Access Security

Alessandro Bocci

name.surname@unipi.it

Advanced Software Engineering (Lab)

21/11/2024

What will you do?

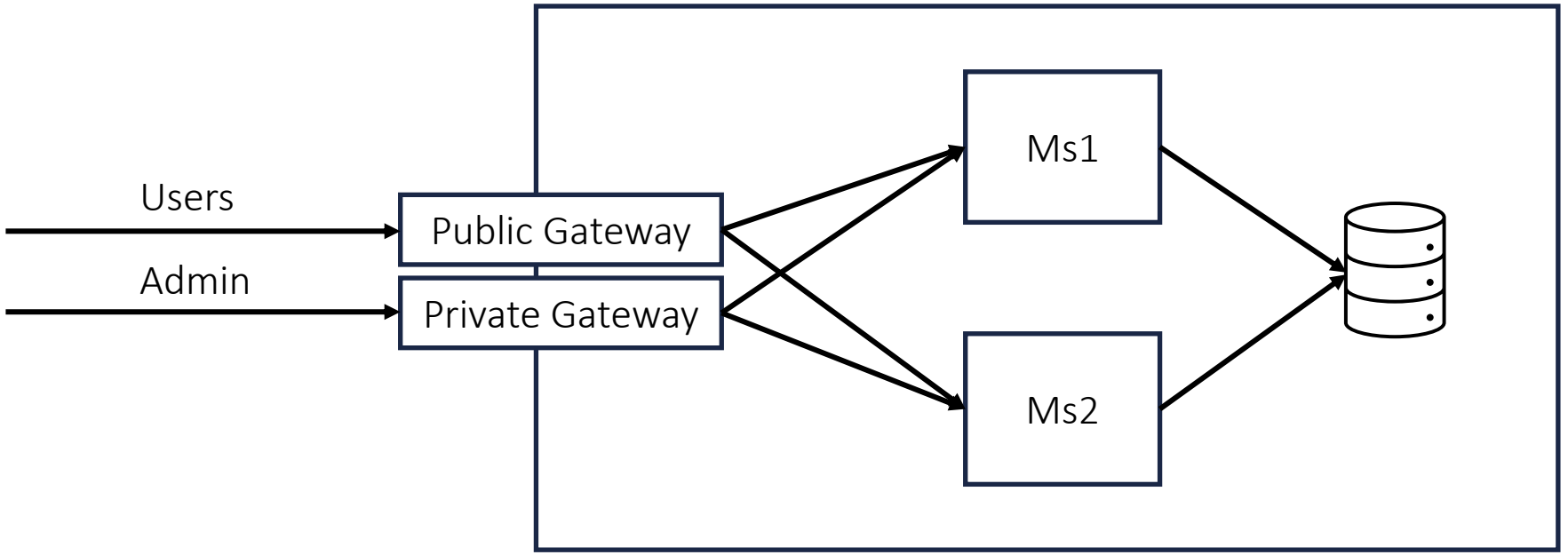
Employ mechanisms to protect a microservice architecture communication.

In particular:

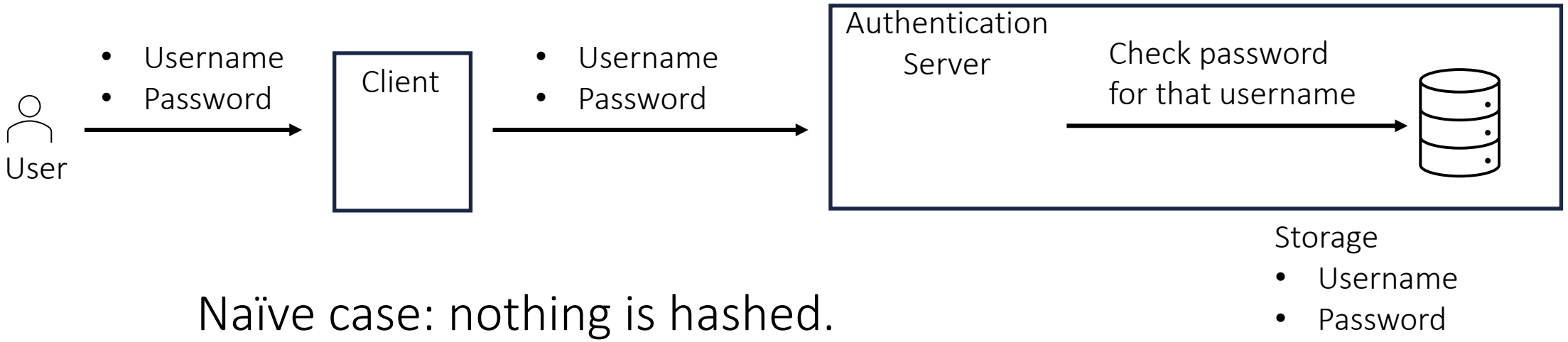
- Manage users' credentials.
- Authentication.
- Authorization.

Access to the architecture

- Different APIs for ‘general’ users and ‘special’ users.
- Better if different networks, e.g. one public and one with a private VPN.



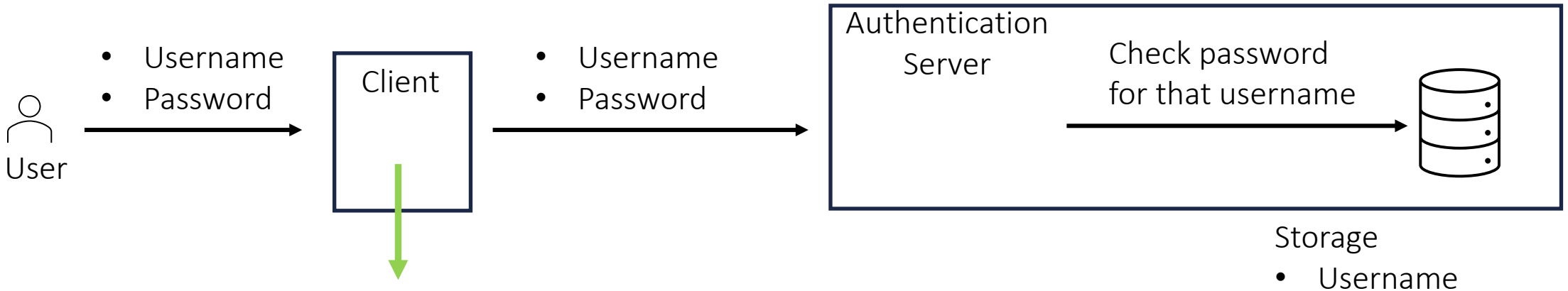
Credentials management



Naïve case: nothing is hashed.

(Client and Server use HTTPS)

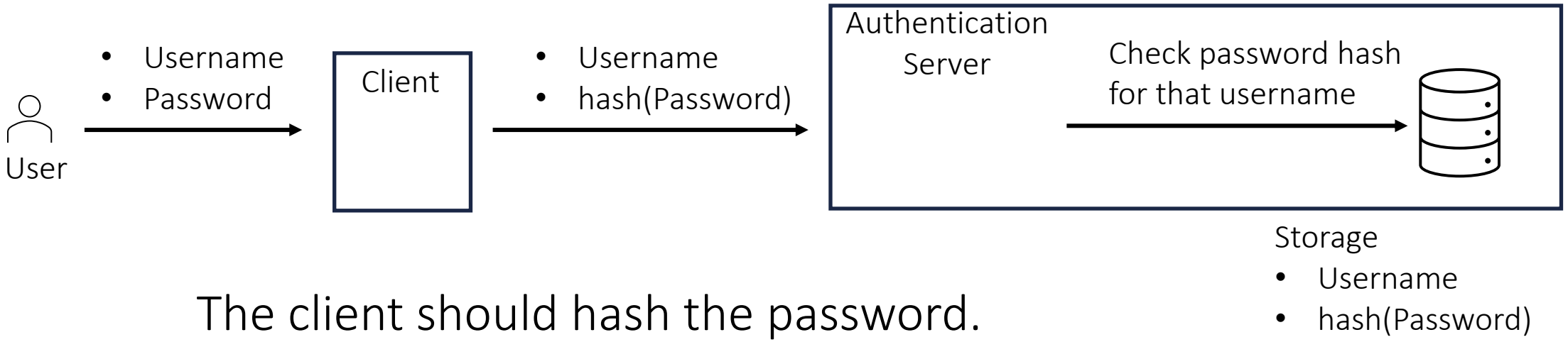
Credentials management



The client should hash the password.

This to avoid that the server knows a password for a user and try to use it for a different service.

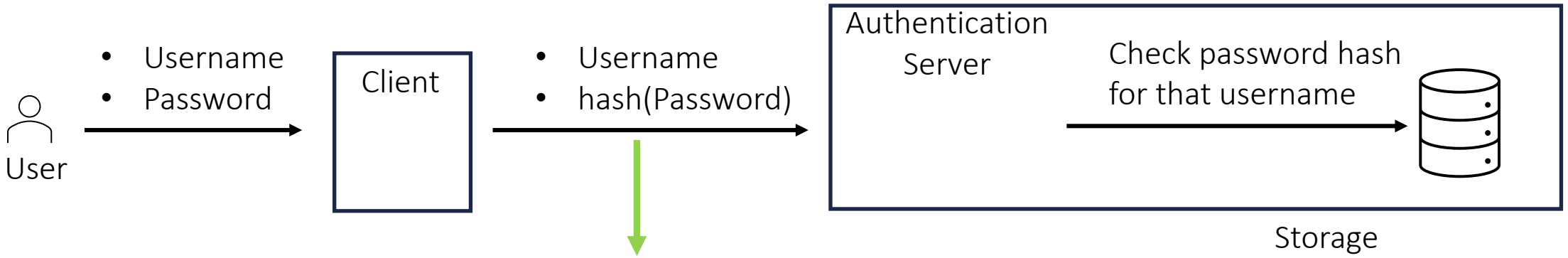
Credentials management



The client should hash the password.

This to avoid that the server knows a password for a user and try to use it for a different service.

Credentials management



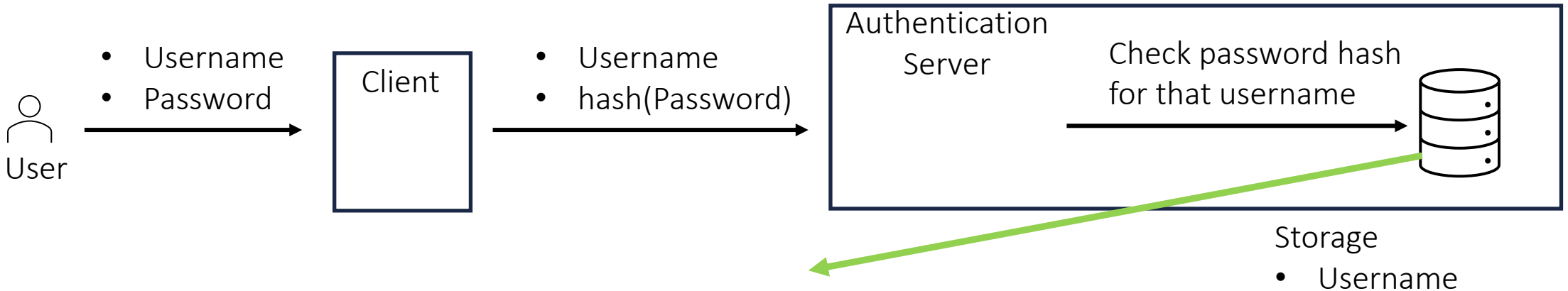
Q: How to send them in REST?

A: As encrypted (HTTPS) payload.

Using query strings (`?user=aa&pw=aa`) is not safe!

They can be cached by browsers, exposed via redirection etc.

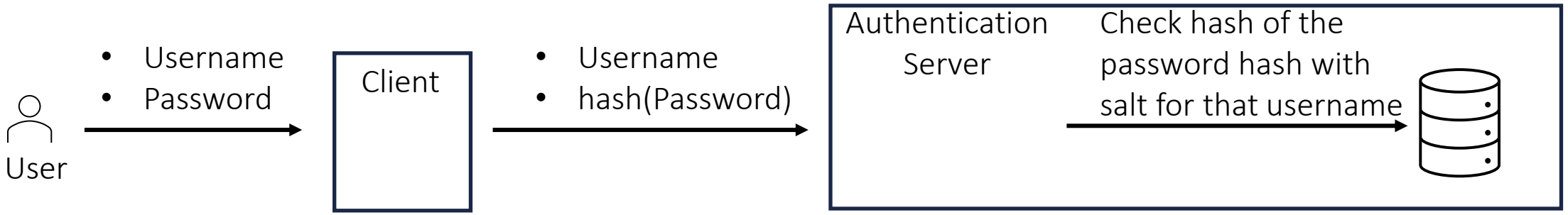
Credentials management



The Authentication Server should store the password hashed with salt (a random value).

This to make brute force and rainbow tables attacks harder.

Credentials management

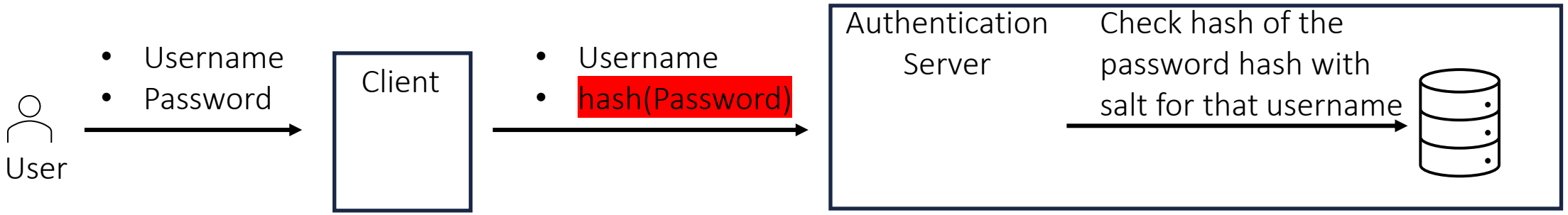


Storage
- Username
- Salt
- hash(hash(Password)+Salt)

The Authentication Server should store the password hashed with salt (a random value).

This to make brute force and rainbow tables attacks harder.

Credentials management

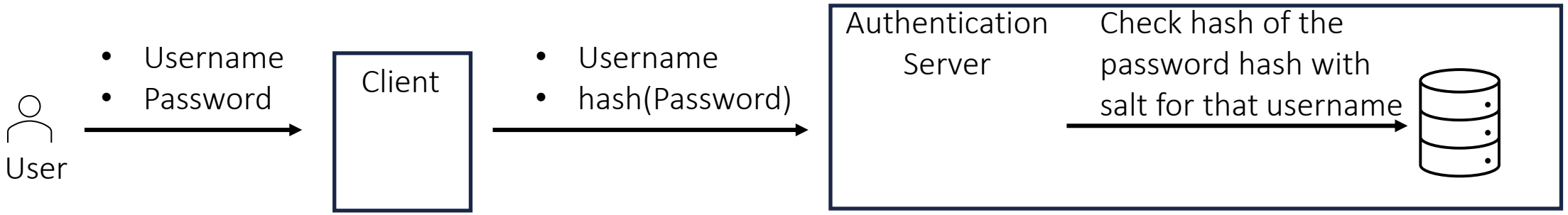


Storage
- Username
- Salt
- hash(hash(Password)+Salt)

Authentication server:

1. Retrieves Salt and hash(hash(Password)+Salt).
2. Applies hash to the received password with Salt.
3. Compare the hashed value with the retrieved one.

Credentials management

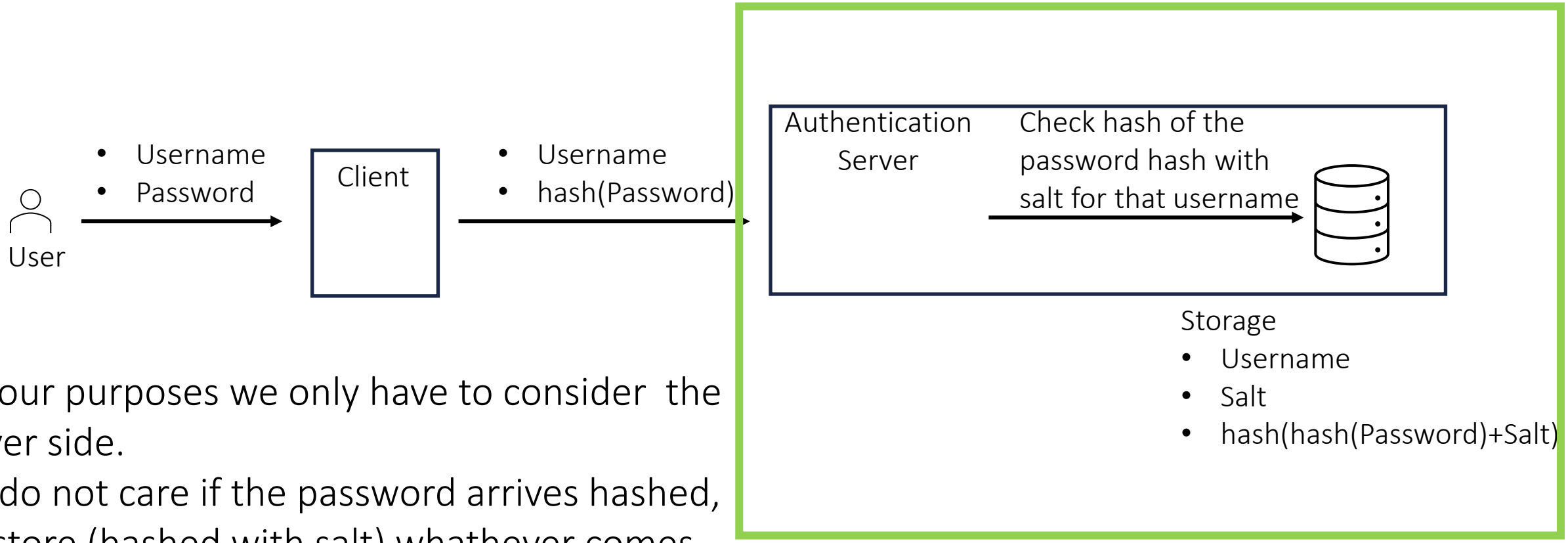


Storage
- Username
- Salt
- hash(hash(Password)+Salt)

In general, services add another authentication channel (Multi-Factor Authentication), such as SMS, email, mobile app notification, etc.

Password should be changed periodically.

Credentials management



For our purposes we only have to consider the server side.

We do not care if the password arrives hashed, we store (hashed with salt) whathever comes.

Cryptographic hash functions

Cryptographic hash functions (CHFs) have the same properties of hash functions plus:

- Pre-image Resistance: It's computationally infeasible to derive the original input from the hash.
- Second Pre-image Resistance: it's hard to find a different input that results in the same hash as a given input.
- Collision Resistance: It's hard to find two different inputs that produce the same hash.
- Avalanche Effect: A small change in input results in a significantly different hash.

You can use libraries that hash data for you with secure algorithms.

Authentication

who are you ?

401 Unauthorized

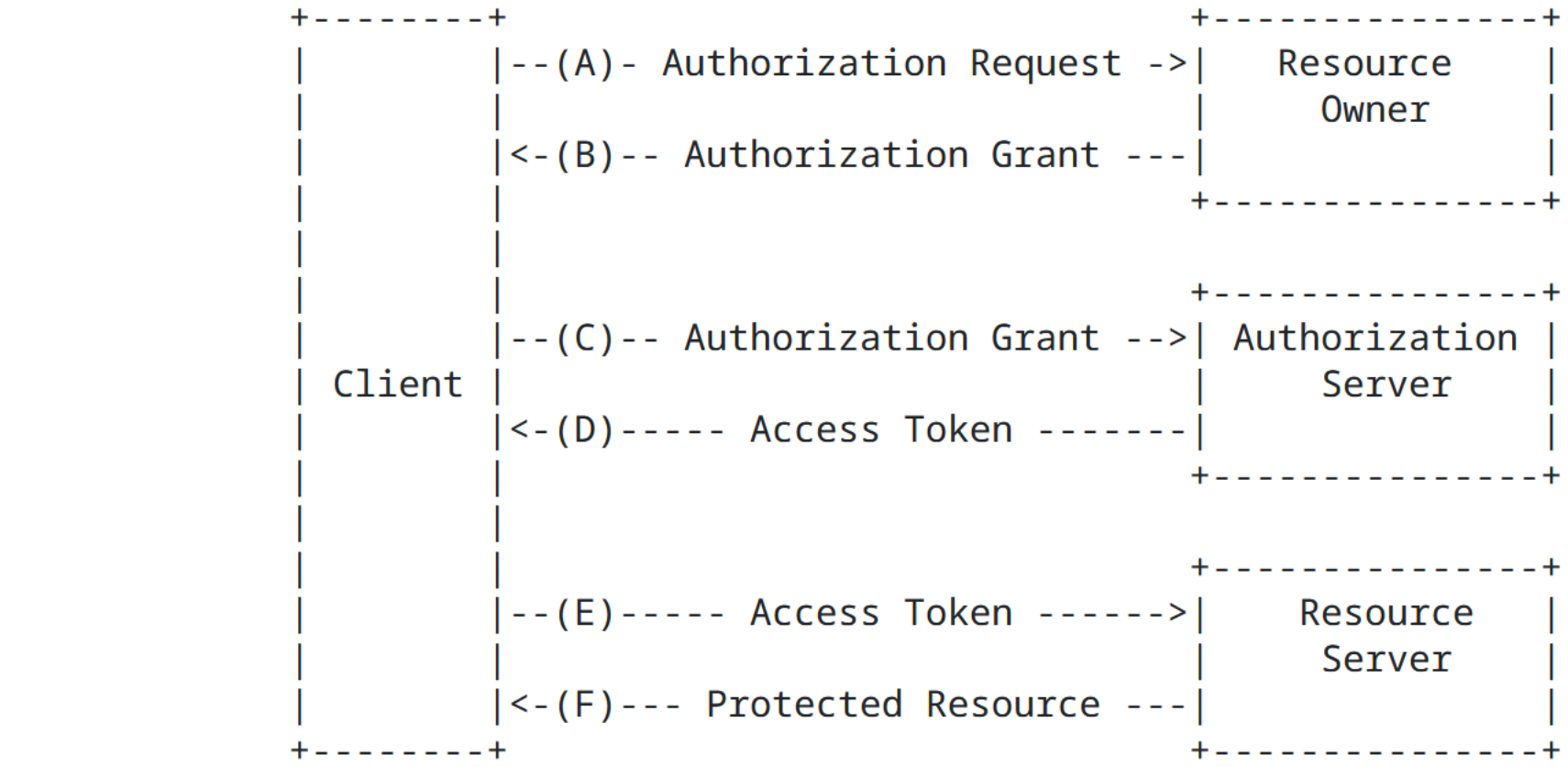
Authorization

what you can do?

403 Forbidden

Authorization – Oauth2.0

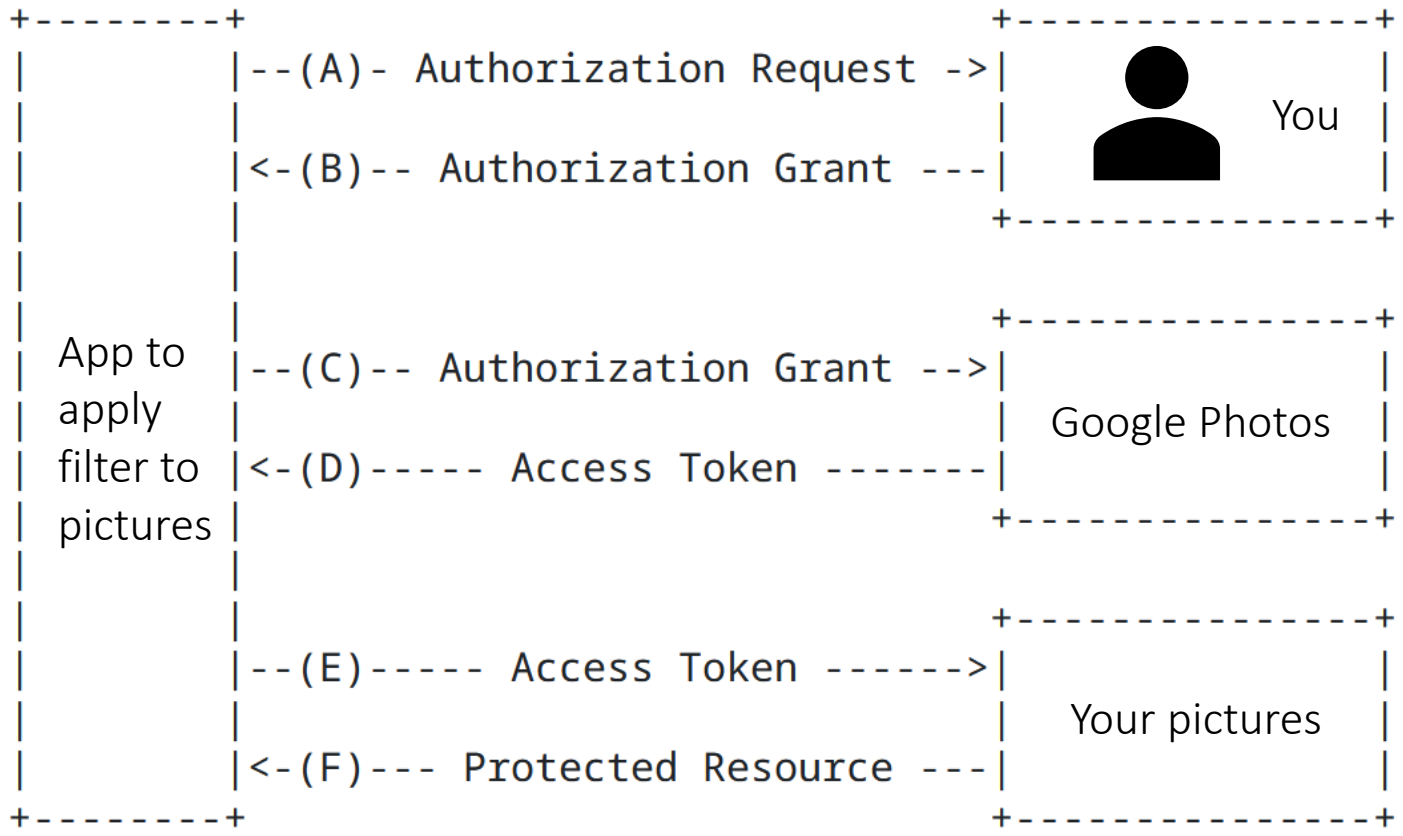
Oauth2.0 is standard framework to enable a third-party application to obtain limited access to an HTTP service on behalf of a Resource Owner.



Specs:
RFC 6749
RFC 6750
RFC 7662
(and more)

Authorization – Oauth2.0 (example)

Imagine using a service to apply filters to your pictures stored on Google Photos.



Authorization – Oauth2.0

A grant is a credential representing the resource owner's authorization (e.g., username and password login, session cookies, id token).

The specification is focused on the authorization (protocol, grant types flows etc.) but do not specify:

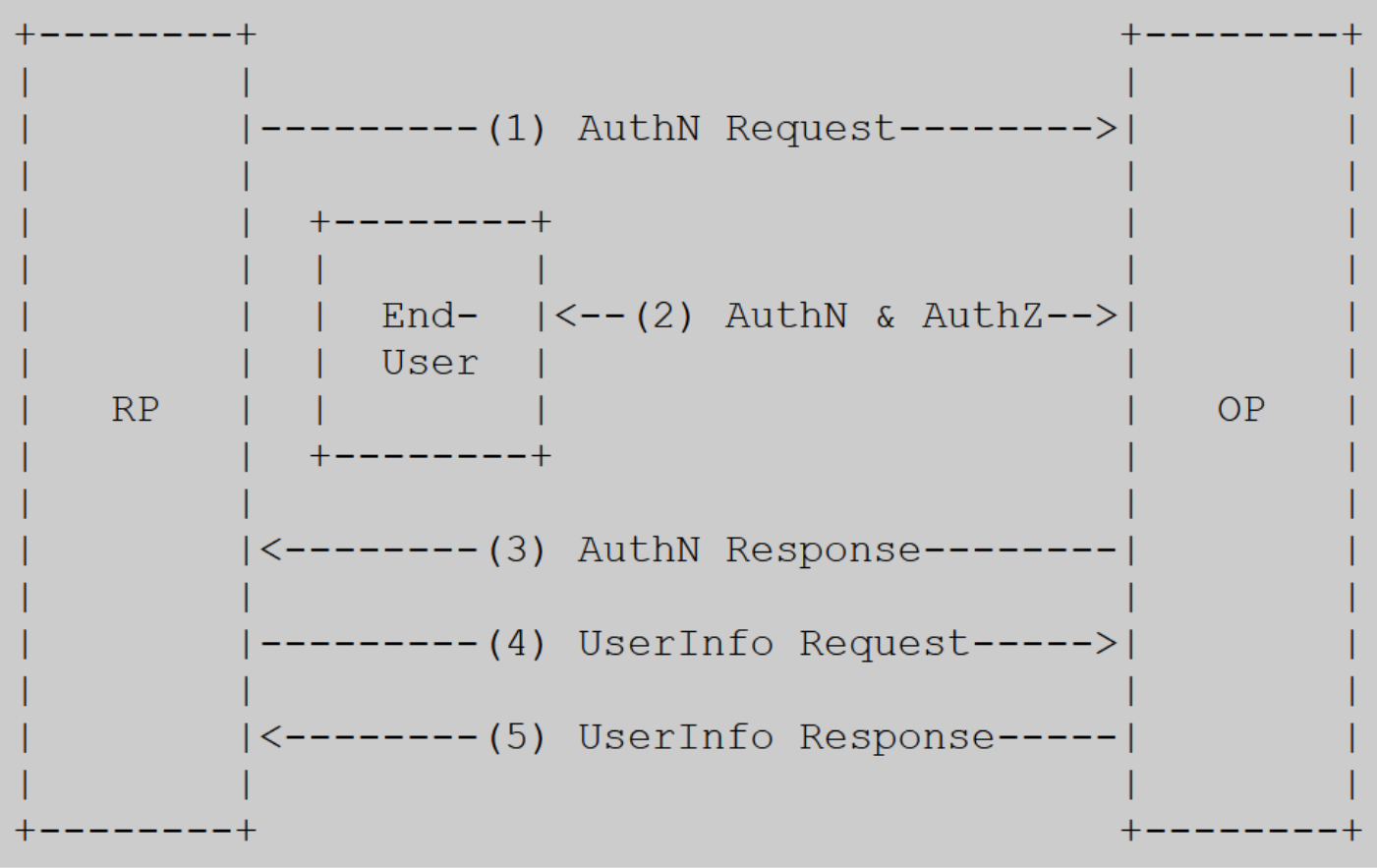
- How the resource owner is authenticated.
- The format of the tokens.
- How the Resource server validate the Access token.

Authentication – OpenID Connect

OpenID Connect (OIDC) is an identity layer built on top of OAuth 2.0.

- It provides authentication and user identity information in a standardized way.
- It allows users to log in to multiple applications with a single set of credentials, improving security and user experience.
- The primary extension that OpenID Connect makes to OAuth 2.0 is to enable End-Users to be Authenticated with the ID Token data structure.

OpenID Connect - Protocol



1. The RP (Relying Party, e.g. client) sends a request to the OpenID Provider (OP, e.g. authentication server).
2. The OP authenticates the End-User and obtains authorization (e.g. using the web interface of OP).
3. The OP responds to the RP with an ID Token and usually an Access Token.
4. The RP can send a request with the Access Token to the UserInfo Endpoint.
5. The UserInfo Endpoint returns Claims about the End-User.

https://openid.net/specs/openid-connect-core-1_0.html

Tokens

We have two type of tokens:

- ID Tokens, for authentication and access to user info.

  Stored and used client-side.

- Access Tokens, for authorization.

  Sent in every request after the login.

  HTTP Header `Authorization: Bearer <access_token>`

Tokens

Tokens should be short-lived to avoid impersonation.

- Both Oauth2.0 and OIDC have specifications to refresh the tokens.

Tokens format:

- OIDC specification represent tokens with JSON Web Tokens (JWT).
- Recently, it is a widespread good practice to use JWT also for authorization.

JSON Web Tokens

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

The representation consists in three Base64-URL-encoded strings separated by dots:

- Header
- Payload
- Signature (hash of encoded header and payload plus a private key)

Encoding(Header.Payload.Signature)

JSON Web Tokens

The header typically is divided in:

- alg: signature algorithm (HS256, RSA, etc.)
- typ: type of the token (the string “JWT”)

The payload contains claims of three types:

- Registered: predefined claims such as **iss** (issuer), **exp** (expiration time), **sub** (subject), **aud** (audience), **scope**.
- Public: defined at will.
- Private: custom created to share information.

JSON Web Tokens

Be careful!

A JWT, though protected against tampering, is readable by anyone.

Do not put secret information in the payload or header elements of a JWT unless it is encrypted.

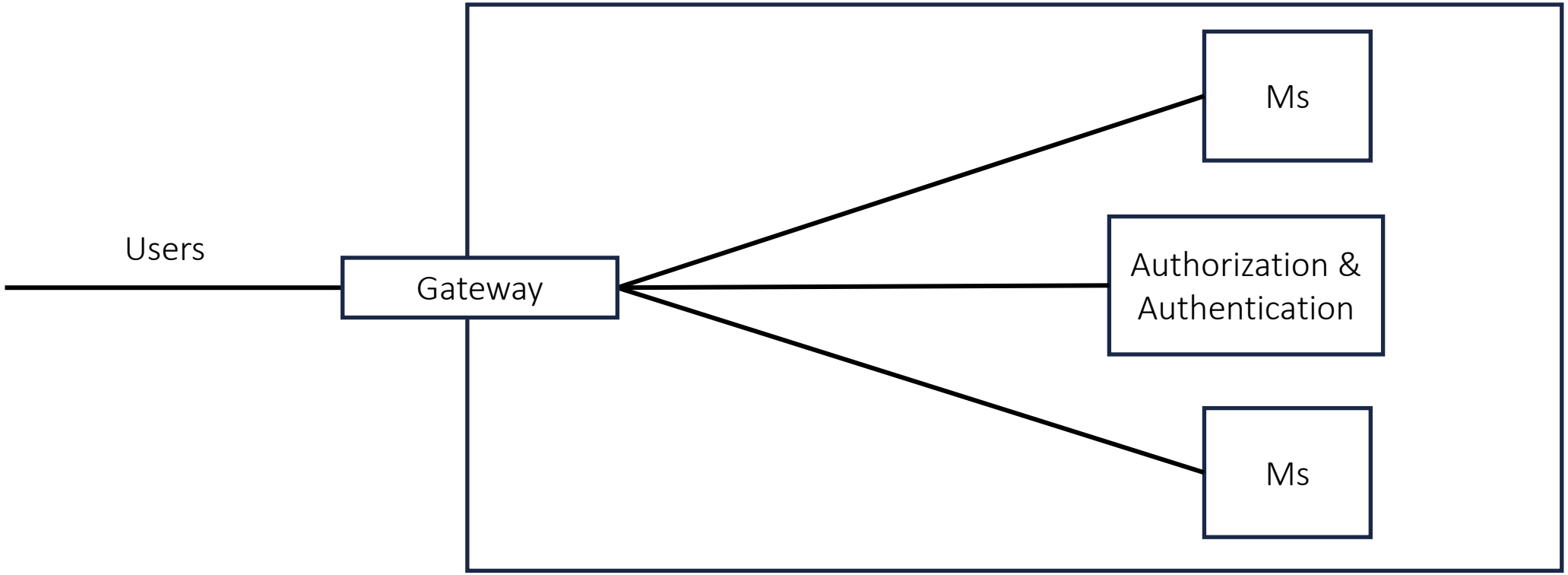
Header:

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

Payload:

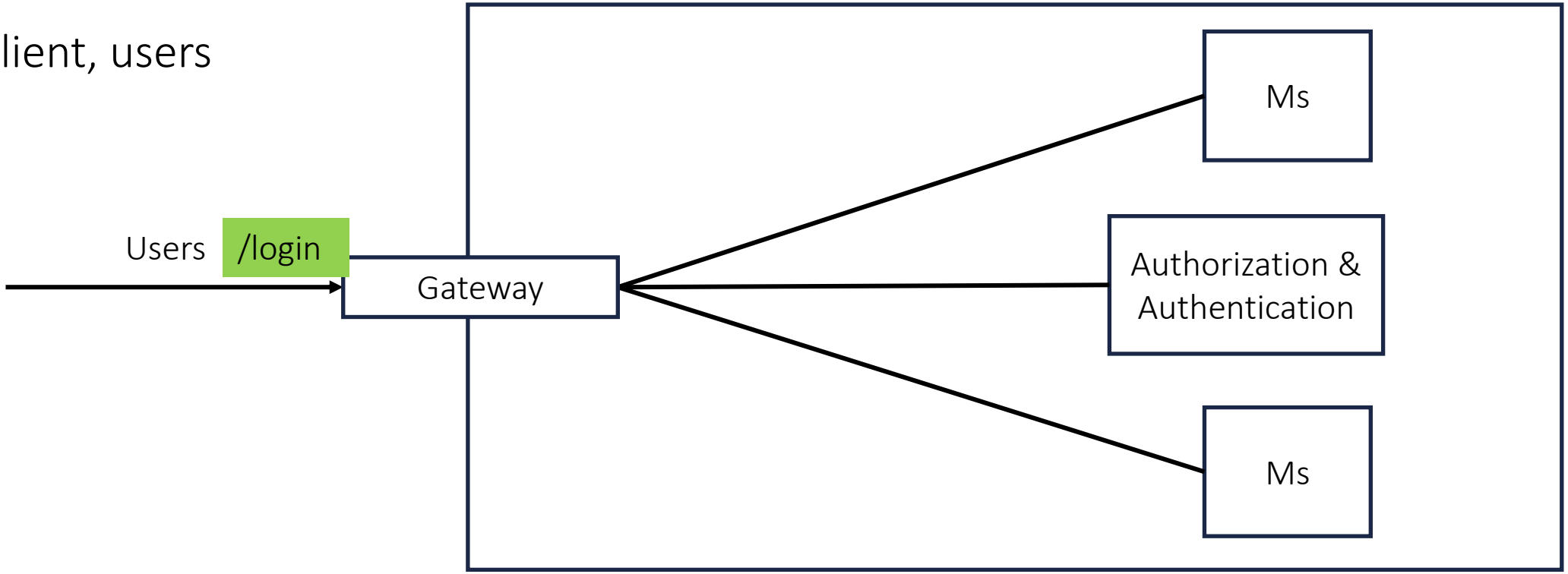
```
{
  "scope": "calc, string",
  "iss": "bjhIRjM1cXpaa21zdWtISnp6ejlMbk44bTlNZjk3dXE=",
  "sub": "user_id_YzEzMGdoMHJnOHBiOG1ibDhyNTA=",
  "aud": "https://localhost:5000/oauth2/authorize",
  "jti": "1516239022",
  "exp": "2021-05-17T07:09:48.000+0545"
}
```

What about microservices?



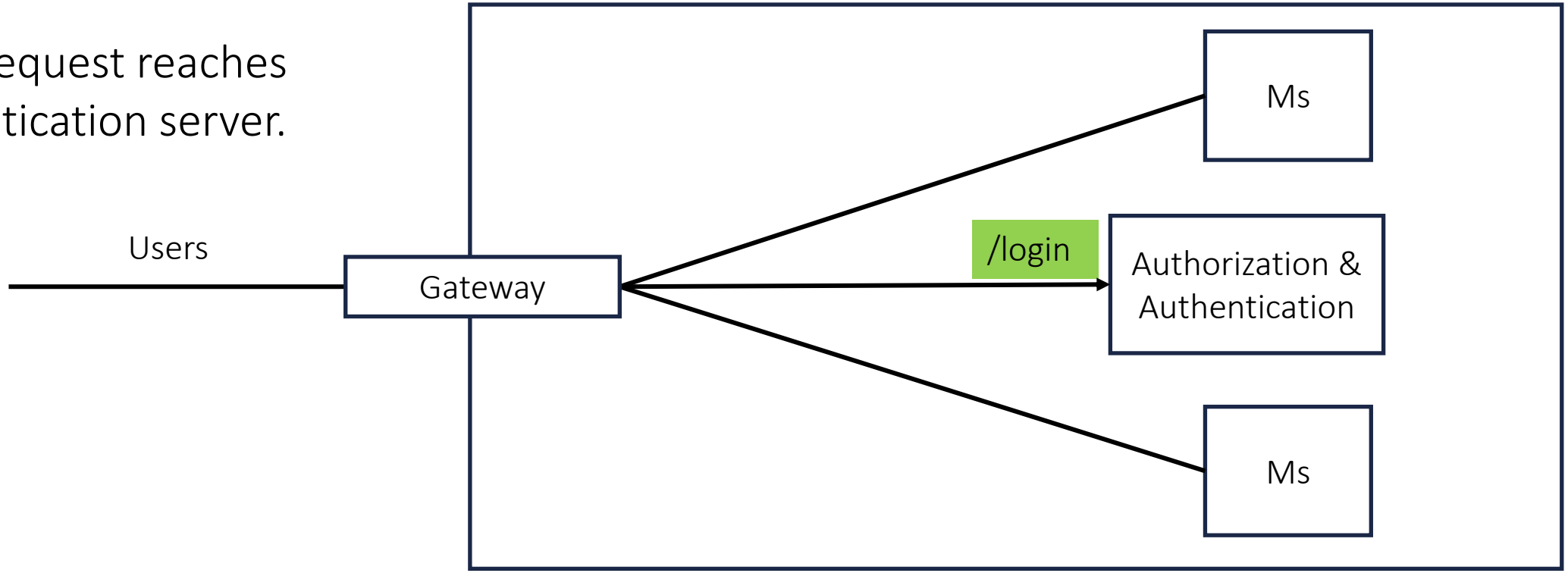
What about microservices?

Using the client, users can login.



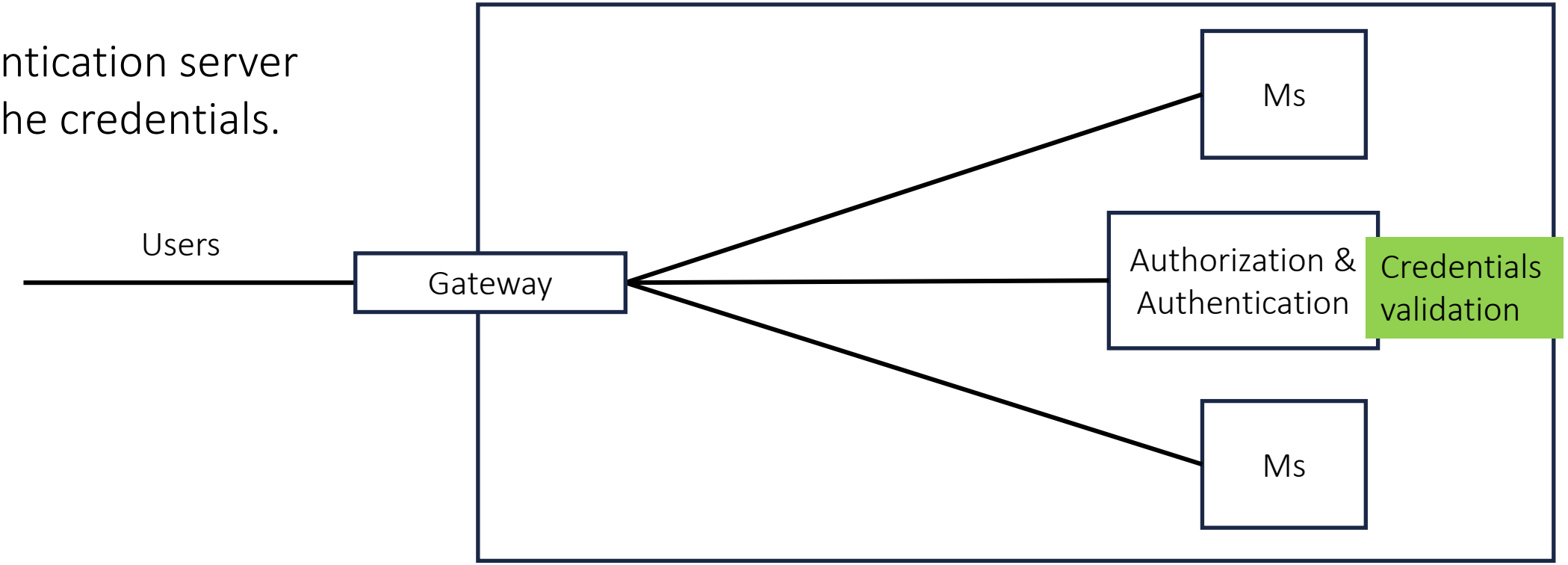
What about microservices?

The login request reaches the authentication server.



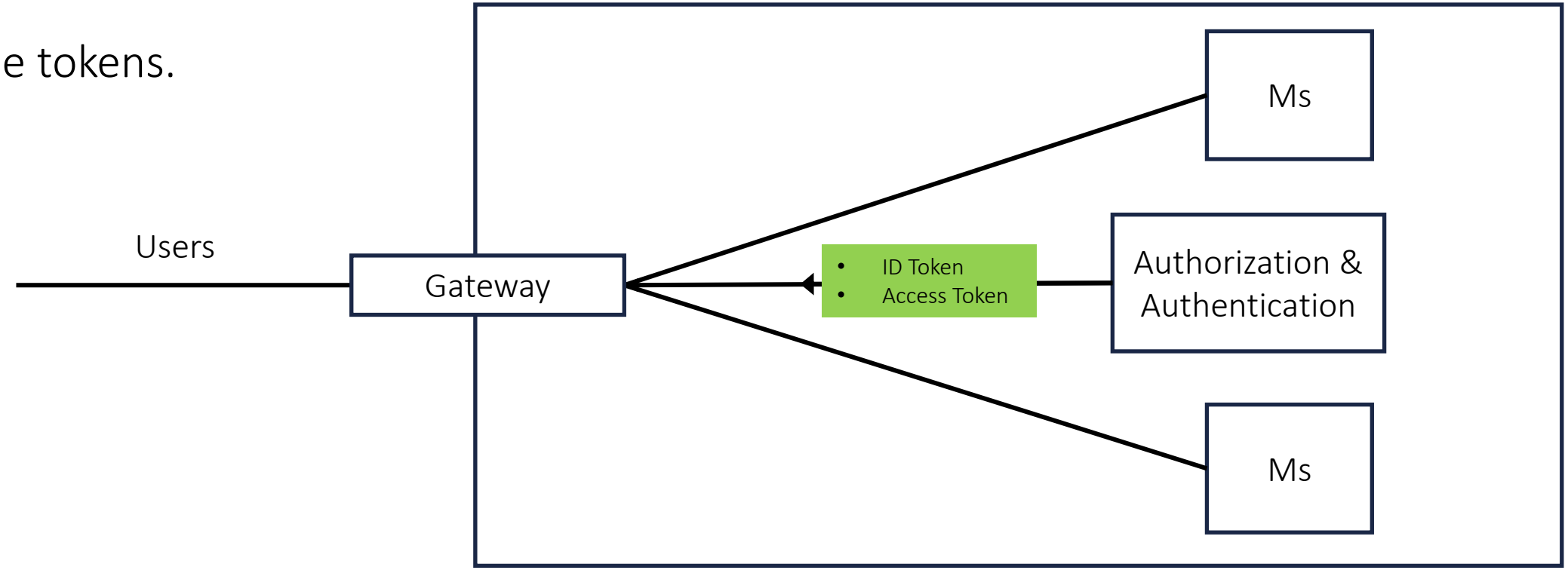
What about microservices?

The authentication server validates the credentials.



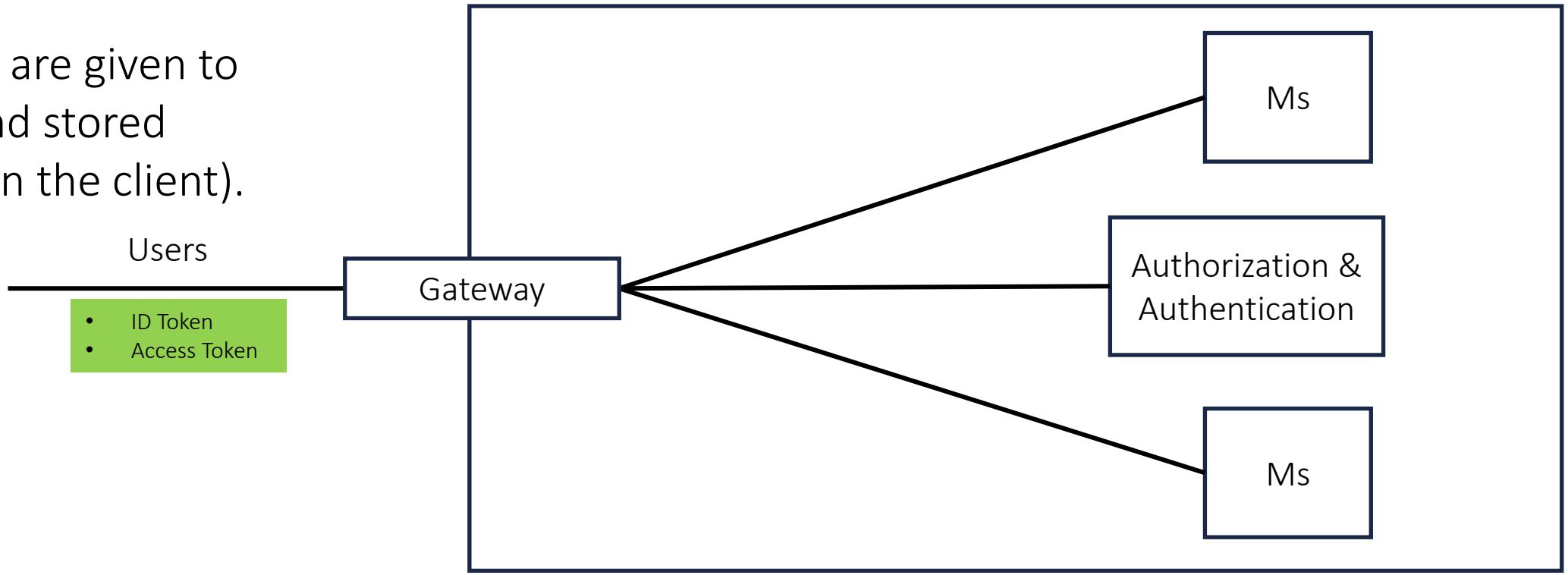
What about microservices?

It issues the tokens.



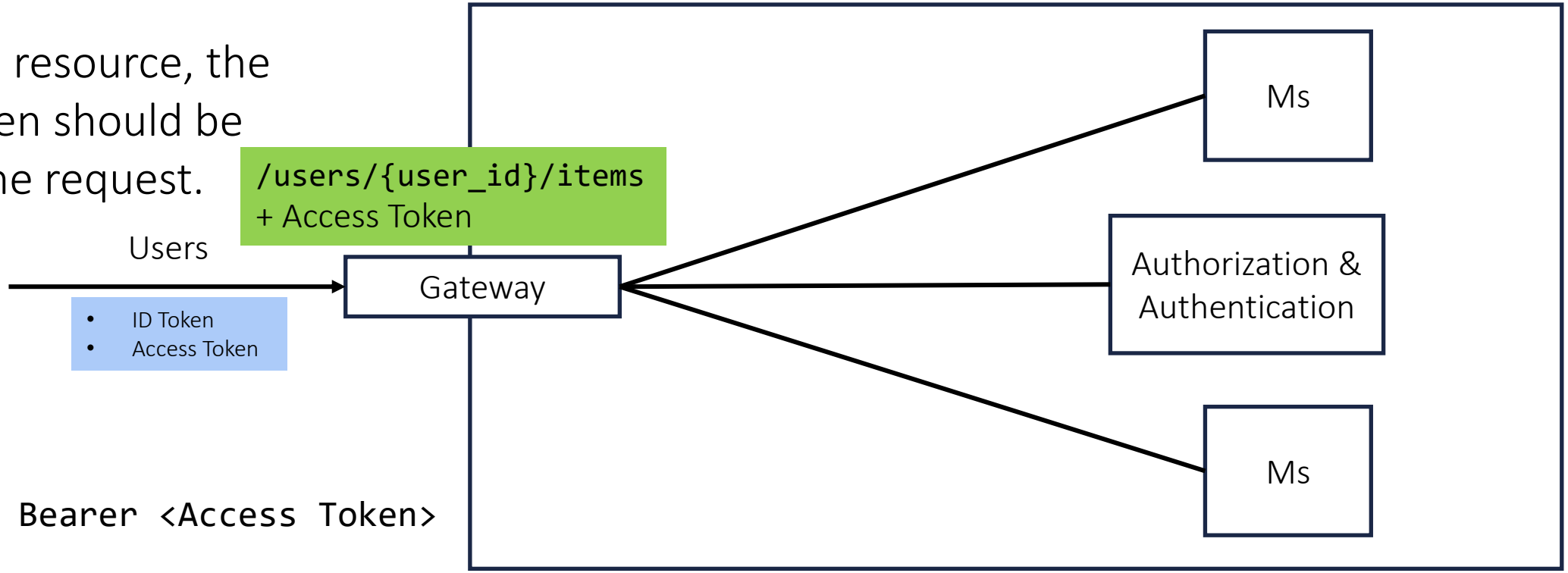
What about microservices?

The tokens are given to the user and stored (generally in the client).



What about microservices?

To access a resource, the Access Token should be added in the request.

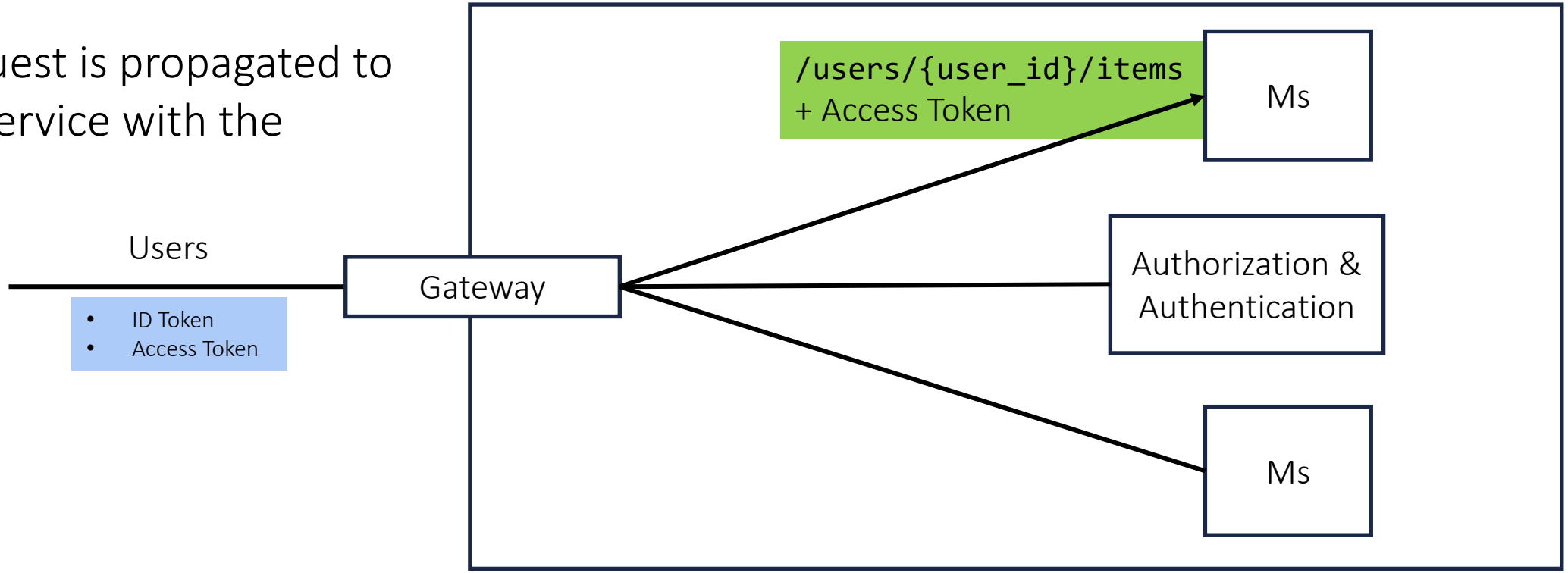


HTTP Header:

```
Authorization: Bearer <Access Token>
```

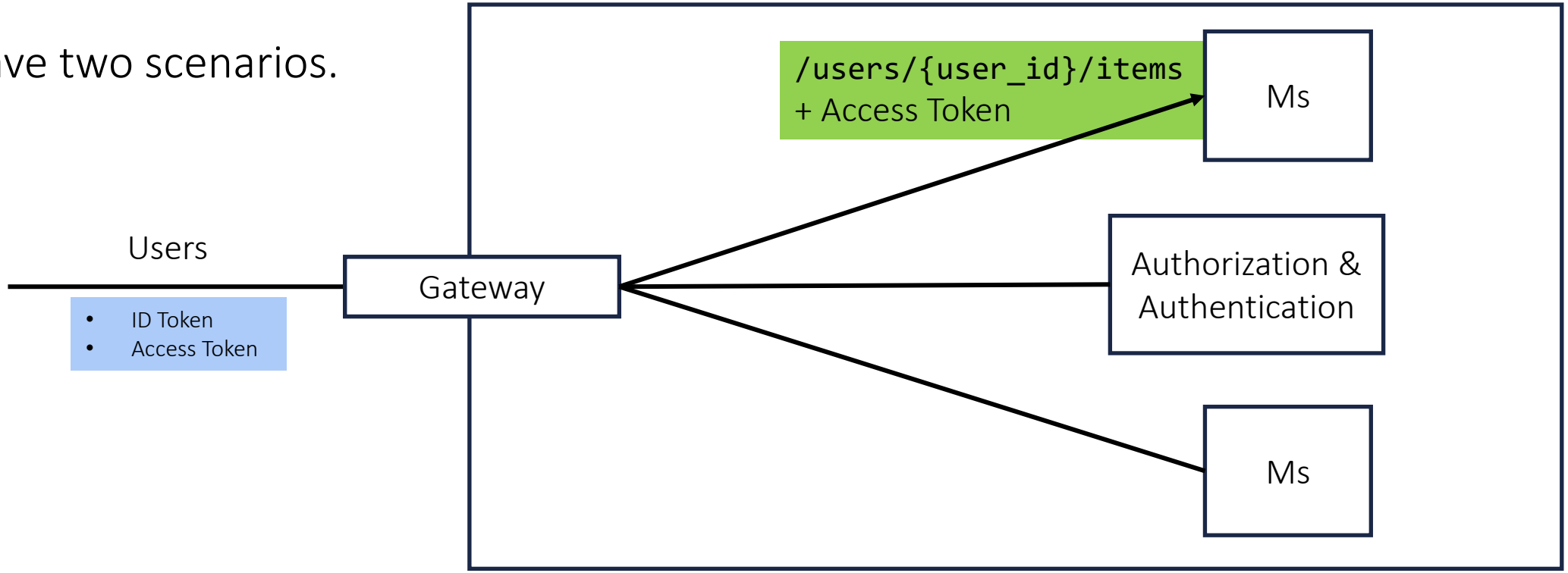
What about microservices?

To the request is propagated to the microservice with the resource.



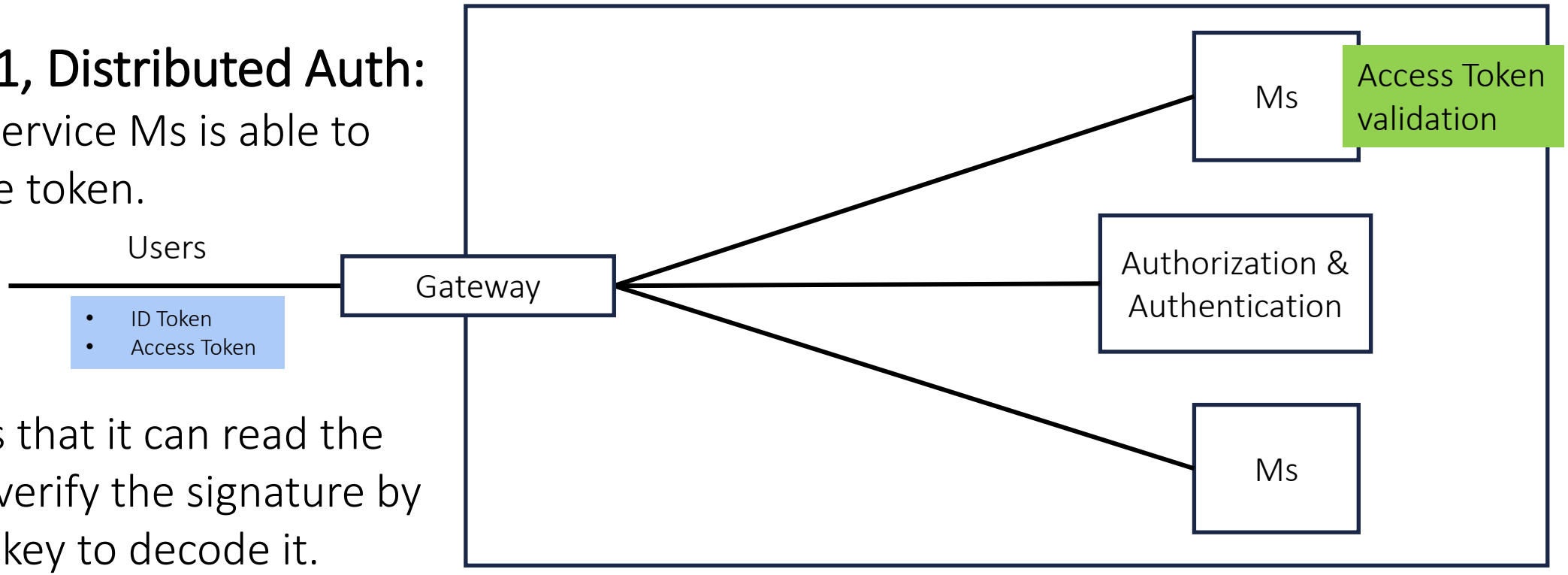
What about microservices?

Now we have two scenarios.



What about microservices?

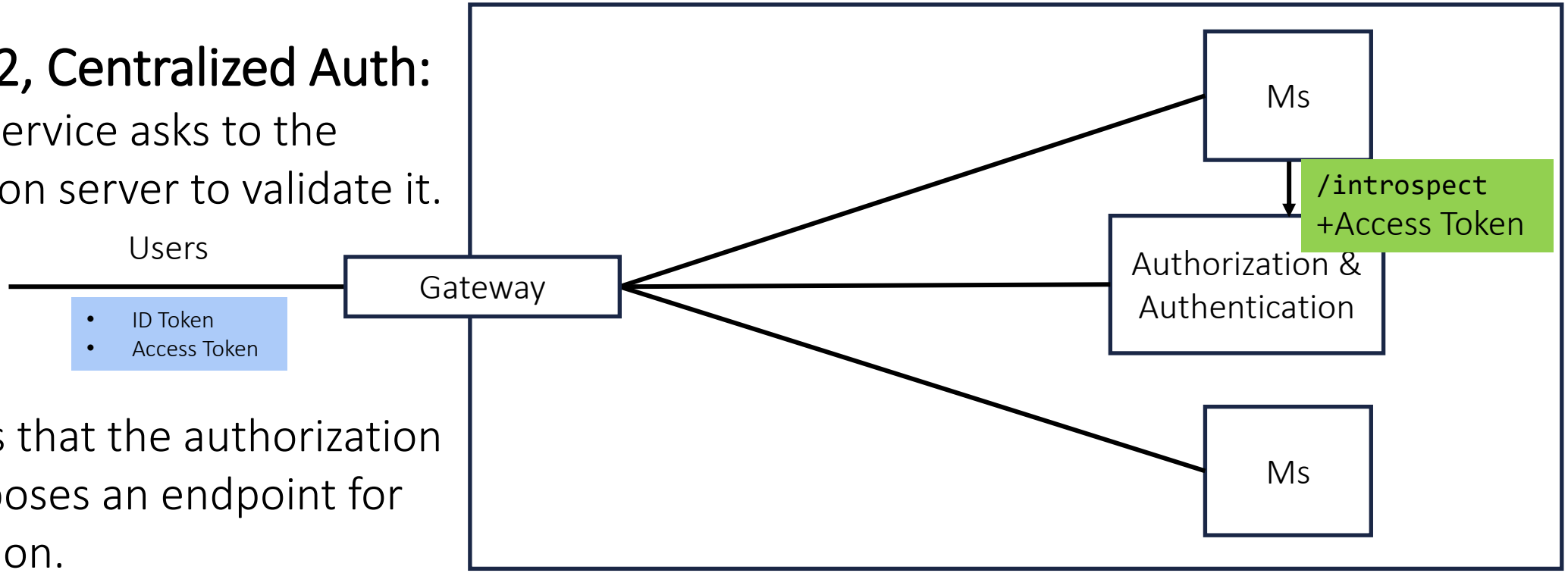
Scenario 1, Distributed Auth:
The microservice Ms is able to validate the token.



This means that it can read the token and verify the signature by having the key to decode it.

What about microservices?

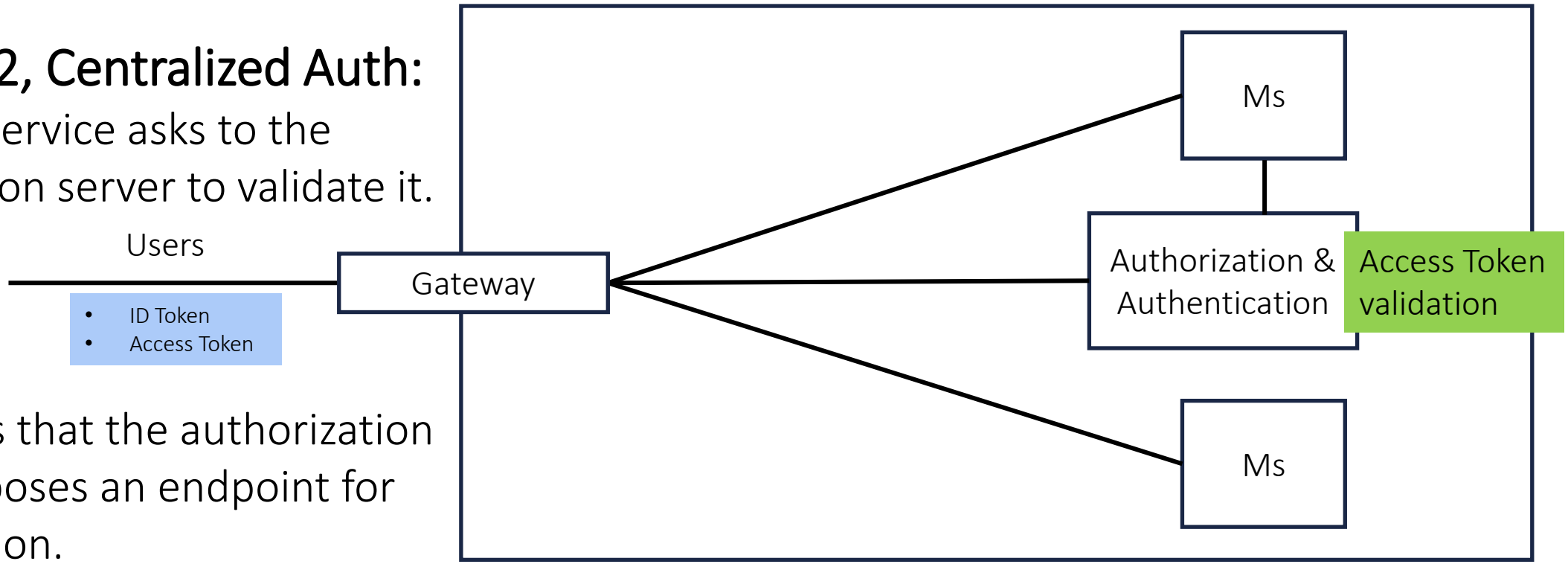
Scenario 2, Centralized Auth:
The microservice asks to the authorization server to validate it.



This means that the authorization service exposes an endpoint for the validation.

What about microservices?

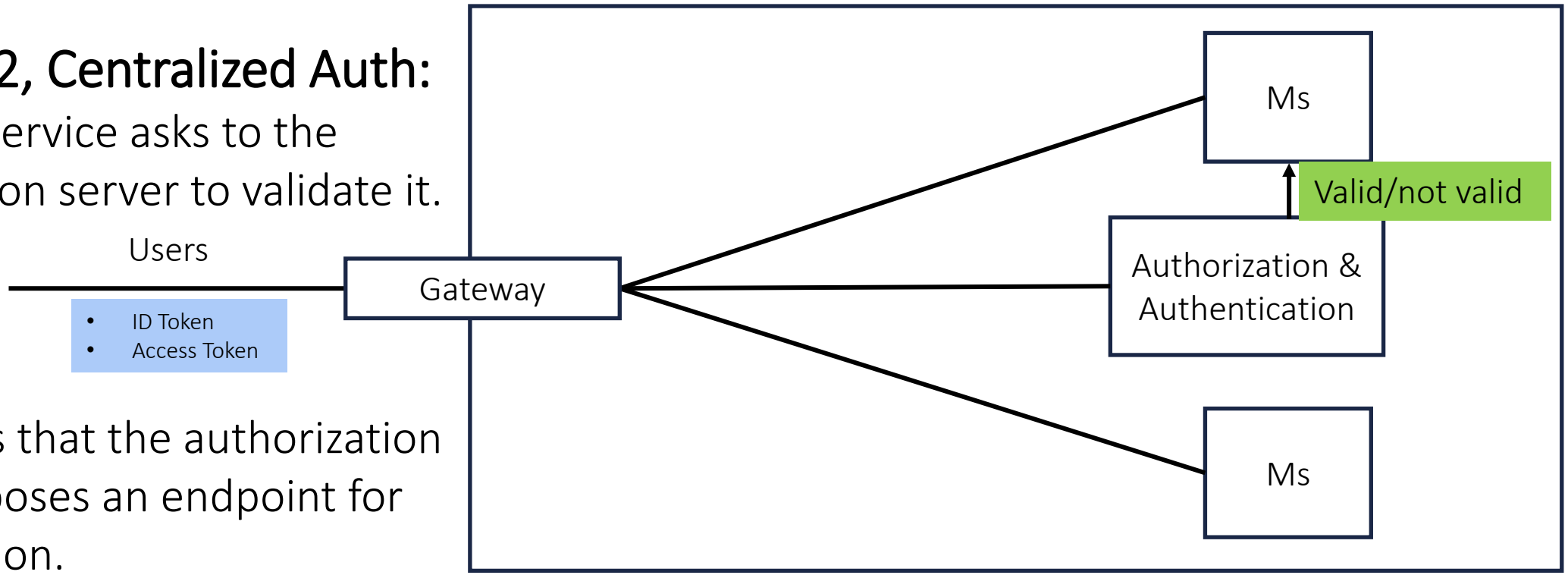
Scenario 2, Centralized Auth:
The microservice asks to the authorization server to validate it.



This means that the authorization service exposes an endpoint for the validation.

What about microservices?

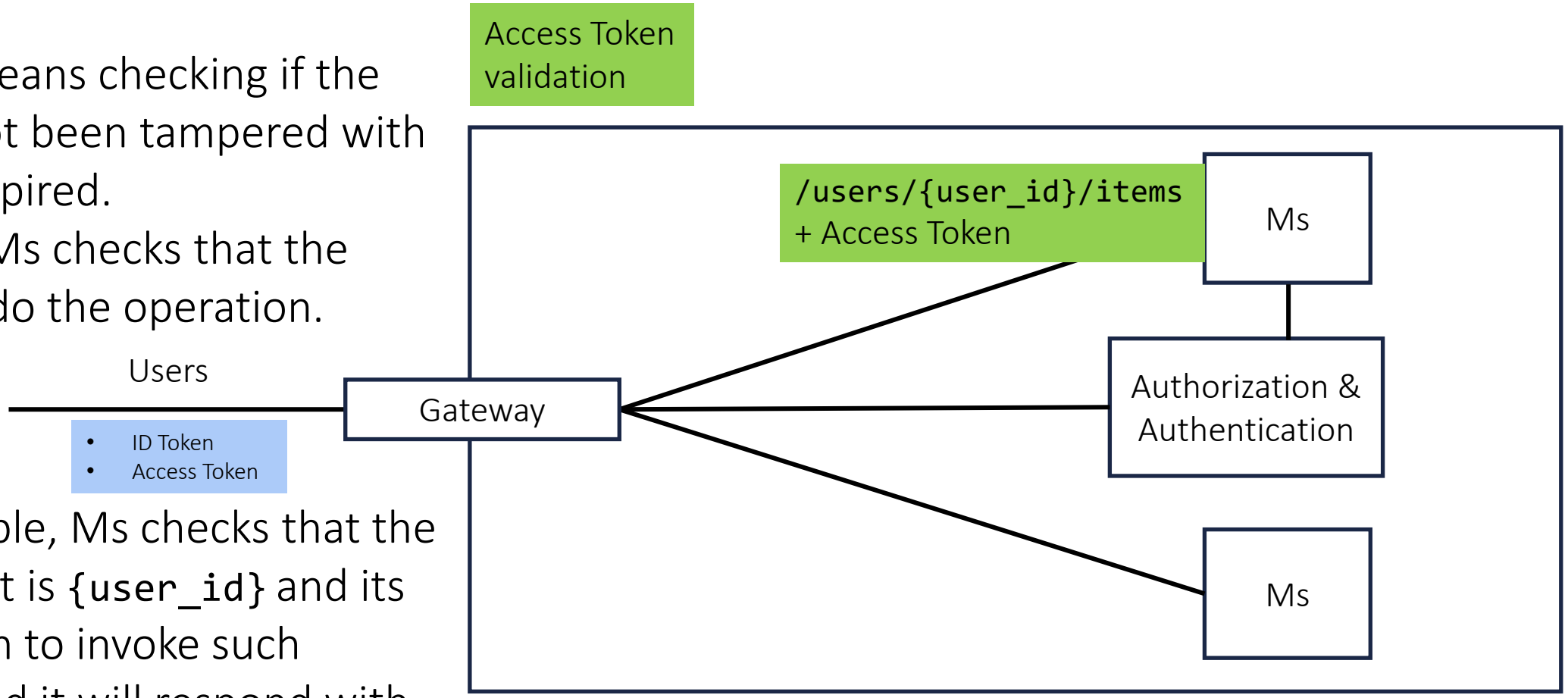
Scenario 2, Centralized Auth:
The microservice asks to the authorization server to validate it.



This means that the authorization service exposes an endpoint for the validation.

What about microservices?

Validation means checking if the token has not been tampered with and is not expired.
If it is valid, Ms checks that the subject can do the operation.



In this example, Ms checks that the token subject is `{user_id}` and its authorization to invoke such endpoint (and it will respond with the items of the user).

Some general considerations

- A possible solution is to perform validation of tokens in the gateway. This can simplify the interactions among microservices but it reduces the security of the architecture (‘trust the network’).
- It is possible to separate authentication and authorization in two microservices. The login flow changes but the other parts are similar.
- Scenario 1 consists in **distributed authorization**, it reduces the burden on the authorization service but the key to validate tokens should be distributed.
- Scenario 2 is the opposite, it **centralize** the **authorization** with the risk of slowing down the performance if the authorization server is overloaded (it can be scaled but it is not trivial).

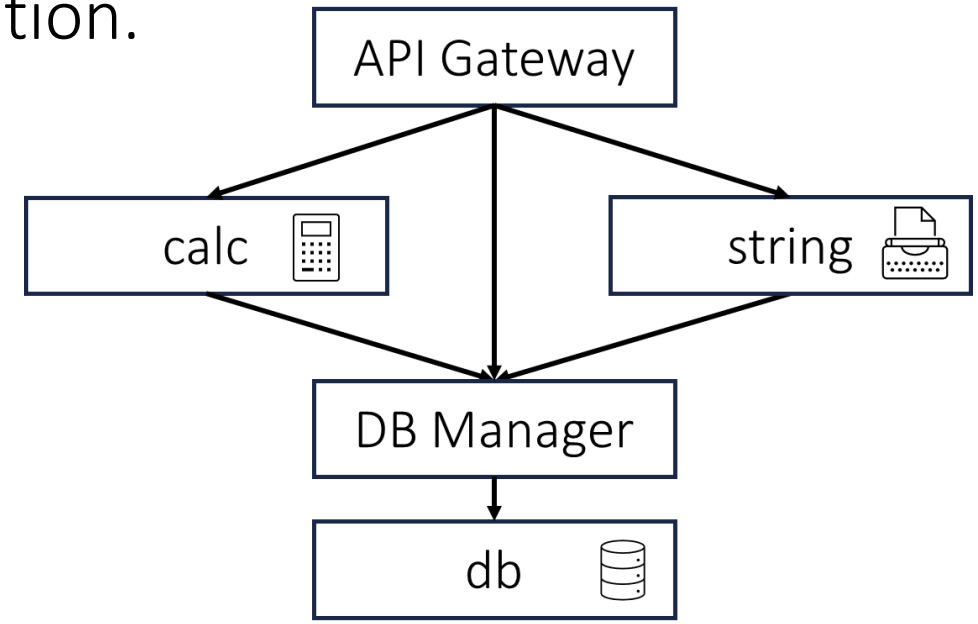
Disclaimer

- To simplify the things, we do not implement OIDC.
- Considering that we do not develop a client, the ID Token is never really used by us.
- For these reasons, for this lab and for the project, the authentication should be done with the schema shown before to obtain only the Access Token that must be used following OAuth 2.0 specifications.

Today's Lab

Work on your project code or use **`microase`** (your own or from Moodle).

- Choose the authorization policies for the endpoints (details in next slide).
- Choose the authorization scenario (1- distributed vs 2- centralized).
- Create a microservice that acts as authentication and authorization server.
- [optional] Refresh the token after expiration.



Authorization policies

Which user can do what. Example in `microase`:

- Role `Math_user`: can do `calc` operations.
- Role `Text_user`: can do `string` operations.
- Role `All_user`: can do every operations.

The User record in the DB is: `<ID, Role, Username, Password, Salt>`

Note: In the Project, we do not have different roles.

Authorization policies – Full flow

1. The user is created (i.e. by registration) and the corrisponding record is created in the DB.
2. When a user performs a login, the credentials must be checked against the corresponding record in the DB.
3. If the credentials match, the Access Token is generated and sent in the response.
4. In the Access Tokens scope should be present which microservice can access the user and which is the subject.
5. When the user calls an endpoint, the Access Token must be in the request to check if the operation can be performed by that user.

For example, a user with role '`Math_user`' can only call the API related to the `calc` microservice.

User registration

For our purposes, it is enough to expose an endpoint which:

1. Receives the user information.
2. Validates the inputs (e.g. password length, email format, etc.).
3. Checks for uniqueness (e.g. avoid the same username or email).
4. Assigns a role to the user and stores the information in the database.
5. Answers with success and the user ID (or with failure if something goes wrong).

Auth server

The auth (authentication and authorization) server is connected with the DB containing user credentials.
If it's the only microservice using the DB, it acts as DB Manager.

It is not mandatory to develop it by yourself!
You can use a third party (dockerizable) service, for example Keycloak.
The only requirements is that it should work with `docker compose up` of the architecture.

I will now give you insights if you want to implement it by yourserlf.

Auth server – OAuth2.0

We use the **Resource Owner Password Credentials Grant** flow (Sec. 4.3 of RFC 6749).

In other cases, when your client is not trusted (e.g. web page), you need to use a different flow to reduce the use of credentials.

Auth server – Endpoints

For our purposes, we need at least the following endpoints:

- **Token Endpoint**: Exchanges the credentials for tokens. It returns the access token and optionally a refresh token. Requests and response for it are specified in RFC 6749 4.2.
- **Introspect Endpoint:** Used in centralised authorization to validate the token. When a microservice receives a request, it calls this endpoint to ask the token validation. It is specified in RFC 7662.

User login flow

A user that is already registered must perform the login before doing any operation.

1. It calls the **login** endpoint of the gateway sending the credentials as payload. The request is redirected to the **Token** endpoint of the auth server.
2. The auth server validates the credentials and sends the access token.
3. When invoking other endpoints, the Access token must be issued with the request for its validation (distributed or centralized).
4. If the validation is centralized, the ‘internal’ microservices invoke the **Introspect** endpoint of the auth server with the Access Token.

Auth server – Access Token creation

Example using `pyjwt`:

```python
import jwt
import datetime

# Define the private key for signing the token
private_key = ... # Load the private key from a file or other source
# Define the token payload with standard OAuth claims
payload = {
  "iss": "https://your-auth-server",          # Issuer: your auth server's URL
  "sub": "user_id_123",                       # Subject: the user's unique ID
  "aud": "https://gateway",                   # Audience: the API/service you're securing
  "iat": datetime.datetime.now(datetime.UTC), # Issued at: current time
  "exp": datetime.datetime.now(datetime.UTC) + datetime.timedelta(hours=1),  # Expiration time
  "scope": "calc",                            # Scopes: permissions granted to this token
  "client_id": "your-client-id",              # Optional: ID of the client app
  "jti": "unique_token_id_123",               # JWT ID: unique identifier for the token
}
# Generate the access token
access_token = jwt.encode(payload, private_key, algorithm="RS256")
```

`pyjwt` can generate automatically the header and the signature.
The optional parts depends on your implementation choices.

Token management

The auth server needs to:

- **Create tokens** (login).
- (centralized scenario) **Validate tokens**.
- (optional) **Refresh tokens**.

To validate tokens, `jwt` library has ha `decode` function and than you have to check the signature, the subject, the scope, etc.

What about token expiration?

Considering that refreshing the tokens is optional, how to manage the expiration without it is not trivial.

- We can ask the user to login again, but what happens during a match?
- What happens if a user logs in twice? Issue a new token? We use the original one?
- What happens if the user logs out? The token is revoked? It is valid until expiration?

The answer is… it is all part of your design choice ☺

Token in the HTTP Header

GET with Access Token in the Header:

```
headers = {"Authorization": f"Bearer {access_token}"}
response = requests.get(f"https://localhost:5000/user/{user_id}/items", headers=headers)
```

Obtain the Access Token in a route (Flask):

```
@app.route('/users/user_id>/items', methods=['GET'])
def get_user_items(user_id):
    # Extract the Authorization header
    auth_header = request.headers.get('Authorization')
    if not auth_header:
        return jsonify({"error": "Missing Authorization header"}), 401
    acces_token = auth_header.removeprefix("Bearer ").strip()
```

Now you have all the ingredients

1. Decide your authorization policies.
2. Decide the Authorization Scenario and implement it.
3. Develop the auth service (or use an existing one).
   - Registration endpoint and user credentials database.
   - Correct user credentials management.
   - Access Token endpoint (+ Introspection if needed).
   - Correct JWT management and expiration scenarios.

Project: how to test with tokens?

You have to register (at least) a user with test credentials.
The simplest way is to create a specific Postman request for registration which, in theory, should be deleted after the testing phase.

Every test should use such credentials to login (obtaining the token) and perform the operation under test.

Project: how to test with tokens?

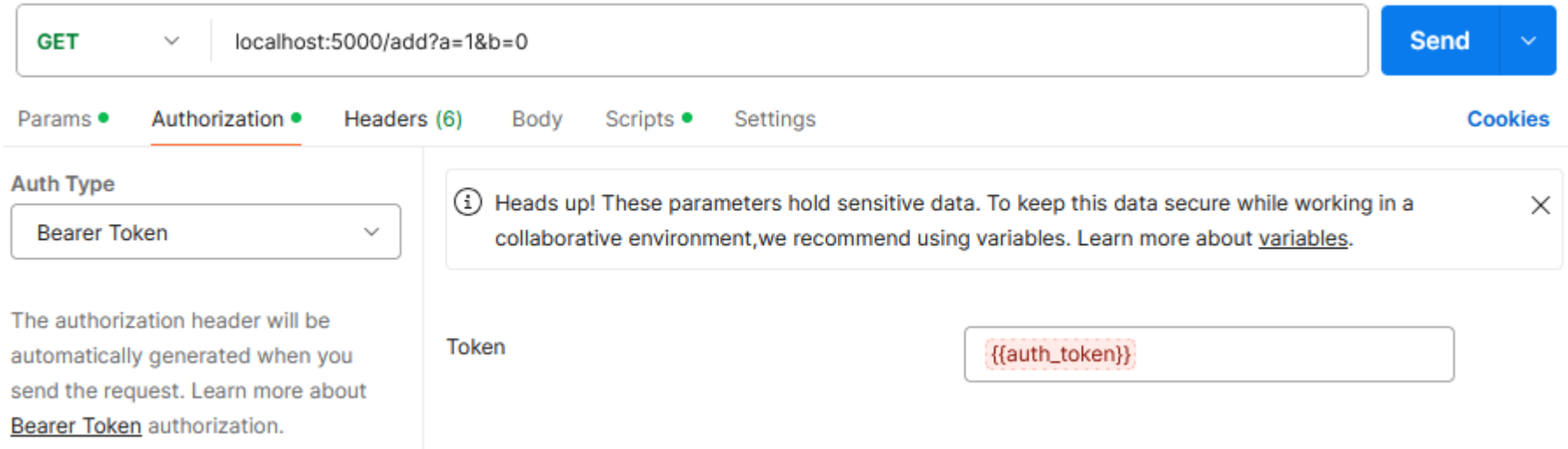
With Postman:

- Create an environment and define a variable (e.g. auth_token) with empty value.
- To obtain the token, use the test credentials to perform a login.
- In the script post-response of the login request write:

```
pm.test("Token received", function () {
    pm.response.to.have.status(200);
    var jsonData = pm.response.json();
    pm.environment.set("auth_token", jsonData.access_token);
});
```

This will save the token in the environment variable.

- Every other request must use the environment variable:



Project: how to test with tokens?

With locust (assuming the test user registered):

- Perform a login in the `on_start()` method that is called when each user is started.
- Save the token and add it to the header of each request.

```
class TokenUser(HttpUser):
    wait_time = between(1, 2)

    def on_start(self):
        # You can also do it only the first time using "if self.token:"
        response = self.client.post("/login",json={"username": "test_user", "password": "test_password"})
        if response.status_code == 200:
            self.token = response.json().get("access_token")
            print("Access token obtained successfully.")
        else:
            self.token = None
            print(f"Failed to obtain token: {response.status_code}, {response.text}")
    @task
    def get_user_items(self):
        if self.token:
            user_id =  # Replace with the actual user_id of testing user (maybe obtained during the login)
            headers = {"Authorization": f"Bearer {self.token}"}
            response = self.client.get(f"/users/{user_id}/items",headers=headers)
```

Project: how to test with tokens?

What about testing of microservice in isolation?

1. If you use centralized authorization:
   - The token validation is an external dependency.
   - You have to mock it.

2. If you use distributed authorization:
   - Generate a mock token that is valid, or
   - Create mock code that skip the token validation. This will not test the authorization part in isolation but could simplify the things for you.

Lab take away

- How to manage user credentials.
- Authentication and Authorization with microservices.
- We scratched the surface, exist several security mechanism and protocols to enhance authentication and authorization.

Project take away

- ❑ Manage the credentials securely.
- ❑ Use authorization with Oauth2.0.
- ❑ Use JSON Web Tokens.