readme.md for @serve.zone/dcrouter

dcrouter: The all-in-one gateway for your datacenter. 🌐

A comprehensive traffic routing solution that provides unified gateway capabilities for HTTP/HTTPS, TCP/SNI, email (SMTP), DNS, RADIUS, VPN, and remote edge ingress — all from a single process. Designed for enterprises requiring robust traffic management, automatic TLS certificate provisioning, VPN-based access control, distributed edge networking, and enterprise-grade email infrastructure.

Issue Reporting and Security

For reporting bugs, issues, or security vulnerabilities, please visit community.foss.global/. This is the central community hub for all issue reporting. Developers who sign and comply with our contribution agreement and go through identification can also get a code.foss.global/ account to submit Pull Requests directly.

Table of Contents

Features

□□ Universal Traffic Router

- **HTTP/HTTPS routing** with domain matching, path-based forwarding, and automatic TLS
- **HTTP/3 (QUIC) enabled by default** — qualifying HTTPS routes automatically get QUIC/H3 support with zero configuration
- **TCP/SNI proxy** for any protocol with TLS termination or passthrough
- **DNS server** (Rust-powered via SmartDNS) with authoritative zones, dynamic record management, and DNS-over-HTTPS
- **Multi-protocol support** on the same infrastructure via SmartProxy

□□ Complete Email Infrastructure (powered by smartmta)

- **Multi-domain SMTP server** on standard ports (25, 587, 465)
- **Pattern-based email routing** with four action types: forward, process, deliver, reject

- **DKIM signing & verification**, SPF, DMARC authentication stack
- **Enterprise deliverability** with IP warmup schedules and sender reputation tracking
- **Bounce handling** with automatic suppression lists
- **Hierarchical rate limiting** — global, per-domain, per-sender

□□ Enterprise Security

- **Automatic TLS certificates** via ACME (smartacme v9) with Cloudflare DNS-01 challenges
- **Smart certificate scheduling** — per-domain deduplication, controlled parallelism, and account rate limiting handled automatically
- **Per-domain exponential backoff** — failed provisioning attempts are tracked and backed off to avoid hammering ACME servers
- **IP reputation checking** with caching and configurable thresholds
- **Content scanning** for spam, viruses, and malicious attachments
- **Security event logging** with structured audit trails

□□ RADIUS Server

- **MAC Authentication Bypass (MAB)** for network device authentication
- **VLAN assignment** based on exact MAC, OUI prefix, or wildcard patterns
- **RADIUS accounting** for session tracking, traffic metering, and billing
- **Real-time management** via OpsServer API

□□ Remote Ingress (powered by remoteingress)

- **Distributed edge networking** — accept traffic at remote edge nodes and tunnel it to the hub
- **Edge registration CRUD** with secret-based authentication
- **Auto-derived ports** — edges automatically pick up ports from routes tagged with `remoteIngress.enabled`
- **Connection tokens** — generate a single opaque base64url token containing hubHost, hubPort, edgeId, and secret for easy edge provisioning
- **Real-time status monitoring** — connected/disconnected state, public IP, active tunnels, heartbeat tracking
- **OpsServer dashboard** with enable/disable, edit, secret regeneration, token copy, and delete actions

□□ VPN Access Control (powered by smartvpn)

- **WireGuard + native transports** — standard WireGuard clients (iOS, Android, macOS, Windows, Linux) plus custom WebSocket/QUIC tunnels
- **Route-level VPN gating** — mark any route with `vpn: { enabled: true }` to restrict access to VPN clients only, or `vpn: { enabled: true, mandatory: false }` to add VPN clients alongside existing access rules
- **Tag-based access control** — assign `serverDefinedClientTags` to clients and restrict routes with `allowedServerDefinedClientTags`
- **Constructor-defined clients** — pre-define VPN clients with tags in config for declarative, code-driven setup
- **Rootless operation** — uses userspace NAT (smoltcp) with no root required
- **Destination policy** — configurable `forceTarget`, `block`, or `allow` with allowList/blockList for granular traffic control
- **Client management** — create, enable, disable, rotate keys, export WireGuard/SmartVPN configs via OpsServer API and dashboard
- **IP-based enforcement** — VPN clients get IPs from a configurable subnet; SmartProxy enforces `ipAllowList` per route
- **PROXY protocol v2** — the NAT engine sends PP v2 on outbound connections to preserve VPN client identity

⚡ High Performance

- **Rust-powered proxy engine** via SmartProxy for maximum throughput
- **Rust-powered MTA engine** via smartmta (TypeScript + Rust hybrid) for reliable email delivery
- **Rust-powered DNS engine** via SmartDNS for high-performance UDP and DNS-over-HTTPS
- **Connection pooling** for outbound SMTP and backend services
- **Socket-handler mode** — direct socket passing eliminates internal port hops
- **Real-time metrics** via SmartMetrics (CPU, memory, connections, throughput)

□□ Persistent Storage & Caching

- **Multiple storage backends**: filesystem, custom functions, or in-memory
- **Embedded cache database** via smartdata + smartdb (MongoDB-compatible)
- **Automatic TTL-based cleanup** for cached emails and IP reputation data

□□ OpsServer Dashboard

- **Web-based management interface** with real-time monitoring
- **JWT authentication** with session persistence
- **Live views** for connections, email queues, DNS queries, RADIUS sessions, certificates, remote ingress edges, VPN clients, and security events
- **Domain-centric certificate overview** with backoff status and one-click reprovisioning
- **Remote ingress management** with connection token generation and one-click copy
- **Read-only configuration display** — DcRouter is configured through code
- **Smart tab visibility handling** — auto-pauses all polling, WebSocket connections, and chart updates when the browser tab is hidden, preventing resource waste and tab freezing

□□ Programmatic API Client

- **Object-oriented API** — resource classes (`Route`, `Certificate`, `ApiToken`, `RemoteIngress`, `Email`) with instance methods
- **Builder pattern** — fluent `.setName().setMatch().save()` chains for creating routes, tokens, and edges
- **Auto-injected auth** — JWT identity and API tokens included automatically in every request
- **Dual auth modes** — login with credentials (JWT) or pass an API token for programmatic access
- **Full coverage** — wraps every OpsServer endpoint with typed request/response pairs

Installation

```
pnpm add @serve.zone/dcrouter
# or
npm install @serve.zone/dcrouter
```

Prerequisites

- **Node.js 20+** with ES module support
- Valid domain with DNS control (for ACME certificate automation)
- Cloudflare API token (for DNS-01 challenges) — optional

Quick Start

Basic HTTP/HTTPS Router

```
import { DcRouter } from '@serve.zone/dcrouter';

const router = new DcRouter({
  smartProxyConfig: {
    routes: [
      {
        name: 'web-app',
        match: { domains: ['example.com', 'www.example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.10', port: 8080 }],
          tls: { mode: 'terminate', certificate: 'auto' }
        }
      }
    ],
    acme: {
      email: 'admin@example.com',
      enabled: true,
      useProduction: true
    }
  }
});

await router.start();
```

Basic Email Server

```
import { DcRouter } from '@serve.zone/dcrouter';

const router = new DcRouter({
  emailConfig: {
    ports: [25, 587, 465],
```

```
    hostname: 'mail.example.com',
    domains: [
      {
        domain: 'example.com',
        dnsMode: 'external-dns'
      }
    ],
    routes: [
      {
        name: 'process-all',
        match: { recipients: '*@example.com' },
        action: {
          type: 'process',
          process: { scan: true, dkim: true, queue: 'normal' }
        }
      }
    ]
  }
});

await router.start();
```

Full Stack with Dashboard

```
import { DcRouter } from '@serve.zone/dcrouter';

const router = new DcRouter({
  // HTTP/HTTPS routing
  smartProxyConfig: {
    routes: [
      {
        name: 'website',
        match: { domains: ['example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.10', port: 80 }],
          tls: { mode: 'terminate', certificate: 'auto' }
        }
```

```
    }
  ],
  acme: { email: 'ssl@example.com', enabled: true, useProduction: true }
},

// Email system (powered by smartmta)
emailConfig: {
  ports: [25, 587, 465],
  hostname: 'mail.example.com',
  domains: [{ domain: 'example.com', dnsMode: 'external-dns' }],
  routes: [
    {
      name: 'inbound-mail',
      match: { recipients: '*@example.com' },
      action: { type: 'process', process: { scan: true, dkim: true, queue: 'normal' } }
    }
  ]
},

// Authoritative DNS
dnsNsDomains: ['ns1.example.com', 'ns2.example.com'],
dnsScopes: ['example.com'],
publicIp: '203.0.113.1',
dnsRecords: [
  { name: 'example.com', type: 'A', value: '203.0.113.1' },
  { name: 'www.example.com', type: 'CNAME', value: 'example.com' }
],

// RADIUS authentication
radiusConfig: {
  authPort: 1812,
  acctPort: 1813,
  clients: [
    { name: 'switch-1', ipRange: '192.168.1.0/24', secret: 'radius-secret', enabled: true }
  ],
  vlanAssignment: {
    defaultVlan: 100,
    allowUnknownMacs: true,
    mappings: [
```

```
        { mac: 'aa:bb:cc:dd:ee:ff', vlan: 10, enabled: true },
        { mac: 'aa:bb:cc', vlan: 20, enabled: true }  // OUI prefix
      ]
    },
    accounting: { enabled: true, retentionDays: 30 }
  },

  // Remote Ingress — edge nodes tunnel traffic to this hub
  remoteIngressConfig: {
    enabled: true,
    tunnelPort: 8443,
    hubDomain: 'hub.example.com',
  },

  // VPN — restrict sensitive routes to VPN clients
  vpnConfig: {
    enabled: true,
    serverEndpoint: 'vpn.example.com',
    clients: [
      { clientId: 'dev-laptop', serverDefinedClientTags: ['engineering'] },
    ],
  },

  // Persistent storage
  storage: { fsPath: '/var/lib/dcrouter/data' },

  // Cache database
  cacheConfig: { enabled: true, storagePath: '~/.serve.zone/dcrouter/tsmdb' },

  // TLS & ACME
  tls: { contactEmail: 'admin@example.com' },
  dnsChallenge: { cloudflareApiKey: process.env.CLOUDFLARE_API_KEY }
});

await router.start();
// OpsServer dashboard available at http://localhost:3000
```

Architecture

System Overview

```
graph TB
    subgraph "External Traffic"
        HTTP[HTTP/HTTPS Clients]
        SMTP[SMTP Clients]
        TCP[TCP Clients]
        DNS[DNS Queries]
        RAD[RADIUS Clients]
        EDGE[Edge Nodes]
        VPN[VPN Clients]
    end

    subgraph "DcRouter Core"
        DC[DcRouter Orchestrator]
        SP[SmartProxy Engine<br/><i>Rust-powered</i>]
        ES[smartmta Email Server<br/><i>TypeScript + Rust</i>]
        DS[SmartDNS Server<br/><i>Rust-powered</i>]
        RS[SmartRadius Server]
        RI[RemoteIngress Hub<br/><i>Rust data plane</i>]
        VS[SmartVPN Server<br/><i>Rust data plane</i>]
        CM[Certificate Manager<br/><i>smartacme v9</i>]
        OS[OpsServer Dashboard]
        MM[Metrics Manager]
        SM[Storage Manager]
        CD[Cache Database]
    end

    subgraph "Backend Services"
        WEB[Web Services]
        MAIL[Mail Servers]
        DB[Databases]
        API[Internal APIs]
    end

    HTTP --> SP
    TCP --> SP
    SMTP --> ES
    DNS --> DS
```

```
    RAD --> RS
    EDGE --> RI
    VPN --> VS

    DC --> SP
    DC --> ES
    DC --> DS
    DC --> RS
    DC --> RI
    DC --> VS
    DC --> CM
    DC --> OS
    DC --> MM
    DC --> SM
    DC --> CD

    SP --> WEB
    SP --> API
    ES --> MAIL
    ES --> DB
    RI --> SP

    CM -.-> SP
    CM -.-> ES
```

Core Components

Component	Package	Description
DcRouter	`@serve.zone/dcrouter`	Central orchestrator — starts, stops, and coordinates all services
SmartProxy	`@push.rocks/smartproxy`	High-performance HTTP/HTTPS and TCP/SNI proxy with route-based config (Rust engine)
UnifiedEmailServer	`@push.rocks/smartmta`	Full SMTP server with pattern-based routing, DKIM, queue management (TypeScript + Rust)
DNS Server	`@push.rocks/smartdns`	Authoritative DNS with dynamic records and DKIM TXT auto-generation (Rust engine)

Component	Package	Description
SmartAcme	`@push.rocks/smartacme`	ACME certificate management with per-domain dedup, concurrency control, and rate limiting
RADIUS Server	`@push.rocks/smartradius`	Network authentication with MAB, VLAN assignment, and accounting
RemoteIngress	`@serve.zone/remoteingress`	Distributed edge tunneling with Rust data plane and TS management
OpsServer	`@api.global/typedserver`	Web dashboard + TypedRequest API for monitoring and management
MetricsManager	`@push.rocks/smartmetrics`	Real-time metrics collection (CPU, memory, email, DNS, security)
StorageManager	built-in	Pluggable key-value storage (filesystem, custom, or in-memory)
CacheDb	`@push.rocks/smartdb`	Embedded MongoDB-compatible database (LocalSmartDb) for persistent caching

How It Works

DcRouter acts purely as an **orchestrator** — it doesn't implement protocols itself. Instead, it wires together best-in-class packages for each protocol:

1. **On `start()`**: DcRouter initializes OpsServer (default port 3000, configurable via `opsServerPort`), then spins up SmartProxy, smartmta, SmartDNS, SmartRadius, RemoteIngress, and SmartVPN based on which configs are provided. Services start in dependency order via `ServiceManager`.
2. **During operation**: Each service handles its own protocol independently. SmartProxy uses a Rust-powered engine for maximum throughput. smartmta uses a hybrid TypeScript + Rust architecture for reliable email delivery. RemoteIngress runs a Rust data plane for edge tunnel networking. SmartVPN runs a Rust data plane for WireGuard and custom transports. SmartAcme v9 handles all certificate operations with built-in concurrency control and rate limiting.
3. **On `stop()`**: All services are gracefully shut down in parallel, including cleanup of HTTP agents and DNS clients.

Rust-Powered Architecture

DcRouter itself is a pure TypeScript orchestrator, but several of its core sub-components ship with **compiled Rust binaries** for performance-critical paths. At runtime each package detects the platform, unpacks the correct binary, and communicates with TypeScript over IPC/FFI — so you get the ergonomics of TypeScript with the throughput of native code.

Component	Rust Binary	What It Handles
SmartProxy	`smartproxy-bin`	All TCP/TLS/HTTP proxy networking, NFTables integration, connection metrics
smartmta	`mailer-bin`	SMTP server + client, DKIM/SPF/DMARC, content scanning, IP reputation
SmartDNS	`smartdns-bin`	DNS server (UDP + DNS-over-HTTPS), DNSSEC, DNS client resolution
RemoteIngress	`remoteingress-bin`	Edge tunnel data plane, multiplexed streams, heartbeat management
SmartVPN	`smartvpn_daemon`	WireGuard (boringtun), Noise IK handshake, QUIC/WS transports, userspace NAT (smoltcp)
SmartRadius	—	Pure TypeScript (no Rust component)

Configuration Reference

`IDcRouterOptions`

```
interface IDcRouterOptions {
  // ── Base ──────────────────────────────────────────────────
  /** Base directory for all dcrouter data. Defaults to ~/.serve.zone/dcrouter */
  baseDir?: string;

  // ── Traffic Routing ──────────────────────────────────────
  /** SmartProxy config for HTTP/HTTPS and TCP/SNI routing */
  smartProxyConfig?: ISmartProxyOptions;

  // ── Email ────────────────────────────────────────────────
  /** Unified email server configuration (smartmta) */
  emailConfig?: IUnifiedEmailServerOptions;

  /** Custom email port mapping overrides */
  emailPortConfig?: {
    portMapping?: Record<number, number>;
    portSettings?: Record<number, any>;
```

```
  receivedEmailsPath?: string;
};

// ── DNS ──────────────────────────────────────────────────────────
/** Nameserver domains — get A records automatically */
dnsNsDomains?: string[];
/** Domains this server is authoritative for */
dnsScopes?: string[];
/** Public IP for NS A records */
publicIp?: string;
/** Ingress proxy IPs (hides real server IP) */
proxyIps?: string[];
/** Custom DNS records */
dnsRecords?: Array<{
  name: string;
  type: 'A' | 'AAAA' | 'CNAME' | 'MX' | 'TXT' | 'NS' | 'SOA';
  value: string;
  ttl?: number;
  useIngressProxy?: boolean;
}>;

// ── RADIUS ──────────────────────────────────────────────────────
/** RADIUS server for network authentication */
radiusConfig?: {
  authPort?: number;                  // default: 1812
  acctPort?: number;                  // default: 1813
  clients: IRadiusClient[];
  vlanAssignment?: IVlanManagerConfig;
  accounting?: { enabled: boolean; retentionDays?: number };
};

// ── Remote Ingress ──────────────────────────────────────────────
/** Remote Ingress hub for edge tunnel connections */
remoteIngressConfig?: {
  enabled?: boolean;                  // default: false
  tunnelPort?: number;                // default: 8443
  hubDomain?: string;                 // External hostname for connection tokens
  tls?: {
    certPath?: string;
```

```
    keyPath?: string;
  };
};

// ── VPN ──────────────────────────────────────────────────────────
/** VPN server for route-level access control */
vpnConfig?: {
  enabled?: boolean;                  // default: false
  subnet?: string;                    // default: '10.8.0.0/24'
  wgListenPort?: number;              // default: 51820
  dns?: string[];                     // DNS servers pushed to VPN clients
  serverEndpoint?: string;            // Hostname in generated client configs
  clients?: Array<{                   // Pre-defined VPN clients
    clientId: string;
    serverDefinedClientTags?: string[];
    description?: string;
  }>;
  destinationPolicy?: {               // Traffic routing policy
    default: 'forceTarget' | 'block' | 'allow';
    target?: string;                  // IP for forceTarget (default: '127.0.0.1')
    allowList?: string[];             // Pass through directly
    blockList?: string[];             // Always block (overrides allowList)
  };
};

// ── HTTP/3 (QUIC) ────────────────────────────────────────────────
/** HTTP/3 config — enabled by default on qualifying HTTPS routes */
http3?: {
  enabled?: boolean;                  // default: true
  quicSettings?: {
    maxIdleTimeout?: number;          // default: 30000ms
    maxConcurrentBidiStreams?: number;  // default: 100
    maxConcurrentUniStreams?: number;   // default: 100
    initialCongestionWindow?: number;
  };
  altSvc?: {
    port?: number;                    // default: listening port
    maxAge?: number;                  // default: 86400s
  };
```

```
  udpSettings?: {
    sessionTimeout?: number;           // default: 60000ms
    maxSessionsPerIP?: number;         // default: 1000
    maxDatagramSize?: number;          // default: 65535
  };
};

// ── OpsServer ──────────────────────────────────────────────
/** Port for the OpsServer web dashboard (default: 3000) */
opsServerPort?: number;

// ── TLS & Certificates ─────────────────────────────────────
tls?: {
  contactEmail: string;
  domain?: string;
  certPath?: string;
  keyPath?: string;
};
dnsChallenge?: { cloudflareApiKey?: string };

// ── Storage & Caching ──────────────────────────────────────
storage?: {
  fsPath?: string;
  readFunction?: (key: string) => Promise<string>;
  writeFunction?: (key: string, value: string) => Promise<void>;
};
cacheConfig?: {
  enabled?: boolean;                   // default: true
  storagePath?: string;                // default: '~/.serve.zone/dcrouter/tsmdb'
  dbName?: string;                     // default: 'dcrouter'
  cleanupIntervalHours?: number;       // default: 1
  ttlConfig?: {
    emails?: number;                   // default: 30 days
    ipReputation?: number;             // default: 1 day
    bounces?: number;                  // default: 30 days
    dkimKeys?: number;                 // default: 90 days
    suppression?: number;              // default: 30 days
  };
};
```

```
}
```

HTTP/HTTPS & TCP/SNI Routing

DcRouter uses SmartProxy for all HTTP/HTTPS and TCP/SNI routing. Routes are pattern-matched by domain, port, or both.

HTTPS with Auto-TLS

```
{
  name: 'api-gateway',
  match: { domains: ['api.example.com'], ports: [443] },
  action: {
    type: 'forward',
    targets: [{ host: '192.168.1.20', port: 8080 }],
    tls: { mode: 'terminate', certificate: 'auto' }
  }
}
```

TLS Passthrough (SNI Routing)

```
{
  name: 'secure-backend',
  match: { domains: ['secure.example.com'], ports: [8443] },
  action: {
    type: 'forward',
    targets: [{ host: '192.168.1.40', port: 8443 }],
    tls: { mode: 'passthrough' }
  }
}
```

TCP Port Range Forwarding

```
{
  name: 'database-cluster',
```

```
  match: { ports: [{ from: 5432, to: 5439 }] },
  action: {
    type: 'forward',
    targets: [{ host: '192.168.1.30', port: 'preserve' }],
    security: { ipAllowList: ['192.168.1.0/24'] }
  }
}
```

HTTP Redirect

```
{
  name: 'http-to-https',
  match: { ports: [80] },
  action: { type: 'redirect', redirect: { to: 'https://{domain}{path}' } }
}
```

HTTP/3 (QUIC) Support

DcRouter ships with **HTTP/3 enabled by default** □□. All qualifying HTTPS routes on port 443 are automatically augmented with QUIC/H3 configuration — no extra setup needed. Under the hood, SmartProxy's native HTTP/3 support (via `IRouteQuic`) handles QUIC transport, Alt-Svc advertisement, and HTTP/3 negotiation.

How It Works

When DcRouter assembles routes in `setupSmartProxy()`, it automatically augments qualifying routes with:

- `match.transport: 'all'` — listen on both TCP (HTTP/1.1 + HTTP/2) and UDP (QUIC/HTTP/3) on the same port
- `action.udp.quic` — QUIC configuration with `enableHttp3: true` and `altSvcMaxAge: 86400`

Browsers that support HTTP/3 will discover it via the `Alt-Svc` header on initial TCP responses, then upgrade to QUIC for subsequent requests.

What Gets Augmented

A route qualifies for HTTP/3 augmentation when **all** of these are true:

- Port includes **443** (single number, array, or range)
- Action type is `forward` (not `socket-handler`)
- **TLS is enabled** (passthrough, terminate, or terminate-and-reencrypt)
- Route is **not** an email route (ports 25/587/465)
- Route doesn't already have `transport: 'all'` or existing `udp.quic` config

Zero-Config (Default Behavior)

```
// HTTP/3 is ON by default — this route automatically gets QUIC/H3:
const router = new DcRouter({
  smartProxyConfig: {
    routes: [{
      name: 'web-app',
      match: { domains: ['example.com'], ports: [443] },
      action: {
        type: 'forward',
        targets: [{ host: '192.168.1.10', port: 8080 }],
        tls: { mode: 'terminate', certificate: 'auto' }
      }
    }]
  }
});
```

Per-Route Opt-Out

Disable HTTP/3 on a specific route using `action.options.http3`:

```
{
  name: 'legacy-app',
  match: { domains: ['legacy.example.com'], ports: [443] },
  action: {
    type: 'forward',
    targets: [{ host: '192.168.1.50', port: 8080 }],
    tls: { mode: 'terminate', certificate: 'auto' },
    options: { http3: false }  // ← This route stays TCP-only
  }
}
```

Global Opt-Out

Disable HTTP/3 across all routes:

```
const router = new DcRouter({
  http3: { enabled: false },
  smartProxyConfig: { routes: [/* ... */] }
});
```

Custom QUIC Settings

Fine-tune QUIC parameters globally:

```
const router = new DcRouter({
  http3: {
    quicSettings: {
      maxIdleTimeout: 60000,          // 60s idle timeout
      maxConcurrentBidiStreams: 200,   // More parallel streams
      maxConcurrentUniStreams: 50,
    },
    altSvc: {
      maxAge: 3600,                   // 1 hour Alt-Svc cache
    },
    udpSettings: {
      sessionTimeout: 120000,         // 2 min UDP session timeout
      maxSessionsPerIP: 500,
    }
  },
  smartProxyConfig: { routes: [/* ... */] }
});
```

Programmatic Routes

Routes added at runtime via the Route Management API also get HTTP/3 augmentation automatically — the `RouteConfigManager` applies the same augmentation logic when merging programmatic routes.

Email System

The email system is powered by `@push.rocks/smartmta`, a TypeScript + Rust hybrid MTA. DcRouter configures and orchestrates smartmta's **UnifiedEmailServer**, which handles SMTP sessions, route matching, delivery queuing, DKIM signing, and all email processing.

Email Domain Configuration

Domains define *infrastructure* — how DNS and DKIM are handled for each domain:

Forward Mode

Simple forwarding without local DNS management:

```
{
  domain: 'forwarded.com',
  dnsMode: 'forward',
  dns: { forward: { skipDnsValidation: true, targetDomain: 'mail.target.com' } }
}
```

Internal DNS Mode

Uses DcRouter's built-in DNS server (requires `dnsNsDomains` + `dnsScopes`):

```
{
  domain: 'mail.example.com',
  dnsMode: 'internal-dns',
  dns: { internal: { mxPriority: 10, ttl: 3600 } },
  dkim: { selector: 'mail2024', keySize: 2048, rotateKeys: true, rotationInterval: 90 }
}
```

External DNS Mode

Uses existing DNS infrastructure with validation:

```
{
  domain: 'mail.external.com',
  dnsMode: 'external-dns',
  dns: { external: { requiredRecords: ['MX', 'SPF', 'DKIM', 'DMARC'] } },
```

```
  rateLimits: {
    inbound: { messagesPerMinute: 100, connectionsPerIp: 10 },
    outbound: { messagesPerMinute: 200 }
  }
}
```

Email Route Actions

Routes define *behavior* — what happens when an email matches:

Forward □□

Routes emails to an external SMTP server:

```
{
  name: 'forward-to-internal',
  match: { recipients: '*@company.com' },
  action: {
    type: 'forward',
    forward: {
      host: 'internal-mail.company.com',
      port: 25,
      auth: { user: 'relay-user', pass: 'relay-pass' },
      addHeaders: { 'X-Forwarded-By': 'dcrouter' }
    }
  }
}
```

Process ⚙

Full MTA processing with content scanning and delivery queues:

```
{
  name: 'process-notifications',
  match: { recipients: '*@notifications.company.com' },
  action: {
    type: 'process',
    process: { scan: true, dkim: true, queue: 'priority' }
  }
}
```

Deliver □□

Local mailbox delivery:

```
{
  name: 'deliver-local',
  match: { recipients: '*@local.company.com' },
  action: { type: 'deliver' }
}
```

Reject □□

Reject with custom SMTP response code:

```
{
  name: 'reject-spam-domain',
  match: { senders: '*@spam-domain.com', sizeRange: { min: 1000000 } },
  action: {
    type: 'reject',
    reject: { code: 550, message: 'Message rejected due to policy' }
  }
}
```

Route Matching

Routes support powerful matching criteria:

```
// Recipient patterns
match: { recipients: '*@example.com' }           // All addresses at domain
match: { recipients: 'admin@*' }                 // "admin" at any domain
match: { senders: ['*@trusted.com', '*@vip.com'] } // Multiple sender patterns

// IP-based matching (CIDR)
match: { clientIp: '192.168.0.0/16' }
match: { clientIp: ['10.0.0.0/8', '172.16.0.0/12'] }

// Authentication state
match: { authenticated: true }
```

```
// Header matching
match: { headers: { 'X-Priority': 'high', 'Subject': /urgent|emergency/i } }

// Size and content
match: { sizeRange: { min: 1000, max: 5000000 }, hasAttachments: true }
match: { subject: /invoice|receipt/i }
```

Email Security Stack

- **DKIM** — Automatic key generation, signing, and rotation for all domains
- **SPF** — Sender Policy Framework verification on inbound mail
- **DMARC** — Domain-based Message Authentication verification
- **IP Reputation** — Real-time IP reputation checking with caching
- **Content Scanning** — Spam, virus, and attachment scanning
- **Rate Limiting** — Hierarchical limits (global → domain → sender)
- **Bounce Management** — Automatic bounce detection and suppression lists

Email Deliverability

- **IP Warmup Manager** — Multi-stage warmup schedules for new IPs
- **Sender Reputation Monitor** — Per-domain reputation tracking and scoring
- **Connection Pooling** — Pooled outbound SMTP connections per destination

DNS Server

DcRouter includes an authoritative DNS server built on smartdns. It handles standard UDP DNS on port 53 and DNS-over-HTTPS via SmartProxy socket handler.

Enabling DNS

DNS is activated when both `dnsNsDomains` and `dnsScopes` are configured:

```
const router = new DcRouter({
  dnsNsDomains: ['ns1.example.com', 'ns2.example.com'],
  dnsScopes: ['example.com'],
  publicIp: '203.0.113.1',
  dnsRecords: [
```

```
    { name: 'example.com', type: 'A', value: '203.0.113.1' },
    { name: 'www.example.com', type: 'CNAME', value: 'example.com' },
    { name: 'example.com', type: 'MX', value: '10:mail.example.com' },
    { name: 'example.com', type: 'TXT', value: 'v=spf1 a mx ~all' }
  ]
});
```

Automatic DNS Records

DcRouter auto-generates:

- **NS records** for all domains in `dnsScopes`
- **SOA records** for authoritative zones
- **A records** for nameserver domains (`dnsNsDomains`)
- **MX, SPF, DKIM, DMARC records** for email domains with `internal-dns` mode
- **ACME challenge records** for certificate provisioning

Ingress Proxy Support

When `proxyIps` is configured, A records with `useIngressProxy: true` (default) will use the proxy IP instead of the real server IP — hiding your origin:

```
{
  proxyIps: ['198.51.100.1', '198.51.100.2'],
  dnsRecords: [
    { name: 'example.com', type: 'A', value: '203.0.113.1' },  // Will resolve to 198.51.100.1
    { name: 'ns1.example.com', type: 'A', value: '203.0.113.1', useIngressProxy: false }  //
Stays real IP
  ]
}
```

RADIUS Server

DcRouter includes a RADIUS server for network access control, built on smartradius.

Configuration

```
const router = new DcRouter({
  radiusConfig: {
    authPort: 1812,
    acctPort: 1813,
    clients: [
      {
        name: 'core-switch',
        ipRange: '192.168.1.0/24',
        secret: 'shared-secret',
        enabled: true
      }
    ],
    vlanAssignment: {
      defaultVlan: 100,
      allowUnknownMacs: true,
      mappings: [
        { mac: 'aa:bb:cc:dd:ee:ff', vlan: 10, enabled: true },   // Exact MAC
        { mac: 'aa:bb:cc', vlan: 20, enabled: true },             // OUI prefix
      ]
    },
    accounting: {
      enabled: true,
      retentionDays: 30
    }
  }
});
```

Components

Component	Purpose
RadiusServer	Main RADIUS server handling auth + accounting requests
VlanManager	MAC-to-VLAN mapping with exact, OUI, and wildcard patterns
AccountingManager	Session tracking, traffic metering, start/stop/interim updates

OpsServer API

RADIUS is fully manageable at runtime via the OpsServer API:

- Client management (add/remove/list NAS devices)
- VLAN mapping CRUD operations
- Session monitoring and forced disconnects
- Accounting summaries and statistics

Remote Ingress

DcRouter can act as a **hub** for distributed edge nodes using `@serve.zone/remoteingress`. Edge nodes accept incoming traffic at remote locations and tunnel it back to the hub over a single multiplexed connection. This is ideal for scenarios where you need to accept traffic at multiple geographic locations but process it centrally.

Enabling Remote Ingress

```
const router = new DcRouter({
  remoteIngressConfig: {
    enabled: true,
    tunnelPort: 8443,
    hubDomain: 'hub.example.com',  // Embedded in connection tokens
  },
  // Routes tagged with remoteIngress are auto-derived to edge listen ports
  smartProxyConfig: {
    routes: [
      {
        name: 'web-via-edge',
        match: { domains: ['app.example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.10', port: 8080 }],
          tls: { mode: 'terminate', certificate: 'auto' }
        },
        remoteIngress: { enabled: true }  // Edges will listen on port 443
      }
    ]
  }
});
```

```
await router.start();
```

Edge Registration

Edges are registered via the OpsServer API (or dashboard UI). Each edge gets a unique ID and secret:

```
// Via TypedRequest API
const createReq = new TypedRequest<IReq_CreateRemoteIngress>(
  'https://hub:3000/typedrequest', 'createRemoteIngress'
);
const { edge } = await createReq.fire({
  identity,
  name: 'edge-nyc-01',
  autoDerivePorts: true,
  tags: ['us-east'],
});
// edge.secret is returned only on creation — save it!
```

Connection Tokens □□

Instead of configuring edges with four separate values (hubHost, hubPort, edgeId, secret), DcRouter can generate a single **connection token** — an opaque base64url string that encodes everything:

```
// Via TypedRequest API
const tokenReq = new TypedRequest<IReq_GetRemoteIngressConnectionToken>(
  'https://hub:3000/typedrequest', 'getRemoteIngressConnectionToken'
);
const { token } = await tokenReq.fire({ identity, edgeId: 'edge-uuid' });
// token = "eyJoIjoiaHViLmV4YW1wbGUuY29tIiwicCI6ODQ0MywiZSI6I..."

// On the edge side, just pass the token:
const edge = new RemoteIngressEdge({ token });
await edge.start();
```

The token is generated using `remoteingress.encodeConnectionToken()` and contains `{ hubHost, hubPort, edgeId, secret }`. The `hubHost` comes from `remoteIngressConfig.hubDomain` (or can be

overridden per-request).

In the OpsServer dashboard, click **"Copy Token"** on any edge row to copy the connection token to your clipboard.

Auto-Derived Ports

When routes have `remoteIngress: { enabled: true }`, edges with `autoDerivePorts: true` (default) automatically pick up those routes' ports. You can also use `edgeFilter` to restrict which edges get which ports:

```
{
  name: 'web-route',
  match: { ports: [443] },
  action: { /* ... */ },
  remoteIngress: {
    enabled: true,
    edgeFilter: ['us-east', 'edge-uuid-123']  // Only edges with matching id or tags
  }
}
```

Dashboard Actions

The OpsServer Remote Ingress view provides:

Action	Description
Create Edge Node	Register a new edge with name, ports, tags
Enable / Disable	Toggle an edge on or off
Edit	Modify name, manual ports, auto-derive setting, tags
Regenerate Secret	Issue a new secret (invalidates the old one)
Copy Token	Generate and copy a base64url connection token to clipboard
Delete	Remove the edge registration

VPN Access Control

DcRouter integrates `@push.rocks/smartvpn` to provide VPN-based route access control. VPN clients connect via standard WireGuard or native WebSocket/QUIC transports, receive an IP from a configurable subnet, and can then access routes that are restricted to VPN-only traffic.

How It Works

1. **SmartVPN daemon** runs inside dcrouter with a Rust data plane (WireGuard via `boringtun`, custom protocol via Noise IK)
2. Clients connect and get assigned an IP from the VPN subnet (e.g. `10.8.0.0/24`)
3. **Smart split tunnel** — generated WireGuard configs auto-include the VPN subnet plus DNS-resolved IPs of VPN-gated domains. Domains from routes with `vpn.enabled` are resolved at config generation time, so clients route only the necessary traffic through the tunnel
4. Routes with `vpn: { enabled: true }` get `security.ipAllowList` dynamically injected (re-computed on every client change). With `mandatory: true` (default), the allowlist is replaced; with `mandatory: false`, VPN IPs are appended to existing rules
5. When `allowedServerDefinedClientTags` is set, only matching client IPs are injected (not the whole subnet)
6. SmartProxy enforces the allowlist — only authorized VPN clients can access protected routes
7. All VPN traffic is forced through SmartProxy via userspace NAT with PROXY protocol v2 — no root required

Destination Policy

By default, VPN client traffic is redirected to localhost (SmartProxy) via `forceTarget`. You can customize this with a destination policy:

```
// Default: all traffic → SmartProxy
destinationPolicy: { default: 'forceTarget', target: '127.0.0.1' }

// Allow direct access to a backend subnet
destinationPolicy: {
  default: 'forceTarget',
  target: '127.0.0.1',
  allowList: ['192.168.190.*'],     // direct access to this subnet
  blockList: ['192.168.190.1'],     // except the gateway
}

// Block everything except specific IPs
destinationPolicy: {
```

```
  default: 'block',
  allowList: ['10.0.0.*', '192.168.1.*'],
}
```

Configuration

```
const router = new DcRouter({
  vpnConfig: {
    enabled: true,
    subnet: '10.8.0.0/24',            // VPN client IP pool (default)
    wgListenPort: 51820,               // WireGuard UDP port (default)
    serverEndpoint: 'vpn.example.com', // Hostname in generated client configs
    dns: ['1.1.1.1', '8.8.8.8'],      // DNS servers pushed to clients

    // Pre-define VPN clients with server-defined tags
    clients: [
      { clientId: 'alice-laptop', serverDefinedClientTags: ['engineering'], description: 'Dev
laptop' },
      { clientId: 'bob-phone', serverDefinedClientTags: ['engineering', 'mobile'] },
      { clientId: 'carol-desktop', serverDefinedClientTags: ['finance'] },
    ],

    // Optional: customize destination policy (default: forceTarget → localhost)
    // destinationPolicy: { default: 'forceTarget', target: '127.0.0.1', allowList:
['192.168.1.*'] },
  },
  smartProxyConfig: {
    routes: [
      // □□VPN-only: any VPN client can access
      {
        name: 'internal-app',
        match: { domains: ['internal.example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.50', port: 8080 }],
          tls: { mode: 'terminate', certificate: 'auto' },
        },
        vpn: { enabled: true },
```

```
      },
      // □□VPN + tag-restricted: only 'engineering' tagged clients
      {
        name: 'eng-dashboard',
        match: { domains: ['eng.example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.51', port: 8080 }],
          tls: { mode: 'terminate', certificate: 'auto' },
        },
        vpn: { enabled: true, allowedServerDefinedClientTags: ['engineering'] },
        // → alice + bob can access, carol cannot
      },
      // □□Public: no VPN
      {
        name: 'public-site',
        match: { domains: ['example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.10', port: 80 }],
          tls: { mode: 'terminate', certificate: 'auto' },
        },
      },
    ],
  },
});
```

Client Tags

SmartVPN distinguishes between two types of client tags:

Tag Type	Set By	Purpose
`serverDefinedClientTags`	Admin (via config or API)	**Trusted** — used for route access control
`clientDefinedClientTags`	Connecting client	**Informational** — displayed in dashboard, never used for security

Routes with `allowedServerDefinedClientTags` only permit VPN clients whose admin-assigned tags match. Clients cannot influence their own server-defined tags.

Client Management via OpsServer

The OpsServer dashboard and API provide full VPN client lifecycle management:

- **Create client** — generates WireGuard keypairs, assigns IP, returns a ready-to-use `.conf` file
- **QR code** — scan with the WireGuard mobile app (iOS/Android) for instant setup
- **Enable / Disable** — toggle client access without deleting
- **Rotate keys** — generate fresh keypairs (invalidates old ones)
- **Export config** — download in WireGuard (`.conf`), SmartVPN (`.json`), or scan as QR code
- **Telemetry** — per-client bytes sent/received, keepalives, rate limiting
- **Delete** — remove a client and revoke access

Standard WireGuard clients on any platform (iOS, Android, macOS, Windows, Linux) can connect using the generated `.conf` file or by scanning the QR code — no custom VPN software needed.

Certificate Management

DcRouter uses `@push.rocks/smartacme` v9 for ACME certificate provisioning. smartacme v9 brings significant improvements over previous versions:

How It Works

When a `dnsChallenge` is configured (e.g. with a Cloudflare API key), DcRouter creates a SmartAcme instance that handles DNS-01 challenges for automatic certificate provisioning. SmartProxy calls the `certProvisionFunction` whenever a route needs a TLS certificate, and SmartAcme takes care of the rest.

```
const router = new DcRouter({
  smartProxyConfig: {
    routes: [
      {
        name: 'secure-app',
        match: { domains: ['app.example.com'], ports: [443] },
        action: {
          type: 'forward',
          targets: [{ host: '192.168.1.10', port: 8080 }],
          tls: { mode: 'terminate', certificate: 'auto' }  // ← triggers ACME provisioning
        }
```

```
      }
    ],
    acme: { email: 'admin@example.com', enabled: true, useProduction: true }
  },
  tls: { contactEmail: 'admin@example.com' },
  dnsChallenge: { cloudflareApiKey: process.env.CLOUDFLARE_API_KEY }
});
```

smartacme v9 Features

Feature	Description
Per-domain deduplication	Concurrent requests for the same domain share a single ACME operation
Global concurrency cap	Default 5 parallel ACME operations to prevent overload
Account rate limiting	Sliding window (250 orders / 3 hours) to stay within ACME provider limits
Structured errors	`AcmeError` with `isRetryable`, `isRateLimited`, `retryAfter` fields
Clean shutdown	`stop()` properly destroys HTTP agents and DNS clients

Per-Domain Backoff

DcRouter's `CertProvisionScheduler` adds **per-domain exponential backoff** on top of smartacme's built-in protections. If a DNS-01 challenge fails for a domain:

1. The failure is recorded (persisted to storage)
2. The domain enters backoff: `min(failures² × 1 hour, 24 hours)`
3. Subsequent requests for that domain are rejected until the backoff expires
4. On success, the backoff is cleared

This prevents hammering ACME servers for domains with persistent issues (e.g. missing DNS delegation).

Fallback to HTTP-01

If DNS-01 fails, the `certProvisionFunction` returns `'http01'` to tell SmartProxy to fall back to HTTP-01 challenge validation. This provides a safety net for domains where DNS-01 isn't viable.

Certificate Storage

Certificates are persisted via the `StorageBackedCertManager` which uses DcRouter's `StorageManager`. This means certs survive restarts and don't need to be re-provisioned unless they expire.

Dashboard

The OpsServer includes a **Certificates** view showing:

- All domains with their certificate status (valid, expiring, expired, failed)
- Certificate source (ACME, provision function, static)
- Expiry dates and issuer information
- Backoff status for failed domains
- One-click reprovisioning per domain
- Certificate import and export

Storage & Caching

StorageManager

Provides a unified key-value interface with three backends:

```
// Filesystem backend
storage: { fsPath: '/var/lib/dcrouter/data' }

// Custom backend (Redis, S3, etc.)
storage: {
  readFunction: async (key) => await redis.get(key),
  writeFunction: async (key, value) => await redis.set(key, value)
}

// In-memory (development only — data lost on restart)
// Simply omit the storage config
```

Used for: TLS certificates, DKIM keys, email routes, bounce/suppression lists, IP reputation data, domain configs, cert backoff state, remote ingress edge registrations.

Cache Database

An embedded MongoDB-compatible database (via smartdata + smartdb) for persistent caching with automatic TTL cleanup:

```
cacheConfig: {
  enabled: true,
  storagePath: '~/.serve.zone/dcrouter/tsmdb',
  dbName: 'dcrouter',
  cleanupIntervalHours: 1,
  ttlConfig: {
    emails: 30,         // days
    ipReputation: 1,    // days
    bounces: 30,        // days
    dkimKeys: 90,       // days
    suppression: 30     // days
  }
}
```

Cached document types: `CachedEmail`, `CachedIPReputation`.

Security Features

IP Reputation Checking

Automatic IP reputation checks on inbound connections with configurable caching:

```
// IP reputation is checked automatically for inbound SMTP connections.
// Results are cached according to cacheConfig.ttlConfig.ipReputation.
```

Rate Limiting

Hierarchical rate limits with three levels of specificity:

```
// Global defaults (via emailConfig.defaults.rateLimits)
defaults: {
  rateLimits: {
```

```
    inbound: { messagesPerMinute: 50, connectionsPerIp: 5, recipientsPerMessage: 50 },
    outbound: { messagesPerMinute: 100 }
  }
}

// Per-domain overrides (in domain config)
{
  domain: 'high-volume.com',
  rateLimits: {
    outbound: { messagesPerMinute: 500 }  // Override for this domain
  }
}
```

Precedence: Domain-specific > Pattern-specific > Global

Content Scanning

```
action: {
  type: 'process',
  options: {
    contentScanning: true,
    scanners: [
      { type: 'spam', threshold: 5.0, action: 'tag' },
      { type: 'virus', action: 'reject' },
      { type: 'attachment', blockedExtensions: ['.exe', '.bat', '.scr'], action: 'reject' }
    ]
  }
}
```

OpsServer Dashboard

The OpsServer provides a web-based management interface served on port 3000 by default (configurable via `opsServerPort`). It's built with modern web components using @design.estate/dees-catalog.

Dashboard Views

View	Description
◫ **Overview**	Real-time server stats, CPU/memory, connection counts, email throughput
◫ **Network**	Active connections, top IPs, throughput rates, SmartProxy metrics
◫ **Email**	Queue monitoring (queued/sent/failed), bounce records, security incidents
◫ **Routes**	Merged route list (hardcoded + programmatic), create/edit/toggle/override routes
◫ **API Tokens**	Token management with scopes, create/revoke/roll/toggle
◫ **Certificates**	Domain-centric certificate overview, status, backoff info, reprovisioning, import/export
◫ **RemoteIngress**	Edge node management, connection status, token generation, enable/disable
◫ **VPN**	VPN client management, server status, create/toggle/export/rotate/delete clients
◫ **RADIUS**	NAS client management, VLAN mappings, session monitoring, accounting
◫ **Logs**	Real-time log viewer with level filtering and search
⚙ **Configuration**	Read-only view of current system configuration
◫ **Security**	IP reputation, rate limit status, blocked connections

API Endpoints

All management is done via TypedRequest over HTTP POST to `/typedrequest`:

```
// Authentication
'adminLoginWithUsernameAndPassword'   // Login with credentials → returns JWT identity
'verifyIdentity'                       // Verify JWT token validity
'adminLogout'                          // End admin session

// Statistics & Health
'getServerStatistics'                  // Uptime, CPU, memory, connections
'getHealthStatus'                      // System health check
'getCombinedMetrics'                   // All metrics in one call

// Email Operations
'getAllEmails'                          // List all emails (queued/sent/failed)
```

```
'getEmailDetail'                      // Full detail for a specific email
'resendEmail'                         // Re-queue a failed email

// Certificates
'getCertificateOverview'              // Domain-centric certificate status
'reprovisionCertificate'              // Reprovision by route name (legacy)
'reprovisionCertificateDomain'        // Reprovision by domain (preferred)
'importCertificate'                   // Import a certificate
'exportCertificate'                   // Export a certificate
'deleteCertificate'                   // Delete a certificate

// Remote Ingress
'getRemoteIngresses'                  // List all edge registrations
'createRemoteIngress'                 // Register a new edge
'updateRemoteIngress'                 // Update edge settings
'deleteRemoteIngress'                 // Remove an edge
'regenerateRemoteIngressSecret'       // Issue a new secret
'getRemoteIngressStatus'              // Runtime status of all edges
'getRemoteIngressConnectionToken'     // Generate a connection token for an edge

// Route Management (JWT or API token auth)
'getMergedRoutes'                     // List all routes (hardcoded + programmatic)
'createRoute'                         // Create a new programmatic route
'updateRoute'                         // Update a programmatic route
'deleteRoute'                         // Delete a programmatic route
'toggleRoute'                         // Enable/disable a programmatic route
'setRouteOverride'                    // Override a hardcoded route
'removeRouteOverride'                 // Remove a hardcoded route override

// API Token Management (admin JWT only)
'createApiToken'                      // Create API token → returns raw value once
'listApiTokens'                       // List all tokens (without secrets)
'revokeApiToken'                      // Delete an API token
'rollApiToken'                        // Regenerate token secret
'toggleApiToken'                      // Enable/disable a token

// Configuration (read-only)
'getConfiguration'                    // Current system config
```

```
// Logs
'getRecentLogs'                    // Retrieve system logs with filtering
'getLogStream'                     // Stream live logs

// VPN
'getVpnClients'                    // List all registered VPN clients
'getVpnStatus'                     // VPN server status (running, subnet, port, keys)
'createVpnClient'                  // Create client → returns WireGuard config (shown
once)
'deleteVpnClient'                  // Remove a VPN client
'enableVpnClient'                  // Enable a disabled client
'disableVpnClient'                 // Disable a client
'rotateVpnClientKey'               // Generate new keys (invalidates old ones)
'exportVpnClientConfig'            // Export WireGuard (.conf) or SmartVPN (.json) config
'getVpnClientTelemetry'            // Per-client bytes sent/received, keepalives

// RADIUS
'getRadiusSessions'                // Active RADIUS sessions
'getRadiusClients'                 // List NAS clients
'getRadiusStatistics'              // RADIUS stats
'setRadiusClient'                  // Add/update NAS client
'removeRadiusClient'               // Remove NAS client
'getVlanMappings'                  // List VLAN mappings
'setVlanMapping'                   // Add/update VLAN mapping
'removeVlanMapping'                // Remove VLAN mapping
'testVlanAssignment'               // Test what VLAN a MAC gets
```

API Client

DcRouter ships with a typed, object-oriented API client for programmatic management of a running instance. Install it separately or import from the main package:

```
pnpm add @serve.zone/dcrouter-apiclient
# or import from the main package:
# import { DcRouterApiClient } from '@serve.zone/dcrouter/apiclient';
```

Quick Example

```
import { DcRouterApiClient } from '@serve.zone/dcrouter/apiclient';

const client = new DcRouterApiClient({ baseUrl: 'https://dcrouter.example.com' });
await client.login('admin', 'password');

// OO resource instances with methods
const { routes } = await client.routes.list();
await routes[0].toggle(false);

// Builder pattern for creation
const newRoute = await client.routes.build()
  .setName('api-gateway')
  .setMatch({ ports: 443, domains: ['api.example.com'] })
  .setAction({ type: 'forward', targets: [{ host: 'backend', port: 8080 }] })
  .setTls({ mode: 'terminate', certificate: 'auto' })
  .save();

// Manage certificates
const { certificates, summary } = await client.certificates.list();
await certificates[0].reprovision();

// Create API tokens with builder
const token = await client.apiTokens.build()
  .setName('ci-token')
  .setScopes(['routes:read', 'routes:write'])
  .setExpiresInDays(90)
  .save();
console.log(token.tokenValue); // only available at creation

// Remote ingress edges
const edge = await client.remoteIngress.build()
  .setName('edge-nyc-01')
  .setListenPorts([80, 443])
  .save();
const connToken = await edge.getConnectionToken();

// Read-only managers
const health = await client.stats.getHealth();
const config = await client.config.get();
```

```
const { logs } = await client.logs.getRecent({ level: 'error', limit: 50 });
```

Resource Managers

Manager	Operations
`client.routes`	`list()`, `create()`, `build()` → Route: `update()`, `delete()`, `toggle()`, `setOverride()`, `removeOverride()`
`client.certificates`	`list()`, `import()` → Certificate: `reprovision()`, `delete()`, `export()`
`client.apiTokens`	`list()`, `create()`, `build()` → ApiToken: `revoke()`, `roll()`, `toggle()`
`client.remoteIngress`	`list()`, `getStatuses()`, `create()`, `build()` → RemoteIngress: `update()`, `delete()`, `regenerateSecret()`, `getConnectionToken()`
`client.stats`	`getServer()`, `getEmail()`, `getDns()`, `getSecurity()`, `getConnections()`, `getQueues()`, `getHealth()`, `getNetwork()`, `getCombined()`
`client.config`	`get(section?)`
`client.logs`	`getRecent()`, `getStream()`
`client.emails`	`list()` → Email: `getDetail()`, `resend()`
`client.radius`	`.clients`, `.vlans`, `.sessions` sub-managers + `getStatistics()`, `getAccountingSummary()`

See the full API client documentation for detailed usage of every manager, builder, and resource class.

API Reference

DcRouter Class

```
import { DcRouter } from '@serve.zone/dcrouter';

const router = new DcRouter(options: IDcRouterOptions);
```

Methods

Method	Description

`start(): Promise<void>`	Start all configured services
`stop(): Promise<void>`	Gracefully stop all services
`updateSmartProxyConfig(config): Promise<void>`	Hot-update SmartProxy routes
`updateEmailConfig(config): Promise<void>`	Hot-update email configuration
`updateEmailRoutes(routes): Promise<void>`	Update email routing rules at runtime
`updateRadiusConfig(config): Promise<void>`	Hot-update RADIUS configuration
`getStats(): any`	Get real-time statistics from all services

Properties

Property	Type	Description
`options`	`IDcRouterOptions`	Current configuration
`smartProxy`	`SmartProxy`	SmartProxy instance
`smartAcme`	`SmartAcme`	SmartAcme v9 certificate manager instance
`emailServer`	`UnifiedEmailServer`	Email server instance (from smartmta)
`dnsServer`	`DnsServer`	DNS server instance
`radiusServer`	`RadiusServer`	RADIUS server instance
`remoteIngressManager`	`RemoteIngressManager`	Edge registration CRUD manager
`tunnelManager`	`TunnelManager`	Tunnel lifecycle and status manager
`vpnManager`	`VpnManager`	VPN server lifecycle and client CRUD manager
`storageManager`	`StorageManager`	Storage backend
`opsServer`	`OpsServer`	OpsServer/dashboard instance
`metricsManager`	`MetricsManager`	Metrics collector
`cacheDb`	`CacheDb`	Cache database instance
`certProvisionScheduler`	`CertProvisionScheduler`	Per-domain backoff scheduler for cert provisioning
`certificateStatusMap`	`Map<string, ...>`	Domain-keyed certificate status from SmartProxy events

Re-exported Types

DcRouter re-exports key types for convenience:

```
import {
  DcRouter,
  IDcRouterOptions,
  UnifiedEmailServer,
  type IUnifiedEmailServerOptions,
  type IEmailRoute,
  type IEmailDomainConfig,
  type IHttp3Config,
} from '@serve.zone/dcrouter';
```

Sub-Modules

DcRouter is published as a monorepo with separately-installable interface and web packages:

Package	Description	Install
`@serve.zone/dcrouter`	Main package — the full router	`pnpm add @serve.zone/dcrouter`
`@serve.zone/dcrouter-interfaces`	TypedRequest interfaces for the OpsServer API	`pnpm add @serve.zone/dcrouter-interfaces`
`@serve.zone/dcrouter-apiclient`	OO API client with builder pattern	`pnpm add @serve.zone/dcrouter-apiclient`
`@serve.zone/dcrouter-web`	Web dashboard components	`pnpm add @serve.zone/dcrouter-web`

You can also import directly from the main package:

```
import { data, requests } from '@serve.zone/dcrouter/interfaces';
import { DcRouterApiClient } from '@serve.zone/dcrouter/apiclient';
```

Testing

DcRouter includes a comprehensive test suite covering all system components:

```
# Run all tests
pnpm test

# Run a specific test file
tstest test/test.jwt-auth.ts --verbose
```

```
# Run with extended timeout
tstest test/test.opsserver-api.ts --verbose --timeout 60
```

Test Coverage

Test File	Area	Tests
`test.apiclient.ts`	API client instantiation, builders, resource hydration, exports	18
`test.contentscanner.ts`	Content scanning (spam, phishing, malware, attachments)	13
`test.dcrouter.email.ts`	Email config, domain and route setup	4
`test.dns-server-config.ts`	DNS record parsing, grouping, extraction	5
`test.dns-socket-handler.ts`	DNS socket handler and route generation	6
`test.errors.ts`	Error classes, handler, retry utilities	5
`test.http3-augmentation.ts`	HTTP/3 route augmentation, qualification, opt-in/out, QUIC settings	20
`test.ipreputationchecker.ts`	IP reputation, DNSBL, caching, risk classification	10
`test.jwt-auth.ts`	JWT login, verification, logout, invalid credentials	8
`test.opsserver-api.ts`	Health, statistics, configuration, log APIs	8
`test.protected-endpoint.ts`	Admin auth, identity verification, public endpoints	8
`test.storagemanager.ts`	Memory, filesystem, custom backends, concurrency	8

Docker / OCI Container Deployment

DcRouter ships with a production-ready `Dockerfile` and supports environment-variable-driven configuration for OCI container deployments. The container image includes tini as PID 1 (via the base image), proper health checks, and configurable resource limits. When

`DCROUTER_MODE=OCI_CONTAINER` is set, DcRouter automatically reads configuration from environment variables (and optionally from a JSON config file).

Running with Docker

```
docker run -d \
  --ulimit nofile=65536:65536 \
  -e DCROUTER_TLS_EMAIL=admin@example.com \
  -e DCROUTER_PUBLIC_IP=203.0.113.1 \
  -e DCROUTER_DNS_NS_DOMAINS=ns1.example.com,ns2.example.com \
  -e DCROUTER_DNS_SCOPES=example.com \
  -p 80:80 -p 443:443 -p 25:25 -p 587:587 -p 465:465 \
  -p 53:53/udp -p 3000:3000 -p 8443:8443 \
  code.foss.global/serve.zone/dcrouter:latest
```

> ⚡ **Production tip:** Always set `--ulimit nofile=65536:65536` for production deployments. DcRouter will log a warning at startup if the file descriptor limit is below 65536.

Environment Variables

Variable	Description	Default	Example
`DCROUTER_MODE`	Container mode (set automatically in image)	`OCI_CONTAINER`	—
`DCROUTER_CONFIG_PATH`	Path to JSON config file (env vars override)	—	`/config/dcrouter.json`
`DCROUTER_BASE_DIR`	Base data directory	`~/.serve.zone/dcrouter`	`/data/dcrouter`
`DCROUTER_TLS_EMAIL`	ACME contact email	—	`admin@example.com`
`DCROUTER_TLS_DOMAIN`	Primary TLS domain	—	`example.com`
`DCROUTER_PUBLIC_IP`	Public IP for DNS records	—	`203.0.113.1`
`DCROUTER_PROXY_IPS`	Comma-separated ingress proxy IPs	—	`198.51.100.1,198.51.100.2`
`DCROUTER_DNS_NS_DOMAINS`	Comma-separated nameserver domains	—	`ns1.example.com,ns2.example.com`
`DCROUTER_DNS_SCOPES`	Comma-separated authoritative domains	—	`example.com,other.com`

Variable	Description	Default	Example
`DCROUTER_EMAIL_HOSTNAME`	SMTP server hostname	—	`mail.example.com`
`DCROUTER_EMAIL_PORTS`	Comma-separated email ports	—	`25,587,465`
`DCROUTER_CACHE_ENABLED`	Enable/disable cache database	`true`	`false`
`DCROUTER_HEAP_SIZE`	Node.js V8 heap size in MB	`512`	`1024`
`DCROUTER_MAX_CONNECTIONS`	Global max concurrent connections	`50000`	`100000`
`DCROUTER_MAX_CONNECTIONS_PER_IP`	Max connections per source IP	`100`	`200`
`DCROUTER_CONNECTION_RATE_LIMIT`	Max new connections/min per IP	`600`	`1200`

Exposed Ports

The container exposes all service ports:

Port(s)	Protocol	Service
80, 443	TCP	HTTP/HTTPS (SmartProxy)
25, 587, 465	TCP	SMTP, Submission, SMTPS
53	TCP/UDP	DNS
1812, 1813	UDP	RADIUS auth/acct
3000	TCP	OpsServer dashboard
8443	TCP	Remote ingress tunnels
51820	UDP	WireGuard VPN
29000–30000	TCP	Dynamic port range

Building the Image

```
pnpm run build:docker    # Build the container image
pnpm run release:docker  # Push to registry
```

The Docker build supports multi-platform (`linux/amd64`, `linux/arm64`) via tsdocker.

License and Legal Information

This repository contains open-source code licensed under the MIT License. A copy of the license can be found in the license file.

Please note: The MIT License does not grant permission to use the trade names, trademarks, service marks, or product names of the project, except as required for reasonable and customary use in describing the origin of the work and reproducing the content of the NOTICE file.

Trademarks

This project is owned and maintained by Task Venture Capital GmbH. The names and logos associated with Task Venture Capital GmbH and any related products or services are trademarks of Task Venture Capital GmbH or third parties, and are not included within the scope of the MIT license granted herein.

Use of these trademarks must comply with Task Venture Capital GmbH's Trademark Guidelines or the guidelines of the respective third-party owners, and any usage must be approved in writing. Third-party trademarks used herein are the property of their respective owners and used only in a descriptive manner, e.g. for an implementation of an API or similar.

Company Information

Task Venture Capital GmbH
Registered at District Court Bremen HRB 35230 HB, Germany

For any legal inquiries or further information, please contact us via email at hello@task.vc.

By using this repository, you acknowledge that you have read this section, agree to comply with its terms, and understand that the licensing of the code does not imply endorsement by Task Venture Capital GmbH of any derivative works.

Revision #4
Created 2026-03-28 11:14:32 UTC by foss.global Team
Updated 2026-03-31 14:22:44 UTC by foss.global Team