@serve.zone/containerarchive

backup and restore container based services

- readme.md for @serve.zone/containerarchive
- changelog.md for @serve.zone/containerarchive

readme.md for @serve.zone/containerarchive

A high-performance, content-addressed incremental backup engine with built-in deduplication, encryption, compression, and Reed-Solomon error correction — powered by a Rust core with a clean TypeScript API.

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.

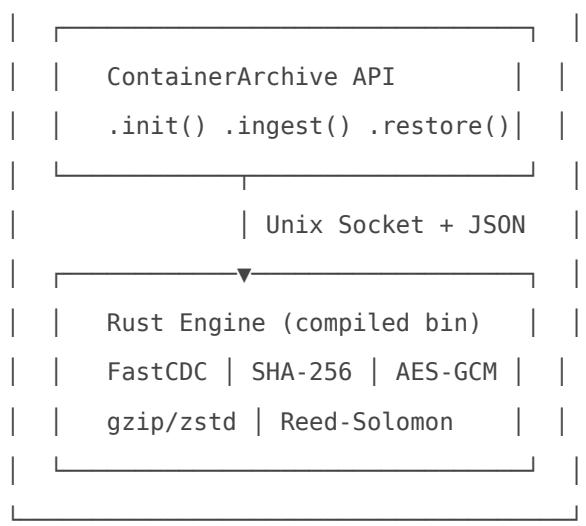
Install

```
pnpm install @serve.zone/containerarchive
```

□□ Architecture

containerarchive uses a **hybrid Rust + TypeScript architecture**. The heavy lifting — chunking, hashing, compression, encryption, pack file I/O, and parity — runs in a compiled Rust binary. The TypeScript layer provides a clean, idiomatic Node.js API and manages data streaming via Unix sockets through the `@push.rocks/smartrust` RustBridge IPC.

```
┌──────────────────────────────────────┐
│  Your Application  (TypeScript/JS)   │
```



Features

Feature	Details
Content-Defined Chunking	FastCDC with gear-based rolling hash — insertions/deletions only affect nearby boundaries
Deduplication	SHA-256 chunk addressing — identical data is stored only once across all snapshots
Compression	gzip or zstd per-chunk compression
Encryption	AES-256-GCM with Argon2id key derivation — passphrase-protected repositories
Pack Files	Chunks are batched into binary pack files with binary `.idx` indexes for fast lookup
Snapshots	Immutable point-in-time snapshots with metadata tags and multi-item support
Reed-Solomon Parity	RS(20,1) erasure coding — recover any single lost pack from a group of 20
Incremental	Only new/changed chunks are stored on each ingest
Streaming	Unix socket streaming between TypeScript and Rust for zero-copy data transfer
Multi-Item Snapshots	Bundle multiple data streams (DB dumps, config tarballs, etc.) into a single snapshot
Verification	Three-level integrity checks: quick, standard, full
Pruning	Retention policies (keep last N, days, weeks, months) with garbage collection

Feature	Details
Repair	Automatic index rebuild, stale lock removal, and parity-based pack recovery

Usage

Initialize a New Repository

```
import { ContainerArchive } from '@serve.zone/containerarchive';

// Unencrypted repository
const repo = await ContainerArchive.init('/path/to/backup-repo');

// Encrypted repository (AES-256-GCM + Argon2id)
const encryptedRepo = await ContainerArchive.init('/path/to/secure-repo', {
  passphrase: 'my-strong-passphrase',
});
```

Open an Existing Repository

```
const repo = await ContainerArchive.open('/path/to/backup-repo');

// With passphrase for encrypted repos
const repo = await ContainerArchive.open('/path/to/secure-repo', {
  passphrase: 'my-strong-passphrase',
});
```

Ingest Data (Single Stream)

```
import * as fs from 'node:fs';

const inputStream = fs.createReadStream('/path/to/database-dump.sql');
const snapshot = await repo.ingest(inputStream, {
  tags: { service: 'postgres', environment: 'production' },
```

```
  items: [{ name: 'database.sql', type: 'database-dump' }],
});

console.log(`Snapshot ${snapshot.id} created`);
console.log(`Original: ${snapshot.originalSize} bytes`);
console.log(`Stored: ${snapshot.storedSize} bytes`);
console.log(`New chunks: ${snapshot.newChunks}, Reused: ${snapshot.reusedChunks}`);
```

Multi-Item Ingest

Bundle multiple data streams into one snapshot:

```
import * as stream from 'node:stream';

const dbDump = fs.createReadStream('/tmp/pg_dump.sql');
const configTar = fs.createReadStream('/tmp/config-volumes.tar');

const snapshot = await repo.ingestMulti([
  { stream: dbDump, name: 'database.sql', type: 'database-dump' },
  { stream: configTar, name: 'config.tar', type: 'volume-tar' },
], {
  tags: { service: 'myapp', type: 'full-backup' },
});

console.log(`Items stored: ${snapshot.items.map(i => i.name).join(', ')}`);
```

Restore Data

```
// Restore an entire snapshot
const restoreStream = await repo.restore(snapshot.id);
const writeStream = fs.createWriteStream('/tmp/restored-dump.sql');
restoreStream.pipe(writeStream);

// Restore a specific item from a multi-item snapshot
const configStream = await repo.restore(snapshot.id, { item: 'config.tar' });
configStream.pipe(fs.createWriteStream('/tmp/restored-config.tar'));
```

List & Filter Snapshots

```
// List all snapshots
const allSnapshots = await repo.listSnapshots();

// Filter by tags
const prodSnapshots = await repo.listSnapshots({
  tags: { environment: 'production' },
});

// Filter by date range
const recentSnapshots = await repo.listSnapshots({
  after: '2026-03-01T00:00:00Z',
  before: '2026-03-22T00:00:00Z',
});

// Get a specific snapshot
const snap = await repo.getSnapshot('snapshot-id-here');
```

Verify Repository Integrity

```
// Quick check — validates index consistency
const quick = await repo.verify({ level: 'quick' });

// Standard — reads pack headers and validates checksums
const standard = await repo.verify({ level: 'standard' });

// Full — decompresses and re-hashes every chunk
const full = await repo.verify({ level: 'full' });

console.log(`OK: ${full.ok}`);
console.log(`Packs checked: ${full.stats.packsChecked}`);
console.log(`Chunks checked: ${full.stats.chunksChecked}`);
```

Prune Old Snapshots

```
// Dry run first
const preview = await repo.prune({ keepLast: 5, keepDays: 30 }, true);
console.log(`Would remove ${preview.removedSnapshots} snapshots, free ${preview.freedBytes}
bytes`);

// Execute for real
const result = await repo.prune({
  keepLast: 5,
  keepDays: 30,
  keepWeeks: 12,
  keepMonths: 6,
});
console.log(`Removed ${result.removedSnapshots} snapshots, ${result.removedPacks} packs`);
```

Repair & Maintenance

```
// Repair — rebuild index, remove stale locks, attempt parity recovery
const repairResult = await repo.repair();
console.log(`Index rebuilt: ${repairResult.indexRebuilt}`);
console.log(`Packs repaired via parity: ${repairResult.packsRepaired}`);

// Rebuild global index from pack .idx files
await repo.reindex();

// Remove stale locks
await repo.unlock();
await repo.unlock({ force: true }); // force-remove all locks
```

Event Subscriptions

Monitor ingest progress and errors with RxJS-based event streams:

```
// Track ingest progress
const sub = repo.on('ingest:progress', (data) => {
  console.log(`${data.operation}: ${data.percentage}% — ${data.message}`);
});
```

```
// Track completed ingests
repo.on('ingest:complete', (data) => {
  console.log(`Snapshot ${data.snapshotId} complete — ${data.newChunks} new chunks`);
});

// Track verification errors
repo.on('verify:error', (error) => {
  console.error(`Verification error in ${error.pack || error.chunk}: ${error.error}`);
});

// Unsubscribe when done
sub.unsubscribe();
```

Close the Repository

```
await repo.close();
```

□□ Repository Structure

When initialized, a repository has the following on-disk layout:

```
backup-repo/
├── config.json          # Repository config (chunking, compression, encryption, parity)
├── packs/
│   ├── data/            # Binary pack files (.pack) and indexes (.idx)
│   └── parity/          # Reed-Solomon parity packs
├── snapshots/           # JSON snapshot manifests
├── index/               # Global chunk index (hash → pack location)
├── keys/                # Encrypted key files (for passphrase-protected repos)
└── locks/               # Advisory locks for concurrent access
```

□□ How It Works

1. **Chunking** — Incoming data is split into variable-size chunks using FastCDC with a gear-based rolling hash. Chunk sizes range from 64 KB to 1 MB (avg 256 KB). Content-defined

boundaries mean that insertions or edits only affect nearby chunks, maximizing dedup across versions.
2. **Hashing** — Each chunk is hashed with SHA-256 for content addressing. If a chunk's hash already exists in the global index, it's deduplicated — only a reference is stored.
3. **Compression** — New chunks are compressed with gzip (default) or zstd before storage. Per-chunk compression flags are stored in the index.
4. **Encryption** — If a passphrase is set, a random 256-bit master key is generated, wrapped with an Argon2id-derived key, and stored in a key file. Every chunk is encrypted with AES-256-GCM using a unique nonce.
5. **Packing** — Compressed (and optionally encrypted) chunks are appended into binary pack files (~8 MB target). Each pack has an associated `.idx` file with chunk offsets, sizes, and flags for O(1) lookup.
6. **Parity** — After every group of 20 data packs, a Reed-Solomon RS(20,1) parity pack is generated. If any single pack in the group is lost or corrupted, it can be fully reconstructed.
7. **Snapshots** — A JSON manifest records the chunk list, tags, sizes, and item metadata. Snapshots are immutable — pruning removes snapshots but never alters existing pack data in-place.
8. **Restore** — The snapshot manifest is read, chunks are looked up in the global index, fetched from pack files, decompressed, decrypted if needed, and streamed back in order via a Unix socket.

□□ API Reference

`ContainerArchive`

Method	Description
`static init(path, options?)`	Create a new repository. Returns `Promise<ContainerArchive>`
`static open(path, options?)`	Open an existing repository. Returns `Promise<ContainerArchive>`
`ingest(stream, options?)`	Ingest a single data stream. Returns `Promise<ISnapshot>`
`ingestMulti(items, options?)`	Ingest multiple streams as one snapshot. Returns `Promise<ISnapshot>`
`restore(snapshotId, options?)`	Restore a snapshot. Returns `Promise<ReadableStream>`
`listSnapshots(filter?)`	List snapshots with optional tag/date filtering. Returns `Promise<ISnapshot[]>`
`getSnapshot(id)`	Get a specific snapshot. Returns `Promise<ISnapshot>`
`verify(options?)`	Verify repository integrity (quick/standard/full). Returns `Promise<IVerifyResult>`

Method	Description
`prune(retention, dryRun?)`	Remove old snapshots and garbage-collect packs. Returns `Promise<IPruneResult>`
`repair()`	Rebuild index, remove stale locks, attempt parity recovery. Returns `Promise<IRepairResult>`
`reindex()`	Rebuild the global index from pack `.idx` files. Returns `Promise<void>`
`unlock(options?)`	Remove advisory locks. Returns `Promise<void>`
`on(event, handler)`	Subscribe to events. Returns `Subscription`
`close()`	Close the repository and terminate the Rust process. Returns `Promise<void>`

Key Interfaces

```
interface ISnapshot {
  id: string;
  version: number;
  createdAt: string;
  tags: Record<string, string>;
  originalSize: number;
  storedSize: number;
  chunkCount: number;
  newChunks: number;
  reusedChunks: number;
  items: ISnapshotItem[];
}

interface IRetentionPolicy {
  keepLast?: number;
  keepDays?: number;
  keepWeeks?: number;
  keepMonths?: number;
}

interface IVerifyResult {
  ok: boolean;
  errors: IVerifyError[];
  stats: {
```

```
    packsChecked: number;
    chunksChecked: number;
    snapshotsChecked: number;
  };
}
```

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.

changelog.md for @serve.zone/containerarchive

2026-03-23 - 0.1.3 - fix(repo)

no changes to commit

2026-03-23 - 0.1.2 - fix(package)

rename package namespace from @push.rocks to @serve.zone

- Updates the published package name to @serve.zone/containerarchive
- Aligns repository, bugs, homepage, README usage examples, and generated commit metadata with the new namespace

2026-03-23 - 0.1.1 - fix(repo)

no changes to commit

2026-03-22 - 0.1.0 - feat(rust-core)

add zstd chunk compression support and rewrite partially referenced packs during prune

- introduce selectable gzip or zstd compression with per-chunk flags persisted in the global index
- restore and verify now detect the compression algorithm from stored flags and validate encrypted chunk handling

- prune now rewrites partially referenced packs to reclaim space and exposes rewritten pack counts in the API
- add coverage for zstd repository ingest, restore, and verify flows
- add project readme documentation

2026-03-22 - 0.0.2 - fix(repository)

no changes to commit

2026-03-22 - 0.0.1 - ingest and recovery

Expanded the backup engine with parity-integrated ingest and faster encrypted restore paths.

- Wired parity generation into the ingest pipeline with automatic triggering after every configured number of packs
- Stored `ParityConfig` in repository `config.json`
- Optimized encrypted restore by storing nonces in global index entries, avoiding IDX re-reads on the fast path with cache fallback
- Improved repair to attempt parity-based pack reconstruction before reindexing

2026-03-21 - 0.0.1 - initial engine and parity foundations

Initial release of the content-addressed incremental backup engine, including core architecture and early parity-enabled ingest capabilities.

- Implemented a Rust-first backup engine with a TypeScript facade using the smartproxy/smartstorage pattern
- Added core backup features including FastCDC chunking, SHA-256 hashing, gzip compression, AES-256-GCM encryption with Argon2id, binary pack files, global index, snapshots, locking, verification, pruning, and repair
- Added Unix socket streaming between TypeScript and Rust through `@push.rocks/smartrust` RustBridge IPC for ingest and restore
- Introduced multi-item ingest, processing multiple items sequentially into a single snapshot with separate chunk lists

- Added Reed-Solomon parity support for pack file groups to enable single-pack-loss recovery
- Extended repair to attempt parity-based recovery for missing pack files
- Verified the implementation with passing integration and Rust unit tests