Node.js backend, functional and type-safe

You build a Node.js service with Express 5 from the first request to container operations, tested, logged, and hardened.

Mike Bild

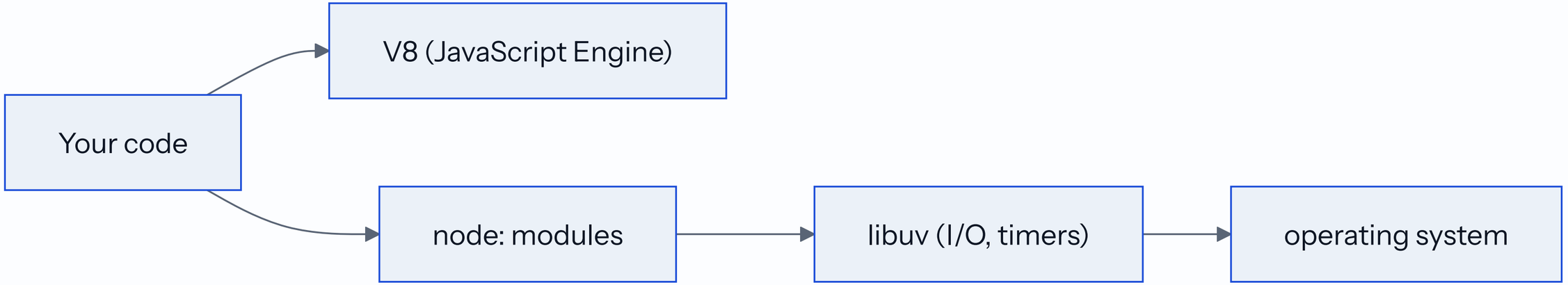
Exercises: github.com/MikeBild/introduction-nodejs

Runtime, npm, event loop, async

Modern Node.js needs little tooling · package.json is a contract · Understand the event loop · Every asynchronous task needs an end

Why Node.js?

- JavaScript on the server, one thread, lots of I/O
- Strong for APIs, CLI tools, and real-time
- Weak under pure CPU load



ESM is the standard

- "type": "module" in package.json
- import and export instead of require
- Top-level await works

```
import { readFile } from 'node:fs/promises'
import { join } from 'node:path'

const configPath = join(import.meta.dirname, 'config.json')
const config = JSON.parse(await readFile(configPath, 'utf8'))
console.log(config)
```

Run TypeScript directly

- Node removes types while loading, stable since 24.12
- tsc --noEmit handles the type check
- Node rejects enums and namespaces

```
// order.ts
export type Order = { id: string; total: number }

// main.ts
import type { Order } from './order.ts'

const formatOrder = (order: Order) => `${order.id}: ${order.total} EUR`
console.log(formatOrder({ id: 'A-1', total: 42 }))
```

Common pitfalls

- Relative imports need the file extension
- require loads ES Modules only without top-level await
- Without strict, TypeScript finds little

What makes up the project baseline

- ESM, node: prefix, top-level await
- TypeScript strict, type check with tsc
- Project baseline without a build step

Dependencies are code

- Every package runs with your privileges
- Versions drift without a lockfile
- Fewer packages, less risk

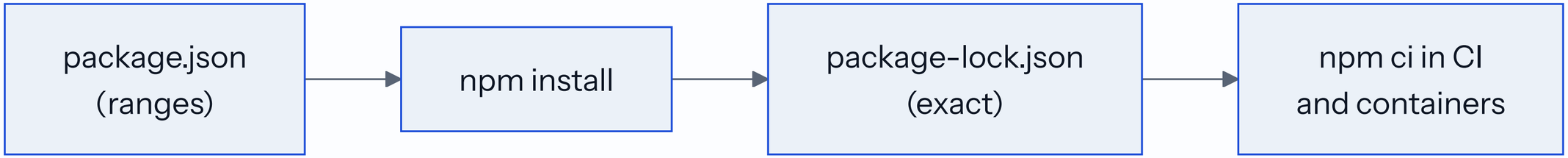
package.json as a contract

- type, engines, exports, scripts
- dependencies vs. devDependencies
- Scripts instead of wiki instructions

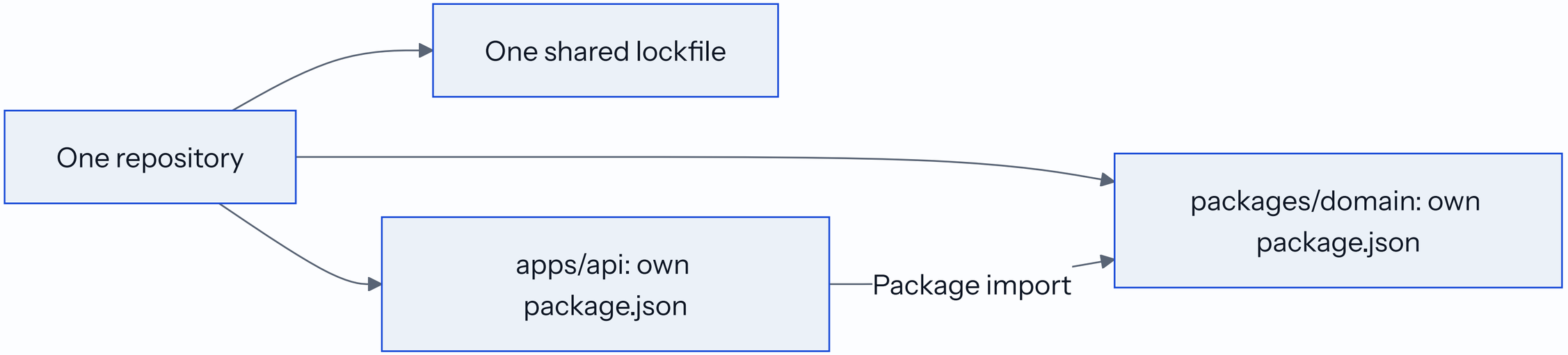
```
{
  "name": "@shop/orders",
  "type": "module",
  "engines": { "node": ">=26" },
  "exports": { ".": "./src/index.ts" },
  "scripts": {
    "dev": "node --watch src/main.ts",
    "test": "vitest run",
    "check": "tsc --noEmit"
  }
}
```

Lockfile and npm ci

- npm install updates the lockfile
- npm ci installs exactly what it contains
- Watch semver ranges, especially for versions below 1.0



Workspaces

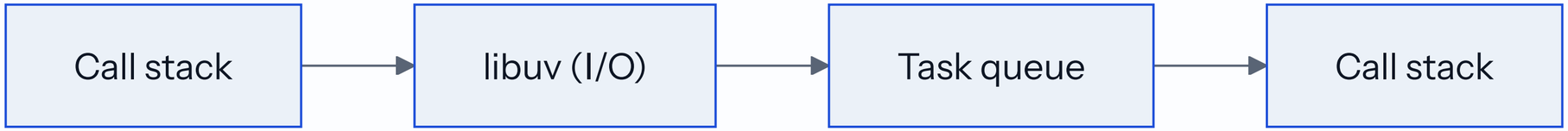

One shared lockfile
One repository
packages/domain: own package.json
apps/api: own package.json
Package import

What makes builds unpredictable or unsafe

- Global installations make builds unpredictable
- postinstall scripts run third-party code
- Typos in package names: typosquatting

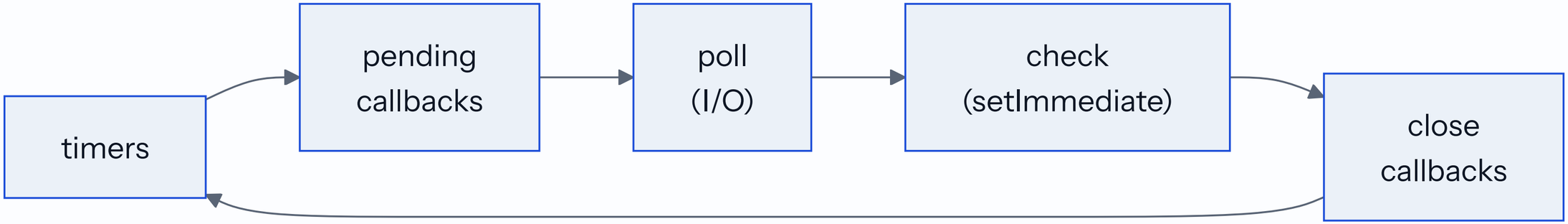
Concurrency ≠ parallelism

- One JavaScript thread runs your code
- I/O runs in the background through libuv
- Callbacks return to the event loop



The phases of the event loop

- timers: due setTimeout and setInterval callbacks
- poll: collect new I/O events
- check: setImmediate callbacks



Order in the I/O callback

```
import { readFile } from 'node:fs'

readFile(import.meta.filename, () => {
  setTimeout(() => console.log('timeout'))
  setImmediate(() => console.log('immediate'))
  Promise.resolve().then(() => console.log('microtask'))
  process.nextTick(() => console.log('nextTick'))
  console.log('sync')
})
// What output follows sync?
```

Blocking the event loop

- Sync I/O, large JSON data, and Crypto block
- Endless microtask chains can starve the loop
- Event loop delay shows whether you are blocking

```
import { monitorEventLoopDelay } from 'node:perf_hooks'

const loopDelay = monitorEventLoopDelay({ resolution: 20 })
loopDelay.enable()
setInterval(() => {
  console.log({ p99_ms: loopDelay.percentile(99) / 1e6 })
  loopDelay.reset()
}, 5_000).unref()
```

Key points

- Waiting is cheap, computing blocks
- nextTick before microtasks before phases
- CPU load belongs in Worker Threads

Waiting without chaos

- Callbacks nest quickly
- Promises are values for future results
- async/await reads like synchronous code

Parallel instead of sequential

```
const loadDashboard = async (customerId: string) => {
  const [customer, orders, invoices] = await Promise.all([
    loadCustomer(customerId),
    loadOrders(customerId),
    loadInvoices(customerId),
  ])
  return { customer, orders, invoices }
}
```

Timeout and cancellation

- AbortController signals: stop
- fetch, streams, and timers understand signal
- AbortSignal.timeout replaces your own timers

```
const loadCustomer = async (customerId: string, signal?: AbortSignal) => {
  const timeoutSignal = AbortSignal.timeout(2_000)
  const response = await fetch(`https://api.example.com/customers/${customerId}`, {
    signal: signal ? AbortSignal.any([signal, timeoutSignal]) : timeoutSignal,
  })
  if (!response.ok) throw new Error(`customer ${customerId}: ${response.status}`)
  return response.json()
}
```

Where asynchronous code swallows errors

- Forgotten await: errors go nowhere
- Unhandled rejections terminate the process
- forEach with async waits for nothing

Three rules for asynchronous work

- Independent work in parallel, dependent work sequentially
- Every I/O operation gets a timeout
- Pass signal through all the way down

Errors, files, streams, CLI

Expected errors are not exceptions · Use the file system and paths safely · Streams and backpressure · Build your own CLI tools

Two kinds of errors

- Expected: not found, invalid, rejected
- Unexpected: bug, network down, disk full
- What matters: who can handle the case usefully

Expected cases as a Result

```
type Result<Value, Failure> = { ok: true; value: Value } | { ok: false; error: Failure }
type OrderFailure = { kind: 'not-found'; orderId: string } | { kind: 'cancelled'; orderId: string }

const findOpenOrder = async (orderId: string): Promise<Result<Order, OrderFailure>> => {
  const order = await loadOrder(orderId)
  if (!order) return { ok: false, error: { kind: 'not-found', orderId } }
  if (order.status === 'cancelled') return { ok: false, error: { kind: 'cancelled', orderId } }
  return { ok: true, value: order }
}
```

Unexpected failures with cause

- throw new Error with cause
- Evaluate error codes like ENOENT
- Log once at the boundary

```
const loadOrder = async (orderId: string) => {
  try {
    return await database.findOrder(orderId)
  } catch (error) {
    throw new Error(`load order ${orderId} failed`, { cause: error })
  }
}
```

Global error handlers

```
const crash = (kind: string) => (error: unknown) => {
  const detail = error instanceof Error ? error.stack : String(error)
  console.error(JSON.stringify({ level: 'fatal', kind, error: detail }))
  process.exit(1)
}

process.on('unhandledRejection', crash('unhandledRejection'))
process.on('uncaughtException', crash('uncaughtException'))
```

How errors get lost

- Empty catch blocks hide bugs
- throw with a string instead of Error loses the stack trace
- Comparing Error messages is fragile

Files are everyday work

- Config, exports, uploads, caches
- Sync I/O blocks the event loop
- External file names are an attack path

node:fs/promises

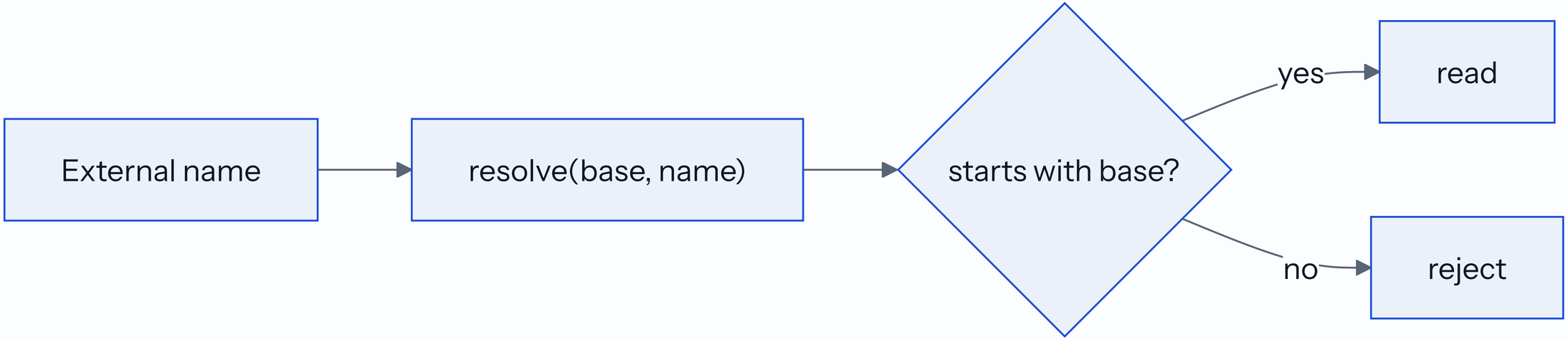
```ts
import { mkdir, rename, writeFile } from 'node:fs/promises'
import { join } from 'node:path'

const dataDirectory = join(import.meta.dirname, 'data')
const reportPath = (id: string) => {
  if (!/^[\w-]+$/.test(id)) throw new Error(`invalid id: ${id}`)
  return join(dataDirectory, `${id}.json`)
}

export const saveReport = async (id: string, report: object) => {
  const target = reportPath(id)
  await mkdir(dataDirectory, { recursive: true })
  await writeFile(`${target}.tmp`, JSON.stringify(report))
  await rename(`${target}.tmp`, target)
}
```

Build paths correctly

- join combines, resolve makes absolute
- The base is import.meta.dirname, not the cwd
- The separator comes from node:path



Prevent Path Traversal

```
import { resolve, sep } from 'node:path'

export const resolveInside = (baseDirectory: string, untrustedName: string) => {
  const base = resolve(baseDirectory)
  const target = resolve(base, untrustedName)
  if (!target.startsWith(base + sep)) throw new Error(`path outside base: ${untrustedName}`)
  return target
}
```

What makes file access error-prone

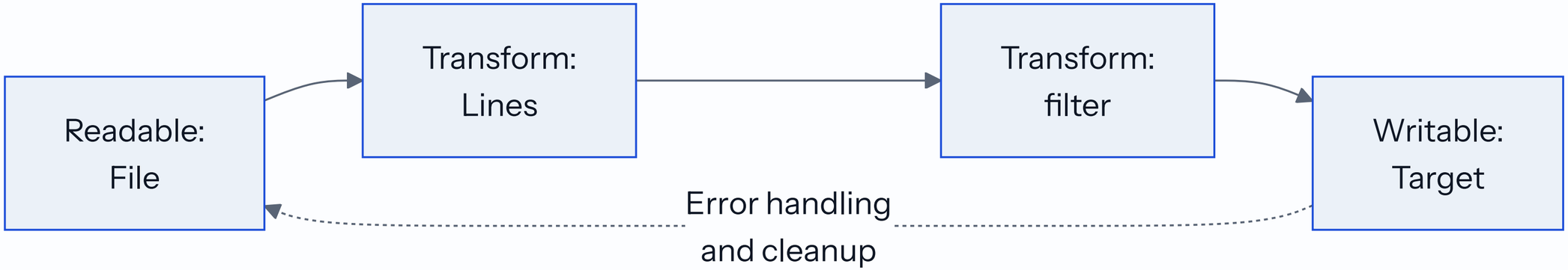
- Check first, then open: race condition
- ENOENT is often not an error, but a case
- Never load large files fully into memory

Doing everything at once goes wrong

- readFile loads the whole file into memory
- With gigabytes: memory full, process dead
- Streams process chunk by chunk

Four types

- Readable and Writable
- Duplex: both, such as a socket
- Transform: changes data on the way, such as gzip



pipeline and Async Generators

```ts
import { createReadStream, createWriteStream } from 'node:fs'
import { pipeline } from 'node:stream/promises'

async function* splitLines(chunks: AsyncIterable<string>) {
  let rest = ''
  for await (const chunk of chunks) {
    const lines = (rest + chunk).split('\n')
    rest = lines.pop() ?? ''
    yield* lines
  }
  if (rest) yield rest
}

async function* onlyPaid(lines: AsyncIterable<string>) {
  for await (const line of lines) if (line.includes(';paid;')) yield `${line}\n`
}

await pipeline(createReadStream('orders.csv', 'utf8'), splitLines, onlyPaid, createWriteStream('paid.csv'))
```

Backpressure

- Fast reader, slow writer
- write() returns false: buffer full
- pipeline handles this automatically

```
import { once } from 'node:events'
import type { Writable } from 'node:stream'

const writeAll = async (target: Writable, rows: Iterable<string>) => {
  for (const row of rows) {
    if (!target.write(row)) await once(target, 'drain')
  }
  target.end()
}
```

What breaks streams and when they pay off

- pipe() does not forward errors
- Ignoring the write() return value fills memory
- Rule of thumb: stream anything large

Tools instead of click paths



Arguments with parseArgs

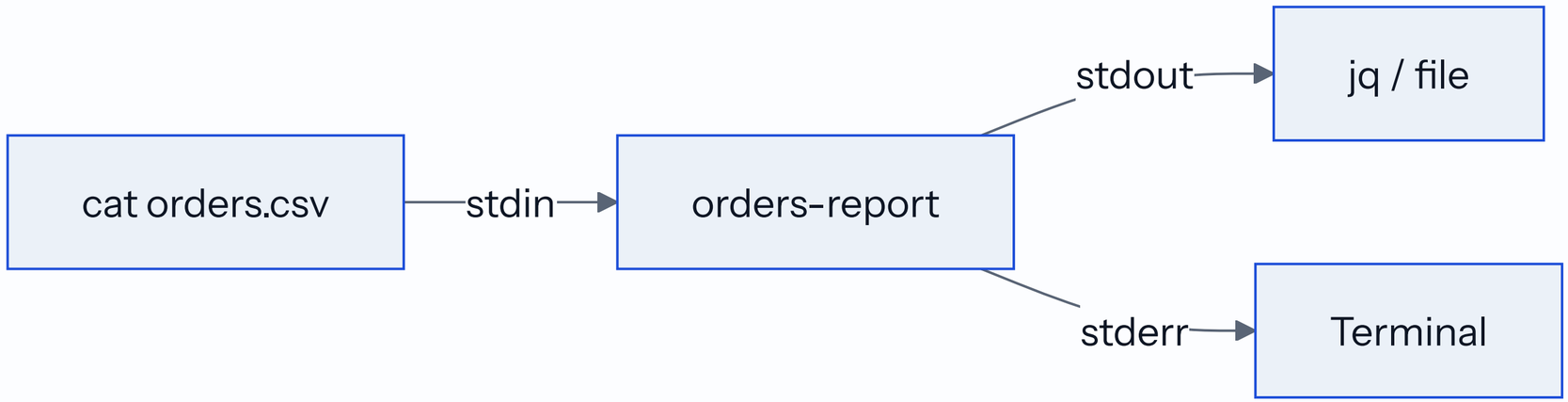
```
#!/usr/bin/env node
import { parseArgs } from 'node:util'

const { values, positionals } = parseArgs({
  options: {
    format: { type: 'string', short: 'f', default: 'json' },
    verbose: { type: 'boolean', short: 'v', default: false },
  },
  allowPositionals: true,
})

const [inputFile] = positionals
if (!inputFile) {
  console.error('usage: orders-report <file> [--format json|csv]')
  process.exitCode = 2
}
```

Input and output

- stdout: data only, machine-readable
- stderr: progress, warnings, errors
- stdin: data from the pipe



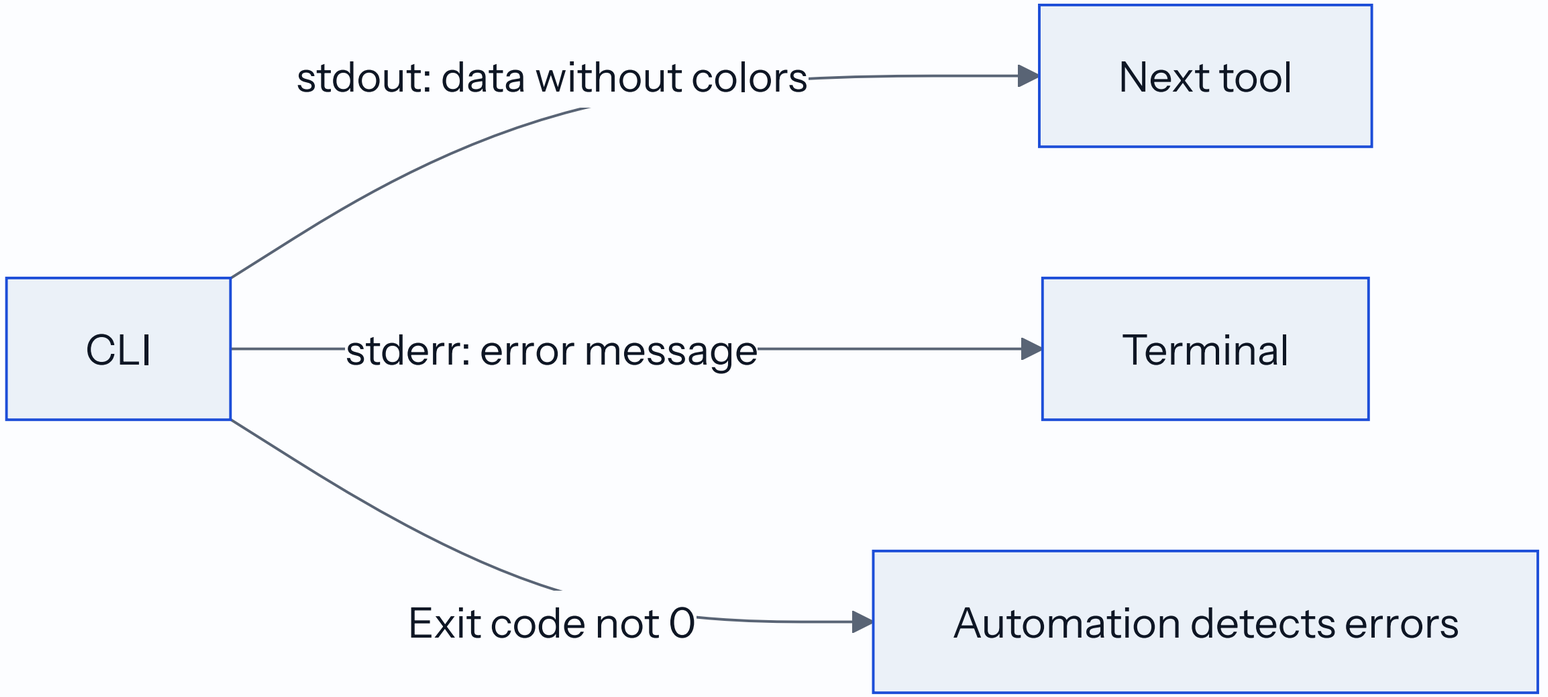
Abort and exit codes

```
import { createInterface } from 'node:readline'

const controller = new AbortController()
process.once('SIGINT', () => controller.abort())

try {
  const lines = createInterface({ input: process.stdin, signal: controller.signal })
  for await (const line of lines) {
    process.stdout.write(`${transformLine(line)}\n`)
  }
} catch (error) {
  console.error(error)
  process.exitCode = 1
}
if (controller.signal.aborted) process.exitCode = 130
```

What gets in the way of CLI tools in pipelines



HTTP API with Express, testing

HTTP API with Express 5 · Input validation with zod · Functional Core, Imperative Shell · Test the core broadly, the shell deliberately

What Express adds on top of node:http

- node:http provides request and response as streams
- You build routing, JSON, and status codes there yourself
- Express adds routes, middleware, and error handling

```
import { createServer } from 'node:http'

createServer((req, res) => {
  if (req.method === 'GET' && req.url === '/health') {
    res.writeHead(200, { 'content-type': 'application/json' })
    res.end(JSON.stringify({ status: 'ok' }))
    return
  }
  res.writeHead(404).end()
}).listen(3000)
```

First routes

```
import express from 'express'

const app = express()
app.use(express.json({ limit: '100kb' }))

app.get('/orders/:orderId', async (req, res) => {
  const order = await findOrder(req.params.orderId)
  if (order) res.json(order)
  else res.status(404).json({ error: 'order not found' })
})

app.post('/orders', async (req, res) => {
  res.status(201).json(await placeOrder(req.body))
})

app.listen(3000)
```

Middleware and routers

- Middleware runs in order before the handler
- next() passes control on, otherwise the middleware responds
- Router groups the routes for one area

```
import { Router, type RequestHandler } from 'express'

const requestId: RequestHandler = (req, res, next) => {
  res.set('x-request-id', req.get('x-request-id') ?? crypto.randomUUID())
  next()
}

const ordersRouter = Router()
  .get('/:orderId', getOrder)
  .post('/', postOrder)

app.use(requestId)
app.use('/orders', ordersRouter)
```

Central error handling

```
import type { ErrorRequestHandler, RequestHandler } from 'express'

const notFound: RequestHandler = (req, res) => {
  res.status(404).json({ error: 'route not found' })
}

const errorHandler: ErrorRequestHandler = (error, req, res, next) => {
  console.error(JSON.stringify({ level: 'error', path: req.path, error: String(error) }))
  res.status(500).json({ error: 'internal error' })
}

app.use(notFound)
app.use(errorHandler)
```

Testing with fetch

- app.listen(0) chooses a free port
- fetch calls the real API without an extra package
- Handlers stay thin, without business logic

```
import { expect, test } from 'vitest'
import { createApp } from './app.ts'
import { listen } from './test-server.ts'

test('unknown order returns 404', async () => {
  await using api = await listen(createApp({ findOrder: async () => undefined }))
  const response = await fetch(`${api.url}/orders/does-not-exist`)
  expect(response.status).toBe(404)
})
```

TypeScript ends at the boundary

- Types exist only at compile time
- JSON from the network is unknown
- An as cast is a claim, not proof

Schema instead of interface

```
import { z } from 'zod'

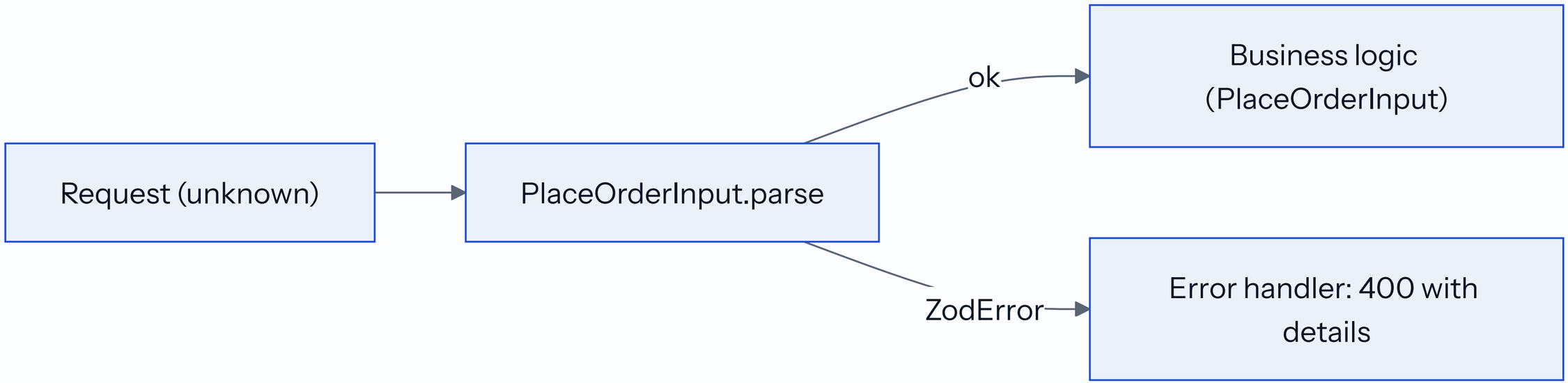
export const OrderItemInput = z.strictObject({
  productId: z.string().min(1),
  quantity: z.number().int().min(1),
})

export const PlaceOrderInput = z.strictObject({
  customerId: z.string().min(1),
  items: z.array(OrderItemInput).min(1),
})

export type PlaceOrderInput = z.infer<typeof PlaceOrderInput>
```

Parse at the boundary

- Parse once at the edge, use only types inside
- parse returns the value or throws a ZodError
- Validation errors as 400 with clear details



zod in Express

```ts
import { z } from 'zod'
import { PlaceOrderInput } from './schemas.ts'

const ListQuery = z.object({
  limit: z.coerce.number().int().min(1).max(100).default(20),
  status: z.enum(['draft', 'placed', 'paid', 'shipped', 'cancelled']).optional(),
})

app.get('/orders', async (req, res) => {
  res.json(await listOrders(ListQuery.parse(req.query)))
})

app.post('/orders', async (req, res) => {
  res.status(201).json(await placeOrder(PlaceOrderInput.parse(req.body)))
})
```

Validation errors as 400

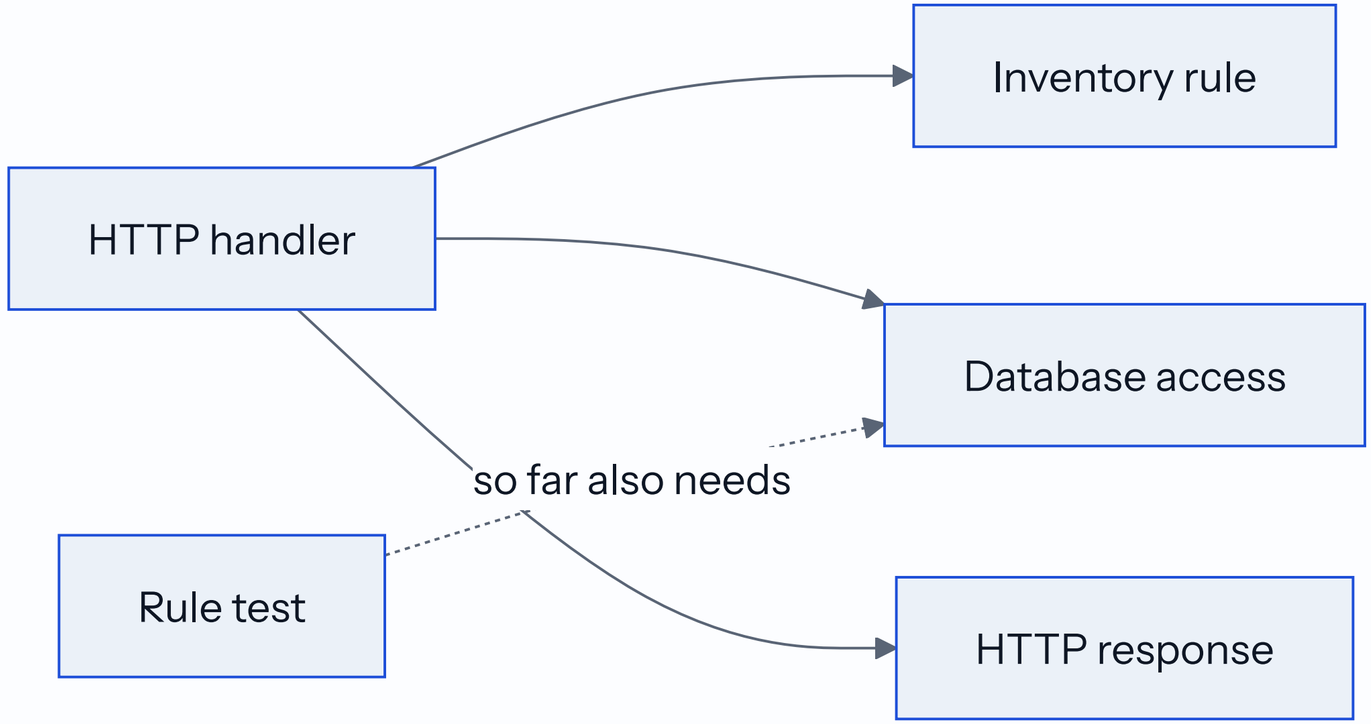
```
import type { ErrorRequestHandler } from 'express'
import { z } from 'zod'

export const errorHandler: ErrorRequestHandler = (error, req, res, next) => {
  if (error instanceof z.ZodError) {
    const issues = error.issues.map(({ path, message }) => ({ path: path.join('.'), message }))
    res.status(400).json({ error: 'invalid input', issues })
    return
  }
  if (error.status >= 400 && error.status < 500) {
    res.status(error.status).json({ error: 'bad request' })
    return
  }
  console.error(error)
  res.status(500).json({ error: 'internal error' })
}
```

Where validation does too much or too little

- Validation in the core duplicates and drifts
- Third-party APIs and the environment are input too
- Validation errors reveal too much

Where is the logic?



Core and shell

- Core only calculates
- Shell reads and writes
- Tests need no mocks



Composition

```ts
import type { OrderItem, Product } from './domain.ts'

const lineTotalCents = (item: OrderItem) => item.quantity * item.priceCents
const sum = (values: number[]) => values.reduce((total, value) => total + value, 0)

export const orderTotalCents = (items: OrderItem[]) => sum(items.map(lineTotalCents))

export const rejected = <Reason extends string>(reason: Reason) => ({ kind: 'rejected', reason }) as const

export const toOrderItem = (product: Product, quantity: number): OrderItem => ({
  productId: product.id,
  quantity,
  priceCents: product.priceCents,
})
```

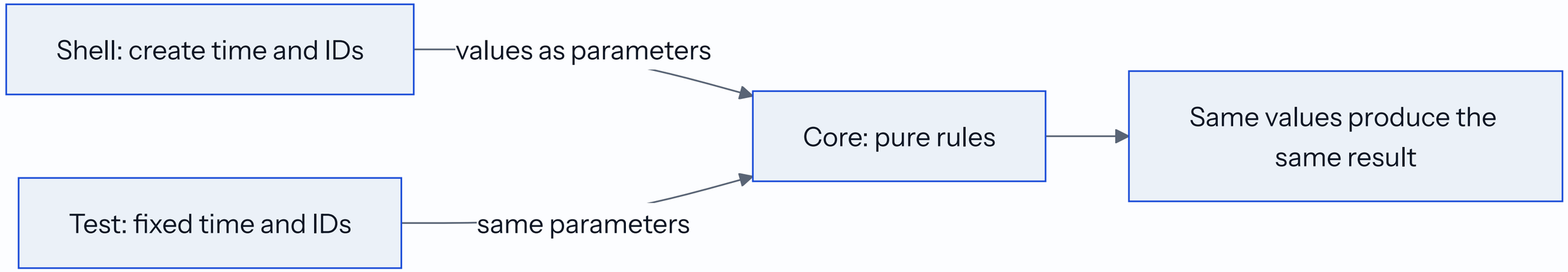
The core decides

```
export const placeOrder = (input: PlaceOrderInput, catalog: Catalog, now: Date) => {
  const items: OrderItem[] = []
  for (const { productId, quantity } of input.items) {
    const product = catalog.get(productId)
    if (!product) return rejected('unknown-product')
    if (product.stock < quantity) return rejected('insufficient-stock')
    items.push(toOrderItem(product, quantity))
  }
  const order = { customerId: input.customerId, items, status: 'placed', createdAt: now.toISOString() } as const
  return { kind: 'placed', order } as const
}
```

The shell stays thin

```
app.post('/orders', async (req, res) => {
  const input = PlaceOrderInput.parse(req.body)
  const catalog = await loadCatalog(input.items.map((item) => item.productId))
  const decision = placeOrder(input, catalog, new Date())
  if (decision.kind === 'rejected') res.status(422).json(decision)
  else res.status(201).json(await saveOrder(decision.order))
})
```

What makes the core impure, and what you gain



Why Vitest?

- One test runner for frontend and backend
- TypeScript and ESM without custom config
- Watch mode, mocks, and fake timers included

Test the core

```
import { expect, test } from 'vitest'
import { placeOrder } from './orders.ts'
import { hammer } from './fixtures.ts'

const now = new Date('2026-01-01T10:00:00Z')
const catalog = new Map([[hammer.id, { ...hammer, stock: 2 }]])

test('placeOrder rejects more than the stock', () => {
  const input = { customerId: 'c-1', items: [{ productId: hammer.id, quantity: 3 }] }
  expect(placeOrder(input, catalog, now)).toEqual({ kind: 'rejected', reason: 'insufficient-stock' })
})
```

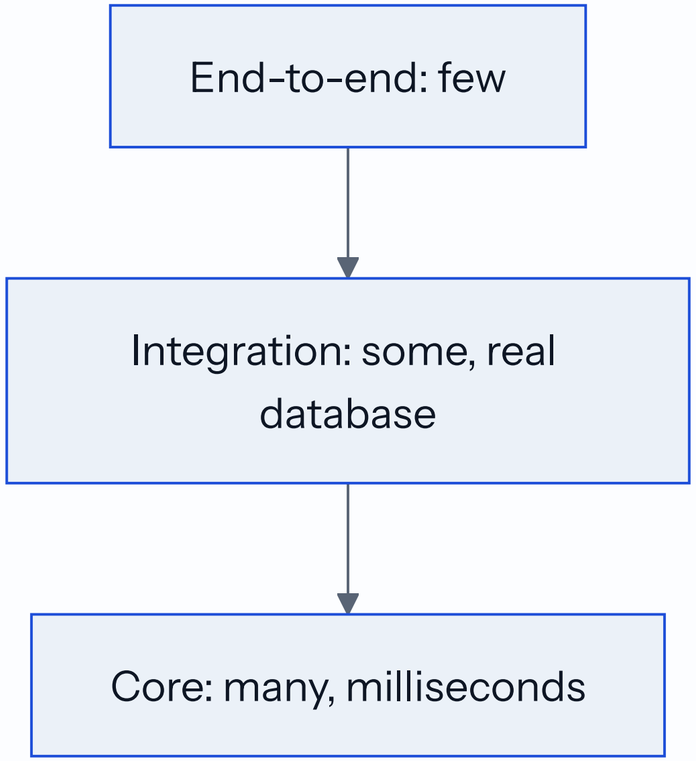
Test the shell

```
import { expect, test, vi } from 'vitest'
import { createApp } from './app.ts'
import { listen } from './test-server.ts'

test('POST /orders saves the order', async () => {
  const saveOrder = vi.fn(async (order: NewOrder) => ({ id: 'o-1', ...order }))
  await using api = await listen(createApp({ loadCatalog: async () => catalog, saveOrder }))
  const response = await fetch(`${api.url}/orders`, {
    method: 'POST',
    headers: { 'content-type': 'application/json' },
    body: JSON.stringify(validInput),
  })
  expect(response.status).toBe(201)
  expect(saveOrder).toHaveBeenCalledOnce()
})
```

How much of each?

- Many fast unit tests for the core
- Some integration tests against a real database
- Few end-to-end tests for the main paths



What makes tests unreliable

- Mocks for everything test only the mocks
- Real time and randomness make tests flaky
- Shared state makes tests depend on order

Config, logging, database

No valid config, no startup · Structured logging with JSON on stdout · Database access with SQL

Config is not code

- URLs, ports, feature flags, secrets
- Different per environment, code stays the same
- One build for all environments

Load once, validate immediately

```
import { z } from 'zod'

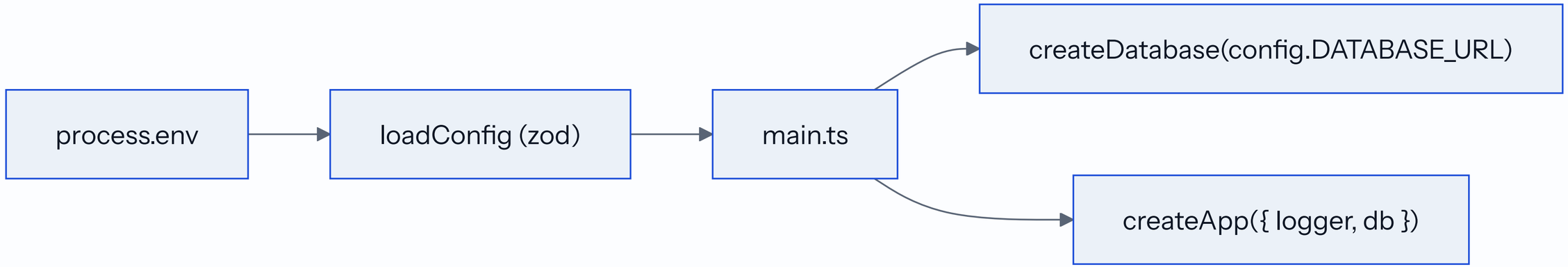
const ConfigSchema = z.object({
  PORT: z.coerce.number().int().min(1).max(65535).default(3000),
  DATABASE_URL: z.url(),
  LOG_LEVEL: z.enum(['debug', 'info', 'warn', 'error']).default('info'),
  JWT_ISSUER: z.url(),
})

export type Config = z.infer<typeof ConfigSchema>

export const loadConfig = (environment: NodeJS.ProcessEnv): Config => {
  const parsed = ConfigSchema.safeParse(environment)
  if (!parsed.success) throw new Error(`invalid config\n${z.prettifyError(parsed.error)}`)
  return Object.freeze(parsed.data)
}
```

Pass config along

- main.ts loads the config
- Functions get what they need
- Tests pass their own values



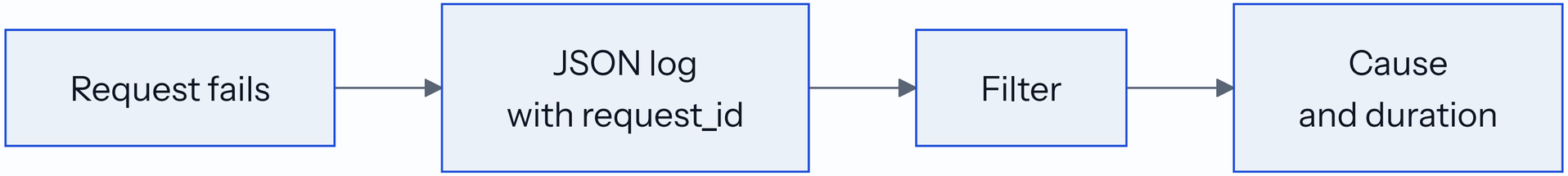
Protect secrets

- Never put them in the repo, not even briefly
- The platform or Secret Manager injects them
- Plan rotation, mask logs

How config hides errors or reveals secrets

- Silent defaults for secrets hide errors
- Config object in the log reveals everything
- NODE_ENV is not an environment selector

Logs that help in operations



Three rules

- Choose log level deliberately
- Context as fields instead of prose
- No secrets, no personal data

```
// schlecht
console.log('Order ' + orderId + ' placed for ' + customerEmail)

// gut
logger.info('order placed', { order_id: orderId, customer_id: customerId })
```

A JSON logger without a package

```ts
type Fields = Record<string, unknown>
const levels = { debug: 10, info: 20, warn: 30, error: 40 } as const
type Level = keyof typeof levels

export const createLogger = (minLevel: Level, bound: Fields = {}): Logger => {
  const at = (level: Level) => (msg: string, fields: Fields = {}) => {
    if (levels[level] < levels[minLevel]) return
    console.log(JSON.stringify({ level, time: new Date(), msg, ...bound, ...fields }))
  }
  return {
    debug: at('debug'), info: at('info'), warn: at('warn'), error: at('error'),
    child: (more) => createLogger(minLevel, { ...bound, ...more }),
  }
}
```

Child logger and request ID

```
import { randomUUID } from 'node:crypto'
import type { RequestHandler } from 'express'

export const requestLogger = (logger: Logger): RequestHandler => (req, res, next) => {
  const requestId = req.get('x-request-id') ?? randomUUID()
  res.locals.logger = logger.child({ request_id: requestId })
  const startedAt = performance.now()
  res.on('finish', () =>
    res.locals.logger.info('request finished', {
      method: req.method, path: req.path, status: res.statusCode,
      duration_ms: Math.round(performance.now() - startedAt),
    }))
  next()
}
```

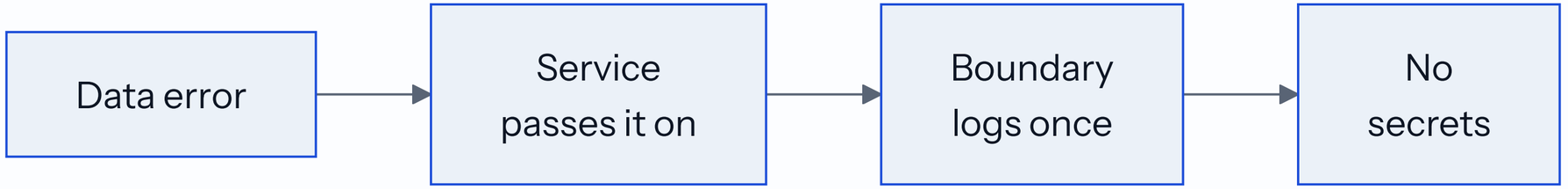
Redaction

- Replace sensitive keys at every depth
- Also redact fields in the child logger
- Safety net, not a free pass for whole objects

```
const sensitiveKeys = new Set(['authorization', 'cookie', 'password', 'email'])

const redact = (value: unknown): unknown =>
  Array.isArray(value)
    ? value.map(redact)
    : value !== null && typeof value === 'object'
      ? Object.fromEntries(
          Object.entries(value).map(([key, nested]) =>
            [key, sensitiveKeys.has(key.toLowerCase()) ? '[redacted]' : redact(nested)]),
        )
      : value
```

When logs become expensive or dangerous



ORM, query builder, or SQL?

- ORM: fast at first, costly in edge cases
- Query builder: type-safe, close to SQL
- SQL directly: full control, little magic

Queries with postgres.js

```ts
type OrderRow = { id: string; customer_id: string; status: OrderStatus; created_at: Date }

export const findOrder = (sql: Sql) => async (orderId: string) => {
  const [order] = await sql<OrderRow[]>`
    select id, customer_id, status, created_at from orders where id = ${orderId}
  `
  return order
}

export const listOrdersByStatus = (sql: Sql) => (status: OrderStatus, limit: number) =>
  sql<OrderRow[]>`
    select id, customer_id, status, created_at from orders
    where status = ${status} order by created_at desc limit ${limit}
  `
```

Transactions

```ts
export const saveOrder = (sql: Sql) => (order: NewOrder) =>
  sql.begin(async (transaction) => {
    const [created] = await transaction<{ id: string }[]>`
      insert into orders (customer_id, status, created_at)
      values (${order.customerId}, ${order.status}, ${order.createdAt})
      returning id
    `
    const itemRows = order.items.map((item) => ({
      order_id: created!.id, product_id: item.productId,
      quantity: item.quantity, price_cents: item.priceCents,
    }))
    await transaction`insert into order_items ${transaction(itemRows)}`
    return created!.id
  })
```

Migrations

- Schema changes as numbered SQL files
- Versioned in the repo, applied before startup
- Make backward-compatible changes: expand first, then clean up



What makes database access slow or wrong

- N+1: one query per record
- Connection pool too large: database buckles
- numeric as a string: never calculate money as float

Operations, performance, architecture

Start cleanly, report honestly, stop cleanly · Measure first, then optimize · Ports and Adapters without classes

The lifecycle of a service



Liveness and readiness

- Liveness: process stuck? Then restart
- Readiness: accept requests? Otherwise no traffic
- Check dependencies only in readiness

```
import { Router } from 'express'

export const createHealthRoutes = (isShuttingDown: () => boolean, pingDatabase: () => Promise<void>) =>
  Router()
    .get('/livez', (req, res) => {
      res.json({ status: 'ok' })
    })
    .get('/readyz', async (req, res) => {
      const ready = !isShuttingDown() && (await pingDatabase().then(() => true, () => false))
      res.status(ready ? 200 : 503).json({ status: ready ? 'ready' : 'not ready' })
    })
```

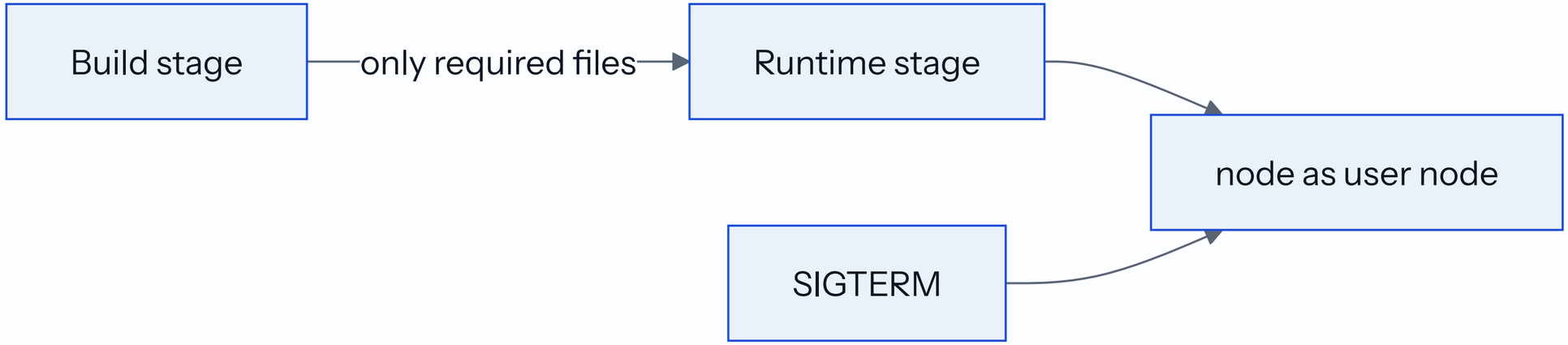
Graceful shutdown

```
const server = app.listen(config.PORT)
let shuttingDown = false

const shutdown = (signal: NodeJS.Signals) => {
  logger.info('shutdown started', { signal })
  shuttingDown = true
  setTimeout(() => process.exit(1), 10_000).unref()
  server.close(async () => {
    await sql.end({ timeout: 5 })
    logger.info('shutdown complete')
    process.exit(0)
  })
}

process.once('SIGTERM', shutdown)
process.once('SIGINT', shutdown)
```

Node in a container



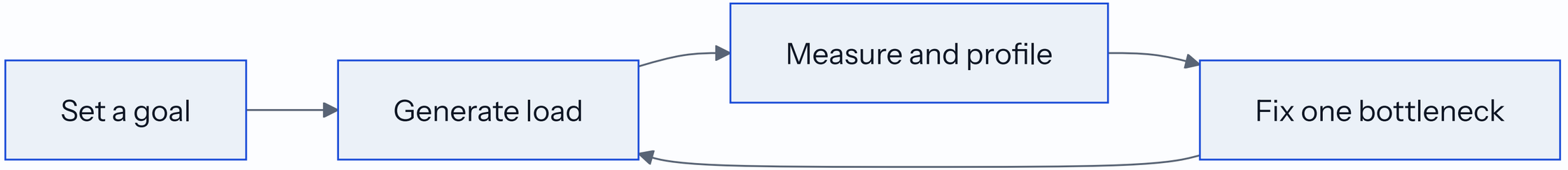
What gets in the way of container operations

- No SIGTERM handler: stops in the middle of the request
- V8 heap larger than the container memory limit
- Health checks that are expensive themselves

Feels slow

- Optimization never ends without a goal
- The average hides slow requests
- Assumptions are usually wrong

The measurement loop



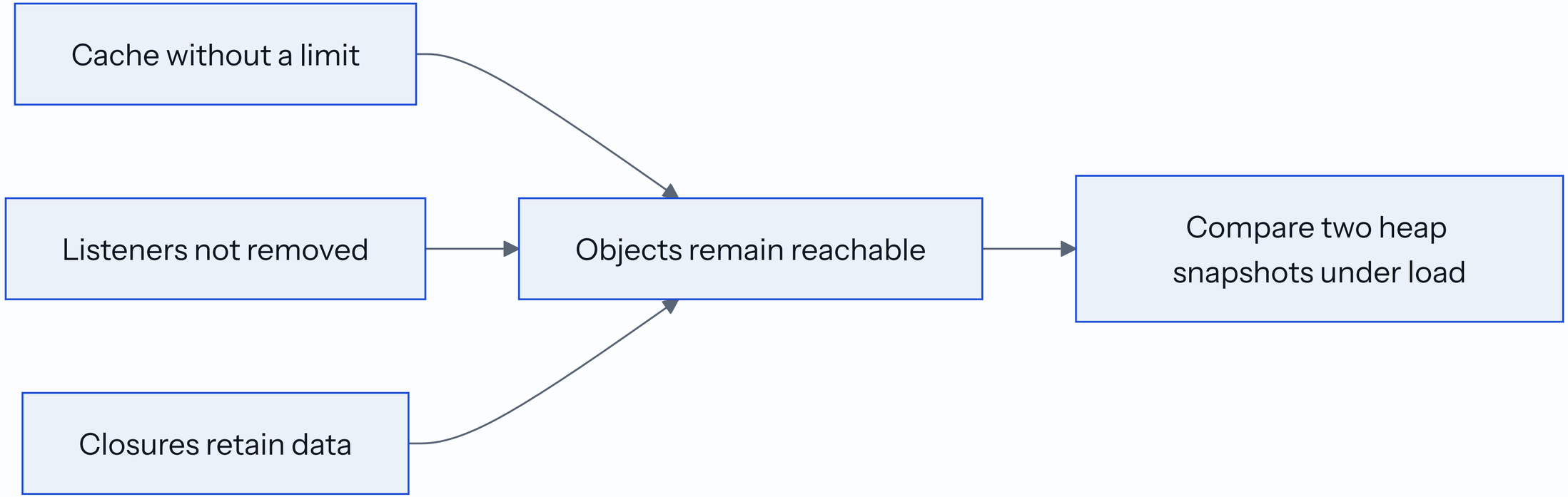
Read a CPU profile

- –cpu-prof writes a profile on exit
- –inspect connects Chrome DevTools live
- Flame graph: wide bars are expensive

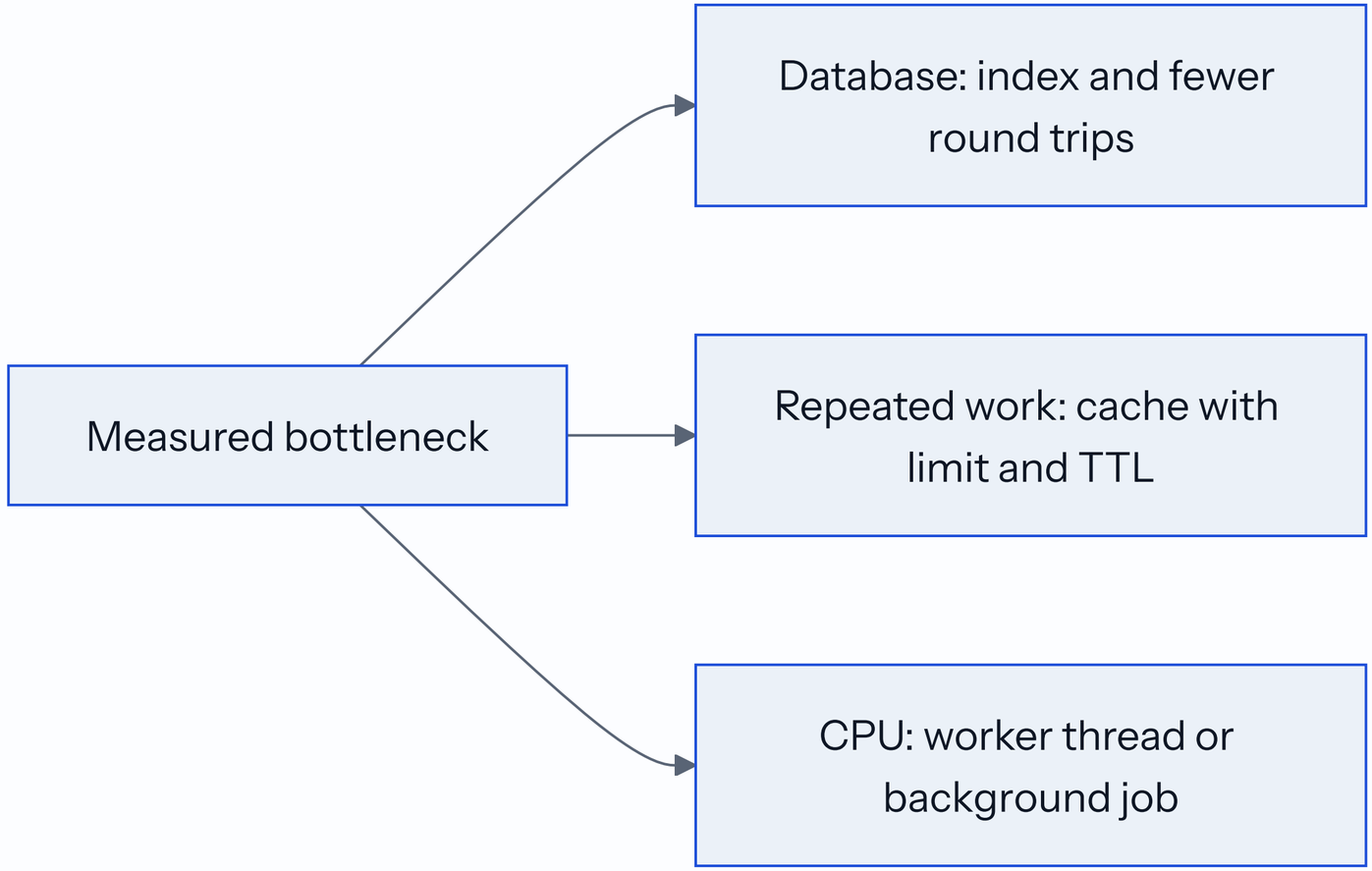
```
import { performance } from 'node:perf_hooks'

performance.mark('report:start')
const report = buildMonthlyReport(orders)
performance.mark('report:end')
const { duration } = performance.measure('report', 'report:start', 'report:end')
logger.info('report built', { duration_ms: Math.round(duration) })
```

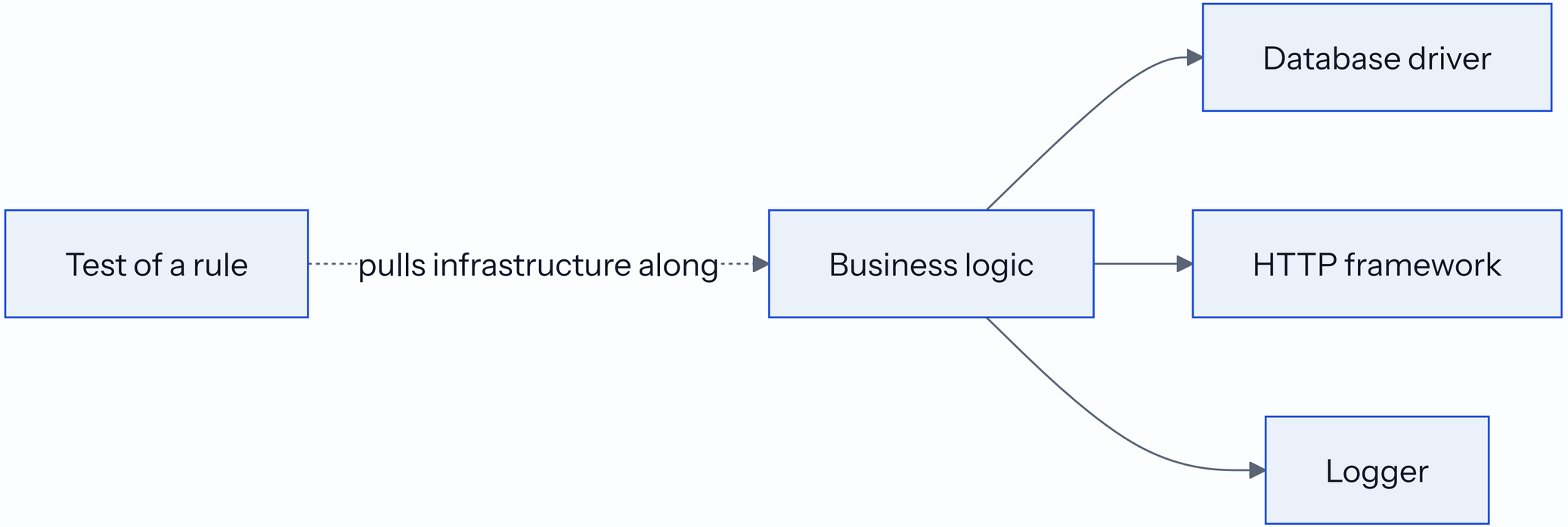
Memory and leaks



Typical levers

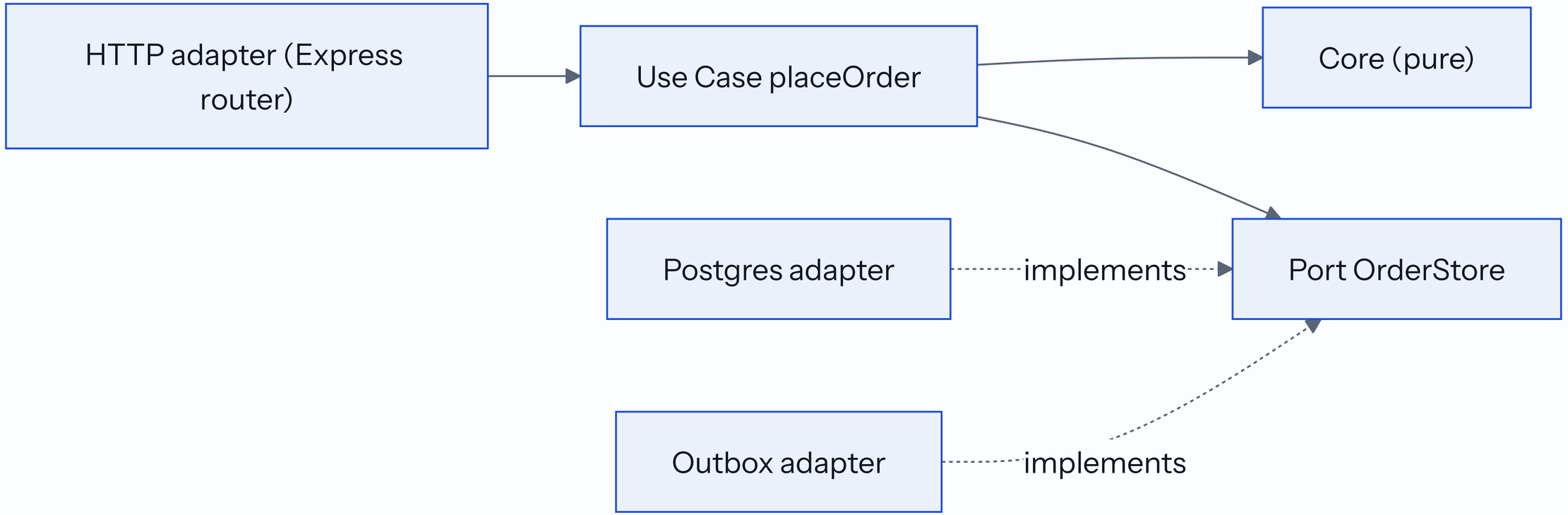


When everything depends on everything



Ports and Adapters

- The core defines what it needs: ports
- Adapters fulfill ports with concrete technology
- Dependencies always point inward



Implemented functionally

```ts
// ports.ts
export type OrderStore = {
  loadCatalog: (productIds: string[]) => Promise<Catalog>
  saveOrder: (order: NewOrder) => Promise<Order>
}

// use case
export const createPlaceOrder = (store: OrderStore, clock: () => Date) =>
  async (input: PlaceOrderInput) => {
    const catalog = await store.loadCatalog(input.items.map((item) => item.productId))
    const decision = placeOrder(input, catalog, clock())
    if (decision.kind === 'rejected') return decision
    return { kind: 'placed', order: await store.saveOrder(decision.order) } as const
  }
```

Wire in main

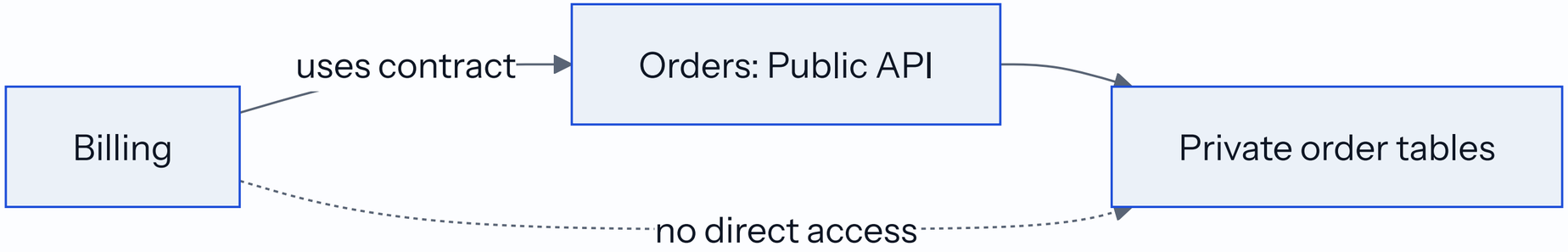
```
// main.ts - Composition Root
const config = loadConfig(process.env)
const logger = createLogger(config.LOG_LEVEL)
const sql = createDatabase(config.DATABASE_URL)

const placeOrder = createPlaceOrder(createPostgresOrderStore(sql), () => new Date())

const app = express()
app.use(express.json({ limit: '100kb' }), requestLogger(logger))
app.use('/orders', createOrderRoutes({ placeOrder }))
app.use(createHealthRoutes(() => shuttingDown, () => pingDatabase(sql)))
app.use(notFound, errorHandler(logger))

const server = app.listen(config.PORT)
```

Modules and trade-offs

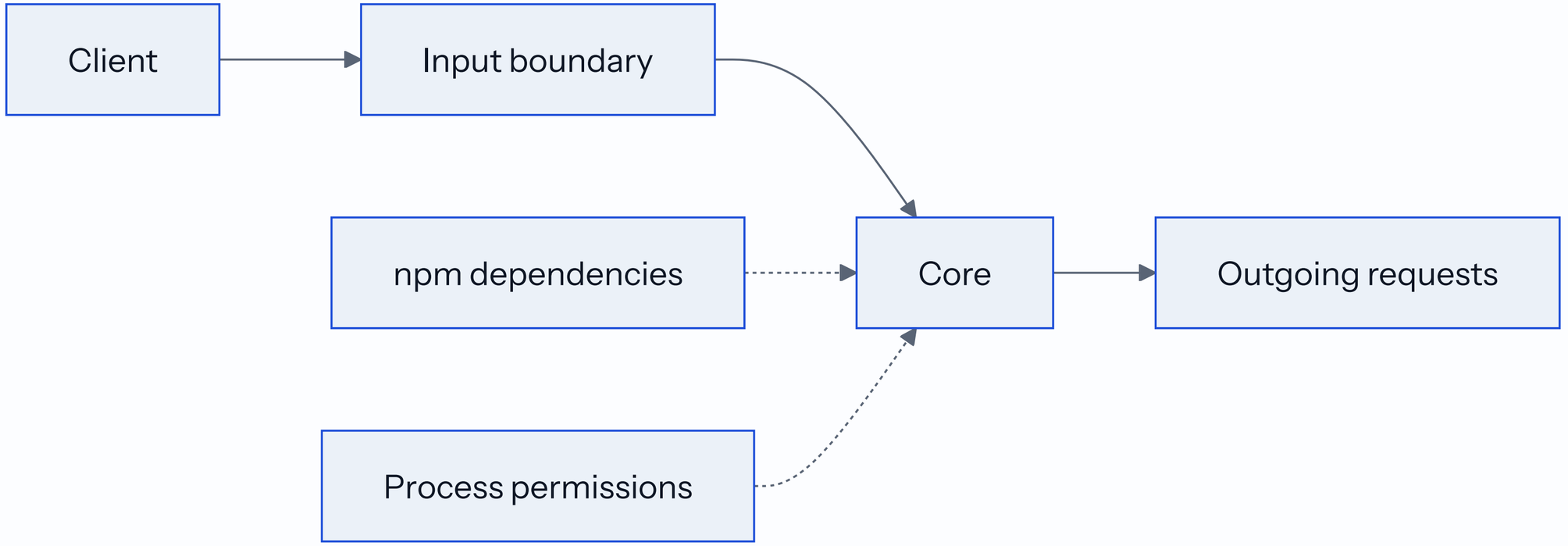


Security

Security for Node.js services

The attack surface

- Input: body, query, headers, files
- Dependencies: hundreds of third-party packages
- Operations: permissions, secrets, resources



Input and injection

- Schema validation at every boundary
- Query parameters instead of string concatenation
- Prototype pollution: never merge objects blindly

```
// dangerous: __proto__ from the body ends up in the object prototype
const unsafeSettings = deepMerge(defaults, req.body)

// safe: only known fields, validated types
const SettingsInput = z.strictObject({ theme: z.enum(['light', 'dark']), pageSize: z.number().int().max(100) })
const settings = { ...defaults, ...SettingsInput.parse(req.body) }
```

Resources and outgoing requests

- Body limit, timeouts, rate limits
- Regex with backtracking: ReDoS
- Check outgoing URLs against an allowlist

Limits and headers in Express

```
import express, { type RequestHandler } from 'express'

const securityHeaders: RequestHandler = (req, res, next) => {
  res.set({
    'content-security-policy': "default-src 'none'; frame-ancestors 'none'",
    'x-content-type-options': 'nosniff',
    'referrer-policy': 'no-referrer',
  })
  next()
}

app.disable('x-powered-by')
app.use(securityHeaders, express.json({ limit: '100kb' }))

const server = app.listen(config.PORT)
server.requestTimeout = 10_000
server.headersTimeout = 5_000
```

Supply chain

- Lockfile and npm ci, always
- npm audit in CI, automate updates
- Check new packages: maintenance, size, scripts

Least privilege

- Containers without root, read-only file system
- Only the secrets the service needs
- Permission Model: --permission with --allow-fs-read and --allow-net

Questions?

Mike Bild · mike.bild@gmail.com · mikebild.dev

Workshops für Teams, die Architekturentscheidungen selbst treffen