Sicherheit in der Smart Systems Service Infrastructure (S^3I)

Ein KWH4.0-Standpunkt

23.3.2022

Kompetenzzentrum Wald und Holz 4.0
c/o RIF Institut für Forschung und Transfer e.V. (Projektkoordination)
Joseph-von-Fraunhofer-Straße 20
D-44227 Dortmund
www.kwh40.de

Kontakt

Kompetenzzentrum Wald und Holz 4.0
c/o RIF Institut für Forschung und Transfer e.V. (Projektkoordination)
Joseph-von-Fraunhofer-Straße 20
D-44227 Dortmund
www.kwh40.de

Ansprechpartner: Dipl.-Ing. Frank Heinze
Tel. +49 (0) 231 9700-781
frank.heinze@rt.rif-ev.de

Verantwortlicher Autor: Jiahang Chen, MMI

Autoren

RIF Institut für Forschung und Transfer e.V. (Koordinator)
Geschäftsführung: Dr. Svenja Rebsch, Dipl.-Inf. Michael Saal
Joseph-von-Fraunhofer Str. 20, 44227 Dortmund

Werkzeugmaschinenlabor (WZL), RWTH Aachen
Institutsleiter: Prof. Dr.-Ing. Christian Brecher
Steinbachstraße 19, 52074 Aachen

Institut für Mensch-Maschine-Interaktion (MMI), RWTH Aachen
Institutsleiter: Prof. Dr.-Ing. Jürgen Roßmann
Ahornstraße 55, 52074 Aachen

Institut für Arbeitswissenschaft (IAW), RWTH Aachen
Institutsleiterin: Prof. Dr.-Ing. Verena Nitsch
Bergdriesch 27, 52062 Aachen

Landesbetrieb Wald und Holz Nordrhein-Westfalen

Wald und Holz NRW, Lehr- und Versuchsforstamt Arnsberger Wald
Forstliches Bildungszentrum für Waldarbeit und Forsttechnik
Leitung: FD Thilo Wagner
Alter Holzweg 93, 59755 Arnsberg

Förderhinweise

Das Vorhaben „Kompetenzzentrum Wald und Holz 4.0“ wurde gefördert durch das Land Nordrhein-Westfalen unter Einsatz von Mitteln aus dem Europäischen Fonds für regionale Entwicklung (EFRE).

EFRE.NRW
Investitionen in Wachstum und Beschäftigung

EUROPÄISCHE UNION
Investition in unsere Zukunft
Europäischer Fonds
für regionale Entwicklung

Das Vorhaben „iWald“ wird gefördert durch:
Bundesministerium für Ernährung und Landwirtschaft aufgrund eines Beschlusses des Deutschen Bundestages (FKZ 22012718).

Bundesministerium für Ernährung und Landwirtschaft

FNR
Fachagentur Nachwachsende Rohstoffe e.V.

Version	Datum	Abschnitte	Änderungen
0.9	17.12.2021	Alle	Erste interne Version
1.0	23.03.2022	Alle	Erste Version

Inhalt

1 Einführung 4
2 Klassische Autorisierung 4
2.1 OAuth 2.0 vs. OpenID Connect 4
2.2 Tokens 5
2.2.1 Access Token vs. ID Token 5
2.2.2 Refresh Token vs. Offline Token 7
2.3 Flows 7
2.3.1 Authorization Code 8
2.3.2 PKCE 8
2.3.3 Client Credentials 9
2.3.4 Device Authorization Grant 10
2.3.5 Resource Owner Password Credentials 11
2.4 Neuerungen in OAuth 2.1 11
3 Keycloak-Features im S³I-IdentityProvider 12
3.1 Realm 12
3.2 User 12
3.3 Client 12
3.4 Scopes 13
4 S³I-Autorisierung 13
4.1 In zentralen S³I-Diensten 13
4.1.1 S³I-IdentityProvider 13
4.1.2 S³I-Directory und -Repository 13
4.1.3 S³I-Broker 16
4.1.4 S³I-Registration 17
4.2 In WH4.0-Dingen 18
4.2.1 Dezentrale Autorisierung 18
4.2.2 Zentrale Autorisierung 19

1 Einführung

Ein wichtiger Aspekt in einem verteilten Internet der Dinge ist die Sicherheit. Darunter ist unter anderem

- Authentisierung: Ich bin es!
- Authentifizierung: Ist er es wirklich?
- Autorisierung: Was darf er?

zu verstehen. In dem vorliegenden Dokument ist das grundlegende Konzept der IoT-Sicherheit und deren Umsetzung in der ***Smart Systems Service Infrastructure*** (S³I) vorgestellt. Für ein grundlegendes Verständnis der S³I sei auf den entsprechenden KWH4.0-Standpunkt unter www.kwh40.de verwiesen.

2 Klassische Autorisierung

Dieses Kapitel stellt die klassische Autorisierung nach dem OAuth 2.0-Protokoll[1] sowie dem darauf basierenden OpenID Connect-Standard (OIDC)[2] dar. Die konkrete Umsetzung in der S³I sind in Kapitel 4 beschrieben.

2.1 OAuth 2.0 vs. OpenID Connect

OAuth 2.0 ist ein offenes industrielles Standardprotokoll, das einen sicheren API-Zugriff ermöglicht, indem eine Autorisierung während der Datenübermittlung unter Anwendung der im Protokoll definierten Autorisierungsflüsse (siehe Kapitel 2.3) durchgeführt wird.

Im OAuth 2.0-Protokoll werden die folgenden Rollen definiert:

- ***Resource Owner***: Entität, die Zugriff auf eine (geschützte) Ressource gewähren kann. Typischerweise ist dies der Endbenutzer.
- ***Client***: Anwendung, die den Zugriff auf eine geschützte Ressource im Namen des Resource Owners anfordert. Bei der Registrierung werden eine ID und ein Secret (Password) einem Client zugewiesen. Client ID ist ein sogenannter öffentlicher Identifier, der eindeutig ist. Client Secret ist nur beim Autorization Server und dem Client selbst bekannt.
- ***Resource Server***: Dienst, der die geschützten Ressourcen hostet. Typischerweise stellt der Dienst APIs bereit, die die Anfragen nach geschützten Ressourcen annehmen und darauf reagieren können.
- ***Authorization Server***: Server, der den Resource Owner authentifiziert und nach der Autorisierung ein Access Token ausstellt und an den Client zurückgibt.

Clients unterteilen sich im Kontext von OAuth 2.0 weiter in *confidential clients*, *public clients*[3] *und bearer-only clients*. Confidential clients sind in der Lage, sich sicher beim Authorization Server zu authentifizieren und ihr Client Secret sicher aufzubewahren. Beispiele für confidential clients sind unter anderem Frontend-Programme, die unmittelbar auf einem Server betrieben werden, und somit die Secrets gut aufbewahren können. *Public Clients* bekommen nach der Registrierung kein Client Secret, weil sie nicht in der Lage sind, ihr Secret sicher aufzubewahren, deswegen dürfen sie nicht eigenständig beim Authorization Server ihr Secret gegen Access Token austauschen (Client Credentials Grant), sondern nur in Namen eines Resource Owners (User) den Zugriff auf Ressourcen anfordern (Authorization Code PKCE, siehe Abschnitt 2.3.2). Beispiele sind Browser-Apps (z.B. Jupyter-Notebook), mobile Apps,

[1] https://oauth.net/2/
[2] https://openid.net/connect/
[3] https://oauth.net/2/client-types/

Desktop Apps etc. Bearer-only Clients sind von der Art her passiv reagierend und dürfen sich weder selbstständig (Client Credentials Grant) noch über einen User anmelden. Typische Anwendungen sind Backend-Programme wie REST-API-Server, Proxy etc., welche niemals eine Anmeldung initiieren und nur Token von anderen empfangen und validieren.

Ergebnis einer OAuth 2.0-Autorisierung ist ein ***Access Token*** (siehe Kapitel 2.2), welches genutzt wird, um einen sicheren Zugriff auf eine geschützte Ressource zu erhalten.

Weil Access Token statt einer Identitätssemantik nur eine Autorisierungssemantik beschreiben, wird das OAuth 2.0-Protokoll im OpenID Connect-Protokoll um eine Identitätsschicht erweitert, um ein Rahmenwerk für verteilte Identitäten zu schaffen. Dieser Standard erlaubt Anwendungen und Authentifizierungsdiensten, persönliche Informationen (z.B. Vor- und Nachname) und Identitätsattribute (z.B. URI-basierte Identitäten) eines Users in Form eines ***ID Token***s (auch siehe Abschnitt 2.2.1) nach einem standardisierten Verfahren untereinander auszutauschen.

2.2 Tokens

2.2.1 Access Token vs. ID Token

Access Tokens[4] sind Berechtigungsnachweise, die von einem Client für den Zugriff auf geschützte Ressourcen verwendet werden. Sie enthalten u.a. diverse Informationen darüber, welche Berechtigung der Client im Namen des Resource Owners besitzt sowie einen Gültigkeitszeitraum.

Access Tokens bieten eine Abstraktionsschicht, die verschiedene Berechtigungskonstrukte (z.B. Benutzername und Passwort) durch ein einziges Token ersetzen, das vom Reource Server verstanden wird. Diese Abstraktionsschicht macht es für den Resource Server überflüssig, eine breite Palette von Authentifizierungsmethoden zu verstehen.

Access Tokens werden vom Authorization Server ausgestellt und vom Resource Server validiert[5]. Sie können je nach Sicherheitsanforderungen des Resource Servers unterschiedliche Formate, Strukturen und Verwendungsmethoden (z.B. welche Claims vom Token werden beim Resource Server validiert) haben. Klassische Umsetzung des Access Token ist JSON Web Token[6] (JWT). Diese lassen sich mit JSON Web Signature (JWS)[7] zudem signieren.

Ein JWT besteht aus einem Header (in Abbildung 2-1 rot markiert), seinem Payload (in Abbildung 2-1 blau markiert) und einer Signatur (in Abbildung 2-1 grün markiert)[8]:

[4] https://www.oauth.com/oauth2-servers/access-tokens/
[5] https://auth0.com/docs/tokens/access-tokens/validate-access-tokens
[6] https://datatracker.ietf.org/doc/html/rfc7519
[7] https://datatracker.ietf.org/doc/html/rfc7515
[8] (ISM Work, Lukas Poqué, 2020)

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "exp": 1575562308,
  "sub": "889f68",
  "email_verified": true,
  "preferred_username":"maxmuster"
}
{
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload)
}
```

Abbildung 2-1: Beispielhaftes JSON Web Token

- Der Header wird in Base64URL[9] kodiert und enthält neben dem Typ des für die Signatur verwendeten Hash-Algorithmus auch den Typ des Tokens (z.B. JWT, JWE, JWS)
- Im Payload werden sogenannte Claims angegeben. Hierbei gibt es einige Standard-Claims[10] (immer drei Zeichen lang) wie z.B. „exp“ für die Zeit, die das Token gültig ist („expiration“), oder „sub“ für die Benutzerbeschreibung („subject“ - z.B. Benutzername oder UUID). Alle Standard-Claims werden in RFC 7519[11] vorgestellt.
  Zusätzlich zu den Standard-Claims können auch eigene Claims definiert werden (in Abbildung 2-1 z.B. "preferred_username"). Dies ermöglich eine hohe Flexibilität und dadurch, dass der Payload wie auch die Signatur Base64URL kodiert sind, einen für das Internet gut nutzbaren Token (kurz, einfache Struktur durch JSON …).
- Als letzter Bestandteil des JWT stellt die Signatur sicher, dass das JWT unverändert ist und auch von der erwarteten Stelle generiert wurde. Dafür wird der vorausgegangene Teil des Tokens (schon kodiert und mit einem „.“ getrennt) mit Hilfe des im Header definierten Hash-Algorithmus signiert. Die hier verwendeten Hash-Verfahren können variieren (z.B. HS[256,384,512], RS[256,384,512] etc.). Die Signatur wird ebenfalls Base64URL kodiert. In der Signatur ist maßgeblich der Algorithmus (siehe „alg“-Claim im Header) zur Signierung relevant. Ob der verwendete Schlüssel symmetrisch oder asymmetrisch ist, ist für JWT selbst uninteressant.

Access Tokens müssen bei Transport und Speicherung vertraulich behandelt werden. Die einzigen Parteien, die die Token sehen dürfen, sind die Anwendung (Client) selbst, der Authorization Server und der Resource Server. Die Anwendung sollte sicherstellen, dass Access Tokens für andere Anwendungen auf demselben Gerät unzugänglich sind. Außerdem sollten Access Tokens nur über sichere Verbindungen wie HTTPS transportiert werden, da die Weitergabe über einen unverschlüsselten Kanal das Access Token im Klartext zeigt und vom Dritten direkt abgefangen werden kann. Alternativ kann die Verschlüsselung des JWT benutzt werden, die als JSON Web Encryption (JWE) bezeichnet[12].

Basierend auf OAuth 2.0 ergänzt der OpenID Connect-Standard eine Identitätsschicht. Bei dieser handelt es sich im Wesentlichen darum, Token-basierte Authentifizierung von Usern mittels *ID Token*[13] zu

[9] https://datatracker.ietf.org/doc/html/rfc4648
[10] https://datatracker.ietf.org/doc/html/rfc7519#section-4.1
[11] https://tools.ietf.org/html/rfc7519#section-4
[12] https://tools.ietf.org/html/rfc7516
[13] https://openid.net/specs/openid-connect-core-1_0.html#IDToken

ermöglichen. Eine mögliche Umsetzung von ID Token ist ebenfalls JWT, welches auch mit JWS signiert werden kann. Für ID Token gelten die gleichen Sicherheitsrichtlinien wie für Access Token. Hier muss insbesondere auch auf sicheren Transport und vertrauliche Speicherung geachtet werden. Im Vergleich zum Access Token, das die Information über eine Berechtigung enthält, sind in ID Token nur die persönlichen Informationen (z.B. Vor- und Nachname, Adresse, E-Mail) und die sogenannten Identitätsattributen (z.B. URI-basierter Identifier)[14] hinterlegt.

Beispiel zur Nutzung von ID Token: Eine Web-App hat sich als OAuth-Client bei einem Identity Provider (IdP) registriert. In demselben IdP ist ein User registriert, der sich über die Web-App anmelden möchte, um eine geschützte Web-API aufzurufen. Nachdem einer der OAuth 2.0-Autorisierungsflüsse durchgeführt worden ist, erhält die Web-App zunächst ein ID Token, das Vor- und Nachname, E-Mail-Adresse und Telefonnummer des Users enthält. Damit wird die Web-App darüber informiert, dass sich der im Token enthaltene User schon beim IdP authentifiziert hat. Anschließend erhält die Web-App ein Access Token, das die Web-App nutzt, um auf die Web-API zuzugreifen.

Es lässt sich zusammenfassen:

- ID Token (*für Authentifizierung*) enthält die Identitätsinformationen eines Users und wird meist vom Client genutzt.
- Access Token (*für Autorisierung*) enthält die Berechtigungsinformationen eines Clients im Auftrag eines Users und wird meist vom Resource Server genutzt.

2.2.2 Refresh Token vs. Offline Token

Refresh Token[15] *sind* Token, die eine lange Lebensdauer haben können und vom Client verwendet werden, um beim Authorization Server ein neues *Access Token* anzufordern, wenn das aktuell genutzte Access Token abgelaufen ist oder um eine Änderung an seinen Scopes (siehe Abschnitt 3.4) zur Anpassung an einen veränderten Anwendungsfall vorzunehmen. Refresh Token wird immer zusammen mit dem Access oder ID Token abgeholt.

Offline Token[16] werden in OpenID Connect eingeführt und sind spezielle Refresh Token zur Anfrage nach einem neuen Access Token. Sie sollen aber im Gegensatz zu Refresh Token **nie** ablaufen. Wenn ein Offline Token von einer Anwendung im Namen eines Users angefordert wird, muss die Anwendung es sicher abspeichern und kann es später verwenden, auch wenn der User sich schon von der betreffenden Anwendung abgemeldet hat. Dies ist nützlich, wenn die Anwendungen einige „Offline"-Aktionen im Namen des Resource Owners (User) durchführen müssen. Ein Beispiel ist ein periodisches Backup der Daten jede Nacht in einer Datenbank.

2.3 Flows

Im vorliegenden Abschnitt werden die klassischen Autorisierungsflüsse (auch „Grant-Typ") von OAuth 2.0 sowie deren Erweiterung in OpenID Connect vorgestellt.

[14] https://www.c-sharpcorner.com/article/accesstoken-vs-id-token-vs-refresh-token-what-whywhen/
[15] https://auth0.com/learn/refresh-tokens/
[16] https://github.com/keycloak/keycloak-documentation/blob/main/server_admin/topics/sessions/offline.adoc

2.3.1 Authorization Code

Der Grant-Typ *Authorization Code*[17] wird meist von Web- und mobilen Apps verwendet. Bei der Autorisierung fordert die Applikation (Client) im Namen des Resource Owners (User) den Zugriff auf die geschützte Ressource an.

Bei der Autorisierung wird der Resource Owner zunächst über einen Browser zum Authorization Server weitergeleitet, wo er seine Anmeldedaten und einen sogenannten „state"-Parameter (beliebiger String) eingeben kann. Der Authorization Server leitet aus den Anmeldedaten und dem „state" einen Authorization Code ab. Anschließend wird der Resource Owner mit dem ausgestellten Authorization Code zur Applikation zurückgeleitet. Die Applikation tauscht zum Abschluss beim Authorization Server den Authorization Code zusammen mit ihrer Client-ID und ihrem Secret gegen ein Access Token ein.

Die Nutzung dieses Grant-Typs zeichnet sich dadurch aus, dass die Anmeldedaten des Resource Owners (User) niemals mit der Applikation (Client) geteilt werden und das ausgestellte Access Token auch niemals für den Resource Owner (User) sichtbar ist.

Der Fluss ist in Abbildung 2-2 dargestellt.

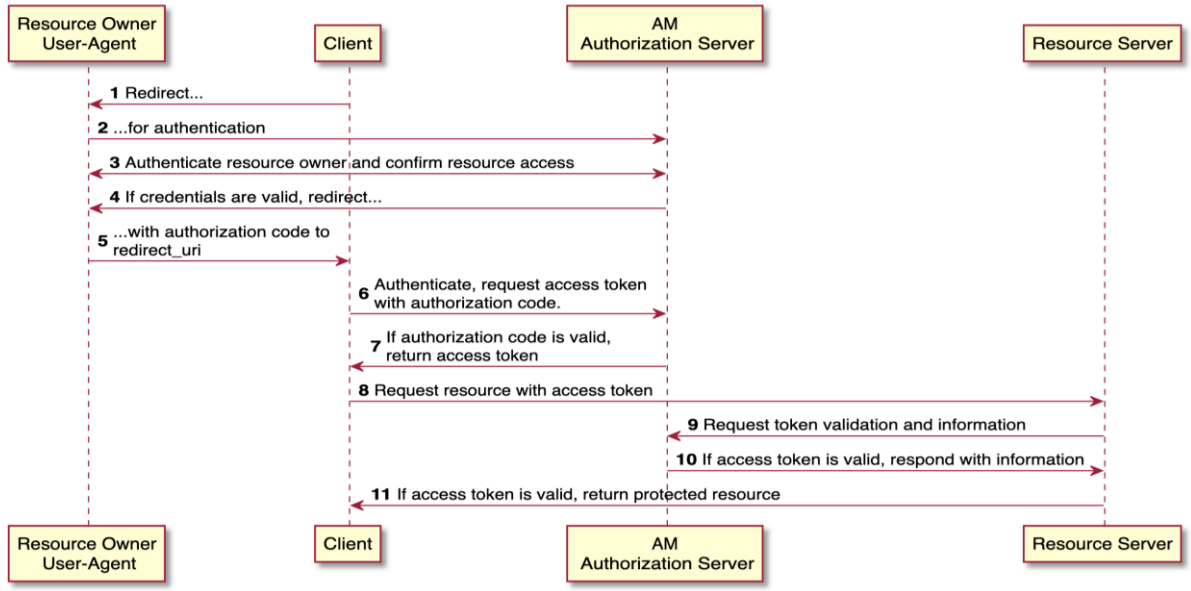

Abbildung 2-2: Authorization Code Flow[18]

2.3.2 PKCE

Proof Key for Code Exchange (kurz PKCE)[19] ist eine Erweiterung des Flows Authorization Code (siehe Abschnitt 2.3.1), um Public Clients (siehe Abschnitt 3.3) beim Austausch mit dem Authorization Server vor möglichen Angriffen (z.B. Injection-Angriff eines Authorization Code) zu schützen. Der PKCE-Grant unterscheidet sich vom Authorization Code Flow hauptsächlich darin, dass bei der Anfrage nach einem Authorization Code ein *Code Verifier* statt dem oben genannten „state" benutzt wird und dabei kein Client Secret (nicht verfügbar für Public Clients, siehe Abschnitt 3.3) zum Eintauschen erforderlich ist.

Wenn eine Applikation die Autorisierung beginnt, erstellt sie zunächst einen sogenannten Code Verifier, anstatt einen Browser zu starten. Der Code Verifier ist eine kryptografisch zufällige Zeichenfolge,

[17] https://auth0.com/docs/authorization/flows/authorization-code-flow
[18] https://developer.forgerock.com/docs/platform/how-tos/script-executing-oauth2-authorization-code-flow-pkce-am
[19] https://developer.okta.com/docs/guides/implement-grant-type/authcodepkce/main/#grant-type-flow

die weiter zur Generierung einer *Code Challenge* verwendet wird. Bei Geräten, die einen SHA256-Hash durchführen können, ist die Code Challenge eine BASE64-kodierte Zeichenfolge des SHA256-Hashs des Code Verifiers. Die Code Challenge und Hash-Methode werden zusammen mit den anderen erforderlichen Parametern an den Authorization Server zur Anfrage nach einem Authorization Code geschickt. Sobald der Authorization Code zurückgeschickt wird, holt sich die Applikation ein Access Token ab, indem sie den Authorization Code zusammen mit dem Code Verifier wiederum an den Authorization Server schickt. Der Authorization Server prüft, ob der Code Verifier und Code Challenge zusammenpassen, damit die Identität des ursprünglichen Anfragenden verifizieren und so potenzielle Injection-Angriffe erkennen.

Der Fluss ist in der nachfolgenden Abbildung 2-3 dargestellt.

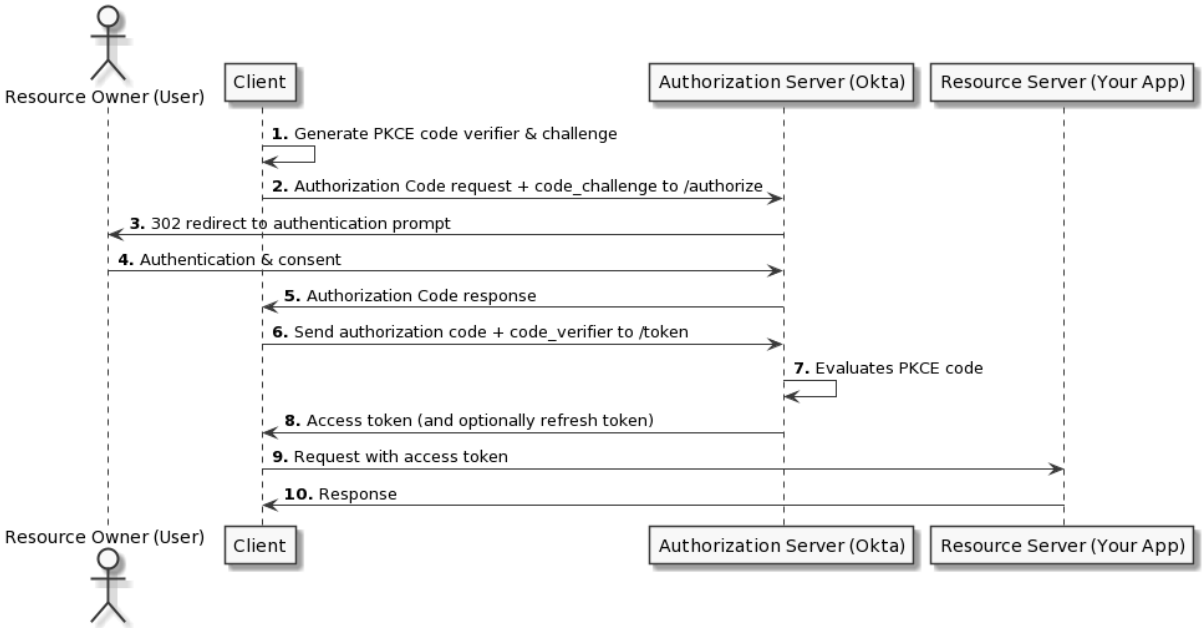

Abbildung 2-3: Authorization Code Flow mit PKCE[20]

2.3.3 Client Credentials

Wenn eine Applikation (Client) selbst der Resource Owner ist, fordert sie den Zugriff auf ihre Ressourcen im eigenen Namen beim Authorization Server an. In diesem Fall wird der Grant-Typ *Client Credentials*[21] für die Autorisierung der Applikation verwendet und damit ist keine Autorisierung eines Users erforderlich. Dazu müssen der Applikation ihre Anmeldedaten (Client ID und Secret) bekannt sein.

Der Fluss ist in Abbildung 2-4 illustriert.

[20] https://developer.okta.com/docs/guides/implement-grant-type/authcodepkce/main/
[21] https://auth0.com/docs/authorization/flows/client-credentials-flow

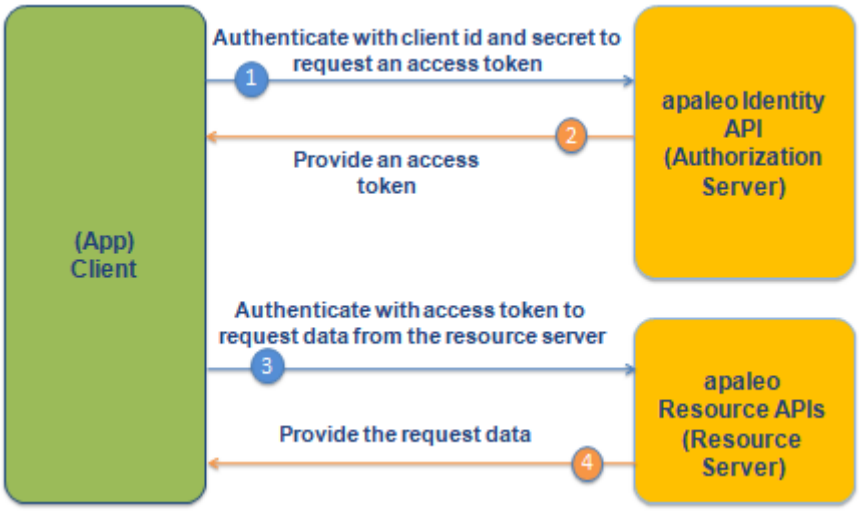

Abbildung 2-4: Client Credentials Flow[22]

2.3.4 Device Authorization Grant

Der Grant-Typ *Device Authorization Grant*[23] wird bei browserlosen oder eingabebeschränkten Geräten wie z.B. Apple TV verwendet. Bei diesem Grant wird ein zuvor generierter Device Code beim Authorization Server gegen ein Access Token eingetauscht.

Der Client muss zunächst einen Device Code beim Authorization Server anfordern. Dieser Code wird gemeinsam mit einer *Verification URL* und einem *User Code* an den Client zurückgeschickt. Die Verification URL wird vom User auf einem anderen Browser-fähigen Gerät aufgerufen. Dort muss er auch den generierten User Code eingeben. Bei erfolgreicher Autorisierung wird das Access Token dann für den Client zur Abholung hinterlegt.

Dieser Flow hat zwei verschiedene Umsetzungen. Eine kann auf eingabebeschränkten Geräten stattfinden und die andere passt zur Nutzung mit einem Browser. Der Fluss ist in Abbildung 2-5 dargestellt.

[22] https://apaleo.dev/guides/start/oauth-connection/client-credentials-grant
[23] https://oauth.net/2/device-flow/

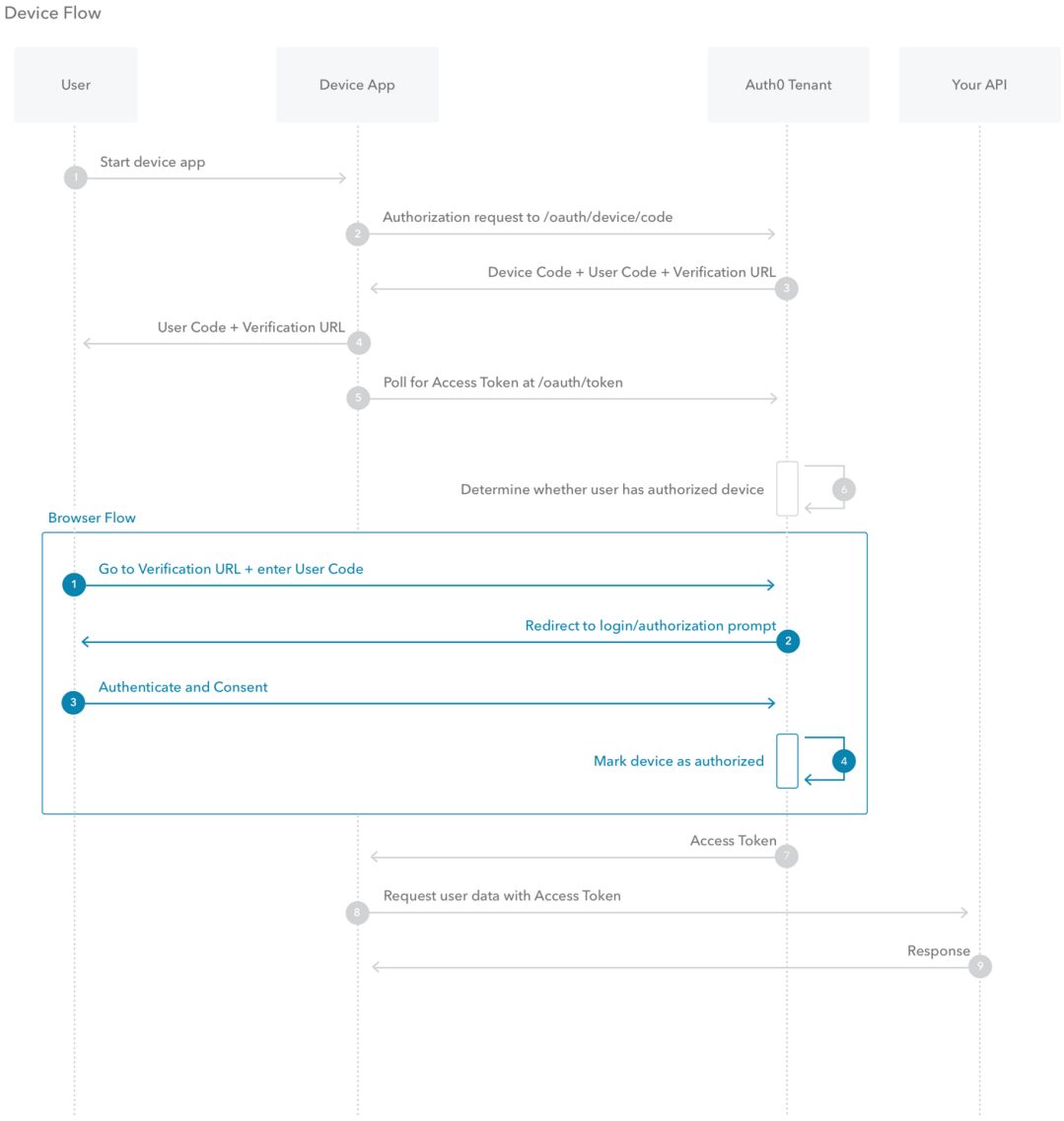

Abbildung 2-5: Device Authorization Flow[24]

2.3.5 Resource Owner Password Credentials

Bei diesem Grant-Typ[25] werden die Anmeldedaten des Resource Owners direkt beim Authorization Server gegen ein Access Token eingetauscht. Ein wesentlicher Nachteil der Nutzung dieses Grant-Typs ist, dass die Applikation die Anmeldedaten entgegennehmen muss, um diese an den Authorization Server weiterzuleiten. Dieser Grant-Typ wird im aktuellen OAuth 2.1-Protokoll komplett untersagt.

2.4 Neuerungen in OAuth 2.1

OAuth 2.1[26] ist ein Work in Progress (WIP)-Protokoll. Unterschiede zu OAuth 2.0 sind hauptsächlich:

- PKCE (Abschnitt 2.3.2) soll statt Authorization Code Flow (Abschnitt 2.3.1) verwendet werden
- Implicit Grant[27] und Resource Owner Password Credentials (Abschnittt 2.3.5) werden nicht mehr unterstützt

[24] https://auth0.com/docs/authorization/flows/device-authorization-flow
[25] https://auth0.com/docs/authorization/flows/resource-owner-password-flow
[26] https://fusionauth.io/blog/2020/04/15/whats-new-in-oauth-2-1/
[27] https://oauth.net/2/grant-types/implicit/

- Refresh Tokens müssen für Public Clients entweder absenderbeschränkt oder einmalig verwendbar sein
- Neue RFCs für die Entwicklung von OAuth-Clients und die Implementierung von Authorization Servern. Z.B. RFC 8252[28] OAuth 2.0 für Native Apps, RFC 7636[29] Proof Key for Code Exchange (PKCE, siehe Abschnitt 2.3.2), OAuth 2.0 for Browser-based Apps[30] und OAuth 2.0 Security Best Current Practice[31]

3 Keycloak-Features im S³I-IdentityProvider

Der S³I-IdentityProvider stellt den zentralen Dienst zur Verwaltung von Identitäten von Personen und WH4.0-Dingen[32] dar und ermöglicht die Authentifizierung und Autorisierung unter der Anwendung von OpenID Connect (basierend auf OAuth 2.0, vergleiche Abschnitt 2.1). Technische Umsetzung des S³I-IdentityProvider erfolgt über die Open-Source-Software Keycloak[33].

Im vorliegenden Kapitel werden die im S³I-IdentityProvider verwendeten grundlegenden Features von Keycloak und deren konkrete Abbildung auf S³I vorgestellt.

3.1 Realm

Ein Realm verwaltet einen Satz von Usern, Clients, Rollen und Gruppen. In Keycloak können mehrere Realms angelegt werden, die aber voneinander unabhängig sind. User, Clients, Rollen und Gruppen werden nur in dem Realm verwaltet, in welchem sie registriert sind.

Im S³I-IdentityProvider ist ein Realm (KWH) explizit für die Verwaltung der Identitäten von WH4.0-Dingen und Personen angelegt.

3.2 User

Ein User (auch Endbenutzer) ist eine Person, die sich zur Wahrnehmung ihrer Aufgaben einer Datenverarbeitungsanlage bedient und dabei in unmittelbarem Kontakt mit der Anlage steht[34]. In Keycloak wird der Begriff des Users weiter spezifiziert. Dabei werden User als Entitäten bezeichnet, die im zentralen IdentityProvider registriert sind, wo ihre Identität verwaltet wird. Mit diesen Identitäten können sie sich weiter in ihre Anwendung einloggen. User können sich mit ihren Attributen wie E-Mail, Username, Adresse, Telefonnummer, Geburtsdatum etc. assoziieren und einen (URI-basierten) einzigartigen Identifier bekommen, der gemeinsam mit den User-Attributen als Identitätsinformationen eines Users abgebildet werden können. Zudem können User Gruppen und Rollen zugewiesen werden.

3.3 Client

Ein Client bezeichnet ein Computerprogramm, das auf dem Endgerät eines Netzwerks ausgeführt wird und mit einem Server kommunizieren kann[35].

Im Kontext von OAuth und OpenID Connect bezeichnet ein Client eine Anwendung, die den Zugriff auf (geschützte) Ressourcen im Namen des Resource Owners anfordern kann. Die in Keycloak angelegten Clients sind in der Lage, die Authentifizierung eines Users, der die Anwendung nutzen will, nach den in OAuth 2.0 definierten Kommunikationsflüssen anzufordern (Authorization Code Grant, siehe 2.3.1).

[28] https://datatracker.ietf.org/doc/html/rfc8252
[29] https://datatracker.ietf.org/doc/html/rfc7636
[30] https://oauth.net/2/browser-based-apps/
[31] https://oauth.net/2/oauth-best-practice/
[32] WH4.0-Dinge = WH4.0-Komponenten (Assets wie Maschinen, Geräte, Sensoren, Personen und ihre Digitalen Zwillinge) + WH4.0-Dienste (Software-Dienste) + WH4.0-Mensch-Maschine-Schnittstellen (MMS)
[33] https://www.keycloak.org/
[34] https://de.wikipedia.org/wiki/Benutzer#cite_note-InfDud-2
[35] https://de.wikipedia.org/wiki/Client#cite_note-1

Des Weiteren können sich Clients als Resource Owner selbst beim Authorization Server authentifizieren, um andere Dienste, die auch durch Keycloak geschützt sind, sicher aufzurufen (Client Credentials, siehe 2.3.3).

3.4 Scopes

Scopes sind Entitäten in Keycloak zur Beschreibung konkreter Rechte, die vom Resource Owner einem Client gewährt werden kann und deren Bedeutung beim Resource Server bekannt sein sollen. Beispielsweise möchte ein User über seine App eine bestimmte API aufrufen. Dafür muss er bei der Token-Anfrage den entsprechenden vorkonfigurierten Scope z.B. API-PUT schon angeben, damit dieser Scope im Token auftaucht. Der Resource Server validiert dann das Token insb. den „scope“-Claim. Weil das im Beispiel genannte „APP-PUT“ im Scope des Tokens enthalten ist, gibt der Server der anfragenden App die Ressourcen frei.

4 S³I-Autorisierung

Das vorliegende Kapitel beschäftigt sich mit der bisherigen Umsetzung der Autorisierung in S³I. Sie lässt sich in Abhängigkeit vom konkreten Anwendungsfall weiter in zentrale (siehe Abschnitt 4.1 und 4.2.2) sowie dezentrale (siehe Abschnitt 4.2.1) Autorisierung unterteilen.

Die in Abschnitt 2.1 eingeführten OAuth2.0-Rollen können hiermit in die WH4.0-Anwendungsfälle gut übertragt werden. Nur darauf zu achten ist, dass in klassischen OAuth Szenarien immer Nutzer (Resource Owner) auf ihre eigenen Sachen zugreifen wollen. In WH4.0 und hier im speziellen ist es aber oft so, dass ein Dritter auf eine Ressource zugreifen will und dazu autorisiert wurde. Ihm „gehört“ (Owner) die Ressource also nicht wirklich. Er hat halt nur Zugriffsrechte. Dasselbe wie beim Baumartenklassifikationsdienst. Da hat der Waldarbeiter vielleicht nur ein Konto und darf ihn daher benutzen. Aber der Dienst „gehört“ ihm deswegen nicht.

4.1 In zentralen S³I-Diensten

Die hier genannten Softwaredienste sind diejenigen zentralen Dienste, die die grundlegenden Features des S³I bereitstellen (S³I-Directory, S³I-Broker …). Die Authentifizierung und Autorisierung bei diesen Diensten erfolgt über *Access Token*, die vom S³I-IdentityProvider zentral ausgestellt werden. In den folgenden Abschnitten wird erläutert, wie Access Token bei den jeweiligen Diensten konkret genutzt werden.

4.1.1 S³I-IdentityProvider

Die Identität der Person (PersonIdentity[36]) wird durch Keycloak-User im S³I-IdentityProvider abgebildet. Hierzu wird im wesentlichen Username/Password verwaltet. Die mit der PersonIdentity assoziierten Attribute (z.B. Vor- und Nachname, E-Mail-Adresse) werden nicht direkt im Keycloak eingestellt, sondern zentral im S³I-Directory ForestML4.0-konform gespeichert.

Darüber hinaus gibt es das Konzept der ThingIdentity. Diese wird im S³I-IdentityProvider durch Keycloak-Clients abgebildet. Für jedes WH4.0-Ding (z.B. Baumartenklassifikationsdienst, iWald-App, Jupyter-HMI, rSFL-Harvester) wird ein einzigartiger Client mit Client ID und Secret registriert, der die Identität des entsprechenden WH4.0-Dings repräsentiert.

4.1.2 S³I-Directory und -Repository

S³I-Directory und -Repository basieren beide auf dem Open-Source-System Eclipse Ditto und nutzen dessen rollenbasierten Autorisierungsmethoden. Im Kontext von OAuth 2.0 stellen beide Resource Server dar. Für den Zugriff auf diese Ressourcen muss ein Berechtigungsnachweis in Form eines Access

[36] https://www.kwh40.de/wp-content/uploads/2020/04/KWH40-Standpunkt-S3I-v2.0.pdf

Tokens vorgelegt werden. Der Dienst validiert das *Access Token* und überprüft dort insbesondere die folgenden Claims:

- „exp“-Claim weist darauf hin, ob das Token abläuft.
- „iss“-Claim stellt dar, wer das Token ausstellt.
- „sub“-Claim stellt die User-ID der anfragenden Partei (d.h. des Token-Halters) beim „Authorization Code Grant“ dar. Bei „Client Credentials Grant“ authentifizieren sich die Clients eigenständig und der „sub“-Claim wird in dem Fall nicht mehr berücksichtigt, weil dort nur eine interne UUID steht (nicht Client-ID, die einen „s3i:“-Namespace hat), die von Keycloak bei der Registrierung automatisch zugewiesen wird.
- „azp“-Claim stellt die Client-ID dar. Dem durch diese ID repräsentierten Client wird erlaubt, im Namen des Resource Owners auf die Ressourcen zuzugreifen
- „group“-Claim beschreibt die Gruppe der anfragenden Partei, falls dieser schon eine Gruppe zugewiesen ist.

Die Zugriffrechte werden durch die sogenannten Policy-Dateien im JSON-Format beschrieben, mit denen es ermöglicht wird, auf einfache Weise eine fein abgestufte Zugriffskontrolle für jedes WH4.0-Ding zu konfigurieren. In jeder Policy werden diverse Rollen definiert, die unterschiedliche Nutzungsrechte auf den Thing-Eintrag haben können. Beispielsweise hat die standardmäßige Rolle „Owner“ vollständige Rechte (Schreiben und Lesen). Rollen können den verschiedensten Parteien (User, Client und Gruppe) zugewiesen werden. Abbildung 4-1 illustriert ein Beispiel der Policy-Datei. In dem grünen Kästchen steht die definierte Rolle und darunter die ID des WH4.0-Dings (im roten Kästchen) sowie die für die Rolle freigegebenen Rechte (im blauen Kästchen).

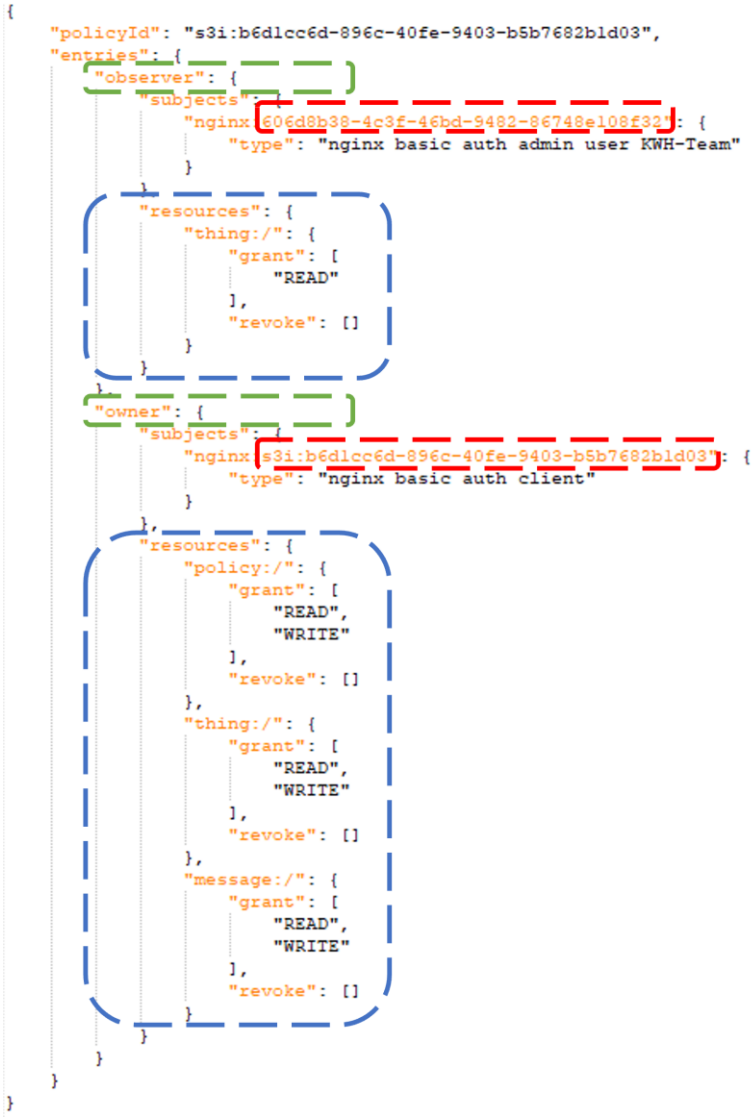

Abbildung 4-1: Beispiel: Policy

Welche Claims bei der Validierung des Tokens berücksichtigt werden, hängt vom verwendeten Grant-Typ ab. Im Kontext von Wald und Holz 4.0 werden folgende Kommunikationsszenarien häufig genutzt:

- **Kommunikation zwischen einem Menschen und dem S³I-Directory/-Repository**: Waldbesitzer benutzt sein Python-HMI, um den Thing-Eintrag seines Harvesters abzufragen. Waldbesitzer holt sich über sein HMI ein Access Token ab und fügt es in einen HTTP-Header ein. In demselben HTTP-Request trägt er die Thing-ID des Harvester ein, um auf den Thing-Eintrag des Harvester zuzugreifen. Wenn das S³I-Directory (Resource Server) den HTTP-Request erhält, wird zunächst der "sub"-Claim des Tokens betrachtet. Dieser gibt an, für wen dieses Token ausgestellt wurde bzw. wer der Token-Halter ist. Um die Zugriffsrechte festzustellen, müssen alle Einträge der Policy-Datei mit der User-ID aus dem „sub"-Claim abgeglichen werden. Da der Waldbesitzer als „Owner" (standardmäßige Rolle; diese kann auf den Thing-Eintrag lesend und schreibend zugreifen) in der Policy-Datei eingetragen ist, wird die angefragten Ressource für ihn freigegeben.
- **Kommunikation zwischen einem Menschen (Gruppenmitglied) und dem S³I-Directory/-Repository**: In demselben Beispiel wie oben greift der Waldbesitzer jetzt auf den Thing-Eintrag des FBZ-Forwarders zu. Dieser Eintrag ist aber aktuell nur für FBZ-Mitglieder zugänglich. Weil der Waldbesitzer dieser Gruppe angehört, gibt er bei der Anfrage nach dem Access Token in

dem Scope „group“ an. Somit wird seine Gruppenzugehörigkeit im „group“-Claim im Access Token aufgeführt. Das S³I-Directory überprüft dann den „group“-Claim und gibt die Ressourcen für den Waldbesitzer frei. (Hinweis: Das Gruppenkonzept für S³I ist noch in Entwicklung)

- **Kommunikation zwischen einem Dienst und dem S³I-Directory/-Repository**: Harvester will auf den Thing-Eintrag des Forwarder zugreifen, um dessen Endpunkt zu erfragen. Der Harvester tauscht seine Client-Credentials gegen ein Access Token beim S³I-IdP ein und bereitet einen HTTP-Request vor, welcher an das S³I-Directory gesendet wird. In diesem Fall schaut sich das Directory den „azp“-Claim statt des „sub“-Claim des Tokens an, weil es sich bei dieser Autorisierung beim S³I-IdP um den Grant-Typ „client credentials“ handelt. Die Client-ID des Harvester erscheint somit im „azp“-Claim. Anschließend daran wird die Client-ID mit allen Einträgen der Policy abgeglichen. Damit wird festgestellt, welche Ressourcen für ihn verfügbar sind.

4.1.3 S³I-Broker

Der S³I-Broker stellt einen optionalen Dienst für die Nachrichten-basierte Kommunikation zwischen WH4.0-Dingen dar. Für die Authentifizierung und Autorisierung beim S³I-Broker werden *Access Token* genutzt. Damit ist der S³I-Broker auch ein Resource Server. Im S³I-Broker wird das AMQP-Protokoll[37] genutzt. Grundlegendes Konzept für AMQP-Massaging ist in Abbildung 4-2 schematisch dargestellt.

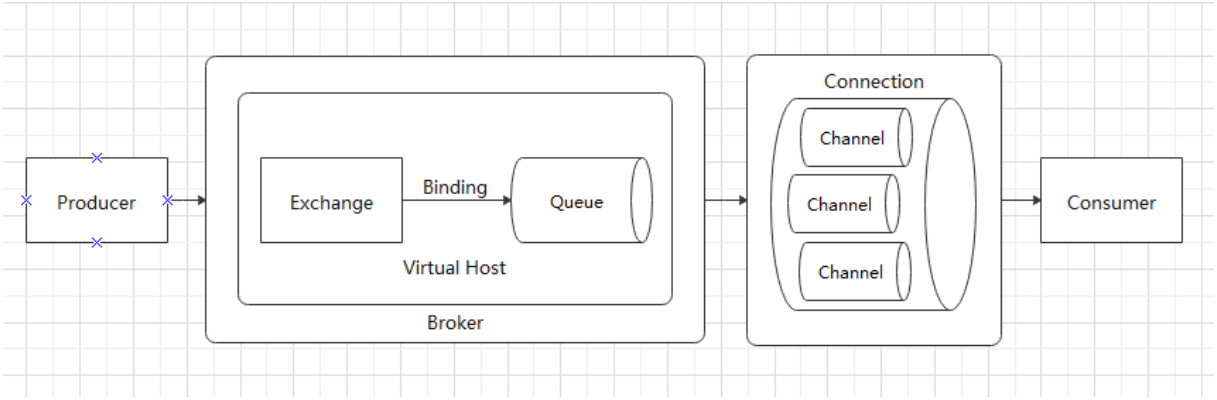

Abbildung 4-2: grundlegendes Konzept für AMQP-Messaging

Die grundlegenden Komponenten (Exchange, Queue, Binding, etc.) von AMQP sind in eine „virtuelle“ Gruppe unterteilt, dies mit dem Konzept des Namespace im Netzwerk verglichen werden kann. Exchange ist die Entität in AMQP, an den die Nachricht ursprünglich geschickt wird. Gemäß dem vorkonfigurierten Binding, welcher den Exchange mit einer oder mehrere Queues verbindet, wird die Nachricht vom Exchange weiter an den entsprechenden Endpunkt (Queue) geroutet. Queue bildet sich als Warteschlange und speichert die Nachricht, bis sie vom Consumer (Empfänger) abgeholt wird.

Folgende Claims in *Access Token* sind dabei besonders wichtig:

- „aud“-Claim bezeichnet den „Zuhörer“ oder Ziel des Tokens – hier muss der Claim „rabbitmq“ lauten.
- „exp“-Claim weist darauf hin, ob das Token abläuft.
- „iss“-Claim stellt dar, wer das Token ausstellt.
- „scope“-Claim stellt die Nutzungsrechte des anfragenden WH4.0-Dings dar. Die Rechte werden als Client-Scopes in S³I-IdP vorkonfiguriert. Sie beziehen sich auf „read“, „write“ und „configure“-Scopes

[37] https://www.amqp.org/

Mit dem „read“-Scope dürfen WH4.0-Dinge Nachrichten aus einer Queue vom S³I-Broker abholen. Z.B. *read:s3i/demo_exchange/s3ibs://s3i:4711* bezeichnet die Berechtigung, mit der alle Nachrichten, die an die Queue s3ibs://s3i:4711 geschickt werden, empfangen werden dürfen.

Mit dem „write“-Scope dürfen WH4.0-Dinge Nachrichten an eine Queue im S³I-Broker schicken. Z.B. *write:s3i/demo_exchange/** bezeichnet die Berechtigung, damit Nachrichten an alle Queues über „demo_exchange“ im „s3i“-Vhost geschickt werden können.

Der „configure“-Scope bezieht sich auf die Konfiguration einer Queue oder eines Exchanges im S³I-Broker. Beispielsweise wird dieses Recht für die Erstellung eines neuen Exchanges benötigt[38].

Es ist darauf zu achten, dass in der aktuellen Nutzung von *Access Tokens* nur die o.g. Claims validiert werden, d.h. die Identität des Nachrichtsenders („sub“-Claim) nicht erneut beim Broker überprüft wird. Es gilt die Annahme, dass ein vom S³I-IdP ausgestelltes und erfolgreich beim S³I-Broker validiertes Access Token sicherstellt, dass eine erfolgreiche Authentifizierung beim S³I-IdP stattgefunden hat und eine weitere Überprüfung der Identität beim Broker nicht mehr nötig ist. Des Weiteren erfolgt die Kommunikation im Broker end-to-end und ggf. verschlüsselt. Der Broker selbst kann Nachrichten nicht einsehen, wenn die Nachrichten verschlüsselt sind. Aus diesem Grund kennt er den in der Nachricht angegebenen Sender nicht, welchen er zur Validierung der Identität im „sub“-Claim verwenden müsste.

Hinweis: Die Ende-zu-Ende-Verschlüsselung von S³I-Broker-Nachrichten mit PGP wird im allgemeinen S³I Standpunkt beschrieben.

Hinweis: Die dem S³I-Broker vorgeschaltete REST API leitet HTTP-Anfragen inkl. der Token an den Broker weiter und realisiert daher keine eigenen Authentifizierungsmechanismen.

4.1.4 S³I-Registration

Die aktuelle technische Umsetzung von S³I-Registration erfolgt über einen REST-basierten Webdienst, S³I-Config. Dieser Dienst wird auf Grundlage von Python-Flask[39] entwickelt mit dem Ziel, Personen und WH4.0-Dinge koordiniert mit allen zentralen S³I-Diensten zu verwalten (Registrieren, Ändern und Löschen).

Die Authentifizierung und Autorisierung bei S³I-Registration erfolgen ebenfalls über *Access Token.* Folgendes Claims werden dabei validiert:

- „sub“-Claim stellt den anfragenden User dar.
- „azp“-Claim stellt die Client-ID dar. Dem durch diese ID repräsentierten Client wird erlaubt, im Namen des Resource Owners auf die Ressourcen zuzugreifen
- „exp“-Claim weist darauf hin, ob das Token abläuft.
- „iss“-Claim bezeichnet den Autorization Server, der das Token ausgestellt hat.

Im Folgenden wird definiert, wie die Ressourcen für den anfragenden User freigegeben werden:

- Jeder kann sich registrieren.
- Eine PersonIdentity darf nur von dem repräsentierten User geändert bzw. gelöscht werden.
- Eine ThingIdentity darf nur von dem besitzenden User angelegt, geändert bzw. gelöscht werden. Die Beziehung zwischen WH4.0-Dingen und User werden im S³I-Directory abgebildet.
- Die Konfiguration eines WH4.0-Dings (z.B. Queue anlegen, Cloud-Kopie im S³I-Repository anlegen) kann nur vom Besitzer verwaltet werden.

[38] https://www.rabbitmq.com/access-control.html
[39] https://flask.palletsprojects.com/en/2.0.x/

4.2 In WH4.0-Dingen

Im Fall, dass WH4.0-Dinge selbst Resource Server sind, findet die Autorisierung des Resource Owners entweder dezentral beim jeweiligen WH4.0-Ding oder zentral beim S³I-IdentityProvider statt. Folgende Anwendungsfälle sind im Kontext von Wald und Holz 4.0 konkretisiert:

- **Kommunikation zwischen Person und (nicht Person-DZ und MMS-) WH4.0-Ding**: User Waldarbeiter (Resource Owner) will über seine HMI (Client) den Baumartenklassifikationsdienst (Resource Server) aufrufen.
- **Kommunikation zwischen WH4.0-Dingen**: Harvester (Resource Owner von sich selbst sowie seinen Daten und dabei auch Client) will die aktuelle Lage des Forwarders (Resource Server) abfragen.

Des Weiteren gibt es noch einige speziellen Fälle, wo mehrere Resource Server in einer Kommunikation beteiligt sind (kaskadierte Autorisierung, Konzept ist noch in Bearbeitung):

- **Kommunikation zwischen Person und WH4.0-Dingen**: User Waldarbeiter (Resource Owner) will über seine HMI (Client) den Baumartenklassifikationsdienst (Resource Server) aufrufen. Der Baumartenklassifikationsdienst muss für die konkrete Nutzung weiter auf das Vertex-Messgerät (Resource Server) und die elektronische Messkluppe (Resource Server) zugreifen.
- **Kaskadierte Kommunikation zwischen WH4.0-Dingen**: Harvester (Resource Owner, Client) will den Befahrbarkeitsdienst (Resource Server) aufrufen. Für die konkrete Nutzung muss der Befahrbarkeitsdienst die Bodenfeuchte eines Bodenfeuchtesensors (Resource Server) abfragen.

4.2.1 Dezentrale Autorisierung

Bei dezentraler Autorisierung verwaltet ein WH4.0-Ding die Zugriffsrechte aller anderen WH4.0-Dinge, die auf ihre Ressourcen zugreifen dürfen, eigenständig. Bei Anfragen werden die Berechtigungen der anfragenden Resource Owner in dem entsprechenden WH4.0-Ding eingesehen und die jeweiligen Ressourcen freigegeben.

Zur Vernetzung von WH4.0-Dingen werden Kommunikationsprotokolle wie OPC UA, MQTT, AMQP, REST und S³I-B verwendet. Die Integration der Verschlüsslung sowie der Signierung gemäß RFC4880[40] in das S³I-B-Protokoll garantiert die Zuverlässigkeit und Integrität der Nachrichten beim Datenaustausch. Im Nachfolgenden wird weiter darauf eingegangen, wie sich ein User durch die Nutzung der Signierung in einer S³I-B-Nachricht authentifiziert.

Die aktuelle Umsetzung der dezentralen Authentifizierung und Autorisierung in WH4.0-Dingen erfolgt über die Signierung[41] der S³I-B-Nachrichten. Jede Nachricht ist mit einem Sender versehen, welcher angibt, von wem die Nachricht erstellt und gesendet wurde. Durch die Signatur der Nachricht wird sichergestellt, dass es sich bei dem Nachrichtsender tatsächlich um den in der Nachricht angegebenen Sender handelt. Somit ist der Sender der Nachricht authentifiziert. Es ist darauf zu achten, dass die Nachricht unabhängig von der verwendeten WH4.0-MMS immer vom jeweiligen sendenden Benutzer (d.h. mit dessen privatem Schlüssel) signiert werden muss, der bei der anzufragenden Ressource Rechte besitzt.

Beispielsweise möchte der Waldbesitzer über seine HMI einen S³I-B-ServiceRequest an den Baumartenklassifikationsdienst senden. Seine HMI muss den privaten Schlüssel des Waldbesitzers kennen, um die Nachricht in seinem Namen zu signieren. Der Baumartenklassifikationsdienst verifiziert die Nachricht mit dem öffentlichen Schlüssel des Waldbesitzers, um den Nachrichtensender zu authentifizieren.

[40] https://datatracker.ietf.org/doc/html/rfc4880
[41] https://www.kwh40.de/wp-content/uploads/2020/04/KWH40-Standpunkt-S3I-v2.0.pdf

Sobald die Signatur verifiziert wird, werden die Nutzungsrechte des Waldbesitzers beim Baumartenklassifikationsdienst eingesehen. Damit wird festgestellt, ob der Dienst vom Waldbesitzer aufgerufen werden darf oder nicht.

Das nächste Beispiel bezieht sich auf die Kommunikation zwischen WH4.0-Dingen. Der Harvester will seine Produktionsdaten in eine Datenbank speichern. Dafür verfasst er einen S³I-B-ServiceRequest, welcher durch den privaten Schlüssel des Harvester direkt signiert wird. Sobald die Datenbank die signierte Nachricht erhält, verifiziert sie die Signatur und überprüft die Identität des in der Nachricht angegebenen Senders. Dadurch wird der Harvester bei der Datenbank authentifiziert. Die in der Datenbank vordefinierten Nutzungsrechte des Harvester stellten weiter fest, ob die Nutzung der Datenbank für den Harvester erlaubt ist oder nicht.

Die Formalisierung der Nutzungsrechte bei dezentralen WH4.0-Dingen muss unabhängig vom S³I von ihrem jeweiligen Besitzer festgestellt werden.

4.2.2 Zentrale Autorisierung

Das Konzept für die zentrale Autorisierung in S³I ist aktuell noch in der Entwicklung.