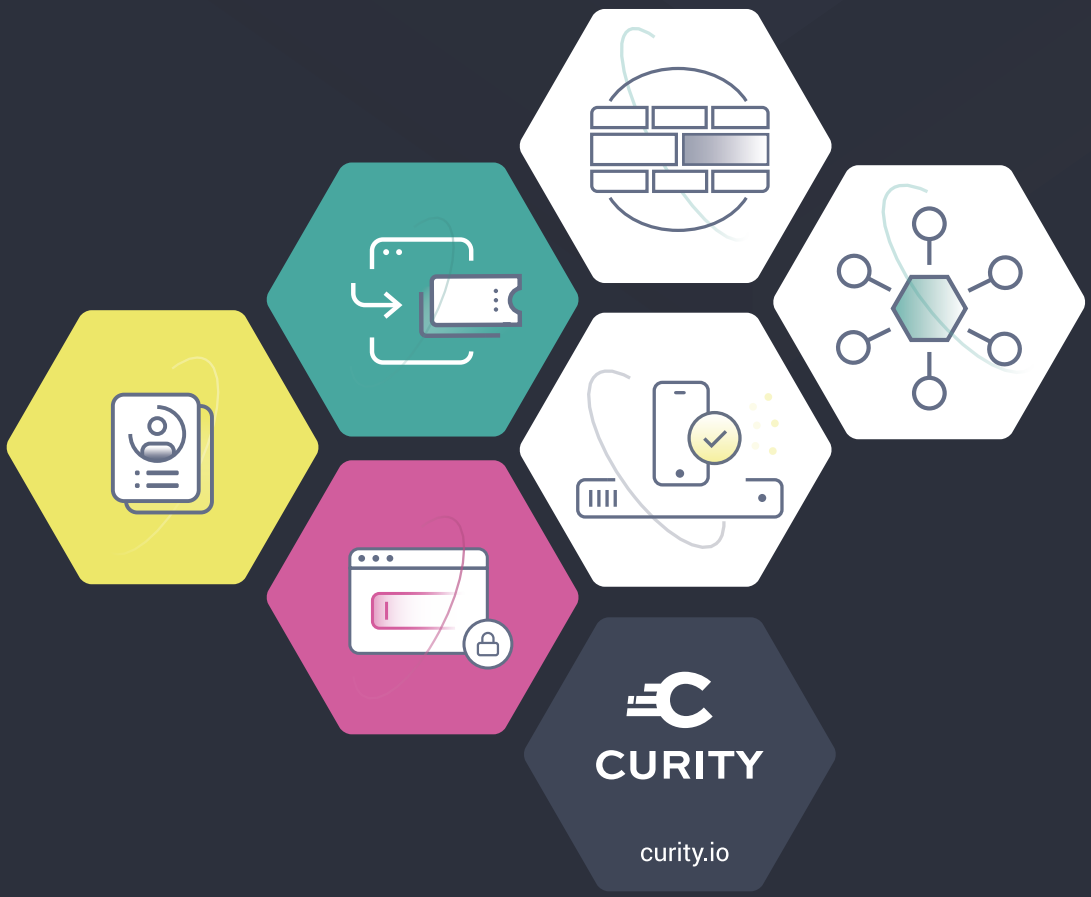
API Security
A Collection of Articles



CURITY

Published by Curity AB in association with Nordic APIs AB

Contact: info@curity.io and info@nordicapis.com

Table of Contents

The API Security Maturity Model 2

API Security: Deep Dive into OAuth and OpenID Connect 8

Using OAuth Within Microservices 17

Coarse Grained Authorization Using Scopes 23

Claims Based Authorization Using OAuth 28

Securing APIs in a Cloud Native Environment 33

Standardized User Management with SCIM 38

More on API Security 44

API Security for the Modern Enterprise

APIs have in recent years grown to be essential to the digital strategy of a modern organization. To ensure that digital assets are securely distributed, and that privacy is maintained at all times, proper access management needs to be in place. Keeping APIs, and the data provided through them, safe and only available to the intended user is a must. And with users who are used to moving through digital systems friction-free, an efficient identification and authorization process has never been more important. By embedding identity information in tokens, you can simplify the access control decisions that will be made in many different places throughout your architecture.

This booklet gathers a selection of articles that cover the most important aspects of securing APIs and microservices. It gives an introduction to related issues, such as how to utilize well-established standards, like OAuth 2, OpenID Connect and SCIM, and how to connect these to your applications, systems and user identities.

We hope you find this useful, and that it helps you secure your current and upcoming APIs.

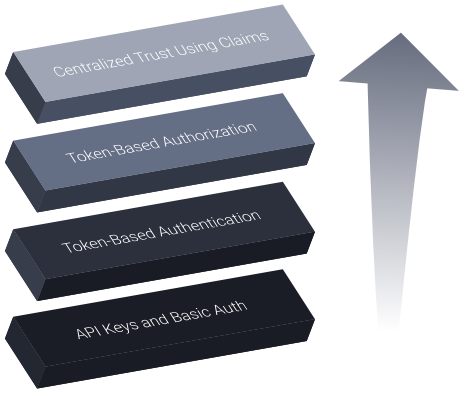
Happy reading from the Curity and Nordic APIs teams!

The API Security Maturity Model

API security has become a forefront issue for modern enterprises. However, there is a spectrum of API security implementations, and not all of them are effective. Too often, APIs only adopt HTTP Basic Authentication, API keys, or token-based authentication, overlooking a major concern: **identity**. To prevent vulnerabilities and reap efficiency benefits, a comprehensive identity focus is critical for fully evolved APIs.

This is why we've created the **API Security Maturity Model**. Inspired by the Richardson Maturity Model, which outlines increasing degrees of web service development maturity, the API Security Maturity Model reframes the model within the context of security. Within this model, security and trust are improved the higher up you go.

- Level 0: API Keys and Basic Authentication
- Level 1: Token-Based Authentication
- Level 2: Token-Based Authorization
- Level 3: Centralized Trust Using Claims



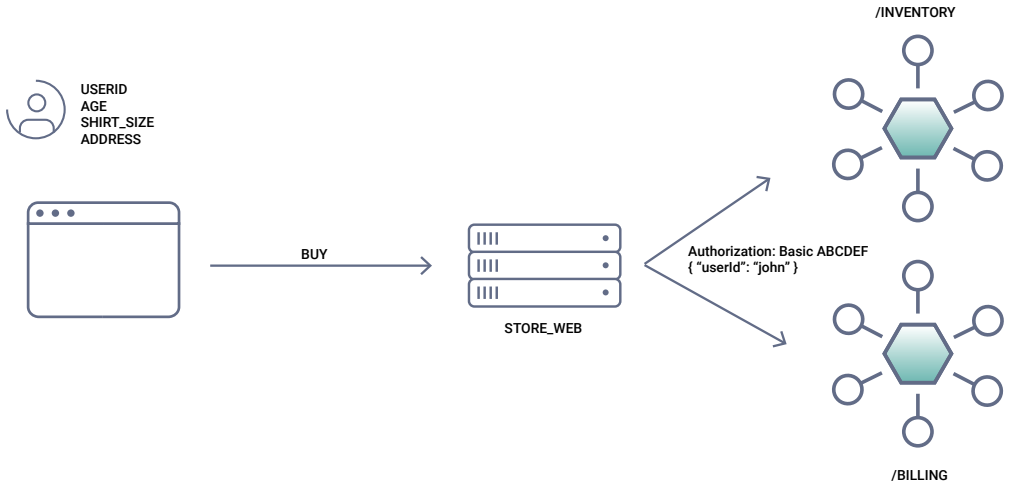
The more evolved API security is, the more identity emphasis it tends to have. So, how do we encapsulate identity with APIs and make it useful? APIs that utilize OAuth and OpenID Connect can take advantage of Claims, an advanced form of trust. Tokens such as JWTs utilizing Subject and Context Attributes can delegate platform-wide trust.

More on the specifics of that below. But first, let's expand on each maturity stage within the model to understand its benefits and drawbacks.

Level 0: API Keys and Basic Authentication

APIs at Level 0 use Basic Authentication or API keys to verify API calls. These are inserted within the header or body of the URL of the API request. This is the level of security that most APIs adopt. Most APIs established this authentication years ago, and unfortunately never evolved from there.

For example, consider an eCommerce store. It makes API calls to a payment API based on user purchases. It sends authentication in the form of an API Key or Basic Authentication in the header to the app and passes it to APIs. The user ID is placed in the Body or URL. In the example below, there are two APIs: `BILLING` and `INVENTORY`. Since HTTP Basic Authentication or API keys only authenticate the `STORE_WEB`, the store must pass on user data.



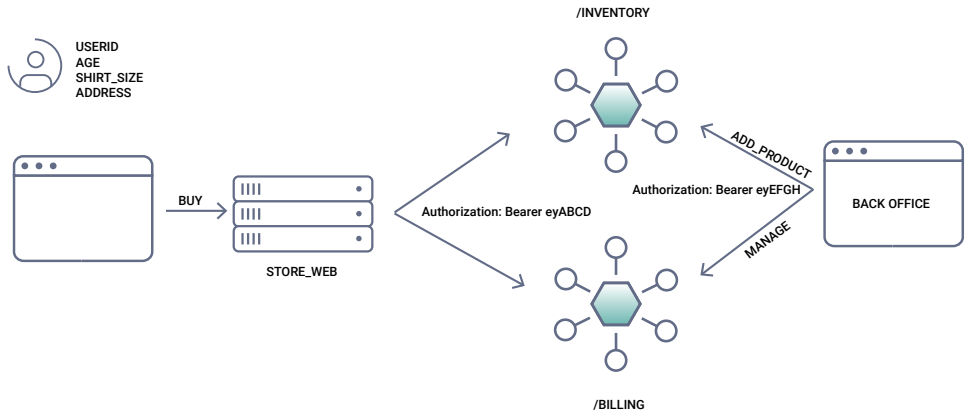
The Problems with Level 0:

You may be thinking: aren't API keys sufficient? Well, this method is actually very basic, wrought with vulnerabilities. Not only are keys constantly compromised, but API key verification relies on machine-machine verification, not bound to the identity of the user at all. Lastly, this method only provides authentication, the act of proving an assertion, and does not cover *authorization* at all.

Level 1: Token-based Authentication

APIs at Level 1 utilize Access Tokens for authentication within a token-based architecture. Such Access Tokens delineate the type of user (machine, app, user, etc.). As this enables privileged access, it helps in environments where the separation of internal and external users is required. Level 1 provides better auditing since user identity is part of the request.

For example, consider token-based authentication at the eCommerce store. When we introduce a back office, the same problem occurs. Custom logic is needed to know if the request is a back-office request with elevated privileges or if it comes from the store web.



The Problems with Level 1:

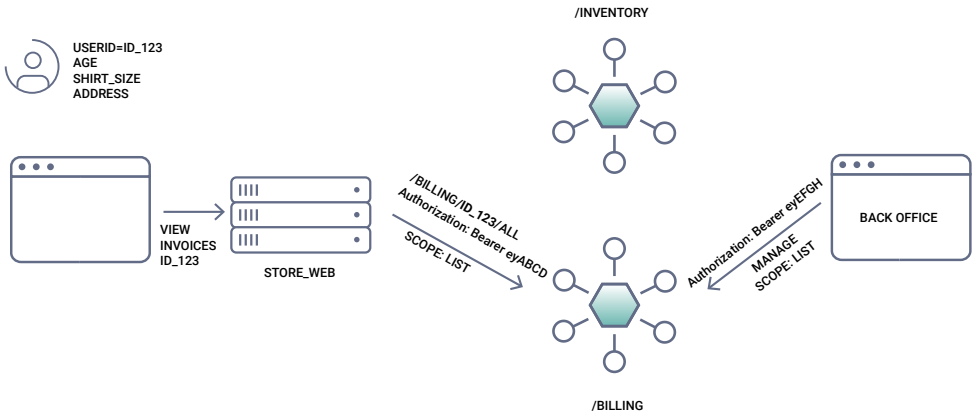
At Level 1, anyone with a token can modify the API, meaning privileged access can be hacked. Furthermore, Level 1 only covers authentication, not authorization. In other words, this strategy doesn't ask *what are you allowed to do?* When only using tokens for authentication, all authorization becomes custom code. Thus, custom mechanisms like if statements must be coded. This is negated in Levels 2 and 3, where you can utilize token data for authorization, thus generalizing authorization logic.

Level 2: Token-based *Authorization*

APIs at Level 2 are more evolved, using token-based architecture for authorization. Authorization delineates privileges for the requesting party, asking *what are you allowed to do?* APIs in Level 2 adopt **OAuth**, a widely adopted authorization standard in which client requests require an OAuth server for authorization. Maintained by IETF, OAuth 2.0 defines varying flows to obtain tokens, enabling the ability to grant access to resources without the need for a password.

One great benefit of OAuth is Scopes. Scopes can be utilized as "named permissions" within a token. These Scopes can specify user privileges. OpenID Connect defines [standard scopes] that can be used to generate standard identity arguments. Or, you can create [custom scopes] for your API. Scopes have more useful data and are better than building 'if' statements into a system.

Let's consider our eCommerce store again. Now, we introduce Scopes, so that the public web store and back-office can have different privileges. However, some operations overlap. The Scope `LIST` is used to list invoices in the billing API. The `ID` to list for is in the URL or passed as a request parameter. Thus, it's possible to manipulate the call to list invoices for another user. Thus, the Scope is not sufficient. Scopes also lock down what the client application is allowed to do; they don't help with the particular user since they are only "names" and not "values." Instead, Claims should be used so that the parameter is baked into the token. Then it's easy to separate back office privileges from `store_web` privileges.



The Problems with Level 2:

One problem in Level 2 is that the system faces the threat of being decompiled. When identity is built directly into the API, logic errors may be discovered and exploited. Level 2 also introduces a higher degree of system complexity, as some API request parameters may rely on other API responses or other conditions. What happens when one API calls another API that fails? Or, what if the data request is full of errors? You can't always assume the data passed from one API to the next is always correct. These realities cause cascading issues of trust, easily becoming an intertangled mess. We call this a "spaghetti of trust."

Level 3: Centralized Trust Using Claims

The final tier, Level 3, is the most evolved API security platform. This practice involves centralized trust with Claims and possibly signed JSON Web Tokens (JWTs). In doing so, we solve all the problems outlined above.

What are **JWT**s? Well, to clarify common misconceptions, a JWT is NOT a protocol. It's a signed piece of data. OAuth flows utilize JWTs to verify transactions. JWTs can be used to share Scopes.

And **Claims**? Claims are essentially assertions. For example, consider a written statement: "Jacob is an identity specialist, says Travis." This claim has a **Subject** (Jacob), an **Attribute** (that he is an identity specialist), and an **Asserting Party** (Travis). If you trust Travis, then you trust the Claim. Many **Attributes** can make up identity. There are **Subject** attributes, like name, age, height, weight, etc. For these attributes, the Asserting Party would be the police or tax authorities. There are also **Context Attributes**, such as the situation, timing, location, weather, etc.

Instead of trusting the attributes themselves, it's far better to trust claims made by common parties. Identity systems use Claims with similar anatomy for verification. If you trust the OAuth Server that issues keys, then you trust the claim being made. To verify a claim (simplified):

- Requesting Party calls the Issuer
- Issuer returns data, signed with a private key
- Requesting Party sends to another party
- Replying Party verifies the signature with a public key

This method solves the issue of trust, by trusting the *issuer* of tokens rather than the claims themselves.

JWTs Require OAuth & OpenID Connect

In cybersecurity, it's rarely encouraged to invent your own traffic rules. For centralized trust to function, authorization systems require the use of stable protocols. Just as street traffic follows common protocols, identity systems require their own shared open standards. These protocols are OAuth and OpenID Connect. Utilizing these standards, an app can share secure, asserted data within JWTs for verification.

Claims: Highly-evolved Identity-based API Security

Trust is a subjective thing. In designing a secure API-based system, should we trust keys, tokens, passwords, machines, or users themselves? The answer is more complex than most API designers think, and maybe pivotal to safeguarding your platform as a whole.

As the API Security Maturity Model displays, **highly mature APIs place trust in very few sources**. Especially, these evolved APIs place trust in the issuer of tokens. This does not guarantee the truth but is the closest representation to validating the identity of requesting parties. Furthermore, standardizing this process with centralized trust removes spaghetti code and wasted effort on custom code.

Essentially, the pinnacle of API security is to trust claims, not attributes. When building an identity-based API security system based on claims, remember some best practices:

- Organize sensitive data only to be reachable by OIDC server
- Include *identity* data in token, not context attributes
- Opaque tokens for the public, JWTs internally
- Limit data exposure only to when the client needs it
- Avoid app-specific rules

Without more advanced security, APIs could easily be made vulnerable with a rogue key left in a GitHub repository. Thus, API providers must make smarter security decisions that safeguard the integrity of the entire platform.

API Security: Deep Dive into OAuth and OpenID Connect

OAuth 2 and OpenID Connect are fundamental to securing your APIs. To protect the data that your services expose, you must use them. They are complicated though, so we wanted to go into some depth about these standards to help you deploy them correctly.

OAuth and OpenID Connect in Context

Always be aware that OAuth and OpenID Connect are part of a larger information security problem. You need to take additional measures to protect your servers and the mobiles that run your apps in addition to the steps taken to secure your API.

Without a holistic approach, your API may be incredibly secure, your OAuth server locked down, and your OpenID Connect Provider tucked away in a safe enclave. Your firewalls, network, cloud infrastructure, or the mobile platform may open you up to attack if you don't also strive to make them as secure as your API.

To account for all three of these security concerns, you have to know who someone is and what they are allowed to do. To authenticate and authorize someone on your servers, mobile devices, and in your API, you need a complete Identity Management System. At the head of API security, enterprise security and mobile security is identity!

Only after you know who someone (or something) is can you determine if they should be allowed to access your data. We won't go into the other two concerns, but don't forget these as we delve deeper into API security.

Start with a Secure Foundation

To address the need for Identity Management in your API, you have to build on a solid base. You need to establish your API security infrastructure on protocols and standards that have been peer-reviewed and are seeing market adoption. For a long time, lack of such standards has been the main impediment for large organizations wanting to adopt RESTful APIs in earnest. This is no longer the case since the advent of the Neo-security Stack:

Neo-security Stack

This protocol suite gives us all the capabilities we need to build a secure API platform. The base of this, OAuth and OpenID Connect, is what we want to go into in this article.

Overview of OAuth

OAuth is a sort of “protocol of protocols” or “meta protocol,” meaning that it provides a useful starting point for other protocols (e.g., OpenID Connect, NAPS, and UMA). This is similar to the way WS-Trust was used as the basis for WS-Federation, WS-Secure, etc., if you have that frame of reference.

Beginning with OAuth is important because it solves a number of important needs that most API providers have, including:

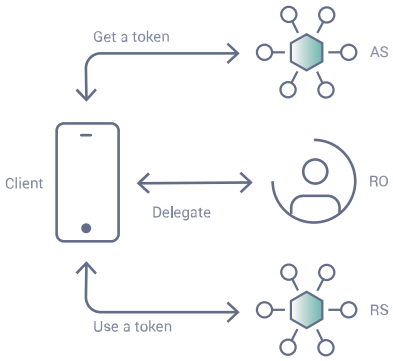
- Delegated access
- Reduction of password sharing between users and third parties (the so called “password anti-pattern”)
- Revocation of access

When the password anti-pattern is followed and users share their credentials with a third-party app, the only way to revoke access to that app is for the user to change their password. Consequently, all other delegated access is revoked as well. With OAuth, users can revoke access to specific applications without breaking other apps that should be allowed to continue to act on their behalf.

Actors in OAuth

There are four primary actors in OAuth:

1. **Resource Owner (RO)**: The entity that is in control of the data exposed by the API, typically an end user
2. **Client**: The mobile app, web site, etc. that wants toaccess data on behalf of the Resource Owner Service (STS) or, colloquially, the OAuth server that issues tokens
3. **Authorization Server (AS):** The Security Token Service (STS) or, colloquially, the OAuth server that issues tokens
4. **Resource Server (RS)**: The service that exposes the data, i.e., the API



Scopes

OAuth defines something called “Scopes.” These are like permissions or delegated rights that the Resource Owner wishes the client to be able to do on their behalf. The client may request certain rights, but the user may only grant some of them or allow others that aren’t even requested. The rights that the client is requesting are often shown in some sort of UI screen. Such a page may not be presented to the user, however. If the user has already granted the client such rights (e.g., in the EULA, employment contract, etc.), this page will be skipped.

What is in the scopes, how you use them, how they are displayed or not displayed, and pretty much everything else to do with scopes are not defined by the OAuth spec. OpenID Connect does define a few, but we’ll get to that in a bit.

Kinds of Tokens and Token Purpose

In OAuth, there are two kinds of tokens, or put in other words, tokens with different purposes:

1. **Access Tokens:** These are tokens that are presented to the API
2. **Refresh Tokens:** These are used by the client to get a new access token from the AS

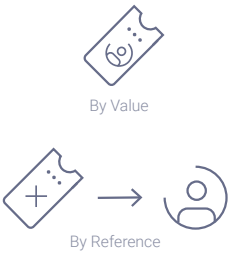
(Another kind of token that OpenID Connect defines is the ID token. We’ll get to that in a bit.)

Think of access tokens like a session that is created for you when you login into a website. As long as that session is valid, you can continue to interact with the website without having to login again. Once that session is expired, you can get a new one by logging in again with your password. Refresh tokens are like passwords in this comparison. Also, just like passwords, the client needs to keep refresh tokens safe. It should persist these in a secure credential store. Loss of these tokens will require the revocation of all consents that users have performed.

NOTE: The Authorization Server may or may not issue a refresh token to a particular client. Issuing such a token is ultimately a trust decision. If you have doubts about a client's ability to keep these privileged tokens safe, don't issue one!

Passing Tokens

As you start implementing OAuth, you'll find that you have more tokens than you ever knew what to do with! How you pass these around your system will certainly affect your overall security. There are two distinct ways in which they are passed:



1. By value
2. By reference

These are analogous to the way programming language pass data identified by variables. The run-time will either copy the data onto the stack as it invokes the function being called (by value) or it will push a pointer to the data (by reference). In a similar way, tokens will either contain all the identity data in them as they are passed around or they will be a reference to that data.

Profiles of Tokens: Token Types

There are different profiles of tokens as well, in the spec this is loosely referred to as the token type.

The two that you need to be aware of are these:

1. Bearer tokens
2. Holder of Key (HoK) tokens

You can think of bearer tokens like cash. If you find a dollar bill on the ground and present it at a shop, the merchant will happily accept it. She looks at the issuer of the bill and trusts that authority. The salesperson doesn't care that you found it somewhere. Bearer tokens are the same. The API gets the bearer token and accepts the contents of the token because it trusts the issuer (the OAuth server). The API does not know if the client presenting the token really

is the one who originally obtained it. This may or may not be a bad thing. Bearer tokens are helpful in some cases, but risky in others. Where some sort of proof that the client is the one to who the token was issued for, HoK tokens should be used.

HoK tokens are like a credit card. If you find a credit card on the street and try to use it at a shop, the merchant will (hopefully) ask for some form of ID or a PIN that unlocks the card. This extra credential assures the merchant that the one presenting the credit card is the one to whom it was issued. If your API requires this sort of proof, you will need HoK key tokens. This profile is still a draft, but you should follow this before doing your own thing.

NOTE: You may have heard of MAC tokens from an early OAuth 2 draft. This proposal was never finalized, and this profile of tokens are never used in practice. Avoid this unless you have a very good reason. Vet that rational on the OAuth mailing list before investing time going down this rabbit trail.

Token Format

We also have different formats of tokens. The OAuth specification doesn't stipulate any particular format of tokens. This was originally seen by many as a negative thing. In practice, however, it's turned out to be a very good thing. It gives immense flexibility. Granted, this comes with reduced interoperability, but a uniform token format isn't one area where interop has been an issue. Quite the contrary! In practice, you'll often find tokens of various formats and being able to switch them around enables interop. Example types include:

- WS-Security tokens, especially SAML tokens
- JWT tokens (which I'll get to next)
- Legacy tokens (e.g., those issued by a Web Access Management system)
- Custom tokens

Custom tokens are the most prevalent when passing them around by reference. In this case, they are randomly generated strings. When passing by val, you'll typically be using JWTs.

JSON Web Tokens

JSON Web Tokens or JWTs (pronounced like the English word "jot") are a type of token that is a JSON data structure that contains information, including:

- The issuer
- The subject or authenticated user (typically the Resource Owner)
- How the user authenticated and when
- Who the token is intended for (i.e., the audience)

These tokens are very flexible, allowing you to add your own claims (i.e., attributes or name/value pairs) that represent the subject. JWTs were designed to be lightweight and to be snuggly passed around in HTTP headers and query strings. To this end, the JSON is split into different parts (header, body, signature) and base-64 encoded.

If it helps, you can compare JWTs to SAML tokens. They are less expressive, however, and you cannot do everything that you can do with SAML tokens. Also, unlike SAML they do not use XML, XML name spaces, or XML Schema. This is a good thing as JSON imposes a much lower technical barrier on the processors of these types of tokens.

JWTs are part of the JSON Identity Suite, a critical layer in the Neo-security Stack. Other things in this suite include JWA for expressing algorithms, JWK for representing keys, JWE for encryption, JWS for signatures, etc. These together with JWT are used by both OAuth (typically) and OpenID Connect. How exactly is specified in the core OpenID Connect spec and various ancillary specs. In the case of OAuth, this is including the Bearer Token spec.

OAuth Flow

OAuth base specification defines different "flows" or message exchange patterns. These interaction types include:

- The code flow (or web server flow)
- Client credential flow
- Resource owner credential flow
- Implicit flow

The code flow is by far the most common; it's probably what you are most familiar with if you've looked into OAuth much. It's where the client is (typically) a web server, and that website wants to access an API on behalf of a user. You've probably used it as a Resource Owner many times, for example, when you log in to a site using certain social network identities. Even when the social network isn't using OAuth 2 per se, the user experience is the same.

Improper and Proper Uses of OAuth

After all this, your head may be spinning. Mine was when I first learned these things. It's normal. To help you orient yourself, I want to stress one really important high-level point:

- **OAuth is not used for authorization.** You might think it's from its name, but it's not.
- **OAuth is also not for authentication.** If you use it for this, expect a breach if your data is of any value.
- **OAuth is also not for federation.**

So what is it for? **It's for delegation, and delegation only!**

This is your plumb line. As you architect your OAuth deployment, ask yourself: In this scenario, am I using OAuth for anything other than delegation? If so, go back to the drawing board.

Consent vs. Authorization

How can it not be for authorization, you may be wondering. The "authorization" of the client by the Resource Owner is really consent. This consent may be enough for the user, but not enough for the API. The API is the one that's actually authorizing the request. It probably takes into account the rights granted to the client by the Resource Owner, but that consent, in and of itself, is not authorization.

To see how this nuance makes a very big difference, imagine you're a business owner. Suppose you hire an assistant to help you manage the finances. You consent to this assistant withdrawing money from the business' bank account. Imagine further that the assistant goes down to the bank to use these newly delegated rights to extract some of the company's capital. The banker would refuse the transaction because the assistant is not authorized — certain paperwork hasn't been filed, for example. So, your act of delegating your rights to the assistant doesn't mean squat. It's up to the banker to decide if the assistant gets to pull money out or not. In case it's not clear, in this analogy, the business owner is the Resource Owner, the assistant is the client, and the banker is the API.

Building OpenID Connect atop OAuth

As I mentioned above, OpenID Connect builds on OAuth. Using everything we just talked about, OpenID Connect constrains the protocol, turning many of the specification's SHOULDs to MUSTs. This profile also adds new endpoints, flows, kinds of tokens, scopes, and more. OpenID Connect (which is often abbreviated OIDC) was made with mobile in mind. For the new kind of tokens that it defines, the spec says that they must be JWTs, which were also designed for low-bandwidth scenarios. By building on OAuth, you will gain both delegated access and federation capabilities with (typically) one product. This means fewer moving parts and reduced complexity.

OpenID Connect is a modern federation specification. It's a passive profile, meaning it's bound to a passive user agent that does not take an active part in the message exchange (though the client does). This exchange flows over HTTP and is analogous to the SAML artifact flow (if that helps). OpenID Connect is a replacement for SAML and WS-Federation. While it's still relatively new, you should prefer it over those unless you have good reason not to (e.g., regulatory constraints).

As I've mentioned a few times, OpenID Connect defines a new kind of token: ID tokens. These are intended for the client. Unlike access tokens and refresh tokens that are opaque to the client, ID tokens allow the client to know, among other things:

- How the user authenticated (i.e., what type of credential was used)
- When the user authenticated
- Various properties about the authenticated user (e.g., first name, last name, shoe size, etc.)

This is useful when your client needs a bit of info to customize the user experience. Many times I've seen people use by value access tokens that contain this info, and they let the client take the values out of the API's token. This means they're stuck if the API needs to change the contents of the access token or switch to using by ref for security reasons. If your client needs data about the user, give it an ID token and avoid the trouble down the road.

The User Info Endpoint and OpenID Connect Scopes

Another important innovation of OpenID Connect is what's called the "User Info Endpoint." It's kind of a mouthful, but it's an extremely useful addition. The spec defines a few specific scopes that the client can pass to the OpenID Connect Provider or OP (which is another name for an AS that supports OIDC):

- openid (required)
- profile
- email
- address
- phone

You can also (and usually will) define others. The first is required and switches the OAuth server into OpenID Connect mode. The others are used to inform the user about what type of data the OP will release to the client. If the user authorizes the client to access these scopes, the OpenID Connect provider will release the respective data (e.g., email) to the client when the client calls the user info endpoint. This endpoint is protected by the access token that the client obtains using the code flow discussed above.

NOTE: An OAuth client that supports OpenID Connect is also called a Relying Party (RP). It gets this name from the fact that it relies on the OpenID Connect Provider to assert the user's identity.

Not Backward Compatible with v. 2

It's important to be aware that OpenID Connect *is not* backward compatible with OpenID 2 (or 1 for that matter). OpenID Connect is effectively version 3 of the OpenID specification. As a major update, it's not interoperable with previous versions. Updating from v. 2 to Connect will require a bit of work. If you've properly architected your API infrastructure to separate the concerns of federation with token issuance and authentication, this change will probably not disrupt much. If that's not the case however, you may need to update each and every app that used OpenID 2.

Conclusion

In this post, I dove into the fundamentals of OAuth and OpenID Connect and pointed out their place in the Neo-security Stack. I said it would be in depth, but honestly, I've only skimmed the surface. Anyone providing an API that is protected by OAuth 2 (which should be all of them that need secure data access), this basic knowledge is a prerequisite for pretty much everyone on your dev team. Others, including product management, ops, and even project management should know some of the basics described above.

This article was originally published on nordicapis.com

Using OAuth Within Microservices

Everyone's excited about microservices, but actual implementation is sparse. Perhaps the reason is that people are unclear on how these services talk to one another; especially tricky is properly maintaining identity and access management throughout a sea of independent services.

Unlike a traditional monolithic structure that may have a single security portal, microservices pose many problems. Should each service have its own independent security firewall? How should identity be distributed between microservices and throughout my entire system? What is the most efficient method for the exchange of user data?

There are smart techniques that leverage common technologies to not only authorize but perform delegation across your entire system. In this article we'll identify how to implement OAuth and OpenID Connect flows using JSON Web Tokens to achieve the end goal of creating a distributed authentication mechanism for microservices — a process of managing identity where everything is self-contained, standardized, secure, and best of all — easy to replicate.

What Are Microservices, Again?

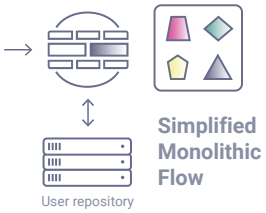
For those readers not well-versed in the web discussion trends of late, the microservice design approach is a way to architect web service suites into independent specialized components. These components are made to satisfy a very targeted function, and are fully independent, deployed as separate environments. The ability to recompile individual units means that development and scaling can be vastly easier within a system using microservices.

This architecture is opposed to the traditional monolithic approach that consolidates all web components into a single system. The downside of a monolithic design is that version control cycles are arduous, and scalability is slow. The entire system must be continuously deployed since it's packaged together.

The move toward microservices could have dramatic repercussions across the industry, allowing SaaS organizations to deploy many small services no longer dependent on large system overhauls, easing development, and on the user-facing side allowing easy pick-and-choose portals for users to personalize services to their individual needs.

Great, so What's the Problem?

The problem we're faced with is that microservices don't lend themselves to the traditional mode of identity control. In a monolithic system security works simply as follows:



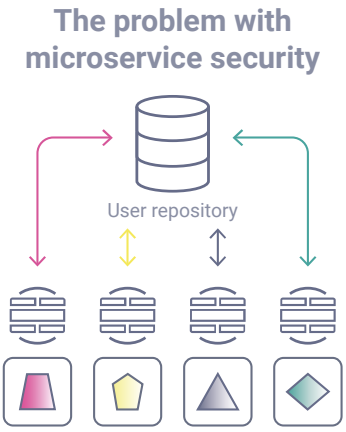
1. Figure out **who** the caller is
2. Pass on credentials to other components when called
3. Store user information in a data repository

Since components are conjoined within this structure, they may share a single security firewall. They share the state of the user as they receive it and may also share access to the same user data repository.

If the same technique were to be applied to individual microservices, it would be grossly inefficient. Having an independent security barrier — or request handler — for each service to authenticate identity is unnecessary. This would involve calling an Authentication Service to populate the object to handle the request and respond in every single instance.

The Solution: OAuth as a Delegation Protocol

There is a method that allows one to combine the benefits of isolated deployment with the ease of a federated identity. To accomplish this OAuth should be interpreted not as Authentication, and not as Authorization, but as Delegation.



In the real world, delegation is where you delegate someone to do something for you. In the web realm, the underlying message is there, yet it also means having the ability to offer, accept, or deny the exchange of data. Considering OAuth as a Delegation protocol can assist in the creation of scalable microservices or APIs.

To understand this process, we'll first lay out a standard OAuth flow for a simple use case. Assume we need to access a user's email account for a simple app that organizes a user's email — perhaps to send SMS messages as notifications. OAuth has the following four main actors as described in the previous article:

- **Resource Owner (RO):** the user
- **Client:** the web or mobile app
- **Authorization Server (AS):** OAuth 2.0 server
- **Resource Server (RS):** where the actual service is stored

A Simplified Example of an OAuth 2 Flow

In our situation, the app (the Client), needs to access the email account (the Resource Server) to collect emails before it can organize them to create the notification system. In a simplified OAuth flow, an approval process would be as follows:

1. The **Client** requests access to the **Resource Server** by calling the **Authorization Server**.
2. The **Authorization Server** redirects to allow the user to authenticate, which is usually performed within a browser. This is essentially signing into an authorization server, not the app.
3. The **Authorization Server** then validates the user credentials and provides an **Access Token** to the client, which can be used to call the **Resource Server**.
4. The **Client** then sends the **Token** to the **Resource Server**.
5. The **Resource Server** asks the **Authorization Server** if the token is valid.
6. The **Authorization Server** validates the **Token**, returning relevant information to the **Resource Server** i.e. time until token expiration, who the token belongs to.
7. The **Resource Server** then provides data to the **Client**. In our case, the requested emails are unbarred and delivered to the **Client**.

An important factor to note within this flow is that the Client — our email notification app — knows nothing about the user at this stage. The token that was sent to the client was completely opaque — only a string of random characters. Though this is a secure exchange, the token data is itself useless to the client. The exchange thus supplies access for the client, but not user information. What if our app needed to customize the User Experience (UX) based on which membership level the user belonged to, a group they were a member of, where they were located, their preferred language, etc.? Many apps provide this type of experience and for that they require additional user information.

The OpenID Connect Flow

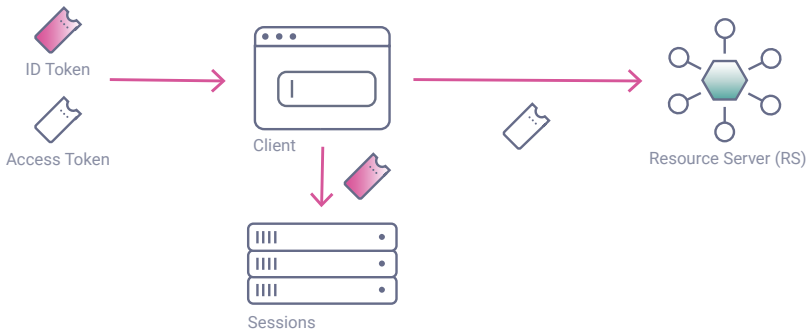
Let's assume that we're enhancing the email service client so that it not only organizes your emails, but also stores them and translates them into another language. In this case, the client will want to retrieve additional user data and store it in its own user sessions.

To give the client something other than the opaque token provided in the OAuth flow, use an alternative flow defined in OpenID Connect. In this process, the Authorization Server, which is also called an OpenID Connect Provider (OP), returns an ID Token along with the Access Token to the client. The flow is as follows:

1. The **Client** requests access to the **Resource Server** by calling the Open ID Connect enabled **Authorization Server**.
2. The **Authorization Server** redirects to allow the user to authenticate.
3. The **Authorization Server** then validates the user credentials and provides an **Access Token** AND an **ID Token** to the client.

4. The **Client** uses this **ID Token** to enhance the UX and typically stores the user data in its own session.
5. The Client then sends the **Access Token** to the **Resource Server**
6. The **Resource Server** responds, delivering the data (the emails) to the **Client**.

Sessions Can Be Created (SSO)



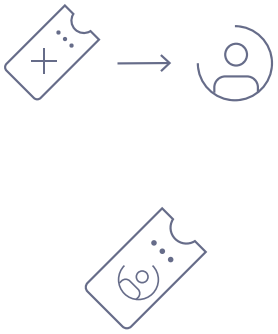
The ID token contains information about the user, such as how they authenticated, the name, email, and any number of custom data points on a user. This ID token takes the form of a JSON Web Token (JWT), which is a coded and signed compilation of JSON documents. The document includes a header, body, and a signature appended to the message. Data + Signature = JWT.

Using a JWT, you can access the public part of a certificate, validate the signature, and understand that this authentication session was issued — verifying that the user has been authenticated. An important facet of this approach is that ID tokens establish trust between the Authorization Server/Open ID Connect Provider and the Client.

Using JWT for OAuth Access Tokens

Even if we don't use OpenID Connect, JWTs can be used for many things. A system can standardize by using JWTs to pass user data among individual services. Let's revisit the formats for OAuth access tokens to see how to smartly implement secure identity control within microservice architecture.

By Reference: Standard Access Token



This type of token contains no information outside of the network, simply pointing to a space where information is located. This opaque string means nothing to a user, and as it's randomized cannot easily be decrypted. This is the standard form of an access token — without extraneous content, simply used for a client to gain access to data.

By Value: JSON Web Token

This type may contain necessary user information that the client requires. The data is compiled and inserted into the message as an access token. This is an efficient method because it erases the need to call again for additional information. If exposed over the web, a downside is that this public user information can be easily read, exposing the data to an unnecessary risk of decryption attempts to crack codes.

The Workaround: External vs. Internal

To limit this risk of exposure, Ideskog recommends splitting the way the tokens are used. What is usually done is as follows:

1. The **Reference Token** is issued by the **Authorization Server**. The client sends back when it's time to call the API.
2. In the middle: The **Authorization Server** validates the token and responds with a **JWT**.
3. The **JWT** is then passed further along in the network.

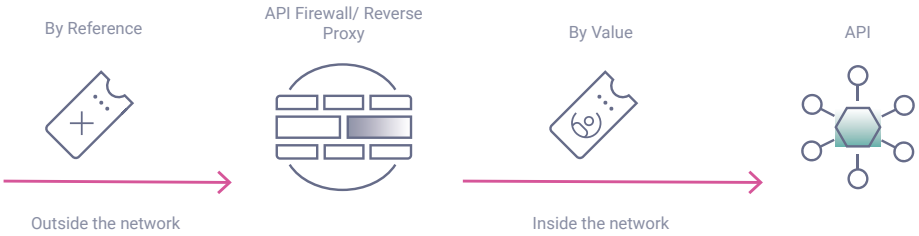
In the middle we essentially create a firewall, an Authorization Server that acts as a token translation point for the API. The Authorization Server will translate the token, either for a simple Reverse Proxy, or a full-scale API Firewall. The Authorization Server shouldn't be in the "traffic path" however — the reverse proxy finds the token and calls the Authorization server to translate it.

Let All Microservices Consume JWT

So, to refresh, with microservice security we have two problems:

- **We need to identify the user multiple times:** We've shown how to leave authentication to OAuth and the OpenID Connect server, so that microservices successfully provide access given someone has the right to use the data.
- **We have to create and store user sessions:** JWTs contain the necessary information to help in storing user sessions. If each service can understand a JSON web token, then you have distributed your identity mechanism, allowing you to transport identity throughout your system.

Token Translation



In microservice architecture, an access token should not be treated as a request object, but rather as an identity object. As the process outlined above requires translation, JWTs should be translated by a front-facing stateless proxy, used to take a reference token and convert it into a value token to then be distributed throughout the network.

Why Do This?

By using OAuth with OpenID Connect, and by creating a standards-based architecture that universally accepts JWTs, the end result is a distributed identity mechanism that is self-contained and easy to replicate. Constructing a library that understands JWT is a very simple task. In this environment, access as well as user data is secured. Creating microservices that communicate well and securely access user information can greatly increase agility of the whole system, as well as increase the quality of the end user experience.

Coarse Grained Authorization Using Scopes

How does one go about securing APIs, microservices, and websites? One way to do this is by focusing on the **authorization** – knowing what the caller is allowed to do with your data. Too often, though, providers rely too heavily on user identity, pairing it way too closely with the design of their APIs.

As OAuth doesn't authenticate by itself, the way these flows are structured means that API access often ultimately relies on user social logins, which is an unfavorable dependency that actually decreases API security and scalability. APIs are far better secured with a proxy in-between the API and authentication mechanism, utilizing scopes that delineate the type of access that the API grants.

Authorization needs to be done on different levels. We now focus on the coarse-grained level, where we can decide if we should at all consider the request. The next article will discuss the fine-grained setup where we can make decisions on a per user/request level.

In this article we'll see why APIs and microservices should decouple user identity from their designs, and how to go about this implementation. We'll review some sample flows, and briefly walk through how OAuth scopes can be used to create a more valuable, knowledgeable API. Following these cues, the end result will emboss state of the art Identity and Access Management (IAM) practices within actual API design, in effect utilizing identity data to secure the entire API lifecycle.

AAA

When an API call is made, we must know who made the request, and if they are allowed to read and access the requested data. Identity and Access Management is also described as **AAA;** an important initialism made up of:

- **Authentication:** Validation that the user is who they say they are.
- **Authorization:** Validation of the user, their application, and privileges.
- **Auditing:** Accounting for user behavior, logging metadata like what is accessed, when it's accessed, with what device, and more.

Caching user logs is great, but it doesn't prevent malformed requests to the server from the onset. We don't want to waste resources, so verifying requests must be processed as early as possible in the code pipeline. So, typically you block this with a proxy. BUT how do you make sure that the proxy knows what to do? How do we instruct the proxy to decipher what user or application is accessing the data, and what data they are allowed to access?

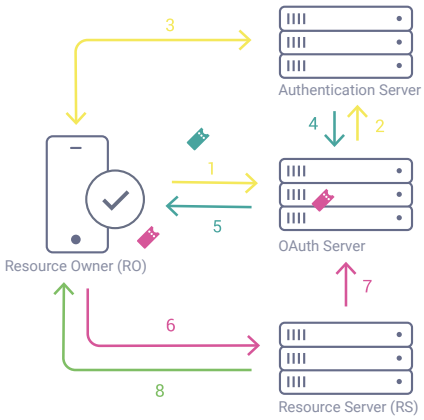
Overview of the OAuth Flow

You may be thinking, just use OAuth, problem solved. Yes, OAuth is a necessary protocol within the security workflow — but OAuth cannot authenticate — a separate server must tell OAuth who the user is. To understand where we're heading, here's a quick overview of a simple OAuth flow. To review, the actors are:

- **Resource Owner** (RO): The user
- **Client**: The application, mobile device, server, or website requesting data from the API
- **Authentication Server**: The login service that authenticates users
- **OAuth Server** (AS): Also called the *Authorization Server*
- **Resource Server** (RS): The API providing data

Let's assume that we have created a mail server with an API that provides information so that a third party app client can sort emails in an improved way. In our walkthrough we'll assume that the app uses Google as an authentication service. There are variants of these flows, but a simple OAuth flow for this scenario would be:

1. The **Client** first requests accesses the OAuth Server.
2. The **OAuth Server** next delegates authentication responsibility to a third-party **Authentication Server**.
3. The **User** enters credentials with the **Authentication Server** to authenticate.
4. The **Authentication Server** tells the **OAuth Server** the authentication was successful.
5. The **OAuth Server** sends the **Token** to the **Client**.
6. The **Client** uses the **Token** to access resources from the **Resource Server**.
7. The **Resource Server** verifies with the **OAuth Server** that the **Token** is valid.
8. The **Resource Server** (API) then sends the data to the **Client** app.



Designing an API with Scopes from the Bottom Up

Located within the OAuth token, **scope** is an interesting data point that you've likely used before. Scope specifies the extent of tokens and are akin to the **permissions** listed on a consent UI. They are extremely useful, as scopes can be used to delineate API access tiers. Furthermore, OAuth doesn't specify that you have to give the same scopes that you are requesting — if the scope changes you must simply notify the client/user. For access management designers, this grants us a lot of power and flexibility in how we handle scopes and identity. We'll see that building an API with scopes hardwired into the design can be extremely helpful.

Scopes are tied to the **client** which is where this becomes useful from a coarse-grained perspective. Consider an Invoicing API, the API can create and list invoices. Customers using the company app should only be able to view the invoices, while the internal Finance systems should be able to create invoices. It doesn't matter if a super admin, or a regular customer logs in with the customer app, they will only ever be able to list invoices. In the next article we'll discuss *which* invoices they will be able to list.

Let's build a sample API to see what we're talking about. We could in example design an **Invoice API** that taps into an e-commerce platform. The API provides access to customers, who are listing the invoices, as well as to employees, who are writing the invoices.

On the other end, **ECommerceApp** is an application that consumes the Invoices API. You can think of ECommerceApp as the client in our OAuth flow. As it's a customer client, it will be limited in scope without editing capability.

So how do we create permissions? To do so means we define the **scopes** in the API. The beautiful magic here is that we assign these scopes with different strengths as follows:

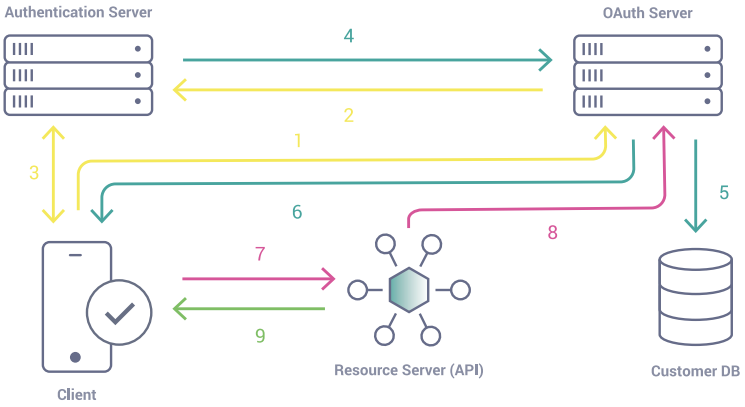
Let OAuth Filter These Scopes

Next we let the OAuth server filter API access based on these scopes. Since we can change our scopes

SCOPE	STRENGTH	BEHAVIOUR
`No Scope`	Weak	User does not have to be a registered customer
`invoice_read`	Medium	User must be a registered customer
`invoice_write`	Strong	The application must be internal and the user needs to login using internal credentials on the corporate network.

throughout the process, when ECommerceApp sends a request with an `invoice_read` scope, a new flow would look like this:

1. The ECommerceApp **Client** makes a request to the **OAuth Server** sending a basic read scope.
2. The **OAuth Server** next delegates authentication responsibility to a third party **Authentication Server**.
3. The **User** enters credentials with Google, the **Authentication Server**, to authenticate.
4. The **Authentication Server** tells the **OAuth Server** the authentication was successful, and sends an **OAuth Token**. Within the token is information that will affect the scopes, namely the ACR (which in this case is *social*) and the *subject* (the username).
5. The **OAuth Server** checks its **Customer Database** to see if the username is in fact a customer.
6. In this case it does find the username among the customer files, and thus grants the client an **Access Token** with the `invoice_read` scope.
7. The ECommerceApp **Client** then sends the **Access Token** to the Invoice API **Resource Server**.



8. The **Resource Server** verifies with the **OAuth Server** that the **Token** is valid.
9. The **Resource Server** (API) then sends the data to the Client app.

Using this flow, the API becomes much more knowledgeable. By returning different scopes in step 6, we give the client the ability to tailor the UI and enable or disable certain functions. Even better is that since the power of the scope told it that sufficient strength was used during authentication, the API now doesn't even care which authentication was originally made or if the user was a customer or not. Empowered with data on the client, the user, and the access granted by the scopes, it can effectively filter data to the proper channels using an improved provisioning schematic.

The Proxy Accepts Only Valid Requests

The last step in decoupling authentication from API design is constructing a proxy to separate our API from the authentication mechanism. The final result is a proxy that only allows access to the Invoice API when the following criteria are met:

- The token exists
- The token is valid
- The token contains one or more scopes. For the Invoice API, that would be one or more of the following:

```
invoice_read
invoice_write
```

Now we have a more secure, strict API front end that only allows access once these three rules are met, blocking unregistered users in the proxy.

Conclusion: Separate the API from Authentication and Client Permissions

In order to build a scalable API infrastructure that is ideal for microservices, you must design your APIs in a way that separates them from authentication. Using scopes to map the permissions, and defining them in your API, can create a robust platform that better protects and informs you as an API provider.

Benefits of this approach also include:

- Overall API security is improved with abstraction;
- API identity control now maps your access tiers, enabling easy enforcement for freemium business models;
- This pattern is simple to grasp and implement;
- Constructing scopes into API design rather than third party authentication means freedom of any authentication method without bothering the APIs with all the details;
- Can help in separating private, public, partner APIs — complementing platform strategy and adding business potential;
- Could be used to inform usage analytics;
- As marketing departments have high demand on smooth customer journeys, this provides a quicker time to market when it comes to authentication.

But perhaps the most critical point is that **one and only one pattern is needed for microservices design**. This increases the ability to not only build APIs, but easily share identity knowledge across an organization, increasing the service maintainability over time. Authentication is a moving target, whereas APIs may not be.

Claims Based Authorization Using OAuth

In the previous article we discussed how to use OAuth Scopes to perform coarse-grained authorization. This helps with the separation of access for different applications. The next step is to continue and map the exact access that the user needs. Given the example with an Invoice API, it's obvious that the coarse-grained *scope* tells us weather or not the application is allowed to perform the listing of invoices, but what the API really needs to know is which invoices the application may list.

This is data tied to the user which can be communicated using claims in the OAuth Access Token.

Attributes vs. Claims

To understand what claims are, we need to start with attributes. Attributes are properties of a user, such as username, name, age, shoesize etc. They can also be attributes about the session, such as when the user logged in, from what location etc.

For that reason, we typically split attributes in two categories: Subject Attributes and Context Attributes.

Subject Attributes are true about the user no matter how the user logged in. Depending on authentication method and orchestration during login, you may receive different Subject Attributes. I.e. if a user logs in using Google, Google will provide certain attributes, and when they login using Active Directory other attributes may be present. It's up to the authentication service to normalize these attributes and make sure the relevant Subject Attributes are always present.

Context Attributes on the other hand tell us something about the circumstances under which the Subject Attributes were established. The time of authentication, the location, what authentication method was used etc. These are relevant when issuing OAuth tokens, because certain properties of a token may be considered more sensitive and should perhaps only be present if we're fairly confident the user is close to the computer. So, if the authentication time is further back in time than say 30 minutes, we can drop properties in the token to weaken its strength.

Now that we understand attributes, we need to look at claims. Attributes are only interesting if we trust the party that issued them or put in other word: if we trust the party that claim them to be true.

A claim has the following form: **Jacob has a Horse, says Travis:**

> *Jacob – is the subject*
> *"has a horse" – is the claim / attribute*
> *Travis – is the asserting party.*
> *If we trust Travis, then we can trust the claim about the horse.*
> *More formally: Subject + Attribute + Issuer = Claim*

Claims

Claims are a first-class citizen in OpenID Connect and are easily transferrable to OAuth. The most common place where you encounter these are in the ID Token which is a Json Web Token (JWT) but they can be generalized to tokens of any purpose and format:

```
{
   sub: janedoe@example.com
   name: Jane Doe
   iat: 1546300800
   exp: 1893456000
   iss: https://login.curity.io
   subscriber_id: ABC_123
   phone_number: +46 123 123 123
}
```

This JWT contains the following claims: “name”, “iat”, “exp”, “subscriber_id”, “phone_number”.

It also has a subject “sub” and an issuer “iss”. The JWT is normally signed by a key that matches what the issuer has documented or published out of band. A receiver of this token can verify that the “iss” field is a host it trusts and use a key provided by that host to verify that the token has indeed been issued by that party. Once that is done, we can rely on the claims in the token and act on the information.

OpenID Connect describes standard claims for ID Tokens, but the claims infrastructure is not limited to only those tokens. Access Tokens can use the same structure, even when using other formats than Json Web Tokens.

This makes claims very powerful when designing systems in need of finer grained authorization.

Scopes Revisited

In the previous article we discussed the scope’s role in authorizing access for the client. But scopes play a deeper role in a claims-based system. Without claims, a scope is just a space separated list of strings with Scope Tokens or Scope Names.

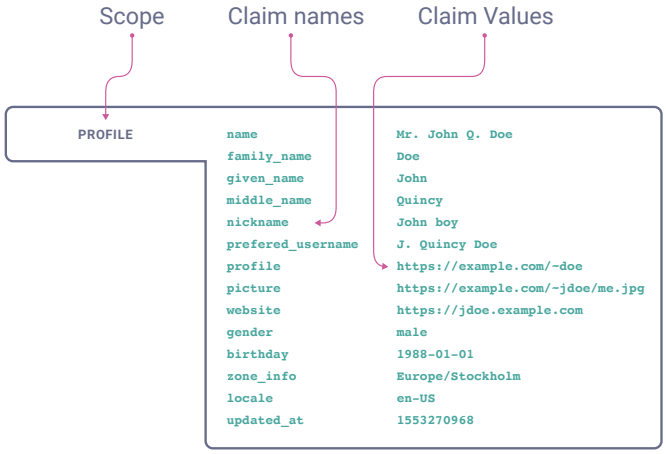
Example: scope = “invoice_read invoice_write openid email”

It’s good to think of the scope parameter as *Scope of Access*. I.e. this lists the things the client needs to access.

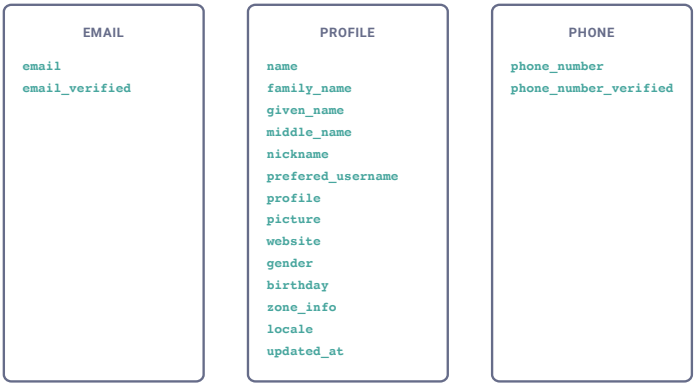
We give each of these meaning in coarse grained authorization, to see if the client should be allowed to query the API. But as discussed in the scope article, even if a scope of invoice_read is present in the token, we don’t know *which* invoice it should be allowed to read.

Group of claims

OpenID Connect defines the scope as a group of claims. So, the “email” scope token is mapped to the “email” and “email_verfied” claim. The claims have values associated to then while the scope token is just a name. This means that a scope is simply a *bag* of claims. Requesting a scope will result in zero or more claims to be issued and present in the tokens.



Each OpenID claim is associated with a scope as shown in the image below.



This can be generalized to arbitrary scopes and claims. When it comes to API access this becomes very useful.

Consider the example of an invoice API. Let's assume we have the following scopes:

```
invoice_list

invoice_read

invoice_write
```

As already discussed, it's not enough to authorize the particular request based on only this information. The API needs to know which invoices that it's allowed to list. Listing the actual invoice IDs in the token is not very efficient, but let's assume we have a claim which is `account_id`. It's the financial account that is associated with the user.

We can now map this to the scopes:

SCOPE	CLAIM ASSOCIATED WITH SCOPE
`invoice_list`	`account_id, role=customer`
`invoice_read`	`account_id, role=customer`
`invoice_write`	`account_id="*", role=finance`

If any of these scopes are requested, the resulting token will contain the claims needed with an associated value for that user. Now the API can easily know if the requested operation should be authorized or not.

What we have done now is create a contract. The API knows that if the role=customer it needs to check the account_id claim to see for which account the invoices should be provided and if the role=finance, it allows writes.

This mitigates many risks. The API no longer needs to rely on data provided from unreliable sources, but can safely operate on the account, and scope information when performing the task. It makes it virtually impossible for a third party to inject a different `account_id` in the request and the API doesn't have to look up additional data about the user.

The example should be considered illustrative but shows the essence of how to use claims.

Describing Login Information

It's not only the API that will benefit from claims being present in the token. When using OpenID Connect, the client (application) might also be interested in knowing details about the user. In many cases it's interested in knowing details about the context in which the user authenticated.

As mentioned previously, the Context Attributes provide this information. Using the OpenID Connect ID Token, the client can determine not only who logged in, but also when and how. These are part of the standard claims that are associated with the *openid* scope implicitly.

- `sub`: who logged in
- `auth_time`: when the login occurred
- `acr`: Short for Authentication Context Class Reference, this stipulates how the login happened.

It's not unusual for applications to require knowledge if the authentication was fresh or if it occurred with SSO, i.e. the user didn't interact. This can be critical when deciding if sensitive data should be displayed or not.

Knowing how the user logged in can also be important, since it can help in decisions around what actions a user may be allowed to take. A user that logged in with a strong authentication could be allowed to change its account details, while a user that didn't may perhaps only view the same data.

Conclusion

Claims are the contents of a token. They are asserted by the issuer, which in this case is the OAuth or OpenID Connect server. They provide a powerful mechanism to help both the API and the Client make qualified authorization decisions.

Claims are based out of the OpenID Connect standard and provide a mechanism to help the API being more fine-grained in the authorization of the requests.

Securing APIs in a Cloud Native Environment

Computer systems built today have very little in common with what we built only a few years ago. Systems have evolved from classic client-server solutions, into distributed systems that span over many data centers and geolocations. DevOps teams are now able to build applications that scale up and down effortlessly, and even build serverless applications that can spin up a server just to serve a single request. It's pretty impressive.

However, one requirement has stayed the same over the years. The system still needs to know who the caller is, or at least know a little something about the caller. Only when the caller is known can the data be released. Identity is a prerequisite to authorization.

So how do we fulfill this requirement in ever changing environments? One way is to adhere to the standards that are available. Following standards ensures interoperability, and well-written standards help with scalability. But which ones to choose? There are hundreds of standards that could apply to the subject. For identity alone there are ~50 different ones that could be relevant depending on the use case.

The Power of Identity Standards OAuth and OpenID Connect

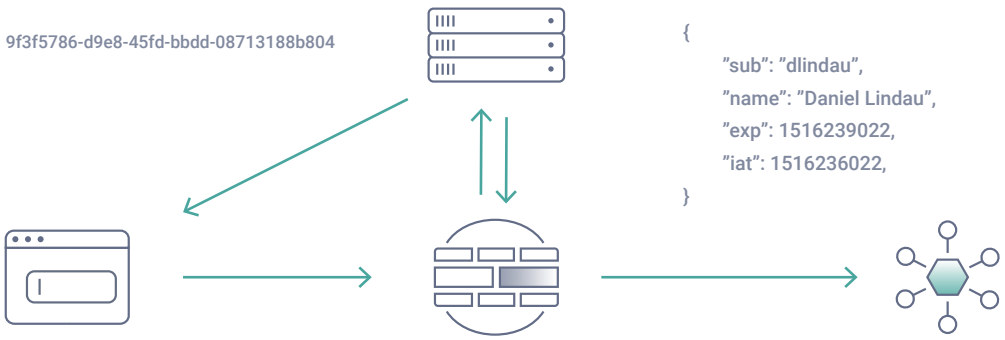
There have been several standard attempts trying to solve these kinds of things, and there will likely be many more in the future. Protocols such as SAML, WS-* and the likes of them have been around for many years and are still quite heavily deployed. They do solve a lot of the use cases, but with the current need for REST-ful access control and identity management, they are quite hard to use. Simply because they're not designed for that. Instead two others have taken over the stage:

OAuth2 and OpenID Connect. This should not come as a surprise to anyone, they have been around a while now and have become de facto standards for digital identity and delegated access. Together, these two make up the core of a secure API platform.

A lot of people have glanced at the core OAuth 2.0 spec, and thought to themselves; "I can implement this". And that's probably true, but there is a lot more to it than the core specifications. OAuth and OpenID Connect is a whole family of specifications, and if we printed them all we would have close to a bookshelf full of specifications to read. Because of this, I never recommend implementing the server part by yourself. All those nuances should be left to be implemented by experts.

So, if I install an OAuth/OpenID server, am I done? No, but you're well on your way. There are still some measures to be made, and I'll give you a few tips on how to avoid some of the pitfalls when deploying largely scaled platforms.

Phantom Token - the Base of a Secure API Platform



The Phantom Token flow is something that we architectured to fulfill the need of hiding data from clients, and at the same time share all the data the APIs need for making their authorization decision. Although it's not a standard in itself, it ties together several standards in a nice and comprehensible way. It's a pattern that we have deployed at all of our customers, with very good results.

The idea is that you allow your OAuth server to issue an opaque access token, a token that is merely a reference to the token data. The opaque token is represented by a random string, so there is no way for a potential attacker to extract any data from it. This also means that the client receiving the token isn't able to read any data from it. This is because the token is not really for the client, it's for the API. When the client uses the token to call an API, the API will have to de-reference the data by using the introspection capability of the OAuth server. This will not scale very well, since all APIs would have to do the same thing for every incoming request and would more or less force the APIs to create their own cache. So instead, we introduce the API gateway.

With the API gateway in place, we can allow it to perform the introspection for the API. This means several things. First, it allows us to move the cache to the API gateway which will give us control over it. Second, we can have the OAuth server respond with more than just the document that explains if the token is valid or not. It can also respond with the access token in the form of a JSON Web Token (JWT). The JWT is a JSON representation of the

token, signed with the private key of the OAuth server. This JWT is then what's passed on with the request to the API. The API can then validate the signature of the token using the public key of the OAuth server and base its authorization decision on the data from the token. This makes for a very scalable platform, since all APIs can make their own authorization decision without asking anyone else. And all we need to distribute to them is the public key.

But now consider this, in a distributed environment, where there are multiple instances of the API gateway. If you're unlucky, the API requests might hit new gateways each time, so the benefits of caching would be lost. To mitigate against this, the OAuth server could be allowed to warm up the cache for the gateway instances. Depending on the gateway, that could mean to push the reference/value token pair to the gateway. Or in other cases, push to some common cache.

Token Validation

When using the Phantom Token flow, the API is able to validate the tokens using the public key of the OAuth server. To obtain the key, it can use the metadata of the server. The metadata and where it's obtained is described in RFC8414 or/and OpenID Connect Discovery depending on the server. So if your OAuth server supports one of these, it means we can get the public keys using http requests. The keys are represented in a JWKS, and look something like this.

```
{
  "keys": [{
     "kty": "RSA",
     "kid": "1555934847",
     "use": "sig",
     "alg": "RS256",
     "n": "rCwwj0H1f2Gl3W6...8QlB9R9M_DxcKRQ",
     "e": "AQAB"
  }]
}
```

This document contains one key with id 1555934847. It could contain a full list of keys.

Let's have a look at a token, and see how to validate it.

```
eyJraWQiOiIxNTU1OTM0ODQ3IiwieDV0IjoiOWdCOW9zRldSRHRSMkhtNGNmVnJnWTBGcmZ-
RIiwiYWxnIjoiUlMyNTYifQ.eyJhdF9oYXNoIjoiV3RDYWN6N3hrNHBHZDE0Y29PeTM3dy-
IsImRlbGVnYXRpb25faWQiOiJiNWZmYjMyZC0zNDdiLTQyYWQtODQzMS03MGEzM2I0N2U-
wMjIiLCJhY3IiOiJ1cm46c2U6Y3VyaXR5OmF1dGhlbnRpY2F0aW9uOmh0bWwtZm9ybT-
podG1sLXByaW1hcnkiLCJzX2hhc2giOiJraUdtTUN0YmNmUy1rZ2FUSTZXLWNRIiwidXB-
kYXRlZF9hdCI6MTU0MDE5NzU2NSwiYXpwIjoidG9vbHMiLCJhdXRoX3RpbWUiOjE1NT-
c3ODMxMjMsInByZWZlcnJlZF91c2VybmFtZSI6ImphY29iIiwiZ2l2ZW5fbmFtZSI6Ik-
phY29iIiwiZmFtaWx5X25hbWUiOiJJZGVza29nIiwiZXhwIjoxNTU3Nzg2NzMzLCJuYmY-
iOjE1NTc3ODMxMzMsImp0aSI6IjExOGYyMDJkLTcyZjctNGI5Zi05MTk0LTU5MDZiYzA-
wNjQwMiIsImlzcyI6Imh0dHBzOi8vbm9yZGljYXBpcy5jdXJpdHkuaW8vfiIsImF1ZCI-
6InRvb2xzIiwic3ViIjoiamFjb2IiLCJpYXQiOjE1NTc3ODMxMzMsInB1cnBvc2UiOiJpZCJ9.
DnY8tSaT2VoDfVUazp28JnKPnl1o0bOaCZRRx6nR31vebG8xkTQLGGD56piiwp6HroehRECtniOx-
OMuPi91w7NBqVky3jbxDNYRyfmbTMxz6TRk2k1M-Tc2d1UrQposSf-GNeMxchVB47pzArUAcnACM-
58vB83RpCzdsbv3VxdLcP9Bp8hGSU3bGKSLDJIEYlWYV9au2qYrwLA2Avzj-ZCv4qK6WxIlcbQdfHk-
w3hsF_JULTxxvMHFwE6EAzxEXu5DRiNVJqn57P_jc4wW5SLkxS0fhBXFG2LZ2tnSGaoNc3JZ5g6L-
nJ-7IXvg14NWtzLM6yPMv5Dw_KxC5bBIFjFw
```

This is a JWT. It has a header in pink, body in grey, and the signature in green. The header and body are encoded JSON documents, and the signature is encoded binary data. The parts are separated with a period ('.') character. If we decode the header, it looks like this

```
{
  "kid": "1555934847",
  "alg": "RS256"
}
```

The "kid" (key ID), points to the key in the JWKS that was used to sign this JWT, and the "alg" (algorithm) describes how it was signed. So to validate the JWT, the API can use the key from the JWKS and validate that the signature is correct. If the validation passes, the data of the body can be trusted, and the API can base its authorization decision on it. Mission accomplished!

That means to validate an incoming token, the API must do the following:

- Get server metadata
- Cache keys locally
- Validate the signature

Important to note here is that if a token comes to the API with a "kid" that is not recognized, it can mean two things. Either the server rolled its keys, or the token comes from an untrusted source. To be sure, the API must first update its keys, and if the kid still isn't found it means that the source is untrusted. This way, the server can roll its keys at any time without getting dropped requests by the API.

This works really well in all environments that can keep a state, like traditional web servers, Docker containers, Kubernetes and so on. But for other things like lambda functions we need something else.

Browser Model

For stateless functions, performing token validation the mentioned way gives a lot of overhead. The function would need to collect the metadata and keys for each request, so we obviously need something else. For these types we can use the same model that the browsers use to validate that websites are trusted while using https. Allow the OAuth Server to create a Certificate Authority (CA) that can issue sub-certificates to use to sign the tokens. The CA is then distributed with the functions, by compiling in or using some other means of the current platform.

The OAuth server can now issue JWTs with slight difference from before

```
{
  "x5c": "MIICojCCAYoC…xMjExMjJaFw0yNDAxMjcxM",
  "alg": "RS256"
}
```

Instead of the "kid" we had from before, we have a "x5c". x5c contains the full certificate that corresponds to the key used to sign the JWT. So to validate the token, the API needs to extract the certificate, validate that it's issued by the CA and validate the signature using the public key of the certificate.

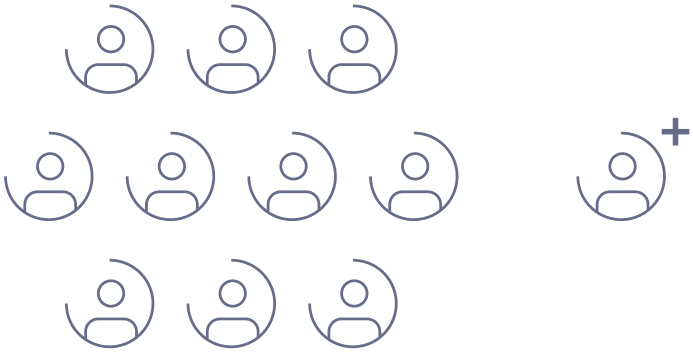
So we have enabled lambda functions to validate JWTs, without the http overhead. And the server can still roll keys by getting a new signing certificate.

Final Thoughts

By following these patterns in your platform, you allow all the components in the platform to be distributed or to dynamically scale. But maybe even more important, it allows you to enforce your access policies in both APIs and gateways. The policy enforcement can be made without calling out to a third party, since all the data needed is provided in the request.

What enables us to create these patterns is the use of standards. We separate the concern of every component in the platform, and by tying them together with the use of open standards we're not only allowing them to scale separately, but we also allow them to be replaceable. Since the glue of the components are standard protocols, it makes it easier to replace components. All of this will make you able to build a truly scalable platform.

Standardized User Management with SCIM



What is SCIM?

SCIM stands for “System for Cross-domain Identity Management” and is firstly a standardized way of representing users, groups, and anything related. Secondly, SCIM helps to standardize methods for acting on this data, such as creating, querying, searching, updating, and deleting. In other words, it’s an API model.

These two parts of SCIM are split into two standards: a Core Schema (RFC7643) controlling how the data is modelled, and a Protocol for interacting with the data (RFC7644).

But Why Do Standards, Such as SCIM, Matter?

The image to the right is from the blockbuster movie Jurassic Park (1993). You’re probably wondering how this is relevant. Well, in this movie the characters are chased by velociraptors into a room, but they unfortunately can’t lock the doors as they are controlled by a computer system. Everyone panics, until this girl discovers that the computer system controlling the doors is a UNIX system, which is a standard she is familiar with. Thanks to this standard she managed to lock the doors and survive – and that’s why standards are so important! Code shouldn’t just work well, it should also be easy to maintain, add to and debug. Using open standards makes code understandable to all developers.

Standards can still be frightening, complex, over engineered, and/or boring. With SCIM, however, all data is represented as JSON and the protocol is built on REST. Also, you're probably already handling users so there is a pretty good chance that you are familiar with some aspects of SCIM. By the end of this article, you'll hopefully be able to look at SCIM and similarly say "I know this."

How to Implement SCIM

SCIM is not meant to replace your existing systems for user management, but rather to act as a standard interface on top of them. These could be anything from SQL databases, LDAP, NoSQL data stores, SOAP, or REST APIs. SCIM has very few requirements as to what needs to be implemented. Therefore, it's recommended to implement the base features and those that make sense for your company and scale up as the need arises.

A huge plus of using a standard interface is that there is no need to document each system separately; if you have a unified way for user management, the documentation is in the specification itself. It's important to note that the SCIM specification focuses on what is needed for user management, not security. Therefore, things like how to secure access to a system and the permissions to access a system are left to other standards like OAuth.

Schema – Also Known as “The Data”

Resource. In SCIM, everything extends from the resource type, and share a set of common attributes. You’re probably already familiar with these sets of attributes, since they are common in pretty much all identity management systems. All SCIM types are identified by the schema in the payload, like the User schema below:

- `id:` Globally unique identifier
- `externalId:` Identifies the source of the data. This could be an ID from your database or a Twitter handle – wherever you got the user or resource from originally
- `meta:` Common metadata, such as a timestamp for when the resource was created and lastModified, as well as where you can find it, or the location (URL) of the given resource.

Here is an example resource as represented in JSON:

Another benefit of SCIM is that you are free to extend these fields with your own schemas or resource types.

Users – as Found in the /Users Endpoint

Central in all identity management systems is the concept of users, and SCIM is naturally no exception. The core schema defines a set of attributes that should be common for most users, as well as some that may not be as common.

So what is a user?

Within almost all systems handling users, we find some common attributes. Amongst these are:

- **Username**
- **Names** (first name, last name, etc.)
- **Contact** (phone numbers, email addresses)
- **Groups**
- **Locales** (time zone, location, etc.)
- **Password** (never visible in payload, i.e. a “write only” attribute)

Passwords are a bit special in SCIM. It’s an attribute handled in the standard, but you can never view a password when you request a user resource. You can still query and do authentication through SCIM, but you can’t get a list of passwords, or a password to a specific user.

Less common attributes are things like social media or instant messaging handles. For example, the specification even mentions ICQ!

Group – as Found in the /Groups Endpoint

Groups aren't really needed for user management, but it's pretty common so we'll include it in this article. Groups in SCIM are not much more than a name and a list of members.

SCIM Protocol – Working with Resources

These parts of the SCIM Protocol should be pretty familiar too, since it's all based on REST:

- `GET`: Fetches an existing resource, either by ID or by search
- `POST`: Sending a Post request to the user's endpoint creates a new resource
- `PUT`: Replace an existing resource
- `PATCH`: Updates attributes on an existing resource
- `DELETE`: Deletes a resource

Endpoints and Search

Each resource type is represented under an endpoint named after the resource type: users under `/Users`, groups under `/Groups`, and so on.

A regular `GET` request to one of those endpoints lists all resources for that resource type, but naturally you want to have some limitations on entries, pagination, and the current position. Browsing is as simple as passing in the "pagination" parameters (and optionally, sorting) in the request. It's also possible to show (include/exclude) specific attributes of interest.

```
{
"totalResults": 100,
"itemsPerPage": 10,
"startIndex": 1,
"schemas": ["urn:ietf:params:scim:api:messages:2.0:ListResponse"], "Resource":
[{
…
}]
}
```

When using `GET` to retrieve information it's often not ideal to show passwords or personal IDs in the URL. While `GET` requests are RESTful, it exposes parameters in the URL. Adding `/.search` to the URL makes it possible to search by `POST` as well, which is ideal for sensitive data like credentials or other personal information.

Filtered Search, Querying Resources

One of the most powerful features of SCIM, and one of the most complex, is the ability to send filtered queries. For example, you can say `/Users?filter=username eq “teddy”`, or in English: *show me all users filtered by the username equal to Teddy*.

Example `GET` requests to `/Users/.search`:

```
/Users?filter=userName eq “teddy”
/Users?filter=emails.value ew “curity.io” and meta.lastModified lt
“2017-01-01T00:00:00Z”
/Users?filter=name.familyName co “O’Malley”
/Users?filter=title pr
/Users?filter=filter=emails[type eq “work” and value co “@example.com”]
/Groups?filter=displayName eq “Curity” or displayName eq “Twobo”
/?filter=(meta.resourceType eq User) or (meta.resourceType eq Group)
```

Example `POST` request:

```
{
“schemas”: [“urn:ietf:params:scim:api:messages:2.0:SearchRequest”], “filter”:
“userName
eq \”teddy\” and password eq \”F&1!b90t111!\””
}
```

/ServiceProviderConfig

A sort of meta thing of SCIM is that it also defines what you as a service provider support. You can choose yourself what you want to support. The /ServiceProviderConfig endpoint is a way to advertise which features are supported by the service provider.

While features like this certainly add additional functionality within niche contexts, not all SCIM features make sense for all implementations. For example, advertising specific service provider features is a great resource for anyone interacting with your SCIM API.

“I Know This”

Hopefully, you’ve found this article valuable and see that SCIM isn’t all that scary — at least not as scary as being chased by dinosaurs. In the future, the image above likely illustrates what you’ll say the next time you come across a SCIM system.

This article was originally published on nordicapis.com

More on API Security

If you want more resources and information regarding identity management, OAuth, OpenID Connect and authentication, then please visit curity.io/resources. You can also sign up quarterly newsletter to get updates on new articles, whitepapers, webinars and more.

About Curity

OAuth and OpenID Connect done better

Curity is the leading supplier of API-driven identity management, providing unified security for digital services.

Curity Identity Server is the world's most powerful OAuth and OpenID Connect Server; it's used for logging in and securing millions of users' access to web and mobile apps over APIs and microservices. Curity Identity Server is built upon open standards and designed for development and operations. We enjoy the trust of large organizations in financial services, telecom, retail, energy and government services with operations across many countries which have chosen Curity for their enterprise-grade API security needs.

To learn more, visit curity.io or contact us at info@curity.io or +46 8-410 737 70.