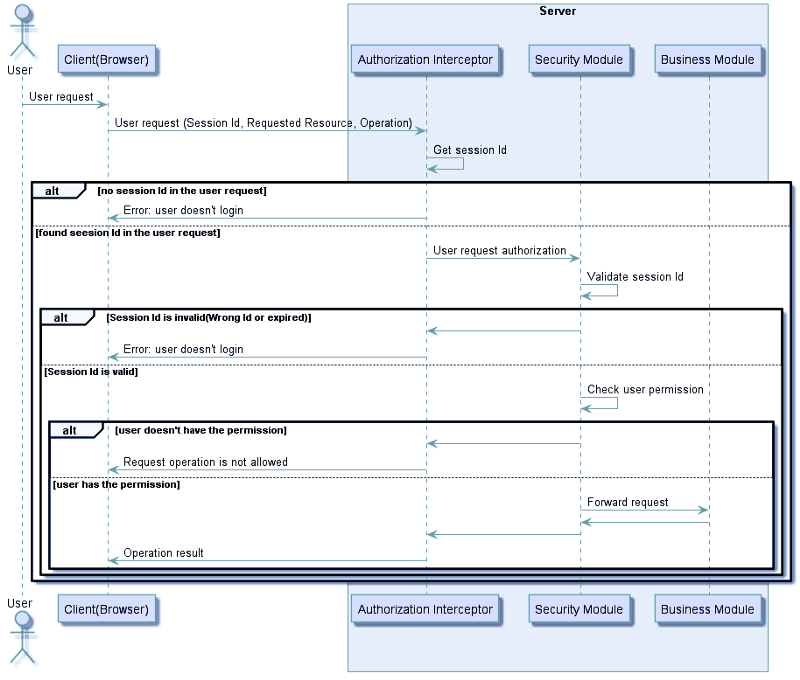
**Discussion on Authentication process of Traditional application and Microservice based application**

**Monolithic Application Authentication and Authorization**

* **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.



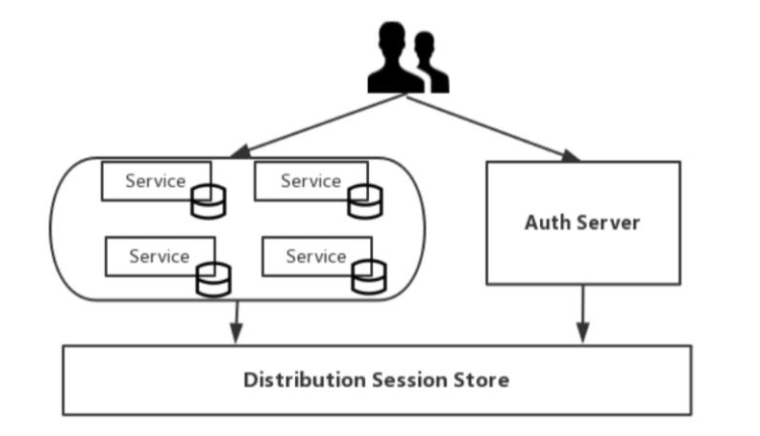
**Microservices authentication and authorization**

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 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)](https://jwt.io/) 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.
* **Payload**

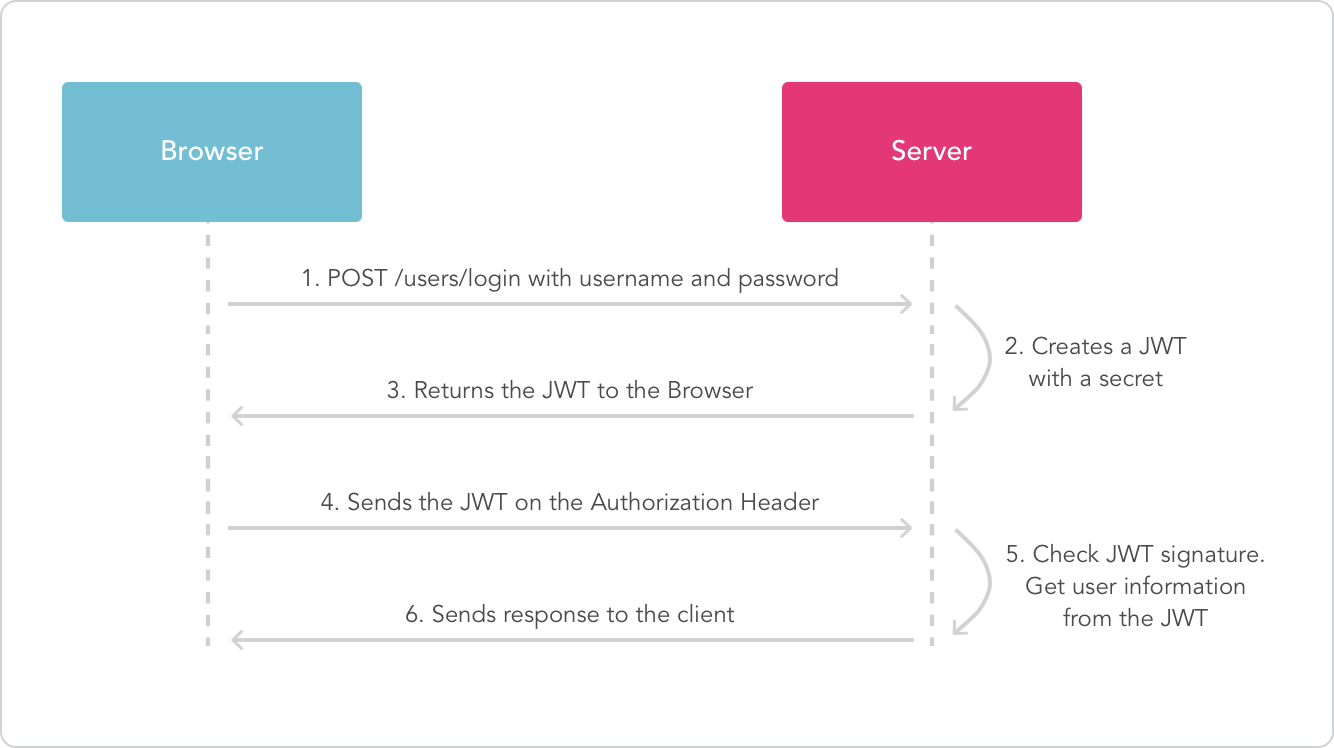
includes standard information such as the user id, expiration date, and user name. It can also add user roles and user-defined information.

* **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.

These three parts are combined using Base64 encoding and become Token strings

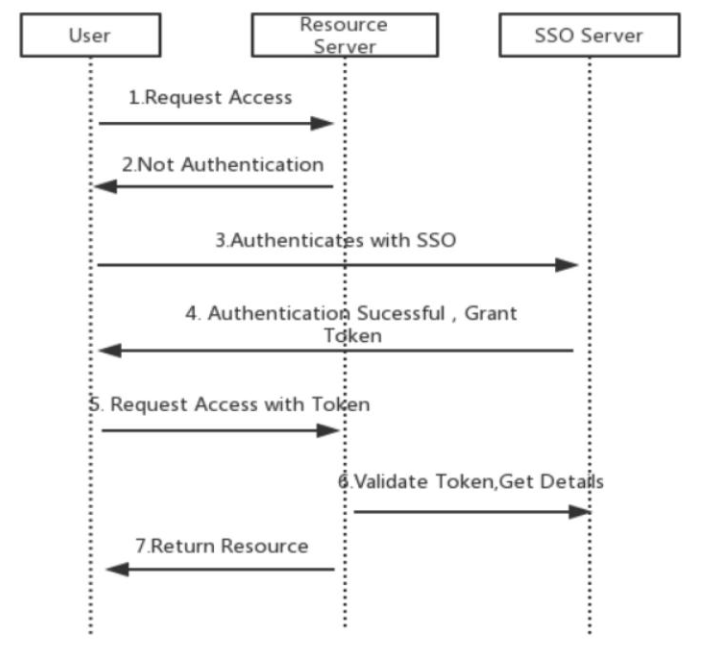
By using token for user authentication, The server does not save the user status. The client needs to send the token to the server for authentication every time the client requests it.

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



**3. Single sign-on**

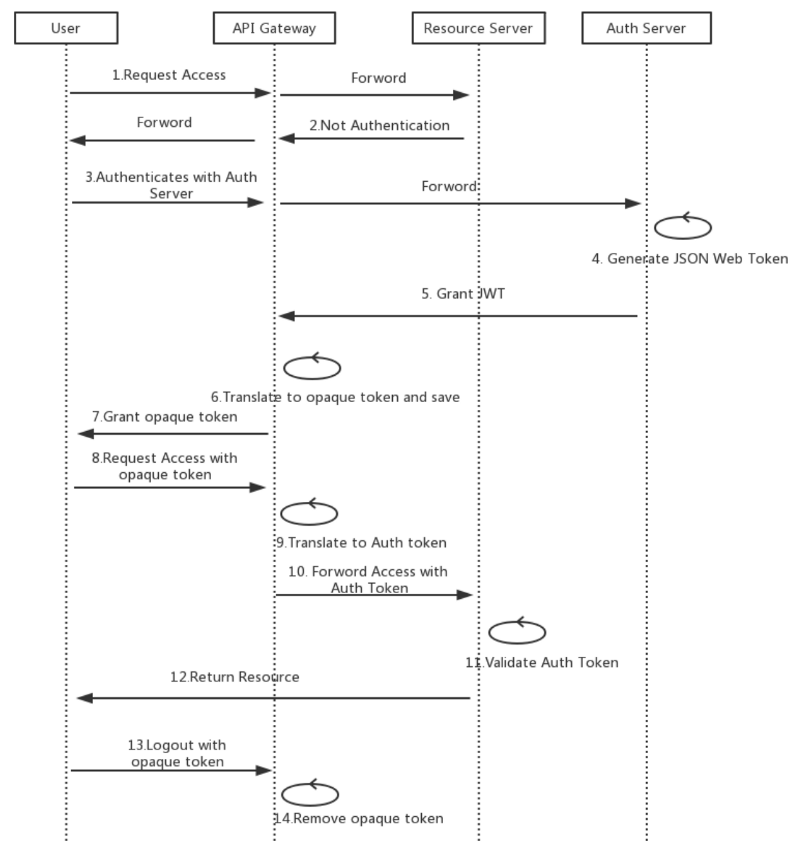
The idea of ​​single sign-on is simple, that is, users only need to log in to the application once, then they can access all the microservices in the application. This solution means that each user-oriented service must interact with the authentication service like the following diagram:



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:



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](https://github.com/settings/tokens" \t "_blank), 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:

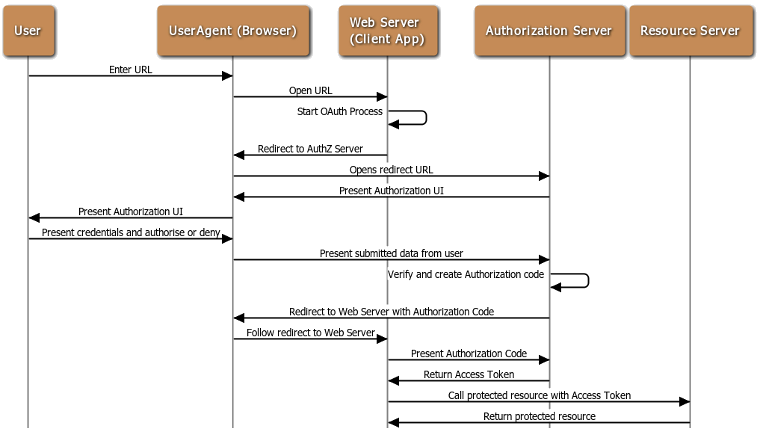
curl -u <username>:f3kdfvf8e882424ed0f3bavmvdl88c01acd34eec https://api.github.com/user

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**

***I****n 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.

**Conclusion:**

Need to be discussed and decided