Single Sign On bei Webanwendungen

Seminararbeit

Daniel Ebert 65926

Sebastian Scherer 65810

21. Dezember 2020

Betreuer:

Prof. Dr. Christoph Karg

Hochschule Aalen
Studiengang IT-Sicherheit

Inhaltsverzeichnis

1 | Single Sign On bei Webanwendungen

Daniel Ebert und Sebastian Scherer

1.1 Einleitung

Durch das ***Single Sign-on (SSO)*** muss sich ein Benutzer nur einmal unter Zuhilfenahme eines einzigen Authentifizierungsverfahrens identifizieren [Mar20]. Danach übernimmt der SSO-Mechanismus die Aufgabe, den Anwender zu authentifizieren und die erkannte Identität zu bestätigen [Mar20]. Dies hat den Vorteil, dass sich der Benutzer nur einmal identifizieren muss und seine Identität sicher an weitere Systeme weitergegeben werden kann, ohne dass sich dieser dort erneut anmelden muss. Dadurch hat der Benutzer Zugriff auf mehrere Systeme, wie z.B. Frontend Anwendungen oder Backend Services, bei dem die Ressourcen für die authentifizierten und autorisierten Benutzer beschränkt ist.

1.1.1 Motivation

SSO ist vor allem in Webanwendungen hilfreich, da der Benutzer dort oft mit vielen verschiedenen Systemen interagieren muss [BK16]. Ohne SSO müssten sich Benutzer bei jedem System individuell und erneut mit einem separaten Account einloggen [BK16]. Somit müsste sich der Benutzer mehrere Passwörter merken und bei jedem System einen manuellen Anmeldemechanismus durchführen. Dies senkt die psychologische Akzeptanz der Benutzer für die Sicherheit, da heutige Anmeldeprozesse Multi-Faktor Authentifizierung verwenden, welche Zeit und Aufwand benötigen.

In der Praxis wird daher in vielen Fällen dasselbe Passwort für mehrere Systeme verwendet. Werden dennoch unterschiedliche Passwörter verwendet, dann sind diese oft einfach gehalten, damit man sich alle Passwörter merken kann. Aus sicherheitstechnischer Sicht ist das nicht gut. Einfache Passwörter könnten z.B. durch Brute-Force oder durch Dictionary Attacks gebrochen werden. Immer dasselbe Passwort zu verwenden ist ebenfalls kritisch. Hat eines der Systeme einen Data Leak, dann könnten Angreifer Zugriff auf alle Systeme mit demselben Passwort erhalten. Dafür könnte zwar auch ein Password Manager verwendet werden [BK16]. Bei diesen hat man allerdings immer noch das Problem der ***Psychological Acceptability*** der Benutzer, da sich diese bei jedem System erneut registrieren und einloggen müssen. Hier schafft SSO Abhilfe.

1.1.2 Vorteile von SSO

Neben den im letzten Absatz erläuterten Vorteil der ***Psychological Acceptability*** im Hinblick auf die Benutzer hat SSO auch Vorteile für die Entwickler. Bei SSO müssen sich diese nur um ein System zur Authentifizierung kümmern [BK16]. Weiter sorgt SSO für eine simplere Administration, da alle Benutzerdaten in einem System aufbewahrt werden [BK16]. Außerdem sind die Benutzer eher dazu bereit, ein sichereres Passwort zu verwenden, wenn sie wissen, dass sie sich nur ein Passwort merken müssen.

1.1.3 Nachteile von SSO

SSO hat auch Nachteile. Zum Beispiel kann es schwierig sein, SSO in bereits existierende Systeme einzubauen [BK16]. Um dem entgegenzuwirken bieten SSO Lösungen wie ***Keycloak*** fertige Adapter für verschiedene Programmiersprachen und Systeme an, wie z.B. für ReactJS [Red20b, OpenID Connect] [Mat20]. Außerdem könnte man durch das SSO System einen ***Single Point of Failure*** haben. Jedoch kann dem entgegengewirkt werden, beispielsweise bietet Keycloak die Möglichkeit an, mehrere Keycloak Instanzen zu verwenden [Red20e]. Diese Instanzen können auf mehrere Server verteilt werden, um die Redundanz zu erhöhen.

1.1.4 Abgrenzung

Es gibt verschiedene Arten von SSO [RR12]. Diese Arbeit konzentriert sich auf SSO für Webanwendungen. Dabei interagiert der Benutzer mit webbasierten Anwendungen und Services.

Bei SSO können verschiedene Protokolle wie ***OAuth 2.0***, ***OpenID Connect(OIDC)*** oder ***SAML 2.0*** eingesetzt werden. Die OAuth 2.0-Spezifikation definiert lediglich ein Delegationsprotokoll, das für die Übermittlung von Autorisierungsentscheidungen über ein Netzwerk von webfähigen Anwendungen und APIs nützlich ist, jedoch kein Authentifizierungsprotokoll. Durch Authentifizierung soll festgestellt werden, dass der Benutzer derjenige ist, für den er sich ausgibt. Mit anderen Worten ist Authentifizierung der Nachweis einer Behauptung, wie z.B. der Identität eines Benutzers im System. In der Regel beweist er das, indem er einer Anwendung eine Reihe von Anmeldedaten, wie Benutzername und Passwort, zur Verfügung stellt. [RS17]

OAuth 2.0 allein ist kein Mechanismus, um festzustellen, wer ein Benutzer ist oder wie er sich authentifiziert hat. Es sagt nur, dass ein Benutzer eine Anwendung delegiert hat, um in seinem Namen zu handeln. OAuth 2.0 stellt diese Delegation in Form eines Access Tokens zur Verfügung, das die Anwendung verwenden kann, um im Namen des Benutzers zu handeln. Das Access Token wird der API vorgelegt, dessen Anwendung weiß, wie sie überprüfen kann, ob das Access Token valide ist. [Okt20]

Wird zum Beispiel in ein Hotel eingecheckt, erhält man eine Schlüsselkarte, mit der das zugewiesene Zimmer betreten werden kann. Die Schlüsselkarte sagt jedoch nichts darüber aus, wer der Hotelgast ist oder wie er sich an der Rezeption authentifiziert hat. Diese Karte kann für die Dauer des Aufenthalts für den Zugang zu dem Hotelzimmer verwendet werden. In ähnlicher Weise zeigt ein OAuth 2.0 Access Token nicht an, wer ein Benutzer ist. Der Token ist ein Schlüssel, welcher für den Zugriff auf Daten verwendet werden kann. [Okt20]

OIDC ist ein Authentifizierungsprotokoll, das eine Erweiterung von OAuth 2.0 ist. OIDC ist ein vollwertiges Authentifizierungs- und Autorisierungsprotokoll. OIDC macht auch starken Gebrauch von den ***JSON Web Token (JWT)***-Standards. Diese Standards definieren ein JSON-Format für Identitäts-Token und Möglichkeiten, diese Daten auf kompakte und webfreundliche Weise digital zu signieren und zu verschlüsseln [JBS15]. OIDC ist ein offener Standard, welcher von der OpenID Foundation im Februar 2014 veröffentlicht wurde [N S14a].

In dieser Arbeit wird nur das OIDC Protokoll betrachtet, da dieses im Gegensatz zu SAML 2.0 speziell für Webanwendungen entwickelt worden ist und das modernere Protokoll ist [Red20c, OpenID Connect vs. SAML]. Außerdem ist OIDC besser für HTML5 / JavaScript-Anwendungen geeignet, da es auf der Client-Seite einfacher zu implementieren ist als SAML 2.0. Da Token im JSON-Format vorliegen, sind sie mittels JavaScript leichter zu verarbeiten [Red20d].

Es gibt verschiedene Implementierungen von SSO-Systemen. Eines der bekanntesten dieser Implementierung ist Keycloak. Keycloak ist Open Source und wird von Red Hat und einer Open Source Community entwickelt. Keycloak ist eine SSO-Lösung für Webanwendungen und RESTful Webdienste, die OpenID Connect unterstützt. Keycloak bietet anpassbare Benutzeroberflächen für Anmeldung, Registrierung, Administration und Kontoverwaltung. Ebenfalls kann die Authentifizierung auch an Identitätsanbieter von Drittanbietern wie Facebook und Google delegiert werden. [Red20c]

Keycloak unterstützt sowohl OIDC als auch SAML 2.0 [Red20b]. Anwendungen, welche mit Keycloak interagieren, müssen eines der beiden Protokolle auswählen [Red20b]. Zusätzlich zu den SSO-Features implementiert Keycloak auch ***Role Based Access Control*** [Red20a]. Dabei können einem Benutzer Rollen wie z.B. ***Kunde*** oder ***Mitarbeiter*** zugewiesen werden. Basierend auf der Rolle des Benutzers hat dieser Zugriff auf andere Ressourcen. Da der Fokus dieser Arbeit auf SSO liegt, wird Role Based Access Control in der Beispielanwendung verwendet, aber nicht erläutert.

1.1.5 Vorgehen

In der ersten Sektion der Arbeit wurde SSO vorgestellt zusammen mit zugehörigen Vor- und Nachteilen, sowie der Motivation und einer Abgrenzung der verschiedenen SSO Protokolle. Die darauffolgende Sektion beschreibt den Aufbau des SSO Protokolls OIDC. In der dritten Sektion werden die Schwachstellen und passenden Gegenmaßnahmen in einem Threat Model aufgeführt. In Sektion vier wird anhand einer Beispielwebanwendung SSO praktisch gezeigt. Zuletzt wird eine Zusammenfassung dieser Arbeit gegeben.

1.2 Aufbau von OpenID Connect

Das SSO-Protokoll OpenID Connect ersetzt mehrere manuelle Authentifizierungen bei verschiedenen Service Providern durch eine einzige manuelle Authentifizierung bei einem ***Identity Provider*** [MMS16]. Ein Identity Provider verwaltet die Identitäten mehrerer Endbenutzer, bietet spezifische Authentifizierungsmechanismen (z.B. Benutzername/Passwort oder 2-Faktoren) und erstellt Authentifizierungs-Token über authentifizierte Endbenutzer [MMS16]. Das Token ist ein signiertes JSON-Web-Token. Diese Authentifizierungs-Token werden von einem ***Service Provider*** verbraucht, der dem Endbenutzer in Abhängigkeit von der Token-Verifizierung den Zugang gewährt oder verweigert [MMS16].

Der Aufbau des OIDC Protokolls wird anhand von Keycloak in den folgenden Sektionen erklärt.

1.2.1 Teilnehmer/Rollen

Die Entitäten beim OIDC können wie folgt kategorisiert werden.

Endbenutzer

Endbenutzer, oder kurz Benutzer, sind Entitäten, die in der Lage sind, sich in das Keycloak System einzuloggen [Red20c, Core Concepts and Terms]. Der Endbenutzer ist der menschliche Teilnehmer. Informationen über den Benutzer und über die Authentifizierung des Benutzers können in ***Claims*** gespeichert werden [N S14a, Terminology]. Claims sind Key-Value Paare. Diese Keys und Values

können beliebig gewählt werden. In OIDC gibt es allerdings Standardclaims wie zum Beispiel `sub`, `iss`, `email` und `address` mit einer vorgeschriebenen Beschreibung und teilweise mit einer vorgeschriebenen Funktion [N S14a, Standard Claims]. Zum Beispiel ist `sub` eine Identifikationsnummer für den Benutzer vom Issuer (`iss`) [N S14a, Standard Claims]. Ein Issuer weist eine `sub` nur einmal zu [N S14a, Claim Stability]. Aus diesem Grund können andere Teilnehmer, wie z.B. der Client, mit der Kombination von `sub` und `iss` einen Benutzer eindeutig identifizieren [N S14a, Claim Stability]. Im Beispiel in Sektion 1.4 gibt es nur einen Issuer (eine Keycloak Instanz), wodurch Benutzer allein durch `sub` eindeutig identifiziert werden können. Für andere Claims wie z.B. `email` fordert das OpenID Connect Protokoll diese Einzigartigkeit nicht [N S14a, Claim Stability].

Clients

Clients sind Entitäten, welche die Authentifizierung eines Benutzers anfordern können. Es gibt drei Arten von Clients [RS17].

Die erste Art von Client ruft andere Services im Namen des authentifizierten Benutzers auf [Har12, Sec. 1.1] [Red20c, Managing Clients]. Das sind z.B. die Frontend- oder Browser-Anwendungen, welche vollständig im Webbrowser laufen und geschützte Ressourcen vom Backend Server anfordern. Obwohl der Code für die Anwendung von einem Web-Server bedient werden muss, wird der Code selbst nicht auf dem Server ausgeführt und der Web-Server behält nichts vom Laufzeitstatus der Anwendung bei [RS17]. Stattdessen geschieht alles, was die Anwendung betrifft, auf dem Computer des Endbenutzers in dessen Speicher des Web-Browsers [RS17].

Bei der zweiten Art handelt es sich um Backend-Anwendungen, die auf einem entfernten Server ausgeführt werden und auf die über Web-Services oder ein Frontend zugegriffen wird [RS17]. Die Konfiguration der Anwendung und ihr Laufzeitzustand werden auf dem Web-Server gehalten und die Browser-Verbindung wird in der Regel über ein Session-Cookie hergestellt. Diese Clients können Ressourcen bereitstellen, welche für die authentifizierten und autorisierten Benutzer beschränkt sind und können auch als geschützter ***Ressourcenserver*** bezeichnet werden.

Die dritte Art sind native Anwendungen, die direkt auf dem Gerät des Endbenutzers laufen [RS17]. Da sich diese Arbeit auf Web SSO fokussiert, werden diese nicht behandelt, und nur der Vollständigkeit halber erwähnt.

OpenID Provider

Der ***OpenID Provider*** kann Benutzer authentifizieren und Claims der Benutzer speichern. Keycloak ist ein Beispiel für einen OpenID Provider. Ebenfalls stellen sie verschiedene Service-Endpunkte für Clients zur Verfügung [Red20c, Keycloak URI Endpoints]. Neben dem Endpunkt für die Authentifizierung eines Benutzers stellen OpenID Provider den ***UserInfo Endpoint*** bereit [N S14a, UserInfo Endpoint] [Red20c, Keycloak URI Endpoints]. Clients können dort mit einem Access Token des Benutzers alle oder einen bestimmten Teil der Claims des Benutzers abrufen [N S14a, UserInfo Endpoint]. Über ***Scopes***, ein Feld des Access Tokens, wird festgelegt, welche Claims abrufen werden können [N S14a, Requesting Claims]. Die verschiedenen Arten und der Aufbau der Tokens wird in der nächsten Sektion 1.2.2 beschrieben.

Es gibt weitere Endpunkte des OpenID Providers, wie zum Beispiel Endpunkte um ***ID***, ***Access*** und ***Refresh Tokens*** anzufordern, um einen Benutzer auszuloggen oder um die Signatur eines Token zu validieren [Red20c, Keycloak URI Endpoints]. OpenID Provider können Benutzer selbst oder über einen externen Identity Provider wie Google, Github oder Stack Overflow authentifizieren [Ide20]

[Red20c, Social Identity Providers]. Dementsprechend werden die Accountdaten lokal oder bei dem externen Identity Provider gespeichert.

Der OpenID Provider ist auch für das Verwalten der Clients verantwortlich. Zum Beispiel müssen sich Clients beim OpenID Provider registrieren, um am SSO teilzunehmen [Red20b, Client Registration] [N S14b]. Zusätzlich kann der OpenID Provider festlegen, auf welche anderen Clients, z.B. Backend Services, und auf welche Informationen und Ressourcen, z.B. Claims, ein Client Zugriff hat [Red20c, Audience Support] [N S14a, ID Token]. Das wird in der Beispielimplementierung in Sektion 1.4.1 näher erläutert.

1.2.2 Tokens

Bei SSO mit OpenID Connect werden verschiedene Arten von Tokens eingesetzt. Bei OpenID Connect haben Tokens das JWT-Format [JBS15]. Tokens werden vom OpenID Provider erstellt und mit der ***JSON Web Signature*** vom OpenID Provider signiert [Jon15]. Tokens können optional über ***JSON Web Encryption*** verschlüsselt werden [Jon15]. Da Tokens bei Webanwendungen üblicherweise über HTTPS übertragen werden und deshalb bereits verschlüsselt sind, ist eine erneute Verschlüsselung mit JSON Web Encryption dort nicht erforderlich. Der OpenID Provider kann verschiedene Arten von Token ausgeben mit jeweils verschiedenen Zielgruppen und Anwendungsfällen [N S14a, Token Endpoint].

Ein JWT setzt sich aus 3 Teilen zusammen. Im Header wird der Algorithmus genannt, welcher für die Signatur des Tokens verwendet wird [Jon15]. Der Signaturalgorithmus kann frei gewählt werden. Keycloak verwendet standardmäßig ***HMAC-SHA256***. Der Payload besteht aus einer Menge von sogenannten ***JWT Claims*** [Jon15]. Ein JWT Claim ist, wie ein Claim bei OpenID Connect, ein Key-Value Paar. Im Payload können auch OpenID Connect Claims übertragen werden. Der dritte Teil enthält die Signatur der anderen zwei Teile. Die drei Teile sind jeweils ***Base64url*** kodiert [Jon15]. In Abbildungen dieser Arbeit wird nur der dekodierte Payload gezeigt.

Im Nachfolgenden werden drei Arten von Tokens, ID Token, Access Token und Refresh Token, beschrieben. Diese drei Token werden nach einer erfolgreichen Authentifizierung eines Benutzers an den Client gesendet.

ID Token

Der ID Token enthält Claims über die Authentifizierung des Benutzers [N S14a, ID Token]. Wenn vom Client gefordert, können optional weitere Claims mit Informationen über den Benutzer enthalten sein [N S14a, ID Token]. Die folgende Auflistung zeigt ein Beispiel für ein ID Token.


```
{
        "exp": 1606058197,
        "iat": 1606057897,
        "auth_time": 1606057896,
        "jti": "c7e74c3e-dc1c-4790-bc47-4b787abe86dd",
        "iss": "http://keycloak/auth/realms/ExampleRealm",
        "aud": "frontend1",
        "sub": "85dd2d59-2ed9-407e-a745-b2f676095d4b",
        "typ": "ID",
        "azp": "frontend1",
        "nonce": "33f922fc-eacc-4ad0-81f7-580e865177df",
```

```
        "email_verified": false,
        "name": "Tom Smith",
        "preferred_username": "tom_smith",
        "given_name": "Tom",
        "family_name": "Smith",
        "email": "tomsmith@web.de"
}
```

Listing 1.1: Beispiel ID Token

Ein Beispiel Claim für die Authentifizierung des Benutzers ist `auth_time`, welches den Zeitpunkt der Authentisierung in Unixzeit (Epoch) angibt [N S14a, ID Token]. Der interessierte Leser findet eine Definition für die meisten hier gezeigten Claims unter [Ide20]. Einen Teil dieser Claims werden später noch näher erläutert, wie z.B. die `nonce` in Sektion 1.2.4 und `aud` in Sektion 1.2.2.

Der ID Token ist nur für den Client, welcher an der Authentifizierung des Benutzers beteiligt war, gedacht [Jus15]. Dieser Client kann durch den ID Token die Benutzererfahrung anpassen [Aut20, ID tokens], z.B. mit einer Willkommensnachricht mit dem Namen des Benutzers. Der ID Token sollte nicht für die Autorisierung verwendet werden, z.B. um geschützte Ressourcen von einem Backend Service abzurufen. Das ist die Aufgabe des im nächsten Abschnitt beschriebenen Access Tokens.

Access Token

Der Access Token wird für die Autorisierung des Benutzers verwendet [Har12]. Man kann ihn als Schlüssel für Ressourcen interpretieren. Er kann Informationen enthalten, die spezifizieren auf welche Ressourcen und Clients man mit dem Access Token Zugriff hat [Red20c, Audience Support]. In Web-SSO kann das Access Token z.B. als `Bearer` Token im HTTP `Authorization` Header an Backend Services enthalten sein [JH12]. Dies wird später in Sektion 1.2.4 näher erläutert.

Abgesehen von der ID des Benutzers, dem `sub` Claim, sollte der Access Token keine weiteren Informationen über den Benutzer oder die Authentifizierung des Benutzers enthalten [Ide20]. Falls ein Client trotzdem Benutzerinformationen benötigt, kann er die erlaubten Benutzer Claims mit dem Access Token bei dem UserInfo Endpunkt abfragen [N S14a, UserInfo Endpoint]. Die nachfolgende Auflistung zeigt ein Beispiel für ein Access Token.

```
{
        "exp": 1606061497,
        "iat": 1606061197,
        "jti": "2239840b-86ff-41da-8e14-7acfbf05cb35",
        "iss": "http://keycloak/auth/realms/ExampleRealm",
        "aud": [
                "backendclientservice1",
                "backendclientservice2"
        ],
        "sub": "85dd2d59-2ed9-407e-a745-b2f676095d4b",
        "typ": "Bearer",
        "azp": "frontend1",
        "nonce": "33f922fc-eacc-4ad0-81f7-580e865177df",
        "scope": "openid email profile"
}
```

Listing 1.2: Beispiel Access Token

Der `scope` Claim beschreibt sowohl welche Benutzer Claims über den UserInfo Endpunkt abgerufen werden können, als auch auf welche Ressourcen man mit dem Access Token Zugriff hat [N S14a, Requesting Claims] [Ide20]. Ein Scope kann eine Menge von Claims beschreiben. Zum Beispiel hat man mit dem `profile` Scope Zugriff auf die Benutzer Claims `name`, `family_name`, `preferred_username` und Weitere beim UserInfo Endpunkt [N S14a, Requesting Claims].

Der `aud` (Audience) Claim gibt an, für welche Clients das Access Token bestimmt ist [N S14a, ID Token]. Erhält ein Client ein Access Token und befindet sich dieser Client nicht in der `aud` Liste des Access Tokens, dann sollten keine geschützten Ressourcen an den Sender zurückgeschickt werden [Jus15].

Refresh Token

ID-, Access- und Refresh Token haben eine Verfallszeit [JBS15, Sec. 4.1.4] [N S14a, ID Token]. Diese Verfallszeit ist im `exp` (Expiration Time) Claim als Unixzeit gespeichert [JBS15, Sec. 4.1.4]. Ist ein Token abgelaufen, dann ist er nicht mehr gültig [JBS15, Sec. 4.1.4]. Speziell das Access Token hat aus Sicherheitsgründen eine kurze Verfallszeit [Okt18]. In Keycloak ist standardmäßig eine Verfallszeit von 5 Minuten für den Access Token eingestellt.

Über den Refresh Token können neue ID-, Access-, und Refresh Tokens erstellt werden ohne, dass sich der Benutzer erneut authentifizieren muss. Der OpenID Provider bietet dafür einen Endpunkt an, bei dem ein Refresh Token durch gewünschte neue Tokens mit erneuerter Verfallszeit ausgetauscht werden kann [Har12, Sec. 1.5]. Das Refresh Token kann auch verwendet werden, um einen Access Token mit weniger Rechten zu erhalten [Ami19]. Das Access Token mit weniger Rechten kann dann an potenziell weniger vertrauenswürdige Clients gesendet werden.

Das folgende Beispiel soll die Motivation für die Verfallszeit erläutern. Ein Benutzer kann sich sowohl über einen PC, als auch über einen Laptop mit demselben Account am System anmelden. Es gibt ein Refresh Token für den PC und einen für den Laptop. Falls der Laptop gestohlen werden sollte, dann kann das Refresh Token für den Laptop an einer Stelle, dem OpenID Provider, gesperrt werden. Ein Angreifer kann dann keine neuen Access Tokens mehr beim OpenID Provider anfordern. Der Benutzer kann weiterhin neue Access Tokens anfordern. Dadurch hat er über seinen PC weiterhin Zugriff auf seinen Account und kann weiter arbeiten. Ein Angreifer könnte allerdings trotzdem mit dem Access Token auf Ressourcen des Benutzers zugreifen, wenn dieser noch nicht abgelaufen ist. Aus diesem Grund muss bei der Wahl der Verfallszeit abgewogen werden. Je geringer die Verfallszeit ist, desto öfters müssen neue Tokens vom OpenID Provider angefordert werden. Dies erhöht die Last des OpenID Provider Servers. Je höher die Verfallszeit ist, desto mehr Zeit hat ein Angreifer, um möglicherweise auf geschützte Ressourcen des Benutzers zuzugreifen. Clients haben allerdings auch die Möglichkeit, Tokens beim OpenID Provider zu validieren. Dies wird in Sektion 1.2.4 näher erläutert.

1.2.3 Protokoll-Flow

Bei SSO kommen verschiedene Flows oder ***Authorization Grants*** zum Einsatz. Wie in der Abgrenzung in Sektion 1.1.4 beschrieben, gibt es verschiedene Protokolle und auch mehrere Varianten innerhalb eines Protokolls. Deshalb werden in dieser Arbeit nicht alle Protokolle und Varianten erläutert. Diese Sektion soll kurz eine Übersicht über die wichtigsten Flows von OIDC bei Webanwendungen bieten, die auf den Authorization Grant von OAuth 2.0 aufbauen.

Der ***Authorization Code Flow*** wird in der nächsten Sektion ausführlich beschrieben, da dieser am besten zum Use Case unseres praktischen Beispiels passt und der meist verwendete Flow ist. Als

Alternative zum Authorization Code Flow bietet das OpenID Connect Protokoll den ***Implicit Flow*** und den ***Hybrid Flow*** an, um einen Benutzer zu authentifizieren. Diese zwei Alternativen haben beide eine bessere Performance [Red20b, Implicit and Hybrid Flow] und eine kürzere Latenz als der Authorization Code Flow [N S20, Code Flow]. Trotzdem sollten beide, zumindest mit Keycloak, aus Sicherheitsgründen nicht verwendet werden, da bei beiden Access Tokens über den Browserverlauf geleakt werden können [Red20b] [Red20c, Implicit Flow].

1.2.4 Authorization Code Flow

Authentifizierung eines Benutzers

Der ***Authorization Code Flow*** ist der Empfohlene, um einen Benutzer zu authentisieren oder um zu überprüfen, ob der Benutzer bereits authentisiert ist [Red20c, Authorization Code Flow].

Der Flow arbeitet mit ***Uniform Resource Identifier (URI)-Redirects*** [Ami19]. Außerdem muss der Client eingehende HTTP-Anfragen vom OpenID Provider erhalten können [Ami19]. Für diese Bedingungen eignet sich z.B. ein Webbrowser. Die Abbildung 1.1 zeigt eine Übersicht des Authorization Code Flows. Der ***Authorization Server***, ***Token***- und ***UserInfo-Endpoint*** werden vom OpenID Provider bereitgestellt [Ami19] [Red20c, Keycloak URI Endpoints].

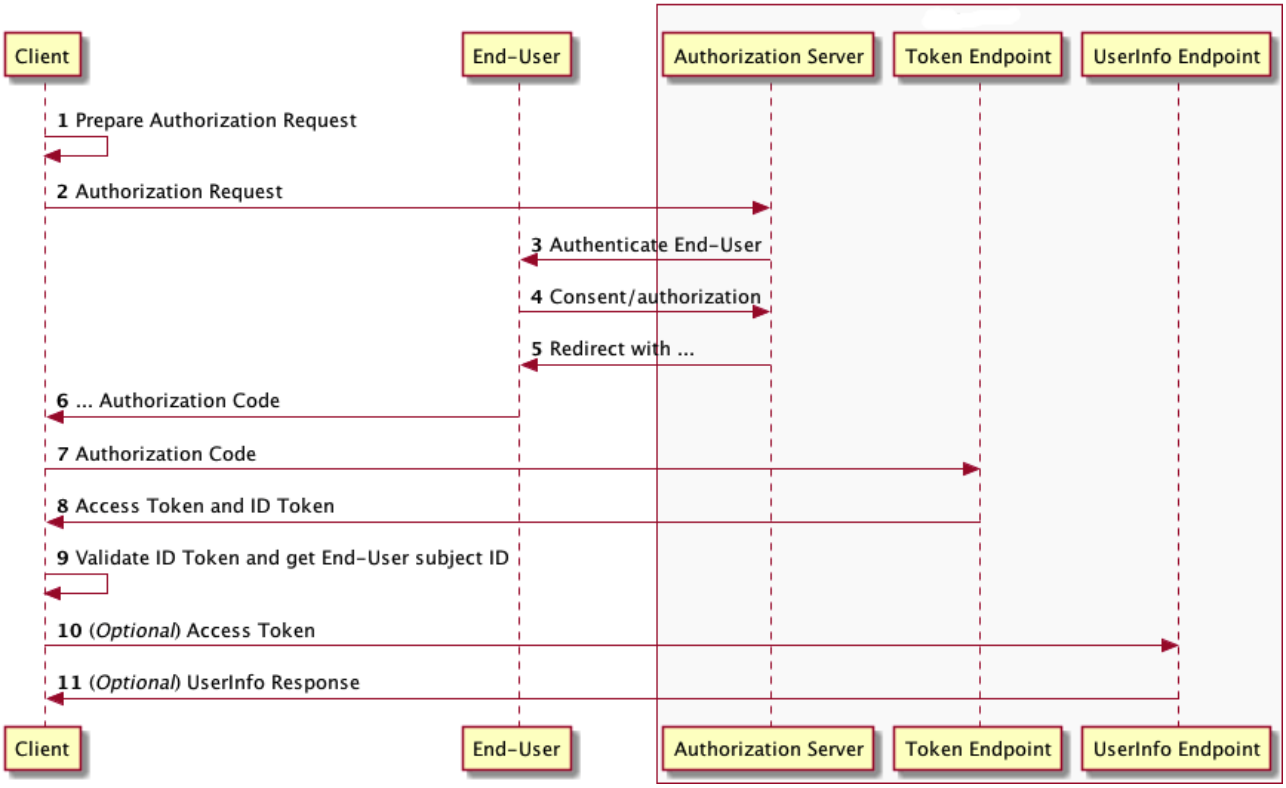

Abbildung 1.1: OIDC Authorization Code Flow Übersicht [Ami19]

Als erstes wird der ***Authorization Request*** vorbereitet. Diese Anfrage enthält mehrere Parameter im Query-String, einer davon ist die Identifikationsnummer des Clients `client_id` [N S14a, Authentication Request]. Diese ID wird bei der Registrierung des Clients beim OpenID Provider angegeben. Durch diese ID kann der OpenID Provider identifizieren, auf welche Scopes der Client Zugriff haben darf. Die Anfrage enthält auch eine Liste von gewünschten Scopes `scope`. Der

`openid` Scope muss immer in der Liste der gewünschten Scopes angegeben werden [N S14a, Authentication Request]. Dieser spezifiziert, dass es sich um eine OpenID Connect Anfrage handelt. Weiter muss eine `redirect_uri` angegeben werden. Nach erfolgreicher oder fehlgeschlagener Authentifizierung wird der Browser des Benutzers zu dieser URI geleitet [N S14a, Authentication Request]. Diese URI muss beim Registrieren des Clients mit angegeben werden. Optional kann in der Anfrage ein `state` hinzugefügt werden, dieser kann vor ***Cross-Site Request Forgery (CSRF)*** schützen [N S14a, Authentication Request]. Ebenfalls optional kann eine `nonce` angegeben werden, welche vor Replay Angriffen schützen kann [N S14a, Authentication Request]. `state` und `nonce` werden später in Schritt 6 und 8 näher betrachtet. Listing 1.3 zeigt ein Beispiel für einen Authorization Request. Über den `response_type=code` wird der Authorization Code Flow ausgewählt. Mit `response_mode=fragment` wird festgelegt, dass später ein Code per URI Fragment übertragen wird [Red20b, JavaScript Adapter Reference]. Letzteres wird später in Schritt 5 und 6 näher erläutert.

```
GET https://keycloak/auth/realms/Test/protocol/openid-connect/auth?
    client_id=frontend1
    &redirect_uri=https%3A%2F%2Ffrontend.com%2F
    &state=0909ff6a-53b2-4253-8690-aff72d2cfff1
    &response_mode=fragment
    &response_type=code
    &scope=openid%20profile
    &nonce=67bad316-d8c1-45d1-9559-2a1c4726ce91
```

Listing 1.3: Beispiel Authorization Request

Im zweiten Schritt wird die Anfrage aus Schritt 1 an den Authorization Server, also den OpenID Provider, gesendet. Bei dieser Anfrage handelt es sich um eine GET HTTP-Anfrage. Was bedeutet, dass der Benutzer nach Schritt 2 nicht auf der Website des Clients, sondern auf der Website des OpenID Provider ist. Dies ist im Hinblick auf die Sicherheit wichtig, da sich der Benutzer im dritten Schritt authentisieren muss und seine Anmeldeinformationen wie z.B. Benutzername und Passwort eingeben muss. Da dies nicht auf der Seite des Clients passiert, sind die Anmeldeinformationen des Benutzers vom Client abgeschirmt [Red20c, How Does Security Work]. Das bedeutet, dass der Client keine Informationen über die Anmeldeinformationen des Benutzers hat. Der Client erhält nur die Tokens.

OIDC schreibt nicht vor, mit welchen Methoden der OpenID Provider den Benutzer in Schritt 3 authentifiziert [N S14a, Authorization Server Authenticates End-User]. Als sicherere Alternative zu Benutzername und Passwort könnte hier z.B. auch eine Multi-Faktor-Authentifizierung eingesetzt werden. Auch die Methode, wenn der Benutzer bereits authentifiziert ist, wird vom OIDC Protokoll nicht vorgeschrieben. Zum Beispiel implementiert Keycloak das über HTTP Cookies. Nach einer erfolgreichen Authentifizierung, wird ein ID Token als `KEYCLOAK_IDENTITY` Cookie für Keycloaks Authorization Endpunkt gesetzt [Jan16]. Bei Anfragen an Keycloaks Authorization Endpunkt ist dieses Token automatisch enthalten. Der Flow für bereits authentifizierte Benutzer ist dann größtenteils gleich, wie der für nicht authentifizierte Benutzer. Allerdings wird der Benutzer dann in Schritt 3 durch das Token und nicht durch die Eingabe von z.B. Benutzername und Passwort authentifiziert.

Nach erfolgreicher Authentifizierung muss der OpenID Provider in Schritt 4 sicherstellen, dass der Client für den Zugriff auf die von ihm gewünschten Scopes berechtigt ist. Im Vorhinein kann der Administrator des OpenID Providers die erlaubten Scopes für jeden Client festlegen [Oli18]. Optional kann in diesem Schritt auch ein interaktiver Dialog mit dem Benutzer stattfinden [N S14a, Authorization Server Obtains End-User Consent]. Dabei wird dem Benutzer die Liste der gewünschten

Scopes angezeigt. Der Benutzer hat hier die Möglichkeit den Scopes zuzustimmen oder die Authentifizierung abzubrechen.

In Schritt 5 und 6 wird der Benutzer bzw. der Browser zur `redirect_uri` umgeleitet. Diese URI ist im Authorization Request von Schritt 1 spezifiziert worden. Die URI enthält einen vom OpenID Provider erstellten Code im Query-String oder im URI Fragment. Dieser Code wird Authorization Code genannt und ist für den Client opak [Sus20]. Seine Struktur wird also nur vom OpenID Provider verstanden. Der Authorization Code enthält Informationen über die Authentifizierung der vorherigen Schritte. Dazu gehören z.B. welche Scopes im Access Token enthalten sein werden. Der Authorization Code hat eine kurze Verfallszeit [Red20c, Authorization Code Flow]. Im Folgenden wird eine beispielhafte Redirect URI gezeigt.

```
https://frontend.com/#
        state=0909ff6a-53b2-4253-8690-aff72d2cfff1&
        code=50342949-85bb-47ba-84ab-ed890b088226.0d9f3aa9-9e37-4c
        b7-a589-06972b3cf410.ef40a086-7a64-4b49-b3f0-11dc5cf92a68
```

Listing 1.4: Beispiel Redirect URI

Durch das im Authorization Request aus Schritt 1 angegebene `response_mode=fragment` werden die Informationen über das URI Fragment übermittelt [Red20b, JavaScript Adapter Reference]. Der in der Redirect URI zurückgegebene `state` muss derselbe sein wie in dem vorigen Authorization Request [N S20, End-User Grants Authorization]. Das verhindert CSRF, was später in der Sektion 1.3.2 des Threat Models näher erläutert wird.

Der Authorization Code kann am Token Endpoint des OpenID Connect Providers zu einem ID, Access und optional einem Refresh Token ausgetauscht werden [N S14a, Token Endpoint]. Dies wird in Schritt 7 und 8 per HTTP POST Anfrage durchgeführt [N S20]. Um potenzielle Replay Angriffe zu verhindern, kann dieser Authorization Code nur einmal verwendet werden und hat eine sehr kurze Verfallszeit [Red20c, Authorization Code Flow].

Durch die POST Anfrage werden die Tokens im HTTP Response Body zurückgegeben. Dadurch werden sie nicht dem Browser ausgesetzt [con16] und z.B. nicht im Browserverlauf gespeichert [Red20c, Implicit Flow]. Im Gegensatz dazu wurden Informationen wie z.B. der Authorization Code dem Browser ausgesetzt. Da der Access Token eine längere Verfallszeit hat und potentiell nicht widerrufen werden kann, sollte der Browser diesen aus Sicherheitsgründen nicht speichern [Red20c, Implicit Flow].

In Schritt 8 muss das ID Token validiert werden. Dazu gehört zum Beispiel das Überprüfen der Signatur des Tokens [N S14a, ID Token Validation]. Wird im Authorization Request in Schritt 1 eine `nonce` angegeben, dann muss der ID Token einen `nonce` Claim enthalten. Die `nonce` des Authorization Request muss mit der `nonce` des ID Tokens übereinstimmen. Außerdem muss die ID des Clients im `aud` Claim des ID Tokens enthalten sein, die `iss` muss die ID des OpenID Providers enthalten und der ID Token darf nicht bereits abgelaufen sein. Der `aud` und `iss` Claim wurden in Sektion 1.2.1 erläutert. Optional können weitere Validierungen stattfinden. Zum Beispiel kann ein Token abgelehnt werden, wenn die `auth_time` zu weit in der Vergangenheit liegt. Diese und alle weiteren optionalen Validierungen werden in [N S14a, ID Token Validation] aufgeführt.

Wie bereits in Sektion 1.2.2 erläutert, kann der ID Token nicht alle Benutzer Claims enthalten. Die durch den `scope` Claim des Access Tokens zugelassenen Benutzer Claims können abschließend optional über den UserInfo Endpunkt angefordert werden.

Zugriff auf geschützte Ressourcen

Der in dieser Sektion vorgestellte Prozess ermöglicht den Zugriff auf Ressourcen in anderen Clients, welche für die authentifizierten und autorisierten Benutzer beschränkt ist. Diese anderen Clients werden dabei, wie schon erwähnt, auch Ressourcen Server genannt. Dabei wird der Access Token dem Ressourcen Server übergeben. Der Standard schreibt nicht vor, wie dieses Übergeben des Access Tokens stattfinden soll [Har12, Sec. 7]. Allerdings gibt der Standard an, dass das Access Token üblicherweise über den Authorization Header der HTTP Anfrage als sogenanntes Bearer Token übermittelt werden sollte [Har12, Sec. 7].

Bearer Token bedeutet, dass jeder, der im Besitz des Tokens ist, das Token einsetzten kann [JH12, Sec. 1.2]. Das bedeutet, dass der Ressourcen Server das Token an andere Clients senden kann, um von diesen Ressourcen abzurufen, z.B. in einer Microservice Architektur [Sti18b]. Aus diesem Grund sollte man aus Sicherheitsgründen, vor allem wenn es möglicherweise nicht vertrauenswürdige Clients gibt, die erlaubten Scopes im Access Tokens analysieren und diese so weit wie möglich beschränken. Außerdem kann, wie in Sektion 1.2.2 beschrieben ist, über den Audience Claim im Access Token angegeben werden, für welche Clients das Access Token bestimmt ist. Der Ressourcen Server sollte validieren, dass seine Client ID im Audience Claim enthalten ist, bevor er angeforderte geschützte Daten zurücksendet [Red20c, Audience Support].

Validieren des Access Tokens

Außer der im letzten Abschnitt erläuterten Validierung des Audience Claims, müssen weitere Informationen überprüft werden, bevor geschützte Daten vom Ressourcen Server herausgegeben werden. Dazu gehört die Überprüfung, ob das Token bereits abgelaufen ist [JBS15, Sec. 4.1.4]. Außerdem kann ein Ressourcen Server, basierend auf den erlaubten Scopes des Access Tokens, Entscheidungen treffen [Har12, Sec. 3.3]. Zum Beispiel kann er die Herausgabe geschützter Daten aufgrund nicht vorhandener erlaubten Scopes verweigern [Har12, Sec. 3.3].

Zusätzlich muss das Access Token validiert werden. Grundsätzlich gibt es zwei verschiedene Varianten, wie das Access Token validiert werden kann. In dieser Sektion werden diese als Online und als Offline Variante bezeichnet.

In Abbildung 1.2 ist eine Übersicht der Offline Variante gezeigt. Dabei wird der Public Key, dessen Private Key vom OpenID Provider für die Signatur des Access Tokens verwendet wurde, vom OpenID Provider abgerufen. In der Abbildung ist Keycloak der OpenID Provider und der Client wird als Service bezeichnet. Der Client kann die Signatur des Access Tokens mit dem Public Key validieren.

Bei der Offline Variante wird der Public Key oft gecached [Sti18a]. Das verringert die Latenz der Anfrage und die Auslastung des OpenID Provider Servers [Sti18a]. Allerdings kann das Access Token dann nicht widerrufen und invalidiert werden. Aus diesem Grund haben Access Tokens im OpenID Connect Protokoll eine kurze Verfallszeit. Trotzdem bleibt dabei ein Zeitraum, in dem der erlaubte Zugriff eines Benutzers durch seine bereits existierenden Access Tokens nicht eingeschränkt werden kann. Um dieses Problem zu lösen, kann die Online Variante zur Validierung verwendet werden.

Eine Übersicht der Online Variante wird in Abbildung 1.3 gezeigt. Hier wird das Access Token bei jeder Anfrage an den OpenID Provider gesendet, um ihn dort zu verifizieren. Der OpenID Provider kann dann zusätzlich überprüfen, ob sich der Benutzer ausgeloggt hat, ob der Benutzer gesperrt wurde oder ob ein Administrator des OpenID Providers alle Access Tokens bis zu einem bestimmen Zeitpunkt gesperrt hat [Sti18a]. Diese Maßnahmen greifen dann mit sofortiger Wirkung. Der Nachteil der Online Variante ist, dass jede Anfrage eine höhere Latenz hat und der OpenID Provider

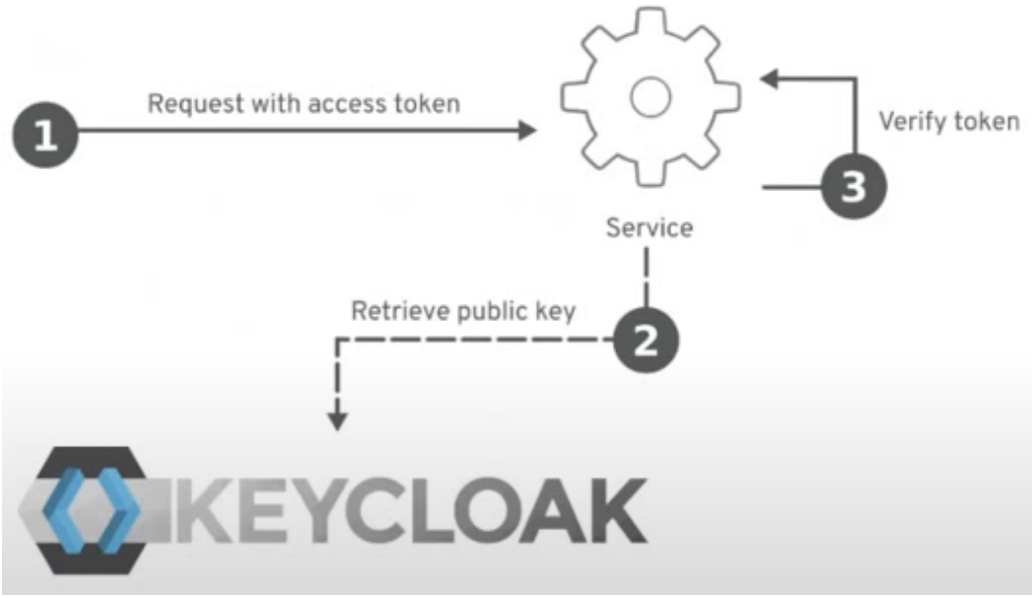

Abbildung 1.2: Access Token Verifizierung Offline Variante [Sti18a]

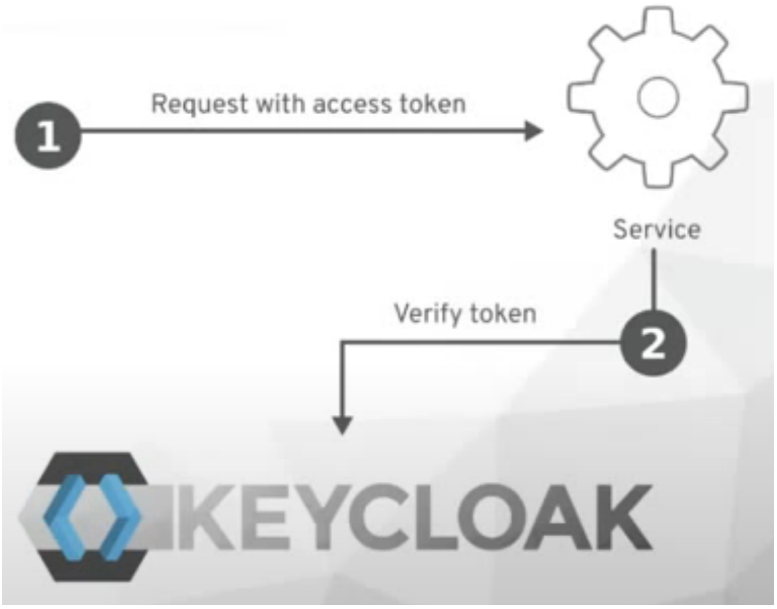

Abbildung 1.3: Access Token Verifizierung Online Variante [Sti18a]

Server eine höhere Auslastung hat. Keycloak bietet für diese Variante einen separaten Endpunkt zur Validierung an.

Den Nachteilen der Online Variante kann entgegengewirkt werden. Wie bereits in Sektion 1.1 erwähnt, bietet Keycloak die Möglichkeit an, mehrere Keycloak Instanzen zu verwenden [Red20e]. Dadurch kann die Last für den OpenID Provider auf mehrere Server verteilt werden. Darüber hinaus kann die Latenzzeit reduziert werden, indem z.B. der OpenID Provider Server und der Ressourcen Server im selben Rechenzentrum oder Rack betrieben werden.

1.3 Threat Model

In dieser Sektion werden mögliche Schwachstellen des OpenID Connect Protokolls erläutert. Weiter wird aufgeführt, wie diese mitigiert werden können. Eine Liste potenzieller Schwachstellen und was gegen sie getan werden muss, kann in dem von der IETF herausgegebenen Dokument ***OAuth 2.0 Threat Model and Security Considerations*** [LMH13] gefunden werden. Die am häufigsten vorkommenden und schwerwiegendsten Schwachstellen werden hier erörtert. [Red20d]

Ein großer Teil der Sicherheit von OpenID Connect und OAuth 2.0 basiert auf der Annahme, dass Transport Layer Security (TLS) verwendet wird, um die Kommunikation zwischen den beteiligten Parteien zu sichern [MMS16]. Es wird davon ausgegangen, dass die entsprechenden TLS-Kanäle sicher sind. Die Protokolle selbst sind frei von Kryptographie. Ebenfalls wird davon ausgegangen, dass der Endbenutzer nicht dazu verleitet werden kann vom Angreifer generierte TLS-Zertifikate als gültige Zertifikate für echte Clients zu akzeptieren [MMS16].

Obwohl OAuth 2.0 und OpenID Connect Sicherheitsprotokolle sind, garantiert ihre Verwendung allein noch keine Sicherheit. Sie müssen korrekt eingesetzt und verwaltet werden.

1.3.1 OAuth 2.0 zur Authentifizierung

Wie bereits erklärt, ist OAuth 2.0 ein fester Bestandteil für das Delegieren von Autorisierungen im Web und damit von SSO. Eine wichtige Einschränkung von OAuth 2.0 ist die Tatsache, dass es für die Autorisierung und nicht für die Authentifizierung konzipiert wurde 1.1.4. Die Verwendung von OAuth 2.0 zur Authentifizierung führt daher zu ernsthaften Schwachstellen.

Die Häufigkeit dieser Schwachstelle wurde in einer Feldstudie mit über 149 beliebten Anwendungen untersucht [Che+14]. Das Ergebnis ist, dass 89 Anwendungen (59,7%) OAuth 2.0 für Authentifizierung verwendet haben und damit anfällig sind [Che+14]. Diese Studie verdeutlicht die Schwere dieses Basis-Threats.

Eine weitere Studie, die die Häufigkeit dieser Schwachstelle unterstreicht, wurde von Zhou et al. durchgeführt [ZE14].

Daher muss OpenID Connect verwendet werden, welches auf OAuth 2.0 aufbaut, indem es Identitätsmanagement und Benutzerauthentifizierung bietet.

1.3.2 Client Schwachstellen

In dieser Sektion werden häufige Angriffen auf Clients präsentiert und praktische Möglichkeiten zur Verhinderung dieser Angriffe gezeigt.

Der Client hat Geheimnisse, wie Access und Refresh Tokens, für die er sicherstellen muss, dass diese an einem Ort aufbewahrt wird, der für Außenstehende nicht leicht zugänglich ist. Des Weiteren muss der Client darauf achten, dass diese Geheimnisse nicht versehentlich in Audit-Protokollen abgelegt werden, wo ein Dritter nach ihnen suchen könnte.

Cross-Site-Request-Forgery

Cross-Site-Request-Forgery (CSRF) tritt auf, wenn der Browser des Benutzers veranlasst wird eine unerwünschte Aktion durch eine Anfrage an eine Webanwendung auszuführen, auf der der Benutzer gerade per Cookie authentifiziert ist. Wenn ein Benutzer bei einer Anwendung angemeldet ist,

kann ein Angreifer den Browser des Benutzers manipulieren, sodass er eine Anfrage stellt, an die das Cookie automatisch angehängt wird und die Anfrage als angemeldeter Benutzer ausgeführt wird.

Um den Angriff durchzuführen, kann der Angreifer einen OAuth-Flow starten und einen Authorization Code vom Authorization Server erhalten. Der Angreifer veranlasst den Client des Opfers, den Authorization Code des Angreifers in einer bösartigen Anfrage zu verwenden, um den Code gegen einen Access Code einzutauschen. Der Ressourcenbesitzer hätte somit seine Client-Anwendung mit dem Autorisierungskontext des Angreifers verbunden. Dies hat katastrophale Folgen, wenn das OAuth-Protokoll zur Authentifizierung verwendet wird. [RS17]

Die wirksamste Mitigation ist das Hinzufügen eines unvorhersehbaren Elements oder State-Tokens in jeder Anfrage. Der Zustandsparameter des Authorization Codes kann zur Vermeidung von CSRF verwendet werden. Dieser Parameter wird vom Client bei der Initialisierung der Session erstellt und an den Authorization Server gesendet, welcher diesen zurück sendet, um den Status zwischen jeder Anfrage und Antwort aufrechtzuerhalten. Versucht nun ein Angreifer dem Benutzer eine bösartige URL unterzuschieben, wird überprüft, ob die URL dieses State-Token enthält. Ist das State-Token nicht enthalten, verwirft der Client des Benutzers die URL des Angreifers und CSRF ist nicht mehr möglich. [RS17]

Dieses State-Token wurde in Sektion 1.2.4 bereits gezeigt. Das generierte State-Token kann in einem State-Cookie gespeichert werden.

Keycloak implementiert diesen Teil der Spezifikation vollständig, so dass alle Anmeldungen geschützt sind. Zum Beispiel ist die Keycloak Admin Console eine reine JavaScript/HTML5 Anwendung, die REST-Anfragen an die Keycloak Admin REST-API des Backend macht. Diese Anfragen erfordern alle eine Bearer-Token-Authentifizierung und werden über JavaScript-Ajax-Anfragen durchgeführt. CSRF ist hier nicht anwendbar. [Red20c]

Der einzige Teil von Keycloak, der für CSRF anfällig ist, sind die Seiten zur Verwaltung der Benutzerkonten. Um dieses Problem zu entschärfen, setzt Keycloak ein Status-Cookie und bettet den Wert dieses Status-Cookies auch in versteckte Formularfelder oder Abfrageparameter in Aktionslinks ein. Dieser Abfrage- oder Formularparameter wird gegen das Status-Cookie geprüft, um zu verifizieren, dass die Anfrage durch den Benutzer erfolgt ist. [Red20c]

Registrierung der `redirect_uri`

Es ist äußerst wichtig, bei der Auswahl der registrierten `redirect_uri` besonders darauf zu achten, dass die `redirect_uri` so spezifisch wie möglich ist. Wenn die `redirect_uri` nicht spezifisch ist, werden Token-Hijacking-Angriffe möglich. Wenn es möglich ist zu allgemeine `redirect_uri` zu registrieren, ist es für einem Angreifer möglich, sich als ein Client auszugeben, welcher einen breiteren Zugriffsbereich hat [Red20c]. Dies könnte zum Beispiel passieren, wenn zwei Clients die gleiche Domain besitzen [Red20c].

Der Hauptgrund dafür ist, dass Authorization Server unterschiedliche `redirect_uri` Validierungsrichtlinien verwenden können. Die einzige zuverlässig sichere Validierungsmethode, die der Authorization Server anwenden sollte, ist die exakte Übereinstimmung [RS17]. Alle anderen Lösungen, die auf Regular Expressions basieren oder Unterverzeichnisse des registrierten `redirect_uri` zulassen, sind suboptimal und gefährlich [RS17].

Diebstahl von Tokens

Das Hauptziel eines Angreifers ist der Diebstahl eines Access Tokens. Mit dem Access Token könnte der Angreifer Operationen durchführen, für die er nicht autorisiert ist.

Access Token werden an Ressourcenserver meist durch Übergabe des Inhaber-Tokens im Header (`Authorization: Bearer <Access Token>`) gesendet, jedoch definiert RFC 6750 weitere Methoden [JH12]. Einer davon ist der URI-Abfrageparameter, der es ermöglicht Access Tokens im URI über dem Abfrageparameter `access_token` zu senden. Diese Methode kann zum Stehlen von Access Tokens verwendet werden.

Wenn ein Angreifer in der Lage ist einen Link zu dieser Zielseite zu platzieren, dann wird im Referer-Header das Access Token offengelegt, da der Referrer die gesamte URL enthält [RS17]. Die Verwendung des Authorization Headers vermeidet diese Art von Problemen, da das Access Token nicht in der URL erscheint [RS17].

XSS bei Endpunkten

Wenn eine REST-API eingesetzt wird, bei der die Antwort von Benutzereingaben abhängt, ist das Risiko, auf eine XSS-Schwachstelle zu stoßen, hoch. Ein Endpunkt kann anfällig für Cross-Site-Scripting (XSS)-Angriffe sein, wenn der Ressourcenserver `access_token` als URI-Parameter unterstützt. Dann kann der Angreifer eine URI fälschen, welche den XSS-Angriff in einem Inputwert enthält und dann Social Engineering einsetzen, um ein Opfer dazu zu bringen, diesem Link zu folgen. [RS17]

Ist der Endpunkt für XSS anfällig ist, kann ein Angreifer eine bösartige URI fälschen, die auf die geschützte Ressource verweist. Sollte das Opfer diese Aufrufen, ist der Angriff abgeschlossen, wodurch die Ausführung des JavaScripts erzwungen wird [RS17]. Der bösartige XSS-Code kann beispielsweise Daten extrahieren, um dem Angreifer die Möglichkeit zu geben, sich als authentifizierter Benutzer auszugeben.

Die Output Sanitization ist der bevorzugte Ansatz zur Verteidigung gegen XSS. Des Weiteren kann der richtige ***Content-Type*** in einer HTTP-Antwort zusätzlichen Schutz bieten. Wird beispielsweise der Content-Type `application/json` verwendet, wird die Antwort in JSON zurückgegeben und somit der XSS-Code nicht vom Browser ausgeführt. Ein weiterer nützlicher Response-Header, der von allen Browsern unterstützt wird, ist `X-Content-Type-Options: nosniff`. Dieser Sicherheitsheader verhindert, dass die Browser ***MIME Sniffing*** für eine Antwort verwenden, die vom angegebenen Content-Type abweicht. Ebenso kann der Sicherheitsheader `X-XSS-Protection` gesetzt werden, der automatisch den XSS-Filter aktiviert, welcher in den meisten neueren Webbrowsern eingebaut ist. Ein weiterer nützlicher Response-Header gegen XSS ist `Content-Security-Policy (CSP)`, der in modernen Browsern deklariert, welche dynamischen Ressourcen über einen HTTP-Header geladen werden dürfen.

Die letzte Gegenmaßnahme, die ein Client implementieren kann, um jede Möglichkeit auszuschließen, dass ein bestimmter Endpunkt für XSS anfällig ist, ist das Übergeben von Access Tokens in den Anfrageparametern nicht zu unterstützen [RS17]. XSS wäre auf dem Endpunkt theoretisch noch möglich, aber nicht ausnutzbar, da ein Angreifer keine Möglichkeit hat, eine URI zu fälschen, die auch das Access Token enthält[RS17].

Daher muss das Access Token über den Authorization Header gesendet werden. In Keycloak wird die Authentifizierung über Authorization Header erzwungen [Red20c].

Token Replays

In einer vorherigen Sektion wurde gezeigt, wie es möglich ist, ein Access Token durch XSS zu stehlen. Selbst wenn die geschützte Ressource über HTTPS läuft, kann der Angreifer, sobald er das Access Token besitzt, auf die geschützte Ressource zugreifen. Aus diesem Grund ist es wichtig, ein

Access Token zu haben, welches eine kurze Verfallszeit hat, um das Risiko des Token Replays zu minimieren. Selbst wenn es einem Angreifer gelingt, an ein Access Token zu gelangen, nimmt die Schwere des Angriffs ab, wenn dieses bereits abgelaufen ist oder kurz davorsteht. In der Sektion 1.4.1 wird gezeigt, wie die Verfallszeit des Access Tokens in Keycloak bestimmt werden kann.

OIDC und OAuth 2.0 sind frei von Kryptographie. Stattdessen verlassen sich die Protokolle vollständig auf das Vorhandensein von TLS über die verschiedenen Verbindungen hinweg. Aus diesem Grund gilt es die Verwendung von TLS im gesamten System so weit wie möglich durchzusetzen.

Dabei kann ***HTTP Strict Transport Security (HSTS)*** helfen. HSTS erlaubt es zu definieren, dass Browser oder andere konforme User-Agents nur über sichere HTTPS-Verbindungen interagieren sollten und niemals über das unsichere HTTP-Protokoll. HSTS kann über den zusätzlichen HTTP-Header `Strict-Transport-Security` hinzugefügt werden. Jedes Mal, wenn versucht wird, den Endpunkt des Client-Backends mit dem Browser über HTTP zu erreichen, würde der Browser einen Redirect zu HTTPS durchführen. Dadurch wird jede unerwartete unverschlüsselte Kommunikation (wie z.B. Protokoll-Downgrade-Angriffe) vermieden. [RS17]

Jedoch sollte nicht alleine auf HSTS vertraut werden, da es User-Agents gibt, die diesen Header nicht unterstützen und daher sollte in der Produktion nur HTTPS, beziehungsweise TLS, eingesetzt werden. In der Sektion 1.4.1 wird gezeigt, wie dies in Keycloak konfiguriert werden kann.

1.3.3 Authorization Server Schwachstellen

Der Authorization Server ist die komplexeste Komponente im SSO- und ODIC-System, und daher am schwersten zu sichern. In dieser Sektion werden dessen Schwachstellen präsentiert und gezeigt wie diese mitigiert werden müssen.

Für den Authorization Server gelten alle allgemeinen Sicherheitsmaßnahmen für den Einsatz sicherer Webserver. Besonders hervorzuheben sind Server-Logs, die Verwendung von TLS und eine sichere Hosting-Umgebung mit angemessenen Konto-Zugriffskontrollen.

Session Hijacking

In der Sektion 1.2.4 wird erklärt, dass der Client einen Redirect-Schritt unternehmen muss, um an ein Access Token vom Authorization Server zu gelangen. Dieser Redirect veranlasst den Browser, eine Anfrage an den Client zu stellen, einschließlich des Authorization Codes.

Session Hijacking kann auftreten, wenn vertrauliche Clients verwendet werden. Bei diesert Art Clients verlässt der Authorization Code den Server und gelangt zum Benutzer. Dabei verbleibt der Authorization Code in der Browser-Historie.

Gelangt nun ein Angreifer auf den Rechner des Benutzers, kann sich dieser mit seinen eigenen Anmeldedaten anmelden, manipuliert jedoch den Redirect zum Client und injiziert den Authorization Code aus der Session des Benutzers, der im Browser-Verlauf gespeichert ist. Das Resultat ist, dass der Angreifer dann Zugriff auf die Ressource des ursprünglichen Benutzers hat.

Als Mitigation gegen diese Schwachstelle empfiehlt die OAuth 2.0 Spezifikation, dass Authorization Codes nur einmal vom Client verwendet werden dürfen [D H12a]. Wenn ein Authorization Code mehr als einmal verwendet wird, muss der Authorization Server die Anfrage ablehnen und sollte alle Access Tokens zurücknehmen, die zuvor auf der Grundlage dieses Authorization Codes ausgegeben wurden [D H12a].

Diese Gegenmaßnahme wird durch Keycloak unterstützt. Ein Authorization Code kann nur einmal verwendet werden. Als zusätzliche Maßnahme ist die Verfallszeit für Authorization Codes sehr kurz - einige Sekunden, bei der der Benutzer seinen Authorization Code gegen ein Access Token eintauschen kann [Red20c, Sec. 19].

Redirection URI Manipulation

Bei der Verwendung des Authorization Code Flows 1.2.4 kann der Client eine Redirect-URI über den Parameter `redirect_uri` angeben. Wenn ein Angreifer den Wert des Redirect-URIs manipulieren kann, kann er den Authorization Server veranlassen, den Benutzer einer Ressource zu einem URI unter der Kontrolle des Angreifers mit dem Authorization Code des Angreifers umzuleiten. [D H12b]

Ein Angreifer kann einen Account bei einem Client erstellen und den Authorization Flow einleiten. Wenn der Benutzeraccount des Angreifers an den Authorization Server gesendet wird, um Zugriff zu gewähren, greift der Angreifer auf den vom Client bereitgestellten Authorization-URI zu und ersetzt den Redirect-URI des Clients durch einen URI unter der Kontrolle des Angreifers. Der Angreifer trickst dann den Benutzer einer Ressource aus, dem manipulierten Link zu folgen, um den Zugriff auf den Client zu autorisieren. [D H12b]

Am Authorization Server wird dann dem Opfer über einen Client eine gültige Anfrage gestellt, welche dieser autorisiert. Das Opfer wird dann an einen Ressourcen-Endpunkt unter der Kontrolle des Angreifers mit dem Authorization Code umgeleitet. Der Angreifer schließt den Authorization Flow ab, indem er den Authorization Code an den Client, unter Verwendung der ursprünglichen Redirect-URI, sendet. Der Client tauscht den Authorization Code mit einem Access Token und verknüpft es mit dem Client-Account des Angreifers, welcher nun Zugang zu den geschützten Ressourcen erhalten kann. [D H12b]

Um dieser Schwachstelle entgegenzuwirken, stellt der Authorization Server sicher, dass der Redirect-URI, der zum Erhalt des Authorization Codes verwendet wird, mit dem Redirect-URI identisch ist, welcher beim Austausch des Authorization Codes gegen ein Access Token bereitgestellt wird [D H12b][Red20c, Sec. 19]. Der Authorization Server muss von Clients verlangen, ihre Redirect-URIs zu registrieren [D H12b]. Wenn in der Anfrage ein Redirect-URI angegeben wird, muss der Authorization Server diesen gegen den registrierten Wert der Redirect-URI validieren [Red20c, Sec. 19].

Open Redirector

Der Authorization Server und der Client-Redirect-Endpunkt können so konfiguriert sein, sodass sie als `offene Redirector` funktionieren. Ein offener Redirector ist ein Endpunkt, der einen Parameter verwendet, um einen Benutzer ohne jegliche Validierung automatisch an den durch den Parameterwert angegebenen Ort umzuleiten. [D H12b]

Offene Redirects können bei Phishing-Angriffen verwendet werden oder von einem Angreifer, um Benutzer dazu zu bringen, bösartige Seiten zu besuchen, indem die URI eines bekannten Ziels verwendet wird. [D H12b]

Wenn der Authorization Server dem Client erlaubt, nur einen Teil der Redirect-URI zu registrieren, kann ein Angreifer darüber hinaus einen vom Client betriebenen offenen Redirector verwenden, um eine Redirect-URI zu konstruieren, die die Authorization Server-Validierung umgeht und Authorization Codes oder Access Tokens auf einen Endpunkt unter der Kontrolle des Angreifers sendet. [D H12b]

Daher erfordert Keycloak, dass alle Clients mindestens ein Redirect-URI-Muster registrieren müssen. Jedes Mal, wenn ein Client Keycloak um eine Umleitung anfragt, überprüft Keycloak diesen Redirect-URI anhand der Liste der gültigen registrierten URI-Muster. Um offene Redirect-Angriffe abzuschwächen, müssen Clients möglichst spezifische URI-Muster verwenden. [Red20c, Sec. 19]

1.3.4 Token Schwachstellen

Die Schwachstellen der vorausgegangenen Sektionen hatten es meist zum Ziel ein Access Token oder einen Authorization Code zu erhalten. In dieser Sektion soll gezeigt werden welche Schwachstellen der Umgang mit den Tokens und Codes hat und wie diese Risiken minimiert werden können.

Die OAuth 2.0 und OIDC-Spezifikationen verwenden Bearer-Token, sodass jeder, der im Besitz des Tokens ist, dieses verwenden kann, unabhängig davon wer er ist. Diese Bearer verwenden alle Klartextzeichen und benötigen daher TLS als Grundlage für die Sicherheit. [D H12b]

In den Sektionen der vorherigen Sektionen wurde präsentiert, wie Access Tokens oder Authorization Tokens aus Clients gestohlen werden können. Bearer-Token für Clients sind transparent, damit keine kryptographischen Operationen durchgeführt werden müssen. Gelangt ein Angreifer an ein Token, ist er in der Lage, auf alle Ressourcen zuzugreifen, die in den Anwendungsbereich dieses speziellen Tokens fallen. Abgesehen vom Hijacking, welches in den vorherigen Sektionen bereits behandelt wurde, gibt es die folgenden Bedrohungen. Diese sind in vielen anderen tokenbasierten Protokollen ebenfalls zu finden:

- ***Token Modification***: Ein Angreifer kann ein gefälschtes Token herstellen oder ein vorhandenes gültiges Token modifizieren, wodurch der Client-Ressourcenserver veranlasst wird, dem Client unangemessenen Zugriff zu gewähren. Beispielsweise kann ein Angreifer ein Token herstellen, um Zugang zu Informationen zu erhalten, die er zuvor nicht einsehen konnte. Alternativ könnte die Gültigkeit des Tokens selbst verlängert werden. [JH12]
- ***Token Replay***: Es wird ein altes, abgelaufenes Token wiederverwendet, um an valide Daten vom Ressourcenserver zu gelangen. Der Ressourcenserver sollte in diesem Fall einen Fehler zurückgeben. [JH12]
- ***Token Redirect***: Ein Angreifer verwendet ein Token, das für einen bestimmten Ressourcenserver generiert wurde, um Zugriff auf einen anderen Ressourcenserver zu erhalten Hierbei glaubt das Opfer fälschlicherweise, dass das Token für es gültig ist. [JH12]
- ***Token Disclosure***: Ein Token könnte sensible Informationen über das System enthalten und der Angreifer könnte an Informationen gelangen, für die er nicht berechtigt ist. [JH12]

Eine Gegenmaßnahme, die ein Client anwenden kann, besteht darin, den Umfang des Tokens auf das für seine Aufgaben erforderliche Minimum zu beschränken [D H12b]. Dieser Ansatz der minimalen Privilegien schränkt ein, wofür das Token verwendet werden kann, wenn es erfasst wird. Um die Auswirkungen auf die Benutzererfahrung so gering wie möglich zu halten, kann ein Client während der Autorisierungsphase nach allen geeigneten Bereichen fragen und dann das Refresh-Token verwenden, um Access Token mit eingeschränktem Bereich zu erhalten, mit denen die Ressource direkt aufgerufen werden kann [D H12b]. Diese Gegenmaßnahme wird insofern durch Keycloak unterstützt, dass in Umgebungen zwischen den einzelnen Services, unter denen das Vertrauen gering ist, der Empfängerkreis der Tokens limitiert werden kann [Red20c, Sec. 19]. Dies kann sowohl für die Scopes, als auch für die Audience konfiguriert werden [Red20c, Sec. 19].

Um Angriffe zu minimieren, die sich aus den Injektionen in den Speicher ergeben, ist das Token im Transient Memory zu belassen. Browser löschen dann das Token, wenn der Browser vom Benutzer geschlossen wird.

Eine große Bandbreite von Bedrohungen kann durch den Schutz der Inhalte des Tokens abgemildert werden, durch Verwendung einer digitalen Signatur oder eines Message Authentication Codes (MAC) [JH12]. Der OAuth 2.0 Standard spezifiziert weder die Kodierung noch den Inhalt des Tokens, diese müssen selbst bestimmt werden. Der Schutz der Token-Integrität muss ausreichend sein, um eine Veränderung des Tokens zu verhindern [JH12]. In Keycloak werden JWTs verwendet, welche signiert werden [Red20c, Sec. 19].

Um Token Redirects zu mitigieren, ist es wichtig, dass der Authorization Server die Identität der vorgesehenen Empfänger in das Token aufnimmt [JH12][Red20c, Sec. 19]. In Keycloak wird durch den sub-Claim die Benutzer-ID mitaufgenommen.

Gegen Token Replay muss die Verfallszeit des Tokens gering sein [JH12]. Eine Möglichkeit, dies zu erreichen, besteht darin, ein Gültigkeitszeitfeld in den geschützten Teil des Tokens zu setzen. Je kürzer die Verfallszeit ist, desto geringer sind die Auswirkungen. Empfohlen wird eine Stunde oder weniger [JH12]. Da in Keycloak JWTs eingesetzt werden, haben die Tokens automatisch eine Ablaufzeit, welche durch den Keycloak Admin gesetzt werden kann. Des Weiteren können alle Tokens bei einer Kompromittierung widerrufen werden.

1.4 Umsetzung von SSO innerhalb einer Beispielwebanwendung

Abbildung 1.4 zeigt einen Überblick über die Architektur und die Flows der implementierten Beispielwebanwendung. Jedes Rechteck stellt einen Microservice dar. Jeder Microservice läuft in einem separaten Docker Container. Pfeile stellen Flows bzw. Interaktionen zwischen den Services dar. Die Richtung der Pfeile gibt an, von welchem Service der Flow ausgeht.

Die Namen und Funktionsweisen der Flows in Abbildung 1.4 korrespondieren mit den in Sektion 1.2.3 beschriebenen Flows. Frontend 1 und 2 sind die in Sektion 1.2.1 beschriebenen Clients der ersten Art. Diese können mit dem Authorization Code Flow einen Benutzer authentifizieren. Dabei kommt Keycloak als OpenID Provider zum Einsatz. Keycloak speichert die Konfiguration der Clients und die Daten der Benutzer in einer Postgres Datenbank (DB) ab. Backend 1 und 2 nehmen am SSO Teil, stoßen selbst allerdings keinen Authentifizierungsflow an. In Sektion 1.2.1 sind das die Ressourcenserver, also Clients der zweiten Art. Die Frontend Clients können HTTP Post Anfragen an die Backend Clients senden. In diesen Anfragen kann ein Access Token enthalten sein. Dieser Access Token wird von den Backend Clients beim OpenID Provider Keycloak validiert. Bei erfolgreicher Validierung senden die Backend Clients die Identifikationsnummer des authentifizierten Benutzers, den 'sub' Claim, mit HTTP Status Code 200 zurück.

Alle Docker Container werden mit Docker-Compose gestartet und konfiguriert. Docker-Compose ist ein Tool für Multi-Container-Anwendungen [Aan20]. Über Docker-Compose wird ein Netzwerk erstellt und konfiguriert. Dadurch können die Komponenten untereinander kommunizieren. Außerdem kann der Benutzer dadurch über den Browser mit den Komponenten kommunizieren. Über Docker-Compose wird ebenfalls ein persistenter Speicher für die Komponente DB (Datenbank) aufgesetzt und Umgebungsvariablen für die Anwendungen in den Containern gesetzt.

In Abbildung 1.5 ist die Struktur des src Ordners zu sehen, welcher die von uns erstellen Dateien dieser Beispielwebanwendung enthält. Dieser src Ordner ist Teil des Git Repositories, welches un-

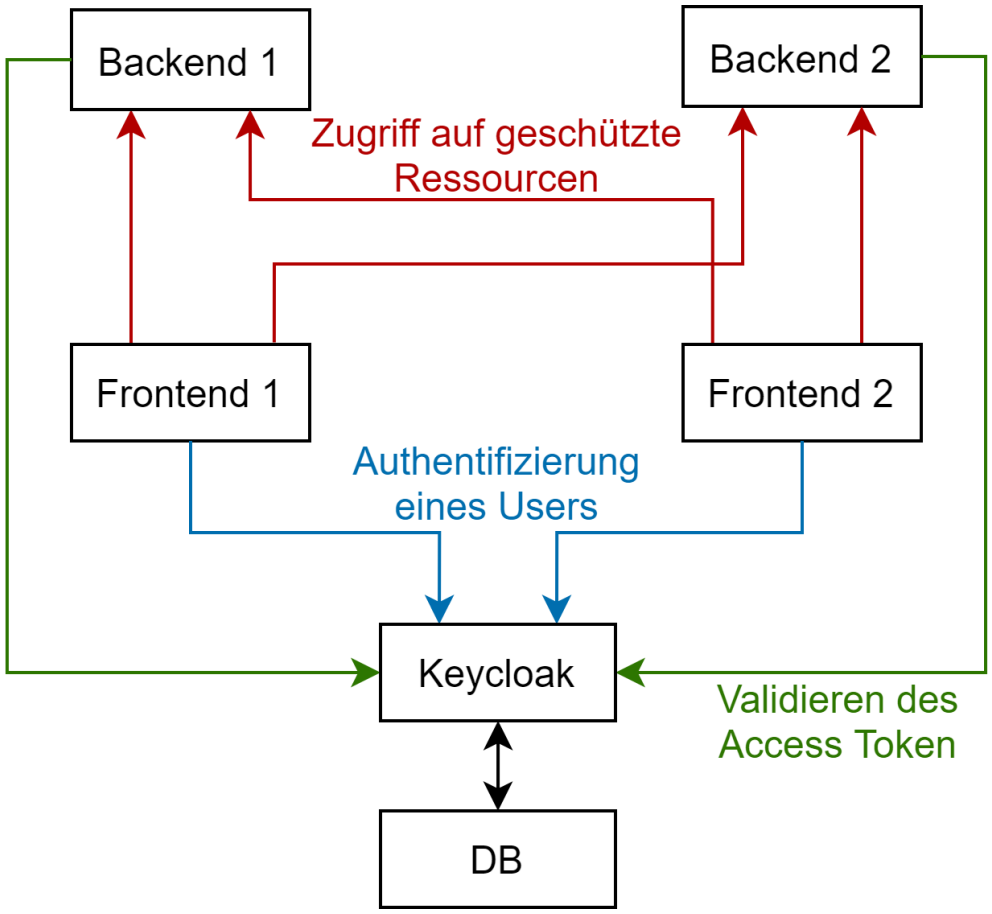

Abbildung 1.4: Überblick der Beispielwebanwendung

```
sichere_webanwendungen/src
├── README.md            # Startvoraussetzungen und wie man das Beispiel startet
├── docker-compose.yml   # Docker-Compose Konfiguration (Netzwerk, ...)
├── .env                 # Umgebungsvariablen für Keycloak und DB Container
├── postgres_data        # Dateien der Postgres Datenbank
│   ├── ...
├── frontend1            # src Dateien der Frontend 1 Applikation
│   ├── Dockerfile
│   ├── README.md
│   └── src
│       ├── App.js
│       ├── ...
├── frontend2            # src Dateien der Frontend 2 Applikation
│   ├── Dockerfile
│   ├── README.md
│   └── src
│       ├── App.js
│       ├── ...
└── service              # src Dateien der Backend 1 und 2 Services
    ├── Dockerfile
    ├── requirements.txt
    └── service.py
```

Abbildung 1.5: Struktur des Git-Verzeichnisses

ter [Seb20a] heruntergeladen werden kann ist. Dort ist in der Datei 'src/README.md' [Seb20b] die Anleitung zum Installieren und Starten dieser Beispielwebanwendung, sowie die Anmeldeinformationen für die von uns erstelle VMWare VM, die Keycloak Admin Console und einen registrierten Beispielbenutzer der Webanwendung. In der VMWare VM ist, wie in der 'src/README.md' näher erläutert, dieses Repository bereits heruntergeladen und die erforderlichen Befehle zur Installation wurden bereits ausgeführt. In der VM kann die Beispielwebanwendung mit 'cd /home/user/sichere_webanwendungen/src && docker-compose up' gestartet werden.

Die `docker-compose.yml` Datei enthält die Konfiguration für Docker-Compose. In `.env` wird der Pfad zum `postgres_data` Ordner gesetzt, sowie das Adminpasswort für Keycloak. Keycloak ist in dieser Beispielwebanwendung bereits konfiguriert. Zum Beispiel sind Frontend 1 und 2 bereits in Keycloak als Clients eingetragen.

Diese Konfiguration ist in der Datenbank bzw. dem `postgres_data` Ordner gespeichert. Die Ordner `frontend1` und `frontend2` enthalten den Source Code für die Frontend Applikationen. Die zwei Frontend Applikationen sind außer verschiedener Farben im HTML identisch. Backend 1 und 2 sind ebenfalls identisch. Ihr Source Code ist im Ordner `service` zu finden.

Für Keycloak und die Datenbank waren bereits Docker Container Images vorhanden. Für die Frontend und Backend Komponenten haben wir neue Docker Container Images erstellt. Diese haben wir im Dockerhub unter [Dan20] hochgeladen. Das Dockerfile zum Bauen des jeweiligen Image ist im Source Folder der jeweiligen Komponente. Die Images können, müssen aber nicht lokal gebaut werden. Mit Docker-Compose werden diese automatisch von Dockerhub heruntergeladen. Die Anleitung zum lokalen Bauen der Container Images ist ebenfalls in der src/README.md Datei des Repositories [Seb20b].

Im src Ordner des geklonten Git Repositories [Seb20a] kann der Befehl `docker-compose up` ausgeführt werden, um alle Container zu starten. Nach circa einer Minute ist Frontend 1 im Browser unter `localhost:3000` und Frontend 2 unter `localhost:3001` erreichbar. Im Folgenden wird eine Übersicht der Implementierung und Konfiguration der Komponenten und Flows gezeigt, sowie näher erläutert. Die genaue Dokumentation zur Implementierung des Systems kann im Code, z.B. im Git Repository [Seb20a] oder in der VM, über DocStrings, Kommentaren und einer Dokumentation nachgelesen werden.

1.4.1 Keycloak

Der Keycloak Authorization Server ist unter `localhost:8080` erreichbar. Die Security Admin Console ist eine GUI zum Konfigurieren von Keycloak für Keycloak Administratoren. Über die URL `localhost:8080/auth/admin/` ist diese GUI erreichbar. Im Beispiel kann man sich dort mit dem Benutzernamen `admin` und Passwort `iuq123` anmelden. Als Alternative zur GUI kann Keycloak auch über die sogenannte Admin CLI konfiguriert werden.

Realms

Keycloak hat ein Konzept namens ***Realms***. Jeder Realm hat eine Menge von Clients und Benutzern. Ein Realm ist wie eine Sandbox. Die Clients und Benutzer eines Realms sind isoliert von Clients und Benutzern anderer Realms [Red20c, Core Concepts and Terms].

Beim ersten Start von Keycloak erstellt Keycloak einen vordefinierten Realm. Dieser wird ***master*** Realm genannt. Der ***master*** Realm enthält unter anderem den Keycloak Administrator-Benutzer. Außerdem fügt Keycloak die Security Admin Console und Admin CLI automatisch als Clients dem

master Realm hinzu. Aus Sicherheitsgründen sollten weitere Benutzer und Clients in einem neuen Realm konfiguriert werden, damit versehentliche Änderungen der Realmeinstellungen weniger schwerwiegend sind [Red20c, Master Realm]. Es können auch weitere Administrator-Benutzer für Realms erstellt werden. Administrator-Benutzer im Master Realm haben Zugriff auf alle Realms, Administrator-Benutzer in anderen Realms haben nur Zugriff auf ihren eigenen Realm [Red20c, Master Realm].

Für das Beispiel wurde ein neuer Realm mit dem Namen `Test` erstellt. In der Security Admin Console ist der Test Realm standardmäßig ausgewählt. Dort kann in der linken Navigationsleiste unter ***Realm Settings*** der ausgewählte Realm konfiguriert werden. Zum Beispiel kann dort im Tokens Tab die Verfallszeit des Access Tokens gesetzt werden. Weiter kann im Login Tab eingestellt werden, dass alle Anfragen an Keycloak über TLS verschlüsselt werden müssen.

Clients

In der linken Navigationsleiste der Security Admin Console können unter ***Clients*** alle Clients des Realms angezeigt werden und neue Clients erstellt werden. Im Beispiel wurden die folgenden vier Clients erstellt: `frontend1`, `frontend2`, `service1`, `service2`. Die Konfiguration der Clients kann unter `Edit` angezeigt werden.

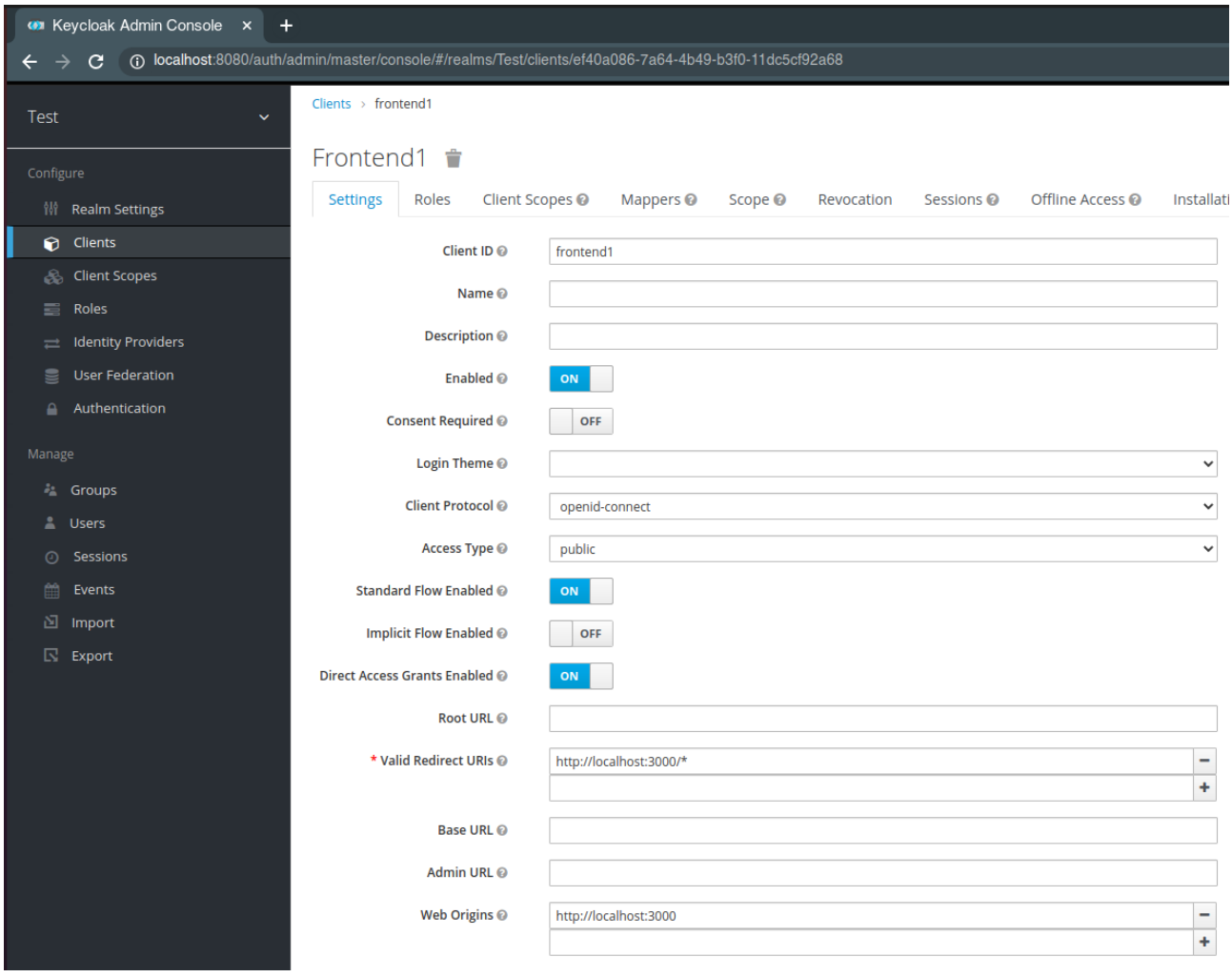

Abbildung 1.6: Keycloak `frontend1` Client Konfiguration

In Abbildung 1.6 ist die Konfiguration des `frontend1`-Clients gezeigt. Die folgende Auflistung beschreibt wichtige Felder in Abbildung 1.6:

- ***Client ID:*** `frontend1` ist die Client ID für Frontend 1.
- ***Consent Required:*** Setzt fest, ob der Benutzer den von Clients gewünschten Scopes zustimmen muss. Das ist der im Authorization Code Flow in Sektion 1.2.4 beschriebene Schritt 4.
- ***Client Protocol:*** Legt das Protokoll OIDC fest, welches der Client bei der Kommunikation mit Keycloak verwendet.
- ***Access Type:*** In Sektion 1.2.1 wurden zwei Arten von Clients für Web SSO beschrieben. Der Wert ***public*** ist für Clients der ersten Art, die Frontend Applikationen. Backend 1 und 2 haben die Client ID `service1` und `service2`. Diese sind Clients der zweiten Art. Bei diesen ist der Access Type auf `bearer-only` gesetzt. Dadurch können die Backend Services keinen Authorization Code Flow anstoßen.
- ***Valid Redirect URIs:*** Hier können möglichst präzise Validierungsrichtlinien für Redirect URIs gesetzt werden, nach 1.3.2.
- ***Web Origins:*** Dadurch ist für den Client der HTTP Header `Access-Control-Allow-Origin: http://localhost:3000` in Nachrichten von Keycloak zum Client enthalten. Dieses Feld muss auf die URL des Frontends gesetzt werden, um CORS zu erlauben. Ansonsten akzeptiert der Browser eingehende Anfragen bzw. Redirects von Keycloak nicht.

Client Scopes

In der linken Navigationsleiste der Security Admin Console kann auf die Keycloak Client Scopes zugegriffen werden. Für ein Keycloak Client Scope können beliebig viele sogenannte ***Protocol Mapper*** definiert werden. Mit Protocol Mappern können die Claims von z.B. dem ID- und Access Token und die zurückgegebenen Claims des UserInfo Endpunkts konfiguriert werden. Dazu gehört unter anderem das Hinzufügen weiterer Scopes im `scopes` Claim und weiterer Audiences im `aud` Claim des Access Tokens. Es können auch neue Claims erstellt werden, welche dem ID- oder Access Token hinzugefügt oder vom UserInfo Endpunkt zurückgegeben werden. Über Protocol Mapper können Benutzern auch Rollen für Role Based Access Control zugewiesen werden. Wie in der Abgrenzung in Sektion 1.1.4 erläutert, wird Role Based Access Control in dieser Arbeit jedoch nicht näher betrachtet.

Im Folgenden werden zwei Beispiele für Keycloak Client Scopes gezeigt. Im ersten Beispiel werden die zwei Backend Clients mit Client ID `service1` und `service2` dem `aud` Claim des Access Tokens hinzugefügt. Unter `Client Scopes` und `Create` wird dafür, wie in Abbildung 1.7 gezeigt, ein neuer Client Scope erstellt mit dem Namen `backend_audience`. Ein wichtiges Feld in 1.7 ist `Include In Token Scope`. Ist dieses Feld `ON`, dann ist der Name des Client Scopes im `scopes` Claim des Access Tokens enthalten. In diesem Fall ist das Feld `OFF`, da nur der `aud` Claim angepasst werden soll.

Für den Client Scope werden im `Mappers` Tab über `Create` zwei Protocol Mapper vom Typ `Audience` erstellt, jeweils einen Protocol Mapper für jeden Backend Client. Die Konfiguration für den ersten Backend Client wird in Abbildung 1.8 gezeigt. Im `Included Client Audience` Feld kann aus einer Liste aller Clients der Client ausgewählt werden, welcher dem `aud` Claim hinzugefügt werden soll. Für den zweiten Protocol Mapper ist an dieser Stelle der Client mit der ID `service2` eingetragen.

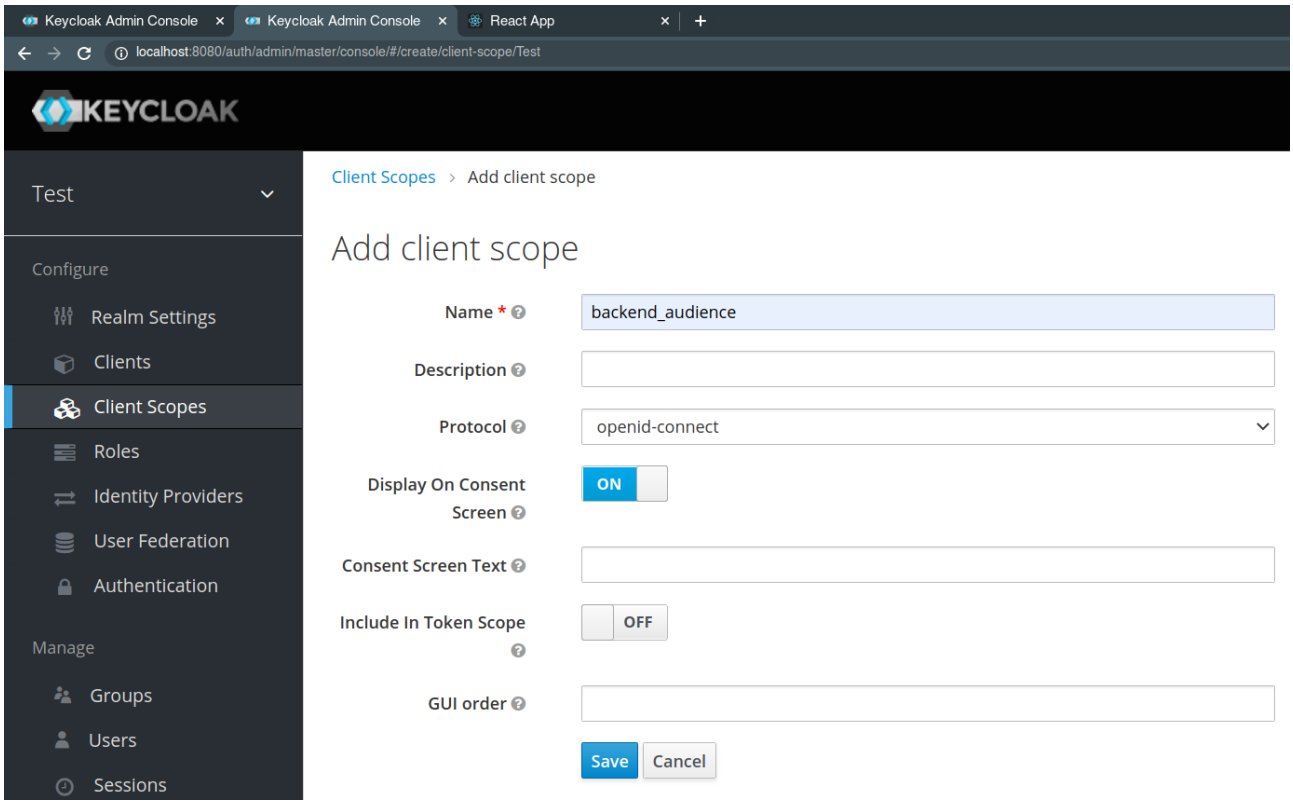

Abbildung 1.7: Keycloak Client Scope für Audience Claims

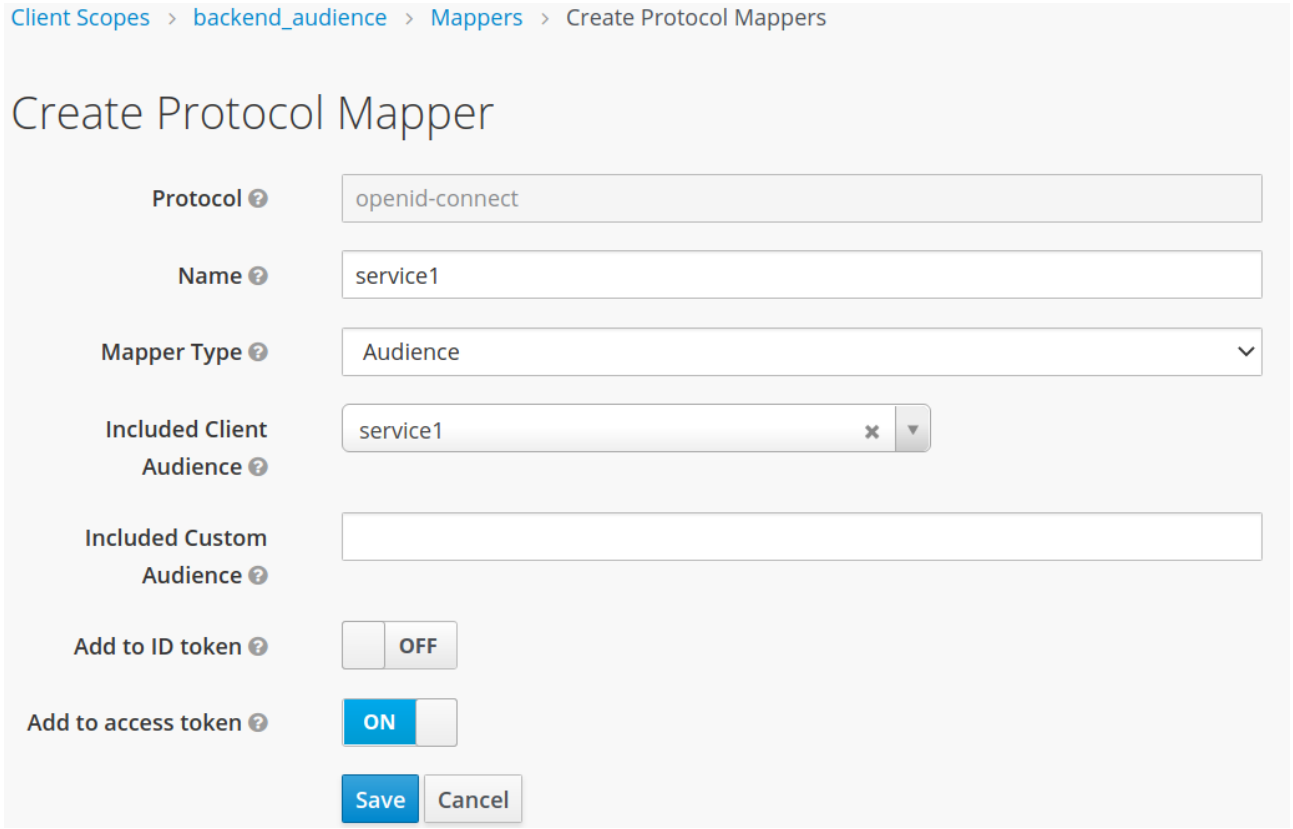

Abbildung 1.8: Keycloak Protocol Mapper für Audience Claims

Über die `Add to ID token` und `Add to access token` Felder wird festgelegt, dass die Änderung nur beim aud Claim des Access Tokens stattfinden soll.

Abschließend kann der Client Scope einem oder mehreren Clients hinzugefügt werden. Die im Client Scope definierten Änderungen werden bei den ausgewählten Clients angewandt. In diesem Beispiel sollen die aud Claims der Access Tokens an die zwei Frontend Applikationen angepasst werden. In Abbildung 1.9 wird das Hinzufügen des `backend_audience` Client Scopes zum `frontend1` Client gezeigt. Im `Client Scopes` Tab des `frontend1` Clients wird dazu der `backend_audience` Client Scope ausgewählt. Dieser kann dann mit ***Add selected*** der Liste der zugewiesenen Standard Client Scopes für den Client hinzugefügt werden.

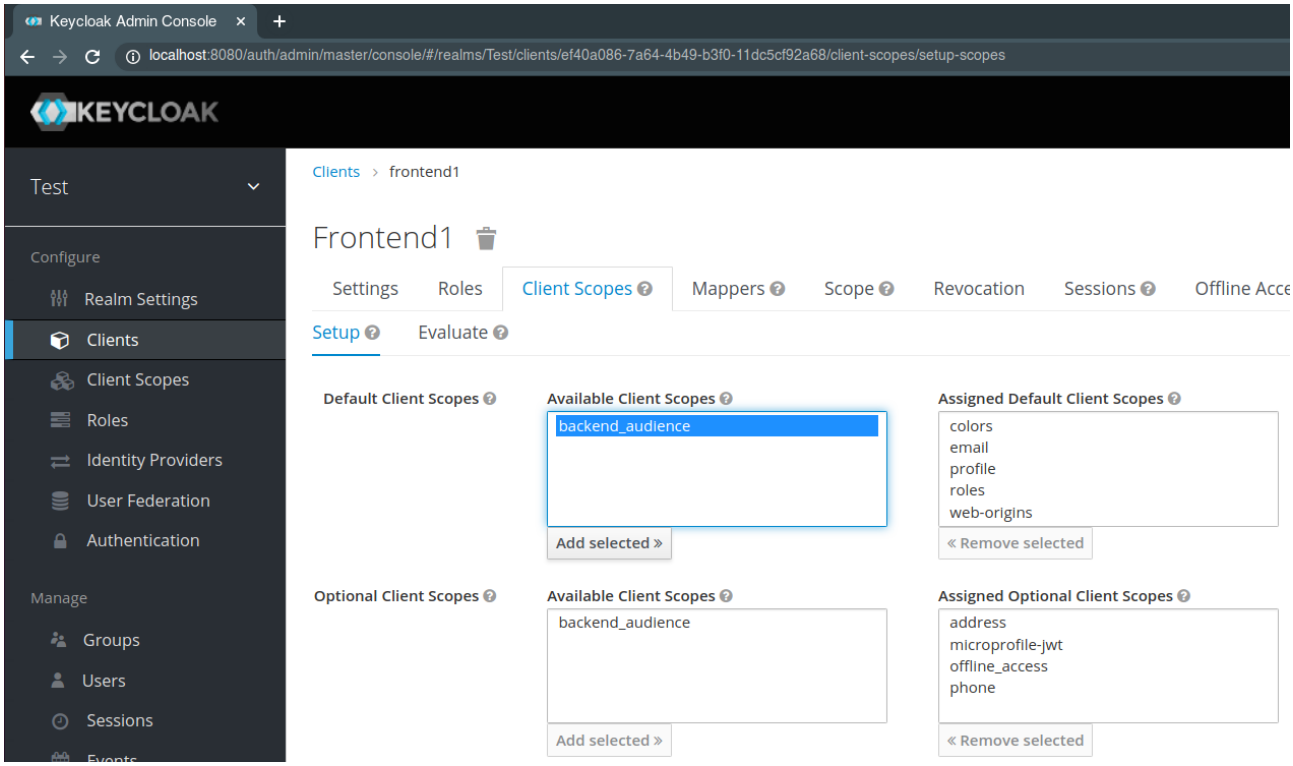

Abbildung 1.9: Zuweisung eines Keycloak Client Scopes zu einem Client

Im zweiten Beispiel soll ein neuer Claim `GUI_Color` erstellt werden. Dieser Claim soll im ID Token enthalten sein und vom UserInfo Endpunkt zurückgegeben werden. Die `GUI_Color` soll nicht bei allen Benutzern gleich sein, sondern sie soll der Lieblingsfarbe des Benutzers entsprechen.

In Keycloak können Benutzer Attribute haben. Attribute sind Key-Value Paare. Diese können z.B. in der Security Admin Console, der Admin CLI oder beim Registrieren des Benutzers gesetzt werden. In Abbildung 1.10 wurde in der linken Navigationsleiste unter `Users` das `Attributes` Tab des `admin` Benutzers ausgewählt und das Attribut `favorite_color` mit dem Value `blue` hinzugefügt. Dieses Attribut kann über einen Protocol Mapper referenziert werden.

Davor wird ein neuer Client Scope erzeugt. Die Konfiguration dieses Client Scopes ist in Abbildung 1.11 gezeigt. Anders als im ersten Beispiel ist `Include In Token Scope` auf `ON`, damit der Name des Client Scopes `colors` im `scopes` Claim des Access Tokens enthalten ist.

Danach wird diesem Client Scope der in Abbildung 1.12 gezeigte Protocol Mapper hinzugefügt. Als ***Mapper Type*** wird in diesem Fall `User Attribute` ausgewählt. Das ***User Attribute*** Feld enthält den Namen des für den Benutzer angelegten Attributs. ***Token Claim Name*** setzt den Key des Claims

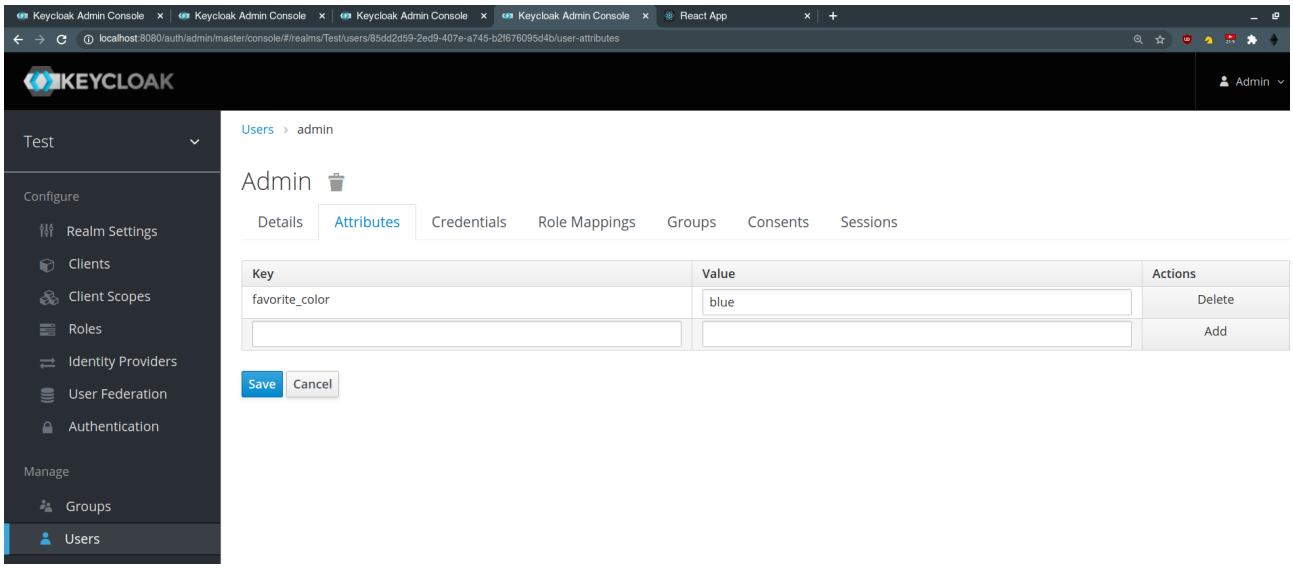

Abbildung 1.10: Keycloak Client Benutzer Attribut

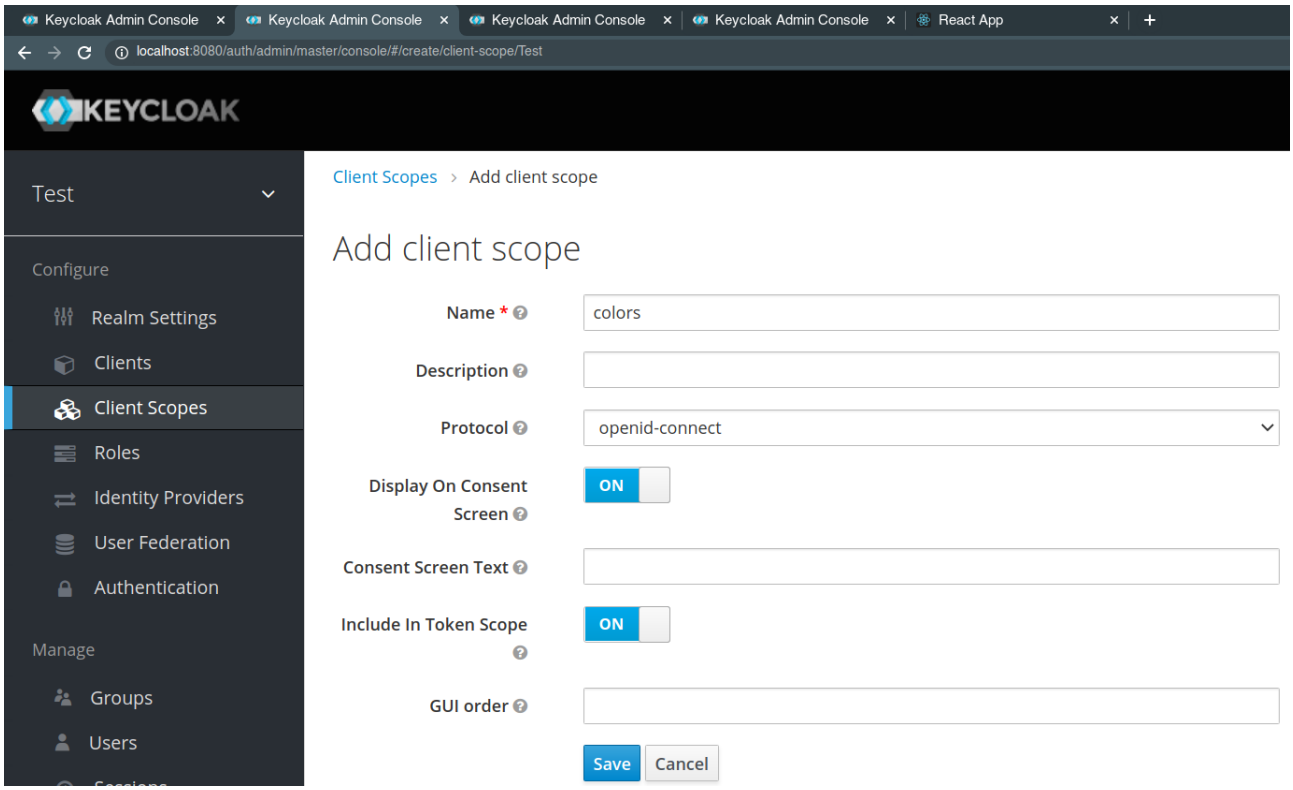

Abbildung 1.11: Keycloak Client Scope für ein Benutzer Attribut

auf `GUI_color`. `Add to ID token` und `Add to userinfo` ist beides auf `ON`, damit der Claim sowohl im ID Token enthalten ist als auch vom UserInfo Endpunkt zurückgegeben werden kann. Der Value dieses Claims ist dann der Wert des `favourite_color` Attributes des authentifizierten Benutzers. Wenn für einen Benutzer kein `favourite_color` Attribut hinzugefügt ist, dann ist der Claim nicht im ID Token und nicht in Antworten des UserInfo Endpunkts enthalten. Wie im ersten Beispiel muss der Client Scope den zwei Fontend Clients hinzugefügt werden.

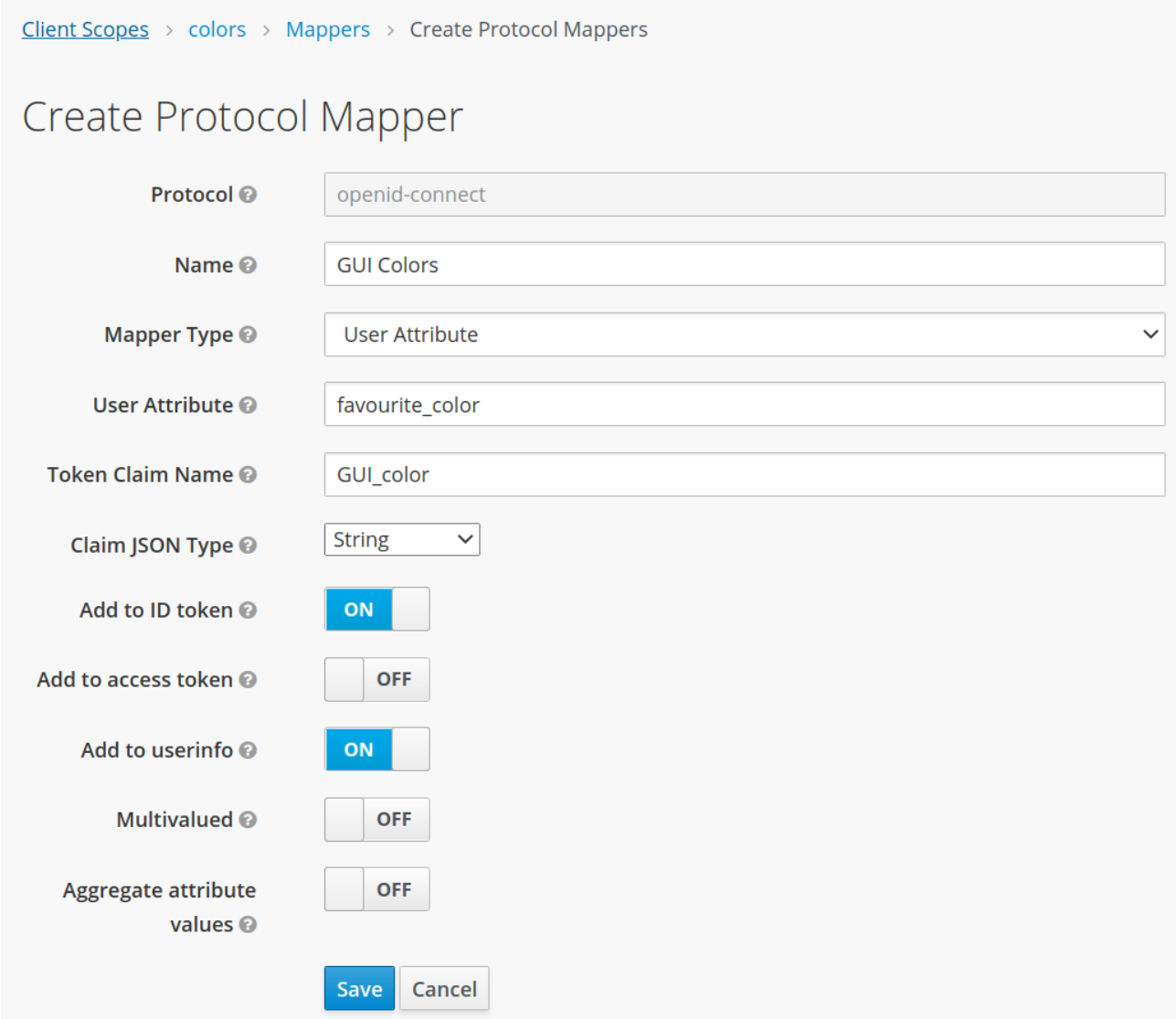

Abbildung 1.12: Keycloak Protocol Mapper für `GUI_color` Claim

1.4.2 Frontend

Das Frontend verwendet das Web Framework ***ReactJS*** [Fac20]. Für Keycloak gibt es sogenannte Adapter. Diese Adapter sind für verschiedenen Programmiersprachen und Systeme wie ReactJS verfügbar. Im Frontend wird der Adapter mit dem Namen `react-keycloak` verwendet [Mat20]. Ein Adapter ist mehr als eine Programmbibliothek oder ein Paket [Fac20]. Die Adapter sind eng in das System, wie ReactJS, integriert [Fac20]. Zum Beispiel implementiert der Adapter den Authorization Code Flow zur Authentifizierung eines Benutzers. Authentifiziert sich der Benutzer das erste Mal, dann kann dieser Flow mit dem Aufruf der `login()` Funktion initiiert werden. Wenn sich der Benutzer bereits bei einem anderen Client authentifiziert hat, führt der Adapter den Authorization Code Flow beim Aufrufen der Website im Hintergrund aus, um den Benutzer ohne erneute Eingabe von Anmeldeinformationen wie Benutzernamen und Passwort zu authentifizieren. Im nachfolgenden Beispiel wird das anhand von Screenshots der Anwendung noch einmal ausführlicher beschrieben.

In 1.13 ist eine Abbildung der Website von `frontend1` zu sehen. Dabei hat sich der Benutzer noch nicht authentifiziert. Über die `GET SERVICE 1 DATA` und `GET SERVICE 2 DATA` Buttons wird jeweils eine Funktion aufgerufen, welche eine HTTP Post Anfrage an den jeweiligen Backend Service,

die Clients mit ID `service1` und `service2`, sendet. Ist ein Benutzer authentifiziert, dann wird der Access Token im Bearer Header der Anfrage mitgesendet. Die Backend Services überprüfen, ob eingehende Nachrichten einen validen Access Token enthalten. Bei Erfolg liefern diese die Identifikationsnummer für den Benutzer, den `sub` Claim, mit dem HTTP-Statuscode 200 zurück. Andernfalls wird, wie in Abbildung 1.14 gezeigt, `FAILED` mit Statuscode 402 zurückgegeben und angezeigt. Der Backend Service wird in der nächsten Sektion 1.4.3 näher erläutert.

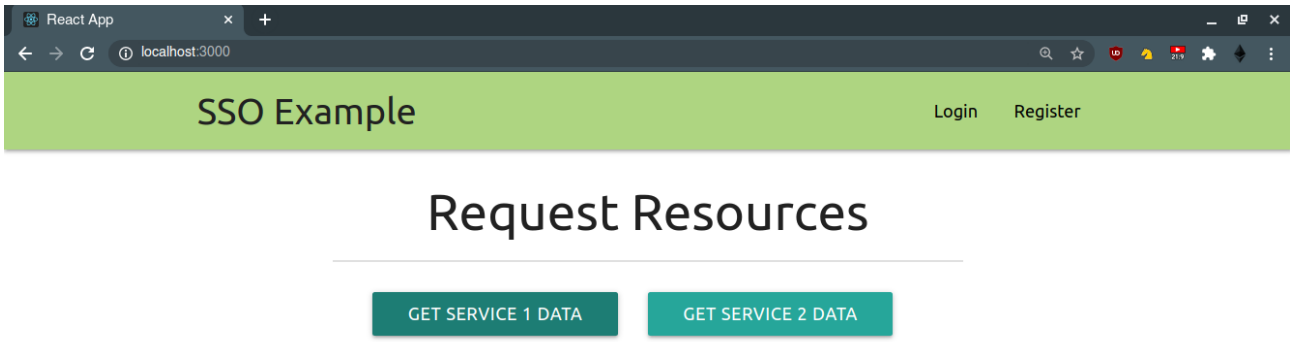

Abbildung 1.13: Frontend Client GUI - Nicht Authentifiziert

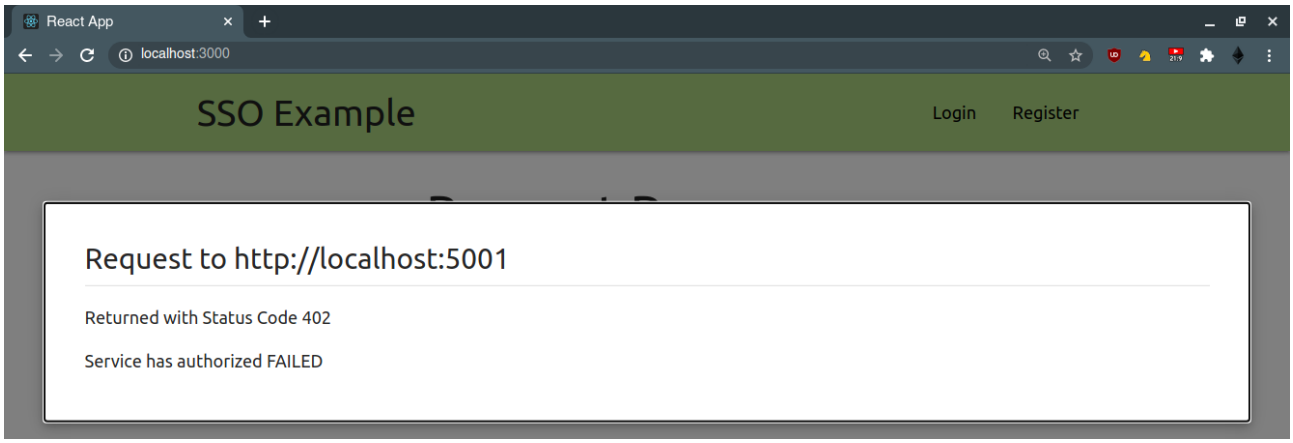

Abbildung 1.14: Frontend Client GUI - Nicht Authentifizierte Anfrage an Backend

In der oberen Navigationsleiste sind auf der rechten Seite zwei Buttons, über die sich Benutzer authentifizieren oder registrieren können. Für beides bietet der Keycloak Adapter Funktionen an, welche beim Klicken des jeweiligen Buttons ausgeführt werden. Wenn diese Buttons nicht angezeigt werden, muss im Browser herausgezoomt oder das Browser Fenster vergrößert werden. Beim Klicken des `Login` Buttons wird der Authorization Code Flow initiiert. Der Benutzer wird dann zu Keycloaks Authorization Endpunkt geleitet. Das ist in Abbildung 1.15 dargestellt. In der Adresszeile sieht man die Addresse des Authorization Endpunkts und einen Teil des Query-Strings, der die Parameter des Authorization Requests enthält. Der Benutzer kann sich hier z.B. als Admin Benutzer mit 'admin:iuq123' (Username:Password), oder mit dem bereits registrierten Beispielbenutzer 'user:user' einloggen. Für beide Benuter sind unterschiedliche Lieblingsfarben als Attribute konfiguriert.

Nach einer erfolgreichen Authentifizierung mit dem Authorization Code Flow wird der Benutzer zurück auf die Website des Frontend Clients geleitet. Anfragen an den Backend Service sind nun erfolgreich. Das ist in Abbildung 1.16 dargestellt. Access Tokens werden automatisch im Hintergrund erneuert. Dabei sendet der Adapter Anfragen mit dem Refresh Token an Keycloaks Token End-

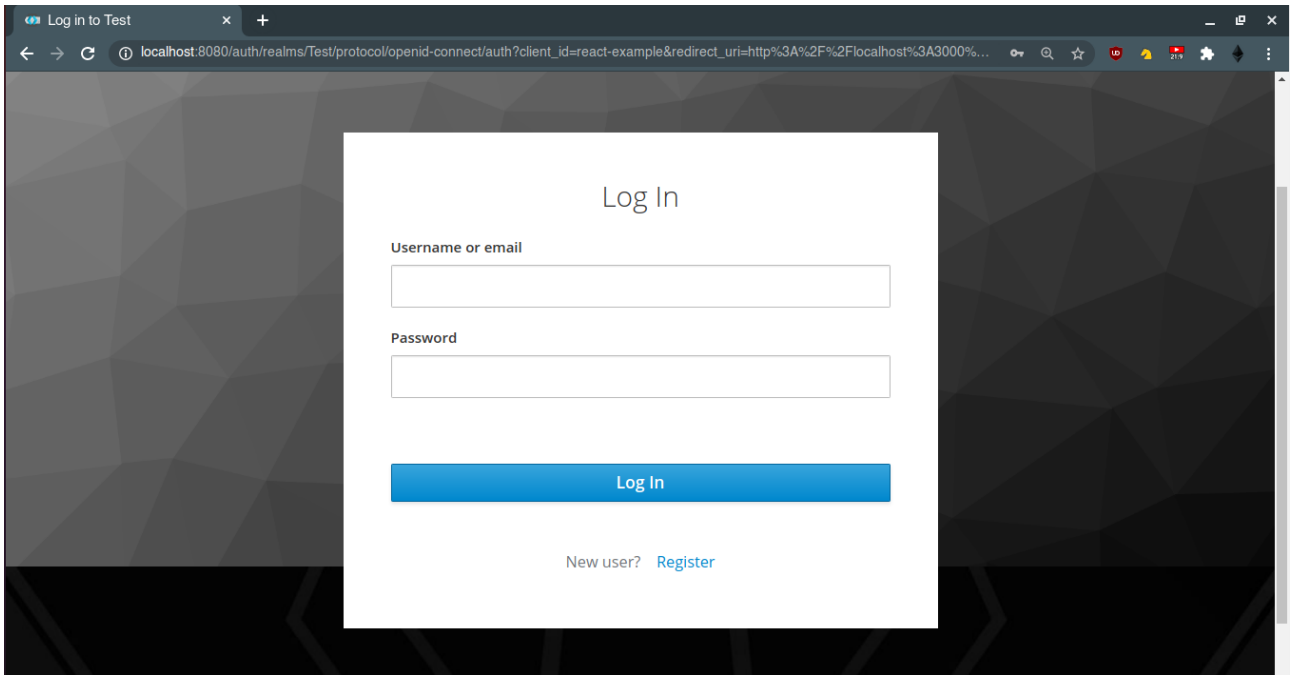

Abbildung 1.15: Keycloak GUI - Login Form des Authorization Endpunkts

punkt. Standardmäßig hat der Refresh Token in Keycloak eine Verfallszeit von 10 Stunden. Über den Token Endpunkt ruft der Adapter ebenfalls automatisch neue Refresh und ID Token ab.

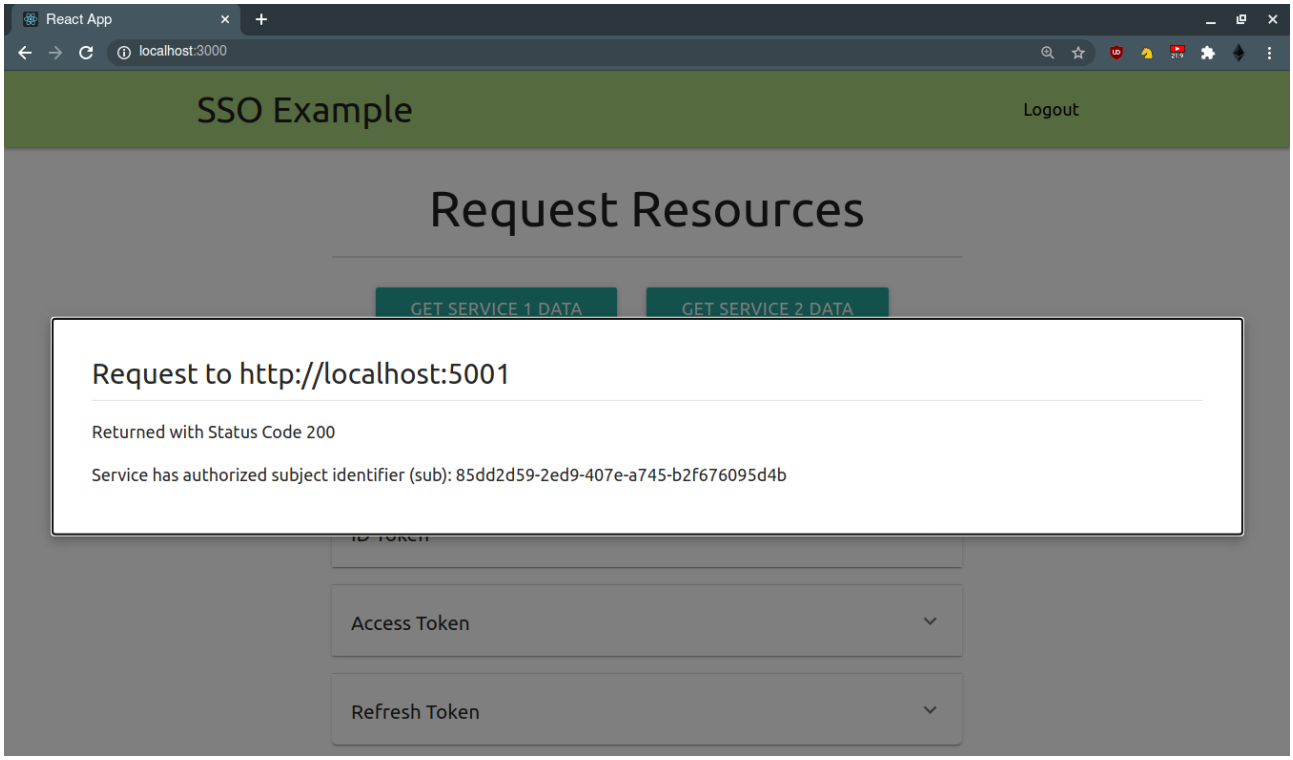

Abbildung 1.16: Frontend Client GUI - Authentifizierte Anfrage an Backend

Der Adapter stellt außerdem APIs zum Abrufen von Tokens zur Verfügung. Ist ein Benutzer authentifiziert, dann werden ID-, Access-, und Refresh Token in der GUI dekodiert angezeigt. Abbildung 1.17 zeigt davon einen Ausschnitt.

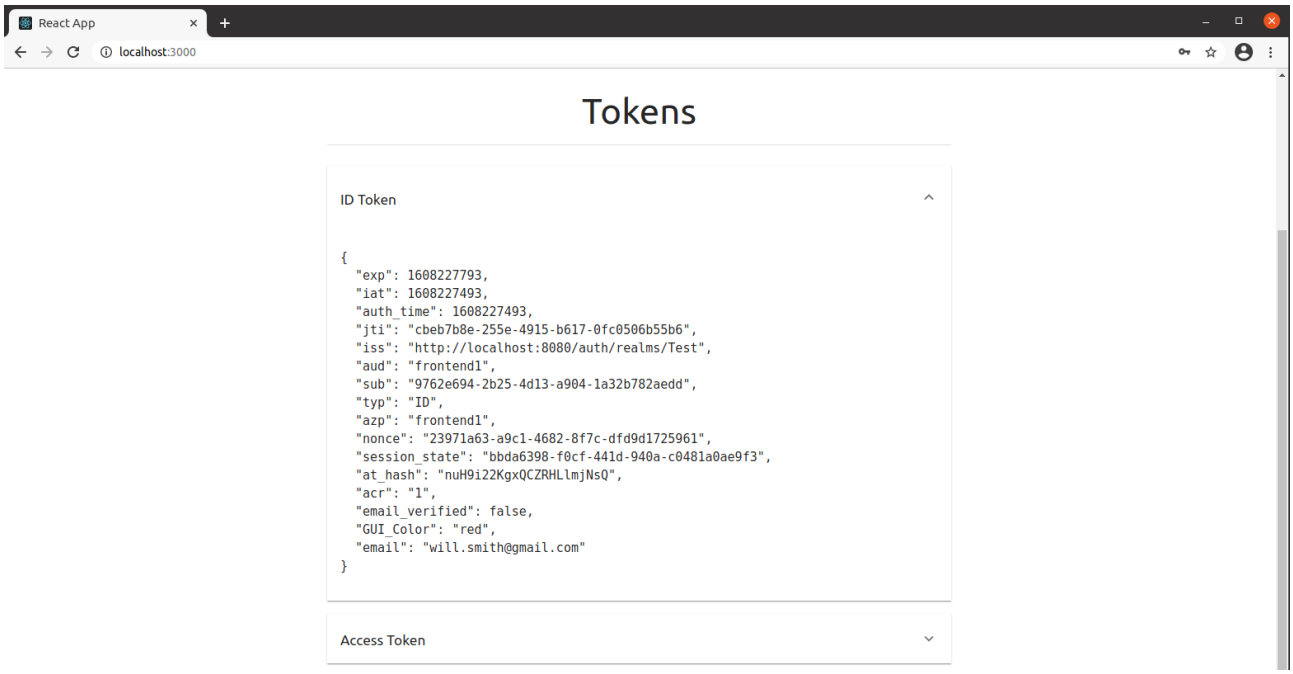

Abbildung 1.17: Frontend Client GUI - Token

Durch die erfolgreichen Authentifizierung mit dem Authorization Code Flow hat Keycloak den in Sektion 1.2.4 beschriebenen `KEYCLOAK_IDENTITY` Cookie für Keycloaks Authorization Endpunkt gesetzt. Dieser Cookie enthält einen ID Token. In Abbildung 1.18 ist manuell zum Authorization Endpunkt navigiert worden und über die Chrome Developer Console kann der Cookie angezeigt werden. Durch diesen Cookie wird, wie in Abbildung 1.19 gezeigt, der Benutzer beim Aufruf des zweiten Frontend Clients automatisch authentisiert.

1.4.3 Backend

Die zwei Backend Services lauschen auf verschiedenen Ports, sind aber im Hinblick auf die Funktionsweise gleich. Sie sind in Python geschrieben und verwenden das Web MicroFramework ***Flask*** [Arm20]. `Flask-CORS`, eine Erweiterung von Flask, wird verwendet, um CORS für die Sender aller eingehenden Anfragen zu erlauben [Cor20]. `PyJWT` implementiert den JSON Web Token (JWT) Standard (RFC 7519 [JBS15]) [Jos20]. Das Backend verwendet PyJWT um Access Tokens zu decodieren.

Eingehende Anfragen werden in zwei Schritten abgearbeitet. Im ersten Schritt wird der Access bzw. Bearer Token aus dem HTTP Authorization Header extrahiert. Schlägt dies fehl, z.B. wenn die Anfrage keinen HTTP Authorization Header enthält, dann wird der Text `FAILED` im HTTP Message Body mit Statuscode 402 zurückgegeben. Andernfalls wird der extrahierte Token an die in der Auflistung 1.5 gezeigten Funktion übergeben.

Dabei wird der Token als erstes über die PyJWT Bibliothek dekodiert. Die Signatur des Tokens wird dort noch nicht validiert. Danach wird überprüft, ob der Token einen `aud` Claim enthält und ob die ID des jeweiligen Backend Clients im `aud` Claim enthalten ist. Die Motivation für diese Art der Überprüfung wurde in Sektion 1.2.4 erläutert. Für die Validierung der Signatur wird die in Sektion 1.2.4 beschriebene Online Validierung eingesetzt, bei der der Token über Keycloak validiert wird. Dabei wird eine Anfrage an Keycloaks UserInfo Endpunkt mit dem Token im HTTP Authorization Header als Bearer Token gesendet. Bei valider Signatur liefert die Keycloak Anfrage den Statuscode 200 und

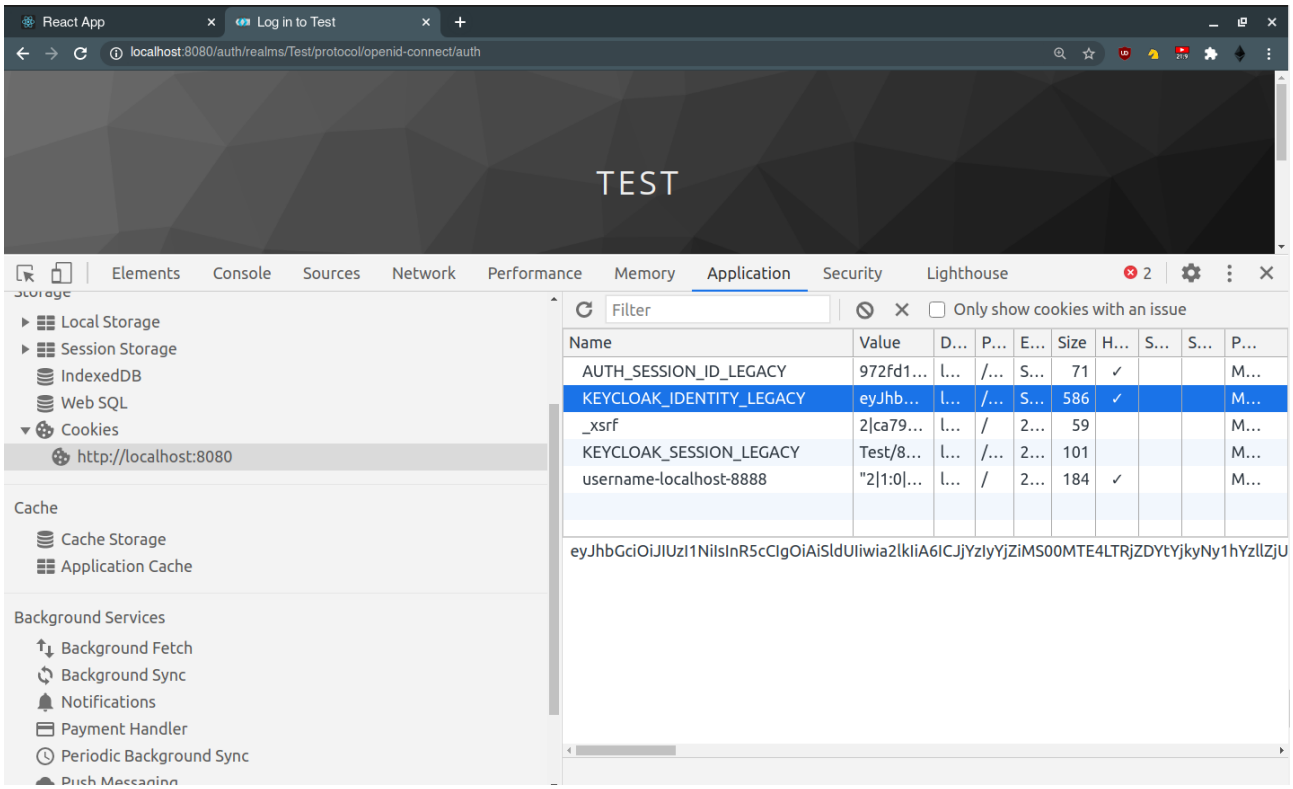

Abbildung 1.18: ID Token Cookie für Keycloak

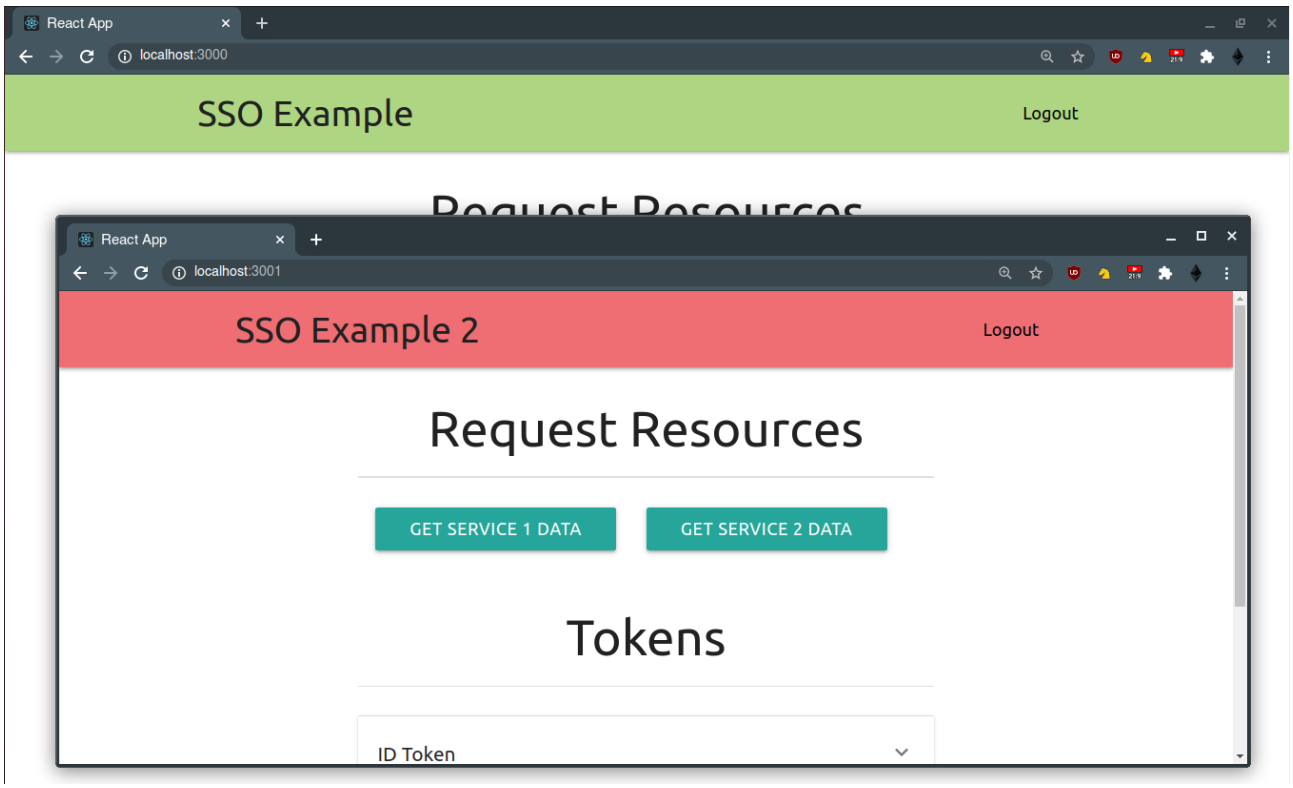

Abbildung 1.19: Snapshot nach Aufruf des zweiten Frontends

die für den UserInfo Endpunkt und den Access Token zugelassenen Claims im HTTP Message Body zurück.

Die Funktion in Auflistung 1.5 gibt einen Tuple mit zwei Elementen zurück. Das erste Element wird als HTTP Message Body und das zweite Element als Statuscode an den Sender der Anfrage an das Backend zurückgesendet. Bei nicht erfolgreicher Validierung des Tokens wird der Statuscode 402 mit Fehlermeldung im HTTP Message Body zurückgegeben. Andernfalls wird die ID des Benutzers des Access Tokens und Statuscode 200 zurückgegeben.

```
def validate_token(token):
  # Decode Access Token without validating it.
  try:
    decoded_token = jwt.decode(token, verify=False)
  except Exception:
    return 'FAILED', 402
  # Invalid if this client's ID is not in 'aud' claim
  if 'aud' not in decoded_token:
    return 'FAILED: no "aud" claim in token', 402
  if client_ID not in decoded_token['aud']:
    return f'FAILED: {client_ID} not in "aud" claim', 402
  # Online Signatur Validation
  url = (f'http://{KEYCLOAK_HOST}/auth/realms/{REALM_NAME}' +
         '/protocol/openid-connect/userinfo')
  headers = {"Authorization": f'Bearer {token}'}
  r = requests.get(url, headers=headers)
  if r.status_code != 200:
    return 'Invalid signature', 402
  r_json = json.loads(r.text)
  return f'subject identifier (sub): {r_json["sub"]}', 200
```

Listing 1.5: Token Validierung im Backend

1.5 Zusammenfassung

In dieser Arbeit ist SSO bei Webanwendungen vorgestellt worden. Dabei wurde speziell das OIDC Protokoll und Keycloak, sowie die Sicherheit dieser analysiert. Keycloak unterstützt OIDC und implementiert geeignete Gegenmaßnahmen, um sich und Keycloaks Benutzer vor den unter anderem in Sektion 1.3 Threat Model vorgestellten möglichen Threats bei Web SSO zu schützen. Aus diesem Grund kann Keycloak und dessen Adapter, wie in der implementierten und in Sektion 1.4 vorgestellten Beispielwebanwendung verwendet werden, um Web SSO sicher zu implementieren.

Literatur

[Aan20] Aanand Prasad et al. ***Docker Compose Github***. 2020. url: https://github.com/docker/compose (besucht am 29. 11. 2020).

[Ami19] Amin Saqi. ***A Survey on SSO Authentication Protocols: Security and Performance***. 2019. url: https://medium.com/@aminsaqi/a-survey-on-sso-authentication-protocols-security-and-performance-287dcb634bdd (besucht am 29. 11. 2020).

[Arm20] Armin Ronacher et al. ***Flask Github***. 2020. url: https://github.com/pallets/flask (besucht am 30. 11. 2020).

[Aut20] Auth0. ***Auth0 Docs***. 2020. url: https://auth0.com/docs/tokens (besucht am 29. 11. 2020).

[BK16] Tayibia Bazaz und Aqeel Khalique. „A Review on Single Sign on Enabling Technologies and Protocols“. In: ***International Journal of Computer Applications*** 151 (Okt. 2016), S. 18–25. doi: 10.5120/ijca2016911938.

[Che+14] Eric Chen u. a. „OAuth Demystified for Mobile Application Developers“. In: ***Proceedings of the ACM Conference on Computer and Communications Security (CCS)***. ACM - Association for Computing Machinery, Nov. 2014. url: https://www.microsoft.com/en-us/research/publication/oauth-demystified-for-mobile-application-developers/.

[con16] connect2id. ***OpenID Connect explained***. 2016. url: https://connect2id.com/learn/openid-connect (besucht am 29. 11. 2020).

[Cor20] Cory Dolphin et al. ***Flask-CORS Github***. 2020. url: https://github.com/corydolphin/flask-cors (besucht am 30. 11. 2020).

[D H12a] D. Hardt. ***The OAuth 2.0 Authorization Framework - Authorization Response***. 2012. url: https://tools.ietf.org/html/rfc6749#section-4.1.2 (besucht am 07. 12. 2020).

[D H12b] D. Hardt. ***The OAuth 2.0 Authorization Framework - Security Considerations***. 2012. url: https://tools.ietf.org/html/rfc6749#section-10 (besucht am 07. 12. 2020).

[Dan20] Daniel Ebert. ***Ebert Dockerhub Docker Images***. 2020. url: https://hub.docker.com/u/danielebert00 (besucht am 29. 11. 2020).

[Fac20] Facebook Inc. ***React***. 2020. url: https://reactjs.org/ (besucht am 29. 11. 2020).

[Har12] D. Hardt. ***The OAuth 2.0 Authorization Framework***. RFC 6749. RFC Editor, Okt. 2012, S. 1–76. url: https://tools.ietf.org/html/rfc6749.

[Ide20] IdentityServer. ***IdentityServer3 - Terminology***. 2020. url: https://identityserver.github.io/Documentation/docsv2/overview/terminology.html (besucht am 29. 11. 2020).

[Jan16] Jannik Hüls. ***Single Sign-On mit Keycloak als OpenID Connect Provider***. 2016. url: https://blog.codecentric.de/2016/08/single-sign-mit-keycloak-als-openid-connect-provider/ (besucht am 29. 11. 2020).

[JBS15] M. Jones, J. Bradley und N. Sakimura. ***JSON Web Token (JWT)***. RFC 7519. RFC Editor, Mai 2015, S. 1–30. url: https://www.rfc-editor.org/rfc/rfc7519.txt.

[JH12] M. Jones und D. Hardt. ***The OAuth 2.0 Authorization Framework: Bearer Token Usage***. RFC 6750. RFC Editor, Okt. 2012, S. 1–18. url: https://tools.ietf.org/html/rfc6750.txt.

[Jon15] M. Jones. ***JSON Web Signature (JWS)***. RFC 7515. RFC Editor, Mai 2015, S. 1–59. url: https://tools.ietf.org/html/rfc7515.txt.

[Jos20] José Padilla et al. ***PyJWT Github***. 2020. url: https://github.com/jpadilla/pyjwt (besucht am 30. 11. 2020).

[Jus15] Justin Richer. ***User Authentication with OAuth 2.0 - ID Token***. 2015. url: https://oauth.net/articles/authentication/ (besucht am 01. 12. 2020).

[LMH13] T. Lodderstedt, M. McGloin und P. Hunt. ***OAuth 2.0 Threat Model and Security Considerations***. RFC 6819. RFC Editor, Jan. 2013, S. 1–71. url: https://www.rfc-editor.org/rfc/rfc6819.txt.

[Mar20] Margaret Rouse. ***single sign-on (SSO)***. 2020. url: https://searchsecurity.techtarget.com/definition/single-sign-on (besucht am 01. 12. 2020).

[Mat20] Mattia Panzeri et al. ***@react-keycloak/web***. 2020. url: https://www.npmjs.com/package/@react-keycloak/web (besucht am 29. 11. 2020).

[MMS16] Vladislav Mladenov, Christian Mainka und Jörg Schwenk. ***On the security of modern Single Sign-On Protocols: Second-Order Vulnerabilities in OpenID Connect***. 2016. arXiv: 1508.04324 [cs.CR].

[N S14a] N. Sakimura et al. ***OpenID Connect Core 1.0***. 2014. url: https://openid.net/specs/openid-connect-core-1_0.html (besucht am 29. 11. 2020).

[N S14b] N. Sakimura et al. ***OpenID Connect Dynamic Client Registration***. 2014. url: https://openid.net/specs/openid-connect-registration-1_0.html (besucht am 01. 12. 2020).

[N S20] N. Sakimura et al. ***OpenID Connect Basic Client Implementer's Guide 1.0 - draft 40***. 2020. url: https://openid.net/specs/openid-connect-basic-1_0.html#CodeOK (besucht am 29. 11. 2020).

[Okt18] Okta, Inc. ***Access Token Lifetime***. 2018. url: https://www.oauth.com/oauth2-servers/access-tokens/access-token-lifetime/ (besucht am 30. 11. 2020).

[Okt20] Okta, Inc. ***Authorization vs Authentication***. 2020. url: https://www.oauth.com/oauth2-servers/openid-connect/authorization-vs-authentication/ (besucht am 25. 11. 2020).

[Oli18] Olivier Rivat. ***Using Client Scope with RedHat SSO Keycloak***. 2018. url: https://www.janua.fr/using-client-scope-with-redhat-sso-keycloak/ (besucht am 01. 12. 2020).

[Red20a] Redhat, Inc. ***Authorization Services Guide - Role-Based Policy***. 2020. url: https://www.keycloak.org/docs/latest/authorization_services/#_policy_rbac (besucht am 29. 11. 2020).

[Red20b] Redhat, Inc. ***Securing Applications and Services Guide***. 2020. url: https://www.keycloak.org/docs/latest/securing_apps/ (besucht am 29. 11. 2020).

[Red20c] Redhat, Inc. ***Server Administration Guide***. 2020. url: https://www.keycloak.org/docs/latest/server_admin/index.html (besucht am 28. 11. 2020).

[Red20d] Redhat, Inc. ***Server Administration Guide - Red Hat Single Sign-On 7.0***. 2020. url: https://access.redhat.com/documentation/en-us/red_hat_single_sign-on/7.0/html/server_administration_guide/index (besucht am 25. 11. 2020).

[Red20e] Redhat, Inc. ***Server Installation and Configuration Guide - Standalone Clustered Mode***. 2020. url: https://www.keycloak.org/docs/latest/server_installation/#_standalone-ha-mode (besucht am 29. 11. 2020).

[RR12] Vedala Radha und D. Reddy. „A Survey on Single Sign-On Techniques“. In: ***Procedia Technology*** 4 (Dez. 2012), S. 134–139. doi: 10.1016/j.protcy.2012.05.019.

[RS17] Justin Richer und Antonio Sanso. ***OAuth 2 in Action***. 1. Aufl. Shelter Island, NY 11964: Manning Publications, 2017.

[Seb20a] Sebastian Scherer and Daniel Ebert. ***Single Sign On bei Webanwendungen***. 2020. url: https://github.com/IyotakeTatanka/sichere_webanwendungen (besucht am 29. 11. 2020).

[Seb20b] Sebastian Scherer and Daniel Ebert. ***Single Sign On bei Webanwendungen - src README***. 2020. url: https://github.com/IyotakeTatanka/sichere_webanwendungen/tree/master/src (besucht am 29. 11. 2020).

[Sti18a] Stian Thorgersen. ***Securing apps and services with Keycloak authentication | DevNation Tech Talk - Time: 10:30***. 2018. url: https://www.youtube.com/watch?v=mdZauKsMDiI%5C&feature=youtu.be%5C&list=WL%5C&t=630 (besucht am 29. 11. 2020).

[Sti18b] Stian Thorgersen. ***Securing apps and services with Keycloak authentication | DevNation Tech Talk - Time: 12:25***. 2018. url: https://www.youtube.com/watch?v=mdZauKsMDiI%5C&feature=youtu.be%5C&list=WL%5C&t=745 (besucht am 29. 11. 2020).

[Sus20] Susan Harper et al. ***OpenID Connect & OAuth 2.0 API - token Response properties***. 2020. url: https://developer.okta.com/docs/reference/api/oidc/#response-properties (besucht am 01. 12. 2020).

[ZE14] Yuchen Zhou und David Evans. „SSOScan: Automated Testing of Web Applications for Single Sign-On Vulnerabilities“. In: ***23rd USENIX Security Symposium (USENIX Security 14)***. San Diego, CA: USENIX Association, Aug. 2014, S. 495–510. isbn: 978-1-931971-15-7. url: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou.

Eidesstattliche Erklärung

Wir versichern, dass wir die Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen oder anderen Quellen entnommen sind, sind als solche kenntlich gemacht. Die Abhandlung wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Aalen, den 21. Dezember 2020

Daniel Ebert

Sebastian Scherer