API Router (Gateway) for routers.systems — Design & Operations Guide

Audience: You (Rex) and collaborators building/operating a unified API entry point at `api.routers.systems` that routes traffic to multiple product backends (e.g., Trade Router at `trade.routers.systems`, Task/Agent Orchestrator at `task.routers.systems`).

Goal: Ship a production-grade gateway that's simple now, scalable later, with clear policies for auth, versioning, routing, rate-limits, observability, and zero-downtime migration from the current setup.

---

0) TL;DR Launch Plan

1. **Stand up a new gateway service** (containerized) on Render as `api-router`.
2. **Attach a staging domain** (e.g. `api-new.routers.systems`) to the gateway.
3. Configure **path-based routing**:
4. `/v1/trade/*` → Trade backend (existing Render service URL)
5. `/v1/task/*` → Task backend (new service URL)
6. Add **baseline security** (TLS, headers, CORS), **auth** (JWT/OIDC), **rate-limits**, **health checks**, and **structured logs**.
7. **Test end-to-end** on `api-new.routers.systems`.
8. **Swap domain**: detach `api.routers.systems` from the current Trade service, attach to the gateway. Keep legacy routes working via proxy/redirect.
9. Publish **OpenAPI** and **client SDKs**, define **SLOs**, and set up **alerts**.

---

1) Architecture Overview

1.1 Components

- **API Router (Gateway)** @ `api.routers.systems` — single entrypoint (HTTP/2/TLS).
- **Product backends** — e.g., Trade API service, Task/Agent API service; each can scale independently.
- **Identity Provider** @ `id.routers.systems` (OIDC/JWT) — issue/validate access tokens with product scopes.
- **Observability stack** — logs, metrics, traces; central dashboards and alerts.
- **(Optional) Edge/WAF** — Cloudflare or similar for DDoS, caching of safe GETs, geo rules.

1.2 Routing Strategy

- **Unified host**: `api.routers.systems`
- **Versioned, namespaced paths**:
- `/v1/trade/...`

- `/v1/task/...`
- Future: `/v1/data/...` etc.
- Optional **product header**: `X-Router-Product: trade|task` for internal tools/telemetry.

1.3 Deployment topology (Render)

- Each service is a separate Render Web Service.
- The gateway uses **reverse proxy** to reach backends via their HTTPS origins.
- Domain attachments are moved in the Render dashboard or via blueprint (IaC) when cutting over.

---

2) Domains & TLS

- **Primary**: `api.routers.systems` → Gateway service
- **Staging**: `api-new.routers.systems` → Gateway service (temporary)
- **Product hosts (optional)**: `api.trade.routers.systems`, `api.task.routers.systems` (aliases or direct)
- **TLS**: Let Render manage certs for attached domains; HSTS enabled at gateway response layer.
- **Note on wildcards**: `*.routers.systems` doesn't cover deeper sub-subdomains like `api.task.routers.systems` — issue specific certs if needed (Render handles this per attached domain).

---

3) API Standards (long-term contract)

3.1 Versioning

- Path-level versioning: `/v1/...`, `/v2/...`.
- **No breaking changes** in a live version; add fields, never repurpose.
- Deprecate with headers: `Deprecation: true`, `Sunset: <date>`, link to migration docs.

3.2 Content & Conventions

- **Content-Type**: `application/json; charset=utf-8` (default).
- **Errors**: RFC 7807 `application/problem+json` structure:

```
{
  "type": "https://docs.routers.systems/errors/invalid-argument",
  "title": "Invalid argument",
  "status": 400,
  "detail": "field X must be > 0",
  "instance": "/v1/trade/orders/123"
}
```

- **Pagination**: `?page` / `?page_size` or cursor (`?cursor`, `?limit`); advertise `Link` headers.

- **Idempotency**: Support `Idempotency-Key` on POST endpoints that create resources; store dedupe keys for 24–72h.
- **Retries**: Clients may retry idempotent requests on 408/429/5xx with backoff; gateway handles safe server-side retries for idempotent upstreams.

3.3 Rate-limit Headers

- Send **standardized headers** with 200/429 responses:
- `RateLimit-Limit: <requests>/<window>`
- `RateLimit-Remaining: <n>`
- `RateLimit-Reset: <epoch-secs>`
- On 429: `Retry-After: <seconds>`

3.4 CORS

- Allow only known app origins (e.g., `https://trade.routers.systems`, `https://task.routers.systems`).
- Preflight cache: `Access-Control-Max-Age: 600`.

---

4) Authentication & Authorization

4.1 Model

- **OIDC** via `id.routers.systems` (or a trusted provider).
- **JWT Access Tokens** (RS256/ES256) with claims:
- `iss`, `sub`, `exp`, `aud` (e.g., `routers-api`), `scope` (space-separated), and optional `org`, `tier`.
- **Scopes**:
- Trade: `trade.read`, `trade.write`, `trade.webhook`
- Task: `task.read`, `task.write`, `task.admin`
- **Gateway policy** (coarse): ensure `aud` and at least one required scope for the prefix.
- **Service policy** (fine): resource-level checks inside each backend.

4.2 Centralized vs Delegated AuthZ

- **Simple start**: backends validate JWT; gateway just forwards `Authorization` and strips hop-by-hop headers.
- **Centralized later**: gateway verifies JWT (JWKS), enforces coarse scopes, injects `X-Principal-*` headers; backends trust gateway.

---

5) Security Baseline

- **TLS** everywhere; enforce **HSTS**: `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`.
- **Headers**:

- `X-Content-Type-Options: nosniff`
- `X-Frame-Options: DENY`
- `Referrer-Policy: no-referrer`
- `Permissions-Policy: geolocation=(), microphone=(), camera=()`
- **Request size limit**: 1–10 MB depending on endpoints; reject early at gateway.
- **WAF/Edge** (optional): Cloudflare in front for DDoS shielding and IP rate rules.
- **Secrets**: store in Render environment variables; rotate regularly.
- **PII**: avoid logging PII; redact tokens; structured logs only.

---

6) Observability & SLOs

6.1 Logging

- **Structured JSON logs**: `timestamp`, `trace_id`, `route`, `product`, `status`, `latency_ms`, `bytes_in/out`, `client_ip` (hashed).
- Generate/propagate **Request ID** header: `X-Request-Id` and **W3C Trace Context** `traceparent`.

6.2 Metrics (per product & per route)

- RPS, p50/p90/p99 latency, error rate by class (4xx/5xx), 429 rate, upstream saturation.
- Rate-limit counters, JWT validation failures, CORS preflight counts.

6.3 Tracing

- Propagate `traceparent`; sample a small percentage in production; capture upstream spans.

6.4 SLOs (initial)

- Availability (monthly): **99.9%** for gateway; latency SLO p95 **< 250 ms** for light GETs.
- Page alerts on 5xx rate spikes or p95 regression > 20% over baseline.

---

7) Rate Limiting & Quotas

- **Buckets**:
- Global per product (e.g., Trade, Task)
- Per API key / per `sub` (user)
- Optionally per IP (abuse guard)
- **Enforcement**: return 429 with headers above.
- **Burst** vs **steady state** windows; different tiers by plan.
- Start coarse (gateway-level) and refine inside services if needed.

---

8) Implementation Options (choose one now, keep others as future paths)

Option A — Node/Express Gateway (max flexibility, simple to ship)

- Libraries: `express`, `http-proxy-middleware`, `helmet`, `cors`, `compression`, `pino-http`, `express-rate-limit`, `jose` (JWT/JWKS).
- Pros: easy JWT verification and custom logic (headers, idempotency, metrics).
- Cons: your code to maintain (but small).

server.ts (excerpt)

```
import express from 'express';
import helmet from 'helmet';
import cors from 'cors';
import compression from 'compression';
import pino from 'pino-http';
import rateLimit from 'express-rate-limit';
import { createProxyMiddleware } from 'http-proxy-middleware';
import { createLocalJWKSet, jwtVerify } from 'jose';
import fetch from 'node-fetch';

const TRADE_API = process.env.TRADE_API_URL!; // e.g. https://trade-
api.onrender.com
const TASK_API  = process.env.TASK_API_URL!;  // e.g. https://task-
api.onrender.com
const OIDC_ISS  = process.env.OIDC_ISSUER_URL!; // https://id.routers.systems
const OIDC_AUD  = process.env.OIDC_AUDIENCE!;   // routers-api

const app = express();

app.disable('x-powered-by');
app.use(helmet({
  contentSecurityPolicy: false, // APIs typically disable CSP
  crossOriginEmbedderPolicy: false,
}));
app.use(compression());
app.use(pino());

// CORS: restrict to known frontends
const allowed = new Set([
  'https://trade.routers.systems',
  'https://task.routers.systems',
]);
app.use(cors({
  origin: (origin, cb) => {
    if (!origin || allowed.has(origin)) return cb(null, true);
    return cb(new Error('CORS blocked'));
  },
  credentials: false,
  maxAge: 600,
}));

// Body size limit
app.use(express.json({ limit: '2mb' }));

// Rate limit (coarse global example)
const rl = rateLimit({
  windowMs: 60_000, // 1 min
  max: 600,         // 600 req/min by default
  standardHeaders: true,
  legacyHeaders: false,
});
app.use('/v1', rl);

// JWT verification middleware (coarse gateway auth)
let jwks: ReturnType<typeof createLocalJWKSet> | null = null;
async function getJWKS() {
  if (jwks) return jwks;
  const res = await fetch(`${OIDC_ISS}/.well-known/jwks.json`);
  const data = await res.json();
  jwks = createLocalJWKSet(data);
  return jwks;
}

async function verifyJWT(req, res, next) {
  try {
    const auth = req.headers['authorization'];
    if (!auth?.startsWith('Bearer ')) return res.status(401).json({ error:
'missing token' });
    const token = auth.slice(7);
    const { payload } = await jwtVerify(token, await getJWKS(), {
      issuer: OIDC_ISS,
      audience: OIDC_AUD,
    });
    // basic scope check by path
    const scope = String(payload.scope || '');
    if (req.path.startsWith('/v1/trade') && !/\btrade\.(read|write|admin)
\b/.test(scope))
      return res.status(403).json({ error: 'insufficient scope' });
    if (req.path.startsWith('/v1/task') && !/\btask\.(read|write|admin)
\b/.test(scope))
      return res.status(403).json({ error: 'insufficient scope' });
    // inject identity headers downstream (optional)
    res.setHeader('X-Principal-Sub', String(payload.sub));
    res.setHeader('X-Principal-Scopes', scope);
    next();
  } catch (e) {
    return res.status(401).json({ error: 'invalid token' });
  }
}

app.use('/v1', verifyJWT);

// Proxy helpers
const commonProxy = {
  changeOrigin: true,
  xfwd: true,
  onProxyReq: (proxyReq, req) => {
    proxyReq.setHeader('X-Request-Id', req.headers['x-request-id'] ||
crypto.randomUUID());
  },
};

app.use('/v1/trade', createProxyMiddleware({ target: TRADE_API, pathRewrite: {
'^/v1/trade': '' }, ...commonProxy }));
app.use('/v1/task',  createProxyMiddleware({ target: TASK_API,  pathRewrite: {
'^/v1/task':  '' }, ...commonProxy }));

app.get('/healthz', (_, res) => res.status(200).send('ok'));
app.get('/readyz',  (_, res) => res.status(200).send('ready'));
app.get('/v1/_meta', (_, res) => res.json({ service: 'api-router', version:
process.env.GIT_SHA || 'dev' }));

const port = process.env.PORT || 8080;
app.listen(port, () => console.log(`api-router listening on :${port}`));
```

Dockerfile

```
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --omit=dev
COPY . .
EXPOSE 8080
CMD ["node", "dist/server.js"]
```

Option B — Caddy (simple reverse proxy + headers)

- Pros: minimal config, great TLS, fast. Cons: JWT/forward-auth requires plugins or delegated auth to backends.

Caddyfile

```
{
  servers {
    protocols h1 h2
  }
}

api.routers.systems {
  encode zstd gzip

  header {
    Strict-Transport-Security "max-age=31536000; includeSubDomains; preload"
    X-Content-Type-Options "nosniff"
    X-Frame-Options "DENY"
    Referrer-Policy "no-referrer"
    Permissions-Policy "geolocation=(), microphone=(), camera=()"
  }

  @trade path /v1/trade/*
  handle @trade {
    reverse_proxy https://trade-api.onrender.com
  }

  @task path /v1/task/*
  handle @task {
    reverse_proxy https://task-api.onrender.com
  }

  handle {
    respond 404
  }
}
```

Option C — Traefik (built-in middlewares)

- Pros: dynamic config, forwardAuth, rate-limit middlewares. Cons: more moving parts.

traefik.yml (static)

```
entryPoints:
  websecure:
    address: ":443"
providers:
  file:
    filename: /etc/traefik/dynamic.yml
api: { dashboard: false }
log: { level: INFO }
accessLog: {}
```

dynamic.yml (routes/middlewares)

```
http:
  middlewares:
    securityHeaders:
      headers:
        stsSeconds: 31536000
        stsIncludeSubdomains: true
        contentTypeNosniff: true
        frameDeny: true
        referrerPolicy: "no-referrer"
    rateDefault:
      rateLimit:
        average: 10
        burst: 20
        period: 1s

  routers:
    trade:
      rule: "PathPrefix(`/v1/trade`)"
      service: trade
      entryPoints: [websecure]
      middlewares: [securityHeaders, rateDefault]
    task:
      rule: "PathPrefix(`/v1/task`)"
      service: task
      entryPoints: [websecure]
      middlewares: [securityHeaders, rateDefault]

  services:
    trade:
      loadBalancer:
        servers:
          - url: "https://trade-api.onrender.com"
    task:
      loadBalancer:
        servers:
          - url: "https://task-api.onrender.com"
```

Auth with Traefik: add a `forwardAuth` middleware pointing to an internal auth service that validates JWT and returns 2xx/4xx.

---

9) Render Deployment

9.1 Manual (UI) Flow

1. Create **Web Service** `api-router` with your chosen option (Node, Caddy, Traefik) using a Dockerfile.
2. Add environment variables:
3. `TRADE_API_URL`, `TASK_API_URL`
4. `OIDC_ISSUER_URL`, `OIDC_AUDIENCE` (if doing gateway JWT validation)
5. Attach domain `api-new.routers.systems` and deploy.
6. Test health, routing, CORS, auth, and rate limits.
7. **Cutover**: detach `api.routers.systems` from the Trade service, attach to `api-router`.
8. Keep legacy routes working (gateway proxies `/` → `/v1/trade` if you need a grace period) and announce deprecation.

9.2 Blueprint (example)

Use as a starting point; adapt to current Render blueprint schema.

```
services:
  - type: web
    name: api-router
    env: docker
    plan: starter
    region: oregon
    domains:
      - api.routers.systems
      - api-new.routers.systems
    healthCheckPath: /healthz
    autoDeploy: true
    envVars:
      - key: TRADE_API_URL
        value: https://trade-api.onrender.com
      - key: TASK_API_URL
        value: https://task-api.onrender.com
      - key: OIDC_ISSUER_URL
        value: https://id.routers.systems
      - key: OIDC_AUDIENCE
        value: routers-api
```

---

10) OpenAPI & SDKs

- Publish at gateway:
- Combined: `/openapi.json` (with tags per product: `Trade`, `Task`).
- Or split: `/openapi-trade.json`, `/openapi-task.json`.
- **Compose** specs with an automated job (e.g., `openapi-merge-cli` or a tiny script) and validate in CI.
- Generate client SDKs (TypeScript/Go/Python) per release; host at `docs.routers.systems`.

openapi-merge.config.json (example)

```
{
  "output": {
    "file": "dist/openapi.json",
    "version": 3
  },
  "apis": [
    { "id": "trade", "path": "./specs/openapi-trade.yaml" },
    { "id": "task",  "path": "./specs/openapi-task.yaml" }
  ]
}
```

---

11) Testing & Verification

- **Smoke**: `GET /healthz`, `/v1/_meta`, `/v1/trade/…` & `/v1/task/…` endpoints with and without tokens.
- **CORS**: preflight from `trade.routers.systems` and `task.routers.systems`.
- **Auth**: expired token, wrong `aud`, missing scope, success.
- **Rate-limit**: exceed limit → observe `429` and headers.
- **Perf**: load-test safe GETs (e.g., k6) to establish latency baselines.
- **Chaos**: take down a backend; ensure gateway surfaces 502 with trace ID and recovers quickly.

---

12) Runbooks

12.1 Domain Cutover (Zero-Downtime)

1. Verify gateway on `api-new.routers.systems`.

2. Freeze trade deployments briefly.
3. Detach `api.routers.systems` from Trade service; attach to gateway.
4. Validate `/v1/trade/*` paths; tail logs for 4xx/5xx anomalies.
5. Announce completion; unfreeze trade.
6. For a deprecation window, proxy old root routes → `/v1/trade/*` or send 301 with `Sunset` header.

12.2 Rollback

- Detach domain from gateway; reattach to Trade service. Keep the gateway running on `api-new.routers.systems` for diagnosis.

12.3 Add a New Product (`/v1/data` example)

1. Add backend service, expose health.
2. Add route rule in gateway.
3. Update OpenAPI; publish.
4. Update rate-limit policy and scopes.
5. Announce beta.

---

13) Error Handling & User Experience

- Always include a **trace ID** in error responses; log map `trace_id → cause`.
- Prefer precise `4xx` (400, 401, 403, 404, 409, 422) over generic 400.
- Circuit break flapping upstreams; return 503 with `Retry-After`.
- For webhooks, support **retries** and **signature verification**.

---

14) Governance & Naming

- Repos:
- `routers/api-router` (gateway)
- `routers/specs` (OpenAPI, SDK gen)
- Branching: trunk-based with short-lived PRs; CI runs lint/tests and spec validation.
- Labels & scopes: keep `trade.*`, `task.*`; expand with new products as `data.*` etc.

---

15) Future Enhancements

- **Centralized auth at gateway** with signed user context headers to backends.
- **Envoy** gateway for advanced policies (ext_authz, local/global rate limit, retries, outlier detection).
- **mTLS** between gateway and backends.
- **Multi-region** failover; weighted DNS; canaries/blue-green.
- **Edge caching** of safe GETs via Cloudflare with `Cache-Control` & ETags.
- **Request validation** at gateway using OpenAPI (reject malformed at edge).

16) Appendices

A) Example `.env` (Node gateway)

```
PORT=8080
TRADE_API_URL=https://trade-api.onrender.com
TASK_API_URL=https://task-api.onrender.com
OIDC_ISSUER_URL=https://id.routers.systems
OIDC_AUDIENCE=routers-api
GIT_SHA=dev
```

B) Standard Response Headers

```
X-Request-Id: <uuid>
Traceparent: 00-<trace>-<span>-01
RateLimit-Limit: 600;w=60
RateLimit-Remaining: 534
RateLimit-Reset: 1732406400
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

C) Health & Readiness Contract

- `GET /healthz` → `200 ok` when process is alive.
- `GET /readyz` → `200 ready` when upstreams are reachable and config loaded.
- `GET /v1/_meta` → `{ service, version, build_time }`.

This guide is deliberately practical: you can ship Option A (Node gateway) in hours, then evolve toward Traefik/Envoy if/when you need richer edge policies. Stick to the contracts above and your future services will slot into `api.routers.systems` cleanly with minimal churn.