Spring  
Security

short line

Your Name  
4th September, 20XX

# SHA256

**What SHA-256 Is and Its Output:**

SHA-256 is a hash function that takes an **input of any size** and produces **a fixed-size output**, which is a **256-bit string** of letters and numbers. This output is often referred to as a "fingerprint" or "digest."

**Key Properties of SHA-256 (and Hash Functions in General):**

* **Deterministic Scrambling:** For the same input, SHA-256 will always produce the exact same output. The process scrambles the data in a predictable manner.
* **Quote:** "given the same input it will always produce exactly the same output"
* **Fixed Size Output:** Regardless of the input size (a single letter or the entire Lord of the Rings series), the output will always be 256 bits long.
* **Quote:** "no matter what the size of the input the output will always be of a fixed length"
* **Irreversibility:** Given only the SHA-256 hash (the fingerprint), it should be computationally infeasible to determine the original input. Even a tiny change in the input results in a drastically different output, making reverse engineering extremely difficult.
* **Quote:** "if you only have access to the output the input should be unguessable" and "if just one letter changes in the input the output changes dramatically right it didn't just change by one character it changed like everything changed it seemingly changed in a random and complete way"

**Why SHA-256 is Special:**

1. **256-bit Output Length:** This specific length is a defining characteristic of SHA-256.
2. **Security:** SHA-256 is currently considered secure, meaning no significant exploitable vulnerabilities have been found. Unlike older hash functions like SHA-1 and MD5, no practical methods for easily creating collisions (different inputs producing the same hash) have been discovered.
3. **Speed:** SHA-256 is relatively fast, making it efficient for hashing large amounts of data quickly. However, this speed makes it less suitable for password hashing, where slower functions are preferred to hinder brute-force attacks.

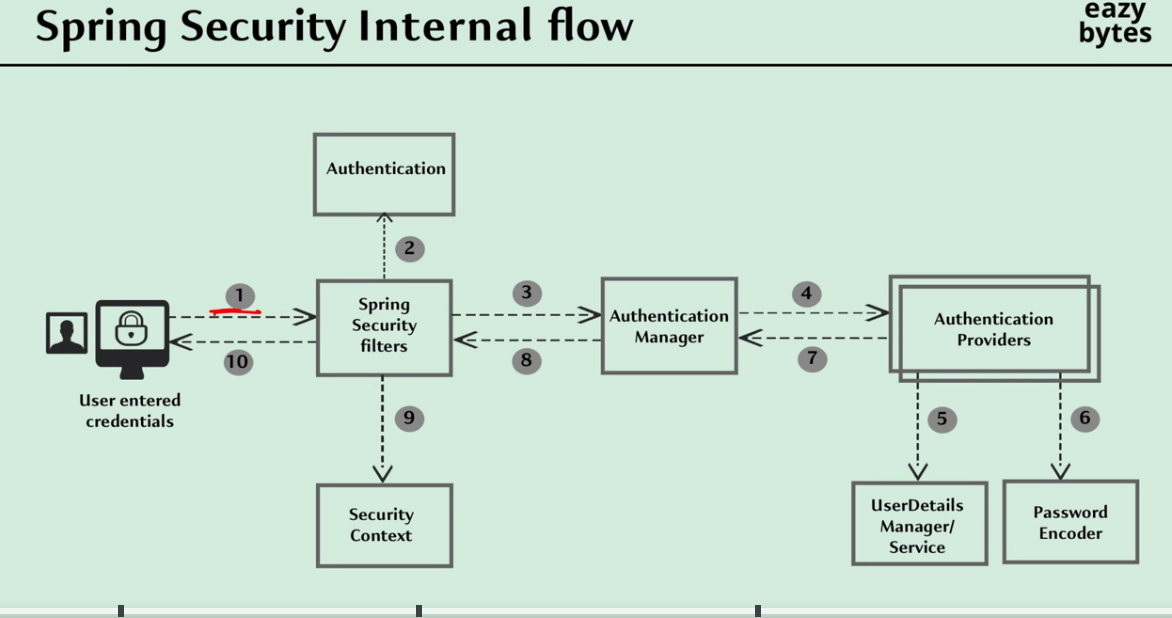
**Real-World Use Cases of SHA-256:**

* **Anti-Virus Software:** When software is published (e.g., word.exe), a SHA-256 fingerprint is often provided alongside it. When a user downloads the software, their computer can calculate the SHA-256 hash of the downloaded file and compare it to the published fingerprint. If they match, it confirms the integrity of the file and assures the user it hasn't been tampered with or infected with malware.
* **Authentication (Auth):** SHA-256 plays a role in authentication processes. Websites often use it to generate authentication tokens. For example, upon successful login, a server might take a user's identifier (like a username) and append a secret key (known only to the server). This combined data is then hashed using SHA-256 to create a unique token. This token is then used to verify the user's identity in subsequent requests without needing to send the password repeatedly. The irreversibility of SHA-256 ensures that even if an attacker intercepts the token, they cannot easily derive the secret key or other sensitive information.
* **Blockchains (e.g., Bitcoin):** Bitcoin utilizes SHA-256 as its hashing algorithm. In a blockchain, each block of transactions contains a hash of its own data, as well as the hash of the previous block. This chaining of hashes ensures the integrity and immutability of the blockchain. If any data in a previous block is altered, its hash will change, which in turn will affect the hash of all subsequent blocks, making tampering easily detectable. The computational cost of generating new blocks makes it extremely difficult for malicious actors to retroactively change transaction history.

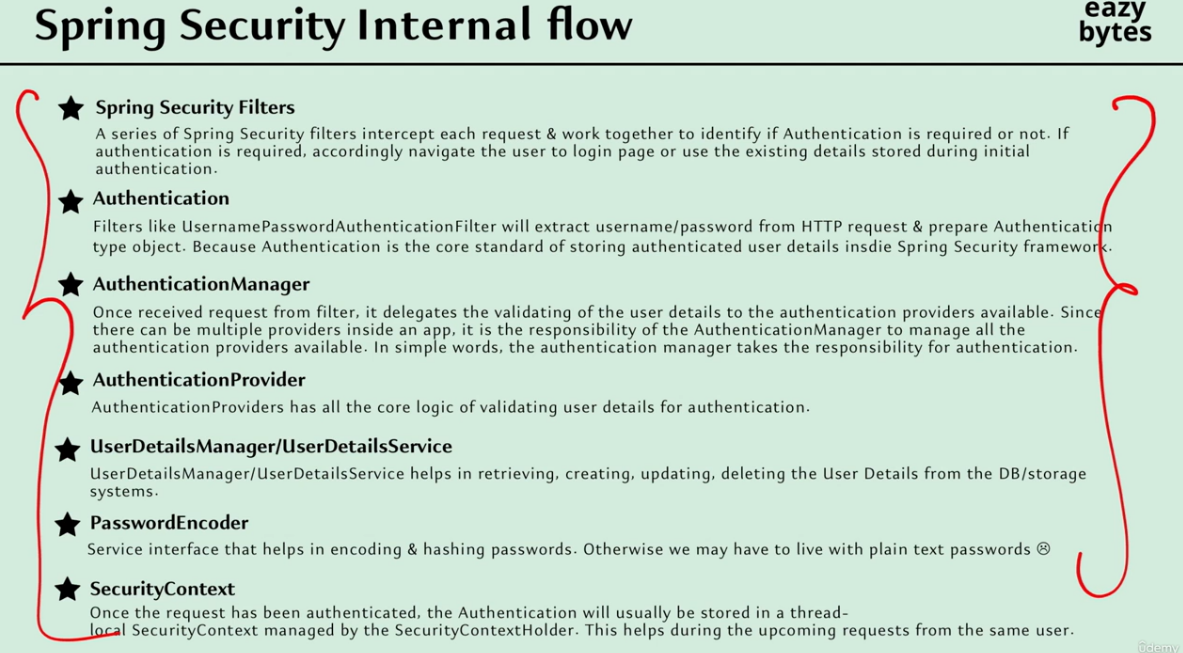
**Conclusion:**

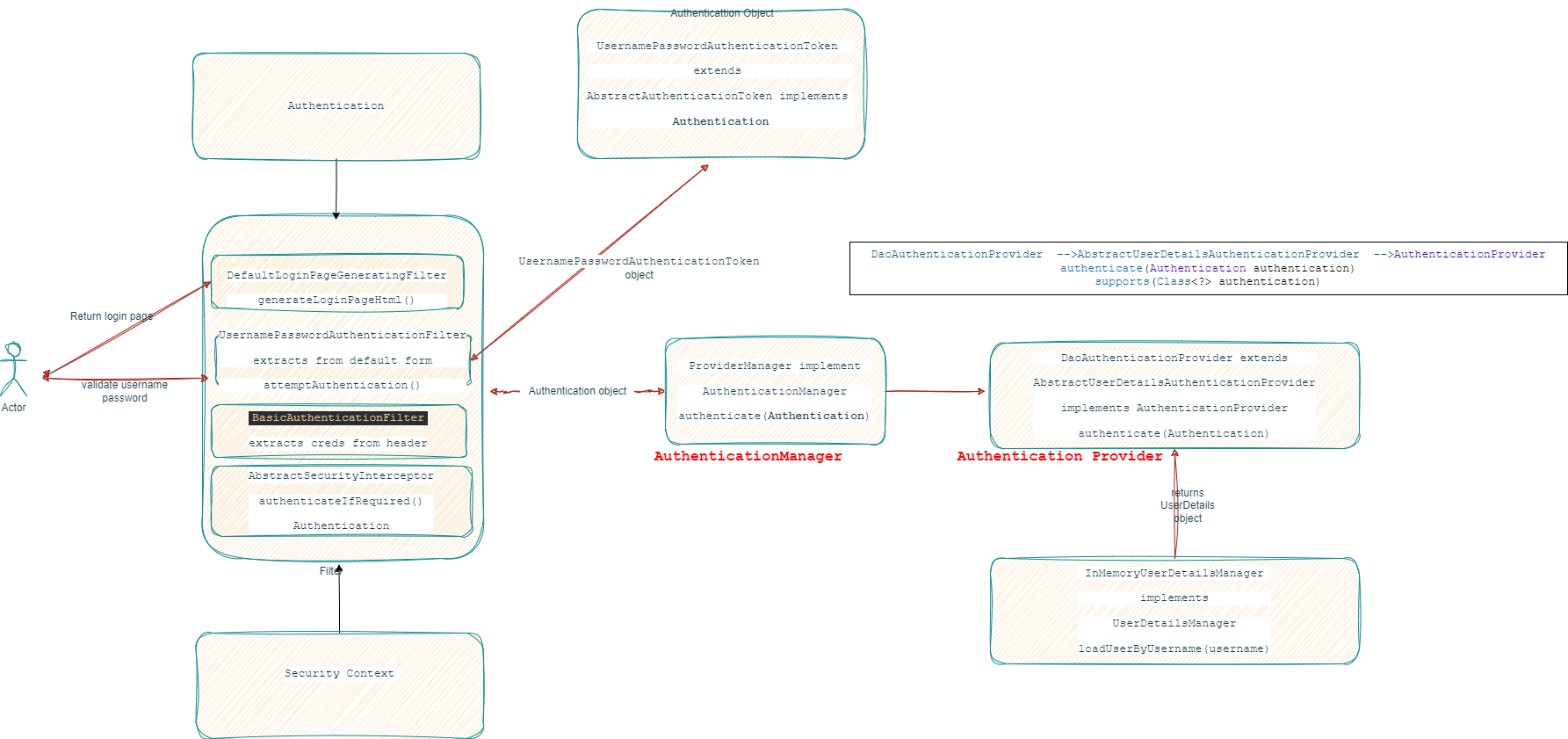
SHA-256 is a fundamental cryptographic hash function with crucial properties like deterministic scrambling, fixed output size, and irreversibility. These properties make it invaluable for various real-world applications, including verifying file integrity, securing authentication processes, and underpinning the security of blockchain technologies like Bitcoin. While fast and efficient for many purposes, its speed makes it less suitable for password hashing, where slower, more computationally intensive functions are preferred. The continued security and widespread adoption of SHA-256 highlight its importance in modern digital security.

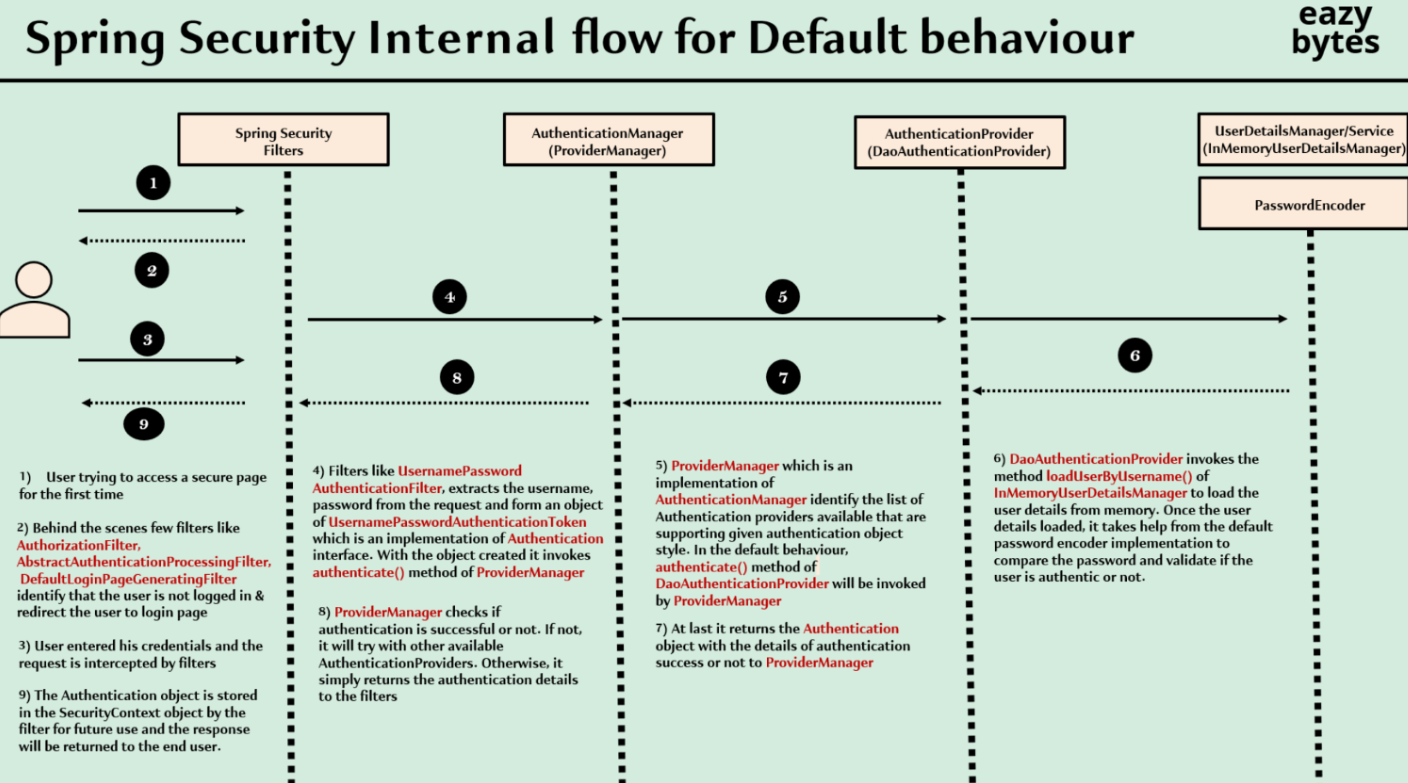
# Spring Security internal flow

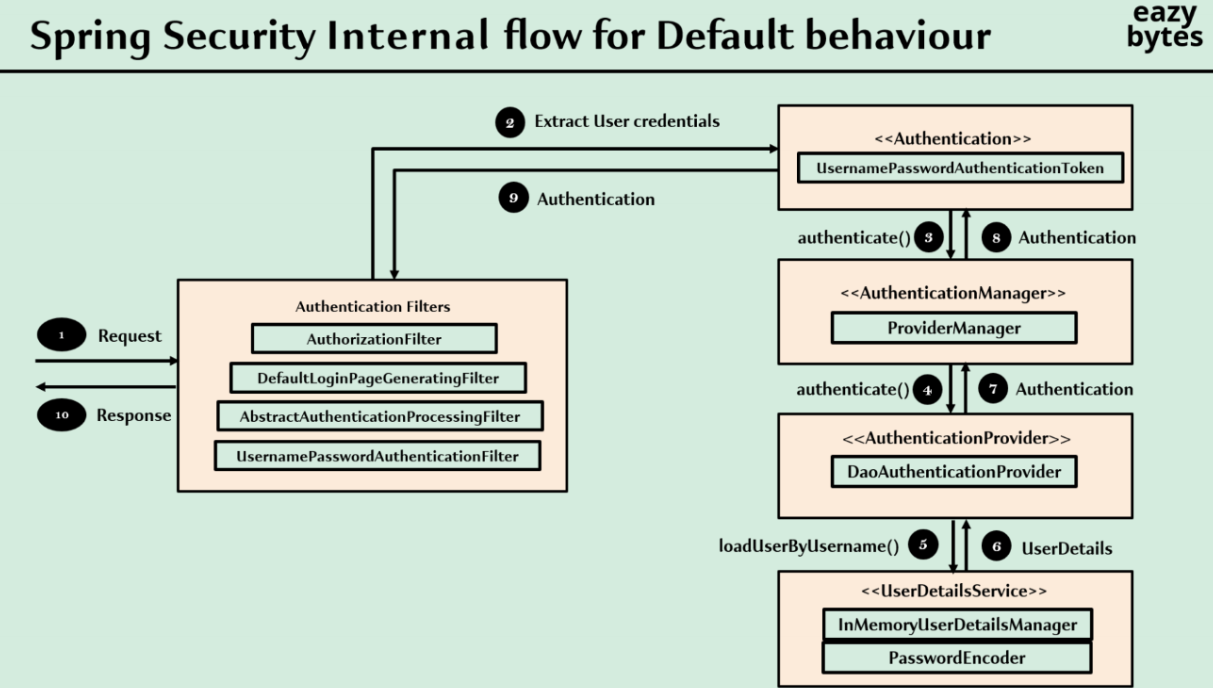


1. Spring security filters extract the username and password into the Authentication object.
2. The same Authentication object we are going to send as an input to the AuthenticationManager
3. AuthenticationManager passed the same Authentication object to the Authentication Provider.
4. Authentication Providers extract the username from the Authentication object and pass the username to the loadUserByUsername() method available inside the UserDetailService interface.
5. loadUserByUsername method from UserDetailService will extract UserDetails from DB/Memory etc





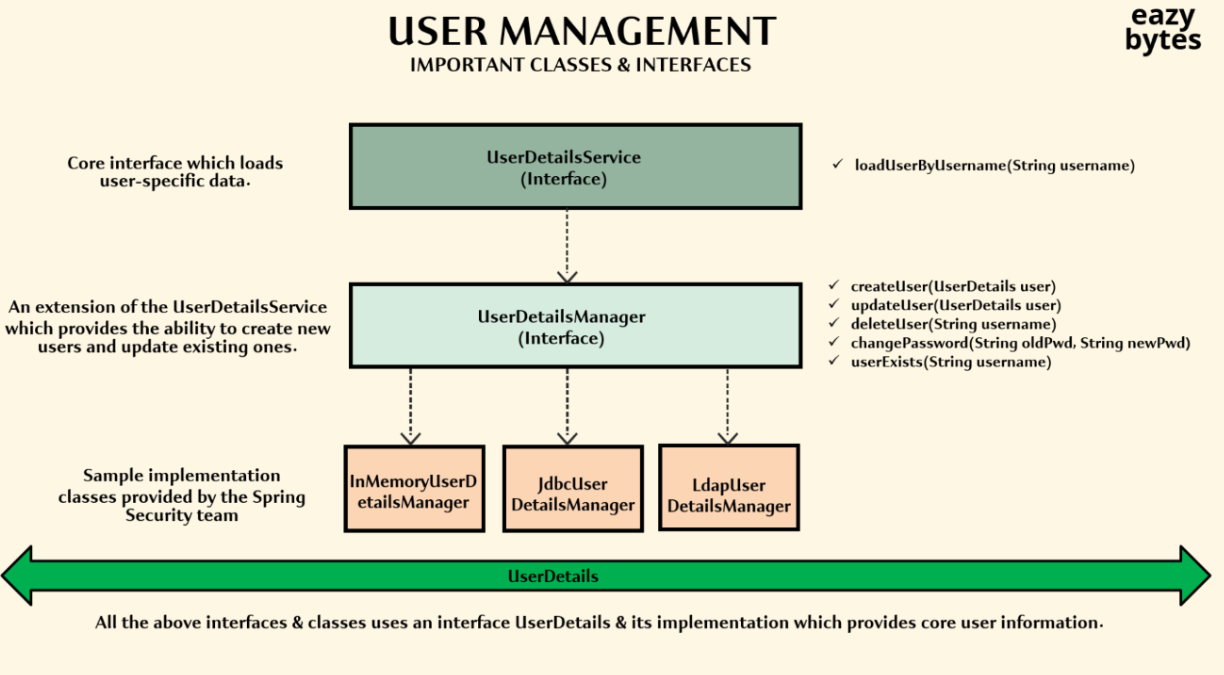




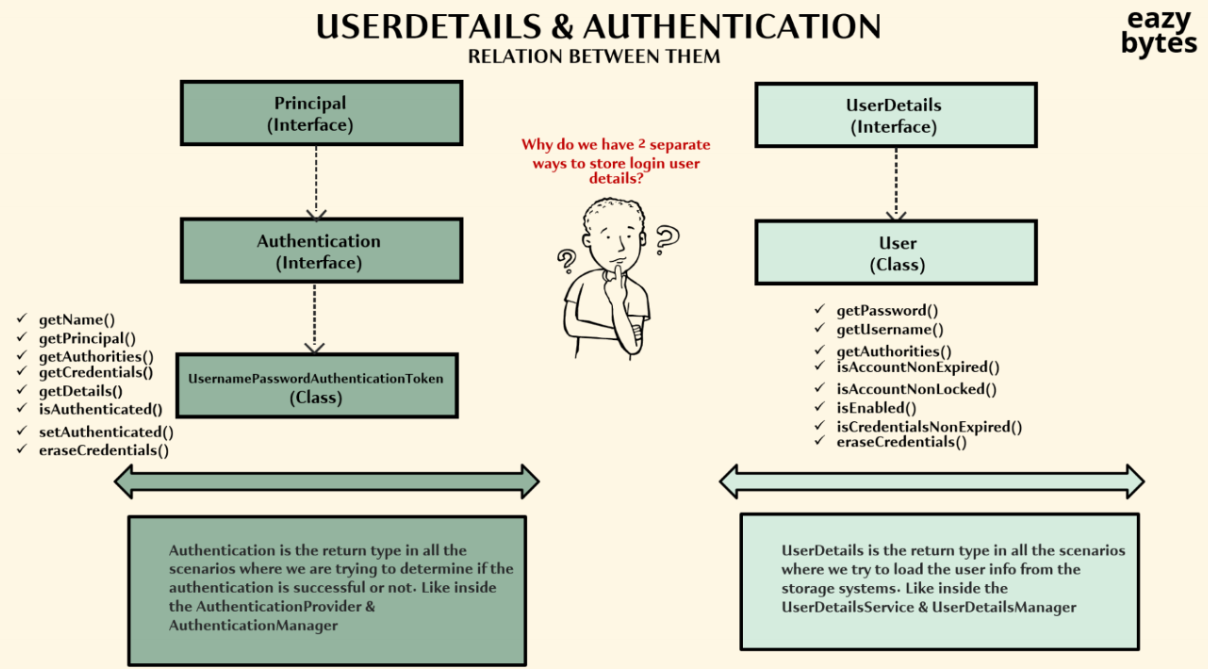
Important items

1. **org.springframework.security.core.userdetails.User** This Class will contain the User details fetched from DB/Memory/Application properties like username, password, authorities(Roles), accountExpired. accountLocked etc.
2. org.springframework.security.core.userdetails.UserDetails Interface implemented by user
3. org.springframework.security.core.userdetail.UserDetailsService Interface which exposes loadUserByUsername method to fetch User
4. **org.springframework.security.provisioning.UserDetailsManager** Interface which provides a lot of other Methods to create User, update user, change password etc.
5. **Authentication** This Class on other side of authentication flow which contains the username password details entered by user after comparison isAuthenticated() is set as true and password removed using eraseCredentials()

Difference between **UserDetailsService and UserDetailsManager.** Manager has additional functionalities related to CRUD operation of User. It is useful when we try to Utilize Spring security to create a user, change password, delete password etc. If we are just using Spring security for authentication then Service is enough which loads User details with methods loadUserByUsername



Difference between **UserDetails/User and Authentication .** User is used to fetch the user details from DB of any system which is storing user information. Authentication is used to fetch the credentials from the user request. Both are matched by Authentication Providers /managers and if user if authentication is successful then in AUthentication is isAuthenticated() is set as true for future use also eraseCredentials() is called to remorse password like details



# JWT

In general, there are two types of tokens

1. **Opaque Tokens** are typically random strings with no inherent meaning. Used to reference authentication information stored on the server-side.
2. **JSON Web Tokens (JWT)** are self-contained tokens that consist of three parts: a header, a payload, and a signature, encoded in Base64 URL. Requires a call to the authorization server or a dedicated introspection endpoint to validate and obtain user information. Encodes user information and claim directly within the token. Can be validated locally by verifying the token's signature using a public key, without needing a server call.

Opaque tokens are suitable for scenarios where token validation by a central server is feasible, such as within a secure internal network where as JWT tokens are ideal for stateless, distributed systems where quick token validation is needed without frequent server calls.

**ROLE OF TOKENS** IN AUTHN & AUTHZ

A Token can be a plain string of format universally unique identifier (UUID) or it can be of type JSON Web Token (JWT) usually that get generated when the user authenticated for the first time during login.

On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it’s valid, returns the secure resource to the client.

|  |  |  |
| --- | --- | --- |
| Client will receive the token after successful login in a header/query string etc.  Client system must make sure to send the same token value on all the further request to the backend |  | Server will send token and send to client and stores client details and token in memory  Sever will validate the token and return result |

**Key Advantages of Token-Based Authentication**

**Security**

* Limited exposure of user credentials within the network
* Tokens can be revoked during suspicious activities without invalidating the user credentials

**Reusability**

* Tokens can be used across different domains and services, making them suitable for single sign-on (SSO) systems

**Cross-Platform Compatibility**

* Tokens can be used across various platforms and devices, including web applications, mobile apps, and IoT devices

**Expiration**

* Tokens can have specific expiration times set, ensuring tokens are valid only for a predefined duration

**Self-contained**

* Tokens are self-contained and carry all the necessary information about the user, roles/authorities etc.

**Statelessness**

* The token contains all the information to identify the user, eliminating the need for the session state
* If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on
* JWT means JSON Web Token. It is a token implementation which will be in the JSON format and designed to use for the web requests.
* JWT is the most common and favourite token type that many systems use these days due to its special features and advantages.
* JWT tokens can be used both in the scenarios of Authorization/Authentication along with Information exchange which means you can share certain user related data in the token itself which will reduce the burden of maintaining such details in the sessions on the client/server side.

A JWT token has 3 parts each separated by a period(.) Below is a sample JWT token,

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjQ3Mjg0MiIsIm5hbWUiOiJBbWl0IFNhaGEiLCJlbWFpbCI6ImFtaXQ4Ny5zQHRjcy5jb20ifQ.OWyFwQWMwXie7FNZa1mPp-BOZeRllUWwfigwmrllg8s

* + 1. Header
    2. Payload
    3. Signature(Optional)

**Header** Inside the JWT header, we store metadata/info related to the token. If we chose to sign the token, the header contains the name of the algorithm that generates the signature.

{

"alg": "HS256",

"typ": "JWT"

}

**Payload** In the body, we can store details related to user, roles etc. which can be used later for Authentication and Authorization. Though there is no such limitation what we can send and how much we can send in the body, but we should put our best efforts to keep it as light as possible.

{

"id": "472842",

"name": "Amit Saha",

"email": "amit87.s@tcs.com"

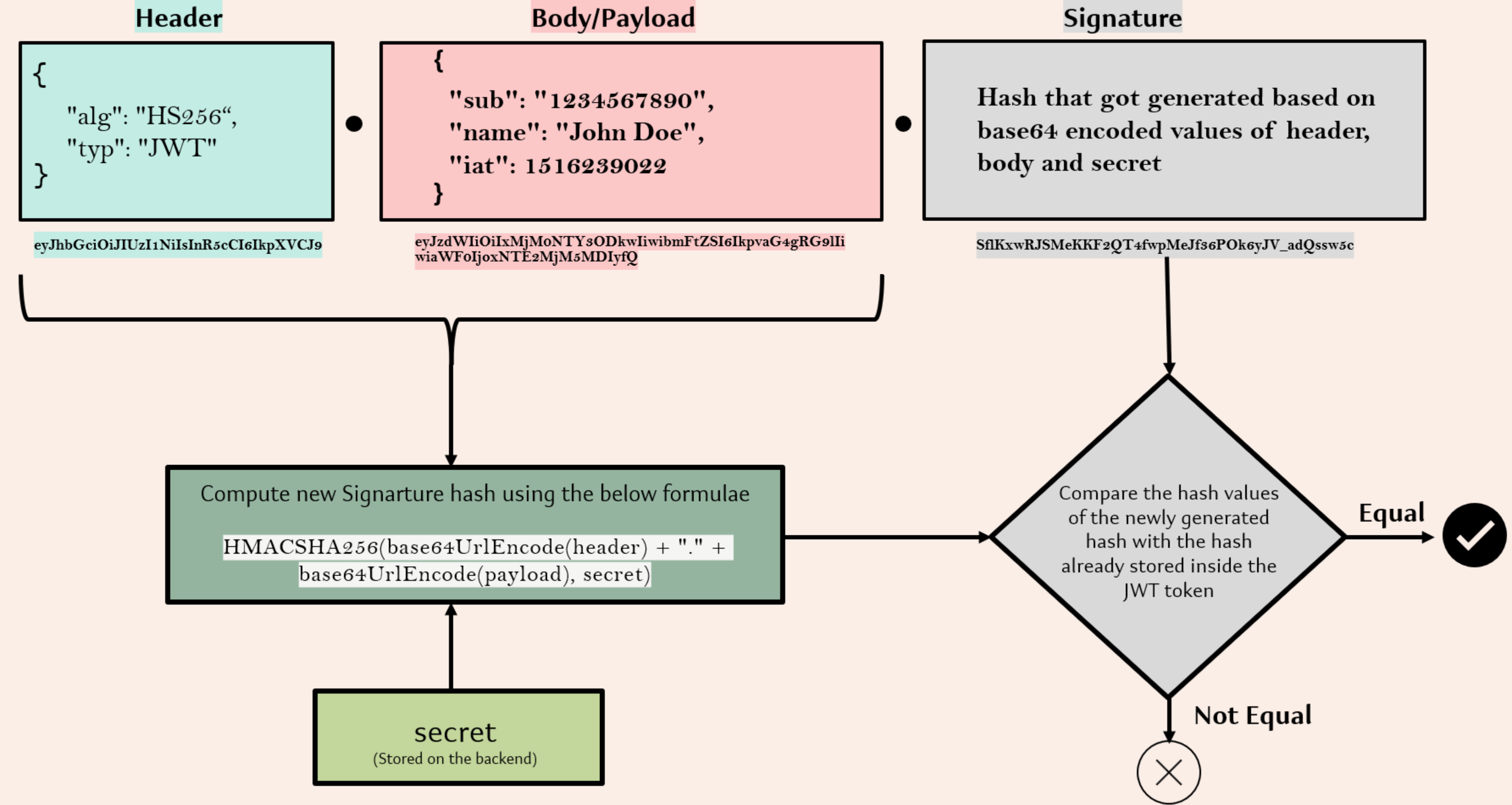
}

Signature

* The last part of the token is the digital signature. This part can be optional if the party that you share the JWT token is internal and that someone who you can trust but not open in the web.
* But if you are sharing this token to the client applications which will be used by all the users in the open web then we need to make sure that no one changed the header and body values like Authorities, username etc.
* To make sure that no one tampered the data on the network, we can send the signature of the content when initially the token is generated. To create the signature part, you must take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that.
* For example, if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way

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

* The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.



# OAUTH2

[OAuth 2.0](https://tools.ietf.org/html/rfc6749), which stands for “Open Authorization”, is a standard designed to allow a website or application to access resources hosted by other web apps on behalf of a user.

OAuth 2.0 is an authorization protocol and NOT an authentication protocol

OAuth 2.0 uses Access Tokens. An **Access Token** is a piece of data that represents the authorization to access resources on behalf of the end-user. OAuth 2.0 doesn’t define a specific format for Access Tokens. However, in some contexts, the JSON Web Token (JWT) format is often used. This enables token issuers to include data in the token itself. Also, for security reasons, Access Tokens may have an expiration date.

**OAuth 2.0 Terminology - Standard Definitions**

**Resource Owner** The entity that can grant access to a protected resource. Typically, the end-user who owns the data (such as their email, profile, etc.). In OAuth flows, this is the person who authorizes an application to access their account.

**Client** The application requesting access to protected resources on behalf of the Resource Owner. This can be a website, mobile app, desktop application, or any other software that needs to access protected resources but cannot directly use the Resource Owner's credentials.

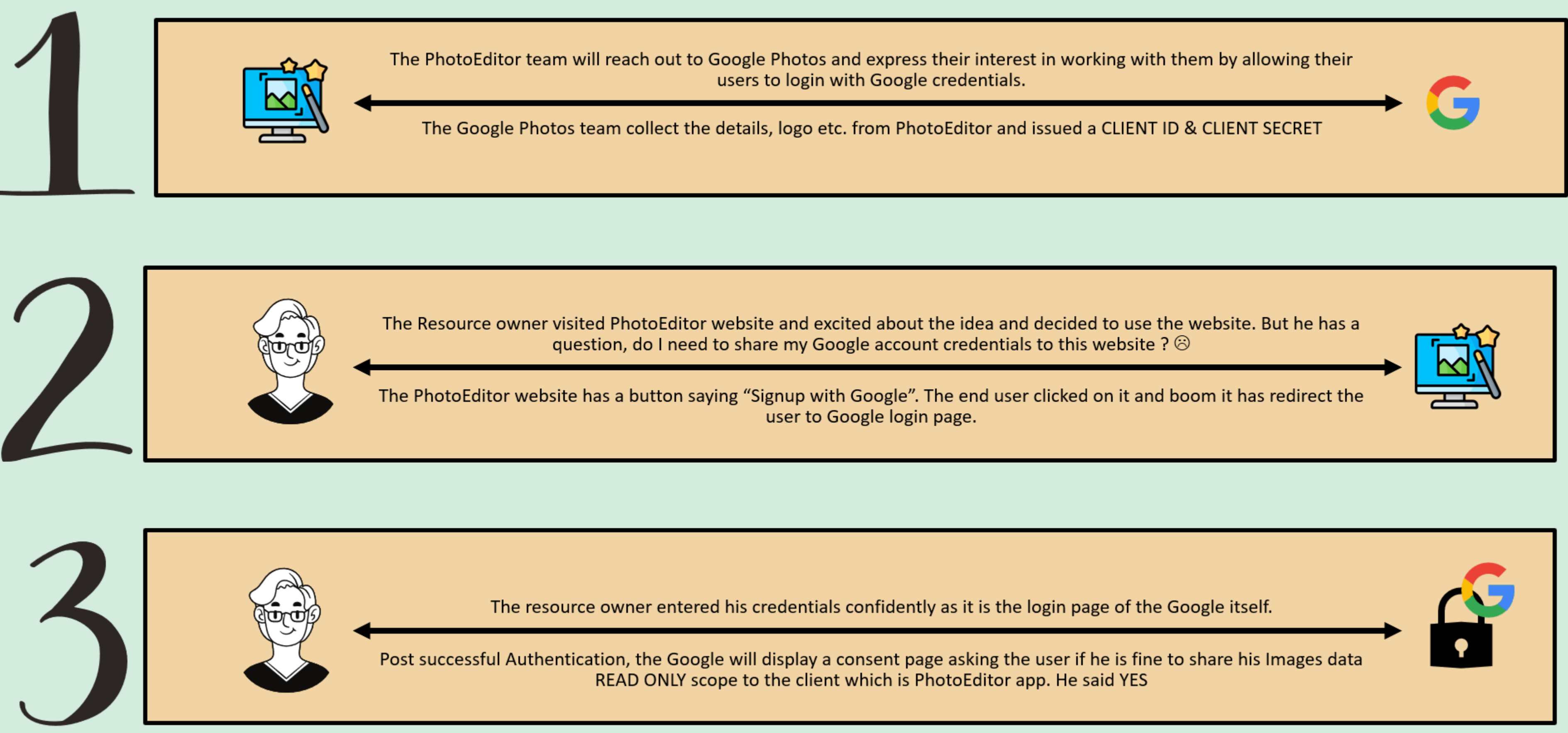
**Authorization Server** The server that authenticates the Resource Owner, obtains authorization decisions, and issues access tokens. It's responsible for validating user identity and handling the OAuth authorization process. It maintains information about clients and their allowed access scopes.

**Resource Server** The server that hosts the protected resources the Client wants to access. It accepts and validates access tokens from the Client and serves the requested resources if the token grants appropriate permissions. Often, this may be the same physical server as the Authorization Server but with a distinct logical role.

**Scopes** Granular permissions that define the extent of access granted to the Client. Scopes limit what the Client can do with the access token. Examples include read-only access to emails, write access to calendar events, or access to profile information. The Authorization Server enforces these scope limitations when issuing tokens.

These definitions establish the core components and roles within the OAuth 2.0 framework as formally specified in RFC 6749. Each plays a crucial part in enabling secure delegated access without sharing passwords across applications.

Sample OAUTH flow

A close-up of a text

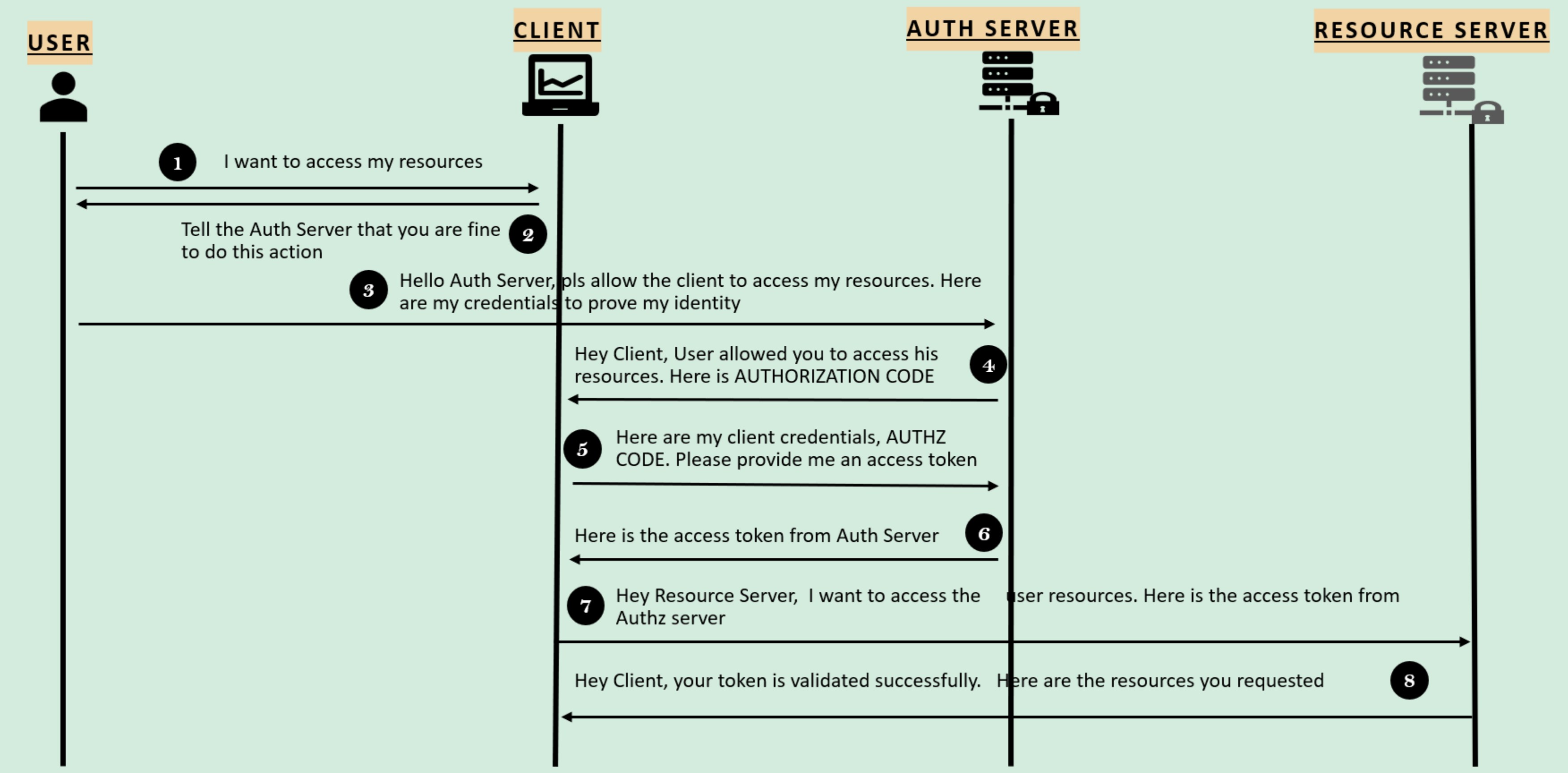
AI-generated content may be incorrect.

OAuth Grant Types

1. Authorization Code – When end user is involved
2. PKCE
3. Client Credentials
4. Device Code
5. Refresh Token
6. Legacy: Implicit Flow
7. Legacy: Password Grant

## Authorization Code

The Authorization Code grant is the most common OAuth 2.0 flow and is designed for applications that can securely store a client secret. It's optimized for security, particularly for web applications where the client can maintain confidentiality.



**Steps**

1. **Authorization Request**: The client redirects the user to the authorization server with its client ID, requested scope, a local state parameter, and a redirect URI.
2. **User Authentication & Consent**: The user logs in to the authorization server and approves the requested permissions.
3. **Authorization Code Response**: The authorization server redirects back to the client's redirect URI with a temporary authorization code and the original state parameter.
4. **Token Request**: The client sends a server-to-server request to the authorization server with the authorization code, client ID, client secret, and redirect URI.
5. **Token Response**: The authorization server validates the request and returns an access token, typically with a refresh token.
6. **Resource Access**: The client uses the access token to request protected resources from the resource server.

In the step 3, where client is making a request to Auth Server endpoint, must send the below important details,

* **client\_id** — the id which identifies the client application by the Auth Server. This will be granted when the client registers first time with the Auth server.
* **redirect\_uri** — the URI value which the Auth server needs to redirect post successful authentication. If a default value is provided during the registration, then this value is optional
* **scope** - like authorities. Specifies level of access that client is requesting like READ
* **state** — CSRF token value to protect from CSRF attacks
* **response\_type** — With the value ‘token’ which indicates that we want to follow implicit grant type

In the step 5 where client after received an authorization code from Auth server, it will again make a request to Auth server for a token with the below values,

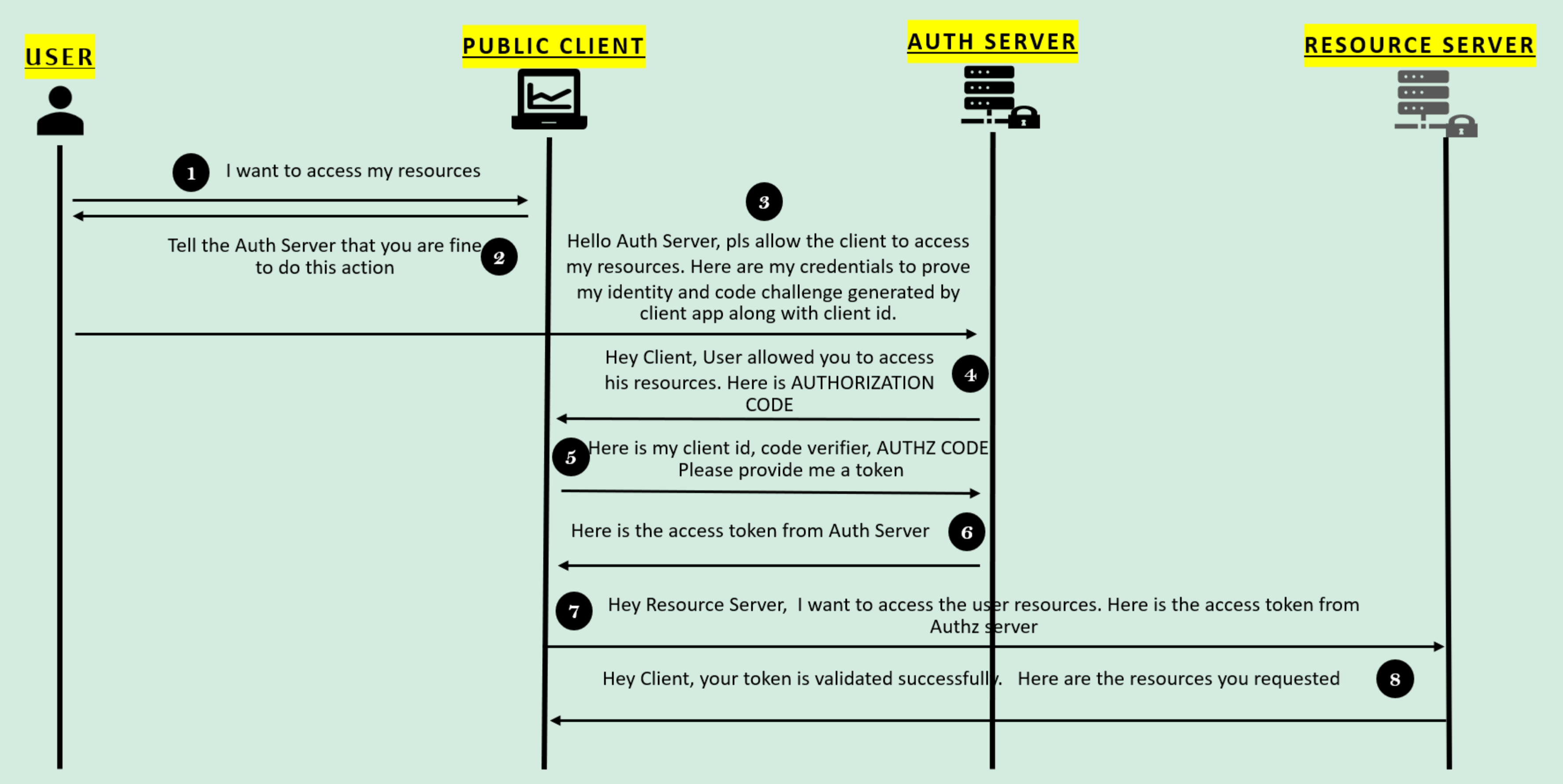
* code - the authorization code received from the above steps
* client\_id & client\_secret — the client credentials which are registered with the auth server. Please note that these are not user credentials
* grant\_type — With the value ‘authorization\_code’ which identifies the kind of grant type is used
* redirect\_uri

**Security Features**

* Authorization code is short-lived and single-use
* Full authentication flow happens server-to-server, not in the browser
* State parameter protects against CSRF attacks
* PKCE extension available for additional security with public clients
* Separates authentication from token issuance

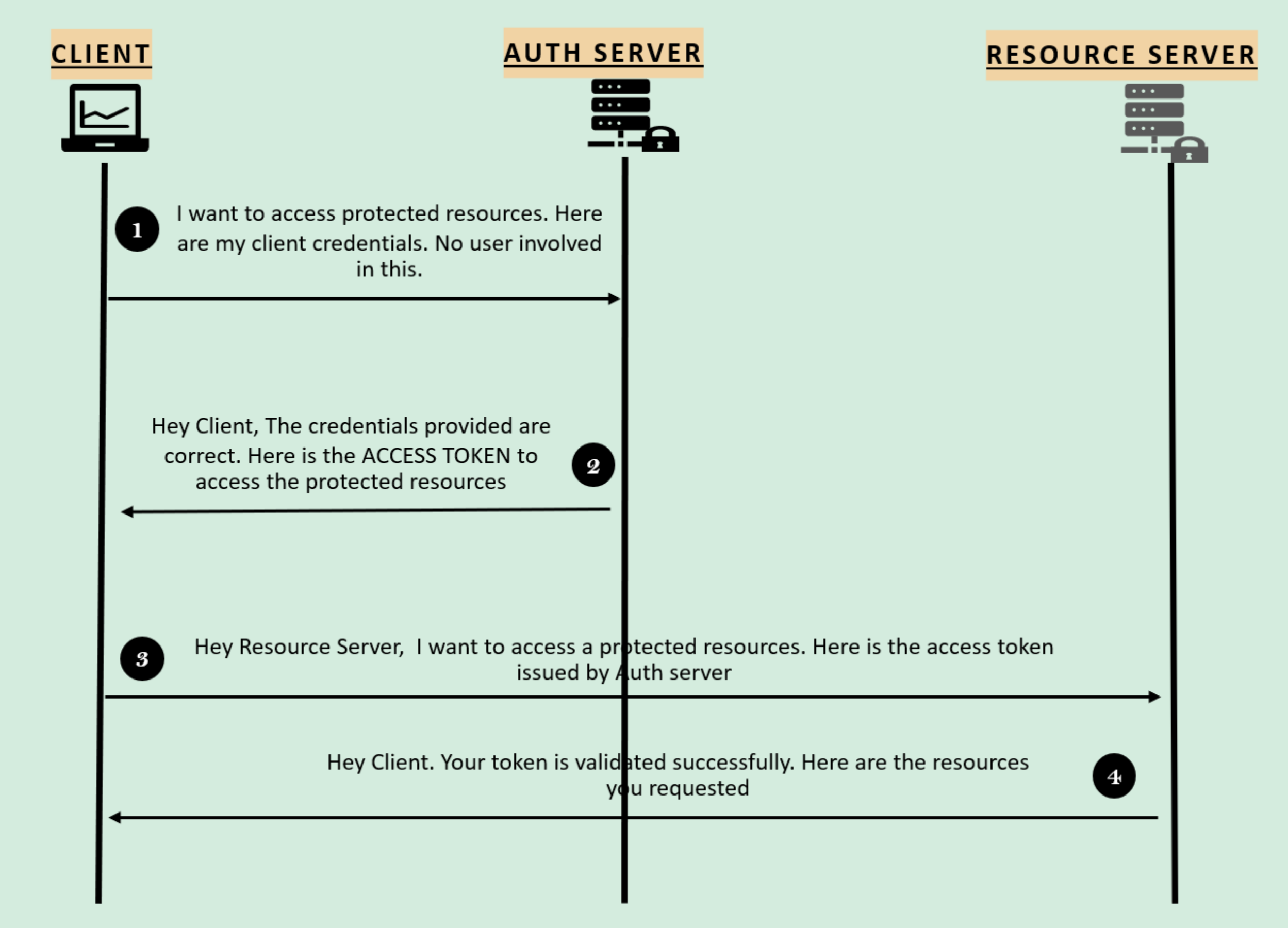
## PKCE

* When public clients (e.g., native and single-page applications) request Access Tokens, some additional security concerns are posed that are not mitigated by the Authorization Code Flow alone. This is because public clients cannot securely store a Client Secret.
* Given these situations, OAuth 2.0 provides a version of the Authorization Code Flow for public client applications which makes use of a Proof Key for Code Exchange (PKCE).
* The PKCE-enhanced Authorization Code Flow follows below steps,
  + Once user clicks login, client app creates a cryptographically random code\_verifier and from this generates a code\_challenge.
  + code challenge is a Base64-URL-encoded string of the SHA256 hash of the code verifier.
  + Redirects the user to the Authorization Server along with the code\_challenge.
  + Authorization Server stores the code\_challenge and redirects the user back to the application with an authorization code, which is good for one use.
  + Client App sends the authorization code and the code\_verifier(created in step 1) to the Authorization Server.
  + Authorization Server verifies the code\_challenge and code\_verifier. If they are valid it responds with ID Token and Access Token (and optionally, a Refresh Token).



## Client Credentials

The Client Credentials grant flow is one of the OAuth 2.0 authorization flows specifically designed for server-to-server authentication when there's no user context involved. It's the simplest OAuth 2.0 grant type and is used when the client itself is the resource owner.



**Steps**

1. The client (usually a server/service) authenticates with the authorization server using its client ID and client secret.
2. The authorization server validates these credentials.
3. If valid, the authorization server returns an access token to the client.
4. The client uses this access token to access protected resources on the resource server.

**Security Considerations**

* The client secret must be kept secure and never exposed to public clients
* Usually limited to confidential clients (servers that can securely store credentials)
* Scopes should be strictly limited to only what's necessary
* No refresh tokens are typically issued since the client can always request a new token

This flow is ideal for backend services that need to communicate with other APIs without user interaction, providing a secure authentication mechanism for service-to-service communication.

Authorization server examples

1. Key Cloak
2. Okta
3. Amazon Cognito
4. Forgerock
5. Fusion auth

# Authorization flow types

## Client credentials

Between 2 api