readme.md for @push.rocks/smartdns

A TypeScript-first DNS toolkit powered by high-performance Rust binaries — covering everything from simple record lookups to running a full authoritative DNS server with DNSSEC, DNS-over-HTTPS, and automatic Let's Encrypt certificates.

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.

Install

```
pnpm install @push.rocks/smartdns
```

Architecture at a Glance □□

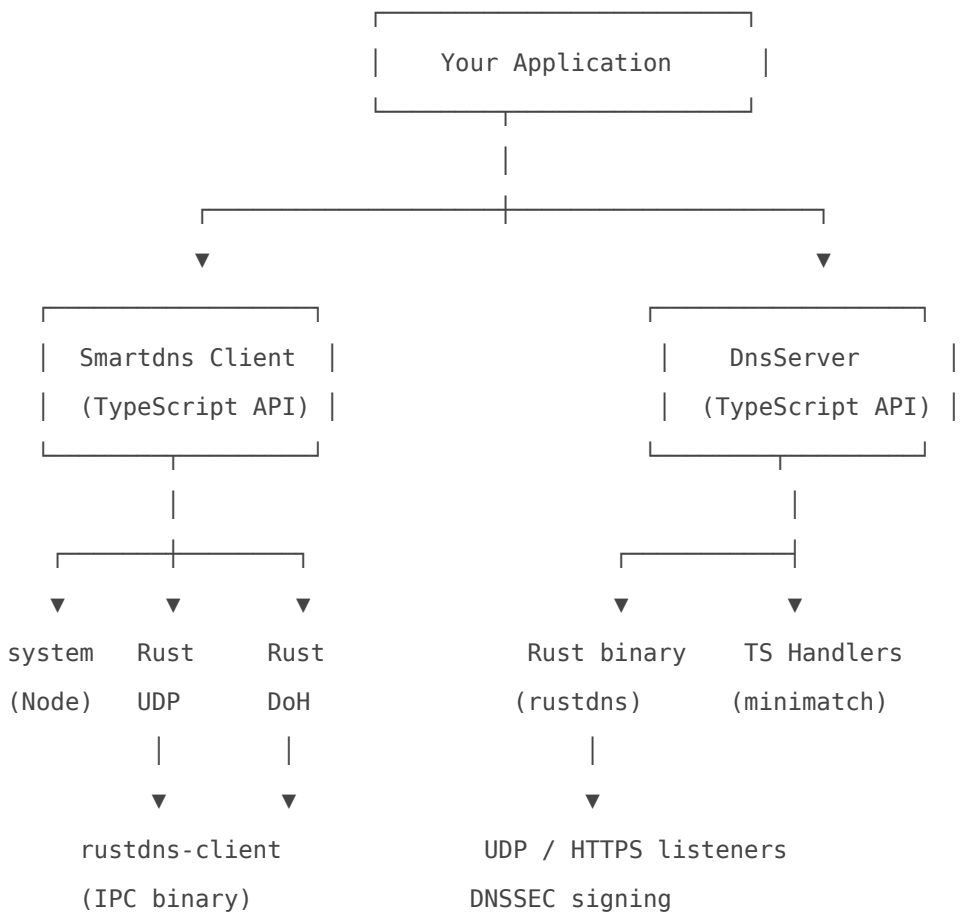
smartdns ships as **three entry points** that you can import independently:

Entry point	What it does
`@push.rocks/smartdns/client`	DNS resolution & record queries (UDP, DoH, system resolver)
`@push.rocks/smartdns/server`	Full DNS server — UDP, DoH, DNSSEC, ACME
`@push.rocks/smartdns`	Convenience re-export of both modules

Both the **client** and the **server** delegate performance-critical work to compiled **Rust binaries** that ship with the package:

- `rustdns` — The server binary: network I/O, packet parsing, DNSSEC signing
- `rustdns-client` — The client binary: UDP wire-format queries, RFC 8484 DoH resolution

TypeScript retains the public API, handler registration, ACME orchestration, and strategy routing. Communication between TypeScript and Rust happens over stdin/stdout JSON IPC via `@push.rocks/smartrust`.



Usage

Quick Start

```
// DNS client — resolve records
import { Smartdns } from '@push.rocks/smartdns/client';

const dns = new Smartdns({});
const records = await dns.getRecordsA('example.com');
console.log(records);
```

```
// DNS server — serve records
import { DnsServer } from '@push.rocks/smartdns/server';

const server = new DnsServer({
  udpPort: 5333,
  httpsPort: 8443,
  httpsKey: '...pem...',
  httpsCert: '...pem...',
  dnssecZone: 'example.com',
});

server.registerHandler('*.example.com', ['A'], (question) => ({
  name: question.name,
  type: 'A',
  class: 'IN',
  ttl: 300,
  data: '192.168.1.100',
}));

await server.start();
```

Or import from the unified entry point:

```
import { dnsClientMod, dnsServerMod } from '@push.rocks/smartdns';

const client = new dnsClientMod.Smartdns({});
const server = new dnsServerMod.DnsServer({ /* ... */ });
```

□□ DNS Client

The `Smartdns` class resolves DNS records using a configurable strategy that combines the system resolver, raw UDP queries, and DNS-over-HTTPS — all backed by a Rust binary for the wire-format transports.

Constructor Options

```
interface ISmartDnsConstructorOptions {
  strategy?: 'doh' | 'udp' | 'system' | 'prefer-system' | 'prefer-udp'; // default: 'prefer-system'
  allowDohFallback?: boolean; // fallback to DoH when system fails (default: true)
  timeoutMs?: number; // per-query timeout in milliseconds
}
```

Resolution Strategies

Strategy	Behavior
`prefer-system`	◫ Try the OS resolver first, fall back to Rust DoH. Honors `/etc/hosts`.
`system`	◫ Use only the Node.js system resolver. No Rust binary needed.
`doh`	◫ Use only DNS-over-HTTPS (RFC 8484 wire format via Cloudflare). Rust-powered.
`udp`	⚡ Use only raw UDP queries to upstream resolver (Cloudflare 1.1.1.1). Rust-powered.
`prefer-udp`	⚡ Try Rust UDP first, fall back to Rust DoH if UDP fails.

The Rust binary (`rustdns-client`) is spawned **lazily** — only on the first query that needs it. This means `system`-only usage incurs zero Rust overhead.

Querying Records

```
const dns = new Smartdns({ strategy: 'prefer-udp' });

// Type-specific helpers
const aRecords    = await dns.getRecordsA('example.com');
const aaaaRecords = await dns.getRecordsAAAA('example.com');
const txtRecords  = await dns.getRecordsTxt('example.com');

// Generic query — supports A, AAAA, CNAME, MX, TXT, NS, SOA, PTR, SRV
const mxRecords = await dns.getRecords('example.com', 'MX');

// Nameserver lookup
const nameservers = await dns.getNameServers('example.com');
```

Every query returns an array of `IDnsRecord`:

```
interface IDnsRecord {
  name: string;
  type: string;          // 'A', 'AAAA', 'TXT', 'MX', etc.
  dnsSecEnabled: boolean; // true if upstream AD flag was set
  value: string;
}
```

DNSSEC Detection □□

When using `doh`, `udp`, or `prefer-udp` strategies, the Rust binary sends queries with the EDNS0 DO (DNSSEC OK) bit set and reports the AD (Authenticated Data) flag from the upstream response:

```
const dns = new Smartdns({ strategy: 'udp' });
const records = await dns.getRecordsA('cloudflare.com');
console.log(records[0].dnsSecEnabled); // true — upstream validated DNSSEC
```

Checking DNS Propagation

Wait for a specific record to appear — essential after making DNS changes:

```
const propagated = await dns.checkUntilAvailable(
  'example.com',
  'TXT',
  'verification=abc123',
  50,   // max check cycles (default: 50)
  500   // interval in ms (default: 500)
);

if (propagated) {
  console.log('Record is live!');
}
```

The method alternates between system resolver and the configured strategy on each cycle for maximum coverage.

Configuring the System DNS Provider

Override the global Node.js DNS resolver for all subsequent lookups:

```
import { makeNodeProcessUseDnsProvider } from '@push.rocks/smartdns/client';

makeNodeProcessUseDnsProvider('cloudflare'); // 1.1.1.1 / 1.0.0.1
makeNodeProcessUseDnsProvider('google');     // 8.8.8.8 / 8.8.4.4
```

Cleanup

When you're done with a `Smartdns` instance (especially one using Rust strategies), call `destroy()` to kill the Rust child process:

```
const dns = new Smartdns({ strategy: 'udp' });
// ... do queries ...
dns.destroy(); // kills rustdns-client process
```

□□ DNS Server

The `DnsServer` class runs a production-capable authoritative DNS server backed by a Rust binary. It supports standard UDP DNS (port 53), DNS-over-HTTPS, DNSSEC signing, and automated Let's Encrypt certificates.

Server Options

```
interface IDnsServerOptions {
  udpPort: number;               // Port for UDP DNS queries
  httpsPort: number;             // Port for DNS-over-HTTPS
  httpsKey: string;              // PEM private key (path or content)
  httpsCert: string;             // PEM certificate (path or content)
  dnssecZone: string;            // Zone for DNSSEC signing
  primaryNameserver?: string;    // SOA mname field (default: 'ns1.{dnssecZone}')
  udpBindInterface?: string;     // IP to bind UDP (default: '0.0.0.0')
  httpsBindInterface?: string;   // IP to bind HTTPS (default: '0.0.0.0')
  manualUdpMode?: boolean;       // Don't auto-bind UDP socket
  manualHttpsMode?: boolean;     // Don't auto-bind HTTPS server
  enableLocalhostHandling?: boolean; // RFC 6761 localhost (default: true)
}
```

Basic Server

```
import { DnsServer } from '@push.rocks/smartdns/server';

const server = new DnsServer({
  udpPort: 5333,
  httpsPort: 8443,
  httpsKey: '...pem...',
  httpsCert: '...pem...',
  dnssecZone: 'example.com',
});

// Register handlers
server.registerHandler('example.com', ['A'], (question) => ({
  name: question.name,
  type: 'A',
  class: 'IN',
  ttl: 300,
  data: '93.184.215.14',
}));

server.registerHandler('example.com', ['TXT'], (question) => ({
  name: question.name,
  type: 'TXT',
  class: 'IN',
  ttl: 300,
  data: 'v=spf1 include:_spf.example.com ~all',
}));

await server.start();
// DNS Server started (UDP: 0.0.0.0:5333, HTTPS: 0.0.0.0:8443)
```

Handler System □□

Handlers use **glob patterns** (via `minimatch`) to match incoming query names. Multiple handlers can contribute records to the same response.

```
// Exact domain
server.registerHandler('example.com', ['A'], handler);

// All subdomains
server.registerHandler('*.example.com', ['A'], handler);

// Specific pattern
server.registerHandler('db-*.internal.example.com', ['A'], (question) => {
  const id = question.name.match(/db-(\d+)/)?.[1];
  return {
    name: question.name,
    type: 'A',
    class: 'IN',
    ttl: 60,
    data: `10.0.1.${id}`,
  };
});

// Catch-all
server.registerHandler('*', ['A'], (question) => ({
  name: question.name,
  type: 'A',
  class: 'IN',
  ttl: 300,
  data: '127.0.0.1',
}));

// Multiple record types
server.registerHandler('example.com', ['MX'], (question) => ({
  name: question.name,
  type: 'MX',
  class: 'IN',
  ttl: 300,
  data: { preference: 10, exchange: 'mail.example.com' },
}));

// Unregister a handler
server.unregisterHandler('example.com', ['A']);
```

When no handler matches, the server automatically returns an **SOA record** for the zone.

DNSSEC

DNSSEC is enabled automatically when you set the `dnssecZone` option. The Rust backend handles:

- **Key generation** — ECDSA P-256 (algorithm 13) by default
- **DNSKEY / DS record** generation
- **RRSIG signing** for all responses
- **NSEC records** for authenticated denial of existence

```
const server = new DnsServer({
  udpPort: 53,
  httpsPort: 443,
  httpsKey: '...',
  httpsCert: '...',
  dnssecZone: 'secure.example.com',
});

// Just register handlers as usual — signing is automatic
server.registerHandler('secure.example.com', ['A'], (q) => ({
  name: q.name,
  type: 'A',
  class: 'IN',
  ttl: 300,
  data: '10.0.0.1',
}));

await server.start();
```

Supported algorithms: **ECDSAP256SHA256** (13), **ED25519** (15), **RSASHA256** (8).

SOA Records

The server auto-generates SOA records for zones when no specific handler matches. Customize the primary nameserver:

```
const server = new DnsServer({
  // ...
  dnssecZone: 'example.com',
  primaryNameserver: 'ns1.example.com', // defaults to 'ns1.{dnssecZone}'
});
```

```
// Generated SOA includes:
// mname:   ns1.example.com
// rname:   hostmaster.example.com
// serial:  Unix timestamp
// refresh: 3600, retry: 600, expire: 604800, minimum: 86400
```

Let's Encrypt Integration □□

Built-in ACME DNS-01 challenge support for automatic SSL certificates:

```
const server = new DnsServer({
  udpPort: 53,
  httpsPort: 443,
  httpsKey: '/path/to/key.pem',
  httpsCert: '/path/to/cert.pem',
  dnssecZone: 'example.com',
});

await server.start();

const result = await server.retrieveSslCertificate(
  ['example.com', 'www.example.com'],
  {
    email: 'admin@example.com',
    staging: false,
    certDir: './certs',
  }
);

if (result.success) {
  console.log('Certificate installed!');
  // The server automatically:
  // 1. Registers temporary _acme-challenge TXT handlers
  // 2. Completes DNS-01 validation
  // 3. Updates the HTTPS server with the new cert
  // 4. Cleans up challenge handlers
}
```

Interface Binding

Restrict the server to specific network interfaces:

```
// Localhost only — great for development
const server = new DnsServer({
  // ...
  udpBindInterface: '127.0.0.1',
  httpsBindInterface: '127.0.0.1',
});

// Different interfaces per protocol
const server = new DnsServer({
  // ...
  udpBindInterface: '192.168.1.100',
  httpsBindInterface: '10.0.0.50',
});
```

Manual Socket Handling □□

For clustering, load balancing, or custom transports, take control of socket management:

```
import { DnsServer } from '@push.rocks/smartdns/server';
import * as dgram from 'dgram';

// Manual UDP mode — you control the socket
const server = new DnsServer({
  // ...
  manualUdpMode: true,
});

await server.start(); // HTTPS auto-binds, UDP does not

const socket = dgram.createSocket('udp4');
socket.on('message', (msg, rinfo) => {
  server.handleUdpMessage(msg, rinfo, (response, responseRinfo) => {
    socket.send(response, responseRinfo.port, responseRinfo.address);
  });
});
```

```
socket.bind(5353);
```

Full manual mode (both protocols):

```
const server = new DnsServer({
  // ...
  manualUdpMode: true,
  manualHttpsMode: true,
});

await server.start(); // Neither protocol binds automatically
```

Process individual DNS packets directly:

```
// Synchronous (TypeScript fallback)
const response = server.processRawDnsPacket(packetBuffer);

// Asynchronous (via Rust bridge — includes DNSSEC signing)
const response = await server.processRawDnsPacketAsync(packetBuffer);
```

Load Balancing Example

```
import * as dgram from 'dgram';
import * as os from 'os';

const numCPUs = os.cpus().length;

for (let i = 0; i < numCPUs; i++) {
  const socket = dgram.createSocket({ type: 'udp4', reuseAddr: true });

  socket.on('message', (msg, rinfo) => {
    server.handleUdpMessage(msg, rinfo, (response, rinfo) => {
      socket.send(response, rinfo.port, rinfo.address);
    });
  });

  socket.bind(53);
}
```

Stopping the Server

```
await server.stop();
```

This gracefully shuts down the Rust process and releases all bound sockets.

□□ Rust Crate Structure

The Rust workspace (`rust/crates/`) contains five crates:

Crate	Purpose
`rustdns`	Server binary — IPC management loop, handler callback routing
`rustdns-client`	Client binary — stateless UDP/DoH query proxy
`rustdns-protocol`	DNS wire format parsing, encoding, and RDATA decode/encode
`rustdns-server`	Async UDP + HTTPS servers (tokio, hyper, rustls)
`rustdns-dnssec`	ECDSA/ED25519 key generation and RRset signing

Pre-compiled binaries for `linux_amd64` and `linux_arm64` are included in `dist_rust/`. Cross-compilation is handled by `@git.zone/tsrust`.

□□ Testing

```
# Run all tests
pnpm test

# Run specific test file
tstest test/test.client.ts --verbose
tstest test/test.server.ts --verbose
```

Example test:

```
import { expect, tap } from '@git.zone/tstest/tapbundle';
import { Smartdns } from '@push.rocks/smartdns/client';

tap.test('resolve A records via UDP', async () => {
  const dns = new Smartdns({ strategy: 'udp' });
  const records = await dns.getRecordsA('google.com');
  expect(records).toBeArray();
  expect(records[0]).toHaveProperty('type', 'A');
  expect(records[0]).toHaveProperty('value');
  dns.destroy();
});

tap.test('detect DNSSEC via DoH', async () => {
  const dns = new Smartdns({ strategy: 'doh' });
  const records = await dns.getRecordsA('cloudflare.com');
  expect(records[0].dnsSecEnabled).toBeTrue();
  dns.destroy();
});

export default tap.start();
```

License and Legal Information

Trademarks

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 11:10:34 UTC by foss.global Team
Updated 2026-03-28 12:17:21 UTC by foss.global Team