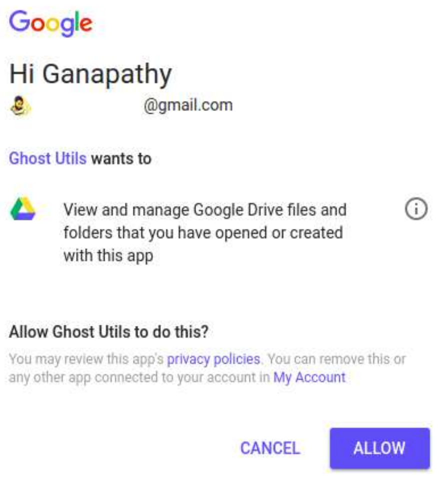
OAuth

OAuth is an authorization framework that enables a third-party application to obtain limited access to an HTTP service. It doesn't provide authentication; it provides authorization.
<u>Authorization, Not Authentication:</u> It doesn't verify who the user is (authentication), but it determines what actions or data a third-party application can access (authorization).
<u>Third-Party Access:</u> OAuth allows third-party apps to request permission to access a user's resources (such as email, social media, or data) from a service (such as Google, Facebook, GitHub) on behalf of the user.
<u>Has 2 versions, 1.0 and 2.0. Will learn about 2.0.</u>
Essentially, OAuth 2.0 allows arbitrary **clients** (for example, a first-party iOS application or a third-party web application) to access **user's** (resource owner's) **resources** on **resource servers** via **authorization servers** in a secure, reliable, and efficient manner.

Terminologies:

<u>Client</u>: your app. It contains-
- Client ID
- Client Secret: A secret only the client & auth server know. So that hacker bhaia can't impersonate you and access the resources from the resource server. It prevents phishing attacks.

<u>Resource Owner</u>: the user who owns the resources
<u>Authorization Server</u>: Google/Github has a server that identifies/decides the user with the appropriate authorizations.
<u>Resource Server</u>: Authorization Server jokhon Client ke authorize korbe, tokhon Client Resource Server er kache request pathaite parbe resource neyar jonno. Then Resource Server will send back resources to the Client, not the User.
Auth server & Resource server can be one, or different.
<u>Scopes</u>: what types of facilities do you want to get from the resource server. Client sends the scope list to auth server then based on that, auth server shows consent.
<u>Consent</u>: Authorization Server last barer moto dekhabe je client wants to access your scopes. Client doesn't show the consent, auth server does. Hence, client do any foul play


Google

Hi Ganapathy

@gmail.com

Ghost Utils wants to

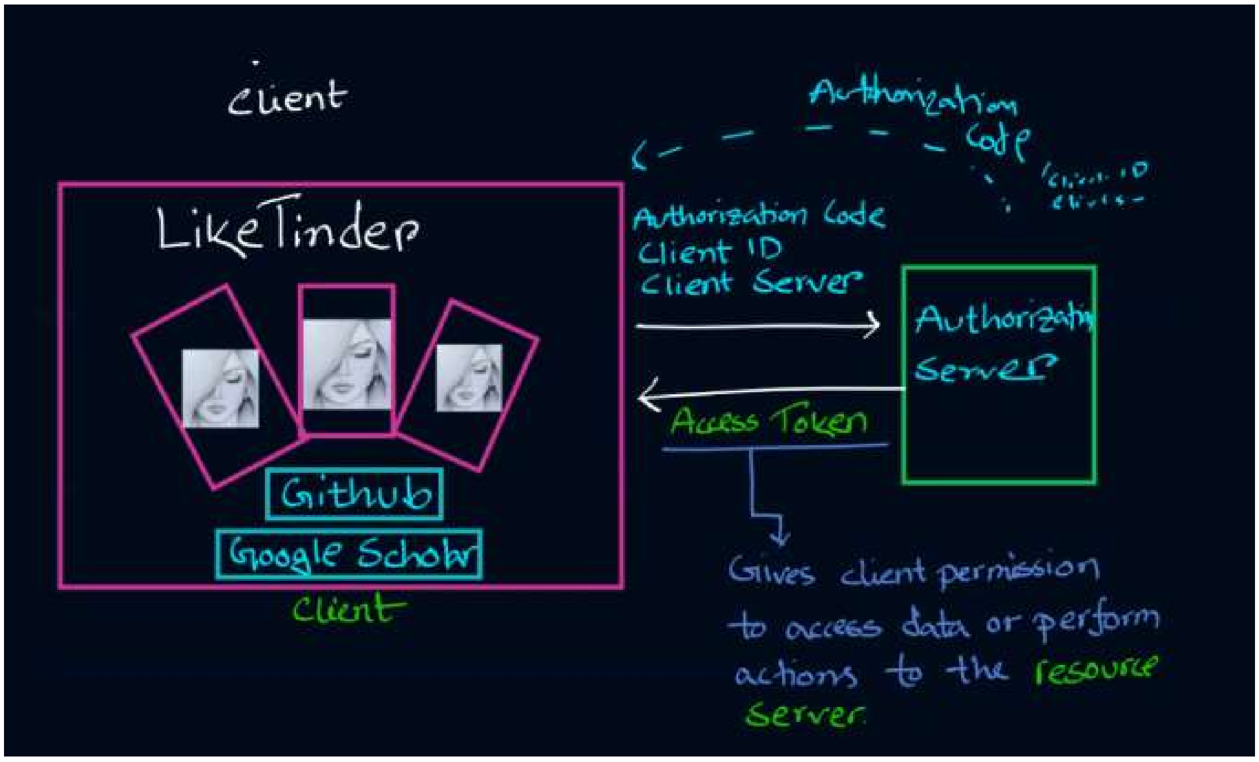
View and manage Google Drive files and folders that you have opened or created with this app

Allow Ghost Utils to do this?

You may review this app's privacy policies. You can remove this or any other app connected to your account in My Account

CANCEL ALLOW

Authorization Code: **Short validity**.
Authorization Server first ei client ke ekta Authorization Code dey. Then client Client ID, Client Secret abar Authorization Server kei pathay dey. Authorization Server tokhon client ke ekta Access Token dibe. Ei Access Token tokhon client ke Resource Server e access dey. Ei access token e ki ki ase sheta client bujhe na. Client just Access Token ta Resource Server e pathay dey, Resource Server take data ta pathay dey.

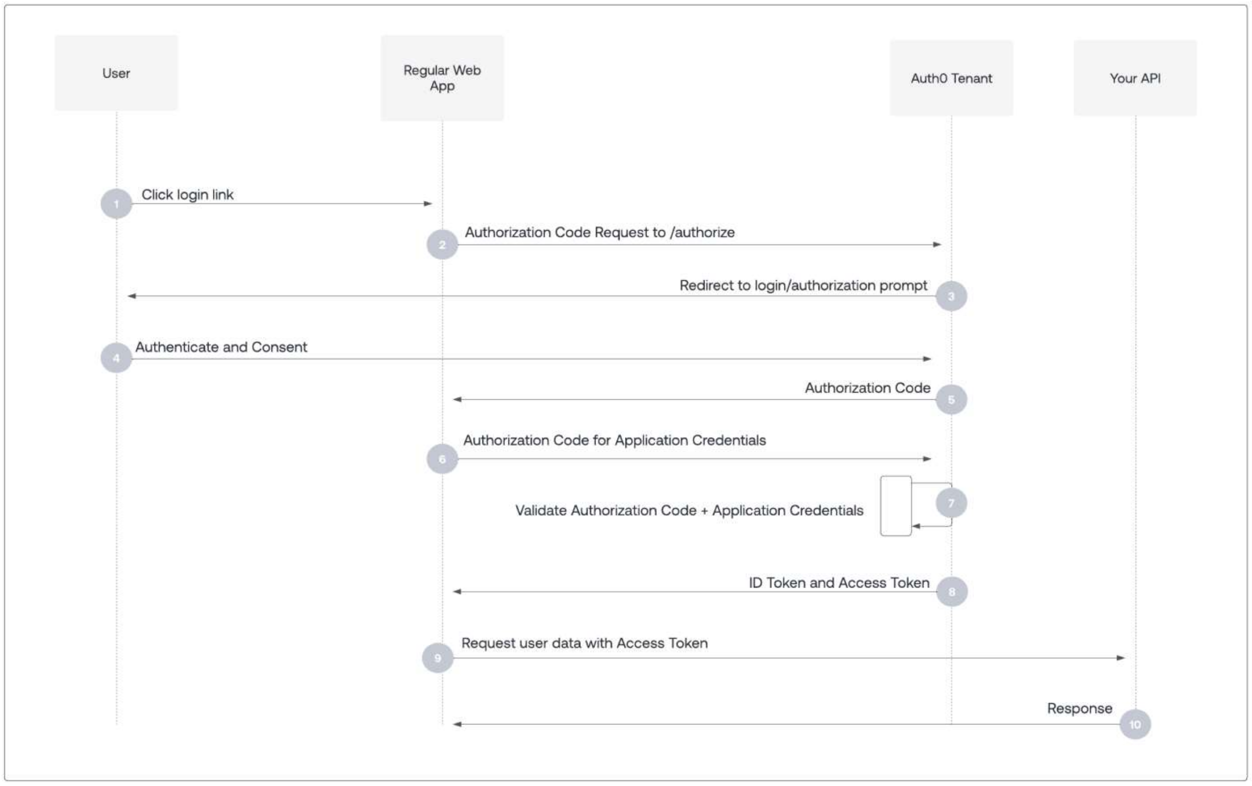


Access Tokens: There are two types of tokens: access tokens and refresh tokens. Anyone with a valid **access token** can access protected resources; usually it is short-lived so that even if there is a security breach and the access token is leaked, the damage can be quickly controlled. When an access token expires, developers can use an optional **refresh token** to request a new access token without having to ask the user to enter their credentials again.

Authorization Flow

OAuth flows are different ways of retrieving Access Tokens

Authorization code flow


User
Regular Web App
Auth0 Tenant
Your API
1
Click login link
2
Authorization Code Request to /authorize
3
Redirect to login/authorization prompt
4
Authenticate and Consent
5
Authorization Code
6
Authorization Code for Application Credentials
7
Validate Authorization Code + Application Credentials
8
ID Token and Access Token
9
Request user data with Access Token
10
Response

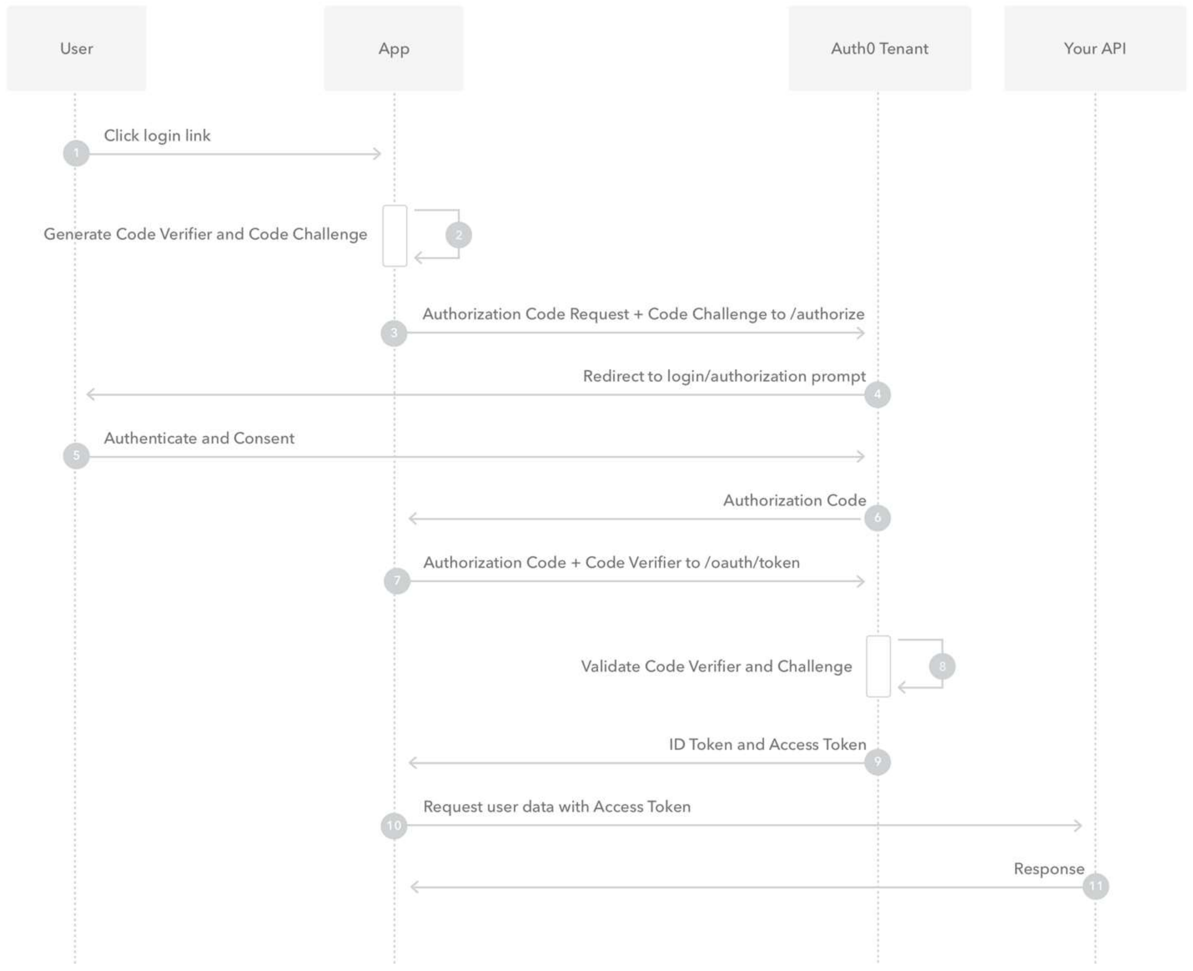
***** applicable for all the flows:**
Auth0 Tenant → Authorization server
Your API → Resource Server

Authorization Code Flow with Proof Key for Code Exchange (PKCE)

Native apps: Cannot securely store a Client Secret. Decompiling the app will reveal the Client Secret, which is bound to the app and is the same for all users and devices.
May make use of a custom URL scheme to capture redirects (e.g., MyApp://) potentially allowing malicious applications to receive an Authorization Code from your Authorization Server.

Single-page apps: Cannot securely store a Client Secret because their entire source is available to the browser.

Given these situations, OAuth 2.0 provides a version of the Authorization Code Flow which makes use of a Proof Key for Code Exchange (PKCE).
The application creates a secret (Code Verifier) that will be verified by the server. Additionally, the app creates a transform value of the Code Verifier (Code Challenge) and sends this value to Authorization server. This way, a malicious attacker can only intercept the Authorization Code, and they cannot exchange it for a token without the Code Verifier.


User
App
Auth0 Tenant
Your API
1 Click login link
Generate Code Verifier and Code Challenge 2
3 Authorization Code Request + Code Challenge to /authorize
Redirect to login/authorization prompt 4
5 Authenticate and Consent
Authorization Code 6
7 Authorization Code + Code Verifier to /oauth/token
Validate Code Verifier and Challenge 8
ID Token and Access Token 9
10 Request user data with Access Token
Response 11

Implicit Flow

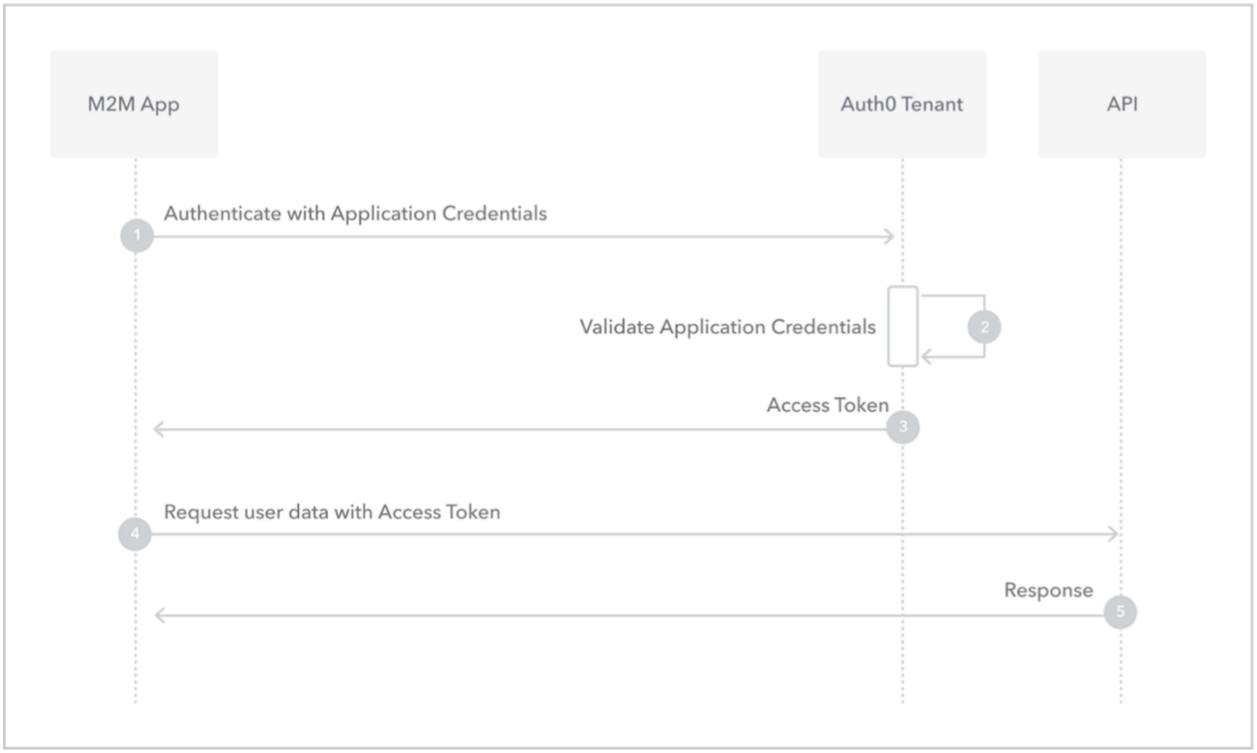
In the case of authorization code flow, for step 2, App sends the secret along with other info. So, there's no need for step 5-7

Resource Owner Password Credentials Flow

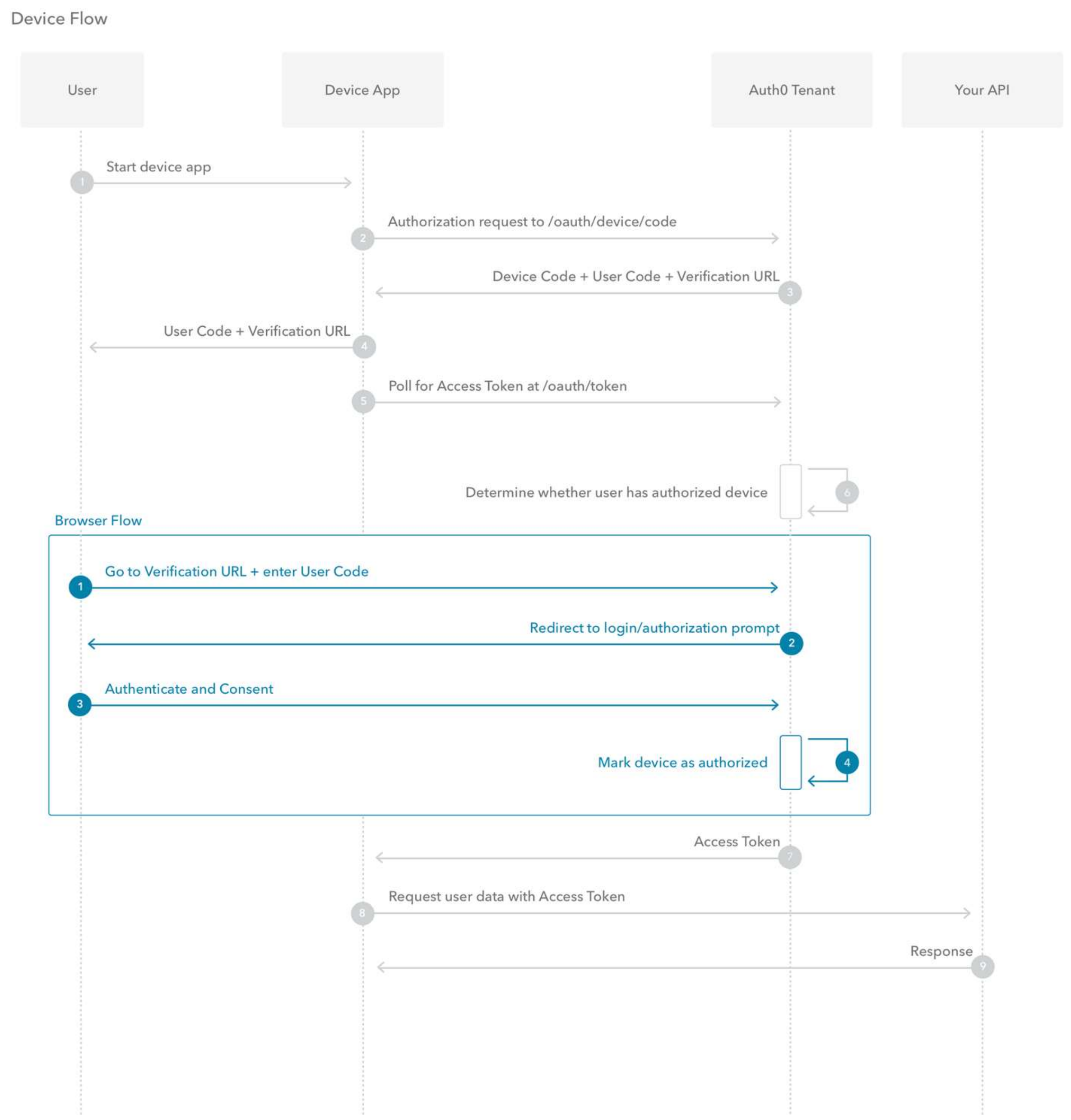
Here, in step 2→ the app will directly send user credentials to the authorization server. So no need for step 3,4.

Client Credentials Flow

No need for the user. App directly communicates with the server. Auth server authenticates the app using clientID, secret etc. Then the app directly accesses the resources from the resource server.


M2M App
Auth0 Tenant
API
1
Authenticate with Application Credentials
Validate Application Credentials
2
Access Token
3
4
Request user data with Access Token
Response
5

Device Authorization Flow


Device Flow
User
Device App
Auth0 Tenant
Your API
1 Start device app
2 Authorization request to /oauth/device/code
3 Device Code + User Code + Verification URL
4 User Code + Verification URL
5 Poll for Access Token at /oauth/token
6 Determine whether user has authorized device
Browser Flow
1 Go to Verification URL + enter User Code
2 Redirect to login/authorization prompt
3 Authenticate and Consent
4 Mark device as authorized
7 Access Token
8 Request user data with Access Token
9 Response

Device Flow

1. The user starts the app on the device.
2. The device app requests authorization from the Auth0 Authorization Server using its Client ID.
3. The Auth0 Authorization Server responds with a device_code, user_code, verification_uri, verification_uri_complete expires_in, and polling interval.
4. The device app asks the user to activate using their computer or smartphone. The app may accomplish this by:
    a. asking the user to visit the verification_uri and enter the user_code after displaying these values on-screen

    b. asking the user to interact with either a QR Code or shortened URL with embedded user code generated from the verification_uri_complete
    c. directly navigating to the verification page with embedded user code using verification_uri_complete, if running natively on a browser-based device
5. The device app begins polling your Auth0 Authorization Server for an Access Token using the time period specified by interval and counting from receipt of the last polling request's response. The device app continues polling until either the user completes the browser flow path or the user code expires.
6. When the user successfully completes the browser flow path, your Auth0 Authorization Server responds with an Access Token (and optionally, a Refresh Token). The device app should now forget its device_code because it will expire.
7. Your device app can use the Access Token to call an API to access information about the user.
8. The API responds with requested data.

Browser Flow

1. The user visits the verification_uri on their computer, enters the user_code and confirms that the device that is being activated is displaying the user_code. If the user visits the verification_uri_complete by any other mechanism (such as by scanning a QR code), only the device confirmation will be needed.
2. Your Auth0 Authorization Server redirects the user to the login and consent prompt, if needed.
3. The user authenticates using one of the configured login options and may see a consent page asking to authorize the device app.
4. Your device app is authorized to access the API.

When to use which one and Example -

Flow	Client Type
Authorization Code	Less trusted apps (third-party apps requesting access to your platform)
Resource Owner Password Credentials	Highly trusted apps (first-party apps). This flow is considered less secure and should only be used in cases where the user and application trust each other fully. **Example**: A company's internal application asks employees to enter their credentials directly for accessing protected resources.
Implicit	Single-page Javascript Web Applications (for example, Google Fonts) that run entirely in the browser.
Client Credentials	Non-interactive programs for machine-to-machine communications (for example, background services and daemons) where no user is involved. **Example**: A backend service wants to access an API that provides weather data, using its own credentials without user intervention.
Authorization Code Flow with Proof Key for Code Exchange (PKCE)	Single-page app, Native apps. **Example**: A mobile banking app wants to authenticate users via Google OAuth without exposing sensitive data in the URL or storing any secrets on the device.
Device Flow	devices that cannot directly display a web-based login, such as smart TVs, gaming consoles, or IoT devices.

Passkey

Passkeys are a safer and easier alternative to passwords. With passkeys, users can sign in to apps and websites with a biometric sensor (such as a fingerprint or facial recognition), PIN, or pattern, freeing them from having to remember and manage passwords.
The W3C has published WebAuthn, a standard that establishes a universal method for passwordless user authentication using public-key cryptography. The credentials generated through this method, commonly known as passkeys, allow users to authenticate themselves solely using their device and features like biometric recognition. This eliminates the need to remember multiple complex passwords or undergo an additional step for identity confirmation through a second authentication factor.
Passkeys are safer:

- Developers only save a public key to the server instead of a password, meaning there's far less value for a bad actor to hack into servers, and far less cleanup to do in the event of a breach.
- Passkeys protect users from phishing attacks. Passkeys work only on their registered websites and apps; a user cannot be tricked into authenticating on a deceptive site because the browser or OS handles verification.
- Passkeys reduce costs for sending SMS, making them a safer and more cost-effective means for two-factor authentication.

https://youtu.be/2xdV-xut7EQ

What are passkeys?

A passkey is a digital credential, tied to a user account and a website or application. When a user wants to sign in to a service that uses passkeys, their **browser or operating system** will help them select and use the right passkey. To make sure only the rightful owner can use a passkey, the **system** will ask them to **unlock** their device. This may be performed with a biometric sensor (such as a fingerprint or facial recognition), PIN, or pattern.
To create a passkey, you use the **WebAuthn API** (Web Authentication: An API for accessing Public Key Credentials).

Anatomy of a passkey system

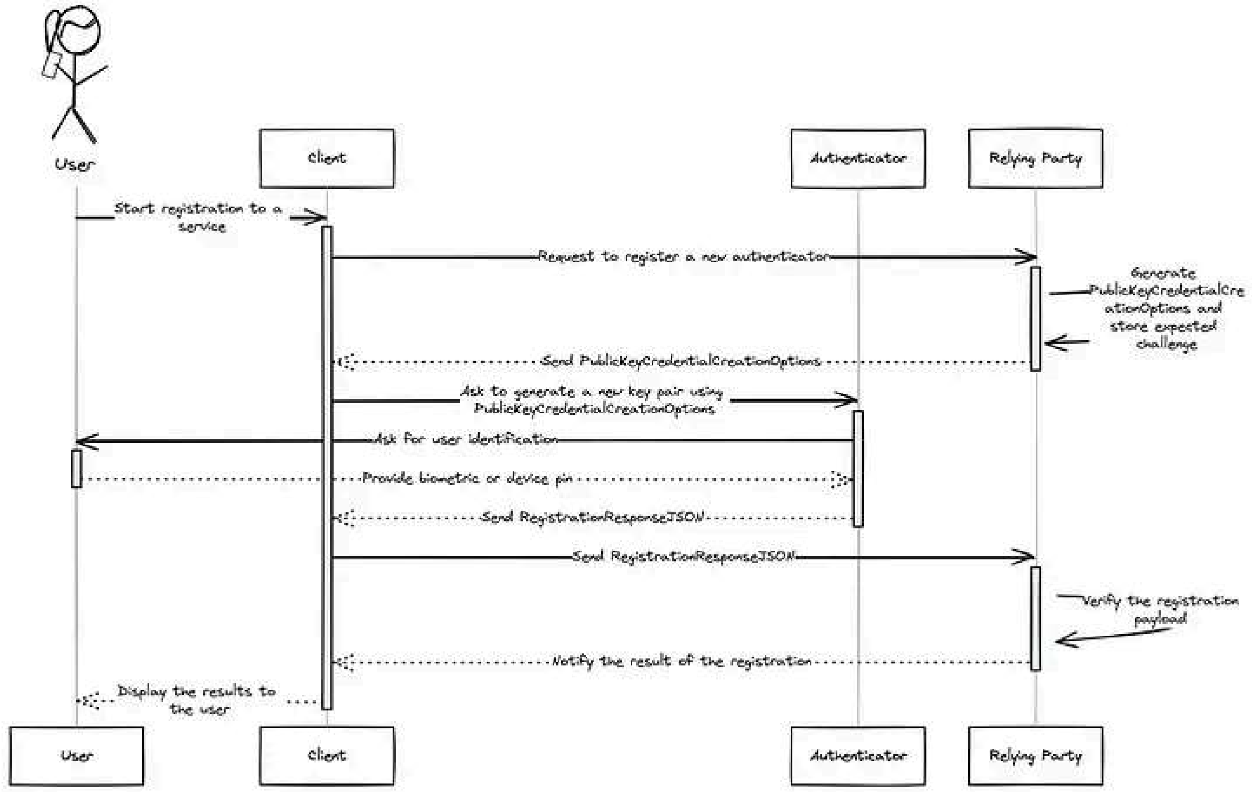
A passkey system consists of a few components:

- **Authenticator: T**his is the device or system that verifies the user's identity. It handles the authentication process, often using biometric methods (like fingerprints or facial recognition) or other forms of secure verification to ensure that the person trying to log in is indeed the authorized user.
- **Relying Party:** This is the service or application that the user is trying to access. It relies on the authenticator to verify the user's identity and grant access to its resources.

- **Password manager:** Software installed on the end user's device(s) that serves, stores and syncs passkeys, such as the Google Password Manager.

How to register a new passkey

Whether the user is already registered to the service or the user wants to create an account with the passkey, the basic flow is more or less the same.


User
Client
Authenticator
Relying Party
Start registration to a service
Request to register a new authenticator
Generate PublicKeyCredentialCreationOptions and store expected challenge
Send PublicKeyCredentialCreationOptions
Ask to generate a new key pair using PublicKeyCredentialCreationOptions
Ask for user identification
Provide biometric or device pin
Send RegistrationResponseJSON
Send RegistrationResponseJSON
Verify the registration payload
Notify the result of the registration
Display the results to the user
User
Client
Authenticator
Relying Party

1. The user intends to register for a service using a client, either a mobile app or a website.
2. The client initiates the registration process, requesting the relying party to register a new authenticator.
3. The relying party responds with a JSON (PublicKeyCredentialCreationOptions) that includes a challenge and additional metadata, such as relying party information and accepted authenticator types. Additionally, the relying party stores the challenge for this registration session, as it will need to be verified when the response is received.
4. The client uses this response to ask the authenticator to generate a new key pair that will be used to authenticate the user to the relying party.
5. The authenticator (which can be an external device or a smartphone) verifies the server payload and requests user identification (via biometrics or device PIN).
6. If all goes well, the authenticator generates a new private and public key pair and returns a JSON (RegistrationResponseJSON) to the client as a result, containing the newly created public key, attestation information, and other relevant details. It should be noted that the private key remains within the authenticator and, in the case of smartphones, it is stored in a separate hardware component known as the secure enclave or Trusted Execution Environment.

7. The client sends this attestation object to the relying party to confirm and verify the new registration.
8. The relying party verifies the received payload, checking the authenticity of the attestation object, and ensures that the user's registration meets the specified criteria (e.g., correct challenge response, proper attestation format).
9. After verifying the registration payload, the relying party notifies the user of the result. If successful, it stores relevant information about the registered credential that will be used for future authentication.

For step 3 ⇒ Fetch important information from the backend

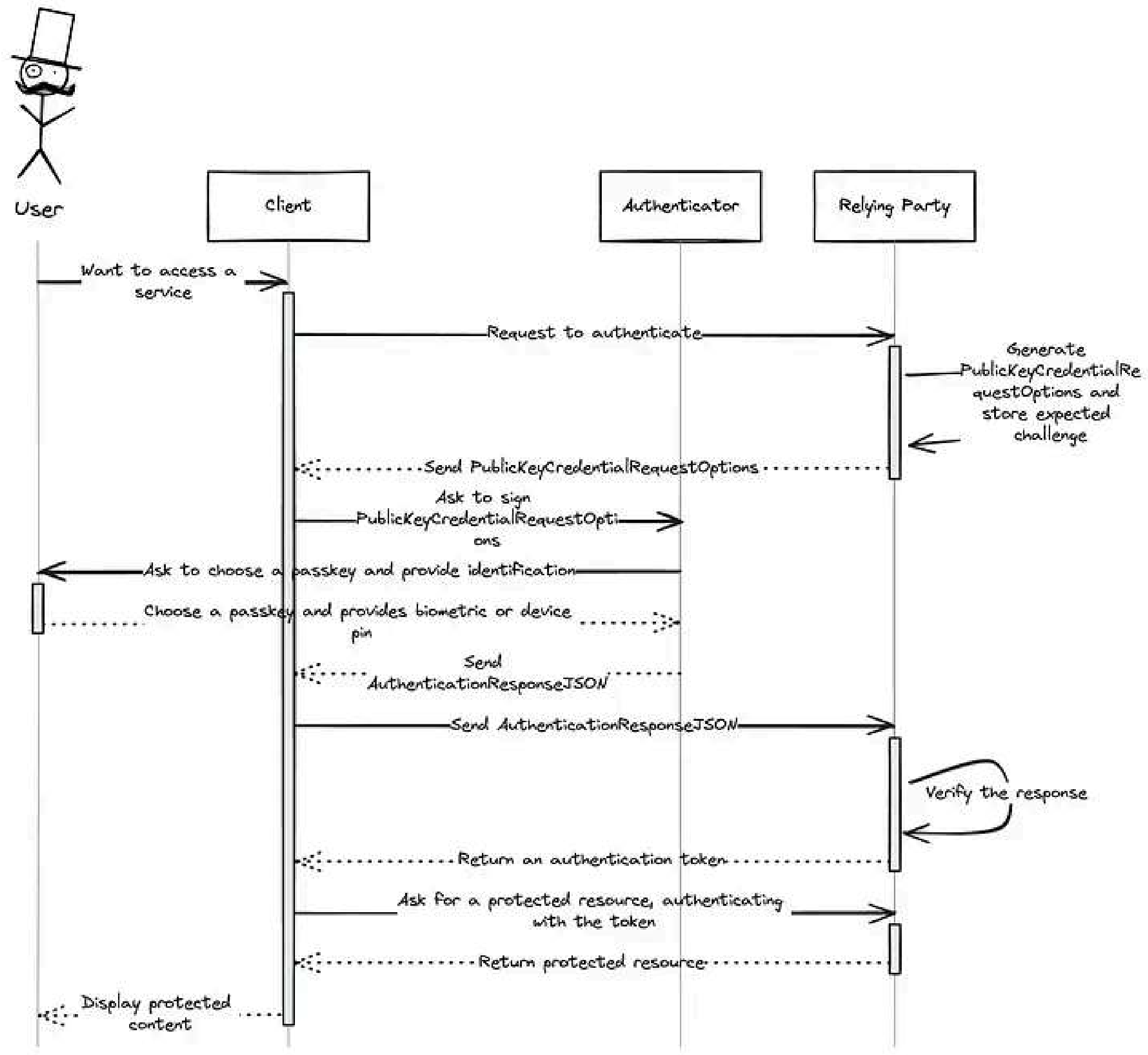
When the user clicks the button, fetch important information to call navigator.credentials.create() from the backend:

- **challenge**: A server-generated challenge in ArrayBuffer for this registration. This is required but unused during registration unless doing attestation—an advanced topic not covered here.
- **RP ID**: Provide the relying party's ID in the form of a web domain.
- **User information**: The user's ID, username and a display name.
- **Credentials to exclude**: Information about previously stored passkeys to prevent duplicate registration.
- **Passkey types**: Whether to use the device itself ("platform authenticator") as an authenticator, or a detachable security key ("cross-platform / roaming authenticator"). Additionally, callers can specify whether to make the credential discoverable so that the user can select an account to sign in with.

Attestation refers to a mechanism used to verify that the authenticator (the device or system performing the authentication) is genuine and meets certain security standards.

Authentication

Once a passkey is registered, the user can use it to authenticate with the relying party and gain access to a protected resource. Let's see the procedure that allows the user to access the service:


User
Client
Authenticator
Relying Party
Want to access a service
Request to authenticate
Generate PublicKeyCredentialRequestOptions and store expected challenge
Send PublicKeyCredentialRequestOptions
Ask to sign PublicKeyCredentialRequestOptions
Ask to choose a passkey and provide identification
Choose a passkey and provides biometric or device pin
Send AuthenticationResponseJSON
Send AuthenticationResponseJSON
Verify the response
Return an authentication token
Ask for a protected resource, authenticating with the token
Return protected resource
Display protected content

1. The user wants to access a service using a mobile app or a website and needs to identify themselves.
2. The client initiates the authentication process by requesting the relying party to authenticate.
3. The relying party sends a JSON with the authentication options (PublicKeyCredentialRequestOptions) that include a challenge and information about the relying party. Additionally, it stores the challenge for this authentication session, as it will need to be verified when the response is received.
4. The client sends this JSON to the authenticator and requests it to sign it using one of the passkeys linked to the relying party.

5. If the authenticator has multiple passkeys associated with the relying party, the user is prompted to select a passkey to proceed with the authentication process. Once selected, the user is prompted to provide identification via biometrics or the device PIN in order to sign the challenge with the private key stored within the authenticator.
6. If the operation proceeds successfully, the authenticator returns to the client a JSON (AuthenticationResponseJSON) containing the signed challenge, authenticator details, and other relevant client data.
7. The client sends this authentication response to the relying party.
8. The relying party first verifies whether the authenticator used is registered. Then, it checks the response's authenticity and the correctness of the challenge response.
9. After successful verification, the relying party grants access to the user, allowing them to use the service, returning, for example, an authentication token that can be used for the next requests.
10. The user is now authenticated and can use the mobile app or website.