Microservices with Node.js, Express 5, and Docker Compose

You split the workshop shop into services along domain boundaries and connect them with contracts, events, outbox, and sagas.

Mike Bild

Exercises: github.com/MikeBild/introduction-microservices

Why, when, and where to cut

Microservices solve team problems, not code problems · Cut at language boundaries, not tables · Modules with clear boundaries before services

What is a microservice?

- A process with one domain task and its own data
- Deployable independently without stopping the others
- Reachable only through its API or its events

What you gain, what you pay



Conway’s Law applies in both directions

- Systems reflect the communication paths of the teams
- One service per team saves coordination on every release
- One service built by three teams is a distributed monolith

Modulith first

- One deployment with domain modules and clear interfaces
- Boundaries are checked in code, not just discussed
- A module becomes a service only when a team needs it

Measure what changes together

```ts
type Commit = { id: string; modules: readonly string[] }

const together = (commits: readonly Commit[], a: string, b: string) =>
  commits.filter((c) => c.modules.includes(a) && c.modules.includes(b)).length

// orders and pricing change together in 3 of 3 commits:
// do not separate them; otherwise every change becomes a distributed release
```

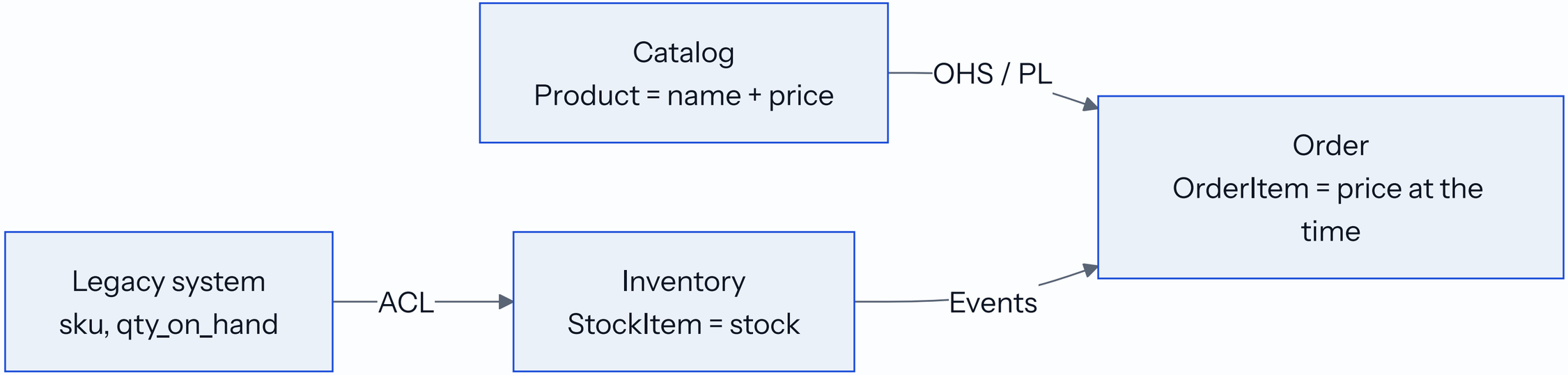
One word, three meanings

- Sales: a Product has a name, price, and description
- Inventory: a Product has stock, reservation, and storage location
- Accounting: a Product has a tax rate and revenue account

You find boundaries in conversation

- A word changes its meaning
- A rule applies here but not next door
- One team cannot translate the other team’s sentence

The Context Map shows the edges



The Anti-Corruption Layer translates in one place

```
import { z } from 'zod'

type StockItem = { productId: string; onHand: number; reserved: number }

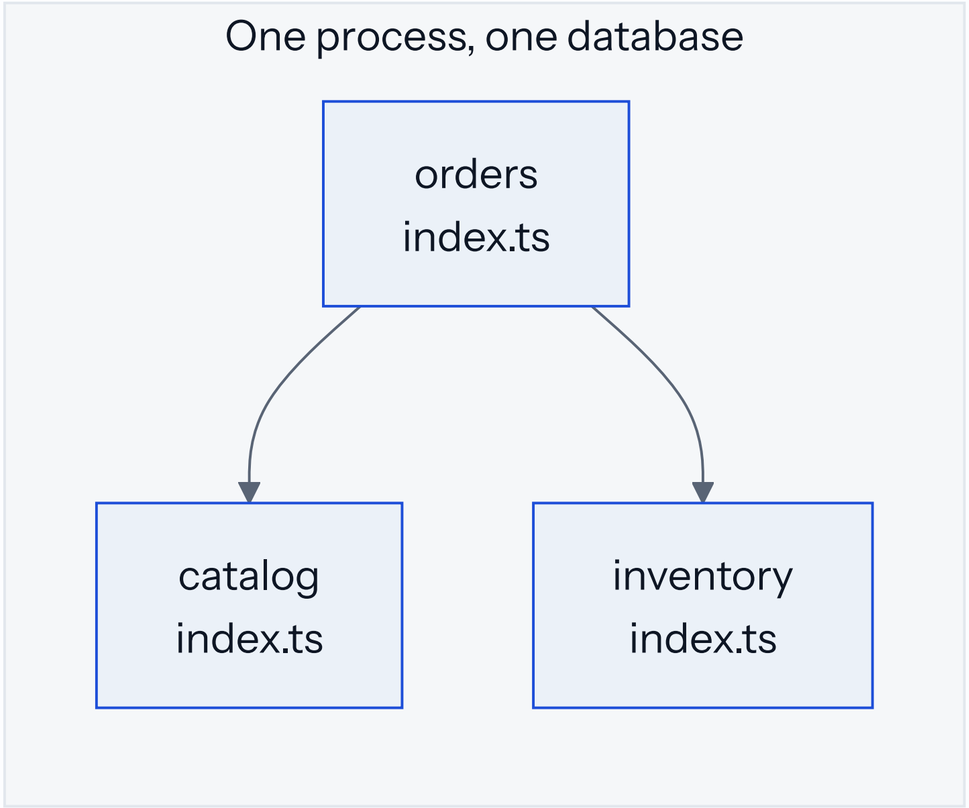
const LegacyStockRow = z.object({
  sku: z.string().min(1),
  qty_on_hand: z.coerce.number().int().min(0),
  qty_blocked: z.coerce.number().int().min(0).default(0),
})

export const fromLegacyStock = (raw: unknown): StockItem => {
  const row = LegacyStockRow.parse(raw)
  return { productId: row.sku, onHand: row.qty_on_hand, reserved: row.qty_blocked }
}
```

Context is not the same as service

- The Bounded Context is the model boundary
- The service is the deployment unit
- One context can support one or more services

One deployment, multiple modules



Only index.ts is public

```ts
// modules/catalog/index.ts - the public interface
export { priceOf } from './internal/prices.ts'

// modules/orders/index.ts
import { priceOf } from '../catalog/index.ts'
import { total } from './internal/total.ts'

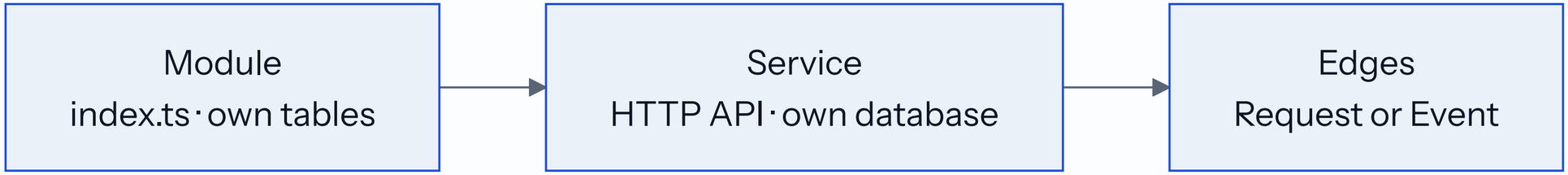
export const priceItems = (items: Array<{ productId: string; quantity: number }>) => {
  const priced = items.map((item) => ({ ...item, priceCents: priceOf(item.productId) ?? 0 }))
  return { items: priced, totalCents: total(priced) }
}
```

A boundary needs a test

```
const importPattern = /(?:import|export)[^'"]*?from\s+['"]([^'"]+)['"]/g

export const checkBoundaries = (root: string): Violation[] =>
  tsFiles(join(root, 'modules')).flatMap((file) =>
    [...readFileSync(file, 'utf8').matchAll(importPattern)]
      .map((match) => resolve(dirname(file), match[1] ?? ''))
      .filter((target) => moduleOf(target) !== moduleOf(file))
      .filter((target) => !target.endsWith(`${moduleOf(target)}${sep}index.ts`))
      .map((target) => ({ file, imports: target })),
  )
```

From module to service



The first service and its contracts

One service, one process, one task · The contract belongs to the service that offers it · A synchronous call couples availability · Start the whole system with one command

The catalog service

- Returns products and prices, nothing else
- Owns the products table alone
- Runs as its own process on port 3001

Separate app and server

```ts
import express from 'express'
import type { Database } from 'node:sqlite'

type Row = { id: string; name: string; price_cents: number }

export const createCatalogApp = ({ db }: { db: Database }) => {
  const app = express()
  app.get('/products/:productId', (req, res) => {
    const row = db.prepare('SELECT * FROM products WHERE id = ?').get(req.params.productId) as Row | undefined
    if (row) res.json({ id: row.id, name: row.name, priceCents: row.price_cents })
    else res.status(404).json({ error: 'product not found' })
  })
  return app
}
```

Configuration comes from the environment

```ts
import { Database } from 'node:sqlite'
import { createCatalogApp } from './app.ts'

export const portFromEnv = (env: NodeJS.ProcessEnv = process.env) => {
  const port = Number(env.PORT ?? 3001)
  if (!Number.isInteger(port) || port < 0 || port > 65535) throw new Error(`invalid PORT: ${env.PORT}`)
  return port
}

const db = new Database(process.env.DB_PATH ?? 'catalog.db')
createCatalogApp({ db }).listen(portFromEnv())
```

Test without network setup

- app.listen(0) chooses a free port
- An in-memory database keeps every test independent
- fetch calls the real API, just like any later client

```
test('unknown product returns 404', async () => {
  await using api = await listen(createCatalogApp({ db: openCatalogDb(':memory:') }))
  const response = await fetch(`${api.url}/products/p-9`)
  expect(response.status).toBe(404)
})
```

The contract is the only coupling

- Paths, methods, and status codes
- Request and response bodies with types
- Events with names and data

One source: zod schemas

```ts
import { z } from 'zod'

export const PlaceOrderInput = z.strictObject({
  customerId: z.string().min(1),
  items: z.array(z.strictObject({
    productId: z.string().min(1),
    quantity: z.number().int().min(1),
  })).min(1),
})

export const Order = z.object({
  id: z.string(),
  status: z.enum(['draft', 'placed', 'paid', 'shipped', 'cancelled']),
  createdAt: z.iso.datetime(),
})
```

OpenAPI 3.1 from the schema

```
import express from 'express'
import { z } from 'zod'
import { Order, PlaceOrderInput } from '@werkstatt/contracts'

const app = express()
const json = (schema: z.ZodType) => ({ 'application/json': { schema: z.toJSONSchema(schema) } })

app.get('/openapi.json', (req, res) => {
  res.json({
    openapi: '3.1.0',
    info: { title: 'orders', version: '1.0.0' },
    paths: { '/orders': { post: {
      requestBody: { required: true, content: json(PlaceOrderInput) },
      responses: { '201': { description: 'Bestellung angelegt', content: json(Order) } },
    } } },
  })
})
```

Validate at the boundary and return Problem Details

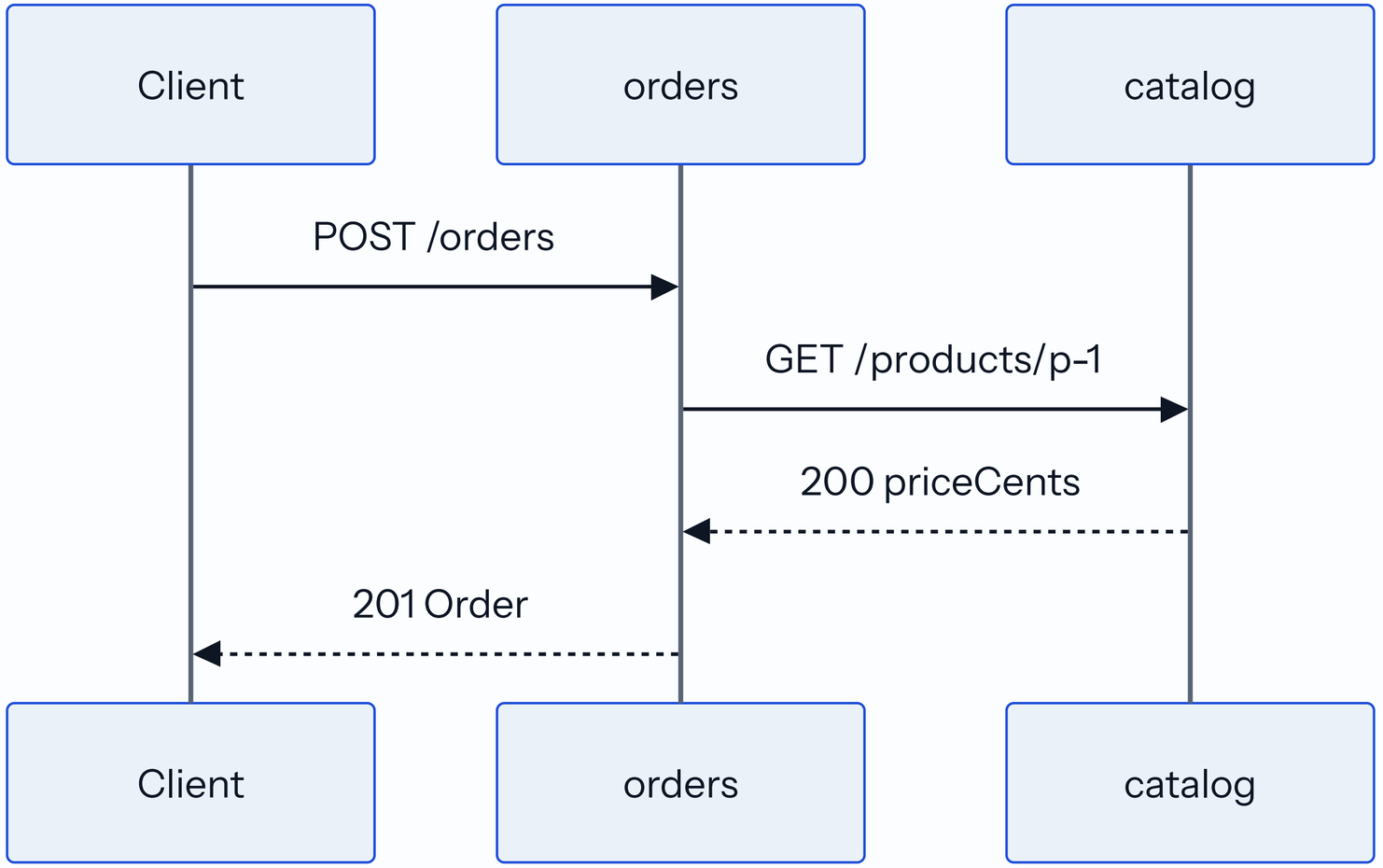
```
import { z } from 'zod'

app.use(express.json({ limit: '100kb' }))
app.post('/orders', async (req, res) => {
  const parsed = PlaceOrderInput.safeParse(req.body)
  if (!parsed.success) {
    res.status(400).type('application/problem+json').json({
      type: 'about:blank',
      title: 'invalid order',
      status: 400,
      detail: z.prettifyError(parsed.error),
    })
    return
  }
  res.status(201).json(await placeOrder(parsed.data))
})
```

Change contracts without breaking clients

- New optional fields and new endpoints are safe
- Renaming, deleting, and new required fields break clients
- Breaking Changes need a new version with a transition

orders asks catalog for prices



Availability multiplies

- Two services at 99 percent produce about 98 percent
- Every additional synchronous call lowers availability
- A slow service ties up threads and connections at the caller

Every call has a timeout

```ts
import { Product } from '@werkstatt/contracts'

export const createCatalogClient = (baseUrl: string, { timeoutMs = 500 } = {}) => ({
  async product(productId: string) {
    const response = await fetch(`${baseUrl}/products/${encodeURIComponent(productId)}`, {
      signal: AbortSignal.timeout(timeoutMs),
    }).catch((cause) => {
      throw new CatalogUnavailable('catalog not reachable', { cause })
    })
    if (response.status === 404) return undefined
    if (!response.ok) throw new CatalogUnavailable(`status ${response.status}`)
    return Product.parse(await response.json())
  },
})
```

What do you do when catalog is missing?

- Respond with 503 and let the client ask again later
- Continue working with a local copy of the prices
- Accept the order and confirm the price later

One image per service

```
FROM node:26-alpine
ARG SERVICE
ENV NODE_ENV=production SERVICE=${SERVICE} DB_PATH=/data/${SERVICE}.db
WORKDIR /app
COPY package.json package-lock.json ./
COPY packages ./packages
COPY services ./services
RUN npm ci --omit=dev && mkdir -p /data && chown node:node /data
USER node
CMD ["sh", "-c", "exec node services/${SERVICE}/src/server.ts"]
```

compose.yaml describes the system

```yaml
services:
  catalog:
    build: { context: ., args: { SERVICE: catalog } }
    environment: { PORT: '3001' }
    volumes: ['catalog-data:/data']
  orders:
    build: { context: ., args: { SERVICE: orders } }
    environment:
      PORT: '3002'
      CATALOG_URL: http://catalog:3001
      SUBSCRIBERS: http://inventory:3003/inbox
    ports: ['3002:3002']
    depends_on: [catalog]
volumes:
  catalog-data:
```

Live and ready are two questions

```
import { Router } from 'express'

export const healthRouter = (checks: Checks, state: { draining: boolean }) =>
  Router()
    .get('/health/live', (req, res) => {
      res.json({ status: 'ok' })
    })
    .get('/health/ready', async (req, res) => {
      const results = await Promise.all(Object.values(checks).map((check) => Promise.resolve().then(check).catch(() => false)))
      const ready = !state.draining && results.every(Boolean)
      res.status(ready ? 200 : 503).json({ ready })
    })
```

Start when ready, stop without aborting

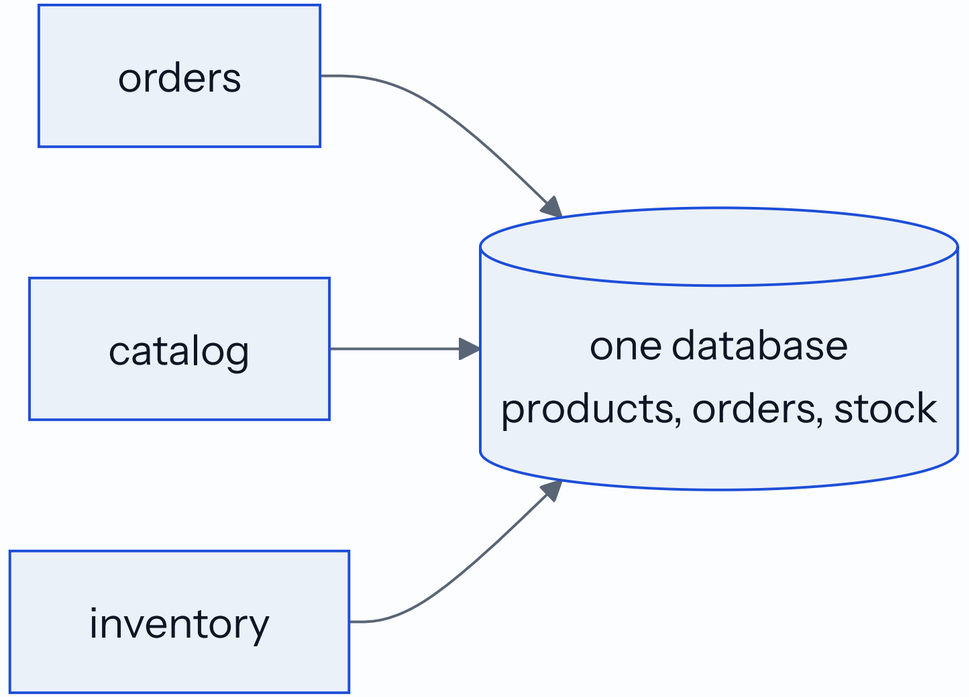
- healthcheck queries /health/ready inside the container
- condition service_healthy waits for the green check
- SIGTERM: first not ready, then finish running requests

```
orders:
  depends_on:
    catalog: { condition: service_healthy }
    inventory: { condition: service_healthy }
  healthcheck:
    test: ['CMD', 'wget', '-qO-', 'http://localhost:3002/health/ready']
    interval: 5s
    retries: 5
```

Data, events, and consistency

Each service owns its data alone · Events decouple time and availability · Save and publish in one transaction · Sagas replace distributed transactions

The shared database couples secretly



One database per service

- catalog owns products, orders owns orders, inventory owns stock
- Other services access the data only through API or events
- Each service chooses the database that fits its access patterns

External data: ask or copy



The copy applies only newer data

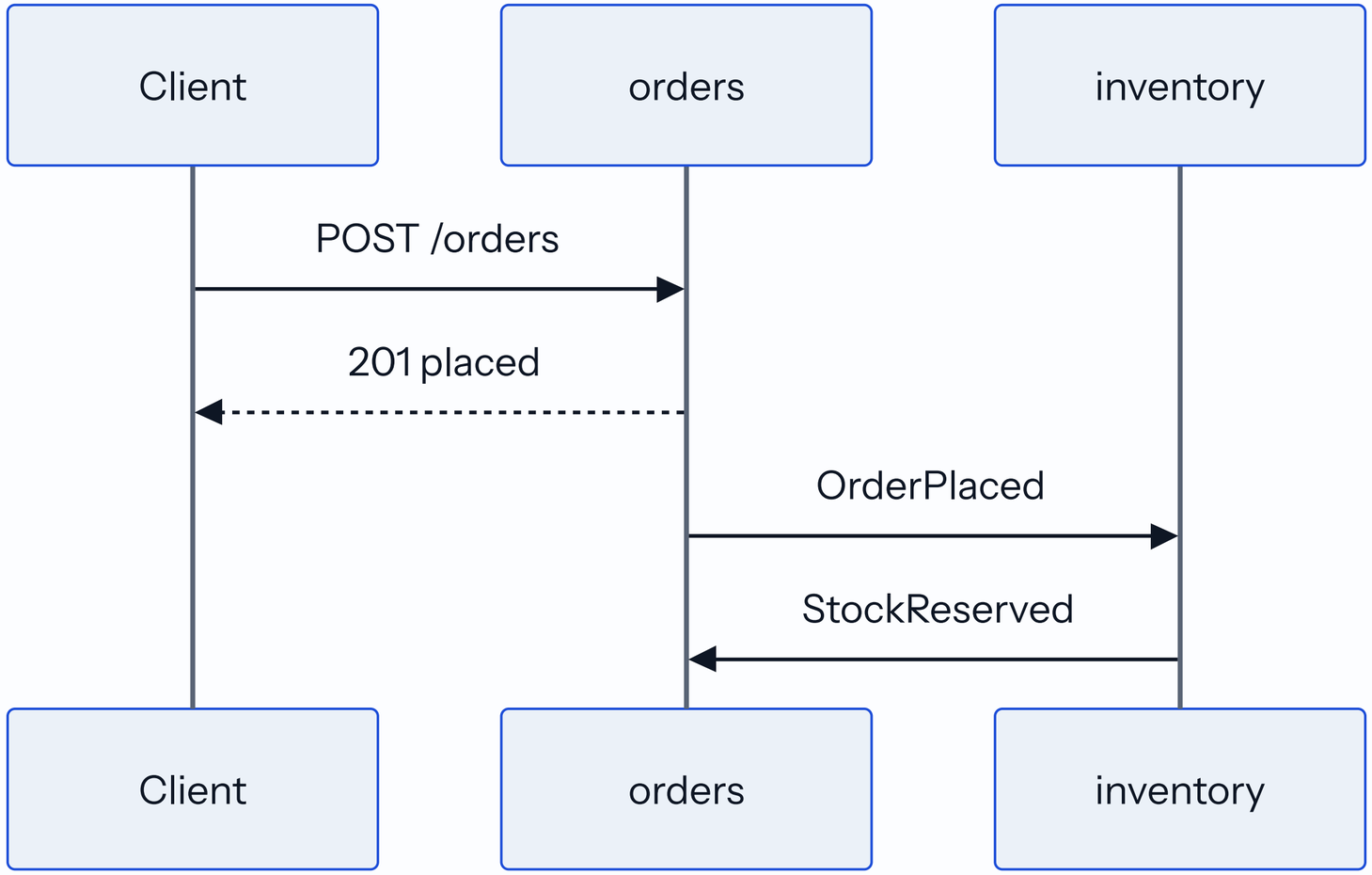
```
const upsert = db.prepare(`
  INSERT INTO product_view (product_id, name, price_cents, version) VALUES (?, ?, ?, ?)
  ON CONFLICT (product_id) DO UPDATE
    SET name = excluded.name, price_cents = excluded.price_cents, version = excluded.version
    WHERE excluded.version > product_view.version`)

export const apply = (change: ProductChanged): boolean =>
  upsert.run(change.productId, change.name, change.priceCents, change.version).changes > 0
```

An event reports a fact

- A Command wants something and may be rejected
- An event reports what has already happened
- The sender does not know the receivers

Events decouple time and availability



An event has an envelope

```
import { z } from 'zod'

export const OrderPlaced = z.object({
  id: z.string().min(1),
  type: z.literal('OrderPlaced'),
  occurredAt: z.iso.datetime(),
  data: z.object({
    orderId: z.string().min(1),
    items: z.array(z.object({ productId: z.string(), quantity: z.number().int().min(1) })).min(1),
  }),
})

export const ShopEvent = z.discriminatedUnion('type', [OrderPlaced, StockReserved, StockRejected])
```

The inbox makes duplicate delivery harmless

```
app.post('/inbox', (req, res) => {
  const parsed = ShopEvent.safeParse(req.body)
  if (!parsed.success) {
    res.status(400).json({ error: 'invalid event' })
    return
  }
  const event = parsed.data
  const fresh = transaction(db, () => {
    if (!firstDelivery(db, event.id)) return false
    if (event.type === 'OrderPlaced') reserve(event)
    return true
  })
  res.status(fresh ? 202 : 200).json({ duplicate: !fresh })
})
```

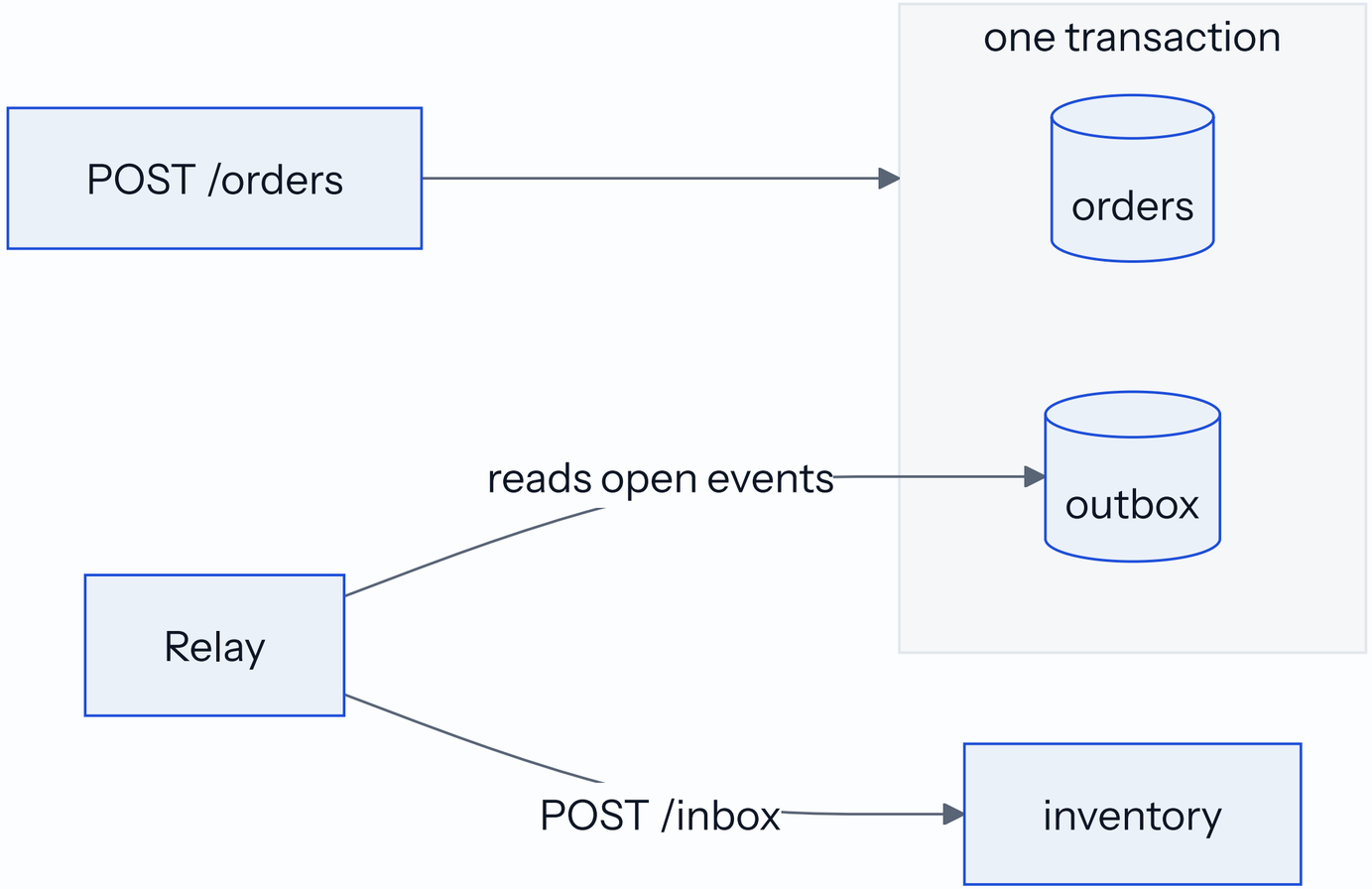
HTTP is enough; a broker scales

- HTTP and Outbox: no infrastructure, receivers are in configuration
- Brokers such as Kafka, NATS, or RabbitMQ: fan-out, buffer, and replay
- Cloud services such as SQS or EventBridge: operated by the provider

Two writes, one crash

```
app.post('/orders', async (req, res) => {
  const order = await priceAndCreate(req.body)
  db.prepare('INSERT INTO orders (id, body) VALUES (?, ?)').run(order.id, JSON.stringify(order))
  // Crash here: order saved, event never sent
  await publish(orderPlaced(order))
  res.status(201).json(order)
})
```

The event first lives in its own database



Order and event in one transaction

```
export const transaction = <T>(db: Database, fn: () => T): T => {
  db.exec('BEGIN')
  try {
    const result = fn()
    db.exec('COMMIT')
    return result
  } catch (error) {
    db.exec('ROLLBACK')
    throw error
  }
}

transaction(db, () => {
  db.prepare('INSERT INTO orders (id, body) VALUES (?, ?)').run(order.id, JSON.stringify(order))
  db.prepare('INSERT INTO outbox (event) VALUES (?)').run(JSON.stringify(orderPlaced(order)))
})
```

The relay delivers at least once

```
export const relayOnce = async (db: Database, publish: (event: ShopEvent) => Promise<void>) => {
  const rows = db.prepare('SELECT seq, event FROM outbox WHERE published_at IS NULL ORDER BY seq LIMIT 100').all() as Row[]
  for (const row of rows) {
    try {
      await publish(JSON.parse(row.event))
    } catch {
      break
    }
    db.prepare('UPDATE outbox SET published_at = ? WHERE seq = ?').run(new Date().toISOString(), row.seq)
  }
}
```

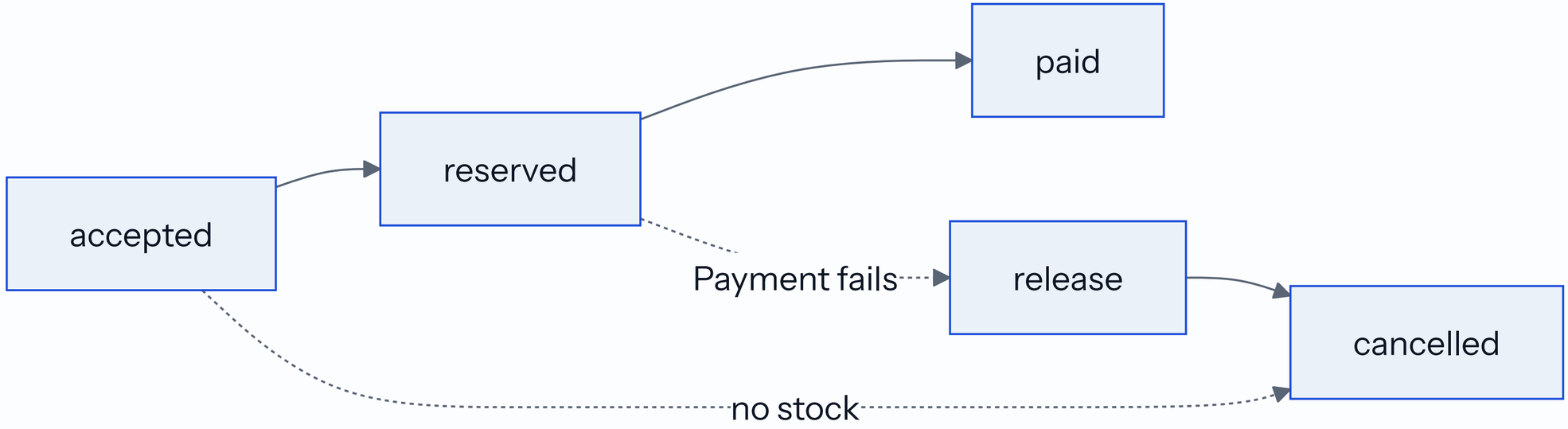
Production variants

- Polling relay in the service, as in the workshop shop
- Change Data Capture reads the database log, for example with Debezium
- A job regularly cleans up published entries

No rollback across service boundaries

- Ordering, reserving, and paying are in three services
- Each service has only its own local transaction
- A Saga replaces a shared distributed transaction here

Compensation is a domain action



The saga as a Pure Function

```
// Excerpt from the compensation policy:
if (event.type === 'PaymentFailed' && state.step === 'paying') {
  return {
    state: { ...state, step: 'releasing' },
    commands: [{ type: 'ReleaseStock', orderId }],
  }
}
// StockReleased -> CancelOrder
// OrderCancelled -> cancelled
// Error -> retry or resolve manually
```

Duplicates and latecomers

- A duplicate StockReserved must not trigger a second payment
- StockReserved after an abort request triggers ReleaseStock
- Each state defines which message still has an effect

Choreography or orchestration?

- Choreography: each service reacts to events from the others
- Orchestration: one service controls the steps with commands
- Workflow engines pay off for long flows with wait times

Resilience, observability, and tests

Timeouts, retries, and idempotency belong together · One request, one trace ID across all services · Contract tests instead of an end-to-end zoo

Not every error is temporary

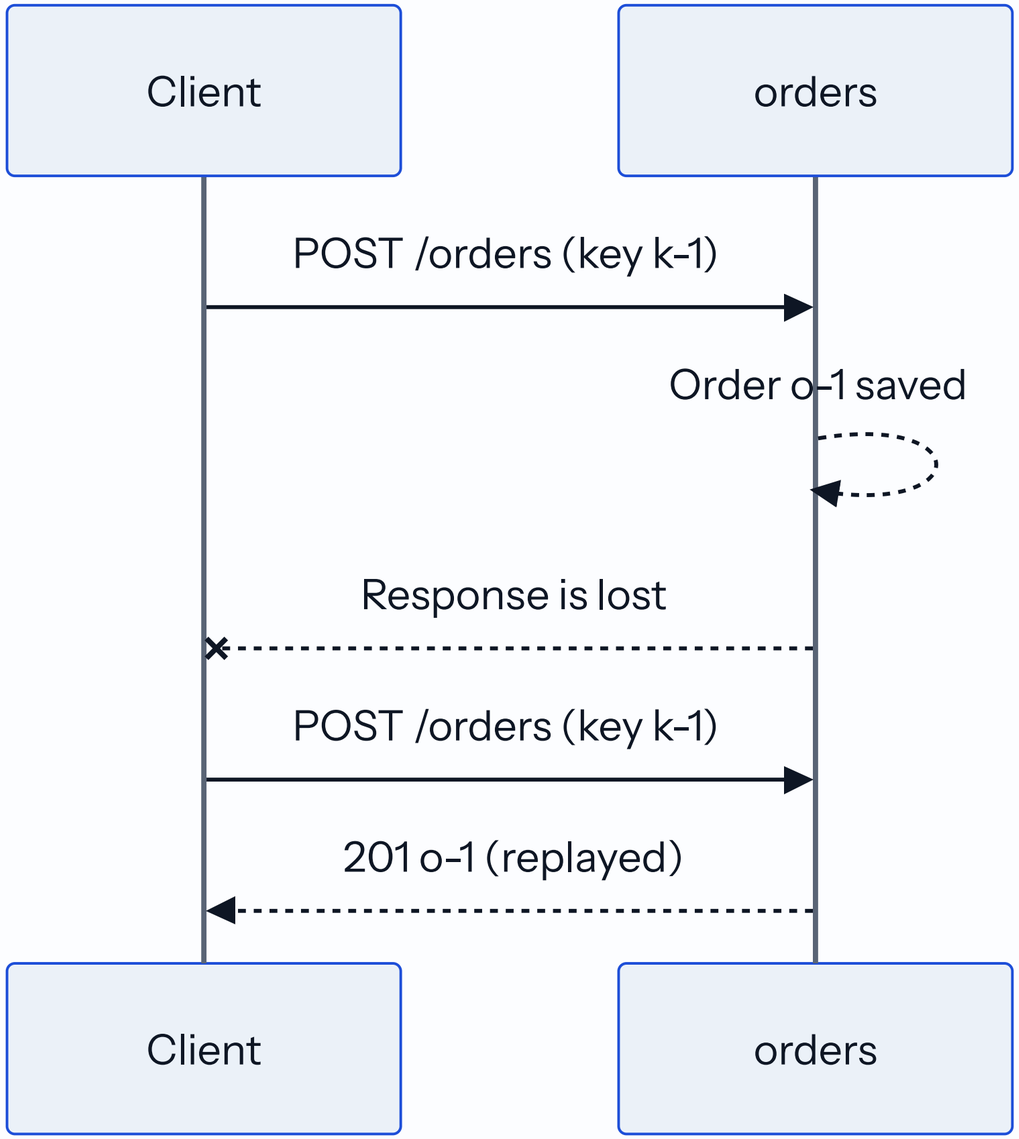
- Retry: network errors, timeouts, 429, 502, 503, 504
- Do not retry: 400, 404, 409, 422, and your own bugs
- The server’s Retry-After wins over your own wait time

Backoff with jitter

```ts
import { setTimeout } from 'node:timers/promises'

export const withRetry = async <T>(fn: (attempt: number) => Promise<T>, { retries = 3, baseMs = 100, maxMs = 2000 } = {}) => {
  for (let attempt = 0; ; attempt++) {
    try {
      return await fn(attempt)
    } catch (error) {
      if (attempt >= retries || !isRetryable(error)) throw error
      await setTimeout(Math.round(Math.random() * Math.min(maxMs, baseMs * 2 ** attempt)))
    }
  }
}
```

A retry must not order twice



Idempotency key as middleware

```
export const idempotency = (store = new Map<string, Stored>()): RequestHandler => (req, res, next) => {
  const key = req.get('idempotency-key')
  if (!key) return next()
  const fingerprint = sha256(JSON.stringify(req.body))
  const seen = store.get(key)
  if (seen && seen.fingerprint !== fingerprint) return void res.status(422).end()
  if (seen) return void res.set('idempotent-replayed', 'true').status(seen.status).json(seen.body)
  const json = res.json.bind(res)
  res.json = (body) => (store.set(key, { fingerprint, status: res.statusCode, body }), json(body))
  next()
}
```

When a service is down for a long time

- A Circuit Breaker stops calls after many errors
- A Bulkhead limits concurrent calls per dependency
- A fallback returns a worse but usable response

Three logs, one error

- One request runs through orders, catalog, and inventory
- Each service writes its own logs
- Without a shared ID, nothing fits together

W3C Trace Context

```
traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01
             version
                trace-id: 32 hex, same for the whole request
                                                parent-id: 16 hex, new per hop
                                                                    flags
```

Pass the context on

```
const storage = new AsyncLocalStorage<TraceContext>()

export const tracing: RequestHandler = (req, res, next) => {
  const incoming = parseTraceparent(req.get('traceparent'))
  const context = { traceId: incoming?.traceId ?? hex(16), spanId: hex(8) }
  res.set('traceparent', formatTraceparent(context))
  storage.run(context, next)
}

export const tracedFetch = (url: string, init: RequestInit = {}) => {
  const headers = new Headers(init.headers)
  const trace = storage.getStore()
  if (trace) headers.set('traceparent', formatTraceparent(trace))
  return fetch(url, { ...init, headers })
}
```

Logs as JSON with a trace ID

```ts
export const log = (level: 'info' | 'warn' | 'error', message: string, fields: Record<string, unknown> = {}) => {
  const trace = storage.getStore()
  console.log(JSON.stringify({
    time: new Date().toISOString(),
    level,
    message,
    trace_id: trace?.traceId,
    span_id: trace?.spanId,
    ...fields,
  }))
}

log('info', 'order placed', { orderId: 'o-1', totalCents: 2598 })
```

Three signals per service are enough to start

- Rate, errors, and duration per service and endpoint
- Alerts on symptoms that users feel
- OpenTelemetry collects traces, metrics, and logs uniformly

The end-to-end zoo turns slow and red

- All services must run in the right version at the same time
- A test failure does not show which service is at fault
- Every pipeline waits for the same staging environment

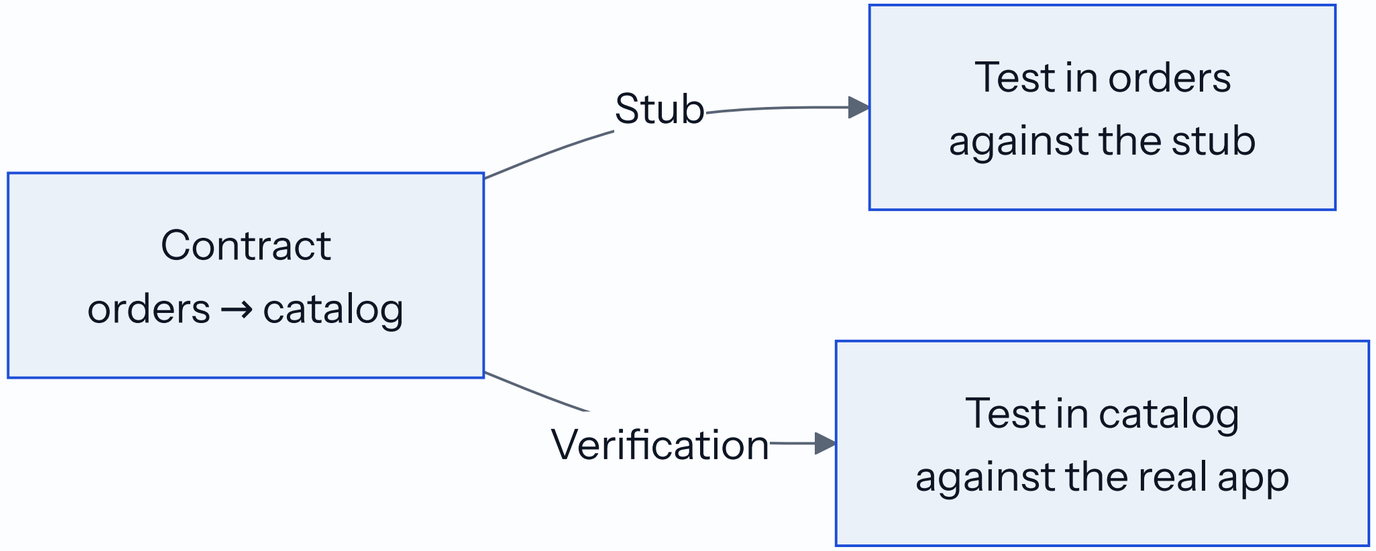
The consumer writes down what it needs

```
import { z } from 'zod'

const PriceView = z.object({ id: z.string(), priceCents: z.number().int().min(0) })

export const catalogContract: Contract = {
  consumer: 'orders',
  provider: 'catalog',
  interactions: [
    {
      description: 'liefert den Preis von p-1',
      request: { method: 'GET', path: '/products/p-1' },
      response: { status: 200, schema: PriceView, example: { id: 'p-1', name: 'Hammer', priceCents: 1299 } },
    },
    { description: 'kennt p-unknown nicht', request: { method: 'GET', path: '/products/p-unknown' }, response: { status: 404 } },
  ],
}
```

One contract, two tests



The provider verifies every contract

```
export const verifyProvider = async (app: Express, contract: Contract) => {
  await using provider = await listen(app)
  const results = []
  for (const { description, request, response } of contract.interactions) {
    const actual = await fetch(`${provider.url}${request.path}`, { method: request.method })
    const parsed = response.schema?.safeParse(await actual.json())
    const ok = actual.status === response.status && (parsed?.success ?? true)
    results.push({ description, ok })
  }
  return results
}
```

Pact and brokers for many teams

- Pact stores contracts as JSON and supports many languages
- A broker distributes contracts and stores results
- can-i-deploy checks all consumers before every deployment

Questions?

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

Workshops für Teams, die Architekturentscheidungen selbst treffen