corelib

User Guide · version 0.4.0 · 2026-10-16

A multi-runtime TypeScript toolkit: resilient HTTP, logging, configuration and databases (`@ckirg/corelib`), Nasdaq market data and streaming (`@ckirg/corelib-markets`), and an edge proxy service (`@ckirg/corelib-cloud`).

Example convention: a heading tagged **ran locally** was executed against the repository's built packages (using local stub servers and scratch files; the output shown is real, with timestamps, ports and ANSI colours trimmed). A heading tagged **illustrative** needs credentials, a paid or public network service, or a deployment, so it was not run and its output is omitted or marked illustrative.

Contents

1. Overview
2. Installation and requirements
3. Quick start
4. Core concepts
5. HTTP requests: endPoint, endPoints, RequestProxied
6. Logging
7. Configuration with ConfigManager
8. Databases
9. Runtime utilities, cron and the native core
10. Market data (corelib-markets)
11. Edge proxy service (corelib-cloud)
12. Configuration reference
13. Recipes
14. Troubleshooting
15. Where the docs and the code disagree
16. Appendix: quick reference

1. Overview

corelib is a pnpm monorepo that publishes three npm packages for Node.js (24+), Bun and Deno, plus a Rust native addon that the TypeScript packages load over N-API.

Package	Use it for
`@ckirg/corelib`	Resilient HTTP (`endPoint` , `endPoints` , `RequestProxied`), structured `logger` , `ConfigManager` , SQLite/PostgreSQL `createDatabase` , runtime helpers, cron include/exclude, native `Core` .
`@ckirg/corelib-markets`	Nasdaq market status and monitor, symbol directory (`MarketSymbols`), CNN Fear and Greed, Yahoo historical data, Nasdaq 100 list, real-time Alpaca / Finnhub / Yahoo streaming.
`@ckirg/corelib-cloud`	A Hono application (`createRouter`) that exposes the above as an authenticated HTTP service on Cloudflare Workers, AWS Lambda or Cloud Run.

This guide is for application developers who consume these packages. It does not cover contributing to the repository.

Smallest useful program ran locally

```
// hello.mjs
import { detectRuntime, getMode, getPlatform, sleep, Core } from '@ckirg/corelib';

console.log(detectRuntime(), getMode(), getPlatform());
const t = Date.now();
await sleep(100);
console.log('slept at least 95 ms:', Date.now() - t >= 95);
console.log('native addon loaded:', Core.isFfiAvailable());
```

Output

```
node development linux
slept at least 95 ms: true
native addon loaded: true
```

2. Installation and requirements

Requirement	Value (from the repo manifests)
Node.js	24 or newer (`engines.node >=24`). Bun 1.1+ and Deno 2.0+ are supported by the isomorphic core.
Package manager for this monorepo	pnpm 11.5.2 (`packageManager` field)
Module system	ESM only (`"type": "module"`). Use `.mjs` files or `"type": "module"` in your `package.json` .
Native binary	Prebuilt `corelib-rust-*.node` for linux-x64, linux-arm64, win32-x64, darwin-x64, darwin-arm64, fetched by a `postinstall` script.

Install into your application illustrative

```
npm install @ckirg/corelib @ckirg/corelib-markets
# or: pnpm add @ckirg/corelib @ckirg/corelib-markets
# or: bun add  @ckirg/corelib @ckirg/corelib-markets
```

Not run: it needs the network. `@ckirg/corelib-markets` depends on a package from JSR, so add this line to your project's `.npmrc` first, or the install fails with a 404 on the transitive `@jsr/...` package:

```
@jsr:registry=https://npm.jsr.io
```

Native binary caveats illustrative

- **Bun** blocks dependency postinstall scripts. Add `"trustedDependencies": ["@ckirg/corelib"]` to your `package.json` .
- The postinstall **skips** the download when `MODE=development` or under GitHub Actions. Do not put `MODE=development` in a consuming app's `.env` .
- Fetch it manually if it was skipped: `node node_modules/@ckirg/corelib/scripts/postinstall.js`
- If the binary is missing the library still loads; only the native features (`Core` , streaming) are disabled. Check `Core.isFfiAvailable()` .

> The per-package READMEs still say the packages are "Not on the public npm registry" and (ts-core, ts-markets) show a v0.1.17 tarball URL. The root README and the current release (0.4.0) say they are on npm. See section 15.

Working from a clone of the monorepo illustrative

```
pnpm install
pnpm build-all        # tsup builds every package into dist/
pnpm test-all:run     # one-shot test run
pnpm typecheck-all
```

The examples in this guide were run against the already-built `dist/` folders of a clone, without modifying it.

3. Quick start

First resilient request, with a result you can branch on ran locally

A throw-away local server stands in for a real API so the example is self-contained. Replace the URL with your own.

```
// quick.mjs
import http from 'node:http';
import { endPoint } from '@ckirg/corelib';

const srv = http.createServer((req, res) => {
  res.setHeader('content-type', 'application/json');
  res.end(JSON.stringify({ symbol: 'AAPL', price: 189.5 }));
}).listen(0, async () => {
  const result = await endPoint(`http://127.0.0.1:${srv.address().port}/quote/AAPL`);
  if (result.status === 'success') {
    console.log(result.value.status, result.value.body);
  } else {
    console.error('failed:', result.reason);
  }
  srv.close();
});
```

Output
```
200 { symbol: 'AAPL', price: 189.5 }
```

That is the whole pattern: nothing throws, every call returns `{ status: 'success', value }` or `{ status: 'error', reason }`.

4. Core concepts

- **Result unions instead of exceptions.** HTTP calls return `RequestResult` (`value` is a `SerializedResponse` with `ok, status, statusText, headers, url, redirected, type, body` and a `cid` correlation id). Database calls return `DatabaseResult`. Market calls return similar unions. Always branch on `status`.
- **Layered configuration.** Priority is CLI flags > `CORELIB_*` environment variables > config file > built-in defaults (section 7).
- **Runtime detection.** The logger and several helpers choose an implementation for node, bun, deno, cloudflare, aws-lambda or gcp-cloudrun. Set `RUNTIME=` to force it.
- **Native core.** Streaming engines and a few helpers run in Rust. They degrade gracefully when the binary is absent.

5. HTTP requests

Retries use full-jitter backoff. GET, HEAD and OPTIONS are retried on network errors, 5xx and up to five 429 responses; other 4xx responses are not retried. Defaults: 50 s timeout, 5 retries, 3 s backoff cap. The examples below run against a small local stub server (full code in the recipe in section 13); `base` is its URL.

5.1 endPoint and endPoints

Successful GET ran locally

```
const ok = await endPoint(`${base}/quote/AAPL`);
console.log(ok.status, ok.value.status, ok.value.body, ok.cid);
```

Output
```
success 200 { symbol: 'AAPL', price: 189.5 } 1
```

HTTP error: a value, not an exception ran locally

```
const nf = await endPoint(`${base}/nope`);
console.log(nf.status, nf.reason.status, nf.reason.statusText);
```

Output
```
error 404 Not Found
```

> On an error result the response `body` is the literal string `"[Body already consumed]"`, so the server's error payload is not available through `reason.body` (observed on 0.4.0; see section 15). A warning is also logged.

Retries on 5xx, with tuned retry options ran locally

The server answers 503 twice, then 200. Options are passed to ky.

```
const fl = await endPoint(`${base}/flaky`, { retry: { limit: 4, backoffLimit: 200 } });
console.log(fl.status, fl.value.body, 'server hits:', hits);
```

Output
```
success { ok: true } server hits: 3
```

POST with JSON and an auth header ran locally

```
const post = await endPoint(`${base}/echo`, {
  method: 'POST',
  json: { a: 1 },
  headers: { Authorization: 'Bearer YOUR_TOKEN' },
});
console.log(post.value.body);

// A pre-serialised string body gets content-type: application/json automatically
const post2 = await endPoint(`${base}/echo`, { method: 'POST', body: JSON.stringify({ b: 2 }) });
console.log(post2.value.body);
```

Output
```
{
  method: 'POST',
  body: '{"a":1}',
  ct: 'application/json',
  auth: 'Bearer YOUR_TOKEN'
}
{ method: 'POST', body: '{"b":2}', ct: 'application/json' }
```

Only GET, HEAD and OPTIONS are retried by default, so a POST that fails is not replayed.

Several URLs in parallel; network failures ran locally

```
const many = await endPoints([`${base}/quote/AAPL`, `${base}/nope`]);
console.log(many.map(r => r.status + ':' + (r.status === 'success' ? r.value.status : r.reason.status)));

// Connection refused (nothing listening): reason is a serialised Error
const dead = await endPoint(`http://127.0.0.1:${closedPort}/`, { retry: { limit: 0 } });
console.log(dead.status, dead.reason.name);
```

Output
```
[ 'success:200', 'error:404' ]
error NetworkError
```

Results come back in input order. Config keys `retrieve.timeout`, `retrieve.retry.limit` and `retrieve.retry.backoffLimit` change the defaults globally (timeout is clamped to 1 000..120 000 ms, retry limit to 0..10, backoff to 0..60 000 ms); options you pass per call win over config.

> If you call `endPoint` before `ConfigManager.initialize()` resolves, a one-time warning is logged (`ConfigManager.get("retrieve.timeout") called before initialize() resolved`) and the seeded defaults are used. Call `await ConfigManager.getInstance().initialize()` at start-up to avoid it (it is suppressed when `NODE_ENV=production`).

5.2 RequestProxied

`new RequestProxied(proxyBaseUrls)` sends each request to a proxy as `<proxy>/?url=<target>` (the proxy is expected to behave like the ts-cloud `GET /api/v1/ky?url=` route). Single requests rotate round-robin and fall back to every other proxy before failing; `endPoints` spreads the batch across proxies. Signature: `endPoint(url, suffix = "", options = {})`.

Rotation and a path suffix ran locally

Two local stub proxies answer with their own name (`via`), the path they received and the `url` query parameter. The suffix call lands on proxy-b because the round-robin pointer was on its turn.

```
const client = new RequestProxied([proxyA, proxyB]);
for (let i = 0; i < 3; i++) {
  const r = await client.endPoint('https://api.example.com/quote/AAPL');
  console.log(r.status, r.value.body.via);
}
const many = await client.endPoints(['https://t1.example.com', 'https://t2.example.com']);
console.log(many.map(r => r.value.body.via));

// Append a path to the proxy base, and pass ky options as the third argument
const s = await client.endPoint('https://api.example.com/q', '/fetch', { headers: { 'x-trace': '1' } });
console.log(s.value.body);
```

Output

```
success proxy-a
success proxy-b
success proxy-a
[ 'proxy-a', 'proxy-b' ]
{ via: 'proxy-b', path: '/fetch', target: 'https://api.example.com/q' }
```

Fallback when a proxy returns 502 ran locally

```
const mixed = new RequestProxied([badProxy502, goodProxy]);
// Argument order is (url, suffix, options): pass '' as the suffix when you only want options
const m = await mixed.endPoint('https://api.example.com/x', '', { retry: { limit: 0 } });
console.log(m.status, m.value.body.via);
```

Output

```
success good
```

> Argument order is `endPoint(url, suffix, options)` , unlike `endPoint(url, options)` of the standalone function. Passing the options object second puts it in the `suffix` slot (observed: it was stringified into the request path as `/[object Object]`).

Only network errors and proxy-originated 502/503/504 count against a proxy's health; a 404 from the target does not. After **3 consecutive** failures a proxy is removed for the lifetime of the instance.

Validation errors ran locally

```
try { new RequestProxied([]); } catch (e) { console.log(e.message); }
try { new RequestProxied(['nope']); } catch (e) { console.log(e.message); }
const bad = await client.endPoint('https://api.example.com/q', 'https://evil.example.org/');
console.log(bad.status, bad.reason.message);   // a suffix may not leave the proxy host
```

Output

```
RequestProxied: at least one proxy URL is required
RequestProxied: invalid proxy URL "nope"
error Invalid proxy suffix
```

6. Logging

`logger` has a strict signature `(msg: string, extras?: object)` with levels `trace, debug, info, warn, error, fatal` , plus `child(bindings)` , `setTelemetry('on'|'off')` , `level` , `flush()` . It is built on pino and picks an implementation per runtime.

Env var	Default	Meaning
`LOG_LEVEL`	`info`	Minimum level (Node implementation).
`LOG_PRETTY`	on unless `NODE_ENV=production`	`true` / `false` forces coloured text vs. one JSON object per line.
`NODE_ENV`	development	`production` switches `getMode()` and disables pretty output.

Basic and structured logging ran locally

```
import { logger } from '@ckirg/corelib';
logger.info('Application started');
logger.error('Database connection failed', { host: 'localhost', port: 5432 });
```

Output

```
[2026-10-16 05:06:05.270 +0300] INFO: Application started
[2026-10-16 05:06:05.278 +0300] ERROR: Database connection failed
    host: "localhost"
    port: 5432
```

(On a terminal the output is also colourised; ANSI codes were stripped here. The logger writes to stdout.)

Child loggers and JSON output ran locally

```
const log = logger.child({ section: 'billing' });
log.debug('hidden at info level');
log.info('Invoice created', { invoiceId: 42, amount: 19.99 });
```

Output

```
$ LOG_PRETTY=false node logs.mjs
{"level":30,"time":1792116378343,"pid":3355647,"hostname":"vmi3624072","section":"billing","invoiceId":42,"amount":19.99,"msg"
:"Invoice created"}

$ LOG_LEVEL=debug LOG_PRETTY=false node logs.mjs
{"level":20,"time":1792116378627,"pid":3355663,"hostname":"vmi3624072","section":"billing","msg":"hidden at info level"}
{"level":30,"time":1792116378629,"pid":3355663,"hostname":"vmi3624072","section":"billing","invoiceId":42,"amount":19.99,"msg"
:"Invoice created"}
```

Telemetry on a record ran locally

```
logger.setTelemetry('on');
log.warn('with telemetry', { telemetry: true });
```

With telemetry on and `{ telemetry: true }` in the extras, the record gains a `telemetry` object (`runtime, os, arch, pid, ppid, cwd, uptime, osVersion, loadAvg, memory, ...`) from `getSysInfo()`, cached for 5 seconds (the full output, not shown here, is long).

> **The telemetry object includes an `env` map of your whole process environment.** Values are redacted only for variable names that look secret (for example tokens, keys, passwords). Other values, such as `PATH`, `HOME` and URLs like `CORELIB_TURSO_URL`, appear verbatim. Leave telemetry off, or do not ship these records to a third party, unless that is acceptable (observed on 0.4.0).

7. Configuration with ConfigManager

`ConfigManager` is a singleton `EventEmitter`. `get(path)` (instance or static form) is synchronous, uses dot notation and returns `undefined` for a missing key. Bundled defaults are seeded at construction, so reads are safe before `initialize()`, but gate long-lived components on `whenReady()`.

Priority, highest first: **CLI flags** (`--a-b-c value` or `--a-b-c=value`), **environment** (`CORELIB_A_B_C`), **config file** (`-C file` / `--config file`), **bundled defaults**. Values are parsed: `true` / `false`, numbers, and JSON arrays/objects.

Initialise, read, update, subscribe ran locally

```
import { ConfigManager } from '@ckirg/corelib';
const config = ConfigManager.getInstance();
await config.initialize();     // concurrent calls share one run
console.log(config.isInitialized, config.get('retrieve.timeout'), config.get('retrieve.retry.limit'));
console.log(ConfigManager.get('database.port'));      // not a default key

config.on('change:database.port', (p) => console.log('port changed to', p));
config.updateValue('database.port', 6000);
console.log(ConfigManager.get('database.port'));
```

Output

```
true 50000 5
undefined
port changed to 6000
6000
```

`updateValue` also emits a generic `change` event `{ path, value }`; `initialize()` emits `initialized`. Other methods: `getConfig()`, `toJsonString()`, `toBuffer()`, `whenReady()`, `loadExternalConfig(source)`.

Command-line overrides ran locally

```
// c2.mjs prints retrieve, database.port and feature.flags after initialize()
$ node c2.mjs --retrieve-retry-limit 2 --database-port=7000 --feature-flags '["a","b"]'
```

Output

```
{"timeout":50000,"retry":{"limit":2,"backoffLimit":3000}} 7000 [ 'a', 'b' ]
```

Environment overrides ran locally

```
$ CORELIB_RETRIEVE_RETRY_LIMIT=3 CORELIB_RETRIEVE_TIMEOUT=1500 node c2.mjs
```

Output

```
{"timeout":1500,"retry":{"limit":3,"backoffLimit":3000}} undefined undefined
```

Env names map to **lower-case** keys with `_` as the separator, so camelCase keys cannot be set that way. `CORELIB_RETRIEVE_RETRY_BACKOFFLIMIT=100` produced a stray key: `{"limit":5,"backoffLimit":3000,"backofflimit":100}`. Use the CLI form for those: `--retrieve-retry-backoffLimit=100` gave `{"limit":5,"backoffLimit":100}`.

Layered config file (YAML, TOML, JSON, JSON5, JSONC, INI) ran locally

A file is layered as `commonAll`, then your app's section (`common`, then `<linux|windows>.<development|production>`). The app name is the folder name of the nearest workspace root (a directory with `pnpm-workspace.yaml` or a `package.json` with `workspaces`), else the current directory's name; the match is case-insensitive. Here the working directory is called `app2`.

```
# settings.yaml
commonAll:
  retrieve:
    timeout: 20000
app2:
  common:
    greeting: hello
  linux:
    development:
      database:
        port: 5433
    production:
      database:
        port: 5432

$ node c2.mjs -C settings.yaml
$ NODE_ENV=production node c2.mjs --config=settings.yaml
```

Output

```
{"timeout":20000,"retry":{"limit":5,"backoffLimit":3000}} 5433 undefined
{"timeout":20000,"retry":{"limit":5,"backoffLimit":3000}} 5432 undefined
```

A `.enc` file is decrypted first; an `http(s)://` source is fetched with `endPoint`. Remote and encrypted loading was not run here.

Error case: config file not found ran locally

```
$ node c3.mjs -C ./missing.yaml
```

Output

```
Error: ENOENT: no such file or directory, open './missing.yaml'
    at readFileSync (node:fs:483:20)
    ...
```

`initialize()` rejects and can be retried; wrap it in `try/catch` if the file is optional.

Do not cache a sub-section at module scope (`const db = ConfigManager.get('database')`): a later override can be missed. Read `ConfigManager.get('database.port')` where you use it.

8. Databases

`createDatabase(config)` returns a `Database` with `query(sql, params?)`, `transaction(callback)` and `disconnect()`. `dialect` is `'sqlite'` (LibSQL: local `file:` URLs or Turso `libsql://`) or `'postgres'`. Params are a positional array (`?`) or an object (`:name`). Config also accepts `mode` (`'stateless' | 'stateful'`), `timeoutMs`, `logger`, `paramRedactor`, `getTraceId`; SQLite adds `authToken`, `journalMode`, `synchronous`, `maxConnections`.

Create, insert, select ran locally

```
import { createDatabase } from '@ckirg/corelib';
const db = await createDatabase({ dialect: 'sqlite', url: 'file:./app.db', mode: 'stateful' });

await db.query('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, active INTEGER)');
console.log(JSON.stringify(await db.query('INSERT INTO users (name, active) VALUES (?, ?)', ['ada', 1])));
await db.query('INSERT INTO users (name, active) VALUES (:n, :a)', { n: 'grace', a: 0 });   // named params
console.log(JSON.stringify(await db.query('SELECT * FROM users WHERE active = ?', [1])));
```

Output

```
{"status":"success","value":{"rows":[],"affectedRows":1,"lastInsertId":"1"}}
{"status":"success","value":{"rows":[{"id":1,"name":"ada","active":1}],"affectedRows":0}}
```

Transactions: commit and roll back ran locally

Return `{ status: 'success', value }` from the callback to commit, or an `error` result to roll back.

```
const tx = await db.transaction(async () => {
  await db.query('INSERT INTO users (name, active) VALUES (?, 1)', ['linus']);
  return { status: 'success', value: 'committed' };
});
const tx2 = await db.transaction(async () => {
  await db.query('INSERT INTO users (name, active) VALUES (?, 1)', ['rolled-back']);
  return { status: 'error', reason: { message: 'abort' } };
});
console.log(JSON.stringify(tx), JSON.stringify(tx2));
console.log(JSON.stringify((await db.query('SELECT count(*) AS n FROM users')).value.rows));
```

Output

```
{"status":"success","value":"committed"} {"status":"error","reason":{"message":"abort"}}
[{"n":3}]
```

Three rows (ada, grace, linus): the rolled-back insert is absent.

SQL error as a value; PostgreSQL ran locally

```
const bad = await db.query('SELECT * FROM nope');
console.log(bad.status, bad.reason.message);
await db.disconnect();

// PostgreSQL has the same API (illustrative, needs a server):
// const pg = await createDatabase({ dialect: 'postgres', url: 'postgres://user:pass@localhost:5432/dbname', mode:
'stateful' });
```

Output

```
error SQLITE_ERROR: no such table: nope
```

The PostgreSQL line is illustrative and was not run. Other public exports: `SqliteDb, PostgresDb, SqliteDriver, PostgresDriver, wrapSuccess, wrapError, handleDbError, redactParams, defaultRedactor, runInTransaction, getActiveTransaction, getActiveTxId, transactionStorage`.

9. Runtime utilities, cron and the native core

Function	Returns
`detectRuntime()`	`"node" | "bun" | "deno" | "cloudflare" | "aws-lambda" | "gcp-cloudrun" | ...` (override with `RUNTIME=`)
`getEnv(key)`, `getAllEnv()`	Env access that works on Node and Deno.
`getMode()`, `getPlatform()`	`"development"|"production"` from `NODE_ENV`; `"linux"|"windows"`.
`getCwd()`, `getDirname()`, `getTempDir()`	Paths.
`readTextFileSync(f)`, `existsSync(f)`	File helpers across runtimes.
`sleep(ms)`	Promise delay.
`getSysInfo()`	Process and system info; secret-looking env values redacted (see the telemetry warning in section 6).
`nextCid()`	Process-wide counter used as a correlation id.

Environment helpers and system info ran locally

```
import { detectRuntime, getMode, getPlatform, getEnv, getSysInfo } from '@ckirg/corelib';
console.log(detectRuntime(), getMode(), getPlatform(), getEnv('HOME'));
console.log(Object.keys(getSysInfo()));
```

Output

```
node development linux /home/user
[
  'runtime', 'os',
  'arch',    'pid',
  'ppid',    'cwd',
  'uptime',  'osVersion',
  'loadAvg', 'memory',
  'env'
]
```

Cron include/exclude

`includeExcludeCron(includeExprs, excludeExprs, handler, timezone?, onError?)` returns a croner job that ticks every second and runs `handler` when any include expression matches and no exclude does. Expressions may have 5, 6 (with seconds) or 7 fields and are validated up front. A throwing or rejecting handler goes to `onError` (or the logger) and does not stop the job.

Run every second except even seconds ran locally

```
const job = includeExcludeCron(['* * * * * *'], ['*/2 * * * * *'], () => {
  console.log('tick, second is odd:', new Date().getSeconds() % 2 === 1);
}, 'UTC');
await sleep(4200);
job.stop();
```

Output

```
tick, second is odd: true
tick, second is odd: true
```

Invalid expression fails fast ran locally

```
try { includeExcludeCron(['bad cron'], [], () => {}); } catch (e) { console.log(e.message); }
```

Output

```
CronPattern: invalid configuration format ('bad cron'), exactly five, six, or seven space separated parts are required.
```

Native core (FFI)

Detect and call the Rust addon ran locally

```
import { Core } from '@ckirg/corelib';
if (Core.isFfiAvailable()) {
  console.log(Core.getVersion(), Core.logAndDouble('Double me', 21));
}
Core.run('maintenance-task', { verbose: true });   // logs only; returns undefined
```

Output

```
0.2.0 42
[2026-10-16 05:06:23.579 +0300] INFO: Running on node. FFI: true
    section: "Core"
[2026-10-16 05:06:23.589 +0300] INFO: Task: maintenance-task
    section: "Core"
    verbose: true
```

Without the binary `isFfiAvailable()` is `false` and the other calls are not safe to rely on. (The version string above is the addon's own, not the npm package version; see section 15.)

10. Market data (corelib-markets)

Most calls here reach live services (Nasdaq, CNN, Yahoo, brokers). Examples that hit them are marked illustrative. Where possible, the local examples redirect the package to a stub server by overriding config keys with `ConfigManager.updateValue`.

10.1 Market status and sleep calculation

`MarketStatus.getStatus()` reads `markets.nasdaq.statusEndpoint` and returns `NasdaqResult<NasdaqMarketInfo>`. `getSleepDuration(info)` returns milliseconds until the next pre-market or open (0 when open; 300 000 if the next trade date cannot be parsed).

Sleep until the next event ran locally

The system clock is faked to 03:00 New York time on 9 Mar 2026 (07:00 UTC, EDT); the info object is trimmed to the fields that matter.

```
const info = { mrktStatus: 'Closed', nextTradeDate: 'Mar 10, 2026', pmOpenRaw: '2026-03-09T04:00:00',
               openRaw: '2026-03-09T09:30:00', closeRaw: '2026-03-09T16:00:00', ahCloseRaw: '2026-03-09T20:00:00',
               /* ...remaining NasdaqMarketInfo fields... */ };
Date.now = () => new Date('2026-03-09T07:00:00Z').getTime();
console.log(MarketStatus.getSleepDuration(info));                          // 1 hour
console.log(MarketStatus.getSleepDuration({ ...info, mrktStatus: 'Open' })); // already open
```

Output

```
3600000
0
```

`Fetch status and monitor phase changes against a stub` ran locally

```
cfg.updateValue('markets.nasdaq.statusEndpoint', `http://127.0.0.1:${port}/api/market-info`);
const st = await MarketStatus.getStatus();
console.log(st.status, st.value.mrktStatus);

const monitor = new MarketMonitor({ liveIntervalSec: 15, closedIntervalSec: 1800 });
monitor.on('status-change', (phase, data, heuristic) => {
  console.log('phase', phase, 'heuristic', !!heuristic, data.nextTradeDate);
  monitor.stop();
});
monitor.on('stopped', () => console.log('stopped; running =', monitor.isRunningState));
monitor.start();
```

Output

```
success Closed
phase closed heuristic false Mar 10, 2026
stopped; running = false
```

`MarketMonitor` options: `liveIntervalSec, closedIntervalSec, warnIntervalSec, proxies` (proxies are called at `<proxy>/api/v1/markets/nasdaq/status`, i.e. a ts-cloud service). Phases: `"open" | "pre-market" | "after-hours" | "closed"`. It emits `status-change` after the first successful poll and on each change, falls back to a time-based heuristic during fetch failures (`heuristic = true`), and emits `stopped`. Properties: `isRunningState`, `currentPhase`.

Against the real Nasdaq API illustrative

```
const result = await MarketStatus.getStatus();
if (result.status === 'success') {
  console.log(`Nasdaq Status: ${result.value.mrktStatus}`);
  console.log(`Sleeping ${MarketStatus.getSleepDuration(result.value)} ms`);
}
```

10.2 MarketSymbols (symbol directory)

`new MarketSymbols(db?, ingestors?)`. `get(symbol)` returns a `MarketSymbolRow` (`symbol, name, type 'rt'|'eod', class, ts, active`) or `null`; `refresh()` forces a reload; `close()` releases the store. The backend is the constructor argument (a path means SQLite, `mongodb://` or `mongodb+srv://` means MongoDB, `{ dbUrl, dbToken }` means Turso) or `markets.nasdaq.symbols.store` (`sqlite` default, `turso`, `mongodb`). On Node it searches store, then Nasdaq API, then ingestors; on edge runtimes API first.

Load the directory from stub files and look up symbols ran locally

The refresh normally downloads two files from Nasdaq's FTP/HTTPS directory. Here the two URL keys point at a local server serving files in the same format, and `minRows` (default 1000; a guard against truncated downloads) is lowered for the demo.

```
cfg.updateValue('markets.nasdaq.symbols.nasdaqListedUrl', `${base}/nasdaqlisted.txt`);
cfg.updateValue('markets.nasdaq.symbols.otherListedUrl',  `${base}/otherlisted.txt`);
cfg.updateValue('markets.nasdaq.symbols.minRows', 2);
const symbols = new MarketSymbols('./symbols-demo.sqlite');
for (const s of ['AAPL', 'IBM', 'QQQ']) console.log(JSON.stringify(await symbols.get(s)));
await symbols.close();
```

Output

```
{"symbol":"AAPL","type":"rt","class":"stocks","name":"Apple Inc. - Common Stock","ts":1792108279630,"active":true}
{"symbol":"IBM","type":"rt","class":"stocks","name":"International Business Machines","ts":1792108279630,"active":true}
{"symbol":"QQQ","type":"rt","class":"etf","name":"Invesco QQQ Trust","ts":1792108279630,"active":true}
```

Other backends illustrative

```
new MarketSymbols({ dbUrl: 'libsql://your-db.turso.io', dbToken: 'YOUR_TURSO_TOKEN' });   // Turso
new MarketSymbols('mongodb+srv://user:YOUR_PASSWORD@cluster.example.mongodb.net/corelib');  // MongoDB
new MarketSymbols(undefined, ['https://script.google.com/macros/s/YOUR_ID/exec']);       // + custom ingestor

// or select by configuration:
//   export CORELIB_MARKETS_NASDAQ_SYMBOLS_STORE=mongodb
//   export MONGODB_URI="mongodb+srv://user:YOUR_PASSWORD@cluster.example.mongodb.net/corelib"
```

- MongoDB runs on Node, Lambda and Cloud Run, not on Cloudflare Workers (use Turso there). Switching backend starts empty; there is no migration.
- A circuit breaker (`markets.nasdaq.symbols.breaker`, default 30 000 ms) applies to MongoDB only.
- A misconfigured store is only **logged** by `get()` (`SearchDb query failed`), which then falls back to the Nasdaq API; `refresh()` throws. If lookups unexpectedly hit the API, read the logs.

10.3 Nasdaq API wrapper, headers, Fear and Greed, Top 100, historical data

Header spoofing and input validation, offline ran locally

```
import { CnnFearAndGreed, getNasdaqHeaders } from '@ckirg/corelib-markets';
console.log(JSON.stringify(await CnnFearAndGreed.getFearAndGreed('15-03-2026')));   // rejected before any request
console.log(Object.keys(getNasdaqHeaders('https://api.nasdaq.com/api/quote/AAPL/info')).join(', '));
```

Output

```
{"status":"error","reason":{"message":"Invalid date: expected YYYY-MM-DD or \"Historical\""}}
accept, accept-language, origin, referer, sec-ch-ua, sec-ch-ua-mobile, sec-ch-ua-platform, user-agent
```

Add or override headers with the config key `markets.nasdaq.headers` (an object); the Chrome version used in the user agent is `markets.chromeVersion`.

Live data calls illustrative

```
import { ApiNasdaqUnlimited, CnnFearAndGreed, CnnFearAndGreedFilter, Historical, getSymbolsTop100 }
  from '@ckirg/corelib-markets';

// Nasdaq: status === 'success' checks the API's own rCode too
const quote = await ApiNasdaqUnlimited.endPoint('https://api.nasdaq.com/api/quote/AAPL/info?assetclass=stocks');

// CNN Fear and Greed: current, full history, or one metric on one date (YYYY-MM-DD)
const current = await CnnFearAndGreed.getFearAndGreed();
const history = await CnnFearAndGreed.getFearAndGreed('Historical', 'full');
const vix     = await CnnFearAndGreed.getFearAndGreed('2026-03-15', CnnFearAndGreedFilter.MarketVolatilityVix);
if (current.status === 'success') console.log(current.value.score, current.value.rating);

// Yahoo Finance v3 daily bars; interval is '1d' | '1wk' | '1mo'
const bars = await Historical.getData('AAPL', { period1: '2023-01-01', interval: '1d' });
if (bars.status === 'success') console.log(bars.value.at(-1));   // { symbol, date, open, high, low, close, volume,
adjClose }

// Nasdaq 100 constituents, cached in-process and shared between concurrent callers
const top = await getSymbolsTop100();
```

These were not run (public network). Other exports: `ApiNasdaqQuotes`, `NasdaqPolling`, the `Markets` namespace object, `SymbolStore` and `SymbolStoreUnavailableError` types.

10.4 Real-time streaming

`AlpacaStreaming`, `FinnhubStreaming` and `YahooStreaming` are `EventEmitter`s backed by the Rust WebSocket engine. Construct (synchronous, wrap in `try/catch`), `await init(config)`, `await start()`, `subscribe([...])`. Events: `pricing`, `log`, `connected`, `disconnected`, `reconnecting`, `error`; Alpaca and Yahoo also emit `market` and `silence-reconnect`.

Alpaca, Finnhub and Yahoo illustrative

```
import { AlpacaStreaming, FinnhubStreaming, YahooStreaming } from '@ckirg/corelib-markets';

// Alpaca: credentials via init() or APCA_API_KEY_ID / APCA_API_SECRET_KEY
const alpaca = new AlpacaStreaming();
await alpaca.init({ keyId: 'YOUR_KEY', secretKey: 'YOUR_SECRET', silenceSeconds: 60 });
await alpaca.start();
alpaca.subscribe(['AAPL', 'TSLA']);                       // or { trades?, quotes?, bars? }
alpaca.on('pricing', (d) => console.log(`${d.symbol}: $${d.price}`));
alpaca.on('error', (e) => console.error('error', e));

// Finnhub: token via init() or FINNHUB_API_KEY; payload fields are camelCase
const finnhub = new FinnhubStreaming();
await finnhub.init({ token: 'YOUR_FINNHUB_API_KEY' });
await finnhub.start();
await finnhub.subscribe(['MSFT']);
finnhub.on('pricing', (d) => console.log(d.messageType, d.symbol, d.price, d.timestamp));

// Yahoo: no credentials; the ticker field is `id`, not `symbol`
const yahoo = new YahooStreaming();
await yahoo.init({ silenceSeconds: 45 });
await yahoo.start();
yahoo.subscribe(['AAPL', 'BTC-USD']);
yahoo.on('pricing', (d) => console.log(`${d.id}: $${d.price}`));

alpaca.stop(); yahoo.stop(); await finnhub.stop();
```

> Connecting needs credentials or a live feed, so `start()` and live data are not run here. Measured locally on 0.4.0: `await new AlpacaStreaming().init({})` with no credentials resolves in a few milliseconds (it opens the persistence database and does not connect). However, in 0.4.0 a Node process that has constructed any streaming client (`AlpacaStreaming`, `FinnhubStreaming` or `YahooStreaming`) does not exit on its own, even after `init()` and `stop()`; end such scripts with `process.exit()`. This is fixed on `main` after 0.4.0: from the next release a constructed or stopped client lets the process exit, and a running feed still keeps it alive. Supply credentials (or the environment variables) before calling `start()`. Use `stream.clean()` to wipe the persistence database (`dbPath`, default under the temp dir).

11. Edge proxy service (corelib-cloud)

`createRouter(logger?, { platform? })` returns a Hono app. Its `app.request(path, init, env)` method lets you exercise it in-process without a server; on a platform you use the entry bundles (`@ckirg/corelib-cloud/aws`, `/cloudflare`, `/cloudrun`). The package also exports the sub-routers (`kyRouter`, `nasdaqRouter`, `marketStatusRouter`, `historicalRouter`, `sqlRouter`), the `Cloud` namespace and the `AppEnv` type.

Route	Auth	Purpose
`GET /health`, `GET /api/v1/health`	none	Liveness (`platform`, `service`, `version`).
`POST /api/v1/ky`	bearer	Resilient proxy; body `{ url, options? }` or `{ endPoints: [...] }`.
`GET /api/v1/ky?url=`	bearer	Transparent single-URL proxy (what `RequestProxied` calls).
`POST|GET /api/v1/markets/nasdaq`	bearer	Nasdaq proxy, `nasdaq.com` hosts only.
`GET /api/v1/markets/nasdaq/status`	bearer	Market status.
`GET /api/v1/markets/nasdaq/groups/top100`	bearer	Nasdaq 100 symbols.
`POST /api/v1/markets/nasdaq/historical`	bearer	Body `{ symbol, options }`.
`POST /api/v1/sql`	bearer	Body `{ sql, params? }` on `CORELIB_TURSO_URL`.

Health and authentication states ran locally

```
import { createRouter } from '@ckirg/corelib-cloud';
const app = createRouter(undefined, { platform: 'local-demo' });
const show = async (label, p, init, env) => {
  const r = await app.request(p, init, env);
  console.log(label, r.status, await r.text());
};
await show('health', '/health');
await show('no token configured', '/api/v1/markets/nasdaq/status');
const env = { CORELIB_API_TOKEN: 'demo-token' };
await show('missing header', '/api/v1/ky?url=https://example.com', {}, env);
await show('wrong token', '/api/v1/ky?url=https://example.com', { headers: { authorization: 'Bearer nope' } }, env);
```

Output

```
health 200 {"status":"ok","timestamp":"2026-10-16T02:06:59.789Z","service":"ts-cloud","platform":"local-demo"}
no token configured 503 {"status":"error","reason":{"message":"Service authentication is not configured"}}
missing header 401 {"status":"error","reason":{"message":"Unauthorized"}}
wrong token 401 {"status":"error","reason":{"message":"Unauthorized"}}
```

The service is fail-closed: with no `CORELIB_API_TOKEN` every `/api/v1` route except health answers 503. Wrong or missing tokens answer 401 with a `WWW-Authenticate` header.

Target restrictions (SSRF guard) ran locally

```
const auth = { authorization: 'Bearer demo-token' };
await show('loopback blocked', '/api/v1/ky?url=http://127.0.0.1:8080/', { headers: auth }, env);
await show('metadata blocked', '/api/v1/ky', { method: 'POST', headers: { ...auth, 'content-type': 'application/json' },
  body: JSON.stringify({ url: 'http://169.254.169.254/latest/meta-data' }) }, env);
await show('non-nasdaq host', '/api/v1/markets/nasdaq', { method: 'POST', headers: { ...auth, 'content-type':
'application/json' },
  body: JSON.stringify({ url: 'https://example.com/x' }) }, env);
```

Output

```
loopback blocked 400 {"status":"error","reason":{"message":"Target address is not allowed"}}
metadata blocked 400 {"status":"error","reason":{"message":"Target address is not allowed"}}
non-nasdaq host 400 {"status":"error","reason":{"message":"Target host is not on the allow-list"}}
```

The guard inspects the URL text only; a public hostname that resolves to a private address, and redirects, are not detected. Keep the token secret.

SQL endpoint against a local SQLite file ran locally

The handler reads `CORELIB_TURSO_URL` and `CORELIB_TURSO_TOKEN` from the bindings or `process.env`; a `file:` URL works locally.

```
const env = { CORELIB_API_TOKEN: 'demo-token', CORELIB_TURSO_URL: 'file:./cloud-demo.db', CORELIB_TURSO_TOKEN: '' };
const sql = async (body) => {
  const r = await app.request('/api/v1/sql', { method: 'POST',
    headers: { authorization: 'Bearer demo-token', 'content-type': 'application/json' },
    body: JSON.stringify(body) }, env);
  console.log(r.status, await r.text());
};
await sql({ sql: 'CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT, active INTEGER)' });
await sql({ sql: 'INSERT INTO users (name, active) VALUES (?, ?)', params: ['ada', 1] });
await sql({ sql: 'SELECT * FROM users WHERE active = ?', params: [1] });
await sql({ sql: 'SELECT * FROM users', params: 'oops' });
await sql({});
```

Output

```
200 {"status":"success","value":{"rows":[],"affectedRows":0,"lastInsertId":"0"}}
200 {"status":"success","value":{"rows":[],"affectedRows":1,"lastInsertId":"1"}}
200 {"status":"success","value":{"rows":[{"id":1,"name":"ada","active":1}],"affectedRows":0}}
400 {"status":"error","reason":{"message":"Invalid params: expected an array or an object"}}
400 {"status":"error","reason":{"message":"Missing SQL query"}}
```

Calls to a deployed service illustrative

```
export CORELIB_API_TOKEN=YOUR_TOKEN
curl https://your-service-url/health

# generic proxy, single and bulk
curl -X POST https://your-service-url/api/v1/ky \
  -H "Authorization: Bearer $CORELIB_API_TOKEN" -H "Content-Type: application/json" \
  -d '{ "url": "https://api.github.com/zen" }'
curl -X POST https://your-service-url/api/v1/ky \
  -H "Authorization: Bearer $CORELIB_API_TOKEN" -H "Content-Type: application/json" \
  -d '{ "endPoints": [ { "url": "https://api.a.com/status" }, { "url": "https://api.b.com/data", "options": { "timeout":
2000 } } ] }'

# transparent GET (this is what RequestProxied calls)
curl -H "Authorization: Bearer $CORELIB_API_TOKEN" "https://your-service-url/api/v1/ky?url=https://api.github.com/zen"

# market data
curl -H "Authorization: Bearer $CORELIB_API_TOKEN" https://your-service-url/api/v1/markets/nasdaq/status
curl -H "Authorization: Bearer $CORELIB_API_TOKEN" https://your-service-url/api/v1/markets/nasdaq/groups/top100
curl -X POST https://your-service-url/api/v1/markets/nasdaq/historical \
  -H "Authorization: Bearer $CORELIB_API_TOKEN" -H "Content-Type: application/json" \
  -d '{ "symbol": "AAPL", "options": { "period1": "2023-01-01", "interval": "1d" } }'
```

Build and deploy illustrative

From a clone, `pnpm --filter @ckirg/corelib-cloud build` (script `build` in `ts-cloud`) produces `dist/cloudflare/worker.js`, `dist/aws/handler.mjs` and `dist/cloudrun/server.js`. Deploying is outside the scope of the local examples and was not run:

```
# Cloudflare (from ts-cloud/)
pnpm exec wrangler secret put CORELIB_API_TOKEN
pnpm exec wrangler deploy

# AWS Lambda (from the repo root; needs docker, pnpm, zip)
scripts/package-lambda.sh
aws lambda update-function-code --function-name corelib --zip-file fileb://ts-cloud/dist/function.zip

# Cloud Run
gcloud run deploy ts-cloud --source . --command node --args dist/cloudrun/server.js
```

Environment for deployments: `CORELIB_API_TOKEN` (required), `CORELIB_TURSO URL` and `CORELIB_TURSO_TOKEN` (SQL route only), `PORT` (Cloud Run, default 3000). Keep the Worker's `[alias]` stubs in `wrangler.toml`; removing them breaks the bundle. On Lambda the native module must be built against glibc 2.34, which is why the script builds inside `amazonlinux:2023`.

12. Configuration reference

Config keys (bundled defaults)

Key	Type	Default	Meaning
`retrieve.timeout`	ms	50000	Request timeout; clamped 1000..120000.
`retrieve.retry.limit`	number	5	Retries; clamped 0..10.
`retrieve.retry.backoffLimit`	ms	3000	Max backoff; clamped 0..60000.
`markets.chromeVersion`	string	"146"	Chrome version in spoofed Nasdaq headers.
`markets.nasdaq.headers`	object	unset	Extra/override headers for Nasdaq calls.
`markets.nasdaq.statusEndpoint`	URL	https://api.nasdaq.com/api/market-info	Market status source.
`markets.nasdaq.monitor.liveIntervalSec`	s	10	Poll interval while market is active.
`markets.nasdaq.monitor.closedIntervalSec`	s	3600	Poll interval when closed.
`markets.nasdaq.monitor.warnIntervalSec`	s	60	Failure-warning throttle.
`markets.nasdaq.symbols.nasdaqListedUrl`, `otherListedUrl`	URL	ftp://ftp.nasdaqtrader.com/...	Primary symbol directory sources.
`markets.nasdaq.symbols.nasdaqListedFallbackUrl`, `otherListedFallbackUrl`	URL	https://www.nasdaqtrader.com/dynamic/symdir/...	Fallback sources.
`markets.nasdaq.symbols.initialBackoffMs`, `maxRetryBackoffMs`, `maxFetchRetries`	number	1000, 3600000, 10	Directory download retry policy.
`markets.nasdaq.symbols.minRows`	number	1000 (code default; not in the JSON)	Refuse a directory that parses to fewer rows.
`markets.nasdaq.symbols.store`	string	sqlite	`sqlite` | `turso` | `mongodb`.
`markets.nasdaq.symbols.breaker`	ms	30000	MongoDB circuit-breaker window.
`markets.nasdaq.symbols.turso.urlEnv`, `tokenEnv`	string	CORELIB_TURSO_URL, CORELIB_TURSO_TOKEN	Names of the env vars holding Turso settings.
`markets.nasdaq.symbols.mongodb.uriEnv`, `database`, `collection`	string	MONGODB_URI, corelib, nasdaq_symbols	MongoDB settings.
`markets.nasdaq.quotes.concurrencyLimit`	number	5	Parallelism for quote fetching.

Environment variables

Variable	Used by
`CORELIB_*`	Any config key, lower-cased, `_` becomes `.` (single-word segments only).
`LOG_LEVEL`, `LOG_PRETTY`, `NODE_ENV`, `RUNTIME`, `MODE`	Logger, `getMode()`, runtime detection, postinstall skip (`MODE=development`).
`APCA_API_KEY_ID`, `APCA_API_SECRET_KEY`, `FINNHUB_API_KEY`	Streaming credentials.
`CORELIB_TURSO_URL`, `CORELIB_TURSO_TOKEN`, `MONGODB_URI`	Symbol store, ts-cloud SQL route.
`CORELIB_API_TOKEN`, `PORT`	ts-cloud.

Command-line flags

`-C file` / `--config file` / `--config=file` load a config file (or URL); every other `--a-b-c value` / `--a-b-c=value` / `--a-b-c` (true) sets key `a.b.c`. Keys containing `__proto__`, `constructor` or `prototype` segments are dropped.

13. Recipes

A self-contained HTTP test harness ran locally

This is the stub server used by the section 5 examples. It serves a JSON quote, a flaky endpoint (two 503s then 200), an echo endpoint and a 404.

```
// stub.mjs
import http from 'node:http';
import { endPoint } from '@ckirg/corelib';

let hits = 0;
const srv = http.createServer((req, res) => {
  const u = new URL(req.url, 'http://x');
  res.setHeader('content-type', 'application/json');
  if (u.pathname === '/quote/AAPL') return res.end(JSON.stringify({ symbol: 'AAPL', price: 189.5 }));
  if (u.pathname === '/flaky') {
    if (++hits < 3) { res.statusCode = 503; return res.end('busy'); }
    return res.end('{"ok":true}');
  }
  if (u.pathname === '/echo') {
    let b = ''; req.on('data', (c) => (b += c));
    return req.on('end', () => res.end(JSON.stringify({ method: req.method, body: b,
      ct: req.headers['content-type'], auth: req.headers.authorization })));
  }
  res.statusCode = 404; res.end('{"error":"not found"}');
}).listen(0, async () => {
  const base = `http://127.0.0.1:${srv.address().port}`;
  console.log((await endPoint(`${base}/flaky`, { retry: { limit: 4, backoffLimit: 200 } })).value.body, 'hits', hits);
  srv.close();
});
```

Output

```
{ ok: true } hits 3
```

Production-style start-up: config, logger, then work ran locally

```
// main.mjs  (run as: NODE_ENV=production LOG_LEVEL=info node main.mjs --retrieve-retry-limit=2)
import { ConfigManager, logger } from '@ckirg/corelib';

const config = ConfigManager.getInstance();
await config.initialize();                      // CLI > env > file > defaults
const log = logger.child({ section: 'main' });
log.info('starting', { retryLimit: config.get('retrieve.retry.limit') });
```

Output

```
{"level":30,"time":1792108608111,"pid":3257669,"hostname":"vmi3624072","section":"main","retryLimit":2,"msg":"starting"}
```

In production mode the logger writes one JSON object per line, which is what log shippers expect; `retryLimit` is `2` from the CLI flag.

Proxy all traffic through your own ts-cloud deployment illustrative

```
import { RequestProxied } from '@ckirg/corelib';
const client = new RequestProxied(['https://your-service-url/api/v1/ky']);
// the proxy expects an Authorization header: pass it per call via options
const r = await client.endPoint('https://api.github.com/zen', '', {
  headers: { Authorization: 'Bearer YOUR_TOKEN' },
});
```

Because `RequestProxied` builds `<proxy>/?url=<target>`, the base you pass already includes the route path (`/api/v1/ky`), and the transparent GET route returns the target's body and status directly.

14. Troubleshooting

Symptom	Cause and fix
"module not found" for the `.node` addon, or `Core.isFfiAvailable()` is false	postinstall skipped (`MODE=development`, CI, Bun without `trustedDependencies`). Run `node node_modules/@ckirg/corelib/scripts/postinstall.js`.
`npm install` 404 on `@jsr/...`	Add `@jsr:registry=https://npm.jsr.io` to `.npmrc`.
Warning "ConfigManager.get(...) called before initialize() resolved"	Await `ConfigManager.getInstance().initialize()` first; the warning is one-time and silent in production.
`ENOENT ... open './x.yaml'` from `initialize()`	The `-C` path does not exist; `initialize()` rejects (retryable).
A `CORELIB_..._BACKOFFLIMIT` env var creates a lower-case twin key	Env names cannot express camelCase; use `--retrieve-retry-backoffLimit` or a config file.
`RequestProxied: at least one proxy URL is required` / `invalid proxy URL`	Constructor validation; pass absolute URLs.
`Invalid proxy suffix` result	The suffix must stay on the proxy host (no absolute or `//` URLs).
Proxy silently stops being used	Removed after 3 consecutive proxy-level failures (network error, 502/503/504); create a new `RequestProxied` to reset.
HTTP error result has `body: "[Body already consumed]"`	Known behaviour on error results; use `reason.status` and logs.
ts-cloud answers 503 "Service authentication is not configured"	Set `CORELIB_API_TOKEN` (binding, secret or env var).
ts-cloud 400 "Target address is not allowed" / "not on the allow-list"	The guard refuses loopback, private, link-local and `.local` / `.internal` hosts on `/ky`, and non-`nasdaq.com` hosts on `/markets/nasdaq`.
`MarketSymbols.get` unexpectedly calls the Nasdaq API	Store misconfigured or unreachable; `get()` only logs `SearchDb query failed`. Check `store`, `MONGODB_URI`, `CORELIB_TURSO_*`.
Symbol refresh error "parsed to N rows, expected at least 1000"	The downloaded file was an error page or truncated; the guard protects existing data. Lower `minRows` only for test stubs.
Lambda deploys but Rust features are off	The addon needs glibc newer than 2.34; build the package with `scripts/package-lambda.sh`.

15. Where the docs and the code disagree

These were found while verifying the examples. The guide follows the code's behaviour where it could be observed.

1. **Install instructions.** The `ts-core`, `ts-markets` and `ts-cloud` READMEs say "Not on the public npm registry" (ts-core and ts-markets show a v0.1.17 GitHub tarball install); the root README says they are on the public npm registry (version 0.4.0 in all manifests). `npm view @ckirg/corelib version` returns 0.4.0, so the root README is right and the package READMEs are out of date.
2. **Tool versions.** The root README shows a TypeScript 5.0+ badge while the catalog pins TypeScript `^6.0.3`.
3. **RequestProxied pruning.** The ts-core README says unhealthy proxies are pruned "on first failure"; the code (and its doc comment) removes a proxy after 3 consecutive failures.
4. **Database config type.** The ts-core README example `createDatabase({ dialect: 'sqlite', url })` omits `mode`, which `BaseDbConfig` declares as required (`'stateless' | 'stateful'`). It runs at runtime without it, but TypeScript users get a type error.
5. **RequestUnlimited.** READMEs describe a `RequestUnlimited` class; the package exports the functions `endPoint` and `endPoints` (plus `RequestProxied`), not a class of that name.
6. **Native addon version.** `Core.getVersion()` returned `0.2.0` from the bundled `ts-core/corelib-rust.node` while the package is 0.4.0.
7. **Error bodies.** Failed HTTP results carry `body: "[Body already consumed]"`, so the server's error payload is not retrievable from `reason.body`.
8. **Streaming clients keep the process alive (0.4.0).** `AlpacaStreaming.init({})` with no credentials resolves in milliseconds, but in 0.4.0 a process that merely constructs an `AlpacaStreaming`, `FinnhubStreaming` or `YahooStreaming` never exits naturally, even after `init()` and `stop()`. The cause was strong native callbacks holding Node's event loop open. It is fixed on `main` after 0.4.0 (callbacks are weak; a running feed holds a keep-alive timer until `stop()`), so on 0.4.0 end such scripts with `process.exit()`.

16. Appendix: quick reference

@ckirg/corelib

Export	Summary
`endPoint(url, ky?)`, `endPoints(urls, ky?)`	Resilient fetch; `RequestResult`.
`new RequestProxied(proxies).endPoint(url, suffix?, ky?)` / `.endPoints(urls, suffix?, ky?)`	Rotating proxy pool.
`logger`, `StrictLogger`, `LogMethod`	Structured logging.
`ConfigManager`	`getInstance, initialize, whenReady, isInitialized, get, getConfig, updateValue, loadExternalConfig, toJsonString, toBuffer`; events `initialized, change, change:<path>`.
`createDatabase`, `SqliteDb`, `PostgresDb`, `DatabaseResult`, `wrapSuccess/wrapError`	SQL access.
`detectRuntime, getEnv, getAllEnv, getMode, getPlatform, getCwd, getDirname, getTempDir, readTextFileSync, existsSync, getRequire, sleep, getSysInfo, nextCid, includeExcludeCron`	Runtime helpers.
`Core`, `isFfiAvailable`, `getVersion`, `logAndDouble`, `coreFFI`	Native addon.

@ckirg/corelib-markets

`MarketStatus.getStatus/getSleepDuration`, `MarketMonitor`, `MarketSymbols` (`get, refresh, close`), `ApiNasdaqUnlimited.endPoint`, `ApiNasdaqQuotes`, `getNasdaqHeaders`, `CnnFearAndGreed.getFearAndGreed` with `CnnFearAndGreedFilter`, `Historical.getData`, `NasdaqPolling`, `getSymbolsTop100`, `AlpacaStreaming`, `FinnhubStreaming`, `YahooStreaming`, namespace `Markets`.

@ckirg/corelib-cloud

`createRouter(logger?, { platform? })`, `Cloud`, `kyRouter`, `nasdaqRouter`, `marketStatusRouter`, `historicalRouter`, `sqlRouter`, type `AppEnv`; subpath bundles `/aws`, `/cloudflare`, `/cloudrun`.

Monorepo scripts (from a clone)

Command	Does
`pnpm build-all`	Build all packages.
`pnpm test-all:run`	Run all unit tests once.
`pnpm typecheck-all`, `pnpm lint-all`, `pnpm lint:strict`	Type check and Biome.
`pnpm verify:fast`, `pnpm verify:full`	Pre-commit style and full pre-release checks.
`pnpm test:integration` (`:record`, `:live`)	Integration tests; live variants hit real services.
`pnpm docs-all`	Generate API docs.