readme.md for @push.rocks/smartfs

Modern, pluggable filesystem module with fluent API, Web Streams, Rust-powered durability, and multiple storage backends.

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.

Features

- **Fluent API** — Action-last chainable interface for elegant, readable code
- **Pluggable Providers** — Swap backends (Node.js fs, in-memory, Rust) without changing a line of application code
- **Rust Provider** — XFS-safe `fsync` durability, cross-compiled binary via IPC for production-grade reliability
- **Web Streams** — True chunked streaming with the Web Streams API (including over IPC for the Rust provider)
- **Transactions** — Atomic multi-file operations with automatic rollback on failure
- **File Watching** — Event-based filesystem monitoring with debounce, filters, and recursive watching
- **Tree Hashing** — Deterministic SHA-256 directory hashing for cache-busting and change detection
- **Directory Copy & Move** — Full directory tree operations with conflict handling, filtering, and timestamp preservation
- ⚡ **Async-Only** — Modern `async`/`await` patterns throughout — no sync footguns
- **TypeScript-First** — Full type safety, IntelliSense, and exported interfaces
- **Multi-Runtime** — Works on Node.js, Bun, and Deno

Installation

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

Quick Start

```
import { SmartFs, SmartFsProviderNode } from '@push.rocks/smartfs';

// Create a SmartFS instance with the Node.js provider
const fs = new SmartFs(new SmartFsProviderNode());

// Write a file
await fs.file('/path/to/file.txt')
  .encoding('utf8')
  .write('Hello, World!');

// Read it back
const content = await fs.file('/path/to/file.txt')
  .encoding('utf8')
  .read();

console.log(content); // "Hello, World!"
```

API Overview

□□ File Operations

The fluent API uses an **action-last pattern** — configure first, then execute:

```
// Read
const content = await fs.file('/path/to/file.txt')
  .encoding('utf8')
```

```
  .read();

// Write
await fs.file('/path/to/file.txt')
  .encoding('utf8')
  .mode(0o644)
  .write('content');

// Atomic write (write to temp file, then rename — crash-safe)
await fs.file('/path/to/file.txt')
  .atomic()
  .write('content');

// Append
await fs.file('/path/to/file.txt')
  .append('more content');

// Copy with preserved timestamps
await fs.file('/source.txt')
  .preserveTimestamps()
  .copy('/destination.txt');

// Move / rename
await fs.file('/old.txt').move('/new.txt');

// Delete
await fs.file('/path/to/file.txt').delete();

// Existence check
const exists = await fs.file('/path/to/file.txt').exists();

// Stats (size, timestamps, permissions, etc.)
const stats = await fs.file('/path/to/file.txt').stat();
```

□□ Directory Operations

```
// Create directory (recursive by default)
await fs.directory('/path/to/nested/dir').create();
```

```
// List contents
const entries = await fs.directory('/path/to/dir').list();

// List recursively with glob filter and stats
const tsFiles = await fs.directory('/src')
  .recursive()
  .filter('*.ts')
  .includeStats()
  .list();

// Filter with RegExp
const configs = await fs.directory('/project')
  .filter(/\.config\.(ts|js)$/)
  .list();

// Filter with function
const largeFiles = await fs.directory('/data')
  .includeStats()
  .filter(entry => entry.stats && entry.stats.size > 1024)
  .list();

// Delete directory recursively
await fs.directory('/path/to/dir').recursive().delete();

// Check existence
const exists = await fs.directory('/path/to/dir').exists();
```

□□ Directory Copy & Move

Copy or move entire directory trees with fine-grained control:

```
// Basic copy
await fs.directory('/source').copy('/destination');

// Basic move
await fs.directory('/old-location').move('/new-location');

// Copy with options
```

```
await fs.directory('/source')
  .filter(/\.ts$/)           // Only copy TypeScript files
  .overwrite(true)           // Overwrite existing files
  .preserveTimestamps(true)  // Keep original timestamps
  .copy('/destination');

// Ignore filter for copy (copy everything regardless of list filter)
await fs.directory('/source')
  .filter('*.ts')
  .applyFilter(false)
  .copy('/destination');

// Handle target directory conflicts
await fs.directory('/source')
  .onConflict('merge')       // Default: merge contents
  .copy('/destination');

await fs.directory('/source')
  .onConflict('error')       // Throw if target exists
  .copy('/destination');

await fs.directory('/source')
  .onConflict('replace')     // Delete target first, then copy
  .copy('/destination');
```

Configuration Options:

Method	Default	Description
`filter(pattern)`	none	Filter files by glob, regex, or function
`applyFilter(bool)`	`true`	Whether to apply filter during copy/move
`overwrite(bool)`	`false`	Overwrite existing files at destination
`preserveTimestamps(bool)`	`false`	Preserve original file timestamps
`onConflict(mode)`	`'merge'`	`'merge'`, `'error'`, or `'replace'`

□□ Streaming Operations

SmartFS uses the **Web Streams API** for efficient, memory-friendly handling of large files. All providers — including the Rust provider over IPC — support true chunked streaming:

```
// Read stream
const readStream = await fs.file('/large-file.bin')
  .chunkSize(64 * 1024) // 64 KB chunks
  .readStream();

const reader = readStream.getReader();
while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  // Process chunk (Uint8Array)
}

// Write stream
const writeStream = await fs.file('/output.bin').writeStream();
const writer = writeStream.getWriter();

await writer.write(new Uint8Array([1, 2, 3]));
await writer.write(new Uint8Array([4, 5, 6]));
await writer.close();

// Pipe one stream to another
const input = await fs.file('/input.txt').readStream();
const output = await fs.file('/output.txt').writeStream();
await input.pipeTo(output);
```

□□ Transactions

Execute multiple file operations atomically with automatic rollback on failure:

```
// Simple transaction — all-or-nothing
await fs.transaction()
  .file('/file1.txt').write('content 1')
  .file('/file2.txt').write('content 2')
  .file('/file3.txt').delete()
  .commit();

// Transaction with error handling
const tx = fs.transaction()
  .file('/important.txt').write('critical data')
```

```
  .file('/backup.txt').copy('/backup-old.txt')
  .file('/temp.txt').delete();

try {
  await tx.commit();
  console.log('Transaction completed successfully');
} catch (error) {
  console.error('Transaction failed and was rolled back:', error);
  // All operations are automatically reverted
}
```

File Watching

Monitor filesystem changes with event-based watching:

```
// Watch a single file
const watcher = await fs.watch('/path/to/file.txt')
  .onChange(event => console.log('Changed:', event.path))
  .start();

// Watch a directory recursively with filters and debounce
const dirWatcher = await fs.watch('/src')
  .recursive()
  .filter(/\.ts$/)
  .debounce(100) // ms
  .onChange(event => console.log('Changed:', event.path))
  .onAdd(event => console.log('Added:', event.path))
  .onDelete(event => console.log('Deleted:', event.path))
  .start();

// Watch with a function filter
const customWatcher = await fs.watch('/src')
  .recursive()
  .filter(path => path.endsWith('.ts') && !path.includes('test'))
  .onAll(event => console.log(`${event.type}: ${event.path}`))
  .start();

// Stop watching
await dirWatcher.stop();
```

□□ Tree Hashing (Cache-Busting)

Compute a deterministic hash of all files in a directory — ideal for cache invalidation, change detection, and build triggers:

```
// Hash all files in a directory recursively
const hash = await fs.directory('/assets')
  .recursive()
  .treeHash();
// → "a3f2b8c9d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a1"

// Hash only specific file types
const cssHash = await fs.directory('/styles')
  .filter(/\.css$/)
  .recursive()
  .treeHash();

// Use a different algorithm
const sha512Hash = await fs.directory('/data')
  .recursive()
  .treeHash({ algorithm: 'sha512' });
```

How it works:

- Files are sorted by path for deterministic ordering
- Hashes relative path + file contents (streaming, memory-efficient)
- Does **not** include metadata (mtime/size) — pure content-based
- Same content always produces the same hash, regardless of timestamps

Use cases:

- □□ Cache-busting static assets
- □□ Detecting when served files have changed
- □□ Incremental build triggers
- □ Content integrity verification

Providers

SmartFS supports multiple storage backends through its provider architecture. Swap providers without changing any application code.

□□ Node.js Provider

Uses Node.js `fs/promises` for local filesystem operations. The default choice for most applications:

```
import { SmartFs, SmartFsProviderNode } from '@push.rocks/smartfs';

const fs = new SmartFs(new SmartFsProviderNode());
```

Capability	Status
File watching	□
Atomic writes	□
Transactions	□
Streaming	□
Symbolic links	□
File permissions	□

□□ Rust Provider

A high-durability provider powered by a cross-compiled Rust binary that communicates via JSON-over-IPC. The Rust provider adds **XFS-safe `fsync` guarantees** that the Node.js `fs` module cannot provide — after every metadata-changing operation (`write`, `rename`, `unlink`, `mkdir`), the parent directory is explicitly `fsync`'d to ensure durability on delayed-logging filesystems like XFS.

```
import { SmartFs, SmartFsProviderRust } from '@push.rocks/smartfs';

const fs = new SmartFs(new SmartFsProviderRust());

// Use it exactly like any other provider
await fs.file('/data/important.json')
  .atomic()
  .write(JSON.stringify(data));

// Don't forget to shut down when done
const provider = fs.provider as SmartFsProviderRust;
await provider.shutdown();
```

Capability	Status
File watching	□ (via `notify` crate)

Capability	Status
Atomic writes	□ (with fsync + parent fsync)
Transactions	□ (with batch fsync)
Streaming	□ (chunked IPC)
Symbolic links	□
File permissions	□

Key advantages over the Node.js provider:

- `fsync` on parent directories after all metadata changes (crash-safe on XFS)
- Atomic writes with `fsync` → `rename` → `fsync parent` sequence
- Batch `fsync` for transactions (collect affected directories, sync once at end)
- Cross-device move with fallback (`EXDEV` handling)
- Uses the `notify` crate for reliable file watching

□□ Memory Provider

In-memory virtual filesystem — perfect for testing:

```
import { SmartFs, SmartFsProviderMemory } from '@push.rocks/smartfs';

const fs = new SmartFs(new SmartFsProviderMemory());

// All operations work in memory — fast, isolated, no cleanup needed
await fs.file('/virtual/file.txt').write('data');
const content = await fs.file('/virtual/file.txt').encoding('utf8').read();

// Clear all data between tests
(fs.provider as SmartFsProviderMemory).clear();
```

Capability	Status
File watching	□
Atomic writes	□
Transactions	□
Streaming	□
Symbolic links	□
File permissions	□

□□ Custom Providers

Build your own provider by implementing the `ISmartFsProvider` interface:

```
import type { ISmartFsProvider } from '@push.rocks/smartfs';

class MyS3Provider implements ISmartFsProvider {
  public readonly name = 's3';
  public readonly capabilities = {
    supportsWatch: false,
    supportsAtomic: true,
    supportsTransactions: true,
    supportsStreaming: true,
    supportsSymlinks: false,
    supportsPermissions: false,
  };

  // Implement all required methods...
  async readFile(path: string, options?) { /* ... */ }
  async writeFile(path: string, content, options?) { /* ... */ }
  // ... etc
}

const fs = new SmartFs(new MyS3Provider());
```

Advanced Usage

Encoding Options

```
// UTF-8 (default for text)
await fs.file('/file.txt').encoding('utf8').write('text');

// Binary (Buffer)
const buffer = Buffer.from([0x48, 0x65, 0x6c, 0x6c, 0x6f]);
await fs.file('/file.bin').write(buffer);
const data = await fs.file('/file.bin').read(); // Returns Buffer
```

```
// Base64
await fs.file('/file.txt').encoding('base64').write('SGVsbG8=');

// Hex
await fs.file('/file.txt').encoding('hex').write('48656c6c6f');
```

File Permissions

```
// Set file mode
await fs.file('/script.sh')
  .mode(0o755)
  .write('#!/bin/bash\necho "Hello"');

// Set directory mode
await fs.directory('/private')
  .mode(0o700)
  .create();
```

Complex Filtering

```
const recentLargeTs = await fs.directory('/src')
  .recursive()
  .includeStats()
  .filter(entry => {
    if (!entry.stats) return false;
    return entry.isFile &&
           entry.name.endsWith('.ts') &&
           entry.stats.size > 1024 &&
           entry.stats.mtime > new Date('2024-01-01');
  })
  .list();
```

Transaction Operations

```
const tx = fs.transaction();

// Build up operations
tx.file('/data/file1.json').write(JSON.stringify(data1));
tx.file('/data/file2.json').write(JSON.stringify(data2));
tx.file('/data/file1.json').copy('/backup/file1.json');
tx.file('/data/old.json').delete();

// Execute atomically — all succeed or all revert
await tx.commit();
```

Type Definitions

SmartFS is fully typed. All interfaces and types are exported:

```
import type {
  // Provider interface
  ISmartFsProvider,
  IProviderCapabilities,
  TWatchCallback,
  IWatcherHandle,

  // Core types
  TEncoding,        // 'utf8' | 'utf-8' | 'ascii' | 'base64' | 'hex' | 'binary' | 'buffer'
  TFileMode,        // number
  IFileStats,
  IDirectoryEntry,

  // Watch types
  TWatchEventType,  // 'add' | 'change' | 'delete'
  IWatchEvent,
  IWatchOptions,

  // Operation types
  TTransactionOperationType, // 'write' | 'delete' | 'copy' | 'move' | 'append'
  ITransactionOperation,
  IReadOptions,
```

```
  IWriteOptions,
  IStreamOptions,
  ICopyOptions,
  IListOptions,
} from '@push.rocks/smartfs';
```

Error Handling

SmartFS throws descriptive errors that mirror POSIX conventions:

```
try {
  await fs.file('/nonexistent.txt').read();
} catch (error) {
  console.error(error.message);
  // "ENOENT: no such file or directory, open '/nonexistent.txt'"
}

// Transactions automatically rollback on error
try {
  await fs.transaction()
    .file('/file1.txt').write('data')
    .file('/readonly/file2.txt').write('data') // fails
    .commit();
} catch (error) {
  // file1.txt is reverted to its original state
  console.error('Transaction failed:', error);
}
```

Performance Tips

1. **Use streaming** for large files (> 1MB) — avoids loading entire files into memory
2. **Batch operations** with transactions for durability and performance
3. **Use the memory provider** for testing — instant, isolated, no disk I/O
4. **Enable atomic writes** for critical data — prevents partial writes on crash
5. **Debounce watchers** to reduce event noise during rapid changes
6. **Use** `treeHash` instead of reading individual files for change detection
7. **Use the Rust provider** on XFS or when you need guaranteed durability

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 11:13:43 UTC by foss.global Team
Updated 2026-03-28 12:20:29 UTC by foss.global Team