JANZEER BLOCKCHAIN

# Exchange & Custodian Integration Guide

Node · deposits · withdrawals · supply

Document v1.0 · Janzeer Network · 2026

# Janzeer — Exchange and Custodian Integration Guide

This guide is for a service that holds JNZ for its users: an exchange, a custodian, a payment processor, a swap service. It lists what such a service needs and nothing else. The full API is in the Developer Guide; the byte-level signing rules are in the Yellow Paper.

Janzeer is its own Layer-1 network. It is **not** EVM-compatible: an Ethereum library, an Ethereum wallet or an Ethereum address derivation will not work, even though addresses look alike.

## 1. Facts at a glance

| Item | Value |
|---|---|
| Network id | `janzeer` (mainnet). It is part of every signed transaction, so a transaction cannot be replayed on another network. |
| Coin | JNZ, 8 decimal places. Amounts are decimal strings in the API (`"2.5"`). |
| Supply | 100,000,000 JNZ, fixed at genesis. Nothing is minted or burned; block rewards are paid out of a treasury account. |
| Account model | Accounts with balances and a per-sender nonce (no UTXOs). |
| Block time | 15 seconds. |
| Finality | **One block.** A transaction in a committed block is final; there are no reorganisations and no confirmation count. |
| Address | `0x` + 40 hex characters. |
| Signature | secp256k1, deterministic (RFC 6979), low-S, DER encoded as Base64, over a double-SHA-256 digest. |
| Smart contracts | None. One native token standard (JZT-1) with create, mint, burn, cap and transfer. |
| Fee | Paid in JNZ. Transfer: minimum `0.01`. There is no fee market; the minimum is always enough. |
| Memo | Optional `data` field on a transfer, up to 256 bytes. |
| Minimum transfer | `0.1` JNZ (any amount is accepted when a memo is attached). |
| API | REST under `/api/v1`, JSON-RPC 2.0 at `POST /rpc`, WebSocket at `/rpc/ws`. |
| SDKs | TypeScript, Dart, Kotlin/JVM, Python, Go; a command-line client `jz`. All open source. |
| Explorer | https://explorer.janzeer.org |

## 2. Run your own node

A service should read the chain from a node it runs. The node is a single native binary; no JVM or database server is needed.

1. Download the bundle for your platform (Linux x64, Linux arm64, Windows x64, an Arch Linux package or the Docker image `ghcr.io/janzeerorg/janzeer-node`) from https://janzeer.org/download or the GitHub releases.
2. Verify it. Every release carries a `SHA256SUMS` file signed with the project's minisign key:

```
minisign -Vm SHA256SUMS -P RWQBrkWzg4V5n5wUBa9GNPpa95Z7WFT2EY5QrabMv/DP+KG+gJsPIgdr
sha256sum -c SHA256SUMS --ignore-missing
```

3. Unpack and start it with the launcher in the bundle. The node creates its own identity, connects to the network and synchronises. It is a **watch-only** node: it follows and verifies the chain, it does not vote, and it needs no inbound port.
4. Wait until `GET /api/v1/info/sync` reports `"syncStatus": "SYNCHRONIZED"`.

Operational notes:

- The API listens on port `7019`. Keep it on a private interface; the node holds no keys, but the API is yours.
- A committed block is re-executed and verified by your node before it is stored, so your node does not trust the peers it syncs from.
- `GET /actuator/health` is the liveness check. `GET /api/v1/info/update` tells you when a newer release exists; the bundle's `update.sh` installs it after verifying the signature. Releases and any protocol change are announced on the project channel (https://t.me/janzeerorg).
- Read calls are not rate-limited. `POST` submits and `/rpc` allow 20 requests per second per client IP with a burst of 40 (`429` with `Retry-After` beyond that).

You may use the public node at `https://explorer.janzeer.org` for testing. Do not build a production service on it.

## 3. Addresses

- An address is the first 20 bytes of `keccak256(compressed public key)`, written as `0x` + 40 hex characters. This is **not** the Ethereum rule (Ethereum hashes the uncompressed key), so the same private key gives a different address than on Ethereum.
- The canonical form is lower-case. A mixed-case checksum form exists and is accepted; compare addresses case-insensitively and store them lower-case.
- Validate before you accept an address from a user. Offline: `isValidAddress` in the TypeScript SDK and its twins in the other SDKs. Online: `POST /api/v1/wallets/validateAddress` returns `400` for a malformed address.
- Any valid secp256k1 private key is an account. An account does not need to be "activated": it exists as soon as it receives coins.

### Deposit addresses

Both common models work:

- **One address per user.** Generate a key per user, or derive them from one seed with the SDK's key derivation (`derivePath`). Watch all of them (section 4).
- **One shared address plus a memo.** Give each user a memo value and ask them to put it in the transfer's memo field (`data`, up to 256 bytes). The official wallets have a memo field on the send screen.

## 4. Detecting deposits

Because finality is one block, a deposit can be credited as soon as its transaction is reported `FINAL`.

**Push (recommended).** Open a WebSocket to `/rpc/ws` and subscribe to your addresses:

```
{"jsonrpc":"2.0","id":1,"method":"janzeer_subscribe",
 "params":{"kind":"addressActivity","params":{"addresses":["0x…","0x…"]}}}
```

Every final transaction that touches a watched address arrives as a notification with its block height, block hash and the transaction. One subscription takes up to 1,000 addresses and one connection up to 16 subscriptions.

**Block scan (the safety net).** Subscribe to `newBlocks` with `fromHeight` set to the last height you processed, or poll `janzeer_getBlocks` (`fromHeight`, `toHeight`, `includeTransactions: true`, up to 100 blocks per call). Blocks arrive in order, so after a restart you continue from your stored height and miss nothing. Run this even when you use the push feed.

**Per address.** `janzeer_getTransactionsByAddress` (`address`, `page`, `size`) lists final transfers sent or received, plus the address's pending entries.

Rules for crediting:

1. Credit only a transaction whose status is `FINAL` and whose receipt says `successful: true`.
2. Use the transaction hash as the idempotency key. A hash never appears in two blocks.
3. Credit the `amount` field. The fee is paid by the sender on top of the amount; the recipient receives the amount in full.
4. Block rewards also arrive at validator wallets as transactions of type `reward`. Ignore them unless the address is a validator's.
5. Never credit a `PENDING` transaction. A pending transaction can be replaced by its sender or expire.

## 5. Sending withdrawals

The node never holds keys. You build and sign the transaction in your own system and submit the signed result.

1. Read the next nonce: `janzeer_getNonce` (`address`). It already counts your own pending transactions.
2. Build and sign a transfer with an SDK (`TxBuilder` in each SDK), with fee `"0.01"` and the network id `janzeer`. Signing needs no network access, so it can run on an offline or isolated machine.
3. Submit: `janzeer_sendTransfer` with the signed object, or `POST /api/v1/transactions/transfers`. The answer is `{hash, status: "PENDING"}`. Submitting

the same signed transaction again is harmless.

4. Wait: `janzeer_waitForFinality` (`hash`, `timeoutMs` up to 60,000) or poll `janzeer_getTransactionByHash`. Mark the withdrawal done at `FINAL`.

Nonce and mempool rules that matter for a hot wallet:

- Nonces are sequential per sender and start at 0. A transaction with a gap is refused.
- At most 64 transactions from one sender may be pending at a time. For higher throughput, use several hot wallets.
- A pending transaction can be replaced only by one with the **same nonce and a strictly higher fee**.
- A pending transaction that is not included within 6 hours is dropped. If that happens, sign it again with the nonce the node reports.
- Errors name their cause: `INVALID_NONCE`, `INSUFFICIENT_BALANCE`, `INCORRECT_SIGNATURE`, `INCORRECT_ADDRESS` (REST: in the error payload's `type`; JSON-RPC: code `-32001` with `data.type`).

A complete send, in TypeScript (`npm install janzeer-sdk`):

```
import { Account, JanzeerClient, JanzeerRpc, TxBuilder, waitForFinality } from
'janzeer-sdk';

const client = new JanzeerClient('http://127.0.0.1:7019');
const rpc = new JanzeerRpc('http://127.0.0.1:7019');
const hot = Account.fromMnemonic(process.env.HOT_WALLET_MNEMONIC!);

const tx = hot.signTx(TxBuilder.transfer({
  from: hot.address, to: '0x598b1301acef3baba6ce25e38dd17b723f7b98b1',
  amount: '1.25', data: 'withdrawal 8841', nonce: await client.nonce(hot.address),
}));
await client.submit(tx);                        // {hash, status: "PENDING"}
const done = await waitForFinality(rpc, tx.hash);
console.log(done.blockHeight, done.receipt?.successful);
```

The same flow exists in every SDK; each ships it as a runnable file in its `examples/` folder.

## 6. Balances and reconciliation

- `janzeer_getAccount` (`address`) returns `committedBalance` (final), `balance` (spendable: committed minus your pending debits), `nextNonce` and `pendingCount`.
- A balance changes only through transactions in blocks, so the sum of credited deposits minus withdrawals and fees must equal `committedBalance` at any block height.
- `janzeer_getTip` gives the current height and hash. Store the height you reconciled at.

## 7. Supply figures

Available since node 0.1.14:

| Endpoint | Returns |
|---|---|
| `GET /api/v1/supply/total` | Total supply as one plain number (`100000000`). |
| `GET /api/v1/supply/circulating` | Circulating supply as one plain number. |
| `GET /api/v1/supply` | Both figures plus the non-circulating accounts and their current balances. |

**Total supply** is fixed. **Circulating supply** is counted the way market-data sites count it: the total minus every coin held by the protocol or by the project, at current balances. Those accounts are the key-less treasury (`0x0000000000000000000000000000000000000000`, which pays block rewards and receives 20% of every fee), the founder and ecosystem genesis allocations, the founding validators' wallets, and any other wallet the project holds. `GET /api/v1/supply` lists each of them with its address and balance, so the figure can be recomputed by anyone. `GET /api/v1/explorer/info` reports the same circulating figure (before 0.1.14 it subtracted the treasury only).

## 8. JZT-1 tokens

Tokens issued on the native token standard live in the same accounts. Token deposits arrive through the same `addressActivity` feed as transactions of type `token`; balances are read with `janzeer_getTokenBalances`. Token amounts are integers in the token's base units, sent as strings; each token defines its own number of decimals. The fee of a token transfer is paid in JNZ.

## 9. Checklist before going live

- ☐ Your own node is synchronised and monitored (`/actuator/health`, `/api/v1/info/sync`).
- ☐ Addresses are validated on input and compared case-insensitively.
- ☐ Deposits are credited only at `FINAL` with a successful receipt, once per transaction hash.
- ☐ The block scan resumes from a stored height after a restart.
- ☐ Withdrawals handle `INVALID_NONCE` by re-reading the nonce, and stay below 64 pending per hot wallet.
- ☐ Signing keys never touch the node; the node's API is not exposed to the internet.
- ☐ A test deposit and a test withdrawal have been run end to end on mainnet with a small amount.

## 10. Contact

Technical questions: open an issue at https://github.com/janzeerOrg/janzeer-node/issues or write to info@janzeer.org. The project answers integration questions directly and can

provide a test amount of JNZ for an integration run.

---

*Document status: version 1.1 (2026-11-13). Reflects node 0.1.14.*