC4178. URL: https://rfc-editor.org/rfc/rfc4178.txt.

[NFB96] Henrik Nielsen, Roy T. Fielding und Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. Mai 1996. DOI: 10.17487/RFC1945. URL: https://rfc-editor.org/rfc/rfc1945.txt.

[PR85] J. Postel und J. Reynolds. *File Transfer Protocol*. RFC 959. Okt. 1985. DOI: 10.17487/RFC0959. URL: https://rfc-editor.org/rfc/rfc959.txt.

[RD08] Eric Rescorla und Tim Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://rfc-editor.org/rfc/rfc5246.txt.

[Res15] Julian Reschke. *The 'Basic' HTTP Authentication Scheme*. RFC 7617. Sep. 2015. DOI: 10.17487/RFC7617. URL: https://rfc-editor.org/rfc/rfc7617.txt.

[Res18] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://rfc-editor.org/rfc/rfc8446.txt.

[SAB15] Rifaat Shekh-Yusef, David Ahrens und Sophie Bremer. *HTTP Digest Access Authentication*. RFC 7616. Sep. 2015. DOI: 10.17487/RFC7616. URL: https://rfc-editor.org/rfc/rfc7616.txt.

[Sak+14] Nat Sakimura, John Bradley, Michael B. Jones, Breno de Medeiros und Chuck Mortimore. *OpenID Connect Core 1.0 incorporating errata set 1*. OpenID Connect Core 1.0 incorporating errata set 1. Nov. 2014. URL: https://openid.net/specs/openid-connect-core-1_0.html.

# A Anhang

## A.1 Abbildungen

**Abbildung 1:** *Beispiel SOAP-Kommunikation*

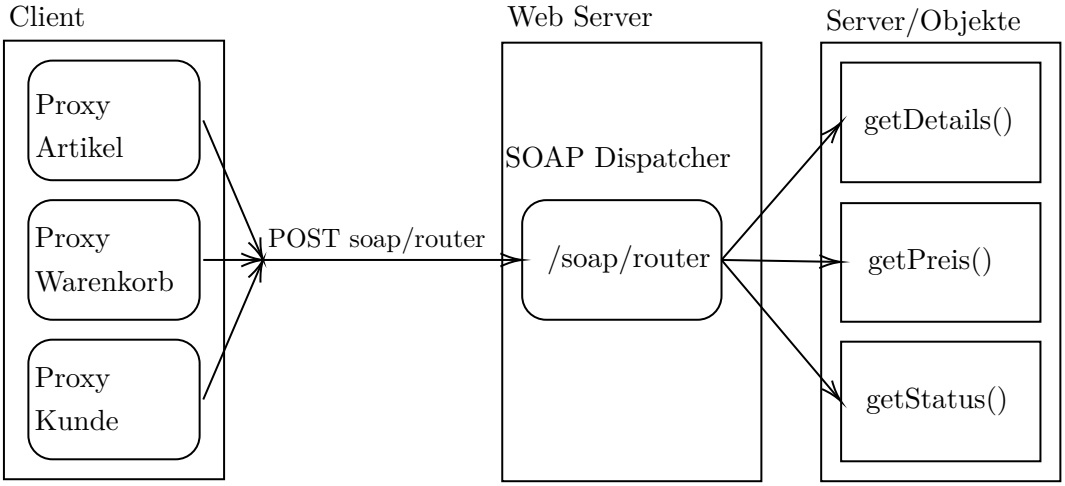



**Abbildung 2:** *Beispiel REST-Kommunikation*

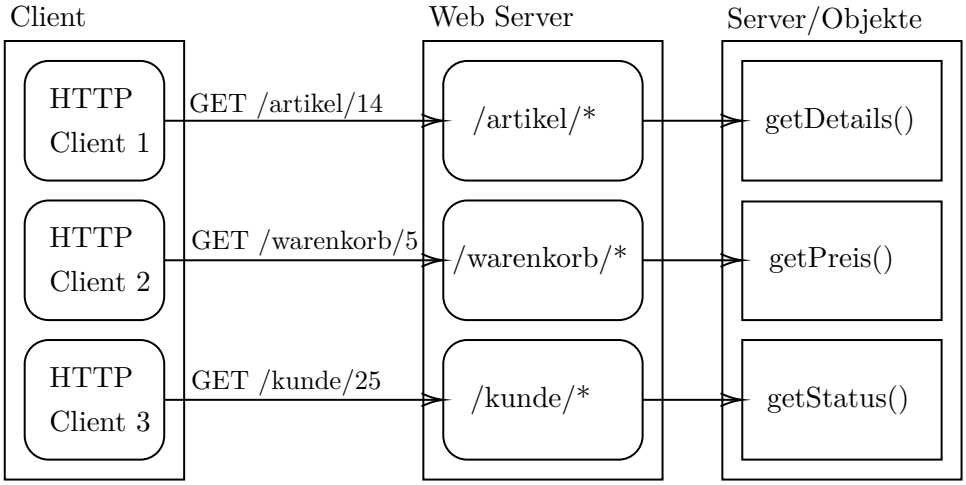


**Abbildung 3:** *Schichtenmodell*

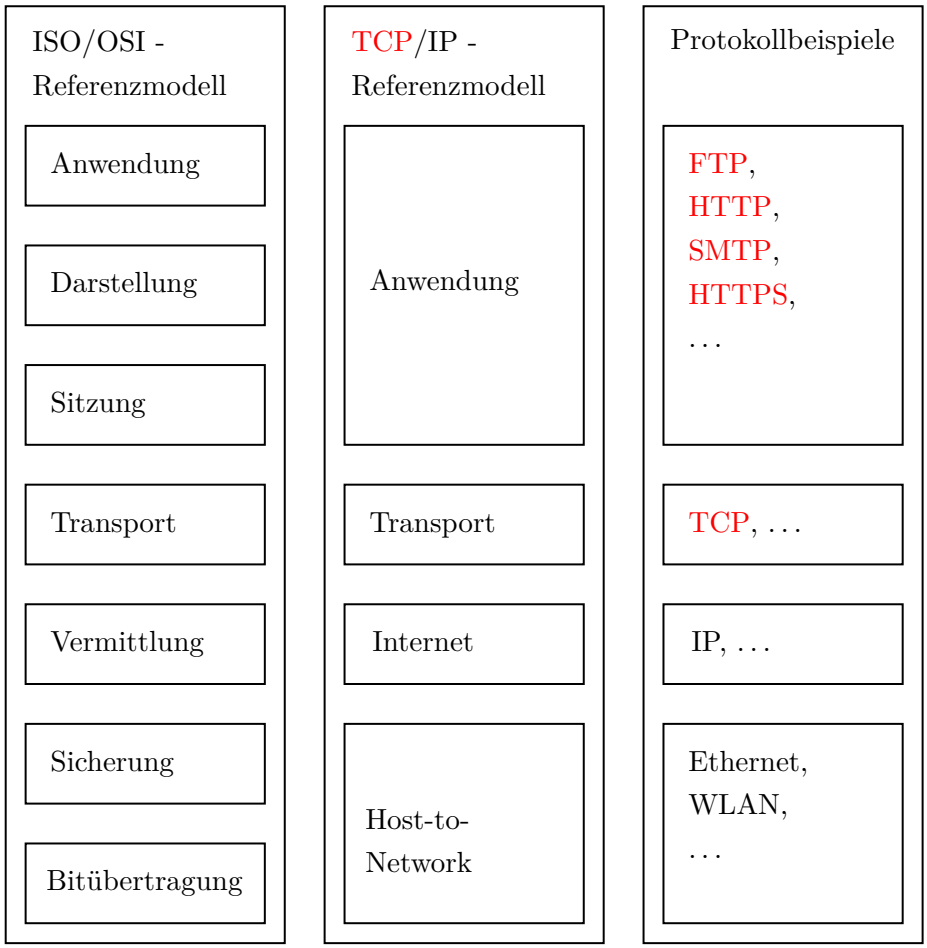



**Abbildung 4:** *OAuth 2.0 Abstrakter Protokoll Fluss*
*Quelle: Dick Hardt.* The OAuth 2.0 Authorization Framework. *RFC 6749. Okt. 2012.* DOI: 10.17487/RFC6749. URL: https://rfc-editor.org/rfc/rfc6749.txt

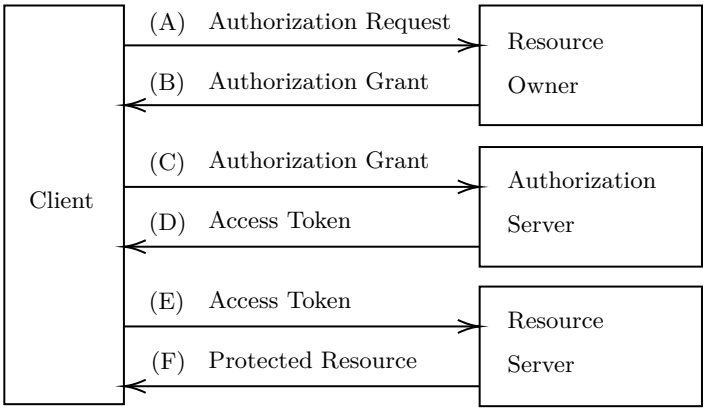
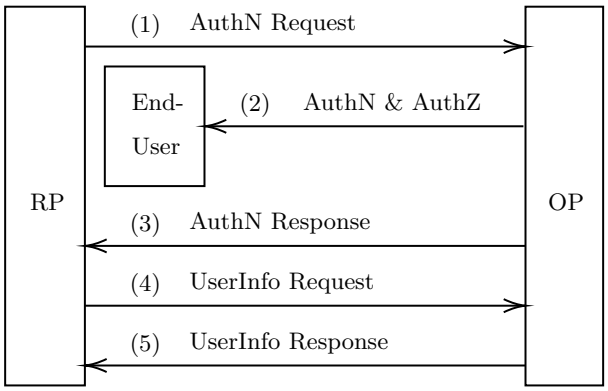


**Abbildung 5:** *OpenID Connect Core Abstrakter Protokoll Fluss*
*Quelle: Nat Sakimura u. a.* OpenID Connect Core 1.0 incorporating errata set 1. *OpenID Connect Core 1.0 incorporating errata set 1. Nov. 2014.* URL: https://openid.net/specs/openid-connect-core-1_0.html




## A.2 Listings

**Listing 1:** *SOAP Beispiel*

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
    <soap:Header>
        <m:RequestID xmlns:m="http://www.meinEigenerNamensraum.de/database">
            a3f5c109b
        </m:RequestID>
    </soap:Header>
    <soap:Body>
        <m:DbResponse xmlns:m="http://www.meinEigenerNamensraum.de/database">
            <m:title value="Folgen, Integrale, Flaechenberechnung">
                <m:choice value="1">Analysis</m:choice>
                <m:choice value="2">Lineare Algebra</m:choice>
            </m:title>
        </m:DbResponse>
    </soap:Body>
</soap:Envelope>
```

**Listing 2:** *JSON Web Token Beispiel*

```
{
  "iss": "https://server.example.com",
  "sub": "24400320",
  "aud": "s6BhdRkqt3",
  "nonce": "n-0S6_WzA2Mj",
  "exp": 1311281970,
  "iat": 1311280970,
  "auth_time": 1311280969,
  "acr": "urn:mace:incommon:iap:silver"
}
```

## A.3 Tabellen

**Tabelle 1:** *Vergleich: SOAP vs. REST*

| Thema \ Architektur | SOAP | REST |
|---|---|---|
| Schnittstellenbeschreibung | WSDL | keine, generisch |
| Adressmodell | URI | URI |
| Schnittstelle | anwendungsspezifisch | generisch durch HTTP-Methoden |
| Transport | HTTP, SMTP, ... | HTTP |
| Standards | W3C | keine, da Architekturstil |

**Tabelle 2:** *HTTP Digest Client Antwortfelder*

| Bezeichner | = | Wert |
|---|---|---|
| username | | Benutzername (Klartext) |
| realm | | Bereichsname |
| nonce | | nonce des Servers |
| uri | | Angeforderte Adresse |
| qop | | Sicherheitsqualität des Clienten |
| nc | | Anzahl der Anfragen (hexadezimal) |
| cnonce | | nonce des Clienten |
| response | | Zusammengesetzter Hashwert |
| opaque | | opaque des Servers |

**Eidesstattliche Erklärung**

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema

Architekturen und Verfahren zur Absicherung des Informationsausstauschs im Client-Server Modell

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

Name: Bergmann, Sven

Aachen, den 15.11.2021

S. Bergmann

Unterschrift der Studentin / des Studenten