readme.md for @push.rocks/smartproxy

A high-performance, Rust-powered proxy toolkit for Node.js — unified route-based configuration for SSL/TLS termination, HTTP/HTTPS reverse proxying, WebSocket support, UDP/QUIC/HTTP3, load balancing, custom protocol handlers, and kernel-level NFTables forwarding via `@push.rocks/smartnftables`.

󠀠󠀠 Installation

```
npm install @push.rocks/smartproxy
# or
pnpm add @push.rocks/smartproxy
```

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.

What is SmartProxy?

SmartProxy is a production-ready proxy solution that takes the complexity out of traffic management. Under the hood, all networking — TCP, UDP, TLS, HTTP reverse proxy, QUIC/HTTP3, connection tracking, security enforcement, and NFTables — is handled by a **Rust engine** for maximum performance, while you configure everything through a clean TypeScript API with full type safety.

Whether you're building microservices, deploying edge infrastructure, proxying UDP-based protocols, or need a battle-tested reverse proxy with automatic Let's Encrypt certificates,

SmartProxy has you covered.

⚡ Key Features

Feature	Description
Rust-Powered Engine	All networking handled by a high-performance Rust binary via IPC
Unified Route-Based Config	Clean match/action patterns for intuitive traffic routing
Automatic SSL/TLS	Zero-config HTTPS with Let's Encrypt ACME integration
Flexible Matching	Route by port, domain, path, protocol, client IP, TLS version, headers, or custom logic
High-Performance	Choose between user-space or kernel-level (NFTables) forwarding
UDP & QUIC/HTTP3	First-class UDP transport, datagram handlers, QUIC tunneling, and HTTP/3 support
⚖ **Load Balancing**	Round-robin, least-connections, IP-hash with health checks
Enterprise Security	IP filtering, rate limiting, basic auth, JWT auth, connection limits
WebSocket Support	First-class WebSocket proxying with ping/pong keep-alive
Custom Protocols	Socket and datagram handlers for implementing any protocol in TypeScript
Live Metrics	Real-time throughput, connection counts, UDP sessions, and performance data
Dynamic Management	Add/remove ports and routes at runtime without restarts
PROXY Protocol	Full PROXY protocol v1/v2 support for preserving client information
Consumer Cert Storage	Bring your own persistence — SmartProxy never writes certs to disk

Quick Start

Get up and running in 30 seconds:

```
import { SmartProxy, SocketHandlers } from '@push.rocks/smartproxy';


// Create a proxy with automatic HTTPS
```

```
const proxy = new SmartProxy({
  acme: {
    email: 'ssl@yourdomain.com',
    useProduction: true
  },
  routes: [
    // HTTPS route with automatic Let's Encrypt cert
    {
      name: 'https-app',
      match: { ports: 443, domains: 'app.example.com' },
      action: {
        type: 'forward',
        targets: [{ host: 'localhost', port: 3000 }],
        tls: { mode: 'terminate', certificate: 'auto' }
      }
    },
    // HTTP → HTTPS redirect
    {
      name: 'http-redirect',
      match: { ports: 80, domains: 'app.example.com' },
      action: {
        type: 'socket-handler',
        socketHandler: SocketHandlers.httpRedirect('https://{domain}:443{path}', 301)
      }
    }
  ]
});

await proxy.start();
console.log('Proxy running with automatic HTTPS!');
```

Core Concepts

Route-Based Architecture

SmartProxy uses a powerful **match/action** pattern that makes routing predictable and maintainable:

```
{
  name: 'api-route',
  match: {
    ports: 443,
    domains: 'api.example.com',
    path: '/v1/*'
  },
  action: {
    type: 'forward',
    targets: [{ host: 'backend', port: 8080 }],
    tls: { mode: 'terminate', certificate: 'auto' }
  }
}
```

Every route consists of:

- **Match** — What traffic to capture (ports, domains, paths, transport, protocol, IPs, headers)
- **Action** — What to do with it (`forward` or `socket-handler`)
- **Security** (optional) — IP allow/block lists, rate limits, authentication
- **Headers** (optional) — Request/response header manipulation with template variables
- **Name/Priority** (optional) — For identification and ordering

TLS Modes

SmartProxy supports three TLS handling modes:

Mode	Description	Use Case
`passthrough`	Forward encrypted traffic as-is (SNI-based routing)	Backend handles TLS
`terminate`	Decrypt at proxy, forward plain HTTP to backend	Standard reverse proxy
`terminate-and-reencrypt`	Decrypt at proxy, re-encrypt to backend. HTTP traffic gets full per-request routing (Host header, path matching) via the HTTP proxy; non-HTTP traffic uses a raw TLS-to-TLS tunnel	Zero-trust / defense-in-depth environments

Common Use Cases

↪️ HTTP to HTTPS Redirect

```
import { SmartProxy, SocketHandlers } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [{
    name: 'http-to-https',
    match: { ports: 80, domains: ['example.com', '*.example.com'] },
    action: {
      type: 'socket-handler',
      socketHandler: SocketHandlers.httpRedirect('https://{domain}:443{path}', 301)
    }
  }]
});
```

⚖️ Load Balancer with Health Checks

```
import { SmartProxy } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [{
    name: 'load-balancer',
    match: { ports: 443, domains: 'app.example.com' },
    action: {
      type: 'forward',
      targets: [
        { host: 'server1.internal', port: 8080 },
        { host: 'server2.internal', port: 8080 },
        { host: 'server3.internal', port: 8080 }
      ],
      tls: { mode: 'terminate', certificate: 'auto' },
      loadBalancing: {
        algorithm: 'round-robin',
        healthCheck: {
          path: '/health',
          interval: 30000,
          timeout: 5000,
          unhealthyThreshold: 3,
```

```
          healthyThreshold: 2
        }
      }
    }
  }]
});
```

🔌 WebSocket Proxy

```
import { SmartProxy } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [{
    name: 'websocket',
    match: { ports: 443, domains: 'ws.example.com', path: '/socket' },
    priority: 100,
    action: {
      type: 'forward',
      targets: [{ host: 'websocket-server', port: 8080 }],
      tls: { mode: 'terminate', certificate: 'auto' },
      websocket: {
        enabled: true,
        pingInterval: 30000,
        pingTimeout: 10000
      }
    }
  }]
});
```

🚦 API Gateway with Rate Limiting

```
import { SmartProxy } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [{
    name: 'api-gateway',
    match: { ports: 443, domains: 'api.example.com', path: '/api/*' },
```

```
    priority: 100,
    action: {
      type: 'forward',
      targets: [{ host: 'api-backend', port: 8080 }],
      tls: { mode: 'terminate', certificate: 'auto' }
    },
    headers: {
      response: {
        'Access-Control-Allow-Origin': '*',
        'Access-Control-Allow-Methods': 'GET, POST, PUT, DELETE, OPTIONS',
        'Access-Control-Allow-Headers': 'Content-Type, Authorization',
        'Access-Control-Max-Age': '86400'
      }
    },
    security: {
      rateLimit: {
        enabled: true,
        maxRequests: 100,
        window: 60,
        keyBy: 'ip'
      }
    }
  }]
});
```

🔌 Custom Protocol Handler (TCP)

SmartProxy lets you implement any protocol with full socket control. Routes with JavaScript socket handlers are automatically relayed from the Rust engine back to your TypeScript code:

```
import { SmartProxy, SocketHandlers } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [
    // Use pre-built handlers
    {
      name: 'echo-server',
      match: { ports: 7777, domains: 'echo.example.com' },
      action: { type: 'socket-handler', socketHandler: SocketHandlers.echo }
```

```
    },
    // Or create your own custom protocol
    {
      name: 'custom-protocol',
      match: { ports: 9999, domains: 'custom.example.com' },
      action: {
        type: 'socket-handler',
        socketHandler: async (socket) => {
          console.log(`New connection on custom protocol`);
          socket.write('Welcome to my custom protocol!\n');

          socket.on('data', (data) => {
            const command = data.toString().trim();
            switch (command) {
              case 'PING': socket.write('PONG\n'); break;
              case 'TIME': socket.write(`${new Date().toISOString()}\n`); break;
              case 'QUIT': socket.end('Goodbye!\n'); break;
              default: socket.write(`Unknown: ${command}\n`);
            }
          });
        }
      }
    }
  ]
});
```

Pre-built Socket Handlers:

Handler	Description
`SocketHandlers.echo`	Echo server — returns everything sent
`SocketHandlers.proxy(host, port)`	TCP proxy to another server
`SocketHandlers.lineProtocol(handler)`	Line-based text protocol
`SocketHandlers.httpResponse(code, body)`	Simple HTTP response
`SocketHandlers.httpRedirect(url, code)`	HTTP redirect with template variables (`{domain}`, `{path}`, `{port}`, `{clientIp}`)
`SocketHandlers.httpServer(handler)`	Full HTTP request/response handling
`SocketHandlers.httpBlock(status, message)`	HTTP block response
`SocketHandlers.block(message)`	Block with optional message

UDP Datagram Handler

Handle raw UDP datagrams with custom TypeScript logic — perfect for DNS, game servers, IoT protocols, or any UDP-based service:

```
import { SmartProxy } from '@push.rocks/smartproxy';
import type { IRouteConfig, TDatagramHandler, IDatagramInfo } from '@push.rocks/smartproxy';

// Custom UDP echo handler
const udpHandler: TDatagramHandler = (datagram, info, reply) => {
  console.log(`UDP from ${info.sourceIp}:${info.sourcePort} on port ${info.destPort}`);
  reply(datagram); // Echo it back
};

const proxy = new SmartProxy({
  routes: [{
    name: 'udp-echo',
    match: {
      ports: 5353,
      transport: 'udp'        // Listen for UDP datagrams
    },
    action: {
      type: 'socket-handler',
      datagramHandler: udpHandler,  // Process each datagram
      udp: {
        sessionTimeout: 60000,     // Session idle timeout (ms)
        maxSessionsPerIP: 100,
        maxDatagramSize: 65535
      }
    }
  }]
});

await proxy.start();
```

QUIC / HTTP3 Forwarding

Forward QUIC traffic to backends with optional protocol translation (e.g., receive QUIC, forward as TCP/HTTP1):

```
import { SmartProxy } from '@push.rocks/smartproxy';
import type { IRouteConfig } from '@push.rocks/smartproxy';

const quicRoute: IRouteConfig = {
  name: 'quic-to-backend',
  match: {
    ports: 443,
    transport: 'udp',
    protocol: 'quic'           // □□Match QUIC protocol
  },
  action: {
    type: 'forward',
    targets: [{
      host: 'backend-server',
      port: 8443,
      backendTransport: 'tcp'  // □□Translate QUIC → TCP for backend
    }],
    tls: {
      mode: 'terminate',
      certificate: 'auto'     // □□QUIC requires TLS 1.3
    },
    udp: {
      quic: {
        enableHttp3: true,
        maxIdleTimeout: 30000,
        maxConcurrentBidiStreams: 100,
        altSvcPort: 443,       // Advertise in Alt-Svc header
        altSvcMaxAge: 86400
      }
    }
  }
};

const proxy = new SmartProxy({
  acme: { email: 'ssl@example.com' },
  routes: [quicRoute]
});
```

�� Best-Effort Backend Protocol (H3 > H2 > H1)

SmartProxy automatically uses the **highest protocol your backend supports** for HTTP requests. The backend protocol is independent of the client protocol — a client using HTTP/1.1 can be forwarded over HTTP/3 to the backend, and vice versa.

```
const route: IRouteConfig = {
  name: 'auto-protocol',
  match: { ports: 443, domains: 'app.example.com' },
  action: {
    type: 'forward',
    targets: [{ host: 'backend', port: 8443 }],
    tls: { mode: 'terminate', certificate: 'auto' },
    options: {
      backendProtocol: 'auto'  // ��Default — best-effort selection
    }
  }
};
```

How protocol discovery works (browser model):

1. First request → TLS ALPN probe detects H2 or H1
2. Backend response inspected for `Alt-Svc: h3=":port"` header
3. If H3 advertised → cached and used for subsequent requests via QUIC
4. Graceful fallback: H3 failure → H2 → H1 with automatic cache invalidation

`backendProtocol`	Behavior
`'auto'` (default)	Best-effort: H3 > H2 > H1 with Alt-Svc discovery
`'http1'`	Always HTTP/1.1
`'http2'`	Always HTTP/2 (hard-fail if unsupported)
`'http3'`	Always HTTP/3 via QUIC (hard-fail if unsupported)

> **Note:** WebSocket upgrades always use HTTP/1.1 to the backend regardless of `backendProtocol`, since there's no performance benefit from H2/H3 Extended CONNECT for tunneled connections, and backend support is rare.

🔀 Dual-Stack TCP + UDP Route

Listen on both TCP and UDP with a single route — handle each transport with its own handler:

```
const dualStackRoute: IRouteConfig = {
  name: 'dual-stack-dns',
  match: {
    ports: 53,
    transport: 'all'  // 👈 Listen on both TCP and UDP
  },
  action: {
    type: 'socket-handler',
    socketHandler: handleTcpDns,      // 👈 TCP connections
    datagramHandler: handleUdpDns,    // 👈 UDP datagrams
  }
};
```

⚡ High-Performance NFTables Forwarding

For ultra-low latency on Linux, use kernel-level forwarding via `@push.rocks/smartnftables` (requires root):

```
import { SmartProxy } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [{
    name: 'nftables-fast',
    match: { ports: 443, domains: 'fast.example.com' },
    action: {
      type: 'forward',
      forwardingEngine: 'nftables',
      targets: [{ host: 'backend', port: 8080 }],
      tls: { mode: 'terminate', certificate: 'auto' },
      nftables: {
        protocol: 'tcp',
        preserveSourceIP: true  // Backend sees real client IP
      }
    }
```

```
  }]
});
```

🔒 SNI Passthrough (TLS Passthrough)

Forward encrypted traffic to backends without terminating TLS — the proxy routes based on the SNI hostname alone:

```
import { SmartProxy } from '@push.rocks/smartproxy';

const proxy = new SmartProxy({
  routes: [{
    name: 'sni-passthrough',
    match: { ports: 443, domains: 'secure.example.com' },
    action: {
      type: 'forward',
      targets: [{ host: 'backend-that-handles-tls', port: 8443 }],
      tls: { mode: 'passthrough' }
    }
  }]
});
```

🔧 Advanced Features

🎯 Dynamic Routing

Route traffic based on runtime conditions using function-based host/port resolution:

```
const proxy = new SmartProxy({
  routes: [{
    name: 'dynamic-backend',
    match: {
      ports: 443,
      domains: 'app.example.com'
    },
    action: {
      type: 'forward',
```

```
      targets: [{
        host: (context) => {
          return context.path?.startsWith('/premium')
            ? 'premium-backend'
            : 'standard-backend';
        },
        port: 8080
      }],
      tls: { mode: 'terminate', certificate: 'auto' }
    }
  }]
});
```

> **Note:** Routes with dynamic functions (host/port callbacks) are automatically relayed through the TypeScript socket handler server, since JavaScript functions can't be serialized to Rust.

□□ Protocol-Specific Routing

Restrict routes to specific application-layer protocols. When `protocol` is set, the Rust engine detects the protocol after connection (or after TLS termination) and only matches routes that accept that protocol:

```
// HTTP-only route (rejects raw TCP connections)
const httpOnlyRoute: IRouteConfig = {
  name: 'http-api',
  match: {
    ports: 443,
    domains: 'api.example.com',
    protocol: 'http',   // Only match HTTP/1.1, HTTP/2, and WebSocket upgrades
  },
  action: {
    type: 'forward',
    targets: [{ host: 'api-backend', port: 8080 }],
    tls: { mode: 'terminate', certificate: 'auto' }
  }
};
```

```
// Raw TCP route (rejects HTTP traffic)
const tcpOnlyRoute: IRouteConfig = {
  name: 'database-proxy',
  match: {
    ports: 5432,
    protocol: 'tcp',    // Only match non-HTTP TCP streams
  },
  action: {
    type: 'forward',
    targets: [{ host: 'db-server', port: 5432 }]
  }
};
```

> **Note:** Omitting `protocol` (the default) matches any protocol. For TLS routes, protocol detection happens *after* TLS termination — during the initial SNI-based route match, `protocol` is not yet known and the route is allowed to match. The protocol restriction is enforced after the proxy peeks at the decrypted data.

🔒 Security Controls

Comprehensive per-route security options:

```
{
  name: 'secure-api',
  match: { ports: 443, domains: 'api.example.com' },
  action: {
    type: 'forward',
    targets: [{ host: 'api-backend', port: 8080 }],
    tls: { mode: 'terminate', certificate: 'auto' }
  },
  security: {
    // IP-based access control
    ipAllowList: ['10.0.0.0/8', '192.168.*'],
    ipBlockList: ['192.168.1.100'],

    // Connection limits
    maxConnections: 1000,
```

```
    // Rate limiting
    rateLimit: {
      enabled: true,
      maxRequests: 100,
      window: 60
    },

    // Authentication
    basicAuth: { users: [{ username: 'admin', password: 'secret' }] },
    jwtAuth: { secret: 'your-jwt-secret', algorithm: 'HS256' }
  }
}
```

Security options are configured directly on each route's `security` property — no separate helpers needed.

□□ Runtime Management

Control your proxy without restarts:

```
// Dynamic port management
await proxy.addListeningPort(8443);
await proxy.removeListeningPort(8080);
const ports = await proxy.getListeningPorts();

// Update routes on the fly (atomic, mutex-locked)
await proxy.updateRoutes([...newRoutes]);

// Get real-time metrics
const metrics = proxy.getMetrics();
console.log(`Active connections: ${metrics.connections.active()}`);
console.log(`Bytes in: ${metrics.totals.bytesIn()}`);
console.log(`Requests/sec: ${metrics.requests.perSecond()}`);
console.log(`Throughput in: ${metrics.throughput.instant().in} bytes/sec`);

// UDP metrics
console.log(`UDP sessions: ${metrics.udp.activeSessions()}`);
console.log(`Datagrams in: ${metrics.udp.datagramsIn()}`);
```

```
// Get detailed statistics from the Rust engine
const stats = await proxy.getStatistics();

// Certificate management
await proxy.provisionCertificate('my-route-name');
await proxy.renewCertificate('my-route-name');
const certStatus = await proxy.getCertificateStatus('my-route-name');

// NFTables status
const nftStatus = await proxy.getNfTablesStatus();
```

🔀 Header Manipulation

Transform requests and responses with template variables:

```
{
  action: {
    type: 'forward',
    targets: [{ host: 'backend', port: 8080 }]
  },
  headers: {
    request: {
      'X-Real-IP': '{clientIp}',
      'X-Request-ID': '{uuid}',
      'X-Forwarded-Proto': 'https'
    },
    response: {
      'Strict-Transport-Security': 'max-age=31536000',
      'X-Frame-Options': 'DENY'
    }
  }
}
```

🔀 PROXY Protocol Support

Preserve original client information through proxy chains:

```
const proxy = new SmartProxy({
  // Accept PROXY protocol from trusted load balancers
  acceptProxyProtocol: true,
  proxyIPs: ['10.0.0.1', '10.0.0.2'],

  // Forward PROXY protocol to backends
  sendProxyProtocol: true,

  routes: [...]
});
```

🔑 Custom Certificate Provisioning

Supply your own certificates or integrate with external certificate providers:

```
const proxy = new SmartProxy({
  certProvisionFunction: async (domain: string) => {
    // Return 'http01' to let the built-in ACME handle it
    if (domain.endsWith('.example.com')) return 'http01';

    // Or return a static certificate object
    return {
      publicKey: myPemCert,
      privateKey: myPemKey,
    };
  },
  certProvisionFallbackToAcme: true,  // Fall back to ACME if callback fails
  routes: [...]
});
```

💾 Consumer-Managed Certificate Storage

SmartProxy **never writes certificates to disk**. Instead, you own all persistence through the `certStore` interface. This gives you full control — store certs in a database, cloud KMS, encrypted vault, or wherever makes sense for your infrastructure:

```
const proxy = new SmartProxy({
  routes: [...],
```

```
  certProvisionFunction: async (domain) => myAcme.provision(domain),

  // Your persistence layer — SmartProxy calls these hooks
  certStore: {
    // Called once on startup to pre-load persisted certs
    loadAll: async () => {
      const certs = await myDb.getAllCerts();
      return certs.map(c => ({
        domain: c.domain,
        publicKey: c.certPem,
        privateKey: c.keyPem,
        ca: c.caPem,      // optional
      }));
    },

    // Called after each successful cert provision
    save: async (domain, publicKey, privateKey, ca) => {
      await myDb.upsertCert({ domain, certPem: publicKey, keyPem: privateKey, caPem: ca });
    },

    // Optional: called when a cert should be removed
    remove: async (domain) => {
      await myDb.deleteCert(domain);
    },
  },
});
```

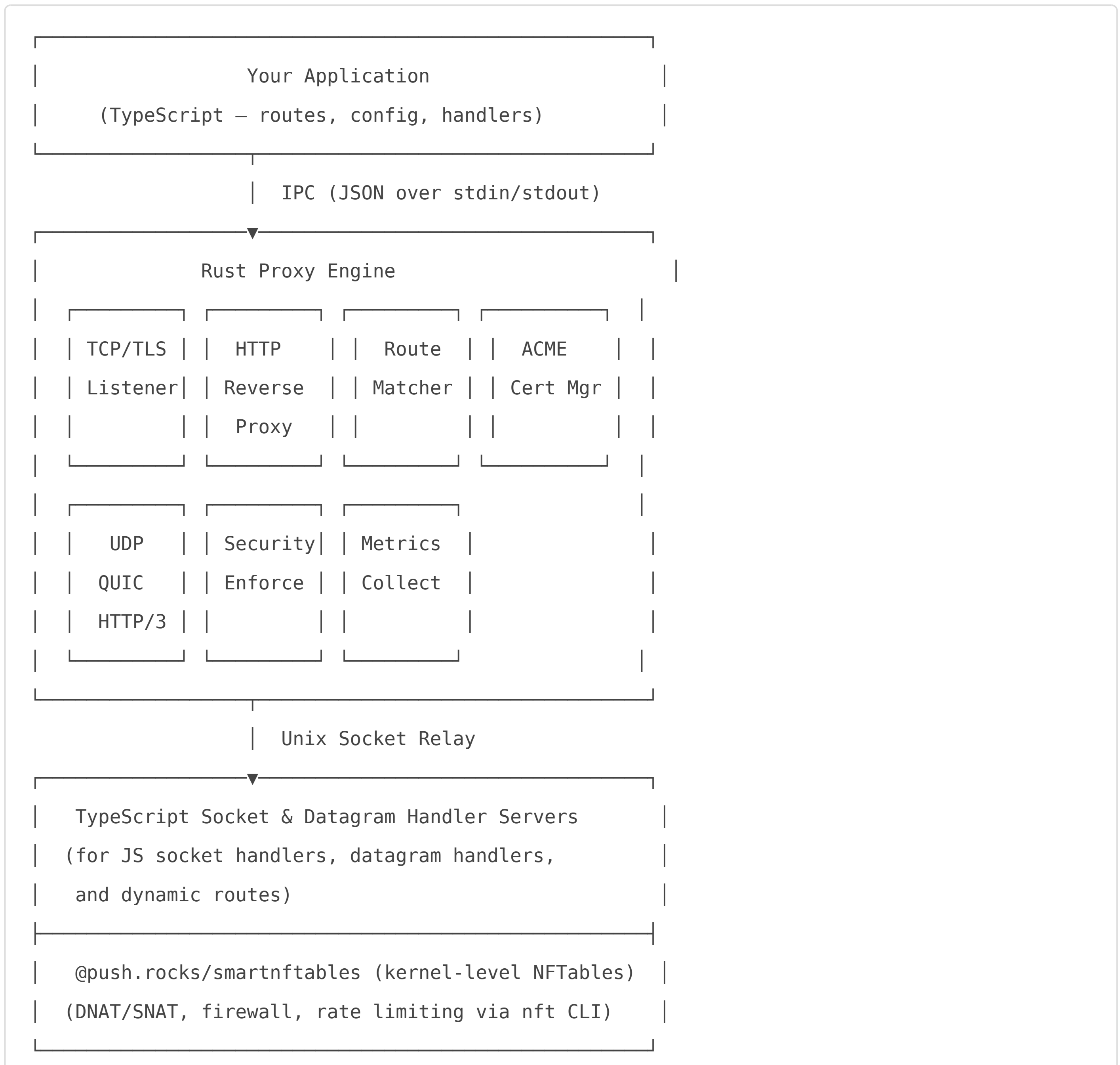
Startup flow:

1. Rust engine starts
2. Default self-signed `*` fallback cert is loaded (unless `disableDefaultCert: true`)
3. `certStore.loadAll()` is called → all returned certs are loaded into the Rust TLS stack
4. `certProvisionFunction` runs for any remaining `certificate: 'auto'` routes (skipping domains already loaded from the store)
5. After each successful provision, `certStore.save()` is called

This means your second startup is instant — no re-provisioning needed for domains that already have valid certs in your store.

Architecture

SmartProxy uses a hybrid **Rust + TypeScript** architecture:



- **Rust Engine** handles all networking: TCP, UDP, TLS, QUIC, HTTP proxying, connection management, security, and metrics
- **TypeScript** provides the npm API, configuration types, validation, and handler callbacks
- **NFTables** managed by `@push.rocks/smartnftables` — kernel-level DNAT/SNAT forwarding, firewall rules, and rate limiting via the `nft` CLI
- **IPC** — The TypeScript wrapper uses JSON commands/events over stdin/stdout to communicate with the Rust binary

- **Socket/Datagram Relay** — Unix domain socket servers for routes requiring TypeScript-side handling (socket handlers, datagram handlers, dynamic host/port functions)

📋 Route Configuration Reference

Match Criteria

```
interface IRouteMatch {
  ports: TPortRange;                     // Required — port(s) to listen on
  transport?: 'tcp' | 'udp' | 'all';    // Transport protocol (default: 'tcp')
  domains?: string | string[];           // 'example.com', '*.example.com'
  path?: string;                         // '/api/*', '/users/:id'
  clientIp?: string[];                   // ['10.0.0.0/8', '192.168.*']
  tlsVersion?: string[];                 // ['TLSv1.2', 'TLSv1.3']
  headers?: Record<string, string | RegExp>;  // Match by HTTP headers
  protocol?: 'http' | 'tcp' | 'udp' | 'quic' | 'http3';  // Application-layer protocol
}

// Port range supports single numbers, arrays, and ranges
type TPortRange = number | Array<number | { from: number; to: number }>;
```

Action Types

Type	Description
`forward`	Proxy to one or more backend targets (with optional TLS, WebSocket, load balancing, UDP/QUIC)
`socket-handler`	Custom socket/datagram handling function in TypeScript

Target Options

```
interface IRouteTarget {
  host: string | string[] | ((context: IRouteContext) => string | string[]);
  port: number | 'preserve' | ((context: IRouteContext) => number);
  tls?: IRouteTls;                  // Per-target TLS override
  priority?: number;                // Target priority
```

```
  match?: ITargetMatch;             // Sub-match within a route (by port, path, headers,
method)
  websocket?: IRouteWebSocket;
  loadBalancing?: IRouteLoadBalancing;
  sendProxyProtocol?: boolean;
  headers?: IRouteHeaders;
  advanced?: IRouteAdvanced;
  backendTransport?: 'tcp' | 'udp'; // Backend transport (e.g., receive QUIC, forward as TCP)
}
```

TLS Options

```
interface IRouteTls {
  mode: 'passthrough' | 'terminate' | 'terminate-and-reencrypt';
  certificate?: 'auto' | {
    key: string;
    cert: string;
    ca?: string;
    keyFile?: string;
    certFile?: string;
  };
  acme?: {
    email: string;
    useProduction?: boolean;
    challengePort?: number;
    renewBeforeDays?: number;
  };
  versions?: string[];
  ciphers?: string;
  honorCipherOrder?: boolean;
  sessionTimeout?: number;
}
```

WebSocket Options

```
interface IRouteWebSocket {
  enabled: boolean;
```

```
  pingInterval?: number;     // ms between pings
  pingTimeout?: number;      // ms to wait for pong
  maxPayloadSize?: number;   // Maximum frame payload
  subprotocols?: string[];   // Allowed subprotocols
  allowedOrigins?: string[]; // CORS origins
}
```

Load Balancing Options

```
interface IRouteLoadBalancing {
  algorithm: 'round-robin' | 'least-connections' | 'ip-hash';
  healthCheck?: {
    path: string;
    interval: number;        // ms
    timeout: number;         // ms
    unhealthyThreshold: number;
    healthyThreshold: number;
  };
}
```

Backend Protocol Options

```
// Set on action.options
{
  action: {
    type: 'forward',
    targets: [...],
    options: {
      backendProtocol: 'auto' | 'http1' | 'http2' | 'http3'
    }
  }
}
```

Value	Backend Behavior
`'auto'`	Best-effort: discovers H3 via Alt-Svc, probes H2 via ALPN, falls back to H1
`'http1'`	Always HTTP/1.1 (no ALPN probe)

Value	Backend Behavior
`'http2'`	Always HTTP/2 (hard-fail if handshake fails)
`'http3'`	Always HTTP/3 over QUIC (3s connect timeout, hard-fail if unreachable)

UDP & QUIC Options

```
interface IRouteUdp {
  sessionTimeout?: number;       // Idle timeout per UDP session (ms, default: 60000)
  maxSessionsPerIP?: number;     // Max concurrent sessions per IP (default: 1000)
  maxDatagramSize?: number;      // Max datagram size in bytes (default: 65535)
  quic?: IRouteQuic;
}

interface IRouteQuic {
  maxIdleTimeout?: number;                 // QUIC idle timeout (ms, default: 30000)
  maxConcurrentBidiStreams?: number;        // Max bidi streams (default: 100)
  maxConcurrentUniStreams?: number;         // Max uni streams (default: 100)
  enableHttp3?: boolean;                   // Enable HTTP/3 (default: false)
  altSvcPort?: number;                     // Port for Alt-Svc header
  altSvcMaxAge?: number;                   // Alt-Svc max age in seconds (default: 86400)
  initialCongestionWindow?: number;        // Initial congestion window (bytes)
}
```

Exports Reference

```
import {
  // Core
  SmartProxy,                        // Main proxy class
  SocketHandlers,                    // Pre-built socket handlers (echo, proxy, block,
httpRedirect, httpServer, etc.)

  // Route Utilities
  mergeRouteConfigs,                 // Deep-merge two route configs
  findMatchingRoutes,                // Find routes matching criteria
  findBestMatchingRoute,             // Find best matching route
  cloneRoute,                        // Deep-clone a route
```

```
  generateRouteId,                 // Generate deterministic route ID
  RouteValidator,                  // Validate route configurations
} from '@push.rocks/smartproxy';
```

All routes are configured as plain `IRouteConfig` objects with `match` and `action` properties — see the examples throughout this document.

📚 API Documentation

SmartProxy Class

```
class SmartProxy extends EventEmitter {
  constructor(options: ISmartProxyOptions);

  // Lifecycle
  start(): Promise<void>;
  stop(): Promise<void>;

  // Route Management (atomic, mutex-locked)
  updateRoutes(routes: IRouteConfig[]): Promise<void>;

  // Port Management
  addListeningPort(port: number): Promise<void>;
  removeListeningPort(port: number): Promise<void>;
  getListeningPorts(): Promise<number[]>;

  // Monitoring & Metrics
  getMetrics(): IMetrics;              // Sync — returns cached metrics adapter
  getStatistics(): Promise<any>;       // Async — queries Rust engine

  // Certificate Management
  provisionCertificate(routeName: string): Promise<void>;
  renewCertificate(routeName: string): Promise<void>;
  getCertificateStatus(routeName: string): Promise<any>;
  getEligibleDomainsForCertificates(): string[];

  // NFTables (managed by @push.rocks/smartnftables)
```

```
  getNfTablesStatus(): INftStatus | null;

  // Events
  on(event: 'error', handler: (err: Error) => void): this;
  on(event: 'certificate-issued', handler: (ev: ICertificateIssuedEvent) => void): this;
  on(event: 'certificate-failed', handler: (ev: ICertificateFailedEvent) => void): this;
}
```

Configuration Options

```
interface ISmartProxyOptions {
  routes: IRouteConfig[];                   // Required: array of route configs

  // ACME/Let's Encrypt
  acme?: {
    email: string;                          // Contact email for Let's Encrypt
    useProduction?: boolean;                // Use production servers (default: false)
    port?: number;                          // HTTP-01 challenge port (default: 80)
    renewThresholdDays?: number;            // Days before expiry to renew (default: 30)
    autoRenew?: boolean;                    // Enable auto-renewal (default: true)
    renewCheckIntervalHours?: number;       // Renewal check interval (default: 24)
  };

  // Custom certificate provisioning
  certProvisionFunction?: (domain: string) => Promise<ICert | 'http01'>;
  certProvisionFallbackToAcme?: boolean;    // Fall back to ACME on failure (default: true)
  certProvisionTimeout?: number;            // Timeout per provision call (ms)
  certProvisionConcurrency?: number;        // Max concurrent provisions

  // Consumer-managed certificate persistence (see "Consumer-Managed Certificate Storage")
  certStore?: ISmartProxyCertStore;

  // Self-signed fallback
  disableDefaultCert?: boolean;             // Disable '*' self-signed fallback (default:
false)

  // Rust binary path override
  rustBinaryPath?: string;                  // Custom path to the Rust proxy binary
```

```
// Global defaults
defaults?: {
  target?: { host: string; port: number };
  security?: { ipAllowList?: string[]; ipBlockList?: string[]; maxConnections?: number };
};

// PROXY protocol
proxyIPs?: string[];                      // Trusted proxy IPs
acceptProxyProtocol?: boolean;            // Accept PROXY protocol headers
sendProxyProtocol?: boolean;              // Send PROXY protocol to targets

// Timeouts
connectionTimeout?: number;               // Backend connection timeout (default: 60s)
initialDataTimeout?: number;              // Initial data/SNI timeout (default: 60s)
socketTimeout?: number;                   // Socket inactivity timeout (default: 60s)
maxConnectionLifetime?: number;           // Max connection lifetime (default: 1h)
inactivityTimeout?: number;               // Inactivity timeout (default: 75s)
gracefulShutdownTimeout?: number;         // Shutdown grace period (default: 30s)

// Connection limits
maxConnectionsPerIP?: number;             // Per-IP connection limit (default: 100)
connectionRateLimitPerMinute?: number;    // Per-IP rate limit (default: 300/min)

// Keep-alive
keepAliveTreatment?: 'standard' | 'extended' | 'immortal';
keepAliveInactivityMultiplier?: number;   // (default: 4)
extendedKeepAliveLifetime?: number;       // (default: 1h)

// Metrics
metrics?: {
  enabled?: boolean;
  sampleIntervalMs?: number;
  retentionSeconds?: number;
};

// Behavior
enableDetailedLogging?: boolean;          // Verbose connection logging
enableTlsDebugLogging?: boolean;          // TLS handshake debug logging
```

```
}
```

ISmartProxyCertStore Interface

```
interface ISmartProxyCertStore {
  /** Called once on startup to pre-load persisted certs */
  loadAll: () => Promise<Array<{
    domain: string;
    publicKey: string;
    privateKey: string;
    ca?: string;
  }>>;

  /** Called after each successful cert provision */
  save: (domain: string, publicKey: string, privateKey: string, ca?: string) => Promise<void>;

  /** Optional: remove a cert from storage */
  remove?: (domain: string) => Promise<void>;
}
```

IMetrics Interface

The `getMetrics()` method returns a cached metrics adapter that polls the Rust engine:

```
const metrics = proxy.getMetrics();

// Connection metrics
metrics.connections.active();              // Current active connections
metrics.connections.total();               // Total connections since start
metrics.connections.byRoute();             // Map<routeName, activeCount>
metrics.connections.byIP();                // Map<ip, activeCount>
metrics.connections.topIPs(10);            // Top N IPs by connection count

// Throughput (bytes/sec)
metrics.throughput.instant();              // { in: number, out: number }
metrics.throughput.recent();               // Recent average
metrics.throughput.average();              // Overall average
```

```
metrics.throughput.byRoute();              // Map<routeName, { in, out }>

// Request rates
metrics.requests.perSecond();              // Requests per second
metrics.requests.perMinute();              // Requests per minute
metrics.requests.total();                  // Total requests

// UDP metrics
metrics.udp.activeSessions();              // Current active UDP sessions
metrics.udp.totalSessions();              // Total UDP sessions since start
metrics.udp.datagramsIn();                // Datagrams received
metrics.udp.datagramsOut();               // Datagrams sent

// Cumulative totals
metrics.totals.bytesIn();                  // Total bytes received
metrics.totals.bytesOut();                 // Total bytes sent
metrics.totals.connections();              // Total connections

// Backend metrics
metrics.backends.byBackend();              // Map<backend, IBackendMetrics>
metrics.backends.protocols();              // Map<backend, protocol>
metrics.backends.topByErrors(10);          // Top N error-prone backends

// Percentiles
metrics.percentiles.connectionDuration();  // { p50, p95, p99 }
metrics.percentiles.bytesTransferred();    // { in: { p50, p95, p99 }, out: { p50, p95, p99 }
}
```

🛠️ Troubleshooting

Certificate Issues

- ✅ Ensure domain DNS points to your server
- ✅ Port 80 must be accessible for ACME HTTP-01 challenges
- ✅ Check DNS propagation with `dig` or `nslookup`
- ✅ Verify the email in ACME configuration is valid
- ✅ Use `getCertificateStatus('route-name')` to check cert state

Connection Problems

- □ Check route priorities (higher number = matched first)
- □ Verify security rules aren't blocking legitimate traffic
- □ Test with `curl -v` for detailed connection output
- □ Enable debug logging with `enableDetailedLogging: true`

Rust Binary Not Found

SmartProxy searches for the Rust binary in this order:

1. `rustBinaryPath` option in `ISmartProxyOptions`
2. `SMARTPROXY_RUST_BINARY` environment variable
3. Platform-specific npm package (`@push.rocks/smartproxy-linux-x64`, etc.)
4. `dist_rust/rustproxy` relative to the package root (built by `tsrust`)
5. Local dev build (`./rust/target/release/rustproxy`)
6. System PATH (`rustproxy`)

QUIC / HTTP3 Caveats

- **GREASE frames are disabled.** The underlying h3 crate sends GREASE frames by default to test protocol extensibility. However, some HTTP/3 clients and servers don't properly ignore unknown frame types, causing 400/500 errors or stream hangs (h3#206). SmartProxy disables GREASE on both the server side (for incoming H3 requests) and the client side (for H3 backend connections) to maximize compatibility.
- **HTTP/3 is pre-release.** The h3 ecosystem (h3 0.0.8, h3-quinn 0.0.10, quinn 0.11) is still pre-1.0. Expect rough edges.

Performance Tuning

- □ Use NFTables forwarding for high-traffic routes (Linux only)
- □ Enable connection keep-alive where appropriate
- □ Use `getMetrics()` and `getStatistics()` to identify bottlenecks
- □ Adjust `maxConnectionsPerIP` and `connectionRateLimitPerMinute` based on your workload
- □ Use `passthrough` TLS mode when backend can handle TLS directly

□□ Best Practices

1. 🔲 **Use Helper Functions** — They provide sensible defaults and prevent common mistakes
2. 🔲 **Set Route Priorities** — More specific routes should have higher priority values
3. 🔲 **Enable Security** — Always use IP filtering and rate limiting for public-facing services
4. 🔲 **Monitor Metrics** — Use the built-in metrics to catch issues early
5. 🔲 **Certificate Monitoring** — Set up alerts before certificates expire
6. 🔲 **Graceful Shutdown** — Always call `proxy.stop()` for clean connection termination
7. 🔲 **Validate Routes** — Use `RouteValidator.validateRoutes()` to catch config errors before deployment
8. 🔲 **Atomic Updates** — Use `updateRoutes()` for hot-reloading routes (mutex-locked, no downtime)
9. 🔲 **Use Socket Handlers** — For protocols beyond HTTP, implement custom socket handlers instead of fighting the proxy model
10. 🔲 **Use `certStore`** — Persist certs in your own storage to avoid re-provisioning on every restart

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 #3
Created 2026-03-28 11:11:41 UTC by foss.global Team
Updated 2026-03-28 12:18:33 UTC by foss.global Team