iBunnyApp Integration Guide

A complete, machine-readable specification for adding Continue with iBunnyApp single sign-on, a shared friends graph, cross-game messaging, wagers and Bunnycoin balances to any Capygram blockchain game.

Audience: AI coding assistants (Lovable, ChatGPT, Claude, Cursor, Copilot) and the developers directing them.
How to use this document: upload it to your AI assistant and say: "Integrate my game with iBunnyApp exactly as specified in this PDF." Every endpoint, header, payload, error code and security rule the assistant needs is here.

Base URL: https://www.ibunnyapp.com | API version: v1 | All responses are JSON. All endpoints support CORS preflight (OPTIONS).

Instructions for the AI assistant

- Treat this document as the authoritative contract. Do not invent endpoints, headers, scopes or field names that are not listed here.
- Never place the studio API key (`bnk_...`) in client-side code, a browser bundle, a mobile app, or a public repository. It belongs only in server-side environment variables.
- All calls to `/api/public/v1/*` must be made from the game's backend. The browser never talks to iBunnyApp directly.
- Use the OAuth 2.0 Authorization Code flow with PKCE. Implicit flow and password grant are not supported.
- Store the player's iBunnyApp user id (a UUID) on the game's own user record. That id is the permanent join key between the game account and the iBunnyApp account.
- Send a unique `external_ref` on every Bunnycoin balance change so a retry is never applied twice.

1. What the integration gives your game

Capability	What the player gets	Endpoint
Single sign-on	One iBunnyApp account signs them into every Capygram game. No new password.	OAuth 2.0 + /v1/me
Account linking	An existing game account can be linked to their iBunnyApp profile and keep its progress.	OAuth 2.0 + /v1/me
Shared friends	Their iBunnyApp friends appear inside your lobby on the very first session.	/v1/friends
Cross-game messages	They read and send iBunnyApp messages without leaving your game.	/v1/messages
Bunnycoin	One balance that travels across every game on the network.	/v1/bunnycoin
Escrowed wagers	Friend-vs-friend wagers held and paid out by iBunnyApp, never by your game.	/v1/wagers

2. One-time studio setup

- Create a studio account at https://www.ibunnyapp.com/auth.
- Open /studio and register your game: name, slug, URL, tagline, emoji.
- Create an API key for that game. The key is shown once - copy it immediately and store it as a server-side secret, for example `IBUNNY_API_KEY`. Only a SHA-256 hash is kept by iBunnyApp, so a lost key must be revoked and replaced.
- Register your OAuth redirect URI (see section 3). It must be an exact, absolute HTTPS URL, for example `https://yourgame.com/auth/ibunnyapp/callback`.
- Keys can be revoked at any time from /studio. A revoked key immediately returns `invalid_api_key`.

3. Authentication model

Every player-scoped request carries two credentials, and both are required:

Header	Meaning	Where it comes from
`x-bunnycoin-key: bnk_...`	Which game is asking.	Your studio API key from /studio (server-side secret).
`Authorization: Bearer <token>`	Which player they are asking about.	The player's iBunnyApp OAuth access token.

Reads execute as the player under row-level security, so a token can only ever reach that player's own data. There is no endpoint that returns another player's private data.

3.1 Discovering the authorization server

Do not hardcode the authorization server. Fetch the protected-resource metadata document, then the authorization server's OpenID configuration:

```
GET https://www.ibunnyapp.com/.well-known/oauth-protected-resource

{
  "resource": "https://www.ibunnyapp.com",
  "authorization_servers": ["https://<auth-host>/auth/v1"],
  "scopes_supported": ["openid", "email", "profile"],
  "bearer_methods_supported": ["header"],
  "resource_documentation": "https://www.ibunnyapp.com/developers"
}

# Then read the authorization server's endpoints:
GET <authorization_servers[0]>/.well-known/openid-configuration
# -> authorization_endpoint, token_endpoint, registration_endpoint, issuer
```

3.2 Authorization Code flow with PKCE

- Generate a `code_verifier` (43-128 random URL-safe chars) and a `code_challenge` = base64url(SHA-256(verifier)). Store the verifier and a random `state` in the player's server-side session.
- Redirect the player's browser to the `authorization_endpoint` with: `response_type=code`, `client_id`, `redirect_uri`, `scope=openid email profile`, `state`, `code_challenge`, `code_challenge_method=S256`.
- The player signs in and approves your game on the iBunnyApp consent screen. They can revoke that approval from their dashboard at any time.
- iBunnyApp redirects back to your `redirect_uri` with `code` and `state`. Verify the state matches, then exchange the code at the `token_endpoint` from your backend.
- Store `access_token`, `refresh_token` and `expires_in` server-side, keyed to your own user row. Refresh with `grant_type=refresh_token` when the access token expires.

```
// STEP 1 - build the authorize URL (Node / TypeScript, server-side)
import crypto from "node:crypto";

const b64url = (b: Buffer) => b.toString("base64url");
const verifier = b64url(crypto.randomBytes(32));
const challenge = b64url(crypto.createHash("sha256").update(verifier).digest());
const state = b64url(crypto.randomBytes(16));
// persist { verifier, state } in the player's session (NOT in a cookie readable by JS)

const j = (r: Response) => r.json();
const BASE = "https://www.ibunnyapp.com";
const meta = await fetch(`${BASE}/.well-known/oauth-protected-resource`).then(j);
const as = meta.authorization_servers[0];
const oidc = await fetch(`${as}/.well-known/openid-configuration`).then(j);

const url = new URL(oidc.authorization_endpoint);
url.searchParams.set("response_type", "code");
url.searchParams.set("client_id", process.env.IBUNNY_CLIENT_ID!);
url.searchParams.set("redirect_uri", "https://yourgame.com/auth/ibunnyapp/callback");
url.searchParams.set("scope", "openid email profile");
url.searchParams.set("state", state);
url.searchParams.set("code_challenge", challenge);
url.searchParams.set("code_challenge_method", "S256");
redirect(url.toString());

// STEP 2 - callback: verify state, exchange code, fetch the player
```

```
const { code, state } = query;
if (state !== session.state) throw new Error("state_mismatch");

const tokens = await fetch(oidc.token_endpoint, {
  method: "POST",
  headers: { "content-type": "application/x-www-form-urlencoded" },
  body: new URLSearchParams({
    grant_type: "authorization_code",
    code,
    redirect_uri: "https://yourgame.com/auth/ibunnyapp/callback",
    client_id: process.env.IBUNNY_CLIENT_ID!,
    code_verifier: session.verifier,
  }),
}).then(r => r.json());   // { access_token, refresh_token, expires_in }

const me = await fetch("https://www.ibunnyapp.com/api/public/v1/me", {
  headers: {
    "x-bunnycoin-key": process.env.IBUNNY_API_KEY!,
    Authorization: `Bearer ${tokens.access_token}`,
  },
}).then(r => r.json());

// me.user.id is the permanent iBunnyApp user id -> store it on your user row.
await db.users.upsert({ ibunny_user_id: me.user.id, username: me.user.username });
```

Calling /v1/me after the OAuth round-trip is required: it records the player's connection to your game. Until that happens, server-to-server calls for that player return `player_not_connected`.

3.3 Linking an existing game account

Linking is the same flow, started while the player is already signed in to your game. On callback, attach `me.user.id` to the existing account row instead of creating a new one. Enforce a unique constraint on that column so one iBunnyApp account maps to at most one account in your game, and offer an unlink action that clears the column and deletes the stored tokens.

4. API reference

Method & path	Purpose
`GET /api/public/v1/me`	Player profile, username, avatar, bio and Bunnycoin balance. Records the game connection.
`GET /api/public/v1/friends`	Accepted friends plus pending requests, with direction.
`GET /api/public/v1/messages`	Inbox (100 most recent). Add ?with=<uuid> for one conversation.
`POST /api/public/v1/messages`	Send a message: { recipient_id, body }.
`GET /api/public/v1/wagers`	The player's wagers and their status.
`POST /api/public/v1/wagers`	Create, respond, cancel or report a wager (see 4.5).
`GET /api/public/v1/bunnycoin`	Bunnycoin balance.
`POST /api/public/v1/bunnycoin`	Credit or debit Bunnycoin: { delta, memo?, external_ref? }.

4.1 GET /v1/me

```
curl https://www.ibunnyapp.com/api/public/v1/me \
  -H "x-bunnycoin-key: $IBUNNY_API_KEY" \
  -H "Authorization: Bearer $PLAYER_ACCESS_TOKEN"

200 {
  "user": { "id": "uuid", "username": "capyking", "display_name": "Capy King",
            "avatar_url": "https://...", "bio": "...", "created_at": "2026-01-01T00:00:00Z" },
  "bunnycoin_balance": 1250,
  "currency": "BUNNY"
}
```

4.2 GET /v1/friends

```
200 {
  "friends": [
    { "friendship_id": "uuid", "status": "accepted", "direction": "outgoing",
      "player": { "id": "uuid", "username": "hoppy", "display_name": "Hoppy",
                  "avatar_url": "https://..." } }
  ],
  "pending": [
    { "friendship_id": "uuid", "status": "pending", "direction": "incoming",
      "player": { "id": "uuid", "username": "nibbles", "...": "..." } }
  ]
}
// direction "outgoing" = this player sent it; "incoming" = they received it.
// Only the recipient of a pending request can accept it (enforced server-side).
```

4.3 GET /v1/messages

```
GET /api/public/v1/messages                  -> 100 most recent, newest first
GET /api/public/v1/messages?with=<user_uuid> -> just that conversation

200 { "messages": [ { "id": "uuid", "body": "gg", "sender_id": "uuid",
                      "recipient_id": "uuid", "read_at": null,
                      "created_at": "2026-08-05T12:00:00Z",
                      "origin_game_id": "uuid",
                      "sender": { "id": "uuid", "username": "hoppy", "...": "..." } } ] }
```

4.4 POST /v1/messages

```
curl -X POST https://www.ibunnyapp.com/api/public/v1/messages \
  -H "x-bunnycoin-key: $IBUNNY_API_KEY" \
  -H "Authorization: Bearer $PLAYER_ACCESS_TOKEN" \
  -H "content-type: application/json" \
  -d '{ "recipient_id": "uuid", "body": "rematch?" }'

201 { "message": { "id": "uuid", "created_at": "..." } }
// body: 1-2000 characters. origin_game_id is stamped with your game automatically.
// 400 invalid_body (validation) | 400 send_failed (recipient not reachable)
```

4.5 Wagers

Wagers are escrowed by iBunnyApp. Your game never holds funds: it only creates, responds to and reports wagers. Payout happens when both participants report the same winner.

```
GET  /api/public/v1/wagers
200 { "wagers": [ { "id": "uuid", "amount": 100, "status": "active", "note": "bo3",
                    "game_id": "uuid", "challenger_id": "uuid", "opponent_id": "uuid",
                    "challenger_result": "uuid|null", "opponent_result": "uuid|null",
                    "winner_id": null, "created_at": "..." } ] }
```

```
POST /api/public/v1/wagers    (one of four actions)
{ "action": "create",  "opponent_id": "uuid", "amount": 100, "note": "bo3" }
{ "action": "respond", "wager_id": "uuid", "accept": true }
{ "action": "cancel",  "wager_id": "uuid" }
{ "action": "report",  "wager_id": "uuid", "winner_id": "uuid" }

200 { "result": ... }
```

- create: the opponent must already be an accepted friend, amount 1-1,000,000, and the challenger needs the balance. Stake is escrowed on acceptance.
- respond: only the opponent may accept or decline a pending wager.
- cancel: only the challenger, and only while the wager is still pending.
- report: either participant reports the winner. Matching reports pay out; conflicting reports move the wager to a disputed state, and a participant may re-report to resolve it, so stakes are never permanently locked.

4.6 Bunnycoin

```
GET  /api/public/v1/bunnycoin
200 { "user_id": "uuid", "balance": 1250, "currency": "BUNNY" }

POST /api/public/v1/bunnycoin
{ "delta": 50, "memo": "level 3 reward", "external_ref": "yourgame:reward:98f1c2" }
200 { "user_id": "uuid", "balance": 1300, "currency": "BUNNY" }

// delta: non-zero integer, |delta| <= 1,000,000. Negative debits, positive credits.
// external_ref: your own unique id for this move. Replaying the same ref for the
// same player is rejected with duplicate_external_ref instead of double-applying.
```

4.7 Server-to-server mode (no player token)

Background jobs such as leaderboard payouts or refunds can omit the player's bearer token and pass `user_id` instead. This works only on the Bunnycoin endpoints, and only for players who have connected your game.

```
# header on both calls:  x-bunnycoin-key: $IBUNNY_API_KEY   (no Authorization header)

GET  /api/public/v1/bunnycoin?user_id=<uuid>
POST /api/public/v1/bunnycoin
     { "user_id": "<uuid>", "delta": -25,
       "external_ref": "yourgame:entryfee:4412" }

// Bearer token sent AND user_id disagrees with it -> 403 user_id_mismatch.
// Player never connected your game               -> 403 player_not_connected.
```

5. Errors

Errors are always JSON: `{ "error": "code" }`. Match on the code, never on prose.

Code	Status	Meaning and fix
invalid_api_key	401	Missing, wrong or revoked x-bunnycoin-key. Check the server secret.

Code	Status	Meaning and fix
`invalid_access_token`	401	Player token missing, expired or malformed. Refresh it, then retry.
`player_not_connected`	403	Player has not connected your game. Run the OAuth flow and call /v1/me.
`user_id_mismatch`	403	The user_id in the request is not the owner of the bearer token.
`invalid_user_id`	400	user_id is not a UUID.
`invalid_body`	400	Payload failed validation. Compare with the schemas in section 4.
`invalid_with`	400	?with= is not a UUID.
`insufficient_balance`	409	Not enough Bunnycoin for the debit or wager stake.
`duplicate_external_ref`	409	This external_ref was already applied. Treat as success; do not retry.
`wallet_not_found`	409	The player has no wallet yet. Call /v1/me first.
`not_friends`	409	Wager opponent is not an accepted friend.
`cannot_wager_yourself`	409	Challenger and opponent are the same player.
`invalid_amount`	409	Wager amount outside 1-1,000,000.
`wager_not_found`	409	Unknown wager id.
`wager_not_pending`	409	Action requires a pending wager (respond, cancel).
`wager_not_active`	409	Action requires an active wager (report).
`not_a_participant`	409	This player is not in that wager.
`not_your_wager`	409	Only the challenger may cancel.
`invalid_winner`	409	winner_id is not one of the two participants.
`lookup_failed`	500	Transient read failure. Retry with backoff.
`move_failed` / `wager_failed` / `send_failed`	400/500	Unclassified failure. Log the request and retry once.

Retry policy

- Retry 500 and network errors with exponential backoff (for example 1s, 2s, 4s; three attempts).
- Never blindly retry a 4xx - fix the request instead.
- Always attach a stable `external_ref` to balance changes so a retry after a timeout cannot double-spend.

6. Security requirements

- Keep the API key server-side. Never ship `bnk_...` to a browser, mobile binary, game client, config file in a public repo, or log output.
- Proxy everything. Your game client calls your backend; your backend calls iBunnyApp.
- Store tokens encrypted at rest and never expose access or refresh tokens to the client.
- Validate state on every OAuth callback and use a single-use PKCE verifier bound to that session.
- Exact redirect URIs. Register absolute HTTPS URLs; never accept an open redirect or a player-supplied `redirect_uri`.
- Trust the token, not the input. Derive the player's identity from their access token; never let a client tell your backend which iBunnyApp user it is acting for.
- Rate-limit your own endpoints that fan out to iBunnyApp, especially message sending and balance changes.
- Rotate on suspicion. Revoke and re-create the key from /studio if it may have leaked; revocation takes effect immediately.
- Respect disconnects. If a player disconnects your game from their iBunnyApp dashboard, their calls start failing - handle it as a normal signed-out state, not a crash.

7. Integration checklist

1. Game registered in /studio; API key stored as a server-side secret.
2. Redirect URI registered and matches the callback route exactly.
3. Discovery documents fetched at runtime instead of hardcoded endpoints.
4. Authorization Code + PKCE flow implemented, with state verification.
5. Tokens stored server-side, refreshed before expiry.
6. GET /v1/me called on callback; iBunnyApp user id persisted (unique column).
7. Existing-account linking and unlinking both work.
8. Friends rendered in the lobby from /v1/friends, including pending requests.
9. Inbox reads and sends wired to /v1/messages.
10. Bunnycoin credits and debits always send a unique external_ref.
11. Wager create / respond / cancel / report handled, including the disputed path.
12. Every documented error code handled with a clear player-facing message.
13. No iBunnyApp credential reachable from the client bundle.

Reference and support: https://www.ibunnyapp.com/developers (live API reference), /studio (games and keys), /bunnycoin (how Bunnycoin works).