African Free Fire Community
africanfreefirecommunity.com

INTEGRATION GUIDE FOR PARTNER ORGANISATIONS

Sign in with AFC

Partner Integration Guide

AFC is an **OpenID Connect provider**. This guide is everything a partner organisation needs to add "Sign in with AFC" to its site: the exact endpoints, the scope catalogue, the claims each scope produces, the rules that decide what actually reaches you, and a complete worked example.

PROTOCOL	FLOW	VERSION	ISSUED
OpenID Connect	**Auth code + PKCE**	**1.2**	**4 August 2026**

ABOUT THIS DOCUMENT

Audience: developers (and AI coding assistants) integrating "Sign in with AFC" into a partner website or application.

Status of this document: written against the AFC implementation as it stands on 2026-08-04. Every endpoint path, parameter name, claim name, error string and lifetime below was read out of the AFC backend source, not out of the OpenID Connect specification. Where AFC differs from what a spec-reading developer would expect, that difference is called out.

Questions about anything in this guide: **info@africanfreefirecommunity.com**.

A few items below are noted as not yet decided by AFC. They do not block an integration; they are flagged so you do not build on something provisional.

CONTENTS

1. What this is

AFC (African Free Fire Community) is an **OpenID Connect provider**. Your site is the **relying party**. A player who already has an AFC account clicks "Sign in with AFC" on your site, is sent to AFC to sign in and approve what is shared, and is sent back to you with an authorization code. You exchange that code for an ID token, an access token and a refresh token. From then on you can read that player's AFC data from the userinfo endpoint, without the player signing in again, until they revoke you.

The important difference from "Sign in with Google" is this: **AFC decides, per partner, the maximum set of fields you may ever receive, and the player then consents on top of that.** Google gives every developer the same menu and lets the user pick from it. AFC gives *your organisation* a menu that an AFC admin configured specifically for you, and the player picks from that. On top of both, AFC applies its own protection rules (for example, players under 18 never release contact data). The practical consequence, and the single thing integrations most often get wrong: **you can request a scope you were approved for, the player can approve it, and you can still legitimately receive no claim for it.** Your code must treat every claim other than `sub` as optional. See **section 7 (page 19)**.

AFC implements the standard protocol with no AFC dialect. Any compliant OIDC client library works unmodified. There is no AFC SDK to install.

2. Getting set up: what AFC issues you

Apply here: `https://africanfreefirecommunity.com/partners/apply`. You do not need an AFC account to apply, and there is nothing to install. Fill the form in, and AFC emails you a link to track the application.

There is still **no self-serve app registration**: the form is an application, not a signup. Dynamic Client Registration (RFC 7591) is disabled on the server, and an AFC admin decides every application by hand. What the form removes is the part where you emailed your redirect URIs to a person who then retyped them.

info@africanfreefirecommunity.com still works, for anything the form does not cover.

What the form asks you for

YOU PROVIDE	NOTES
Which product you want	"Sign in with AFC", the Data API, or both. This is routing, not a request for data.
Organisation name, and the name shown to players	The second is what appears on the consent screen ("*Name* wants to use your AFC account"). Leave it blank to use your organisation name.
Website	Required. Shown on the player's Connected Apps page.
Contact name and email	**Every reply goes to that address**, including your credentials link. Check it before you send.
Logo	Optional file, PNG, JPG or WEBP, up to 2 MB. AFC hosts its own copy rather than loading yours, because it renders on the consent screen.
One or more redirect URIs	Required for "Sign in with AFC". **Checked against the real policy the moment you submit**, so a wildcard or a query string is refused while you are still looking at the form rather than at your first failed sign-in. See below.
Post-logout redirect URIs	Optional. Only if you send players to AFC's end session endpoint (**section 12 (page 27)**). Same rules.
Disconnection webhook URL	Optional, `https`, and it must be a **publicly reachable address** (**section 13 (page 30)**). Unlike your redirect URIs, this is the one address AFC's own server calls, so `localhost` and private addresses are refused. Where AFC POSTs a signed notice that a player disconnected you.
What you are building, and what you need from AFC	Two free-text answers, and the two the decision actually turns on.

You are not asked to pick scopes, and that is deliberate. Describe what you are building and what you need it for, in your own words, and AFC works out which fields that amounts to and grants the minimum that does the job. A checklist would invite you to tick everything, and it would make the answer "approved as requested" rather than "then you need `afc.ranking` and nothing else". Write the two answers as if explaining the product to somebody, because that is what they are for.

What happens next

OUTCOME	WHAT YOU RECEIVE
Changes requested	An email naming exactly what to fix, and a link that opens your application for editing. Sending it back returns it to AFC's queue automatically. Nothing is lost and you do not reapply.
Approved	An email with a **single-use link** to collect your credentials, and your application page stays available.
Not approved	An email with the reason. You are welcome to apply again if what AFC raised changes; a rejected application is closed rather than reopened.

Collecting your credentials

AFC never emails a client secret or an API key. An inbox is permanent, gets forwarded, and is searchable by anyone who later reaches that mailbox, which makes it the worst place to leave a credential. The approval email carries a link to a page that shows the secret once instead.

Three things follow, and they are worth knowing before you click:

- **The link works once.** Opening it mints your secret and burns the link.
- **The secret is displayed a single time**, on that page. Have your password manager or your server's secret store open before you continue.
- **The link expires after 72 hours.** If you miss it, or lose the secret later, ask AFC and they will send a new one. Collecting again issues a *fresh* secret rather than revealing the old one, so anything already configured with the previous secret stops working at that moment.

That last point is also a tripwire worth understanding: because collecting rotates rather than reveals, an intercepted link is detectable. If you open your link and are told the credentials have already been collected, somebody else opened it. Tell AFC.

YOU RECEIVE	NOTES
`client_id`	A public identifier. Safe to put in a redirect URL and to keep in your configuration.
`client_secret`	**Shown once, on the credentials page.** AFC stores only a hash of it and cannot show it to you again. Keep it server side only.
An API key	Data API partners only. Same rule: shown once, stored hashed.
The set of scopes you are approved for	Anything outside this set is refused at the authorize endpoint before the player sees anything.

The redirect URI policy

Register as many redirect URIs as you need. Production, staging and each developer's local machine are all legitimate, and AFC does not ration them. There is no need to ask for a second application, and no need to come back to AFC every time you add an environment.

Matching is exact. The `redirect_uri` you send on the authorize request must be one of the registered URIs, character for character. A trailing slash difference, an added query parameter, `http` where `https` was registered, or a different port all fail.

AFC applies six rules when your URIs are registered or edited. A URI that breaks one is refused at that moment, with a message naming which URI and why, so nothing surprising happens later at sign-in time. These are the rules your list must satisfy:

#	RULE	ACCEPTED	REFUSED, WITH AFC'S MESSAGE
1	**HTTPS everywhere a real player goes.** Plain `http` is accepted for loopback only.	`https://partner.example/auth/afc/callback`, `http://localhost:3000/cb`, `http://127.0.0.1:8000/cb`, `http://[::1]:3000/cb`	`http://staging.partner.example/cb` gives *"Redirect URI 'http://staging.partner.example/cb' uses http. Plain http is only allowed for localhost and 127.0.0.1; every other address must use https."*
2	**A full absolute address, with a scheme AFC accepts.** Only `http` and `https` exist here. A path is welcome but not required.	`https://partner.example/cb`, `https://partner.example`	`partner.example/cb`, `ftp://partner.example/cb` and `javascript:alert(1)` give *"...must start with https:// (or http:// for localhost)."* `https:///cb` gives *"...is not a full address. Include the host..."*
3	**No wildcards, and therefore no query string.** `*` and `?` are both refused, anywhere in the URI.	`https://partner.example/cb`	`https://*.partner.example/cb`, `https://partner.example/*` and `https://partner.example/cb?tenant=acme` all give *"...Wildcards are not allowed: AFC matches redirect URIs exactly, so register each address in full."*
4	**No fragment.**	`https://partner.example/cb`	`https://partner.example/cb#anything` and a bare trailing `#` give *"...must not contain a '#' fragment."*
5	**No credentials in the address.** Nothing before the host.	`https://partner.example/cb`	`https://user:secret@partner.example/cb` and `https://user@partner.example/cb` give *"Redirect URI '...' contains a username or password before the host. Remove everything up to the '@': AFC would have to store that credential and send every player through it."*
6	**At least one redirect URI.** An application with none can never complete a sign-in.	any legal URI	an empty list gives *"At least one redirect URI is required."* (post-logout URIs are exempt: an empty list is legal there, and you can clear them later)

Five of those need a word of explanation.

The loopback exception exists so you can develop against your own machine without a certificate. It is safe precisely because loopback traffic never leaves that machine, so there is no network hop on which an authorization code could be intercepted. `localhost.partner.example` is a REMOTE host that merely begins with the word, and is refused: the whole host has to be loopback, not the start of it. The same goes for `127.0.0.1.partner.example` and for a bare `notlocalhost`.

Wildcards are refused rather than ignored because AFC matches exactly. A registered `https://*.partner.example/cb` could never match anything, and refusing it at registration tells you that immediately instead of leaving you to debug a redirect URI mismatch.

Rule 3 catches ordinary query strings as a side effect, and that is intended. `?` is treated as a pattern character, so `https://partner.example/cb?tenant=acme` is refused even though nothing in it is a wildcard. Register the bare path and carry per-request data in `state` instead, which is where it belongs: `state` survives the round trip, is compared by you on the callback, and does not have to be registered in advance.

Credentials are refused because AFC would have to keep them. `https://user:secret@partner.example/cb` is a legal URI, and that is exactly the problem: RFC 3986 section 3.2.1 deprecates the userinfo component because it puts a password somewhere it gets logged, copied into a support ticket and pasted into a browser bar. Registering one would have AFC storing your secret and sending every player through it. If your callback needs to authenticate the caller, do it inside your own handler, not in the URL.

A refusal changes nothing. One bad URI fails the whole list, and the application keeps the set it already had. You will not end up half updated.

The same six rules are applied on every path an AFC administrator can edit your application by, so a URI that one screen refuses is not quietly accepted by another.

Post-logout redirect URIs are registered here too, and the same six rules apply to them, because they are the other place AFC sends a real player to a URL of yours. You only need them if you send players to AFC's end session endpoint; see section 12 (page 27).

3. Discovery: bootstrap from one URL

AFC publishes a standard OpenID Connect discovery document. **Read it at runtime and drive your integration from it.** Do not hardcode endpoint paths. A compliant OIDC client library can bootstrap the entire integration from this single URL, and that is the recommended way to integrate.

```
GET https://api.africanfreefirecommunity.com/sso/.well-known/openid-configuration
```

The public base URL is `https://api.africanfreefirecommunity.com`. The SSO surface is mounted at the `/sso/` prefix on the AFC API, NOT on the `africanfreefirecommunity.com` website origin, so every path in this guide hangs off that host. Put it in an environment variable rather than in source, so you can point at a test host if AFC gives you one.

A trailing slash is accepted: `/sso/.well-known/openid-configuration` and `/sso/.well-known/openid-configuration/` both return the document. The response carries `Access-Control-Allow-Origin: *`, so it is readable from a browser.

What is in the document

Concrete example, with the values AFC's configuration actually produces:

JSON

```
{
  "issuer": "https://api.africanfreefirecommunity.com/sso",
  "authorization_endpoint": "https://api.africanfreefirecommunity.com/sso/authorize/",
  "token_endpoint": "https://api.africanfreefirecommunity.com/sso/token/",
  "userinfo_endpoint": "https://api.africanfreefirecommunity.com/sso/userinfo/",
  "jwks_uri": "https://api.africanfreefirecommunity.com/sso/.well-known/jwks.json",
  "scopes_supported": [
    "openid", "profile", "email",
    "afc.freefire", "afc.team", "afc.history",
    "afc.stats", "afc.ranking", "afc.standing"
  ],
  "response_types_supported": [
    "code", "token", "id_token", "id_token token",
    "code token", "code id_token", "code id_token token"
  ],
  "subject_types_supported": ["pairwise"],
  "id_token_signing_alg_values_supported": ["RS256", "HS256"],
  "token_endpoint_auth_methods_supported": ["client_secret_post", "client_secret_basic"],
  "code_challenge_methods_supported": ["plain", "S256"],
  "claims_supported": ["sub"],
  "prompt_values_supported": ["none", "login"],
  "client_id_metadata_document_supported": false,
  "end_session_endpoint": "https://api.africanfreefirecommunity.com/sso/logout/"
}
```

Note the `issuer` value has **no trailing slash** and **includes the `/sso` path segment**. Use it verbatim when validating the `iss` claim of an ID token.

Four things to read carefully in the discovery document

Three of these are values a partner could reasonably misread, and the first is a confirmation rather than a warning. The discovery document is generated by the underlying library, so some of its values describe what the library CAN do rather than what AFC will accept from you.

1. `subject_types_supported` **says** `["pairwise"]`**, and it means it.** The same player gets a different `sub` at every partner. Build for pairwise. See **section 8** (page 21).
2. `claims_supported` **lists only** `["sub"]`**.** This is not the claim catalogue. AFC computes claims per request, so the library cannot enumerate them at discovery time. The real catalogue is **section 6** (page 17) of this document.
3. `response_types_supported` **lists seven values, including implicit and hybrid types.** In practice partner applications are created with the authorization code grant, so `response_type=code` **is the only one that will work for you.** Do not use `token`, `id_token`, or any hybrid combination.
4. `code_challenge_methods_supported` **includes** `"plain"`**.** Do not use `plain`. Use `S256`. PKCE with `plain` provides no protection against an attacker who can observe the authorization request.

`end_session_endpoint` **is** present: RP-initiated logout is enabled, so you can offer the player "sign out of AFC too" instead of leaving them signed in at AFC after they sign out of your site. Read the endpoint from discovery like every other one. What it does, and what it deliberately does not do, is **section 12** (page 27).

Two things you will **not** find in the discovery document, because AFC does not implement them: `backchannel_logout_supported` and `frontchannel_logout_supported`. AFC does not call you when a player's session ends elsewhere. The one thing AFC does push to you is the disconnection webhook in **section 13** (page 30), and that is registered with AFC rather than advertised in discovery.

4. Endpoint reference (exact paths)

All paths are relative to the AFC API base URL. Every path below was read from AFC's URL configuration, not assumed from library defaults. **Note the trailing slashes: they are part of the path.**

PURPOSE	METHOD	PATH	AUTH
Discovery document	GET	`/sso/.well-known/openid-configuration`	none
JSON Web Key Set	GET	`/sso/.well-known/jwks.json`	none
Authorization	GET	`/sso/authorize/`	player's AFC browser session
Token	POST	`/sso/token/`	client credentials
UserInfo	GET or POST	`/sso/userinfo/`	`Authorization: Bearer <access_token>`
Token revocation	POST	`/sso/revoke_token/`	client credentials
End session (sign the player out)	GET or POST	`/sso/logout/`	`id_token_hint` identifies you

Notes on each:

- `/sso/authorize/` **is AFC's own view, not the library's.** AFC mounts its consent screen at this path ahead of the library's route, so it takes precedence. This is where the four refusal checks in section 10 (page 23) run.
- **The revocation path is** `/sso/revoke_token/`**, not** `/sso/revoke/`**.** This trips people up. It follows RFC 7009 semantics.
- `/sso/introspect/` is registered by the underlying library but is **not part of the partner contract**. Do not build against it. It is not part of the partner contract today. Ask info@africanfreefirecommunity.com if you have a case for it.
- `/sso/logout/` **is AFC's own view too, not the library's**, for the same reason: the library's version deletes every token the player holds at **every** partner, and AFC's replaces that with a deletion scoped to the partner that asked. It is the `end_session_endpoint` in discovery. See **section 12 (page 27)**.
- `/sso/register/` (Dynamic Client Registration) is **disabled**. Applications are created by an AFC admin only.
- `/sso/me/connected-apps/` exists but is **not for partners**. It is AFC's own player-facing API behind the Connected Apps page in the player's AFC profile, and it authenticates with an AFC session token, not with your OAuth tokens.

5. The authorization code flow with PKCE, step by step

PKCE is REQUIRED. An integration without it will fail.

AFC sets `PKCE_REQUIRED: True`. This is not advisory and it is not only for public clients. **Even though you are a confidential client with a `client_secret`, you must send `code_challenge` and `code_challenge_method` on the authorization request and `code_verifier` on the token request.** If you omit them, the authorization request is rejected or the token exchange fails. This is the single most common cause of a first integration attempt failing.

Generating the verifier and challenge:

The `code_verifier` is a high-entropy random string, 43 to 128 characters, from the URL-safe alphabet. The `code_challenge` is the base64url encoding (no padding) of the SHA-256 digest of the verifier.

Node.js:

```javascript
import crypto from "node:crypto";

const base64url = (buf) =>
  buf.toString("base64").replace(/\+/g, "-").replace(/\//g, "_").replace(/=+$/, "");

const codeVerifier = base64url(crypto.randomBytes(32));                  // 43 chars
const codeChallenge = base64url(
  crypto.createHash("sha256").update(codeVerifier).digest()
);                                                                       // 43 chars
```

Python:

```python
import base64, hashlib, secrets

code_verifier = base64.urlsafe_b64encode(secrets.token_bytes(32)).decode().rstrip("=")
code_challenge = base64.urlsafe_b64encode(
    hashlib.sha256(code_verifier.encode()).digest()
).decode().rstrip("=")
```

Shell (for manual testing):

```shell
CODE_VERIFIER=$(openssl rand -base64 32 | tr '+/' '-_' | tr -d '=')
CODE_CHALLENGE=$(printf '%s' "$CODE_VERIFIER" | openssl dgst -binary -sha256 | openssl base64 | tr '+/' '-_' | tr -d '=')
```

Store the verifier server side, bound to this one browser session, in an httpOnly cookie or a server-side session. You need it again at step 4, and only the browser that started the flow may complete it.

Step 1: Redirect the player to the authorization endpoint

```
GET https://api.africanfreefirecommunity.com/sso/authorize/
    ?response_type=code
    &client_id=<your client_id>
    &redirect_uri=https%3A%2F%2Fpartner.example%2Fauth%2Fafc%2Fcallback
    &scope=openid%20profile%20afc.freefire
    &state=<random, unguessable, stored in the session>
    &nonce=<random, unguessable, stored in the session>
    &code_challenge=<the S256 challenge>
    &code_challenge_method=S256
```

PARAMETER	REQUIRED	VALUE
`response_type`	yes	`code`. Nothing else works for a partner application.
`client_id`	yes	The identifier AFC issued you.
`redirect_uri`	yes	Must match a registered URI exactly, and must be URL-encoded in the query string.
`scope`	yes	Space-separated. **Must include** `openid`. Everything else is from **section 6** (page 17).
`state`	yes in practice	CSRF defence. Random, stored in the session, compared on the callback.
`nonce`	strongly recommended	Random, stored in the session, compared against the `nonce` claim in the ID token. Replay defence.
`code_challenge`	**yes**	The base64url SHA-256 of your verifier.
`code_challenge_method`	**yes**	`S256`.
`approval_prompt`	no	Send `auto` to ask AFC to reuse an existing approval and skip the consent screen. AFC's default is to show the screen every time. See **section 9** (page 22) for when `auto` is ignored.

What the player sees. If they are not signed in to AFC, AFC bounces them to the AFC website login carrying the full authorization request, then returns them here. That extra hop is normal and invisible to you. Once signed in, they get AFC's consent screen: your organisation's name and logo, a plain-language list of exactly what you will be able to see, and Allow / Deny buttons. The screen renders in the player's own language (English, French or Portuguese).

Step 2: AFC redirects back to your `redirect_uri`

On approval:

```
https://partner.example/auth/afc/callback?code=abc123...&state=<the state you sent>
```

On denial, or on a protocol-level error that AFC considers safe to redirect (see **section 10** (page 23) for those it does not):

```
https://partner.example/auth/afc/callback?error=access_denied&state=<the state you sent>
```

Before doing anything else, compare the returned `state` against the value you stored. If they differ, or the stored value is missing, abort. Do not exchange the code.

Step 3: Exchange the code for tokens

The authorization code is **single use** and **expires in 60 seconds**. Exchange it immediately, server side.

```shell
curl -X POST https://api.africanfreefirecommunity.com/sso/token/ \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=authorization_code" \
  -d "code=abc123..." \
  -d "redirect_uri=https://partner.example/auth/afc/callback" \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET" \
  -d "code_verifier=$CODE_VERIFIER"
```

`redirect_uri` must be **byte-identical** to the one you sent at step 1. `code_verifier` is the raw verifier, not the challenge.

Client authentication: AFC accepts `client_secret_post` (the form fields shown above) and `client_secret_basic` (HTTP Basic with `client_id` as the username and `client_secret` as the password). Either is fine. Pick one.

Response, 200:

```json
{
  "access_token": "kR2h...",
  "expires_in": 36000,
  "token_type": "Bearer",
  "scope": "openid profile afc.freefire",
  "refresh_token": "9Xz4...",
  "id_token": "eyJhbGciOiJSUzI1NiIsImtpZCI6..."
}
```

Note the `scope` field: it is the scope actually granted. It may be narrower than what you asked for. Read it rather than assuming.

Response, 400, for a replayed code, an expired code, a wrong verifier, a mismatched `redirect_uri`, or bad client credentials. The body is the standard OAuth 2.0 error object, for example `{"error": "invalid_grant"}`.

Step 4: Verify the ID token

The ID token is a JWT signed with **RS256**, using a key published at the JWKS endpoint. Verify it. Do not decode it without verifying.

Check all of:

CHECK	EXPECTED VALUE
Signature	Valid against a key from `jwks_uri`, matched by the `kid` in the JWT header.
`iss`	Exactly the `issuer` from discovery, for example `https://api.africanfreefirecommunity.com/sso`.
`aud`	Your `client_id`.
`exp`	In the future.
`nonce`	Equal to the nonce you generated at step 1.

Decoded payload (claims beyond the standard set depend on your scopes and the four gates):

```json
{
  "iss": "https://api.africanfreefirecommunity.com/sso",
  "aud": "YOUR_CLIENT_ID",
  "sub": "3f7a1c9e0b5d4a28f61c8e0d7b3a5f92c4e18d60a7b2f34c9d1e5a08b6c7f2d3",
  "exp": 1786000000,
  "iat": 1785964000,
  "auth_time": 1785964000,
  "nonce": "the nonce you sent",
  "at_hash": "...",
  "preferred_username": "PlayerName",
  "country": "NG",
  "locale": "en",
  "ff_uid": "8390224792"
}
```

AFC puts the same claims in the ID token as at the userinfo endpoint. This is deliberate: the two are computed by one function so they cannot disagree. You may read the profile data straight out of the verified ID token on first login and skip the userinfo call.

Step 5: Read the userinfo endpoint

Use this later, whenever you want fresh data. The claims are recomputed on every call, so a field AFC has since switched off disappears here even though your access token is still valid.

```shell
curl https://api.africanfreefirecommunity.com/sso/userinfo/ \
  -H "Authorization: Bearer kR2h..."
```

Response, 200:

```json
{
  "sub": "3f7a1c9e0b5d4a28f61c8e0d7b3a5f92c4e18d60a7b2f34c9d1e5a08b6c7f2d3",
  "preferred_username": "PlayerName",
  "country": "NG",
  "locale": "en",
  "ff_uid": "8390224792"
}
```

`GET` and `POST` both work. An expired, revoked or unknown access token returns **401** with a `WWW-Authenticate: Bearer` header. The exact error body is produced by the underlying OAuth library and is not something AFC customises, so **branch on the status code, not on the body text**.

6. The scope catalogue and the exact claims each one produces

These are all nine scopes AFC defines. There are no others. Request only what you need: every extra scope is another thing the player can refuse, and a wider request forces a fresh consent screen.

The claim names are not what a reader would guess. AFC uses `ff_uid`, `afc_team`, `afc_history`, `afc_stats`, `afc_ranking` and `afc_standing` (underscores), while the scope names use dots (`afc.freefire`, `afc.team`, and so on). Copy the claim names from this table exactly.

SCOPE	WHAT THE PLAYER IS TOLD THEY ARE SHARING	CLAIMS PRODUCED	SHAPE OF THE VALUE
`openid`	"Confirm who you are on AFC"	`sub`	String. 64-character lowercase hex (a SHA-256 digest). Pairwise per partner. **Always present.**
`profile`	"Your in-game name, avatar, country and language"	`preferred_username`, `picture`, `country`, `locale`	`preferred_username`: string, the AFC username. `country`: free-text string, commonly a two-letter code such as `"NG"` but **not schema-enforced**, so treat it as opaque. `locale`: string, one of `"en"`, `"fr"`, `"pt"`. `country` and `locale` are **omitted** when the player has not set them.
`email`	"Your email address"	`email`, `email_verified`	`email`: string. `email_verified`: boolean, and it is **real**: it reflects whether the player completed AFC's emailed activation code.
`afc.freefire`	"Your Free Fire UID"	`ff_uid`	String, the player's Free Fire UID. **Omitted** when the player has no UID on file.
`afc.team`	"Your current team and your role in it"	`afc_team`	Object: `{"name": "Team Name", "role": "captain"}`. `role` falls back to `"member"` when no management role is set. **The whole claim is omitted** when the player is not on a team.
`afc.history`	"Tournaments you have played"	`afc_history`	Array of objects, newest registration first, **capped at 50 entries**: `[{"event": "Event Name", "slug": "event-slug"}]`. Empty array when the player has no history. **No placement or result data is included.**
`afc.stats`	"Your match statistics"	`afc_stats`	Object: `{"matches": 128, "kills": 940}`. Both integers, `0` when the player has no recorded matches. **Only these two figures.**
`afc.ranking`	"Your AFC rank and ranking points"	`afc_ranking`	Object for the most recent scored month: `{"points": 1420, "rank": 17, "month": "2026-07-01"}`. `month` is an ISO date string. **The whole claim is omitted** when the player has never been scored. **No tier is included** (tier is a team attribute on AFC, not a player one).
`afc.standing`	"Whether your AFC account is in good standing"	`afc_standing`	Object: `{"in_good_standing": true}`. `false` when the player is banned or their account is not active.

Three places the scope description overpromises

The consent text is what the player reads; the claim table above is what your code actually receives. Where they differ, the table wins:

- `profile` **releases an avatar as the standard** `picture` **claim** (added 2026-08-03). It is an absolute URL on AFC's API domain. It is **omitted** for players who have not uploaded one, so render a fallback.
- `afc.history` **releases event name and slug only.** No placement, no result, no date.
- `afc.stats` **releases match count and kill count only.** No damage, assists, or per-event breakdown.

`email_verified` is meaningful

`email_verified` is derived from the player's account being active, which on AFC means they completed the emailed activation code at signup (accounts created through Google are active immediately, because Google already verified the address). It is safe to gate account linking on it. It was previously hardcoded to `true`; that was corrected on 2026-08-03.

Claims with no value are absent, not null

AFC strips every claim whose value would be `null` before sending the payload. So an AFC player with no team produces a response with **no `afc_team` key at all**, rather than `"afc_team": null`. Write your parsing accordingly: check for key presence, and never assume `response.afc_team.name` is reachable.

7. The four gates: why a scope you requested can return nothing

This is the part of AFC's model that surprises integrators, so it is stated bluntly. **A field reaches you only if all four of these agree. If any one refuses, the claim is simply absent from the payload. It is not `null`, and it is not an error.**

GATE	CONTROLLED BY	MEANING
1. AFC's per-partner toggle	An AFC admin	Is this field switched on for *your* organisation? Every toggle defaults OFF.
2. The requested scope	You	Did you actually ask for it in this authorization request?
3. Player consent	The player	Did this player press Allow for that scope, and is the grant still live?
4. AFC's own rules	AFC policy, evaluated live	The rules below.

Gate 4, in full

These are applied on **every** ID token issuance and **every** userinfo call, against the player's state at that moment.

1. **Under-18 players never release contact data.** If the player's date of birth puts them under 18, the `email` scope is dropped, whatever your toggle says and whatever the player consented to. **This fails closed:** a player with no date of birth on file, or an unparseable one, is treated as a minor and gets no `email` either. Expect a meaningful share of your players to have no `email` claim.
2. **A player with statistics hidden releases no statistics.** AFC players have a `stats_visible` setting. When it is off, the `afc.stats` scope is dropped and no `afc_stats` claim appears. This too fails closed: a player for whom the flag is unset is treated as hidden.
3. **A suspended account releases only its standing.** If the AFC account is not active, the effective scope set collapses to `openid` plus `afc.standing`. You still get `sub`, and you still get `afc_standing` (which will report `in_good_standing: false`) if you were approved for it. **Everything else disappears**, including profile and email, even though the token itself is still valid. This is deliberate: whether a player is in good standing is the one question a partner has a legitimate reason to ask about a suspended account.

Note the interaction with the authorize endpoint: a player whose account is already suspended cannot start a new sign-in at all (they get a refusal, see section 10 (page 23)). Gate 4 rule 3 is what happens to a player who is suspended **after** you already hold their tokens.

Gates are re-evaluated live, on every call

There is no caching. If AFC switches off your `afc.freefire` toggle this afternoon, the `ff_uid` claim vanishes from your **next userinfo call** with a token issued this morning. You do not get an error and you do not get a token invalidation. The field is just no longer there.

You do not get a webhook either. The disconnection webhook in section 13 (page 30) fires when a player disconnects you altogether, not when a single field stops being shared, so a claim quietly disappearing is something your code has to notice on its own.

What this means for your code

```javascript
// WRONG. Throws for a player with no team, or one whose scope was gated.
const teamName = userinfo.afc_team.name;

// WRONG. "" is a plausible value for country; this silently mislabels it.
const country = userinfo.country || "unknown";

// RIGHT.
const teamName = userinfo.afc_team?.name ?? null;
const hasTeam = "afc_team" in userinfo;
const country = "country" in userinfo ? userinfo.country : null;
```

Design your data model so that every AFC field except `sub` is nullable, and so that a field disappearing on a later refresh is a normal event rather than a data-integrity failure. Do not, for example, wipe a stored team name just because one userinfo call omitted it: decide explicitly whether absence means "no longer applies" or "no longer shared".

8. The `sub` claim is pairwise: the only safe key to store

`sub` is the primary key for an AFC player on your side. Store it. Key on it. Never key on anything else.

AFC issues **pairwise** subject identifiers. The `sub` for one player is derived from AFC's server secret, your application's internal record and the player's internal record, and it is a 64-character SHA-256 hex digest. The consequences:

- **The same player has a different `sub` at every partner.** Two partner sites cannot compare their `sub` values to work out that they are looking at the same person. That is the point.
- `sub` **is opaque.** It encodes nothing you can decode, and you cannot compute it yourself.
- `sub` **is stable for you** across every sign-in, token refresh and userinfo call, for as long as the player's AFC account and your application record both exist.

Store it as a string of exactly 64 characters and treat it as an opaque token.

Do not key on email or username

Both change, and both will break your account linking:

- **Email** changes when a player updates their AFC account, and is **absent entirely** for every player under 18 and for every player whose date of birth AFC does not have. You cannot rely on receiving it at all.
- **Username** (`preferred_username`) is an in-game name that AFC players do change.

If you already have local accounts and want to link them to AFC accounts by email on first sign-in, that is a reasonable one-time bootstrap. But persist the `sub` at that moment and match on `sub` from then on.

Operational caveat to be aware of

AFC's `sub` derivation is tied to a server-side secret. AFC has documented internally that rotating that secret would change every partner's view of every player and break all existing account links. AFC is aware of this and it is not expected to happen, but if you ever see every returning player arrive as a new `sub`, that is the cause and you should contact AFC rather than merging accounts. If you ever see this, contact info@africanfreefirecommunity.com immediately rather than merging accounts.

9. Re-consent when you widen your scopes

If you request a scope the player has not already approved for you, AFC shows the consent screen again. You cannot suppress this.

The rule as implemented:

- AFC compares the scopes on this request against the scopes on the player's **live, unexpired tokens** for your application. Expired tokens do not count as evidence of consent.
- If the request asks for **anything** not in that set, AFC forces the consent screen.
- **Sending `approval_prompt=auto` does not suppress it.** AFC overrides your `auto` with `force` in exactly this case. This is enforced in AFC's own code rather than left to the library, specifically so that a configuration change cannot relax it.
- Narrowing does not re-prompt. Asking for fewer scopes than the player already approved is fine.
- By default, AFC shows the consent screen **on every authorization**, whether or not anything widened. Sending `approval_prompt=auto` is what asks AFC to reuse a live approval, and AFC honours that only when nothing widened.

Two practical consequences:

1. **You cannot silently gain a scope.** If AFC switches on a new toggle for you, the first time you actually request that scope the player is asked again. Plan the user-facing moment: a player mid-task being bounced to an AFC consent screen needs to understand why.
2. **Do not request scopes speculatively.** Requesting the full catalogue "in case we need it later" means a longer consent list, more refusals, and more absent claims.

There is also no per-partner bypass. AFC pins off the underlying library's "skip the consent screen" flag on every save, because every partner application is a third party by definition.

10. Errors and refusals

AFC refusals render a page, they do not redirect

Four checks run at `/sso/authorize/` **before** anything is rendered and before any code is issued. When one fails, AFC returns **HTTP 400 with a plain-text message rendered on AFC's own page**. It does **not** redirect to your `redirect_uri` with an `error` parameter.

This is deliberate, and it is a security property, not a bug. Bouncing a refusal to whatever `redirect_uri` sits in the query string would make AFC an open redirector: an attacker could craft a link that fails on purpose and lands the player on a phishing page carrying an AFC referrer. Nothing in a refusal path echoes attacker-controlled input into a `Location` header. AFC has an explicit regression test asserting that no refusal ever returns a 302.

What this means for you: a refused authorization never comes back to your callback. The player is left on an AFC error page. You will see the flow start and never complete. Do not build a "the callback always fires" assumption into your session handling; time out pending flows.

The four refusals, with the exact English messages:

CONDITION	HTTP	MESSAGE
The `client_id` matches no application	400	`Unknown application.`
Your application's status is suspended	400	`This application is suspended.`
The player's AFC account is not active	400	`Your AFC account cannot sign in to partner sites.`
You requested a scope your organisation is not approved for	400	`This application requested data it is not approved for.`

Do not string-match these messages. They are translated into the player's language, so a French player sees the French text. They are shown to a human, not parsed by a machine. If you need to detect these conditions, detect the absence of a callback.

The scope refusal is worth understanding: AFC checks your requested scopes against your approved set **at the front door**, before the consent screen. The claim-building layer would strip an over-broad scope anyway, but letting the request through would show the player a consent screen listing data your organisation is not permitted to receive. So a request for even one unapproved scope fails the whole authorization. Keep your requested scope list in sync with what AFC approved for you.

Redirect URI mismatch

A `redirect_uri` that does not match a registered URI is rejected by the underlying library, and it also does not redirect (verified by test). The player sees an error page at AFC. Same practical consequence: no callback fires.

Errors that do come back to your callback

Once the request is valid and the consent screen has rendered, ordinary OAuth errors return to your `redirect_uri` as query parameters. The one you will actually see is the player pressing Deny:

```
https://partner.example/auth/afc/callback?error=access_denied&state=<your state>
```

Always handle `error` on the callback before looking for `code`.

Token endpoint errors

Standard OAuth 2.0 JSON errors with HTTP 400, for example:

```json
{"error": "invalid_grant"}
```

Common causes: the code was already used (codes are single use and AFC tests this explicitly), the code expired (60 seconds), the `code_verifier` does not match the `code_challenge`, the `redirect_uri` differs from step 1, or the client credentials are wrong.

UserInfo endpoint errors

401 for an expired, revoked or unknown access token, with a `WWW-Authenticate: Bearer` header. Branch on the status code. On 401, refresh the access token and retry once; if the refresh also fails, treat the player as disconnected and prompt them to sign in with AFC again.

11. Token lifetimes, refresh and revocation

These are the values in AFC's configuration as it stands. AFC does not override the library defaults for any of them, so they are the library defaults. **Read `expires_in` from the token response rather than hardcoding these, in case AFC tunes them.**

ITEM	LIFETIME	NOTES
Authorization code	**60 seconds**	Single use. Exchange it immediately.
Access token	**36000 seconds (10 hours)**	Returned as `expires_in`.
ID token	**36000 seconds (10 hours)**	Same as the access token.
Refresh token	**No expiry**	It remains valid until it is used (and rotated), revoked by you, revoked by the player, or deleted when AFC suspends your application.

Refresh tokens rotate. You must store the new one.

AFC has refresh token rotation enabled, with a **zero-second grace period**. Every successful refresh returns a **new** `refresh_token` and invalidates the one you sent, immediately, with no overlap window.

```shell
curl -X POST https://api.africanfreefirecommunity.com/sso/token/ \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "grant_type=refresh_token" \
  -d "refresh_token=9Xz4..." \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET"
```

The response has the same shape as the authorization code exchange, including a fresh `refresh_token`.

Implications you must design for:

- **Persist the new refresh token in the same transaction that consumes the old one.** If your process crashes between the HTTP response and the database write, that player's connection is dead and they must sign in again.
- **Serialise refreshes per player.** Two concurrent refreshes with the same token means one succeeds and the other gets a 400, and depending on ordering you can lose the valid token. Use a lock or a single-flight pattern.
- There is no grace period, so "retry the same refresh token on a network timeout" is not safe. If a refresh times out, re-read your stored token before retrying.

Note that `scope` is not a parameter you send on refresh: the new tokens carry the scopes the previous ones did. To change scopes, run a fresh authorization (which will re-prompt if you widened, see section 9 (page 22)).

Revoking a token from your side

RFC 7009, at `/sso/revoke_token/`:

```shell
curl -X POST https://api.africanfreefirecommunity.com/sso/revoke_token/ \
  -H "Content-Type: application/x-www-form-urlencoded" \
  -d "token=9Xz4..." \
  -d "client_id=YOUR_CLIENT_ID" \
  -d "client_secret=YOUR_CLIENT_SECRET"
```

Revoking the refresh token kills the ability to mint new access tokens (AFC has a test asserting a subsequent refresh returns 400). Do this when a user deletes their account on your side or unlinks AFC, as a matter of hygiene.

The player can revoke you at any time, without warning

Every AFC player has a Connected Apps page in their AFC profile listing the partner organisations they have signed in to, what each one can see, and a Remove button. Pressing Remove deletes, for your application and that player: all access tokens, all refresh tokens, any unexchanged authorization codes, and all issued ID tokens.

If you have registered a disconnection webhook, AFC tells you. A signed notice is POSTed to your endpoint so you can delete your own copy of that player's data. That is section 13 (page 30), and registering one is the difference between learning about a revocation in seconds and learning about it the next time you happen to make a call.

If you have not registered one, you get no notification. The first you know of it is a 401 from userinfo and a 400 from refresh. Either way, handle a dead connection as a normal state transition (mark the AFC connection as disconnected, invite the player to reconnect), not as an outage: the webhook is best effort and delivery is never guaranteed, so a failing token remains the reliable signal.

The same is true if AFC suspends your application: authorization requests start returning `This application is suspended.` A suspension does **not** fire the webhook, because the player did nothing and their consent still stands.

Signing the player out

Signing a player out of your site does not end their AFC session on its own, but AFC does publish an end session endpoint you can send them to, so you can offer "sign out of AFC too". It ends the AFC session on that device and disconnects you, and only you. See section 12 (page 27).

12. Signing the player out of AFC

AFC publishes an `end_session_endpoint`, so a player signing out of your site can sign out of AFC in the same gesture. Read the URL from discovery; today it resolves to `/sso/logout/`.

This is a browser redirect flow, not a back channel API. You end your own session first, then send the player's browser to AFC. AFC never calls you back as part of it.

The request

```
GET https://api.africanfreefirecommunity.com/sso/logout/
    ?id_token_hint=<the ID token AFC issued you for this player>
    &post_logout_redirect_uri=https%3A%2F%2Fpartner.example%2Fsigned-out
    &state=<random, unguessable, stored in your session>
```

PARAMETER	REQUIRED	VALUE
`id_token_hint`	in practice yes	An ID token AFC issued **you** for **this** player. It is what tells AFC whose session to end and which partner is asking. **An expired one is accepted**, deliberately: a player signing out an hour after they signed in is the normal case, and the hint authorises nothing on its own. Without it (and without `client_id`) AFC cannot tell who is asking and will disconnect nobody.
`client_id`	no	Your `client_id`. If you send both, it must match the one inside `id_token_hint`, or the request is refused.
`post_logout_redirect_uri`	no	Where AFC returns the player afterwards. **Must already be registered on your application** as a post-logout redirect URI, and matching is exact. Omit it and AFC lands the player on the AFC API root instead.
`state`	no	Opaque value, echoed back as a `state` query parameter on the redirect. Use it the same way you use `state` on the sign-in flow.

`logout_hint` and `ui_locales` are part of the OpenID spec but AFC acts on neither. The confirmation page below renders in the player's own AFC language preference, falling back to their `Accept-Language`.

What the player sees

If AFC recognises the player's browser session, it asks first. AFC renders a confirmation page in the player's language, styled like the consent screen: a heading naming your organisation ("*Your Org* is signing you out"), the question "Do you also want to sign out of AFC on this device?", a **Sign out of AFC** button, a **Stay signed in** button, and the note "Signing out here ends your AFC session on this device and disconnects only the site that asked. Your other connected apps are not affected."

Nothing is deleted until the player presses **Sign out of AFC**. Pressing **Stay signed in** refuses the whole request: AFC deletes nothing, ends nothing, and renders a 400 page carrying the error code `logout_denied`. There is no redirect back to you on that path, so do not wait for one.

This prompt is not optional and you cannot suppress it. It is what stops a partner ending a player's AFC session silently, for instance from a hidden iframe.

If AFC does not recognise a browser session (the player's AFC session has already lapsed, or your own server made the call rather than the player's browser), there is nobody to ask, so AFC completes immediately.

What AFC does

AFC DELETES	FOR WHOM
Access tokens, refresh tokens, unexchanged authorization codes, issued ID tokens	The player, at **the partner that asked**, and nobody else
The player's AFC session on that one device	Only when a signed-in player pressed the button themselves

Only you are disconnected. This is the guarantee worth reading twice, in both directions: you cannot use this endpoint to cut a player off from another partner, and no other partner can use it to cut them off from you. It is exactly the same disconnection the player performs by pressing Remove on their AFC Connected Apps page, narrowed to your application.

Only that one device is signed out of AFC. The button says "on this device" and it means it. The player's phone stays signed in to AFC.

A call from your server disconnects you but signs nobody out. Reaching this endpoint without the player's browser deletes your tokens for that player, which makes it equivalent to revoking them at `/sso/revoke_token/`, but it cannot end their AFC session. Otherwise every partner holding a token would have a remote sign-out button for that player.

Sending neither `id_token_hint` nor `client_id` deletes nothing. AFC will not guess which partner is asking.

No webhook fires. The disconnection webhook in section 13 (page 30) is not sent for a logout you initiated, because you already know.

What comes back

OUTCOME	RESPONSE
Done, and you sent a registered `post_logout_redirect_uri`	**302** to that URI, with your `state` appended as a query parameter
Done, and you sent no `post_logout_redirect_uri`	**302** to the AFC API root
The player is signed in to AFC and has not answered yet	**200**, the confirmation page
The player pressed **Stay signed in**	**400** page, error code `logout_denied`. Nothing deleted
`id_token_hint` is malformed, unverifiable, or not an ID token AFC issued	**400** page, error code `invalid_request`. Nothing deleted
`client_id` does not match the one in `id_token_hint`	**400** page, error code `invalid_request`. Nothing deleted
`post_logout_redirect_uri` is not registered on your application	**400** page, error code `invalid_request`. Nothing deleted, and **no redirect**

That last row matters and it is the same principle as the authorization refusals in section 10 (page 23): AFC will not bounce a player to an unregistered URL, because doing so would turn AFC into an open redirector. Register the URL first.

Implementing it

1. Ask AFC to register your post-logout redirect URI. It goes through the same policy as your redirect URIs (**section 2 (page 6)**), so it must be `https` outside loopback, with no wildcard, no query string and no fragment.
2. **Store the `id_token` from the token response**, alongside the access and refresh tokens. You need it here, and it is the one part of the sign-in most integrations throw away.
3. On sign-out, destroy your own session first. Whatever the player chooses at AFC, they are signed out of your site.
4. Then redirect the browser to the `end_session_endpoint` from discovery, carrying `id_token_hint`, your registered `post_logout_redirect_uri` and a `state`.

TYPESCRIPT

```
// app/api/auth/afc/logout/route.ts
// Ends OUR session, then offers the player the chance to end their AFC one.
import { redirect } from "next/navigation";
import { discovery, AFC_CLIENT_ID } from "@/lib/afc";

export async function POST() {
  const { idToken } = await readSession();   // stored at sign-in, see step 2
  await destroyOurSession();                 // our session goes now, unconditionally

  const doc = await discovery();             // end_session_endpoint comes from discovery
  const url = new URL(doc.end_session_endpoint);
  if (idToken) url.searchParams.set("id_token_hint", idToken);
  url.searchParams.set("client_id", AFC_CLIENT_ID);
  url.searchParams.set("post_logout_redirect_uri", "https://partner.example/signed-out");
  url.searchParams.set("state", await newStateBoundToThisBrowser());

  redirect(url.toString());
}
```

`discovery()` and `AFC_CLIENT_ID` are the shared helper from **section 14 (page 35)**, whose `Discovery` type already declares `end_session_endpoint`.

If a player has no `id_token` stored (an older connection, say), do not send them here with nothing. Sign them out of your site and leave AFC alone: a request AFC cannot attribute disconnects nobody and only confuses the player.

13. The disconnection webhook

Register a webhook and AFC tells you the moment a player disconnects you, so you can delete your copy of their data. This is optional. Partners without one are unaffected and simply receive nothing.

Cutting you off inside AFC only stops future reads. The player's data already sitting in your database is beyond AFC's reach, so without this signal a player's "Remove" is only half true. That is what this exists to fix, and honouring it is a condition of holding AFC data (see section 16 (page 42)).

Registering

Give AFC one `https` URL on a **publicly reachable address**. AFC stores it against your application. The application form at `https://africanfreefirecommunity.com/partners/apply` has a field for it, so the simplest moment to register one is when you apply. If you are already a partner, or you decide you want one later, email **info@africanfreefirecommunity.com**.

This field's rules are stricter than your redirect URIs', and the reason is worth knowing. A redirect URI is followed by the PLAYER's browser, so `http://localhost:3000/cb` reaches your own laptop and is a perfectly normal thing to register while you build. This webhook is the one address AFC's OWN SERVER calls, from inside AFC's network, so the same string would point AFC at AFC. Therefore:

- Plain `http` is refused, with no loopback exemption. A signed token is POSTed to this address and clear text would hand it to anybody on the path.
- `localhost`, and any private, loopback or link-local address, is refused.
- **AFC does not follow redirects.** If your endpoint answers `301` or `302`, the signal is treated as failed and is not retried. Register the address that actually receives the POST.

To test a webhook against a local server, use a tunnel that gives you a public `https` hostname rather than pointing AFC at your machine directly.

The endpoint must be reachable without authentication, since AFC sends no credentials. It does not need to be secret: the payload is signed, and verifying that signature is what tells you a request is genuinely from AFC. Treat an unverifiable POST as hostile.

When it fires

EVENT	`REASON`	MEANING
The player pressed Remove on their AFC Connected Apps page	`player_revoked`	Their AFC account still exists. They may connect you again later, and they would arrive with the same `sub`.
The player's AFC account was deleted	`account_deleted`	One signal to every partner they were connected to. The account is gone. If that person ever registers with AFC again, they are a **new** account and reach you as a **new** `sub`, because the pairwise subject is derived from the AFC account itself.

It does **not** fire when: AFC suspends your application; an AFC administrator switches one of your field toggles off; a token merely expires; you revoke a token yourself at `/sso/revoke_token/`; or you send the player through the end session endpoint in section 12 (page 27). In that last case you initiated the disconnection, so AFC does not tell you what you just did.

It also does not fire for a player who was not actually connected to you. Pressing Remove against an organisation they never used revokes nothing and signals nothing.

Note that the account-deletion sweep is deliberately wider than the player's Connected Apps list: that page hides connections whose tokens have lapsed, because a player has nothing to act on there, but a partner whose token merely expired still holds the player's data and still has to be told to delete it.

The request AFC makes

```
POST https://partner.example/afc/disconnected
Content-Type: application/secevent+jwt

eyJhbGciOiJSUzI1NiIsImtpZCI6...   <- the body IS the token, nothing wraps it
```

The body is a compact JWS, modelled on RFC 8417 Security Event Tokens. It is not JSON, so do not run it through a JSON body parser. Read the raw body as text.

Decoded header:

JSON

```
{
  "alg": "RS256",
  "kid": "the same kid AFC publishes at jwks_uri",
  "typ": "secevent+jwt"
}
```

Decoded claims:

JSON

```
{
  "iss": "https://api.africanfreefirecommunity.com/sso",
  "aud": "YOUR_CLIENT_ID",
  "iat": 1786000000,
  "jti": "1f0c8a5e-3d47-4a1b-9c22-7e6d5b8f0a34",
  "sub": "3f7a1c9e0b5d4a28f61c8e0d7b3a5f92c4e18d60a7b2f34c9d1e5a08b6c7f2d3",
  "events": {
    "https://africanfreefirecommunity.com/secevent/player-disconnected": {
      "subject": { "subject_type": "opaque", "id": "3f7a1c9e0b5d4a28...b6c7f2d3" },
      "reason": "player_revoked"
    }
  }
}
```

CLAIM	VALUE
`iss`	Identical to the `issuer` in discovery and to the `iss` of every ID token you have verified.
`aud`	Your `client_id`.
`iat`	Issued-at, seconds since the epoch. There is **no** `exp`: a security event does not stop being true.
`jti`	A UUID, unique per signal and **stable across retries**. This is your deduplication key.
`sub`	The **pairwise** subject for this player at your application, the same 64-character hex string you have been storing since they first signed in (**section 8 (page 21)**). It is never the AFC user id.
`events`	One entry, keyed by the event type URI `https://africanfreefirecommunity.com/secevent/player-disconnected`. Its value carries `subject` (repeating the same `sub`) and `reason`.

The event type is AFC's own URI rather than one of the OpenID RISC types. That is deliberate: "the player disconnected this partner" is not any of the RISC events, and claiming a RISC type AFC does not implement in full would be a lie you might build on. Branch on that URI string; it is stable.

Verifying it

This is an RS256 JWT signed with the same key as your ID tokens, so you already have the machinery. There is no shared secret and no signature header, for two reasons: AFC stores only a hash of your `client_secret` and could not compute an HMAC with it even if it wanted to, and asymmetric signing means one partner cannot forge a signal to another. A shared secret would only ever prove that somebody who knows the secret sent it.

1. Read `kid` from the JOSE header.
2. Fetch AFC's key set from the `jwks_uri` in discovery, the very same key set you use for ID tokens, and select the key with that `kid`.
3. Verify the RS256 signature.
4. Check `iss` equals the `issuer` from discovery.
5. Check `aud` equals your `client_id`.
6. Only then act on it.

TYPESCRIPT

```ts
// app/api/afc/disconnected/route.ts
// AFC POSTs here when a player disconnects us or deletes their AFC account.
// Registered with AFC as this application's disconnection webhook URL.
// Verified against the SAME JWKS used for ID tokens: see lib/afc.ts in section 14.
import { jwtVerify } from "jose";
import { afcJwks, AFC_ISSUER, AFC_CLIENT_ID } from "@/lib/afc";

const EVENT = "https://africanfreefirecommunity.com/secevent/player-disconnected";

export async function POST(req: Request) {
  // The body is the JWT itself, not JSON. Read it as text.
  const token = await req.text();

  let claims;
  try {
    ({ payload: claims } = await jwtVerify(token, await afcJwks(), {
      issuer: AFC_ISSUER,
      audience: AFC_CLIENT_ID,
    }));
  } catch {
    // Not from AFC, or tampered with. 400 is correct and AFC will not retry it.
    return new Response("invalid signal", { status: 400 });
  }

  const event = (claims.events as Record<string, { reason: string }>)?.[EVENT];
  if (!event) return new Response("unknown event", { status: 400 });

  // Idempotent: a retry carries the SAME jti, so seeing one twice is normal.
  if (await alreadyHandled(claims.jti as string)) {
    return new Response(null, { status: 202 });
  }
```

CONTINUES ON THE NEXT PAGE

CONTINUED

```
  // Acknowledge FIRST, delete after. AFC allows ten seconds, and the deletion
  // is our own work, not something AFC should wait on.
  await queueAfcDisconnection(claims.sub as string, event.reason, claims.jti as string);
  return new Response(null, { status: 202 });
}
```

A tampered payload fails this check, which is the point of checking it. Reject and do nothing.

Answering

Any 2xx means accepted. AFC prefers `202 Accepted` but does not insist; `200` and `204` are read the same way.

You have ten seconds. AFC's request times out after that. Acknowledge receipt and do the deletion work asynchronously. A slow answer is treated as a failed answer.

Retries

AFC'S ANSWER TO	BEHAVIOUR
A connection refused, a DNS or TLS failure, a timeout, HTTP 429, or any 5xx	**Retried.** Roughly 20s, 40s, 80s, 160s then 320s, each plus up to 20 seconds of jitter, to a maximum of 5 retries. The jitter is what stops a hundred signals from one account deletion retrying in lockstep against a server that is already struggling.
Any other 4xx	**Not retried.** A wrong URL, a rejected signature or an unknown route will still be wrong in five minutes.
Any 3xx redirect	**Not retried, and not followed.** AFC calls the address you registered and nowhere else, because nobody can inspect where a URL will redirect on the day. Point the registration at the endpoint that actually receives the POST.
Still failing after the last attempt	AFC gives up and logs it. There is no dead letter queue and no replay endpoint.

Every attempt carries the identical token, signed once. `jti` and `iat` do not change across a redelivery, which is precisely what lets you dedupe on `jti` when a retry follows a response AFC never saw.

So: delivery is **at least once, and not guaranteed**. Build for a duplicate, and never treat the absence of a signal as proof a player is still connected.

It can never cost the player their revoke

By the time AFC tries to reach you, the disconnection has already committed on AFC's side. Your server being down does not undo it, delay it, or turn it into an error for the player. This is by design, and the consequence for you is the one stated in section 11 (page 25): a 401 from userinfo and a 400 from refresh remain your reliable signal that a connection is gone. The webhook makes you fast; it does not make you certain.

What to do when you receive one

1. **Find the player by `sub`.** It is the key you already store. There is nothing else in the payload that identifies them, on purpose.
2. **Delete the tokens you hold for them.** Access, refresh and ID tokens. They are already dead on AFC's side.

3. **Delete or anonymise the AFC-sourced data** you copied: in-game name, avatar, country, email, Free Fire UID, team, history, statistics, ranking, standing.
4. **End any local session that exists only because of the AFC sign-in.** A player who disconnected you does not expect to still be logged in through it.
5. **Decide what your own account becomes.** On `player_revoked` the AFC account still exists and the player may reconnect, so keeping a local account they can sign into another way is reasonable. On `account_deleted` there is nothing left to reconnect to.
6. **Do not call AFC back.** There is nothing to acknowledge beyond your HTTP response, and the player's tokens are already gone, so a revocation call would only 400.

14. Worked end-to-end example (Next.js App Router)

A complete, minimal reference integration. Node runtime, App Router, TypeScript. It generates PKCE, redirects, handles the callback, verifies the ID token against JWKS, and reads userinfo.

Dependency: `jose` for JWT verification.

SHELL
```
pnpm add jose
```

Environment variables (`.env.local`, never committed):

```
AFC_ISSUER=https://api.africanfreefirecommunity.com/sso
AFC_CLIENT_ID=your-client-id
AFC_CLIENT_SECRET=your-client-secret
AFC_REDIRECT_URI=https://partner.example/api/auth/afc/callback
```

Shared helper

TYPESCRIPT
```
// lib/afc.ts
// Discovery lookup and JWKS handling for "Sign in with AFC".
// Endpoints are READ FROM DISCOVERY, never hardcoded, so AFC can move them.
import { createRemoteJWKSet } from "jose";

export const AFC_ISSUER = process.env.AFC_ISSUER!;
export const AFC_CLIENT_ID = process.env.AFC_CLIENT_ID!;
export const AFC_CLIENT_SECRET = process.env.AFC_CLIENT_SECRET!;
export const AFC_REDIRECT_URI = process.env.AFC_REDIRECT_URI!;

// Request only what you need. Every extra scope is one more thing the player can
// refuse, and widening this list later forces a fresh consent screen.
export const AFC_SCOPES = "openid profile afc.freefire";

type Discovery = {
  issuer: string;
  authorization_endpoint: string;
  token_endpoint: string;
  userinfo_endpoint: string;
  jwks_uri: string;
  // Sign the player out of AFC as well as out of your site. See section 12.
  end_session_endpoint: string;
};

let cached: { doc: Discovery; at: number } | null = null;

export async function discovery(): Promise<Discovery> {
  // One hour is plenty. The document is stable; refetching per request is wasteful.
  if (cached && Date.now() - cached.at < 3_600_000) return cached.doc;
  const res = await fetch(`${AFC_ISSUER}/.well-known/openid-configuration`);
  if (!res.ok) throw new Error(`AFC discovery failed: ${res.status}`);
  const doc = (await res.json()) as Discovery;
  cached = { doc, at: Date.now() };
  return doc;
}

// Module-level so the key set is fetched once and cached by jose, not per request.
```

CONTINUES ON THE NEXT PAGE

CONTINUED

```
let jwks: ReturnType<typeof createRemoteJWKSet> | null = null;

export async function afcJwks() {
  if (!jwks) jwks = createRemoteJWKSet(new URL((await discovery()).jwks_uri));
  return jwks;
}
```

Step 1: start the flow

TYPESCRIPT

```
// app/api/auth/afc/login/route.ts
// Sends the player to AFC. Generates the PKCE pair, state and nonce, and stashes all
// three in httpOnly cookies so only this browser can complete the flow.
import crypto from "node:crypto";
import { NextResponse } from "next/server";
import {
  AFC_CLIENT_ID, AFC_REDIRECT_URI, AFC_SCOPES, discovery,
} from "@/lib/afc";

const base64url = (buf: Buffer) =>
  buf.toString("base64").replace(/\+/g, "-").replace(/\//g, "_").replace(/=+$/, "");

export async function GET() {
  // PKCE is REQUIRED by AFC. Omitting these two parameters fails the flow.
  const codeVerifier = base64url(crypto.randomBytes(32));
  const codeChallenge = base64url(
    crypto.createHash("sha256").update(codeVerifier).digest()
  );
  const state = base64url(crypto.randomBytes(16));
  const nonce = base64url(crypto.randomBytes(16));

  const { authorization_endpoint } = await discovery();
  const url = new URL(authorization_endpoint);
  url.searchParams.set("response_type", "code");
  url.searchParams.set("client_id", AFC_CLIENT_ID);
  url.searchParams.set("redirect_uri", AFC_REDIRECT_URI);
  url.searchParams.set("scope", AFC_SCOPES);
  url.searchParams.set("state", state);
  url.searchParams.set("nonce", nonce);
  url.searchParams.set("code_challenge", codeChallenge);
  url.searchParams.set("code_challenge_method", "S256");

  const res = NextResponse.redirect(url.toString());
  const opts = {
    httpOnly: true,
    secure: process.env.NODE_ENV === "production",
    sameSite: "lax" as const,
    path: "/",
    maxAge: 600, // the flow should not take ten minutes
  };
  res.cookies.set("afc_verifier", codeVerifier, opts);
  res.cookies.set("afc_state", state, opts);
  res.cookies.set("afc_nonce", nonce, opts);
  return res;
}
```

Step 2: handle the callback

TYPESCRIPT

```
// app/api/auth/afc/callback/route.ts
// Validates state, exchanges the code (60-second window), verifies the ID token
// against AFC's JWKS, then reads userinfo. `sub` is the only field keyed on.
import { NextRequest, NextResponse } from "next/server";
import { jwtVerify } from "jose";
import {
  AFC_CLIENT_ID, AFC_CLIENT_SECRET, AFC_REDIRECT_URI, afcJwks, discovery,
} from "@/lib/afc";

export async function GET(req: NextRequest) {
  const params = req.nextUrl.searchParams;

  // 1. The player pressed Deny, or an OAuth-level error came back. Handle this FIRST.
  const error = params.get("error");
  if (error) {
    return NextResponse.redirect(new URL(`/login?afc_error=${error}`, req.url));
  }

  // 2. CSRF: the state must match what we issued, in this browser.
  const code = params.get("code");
  const state = params.get("state");
  const expectedState = req.cookies.get("afc_state")?.value;
  const verifier = req.cookies.get("afc_verifier")?.value;
  const nonce = req.cookies.get("afc_nonce")?.value;
  if (!code || !state || !expectedState || state !== expectedState || !verifier) {
    return NextResponse.json({ error: "invalid_state" }, { status: 400 });
  }

  const doc = await discovery();

  // 3. Exchange the code. `redirect_uri` must be byte-identical to step 1, and
  //    `code_verifier` is the RAW verifier, not the challenge.
  const tokenRes = await fetch(doc.token_endpoint, {
    method: "POST",
    headers: { "Content-Type": "application/x-www-form-urlencoded" },
    body: new URLSearchParams({
      grant_type: "authorization_code",
      code,
      redirect_uri: AFC_REDIRECT_URI,
      client_id: AFC_CLIENT_ID,
      client_secret: AFC_CLIENT_SECRET,
      code_verifier: verifier,
    }),
  });
  if (!tokenRes.ok) {
    return NextResponse.json(
      { error: "token_exchange_failed", detail: await tokenRes.text() },
      { status: 400 }
    );
  }
  const tokens = await tokenRes.json() as {
    access_token: string;
    refresh_token: string;
    id_token: string;
    expires_in: number;
    scope: string;
```

CONTINUES ON THE NEXT PAGE

CONTINUED

```
  };

  // 4. Verify the ID token: signature, issuer, audience, expiry, then the nonce.
  const { payload } = await jwtVerify(tokens.id_token, await afcJwks(), {
    issuer: doc.issuer,        // "https://.../sso", no trailing slash
    audience: AFC_CLIENT_ID,
  });
  if (payload.nonce !== nonce) {
    return NextResponse.json({ error: "nonce_mismatch" }, { status: 400 });
  }

  // 5. Read userinfo. Optional here (the ID token carries the same claims), but this
  //    is the call you repeat later to get fresh data.
  const infoRes = await fetch(doc.userinfo_endpoint, {
    headers: { Authorization: `Bearer ${tokens.access_token}` },
  });
  if (!infoRes.ok) {
    return NextResponse.json({ error: "userinfo_failed" }, { status: 502 });
  }
  const info = await infoRes.json();

  // 6. EVERY field except `sub` is optional. A missing claim is normal: AFC's toggle
  //    may be off, the player may not have consented, or an AFC rule may have gated it
  //    (under-18 players release no email, hidden stats release nothing, and so on).
  const player = {
    afcSub: info.sub as string,                       // 64-char hex. The ONLY join key.
    username: (info.preferred_username as string) ?? null,
    email: (info.email as string) ?? null,            // often absent, by design
    country: (info.country as string) ?? null,
    locale: (info.locale as string) ?? null,
    freeFireUid: (info.ff_uid as string) ?? null,
    teamName: (info.afc_team as { name?: string } | undefined)?.name ?? null,
  };

  // await upsertUserByAfcSub(player, tokens);   // key on afcSub, never on email
  // Store tokens.refresh_token; it ROTATES on every refresh, so persist the new one.

  const res = NextResponse.redirect(new URL("/", req.url));
  for (const c of ["afc_verifier", "afc_state", "afc_nonce"]) res.cookies.delete(c);
  return res;
}
```

Later: refresh the access token

TYPESCRIPT

```
// lib/afc-refresh.ts
// AFC rotates refresh tokens with NO grace period: the token you send is dead the
// instant this succeeds. Persist the new one before doing anything else, and hold a
// per-player lock so two concurrent refreshes cannot race and lose the live token.
import { AFC_CLIENT_ID, AFC_CLIENT_SECRET, discovery } from "@/lib/afc";

export async function refreshAfcTokens(storedRefreshToken: string) {
  const doc = await discovery();
  const res = await fetch(doc.token_endpoint, {
    method: "POST",
    headers: { "Content-Type": "application/x-www-form-urlencoded" },
    body: new URLSearchParams({
```

CONTINUES ON THE NEXT PAGE

CONTINUED

```
      grant_type: "refresh_token",
      refresh_token: storedRefreshToken,
      client_id: AFC_CLIENT_ID,
      client_secret: AFC_CLIENT_SECRET,
    }),
  });

  // 400 here usually means the player revoked you from their AFC Connected Apps page,
  // or AFC suspended the application. Mark the connection disconnected; do not retry.
  if (!res.ok) throw new Error(`afc_refresh_failed:${res.status}`);

  return res.json() as Promise<{
    access_token: string;
    refresh_token: string;   // NEW. Save it.
    id_token: string;
    expires_in: number;
    scope: string;
  }>;
}
```

Using an off-the-shelf library instead

The hand-rolled version above is written out in full so nothing is ambiguous, but a compliant OIDC client library pointed at the discovery URL is less code and less to get wrong. Whatever you use, confirm it sends PKCE (`code_challenge` / `code_verifier`) and verifies the ID token signature against `jwks_uri`. Configure it with:

- Discovery URL: `<AFC base>/sso/.well-known/openid-configuration`
- Client authentication: `client_secret_post` or `client_secret_basic`
- Response type: `code`
- PKCE method: `S256`
- Scopes: `openid` plus whatever AFC approved for you

15. Common mistakes

1. **Forgetting PKCE.** AFC requires it for confidential clients too. If your library treats PKCE as a public-client-only feature, turn it on explicitly. This is the number one cause of a first integration failing.
2. **Keying on email or username.** Email changes, and is absent entirely for under-18 players and for any player whose date of birth AFC does not hold. Username is an in-game name that players change. **Key on `sub`.**
3. **Assuming a requested scope always returns a value.** All four gates must agree. Treat every claim except `sub` as optional, on every call, including calls that succeeded yesterday.
4. **Assuming a missing claim is `null`.** AFC omits the key entirely. `userinfo.afc_team.name` throws; `userinfo.afc_team?.name` does not.
5. **Hardcoding endpoints instead of reading discovery.** Also: hardcoding `/sso/revoke/` when the real path is `/sso/revoke_token/`, and dropping the trailing slashes, which are part of every AFC path.
6. **Mismatching the redirect URI.** It must match a registered URI exactly at step 1, and be byte-identical again in the token exchange at step 3. A trailing slash, an added query parameter or a different scheme all fail.
7. **Relying on `claims_supported` to enumerate what you will receive.** It lists only `sub`, a library limitation rather than a description of AFC. The real catalogue is **section 6 (page 17)**. (`subject_types_supported` IS accurate: pairwise.)
8. **Losing the rotated refresh token.** Every refresh invalidates the token you sent, with no grace period. Persist the new one atomically, and do not refresh concurrently for the same player.
9. **Assuming every claim is present.** Any field can be absent for a legitimate reason (see section 7). Code defensively rather than indexing straight into the payload.
10. **Expecting a refusal to come back to your callback.** AFC's four refusals render a 400 page at AFC and never redirect. Your callback simply never fires, so time out pending flows.
11. **Sitting on a 60-second authorization code.** Exchange it in the callback handler, not on a queue.
12. **Throwing away the `id_token` after verifying it.** You need it again as the `id_token_hint` when you sign the player out (**section 12 (page 27)**). Store it with the access and refresh tokens.
13. **Sending a player to the end session endpoint with an unregistered `post_logout_redirect_uri`.** AFC refuses with a 400 page rather than redirecting, so the player is stranded at AFC. Register the URL first.
14. **Expecting the end session endpoint to sign the player out of every partner, or out of every device.** It disconnects only the partner that asked, and it ends the AFC session only on the device the player pressed the button on. Both are deliberate.
15. **Parsing the disconnection webhook body as JSON.** The body is a signed JWT and the `Content-Type` is `application/secevent+jwt`. Read the raw body as text, then verify it (**section 13 (page 30)**).
16. **Looking for an HMAC signature header on the webhook.** There is none. It is an RS256 JWT verified against the same JWKS as your ID tokens.

17. **Treating the webhook as exactly-once, or as guaranteed.** AFC retries, so the same `jti` can arrive twice; AFC also eventually gives up, so it can never arrive at all. Dedupe on `jti`, and keep treating a 401 from userinfo as the reliable signal.

16. Security expectations of the partner

AFC is handing you data about real players, some of them minors. These are not optional.

1. **Keep `client_secret` server side.** Never in client-side JavaScript, a mobile app binary, a public repository, or an environment variable exposed to the browser (in Next.js, never prefix it with `NEXT_PUBLIC_`). It is shown once and AFC cannot re-display it. Rotate it through AFC if it is ever exposed.
2. **Run the token exchange server side.** The authorization code and `client_secret` must never touch the browser.
3. **Validate `state` on every callback.** Generate it randomly, store it bound to the browser session, compare it, and reject on mismatch. This is your CSRF defence.
4. **Use `nonce` and check it.** Generate it randomly, send it on the authorization request, and compare it against the `nonce` claim in the verified ID token. This is your replay defence.
5. **Verify the ID token signature against JWKS.** Fetch keys from the `jwks_uri` in discovery, match on the `kid` in the JWT header, and verify `iss`, `aud` and `exp` as well as the signature. Never trust a decoded-but-unverified JWT. Cache the key set, and let your library re-fetch on an unknown `kid` so AFC can rotate signing keys.
6. **HTTPS everywhere.** Redirect URIs, your callback, and every AFC call. No exceptions in production.
7. **Store PKCE verifier, state and nonce in httpOnly, `Secure`, `SameSite=Lax` cookies or a server-side session**, with a short lifetime, and delete them once the flow completes.
8. **Store tokens encrypted at rest and never log them.** Access tokens, refresh tokens and ID tokens are credentials. Keep them out of application logs, error trackers and analytics.
9. **Honour revocation promptly.** A 401 from userinfo or a 400 from refresh means the player or AFC cut you off. Stop using the tokens and delete them.
10. **Register a disconnection webhook, verify every signal, and act on it.** AFC will tell you when a player disconnects you or deletes their AFC account, but only you can delete the copy of their data sitting in your own database. Verify the signature against AFC's JWKS before acting, reject anything that does not verify, and delete what the player asked you to delete. See section 13 (page 30).
11. **Do not re-share AFC data.** What AFC releases to you is scoped to your organisation and consented to by the player for your organisation. Passing it to a third party is outside what the player agreed to.
12. **Request the minimum scopes you need.** Fewer scopes means a shorter consent list, a higher approval rate and less to protect.
13. **Handle minors' data carefully.** AFC deliberately withholds contact data for under-18 players. Design your product so that an account with no email address is fully functional rather than blocked.

17. Quick reference card

```
Apply             https://africanfreefirecommunity.com/partners/apply
                  No AFC account needed. Redirect URIs are policy-checked on submit.
                  Credentials arrive as a ONE-TIME link, never in an email body.

Base URL          https://api.africanfreefirecommunity.com
Issuer            https://api.africanfreefirecommunity.com/sso   (no trailing slash)

Discovery         GET  /sso/.well-known/openid-configuration
JWKS              GET  /sso/.well-known/jwks.json
Authorize         GET  /sso/authorize/
Token             POST /sso/token/
UserInfo          GET  /sso/userinfo/          Authorization: Bearer <access_token>
Revoke            POST /sso/revoke_token/
End session       GET  /sso/logout/           id_token_hint says who is asking

Flow              authorization code + PKCE (S256).  PKCE IS MANDATORY.
Response type     code
Client auth       client_secret_post | client_secret_basic
ID token alg      RS256

Scopes            openid profile email
                  afc.freefire afc.team afc.history
                  afc.stats afc.ranking afc.standing

Claims            sub (64-hex, PAIRWISE, the only safe key)
                  preferred_username, country, locale        <- profile
                  email, email_verified                       <- email
                  ff_uid                                      <- afc.freefire
                  afc_team    {name, role}                    <- afc.team
                  afc_history [{event, slug}] max 50          <- afc.history
                  afc_stats   {matches, kills}                <- afc.stats
                  afc_ranking {points, rank, month}           <- afc.ranking
                  afc_standing {in_good_standing}             <- afc.standing

                  Missing claims are ABSENT, not null. All four gates must agree.

Lifetimes         auth code       60 s, single use
                  access token    36000 s (10 h)
                  id token        36000 s (10 h)
                  refresh token   no expiry, ROTATES on every use, no grace period

Sign out          send the browser to end_session_endpoint with id_token_hint,
                  a REGISTERED post_logout_redirect_uri and state.
                  Disconnects ONLY you. Ends the AFC session on THAT DEVICE only,
                  and only when the player confirms on AFC's page.

Disconnect hook   optional, register an https URL with AFC.
                  POST, Content-Type: application/secevent+jwt, body IS the JWT.
                  RS256, verify against the SAME jwks_uri as ID tokens.
                  Check iss and aud. Dedupe on jti. Answer 2xx within 10 s.
                  reason = player_revoked | account_deleted.
                  Retries 5x with backoff on 5xx/429/timeout. At least once,
                  never guaranteed: a dead token is still your reliable signal.

Not available     dynamic client registration, self-serve app registration,
                  back-channel and front-channel logout, introspection
```

Questions, credential issues, scope changes and application suspensions:

info@africanfreefirecommunity.com