

# Microservices Authentication and Authorization Solutions



Mina Ayoub

Apr 24, 2018 · 11 min read



Microservices Architecture brings many benefits to software applications, including small development teams, shorter development cycles, flexibility in language selection, and enhanced service scalability.

At the same time, many complex problems of distributed systems have also been introduced. One of the challenges is how to implement a flexible, secure and efficient authentication and authorization scheme in the Microservices Architecture. This article will try to conduct a more complete discussion on this issue.

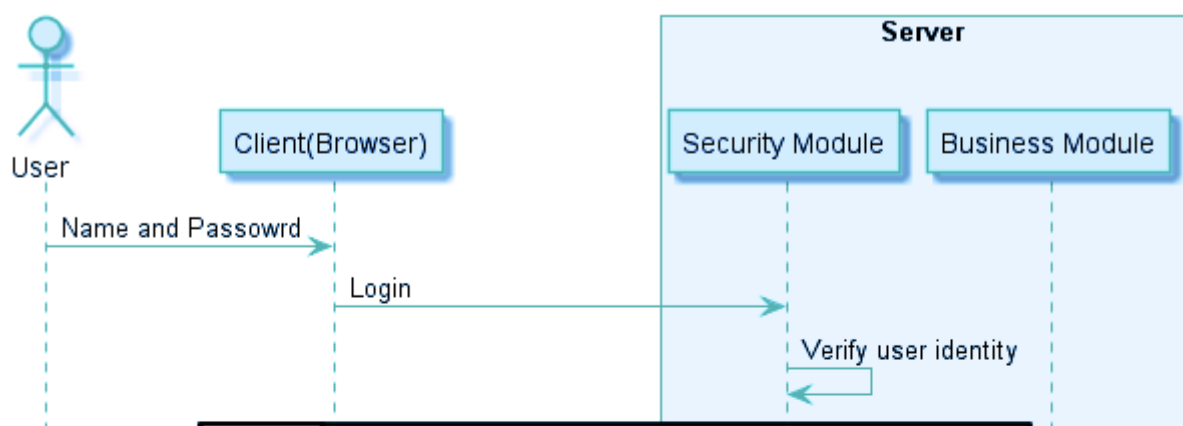
## Monolithic Application Authentication and Authorization

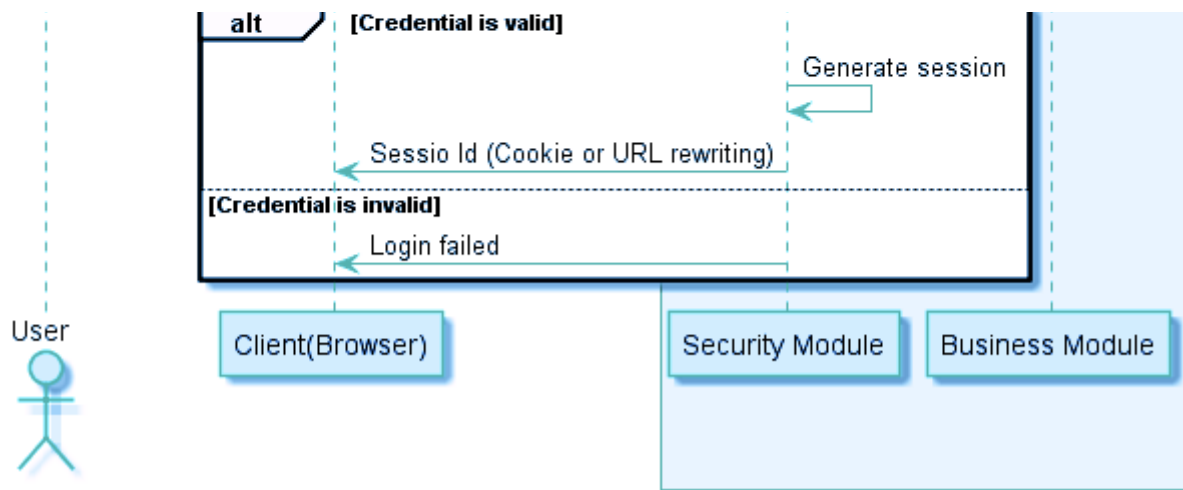
It has been confusing to differentiate between authentication and authorization. In fact, it is very simple.

- **Authentication:** Refers to verify *who you are*, so you need to use username and password for authentication.
- **Authorization:** Refers to *what you can do*, for example access, edit or delete permissions to some documents, and this happens after verification passes.

In the monolithic architecture, the entire application is a process. In the application, a security module is generally used to implement user authentication and authorization.

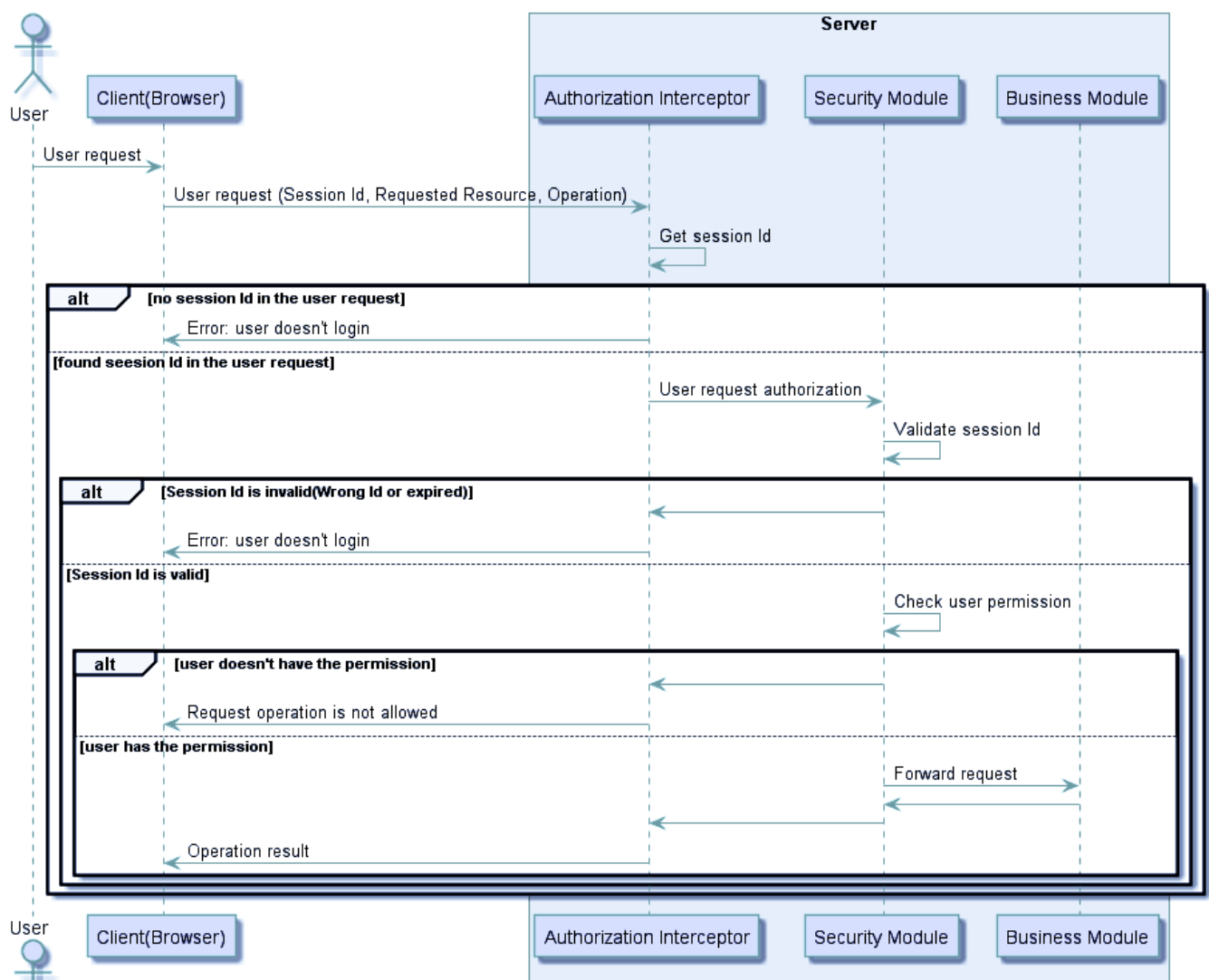
When the user logs in, the security module of the application authenticates the identity of the user. After verifying that the user is legitimate, a session is created for the user, and a unique session ID is associated with the session. A session stores login user information such as User name, Role, and Permission. The server returns the Session Id to the client. The client records the Session Id as a cookie and sends it to the application in subsequent requests. The application can then use the Session Id to verify the user's identity, without having to enter a user name and password for authentication each time.





Monolithic application user login authentication diagram

When the client accesses the application, Session Id is sent to the application along with the HTTP request. The security module generally processes all received client requests through an authorization interceptor. This interceptor first determines whether the Session Id exists. If the Session Id exists, it knows that the user has logged in. Then, by querying the user rights, it is determined whether the user can execute the request or not.





## Microservices authentication and authorization problems

Under the microservice architecture, an application is split into multiple microservice processes, and each microservice implements the business logic of one module in the original single application. After the application is split, the access request for each microservice needs to be authenticated and authorized. If you reference to the implementation of Monolithic application, you will encounter the following problems:

- Authentication and authorization logic needs to be handled in each microservice, and this part of the global logic needs to be implemented repeatedly in each microservice. Although we can use the code base to reuse part of the code, this will in turn cause all micro services to have a dependency on a particular code base and its version, affecting the flexibility of the microservice language/framework selection.
- Microservices should follow the principle of single responsibility. A microservice only handles a single business logic. The global logic of authentication and authorization should not be placed in the microservice implementation.
- HTTP is a stateless protocol. For the server, each time the user's HTTP request is independent. Stateless means that the server can send client requests to any node in the cluster as needed. The stateless design of HTTP has obvious benefits for load balancing. Because there is no state, user requests can be distributed to any server. For services that do not require authentication, such as browsing news pages, there is no problem. However, many services, such as online shopping and enterprise management systems, need to authenticate the user's identity. Therefore, it is necessary to save the user's login status in a manner based on the HTTP protocol so as to prevent the user from needing to perform verification for each request. The traditional way is to use a session on the server side to save the user state. Because the server is stateful, it affects the horizontal expansion of the server.
- The authentication and authorization in the microservices architecture involves scenarios that are more complex, involving users accessing microservice applications, third-party applications accessing microservice applications, and multiple microservice applications accessing each other, and in each scenario, The

following authentication and authorization schemes need to be considered to ensure the security of the application.

# Microservices authentication and authorization technical solutions

## 1. Distributed Session Management

In order to make full use of benefits of the microservice architecture and to achieve the scalability and resiliency of the microservices, the microservices are preferably to be stateless.

This solution can be applied through different ways like:

- **Sticky session**

Which ensures that all requests from a specific user will be sent to the same server who handled the first request corresponding to that user, thus ensuring that session data is always correct for a certain user. However, this solution depends on the load balancer, and it can only meet the horizontally expanded cluster scenario, but when the load balancer is forced suddenly for any reason to shift users to a different server, all of the user's session data will be lost.

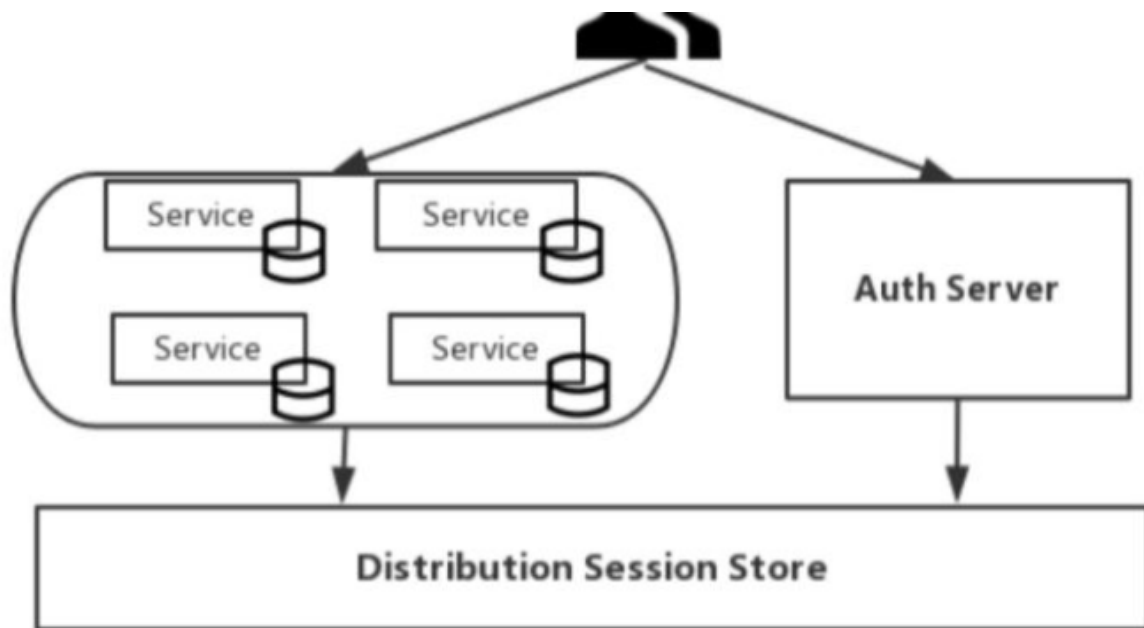
- **Session replication**

Means that each instance saves all session data, and synchronizes through the network. Synchronizing session data causes network bandwidth overhead. As long as the session data changes, the data needs to be synchronized to all other machines. The more instances, the more network bandwidth the synchronization brings.

- **Centralized session storage**

Means that when a user accesses a microservice, user data can be obtained from shared session storage, ensuring that all microservices can read the same session data. In some scenarios, this scheme is very good, and the user login status is opaque. It is also a highly available and scalable solution. But the disadvantage of this solution is that shared session storage requires a certain protection mechanism and therefore needs to be accessed through a secure way.





## 2. Client Token

The traditional way is to use a session on the server side to save the user state. Because the server is stateful, it has an impact on the horizontal expansion of the server. It is recommended to use Token to record user login status in the microservice architecture.

The main difference between Token and Session is where the storage is different. Sessions are stored centrally in the server; Tokens are held by the user themselves and are typically stored in the browser in the form of cookies. The Token holds the user's identity information, and each time the request is sent to the server, the server can therefore determine the identity of the visitor and determine whether it has access to the requested resource.

The Token is used to indicate the user's identity. Therefore, the 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 (RFC 7519) that defines the Token format, defines the Token content, encrypts it, and provides lib for various languages.

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

- **Header**

header contains type, fixed value JWT. Then the Hash algorithm used by JWT.

```
{  
  "typ": "JWT",
```

```
"alg": "HS256"
}
```

- **Payload**

includes standard information such as the user id, expiration date, and user name. It can also add user roles and user-defined information.

```
{
  "id": 123,
  "name": "Mena Meseha",
  "is_admin": true,
  "expire": 1558213420
}
```

- Signature

Token's signature is used by the client to verify the Token's identity and also to verify the message wasn't changed along the way.

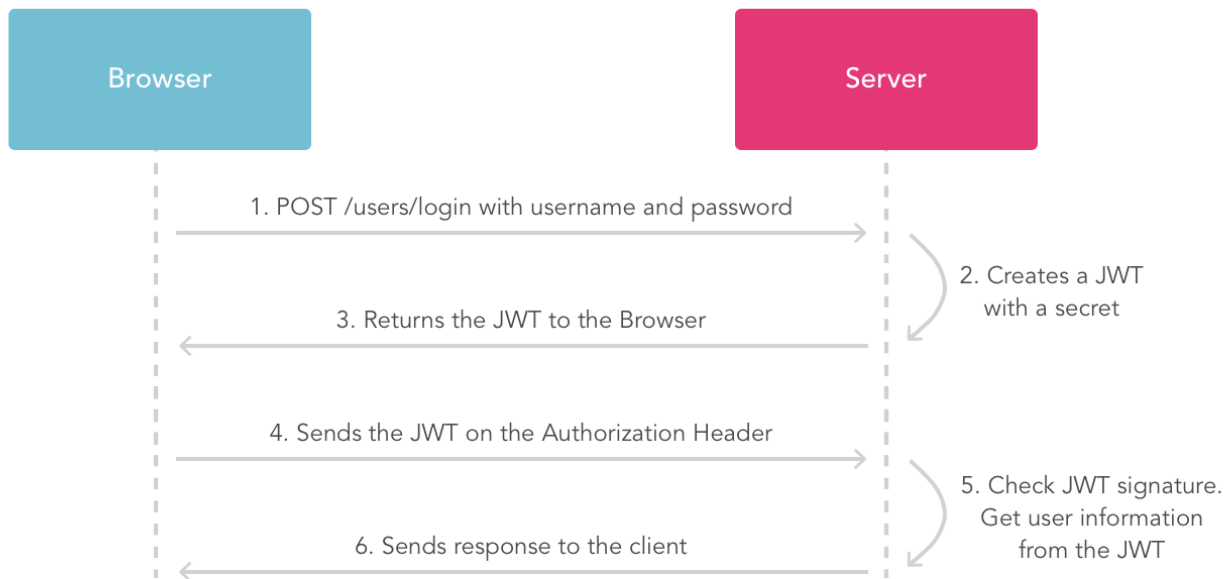
```
HMACSHA256(
    base64UrlEncode(header) + "." +
    base64UrlEncode(payload),
    secret
)
```

These three parts are combined using Base64 encoding and become Token strings that are eventually returned to the client, separated by “.”, The token formed by the above example will be like this:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MTIzLCJuYW1lIjoiTWVuYSBNZXNlaGEiLCJpc19hZG1pbSI6dHJ1ZSwiZXhwaXJlIjoxNTU0MjEzNDIwIiwiaWF0IjpmYy0aK0miLaKFLxb5d3rOC71DHexncHAcQ

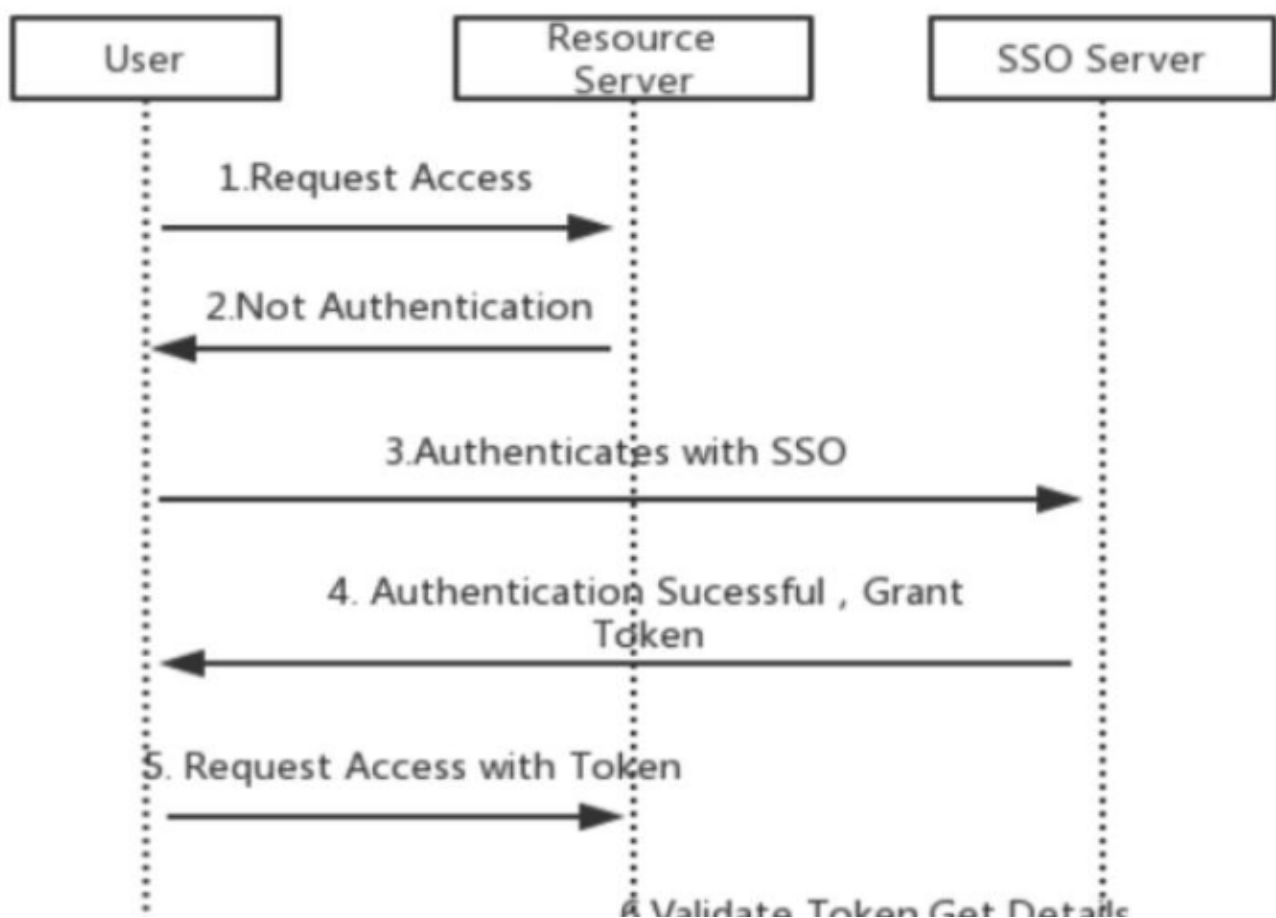
By using token for user authentication, The server does not save the user status. The client needs to send the token to the server for authentication every time the client requests it.

The basic flow of user authentication in token mode is as the following diagram:

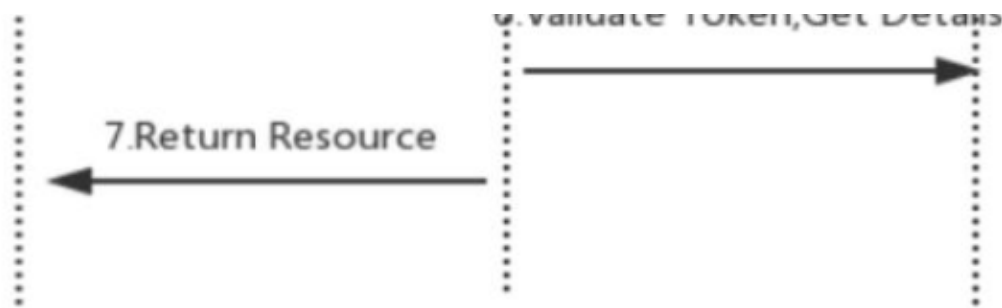


### 3. Single sign-on

The idea of single sign-on is simple, that is, users only need to log in to the application once, then they can access all the microservices in the application. This solution means that each user-oriented service must interact with the authentication service like the following diagram:



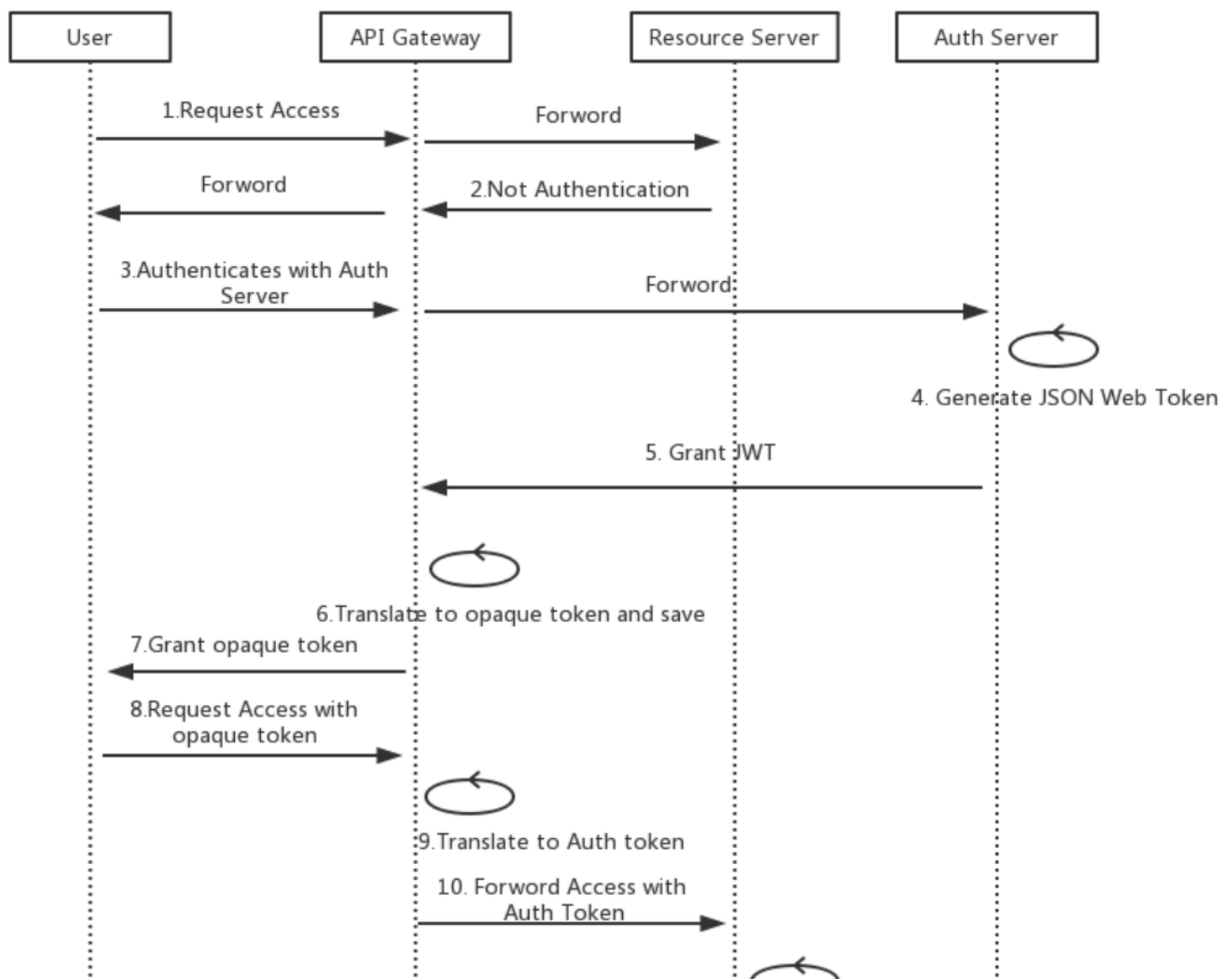


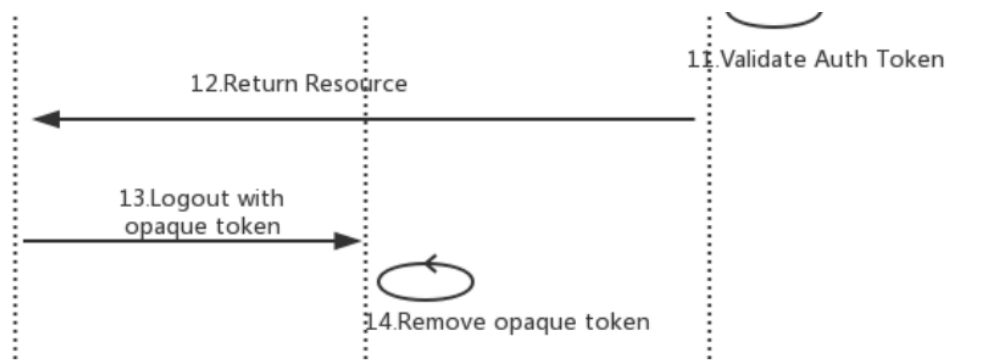


This can result in a lot of very trivial network traffic, repeated work, and it may cause single point of failure. When there are dozens of micro-applications, the drawbacks of this solution will become more apparent.

## 4. Client Token with API Gateway

The authentication process of the user is similar to the basic process of token authentication. The difference is that the API Gateway is added as the entrance of the 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:





Client Token with API Gateway solution

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.

## 5. Third-party application access

### 1. API Token

The third party uses an application-issued API Token to access the application's data. The Token is generated by the user in the application and provided for use by third-party applications. In this case, generally only third-party applications are allowed to access the user's own data of the Token, but not other users' sensitive private data.

For example, Github provides the Personal API Token function. Users can create a Token in Github's developer settings interface, and then use the Token to access the Github API. When creating a Token, you can set which data the Token can access to the user, such as viewing Repo information, deleting Repo, viewing user information, updating user information, and so on.

Using the API Token to Access the Github API is like the following command:

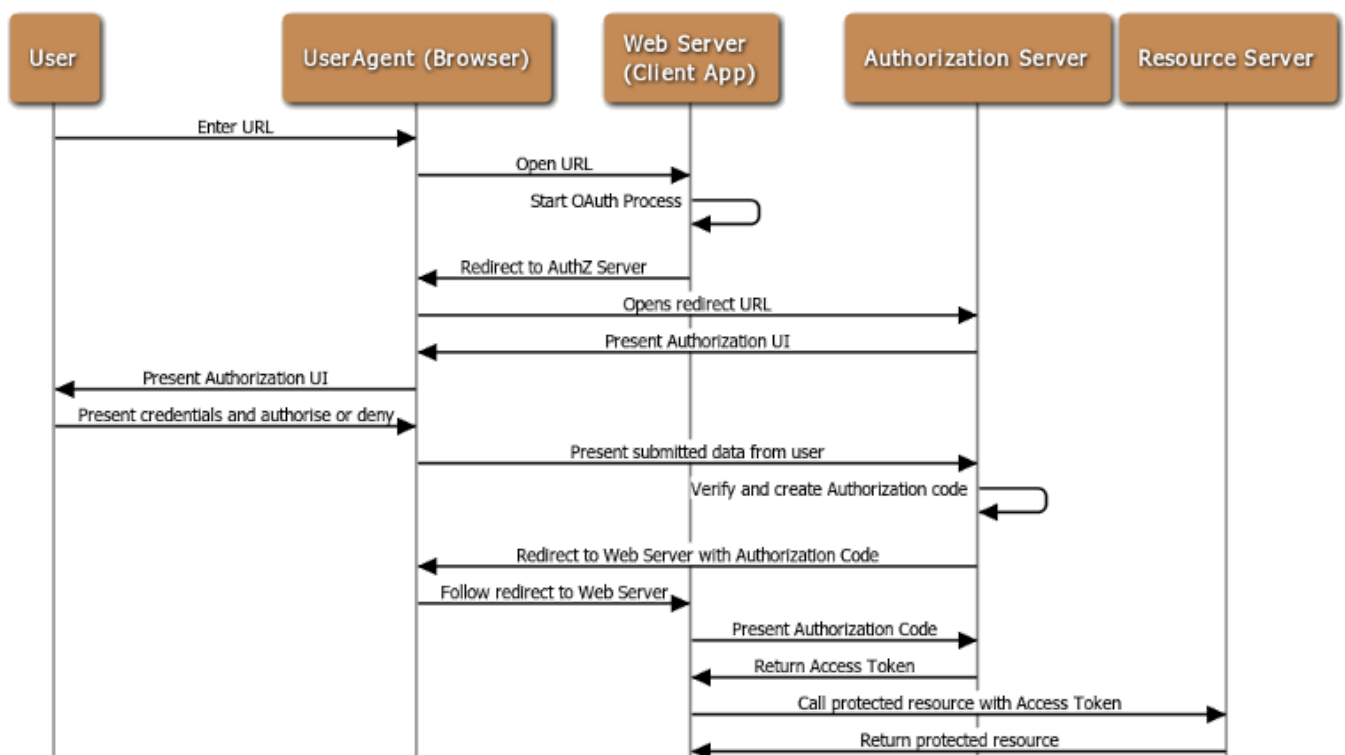
```
curl -u menameseha:f3kdfvf8e882424ed0f3bavmvd188c01acd34eec
https://api.github.com/user
```

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.

### 2. 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.

In Github, for example, some third-party applications such as GitBook or Travis CI, are integrated via OAuth and Github. OAuth has different authentication processes for different scenarios. A typical authentication process is shown in the following diagram:



OAuth authentication process

*In the above example, the resource server and the authorization server are both Github, the client program is GitBook or Travis CI, and the user is a direct user of the client program.*

Someone may wonder why an Authorization Code is used to request Access Token, rather than returning the Access Token to the client directly from the authorization server. The reason why OAuth is designed in this way is to pass through the user agent (browser) during the process of redirecting to the client's Callback URL. If the Access Token is passed directly, there is a risk of being stolen.

By using the authorization code, the client directly interacts with the authorization server when applying for the access token, and the authorization server also authorize the client when processing the client's token request, so it's prevent others from forging the client's identity to use the authentication code.

When implementing user authentication of the microservice itself, OAuth may also be used to delegate user authentication of the microservice to a third-party authentication service provider.

The purpose of using OAuth for user authorization of third-party application access and microservices is different. The former is to authorize private data access rights of users in microservices to third-party applications. Microservices are authorization and resource servers in the OAuth architecture. The purpose of the latter is to integrate and utilize the OAuth authentication service provided by a well-known authentication provider, which simplifies the cumbersome registration operation, in this case the microservice act the role of the client in the OAuth architecture.

Therefore, we need to distinguish between these two different scenarios so as to avoid misunderstandings.

## **6. Mutual Authentication**

In addition to vertical traffic from users and third parties, there is a large amount of horizontal traffic between microservices. These 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. In the microservice operating environment, there may be a large number of microservice instances, and the microservice instances often change dynamically, such as adding service instances as the level expands. In this case, creating and distributing certificates for each service becomes very difficult. We can create a private certificate center (Internal PKI/CA) to provide certificate management for various microservices such as issuing, revoking, and updating.

## **Thinking Summary**

Under the microservice architecture, I prefer to use OAuth and JWT together. OAuth is generally used in third-party access scenarios to manage external permissions, so it is more suitable for integration with API gateways and of course, the underlying Token standard JWT is also possible. The JWT is more lightweight, and for access authentication between microservices it is already sufficient. Of course, from the

perspective of capability of implementation, the distributed session for example, can fully meet the requirements in many scenarios. Specifically, how to choose an authentication scheme depends on your actual needs.

[Microservices](#)   [Authentication](#)   [Security](#)   [Jwt](#)   [Oauth](#)

[About](#)   [Help](#)   [Legal](#)