All the concerns related to security can be broken down into the following two categories:

**Authentication:** Who you are. Use secret credentials to identify yourself.

Is the process of authenticating a user or website/application, that proves that they are who they claim they are, by providing valid credentials for verification. Authentication is usually proven through username and password, sometimes in conjunction with some other information that is known only to the user. This set of information/elements are called [factors](https://dzone.com/articles/multi-factor-authentication-mfa). Based on these factors, authentication mechanism can be divided into following sub-categories:

1. **Single-factor authentication:** Relies only on username and password.
2. **Two-factor authentication:** In addition to the above, requires another piece

of confidential information (e.g. PIN asked by banking website)

1. **Multi-factor authentication:** Uses two or more security factors (e.g. username-

password, a security PIN sent to user’s registered mobile, fingerprint)

Authentication is implemented using **ID tokens**.

**Authorization:** What you can do or what you have access to in the system. This happens after authentication process. It is implemented through policies and rules.

Authorization is implemented using **access tokens**.

[Token-based authentication/authorization](https://dzone.com/articles/jwt-token-lightweight-token-based-authentication) is a technique, where users enter their username and password once and receive a uniquely generated encrypted token in exchange. This token is then used to access protected pages or resources instead of login credentials.

Token-based authentication works by ensuring that each request to a server is accompanied by a signed token, which the server verifies for authenticity and only then responds to the request.

A ‘token’ is a piece of data that is generated by the server containing the information that is required to uniquely identify a user. It is generated as a long string of random characters and numbers. JWT (JSON Web Token) is an example. Use of tokens are preferred for following reasons:

* **Tokens are stateless** which frees the server from maintaining sessions and great for scalability
* **Tokens can be generated from anywhere**. Allows to decouple token generation from verification
* **Fine-grained access control**. Within token payload user roles, permissions even resources can be specified

**Token-based authorization implementation steps:**

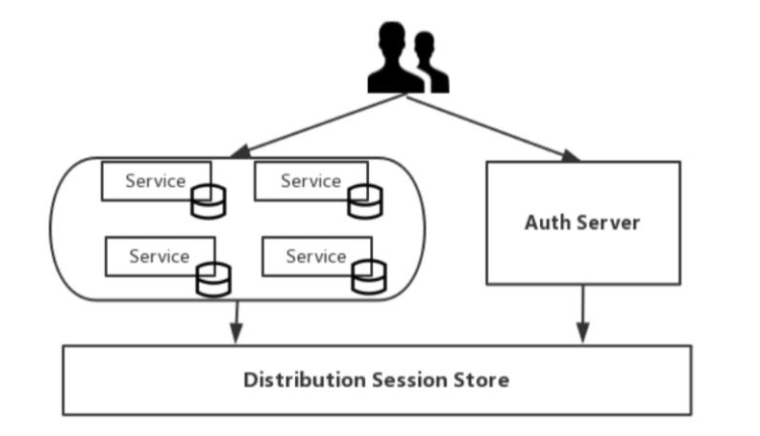
1. User requests access with username/password
2. Application validates credentials
3. On success, application provides a signed token to the client
4. Client stores the token and sends it along with every request
5. Server verifies token and responds with the data

**Authentication/Authorization in Microservices:**

A microservice handles only a single business logic. The global logic of Authentication and Authorization should not be placed in the Microservice implementation.

Authentication/Authorization in Microservice realm might involve the following:

1. **Distributed session management**



1. **Client token**

The main difference between the token and session (used in web monolith applications) is the storage is different. Session is stored centrally in the server. Tokens are held by the user themselves and are typically stored in the browser in form of cookies. The token holds the user’s identity information and each time the request is sent to the server, server can therefore determine the identity of the visitor and ascertain whether it has access to the requested resource.

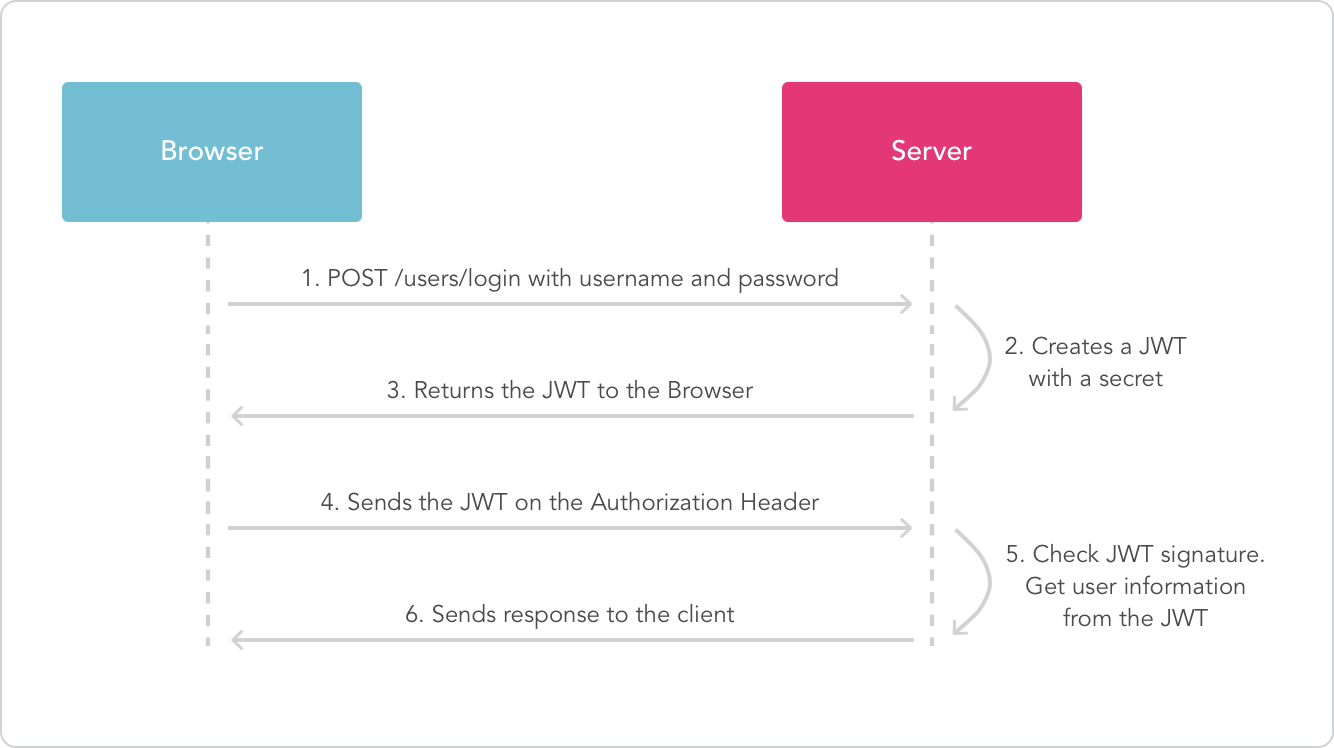
Token used to indicate user’s identity hence content of the token needs to be encrypted to avoid falsification by the requester or the third party. JWT (Json Web Token) is an open standard that defines the token format, defines the token content, encrypts it and provide libraries for various languages to implement.

The structure of JWT is simple and consists of three parts:

* Header (contains type ‘JWT’ and algorithm used)
* Payload (contains user id, name, role and expiration value)
* Signature (used by client to verify token’s identity)

These three above parts are combined using Base64 encoding and become Token strings that are eventually returned to the client separated by ‘.’

By using token for user authentication, the server does not save the user status. The client needs to send the token to server for authentication every time the server requests it.

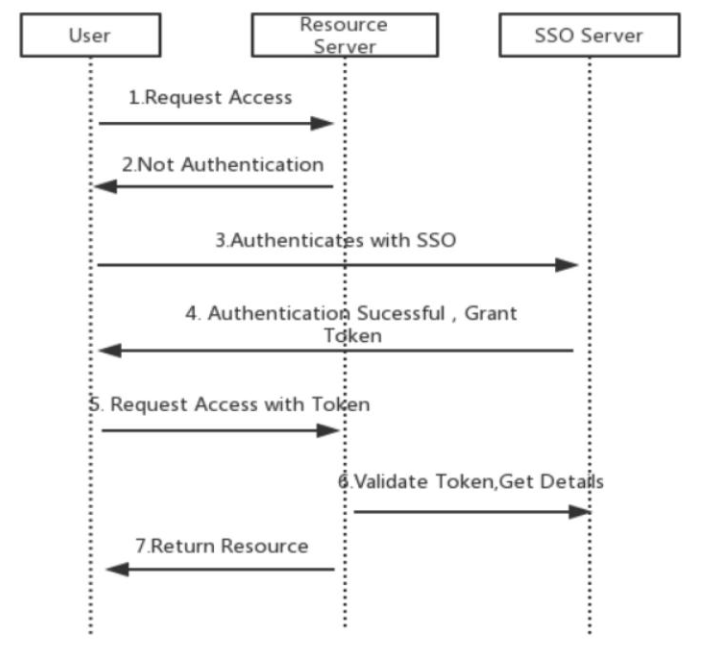


1. **Single sign-on (SSO)**

The idea of single sign-on is simple, i.e. users only need to login to the application once and then they can access all the microservices in the application. This means that each user-oriented service must interact with the authentication service like the following diagram (see next page).

Fortunately, one can leverage existing open standards and open source software to implement SSO for a distributed system. For example, one can rely on [OAuth 2.0 Authorization Framework](https://oauth.net/2/) and [JSON Web Token (JWT)](https://jwt.io/) open protocols. JWT protocol is used to standardize the sharing of a successful authentication result across client apps and resources servers. The protocol allows resources server to trust a client request without double checking with authentication server, which lowers the amount of communication within a distributed system, therefore increases the performance of overall authentication and identification.

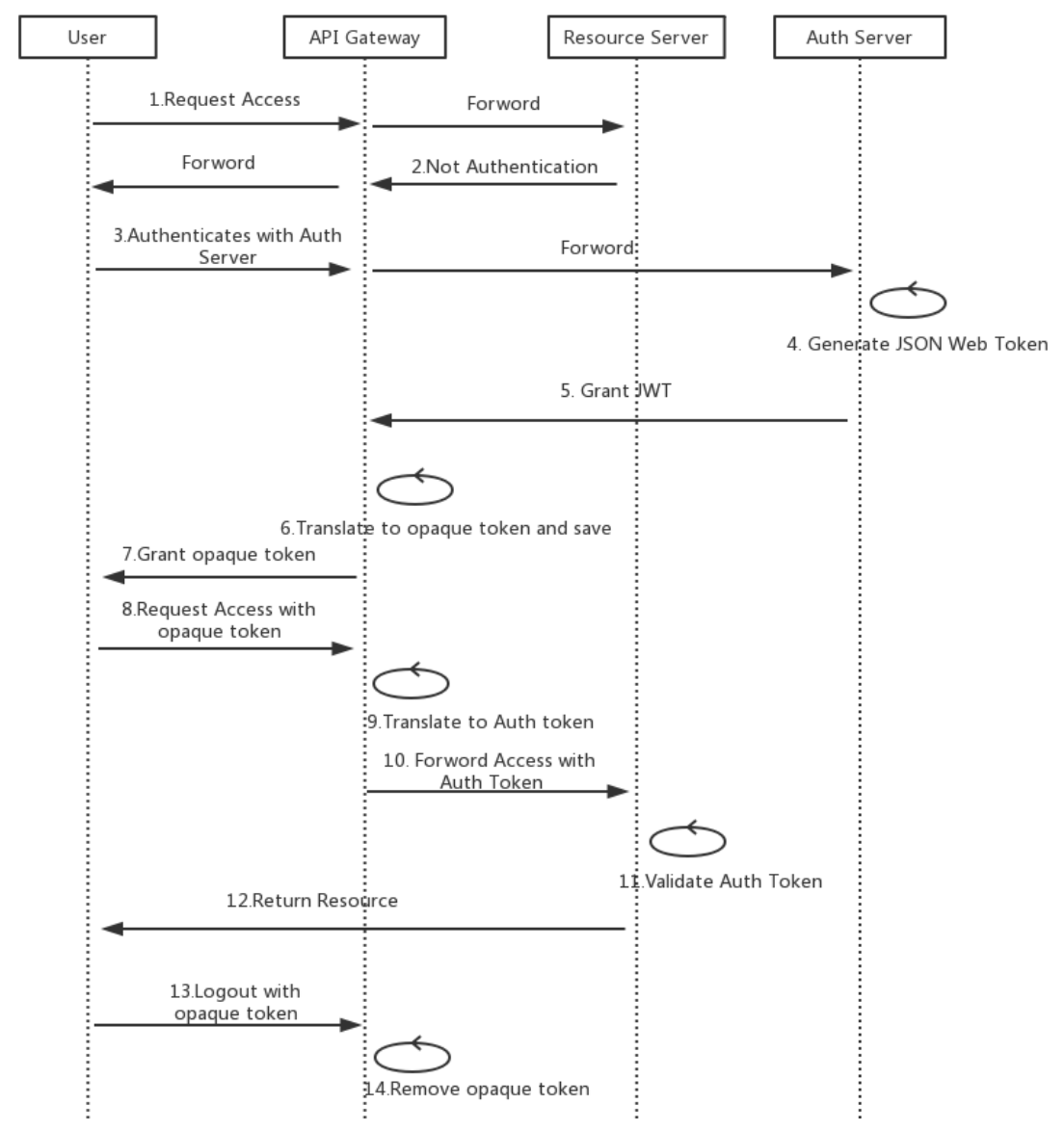
Critical requirement for the authentication server is that the server should implement OAuth 2.0 protocol and use JWT as the bearer token. As long as the authentication server implements the protocols, the rest of facilitating features can be built on any technology.



This can result in a lot of very trivial network traffic, repeated work, and it may cause single point of failure. When there are dozens of micro-applications, the drawbacks of this solution will become more apparent.

1. **Client token with API gateway**

The authentication process of user is similar to the basic process of token authentication. The difference is that API gateway is added as the entrance of external request. This scenario means that all requests go through the API gateway, effectively hiding the microservices. On request, the API gateway translates the original user token into an opaque token that only itself can resolve like the following diagram:



In this case, logging off is not a problem because the API gateway can revoke the user’s token when it logs out and also it adds an extra protection to Auth Token from being decrypted by hiding it from the client.

1. **Third-party application access**

There are two main ways:

1. **API token**

The advantage of using the API Token instead of using the username/password directly to access the API is to reduce the risk of exposing the user’s password, and to reclaim the token’s permissions at any time without having to change the password.

1. **OAuth**

Some third-party applications need to access data from different users or integrate data from multiple users. You may consider using OAuth. With OAuth, when a third-party application accesses a service, the application prompts the user to authorize a third-party application to use the corresponding access authority and generates a token for access according to the user’s permissions.

1. **Mutual authentication**

In addition to vertical traffic from users and third parties, there is a large amount of horizontal traffic between microservices. This traffic may be in the same local area network or across different data centers. Traffic between these microservices exists by third parties. The danger of sniffing and attacking also requires security controls.

Through mutual SSL, mutual authentication between microservices can be achieved, and data transmission between microservices can be encrypted through TLS. A certificate needs to be generated for each microservice, and the microservices are authenticated with each other’s certificates.

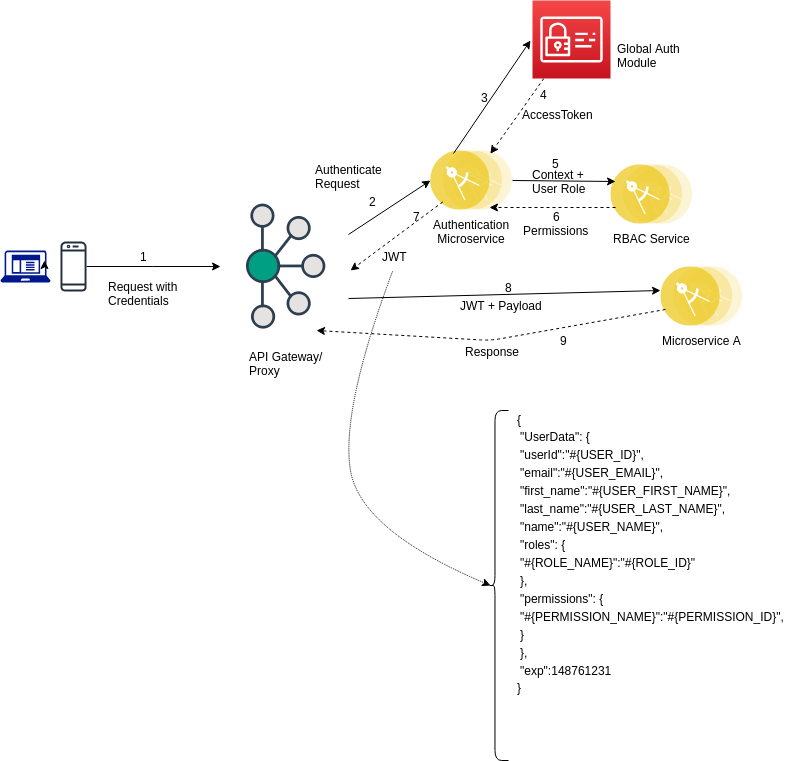
For microservice architecture, OAuth with JWT is a preferred choice.

Three major approaches for Authentication and Authorization in Microservices:

* **Local Authentication/Authorization (microservices themselves are responsible)**
  + Pros
    - Different authentication mechanisms can be implemented for each microservice
    - Authorization can be more fine-grained
  + Cons
    - Code gets bulkier and duplicated
    - Developer needs to have clear idea about permission matrix
    - Probability of making mistakes quite high
* **Global Authentication/Authorization (all or none approach)**
  + Pros
    - No repetition of code
    - Only one place to change
    - Microservice code can focus on business logic only
  + Cons
    - Finer level permission cannot be granted
    - Failure is centralized and will cause everything to stop working
* Global Authentication and Authorization is as part of Microservice
  + Pros
    - Fine grained object permissions are possible
    - Global authentication would be easier to manage
    - Authorization is managed by microservice so no network latency
    - No centralized failure for authorization
  + Cons
    - More code for developers to write
    - Needs some effort to understand what you can do with each permission

The last option looks like the best one. An authentication service can integrate with the LDAP system for verifying the user and then contacting the RBAC (Role-Based Access Control) service to populate the permission matrix based on the role the user is playing in the context of the application, e.g. the same user can be a normal user in one of the applications and an admin in another.

One needs to understand the context from which the user is coming in and RBAC is the place where the context is decoded and populate the relevant set of permissions. The permission matrix was then sent to the microservice as a part of claims in the JWT token. Microservices only apply to those permissions and return what is required to be returned. The following diagram elaborates the concept:



**OAuth 2.0**

An open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. It is basically an authorization protocol, that allows a user to grant limited access to their resources on one site to another site without having to expose their credentials.

OAuth provides to clients a ‘secure delegated access’ to server resources on behalf of a resource owner. It specifies a process for resource owners to authorize third-party access to their server resources without sharing their credentials. This is where access token comes into play.

**OAuth 2.0 roles:**

* Resource owner (RO) Application/end user who owns the data to be shared
* Resource server (RS) Application that holds the protected resources
* Auth server (AS) Application that verifies identity of the user. Issues access tokens
* Client (CL) Application that makes request to RS on behalf of the RO

There are two types of tokens: Access Token and Refresh Token

**Access token** A string representing an authorization issued to the client. Tokens represent specific scopes and duration of access, granted by the resource owner, and enforced by the resource server and authorization server.

The token may denote an identifier used to retrieve the authorization information or may self-contain the authorization information in a verifiable manner (i.e., a token string consisting of some data and a signature).

**Refresh token** Credentials used to obtain access tokens.Refresh tokens are issued to the client by the authorization server and are used to obtain a new access token when the current access token becomes invalid or expires, or to obtain additional access tokens with identical or narrower scope (access tokens may have a shorter lifetime and fewer permissions than authorized by the resource owner).

Unlike access tokens, refresh tokens are intended for use only with authorization servers and are never sent to resource servers.

**Authorization grant** Is a credential representing the resource owner’s authorization (to access its protected resources) used by the client to obtain an access token. OAuth 2.0 specification defines four grant types:

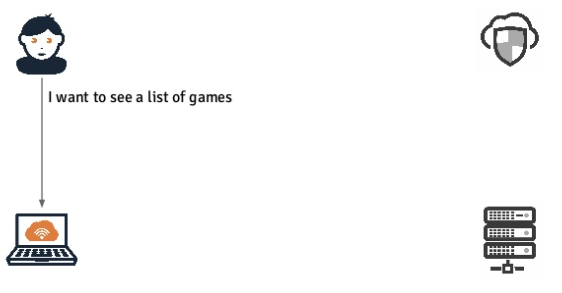
1. Authorization code

2. Implicit grant

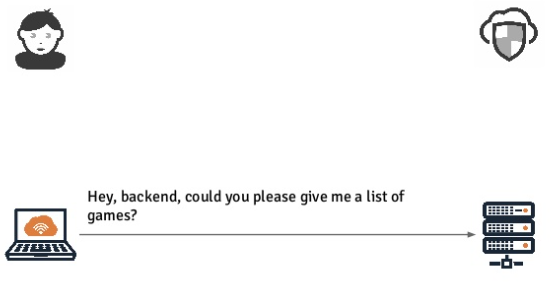
3. Resource Owner Password Credentials (ROPC)

4. Client credentials

**OAuth 2.0 protocol flow 1:** Resource owner requests to client



**OAuth 2.0 protocol flow 2:** Client requests to Resource server



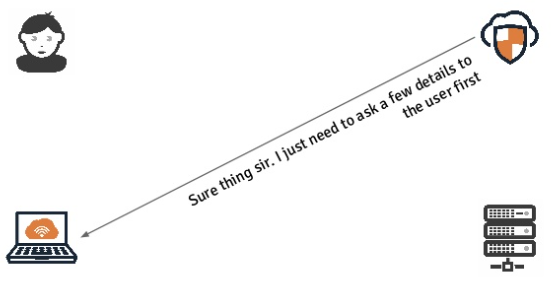
**OAuth 2.0 protocol flow 3:** Resource server responds to client



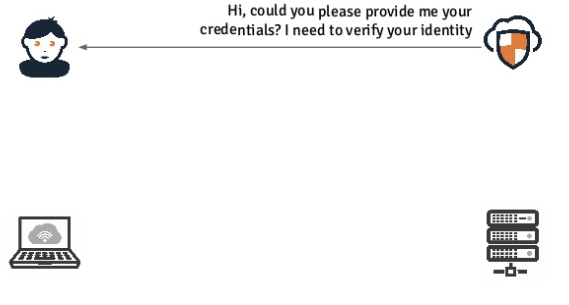
**OAuth 2.0 protocol flow 4:** Client requests to Auth server



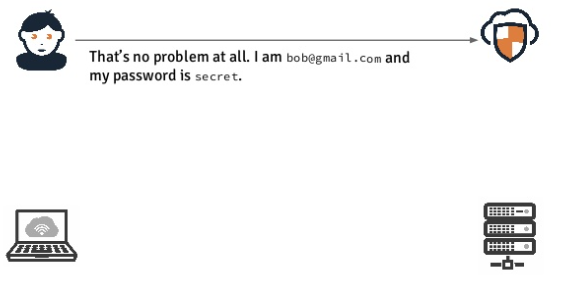
**OAuth 2.0 protocol flow 5:** Auth server responds to client



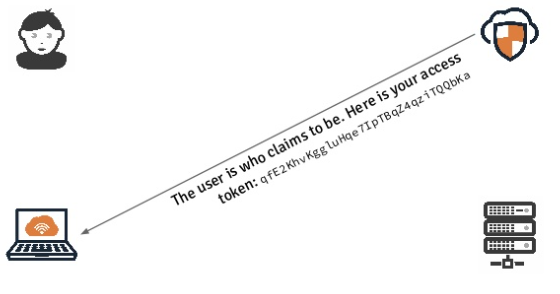
**OAuth 2.0 protocol flow 6:** Auth server requests to Resource owner



**OAuth 2.0 protocol flow 7:** Resource owner responds to Auth server



**OAuth 2.0 protocol flow 8:** Resource owner responds to Auth server



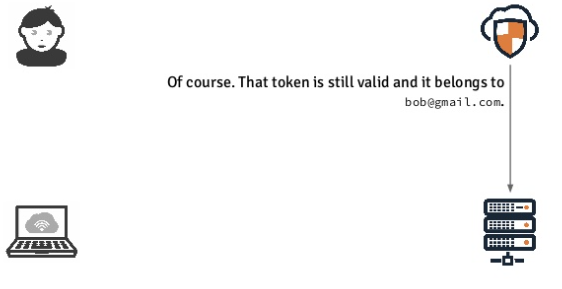
**OAuth 2.0 protocol flow 9:** Client send request to Resource server



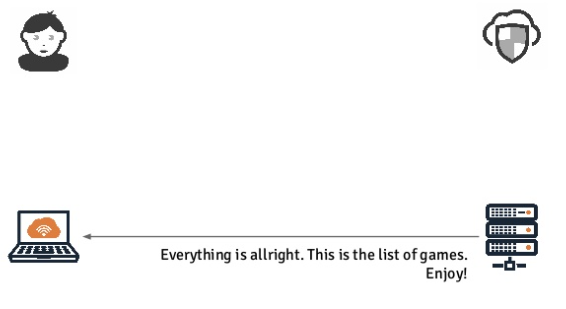
**OAuth 2.0 protocol flow 10:** Resource server send request to Auth server



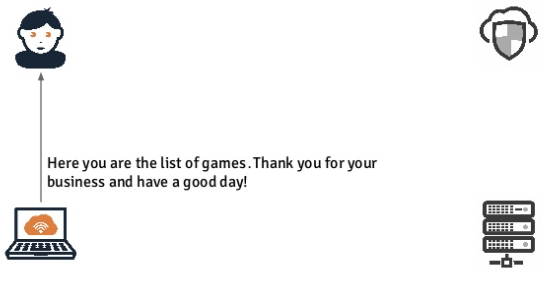
**OAuth 2.0 protocol flow 11:** Auth server responds to Resource server

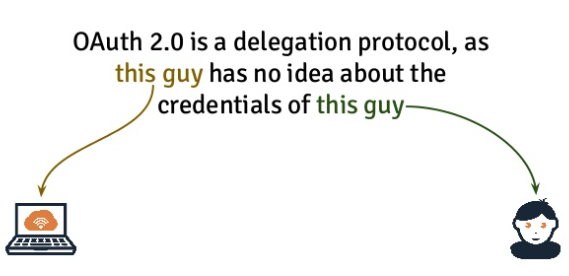


**OAuth 2.0 protocol flow 12:** Resource server responds to Client



**OAuth 2.0 protocol flow 13:** Finally client responds to resource owner





**JWT (JSON Web Token)**

JWT is a standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. They contain information in terms of claims and are specially used in in space constrained environments such as HTTP. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the [HMAC](https://dzone.com/articles/how-to-improve-oauth-security-with-hmac-validation)algorithm) or a public/private key pair using [RSA](https://dzone.com/articles/rsa-duplication-flaws-prime-exponent-and-modulus)or [ECDSA](https://dzone.com/articles/digital-signature-1).

The two striking features of JWT are:

* **Compact:** Because of it relatively small size, JWT can be sent through a

URL, through a POST parameter, or inside an HTTP header quickly.

* **Self-contained:**  JWT contains all the required information about an entity to

avoid querying a database more than once. The recipient of a JWT also does not need to call a server to validate the token.

These tokens can be signed, encrypted, or both. Signed tokens are used to verify the integrity of tokens while the encrypted token is used to hide the claims.

JWT are represented by the series of three strings separated by a dot (.), in terms of format it typically looks like the following:

**AAAAA.BBBBB.CCCCC**

The first part (AAAAA) is header, second part (BBBBB) is payload and the third part (CCCCC) is the signature.

**Header:** Usually the header consists of two key-value pairs. ‘typ’ represents what is the type of this token (JWT) and ‘alg’ denotes the algorithm used for signing this token e.g. SHA, HMAC, RSA etc.

**Payload:** This is the part which consists of claims. Claims are the information about the entity and any additional data. In JWT, the claims are denoted by the key. The following standards are applicable for claims:

* + - The claim name must be unique within a JWT claim set
    - Applications using JWTs should define which specific claims they use and when they are required or optional
    - All the claim names should be short as the core goal of JWT is to be compact

There are three types of claims e.g. Registered, Public and Private

**Signature:** The signature is generated by taking the Base64-encoded header and payload. It is then combined with a secret. Finally, this is signed with the algorithm specified in the header.

The signature is used to validate the validate that the sender of the JWT is who it says it is and to ensure that the message wasn't changed along the way.

**JWT use cases**

**Authentication:** When a user successfully logs in using their credentials, an ID Token is returned. According to the OpenID Connect (OIDC) specs, an ID Token is always a JWT.

**Authorization:** Once a user is successfully logged in, an application may request to access routes, services, or resources on behalf of that user. To do so, it uses an Access Token, which may be in the form of a JWT. Each subsequent request includes the access token.

**Single Sign-on (SSO)** widely uses JWT because of the small overhead of the format, and its ability to easily be used across different domains.

**Information exchange: J**WTs are a good way of securely transmitting information between parties because they can be signed, which means you can be sure that the senders are who they say they are. Additionally, the structure of a JWT allows you to verify that the content hasn't been tampered with.

**Advantages of using JWT**

**Decoupling** The biggest advantage of JWTs (in comparison to user session management using an in-memory random token) is that they enable the delegation of the authentication logic to a third-party server. The application server could be totally separated from the authentication server.

**Stateless** The application server can be completely stateless as JWT’s are self-contained and there is no need to keep tokens in-memory between requests. The authentication server can issue the token, send it back and then immediately discard it.

**Compact** JSON is less verbose than XML, so when it is encoded, a JWT is smaller than a SAML token. This makes JWT a good choice to be passed in HTML and HTTP environments.

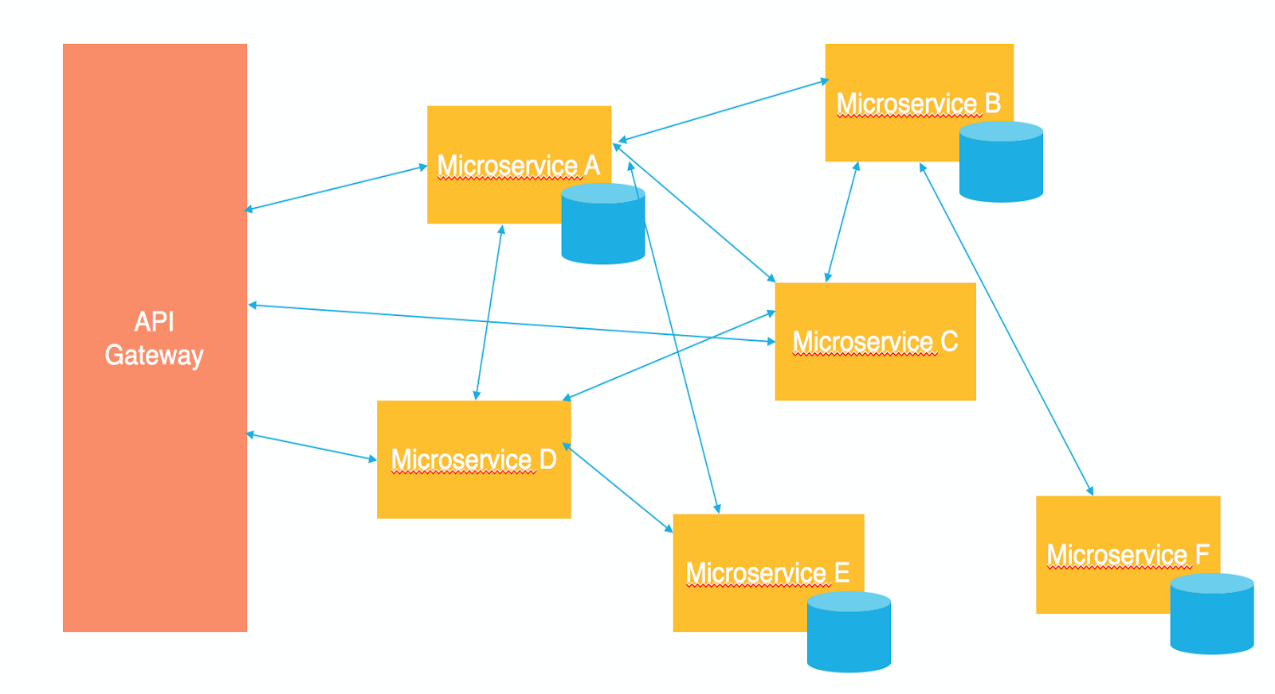
**More Secure** JWT can use a public/private key pair in the form of an X.509 certificate for signing. A JWT can also be symmetrically signed by a shared secret using the HMAC algorithm.

**More Common** JSON parsers are common in most programming languages because they map directly to objects. Conversely, XML doesn't have a natural document-to-object mapping. This makes it easier to work with JWT than SAML assertions.

**Easier to process** JWT is designed for internet scale. This means that it is easier to process on user's devices, especially mobile.

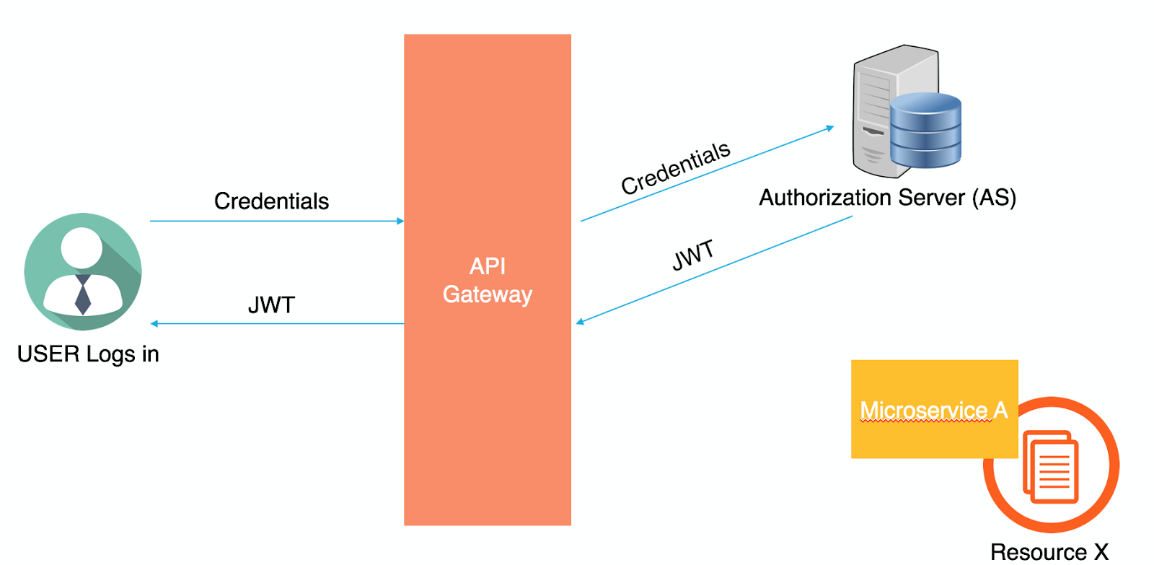
**API Gateway:**

The most obvious approach to communicate with microservice from external world is having an API gateway. This is the entry point of all the services that the application is providing.

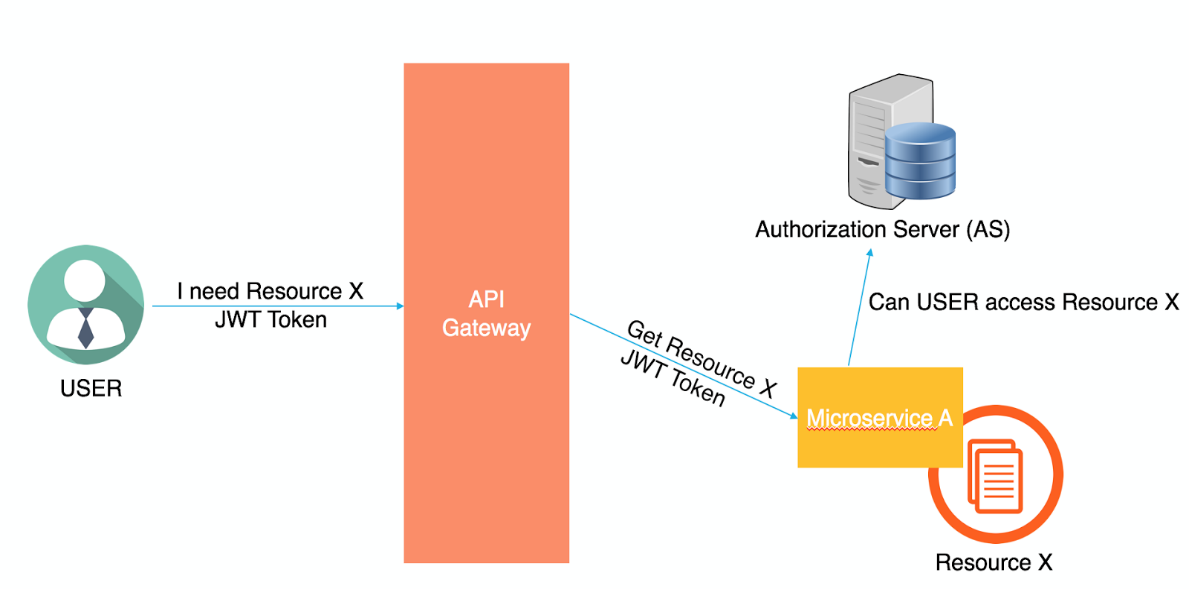


API gateway is responsible for the following things:

1. Service discovery (from the client side)
2. Routing the requests coming from external callers to the right microservices
3. Fanning out to different microservices if different capabilities are requested by an external caller
4. From security PoV, API gateways usually handle the authentication and authorization from the external callers to the microservice level. The most efficient approach to do this is OAuth delegated authorization along with JWT (JSON Web Token)



**STEP 1**



**STEP 2**

* In **STEP 1**, a user starts by sending his credentials to the API gateway which will forward the credentials to the Authorization Server (AS) or the OAuth Server. The AS will generate a JSON Web Token (JWT) and will return it back to the user.
* In **STEP 2**, Whenever the user wants to access a certain resource, he’ll request it from the API Gateway and will send the JWT along with his request. The API Gateway will forward the request with the JWT to the microservice that owns this resource. The microservice will then decide to either grant the user the resource (if the user has the required permissions) or not. Based on the implementation, the microservice can make this decision by itself (if it knows the permissions of this user over this resource) or simply forward the request to one of the Authorization Servers within the environment to determine the user’s permissions.

**Federated Identity Management (FIM) vs Single Sign-on (SSO):**

Each time an enterprise deploys a new application, end users must create a new set of credentials to remember. The result for employees? Too many passwords. In fact, the average user has to remember [at least ten passwords every day](https://www.okta.com/resources/whitepaper-passwordless-future/) - but forgets up to three of those every month.

Nearly 40% of employees reuse the same [two to four passwords](https://www.okta.com/businesses-at-work/2019/)for various accounts, and 10% have just one password for all their applications. This example of weak password hygiene means that it’s now easier than ever for hackers to use stolen credentials to access other critical data—compromising individuals and businesses alike. As it stands, organizations need to be able to provide users with easy access to all of their applications by adopting tools like [federated identity management](https://www.okta.com/blog/2019/05/what-is-federation-and-why-should-your-apps-support-it/) (FIM) and [single sign-on](https://www.okta.com/products/single-sign-on/)(SSO).

But first - it’s important that we understand the nuances of these solutions. How is FIM different from SSO? What are the ideal use cases for each approach? This article will help you understand that and more.

## What is SSO?

Much like the name implies, SSO is a function that allows users to access multiple web applications at once, using just one set of credentials. For businesses that deploy various applications for HR, payroll, and communications, an SSO solution allows employees to access each of those services with just one login. This makes it easier for users to do tlikes they no longer have to remember multiple passwords - and reduces the amount of time IT spends on password resets.

Beyond the workforce, companies can utilize SSO to help customers access various sections of one account. For instance, retail networks with many brands can use SSO to let customers access their accounts with each store from one central dashboard, enhancing their user experience. When shifting between each one, the site re-authenticates users with the same credentials.

## Breaking down Federated Identity Management (FIM)

As a tool, SSO fits within the broader model of FIM. This model was developed to address the constraints posed by early internet infrastructure, where entities on one domain could not access user information stored in other domains. For companies that operated across various domains, this was particularly problematic, as it made it difficult to create streamlined experiences for employees and customers alike.

As a solution, FIM was developed as a set of agreements and standards that help enterprises and applications share user identities. Essentially, it’s an arrangement that can be made among multiple organizations so that subscribers can use the same identifiers to access various applications. In short, it’s what allows you to sign into Spotify with your Facebook account details.

Added to that, in a FIM system, the onus of reviewing and authenticating user credentials is with an identity provider (IdP), not the applications themselves. So, when a user attempts to log into a specific service provider (SP) or application, the SP then communicates with the IdP to authenticate the user. This user identity authorization is often executed through open-sourced [Security Assertion Markup Language (SAML),](https://developer.okta.com/blog/2019/07/30/saml-whats-behind-sso) or other related standards like OAuth or OpenID Connect.

## Where the difference lies

The key difference between SSO and FIM is while SSO is designed to authenticate a single credential across various systems within one organization, federated identity management systems offer single access to a number of applications across various enterprises.

So, while SSO is a function of FIM, having SSO in place won’t necessarily allow for federated identity management. That said, both tools are crucial in supporting organizations with both securing their data and minimizing obstacles in user experience.