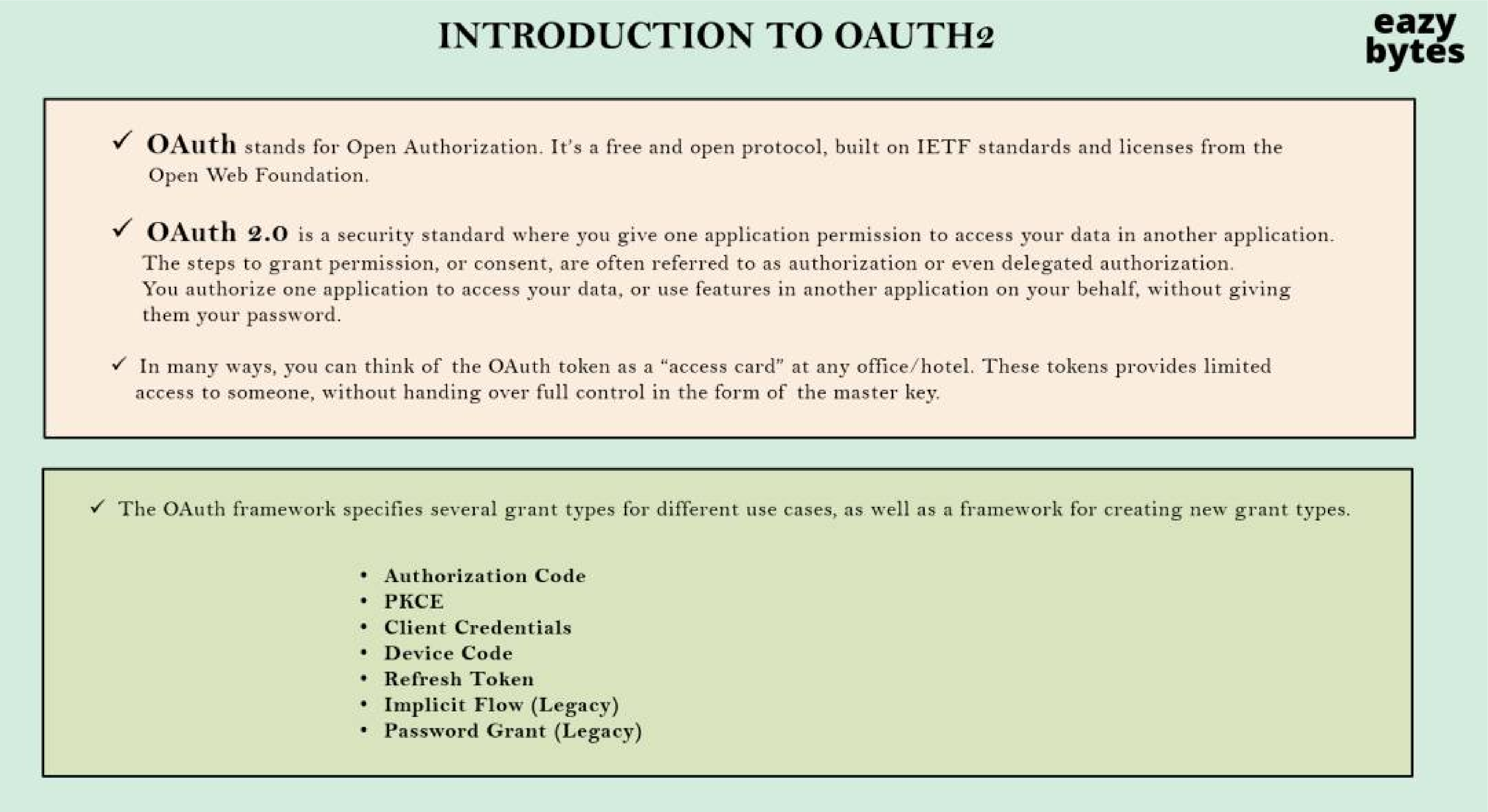
8-OAuth 2.0 Overview



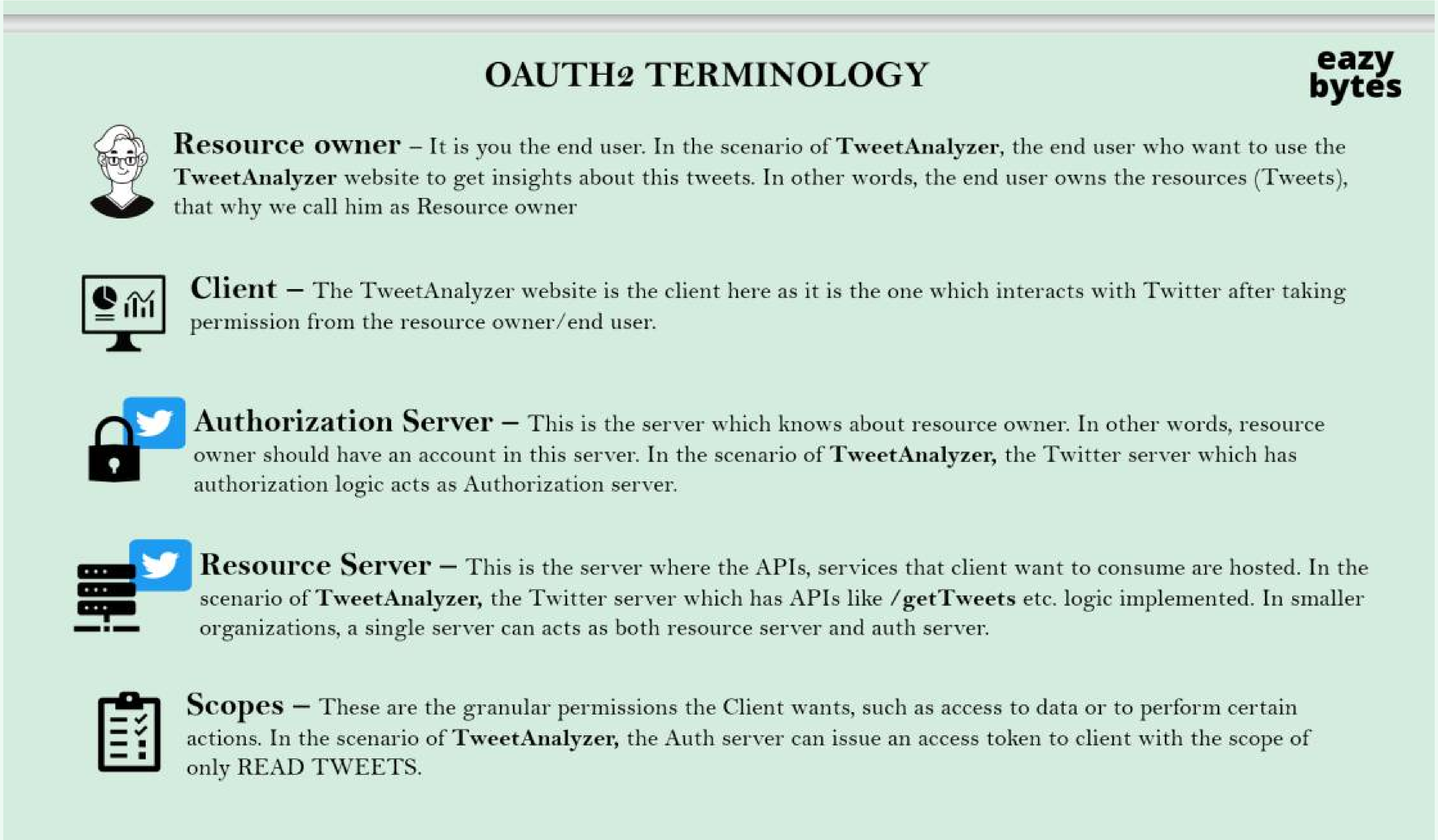
INTRODUCTION TO OAUTH2

eazy bytes

- ✓ **OAuth** stands for Open Authorization. It's a free and open protocol, built on IETF standards and licenses from the Open Web Foundation.
- ✓ **OAuth 2.0** is a security standard where you give one application permission to access your data in another application. The steps to grant permission, or consent, are often referred to as authorization or even delegated authorization. You authorize one application to access your data, or use features in another application on your behalf, without giving them your password.
- ✓ In many ways, you can think of the OAuth token as a "access card" at any office/hotel. These tokens provides limited access to someone, without handing over full control in the form of the master key.

- ✓ The OAuth framework specifies several grant types for different use cases, as well as a framework for creating new grant types.

  - **Authorization Code**
  - **PKCE**
  - **Client Credentials**
  - **Device Code**
  - **Refresh Token**
  - **Implicit Flow (Legacy)**
  - **Password Grant (Legacy)**



OAUTH2 TERMINOLOGY

eazy bytes

Resource owner – It is you the end user. In the scenario of **TweetAnalyzer**, the end user who want to use the **TweetAnalyzer** website to get insights about this tweets. In other words, the end user owns the resources (Tweets), that why we call him as Resource owner

Client – The TweetAnalyzer website is the client here as it is the one which interacts with Twitter after taking permission from the resource owner/end user.

Authorization Server – This is the server which knows about resource owner. In other words, resource owner should have an account in this server. In the scenario of **TweetAnalyzer,** the Twitter server which has authorization logic acts as Authorization server.

Resource Server – This is the server where the APIs, services that client want to consume are hosted. In the scenario of **TweetAnalyzer,** the Twitter server which has APIs like **/getTweets** etc. logic implemented. In smaller organizations, a single server can acts as both resource server and auth server.

Scopes – These are the granular permissions the Client wants, such as access to data or to perform certain actions. In the scenario of **TweetAnalyzer,** the Auth server can issue an access token to client with the scope of only READ TWEETS.

Kindly explain the above image with detailed explanation

The images you've shared provide an overview and detailed explanation of **OAuth 2.0** and its key terminologies. Here's a detailed breakdown of each section:

Image 1: Introduction to OAuth2

1. **What is OAuth?**
    - OAuth stands for **Open Authorization**.
    - It is a free and open protocol developed based on IETF standards.
    - Licensed by the Open Web Foundation, OAuth enables secure access delegation.
2. **What is OAuth 2.0?**
    - OAuth 2.0 is a **security standard** that allows one application to gain permission to access your data in another application.
    - The process of granting access is called **authorization** or **delegated authorization**.
    - This eliminates the need for sharing your password directly with third-party applications.
    - Example: You can allow an app to post on your social media account without giving it full access to your account.
3. **OAuth Token as an Access Card**
    - An OAuth token acts like an **access card**.
    - It provides **limited access** to the resource owner's data without sharing full control or the master credentials (like a username and password).
4. **Grant Types in OAuth**
    - OAuth defines several **grant types** for different use cases, allowing developers to create flexible implementations:
        - **Authorization Code**: Used in server-side apps for secure token exchange.
        - **PKCE (Proof Key for Code Exchange)**: An extension of the Authorization Code for increased security, especially in mobile/desktop apps.
        - **Client Credentials**: Used for machine-to-machine communication.
        - **Device Code**: For devices with limited input capabilities.
        - **Refresh Token**: Used to obtain new tokens without re-authentication.
        - **Implicit Flow (Legacy)**: An older flow, generally avoided now due to security concerns.
        - **Password Grant (Legacy)**: Also a legacy approach; requires sharing the user's password, which is insecure.

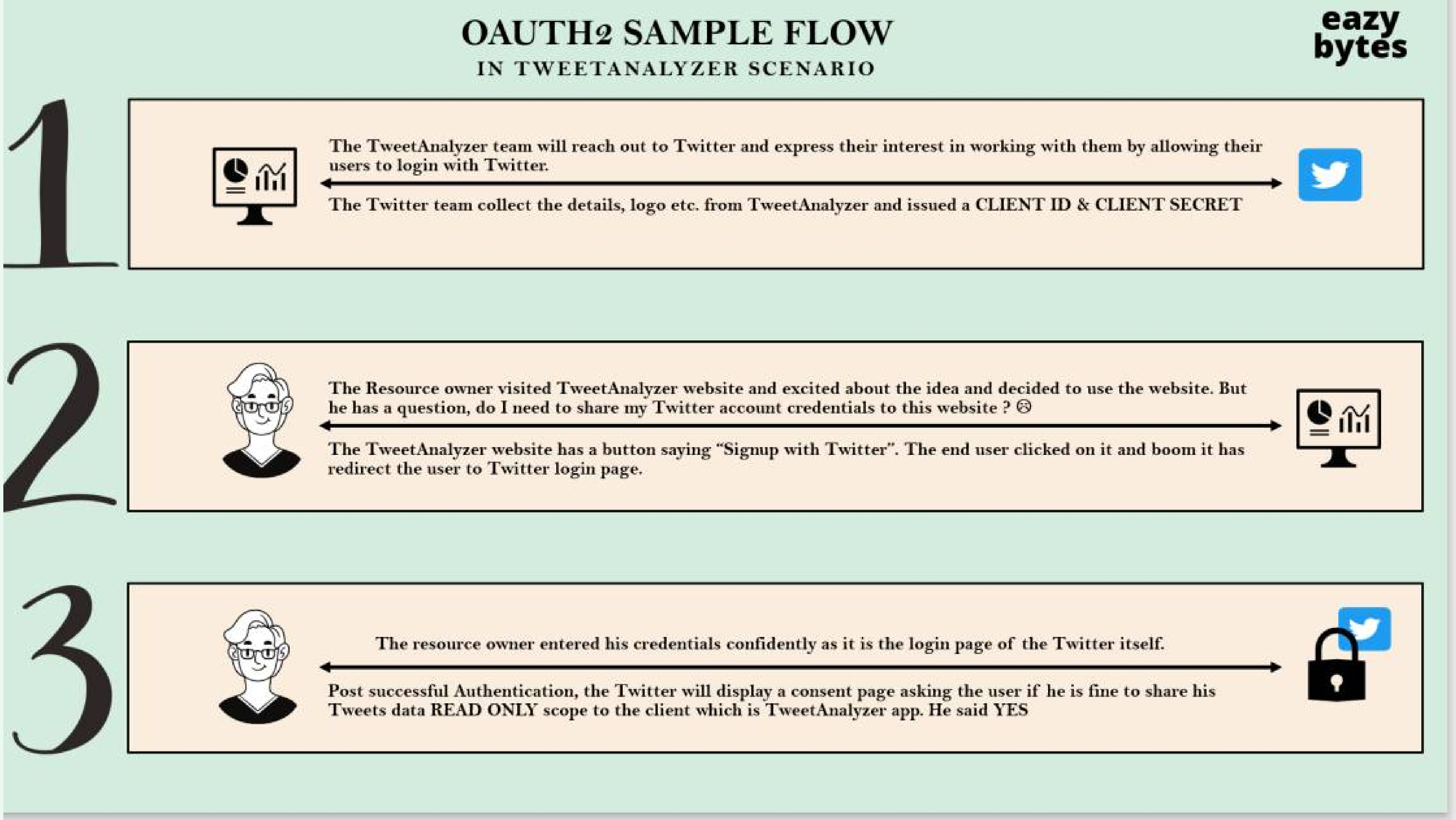
Image 2: OAuth2 Terminology

1. **Resource Owner**
    - The **end-user** or individual who owns the data/resources.
    - In the example of **TweetAnalyzer**, the **resource owner** is the user who wants to analyze their tweets.
2. **Client**
    - The application or entity that requests access to the resource owner's data.

   - Example: **TweetAnalyzer** is the client application that interacts with Twitter after receiving the resource owner’s permission.
3. **Authorization Server**
   - The server that authenticates the resource owner and issues tokens.
   - Example: In the case of TweetAnalyzer, the **Twitter Authorization Server** validates the user's credentials and provides access tokens.
4. **Resource Server**
   - The server hosting the APIs or resources the client application wants to access.
   - Example: The **Twitter Resource Server** contains APIs such as **`` `/getTweets` ``**, where the client (TweetAnalyzer) fetches the required data.
5. **Scopes**
   - Scopes define the **granular permissions** for accessing resources.
   - Example: For TweetAnalyzer, a **scope** might allow only **READ TWEETS**, ensuring limited access.

Summary

OAuth 2.0 provides a secure and flexible way to delegate access permissions between applications. The protocol leverages tokens instead of passwords, ensuring enhanced security and control. The clear division between **resource owners**, **clients**, **authorization servers**, and **resource servers** makes OAuth 2.0 a scalable solution for modern application ecosystems.



OAUTH2 SAMPLE FLOW

IN TWEETANALYZER SCENARIO

1

The TweetAnalyzer team will reach out to Twitter and express their interest in working with them by allowing their users to login with Twitter.

The Twitter team collect the details, logo etc. from TweetAnalyzer and issued a CLIENT ID & CLIENT SECRET

2

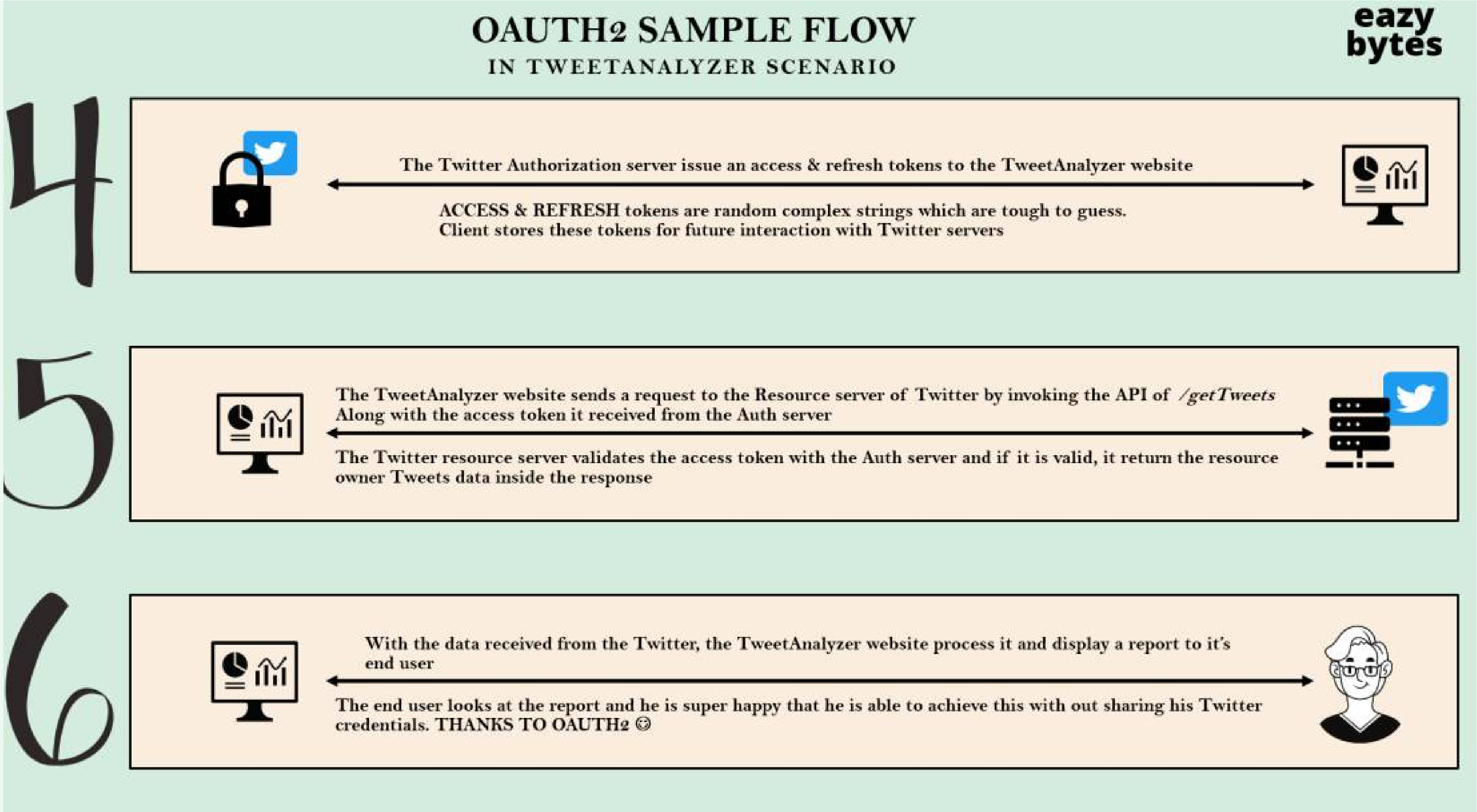
The Resource owner visited TweetAnalyzer website and excited about the idea and decided to use the website. But he has a question, do I need to share my Twitter account credentials to this website ? ☹

The TweetAnalyzer website has a button saying "Signup with Twitter". The end user clicked on it and boom it has redirect the user to Twitter login page.

3

The resource owner entered his credentials confidently as it is the login page of the Twitter itself.

Post successful Authentication, the Twitter will display a consent page asking the user if he is fine to share his Tweets data READ ONLY scope to the client which is TweetAnalyzer app. He said YES



OAUTH2 SAMPLE FLOW

IN TWEETANALYZER SCENARIO

eazy bytes

4

The Twitter Authorization server issue an access & refresh tokens to the TweetAnalyzer website

ACCESS & REFRESH tokens are random complex strings which are tough to guess.
Client stores these tokens for future interaction with Twitter servers

5

The TweetAnalyzer website sends a request to the Resource server of Twitter by invoking the API of */getTweets*
Along with the access token it received from the Auth server

The Twitter resource server validates the access token with the Auth server and if it is valid, it return the resource owner Tweets data inside the response

6

With the data received from the Twitter, the TweetAnalyzer website process it and display a report to it's end user

The end user looks at the report and he is super happy that he is able to achieve this with out sharing his Twitter credentials. THANKS TO OAUTH2 ☺

Kindly explain the above image with detailed explanation

The images depict a **sample OAuth2 flow** for a scenario where **TweetAnalyzer**, a third-party application, interacts with Twitter on behalf of an end-user to analyze their tweets. Here is a detailed explanation of the flow, step by step:

Step 1: Application Registration

- The team behind **TweetAnalyzer** approaches Twitter and expresses interest in enabling users to log in using their Twitter accounts.
- Twitter, in response, collects necessary information (e.g., application details, logo) and issues:
    - A **Client ID**: A public identifier for the TweetAnalyzer application.
    - A **Client Secret**: A confidential key used by TweetAnalyzer to securely authenticate with Twitter.

Step 2: Resource Owner Accesses the Client

- The **resource owner** (end-user) visits the TweetAnalyzer website and shows interest in using its services.
- The resource owner has a valid question: "Do I need to share my Twitter credentials with TweetAnalyzer?"
    - The answer is **NO**.
- TweetAnalyzer provides a "Sign up with Twitter" button. When clicked, the user is redirected to **Twitter's login page**.

Step 3: User Authentication on Authorization Server

- The resource owner confidently enters their credentials on Twitter's login page, knowing that they are not directly sharing them with TweetAnalyzer.
- Post-authentication, Twitter presents a **consent page** to the resource owner:
    - The page asks for permission to allow TweetAnalyzer to access the user's tweet data with a **READ-ONLY** scope.
    - The user grants permission by clicking "YES."

Step 4: Access and Refresh Tokens Issued

- Upon user consent, the **Twitter Authorization Server** generates:
    - An **Access Token**: A short-lived token allowing TweetAnalyzer to access the user's tweet data.
    - A **Refresh Token**: A long-lived token used to request a new Access Token when the current one expires.
- These tokens are securely stored by TweetAnalyzer for future communication with Twitter.

Step 5: API Request to Resource Server

- TweetAnalyzer uses the **Access Token** to send a request to the **Twitter Resource Server** (e.g., an API endpoint like `` `/getTweets` ``) to fetch the user's tweet data.

- The Twitter Resource Server validates the Access Token with the Authorization Server.
- If the token is valid, the Resource Server provides the requested data (e.g., the user's tweets) in the response.

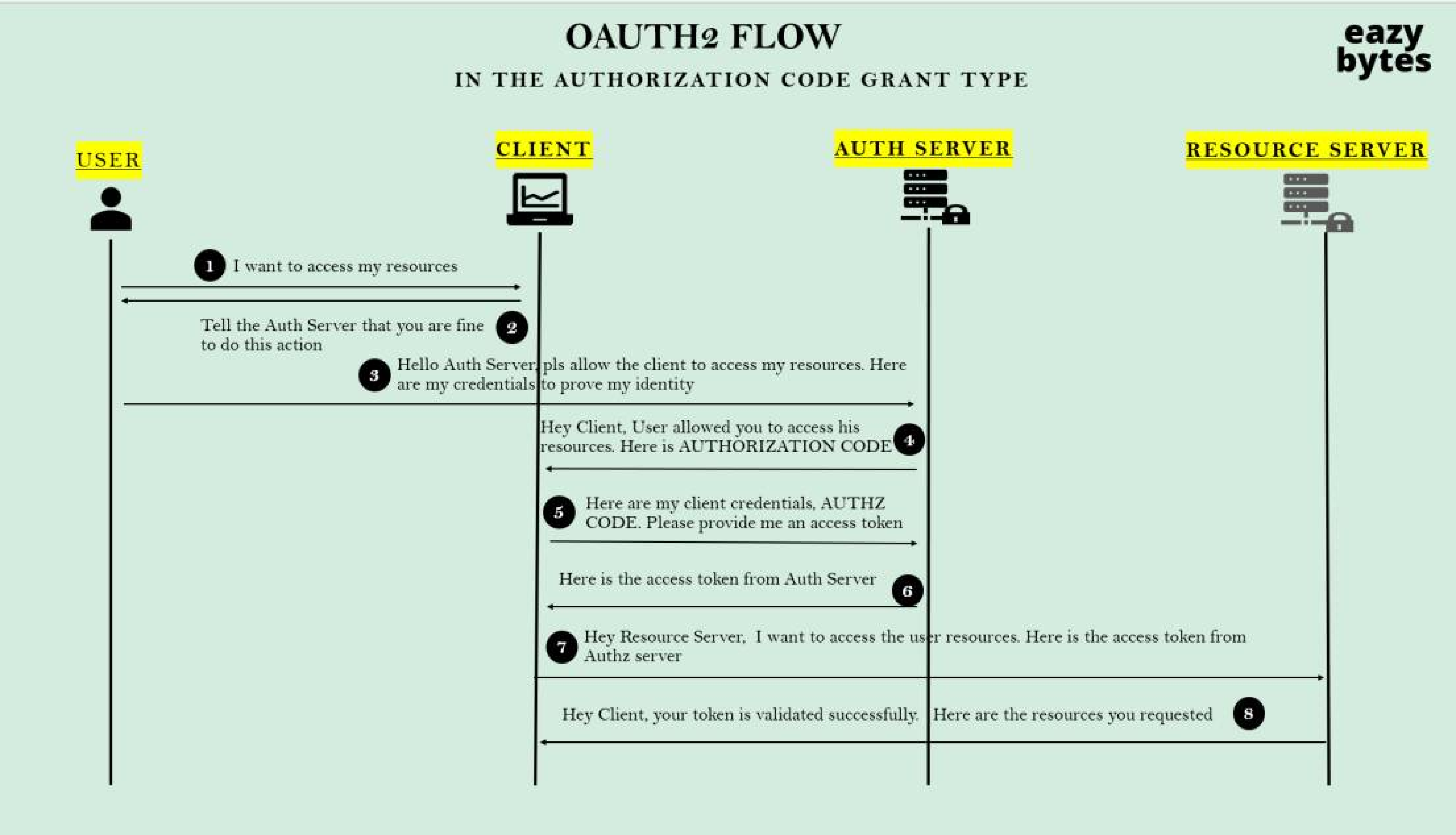
Step 6: Data Processing and Display

- TweetAnalyzer processes the tweet data received from Twitter and generates a report for the user.
- The resource owner (user) views the analyzed report, satisfied that the entire process occurred without sharing sensitive credentials directly with TweetAnalyzer.

Summary

This flow showcases the power of OAuth2 in enabling secure, token-based communication between a third-party application (TweetAnalyzer) and a service provider (Twitter) on behalf of a user. Key benefits include:

- **Separation of credentials**: The user never shares their password with the third-party app.
- **Granular permissions**: The user grants specific scopes (e.g., READ-ONLY access).
- **Token lifecycle**: Tokens ensure security and control over the duration and scope of access.



OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

1 I want to access my resources

2 Tell the Auth Server that you are fine to do this action

3 Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity

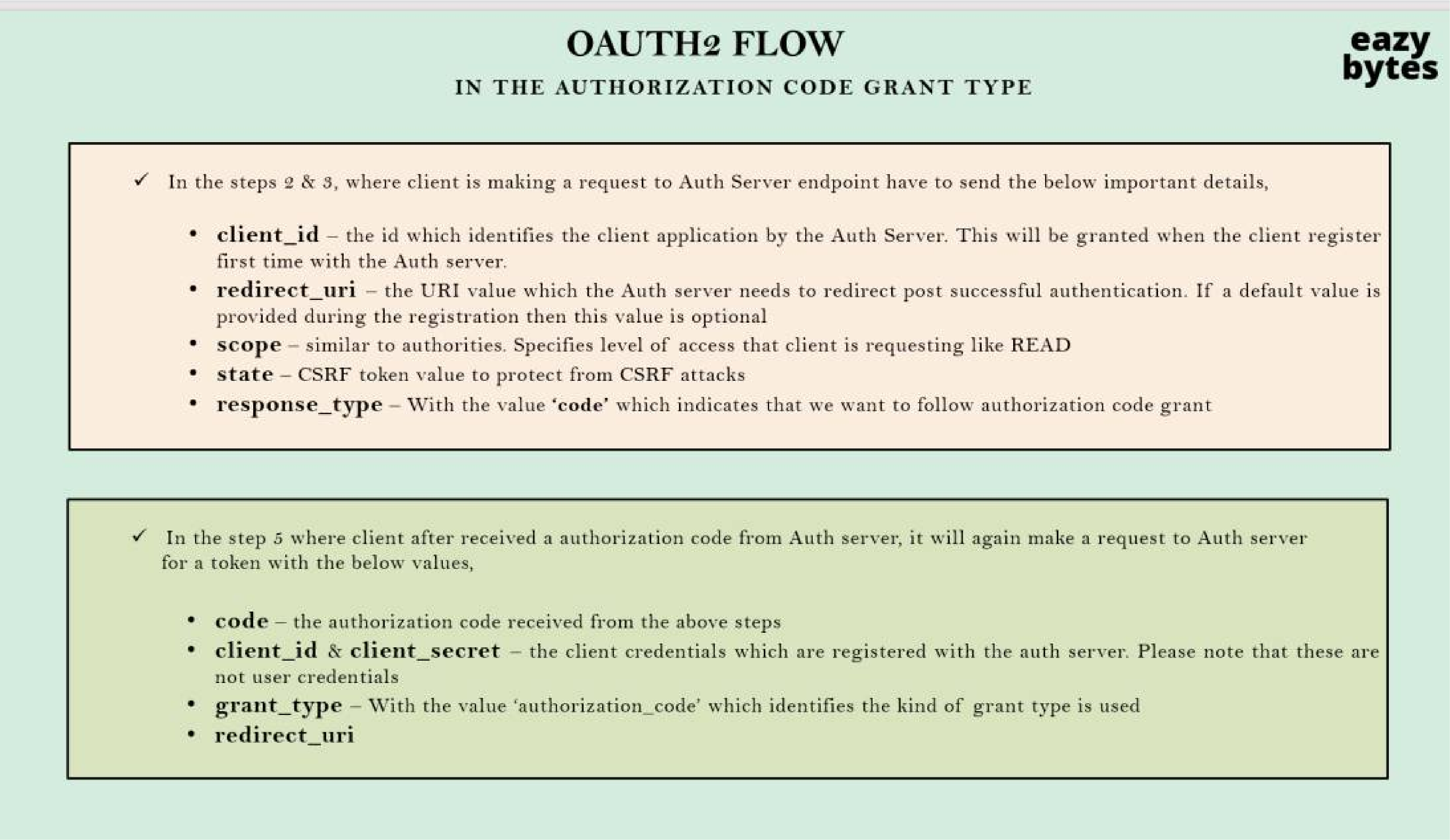
4 Hey Client, User allowed you to access his resources. Here is AUTHORIZATION CODE

5 Here are my client credentials, AUTHZ CODE. Please provide me an access token

6 Here is the access token from Auth Server

7 Hey Resource Server, I want to access the user resources. Here is the access token from Authz server

8 Hey Client, your token is validated successfully. Here are the resources you requested



OAUTH2 FLOW

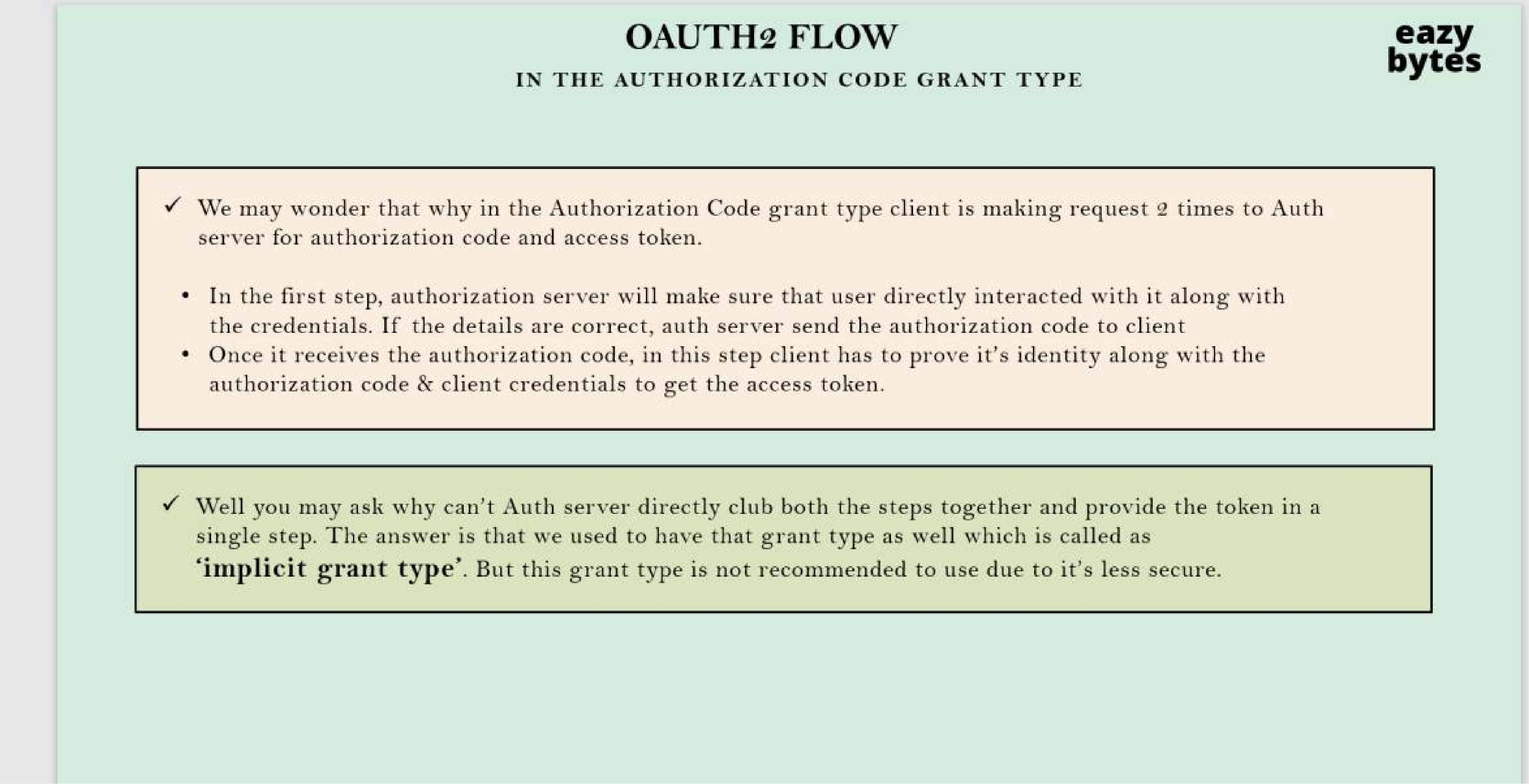
IN THE AUTHORIZATION CODE GRANT TYPE

✓ In the steps 2 & 3, where client is making a request to Auth Server endpoint have to 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 register 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** – similar to 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 **'code'** which indicates that we want to follow authorization code grant

✓ In the step 5 where client after received a 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**



OAUTH2 FLOW

IN THE AUTHORIZATION CODE GRANT TYPE

✓ We may wonder that why in the Authorization Code grant type client is making request 2 times to Auth server for authorization code and access token.

- In the first step, authorization server will make sure that user directly interacted with it along with the credentials. If the details are correct, auth server send the authorization code to client
- Once it receives the authorization code, in this step client has to prove it's identity along with the authorization code & client credentials to get the access token.

✓ Well you may ask why can't Auth server directly club both the steps together and provide the token in a single step. The answer is that we used to have that grant type as well which is called as **'implicit grant type'**. But this grant type is not recommended to use due to it's less secure.

Kindly explain the above image with detailed explanation

The images explain the **OAuth2 Authorization Code Grant Type** flow in detail. Here's a comprehensive breakdown:

Part 1: Sequence Flow of Authorization Code Grant Type

1. **Step 1 (User):**
    - The user (resource owner) expresses the intention to access their resources using a client application.
2. **Step 2 (User → Auth Server):**
    - The user consents to allow the client application to access their resources.
    - The client redirects the user to the **Authorization Server's login page**, where the user provides their credentials.
3. **Step 3 (Client → Auth Server):**
    - The client application sends a request to the Authorization Server with the user's consent.
    - The request includes the **Client ID**, **Redirect URI**, and the **requested scope**.
4. **Step 4 (Auth Server → Client):**
    - The Authorization Server validates the user's credentials and their consent.
    - If valid, it issues an **Authorization Code** to the client application.
5. **Step 5 (Client → Auth Server):**
    - The client application exchanges the Authorization Code for an **Access Token**.
    - The request includes the **Authorization Code**, **Client ID**, **Client Secret**, and other required details.
6. **Step 6 (Auth Server → Client):**
    - The Authorization Server issues an **Access Token** (and optionally a Refresh Token) to the client application.
7. **Step 7 (Client → Resource Server):**
    - The client uses the Access Token to make an API request to the Resource Server to access the user's resources.
8. **Step 8 (Resource Server → Client):**
    - The Resource Server validates the Access Token with the Authorization Server.
    - If valid, it provides the requested resource data to the client.

Part 2: Details of Steps and Parameters

Steps 2 & 3: Request to Authorization Server

The client application sends the following key parameters:

- **client_id:** Identifies the client application registered with the Authorization Server.
- **redirect_uri:** The URI to which the Authorization Server redirects after successful authentication (optional).
- **scope:** Specifies the level of access being requested (e.g., READ).

- **state:** A CSRF protection token for added security.
- **response_type:** Specifies that the request is for an Authorization Code (`` `code` ``).

Step 5: Exchange Authorization Code for Access Token

The client sends the following to the Authorization Server:

- **code:** The Authorization Code received earlier.
- **client_id & client_secret:** The credentials of the client application.
- **grant_type:** Specifies the type of OAuth2 flow (`` `authorization_code` ``).
- **redirect_uri:** Ensures the code is valid for this URI.

Part 3: Security Design

- The flow involves two requests to the Authorization Server:
  1. To obtain the Authorization Code.
  2. To exchange the Authorization Code for an Access Token.
- This design ensures:
  - The **user's credentials** are never shared with the client application.
  - Only the **Authorization Server** interacts directly with the user's credentials.

Why Two Requests?

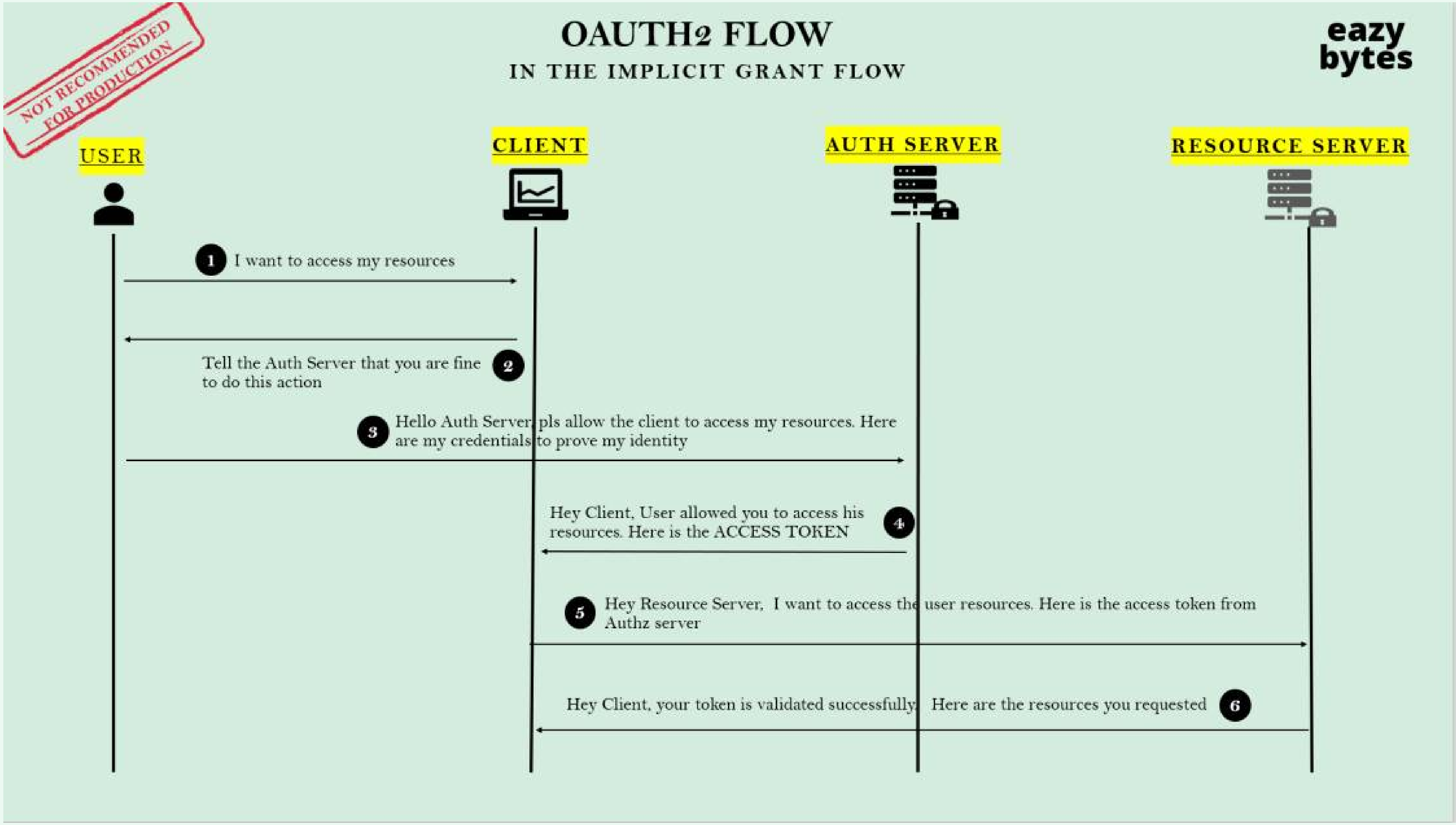
- The Authorization Code is short-lived and used only once.
- The second request requires the **Client Secret**, ensuring only authenticated clients can exchange the code.

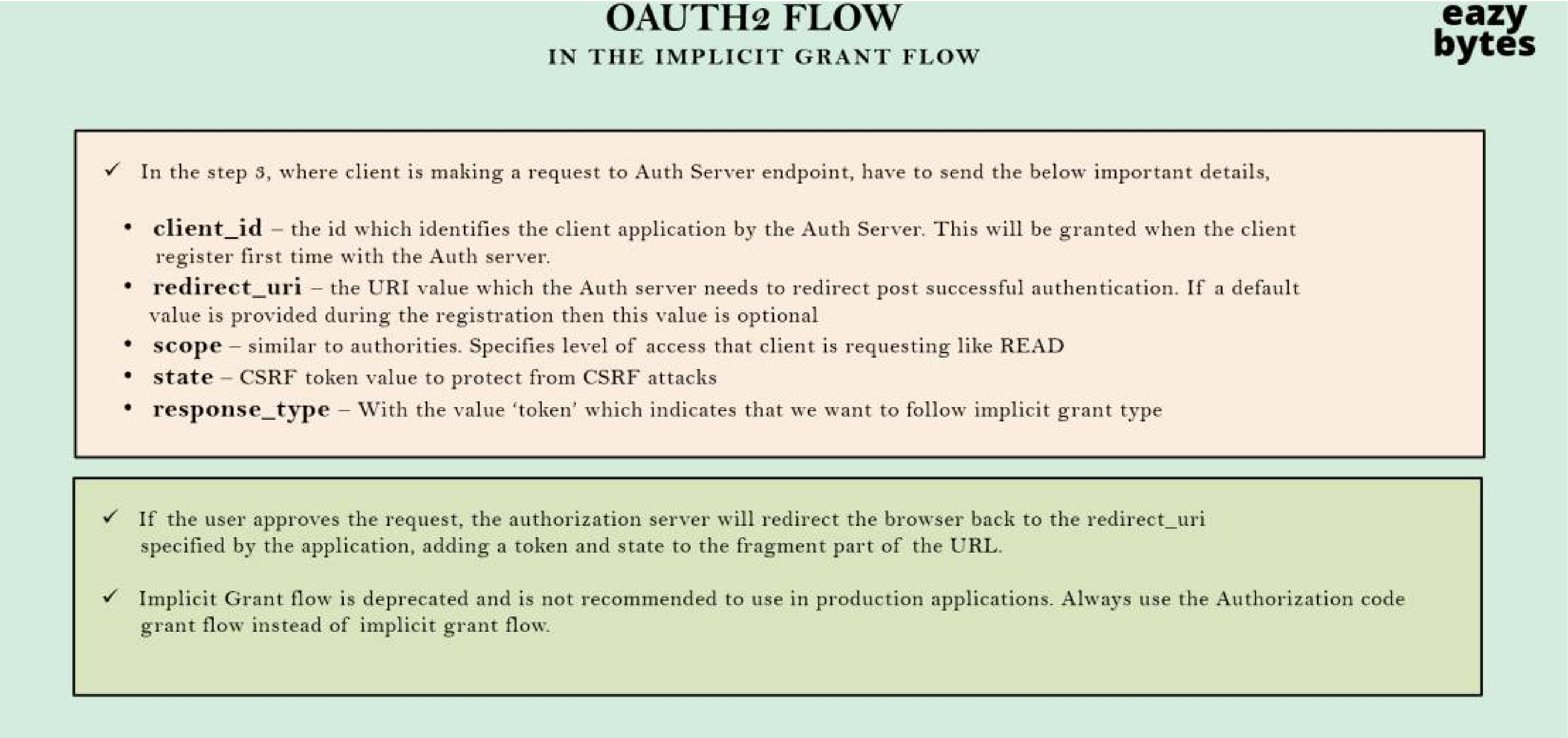
Part 4: Implicit Grant (Legacy)

- An alternative flow called the **Implicit Grant Type** combines the two steps into one.
- However, it is less secure because the Access Token is directly issued to the client, increasing exposure to interception.

Summary

The **Authorization Code Grant Type** is a secure and widely-used OAuth2 flow for server-side applications. It separates user authentication and token issuance, minimizing security risks and ensuring a reliable mechanism for accessing user resources.


NOT RECOMMENDED FOR PRODUCTION
OAUTH2 FLOW
IN THE IMPLICIT GRANT FLOW
eazy bytes
USER
CLIENT
AUTH SERVER
RESOURCE SERVER
1 I want to access my resources
Tell the Auth Server that you are fine to do this action 2
3 Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity
Hey Client, User allowed you to access his resources. Here is the ACCESS TOKEN 4
5 Hey Resource Server, I want to access the user resources. Here is the access token from Authz server
Hey Client, your token is validated successfully. Here are the resources you requested 6



eazy bytes

OAUTH2 FLOW

IN THE IMPLICIT GRANT FLOW

✓ In the step 3, where client is making a request to Auth Server endpoint, have to 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 register 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** – similar to 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

✓ If the user approves the request, the authorization server will redirect the browser back to the redirect_uri specified by the application, adding a token and state to the fragment part of the URL.

✓ Implicit Grant flow is deprecated and is not recommended to use in production applications. Always use the Authorization code grant flow instead of implicit grant flow.

Kindly explain the above image with detailed explanation

The images illustrate the **Implicit Grant Flow** in OAuth 2.0, which is a simplified method for obtaining an access token but is not recommended for production due to security concerns. Here's the detailed explanation:

Part 1: Implicit Grant Flow Sequence

1. **Step 1 (User):**

  - The user (resource owner) expresses the intention to access their resources using a client application.
2. **Step 2 (User → Authorization Server):**
  - The user gives their consent to allow the client application to access their resources.
  - The client application redirects the user to the **Authorization Server's login page**, where the user enters their credentials.
3. **Step 3 (Client → Authorization Server):**
  - The client application sends a request to the Authorization Server with:
    - **client_id**: Identifies the client application.
    - **redirect_uri**: The URI to which the Authorization Server redirects after successful authentication.
    - **scope**: Specifies the level of access the client is requesting (e.g., READ).
    - **state**: A CSRF token for added security.
    - **response_type**: Specifies **`token`**, indicating that the client is following the implicit grant flow.
4. **Step 4 (Authorization Server → Client):**
  - After validating the user's credentials and consent, the Authorization Server directly issues an **Access Token** to the client application.
  - The Access Token is returned as part of the URL fragment in the redirection to the **redirect_uri**.
5. **Step 5 (Client → Resource Server):**
  - The client uses the Access Token to make a request to the Resource Server for accessing the user's resources.
6. **Step 6 (Resource Server → Client):**
  - The Resource Server validates the Access Token with the Authorization Server.
  - If the token is valid, the requested resources are sent back to the client application.

Part 2: Key Details and Parameters in Step 3

When the client sends a request to the Authorization Server, it must include:

1. **client_id**:
  - The unique identifier for the client application, issued during registration with the Authorization Server.
2. **redirect_uri**:
  - The URI where the Authorization Server redirects after successful authentication.
  - Optional if a default value is configured during registration.
3. **scope**:
  - Defines the level of access requested by the client application (e.g., READ or WRITE access).

4. **state**:
    - A CSRF protection token to ensure that the response corresponds to the initial request.
5. **response_type**:
    - The value is `` `token` ``, indicating the client expects an Access Token as a response.

Part 3: Security Concerns of Implicit Grant Flow

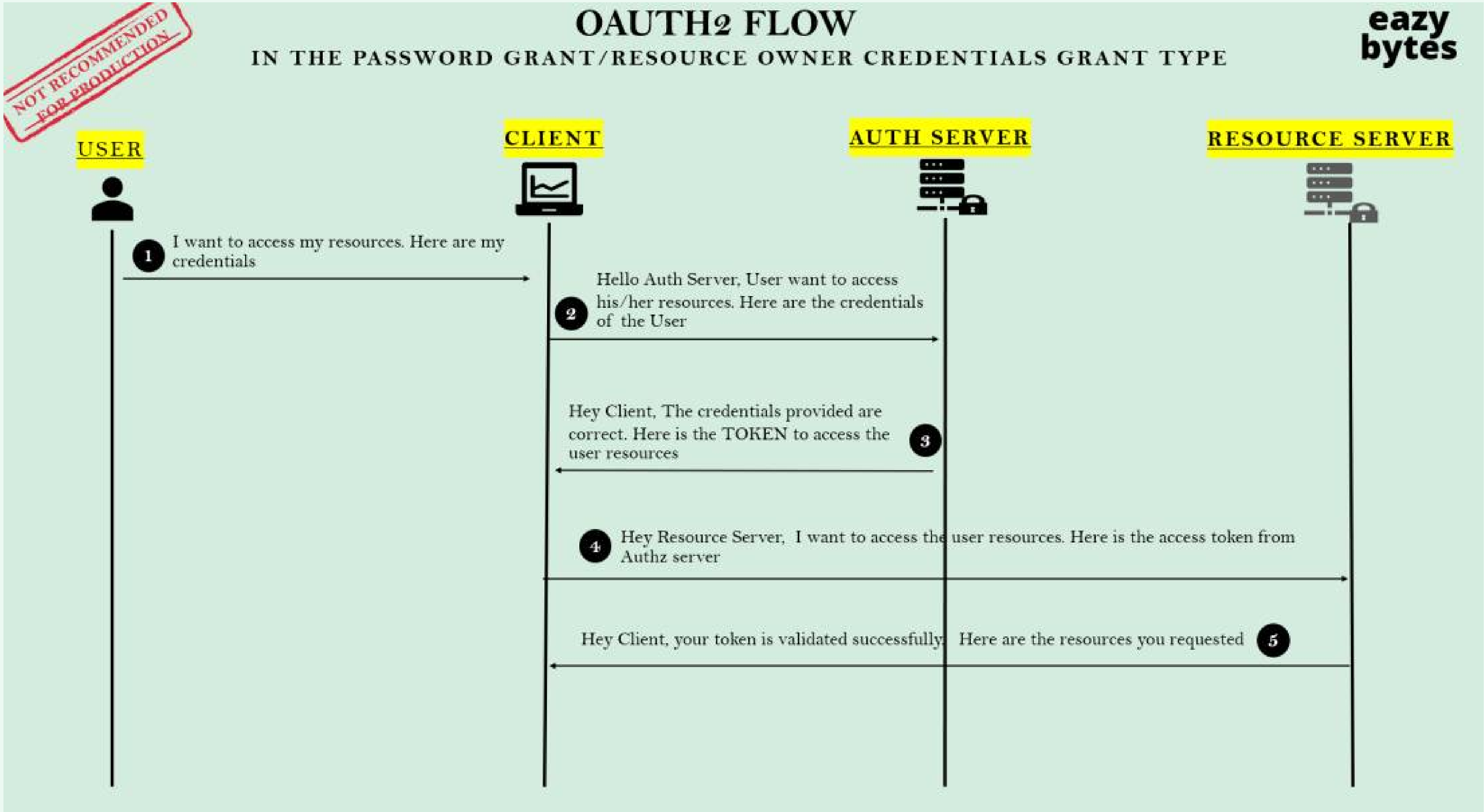
1. **Access Token in URL Fragment:**
    - The Access Token is returned in the URL fragment, making it visible to the browser and potentially accessible to malicious scripts or intermediaries.
2. **No Refresh Token:**
    - Implicit Grant does not provide a Refresh Token, meaning the user must reauthenticate when the Access Token expires.
3. **Vulnerability to Interception:**
    - The Access Token is exposed in the user's browser, increasing the risk of token theft.

Part 4: Deprecation and Recommendations

- **Implicit Grant Flow is Deprecated:**
    - Due to its security weaknesses, the Implicit Grant Flow is no longer recommended for production applications.
- **Use Authorization Code Flow:**
    - For better security, use the **Authorization Code Grant Flow** with Proof Key for Code Exchange (PKCE) to ensure secure token handling.

Summary

The Implicit Grant Flow simplifies the process by directly issuing an Access Token to the client but sacrifices security in the process. It is suitable for limited scenarios like single-page applications but should generally be avoided in favor of more secure alternatives like the Authorization Code Grant Flow.



OAUTH2 FLOW

IN THE PASSWORD GRANT/RESOURCE OWNER CREDENTIALS GRANT TYPE

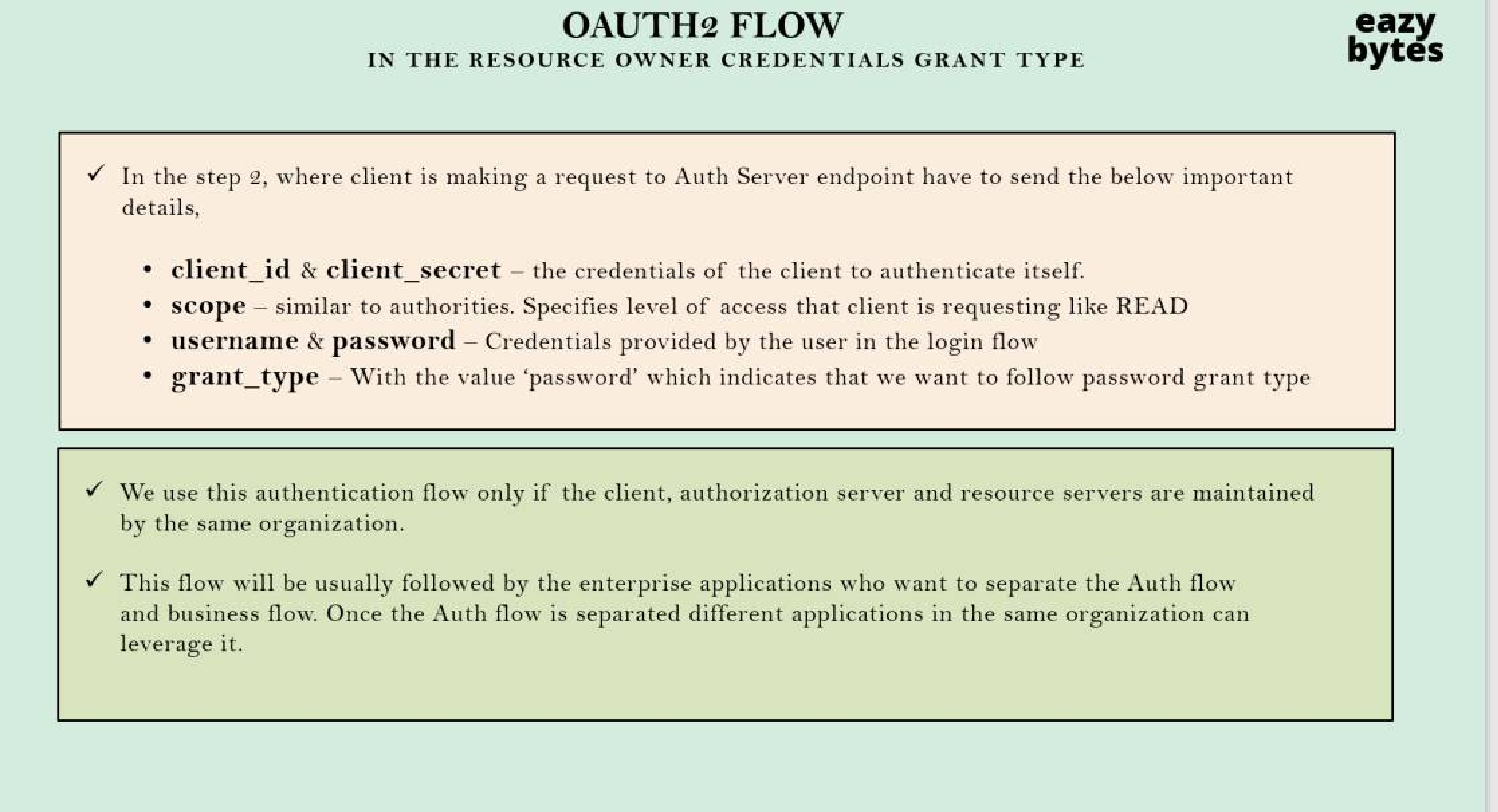
1 I want to access my resources. Here are my credentials

2 Hello Auth Server, User want to access his/her resources. Here are the credentials of the User

3 Hey Client, The credentials provided are correct. Here is the TOKEN to access the user resources

4 Hey Resource Server, I want to access the user resources. Here is the access token from Authz server

5 Hey Client, your token is validated successfully. Here are the resources you requested



OAUTH2 FLOW

IN THE RESOURCE OWNER CREDENTIALS GRANT TYPE

✓ In the step 2, where client is making a request to Auth Server endpoint have to send the below important details,

- **client_id** & **client_secret** – the credentials of the client to authenticate itself.
- **scope** – similar to authorities. Specifies level of access that client is requesting like READ
- **username** & **password** – Credentials provided by the user in the login flow
- **grant_type** – With the value 'password' which indicates that we want to follow password grant type

✓ We use this authentication flow only if the client, authorization server and resource servers are maintained by the same organization.

✓ This flow will be usually followed by the enterprise applications who want to separate the Auth flow and business flow. Once the Auth flow is separated different applications in the same organization can leverage it.

kindly explain the above image with detailed explanation

The images describe the **OAuth2 Password Grant Flow** (or Resource Owner Credentials Grant), a flow that allows the client application to directly handle the user's credentials. This flow is deprecated and **not recommended for production** due to significant security concerns.

Part 1: Sequence of Password Grant Flow

1. **Step 1 (User):**
   - The user (resource owner) provides their credentials (username and password) to the client application, intending to access their resources.
2. **Step 2 (Client → Authorization Server):**
   - The client sends the user's credentials to the Authorization Server, along with:
     - **client_id** and **client_secret** (to authenticate the client itself).
     - **scope** (the requested access permissions, such as READ).
     - **grant_type** (`` `password` ``) to specify this flow.
3. **Step 3 (Authorization Server → Client):**
   - The Authorization Server validates the user's credentials.
   - If valid, it issues an **Access Token** (and optionally a Refresh Token) to the client.
4. **Step 4 (Client → Resource Server):**
   - The client uses the Access Token to request resources from the Resource Server.
5. **Step 5 (Resource Server → Client):**
   - The Resource Server validates the Access Token with the Authorization Server.
   - If the token is valid, the Resource Server returns the requested resources to the client.

Part 2: Parameters Sent in Step 2

When the client sends the user's credentials to the Authorization Server, it includes:

1. **client_id and client_secret:**
   - Identifies and authenticates the client application.
2. **scope:**
   - Specifies the level of access requested by the client (e.g., READ or WRITE).
3. **username and password:**
   - The user’s credentials, provided by the resource owner.
4. **grant_type:**
   - Specifies the flow being used; in this case, `` `password` ``.

Part 3: Use Cases and Restrictions

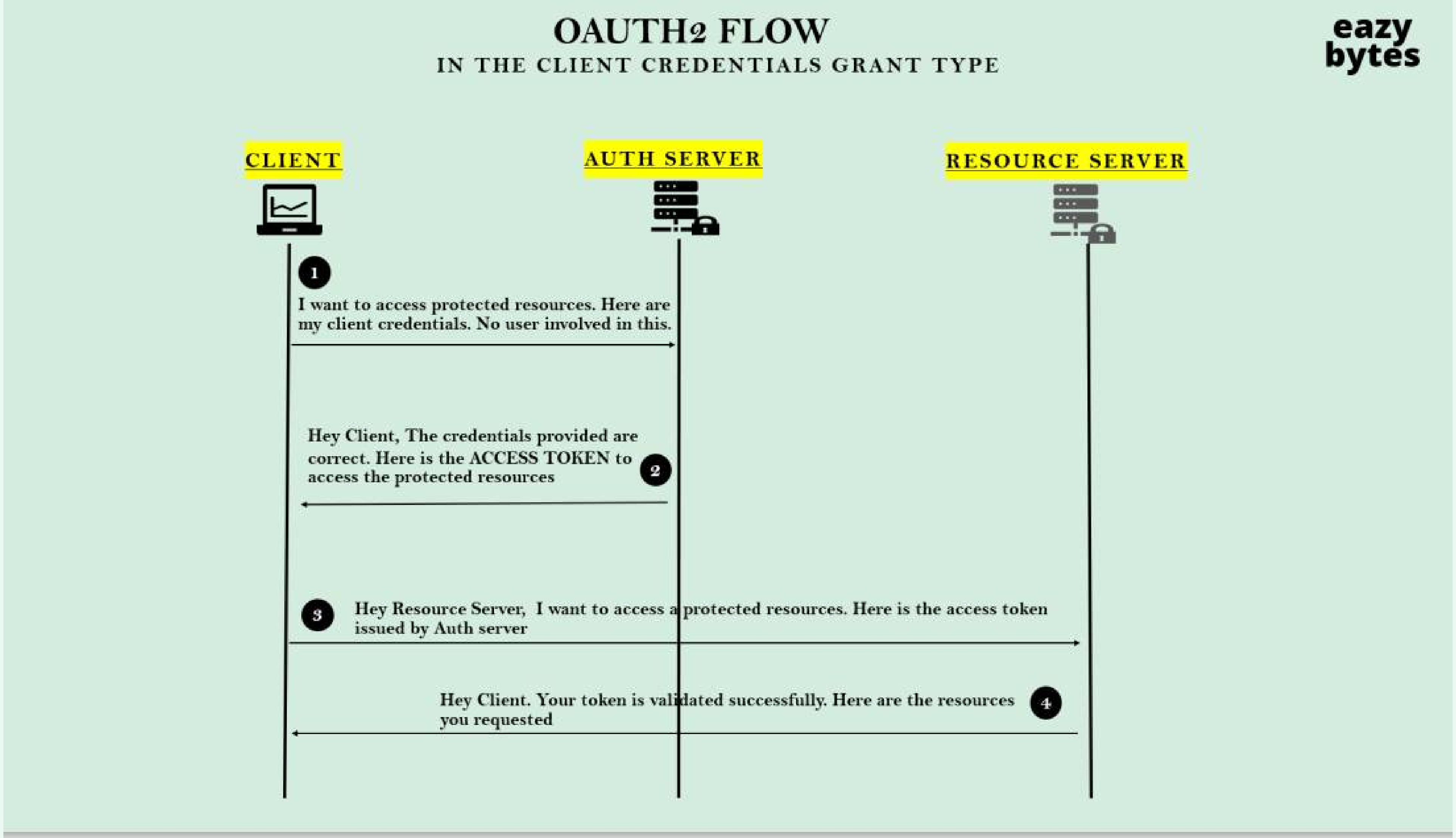
1. **When to Use This Flow:**
    - Only when the client, Authorization Server, and Resource Server are all maintained by the same organization.
    - Suitable for internal enterprise applications where trust can be established.
2. **Why It's Deprecated:**
    - The user's credentials are exposed to the client application, violating the principle of minimal trust.
    - If the client application is compromised, the user's credentials can be stolen.
    - Does not support external third-party applications securely.

Part 4: Enterprise Use and Alternatives

- **Internal Enterprise Use:**
    - This flow may be used for enterprise applications where the client and server are part of the same trusted infrastructure.
- **Recommended Alternative:**
    - Use the **Authorization Code Flow** with PKCE (Proof Key for Code Exchange) for better security and separation of credentials.

Summary

The **Password Grant Flow** directly uses the user's credentials to obtain an Access Token, making it inherently less secure. It is deprecated and should only be used in tightly controlled environments. For modern applications, other flows like the Authorization Code Flow are recommended to ensure security and scalability.



OAUTH2 FLOW

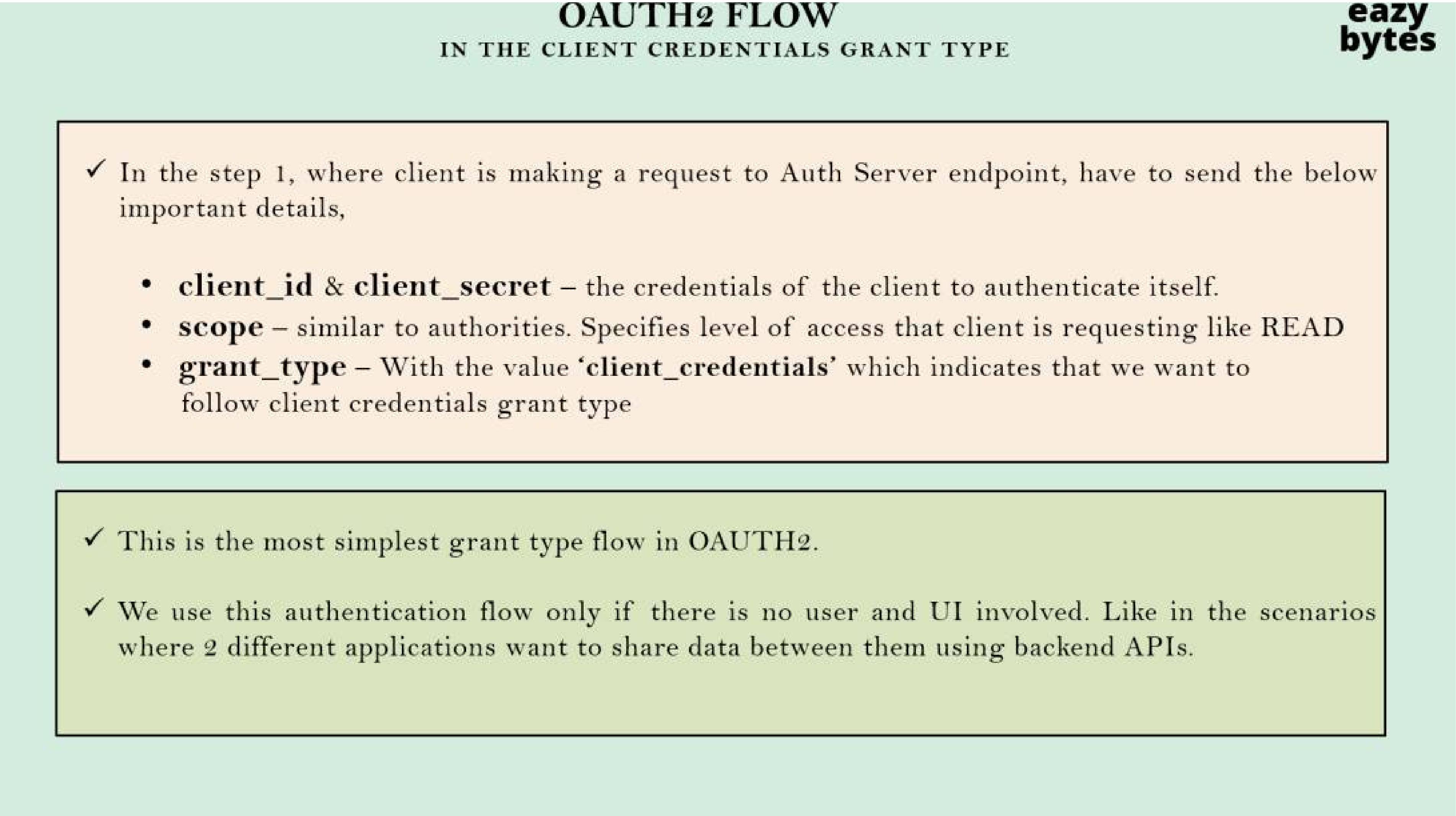
IN THE CLIENT CREDENTIALS GRANT TYPE

1 I want to access protected resources. Here are my client credentials. No user involved in this.

2 Hey Client, The credentials provided are correct. Here is the ACCESS TOKEN to access the protected resources

3 Hey Resource Server, I want to access a protected resources. Here is the access token issued by Auth server

4 Hey Client. Your token is validated successfully. Here are the resources you requested



OAUTH2 FLOW

IN THE CLIENT CREDENTIALS GRANT TYPE

✓ In the step 1, where client is making a request to Auth Server endpoint, have to send the below important details,

- **client_id** & **client_secret** – the credentials of the client to authenticate itself.
- **scope** – similar to authorities. Specifies level of access that client is requesting like READ
- **grant_type** – With the value **'client_credentials'** which indicates that we want to follow client credentials grant type

✓ This is the most simplest grant type flow in OAUTH2.

✓ We use this authentication flow only if there is no user and UI involved. Like in the scenarios where 2 different applications want to share data between them using backend APIs.

Kindly explain the above image with detailed explanation

The images describe the **OAuth2 Client Credentials Grant Type**, which is used for machine-to-machine communication where no user is involved. It is the simplest grant type in OAuth2 and is suitable for backend service communication. Here's a detailed explanation:

Part 1: Sequence of Client Credentials Grant Flow

1. **Step 1 (Client → Authorization Server):**
   - The client (application or service) sends a request to the Authorization Server with its **client credentials** (Client ID and Client Secret).
   - This request includes:
     - **client_id** and **client_secret**: Identifies and authenticates the client.
     - **scope**: Defines the level of access requested by the client (e.g., READ or WRITE).
     - **grant_type**: Specifies `` `client_credentials` ``, indicating the use of this flow.
2. **Step 2 (Authorization Server → Client):**
   - The Authorization Server validates the client's credentials.
   - If valid, it issues an **Access Token** to the client.
3. **Step 3 (Client → Resource Server):**
   - The client uses the Access Token to make a request to the Resource Server for the protected resources.
4. **Step 4 (Resource Server → Client):**
   - The Resource Server validates the Access Token with the Authorization Server.
   - If valid, it provides the requested resources to the client.

Part 2: Parameters in Step 1

When the client makes a request to the Authorization Server, it must include:

1. **client_id and client_secret:**
   - These are issued during the client registration process with the Authorization Server.
2. **scope:**
   - Specifies the permissions or access level requested by the client (e.g., READ or WRITE).
3. **grant_type:**
   - The value is `` `client_credentials` ``, indicating this specific OAuth2 flow.

Part 3: Use Cases and Best Practices

1. **Use Cases:**

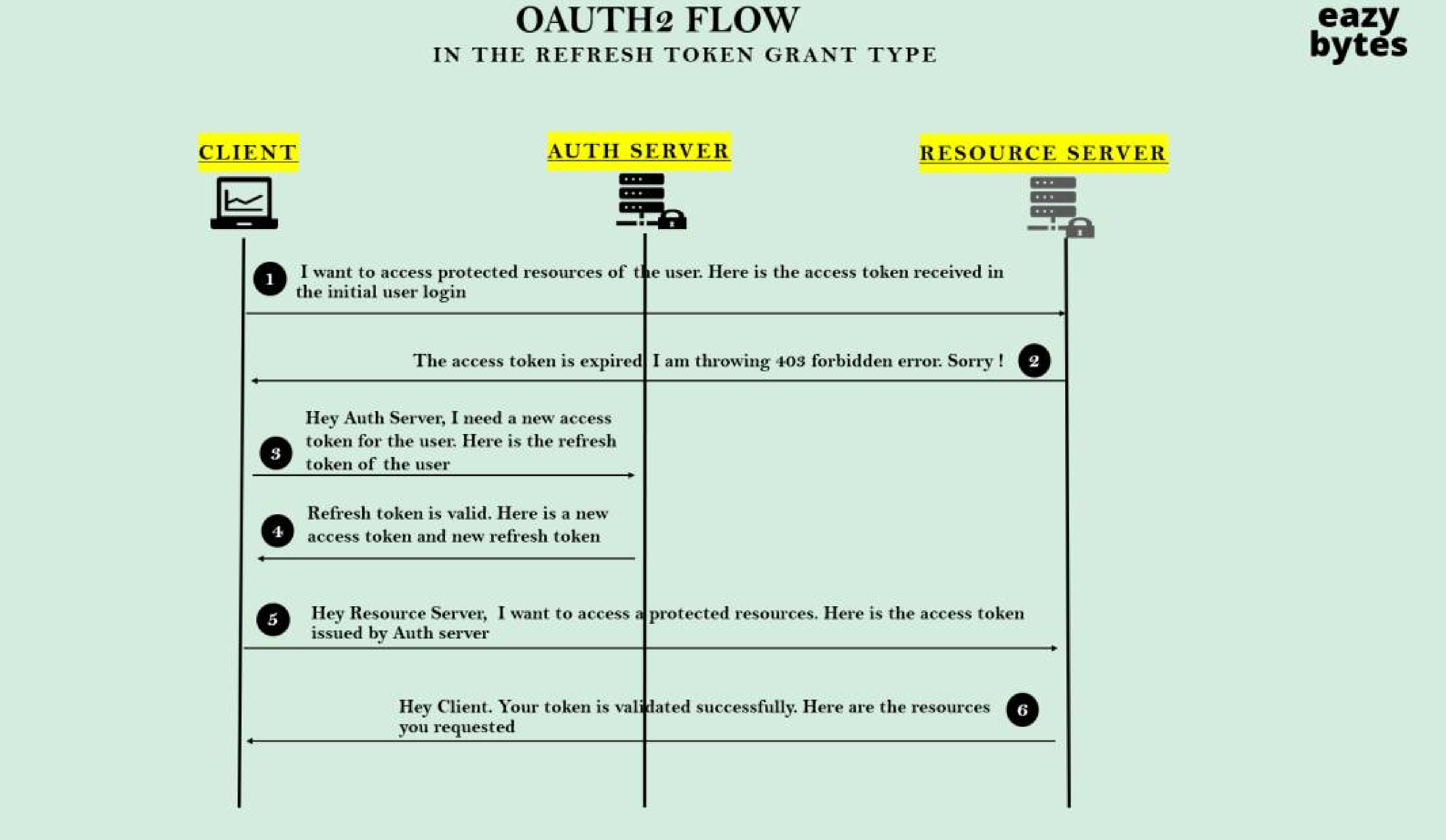
  - Machine-to-machine communication where no user interaction is required.
  - Scenarios where backend services need to share data using APIs (e.g., a payment gateway interacting with a backend API to process transactions).
2. **Best Practices:**
  - Securely store the **Client ID** and **Client Secret** to prevent unauthorized access.
  - Use HTTPS for all communication to protect the credentials and token.

Part 4: Why It's Simple and Secure

1. **No User Interaction:**
  - Unlike other OAuth2 flows, this grant type does not involve user credentials or consent.
2. **Direct Communication:**
  - The Authorization Server directly authenticates the client, issues an Access Token, and the client uses it to access the Resource Server.

Summary

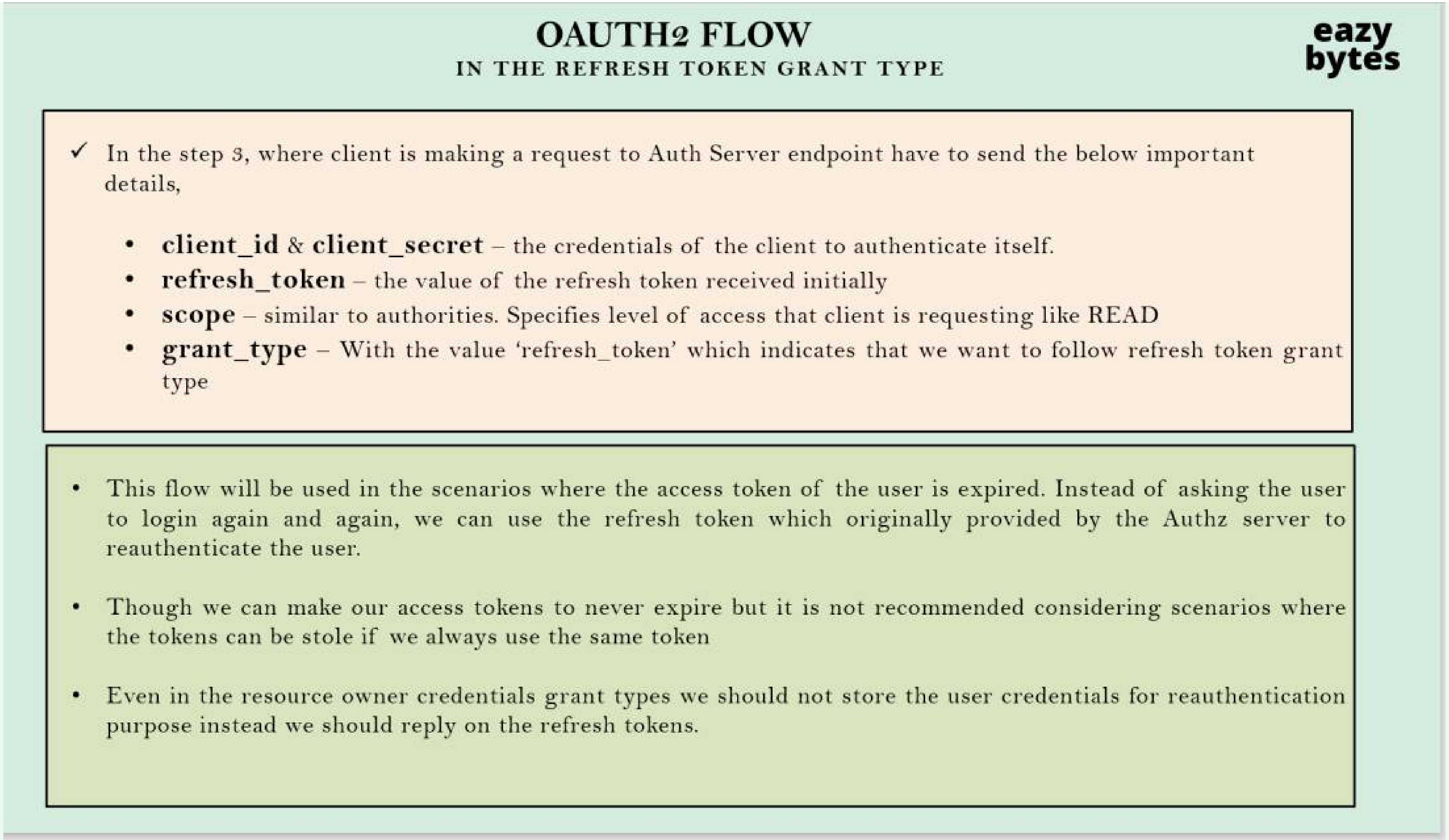
The **Client Credentials Grant Type** is ideal for service-to-service communication in trusted environments. It eliminates the need for user involvement, focusing solely on the authentication of the client application. This makes it a straightforward and efficient solution for backend API integrations. However, secure handling of client credentials is critical to maintain security.



OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE

1. I want to access protected resources of the user. Here is the access token received in the initial user login
2. The access token is expired. I am throwing 403 forbidden error. Sorry !
3. Hey Auth Server, I need a new access token for the user. Here is the refresh token of the user
4. Refresh token is valid. Here is a new access token and new refresh token
5. Hey Resource Server, I want to access a protected resources. Here is the access token issued by Auth server
6. Hey Client. Your token is validated successfully. Here are the resources you requested



OAUTH2 FLOW

IN THE REFRESH TOKEN GRANT TYPE

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

- **client_id** & **client_secret** – the credentials of the client to authenticate itself.
- **refresh_token** – the value of the refresh token received initially
- **scope** – similar to authorities. Specifies level of access that client is requesting like READ
- **grant_type** – With the value 'refresh_token' which indicates that we want to follow refresh token grant type

- This flow will be used in the scenarios where the access token of the user is expired. Instead of asking the user to login again and again, we can use the refresh token which originally provided by the Authz server to reauthenticate the user.
- Though we can make our access tokens to never expire but it is not recommended considering scenarios where the tokens can be stole if we always use the same token
- Even in the resource owner credentials grant types we should not store the user credentials for reauthentication purpose instead we should reply on the refresh tokens.

Kindly explain the above image with detailed explanation

The images describe the **OAuth2 Refresh Token Grant Type**, a mechanism used to refresh access tokens when they expire without requiring the user to log in again. Here is a detailed explanation:

Part 1: Sequence of Refresh Token Grant Flow

1. **Step 1 (Client → Resource Server):**
    - The client uses an **Access Token** received during the initial user login to access protected resources from the Resource Server.
2. **Step 2 (Resource Server → Client):**
    - If the Access Token has expired, the Resource Server returns a `` `403 Forbidden` `` error, indicating that the token is no longer valid.
3. **Step 3 (Client → Authorization Server):**
    - The client sends a request to the Authorization Server to get a new Access Token.
    - This request includes the **Refresh Token** that was issued along with the original Access Token.
4. **Step 4 (Authorization Server → Client):**
    - The Authorization Server validates the Refresh Token.
    - If valid, it issues a new **Access Token** and optionally a new **Refresh Token** to the client.
5. **Step 5 (Client → Resource Server):**
    - The client uses the new Access Token to request resources from the Resource Server.
6. **Step 6 (Resource Server → Client):**
    - The Resource Server validates the new Access Token and provides the requested resources to the client.

Part 2: Parameters in Step 3

When the client requests a new Access Token using the Refresh Token, it sends the following:

1. **client_id and client_secret:**
    - These credentials authenticate the client application with the Authorization Server.
2. **refresh_token:**
    - The Refresh Token issued during the initial authentication is used to request a new Access Token.
3. **scope:**
    - Specifies the permissions or access level requested by the client (e.g., READ or WRITE).

    - It must not exceed the scope granted in the original authentication.
4. **grant_type:**
    - The value is `` `refresh_token` ``, indicating the use of the Refresh Token Grant Type.

Part 3: Use Cases

1. **Token Expiry Management:**
    - Used when an Access Token has expired, allowing the client to obtain a new one without requiring the user to log in again.
2. **Enhanced Security:**
    - Access Tokens are short-lived, minimizing the risk of misuse if compromised.
    - The Refresh Token is used securely to generate new tokens as needed.

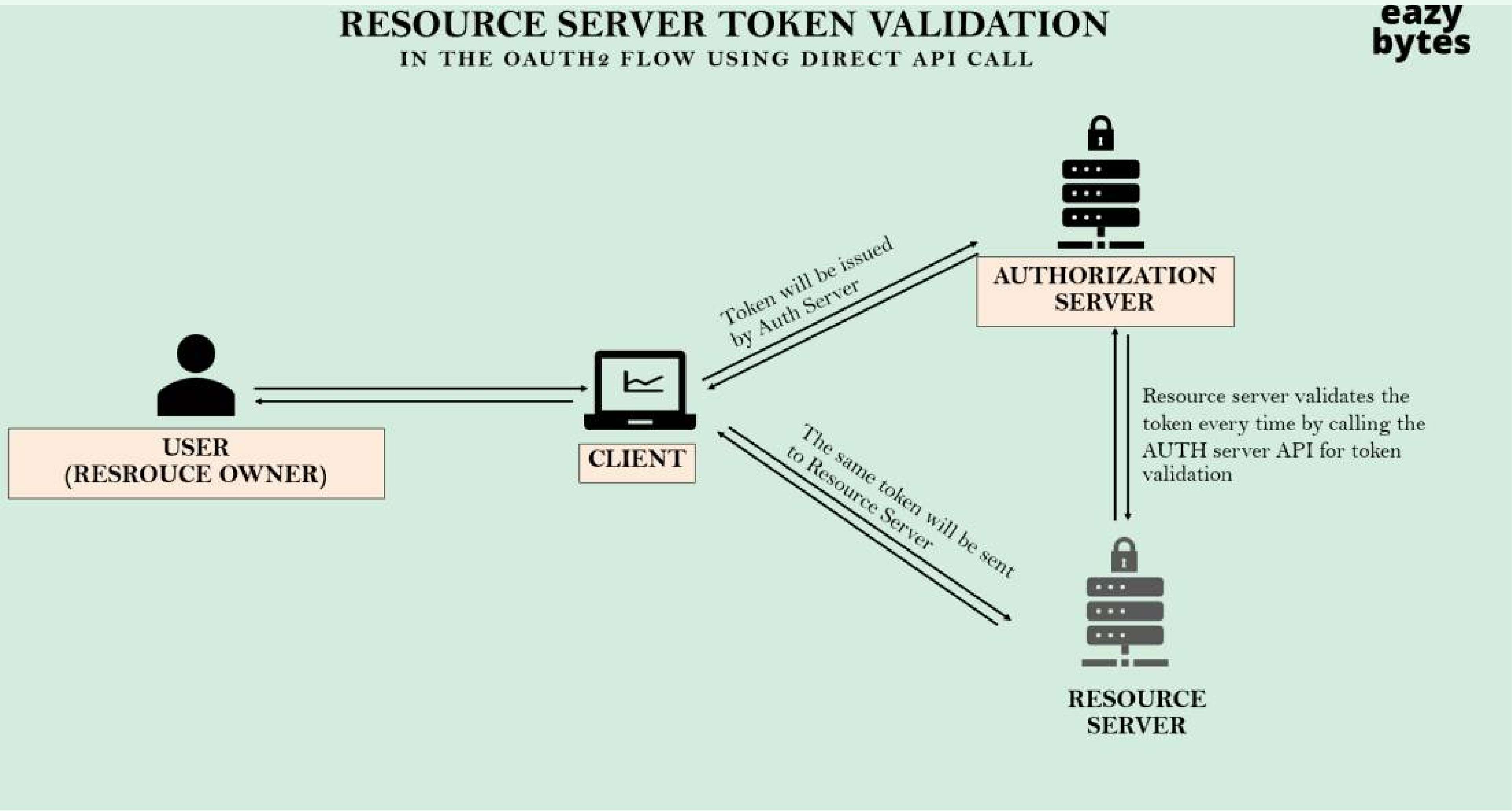
Part 4: Best Practices and Considerations

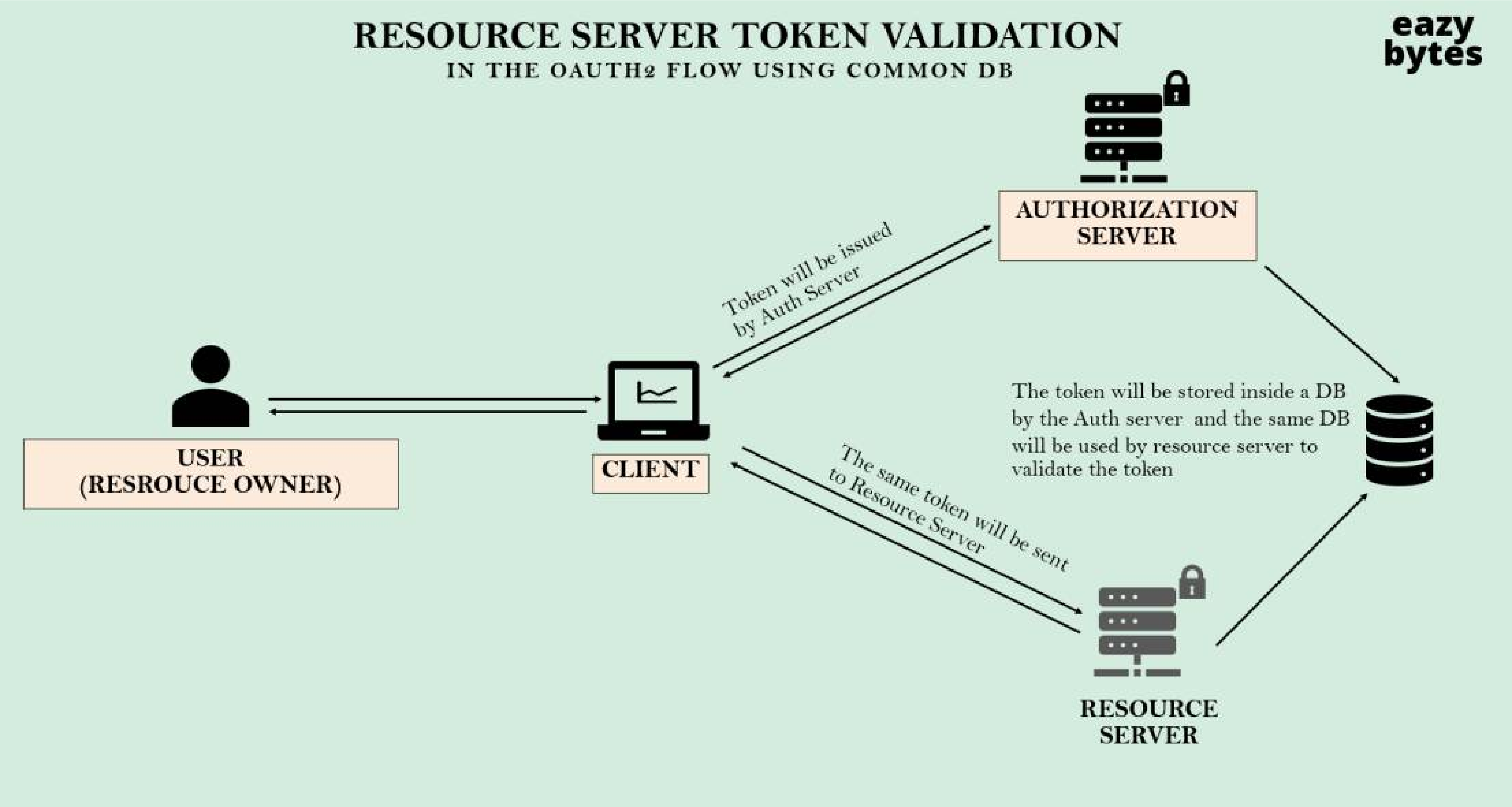
1. **Short-Lived Access Tokens:**
    - Access Tokens should have a short lifespan to reduce the risk of being compromised.
2. **Secure Storage of Refresh Tokens:**
    - Refresh Tokens should be securely stored, as they provide access to obtaining new Access Tokens.
3. **Avoid Infinite Lifetimes:**
    - Avoid making tokens that never expire, as they increase the risk of misuse if stolen.
4. **Use in Trusted Applications:**
    - Refresh Token flow is suitable for applications where the client and server are trusted, such as native mobile apps or web applications.
5. **Do Not Store User Credentials:**
    - Always use Refresh Tokens for reauthentication instead of storing user credentials.

Summary

The **Refresh Token Grant Type** is a secure and user-friendly mechanism to refresh Access Tokens without requiring repeated user authentication. It enhances security by allowing short-lived Access Tokens and reduces friction for users by maintaining seamless access to resources. Proper handling of

Refresh Tokens is critical to maintaining the overall security of the application.


RESOURCE SERVER TOKEN VALIDATION
IN THE OAUTH2 FLOW USING DIRECT API CALL
eazy bytes
USER
(RESROUCE OWNER)
CLIENT
Token will be issued by Auth Server
AUTHORIZATION SERVER
Resource server validates the token every time by calling the AUTH server API for token validation
The same token will be sent to Resource Server
RESOURCE SERVER

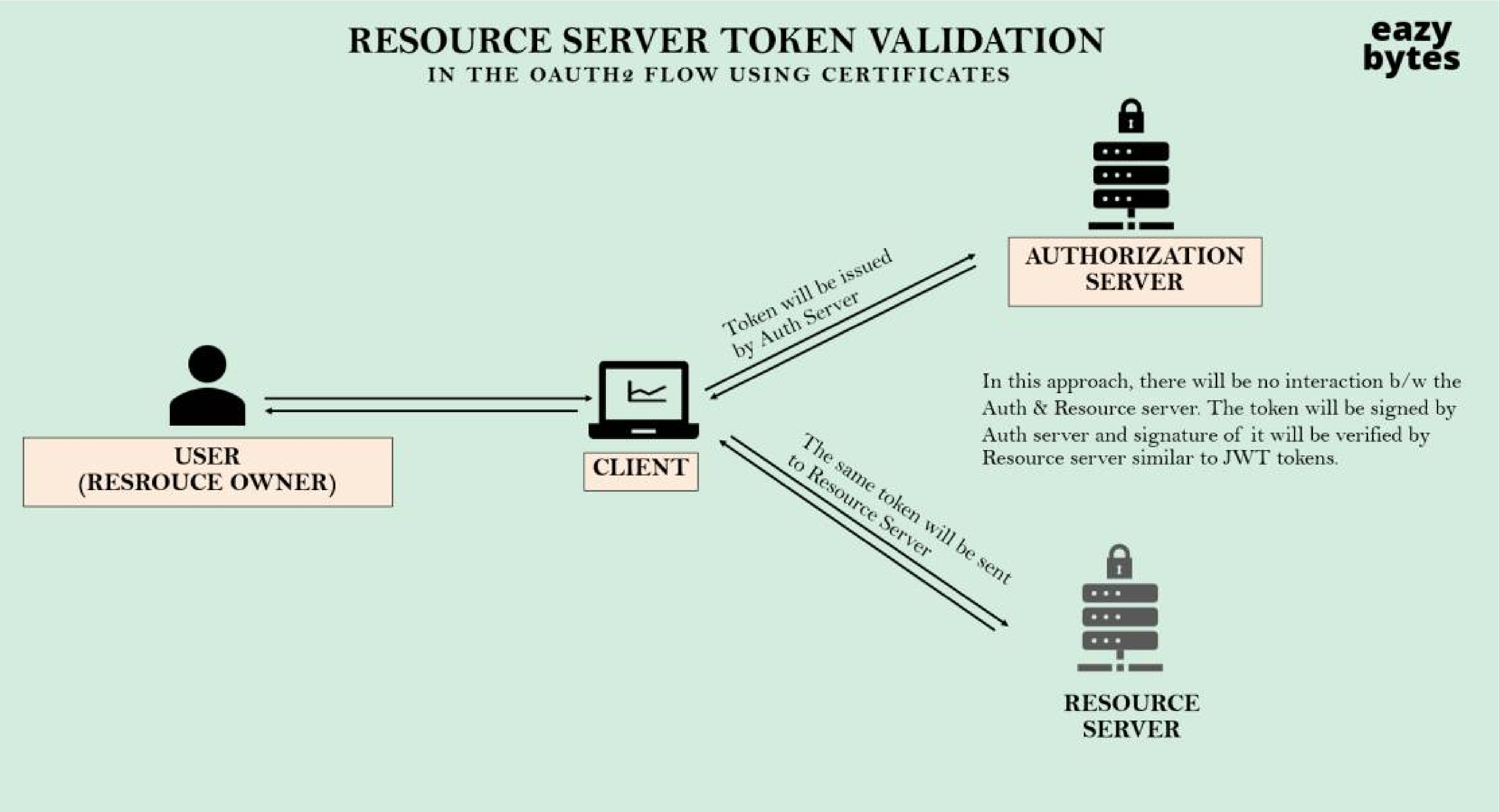


RESOURCE SERVER TOKEN VALIDATION
IN THE OAUTH2 FLOW USING COMMON DB

Token will be issued by Auth Server

The same token will be sent to Resource Server

The token will be stored inside a DB by the Auth server and the same DB will be used by resource server to validate the token


RESOURCE SERVER TOKEN VALIDATION
IN THE OAUTH2 FLOW USING CERTIFICATES
eazy bytes
AUTHORIZATION SERVER
Token will be issued by Auth Server
USER (RESROUCE OWNER)
CLIENT
The same token will be sent to Resource Server
RESOURCE SERVER

In this approach, there will be no interaction b/w the Auth & Resource server. The token will be signed by Auth server and signature of it will be verified by Resource server similar to JWT tokens.

Kindly explain the above image with detailed explanation

The images explain various strategies for **token validation** by the Resource Server in the OAuth2 flow. Token validation ensures that the access tokens issued by the Authorization Server are legitimate and not tampered with. Below is a detailed explanation of each approach:

---

1. Direct API Call to Authorization Server

Process:

- The **Client** obtains an access token from the **Authorization Server** after authentication.
- When the **Resource Server** receives a request from the client, it:
    1. Sends the access token to the Authorization Server for validation using a direct API call.
    2. The Authorization Server verifies the token and confirms its validity.
    3. If valid, the Resource Server grants access to the protected resource.

Advantages:

- **Centralized Validation**: The Authorization Server handles all token validation, ensuring consistency.
- **Dynamic Token Management**: Revocation or updates to tokens are instantly reflected since the Authorization Server performs real-time validation.

Disadvantages:

- **Performance Overhead**: Each request requires a round-trip to the Authorization Server, adding latency.
- **Dependency on Authorization Server Availability**: The Resource Server depends on the Authorization Server to be operational at all times.

---

2. Token Validation Using a Shared Database

Process:

- The Authorization Server stores tokens in a shared database when they are issued.
- The Resource Server:
    1. Checks the validity of the token by querying the shared database directly.
    2. Grants or denies access based on the token's status in the database.

Advantages:

- **Reduced API Calls**: The Resource Server does not need to interact with the Authorization Server for every validation.
- **Scalability**: Ideal for environments with high traffic, as it reduces the load on the Authorization Server.

Disadvantages:

- **Database Synchronization**: Ensuring that the database is up-to-date and secure is crucial.
- **Increased Complexity**: Requires a robust and shared database infrastructure between the Authorization Server and Resource Server.

---

3. Token Validation Using Certificates (e.g., JWT Tokens)

Process:

- The Authorization Server issues self-contained tokens (e.g., JWT) that are signed using a private key or certificate.
- The Resource Server:
  1. Verifies the token's signature using the public key or certificate.
  2. Validates the token without needing to call the Authorization Server.

Advantages:

- **No Dependency on Authorization Server**: The Resource Server can validate tokens independently.
- **High Performance**: Eliminates the need for API calls or database queries, reducing latency.
- **Scalable**: Suitable for distributed systems where multiple Resource Servers handle validation.

Disadvantages:

- **Token Revocation Complexity**: Since tokens are self-contained, revoking them requires additional mechanisms (e.g., token blacklists or short-lived tokens).
- **Token Size**: Self-contained tokens (e.g., JWTs) can be large, potentially increasing network overhead.

Comparison of Approaches

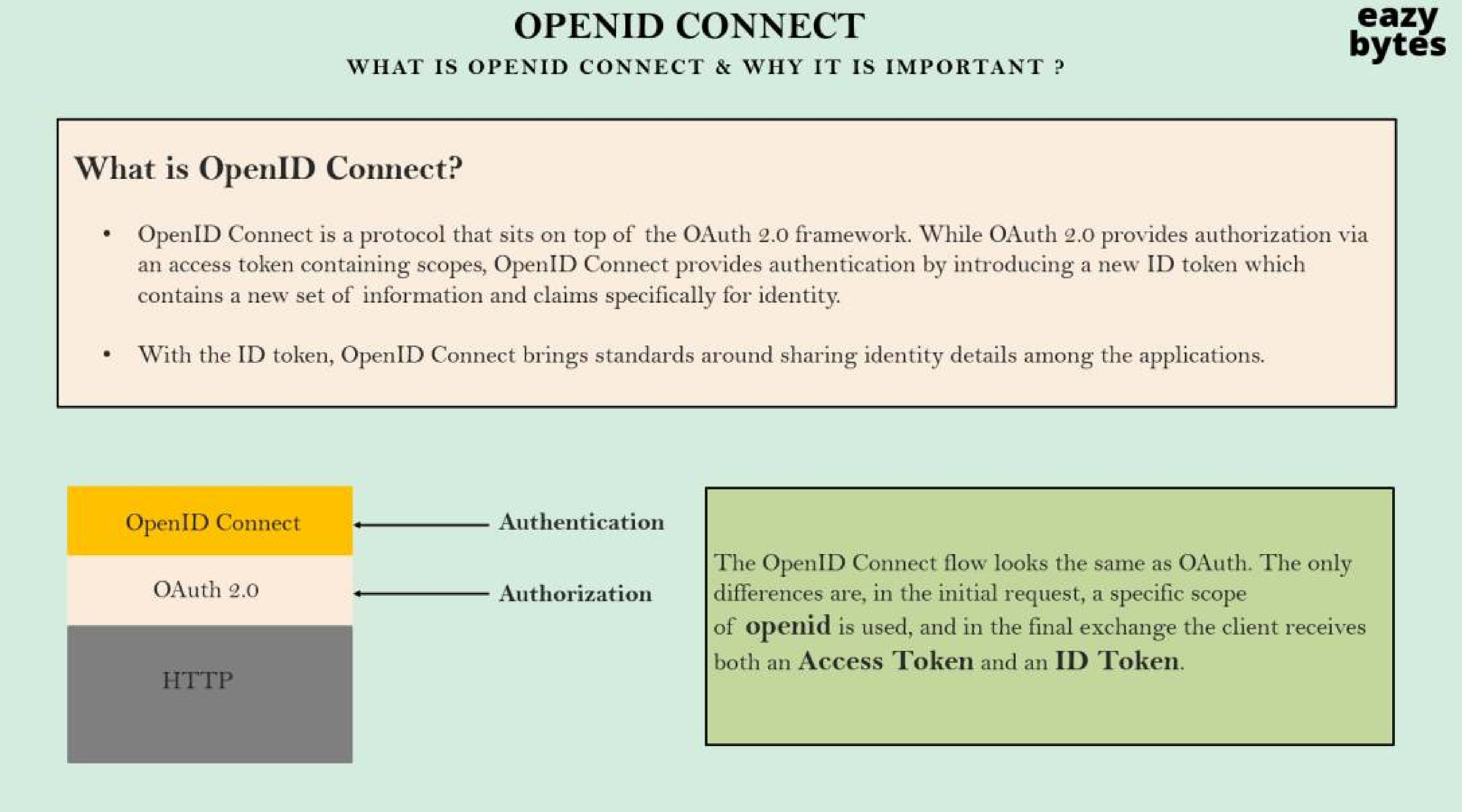
Feature	Direct API Call	Shared Database	Certificates (JWT)
Performance	Moderate (due to API calls)	Better than API calls	Best (independent validation)
Scalability	Limited by API performance	Requires scalable DB	Highly scalable
Token Revocation Handling	Real-time	Near real-time	Requires additional mechanisms

Feature	Direct API Call	Shared Database	Certificates (JWT)
Implementation Complexity	Simple	Moderate	Complex
Dependency	Authorization Server	Shared DB	Public Key/Certificate

---

Summary

Each approach has its trade-offs. The choice depends on the application's architecture, performance requirements, and the level of trust between components. For high-performance and scalable systems, **token validation using certificates** (e.g., JWTs) is often preferred. However, for dynamic token revocation, **direct API calls** or **shared databases** may be more suitable.



OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

What is OpenID Connect?

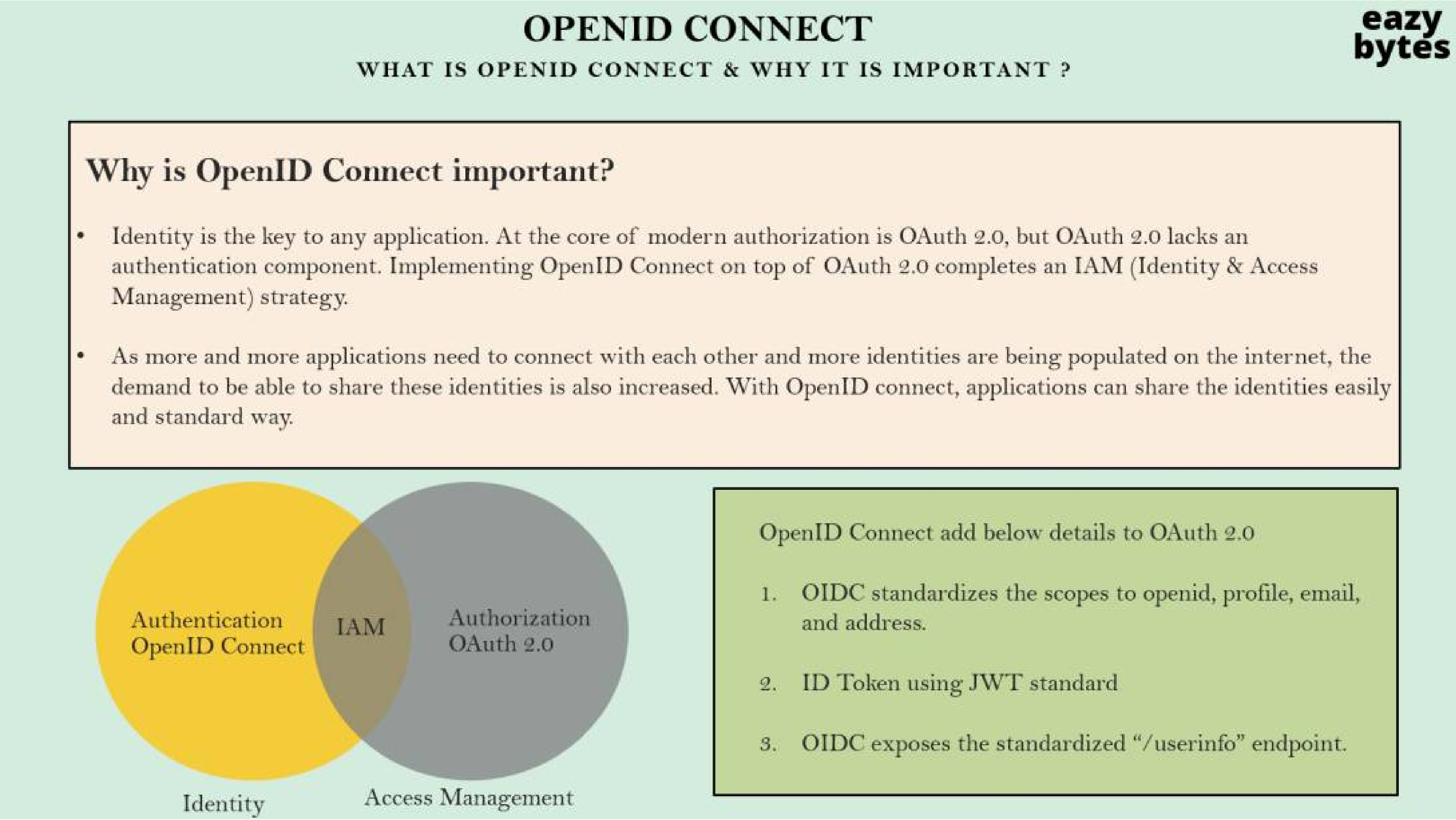
- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of information and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.

OpenID Connect ← **Authentication**

OAuth 2.0 ← **Authorization**

HTTP

The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of **openid** is used, and in the final exchange the client receives both an **Access Token** and an **ID Token**.



OPENID CONNECT

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

Why is OpenID Connect important?

- Identity is the key to any application. At the core of modern authorization is OAuth 2.0, but OAuth 2.0 lacks an authentication component. Implementing OpenID Connect on top of OAuth 2.0 completes an IAM (Identity & Access Management) strategy.
- As more and more applications need to connect with each other and more identities are being populated on the internet, the demand to be able to share these identities is also increased. With OpenID connect, applications can share the identities easily and standard way.

OpenID Connect add below details to OAuth 2.0

1. OIDC standardizes the scopes to openid, profile, email, and address.
2. ID Token using JWT standard
3. OIDC exposes the standardized "/userinfo" endpoint.

Kindly explain the above image with detailed explanation

The images describe **OpenID Connect (OIDC)**, its purpose, and its significance in modern authentication and identity management systems. OpenID Connect is built on top of OAuth 2.0 and focuses on authentication, complementing OAuth's authorization capabilities.

What is OpenID Connect?

1. **Definition:**
    - OpenID Connect is a protocol layered on top of the **OAuth 2.0 framework**.
    - While OAuth 2.0 is designed for **authorization** (granting access to resources), OpenID Connect extends it to include **authentication**, providing a standard way to verify the identity of a user.
2. **Key Features:**
    - **ID Token:** Introduces an ID Token, which contains identity information about the authenticated user (e.g., name, email, unique ID).
    - Enables secure sharing of user identity between applications.
3. **Integration with OAuth 2.0:**
    - OAuth 2.0 is used for resource access (authorization).
    - OpenID Connect is used for verifying the user's identity (authentication).

Why is OpenID Connect Important?

1. **Identity Management:**
    - Modern applications rely heavily on managing user identity securely.
    - OpenID Connect provides a **standardized and interoperable solution** for authentication.
2. **Identity Sharing Across Applications:**
    - As applications increasingly integrate with each other, sharing user identities across systems is crucial.
    - OpenID Connect simplifies identity sharing by standardizing the process.
3. **IAM (Identity and Access Management):**
    - Combines the identity capabilities of OpenID Connect with the access management capabilities of OAuth 2.0.
    - Ensures both authentication (verifying who the user is) and authorization (what the user can access).

Key Additions of OpenID Connect to OAuth 2.0

1. **Standardized Scopes:**
    - Adds specific scopes for identity, such as:
        - `` `openid` ``: Required scope to indicate OpenID Connect usage.
        - `` `profile` ``: Access to basic profile information.

        - `email`: Access to the user's email address.
        - `address`: Access to the user's address.
2. **ID Token:**
    - Issued as part of the authentication process.
    - Based on the **JWT (JSON Web Token)** standard, ensuring secure and self-contained identity data.
3. **Standardized Endpoints:**
    - Adds the `/userinfo` endpoint to fetch additional user information.

How Does OpenID Connect Work?

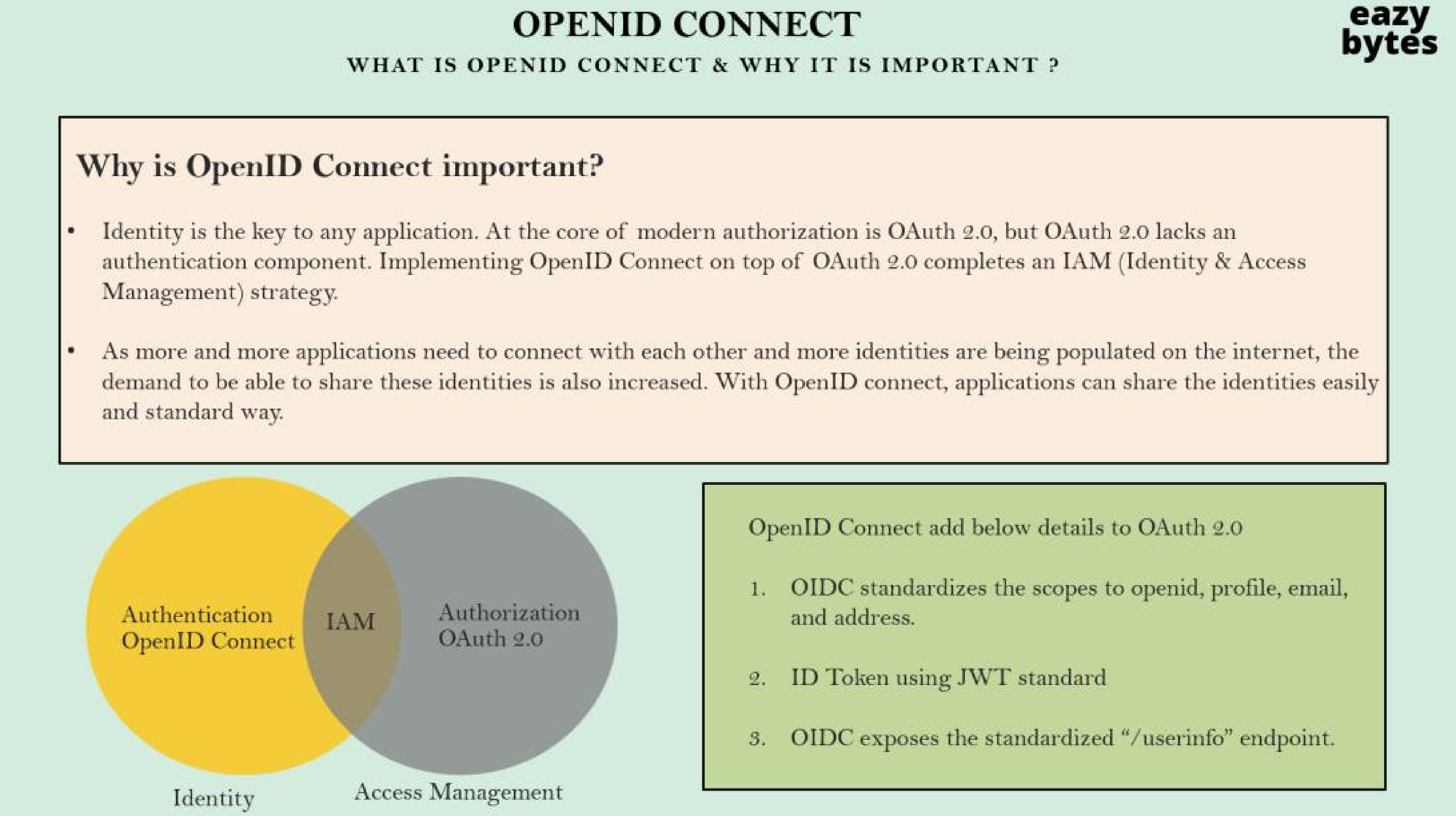
1. **Initial Request:**
    - The client requests authentication, specifying the `openid` scope.
2. **Response:**
    - The server responds with both:
        - **Access Token:** For resource authorization.
        - **ID Token:** For user authentication.
3. **Final Exchange:**
    - The client can use the ID Token to confirm the user’s identity and the Access Token for resource access.

Benefits of OpenID Connect

1. **Unified Protocol:**
    - Combines authentication and authorization under one framework, simplifying development.
2. **Security:**
    - Leverages OAuth 2.0’s secure token mechanisms, ensuring robust identity validation.
3. **Interoperability:**
    - Applications can work seamlessly across platforms and systems using the standard OIDC protocol.

Summary

OpenID Connect extends OAuth 2.0 by adding authentication capabilities, making it a critical component for modern identity and access management systems. By standardizing identity sharing and incorporating ID Tokens, OIDC addresses the growing need for secure and interoperable user authentication.



OPENID CONNECT

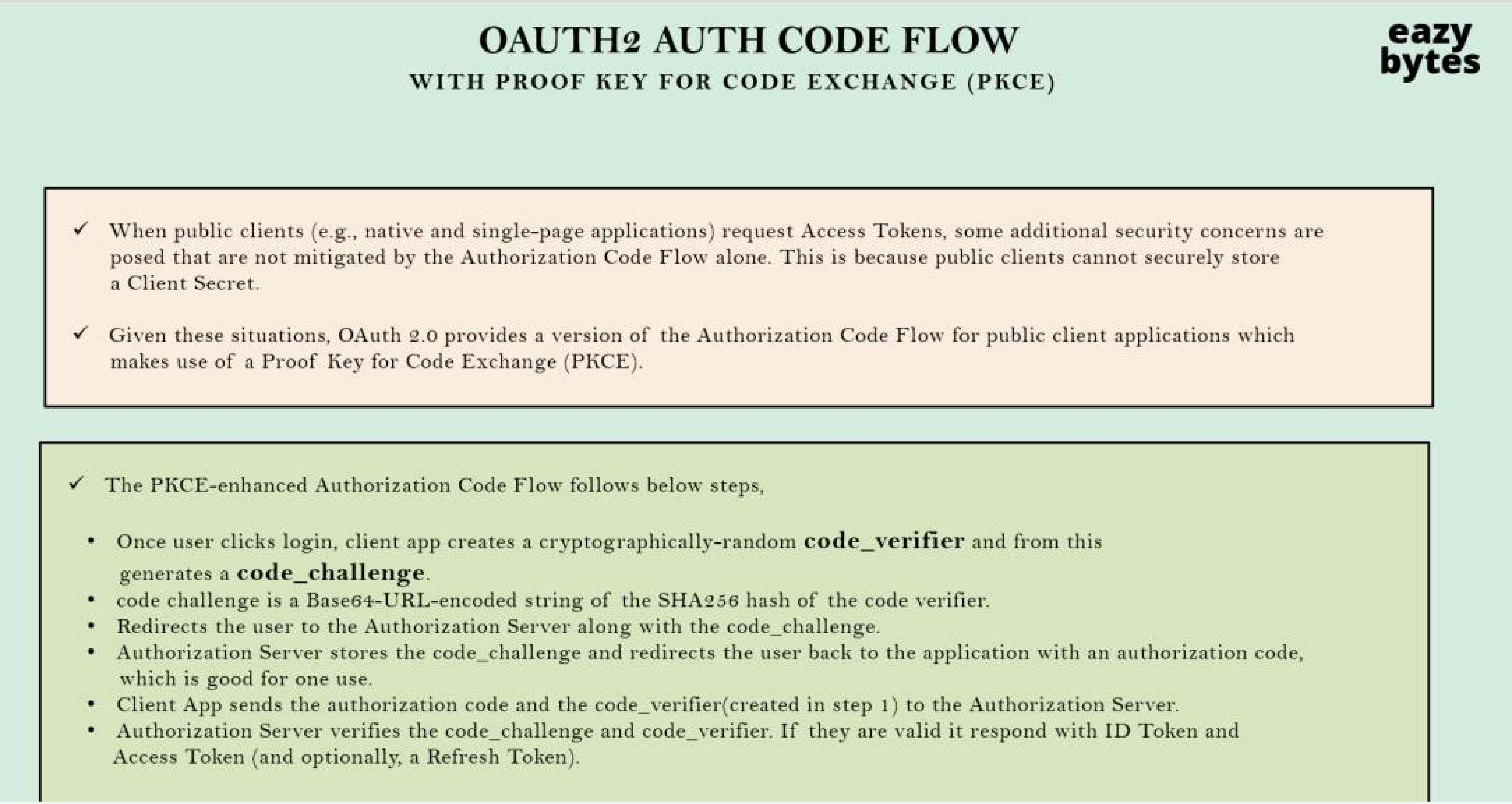
WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

Why is OpenID Connect important?

- Identity is the key to any application. At the core of modern authorization is OAuth 2.0, but OAuth 2.0 lacks an authentication component. Implementing OpenID Connect on top of OAuth 2.0 completes an IAM (Identity & Access Management) strategy.
- As more and more applications need to connect with each other and more identities are being populated on the internet, the demand to be able to share these identities is also increased. With OpenID connect, applications can share the identities easily and standard way.

OpenID Connect add below details to OAuth 2.0

1. OIDC standardizes the scopes to openid, profile, email, and address.
2. ID Token using JWT standard
3. OIDC exposes the standardized "/userinfo" endpoint.



OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

eazy bytes

✓ 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 respond with ID Token and Access Token (and optionally, a Refresh Token).

Kindly explain the above image with detailed explanation

The images explain **OpenID Connect (OIDC)** and the **OAuth2 Authorization Code Flow with PKCE (Proof Key for Code Exchange)**. Below is a detailed explanation of each concept:

OpenID Connect (OIDC): Why It's Important

Key Points:

1. **What Does OIDC Add to OAuth 2.0?**
    - OAuth 2.0 focuses on **authorization** (granting access to resources).
    - OIDC adds **authentication**, verifying the user’s identity and providing details (e.g., name, email).
2. **Completes IAM Strategy:**
    - Combining OIDC and OAuth 2.0 completes an **Identity and Access Management (IAM)** strategy:
        - OIDC handles identity (authentication).
        - OAuth 2.0 handles access management (authorization).
3. **Increasing Demand for Identity Sharing:**
    - As more applications integrate and users interact across systems, secure identity sharing becomes critical.
    - OIDC standardizes this process, making it easier for applications to securely share user identities.

OIDC Key Features:

1. **Scopes:**
    - Adds scopes like `` `openid` ``, `` `profile` ``, `` `email` ``, and `` `address` `` for accessing user information.
2. **ID Token:**
    - A **JWT (JSON Web Token)** containing identity claims about the user.
    - Used for verifying the user’s identity.
3. **Standardized Endpoints:**
    - Includes endpoints like `` `/userinfo` `` for retrieving additional user details.

OAuth 2.0 Authorization Code Flow with PKCE

What is PKCE?

- PKCE (Proof Key for Code Exchange) enhances the **Authorization Code Flow** for **public clients** (e.g., mobile or single-page apps) that cannot securely store a Client Secret.
- PKCE mitigates risks such as **authorization code interception**.

Steps in the PKCE Flow:

4. **Generate `code_verifier` and `code_challenge`:**
   - The client generates a random string (`code_verifier`).
   - A hashed version (`code_challenge`) is derived using SHA256.
5. **Authorization Request:**
   - The client sends the `code_challenge` to the Authorization Server along with the standard OAuth parameters (e.g., client ID, redirect URI, scope).
6. **Authorization Code Issued:**
   - After the user logs in and consents, the Authorization Server issues an **authorization code**, which is tied to the `code_challenge`.
7. **Token Request with `code_verifier`:**
   - The client exchanges the authorization code for an Access Token and ID Token by sending the original `code_verifier`.
8. **Validation by the Authorization Server:**
   - The Authorization Server validates the `code_verifier` against the original `code_challenge`.
   - If valid, it issues tokens (Access Token, Refresh Token, and optionally ID Token).

Benefits of PKCE:

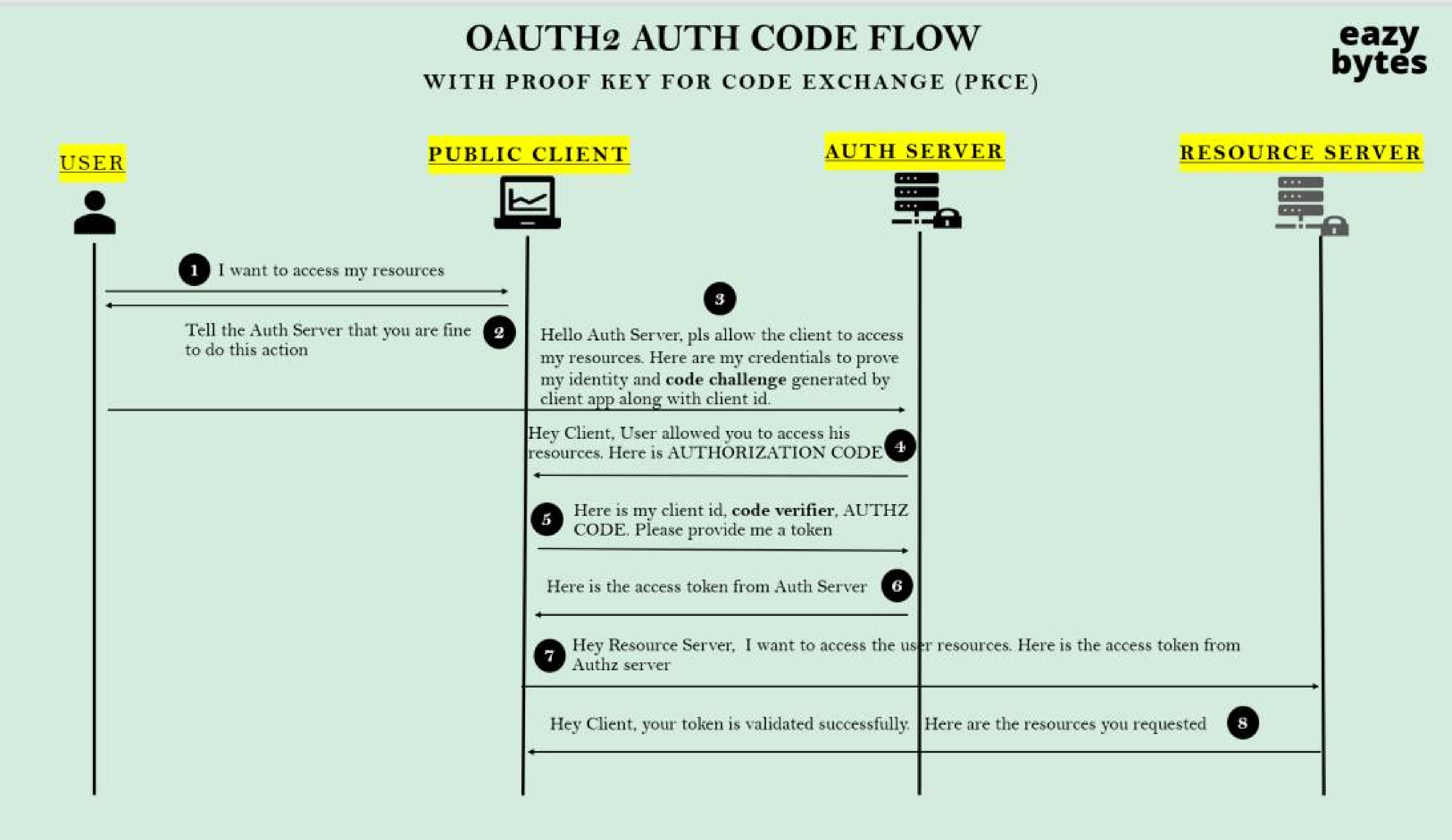
1. **Enhanced Security for Public Clients:**
   - Prevents attackers from using intercepted authorization codes because they lack the original `code_verifier`.
2. **No Need for Client Secrets:**
   - Ideal for public clients that cannot securely store secrets.
3. **Backward Compatible:**
   - Works seamlessly with Authorization Servers that already support the standard Authorization Code Flow.

Comparison of OIDC and PKCE:

Feature	OIDC	OAuth2 + PKCE
Primary Focus	Authentication (user identity)	Authorization (resource access)
Tokens Used	ID Token + Access Token	Access Token + (optional Refresh Token)
Target Applications	Apps needing user authentication	Public clients (mobile, SPA)
Security Enhancement	User identity validation	Mitigates code interception risks

Summary:

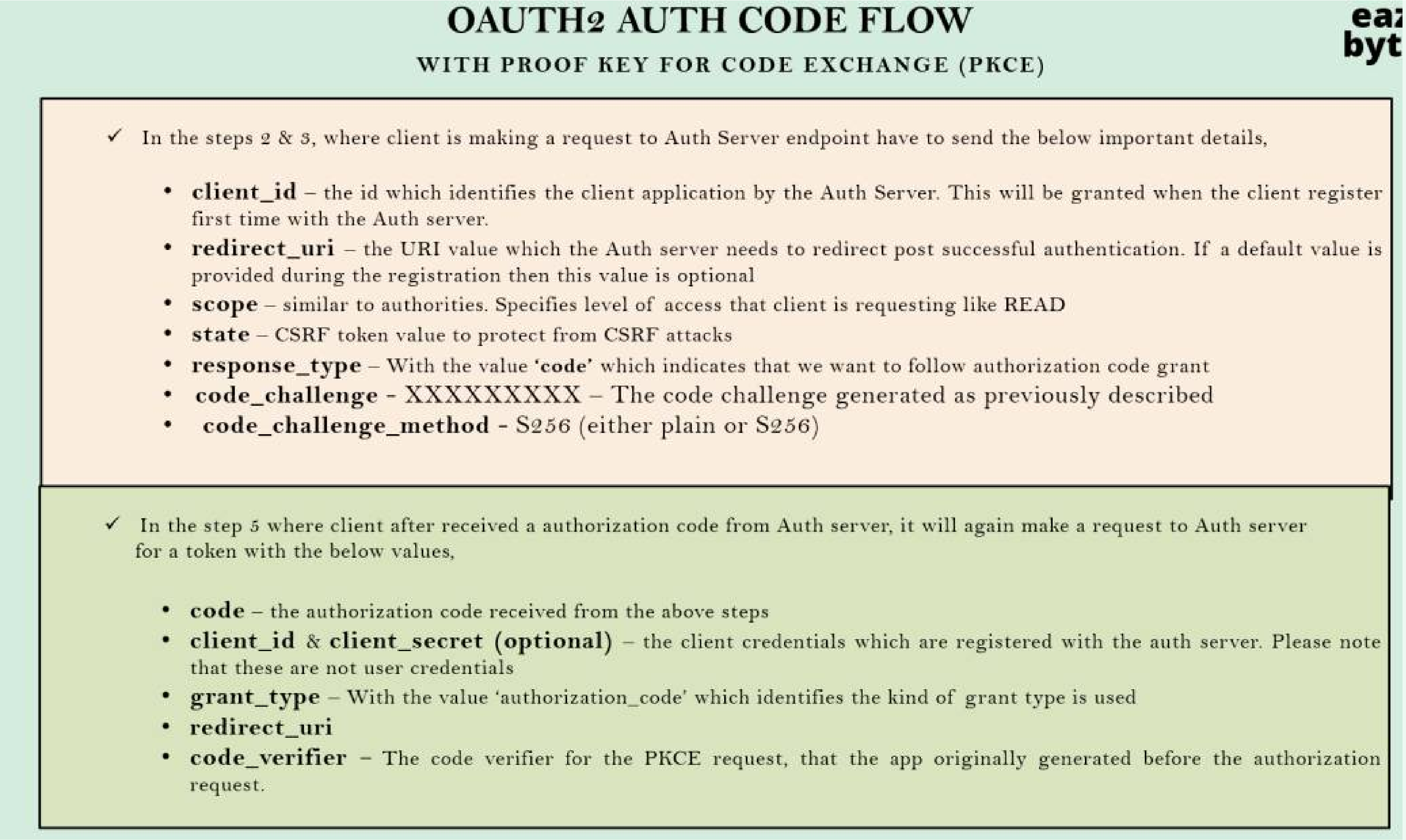
1. **OpenID Connect** adds user authentication to OAuth 2.0, making it essential for identity-focused applications.
2. **PKCE** improves the Authorization Code Flow's security, making it ideal for public client applications where secure storage of secrets is not possible. Both approaches significantly enhance the overall security and usability of OAuth-based systems.



OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

1. I want to access my resources
2. Tell the Auth Server that you are fine to do this action
3. Hello Auth Server, pls allow the client to access my resources. Here are my credentials to prove my identity and **code challenge** generated by client app along with client id.
4. Hey Client, User allowed you to access his resources. Here is AUTHORIZATION CODE
5. Here is my client id, **code verifier**, AUTHZ CODE. Please provide me a token
6. Here is the access token from Auth Server
7. Hey Resource Server, I want to access the user resources. Here is the access token from Authz server
8. Hey Client, your token is validated successfully. Here are the resources you requested



OAUTH2 AUTH CODE FLOW

WITH PROOF KEY FOR CODE EXCHANGE (PKCE)

✓ In the steps 2 & 3, where client is making a request to Auth Server endpoint have to 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 register 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** – similar to 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 **'code'** which indicates that we want to follow authorization code grant
- **code_challenge** - XXXXXXXXX – The code challenge generated as previously described
- **code_challenge_method** - S256 (either plain or S256)

✓ In the step 5 where client after received a 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 (optional)** – 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**
- **code_verifier** – The code verifier for the PKCE request, that the app originally generated before the authorization request.

Kindly explain the above image with detailed explanation

The images provide an overview of the **OAuth 2.0 Authorization Code Flow with Proof Key for Code Exchange (PKCE)** and the details of the flow. Below is a step-by-step explanation:

What is PKCE (Proof Key for Code Exchange)?

PKCE is an extension of the OAuth 2.0 Authorization Code Flow designed to enhance security for **public clients** (e.g., mobile apps, single-page applications) that cannot securely store a Client Secret. It prevents **authorization code interception attacks**.

Step-by-Step Flow

1. **Step 1 (User → Public Client):**
   - The user initiates the flow by requesting access to protected resources.
   - The client prepares to request authorization.
2. **Step 2 (Public Client → Authorization Server):**
   - The client generates a random `code_verifier` (cryptographically secure random string).
   - The client computes the `code_challenge` using the SHA256 hash of the `code_verifier`.
   - The client sends the following to the Authorization Server:
     - **client_id**: Identifies the client.
     - **redirect_uri**: Where the Authorization Server redirects after approval.
     - **scope**: Specifies requested permissions (e.g., READ, WRITE).
     - **state**: Prevents CSRF attacks.
     - **response_type**: Set to `code` to indicate the Authorization Code Flow.
     - **code_challenge**: The hashed value of the `code_verifier`.
     - **code_challenge_method**: Specifies the hashing algorithm (e.g., S256).
3. **Step 3 (Authorization Server → User):**
   - The Authorization Server prompts the user to authenticate and approve the client's access request.
4. **Step 4 (Authorization Server → Public Client):**
   - After the user consents, the Authorization Server sends an **authorization code** to the client.
5. **Step 5 (Public Client → Authorization Server):**
   - The client exchanges the authorization code for an Access Token.
   - The client sends:
     - **authorization code**: Received in Step 4.
     - **code_verifier**: The original random string generated in Step 2.
     - **client_id** and optionally **client_secret**: To authenticate the client.
     - **grant_type**: Set to `authorization_code`.

        - **redirect_uri**: Must match the value in Step 2.

6. **Step 6 (Authorization Server → Public Client):**
    - The Authorization Server validates the `` `code_verifier` `` against the `` `code_challenge` ``.
    - If valid, it issues an **Access Token** (and optionally a Refresh Token).
7. **Step 7 (Public Client → Resource Server):**
    - The client uses the Access Token to request resources from the Resource Server.
8. **Step 8 (Resource Server → Public Client):**
    - The Resource Server validates the token with the Authorization Server and responds with the requested resources.

Parameters in the PKCE Flow

During Authorization Request (Step 2):

- **client_id**: Unique ID of the client app.
- **redirect_uri**: URL to redirect after user consent.
- **scope**: Specifies access permissions (e.g., READ, WRITE).
- **state**: Ensures the request integrity and prevents CSRF attacks.
- **response_type**: `` `code` `` for Authorization Code Flow.
- **code_challenge**: The hashed `` `code_verifier` `` value.
- **code_challenge_method**: The hashing method, typically `` `S256` ``.

During Token Request (Step 5):

- **code**: The authorization code received earlier.
- **client_id and client_secret** (optional): Used to authenticate the client.
- **grant_type**: `` `authorization_code` ``.
- **redirect_uri**: Must match the initial redirect URI.
- **code_verifier**: The original un-hashed value for PKCE validation.

Why Use PKCE?

1. **Enhanced Security:**

    - Prevents attackers from using intercepted authorization codes without the original `` `code_verifier` ``.
2. **No Client Secret Needed:**
    - Suitable for public clients that cannot securely store secrets.
3. **Backward Compatibility:**
    - Works with existing Authorization Code Flow implementations.

Comparison: Authorization Code Flow vs. PKCE

Feature	Authorization Code Flow	Authorization Code Flow with PKCE
Security	Vulnerable to code interception	Protects against interception with `` `code_verifier` ``
Client Secret	Required	Not required for public clients
Suitability	Confidential clients	Public clients
Complexity	Simpler	Slightly more complex

Summary

PKCE adds an extra layer of security to the Authorization Code Flow, making it ideal for public clients like mobile and web applications. It ensures that even if an authorization code is intercepted, it cannot be exchanged for an Access Token without the `` `code_verifier` ``. This makes OAuth 2.0 more secure for a wider range of applications.

Kindly explain how to implement oAuth2 using SpringBoot and OpenId connect.Kindy explain the code with concise and detailed explanation

To implement OAuth2 using Spring Boot and OpenID Connect (OIDC), you can follow these steps. The implementation will include configuring an OAuth2 client in your Spring Boot application to authenticate users via an Authorization Server (e.g., Keycloak, Okta, or Google).

Dependencies

Add the necessary dependencies to your `pom.xml`:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

Application Configuration

Configure your application to act as an OAuth2 client. Add the following properties in your `application.yml`:

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          my-client:
            client-id: YOUR_CLIENT_ID
            client-secret: YOUR_CLIENT_SECRET
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            provider: my-provider
        provider:
          my-provider:
            issuer-uri: https://YOUR_AUTH_SERVER/.well-known/openid-configuration
```

- **`client-id` and `client-secret`:** Provided by the Authorization Server.
- **`issuer-uri`:** URL to fetch OIDC configuration (**`.well-known/openid-configuration`**).
- **`redirect-uri`:** Where the Authorization Server redirects after login.

Security Configuration

Create a security configuration class to enable OAuth2 login.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/public").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(); // Enable OAuth2 login
        return http.build();
    }
}
```

- **`oauth2Login`:** Enables OAuth2 login.
- **Authorization rules:**
    - Public URLs (e.g., **`/`**, **`/public`**) are accessible without authentication.
    - All other URLs require the user to log in.

Controller

Create a simple controller to handle requests and display user information.

```java
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.oidc.user.OidcUser;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @GetMapping("/")
    public String home() {
        return "Welcome to the Spring Boot OAuth2 OIDC Example!";
    }

    @GetMapping("/user")
    public String userInfo(@AuthenticationPrincipal OidcUser oidcUser) {
        return "Hello, " + oidcUser.getFullName() + "! Your email is " + oidcUser.getEmail();
    }
}
```

- **`@AuthenticationPrincipal OidcUser`:** Injects the authenticated user's OIDC data.
- Use methods like `getFullName()` and `getEmail()` to access user information.

OIDC Flow

1. User visits a secured page (e.g., `/user`).
2. Spring Security redirects to the Authorization Server's login page.
3. User logs in and consents to scopes (if required).
4. Authorization Server redirects back to your application with an authorization code.
5. Spring Security exchanges the code for tokens (Access Token, ID Token).
6. The user is authenticated, and you can access their OIDC data.

Running the Application

1. Start your Spring Boot application.
2. Access the public URL (e.g., `` `/` ``).
3. Navigate to a protected endpoint (e.g., `` `/user` ``), which triggers the OAuth2 login flow.

Summary

This implementation demonstrates how to integrate OAuth2 with OpenID Connect in Spring Boot. Key features include:

1. **OIDC Configuration:** Managed via Spring Boot properties (`` `application.yml` ``).
2. **Security Rules:** Define which endpoints are protected.
3. **User Info:** Extract and display user information using `` `OidcUser` ``.

This setup is flexible and can connect to any OIDC-compliant Authorization Server.

How to implement Oauth2 in springBoot.Kindly mention each steps in details

To implement **OAuth2 in Spring Boot**, follow these steps. The implementation will focus on configuring your Spring Boot application as an **OAuth2 client** to authenticate users and secure APIs.

---

1. Add Dependencies

Add the required dependencies to your `` `pom.xml` `` if you are using Maven:

xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

If using Gradle, include the following:

```groovy
implementation 'org.springframework.boot:spring-boot-starter-security'
implementation 'org.springframework.boot:spring-boot-starter-oauth2-client'
implementation 'org.springframework.boot:spring-boot-starter-web'
```

---

2. Configure OAuth2 in `application.yml`

Set up your OAuth2 client credentials and endpoints.

```yaml
spring:
  security:
    oauth2:
      client:
        registration:
          my-client:
            client-id: YOUR_CLIENT_ID
            client-secret: YOUR_CLIENT_SECRET
            scope: openid, profile, email
            authorization-grant-type: authorization_code
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            provider: my-provider
        provider:
          my-provider:
            issuer-uri: https://YOUR_AUTH_SERVER/.well-known/openid-configuration
```

Explanation:

- **`client-id` and `client-secret`:** These are credentials provided by the Authorization Server (e.g., Google, Okta).
- **`scope`:** Specifies the data or permissions requested (e.g., `profile`, `email`).
- **`authorization-grant-type`:** Indicates the flow being used (e.g., `authorization_code` for user login).
- **`redirect-uri`:** The endpoint the Authorization Server redirects to after authentication.
- **`issuer-uri`:** The URL of the Authorization Server's metadata.

---

3. Security Configuration

Define security rules and enable OAuth2 login by configuring `HttpSecurity`.

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/", "/public").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(); // Enable OAuth2 Login
        return http.build();
    }
}
```

Explanation:

- **`requestMatchers`:** Specifies which endpoints are public and which require authentication.
- **`oauth2Login`:** Enables OAuth2 login functionality in Spring Security.

---

4. Create a Controller

Set up a controller to test authentication and display user information.

```java
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.security.oauth2.core.user.OAuth2User;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class UserController {

    @GetMapping("/")
    public String home() {
        return "Welcome to OAuth2 with Spring Boot!";
    }

    @GetMapping("/user")
    public String userInfo(@AuthenticationPrincipal OAuth2User oauth2User) {
        return "Hello, " + oauth2User.getAttribute("name") + "! Your email is " + oauth2User.getAttribute("email");
    }
}
```

Explanation:

- **`@AuthenticationPrincipal OAuth2User`:** Injects the authenticated user details.
- You can access user attributes like `name` and `email` using `getAttribute`.

---

5. Run the Application

1. Start your Spring Boot application.
2. Navigate to `http://localhost:8080` to access the public homepage.
3. Access a protected URL (e.g., `/user`), which will redirect you to the Authorization Server for login.

---

6. Optional: Secure API with Resource Server

If you want to secure APIs using OAuth2 tokens, configure your application as a **Resource Server**.

Add Dependency:

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Configure Resource Server:

```yaml
spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: https://YOUR_AUTH_SERVER/.well-known/openid-configuration
```

Security Configuration:

```java
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain resourceServerSecurityFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt()); // Enable JWT-based authentication
        return http.build();
    }
}
```

---

7. Testing

1. **OAuth2 Login:**
    - Open your browser and navigate to `/user`.
    - You’ll be redirected to the Authorization Server (e.g., Google login page).
    - After login, you’ll see user information.
2. **Token-Based API:**
    - Use a tool like Postman to call the API with an Access Token in the Authorization header:

      ```makefile
      Authorization: Bearer <ACCESS_TOKEN>
      ```

    - The Resource Server validates the token and grants or denies access.

---

8. Summary of Key Components

Component	Description
Dependencies	Include `spring-boot-starter-oauth2-client` or `resource-server`.
Configuration	Define OAuth2 client details in `application.yml`.
SecurityConfig	Secure endpoints and enable OAuth2 login or resource server.
Controller	Access user attributes via `OAuth2User`.

Advantages of OAuth2 with Spring Boot

1. **Simplifies Authentication:** Prebuilt integrations with OAuth2 providers (Google, Okta, Keycloak).
2. **Customizable:** Configure roles, scopes, and resources to fit your needs.
3. **Secure:** Built-in features to handle token validation and user authentication.

This detailed guide should help you implement OAuth2 effectively in your Spring Boot application!