readme.md for @push.rocks/smartacme

A TypeScript-based ACME client and server for certificate management with a focus on simplicity and power. Includes a full RFC 8555-compliant ACME client for Let's Encrypt and a built-in ACME Directory Server for running your own Certificate Authority.

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 add @push.rocks/smartacme
```

Ensure your project uses TypeScript and ECMAScript Modules (ESM).

Usage

`@push.rocks/smartacme` automates the full ACME certificate lifecycle — obtaining, renewing, and storing SSL/TLS certificates from Let's Encrypt. It features a built-in RFC 8555-compliant ACME protocol implementation, pluggable challenge handlers (DNS-01, HTTP-01), pluggable certificate storage backends (MongoDB, in-memory, or your own), structured error handling with smart retry logic, and built-in concurrency control with rate limiting to keep you safely within Let's Encrypt limits.

□□ Quick Start

```
import { SmartAcme, certmanagers, handlers } from '@push.rocks/smartacme';
import * as cloudflare from '@apiclient.xyz/cloudflare';

// 1. Set up a certificate manager (MongoDB or in-memory)
const certManager = new certmanagers.MongoCertManager({
  mongoDbUrl: 'mongodb://localhost:27017',
  mongoDbName: 'myapp',
  mongoDbPass: 'secret',
});

// 2. Set up challenge handlers
const cfAccount = new cloudflare.CloudflareAccount('YOUR_CF_API_TOKEN');
const dnsHandler = new handlers.Dns01Handler(cfAccount);

// 3. Create and start SmartAcme
const smartAcme = new SmartAcme({
  accountEmail: 'admin@example.com',
  certManager,
  environment: 'production', // or 'integration' for staging
  challengeHandlers: [dnsHandler],
});

await smartAcme.start();

// 4. Get a certificate
const cert = await smartAcme.getCertificateForDomain('example.com');
console.log(cert.publicKey);  // PEM certificate chain
console.log(cert.privateKey); // PEM private key

// 5. Clean up
await smartAcme.stop();
```

⚙ SmartAcme Options

```
interface ISmartAcmeOptions {
  accountEmail: string;                    // ACME account email
  accountPrivateKey?: string;              // Optional account key (auto-generated if
omitted)
```

```
  certManager: ICertManager;                // Certificate storage backend
  environment: 'production' | 'integration'; // Let's Encrypt environment
  challengeHandlers: IChallengeHandler[];    // At least one handler required
  challengePriority?: string[];              // e.g. ['dns-01', 'http-01']
  retryOptions?: {                           // Optional retry/backoff config
    retries?: number;                        // Default: 10
    factor?: number;                         // Default: 4
    minTimeoutMs?: number;                   // Default: 1000
    maxTimeoutMs?: number;                   // Default: 60000
  };
  // Concurrency & rate limiting
  maxConcurrentIssuances?: number;           // Global cap on parallel ACME ops (default: 5)
  maxOrdersPerWindow?: number;               // Max orders in sliding window (default: 250)
  orderWindowMs?: number;                    // Sliding window duration in ms (default: 3
hours)
}
```

□□ Getting Certificates

```
// Standard certificate for a single domain
const cert = await smartAcme.getCertificateForDomain('example.com');

// Include wildcard coverage (requires DNS-01 handler)
// Issues a single cert covering example.com AND *.example.com
const certWithWildcard = await smartAcme.getCertificateForDomain('example.com', {
  includeWildcard: true,
});

// Request wildcard only
const wildcardCert = await smartAcme.getCertificateForDomain('*.example.com');
```

Certificates are automatically cached and reused when still valid. Renewal happens automatically when a certificate is within 10 days of expiration. The actual X.509 expiry date is parsed from the issued certificate, ensuring renewal timing is precise.

□□ Certificate Object

The returned `SmartacmeCert` (also exported as `Cert`) object has these properties:

Property	Type	Description
`id`	`string`	Unique certificate identifier
`domainName`	`string`	Domain the cert is issued for
`publicKey`	`string`	PEM-encoded certificate chain
`privateKey`	`string`	PEM-encoded private key
`csr`	`string`	Certificate Signing Request
`created`	`number`	Timestamp of creation
`validUntil`	`number`	Timestamp of expiration

Useful methods:

```
cert.isStillValid();    // true if not expired
cert.shouldBeRenewed(); // true if expires within 10 days
```

□□ Concurrency Control & Rate Limiting

When many callers request certificates concurrently (e.g., hundreds of subdomains under the same TLD), SmartAcme automatically handles deduplication, concurrency, and rate limiting using a built-in task manager powered by `@push.rocks/taskbuffer`.

How It Works

Three constraint layers protect your ACME account:

Layer	What It Does	Default
Per-domain mutex	Only one issuance runs per base domain at a time. Concurrent requests for the same domain automatically wait and receive the same certificate result.	1 concurrent per domain
Global concurrency cap	Limits total parallel ACME operations across all domains.	5 concurrent
Account rate limit	Sliding-window rate limiter that keeps you under Let's Encrypt's 300 orders/3h account limit.	250 per 3 hours

□□ Automatic Request Deduplication

If 100 requests come in for subdomains of `example.com` simultaneously, only **one** ACME issuance runs. All other callers automatically wait and receive the same certificate — no duplicate orders, no wasted rate limit budget.

```
// These all resolve to the same certificate with a single ACME order:
const results = await Promise.all([
  smartAcme.getCertificateForDomain('app.example.com'),
  smartAcme.getCertificateForDomain('api.example.com'),
  smartAcme.getCertificateForDomain('cdn.example.com'),
]);
```

⚡ Configuring Limits

```
const smartAcme = new SmartAcme({
  accountEmail: 'admin@example.com',
  certManager,
  environment: 'production',
  challengeHandlers: [dnsHandler],
  maxConcurrentIssuances: 10,     // Allow up to 10 parallel ACME issuances
  maxOrdersPerWindow: 200,        // Cap at 200 orders per window
  orderWindowMs: 2 * 60 * 60_000, // 2-hour sliding window
});
```

□□ Observing Issuance Progress

Subscribe to the `certIssuanceEvents` stream to observe certificate issuance progress in real-time:

```
smartAcme.certIssuanceEvents.subscribe((event) => {
  switch (event.type) {
    case 'started':
      console.log(`□□Issuance started: ${event.task.name}`);
      break;
    case 'step':
      console.log(`□□Step: ${event.stepName} (${event.task.currentProgress}%)`);
      break;
    case 'completed':
```

```
      console.log(`□ Issuance completed: ${event.task.name}`);
      break;
    case 'failed':
      console.log(`□ Issuance failed: ${event.error}`);
      break;
  }
});
```

Each issuance goes through four steps: **prepare** (10%) → **authorize** (40%) → **finalize** (30%) → **store** (20%).

Certificate Managers

SmartAcme uses the `ICertManager` interface for pluggable certificate storage.

□□ MongoCertManager

Persistent storage backed by MongoDB using `@push.rocks/smartdata`:

```
import { certmanagers } from '@push.rocks/smartacme';

const certManager = new certmanagers.MongoCertManager({
  mongoDbUrl: 'mongodb://localhost:27017',
  mongoDbName: 'myapp',
  mongoDbPass: 'secret',
});
```

□□ MemoryCertManager

In-memory storage, ideal for testing or ephemeral workloads:

```
import { certmanagers } from '@push.rocks/smartacme';

const certManager = new certmanagers.MemoryCertManager();
```

□□ Custom Certificate Manager

Implement the `ICertManager` interface for your own storage backend:

```
import type { ICertManager, Cert } from '@push.rocks/smartacme';

class RedisCertManager implements ICertManager {
  async init(): Promise<void> { /* connect */ }
  async retrieveCertificate(domainName: string): Promise<Cert | null> { /* lookup */ }
  async storeCertificate(cert: Cert): Promise<void> { /* save */ }
  async deleteCertificate(domainName: string): Promise<void> { /* remove */ }
  async close(): Promise<void> { /* disconnect */ }
  async wipe(): Promise<void> { /* clear all */ }
}
```

Challenge Handlers

SmartAcme ships with three built-in ACME challenge handlers. All implement `IChallengeHandler<T>`.

□□ Dns01Handler

Uses Cloudflare (or any `IConvenientDnsProvider`) to set and remove DNS TXT records for `dns-01` challenges:

```
import { handlers } from '@push.rocks/smartacme';
import * as cloudflare from '@apiclient.xyz/cloudflare';

const cfAccount = new cloudflare.CloudflareAccount('YOUR_CF_TOKEN');
const dnsHandler = new handlers.Dns01Handler(cfAccount);
```

DNS-01 is **required** for wildcard certificates and works regardless of server accessibility.

□□ Http01Webroot

Writes challenge response files to a filesystem webroot for `http-01` validation:

```
import { handlers } from '@push.rocks/smartacme';

const httpHandler = new handlers.Http01Webroot({
  webroot: '/var/www/html',
```

```
});
```

The handler writes to `<webroot>/.well-known/acme-challenge/<token>` and cleans up after validation.

□□ Http01MemoryHandler

In-memory HTTP-01 handler — stores challenge tokens in memory and serves them via `handleRequest()`:

```
import { handlers } from '@push.rocks/smartacme';

const memHandler = new handlers.Http01MemoryHandler();

// Integrate with any HTTP server (Express, Koa, raw http, etc.)
app.use((req, res, next) => memHandler.handleRequest(req, res, next));
```

Perfect for serverless or container environments where filesystem access is limited.

□□ Custom Challenge Handler

Implement `IChallengeHandler<T>` for custom challenge types:

```
import type { handlers } from '@push.rocks/smartacme';

interface MyChallenge {
  type: string;
  token: string;
  keyAuthorization: string;
}

class MyHandler implements handlers.IChallengeHandler<MyChallenge> {
  getSupportedTypes(): string[] { return ['http-01']; }
  async prepare(ch: MyChallenge): Promise<void> { /* set up challenge response */ }
  async cleanup(ch: MyChallenge): Promise<void> { /* tear down */ }
  async checkWetherDomainIsSupported(domain: string): Promise<boolean> { return true; }
}
```

Error Handling

SmartAcme provides structured ACME error handling via the `AcmeError` class, which carries full RFC 8555 error information:

```
import { AcmeError } from '@push.rocks/smartacme/ts/acme/acme.classes.error.js';

try {
  const cert = await smartAcme.getCertificateForDomain('example.com');
} catch (err) {
  if (err instanceof AcmeError) {
    console.log(err.status);        // HTTP status code (e.g. 429)
    console.log(err.type);          // ACME error URN (e.g.
'urn:ietf:params:acme:error:rateLimited')
    console.log(err.detail);        // Human-readable message
    console.log(err.subproblems);   // Per-identifier sub-errors (RFC 8555 §6.7.1)
    console.log(err.retryAfter);    // Retry-After value in seconds
    console.log(err.isRateLimited); // true for 429 or rateLimited type
    console.log(err.isRetryable);   // true for 429, 503, 5xx, badNonce; false for 403/404/409
  }
}
```

The built-in retry logic is **error-aware**: non-retryable errors (403, 404, 409) are thrown immediately without wasting retry attempts, and rate-limited responses respect the server's `Retry-After` header instead of using blind exponential backoff.

Domain Matching

SmartAcme automatically maps subdomains to their base domain for certificate lookups:

```
subdomain.example.com → certificate for example.com  □
*.example.com         → certificate for example.com  □
a.b.example.com       → not supported (4+ levels)    □
```

Environment

Environment	Description

`production`	Let's Encrypt production servers. Certificates are browser-trusted. Rate limits apply.
`integration`	Let's Encrypt staging servers. No rate limits, but certificates are **not** browser-trusted. Use for testing.

Complete Example with HTTP-01

```
import { SmartAcme, certmanagers, handlers } from '@push.rocks/smartacme';
import * as http from 'http';

// In-memory handler for HTTP-01 challenges
const memHandler = new handlers.Http01MemoryHandler();

// Create HTTP server that serves ACME challenges
const server = http.createServer((req, res) => {
  memHandler.handleRequest(req, res, () => {
    res.statusCode = 200;
    res.end('OK');
  });
});
server.listen(80);

// Set up SmartAcme with in-memory storage and HTTP-01
const smartAcme = new SmartAcme({
  accountEmail: 'admin@example.com',
  certManager: new certmanagers.MemoryCertManager(),
  environment: 'production',
  challengeHandlers: [memHandler],
  challengePriority: ['http-01'],
});

await smartAcme.start();

const cert = await smartAcme.getCertificateForDomain('example.com');
// Use cert.publicKey and cert.privateKey with your HTTPS server

await smartAcme.stop();
```

```
server.close();
```

□□ ACME Directory Server (Built-in CA)

SmartAcme includes a full RFC 8555-compliant ACME Directory Server, allowing you to run your own Certificate Authority. This is useful for internal PKI, development/testing environments, and air-gapped networks.

Quick Start — ACME Server

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

const acmeServer = new server.AcmeServer({
  port: 14000,
  challengeVerification: false, // Auto-approve challenges (for testing)
  caOptions: {
    commonName: 'My Internal CA',
    certValidityDays: 365,
  },
});

await acmeServer.start();
console.log(acmeServer.getDirectoryUrl()); // http://localhost:14000/directory
console.log(acmeServer.getCaCertPem());    // Root CA certificate in PEM format

// ... use it, then shut down
await acmeServer.stop();
```

Server Options

```
interface IAcmeServerOptions {
  port?: number;                    // Default: 14000
  hostname?: string;                // Default: '0.0.0.0'
  baseUrl?: string;                 // Auto-built from hostname:port if not provided
```

```
  challengeVerification?: boolean;  // Default: true. Set false to auto-approve challenges
  caOptions?: {
    commonName?: string;            // CA subject CN (default: 'SmartACME Test CA')
    validityDays?: number;          // Root cert validity in days (default: 3650)
    certValidityDays?: number;      // Issued cert validity in days (default: 90)
  };
}
```

Using the Server with the Low-Level ACME Client

The `SmartAcme` class connects to Let's Encrypt by default. To use a custom ACME directory (like your own server), use the lower-level `AcmeClient` directly:

```
import { server } from '@push.rocks/smartacme';
import { AcmeCrypto, AcmeClient } from '@push.rocks/smartacme/ts/acme/index.js';

// 1. Start your own CA
const acmeServer = new server.AcmeServer({
  port: 14000,
  challengeVerification: false, // auto-approve for testing
});
await acmeServer.start();

// 2. Create an ACME client pointing at your CA
const accountKey = AcmeCrypto.createRsaPrivateKey();
const client = new AcmeClient({
  directoryUrl: acmeServer.getDirectoryUrl(),
  accountKeyPem: accountKey,
});

// 3. Register an account
await client.createAccount({ termsOfServiceAgreed: true, contact:
['mailto:admin@internal.example.com'] });

// 4. Create an order and issue a certificate
const order = await client.createOrder({
  identifiers: [{ type: 'dns', value: 'myapp.internal' }],
```

```
});

// ... complete challenges, finalize, and download cert
// (challenges auto-approved since challengeVerification is false)

await acmeServer.stop();
```

Server Endpoints

The ACME server implements all RFC 8555 endpoints:

Endpoint	Method	Description
`/directory`	GET	ACME directory with all endpoint URLs
`/new-nonce`	HEAD/GET	Fresh replay nonce
`/new-account`	POST	Account registration/lookup
`/new-order`	POST	Create certificate order
`/order/:id`	POST	Poll order status
`/authz/:id`	POST	Get authorization with challenges
`/challenge/:id`	POST	Trigger or poll challenge validation
`/finalize/:id`	POST	Submit CSR and issue certificate
`/cert/:id`	POST	Download PEM certificate chain

Challenge Verification

By default, the server performs real challenge verification (HTTP-01 fetches the token, DNS-01 queries TXT records). Set `challengeVerification: false` to auto-approve all challenges — useful for testing or internal environments where domain validation isn't needed.

Root CA Certificate

Use `getCaCertPem()` to retrieve the root CA certificate for trust configuration:

```
import * as fs from 'fs';
fs.writeFileSync('/usr/local/share/ca-certificates/my-ca.crt', acmeServer.getCaCertPem());
// Then: sudo update-ca-certificates
```

□□ Architecture

Under the hood, SmartAcme uses a fully custom RFC 8555-compliant ACME protocol implementation (no external ACME libraries). Key internal modules:

Client Modules (`ts/acme/`)

Module	Purpose
`AcmeClient`	Top-level ACME facade — orders, authorizations, finalization
`AcmeCrypto`	RSA key generation, JWK/JWS (RFC 7515/7638), CSR via `@peculiar/x509`
`AcmeHttpClient`	JWS-signed HTTP transport with nonce management and structured logging
`AcmeError`	Structured error class with type URN, subproblems, Retry-After, retryability
`AcmeOrderManager`	Order lifecycle — create, poll, finalize, download certificate
`AcmeChallengeManager`	Key authorization computation and challenge completion
`TaskManager`	Constraint-based concurrency control, rate limiting, and request deduplication via `@push.rocks/taskbuffer`

Server Modules (`ts_server/`)

Module	Purpose
`AcmeServer`	Top-level server facade — start, stop, configuration
`AcmeServerCA`	Self-signed root CA generation and certificate signing via `@peculiar/x509`
`JwsVerifier`	JWS signature verification (inverse of `AcmeCrypto.createJws`)
`NonceManager`	Single-use replay nonce generation and validation
`ChallengeVerifier`	HTTP-01 and DNS-01 challenge verification (with bypass mode)
`AcmeRouter`	Minimal HTTP router with parameterized path support
`MemoryAccountStore`	In-memory ACME account storage
`MemoryOrderStore`	In-memory order, authorization, challenge, and certificate storage

All cryptographic operations use `node:crypto`. The only external crypto dependency is `@peculiar/x509` for CSR generation and certificate signing.

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 11:09:58 UTC by foss.global Team
Updated 2026-03-28 12:16:36 UTC by foss.global Team