JANZEER BLOCKCHAIN

Developer Guide

Run a node · REST API · sign transactions

Document v1.0 · Janzeer Network · 2026

Janzeer Developer Guide

This guide is for app developers and integrators building on **Janzeer**, a fixed-supply, permissionless **stake-free** BFT Layer-1 with a custom, non-custodial HD-wallet crypto scheme. It shows you how to run a node, consume the REST API, and — most importantly — how to build, sign, and submit transactions entirely client-side, because the node never holds user keys. It is practical and example-driven; for exact byte layouts and constants it points you to the Yellow Paper rather than duplicating them.

---

1. Quick start

1.1 Run a node

The node is a **public native download** (closed source — there is no JAR and nothing to build). Get the bundle for your platform from the **Validators** page on janzeer.org, verify its SHA-256, unpack and start it (`./start.sh`, or `start.cmd` on Windows) — the Operator Guide §2 has the details. A freshly started node follows the chain in **watch-only** mode, which is all an application needs:

Service	Default	Notes
REST API	`http://localhost:7019/api/v1/`	response envelope `{timestamp, version, payload}`
JSON-RPC 2.0	`POST http://localhost:7019/rpc` · WebSocket `ws://localhost:7019/rpc/ws`	`janzeer_*` methods, live block/transaction feeds (§2.14)
Explorer + web wallet	`http://localhost:7019/`	served by the node itself
P2P	`9199` (TCP)	outbound only for a watch-only node

The launcher reads `JANZEER_REST_PORT`, `JANZEER_P2P_PORT`, `JANZEER_DATA_DIR` and `JANZEER_ROOT_NODES` from the environment. Sanity-check the node:

```
curl http://localhost:7019/api/v1/info          # networkId, genesisHash, version, syncStatus
curl http://localhost:7019/actuator/health
```

Wait for `"syncStatus":"SYNCHRONIZED"` before you rely on balances or submit transactions. You do not have to run a node to build an app: the public endpoints of the network (`https://explorer.<domain>/api/v1/`, `/rpc`) speak the same API.

1.2 Networks

There is one public network, the mainnet. A bundle belongs to exactly one network id:

Network	`networkId`	Use
Mainnet	`janzeer`	the live chain — the only public network
Private test network	`janzeer-testnet`	a network you run yourself (the local developer network of the SDK conformance kit, or your own nodes); coins on it have no value. There is no public testnet.

The `networkId` is part of every signed transaction preimage (§3) and of the P2P handshake, so a transaction signed for one network is invalid on the other. Read it from `GET /api/v1/info` instead of hard-coding it, as the official wallets do.

1.3 Test coins

There is no public faucet. To develop against a chain with free coins, run a private network: the SDK conformance kit's `e2e/node-up.sh` boots a local 4-node network whose node 1 runs a faucet (`POST /api/v1/faucet` with `{"address": "0x…"}`, rate-limited per address and per IP) and funds a test wallet. A node refuses the faucet on the mainnet id. On the mainnet, coins come from a transfer, from the jPong game's monthly payout, or from block rewards as a validator.

1.4 Genesis config & chain-spec pinning

The genesis state is **config-driven**. The relevant keys are:

Key	Purpose
`janzeer.genesis.anchors`	the bootstrap anchor validators
`janzeer.genesis.signer-secret`	the ceremony key that signs only the two bootstrap blocks (not an anchor)
`janzeer.genesis.founder`	optional founder custody address (carved out of the treasury)
`janzeer.genesis.distribution`	optional ecosystem/distribution pool custody address
`janzeer.dev-fund`	optional `0xaddr:amount` development pre-fund for a private network, drawn from the treasury (refused on the mainnet)
`consensus.network-id`	the network-id string (default `janzeer`), bound into the signed preimage and the P2P handshake

The consensus-critical parameter set (everything under `consensus.*`) is the canonical rule-set fixed at genesis. Each node computes a **chain-spec digest** (a double-SHA256 over those params) and exchanges it in the P2P handshake; a peer whose digest differs is rejected before the challenge. A node that edits any consensus parameter forks itself off the network rather than silently diverging — so integrators can rely on a consistent rule-set across authenticated peers.

---

2. REST API reference

2.1 Conventions

- **Base path:** `/api/v1/` on port `7019`. All methods are `GET` unless noted.
- **Response envelope:** every response is wrapped as:

```
{
  "timestamp": 1719700000000,
  "version": "1.1.0",
  "payload": { }
}
```

The actual result is always in `payload`. The envelope is omitted only for `/docs/**`, `/swagger-resources`, and `/actuator/**`.

- **Errors:** returned as

```
{
  "status": 404,
  "message": "not found",
  "type": "SomeErrorType"
}
```

(`type` is optional.) Validation failures → `400`; missing resources → `404`.

- **Pagination:** list endpoints accept these query params:

Param	Range / default	Meaning
`page`	`≥ 0`, default `0`	zero-based page index
`size`	`1–100`, default `10`	page size
`sortBy`	field name	sort field
`sortDirection`	default `DESC`	`ASC` or `DESC`

A paged payload looks like:

```
{
  "total": 42,
  "list": [ ],
  "page": 0,
  "pageSize": 10,
  "totalPages": 5
}
```

Amounts are returned as decimal strings/`BigDecimal` with **8-decimal** scale (see the Yellow Paper for amount scaling).

2.2 Wallets — `/api/v1/wallets`

Method	Path	Returns
GET	`/wallets/{address}`	confirmed balance (`BigDecimal`, confirmed − unconfirmed, floored at `0`)
GET	`/wallets/{address}/nonce`	next expected nonce for the address
POST	`/wallets/validateAddress`	validate an address string

```
curl http://localhost:7019/api/v1/wallets/0x06e1c0fa9955a700876f8cb0acc7f13fba9fb8ba/nonce
```

```
{ "timestamp": 1719700000000, "version": "1.1.0", "payload": 0 }
```

2.3 Transfers — `/api/v1/transactions/transfers`

Method	Path	Query / body	Returns
GET	`/transactions/transfers`	`?address=&unconfirmed=` + pagination	paged transfers
GET	`/transactions/transfers/{hash}`	—	one transfer by tx hash
POST	`/transactions/transfers`	tx request body	submits a transfer → `201`

2.4 Validators (registration tx) — `/api/v1/transactions/validators`

Method	Path	Returns
GET	`/transactions/validators` (+ pagination)	paged validator-registration txs
GET	`/transactions/validators/{hash}`	one validator tx
POST	`/transactions/validators`	register as a validator → `201`

Registers the node as a validator. Fee is exactly **3**; a **non-refundable** deposit of exactly **2000** goes to the treasury bank. `validatorKey` must be unique. There is no vote, approval, or lock-up — admission is permissionless, and the deposit plus one-seat-per-key are the Sybil guard. Note: the internal transaction/state TYPE names, the `*-promoter-tx` / `promoters-count` config keys, and the DB tables keep the legacy name **promoter** — the endpoints and JSON fields are validator.

2.5 Exit-validators (retirement tx) — `/api/v1/transactions/exit-validators`

Method	Path	Returns
GET	`/transactions/exit-validators` (+ pagination)	paged exit txs
GET	`/transactions/exit-validators/{hash}`	one exit tx
POST	`/transactions/exit-validators`	gracefully retire a validator → `201`

An ExitPromoterTx removes the validator from the active set and slot schedule at the next epoch boundary. Fee ≥ `0.01`; the payload carries only `validatorKey`. The registration deposit is **not** returned — it stays non-refundable by design.

2.6 Rewards — `/api/v1/transactions/rewards` (read-only)

Method	Path	Returns
GET	`/transactions/rewards` (+ pagination)	paged system reward txs
GET	`/transactions/rewards/{hash}`	one reward tx

RewardTx is system-generated (bank → producing validator, `100%`, fee `0`, nonce `0`) and **not** user-submittable.

2.7 Blocks — `/api/v1/blocks/main` and `/api/v1/blocks/genesis`

Both resources share the same shape:

Method	Path	Returns
GET	`/blocks/main/{hash}`	a main block by hash
GET	`/blocks/main/{hash}/previous`	the previous block
GET	`/blocks/main/{hash}/next`	the next block
GET	`/blocks/main` (+ pagination)	paged main blocks

Replace `main` with `genesis` for genesis blocks (epoch-boundary blocks that freeze the active validator set).

2.8 Validators (state) — `/api/v1/validators`

Method	Path	Returns
GET	`/validators` (+ pagination, by registration order)	all validators, ordered by registration; each row carries `status` (`ACTIVE` in this epoch's set · `PENDING` registered after this epoch began, seated at the next boundary · `STANDBY` registered earlier but not seated this epoch · `EXITED` · `BANNED`), `registeredAt`, `lastBlockHeight` / `lastBlockTime` (newest block it produced, null if none) and `joinsAtHeight` (PENDING/STANDBY: the next epoch genesis height) — node 0.1.3
GET	`/validators/active` (+ pagination)	the current active set
GET	`/validators/view` (+ pagination)	validator view records

Election is by **registration order** (longest-registered first, by `createDate` then id), not by votes or rating — there is no delegation.

2.9 Explorer — `/api/v1/explorer/info`

```
curl http://localhost:7019/api/v1/explorer/info
```

Payload fields:

Field	Meaning
`nodesCount`	known nodes
`blocksCount`	total blocks
`transactionsCount`	total txs
`blockProductionTime`	avg block time
`transactionsPerSecond`	TPS
`epochNumber`	current epoch index
`epochDate`	current epoch start
`validatorsCount`	validator count
`maxSupply`	`100,000,000`
`circulatingSupply`	`maxSupply` minus the treasury (up to node 0.1.13); from 0.1.14 minus every protocol- or project-held account — the same figure as `/supply/circulating`

Supply — `/api/v1/supply` (from node 0.1.14). `GET /supply` → `{ total, circulating, nonCirculating: [{label, address, balance}] }` in the usual envelope; `GET /supply/total` and `GET /supply/circulating` answer with **one bare number as `text/plain`**, with no envelope (the form market-data sites poll). Circulating = total − the current balances of the treasury, the founder and ecosystem genesis accounts, the founding validators' wallets and the further project-held wallets the node lists for its chain (labels `treasury`, `founder`, `ecosystem`, `validator`, `project`). See the Integration Guide §7.

2.10 Node info — `/api/v1/info`

Method	Path	Returns
GET	`/info`	`{ nodeKey, host, port, networkId, genesisHash, chainSpecDigest, version, apiVersion, protocolVersion, syncStatus, faucet }` — read `networkId` and sign every transaction with it
GET	`/info/version`	`{ version, apiVersion, protocolVersion }` (build, API, wire protocol — as in `/info`)
GET	`/info/update`	the node's version against the latest published release (daily manifest check, notification only; node 0.1.3)
GET	`/info/sync`	catch-up progress: `syncStatus`, `tipHeight`, `stagedBlocks` (downloaded and verified, not yet committed), `targetHeight`, `peers` — a fresh node stages a whole epoch before its first commit (node 0.1.2)
GET	`/info/uptime`	node uptime
GET	`/info/genesis`	the chain's public `genesis.json` (what a downloaded node boots from)
GET / POST	`/faucet`	private test networks only (a node started with a faucet key on a test network id): status / `{address}` → a 10 JNZ drip (`429` + `Retry-After` inside the cooldown). `404` on the mainnet — there is no public faucet.

2.11 Actuator & API docs (unwrapped)

Path	Purpose
`/actuator/health`	liveness/health
`/actuator/info`	build info
`/actuator/metrics`	metrics
`/actuator/prometheus`	Prometheus scrape endpoint
`/docs/swagger-ui.html`	Swagger UI
`/docs/api-docs`	OpenAPI document

These responses are **not** wrapped in the standard envelope.

2.12 Token transactions — `/api/v1/transactions/tokens`

Method	Path	Query / body	Returns
GET	`/transactions/tokens`	`?tokenId=&unconfirmed=` + pagination	paged token txs
GET	`/transactions/tokens/{hash}`	—	one token tx by hash
POST	`/transactions/tokens`	tx request body	submits a token op → `201`

A single endpoint carries every native-token operation (create / mint / burn / setcap / transfer) via the `op` field. See §4 (Native tokens) for the op set, the signed byte layout, and the authority rules.

2.13 Tokens (state) — `/api/v1/tokens`

Method	Path	Returns
GET	`/tokens` (+ pagination)	all token definitions
GET	`/tokens/{tokenId}`	one token definition + current supply
GET	`/tokens/{tokenId}/holders` (+ pagination)	a token's holders
GET	`/tokens/balances/{address}` (+ pagination)	a holder's token balances

Token amounts (`cap`, `totalSupply`, balances) are **integer base units** carried as decimal strings — a token has its own `decimals` (display-only), so they are **not** coin-scaled like native-coin amounts.

2.14 JSON-RPC 2.0 — `POST /rpc` and `ws://…/rpc/ws`

A second door onto the **same services and rules** as the REST API, in the shape exchange and custody integrators expect. Nothing changes in consensus, the wire protocol between nodes, or the signed-transaction format: a `janzeer_send*` call takes exactly the fields the matching REST `POST` body takes, and the node verifies signatures — it never signs. `GET /rpc` returns the method list (`janzeer_methods` does the same).

- **Envelope:** standard JSON-RPC 2.0 — `{"jsonrpc":"2.0","id":1,"method":"janzeer_getTip","params":{}}` → `{"jsonrpc":"2.0","id":1,"result":{…}}` or `{"jsonrpc":"2.0","id":1,"error":`

`{"code":…,"message":…,"data":…}}`. The REST `{timestamp, version, payload}` envelope is **not** applied. HTTP status is always `200` (`204` for a batch made only of notifications).

- **Params** by name (`{"address":"0x…"}`) or by position (`["0x…"]`, in the order listed below). Batches (arrays of requests) are processed in order; requests without `id` are notifications and get no response.
- **Finality is instant:** a transaction reported `FINAL` sits in a committed block (2f+1 COMMIT quorum). There is no confirmation count to wait for.
- **Error codes:** `-32700` parse, `-32600` invalid request, `-32601` unknown method, `-32602` invalid params (`data` lists the field messages for a malformed transaction), `-32603` internal; application codes `-32000` not found, `-32001` transaction rejected (`data.type` names the rule when known), `-32002` node synchronizing, `-32003` a range or timeout exceeds the server cap, `-32004` rate limit exceeded.
- **Limits:** per client IP the node accepts 20 requests/s with a burst of 40 on `/rpc` and on the REST transaction submits (HTTP `429` + `Retry-After: 1` beyond that; reads elsewhere are unlimited); a `/rpc` body may be at most 512 KiB (`413`) and a batch at most 50 requests. Public nodes add nginx limits in front (operator guide / VPS §9).

Method	Params (positional order)	Result
`janzeer_methods`	—	every method with its parameter order
`janzeer_getInfo`	—	`chainId`, `chainSpecDigest`, `genesisHash`, versions, `nodeKey`, `tip {height, hash, timestamp}`, `epoch`, `syncStatus`, `synchronized`, `peers`, `finality: "instant"`
`janzeer_getChainSpec`	—	the genesis-pinned consensus parameters, fees, deposit, `maxSupply`, `digest`
`janzeer_getStats`	—	the explorer statistics (blocks, transactions, pending, TPS, epoch, supply)
`janzeer_getAccount`	`address`	`balance` (spendable = committed − pending debits), `committedBalance`, `nextNonce`, `committedNonce`, `pendingCount`, `isValidatorWallet`
`janzeer_getBalance` / `janzeer_getNonce`	`address`	the spendable balance / the nonce to sign the next transaction with (pipelined)
`janzeer_getTokenBalances`	`address, page?, size?`	paged JZT-1 balances
`janzeer_getTransactionsByAddress`	`address, page?, size?`	paged final transfers sent or received + `pending` (the address's mempool entries)
`janzeer_sendTransfer` / `janzeer_sendToken` / `janzeer_sendRegisterValidator` / `janzeer_sendExitValidator`	the signed transaction object (the REST `POST` body)	`{hash, status: "PENDING"}`; re-submitting the same transaction is idempotent
`janzeer_sendRawTransaction`	`type, tx` — `transfer | token | registerValidator | exitValidator`	as above
`janzeer_getTransactionByHash`	`hash`	`status` `FINAL` (with `blockHeight`, `blockHash`, `receipt {successful, results[]}`) | `PENDING` | `UNKNOWN`; one shape for every `type`
`janzeer_waitForFinality`	`hash, timeoutMs?`	long-poll (default 30 s, cap 60 s): the `FINAL` transaction, or the

Method	Params (positional order)	Result
		current view with `timedOut: true`
`janzeer_estimateFee`	`kind` — `transfer | token | tokenCreate | registerValidator | exitValidator`	`{fee, rule: "minimum" | "exact", feeMarket: false}`
`janzeer_getTip`	—	the tip block header (`type: "main" | "genesis"`)
`janzeer_getBlockByNumber` / `janzeer_getBlockByHash`	`height` | `hash`, `includeTransactions?`	block header (+ `transactions[]` with receipts)
`janzeer_getBlocks`	`fromHeight, toHeight, includeTransactions?`	up to 100 blocks, ascending
`janzeer_getToken` / `janzeer_listTokens` / `janzeer_getTokenHolders`	`tokenId` / `page?, size?` / `tokenId, page?, size?`	token definition + supply / paged definitions / paged holders
`janzeer_listValidators` / `janzeer_getActiveValidators` / `janzeer_getValidator`	`page?, size?` / — / `key`	`{key, walletAddress, registeredAt, active, banned, exited}`
`janzeer_getEpoch`	—	`index`, `genesisHeight/Hash`, `startTimestamp`, `nextGenesisHeight`, `slotPeriodMs`, `currentSlot`, `currentSlotOwner`, `validators[]`
`janzeer_getGenesis`	—	the chain's public genesis specification (`specVersion`, `networkId`, `anchors[]`, `treasury`, `signerPublicKey`, `mainBlock/genesisBlock {hash, signature}`) — the `genesis.json` a downloaded node boots from
`janzeer_subscribe` (WebSocket only)	`kind, params`	a subscription id; `kind` = `newBlocks` `{fromHeight?, includeTransactions?}` or `addressActivity` `{addresses: [...]}`
`janzeer_unsubscribe` (WebSocket only)	`subscription`	`true` / `false`

Over the WebSocket every method above works unchanged, and subscriptions push notifications:

```
{"jsonrpc":"2.0","method":"janzeer_subscription",
 "params":{"subscription":"1f0c…","kind":"addressActivity",
           "result":{"blockHeight":128,"blockHash":"…","transaction":
{"hash":"…","type":"transfer","status":"FINAL", "…":"…"}}}}
```

`newBlocks` with `fromHeight` replays the committed main blocks from that height before following the tip, in order, so a reconnecting indexer never misses a block. `addressActivity` delivers every **final** transaction whose sender or recipient is a watched address (transfers, token operations, rewards, validator registrations) — the deposit-detection primitive. Limits: 16 subscriptions and a bounded send buffer per connection (a client that cannot keep up is disconnected), 1,000 addresses per subscription.

Worked example — submit and wait:

```
curl -s localhost:7019/rpc -H 'Content-Type: application/json' -d '[
  {"jsonrpc":"2.0","id":1,"method":"janzeer_getNonce","params":
["0x06e1c0fa9955a700876f8cb0acc7f13fba9fb8ba"]},
  {"jsonrpc":"2.0","id":2,"method":"janzeer_sendTransfer","params":
{"timestamp":1788650000000,"fee":"0.01","nonce":0,

"hash":"…","senderAddress":"0x06e1…","senderSignature":"…","senderPublicKey":"…","amount":"2.5
","recipientAddress":"0x598b…"}},
  {"jsonrpc":"2.0","id":3,"method":"janzeer_waitForFinality","params":
{"hash":"…","timeoutMs":30000}}
]'
```

---

3. Building & submitting a transaction (client-side)

Janzeer is **non-custodial**: the node has no key-generation or seed endpoints. Your app/SDK derives keys, builds the signed preimage, hashes, signs, and POSTs the finished transaction. The node only verifies.

3.1 Key derivation (BIP39 + `m/0/0/0`)

The derivation scheme is Janzeer-specific — it resembles BIP39/BIP32 but uses custom salt strings, so do **not** reuse a stock library blindly. Summary (full byte details in the Yellow Paper):

1. **Mnemonic → seed:** `PBKDF2-HMAC-SHA512`, `2048` iterations, `64`-byte output. Password = `NFKD(mnemonic)` UTF-8. Salt = the literal string `@_Janzeer_Blockchain_@` + `NFKD(passphrase)` (passphrase defaults to `""`). Note: this is **not** BIP39's `"mnemonic"` salt prefix.
2. **HD master key:** `I = HMAC-SHA512(key = "@_Janzeer_Blockchain_@", msg = seed)`; `IL[0:32]` = master private key, `IR[32:64]` = chain code. The HMAC key is the **same** salt string, not BIP32's `"Bitcoin seed"`.
3. **Derivation path:** `m/0/0/0` (account / wallet / address), all **non-hardened**, curve secp256k1, compressed 33-byte pubkeys. Non-hardened CKD: `data = compressedParentPub(33B) ‖ ser32(i)`; `I = HMAC-SHA512(chainCode, data)`; `childPriv = (IL + parentPriv) mod n`.
4. **Address:** `"0x" + hex(keccak256(compressedPubKey)[0:20])` — **Keccak-256** (not NIST SHA3) over the **33-byte compressed** pubkey, take the **first 20 bytes**. This is **not** the Ethereum scheme. The **canonical stored/hashed form is all-lowercase hex**; a real EIP-55 mixed-case checksum

is applied at the wallet/API display boundary only (presentational, not consensus-relevant), and all-lowercase input stays accepted.

3.2 The signing pipeline

For every transaction:

1. **Fetch the nonce** for your sender address:

   ```
   curl http://localhost:7019/api/v1/wallets/0x06e1.../nonce
   ```

2. **Build the preimage** by concatenating the base fields and the type-specific payload (byte layouts in the Yellow Paper). The preimage now **begins with the network-id** (`consensus.network-id`, default `janzeer`) for cross-chain replay protection — every preimage starts with the bytes `6a616e7a656572` ("janzeer"). All multi-byte integers are **big-endian**; strings/addresses/keys are their raw UTF-8 / ASCII-hex bytes; amounts use `toScaledLong(x) = x.setScale(8, HALF_UP) × 10^8` as an 8-byte big-endian long.

   Base (every tx): `networkId.utf8 ‖ timestamp(8) ‖ toScaledLong(fee)(8) ‖ nonce(8) ‖ senderAddress.utf8 ‖ payload()` TransferTx payload: `toScaledLong(amount)(8) ‖ recipientAddress.utf8 ‖ (data ?? "").utf8` PromoterTx payload: `toScaledLong(amount)(8) ‖ validatorKey.utf8` ExitPromoterTx payload: `validatorKey.utf8` TokenTx payload (all ops): `op(1) ‖ LP(tokenId) ‖ LP(symbol) ‖ LP(name) ‖ int32(decimals) ‖ LP(cap) ‖ LP(amount) ‖ LP(recipient)` where `LP(x) = int32(len) ‖ utf8(x)` — see §4

   A native-token `TokenTx` is built exactly the same way: the same base preimage, then the token payload above. The base `fee` is still `toScaledLong`-scaled (it is paid in the native coin), but the token `cap` and `amount` fields are **integer base units carried as their decimal string** — a token has its own `decimals` — so they are **not** run through `toScaledLong`.

3. **Hash:** `txHash = doubleSHA256(preimage)`, lowercase hex.

4. **Sign:** ECDSA over secp256k1, deterministic **RFC-6979** (HMAC-SHA256 `k`), **canonical low-S** enforced, **DER**-encoded then **Base64**. The signature is over the **raw 32-byte digest** — no extra hashing — so set `prehash: false` in your signing library.

5. **POST** the finished tx to the matching endpoint.

3.3 Worked Transfer example (JS-ish pseudocode)

```
// 0. Derive keys from mnemonic (Janzeer scheme — see §3.1 / Yellow Paper)
const { privKey, pubKeyCompressed, address } = deriveJanzeer(mnemonic); // m/0/0/0

// 1. Fetch the nonce
const nonce = (await getJSON(
  `http://localhost:7019/api/v1/wallets/${address}/nonce`)).payload;

// 2. Transaction fields
const networkId = "janzeer";   // consensus.network-id (per chain)
const tx = {
  timestamp: Date.now(),
  fee: "0.01",                 // ≥ MINIMUM_FEE
  nonce,
  senderAddress: address,
  amount: "1.5",               // ≥ 0.1 unless a memo is attached
  recipientAddress: "0xRECIPIENT...",
  data: ""                     // optional ≤256B memo
};

// 3. Build the preimage (big-endian; amounts scaled ×10^8)
const preimage = concatBytes(
  utf8(networkId),             // network-id prefix (bytes 6a616e7a656572)
  i64be(tx.timestamp),
  i64be(toScaledLong(tx.fee)),
  i64be(tx.nonce),
  utf8(tx.senderAddress),
  // TransferTx payload:
  i64be(toScaledLong(tx.amount)),
  utf8(tx.recipientAddress),
  utf8(tx.data)
);

// 4. Hash + sign
const digest = sha256(sha256(preimage));            // double-SHA256, 32 bytes
const sigDER = secp256k1.sign(digest, privKey, {    // RFC-6979, low-S
  prehash: false, der: true
});

tx.hash            = toHex(digest);                 // lowercase hex
tx.senderSignature = base64(sigDER);
tx.senderPublicKey = toHex(pubKeyCompressed);       // 33-byte compressed

// 5. Submit
await postJSON("http://localhost:7019/api/v1/transactions/transfers", tx); // → 201
```

3.4 POST request fields by type

Every tx POST carries the common fields: `timestamp`, `fee`, `nonce`, `hash`, `senderAddress`, `senderSignature`, `senderPublicKey`. Then add the type-specific fields:

Tx type	Endpoint	Extra fields
Transfer	`/transactions/transfers`	`amount`, `recipientAddress?`, `data?`
Promoter	`/transactions/validators`	`amount`, `validatorKey`
ExitPromoter	`/transactions/exit-validators`	`validatorKey`
Token	`/transactions/tokens`	`op`, `tokenId?`, `symbol?`, `name?`, `decimals?`, `cap?`, `amount?`, `recipient?`

Minimums and fees (exact): `MINIMUM_FEE = 0.01`, `MINIMUM_AMOUNT = 0.1`, `MAX_DATA_SIZE = 256` bytes; the PromoterTx fee is exactly `3` with a non-refundable deposit of `2000`; ExitPromoterTx fee ≥ `0.01`; the TokenTx CREATE fee is exactly `5` and all other token ops (mint / burn / setcap / transfer) ≥ `0.01`. The token `cap` and `amount` fields are sent as JSON **strings** (integer base-unit decimals) to avoid precision loss. The mempool allows **consecutive** nonces (pipelining): expected nonce = `confirmedNonce + pendingCount`.

Replacing a pending transaction ("speed up"). A pending transaction occupies its nonce; a sender's later nonces wait behind it. To replace it, submit a transaction with the **same nonce and a strictly higher fee** — the node swaps it in and drops the old one (an equal or lower fee is refused: `A pending transaction with nonce N exists; a replacement must pay a higher fee than …`). Only the replacement can ever be charged. `GET /api/v1/transactions/transfers?address=…&unconfirmed=true` lists a wallet's pending transfers; since node 0.1.0 every transfer in the REST response carries its **`nonce`**, which is what a wallet needs to rebuild the transfer. A pending transaction that is not mined within 6 h is evicted. The official wallets expose this as *Speed up*: same recipient, amount and memo, fee chosen by the user (suggested: double).

Timestamp. Set `timestamp` to the current time in milliseconds **when you sign** — it is part of the signed preimage. The node accepts a transaction whose timestamp is at most **60 s ahead** of its clock and at most **6 h old**; inside a block the consensus window is `block − 6 h … block + 2 min`. The six hours are not a delay: they are the mempool lifetime, i.e. how long a signed transaction may wait before a block includes it (normal inclusion takes one or two 15 s slots). Do not pre-sign transactions for later and do not reuse an old signed payload — re-sign with a fresh timestamp. A device with a wrong clock gets `400 Transaction timestamp is in the future (check the device clock)`; an old payload gets `… is older than 6 h — sign it again`. `MAX_DATA_SIZE` counts **UTF-8 bytes**, not characters (an Arabic letter is 2 bytes, an emoji 4).

---

4. Native tokens (JZT-1)

Janzeer supports optional, validator-issued **fungible tokens** that live alongside the native coin on the same chain. They are a **separate ledger**: a token's supply is **not** part of the coin's fixed `100,000,000` supply and never touches emission, the treasury, or block rewards. Every token operation still pays its fee **in the native coin**. Tokens were live from genesis on a fresh network (`token-activation-height` defaults to `0`, pinned in the chain-spec digest).

4.1 One transaction type, an `op` byte

There are no five separate token tx types — every token operation is a single `TokenTx` carrying an `op` byte:

op	Operation	Who may submit
0	CREATE	a registered validator (a PromoterState.walletAddress, active or standby)
1	MINT	the token's issuer (its creator) only
2	BURN	the token's issuer only
3	SETCAP	the token's issuer only
4	TRANSFER	anyone holding a balance

4.2 Signed payload byte layout

The signed preimage is the usual base (networkId ‖ timestamp ‖ toScaledLong(fee) ‖ nonce ‖ senderAddress) followed by one canonical token payload for **every** op, each variable field length-prefixed:

```
op(1) ‖ LP(tokenId) ‖ LP(symbol) ‖ LP(name) ‖ int32(decimals) ‖ LP(cap) ‖ LP(amount) ‖
LP(recipient)
    where LP(x) = int32(len) ‖ utf8(x)      // 4-byte big-endian length, then the UTF-8 bytes
```

- op ∈ { CREATE 0, MINT 1, BURN 2, SETCAP 3, TRANSFER 4 }.
- Absent fields are the empty string (length 0) — e.g. a TRANSFER carries tokenId, amount, and recipient and leaves symbol / name / cap empty.
- **cap and amount are INTEGER base units carried as their decimal STRING — NOT toScaledLong.** A token has its own decimals (0–18, display-only), so its base units are independent of the native coin's 8-decimal scaling.
- The hash and signature are computed over this preimage exactly like any other tx (double-SHA256, RFC-6979 low-S, DER→Base64). See §3 for the full pipeline — a TokenTx is built and signed identically, just with this payload.

4.3 Authority & rules

- **CREATE (op 0)** — only a **registered validator** may create a token. It sets symbol (1–12 chars), name (1–40), decimals (0–18, display-only), cap (max supply), and an optional initialSupply (≤ cap, credited to the creator). The new **tokenId is the CREATE tx hash** (globally unique). There is no symbol registry, so symbols are **not** unique. Pays the exact CREATE fee of 5.
- **MINT / BURN / SETCAP (ops 1-3)** — only the token's **issuer** (its creator) may raise supply, lower supply, or change the cap. The invariant 0 ≤ totalSupply ≤ cap is enforced on both the live and the sync path, and SETCAP cannot drop the cap below the already-minted supply. Fee ≥ 0.01.
- **TRANSFER (op 4)** — **anyone** holding a token balance may transfer it to any address. Fee ≥ 0.01.

These economic/authority rules are enforced in the transaction service (so they run identically on the producer and every re-executing peer) **and** re-checked at mempool intake, so an un-mineable token tx is rejected at submission (HTTP 400) rather than wasting a block slot.

4.4 Reading token state

Use the endpoints in §2.12 (token transactions) and §2.13 (token state): `POST /api/v1/transactions/tokens` to submit an op, and the `GET /api/v1/tokens/*` reads for definitions, supply, holders, and per-address balances. See the REST reference above for full detail.

---

5. Build on Janzeer — official SDKs & conformance

You do not have to re-implement the crypto scheme of §3: five official SDKs do it, are proven byte-for-byte against the node, and wrap the REST and JSON-RPC surfaces of §2 with typed clients. All are Apache-2.0, non-custodial (keys never leave your process) and tested against node 0.1.0 / API 1.1.0.

5.1 The SDKs

Language	Package	Install	Platforms
TypeScript / JavaScript	`janzeer-sdk`	`npm install janzeer-sdk`	Node 20+, browsers (pure JS, works on plain-http pages), Deno/Bun
Dart / Flutter	`janzeer_sdk`	`dart pub add janzeer_sdk`	Dart VM, Flutter mobile/desktop/web
Kotlin / JVM / Android	`org.janzeer:janzeer-sdk-client` (+ `janzeer-sdk-core`)	Gradle/Maven dependency	JVM 11+, Android API 26+, plain Java
Python	`janzeer-sdk` (import `janzeer`)	`pip install "janzeer-sdk[ws]"`	Python 3.10–3.13 on Linux, macOS, Windows; blocking and `asyncio` clients
Go	`github.com/janzeerorg/janzeer-sdk-go`	`go get github.com/janzeerorg/janzeer-sdk-go`	Go 1.26+, no cgo (cross-compiles everywhere)

Each SDK has the same shape: `Mnemonic` / `Account` (keys, addresses, signing), `TxBuilder` (transfer, validator registration/exit, all five JZT-1 token ops) → `UnsignedTx` → `SignedTx`, a typed REST client (`JanzeerClient`), JSON-RPC over HTTP (`JanzeerRpc`) and WebSocket with `newBlocks` / `addressActivity` subscriptions (`JanzeerRpcWs`), `waitForFinality` / `sendAndWait` helpers, a typed error hierarchy (`NonceMismatch`, `TxRejected` with the node's `type`, `RateLimited`, …) and an optional encrypted vault for the phrase at rest. Amounts are decimal strings and responses are parsed losslessly — never through a floating-point number.

The whole integration is six lines in any of them — derive → nonce → build → sign → submit → wait:

```
// TypeScript
const client = new JanzeerClient('http://localhost:7019'); const rpc = new
JanzeerRpc('http://localhost:7019');
const me = Account.fromMnemonic(phrase);
const tx = me.signTx(TxBuilder.transfer({ from: me.address, to, amount: '1.25', nonce: await
client.nonce(me.address) }));
await client.submit(tx);
const final = await waitForFinality(rpc, tx.hash);   // one or two 15-second slots; a
committed block is final
```

```
// Dart
final tx = me.signTx(TxBuilder.transfer(from: me.address, to: to, amount: '1.25', nonce: await
client.nonce(me.address)));
await client.submit(tx);
final fin = await waitForFinality(rpc, tx.hash);
```

```
// Kotlin (Java: TxBuilder.transfer(from, to, "1.25", nonce), Helpers.waitForFinality(rpc,
tx.getHash()))
val tx = me.signTx(TxBuilder.transfer(from = me.address, to = to, amount = "1.25", nonce =
client.nonce(me.address)))
client.submit(tx)
val fin = Helpers.waitForFinality(rpc, tx.hash)
```

```
# Python (asyncio twins: AsyncJanzeerClient, AsyncJanzeerRpc, JanzeerRpcWs, await_finality)
tx = me.sign_tx(TxBuilder.transfer(sender=me.address, to=to, amount="1.25",
nonce=client.nonce(me.address)))
client.submit(tx)
final = wait_for_finality(rpc, tx.hash)
```

```
// Go (every call takes a context; errors are typed values for errors.As)
utx, _ := janzeer.NewTransfer(janzeer.Transfer{Common: janzeer.Common{From: me.Address(),
Nonce: nonce}, To: to, Amount: "1.25"})
tx, _ := me.SignTx(utx)
_, err := c.Submit(ctx, tx)
final, err := rpc.WaitForFinality(ctx, r, tx.Hash)
```

Every SDK repository ships a README with this quickstart, guides (wallet & keys, sending each transaction type, reading chain data, JSON-RPC & subscriptions, errors & nonces, vault, migration), a generated API reference (TypeDoc / dartdoc / Dokka) and runnable examples. The Janzeer explorer wallet, the Telegram mini-app wallet and the Flutter wallet are built on them.

5.2 The conformance kit

Whatever the language, a client must reproduce the crypto scheme **byte-for-byte** or the node rejects its transactions. The public conformance kit (`sdk_conformance/`) is what the SDKs — and any third-party client — are tested against:

- **Vectors** (`vectors/wallet-parity-vectors.json`, format v2, generated by the node's own `WalletParityVectorsTest`): mnemonic → seed → `m/0/0/0` key → address; a raw-key address; for every client-submittable transaction type (transfer with and without memo, validator registration, validator exit, token CREATE and TRANSFER) the exact preimage bytes, hash and

signature; an EIP-55 checksum table; a `toScaledLong` table. Every preimage begins with the network-id bytes (`6a616e7a656572` = "janzeer"). Canonical vector: mnemonic `abandon … about` → `0x06e1c0fa9955a700876f8cb0acc7f13fba9fb8ba`.

- **The e2e flow** (`e2e/SPEC.md`), 14 steps against a real network: version gate, account, subscribe first, build/sign, submit over REST (201), idempotent resubmit over RPC, finality on another node, exact balance arithmetic, the WebSocket notification, and three negative cases (nonce gap → `INVALID_NONCE`, forged signature → `INCORRECT_SIGNATURE`, bad address → `-32602`).
- A vault interop fixture (PBKDF2-SHA256/250k + AES-256-GCM blob) so encrypted phrases move between SDKs.

5.3 Writing your own client

Reproduce the vectors in this order, then run the SPEC flow:

- [ ] `scaled` table: `toScaledLong(0.01) == 1_000_000`, `toScaledLong(1.5) == 150_000_000` (8 decimals, HALF_UP)
- [ ] `abandon … about` → `0x06e1c0fa9955a700876f8cb0acc7f13fba9fb8ba` (PBKDF2 salt and HMAC key `@_Janzeer_Blockchain_@`, path `m/0/0/0`)
- [ ] Address is lowercase, `0x` + 40 hex chars, from `keccak256(compressedPub)[0:20]`; EIP-55 casing is display only
- [ ] Every `preimageHex` matches; the hash is `doubleSHA256(preimage)`
- [ ] Signature is RFC-6979, low-S, DER, Base64, over the raw 32-byte digest (`prehash: false`)
- [ ] Errors: read `payload` when the body has one (the envelope wraps errors too), an array payload is a validation error, `type` names the rejection

The SDK sources are the reference implementations (`janzeer-sdk` `src/crypto`, `janzeer_sdk` `lib/src`, `janzeer-sdk-core`); the authoritative byte layouts are in the Yellow Paper.

---

6. Non-custodial note

Janzeer's node is strictly **non-custodial**:

- Keys are derived and held **only client-side**; the node never sees a private key or seed.
- There are **no key-generation, seed, or signing endpoints** on the node — building and signing transactions is entirely your app's responsibility (see §3).
- The node's role is to **verify** signatures, enforce per-account nonces plus the network-id in the signed preimage (replay protection, including cross-chain) and the negative-balance guard, and apply state transitions.
- The genesis treasury bank `0x0000000000000000000000000000000000000000` is key-less and capture-proof; only protocol reward/fee logic can move its coins.

Because finality requires a live quorum of the active set, a single node cannot finalize on its own — use the local multi-node net (§1.2), at least 4 nodes, for realistic end-to-end testing.

Document status: version 1.1 (official SDKs, 2026-09-14). Reflects node version 0.1.0, including native tokens (JZT-1). Exact byte-level facts (preimage layouts, hashing, amount scaling, derivation constants) live in the Yellow Paper.