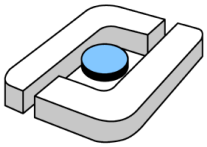

Hochschule Osnabrück

University of Applied Sciences

Fakultät

Ingenieurwissenschaften und Informatik

Wissenschaftliches Projekt

über das Thema:

Implementieren eines Microservice-basierten Authentifizierungsdienstes auf der Basis von OAuth 2.0 mittels Tokens am Beispiel von OpenSlides

Autor:	Gabriel Meyer
	gabriel.meyer@hs-osnabrueck.de
1. Prüfer:	Prof. Dr. A. Scheerhorn

Abgabedatum:	03.07.2020

I Kurzfassung

Authentifizierung ist ein wichtiger Bestandteil verschiedener Anwendungen, um eine eindeutige Zuordnung von Benutzer und Benutzerkonto herzustellen. Dabei haben sich zwei Verfahren etabliert: eine Authentifizierung mittels Sessions und eine Authentifizierung mittels Tokens. Diese Arbeit beschreibt die Entwicklung eines Authentifizierungsdienstes für OpenSlides auf Basis des Autorisierungsframeworks OAuth 2.0. Dafür werden beide Verfahren, Tokens und Sessions, untersucht und Unterschiede aufgezeigt. Anschließend wird die Nutzung des Frameworks OAuth 2.0 beleuchtet. Ziel ist es, einen Authentifizierungsdienst für OpenSlides auf Basis von OAuth 2.0 zu realisieren. Die Ergebnisse zeigen, dass sich Tokens für verteilte Systeme wie OpenSlides besser eignen, da sie clientseitig verwendet werden, sodass Ressourcen eines Servers geringgehalten werden und die Anwendung leicht skalierbar ist. Außerdem hat sich gezeigt, dass das Framework OAuth 2.0 nur mit Tokens umzusetzen ist. Eine Handhabung mit Sessions (insbesondere Session-Cookies) ist nicht vorgesehen. Des Weiteren ist zu beachten, dass mithilfe von OAuth 2.0 keine Authentifizierung stattfindet. Deshalb muss sie zusätzlich zu OAuth 2.0 eingebaut werden, wenn eine Authentifizierung erforderlich ist.

II Inhaltsverzeichnis

III Abbildungsverzeichnis

IV Glossar

Begriff	Definition
Authentifizierung	Durch eine Authentifizierung findet eine eindeutige Zuordnung von Benutzer zu Benutzerkonto statt (vgl. [Bih15]: S. 2). Dies wird beispielsweise im Kontext von OpenSlides mithilfe einer Benutzername-Passwort-Kombination ermittelt. Ein Benutzer kennt nur sein eigenes zum Benutzernamen spezifisches Passwort. Die Anwendung OpenSlides trifft eine eindeutige Zuordnung zu einem Benutzer, wenn die Benutzername-Passwort-Kombination des Benutzers eingegeben wird.
Autorisierung	Durch eine Autorisierung werden einem Benutzer oder einem System innerhalb einer Anwendung Rechte zugeordnet (vgl. [Bih15]: 2). Beispiel: Im Kontext von OpenSlides gibt es verschiedene Rollen, beispielsweise einen Administrator und einen Delegierten. Ein Administrator hat die Rechte dazu, einen neuen Benutzer anzulegen. Ein Delegierter hat solche Rechte nicht. Versucht ein Delegierter einen neuen Benutzer anzulegen, wird der Versuch abgelehnt, da OpenSlides durch eine Autorisierung feststellt, dass keine Berechtigung vorliegt.
Benutzerkonto	Ein virtuelles Objekt, das einen Benutzer in einer Anwendung repräsentiert.
Client	Nach [Lu18b] ist ein Client „*eine Software oder eine Hardware, die (Dienste) … eine(s) (Servers) in Anspruch nimmt*“ ([Lu18b]). Im Allgemeinen kommunizieren Clients mit Servern über Protokolle, beispielsweise HTTP. Ein Client wird HTTP-Client genannt, wenn er über HTTP mit einem HTTP-Server kommuniziert. Zum Beispiel sendet ein HTTP-Client eine Anfrage über HTTP an einen HTTP-Server. Dieser bearbeitet die Anfrage und sendet eine Antwort über HTTP zurück an den HTTP-Client. (vgl. [Lu18b]) Es kann je Anwendung für verschiedene Plattformen (Browser, Mobil) jeweils einen Client geben. In OpenSlides gibt es nur einen HTTP-Client, der in einem Browser aufrufbar ist. In dieser Arbeit

	wird ausschließlich ein HTTP-Client genutzt. Deshalb wird ein HTTP-Client in den nachfolgenden Abschnitten nur als „Client“ bezeichnet.
Cookie	Ein (HTTP-) Cookie ist eine Zeichenfolge, durch die Server und Client Informationen clientseitig setzen. Aufgebaut ist ein Cookie als Schlüssel-Werte-Paar, beispielsweise „test=hallowelt“. ‚test‘ ist der Schlüssel und ‚hallowelt‘ ist der Wert hinter dem Schlüssel. (vgl. [Cz16]) Ein Cookie kann verschiedene Eigenschaften haben: - „HttpOnly“: Ein Cookie mit dieser Eigenschaft kann nur von einem Server gesetzt und gelesen werden. Es ist für einen Client nicht ersichtlich. (vgl. [Mo20f]) - „SameSite“: Diese Eigenschaft verhindert, dass ein Cookie über mehrere Domänen versendet wird. (vgl. [Ch20]) - „Secure“: Durch diese Eigenschaft wird ein Cookie ausschließlich über HTTPS versendet (vgl. [Owa20a]).
Cross-Site Request Forgery (CSRF)	Bei einem „Cross-Site Request Forgery“-Angriff (CSRF-Angriff) wird der Zustand ausgenutzt, dass ein Benutzer an einer Anwendung angemeldet ist. Dazu sendet eine externe (bösartige) Anwendung eine Anfrage an einen Server. Bei der Anfrage werden Cookies automatisch an den Server gesendet. Durch ein Cookie, das die Identifikation eines Benutzers enthält, ist es so, als hätte der angemeldete Benutzer die Anfrage gesendet. Der Server erkennt dabei keinen Unterschied zwischen dem Benutzer und einem Angreifer. (vgl. [Bih15]: 154f)
Cross-Site-Scripting (XSS)	In der Umgebung eines Browsers wird JavaScript in Webanwendungen ausgeführt. Bei einem „Cross-Site-Scripting“-Angriff (XSS-Angriff) werden mithilfe eines JavaScript-Skripts Daten gelesen, die mittels JavaScript in die Browserumgebung geschrieben wurden. Dies schließt den Lokalen Speicher oder Sitzungsspeicher eines Browsers ein. (vgl. [Owa20])
Domäne	Domänen sind ein eindeutig bestimmter Ort im Internet, an dem Server angefragt und Ressourcen angefordert werden (vgl. [Mo87]).

Framework	Ein Framework ist wie ein Werkzeugkasten und bietet verschiedene Bausteine zur Entwicklung einer Anwendung an. Auf diese Weise kann beispielsweise ein Softwareentwickler eine Anwendung schneller fertigstellen als ohne Framework. Ohne Framework müsste er zu Beginn zunächst Bausteine entwerfen, die er für ein Endprodukt benötigt. (vgl. [Da19]) Jedes Framework wird zu einem bestimmten Zweck entwickelt. Es abstrahiert Konzepte und macht diese Entwicklern in Form von Bausteinen zugänglich. (vgl. [Da19])
HTTP-Anfrage	Ein Client, der eine Instanz im Internet über HTTP anspricht, sendet HTTP-Anfragen. In dieser Arbeit wird ausschließlich mit dem Protokoll HTTP gearbeitet, deshalb wird im Nachfolgenden jede HTTP-Anfrage als "Anfrage" bezeichnet.
HTTP-Antwort	Zu jeder Anfrage wird eine HTTP-Antwort zurück an die anfragende Instanz gesendet. In dieser Arbeit wird ausschließlich mit dem Protokoll HTTP gearbeitet, deshalb wird im Nachfolgenden jede HTTP-Antwort als "Antwort" bezeichnet.
Hypertext Transport Protocol (HTTP)	*Hypertext Transfer Protocol* (HTTP) ([Ho20]) ist ein Protokoll und definiert, wie Nachrichten aufgebaut sind, die zwischen zwei Instanzen im Internet ausgetauscht werden (vgl. [Ho20]). Eine Instanz bezeichnet dabei einen Client oder einen Server (vgl. [Mo20]). Jede Nachricht ist durch einen Header und einen Body definiert (vgl. [Mo20e]). Ein Header gibt Meta-Informationen einer Nachricht wieder, während ein Body die eigentlichen Daten einer Nachricht enthält. Die Daten einer Nachricht sind in Form von Schlüssel-Wert-Paaren gesetzt. (vgl. [Io20]) Sogenannte Statuscodes Nachrichten über das Protokoll HTTP zeigen den Status einer Nachricht, zum Beispiel 200 für „OK" und 400/500 für „ein Fehler ist aufgetreten". (vgl. [Lu18]) Des Weiteren gibt es verschiedene Methoden, die über HTTP benutzt werden können. In dieser Arbeit sind drei wichtig: GET, POST

	und DELETE. *GET*: Erfragt Ressourcen hinter einer bestimmten URL (vgl. [Re01]). *POST*: Sendet Daten zu einer bestimmten URL (vgl. [Re01]). *DELETE*: Durch die DELETE-Methode werden Daten an einer angegebenen URL gelöscht (vgl. [Re01]). Zu beachten ist, dass HTTP zustandslos ist und keine Aktionen eines Benutzers speichert (vgl. [Mo20]).
Hypertext Transfer Protocol Secure (HTTPS)	*Hypertext Transfer Protocol Secure* (HTTPS) (vgl. [Lu18c]) erweitert HTTP und baut eine zusätzliche Schicht ein. Durch diese Schicht wird bei der Kommunikation zwischen Client und Server der Server authentifiziert, sodass die Echtheit des Servers bestätigt wird. Die Kommunikation erfolgt daraufhin verschlüsselt. (vgl. [Lu18c])
Identifikation	Eine Identifikation (kurz: *Id*) ist eine Zeichenfolge und ermöglicht eine eindeutige Zuordnung zwischen zwei Objekten. Beispiel: Ein Hund hat die Id „23“. Der Besitzer des Hundes besitzt einen Parameter „Hund-Id“, das den Wert „23“ trägt. Dadurch wird erkannt, dass der Hund mit der Id „23“ dem Besitzer gehört.
Identität	Als Identität wird die Identität eines Benutzers in einer Anwendung bezeichnet. Darunter sind personenbezogene Daten zusammengefasst, wie realer Name und Benutzername. Je nach Anwendung können weitere personenbezogene Daten erhoben werden. (vgl. [Ri17]: S. 246f)
Kommunikation im Hintergrund	Bei dieser Art der Kommunikation werden Anfragen im Hintergrund an einen Server gesendet, ohne dass ein Benutzer dafür eine Aktion tätigt (vgl. [Ri17]: S. 35f).
Kommunikation im Vordergrund	Eine Kommunikation im Vordergrund erfolgt, wenn ein Benutzer eine Aktion tätigt, wodurch eine Anfrage an einen Server gesendet wird (vgl. [Ri17]: S. 36 – 39).
Microservice	Ein Microservice ist ein in sich abgeschlossenes System, das von außen angesprochen wird. Dadurch arbeitet es unabhängig von an-

	deren Microservices. (vgl. [Ric20])
Persistieren	Persistieren meint eine globale und dauerhafte Speicherung von Daten, sodass eine Anwendung (beispielsweise ein Client oder ein Browser) auf sie zugreifen kann, nachdem sie geschlossen wurde.
Protokoll	Ein Protokoll definiert die Kommunikation zwischen zwei Instanzen. Im Kontext dieser Arbeit meint Instanz einen Client oder Server. Durch ein Protokoll ist spezifiziert, welche Parameter eine Nachricht einer Instanz enthält und in welchem Format die Parameter gesendet und empfangen werden. (vgl. [Ho19])
Server	Ein Server ist eine Instanz, die Anfragen bearbeitet und Antworten an anfragende Instanzen zurücksendet. Dabei kommuniziert ein Server über verschiedene Protokolle, beispielsweise HTTP, mit anfragenden Instanzen. Ein Server wird HTTP-Server genannt, wenn er über HTTP mit anderen Instanzen kommuniziert. Ein HTTP-Server arbeitet auf einer bestimmten Domäne und bearbeitet Anfragen über HTTP, die an die jeweilige Domäne gesendet werden. Zu jeder Anfrage generiert er eine Antwort und sendet diese über HTTP an die anfragende Instanz zurück. Eine anfragende Instanz kann ein weiterer HTTP-Server oder ein HTTP-Client sein. (vgl. [Fi19]) In dieser Arbeit wird ausschließlich ein HTTP-Server genutzt. Deshalb bezeichnet der Begriff „Server“ in den nachfolgenden Abschnitten ausnahmslos einen HTTP-Server.
SinglePageApplication	Bei einer SinglePageApplication beschränkt sich eine Web-Anwendung auf eine einzige Seite, bei der Inhalte dynamisch durch JavaScript verändert werden. Die Anwendung wird ausschließlich in einem Browser eines Benutzers ausgeführt und die Rechenlast liegt hauptsächlich beim Client. Dadurch wird die Kommunikation zwischen Client und Server geringgehalten, sodass weniger Daten versendet werden und eine Anwendung insgesamt flüssiger ist. (vgl. [Ye18])
Skalierung	Die Skalierung einer Anwendung meint, wie viele Benutzer gleichzeitig angemeldet sein können. Dies korrespondiert mit den verfügbaren Ressourcen (Prozessor und Arbeitsspeicher) eines Servers.

	(vgl. [Ge19])
Unified Resource Locator (URL)	*Unified Resource Locator* (URL) ([Ry20]) bestimmt eindeutig den Ort einer Ressource (vgl. [Ry20]). Eine URL ist dabei folgendermaßen aufgebaut: „Protokoll“://www.“Domäne“/“Pfad“?“Optionale Parameter“ - *Protokoll* ist das verwendete Protokoll, um die Datei abzurufen (dies ist zum Beispiel HTTP oder HTTPS). - Die *Domäne* ist wie ein Dateisystem, in dem eine Datei abgelegt ist. - Der *Pfad* gibt die Ordnerstruktur an, in dem sich eine Datei befindet. - *Optionale Parameter* sind Parameter, die angegeben werden, um die Suche nach einer Datei näher zu beschreiben. Diese werden *Query-Parameter* genannt. Ist eine angefragte Ressource nicht vorhanden, wird mit einem entsprechenden HTTP-Statuscode von 404, *Ressource nicht gefunden*, geantwortet.
Verteilte Architektur	In einer verteilten Architektur liegen die einzelnen Komponenten auf verschiedenen Servern und sind separate Ressourcen. Deshalb hat jede Komponente eine eigene URL und ist auf einer separaten Domäne zu finden. Dies ermöglicht eine flexible Strukturierung der einzelnen Komponenten. (vgl. [Go12]) Die Zielarchitektur von OpenSlides ist eine Verteilte Architektur. Grundlegend gibt es drei Komponenten in dieser Architektur (vgl. [Ri17]: S. 21 – 30): - Einen Client - Einen Server - Mehrere Ressourcen Der Server verwaltet und schützt die Ressourcen. Das heißt, er gibt Clients Zugang dazu und kann diesen wieder nehmen. (vgl. [Ri17]: S. 21 – 30)

1 Einleitung

In dieser Arbeit wird ein Dienst zur Authentifizierung von Benutzern (*Authentifizierungsdienst*) in der Software OpenSlides (vgl. [OS20]) erstellt. Dieser Authentifizierungsdienst soll auf Basis des Frameworks OAuth 2.0 realisiert werden.

Dabei werden folgende Merkmale untersucht, um den Authentifizierungsdienst zu realisieren:

- Welche Verfahren gibt es, um eine Authentifizierung zu realisieren?
- Wie werden die Verfahren umgesetzt?
- Was sind Vor- und Nachteile der untersuchten Verfahren?
- Wie kann das Framework OAuth 2.0 mit den untersuchten Verfahren zusammenarbeiten?
- Wann ist die Verwendung von OAuth 2.0 sinnvoll?
- Welche Vor- und Nachteile hat die Verwendung von OAuth 2.0?

Neben einer theoretischen Untersuchung des Frameworks OAuth 2.0 (vgl. [Ha12]) wird eine praktische Anwendung erarbeitet, in der das Framework umgesetzt ist. Die praktische Anwendung geschieht am Beispiel von OpenSlides.

OpenSlides ist eine Software, die dazu dient, Versammlungen digital zu verwalten. Benutzer der Software erstellen Anträge, die in Versammlungen diskutiert werden. Mithilfe einer Agenda ist ersichtlich, welche Themen bei einer Versammlung besprochen werden. Benutzer tragen sich auf einer sogenannten Redeliste ein, um zu einem Thema Meinungen und Sachverhalte beizutragen. Ein weiterer Bestandteil der Software sind Wahlen und Abstimmungen. Durch Wahlen werden Personen für Ämter gewählt. Mithilfe von Abstimmungen werden Anträge genehmigt oder abgelehnt. Ausschließlich Administratoren können neue Benutzer anlegen, bestehende ändern oder löschen. (vgl. [OS20])

Nur bereits angelegte Benutzer können sich bei der Software OpenSlides anmelden (vgl. [OS20]). Für die Software ist es erforderlich, dass Benutzer ihre Identität verifizieren. Dadurch wird sichergestellt, dass nur registrierte Benutzer Zugang zu den Inhalten der Software haben. In gleicher Weise sehen registrierte Benutzer ausschließlich Inhalte der Versammlungen, wie Anträge und Wahlen, zu denen sie eingeladen sind (vgl. [OS20]).

Aktuell ist Version 3.2 von OpenSlides veröffentlicht. In dieser Version ist die Software aus architektonischer Sicht in einer Server-Client-Architektur aufgebaut. Dabei ist der Client von OpenSlides als sogenannte SinglePageApplication mithilfe des Frameworks Angular (vgl. [Goo20]) umgesetzt. Zurzeit bestehen Arbeiten an Version 4.0. Ziel der Version 4.0 ist, die Architektur in eine dienstorientierte Architektur zu überführen (beispielsweise durch den Einsatz von Microservices). Komponenten des Servers werden dabei in separate Dienste unterteilt. Ziel der Architektur und Grund für die Umstrukturierung ist die Skalierbarkeit von O-

penSlides. Derzeit ist die Architektur für einige tausend Benutzer ausgelegt, Ziel ist es, dass mehrere hunderttausend gleichzeitig angemeldet sein können.

Das Autorisierungsframework OAuth 2.0 (vgl. [Ha12]) ist durch die „Internet Engineering Task Force“ (IETF) wie folgt definiert:

> „*The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.*“ ([Ha12])

Es ist ein Autorisierungsframework, das festlegt, wie Rechte eines Benutzers (im Kontext des Frameworks genannt „Ressourceneigentümer“ = „*resource owner*“) auf eine Anwendung (im Kontext des Frameworks genannt „Client“ = „*third-party application*“) übertragen werden. Dadurch kann der Client Daten des Ressourceneigentümers abrufen, die bei einem externen Dienst hinterlegt sind (im Kontext des Frameworks genannt „Ressourcenserver“ = „*HTTP service*“). (vgl. [Ri17]: S. 3 – 20)

[Ri17] führt als Beispiel einen Fotodienst und einen Druckerdienst an. Beides sind getrennt voneinander arbeitende Systeme. Ein Benutzer des Druckerdienstes kann seine Fotos, die er beim Fotodienst hinterlegt hat, drucken, wenn er den Druckerdienst dazu autorisiert auf seine Fotos beim Fotodienst zuzugreifen. (vgl. [Ri17]: S. 4 – 14)

Im grundlegenden Ablauf sind nach [Ri17] vier Instanzen an einem Ablauf nach OAuth 2.0 involviert (vgl. [Ri17]: 23):

- Ein *Ressourceneigentümer*, der eine Anwendung autorisiert
- Ein *Client* – die Anwendung, die autorisiert wird
- Ein *Autorisierungsserver*, der eine Autorisierung verwaltet
- Eine *Ressource*, auf die zugegriffen wird und die dem Ressourceneigentümer gehört

Ein *Autorisierungsserver* regelt die Autorisierung eines *Clients*. Rechte eines *Ressourceneigentümers* werden dem Client zugeordnet, sodass er auf *Ressourcen* des Ressourceneigentümers zugreifen kann. Ressourcen sind Daten, die bei einem Autorisierungsserver hinterlegt sind. Nur durch ein vom Autorisierungsserver ausgestelltes Token kann auf die Daten zugegriffen werden. (vgl. [Ri17]: S. 23) Zu beachten ist, dass das Autorisierungsframework OAuth 2.0 nicht zur Authentifizierung gedacht ist und eine Authentifizierung, das heißt Zugang zu einer Anwendung, an anderer Stelle erfolgen muss (vgl. [Ri17]: S. 236).

1.1 Einführung in die Thematik

Diese Arbeit beschreibt die Entwicklung eines Authentifizierungsdienstes für OpenSlides. Dabei kommuniziert der Authentifizierungsdienst mit dem Client von OpenSlides über

HTTP. HTTP speichert keine Aktionen eines Benutzers (vgl. [Mo20]). Deshalb ist es sinnvoll, die Aktionen eines Benutzers softwareseitig zu speichern. Mithilfe einer Benutzerverwaltung können Aktionen einem sogenannten Benutzerkonto aus der Benutzerverwaltung zugeordnet werden. Durch eine Authentifizierung können sich Benutzer anmelden, sodass ein Benutzer eindeutig einem Benutzerkonto aus der Benutzerverwaltung zugeordnet wird. Ermöglicht wird die Authentifizierung eines Benutzers durch einen Schlüssel, den nur der Benutzer selbst kennt. (vgl. [Ba17]: S. 1) Für Version 4.0 von OpenSlides ist eine verteilte Architektur angestrebt. Zurzeit ist die Authentifizierung bei OpenSlides sessionbasiert umgesetzt. Laut [Ok20] weise eine tokenbasierte Authentifizierung eine einfache Handhabung für verteilte Architekturen auf (vgl. [Ok20]). Deshalb wird eine tokenbasierte Authentifizierung mit einer sessionbasierten Authentifizierung verglichen, um Unterschiede herauszustellen. Der Authentifizierungsdienst von OpenSlides wird um das Framework OAuth 2.0 erweitert. Durch OAuth kann eine Anwendung autorisiert werden, um im Auftrag eines Benutzers zu handeln (vgl. [Ri17]: S. 3).

1.2 Ziel der Arbeit

Das Ziel der Arbeit ist es, einen Server zu entwickeln, der mithilfe einer Schnittstelle über HTTP erreichbar ist und als Dienst zur Authentifizierung von Benutzern in das Umfeld von OpenSlides integriert wird. Dabei soll das Framework OAuth 2.0 als Grundlage dienen.

Für das Erreichen des Ziels sind untergeordnete Fragen zu beantworten:

1. Eine Authentifizierung in OpenSlides zu der Version 3.2 erfolgte mittels Sessions. Für den neuen Authentifizierungsdienst sollen Tokens als Mittel zur Authentifizierung eingesetzt werden. Daraus ergeben sich zwei Fragen:
    a. Wie kann eine Authentifizierung mittels Tokens realisiert werden?
    b. Worin unterscheidet sich eine Authentifizierung mittels Tokens im Vergleich zu einer Authentifizierung mittels Sessions?
2. Wie kann das Framework OAuth 2.0 in dem zu entwickelnden Server realisiert werden?
3. Wie kommunizieren Client und Authentifizierungsserver auf Basis von OAuth 2.0?

Die Fragen werden sukzessive in dieser Arbeit aufgegriffen und beantwortet.

1.3 Aufbau der Arbeit

Der Hauptteil der Arbeit gliedert sich in drei Kapitel, um die im vorhergehenden Kapitel aufgestellten Ziele zu erreichen (siehe Kapitel 1.2). In Kapitel 2 wird eine Authentifizierung mittels Tokens mit einer Authentifizierung mittels Sessions verglichen. Dabei wird herausgestellt, welche Eigenschaften Tokens und welche Eigenschaften Sessions haben. Für den Ver-

gleich wird eine Eigenschaft genannt und aufgezeigt, wie die Eigenschaft bei Tokens und bei Sessions umgesetzt ist. Im Anschluss an den Vergleich wird ein kurzes Resümee gezogen. Dabei werden wichtige Punkte zusammengefasst und eine Empfehlung gegeben, welches der beiden Verfahren unabhängig von dieser Arbeit in Hinblick auf das Vorhaben von OpenSlides geeigneter ist. Im Anschluss an den Vergleich wird in Kapitel 3 das Framework OAuth 2.0 erläutert. Dabei werden Kernfunktionalitäten und Anforderungen beschrieben, die bei einer praktischen Umsetzung zu berücksichtigen sind. Kapitel 3 schließt mit weiterführenden Themen zu OAuth 2.0 ab. In Kapitel 4 wird die praktische Umsetzung des Authentifizierungsdienstes beschrieben, die in Kapitel 2 und 3 auf theoretischer Ebene beleuchtet wurde. Außerdem werden dabei Erkenntnisse und Probleme aufgezeigt, die während der praktischen Umsetzung aufgetreten sind. Zum Schluss werden das Vorhaben und die damit verbundene Umsetzung in einem Fazit bewertet. In diesem wird die ursprüngliche Fragestellung aufgegriffen und anschließend erläutert, ob das Framework OAuth 2.0 sinnvoll für das Vorhaben von OpenSlides ist und welche Punkte dabei zu berücksichtigen sind. Schlussendlich ist diese Arbeit als Leitfaden anzusehen, wenn ein Projekt auf Basis des Frameworks OAuth 2.0 entwickelt wird.

2 Beschreibung einer Authentifizierung mittels Tokens und Vergleich zu Sessions

In diesem Kapitel wird eine Umsetzung einer Authentifizierung mittels Tokens beschrieben. Dafür wird zunächst ein Vergleich zwischen Sessions und Tokens aufgestellt, um zu beschreiben, welche Vor- und Nachteile beide Verfahren haben. Anschließend werden verschiedene Implementierungsarten von Tokens betrachtet. Dazu werden sie beschrieben und es wird aufgezeigt, worin sie sich unterscheiden. Ziel ist es zu ermitteln, welches der genannten Verfahren und welche Implementierungsart (ob Session oder Token und welche Implementierungsart von Tokens) sich für den Authentifizierungsdienst bei OpenSlides besser eignet (siehe Kapitel 1).

2.1 Vergleich von Tokens und Sessions

Sowohl Tokens als auch Sessions sind Mittel, um Benutzer einer Internetseite zu authentifizieren. Für einen Benutzer gibt es keinen Unterschied zwischen den beiden Verfahren, da die Kommunikation mit einem Server im Hintergrund erfolgt (siehe Glossar: „Kommunikation im Hintergrund“) (vgl. [Hsu18]).

Im Nachfolgenden, Abschnitt 2.1.3 bis Abschnitt 2.1.8, wird ein detaillierter Vergleich zwischen Tokens und Sessions gezogen. Zuerst werden beide Verfahren im Detail beschrieben. Danach wird ein Aspekt genannt und es wird aufgezeigt, wie der Aspekt bei beiden Verfahren

umgesetzt ist. Als Repräsentanten für Tokens dienen JSON WEB Tokens (JWT), da JWT ein offizieller Standard und bei einer Authentifizierung mittels Tokens am weitverbreitetsten ist (vgl. [Ok20]).

2.1.1 Was sind JSON Web Tokens (JWT)

Hier erfolgt ein kurzer Überblick, was JWT sind, damit Unterschiede zwischen Sessions und Tokens im Vergleich nachvollziehbar sind.

JSON Web Token (JWT) (vgl. [Jo15d]) ist ein offener Standard, der nach RFC7519 definiert ist (vgl. [Au14]). Ein JWT liegt als JSON-Struktur vor. JWT sind in zwei Teilen aufgebaut: einem *Header* und einem *Payload*. Im Header sind Metainformationen festgehalten (vgl. [Jo15d]):

1. Die Art des vorliegenden Tokens (im Fall von JWT ist es „JWT“) (vgl. [Jo15d])
2. Der genutzte Signieralgorithmus (ein JWT kann alternativ verschlüsselt werden, es werden allerdings nur signierte JWT betrachtet) (vgl. [Jo15d])

Für den Header gibt es mehrere Spezifikationen, die unter anderem die Signierung definieren: „*the JSON Object Signing and Encryption standards, or JOSE*“ ([Ri17]: S. 189). Zu den Standards gehören: JSON Web Signatures (JWS) (vgl. [Jo15]), JSON Web Encryption (JWE) (vgl. [Jo15a]) und JSON Web Key (JWK) (vgl. [Jo15b]). (vgl. [Ri17]: S. 189)

Ist kein Algorithmus angegeben, bleibt ein JWT unsigniert (vgl. [Ri17]: S. 184).

Der Payload eines JWTs bezeichnet dessen Inhalt. In ihm stehen systemrelevante Informationen. Es gibt sieben offizielle Werte eines JWT (genannt „*claims*“ ([Ri17]: S. 185)), die im Payload festgelegt sein können. Darüber hinaus können weitere Daten im JSON-Format in ein JWT geschrieben werden, beispielsweise die Identität eines Benutzers. (vgl. [Ri17]: S. 184ff)

Wert	*Beschreibung*
iss	Der Aussteller eines Tokens
sub	Der Verwendungszweck eines Tokens
aud	Der Empfänger eines Tokens. Gibt an, für wen das Token ausgestellt wurde
exp	Ablaufzeitpunkt eines Tokens
nbf	Dieser Wert gibt an, ab welchem Zeitpunkt ein Token gültig ist
iat	Der Ausstellungszeitpunkt eines Tokens

jti	Die Identifikation eines Tokens

Tabelle 1: Offizielle Werte eines JWT
Quelle: In Anlehnung an [Ri17]: S. 185

JWT werden als Base64-kodierte Zeichenfolge übermittelt. Die Zeichenfolge ist in drei Teile geteilt und mittels Punkte getrennt. (vgl. [Ri17]: S. 190) Das nachfolgende Schaubild verdeutlicht dies.

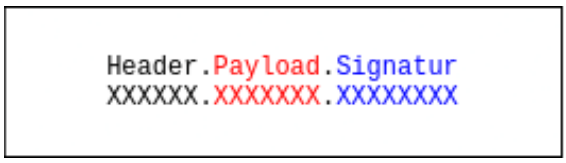

Abbildung 1: Struktur eines JWT
Quelle: In Anlehnung an [WP19]

Die Signatur wird über dem kodierten Header und dem kodierten Payload erstellt. Dazu wird ein Signaturschlüssel verwendet, der nur dem Server bekannt ist. (vgl. [Ri17]: S. 189f) Für einen Client ist der Payload eines JWT ersichtlich, da dieser nur Base64-kodiert ist und somit ohne Hilfsmittel dekodiert werden kann. Dadurch haben Clients Zugriff auf zum Beispiel benutzerbezogene Informationen, ohne dass sie den Signierschlüssel kennen müssen.

2.1.2 Beschreibung zweier Verfahren zur Authentifizierung

Zur Authentifizierung eines Benutzers gibt es verschiedene Verfahren. Wie im Glossar unter „HTTP“ beschrieben, ist HTTP zustandslos und speichert keine Interaktion mit einem Benutzer. Ein Benutzer müsste sich bei jeder Unterseite einer Anwendung erneut anmelden, um Zugang zu seinem Benutzerkonto zu erhalten. Deshalb ist ein Verfahren zur Authentifizierung nützlich. Dadurch wird die erfolgreiche Anmeldung eines Benutzers gespeichert, ein erneutes Anmelden innerhalb einer Sitzung entfällt. (vgl. [WP19])

Bei beiden Verfahren einer Authentifizierung (mittels Tokens sowie mittels Sessions) wird ein Objekt mit der Identität eines Benutzers erstellt. Bei einer tokenbasierten Authentifizierung heißt das Objekt „Token“. Bei einer sessionbasierten Authentifizierung heißt das Objekt „Session-Objekt“. Durch das Objekt erkennt ein Server die gültige Anmeldung eines Benutzers. Die beiden Verfahren unterscheiden sich darin, dass das Objekt entweder clientseitig oder serverseitig gespeichert wird. Bei einer tokenbasierten Authentifizierung wird das erstellte Objekt clientseitig gespeichert, bei einer sessionbasierten Authentifizierung serverseitig.

Für den Vergleich werden beide Verfahren im Detail beschrieben.

Authentifizierung mittels Tokens

Eine Authentifizierung mittels Tokens erfolgt folgendermaßen:

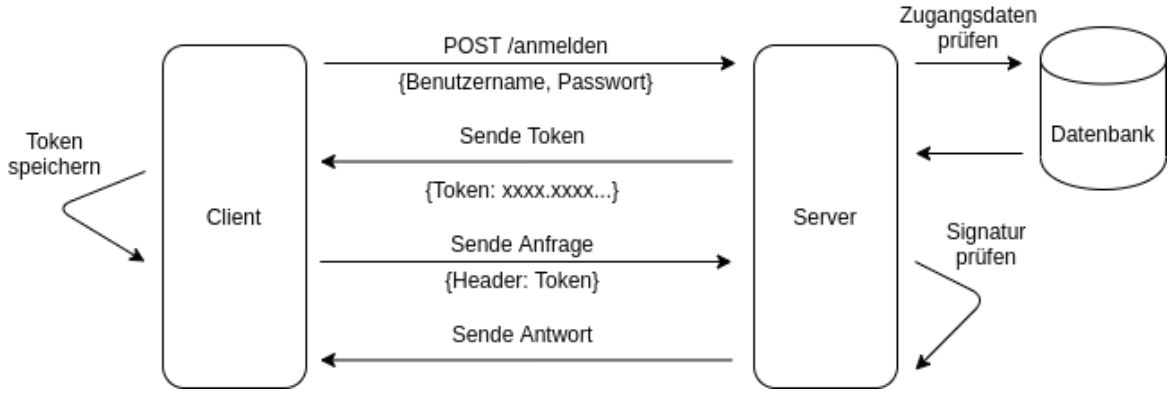

Abbildung 2: Tokenbasierte Authentifizierung

Quelle: In Anlehnung an [Hsu18]

Abbildung 2 zeigt, dass ein Client eine Anfrage an einen Server zum Authentifizieren schickt. Die Anfrage wird mit der POST-Methode versendet und enthält Zugangsdaten des Benutzers, der sich anmeldet. Der Server überprüft die Zugangsdaten, indem er sie mit den Daten in seiner Datenbank abgleicht. Bei einer Übereinstimmung erstellt er ein Token. Dieses sendet der Server mit dem HTTP-Statuscode 200 als Antwort an den Client zurück. (vgl. [Kuk16])

Das Token stellt einen Beweis für die erfolgreiche Authentifizierung des Benutzers dar. In ihm ist die Identität eines Benutzers enthalten sowie weitere Daten, wie *Aussteller, Gültigkeitszeitraum, Rechte, Verwendungszweck, Empfänger, Ausstellungszeitpunkt, Identifikation* (siehe Kapitel 2.1.1). Bei jeder weiteren Anfrage an den Server sendet der Client das erhaltene Token an den Server und fragt Ressourcen an. Das Token wird in den Header einer Anfrage gesetzt. Der Server prüft das Token und verifiziert mit seiner Richtigkeit, dass der Benutzer angemeldet ist und sendet dem Client die angefragten Ressourcen als Antwort. (vgl. [Hsu18])

Authentifizierung mittels Sessions

Im Gegensatz zu einer Authentifizierung mittels Tokens läuft eine Authentifizierung mittels Sessions folgendermaßen ab:

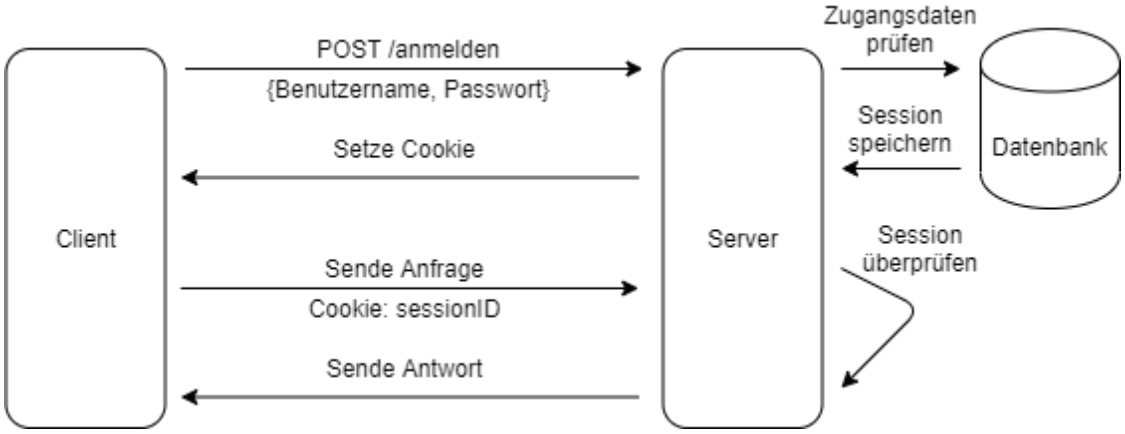

Abbildung 3: Sessionbasierte Authentifizierung

Quelle: In Anlehnung an [Hsu18]

Wie in Abbildung 3 zu sehen sendet ein Client bei einer sessionbasierten Authentifizierung ebenfalls eine Anfrage mit der POST-Methode an einen Server und übermittelt dabei Zugangsdaten eines Benutzers. Der Server überprüft die Zugangsdaten, indem er sie mit den Daten in seiner Datenbank abgleicht. Daraufhin erstellt der Server bei einer Übereinstimmung der Zugangsdaten ein sogenanntes Session-Objekt und speichert dieses Session-Objekt in seinem Arbeitsspeicher. Es enthält die Identität des Benutzers und stellt einen Beweis für seine erfolgreiche Authentifizierung dar. Außerdem beinhaltet das Session-Objekt eine Identifikation in Form einer Zeichenkette. Neben dem HTTP-Statuscode 200 setzt der Server bei dem anfragenden Client die Identifikation in einem Cookie. Dieses wird *Session-Cookie* genannt; die Identifikation in einem Session-Cookie heißt *Session-Id*. Bei jeder weiteren Anfrage an den Server wird automatisch das Session-Cookie gesendet. Der Server liest aus dem Session-Cookie die Identifikation und sucht nach einem Session-Objekt, das die gleiche Identifikation aufweist. Findet der Server ein passendes Session-Objekt, erkennt der Server, dass der Benutzer angemeldet ist und sendet eine Antwort zum Client. (vgl. [Hsu18])

2.1.3 Speicherplatzbedarf von Tokens und Cookies

An dieser Stelle wird überprüft, wie viel Speicherplatz einerseits Tokens andererseits Session-Cookies erfordern. In Hinblick auf Anfragen, die ein Client an einen Server sendet, ist dies wichtig, da bei jeder Anfrage an einen Server ein Token oder ein Session-Cookie gesendet wird (siehe Kapitel 2.1.2). Je weniger Speicherplatz benötigt wird, desto schneller wird das Objekt (Token oder Session-Cookie) gesendet.

Bei einer tokenbasierten Authentifizierung versendet ein Client bei einer Anfrage ein Token. In dem Token sind Informationen enthalten, zum Beispiel der Name sowie Rechte eines Benutzers (siehe Kapitel 2.1.2.1). Bei einer sessionbasierten Authentifizierung versendet ein Client ein Session-Cookie, das lediglich eine Zeichenkette enthält (siehe Kapitel 2.1.2.2).

Degges hat hierfür einen Versuch durchgeführt: Die Zeichenkette „*abc123*“ ([De17]) wurde sowohl in einem Cookie als auch in einem Token gespeichert. Laut Degges benötigte das Cookie einen Speicherplatz von 6 Byte, während das Token einen Speicherplatz von 304 Byte benötigte. (vgl. [De17]) Dies zeigt, dass Tokens mehr als fünfzigmal so groß wie Session-Cookies bei gleichem Inhalt sind.

Deshalb ist der Aspekt „Speicherplatzbedarf“ bei einer sessionbasierte Authentifizierung positiv, bei einer tokenbasierten Authentifizierung negativ zu bewerten.

2.1.4 Sicherheit von Tokens und Sessions

Tokens sind signiert (siehe Kapitel 2.1.1). Durch die Signatur eines Tokens ist sichergestellt, dass es unverfälscht ist. Die Signatur wird unter anderem aus dem Header und dem Payload des Tokens generiert. Ein veränderter Header oder veränderter Payload ergibt eine andere Signatur und die Signatur kann nur durch das Geheimnis, das ausschließlich dem Server bekannt ist, erstellt werden.

Session-Objekte selbst liegen bei einem Server. Ein Session-Cookie stellt die gültige Anmeldung eines Benutzers dar. Bei einer Anfrage an einen Server wird das Session-Cookie eines Clients automatisch an den Server gesendet. Ein Benutzer ist angemeldet, wenn ein Session-Objekt mit derselben Identifikation wie im Session-Cookie gefunden wurde. (siehe Kapitel 2.1.2.2)

Session-Cookies können die Eigenschaften „HttpOnly" und „SameSite“ haben. Mit der Eigenschaft „HttpOnly" sind Cookies vor XSS-Attacken geschützt, da sie nicht mehr clientseitig durch JavaScript abgerufen werden können (vgl. [Mo20f]). Mit der Eigenschaft „SameSite“ sind Cookies vor CSRF-Attacken geschützt, da Cookies mit der Eigenschat „SameSite“ nur von derselben Domäne versendet werden (vgl. [Owa18]). Des Weiteren wird durch die Eigenschaft „Secure“ sichergestellt, dass ein Cookie ausschließlich über HTTPS gesendet wird (vgl. [Mo20f]).

Bei beiden Verfahren, Tokens und Sessions, bestehen Möglichkeiten die Verfahren sicher einzusetzen. Hier sind beide Verfahren positiv zu bewerten.

2.1.5 Performanz eines Clients und eines Servers und Skalierbarkeit einer Anwendung

Bei einer tokenbasierten Authentifizierung speichert ein Client ein Token. Damit bleibt die Auslastung eines Servers (in Hinblick auf Prozessor und Arbeitsspeicher) gering, da dieser nur Tokens ausstellt und sie an Clients sendet. Ein Server muss jedoch das Token verifizieren, indem er dessen Signatur prüft. (vgl. [WP19]) Dies ist ein rechenintensives Verfahren, benötigt aber dennoch nur zwischen 0,5 und 4,1 Millisekunden (ms)[1]. Zudem bleibt die Dauer mit zunehmender Anzahl an gleichzeitig angemeldeten Benutzern konstant. Die Auslastung eines Clients bleibt ebenfalls gering, da er unabhängig von der Anzahl an gleichzeitig angemeldeten Benutzern nur ein einziges Token erhält (vgl. [WP19]).

In Hinblick auf Prozessor und Arbeitsspeicher bleibt die Auslastung eines Clients gering, bei einem Server steigt sie allerdings, wenn eine sessionbasierte Authentifizierung genutzt wird. Dies liegt darin begründet, dass ein Server ein Session-Objekt je angemeldetem Benutzer speichert (und im Arbeitsspeicher ablegt) und nur ein Session-Cookie bei einem Client setzt. (siehe Kapitel 2.1.2).

Für eine sessionbasierte Anwendung gibt es eine Obergrenze an gleichzeitig angemeldeten Benutzern, da für jeden Benutzer ein Teil des Arbeitsspeichers eines Servers benötigt wird. (vgl. [WP19])

Bei einer tokenbasierten Authentifizierung ist die Anzahl nicht durch Ressourcen begrenzt. Dadurch skaliert eine Anwendung mit einer tokenbasierten Authentifizierung besser als eine sessionbasierte Authentifizierung und es sind weniger Ressourcen erforderlich, damit gleich viele Benutzer angemeldet sein können. (vgl. [WP19])

Dieser Punkt, Performanz und Skalierbarkeit, ist für Tokens positiv zu bewerten. Das Prüfen der Signatur eines Tokens ist zwar eine rechenintensive, jedoch nur eine kurzweilige Aufgabe für einen Server. Eine Signatur ändert sich nicht und ist deshalb von gleichbleibender Schwierigkeit. Außerdem sind die vorhandenen Ressourcen eines Servers ausreichend, damit eine beliebige Anzahl an Benutzern gleichzeitig angemeldet sein kann. Für eine sessionbasierte Authentifizierung ist dieser Punkt negativ zu bewerten. Mit zunehmender Anzahl an gleichzeitig angemeldeten Benutzern steigt die Auslastung eines Servers, sodass es eine Obergrenze

[1] Ein Versuchsaufbau wurde mit einem Laptop durchgeführt, der mit vier Kernen (jeder Kern hat eine Taktung von 2 – 3,6 Gigahertz und arbeitet mit zwei Threads), 16 Gigabyte Arbeitsspeicher und einer 256 Gigabyte M.2 SSD ausgestattet ist. Dabei wurde ein RSA-Schlüssel mit einer Schlüssellänge von 2048 Bit genutzt. Folgende sechs Messwerte wurden festgestellt: 4,105 ms, 0,507 ms, 1,027 ms, 0,617 ms, 0,659 ms, 1,074 ms.

gibt und die Ressourcen des Servers erweitert werden müssen, damit sich mehr Benutzer als diese Obergrenze anmelden können.

2.1.6 Gleichzeitige Nutzung mehrerer Verbindungen

In diesem Abschnitt wird untersucht, wie realisiert wird, dass sich Tabs untereinander synchronisieren. Ein Beispiel verdeutlicht, in welchen Szenarien eine Synchronisation zwischen einzelnen Tabs sinnvoll ist:

Bei einem Onlineshop können Waren in den Einkaufswagen gelegt werden. Es kommt vor, dass man Waren miteinander vergleicht, indem man jede einzelne Ware in einem separaten Tab öffnet. Dabei werden sie in den unterschiedlichen Tabs in den Einkaufswagen abgelegt. Am Ende sollen alle Waren im gleichen Einkaufswagen sein, wenn man sie bezahlt. Dies ist der Fall, wenn Tabs untereinander synchronisiert werden.

Bei einer Authentifizierung mittels Tokens benötigt ein Client Zugriff auf einen globalen Speicher eines Browsers, beispielsweise den Lokalen Speicher oder die IndexedDB. Beide speichern Daten in Form von Schlüssel-Wert-Paaren. Die Daten sind global in einem Browser sichtbar, bis sie explizit gelöscht werden. (vgl. [me19]) In Kapitel 2.3.1 wird näher auf verschiedene Speicher eines Browsers eingegangen. Durch den Lokalen Speicher oder die IndexedDB eines Browsers ist eine Synchronisation zwischen verschiedenen Tabs eines Browsers möglich: In einem Tab wird eine Aktion eines Benutzers ausgeführt (Aktion meint als Beispiel „eine Ware in den Einkaufswagen legen“). Die Aktion aktiviert ein Ereignis, wodurch in anderen Tabs erfasst wird, dass die Aktion ausgeführt wurde. (vgl. [Go19])

Wird eine sessionbasierte Authentifizierung verwendet, speichert ein Client nichts in dem globalen Speicher eines Browsers (siehe Kapitel 2.1.2.2). Bei jeder Aktion, die ein Benutzer ausführt, wird eine Anfrage an einen Server gesendet. In der Folge passt der Server das dem Benutzer zugehörige Session-Objekt an. Daraufhin gleichen andere Tabs desselben Browsers die Daten mit dem Session-Objekt beim Server ab. (vgl. [Au18a])

Letztendlich können beide Verfahren ohne Hilfsmittel mehrere Tabs untereinander synchronisieren.

2.1.7 Gleichzeitige Anmeldung in mehreren Tabs

In diesem Abschnitt wird untersucht, wie sichergestellt wird, dass ein Benutzer sich nur einmal anmeldet und angemeldet bleibt, bis er sich aktiv abmeldet. Das heißt, ein Benutzer bleibt angemeldet, wenn er die gleiche Anwendung in einem neuen Tab öffnet oder die Internetseite neu lädt. Je seltener ein Benutzer seine Zugangsdaten eingibt, desto seltener kann ein Angreifer sie erfassen (vgl. [Bih15]: S. 151).

Bei der Nutzung von Tokens kann die Identität eines Benutzers ohne Hilfsmittel nicht gespeichert werden. Dies ist darin begründet, dass das Token clientseitig gespeichert wird und nicht serverseitig. Zudem hält ein Server nicht fest, welcher Benutzer sich bereits angemeldet hat. Dadurch erkennt ein Server die Identität eines Benutzers und die damit verbundene gültige Anmeldung nur, wenn das ausgestellte Token in dem Header einer Anfrage liegt. Dadurch wird ein Benutzer in einem anderen Tab ohne Hilfsmittel nicht angemeldet sein, sondern muss sich in jedem neuen Tab separat anmelden. (vgl. [Go19])

Aufgrund des Session-Cookies kann ein Client bei einer sessionbasierten Authentifizierung in jedem Tab die Identität eines Benutzers abfragen und damit feststellen, ob er angemeldet ist: Das Session-Cookie ist browserweit gültig. Wird der Client in einem neuen Tab aufgerufen, sendet er zunächst eine Anfrage an einen Server, um die Identität des Benutzers abzufragen. Der Server erkennt das Session-Cookie in der Anfrage und sendet eine Bestätigung mit der Identität des Benutzers zurück. (vgl. [Go19])

2.1.8 Modularität der Architektur einer Anwendung

Dieser Abschnitt untersucht, ob eine tokenbasierte oder sessionbasierte Authentifizierung die Architektur einer Anwendung einschränkt.

Ein Kriterium dafür ist die Domäne einer Anwendung. Komponenten einer verteilten Architektur können auf verschiedenen Domänen liegen (siehe Kapitel 1.1). Für den Vergleich werden die Eigenschaften von Session-Cookies herangezogen, da diese zur Kommunikation mit einem Server verwendet werden (siehe Kapitel 2.1.2).

Tokens sind in ihren Werten variabel. Es (im Fall von JWT) kann ein Zeitpunkt festgesetzt werden, an dem das Token ungültig wird (siehe Kapitel 2.1.1). Für Tokens gibt es jedoch keine Restriktionen, die die Nutzung einschränken. Für einen Server ist es gleichwertig, aus welcher Domäne ein Token gesendet wird. (vgl. [WP19])

Im Gegensatz dazu müssen Session-Cookies mit den Eigenschaften „HttpOnly“ und „SameSite“ gesetzt sein, um gegenüber XSS- und CSRF-Angriffen geschützt zu sein (siehe Kapitel 2.1.4). Allerdings können Session-Cookies mit der Eigenschaft „SameSite“ ausschließlich von derselben Domäne agieren, in der sie gesetzt werden (vgl. [Ch20]). Für eine Architektur bedeutet dies: Die Komponenten einer Anwendung müssen sich in einer Domäne befinden. Andernfalls können die Komponenten nicht untereinander kommunizieren. (vgl. [Ch20]) Bei einer verteilten Architektur liegen die einzelnen Komponenten jedoch auf verschiedenen Domänen.

2.1.9 Zusammenfassung des Vergleichs von Tokens und Sessions

Aus den einzelnen Vergleichsaspekten zwischen Tokens und Sessions geht hervor, dass ein Token eine größere Datenmenge als eine Session ist. Jedoch sind Tokens flexibler einsetzbar und bieten eine höhere Modularität einer Anwendung im Vergleich zu Sessions.

Außerdem bedeutet der Einsatz von Sessions eine enge Verknüpfung von Server und Client, da durch Session-Cookies Anfragen und Antworten auf derselben Domäne gesendet werden müssen, um gegen CSRF-Attacken und XSS-Attacken geschützt zu sein.

Aus den genannten Punkten folgt, dass der Einsatz von Tokens in Form von JWT für OpenSlides geeigneter ist.

2.2 Vergleich verschiedener Tokens

In diesem Abschnitt werden zwei weitere Implementierungsarten von Tokens beschrieben und mit JWTs verglichen. Am Ende wird eine Implementierungsart von Tokens ausgewählt, die als Basis für die Entwicklung des Authentifizierungsdienstes bei OpenSlides genutzt wird.

2.2.1 Beschreibung von Branca Tokens

Branca Tokens sind eine Umschreibung für die verschlüsselte Form einer Nachricht, die nach „*IETF XChaCha20-Poly1305 AEAD*“ ([Tu17]) konstruiert wird, so Tuupola (vgl. [Tu17]). Deshalb wird in einem Branca Token kein Header angegeben, sondern nur ein Payload. Der Payload eines Branca Tokens kann ein JSON-Objekt, eine Zeichenfolge oder ein Zahlenformat sein. (vgl. [Tu20b]) Branca Tokens sind nur durch eine inoffizielle Dokumentation (vgl. [Tu20]) beschrieben, die von Mika Tuupola verfasst wurde.

Im Vergleich zu JWT sollen Branca Tokens kleiner sein, wie Tuupola beschreibt. Zum einen ist ein Branca Token mit Base62 kodiert (statt einer Base64-Kodierung wie bei JWT), zum anderen wird bei Branca Tokens nur der Payload kodiert, dadurch entfallen Zeichen, die bei JWTs vorhanden sind. (vgl. [Tu17])

Als Nachweis wird folgender Payload benutzt ([Tu17]):

```
"iss" : "joe",
"exp" : 1300819380,
"http://example.com/is_root" : true
```

Ein JWT mit dem oben genannten Payload umfasst als kodierte Zeichenfolge 168 Zeichen. Ein Branca Token mit gleichem Payload umfasst als kodierte Zeichenfolge lediglich 148 Zeichen. (vgl. [Tu17]) Dies ist ein Unterschied von 20 Zeichen. Der genannten Dokumentation von Branca Tokens zufolge ist die Länge der Zeichenfolge eines Branca Tokens weiterhin

reduzierbar, wenn der Ablaufzeitpunkt (im Payload dargestellt durch „exp“) entfernt wird und zusätzlich ein binärer Serialisierer eingesetzt wird (vgl. [Tu17]). Ein Serialisierer ist dabei ein Hilfsmittel, das ein Objekt (beispielsweise ein JSON-Objekt) systematisch in eine Zahlenfolge kodiert (vgl. [Sch20]). Sind die beiden Punkte berücksichtigt, reduziert sich die Zeichenfolge auf 115 Zeichen (vgl. [Tu17]). Dadurch ist ein Branca Token als kodierte Zeichenfolge um 53 Zeichen kleiner als ein JWT. Das ist ein Unterschied von ungefähr 32%. Ein geringerer Speicherbedarf bedeutet eine schnellere Übertragung, wie bereits in Kapitel 2.1.3 diskutiert (siehe Kapitel 2.1.3).

Ein Nachteil von Branca Tokens ist, dass sie nicht von selbst ihre Gültigkeit verlieren, sondern ein Server oder ein Client dies handhaben muss (vgl. [Tu20]). Dies erfordert eine zusätzliche Überprüfung auf die Gültigkeit eines Branca Tokens (vgl. [Tu17]). Eine zusätzliche Überprüfung bedeutet, dass eine weitere Funktion implementiert und gewartet werden muss.

2.2.2 Beschreibung von Platform-Agnostic-Security-Tokens (PASETO)

Zu Platform-Agnostic Security-Tokens (PASETO) gibt es eine Dokumentation (vgl. [Ar18]), die PASETO wie folgt beschreibt:

> „*Platform-Agnostic SEcurity TOkens (PASETOs) provide a cryptographically secure, compact, and URL-safe representation of claims that may be transferred between two parties. The claims are encoded in JavaScript Object Notation (JSON), version-tagged, and either encrypted using shared-key cryptography or signed using public-key cryptography.*“ ([Ar18])

PASETO sind Tokens, deren Payload kryptografisch gesichert sind (vgl. [Ar18]). Dabei sind PASETO in drei Bereiche unterteilt: Version, Verwendungszweck und Payload. (vgl. [Ar18])

Die *Version* stellt die genutzte Version von PASETO dar. Der *Verwendungszweck* gibt an, ob es sich um ein verschlüsseltes oder ein signiertes Token handelt. Er kann zwei Werte annehmen: „*public, local*“ ([Ar18]). Der Wert „public“ bedeutet, dass der Payload eines PASETO nicht verschlüsselt, aber signiert wird; „local“ gibt an, dass der Payload eines PASETO verschlüsselt wird und nicht signiert. (vgl. [Ar18])

PASETO sind durch einen noch nicht veröffentlichten Standard der Internet Engineering Task Force (IETF) definiert. Dabei betont die IETF, dass es sich bei PASETO um einen Entwurf handelt, der noch in Arbeit ist (vgl. [Ar18]).

2.2.3 Fazit für die Nutzung von Tokens

An den inoffiziellen Dokumentationen von PASETO und Branca Tokens ist zu erkennen, dass die Entwicklung der beiden Implementierungsarten nicht von einer offiziellen Gruppe unterstützt wird. Sie sind lediglich durch inoffizielle Bibliotheken nutzbar, aber nicht durch einen offenen Standard beschrieben. Bei inoffiziellen Entwicklungen besteht die Gefahr, dass

sie stagnieren, Sicherheitsrisiken darstellen oder nur kommerziell verfügbar sind. Deshalb wird in dieser Arbeit Abstand von sowohl PASETO Tokens als auch von Branca Tokens genommen und ausschließlich JWT für die Implementierung des Authentifizierungsdienstes bei OpenSlides genutzt.

2.3 Verwendung von Tokens zur Authentifizierung

Im vorherigen Abschnitt wurde ein Vergleich verschiedener Implementierungsarten von Tokens aufgestellt und eine Implementierungsart für die weitere Nutzung festgelegt. In diesem Kapitel wird untersucht, welche Bedingungen bei der Verwendung von Tokens zu beachten sind. Eine Web-Anwendung muss sicher vor XSS- und CSRF-Attacken sein, deshalb werden die beiden im Glossar beschriebenen Angriffsverfahren berücksichtigt.

2.3.1 Persistieren eines Tokens

In Bezug auf CSRF- und XSS-Angriffen ist es wichtig, das Persistieren eines Tokens zu untersuchen. Es gibt fünf verschiedene Möglichkeiten, wie ein Client ein Token persistieren kann: In dem Lokalen Speicher, in dem Sitzungsspeicher, in der IndexedDB eines Browsers, in dem Arbeitsspeicher eines Tabs oder in einem Cookie.

Die nachfolgende Tabelle gibt dabei einen Überblick, welche Möglichkeit zum Persistieren vor welchen Angriffen schützt.

	Geschützt vor XSS-Attacken	Geschützt vor CSRF-Attacken
Cookies	Ja[2]	Ja[3]
Lokaler Speicher	Nein	Ja
Sitzungsspeicher	Nein	Ja
IndexedDB	Nein	Ja
Arbeitsspeicher	Ja[4]	Ja

Tabelle 2: Schutz von Persistierungsmöglichkeiten vor XSS- und CSRF-Attacken
Quelle: In Anlehnung an [St19]

[2] Ein Schutz vor XSS-Attacken ist dann gegeben, wenn die Eigenschaft „HttpOnly“ gesetzt ist.

[3] Ein Schutz vor CSRF-Attacken ist dann gegeben, wenn die Eigenschaft „SameSite“ gesetzt ist.

[4] Ein Schutz vor XSS-Attacken hängt davon ab, ob eine Maßnahme gegen XSS-Attacken clientseitig besteht, beispielsweise durch ein Framework.

Aus Tabelle 2 ist zu erkennen, dass lediglich Cookies und Arbeitsspeicher gegen beide Angriffe geschützt sind. Bei einem Arbeitsspeicher hängt dies allerdings von dem Framework ab, mit dem der Client entwickelt wurde. (vgl. [St19]) Bei OpenSlides wird der Client mit dem Framework *Angular* entwickelt (siehe Kapitel 1). Durch Angulars Mechaniken werden Eingaben und Manipulationen, beispielsweise das Einspielen von JavaScript, außerhalb einer Anwendung standardmäßig als einfache Zeichenfolge eingelesen, sodass JavaScript nicht ausgeführt wird (vgl. [Goo19a]). Dadurch ist der Client von OpenSlides vor XSS-Attacken geschützt. Cookies sind nur vor beiden Angriffen sicher, wenn sowohl die Eigenschaft „HttpOnly“ (vgl. [Mo20f]) als auch „SameSite“ (vgl. [Ch20]) gesetzt ist. Die Verwendung von Cookies zum Persistieren von Tokens ist aus zwei Gründen jedoch ungeeignet:

- Die Eigenschaft „SameSite“ verhindert, dass Komponenten auf unterschiedlichen Domänen arbeiten.
- Durch die Eigenschaft „HttpOnly“ können Daten im Cookie clientseitig nicht ausgelesen werden. Dies ist aber notwendig, damit ein Client die Identität eines Benutzers kennt.

Daraus resultiert, dass sich der Arbeitsspeicher eines Tabs am besten dazu eignet ein Token zu speichern. Für die Entwicklung des Authentifizierungsdienstes von OpenSlides folgt, dass ein Token nur temporär im Arbeitsspeicher eines Tabs gespeichert und nicht dauerhaft persistiert wird. (vgl. [Au18])

Eine detaillierte Ausführung der Argumente für oder gegen jede Möglichkeit kann im Anhang A nachgelesen werden.

2.3.2 Dauer der Gültigkeit eines Tokens

Im Gegensatz zur sessionbasierten Authentifizierung gibt es in Zusammenhang mit der tokenbasierten Authentifizierung keine Routine, um Nutzer abzumelden. Ein Server hat keine Informationen über gültige und ungültige Tokens, sondern verifiziert nur die Signatur eines Tokens, um seine Richtigkeit zu überprüfen. Ein Token, mit dem ein Benutzer angemeldet ist, wird clientseitig gelöscht, sobald er sich abmeldet. Deshalb ist es wichtig, dass ein Server gültige von ungültigen Tokens unterscheiden kann.

Eine Möglichkeit liegt in dem Ablaufzeitpunkt eines JWT, um ein Token als ungültig zu kennzeichnen. Ein Token verliert seine Gültigkeit, sobald dieser Zeitpunkt erreicht ist. Es ist wichtig, dass ein Token nur wenige Minuten lang gültig ist. (vgl. [Jo15d]) „*Common practice is to keep it around 15 minutes*“, so Gopal ([Go19]). Ein Angreifer kann nur in den wenigen verbliebenen Minuten mit einem Token handeln, nachdem er eines ergreift (vgl. [Go19]). Zum Beispiel: *Ein Token wurde vor vier Minuten ausgestellt, dann gelangt das Token in den*

Besitz eines Angreifers. Dieser hat noch elf Minuten Zeit, in denen das Token gültig ist und er Zugang zu einer Anwendung hat.

Eine weitere Methode, um ungültige Tokens zu erkennen, besteht darin, eine Liste mit ungültigen Tokens zu führen. Darin aufgelistet sind alle Tokens, die bereits genutzt und ungültig wurden. Ein Server gleicht gesendete Tokens mit denen in der Liste, sodass darin aufgeführte Tokens nicht mehr verwendet werden können. Ein Token wird ungültig, weil ein Benutzer sich abmeldet oder weil erkannt wird, dass ein Angreifer ein Token besitzt. Die Liste mit allen ungültigen Tokens wird in einer Datenbank abgespeichert, um dies beim Server zu erkennen. (vgl. [Go19])

2.3.3 Erneuern eines Tokens

Für die Entwicklung des Authentifizierungsdienstes für OpenSlides ist ein weiterer zu berücksichtigender Aspekt das Erneuern eines Tokens. Dabei wird ein altes Token gegen ein neues, gültiges Token ersetzt. Bei einer erfolgreichen Anmeldung eines Clients erhält er nicht nur das Zugangstoken, sondern auch ein sogenanntes Aktualisierungstoken (vgl. [Au16]). Ein Aktualisierungstoken ist ebenfalls ein Token (JWT), wird zum Erneuern eines Zugangstokens ausgestellt und enthält ausschließlich eine Zeichenfolge: Die Identifikation des aktiven Zugangstokens. Die Identifikation ist zusätzlich in einer Datenbank des Servers gespeichert. Jedes Mal, wenn das aktuelle Zugangstoken abläuft, schickt der Client eine Anfrage mit dem Aktualisierungstoken. Daraufhin überprüft der Server das Aktualisierungstoken und die darin enthaltene Identifikation und gleicht sie mit seiner Datenbank ab. Ist eine solche Identifikation in der Datenbank vorhanden, sendet der Server als Antwort den HTTP-Statuscode 200 sowie ein neues Zugangstoken. (vgl. [Go19])

Die Nutzung von Aktualisierungstokens bietet drei Vorteile:

1. Höhere Sicherheit im Vergleich zu einer Anwendung ohne Aktualisierungstoken (vgl. [Bih15]: 151).
2. Ein Aktualisierungstoken erhöht die Nutzerfreundlichkeit (vgl. [Bih15]: 151).
3. Das Eingeben von Zugangsdaten ist nur einmal erforderlich (vgl. [Go19]).

Aus dem vorangegangenen Kapitel geht hervor, dass ein Zugangstoken eine kurze Gültigkeitsdauer hat (siehe Kapitel 2.3.2). Ist kein Aktualisierungstoken vorhanden, muss ein Nutzer sich jedes Mal erneut anmelden, wenn ein Zugangstoken seine Gültigkeit verliert (vgl. [Go19]). Deshalb wird laut Bihis durch die Verwendung eines Aktualisierungstokens neben der Nutzererfahrung auch die Sicherheit der Anwendung erhöht:

„*Making effective use of refresh tokens can minimize the opportunities attackers have to steal user credentials, as well as provide a better user experience by extending their sessions with your client without intervention.*“ ([Bih15]: S. 151)

Eine Aktualisierung im Hintergrund führt dazu, dass ein Benutzer seltener seine Anmeldedaten eingeben muss. Dadurch ist die Gefahr geringer, dass ein Angreifer Zugangsdaten eines Benutzers erhält, weil die Zugangsdaten seltener über HTTP versendet werden. (vgl. [Bih15]: S. 151)

Ein Aktualisierungstoken kann ähnlich wie ein Zugangstoken auf fünf verschiedene Arten persistiert werden (siehe Kapitel 2.3.1): im Lokalen Speicher, im Sitzungsspeicher, in der IndexedDB eines Browsers, in einem Cookie oder im Arbeitsspeicher eines Tabs. Der Lokale Speicher, der Sitzungsspeicher oder die IndexedDB können für OpenSlides nicht verwendet werden, da sie gegenüber XSS-Attacken anfällig sind. Das Speichern im Arbeitsspeicher eines Tabs ist redundant, da dann für ein Aktualisierungstoken die gleichen Bedingungen wie für ein Zugangstoken gelten. Deshalb wird ein Aktualisierungstoken in einem Cookie (mit der Eigenschaft „HttpOnly“) gespeichert.

Im Gegensatz zu einem Zugangstoken hat ein Aktualisierungstoken eine längere Gültigkeit von mehreren Stunden und wird persistiert, da ein Angreifer mit einem Aktualisierungstoken keine geschützten Ressourcen abfragen kann (vgl. [Go19]). Wie bereits festgestellt wird ein Zugangstoken nicht in einem Cookie mit der Eigenschaft „HttpOnly“ gespeichert, da dieses anfällig für CSRF-Attacken ist (siehe Kapitel 2.3.1). Außerdem muss ein Client den Payload eines Zugangstokens auslesen, um nötige Informationen für den Benutzer anzuzeigen. Für den Zugang zu geschützten Inhalten sind sowohl Zugangstoken als auch Aktualisierungstoken nötig. Zudem kann ein Benutzer in jedem Tab angemeldet sein. Ähnlich wie bei einer sessionbasierten Authentifizierung wird beim erstmaligen Aufruf einer Anwendung eine Anfrage an einen Server gesendet, um die Identität abzurufen. Ist ein Aktualisierungstoken vorhanden, erkennt der Server das und sendet ein Zugangstoken zurück. Das Zugangstoken wird wiederum in dem Arbeitsspeicher des Tabs abgelegt. (vgl. [Go19])

Durch ein Aktualisierungstoken können neue Zugangstoken von einem Server erhalten werden. Damit dies nicht unbegrenzt möglich ist, gibt es drei Möglichkeiten ein Aktualisierungstoken als ungültig zu deklarieren (vgl. [Au16]):

1. Ein Aktualisierungstoken besitzt wie ein Zugangstoken einen Ablaufzeitpunkt. Das Aktualisierungstoken ist ungültig, sobald der Zeitpunkt überschritten ist. (vgl. [Au16])
2. Ungültige Aktualisierungstokens werden durch einen Server in einer Liste gespeichert (vgl. [Au16]).
3. Das Anmeldeverfahren wurde geändert. Dies ist beispielsweise der Fall, wenn von einer Benutzername-Passwort-Kombination auf eine Zwei-Faktor-Authentifizierung umgestellt wurde. (vgl. [Au16])

2.4 Vorgehensweise zur Authentifizierung mittels Tokens

In den vorangegangenen Kapiteln wurden verschiedene Aspekte betrachtet, um eine Authentifizierung mittels Tokens zu realisieren. In diesem Kapitel, Kapitel 2.4, werden die Aspekte zusammengefasst und wesentliche Punkte aufgezeigt.

1. Einem Client wird Zugang zu geschützten Inhalten mittels Zugangstokens gewährt. Ein Zugangstoken wird von einem Server nach erfolgreicher Authentifizierung ausgestellt und beim Client im Arbeitsspeicher abgelegt. Zusätzlich zu einem Zugangstoken wird ein Aktualisierungstoken ausgestellt. Das Aktualisierungstoken wird in einem Cookie mit der Eigenschaft „HttpOnly“ abgelegt.
2. Das Zugangstoken ist nur wenige Minuten (15 Minuten) lang gültig. Ein Aktualisierungstoken ist mehrere Stunden gültig.
3. Als Zugang zu Ressourcen benötigt ein Server neben einem Zugangstoken ein Aktualisierungstoken vom Client.

3 Nutzung des Frameworks OAuth 2.0

In diesem Kapitel wird untersucht, welche Vor- und Nachteile das Framework OAuth 2.0 hat und wie es zu implementieren ist.

Zur Erklärung des Ablaufs des Frameworks OAuth 2.0 werden zunächst die Instanzen aufgezeigt, die im Framework involviert sind. Es gibt vier Instanzen, die an dem Ablauf beteiligt sind:

- Ein *Ressourceneigentümer*, beispielsweise ein Benutzer. Im Nachfolgenden wird ausschließlich ein Benutzer betrachtet. (vgl. [Ri17]: S. 5)
- Ein *Client*: Das ist eine Webanwendung, mit dem der Benutzer interagiert, beispielsweise eine SinglePageApplication (wie es am Beispiel „OpenSlides“ der Fall ist). Für eine eindeutige Zuordnung eines Clients ist jeder Client durch eine Identifikation (genannt „Client-Id“) und ein sogenanntes Client-Geheimnis gekennzeichnet. (vgl. [Ri17]: S. 43 – 58)
- Ein sogenannter *Autorisierungsserver*: Er verwaltet die Autorisierung eines Clients. (vgl. [Ri17]: S. 22).
- *Ressourcen*, die geschützt werden und nur für autorisierte Clients mit einem Zugangstoken zugänglich sind. Ressourcen werden dabei von einem eigenen Server, genannt *Ressourcenserver*, verwaltet. Ressourcen sind beispielsweise Fotos bei einem Foto-Dienst (vgl. [Ri17]: S. 59 – 74).

Einen grundlegenden Ablauf stellt folgendes Schaubild dar:

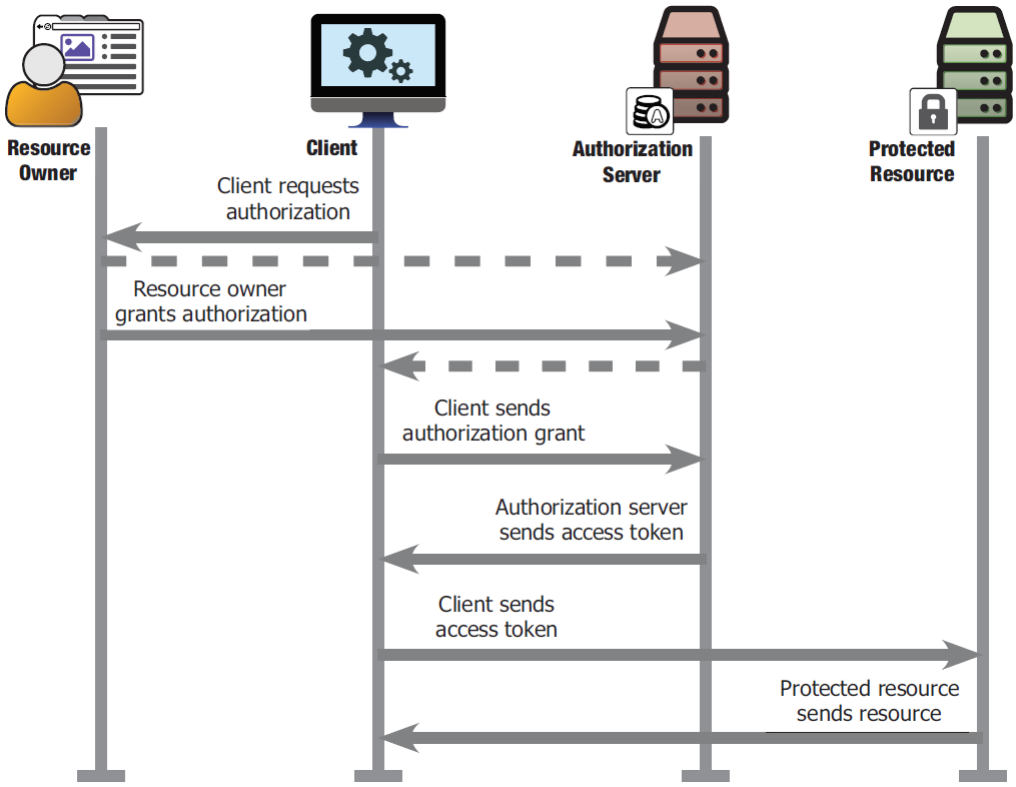

Abbildung 4: Grundlegender Ablauf von OAuth 2.0
Quelle: [Ri17]: S. 12

Abbildung 4 bildet den Ablauf des Autorisierungsframeworks OAuth 2.0 ab. Dabei ist dargestellt, dass ein Ressourceneigentümer einen Client dazu autorisiert auf Ressourcen zuzugreifen. Dies geschieht in folgenden Schritten:

1. Ein Client benötigt Rechte, um auf angefragte Ressourcen zuzugreifen. Er fragt die erforderlichen Rechte beim Benutzer an und leitet ihn zu einem Autorisierungsserver, der die Ressourcen schützt. (vgl. [Ri17]: S. 11ff)
2. Der Autorisierungsserver erfragt eine Bestätigung der Autorisierung des Clients (vgl. [Ri17]: S. 11ff).
3. Danach sendet dieser die Bestätigung zum Autorisierungsserver, nachdem der Benutzer die erforderlichen Rechte dem Client gewährt. (vgl. [Ri17]: S. 11ff)
4. Der Autorisierungsserver erstellt daraufhin ein Zugangstoken für den Client (vgl. [Ri17]: S. 11ff).
5. Dieser sendet das Zugangstoken zu einem Ressourcenserver und fragt Ressourcen an (vgl. [Ri17]: S. 11ff).
6. Der Ressourcenserver sendet nach einer Verifikation des Zugangstokens die angefragten Ressourcen zum Client (vgl. [Ri17]: S. 11ff).

Laut der Definition von OAuth 2.0 in RFC6749 (vgl. [Ha12]) gibt es vier verschiedene Möglichkeiten, das Framework OAuth 2.0 umzusetzen (vgl. [Ha12]). Jede Möglichkeit zeigt einen Weg auf, wie ein Client dazu autorisiert wird, Zugang zu Ressourcen zu erhalten (vgl. [Ha12]):

1. Autorisierung eines Clients durch einen Indirekten Ablauf
2. Autorisierung eines Clients durch einen Autorisierungscode
3. Autorisierung eines Clients durch Zugangsdaten eines Benutzers
4. Autorisierung eines Clients durch Zugangsdaten des Clients

Laut [Pa20] soll der Indirekte Ablauf nicht sicher genug sein und nach Möglichkeit eine Autorisierung mittels Autorisierungscodes genutzt werden (vgl. [Pa20]: S. 49). Aus diesem Grund wurde die *Autorisierung mittels Autorisierungscodes* erweitert: die *Autorisierung mittels Autorisierungscodes nach PKCE*. (vgl. [Pa20]: S. 57 – 60) Die Autorisierung mittels Autorisierungscodes nach PKCE ist für Clients relevant, die ein Client-Geheimnis nicht sicher bewahren können, beispielsweise SinglePageApplications. Dies liegt daran, dass SinglePageApplications Anwendungen sind, die direkt im Browser ausgeführt werden. Dadurch ist die Ausführung einer SinglePageApplication für den Browser ersichtlich. (vgl. [Ri17]: S. 110)

3.1 Beschreibung der verschiedenen Abläufe zur Autorisierung

Die nachfolgenden Kapitel, insbesondere die Entwicklung des Authentifizierungsdienstes von OpenSlides, beziehen sich ausschließlich auf die Autorisierung mittels Autorisierungscodes, da dieser Ablauf nach [Ri17] allen anderen Abläufen zugrunde liege (vgl. [Ri17]: S. 46). Deshalb sind die Abläufe *Autorisierung mittels Autorisierungscodes* und *Autorisierung mittels Autorisierungscodes nach PKCE* im Nachfolgenden beschrieben. Die drei anderen Abläufe können im Anhang B nachgelesen werden.

3.1.1 Autorisierung mittels Autorisierungscode

Das nachfolgende Schaubild stellt den Ablauf einer Autorisierung mittels Autorisierungscodes im Detail dar:

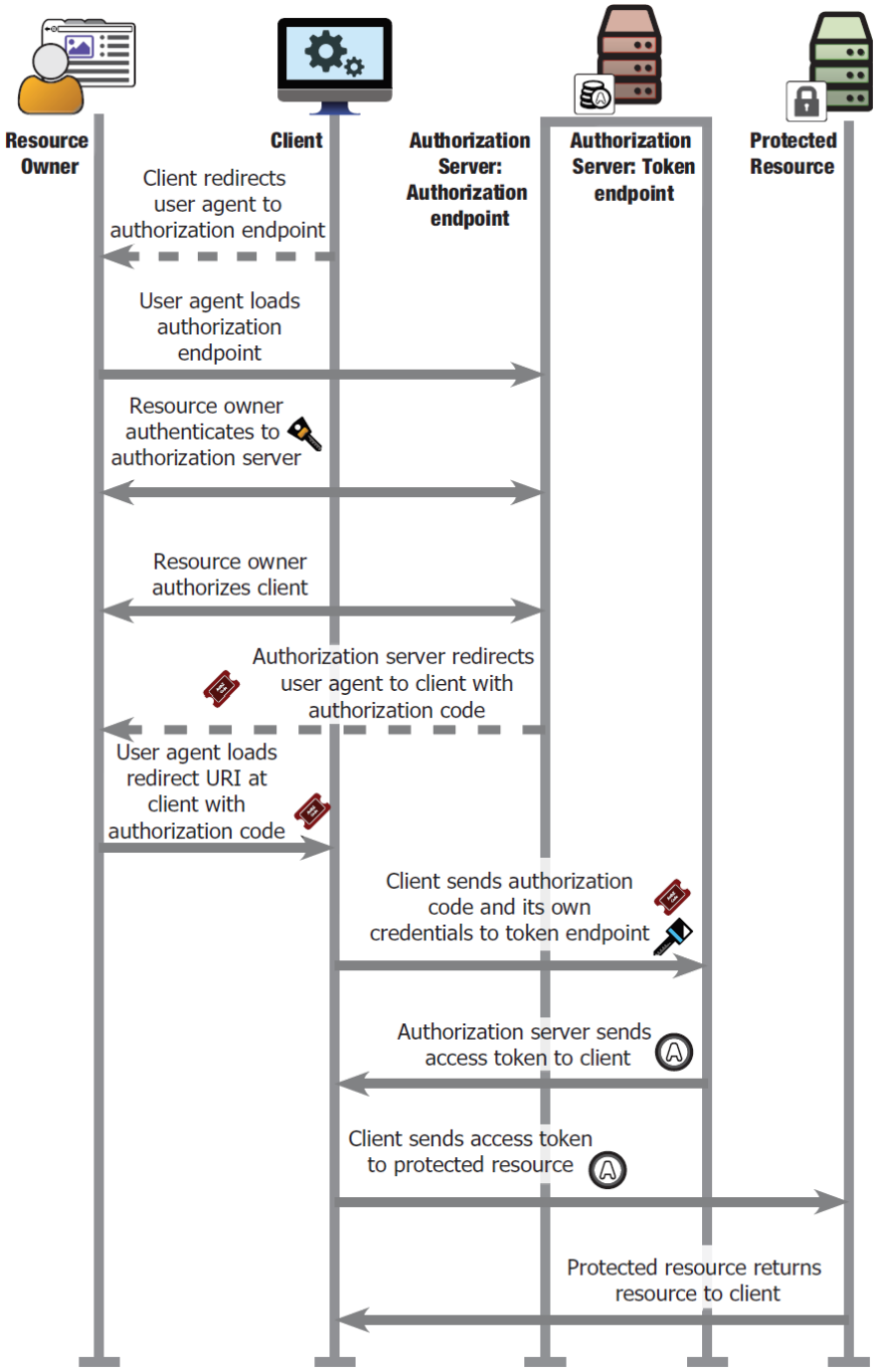

Abbildung 5: Autorisierung mittels Autorisierungscodes
Quelle: [Ri17]: S. 23

Abbildung 5 zeigt: Bei einer Autorisierung mittels Autorisierungscodes leitet ein Client den Browser (= „*User Agent*“) eines Benutzers (= „*Resource owner*“) weiter zum sogenannten

Autorisierungs-Endpunkt (= „Authorization endpoint“) eines Autorisierungsservers. Der Benutzer authentifiziert sich beim Autorisierungsserver und autorisiert daraufhin den Client. In diesem Zusammenhang leitet der Autorisierungsserver den Browser sowie einen Autorisierungscode wieder zurück zum Client. Dieser sendet eine Anfrage mittels der POST-Methode von HTTP zum *Token-Endpunkt (= „Token endpoint“)* des Autorisierungsservers. Die Anfrage enthält den Autorisierungscode und die Client-Id sowie das Client-Geheimnis des Clients. Als Antwort erhält der Client ein Zugangstoken (= „Access Token“). Mit dem Zugangstoken hat der Client Zugriff auf die geschützte Ressource (= „*Protected resource*“). (vgl. [Ri17]: S. 22 – 32)

Ein Vorteil dieses Ablaufs ist, dass Zugangsdaten eines Ressourceneigentümers ausschließlich zwischen Autorisierungsserver und Ressourceneigentümer ausgetauscht werden. Dadurch sind die Zugangsdaten eines Benutzers beim Client nicht ersichtlich. (vgl. [Ha12]) Folglich können sie nicht durch einen Client ergriffen werden, der von einem Angreifer manipuliert wurde, um Zugangsdaten auszulesen.

Die Authentifizierung eines Benutzers (= Authentifizierungsseite) und die Bestätigung der Autorisierung eines Clients (= Bestätigungsseite) finden jeweils auf einer eigenen Internetseite, die von einem Autorisierungsserver ausgeliefert werden, statt (vgl. [Ri17]: S. 79 – 82).

3.1.2 Autorisierung mittels Autorisierungscode nach PKCE

Webanwendungen, die als reine JavaScript-Anwendungen geschrieben sind, wie SinglePage-Applications, können nicht ohne Weiteres in das Framework OAuth 2.0 eingebunden werden, da sie das Client-Geheimnis nicht geheim halten können. (vgl. [Pa20]: S.49) Aus diesem Grund ist die Autorisierung mittels Autorisierungscodes um einen sogenannten „*Proof Key for Code Exchange (PKCE […])*“ ([Ri17]: S. 174) erweitert worden. Dadurch können SinglePageApplications durch eine Autorisierung mittels Autorisierungscodes ohne Client-Geheimnis autorisiert werden. Für die Umsetzung der Erweiterung bedarf es zweier Hilfsmittel: einem sogenannten *Code-Prüfer* (eine Zeichenfolge bestehend aus 43 bis 128 Zeichen (vgl. [Ri17]: S. 175)) und einer *Hash-Funktion* (eine Hash-Funktion ist optional, wird aber empfohlen (vgl. [Ri17]: S. 175)).

Die Autorisierung mittels Autorisierungscodes wird um die Punkte *Code-Prüfer und Hash-Funktion* bei der Erweiterung nach PKCE ergänzt, wie es bei [Ri17] beschrieben ist (vgl. [Ri17]: S. 174 – 178):

1. Zur Autorisierung eines Clients, sendet dieser neben seiner Client-Id einen Base64-kodierten Hash-Wert (vgl. [Pa20]: S. 50) (genannt: „*Code-Challenge*“ ([Pa20]: S. 50)) erzeugt aus einem sogenannten Code-Prüfer sowie die benutzte Hash-Funktion (falls

eine Hash-Funktion angegeben ist) an einen Autorisierungsserver und leitet den Browser eines Benutzers an den Autorisierungsserver weiter. (vgl. [Ri17]: S. 174 – 178)
2. Der Autorisierungsserver speichert den Hash-Wert sowie die genutzte Hash-Funktion ab und generiert einen Autorisierungscode, nachdem der Benutzer die Autorisierung bestätigt hat. (vgl. [Ri17]: S. 174 – 178)
3. Im nächsten Schritt leitet der Autorisierungsserver den Browser zurück zum Client und übergibt dabei den Autorisierungscode. (vgl. [Ri17]: S. 174 – 178)
4. Der Client sendet daraufhin neben dem Autorisierungscode und seiner Client-Id den Code-Prüfer. Der Autorisierungsserver bildet aus dem Code-Prüfer den Hash-Wert unter Verwendung der genannten Hash-Funktion und vergleicht den neu berechneten Hash-Wert mit dem, den der Autorisierungsserver zuvor gespeichert hat. (vgl. [Ri17]: S. 174 – 178)
5. Sind beide Hash-Werte identisch, ist die Anfrage korrekt und der Autorisierungsserver stellt ein Zugangstoken aus (vgl. [Ri17]: S. 174 – 178).

3.2 Implementieren des Frameworks OAuth 2.0

Dieses Kapitel beschreibt eine Implementierung des Frameworks OAuth 2.0 durch Autorisierung mittels Autorisierungscodes. Dabei wird die Implementierung von drei Seiten betrachten: aus Sicht eines Clients, aus Sicht eines Autorisierungsservers und aus Sicht eines Ressourcenservers.

3.2.1 Was sind „Bearer“-Tokens?

„*Bearer*“-Tokens (zu Deutsch: „Inhaber“) (vgl. [Jo12]) sind Tokens, die anzeigen, dass ein bestimmtes System die Tokens erstellt hat (der Aussteller). Die Tokens bieten Zugang zu dem System, unabhängig davon, wer das Token nutzt. Im Kontext von OAuth 2.0 ist der Aussteller ein Autorisierungsserver. Dadurch können „Bearer“-Tokens genutzt werden, um Ressourcen im System des Servers anzufragen. (vgl. [Ri17]: S. 168ff)

„Bearer“-Tokens können auf drei verschiedene Weisen in einer Anfrage eingebunden werden (vgl. [Ri17]: S. 60f):

1. *Authorization-Header*: Ein Bearer-Token kann in den Authorization-Header gesetzt werden. Dazu wird es durch das Wort „Bearer“ eingeleitet, gefolgt von dem Token selbst. Dabei ist es gleichwertig, wie die einzelnen Buchstaben in „Bearer“ geschrieben werden (groß oder klein). (vgl. [Ri17]: S. 60f)
2. *In dem Body einer HTTP-Anfrage*: Ein Bearer-Token kann als Token mit dem Schlüssel „*access_token*“ ([Ri17]: S. 61) in den Body einer HTTP-Anfrage geschrieben werden, sodass es unter dem Schlüssel abgefragt wird. Diese Methode ist nach [Ri17] nicht empfohlen, da sie die Eingabemöglichkeiten beschränkt. (vgl. [Ri17]: S. 61)

3. *Als Query-Parameter*: Ein Bearer-Token kann als dritte Möglichkeit als ein Query-Parameter in einer HTTP-Anfrage übermittelt werden. Ein Bearer-Token als Query-Parameter zu übermitteln ist allerdings nur empfohlen, wenn die beiden zuerst genannten Methoden nicht verfügbar sind. Dies ist der Fall, wenn ein BearerToken nicht in den Authorization-Header oder Body einer Nachricht geschrieben werden kann. (vgl. [Ri17]: S. 61)

3.2.2 Implementieren eines Clients im Kontext von OAuth 2.0

Im Kontext des Frameworks OAuth 2.0 wird ein Client in fünf Schritten implementiert (vgl. [Ri17]: S. 43 – 58):

1. Als erstes wird ein Client bei einem Autorisierungsserver registriert, sodass dem Client eine Client-Id und ein Client-Geheimnis zugeordnet wird (vgl. [Ri17]: S. 43 – 58).
2. In einem zweiten Schritt leitet der Client den Browser eines Benutzers an den Autorisierungs-Endpunkt des Autorisierungsservers, um autorisiert zu werden. Dabei übermittelt der Client dem Autorisierungsserver seine Client-Id, einen Antworttypen (bei einer Autorisierung mittels Autorisierungscodes lautet der Antworttyp „*code*“ ([Ri17]: S. 81f)) sowie eine Weiterleitungs-URL und einen sogenannten Geltungsbereich (aus dem Englischen „*scope*“ ([Ri17]: S. 88)). (vgl. [Ri17]: S. 44 – 50) Der Geltungsbereich gibt an, welche Ressourcen der Client abfragen kann (vgl. [Ri17]: S. 88 – 92). Durch den Antworttypen identifiziert ein Autorisierungsserver, welchen Autorisierungsablauf ein Client anfragt. Auf die Weiterleitungs-URL leitet der Autorisierungsserver den Benutzer weiter. Zur Übermittlung der einzelnen Daten werden diese als Query-Parameter in die URL gesetzt, an die der Benutzer weitergeleitet wird. Daraufhin muss dieser bestätigen, dass der anfragende Client dazu berechtigt ist, auf seine hinterlegten Daten (Ressourcen) zuzugreifen. Der Autorisierungsserver generiert daraufhin einen Autorisierungscode als Antwort und leitet den Browser des Benutzers wieder zurück zum Client (an die angegebene Weiterleitungs-URL). (vgl. [Ri17]: S. 43 – 58)
3. Als dritter Schritt sendet der Client den Autorisierungscode in einer Anfrage mit der POST-Methode an den Token-Endpunkt des Autorisierungsservers (vgl. [Ri17]: S. 49). Bei dieser Anfrage ist das Client-Geheimnis enthalten, das dem Client zugeordnet wurde. Außerdem beinhaltet die Anfrage Client-Id und einen Bewilligungstypen (bei einer Autorisierung mittels Autorisierungscodes lautet der Bewilligungstyp „*authorization_code*“ ([Ri17]: S. 84f) sowie die Weiterleitungs-URL, die bereits in Schritt 2 definiert wurde. (vgl. [Ri17]: S. 49f) Dabei sind Client-Id und Client-Geheimnis als eine Base64-kodierte Zeichenfolge und mittels Doppelpunktes konkateniert in den Autorisierungs-Header der Anfrage gesetzt. Die Weiterleitungs-URL muss in der Anfra-

ge an den Token-Endpunkt gesetzt sein, da dies in der Spezifikation von OAuth 2.0 definiert ist (vgl. [Ri17]: 49). Als Antwort auf die Anfrage sendet der Autorisierungsserver ein Zugangstoken und ein Aktualisierungstoken an den Client zurück. (vgl. [Ri17]: S. 43 – 58)

4. Mit dem Zugangstoken hat der Client Zugang zur Anwendung und kann benutzerbezogenen Informationen innerhalb des Geltungsbereichs abfragen, die beim Ressourcenserver gespeichert sind (vgl. [Ri17]: S. 43 – 58).
5. Als letzter Schritt wird das Zugangstoken mithilfe des Aktualisierungstokens erneuert, sobald dieses abgelaufen ist (vgl. [Ri17]: S. 43 – 58).

3.2.3 Implementieren eines Ressourcenservers im Kontext des Frameworks OAuth 2.0

In diesem Abschnitt wird beschrieben, wie eine Ressource in den Kontext von OAuth 2.0 eingebaut wird, sodass ein Zugriff auf die Ressource nur durch ein von einem Autorisierungsserver ausgestellten Zugangstoken gewährt ist.

Nach [Ri17] sind (geschützte) Ressourcen und Autorisierungsserver oftmals in einem einzigen System realisiert. An dieser Stelle wird untersucht, wie geschützte Ressourcen von einem Autorisierungsserver getrennt, aber trotzdem durch ihn im Kontext von OAuth 2.0 geschützt sind. Die Ressourcen werden von einem *Ressourcenserver* verwaltet. (vgl. [Ri17]: S. 60f) Dadurch können Komponenten getrennt voneinander entwickelt werden, was der Zielarchitektur von OpenSlides entspricht.

Zur Implementierung eines Ressourcenservers auf Basis des Frameworks OAuth 2.0 sind vier Funktionalitäten einzubauen:

1. *Überprüfen, ob ein Zugangstoken in einer Anfrage enthalten ist:* Jeder Client braucht ein gültiges Zugangstoken, um Ressourcen anzufragen. Deshalb überprüft ein Ressourcenserver zuerst, ob ein Zugangstoken in einer Anfrage vorhanden ist. (vgl. [Ri17]: S. 60ff) Ein Zugangstoken ist entweder im Authorization-Header einer Anfrage, im Body einer Anfrage oder als Query-Parameter gesetzt (siehe Kapitel 3.2.1). Es wird an die nächste Funktion übergeben, wenn eines gefunden wird.
2. *Überprüfen der Gültigkeit eines Zugangstokens:* In einem nächsten Schritt überprüft der Ressourcenserver, ob das enthaltene Zugangstoken valide ist. Das bedeutet, er untersucht den Ablaufzeitpunkt des Zugangstokens und ob dieses dem Client zugeordnet ist. (vgl. [Ri17]: S. 62 – 65) Ist der Ablaufzeitpunkt noch nicht überschritten und das Zugangstoken ist dem anfragenden Client zugeordnet, wird es an die nächste Funktion übergeben.

3. *Überprüfen der Gültigkeitsbereiche eines Clients:* Im dritten Schritt gleicht der Ressourcenserver die gegebenen Rechte des anfragenden Clients mit den notwendigen Rechten ab. Der Ressourcenserver sendet einen Fehler als Antwort, wenn für die angefragte Ressource mehr Rechte erforderlich sind, als dem anfragenden Client zugeordnet sind. (vgl. [Ri17]: S. 65 – 70) Nach dem Überprüfen der Geltungsbereiche des Clients folgt eine letzte Überprüfung.
4. *Überprüfen von Benutzern:* Der Inhalt einer Ressource kann je nach Anwendung variieren. Deshalb werden in einem letzten Schritt die Informationen zusammengetragen, die angefragt werden und dem Benutzer zugeordnet sind, der den anfragenden Client autorisierte. (vgl. [Ri17]: S. 70 – 74) „*During the OAuth process, the client never knew that it was talking to […] someone […]. […] This is a powerful design pattern, as it can protect the resource owner's privacy by not revealing personally identifying information unnecessarily.*“ ([Ri17]: S. 72f) Hieraus geht hervor, dass trotz benutzerabhängiger Inhalte einer Ressource personenbezogene Daten eines Benutzers nicht an einen Client gelangen (vgl. [Ri17]: S. 72f).

Bei einer Anfrage an einen Ressourcenserver werden diese vier Funktionalitäten aufgerufen. Die angefragten Ressourcen werden als Antwort erst dann gesendet, wenn alle vier Funktionalitäten die Anfrage bearbeitet haben. Am Ende wird dem anfragenden Client die angefragte Ressource als Antwort gesendet, nachdem die vier Funktionalitäten durchlaufen wurden. (vgl. [Ri17]: S. 60 – 74)

3.2.4 Implementieren eines Autorisierungsservers im Kontext des Frameworks

Ein Autorisierungsserver ist von zentraler Bedeutung für das Framework OAuth 2.0: „*Only the authorization server can authenticate users, register clients, and issue tokens.*“ ([Ri17]: S. 75) Ein Client im Framework OAuth 2.0 wird lediglich dazu bevollmächtigt, Ressourcen abzufragen und ein Ressourcenserver bearbeitet nur Anfragen zu Ressourcen, wie in den oberen beiden Kapiteln beschrieben. Ein Autorisierungsserver verwaltet die Autorisierung von Zugriffen auf Ressourcen. Er kontrolliert, welcher Client Zugriff auf welche Ressourcen hat. (vgl. [Ri17]: S. 75) Dabei zeigt [Ri17] auf, dass zur Entwicklung eines Autorisierungsservers auf Basis des Frameworks OAuth 2.0 ein HTTP-Server benutzt wird, da Komponenten im Kontext von OAuth 2.0 mittels HTTP miteinander kommunizieren (vgl. [Ri17]: S. 77f). Für einen Autorisierungsserver ist eine Datenbank erforderlich.

Die folgenden fünf Punkte sind für eine Entwicklung eines Autorisierungsservers auf Basis des Frameworks OAuth 2.0 zu berücksichtigen:

1. *Registrieren eines Clients:* Ein Client muss bei einem Autorisierungsserver registriert sein, damit er Zugriff auf Ressourcen erhält und eine Autorisierung nach OAuth 2.0 stattfindet. Fragt ein Client einen Autorisierungsserver nach einem Autorisierungscode oder Zugangstoken an, ohne beim Autorisierungsserver registriert zu sein, wird der Vorgang abgebrochen. (vgl. [Ri17]: S. 76 – 78)
2. *Ein Autorisierungs-Endpunkt:* Für eine Autorisierung mittels Autorisierungscodes muss ein Endpunkt definiert sein, bei dem ein Client einen Autorisierungscode erhält (vgl. [Ri17]: S. 78). Für den Vorgang im Kontext von OAuth 2.0 muss ein Client bei einer Anfrage an den Autorisierungs-Endpunkt seine Client-Id, eine Weiterleitungs-URL und einen Antworttypen definieren. (vgl. [Ri17]: S. 78 – 82) Durch die Client-Id gelingt eine eindeutige Zuordnung des Clients beim Autorisierungsserver. Der Antworttyp einer Anfrage eines Clients gibt an, durch welchen Ablauf nach OAuth 2.0 der Client autorisiert wird. Ein Autorisierungsserver unterstützt nur eine Teilmenge von Abläufen nach OAuth 2.0 und kann mit dem Antworttypen entscheiden, ob er eine Anfrage bearbeitet. Wird der angefragte Antworttyp nicht unterstützt, kann er die Anfrage nicht bearbeiten. (vgl. [Ri17]: S. 78-82) Im Fall von einer Autorisierung durch einen Autorisierungscode lautet der Antworttyp „code“ (siehe Kapitel 3.2.2).
3. *Ein Token-Endpunkt:* Zusätzlich zu einem Autorisierungs-Endpunkt muss für eine Autorisierung mittels Autorisierungscodes ein Token-Endpunkt definiert sein, durch den ein Client ein Zugangstoken erhält (vgl. [Ri17]: S. 82f). Eine Anfrage an den Token-Endpunkt eines Autorisierungsservers muss folgende Parameter enthalten: eine Client-Id, ein Client-Geheimnis, eine Weiterleitungs-URL, ein Bewilligungstyp und ein Autorisierungscode (vgl. [Ri17]: S. 49f). Ein Client sendet seine Client-Id sowie sein Client-Geheimnis gesetzt im Autorisierungs-Header seiner Anfrage (siehe Kapitel 3.2.2). Die Weiterleitungs-URL, die bereits in der Anfrage an den Autorisierungs-Endpunkt enthalten ist, muss erneut angegeben werden (siehe Kapitel 3.2.2). Der Bewilligungstyp für den Fall einer Autorisierung durch einen Autorisierungscode lautet „*authorization_code*“ ([Ri17]: S. 84f). Sind alle Parameter in der Anfrage an den Token-Endpunkt enthalten, gibt der Autorisierungsserver als Antwort ein JSON-Objekt mit folgenden Werten zurück: Zugangstoken und Token-Typ (beispielsweise „*Bearer*“ ([Ri17]: S. 86) oder „*Proof of Possession*“ ([Ri17]: S. 86)).
4. *Hinzufügen von Aktualisierungstokens:* Für eine Autorisierung durch das Framework OAuth 2.0 nutzen Clients Aktualisierungstokens, um Zugangstokens zu erneuern. Dieser Vorgang ist sinnvoll, da ein Benutzer dadurch seltener seine Zugangsdaten eingeben muss (siehe Kapitel 2.3.3). Für das Erneuern von Zugangstokens ist bei einem Autorisierungsserver eine URL definiert. An diese URL sendet ein Client sein Aktualisierungstoken, um ein neues vom Autorisierungsserver generiertes Zugangstoken zu erhalten. Der Autorisierungsserver speichert ein Aktualisierungstoken in seiner Daten-

bank neben dem zugehörigen Client ab. Die Aktualisierung eines Zugangstokens ist nur dann gültig, wenn Aktualisierungstoken und Client übereinstimmen. (vgl. [Ri17]: S. 86ff)

5. *Zuweisung von Geltungsbereichen:* Durch Geltungsbereiche werden einem Client verschiedene Rechte zugeordnet. So bekommt er beispielsweise Rechte zum Lesen oder Schreiben oder Löschen (von etwas). Bei der Registrierung eines Clients wird festgelegt, welche Geltungsbereiche er hat. Diese weist ein Autorisierungsserver dem Client in seiner Datenbank zu, sodass eine Zuordnung zwischen Client und Geltungsbereichen erfolgt. Startet ein Client den Ablauf von OAuth 2.0, muss ein Benutzer die Autorisierung des Clients und Zugriff auf dessen Geltungsbereiche sowie die damit verbundenen Rechte bestätigen. Bei der Anfrage eines Clients an den Autorisierungs-Endpunkt sendet der Client zusätzlich seine Geltungsbereiche, auf die er Zugriff bekommt. Ein Autorisierungsserver gleicht die Geltungsbereiche in der Anfrage mit denen aus seiner Datenbank ab, damit einem Client nicht mehr Rechte zugeordnet werden, als registriert wurden. (vgl. [Ri17]: S. 88 – 92)

3.3 Angriffsverfahren im Ablauf von OAuth 2.0

XSS und CSRF (siehe Glossar) sind zwei Angriffsverfahren, die allgemein bei der Entwicklung von Webanwendungen zu berücksichtigen sind. In diesem Abschnitt werden weitere Angriffsverfahren, die bei einer Implementierung von OAuth 2.0 betrachtet werden müssen, beschrieben. Zusätzlich werden Präventivmaßnahmen gegen CSRF-Angriffe im Framework OAuth 2.0 genannt.

3.3.1 Eine geänderte Weiterleitungs-URL

Ein Client sendet eine Autorisierungsanfrage an einen Autorisierungsserver. Bei der Anfrage ist eine Weiterleitungs-URL angegeben, an die der Autorisierungsserver den für die Anfrage benutzten Browser weiterleitet (siehe Kapitel 3.2.2).

Eine Attacke durch eine geänderte Weiterleitungs-URL

Eine Attacke durch eine geänderte Weiterleitungs-URL erfolgt, indem ein Angreifer die Weiterleitungs-URL einer Autorisierungsanfrage ändert. Dadurch wird ein Browser auf die URL weitergeleitet, die der Angreifer bestimmt hat, zum Beispiel dessen Internetseite. Auf diese Weise kann ein Angreifer alle Autorisierungscodes oder sogar Zugangstokens, die von einem Autorisierungsserver erstellt wurden, abhören und für sich nutzen. (vgl. [Bih15]: S. 160)

Bihis führt hierfür als Beispiel einen Autorisierungsserver an, bei dem dynamische Weiterleitungs-URLs zulässig sind (beispielsweise „*goodapp.com/**“ ([Bih15]: S. 160)). Der Stern in der Weiterleitungs-URL ist ein Platzhalter und kann durch verschiedene Zeichenfolgen ersetzt

werden. Das heißt, sowohl ‚goodapp.com/hund' als auch ‚goodapp.com/tisch' treffen als Weiterleitungs-URL zu. In diesem Fall platziert ein Angreifer für seinen Angriff einen Link auf einer Internetseite, der auf den Autorisierungs-Endpunkt eines Autorisierungsservers führt und auf eine von ihm erstellte Unterseite von ‚goodapp.com' weiterleitet. Bei der Weiterleitung übermittelt der Autorisierungsserver einen Autorisierungscode, den der Angreifer nutzt, um ein Zugangstoken und Zugriff auf die damit verbundenen Daten beim Autorisierungsserver zu erhalten. (vgl. [Bih15]: S. 160ff)

Maßnahmen gegen eine Veränderung einer Weiterleitungs-URL

Es gibt zwei Maßnahmen, die getroffen werden können, um einem Angriff durch eine veränderte Weiterleitungs-URL vorzubeugen (vgl. [Bih15]: S. 162):

1. Eine explizite Weiterleitungs-URL statt eines Platzhalters angeben.
2. Keine Autorisierungsserver nutzen, die Platzhalter erlauben (oder bevorzugen).

3.3.2 Imitation eines Benutzers

Eine Autorisierung mittels Autorisierungscodes erfolgt in zwei Schritten: Zuerst wird ein Benutzer authentifiziert, und dann bestätigt er die Autorisierung eines Clients.

Die Imitation eines Benutzers als Attacke

Bei diesem Angriff baut ein Angreifer einen Client, der den ersten Schritt, die Authentifizierung eines Benutzers, umgeht, indem eine für diesen unsichtbare Seite den Autorisierungsprozess einleitet und Eingaben programmatisch durchführt. Der Client sendet Eingaben, wie beispielsweise die Bestätigung der Autorisierung, zum Autorisierungsserver, die dieser erwartet. Dadurch wird der Client mittels Autorisierungscode autorisiert, ohne dass eine Interaktion mit dem Benutzer stattfindet. (vgl. [Lod13]) Es gibt drei Verfahren dafür:

1. Der Benutzer ist bereits beim Autorisierungsserver authentifiziert (vgl. [Lod13]).
2. Der Client sendet eine Anfrage, um für einen Geltungsbereich autorisiert zu werden, der ihm ohne Bestätigung des Benutzers zugeordnet wird. Daraufhin sendet er weitere Anfragen, um für weitere Geltungsbereiche autorisiert zu werden. (vgl. [Lod13])
3. Der Autorisierungsserver authentifiziert den Benutzer, ohne dass dieser seine Zugangsdaten eingeben und mit dem Autorisierungsserver interagieren muss, beispielsweise durch Zertifikate (vgl. [Lod13]).

Maßnahmen gegen eine Imitation eines Benutzers

Eine Präventivmaßnahme gegen die Imitation eines Benutzers ist das explizite Einbinden einer Interaktion eines Benutzers. Dies erfolgt auf drei verschiedenen Wegen:

1. Die Authentifizierung sowie die Bestätigung eines Benutzers befinden sich auf derselben Internetseite eines Autorisierungsservers (vgl. [Lod13]).
2. Ein Benutzer muss ein sogenanntes CAPTCHA ausfüllen (vgl. [Lod13]). Ein CAPTCHA ist ein Fenster, bei dem ein Benutzer verschiedene Bilder sieht und zu einem vorgegebenen Thema passende Bilder anklicken muss. Dadurch wird eine Interaktion mit einem Benutzer erzwungen.
3. Ein Benutzer erhält einen einmaligen Code, den er zur Bestätigung eingeben muss (vgl. [Lod13])

3.3.3 CSRF-Attacken im Kontext von OAuth 2.0

CSRF-Attacken wurden im Rahmen dieser Ausarbeitung bereits bei der Nutzung von Tokens für eine Verbindung zwischen Server und Client betrachtet (siehe Kapitel 2.3.1). Zur Absicherung gegen CSRF-Attacken werden im Allgemeinen Zugangstokens eines Clients in den Arbeitsspeicher eines Tabs gelegt und Aktualisierungstokens in Cookies mit der Eigenschaft „HttpOnly“ gesetzt (siehe Kapitel 2.3.3). Im Kontext von OAuth 2.0 ist diese Vorgehensweise nicht möglich, da im Ablauf von OAuth 2.0 keine Cookies vorgesehen sind.

CSRF-Attacken im Ablauf von OAuth 2.0

Bei der Autorisierung mittels Autorisierungscodes initiiert ein Client den Ablauf mit einer Anfrage und übermittelt dabei einem Autorisierungsserver eine Weiterleitungs-URL (siehe Kapitel 3.1.2). Zu der Weiterleitungs-URL wird der Benutzer zurückgeleitet, sobald er den Client autorisiert. Bei der Weiterleitung übermittelt der Autorisierungsserver einen Autorisierungscode, mit dem ein Client ein Zugangstoken erhält. Ohne Schutz sendet ein Angreifer einen selbst generierten Autorisierungscode an den Client. Dieser erkennt nicht, dass es sich dabei um einen Angriff handelt und setzt den Ablauf mit dem vom Angreifer erhaltenen Autorisierungscode fort. Das bedeutet, der Client nutzt den Autorisierungscode, um ein Zugangstoken zu erhalten. Dieses Zugangstoken hört der Angreifer ab und hat damit Zugriff auf die Daten beim Ressourcenserver. (vgl. [Bih15]: S. 154ff)

Maßnahmen gegen CSRF-Attacken

Auf der Seite eines Clients:

Laut [Ha12] muss ein Client vor CSRF-Angriffen geschützt sein (vgl. [Ha12]). Hierfür kann ein sogenannter „*State*“-Parameter ([Ha12]) genutzt werden. Ein Client setzt bei einer Autorisierungsanfrage an einen Autorisierungsserver eine zufällige Zeichenfolge als „State“-Parameter. Der Autorisierungsserver setzt in seiner Antwort ebenfalls den „State“-Parameter mit demselben Wert. (vgl. [Ha12])

Dieses Vorgehen wird empfohlen: „*When an authorization code or access token is returned back to the redirect URI, your application can validate (the state) to ensure that it was indeed used as part of a valid authorization request initiated by the user, and not by some attacker.*“ ([Bih15]: S. 156). Hieraus geht hervor, dass ein Client nach einer Autorisierungsanfrage den gesendeten und erhaltenen „State“-Parameter miteinander vergleicht. Wie im vorherigen Kapitel (Kapitel 3.2.2) beschrieben, ist der „State“-Parameter bei einem Angriff entweder falsch oder gar nicht gesetzt. Unterscheidet sich der vom Client gesendete „State“-Parameter vom erhaltenen, handelt es sich um einen Angriffsversuch und der Vorgang der Autorisierungsanfrage wird abgebrochen. (vgl. [Bih15]: S. 156ff)

Auf der Seite eines Servers:

Auf der Seite eines Autorisierungsservers kann ebenfalls ein „State“-Parameter genutzt werden, um CSRF-Attacken vorzubeugen, so Richer und Sanso (vgl. [Ri17]: S. 79ff). Ein Autorisierungsserver generiert eine zufällige Zeichenfolge und setzt diese als „State“-Parameter. Dieser wird an die Bestätigungsseite übergeben, auf der ein Benutzer einen Client autorisiert. Nach der Bestätigung der Autorisierung überprüft der Autorisierungsserver den „State“-Parameter. Ist er verändert oder nicht vorhanden, handelt es sich um eine (CSRF-) Attacke. (vgl. [Ri17]: S. 79ff)

3.4 Nutzen von JWT als Zugangstoken

Ein JWT ist eine Implementierungsart, die sich selbst verwaltet. Es hat einen festgelegten Ablaufzeitpunkt und eine Signatur, wodurch die Unverfälschtheit gewährleistet ist. Deshalb ist ein JWT so vertrauenswürdig, dass ein Autorisierungsserver oder ein Ressourcenserver ausschließlich die Signatur eines JWTs prüfen muss, um dessen Echtheit zu verifizieren. Dadurch eignen sich JWTs gut als Zugangstokens im Kontext von OAuth 2.0. (vgl. [Ri17]: S. 183)

3.5 Weiterführende Themen zu OAuth 2.0

Drei weiterführende Themen zu OAuth 2.0 können unter Anhang C nachgelesen werden.

4 Implementierung des Authentifizierungsdienstes

In Kapitel 2 wurde der Einsatz von Tokens theoretisch behandelt, in Kapitel 3 wurde das Framework OAuth 2.0 beleuchtet. In diesem Kapitel folgt eine Untersuchung der praktischen Umsetzung von OAuth 2.0 mittels Tokens. Neben der konkreten Vorgehensweise werden Erkenntnisse sowie aufgetretene Probleme aufgezeigt.

4.1 Grundlegende Einrichtung

Für die Umsetzung der theoretischen Inhalte aus Kapitel 2 und 3 wurde zunächst ein Server geschrieben, der *Authentifizierungsdienst von OpenSlides*. Der Authentifizierungsdienst bearbeitet ausschließlich Anfragen in Bezug auf die Authentifizierung eines Benutzers. Das heißt, nur das Anmelden, das Abmelden und eine Liste mit allen angemeldeten Benutzern sind Bestandteil des Authentifizierungsdienstes von OpenSlides. Er erstellt keine Anträge, Wahlen oder Abstimmungen. Auch das Erstellen neuer Benutzer wird von einem anderen Dienst im Umfeld von OpenSlides übernommen, da Benutzer sich nicht selbst registrieren können.

Nach der Implementierung der grundlegenden Funktionalitäten folgte die Integration des Frameworks OAuth 2.0. Dafür wurden Endpunkte festgelegt, die für eine Autorisierung mittels Autorisierungscodes wichtig sind: Autorisierungs-Endpunkt und Token-Endpunkt. Des Weiteren wurde ein Registrierungs-Endpunkt sowie ein Aktualisierungs-Endpunkt bestimmt, wodurch Clients registriert und Zugangstokens erneuert werden. Durch die festgelegten Endpunkte autorisieren Benutzer externe Anwendungen (außerhalb des Kontextes von OpenSlides), damit diese auf Anträge, Wahlen oder Abstimmungen, die in Zusammenhang mit den Benutzern stehen, zugreifen.

Für die Zielerreichung „*die Implementierung des Authentifizierungsdienstes von OpenSlides*“ gibt es gleichwertig nutzbare Programmiersprachen, beispielsweise JavaScript/TypeScript (Node.js als serverseitige Entwicklungsumgebung), Java und Go. Für die Implementierung des Authentifizierungsdienstes wurde TypeScript als Programmiersprache gewählt. Damit verbunden ist Node.js die serverseitige Entwicklungsumgebung. Der Authentifizierungsdienst wurde mit dem Framework *Express* (vgl. [Str20]) realisiert. Durch Express werden bei einem Server Endpunkte durch URLs definiert, die angesprochen werden können. Außerdem werden Funktionen hinterlegt, die beim Aufruf der URLs ausgeführt werden. (vgl. [Str20a])

4.2 Implementierung eines tokenbasierten Ansatzes

Der Authentifizierungsdienst wurde um eine Bibliothek zu JSON Web Tokens (JWT) erweitert, da er mittels Tokens realisiert wurde. An den Authentifizierungsdienst gab es folgende fünf Anforderungen:

1. *Anmelden*: In der Datenbank bestehende Benutzer können sich mit einer Benutzername-Passwort-Kombination anmelden.
2. *Abmelden*: Benutzer können sich nach erfolgreichem Anmelden abmelden.
3. *Liste mit allen aktiven Sitzungen*: Benutzer mit entsprechenden Rechten (beispielsweise Administratoren) können sehen, wie viele Benutzer zurzeit angemeldet sind.

4. *Eine spezifische Sitzung unter gegebener Identifikation beenden*: Ein Administrator kann eine spezifische Sitzung eines anderen Benutzers explizit beenden und ihn damit vom System abmelden.
5. *Alle aktiven Sitzungen beenden*: Ein Administrator meldet alle anderen Benutzer ab, indem er alle aktiven Sitzungen, bis auf seine eigene, beendet.

Diese fünf Anforderungen sind notwendig, damit der Authentifizierungsdienst für OpenSlides eingesetzt werden kann. Dafür wurde der beschriebenen Vorgehensweise bei einer tokenbasierten Authentifizierung gefolgt (siehe Kapitel 2.4). Das heißt, es werden ein Zugangstoken als Zugang zur Anwendung und ein Aktualisierungstoken zur Aktualisierung eines Zugangstokens genutzt. Des Weiteren wird ein Zugangstoken clientseitig ausschließlich im Arbeitsspeicher eines Tabs gespeichert und nicht in einem globalen Speicher. Intern speichert der Authentifizierungsdienst eine Liste mit Identifikationen ausgestellter Zugangstokens. Dementsprechend erfolgte die konkrete Umsetzung der fünf Anforderungen wie folgt:

1. *Anmelden eines Benutzers:* Beim Server wurde ein Endpunkt mit der URL ‚/login' definiert, die mittels POST-Methode aufgerufen wird. Beim Aufruf dieser URL müssen Zugangsdaten eines Benutzers übermittelt werden, um ihn anzumelden. Deshalb überprüft der Authentifizierungsdienst zunächst, ob Zugangsdaten in der Anfrage enthalten sind. In einem nächsten Schritt gleicht er die erhaltenen Zugangsdaten mit den in seiner Datenbank hinterlegten Daten ab. Stimmen sie überein, schickt er dem Client als Antwort ein Zugangstoken. Des Weiteren setzt der Authentifizierungsdienst beim Client ein Cookie mit Eigenschaft „HttpOnly". Es enthält eine Identifikation der aktiven Sitzung des Clients und dient als Aktualisierungstoken (siehe Kapitel 2.3.3). Serverseitig wird die Identifikation der aktiven Sitzung jedes Clients in der Liste mit Identifikationen ausgestellter Zugangstokens gespeichert. Eine Anfrage (nach Ressourcen) an den Authentifizierungsdienst ist nur gültig, wenn das Zugangstoken in seiner Liste enthalten ist.
2. *Abmelden eines Benutzers:* Für das Abmelden eines Benutzers wurde beim Authentifizierungsdienst ein Endpunkt mit der URL ‚/logout' definiert. Diese ist mit der POST-Methode aufzurufen. Dabei wird das während der Anmeldung gesetzte Cookie serverseitig gelöscht. Die Identifikation der aktiven Sitzung des Clients löscht der Authentifizierungsdienst ebenfalls aus seiner Liste, sodass das aktuelle Zugangstoken ungültig wird.
3. *Liste mit allen aktiven Sitzungen:* Clients können eine Liste mit allen zurzeit angemeldeten Benutzern unter dem Endpunkt mit der URL ‚/list-all-session' erhalten. Dafür iteriert der Authentifizierungsdienst über die Liste mit den Identifikationen der aktiven Zugangstokens und ruft den zu einer Identifikation dazugehörigen Benutzernamen ab.

Diesen speichert der Authentifizierungsdienst in einer separaten List und sendet sie als Antwort zum anfragenden Client.

4. *Löschen einer aktiven Sitzung:* Das Beenden einer aktiven Sitzung wurde unter dem Endpunkt mit der URL ‚/clear-session-by-id' definiert. Dafür wird die URL mittels DELETE-Methode aufgerufen und eine Identifikation übergeben, deren Sitzung beendet werden soll. Der Authentifizierungsdienst durchsucht seine Liste mit den Identifikationen der aktiven Zugangstokens nach der übermittelten Identifikation. Ist sie vorhanden, löscht er sie aus seiner Liste. Damit wird die dazugehörige Sitzung beendet und der Benutzer abgemeldet, der mit dieser Sitzung angemeldet ist. Ist die Identifikation nicht vorhanden, wird ein Fehler als Antwort zurückgesendet.
5. *Löschen aller aktiven Sitzungen:* Das Beenden aller aktiven Sitzungen wurde beim Authentifizierungsdienst von OpenSlides implementiert, indem ein Endpunkt mit der URL ‚/clear-all-sessions-except-themselves' definiert wurde. Die URL wird mit der DELETE-Methode aufgerufen. Beim Aufruf leert der Authentifizierungsdienst seine Liste mit den Identifikationen der aktiven Zugangstokens (Ausnahme bildet das Zugangstoken des anfragenden Clients). Die Sitzung des anfragenden Clients wird nicht beendet, weil dieser daraufhin automatisch abgemeldet würde.

4.3 Erweiterung um das Framework OAuth 2.0

In dem Ablauf des Frameworks OAuth 2.0 sind vier verschiedene Instanzen involviert, wie in Kapitel 3 beschrieben:

- Ein *Ressourceneigentümer*, beispielsweise der Benutzer einer Anwendung.
- Ein *Client*, der vom Ressourceneigentümer autorisiert wird, beispielsweise eine SinglePageApplication
- Ein *Autorisierungsserver*, der die Autorisierung eines Clients verwaltet.
- *Geschützte Ressourcen*, die nur für Clients mit einem vom Autorisierungsserver ausgestellten Zugangstoken zugänglich sind und von einem *Ressourcenserver* verwaltet werden.

Das Framework OAuth 2.0 wurde im Rahmen dieser Ausarbeitung auf zwei Weisen umgesetzt: als Autorisierungsserver und als Client. Dadurch wurde der Ablauf von OAuth 2.0 in einer praktischen Umsetzung beleuchtet, da so die Kommunikation zwischen Autorisierungsserver und Client im Ablauf von OAuth 2.0 untersucht wurde.

1. Der Authentifizierungsdienst von OpenSlides wurde so erweitert, dass eine externe Anwendung durch den Ablauf *Autorisierung mittels Autorisierungscodes* autorisiert wird. Dadurch sind Anträge, Abstimmungen und Wahlen, die im Zusammenhang mit einem Benutzer stehen, für die externe Anwendung zugänglich.

2. Es wurde ein Beispiel-Client mithilfe des Frameworks Angular entwickelt. In dem Client wurde die *Autorisierung mittels Autorisierungscodes nach PKCE* umgesetzt (siehe 3.1.3 für mehr Informationen). Dadurch wurde eine für den Client von OpenSlides nahe Vorgehensweise zur Implementierung des Frameworks OAuth 2.0 erzielt.

4.3.1 Erweitern des Authentifizierungsdienstes um OAuth 2.0

Die Implementierung des Ablaufs *Autorisierung mittels Autorisierungscodes* in den Authentifizierungsdienst erfolgte, indem vier Endpunkte durch vier URLs definiert wurden: ein Autorisierungs-Endpunkt, ein Token-Endpunkt, ein Aktualisierungs-Endpunkt und ein Registrierungs-Endpunkt.

Autorisierungs-Endpunkt: Die Autorisierung eines Clients erfolgt durch eine Weiterleitung mit der GET-Methode von HTTP an die URL ‚/authorize'. Eine Anfrage enthält eine Client-Id, einen „State"-Parameter, eine Weiterleitungs-URL und einen Antworttypen (siehe Kapitel 3.2.4). Der Authentifizierungsdienst sendet einen Fehler, wenn einer dieser Parameter nicht enthalten ist oder ein anderer Antworttyp als „code" angefragt wird. Eine „Code-Challenge" und eine Hash-Funktion können ebenfalls gesendet werden, wenn ein Client eine *Autorisierung mittels Autorisierungscodes nach PKCE* durchführt.

Token-Endpunkt: Ein Client sendet eine Anfrage mit der POST-Methode an die URL ‚/token'. Dabei übermittelt er seine Client-Id, seine Weiterleitungs-URL sowie einen Bewilligungstypen. Des Weiteren ist entweder das Client-Geheimnis des Clients oder ein Code-Prüfer, wenn der Client eine *Autorisierung mittels Autorisierungscodes nach PKCE* durchführt, in der Anfrage enthalten. (siehe Kapitel 3.2.4) Der Authentifizierungsdienst sendet einen Fehler als Antwort, wenn einer der genannten Parameter fehlt. Sind alle übermittelten Parameter vorhanden und stimmen mit den in der Datenbank des Authentifizierungsdienstes gespeicherten überein, sendet dieser als Antwort ein JSON-Objekt mit folgenden Werten: ein Zugangstoken, den Token-Typ „Bearer" und ein Aktualisierungstoken.

Aktualisierungs-Endpunkt: An diesem Endpunkt senden Clients eine Anfrage mit der POST-Methode an die URL ‚/refresh', um ein neues Zugangstokens zu erhalten. Dafür wird ein Aktualisierungstoken sowie eine Client-Id durch die Anfrage übermittelt. Stimmt die Kombination aus Aktualisierungstoken und Client-Id mit der in einer Datenbank hinterlegten überein, sendet der Authentifizierungsdienst als Antwort ein neues gültiges Zugangstoken zurück, andernfalls sendet er einen Fehler.

Registrierungs-Endpunkt: An diesen Endpunkt sendet ein Client eine Anfrage mit der POST-Methode, um sich zu registrieren und eine Client-Id sowie ein Client-Geheimnis zu erhalten. Dabei übermittelt der Client eine Weiterleitungs-URL, einen Namen der Anwendung (die autorisiert wird), Geltungsbereiche (auf die die Anwendung Zugriff bekommt) und einen Be-

willigungstypen. Der Authentifizierungsdienst sendet einen Fehler als Antwort, wenn der Bewilligungstyp anders als „authorization_code“ ist. Andernfalls antwortet der Authentifizierungsdienst mit einer Client-Id sowie einem Client-Geheimnis.

Des Weiteren wurde ein Endpunkt mit der URL ‚/approve‘ bei dem Authentifizierungsdienst festgelegt, der jedoch nur für den Authentifizierungsdienst relevant ist. Der Endpunkt wird nach einer Autorisierung angesprochen und leitet einen Benutzer auf die Weiterleitungs-URL weiter. Dieser Endpunkt bearbeitet die Bestätigung oder Ablehnung einer Autorisierung. (vgl. [Ri17]: S. 79 – 82)

4.3.2 Erweitern des Clients von OpenSlides um OAuth 2.0

In diesem Kapitel wird erläutert, wie der Beispiel-Client um das Framework OAuth 2.0 erweitert wurde. Dafür wurde die *Autorisierung mittels Autorisierungscodes nach PKCE* verwendet, da der Beispiel-Client eine SinglePageApplication ist (ähnlich wie der Client von OpenSlides) (siehe Kapitel 3.1.2).

In dem Ablauf nach *Autorisierung mittels Autorisierungscode nach PKCE* sendet ein Client neben seiner Client-Id einen Code-Challenge sowie eine Hash-Funktion zu einem Autorisierungsserver und leitet einen Benutzer dabei zu ihm weiter. Daraufhin autorisiert der Benutzer den Client und wird zu diesem weitergeleitet. Der Client erhält dabei einen Autorisierungscode. Diesen sendet er mit dem Code-Prüfer, der zuvor zur Berechnung der Code-Challenge benutzt wurde, mittels POST-Methode an den Autorisierungsserver und erhält dafür ein Zugangstoken.

Der Beispiel-Client wurde mithilfe des Frameworks Angular entwickelt. Angular bietet eine Funktion, durch die HTTP-Anfragen versendet werden. Als erster Schritt wurde eine weitere Funktion implementiert, die eine kryptografisch sichere Zeichenfolge mit der Länge 43 erzeugt. Diese ist der Code-Prüfer für Anfragen zu einer Autorisierung. Als Hash-Funktion wird „SHA256“ verwendet (vgl. [Pa20]: S. 50f). Der Client sendet für Anfragen zur Autorisierung folgende Werte (vgl. [Pa20]: S. 50ff):

- den Antworttypen „code“
- seinen Geltungsbereich
- eine Weiterleitungs-URL
- eine beliebige Zeichenfolge als „state“-Parameter
- seine Client-Id
- eine Base64-kodierte Code-Challenge, berechnet aus einem Code-Prüfer und der genutzten Hash-Funktion
- die Hash-Funktion (abgekürzt als „S256“)

Die aufgelisteten Werte setzt er dafür als Query-Parameter in einer URL und leitet einen Benutzer zu dem Autorisierungsserver unter Verwendung der URL weiter. Der Autorisierungsserver übermittelt dem Client danach einen Autorisierungscode sowie den erhaltenen „state"-Parameter, nachdem der Benutzer die Autorisierung bestätigte. Daraufhin überprüft der Client den „state"-Parameter. Ist er identisch zu dem zuvor benutzten, sendet der Client mittels POST-Methode die folgenden Werte (vgl. [Pa20]: S. 53ff):

- den Autorisierungscode mit dem zuvor benutzten Code-Prüfer
- den Bewilligungstypen „authorization_code"
- die zuvor genannte Weiterleitungs-URL

Im Anschluss daran erhält er vom Autorisierungsserver ein Zugangstoken. Dieses speichert er temporär im Arbeitsspeicher.

Erhält er zusätzlich ein Aktualisierungstoken, kann damit auf zwei Arten verfahren werden:

1. Der Autorisierungsserver setzt ein Aktualisierungstoken in einem Cookie beim Client.
2. Die zweite Möglichkeit ist, dass ein Client ein Aktualisierungstoken ebenso wie das Zugangstoken nur temporär im Arbeitsspeicher eines Tabs speichert.

Ein Cookie mit der Eigenschaft „HttpOnly" kann nur serverseitig gesetzt werden, andernfalls ist es anfällig gegenüber XSS-Attacken. Das Speichern eines Aktualisierungstokens in einem Cookie hat zwei Vorteile: es ist browserweit gültig und permanent, bis die Gültigkeit des Cookies abläuft. Dadurch bleibt der Client autorisiert, wenn er in einem zweiten Tab geöffnet oder der aktive Tab aktualisiert wird (siehe Kapitel 2.3.3). Allerdings ist ein Ablauf des Frameworks OAuth 2.0 ausschließlich mit Tokens konzipiert und Cookies sind nicht vorgesehen, wodurch dieses Vorgehen nicht implementiert wird.

Die zweite Möglichkeit, ein Aktualisierungstoken wie ein Zugangstoken ebenfalls im Arbeitsspeicher temporär zu speichern, ist ebenso nicht anwendbar. Durch ein Aktualisierungstoken im Arbeitsspeicher kann der Client sein Zugangstoken erneuern, solange das Aktualisierungstoken gültig ist. Dieses Vorgehen ist jedoch nur solange möglich, bis der aktive Tab geschlossen oder aktualisiert wird. Außerdem gelten für ein Aktualisierungstoken im Arbeitsspeicher eines Tabs die gleichen Bedingungen wie für ein Zugangstoken. Dadurch ist ein Aktualisierungstoken redundant – stattdessen kann die Gültigkeit eines Zugangstokens verlängert werden, was dasselbe Resultat ist.

Andere Möglichkeiten zum Persistieren, wie der Lokale Speicher, der Sitzungsspeicher oder die IndexedDB eines Browsers, können nicht gewählt werden, da diese ebenfalls anfällig für XSS-Attacken sind (siehe Kapitel 2.3.3).

Deshalb bleibt die Frage offen, auf welche Weise ein Aktualisierungstoken gespeichert werden kann, sodass mehrere Verbindungen gleichzeitig benutzt werden können. Die Beantwortung der Frage erfolgt jedoch nicht an dieser Stelle, da dies nicht Gegenstand der Arbeit ist.

4.4 Aufgetretene Probleme und Erkenntnisse

Bei der Entwicklung eines Servers ist die sogenannte „*Cross-Origin-Resource-Sharing*"-Restriktion ([Mo20g]) zu berücksichtigen. Sie ist dann wichtig, wenn ein Client einen Server in einer anderen Domäne anfragt, um Ressourcen anzufordern. Durch die Restriktion wird verhindert, dass Ressourcen an nicht-erlaubte Domänen gesendet werden. Stattdessen werden Anfragen aus nicht-erlaubten Domänen abgelehnt.

Der Authentifizierungsdienst von OpenSlides legte bisher eine Domäne statisch fest, die erlaubt ist (die, in der der Authentifizierungsdienst sich befindet). Das Problem liegt darin, dass Anfragen aus anderen Domänen abgelehnt werden, Anfragen im Ablauf von OAuth 2.0 jedoch domänenübergreifend gesendet werden.

Dieses Problem ist durch einen Beitrag in einem Entwicklerforum gelöst worden. Statt eine Domäne statisch festzulegen, wird eine Liste mit erlaubten Domänen angelegt. Bei jeder Anfrage überprüft eine Funktion dynamisch, ob die Domäne einer Anfrage in der Liste erlaubter Domänen enthalten ist oder nicht. Die Anfrage wird zugelassen, wenn ihre Domäne in der Liste gefunden wird. (vgl. [Ch15])

5 Fazit und Ausblick

Das Ziel dieser Arbeit war es einen Authentifizierungsdienst zu implementieren. Dafür wurde das Framework OAuth 2.0 beleuchtet und Vor- und Nachteile von Tokens diskutiert.

Als Resultat wurde ein Authentifizierungsdienst erstellt, der auf Basis des Frameworks OAuth 2.0 externe Clients dazu autorisiert, auf benutzerbezogene Inhalte in OpenSlides zuzugreifen. Zum Erreichen des Ziels wurde als erstes eine Authentifizierung mittels Tokens umgesetzt und anschließend diese Authentifizierung um das Framework OAuth 2.0 erweitert.

Letztendlich lässt sich festhalten, dass der Einsatz von Tokens in Form von JWT dem Einsatz von Sessions als Authentifizierungsverfahren im Falle von OpenSlides vorzuziehen ist, da die Vorteile von Tokens gegenüber denen von Sessions überwiegen. Tokens sind flexibler einzusetzen und besitzen (im Fall von JWT) einen eingebauten Ablaufzeitpunkt. Dadurch ist OpenSlides dynamisch erweiterbar und einzelne Komponenten liegen verteilt vor. Zudem muss weder im Client noch im Server eine Funktion implementiert sein, die überprüft, ob ein Token gültig ist.

Das Framework OAuth 2.0 wurde in dieser Arbeit auf zwei Weisen umgesetzt. Dadurch zeigte sich die Kommunikation zwischen Server und Client in einem praktischen Ablauf. Die Verwendung von OAuth 2.0 ist generell sinnvoll, wenn eine Anwendung mit externen Anwendungen kommunizieren soll. Das Framework OAuth 2.0 ist hierbei nur mit Tokens realisierbar. Für OpenSlides kann der Einsatz des Frameworks dann bewertet werden, wenn OpenSlides 4 erscheint, da der Authentifizierungsdienst in OpenSlides 4 eingesetzt wird. Zu beachten ist, dass eine Implementierung des Frameworks OAuth 2.0 im Client von OpenSlides erst dann sinnvoll ist, wenn es eine gute Handhabung für Aktualisierungstokens für SinglePageApplications gibt, da zurzeit keine einsetzbare Möglichkeit vorhanden ist (siehe Kapitel 4.3.2).

Durch diese Ausarbeitung wurde der Authentifizierungsdienst von OpenSlides auf Basis des Autorisierungsframeworks OAuth 2.0 realisiert. OAuth 2.0 definiert nur eine Autorisierung, eine mögliche Erweiterung kann durch eine Authentifizierung auf Basis von „OpenID Connect“ erfolgen (siehe Anhang C). Des Weiteren kann eine Zwei-Faktor-Authentifizierung als weitere mögliche Erweiterung implementiert werden. Eine Zwei-Faktor-Authentifizierung authentifiziert einen Benutzer durch zwei Merkmale (Faktoren), die einen Benutzer eindeutig identifizieren sollen. Die Authentifizierung beim Authentifizierungsdienst von OpenSlides erfolgt zurzeit mit einer Benutzername-Passwort-Kombination. Das ist ein Faktor. Der zweite Faktor kann zum Beispiel ein Code sein, den ein Benutzer über eine angegebene E-Mail-Adresse oder Handynummer erhält und eingeben muss. In einer möglichen nachfolgenden Arbeit könnten Vor- und Nachteile einer Zwei-Faktor-Authentifizierung diskutiert sowie der Nutzen für OpenSlides beleuchtet werden.

6 Literaturverzeichnis

[An18]	A. Anand. (2018). *How Cross-Origin Resource Sharing requests affect your app's performance*. Abgerufen am 31. März 2020 von https://medium.com/free-code-camp/the-terrible-performance-cost-of-cors-api-on-the-single-page-application-spa-6fcf71e50147
[Ar18]	S. Arciszewski, S. Haussmann. (2018). *PASETO: Platform-Agnostic-SEcurity-TOkens* (work in progress). Abgerufen am 15. Juni 2020 von https://tools.ietf.org/id/draft-paragon-paseto-rfc-00.html
[Au14]	Auth0, Inc. (2014). *Introduction to JSON Web Tokens*. Abgerufen am 27. Mai 2020 von https://jwt.io/introduction/
[Au16]	Auth0, Inc. (2016). *Understanding Refresh Tokens*. Abgerufen am 31. März 2020 von https://auth0.com/learn/refresh-tokens/
[Au18]	Auth0, Inc. (2018). *Store Tokens*. Abgerufen am 31. März 2020 von https://auth0.com/docs/tokens/guides/store-tokens
[Au18a]	Auth0, Inc. (2018). *Sessions*. Abgerufen am 17. Juni 2020 von https://auth0.com/docs/sessions
[Ba17]	Y. Balaj. (2017). *Token-Based vs. Session-Based Authentication: A survey*.
[Bib20]	Bibliographisches Institut GmbH. (2020). *Persistenz*. Abgerufen am 31. März 2020 von https://www.duden.de/rechtschreibung/Persistenz
[Bih15]	C. Bihis (Packt Publishing Ltd.). (2015). *Mastering OAuth 2.0*. Birmingham.
[Cha19]	A. Chauhan. (2019). *Web Security: Authentication - Cookies vs. Tokens vs. HTML5 Web Storage*. Abgerufen am 31. März 2020 von https://labs.tadigital.com/index.php/2019/05/16/web-security-authentication-cookies-vs-tokens-vs-html5-web-storage/
[Ch15]	Chandru. (2015). *Enable Access-Control-Allow-Origin for multiple domains in nodejs [duplicate]*. Abgerufen am 23. Juni 2020 von https://stackoverflow.com/questions/24897801/enable-access-control-allow-origin-for-multiple-domains-in-nodejs

[Ch20]	CheatSheet Series Team. (2020). *Cross-Site Request Forgery Prevention Cheat Sheet*. Abgerufen am 17. Juni 2020 von https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html
[Cz16]	A. Czernik. (2016). *Cookies – Funktion und Aufbau einfach erklärt*. Abgerufen am 18. Juni 2020 von https://www.datenschutzbeauftragter-info.de/cookies-funktion-und-aufbau-einfach-erklaert/
[Da19]	DATACOM Buchverlag GmbH. (2019). *Framework*. Abgerufen am 26. Mai 2020 von https://www.itwissen.info/Framework-framework.html
[De17]	R. Degges. (2017). *Why JWTs Suck as Session Tokens*. Abgerufen am 09. April 2020 von https://developer.okta.com/blog/2017/08/17/why-jwts-suck-as-session-tokens
[Fi19]	W. Fiege. (2019). *Was ist ein Webserver?*. Abgerufen am 01. Juni 2020 von https://www.hosteurope.de/blog/was-ist-ein-webserver/
[Ge19]	O. Geißler, U. Ostler. (2019). *Was ist Skalierbarkeit?*. Abgerufen am 30. Juni 2020 von https://www.datacenter-insider.de/was-ist-skalierbarkeit-a-852037/
[Go12]	J. Gómez. (2012). *Serviceorientierte Architektur*. Abgerufen am 18. Juni 2020 von https://enzyklopaedie-der-wirtschaftsinformatik.de/wi-enzyklopaedie/lexikon/is-management/Systementwicklung/Softwarearchitektur/Architekturparadigmen/Serviceorientierte-Architektur
[Go19]	T. Gopal et al. (2019). *The Ultimate Guide to handling JWTs on frontend clients (GraphQL)*. Abgerufen am 31. März 2020 von https://hasura.io/blog/best-practices-of-using-jwt-with-graphql/
[Goo20]	Google. (2020). *Angular*. Abgerufen am 12. Juni 2020 von https://angular.io/
[Goo20a]	Google. (2020). *Angular Security*. Abgerufen am 12. Juni 2020 von https://angular.io/guide/security
[Ha12]	D. Hardt. (2012). *The OAuth 2.0 Authorization Framework*. Abgerufen am 20. Mai 2020 von https://tools.ietf.org/html/rfc6749

[Ho19]	Computer Hope. (2019). *Protocol*. Abgerufen am 26. Mai 2020 von https://www.computerhope.com/jargon/p/protocol.htm
[Ho20]	Computer Hope. (2020). *HTTP*. Abgerufen am 26. Mai 2020 von https://www.computerhope.com/jargon/h/http.htm
[Hsu18]	S. Hsu. (2018). *Session vs Token Based Authentication*. Abgerufen am 26. Mai 2020 von https://medium.com/@sherryhsu/session-vs-token-based-authentication-11a6c5ac45e4
[Io20]	1&1 Ionos SE. (2020). *Der HTTP-Request einfach erklärt*. Abgerufen am 15. Juni 2020 von https://www.ionos.de/digitalguide/hosting/hosting-technik/http-request-erklaert/
[Jo12]	M. Jones, D. Hardt. (2012). *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Abgerufen am 14. Juni 2020 von https://tools.ietf.org/html/rfc6750
[Jo15]	M. Jones et al. (2015). *JSON Web Signature (JWS)*. Abgerufen am 17. Juni 2020 von https://tools.ietf.org/html/rfc7515
[Jo15a]	M. Jones, J. Hildebrandt. (2015). *JSON Web Encryption (JWE)*. Abgerufen am 17. Juni 2020 von https://tools.ietf.org/html/rfc7516
[Jo15b]	M. Jones. (2015). *JSON Web Key (JWK)*. Abgerufen am 18. Juni 2020 von https://tools.ietf.org/html/rfc7517
[Jo15c]	M. Jones et al. (2015). *OAuth 2.0 Dynamic Client Registration Protocol*. Abgerufen am 14. Juni 2020 von https://tools.ietf.org/html/rfc7591
[Jo15d]	M. Jones et al. (2015). *JSON Web Token (JWT).* Abgerufen am 09. April 2020 von https://tools.ietf.org/html/rfc7519
[Ka20]	I. Kantor. (2020). *LocalStorage, sessionStorage*. Abgerufen am 12. Juni 2020 von https://javascript.info/localstorage
[Kuk16]	A. Kukic. (2016). *Cookies vs. Tokens: The Definitive Guide*. Abgerufen am 31. März 2020 von https://dzone.com/articles/cookies-vs-tokens-the-definitive-guide

[La18]	Lambros. (2018). *Difference between XSS and CSRF attacks*. Abgerufen am 01. Juni 2020 von https://medium.com/@l4mp1/difference-between-xss-and-csrf-attacks-ff29e5abcd33
[Lod13]	T. Lodderstedt et al. (2013). *OAuth 2.0 Threat Model and Security Considerations*. Abgerufen am 22. Juni 2020 von https://tools.ietf.org/html/rfc6819
[Lu18]	S. Luber, A. Donner. (2018). *Was ist HTTP (Hypertext Transfer Protocol)?*. Abgerufen am 26. Mai 2020 von https://www.ip-insider.de/was-ist-http-hypertext-transfer-protocol-a-691181/
[Lu18b]	S. Luber, A. Donner. (2018). *Was ist ein Client?*. Abgerufen am 01. Juni 2020 von https://www.ip-insider.de/was-ist-ein-client-a-728753/
[Lu18c]	S. Luber, A. Donner. (2018). *Was ist HTTPS (Hypertext Transfer Protocol Secure)?*. Abgerufen am 19. Juni 2020 von https://www.ip-insider.de/was-ist-https-hypertext-transfer-protocol-secure-a-691192/
[Me19]	mediaevent.de. (2019). *Web Storage: Local Storage, Session Storage und Datenbanken*. Abgerufen am 12. Juni 2020 von https://www.mediaevent.de/javascript/web-storage.html
[Mo20]	Mozilla Foundation. (2020). *HTTP*. Abgerufen am 09. April 2020 von https://developer.mozilla.org/en-US/docs/Web/HTTP
[Mo20a]	Mozilla Foundation. (2020). *Window.localStorage*. Abgerufen am 12. Juni 2020 von https://developer.mozilla.org/de/docs/Web/API/Window/localStorage
[Mo20b]	Mozilla Foundation. (2020). *Window.sessionStorage*. Abgerufen am 12. Juni 2020 von
[Mo20c]	Mozilla Foundation. (2020). *IndexedDB*. Abgerufen am 12. Juni 2020 von https://developer.mozilla.org/de/docs/Web/API/IndexedDB_API
[Mo20d]	Mozilla Foundation. (2020). *Using the Web Storage API*. Abgerufen am 12. Juni 2020 von https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

[Mo20e]	Mozilla Foundation. (2020). *HTTP Header*. Abgerufen am 15. Juni 2020 von https://developer.mozilla.org/de/docs/Web/HTTP/Headers
[Mo20f]	Mozilla Foundation. (2020). *Using HTTP cookies*. Abgerufen am 15. Juni 2020 von https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
[Mo20g]	Mozilla Foundation. (2020). *Cross-Origin-Resource-Sharing (CORS)*. Abgerufen am 23. Juni 2020 von https://developer.mozilla.org/de/docs/Web/HTTP/CORS
[Mo87]	P. Mockapetris. (1987). *Domain Names – Implementation and Specification*. Abgerufen am 29. Juni 2020 von https://tools.ietf.org/html/rfc1035
[Ok20]	Okta, Inc. (2020). *What is token-based authentication and who uses it?*. Abgerufen am 15. Juni 2020 von https://www.okta.com/identity-101/what-is-token-based-authentication/
[Op20]	OpenID. (2020). *Welcome to OpenID Connect*. Abgerufen am 12. Juni 2020 von https://openid.net/connect/
[OS20]	OpenSlides-Team. (2020). *OpenSlides*. Abgerufen am 31. März 2020 von https://openslides.com/de
[Owa18]	OWASP Foundation, Inc. (2018). *SameSite*. Abgerufen am 31. März 2020 von https://www.owasp.org/index.php/SameSite
[Owa20]	OWASP Foundation, Inc. (2020). *Cross Site Scripting (XSS)*. Abgerufen am 28. Mai 2020 von https://owasp.org/www-community/attacks/xss/
[Owa20a]	OWASP Foundation, Inc. (2020). *Secure Cookie Flag*. Abgerufen am 03. Juli 2020 von https://owasp.org/www-community/controls/SecureFlag
[Pa20]	A. Parecki (Okta, Inc.). (2020). *OAuth 2 Simplified*. San Francisco.
[Re01]	H. Reibold. (2001). *Hypertext Transfer Protocol*. Abgerufen am 19. Juni 2020 von https://www.tecchannel.de/a/hypertext-transfer-protocol,401210,6
[Ri17]	J. Richer und A. Sanso (Manning Publications). (2017). *OAuth 2 in Action*. New York.
[Ric20]	C. Richardson. (2020). *What are microservices?*. Abgerufen am 29. Juni 2020

	von https://microservices.io/
[Ril20]	Riley, X. (2020). *Branca*. Abgerufen am 05. Juni 2020 von https://branca.io
[Ry20]	Ryte GmbH. (2020). *URL*. Abgerufen am 27. Mai 2020 von https://de.ryte.com/wiki/URL
[Sch20]	H. Schwichtenberg. (2020). *Was ist Serialisierung?*. Abgerufen am 28. Mai 2020 von https://www.it-visions.de/glossar/alle/3469/Serialisierung.aspx
[St19]	D. Stötzel. (2019). *How To Securely Implement Authentication In Single Page Applications*. Abgerufen am 12. Juni 2020 von https://medium.com/better-programming/how-to-securely-implement-authentication-in-single-page-applications-670534da746f
[Str20]	StrongLoop, Inc. et al. (2020). *Express*. Abgerufen am 22. Juni 2020 von https://expressjs.com/de/
[Str20a]	StrongLoop, Inc. et al. (2020). *API-Referenz*. Abgerufen am 22. Juni 2020 von https://expressjs.com/de/4x/api.html
[Tu17]	M. Tuupola. (2017). *Branca as an Alternative to JWT?*. Abgerufen am 28. Mai 2020 von https://appelsiini.net/2017/branca-alternative-to-jwt/
[Tu20]	M. Tuupola. (2020). *Branca Token*. Abgerufen am 28. Mai 2020 von https://github.com/tuupola/branca-spec
[Tu20a]	M. Tuupola. (2020). *Branca Token*. Abgerufen am 18. Juni 2020 von https://branco.io
[WP19]	wp-rocket.me. (2019). *JSON Web Tokens vs. Session Cookies: What's the Difference?*. Abgerufen am 30. Mai 2020 von https://wp-rocket.me/blog/difference-json-web-tokens-vs-session-cookies/
[Ye18]	Yeebase Media GmbH. (2018). *Was ist eigentlich eine Single-Page-Webanwendung?*. Abgerufen am 18. Juni 2020 von https://t3n.de/news/single-page-webanwendung-1023623/

Anhang A: Persistieren eines Tokens

1 Persistieren in einem Cookie

In einem Cookie sind Daten als Zeichenfolgen durch ein Schlüssel-Werte-Paar gespeichert. Cookies sind browserweit gültig. Das heißt, dass sie noch vorhanden sind, nachdem ein Tab geschlossen wurde. Die Verwendung von Cookies zum Persistieren von Tokens hat allerdings zwei Nachteile:

1. Cookies sind nur gegen XSS-Attacken geschützt, wenn die Eigenschaft „HttpOnly“ gesetzt wurde. Dadurch kann ein Client allerdings nicht mehr das JWT aus einem Cookie lesen und kennt die Identität eines Benutzers nicht. (vgl. [Cha19])
2. Gegen CSRF-Attacken sind Cookies nur durch die Eigenschaft „SameSite“ geschützt. Dadurch können sie aber ausschließlich von einer Domäne gesendet werden und die Domänenunabhängigkeit von Tokens entfällt (siehe Kapitel 2.1.8). (vgl. [Kuk16])

2 Persistieren in dem Lokalen Speicher eines Browsers

Der Lokale Speicher eines Browsers (vom Englischen „Local Storage“ (vgl. [Mo20a])) speichert Dateien lokal auf einem Gerät ab. Dabei liegen Dateien domänenabhängig vor. Das heißt, dass jede Domäne einen eigenen Lokalen Speicher hat und keine Dateien abrufen kann, die von einer anderen Domäne gespeichert wurden. Dateien werden nur clientseitig durch JavaScript in den Lokalen Speicher geschrieben oder abgerufen. Außerdem sind Dateien im Lokalen Speicher solange persistiert, bis sie explizit gelöscht werden. (vgl. [Mo20a]) Der Lokale Speicher eines Browsers ist nur zum Speichern kleiner Dateien gedacht. Zudem werden Dateien aus dem Lokalen Speicher eines Browsers bei einer Anfrage nicht automatisch über HTTP zu einem Server gesendet. (vgl. [Ka20])

Aufgrund der genannten Eigenschaften sind im Lokalen Speicher abgespeicherte Dateien vor CSRF-Attacken geschützt, allerdings sind sie dann anfällig für XSS-Attacken, da die Dateien im Lokalen Speicher eines Browsers clientseitig durch JavaScript abgerufen werden.

3 Persistieren in dem Sitzungsspeicher eines Browsers

Der Sitzungsspeicher eines Browsers (vom Englischen „Session Storage“ (vgl. [Mo20b]) dient dazu, Dateien lokal im Browser zu speichern. Für Dateien im Sitzungsspeicher gelten gleiche Bedingungen wie für Dateien im Lokalen Speicher:

- Der Sitzungsspeicher ist domänenabhängig (vgl. [Mo20b])
- Dateien werden nur durch JavaScript geschrieben oder gelesen (vgl. [Mo20b])
- Dateien werden nicht automatisch bei einer Anfrage über HTTP gesendet (sondern erst, wenn sie explizit durch JavaScript abgerufen werden) (vgl. [Ka20])

Im Gegensatz zu dem Lokalen Speicher liegen Dateien im Sitzungsspeicher nur für jeweils eine Sitzung vor. Das heißt, der Sitzungsspeicher einer Domäne wird komplett gelöscht, wenn der Tab, in dem eine Anwendung geöffnet ist, oder der gesamte Browser geschlossen wird. (vgl. [Mo20b])

Ähnlich wie beim Lokalen Speicher eines Browsers sind Dateien im Sitzungsspeicher vor CSRF-Attacken geschützt, vor XSS-Attacken allerdings nicht.

4 Persistieren in der IndexedDB eines Browsers

Die IndexedDB (vgl. [Mo20c]) stellt eine Browser-interne Datenbank dar und lässt sich clientseitig per JavaScript ansprechen. Entworfen wurde IndexedDB, um große Datenmengen strukturiert zu speichern. Ähnlich wie der Lokale Speicher oder der Sitzungsspeicher ist die IndexedDB domänenabhängig. (vgl. [Mo20c]) Zudem werden bei einer Anfrage über HTTP Dateien aus der IndexedDB nicht automatisch an einen Server gesendet (vgl. [Me19]).

Aufgrund der beiden oben genannten Eigenschaften sind Dateien in der IndexedDB vor CSRF-Angriffen geschützt, da sie nicht bei jeder Anfrage automatisch gesendet werden. Sie sind jedoch nicht vor XSS-Angriffen sicher, da sie clientseitig ausgelesen werden.

5 Kein Persistieren

Die letzte Möglichkeit für das Persistieren ist ein temporäres Speichern, indem ein Token nur als Variable im Programm vorliegt und nicht persistiert wird. Dadurch liegt ein Token nur im Arbeitsspeicher. Auf diese Weise wird es nicht persistiert (vgl. [Bib20]), kann aber von einem Client für die Kommunikation mit einem Server genutzt werden. Bei jedem Abmelden eines Nutzers von einer Webseite oder Schließen des Browsertabs wird der Arbeitsspeicher geleert und mit ihm das Token gelöscht. Dennoch bietet diese Herangehensweise zwei Vorteile. Einerseits ist ein Token vor CSRF-Angriffen geschützt, da eine fremde Domäne oder fremde Webseite nicht auf Daten im Arbeitsspeicher zugreifen kann. Andererseits ist ein Token auch vor XSS-Angriffen geschützt, da bei Angular eine Manipulation oder Eingabe zur Laufzeit einer Anwendung als Zeichenfolge eingelesen wird (siehe Kapitel 2.3.1). (vgl. [Go19])

6 Ergebnis und Vorgehensweise für die Entwicklung des Authentifizierungsdienstes von OpenSlides

Cookies wären in Hinblick auf die Sicherheit ein mögliches Mittel, um ein Token sicher zu persistieren und vor Angreifern zu schützen. Allerdings ist dieser Schutz nur solange gewährt, wie die Bestandteile der Software auf der gleichen Domäne arbeiten. In Hinblick auf die Zielarchitektur, eine verteilte Architektur, der Software ist nicht sichergestellt, dass stets die-

selbe Domäne genutzt wird. (vgl. [An18]) In der Zielarchitektur liegen einzelne Komponenten verteilt vor. Deshalb kann die Eigenschaft ”SameSite” der Cookies nicht genutzt werden.

Das Ergebnis dieser Untersuchung ist, dass ein Token nur temporär im Arbeitsspeicher eines Tabs gespeichert und nicht dauerhaft persistiert wird. (vgl. [Au18])

Anhang B: Drei weitere Autorisierungsmöglichkeiten

1 Autorisierung durch einen indirekten Ablauf

Der indirekte Ablauf wurde entwickelt, damit reine JavaScript-Anwendungen, die nur im Browser ausgeführt werden, nach OAuth 2.0 autorisiert werden können (vgl. [Ri17]: S. 94). Dies ist zum Beispiel bei SinglePageApplications der Fall (vgl. [Pa20]: S. 49).

Der Zustand einer SinglePageApplication wird nicht persistiert und die Ausführung ist für den Browser ersichtlich (vgl. [Ri17]: S. 110). Deshalb kann ein Client-Geheimnis nicht sicher geheim gehalten werden (vgl. [Pa20]: S. 49).

Bei einem indirekten Ablauf wird eine Anfrage zu einem Autorisierungsserver gesendet und direkt ein Zugangstoken angefordert. Dabei findet keine Authentifizierung des Clients statt. (vgl. [Ha12]) Nach Parecki ist diese Vorgehensweise jedoch veraltet. Deshalb wurde die Autorisierung mittels Autorisierungscodes nach PKCE eingeführt. (vgl. [Pa20]: S. 57 – 60)

2 Autorisierung durch Zugangsdaten eines Benutzers

Bei einer Autorisierung durch Zugangsdaten eines Benutzers sendet ein Client eine Anfrage an einen Autorisierungsserver, um ein Zugangstoken zu erhalten. Dabei speichert der Client Zugangsdaten eines Benutzers. Durch die Zugangsdaten kann der Client den Benutzer beim Autorisierungsserver authentifizieren und sich selbst autorisieren. Deshalb eignet sich dieser Ablauf, unter anderem wenn kein anderer Ablauf verfügbar ist. (vgl. [Ha12])

3 Autorisierung durch Zugangsdaten des Clients

Bei der Autorisierung durch Zugangsdaten eines Clients ist ein Benutzer nicht involviert. Stattdessen sendet ein Client seine Zugangsdaten an den Autorisierungsserver. Dadurch authentifiziert sich der Client beim Autorisierungsserver, kein Benutzer. Auf diese Weise kann eine Autorisierung zwischen Servern stattfinden, bei der ein Benutzer unbekannt oder nicht vorhanden ist. (vgl. [Ri17]: 97 – 101)

Anhang C: Weiterführende Themen zu OAuth 2.0

1 Registrieren eines Clients

Ein Client muss bei einem Autorisierungsserver registriert sein, damit er OAuth 2.0 zur Kommunikation mit dem Autorisierungsserver nutzen kann (siehe Kapitel 3.2.4). Zur Registrierung eines Clients ist ein Standard (vgl. [Jo15c]) entwickelt worden, der ein Protokoll spezifiziert. Dafür ist bei einem Autorisierungsserver ein Registrierungs-Endpunkt definiert. Ein zu registrierender Client sendet eine Anfrage an den Registrierungs-Endpunkt, um sich zu registrieren. Der Autorisierungsserver bearbeitet die Anfrage und generiert für den Client eine einzigartige Kennung: die Client-Id sowie ein Client-Geheimnis. Die Anfrage wird über HTTP mit der POST-Methode gesendet. Der Client sendet dabei einige Informationen in einer JSON-Struktur an einen Autorisierungsserver, zum Beispiel: Name des Clients, Weiterleitungs-URL und Geltungsbereiche sowie Bewilligungstypen. (vgl. [Ri17]: S. 210) Mit den so erhaltenen Zugangsdaten, Client-Id und Client-Geheimnis, kann der Client das Framework OAuth 2.0 mit dem Autorisierungsserver nutzen (vgl. [Ri17]: S. 210ff).

2 Authentifizieren eines Benutzers durch OAuth 2.0

Das Authentifizieren eines Benutzers durch OAuth 2.0 ist nicht möglich, da OAuth 2.0 ein Autorisierungsframework ist (siehe Kapitel 1). Eine Erweiterung zum Autorisierungsframework OAuth 2.0 stellt OpenID Connect (vgl. [Op20]) dar. OpenID Connect setzt dabei an das Autorisierungsframework OAuth 2.0 an und implementiert eine Schicht über OAuth 2.0, die dazu dient, Benutzer zu authentifizieren und deren Identität abzufragen. (vgl. [Op20])

3 Proof of Possession

Proof of Possession ist eine Methode, um ein Token sowie einen Schlüssel statt eines Zugangstokens von einem Autorisierungsserver zu erhalten (vgl. [Ri17]: S. 283). Zugangstoken haben eine kurze Gültigkeitsdauer, wie bereits in vorhergehenden Kapiteln besprochen (siehe Kapitel 2.3.2). Bekommt ein Angreifer ein Zugangstoken, hat er damit innerhalb einer gewissen Zeitspanne Zugriff auf die Daten bei einem Ressourcenserver. Durch *Proof of Possession* wird dieser Fall verhindert. (vgl. [Ri17]: S. 283)

Das Konzept *Proof of Possession* (PoP) nutzt sogenannte PoP-Tokens statt „Bearer“-Tokens. Dabei erhält ein Client neben einem Zugangstoken einen Schlüssel. Ein Schlüssel kann auf zwei Weisen mit einem Autorisierungsserver ausgetauscht werden: Entweder erstellt ein Autorisierungsserver einen Schlüssel und sendet diesen mit einem Zugangstoken an einen Client oder ein Client generiert selbst einen Schlüssel und übermittelt diesen an einen Autorisierungsserver. (vgl. [Ri17]: S. 283ff)

Ein Autorisierungsserver kennzeichnet *Proof of Possession* mit dem Token-Typen „PoP“, damit ein Client erkennt, dass sowohl ein Zugangstoken als auch ein Schlüssel in der Antwort des Autorisierungsservers enthalten sind (vgl. [Ri17]: S. 287). Mit dem Schlüssel signiert ein Client ein PoP-Token ähnlich wie bei JWT (siehe Kapitel 2.1.1). (vgl. [Ri17]: S. 283ff) Bei einem PoP-Token wird die Signatur allerdings über Teile einer HTTP-Anfrage erstellt, zum Beispiel über die Methode der Anfrage oder dem Zeitpunkt, an dem sie erstellt wurde (vgl. [Ri17]: S. 287f). Dadurch ändert sich die Signatur bei jeder Anfrage und ein PoP-Token ist für einen Angreifer unbrauchbar (außer für eine einzige Anfrage), da er allein mit einem PoP-Token keine neuen, gültigen Anfragen an einen Ressourcenserver erstellen kann. (vgl. [Ri17]: S. 283 – 289)

Nach [Ri17] ist das Konzept zu *Proof of Possession* nur ein Entwurf und noch in Arbeit. Vielmehr sind „Bearer“-Tokens laut [Ri17] bisher der einzige Standard für OAuth 2.0. (vgl. [Ri17]: S. 282f) Deshalb wird in dieser Arbeit Abstand von PoP genommen und ausschließlich „Bearer“-Tokens genutzt.

Erklärung

Hiermit versichere ich, dass ich meine Ausarbeitung für das Wissenschaftliche Projekt selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Datum: 03. Juli 2020 ……

(Unterschrift)