readme.md for @push.rocks/smartbucket

A powerful, cloud-agnostic TypeScript library for object storage that makes S3-compatible storage feel like a modern filesystem. Built for developers who demand simplicity, type-safety, and advanced features like real-time bucket watching, metadata management, file locking, intelligent trash handling, and memory-efficient streaming.

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.

Why SmartBucket?

- **Cloud Agnostic** - Write once, run on AWS S3, MinIO, DigitalOcean Spaces, Backblaze B2, Wasabi, Cloudflare R2, or any S3-compatible storage
- **Modern TypeScript** - First-class TypeScript support with complete type definitions and async/await patterns
- **Real-Time Watching** - Monitor bucket changes with polling-based watcher supporting RxJS and EventEmitter patterns
- **Memory Efficient** - Handle millions of files with async generators, RxJS observables, and cursor pagination
- **Smart Trash System** - Recover accidentally deleted files with built-in trash and restore functionality
- **File Locking** - Prevent concurrent modifications with built-in locking mechanisms
- **Rich Metadata** - Attach custom metadata to any file for powerful organization and search
- **Streaming Support** - Efficient handling of large files with Node.js and Web streams
- **Directory-like API** - Intuitive filesystem-like operations on object storage
- **Fail-Fast** - Strict-by-default API catches errors immediately with precise stack traces

Quick Start

```
import { SmartBucket } from '@push.rocks/smartbucket';

// Connect to your storage
const storage = new SmartBucket({
  accessKey: 'your-access-key',
  accessSecret: 'your-secret-key',
  endpoint: 's3.amazonaws.com',
  port: 443,
  useSsl: true
});

// Get or create a bucket
const bucket = await storage.getBucketByName('my-app-data');

// Upload a file
await bucket.fastPut({
  path: 'users/profile.json',
  contents: JSON.stringify({ name: 'Alice', role: 'admin' })
});

// Download it back
const data = await bucket.fastGet({ path: 'users/profile.json' });
console.log(JSON.parse(data.toString()));

// List files efficiently (even with millions of objects!)
for await (const key of bucket.listAllObjects('users/')) {
  console.log('Found:', key);
}

// Watch for changes in real-time
const watcher = bucket.createWatcher({ prefix: 'uploads/', pollIntervalMs: 3000 });
watcher.changeSubject.subscribe((change) => {
  console.log('Change detected:', change.type, change.key);
});
await watcher.start();
```

Install

```
# Using pnpm (recommended)
pnpm add @push.rocks/smartbucket

# Using npm
npm install @push.rocks/smartbucket --save
```

Usage

Table of Contents

Getting Started

First, set up your storage connection:

```
import { SmartBucket } from '@push.rocks/smartbucket';

// Initialize with your cloud storage credentials
const smartBucket = new SmartBucket({
```

```
  accessKey: 'your-access-key',
  accessSecret: 'your-secret-key',
  endpoint: 's3.amazonaws.com', // Or your provider's endpoint
  port: 443,
  useSsl: true,
  region: 'us-east-1' // Optional, defaults to 'us-east-1'
});
```

For MinIO or self-hosted storage:

```
const smartBucket = new SmartBucket({
  accessKey: 'minioadmin',
  accessSecret: 'minioadmin',
  endpoint: 'localhost',
  port: 9000,
  useSsl: false // MinIO often runs without SSL locally
});
```

Working with Buckets

Creating Buckets

```
// Create a new bucket
const myBucket = await smartBucket.createBucket('my-awesome-bucket');
```

Getting Existing Buckets

```
// Get a bucket reference (throws if not found - strict by default!)
const existingBucket = await smartBucket.getBucketByName('existing-bucket');

// Check first, then get (non-throwing approach)
if (await smartBucket.bucketExists('maybe-exists')) {
  const bucket = await smartBucket.getBucketByName('maybe-exists');
}
```

Removing Buckets

```
// Delete a bucket (must be empty)
await smartBucket.removeBucket('old-bucket');
```

File Operations

Upload Files

```
const bucket = await smartBucket.getBucketByName('my-bucket');

// Simple file upload (returns File object)
const file = await bucket.fastPut({
  path: 'documents/report.pdf',
  contents: Buffer.from('Your file content here')
});

// Upload with string content
await bucket.fastPut({
  path: 'notes/todo.txt',
  contents: 'Buy milk\nCall mom\nRule the world'
});

// Upload with overwrite control
const uploadedFile = await bucket.fastPut({
  path: 'images/logo.png',
  contents: imageBuffer,
  overwrite: true // Set to true to replace existing files
});

// Error handling: fastPut throws if file exists and overwrite is false
try {
  await bucket.fastPut({
    path: 'existing-file.txt',
    contents: 'new content'
  });
} catch (error) {
  console.error('Upload failed:', error.message);
}
```

Download Files

```
// Get file as Buffer
const fileContent = await bucket.fastGet({
```

```
  path: 'documents/report.pdf'
});

// Get file as string
const textContent = fileContent.toString('utf-8');

// Parse JSON files directly
const jsonData = JSON.parse(fileContent.toString());
```

Check File Existence

```
const exists = await bucket.fastExists({
  path: 'documents/report.pdf'
});
```

Delete Files

```
// Permanent deletion
await bucket.fastRemove({
  path: 'old-file.txt'
});
```

Copy & Move Files

```
// Copy file within bucket
await bucket.fastCopy({
  sourcePath: 'original/file.txt',
  destinationPath: 'backup/file-copy.txt'
});

// Move file (copy + delete original)
await bucket.fastMove({
  sourcePath: 'temp/draft.txt',
  destinationPath: 'final/document.txt'
});

// Copy to different bucket
const targetBucket = await smartBucket.getBucketByName('backup-bucket');
await bucket.fastCopy({
```

```
  sourcePath: 'important/data.json',
  destinationPath: 'archived/data.json',
  targetBucket: targetBucket
});
```

Memory-Efficient Listing

SmartBucket provides three powerful patterns for listing objects, optimized for handling **millions of files** efficiently:

Async Generators (Recommended)

Memory-efficient streaming using native JavaScript async iteration:

```
// List all objects with prefix - streams one at a time!
for await (const key of bucket.listAllObjects('documents/')) {
  console.log('Found:', key);

  // Process each file individually (memory efficient!)
  const content = await bucket.fastGet({ path: key });
  processFile(content);

  // Early termination support
  if (shouldStop()) break;
}

// Find objects matching glob patterns
for await (const key of bucket.findByGlob('**/*.json')) {
  console.log('JSON file:', key);
}

// Complex glob patterns
for await (const key of bucket.findByGlob('npm/packages/*/index.json')) {
  console.log('Package index:', key);
}

for await (const key of bucket.findByGlob('images/*.{jpg,png,gif}')) {
  console.log('Image:', key);
}
```

Why use async generators?

- Processes one item at a time (constant memory usage)
- Supports early termination with `break`
- Native JavaScript - no dependencies
- Perfect for large buckets with millions of objects

RxJS Observables

Perfect for reactive pipelines and complex data transformations:

```
import { filter, take, map } from 'rxjs/operators';

// Stream keys as Observable with powerful operators
bucket.listAllObjectsObservable('logs/')
  .pipe(
    filter(key => key.endsWith('.log')),
    take(100),
    map(key => ({ key, timestamp: Date.now() }))
  )
  .subscribe({
    next: (item) => console.log('Log file:', item.key),
    error: (err) => console.error('Error:', err),
    complete: () => console.log('Listing complete')
  });

// Combine with other observables
import { merge } from 'rxjs';

const logs$ = bucket.listAllObjectsObservable('logs/');
const backups$ = bucket.listAllObjectsObservable('backups/');

merge(logs$, backups$)
  .pipe(filter(key => key.includes('2024')))
  .subscribe(key => console.log('2024 file:', key));
```

Cursor Pattern

Explicit pagination control for UI and resumable operations:

```
// Create cursor with custom page size
const cursor = bucket.createCursor('uploads/', { pageSize: 100 });

// Fetch pages manually
while (cursor.hasMore()) {
  const page = await cursor.next();
  console.log(`Page has ${page.keys.length} items`);

  for (const key of page.keys) {
    console.log(`  - ${key}`);
  }

  if (page.done) break;
}

// Save and restore cursor state (perfect for resumable operations!)
const token = cursor.getToken();
// Store token in database or session...

// ... later, in a different request ...
const newCursor = bucket.createCursor('uploads/', { pageSize: 100 });
newCursor.setToken(token); // Resume from saved position!
const nextPage = await newCursor.next();

// Reset cursor to start over
cursor.reset();
```

Convenience Methods

```
// Collect all keys into array (WARNING: loads everything into memory!)
const allKeys = await bucket.listAllObjectsArray('images/');
console.log(`Found ${allKeys.length} images`);
```

Performance Comparison:

Method	Memory Usage	Best For	Supports Early Exit
Async Generator	O(1) - constant	Most use cases, large datasets	Yes
Observable	O(1) - constant	Reactive pipelines, RxJS apps	Yes

Method	Memory Usage	Best For	Supports Early Exit
Cursor	O(pageSize)	UI pagination, resumable ops	Yes
Array	O(n) - grows with results	Small datasets (<10k items)	No

Bucket Watching

Monitor your storage bucket for changes in real-time with the powerful `BucketWatcher`:

```
// Create a watcher for a specific prefix
const watcher = bucket.createWatcher({
  prefix: 'uploads/',        // Watch files with this prefix
  pollIntervalMs: 3000,      // Check every 3 seconds
  includeInitial: false,     // Don't emit existing files on start
});

// RxJS Observable pattern (recommended for reactive apps)
watcher.changeSubject.subscribe((change) => {
  if (change.type === 'add') {
    console.log('New file:', change.key);
  } else if (change.type === 'modify') {
    console.log('Modified:', change.key);
  } else if (change.type === 'delete') {
    console.log('Deleted:', change.key);
  }
});

// EventEmitter pattern (classic Node.js style)
watcher.on('change', (change) => {
  console.log(`${change.type}: ${change.key}`);
});

watcher.on('error', (err) => {
  console.error('Watcher error:', err);
});

// Start watching
await watcher.start();
```

```
// Wait until watcher is ready (initial state built)
await watcher.readyDeferred.promise;
console.log('Watcher is now monitoring the bucket');

// ... your application runs ...

// Stop watching when done
await watcher.stop();
// Or use the alias:
await watcher.close();
```

Watcher Options

```
interface IBucketWatcherOptions {
  prefix?: string;           // Filter objects by prefix (default: '' = all)
  pollIntervalMs?: number;   // Polling interval in ms (default: 5000)
  bufferTimeMs?: number;     // Buffer events before emitting (for batching)
  includeInitial?: boolean;  // Emit existing files as 'add' on start (default: false)
  pageSize?: number;         // Objects per page when listing (default: 1000)
}
```

Buffered Events

For high-frequency change environments, buffer events to reduce processing overhead:

```
const watcher = bucket.createWatcher({
  prefix: 'high-traffic/',
  pollIntervalMs: 1000,
  bufferTimeMs: 2000, // Collect events for 2 seconds before emitting
});

// Receive batched events as arrays
watcher.changeSubject.subscribe((changes) => {
  if (Array.isArray(changes)) {
    console.log(`Batch of ${changes.length} changes:`);
    changes.forEach(c => console.log(`  - ${c.type}: ${c.key}`));
  }
});
```

```
await watcher.start();
```

Change Event Structure

```
interface IStorageChangeEvent {
  type: 'add' | 'modify' | 'delete';
  key: string;           // Object key (path)
  bucket: string;        // Bucket name
  size?: number;         // File size (not present for deletes)
  etag?: string;         // ETag hash (not present for deletes)
  lastModified?: Date;   // Last modified date (not present for deletes)
}
```

Watch Use Cases

- **Sync systems** - Detect changes to trigger synchronization
- **Analytics** - Track file uploads/modifications in real-time
- **Notifications** - Alert users when their files are ready
- **Processing pipelines** - Trigger workflows on new file uploads
- **Backup systems** - Detect changes for incremental backups
- **Audit logs** - Track all bucket activity

Directory Management

SmartBucket provides powerful directory-like operations for organizing your files:

```
// Get base directory
const baseDir = await bucket.getBaseDirectory();

// List directories and files
const directories = await baseDir.listDirectories();
const files = await baseDir.listFiles();

console.log(`Found ${directories.length} directories`);
console.log(`Found ${files.length} files`);

// Navigate subdirectories
const subDir = await baseDir.getSubDirectoryByName('projects/2024');

// Create nested file
```

```
await subDir.fastPut({
  path: 'report.pdf',
  contents: reportBuffer
});

// Get directory tree structure
const tree = await subDir.getTreeArray();

// Get directory path
console.log('Base path:', subDir.getBasePath()); // "projects/2024/"

// Create empty file as placeholder
await subDir.createEmptyFile('placeholder.txt');

// Check existence
const fileExists = await subDir.fileExists({ path: 'report.pdf' });
const dirExists = await baseDir.directoryExists('projects');
```

Streaming Operations

Handle large files efficiently with streaming support:

Download Streams

```
// Node.js stream (for file I/O, HTTP responses, etc.)
const nodeStream = await bucket.fastGetStream(
  { path: 'large-video.mp4' },
  'nodestream'
);

// Pipe to file
import * as fs from 'node:fs';
nodeStream.pipe(fs.createWriteStream('local-video.mp4'));

// Pipe to HTTP response
app.get('/download', async (req, res) => {
  const stream = await bucket.fastGetStream(
    { path: 'file.pdf' },
    'nodestream'
```

```
  );
  res.setHeader('Content-Type', 'application/pdf');
  stream.pipe(res);
});

// Web stream (for modern browser/Deno environments)
const webStream = await bucket.fastGetStream(
  { path: 'large-file.zip' },
  'webstream'
);
```

Upload Streams

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

// Stream upload from file
const readStream = fs.createReadStream('big-data.csv');
await bucket.fastPutStream({
  path: 'uploads/big-data.csv',
  readableStream: readStream,
  nativeMetadata: {
    'content-type': 'text/csv',
    'x-custom-header': 'my-value'
  }
});
```

Reactive Streams with RxJS

```
// Get file as ReplaySubject for reactive programming
const replaySubject = await bucket.fastGetReplaySubject({
  path: 'data/sensor-readings.json'
});

// Multiple subscribers can consume the same data
replaySubject.subscribe({
  next: (chunk) => processChunk(chunk),
  complete: () => console.log('Stream complete')
});
```

```
replaySubject.subscribe({
  next: (chunk) => logChunk(chunk)
});
```

File Locking

Prevent concurrent modifications with built-in file locking:

```
const baseDir = await bucket.getBaseDirectory();
const file = await baseDir.getFile({ path: 'important-config.json' });

// Lock file for 10 minutes
await file.lock({ timeoutMillis: 600000 });

// Check lock status via metadata
const metadata = await file.getMetaData();
const isLocked = await metadata.checkLocked();

// Get lock info
const lockInfo = await metadata.getLockInfo();
console.log(`Lock expires: ${new Date(lockInfo.expires)}`);

// Unlock when done
await file.unlock();

// Force unlock (even if locked by another process)
await file.unlock({ force: true });
```

Lock use cases:

- Prevent concurrent writes during critical updates
- Protect configuration files during deployment
- Coordinate distributed workers
- Ensure data consistency

Metadata Management

Attach and manage rich metadata for your files:

```
const baseDir = await bucket.getBaseDirectory();
const file = await baseDir.getFile({ path: 'document.pdf' });

// Get metadata handler
const metadata = await file.getMetaData();

// Store custom metadata (can be any JSON-serializable value)
await metadata.storeCustomMetaData({
  key: 'author',
  value: 'John Doe'
});

await metadata.storeCustomMetaData({
  key: 'tags',
  value: ['important', 'quarterly-report', '2024']
});

await metadata.storeCustomMetaData({
  key: 'workflow',
  value: { status: 'approved', approvedBy: 'jane@company.com' }
});

// Retrieve metadata
const author = await metadata.getCustomMetaData({ key: 'author' });

// Delete metadata
await metadata.deleteCustomMetaData({ key: 'workflow' });

// Check if file has any metadata
const hasMetadata = await file.hasMetaData();

// Get file type detection
const fileType = await metadata.getFileType({ useFileExtension: true });

// Get file type from magic bytes (more accurate)
const detectedType = await metadata.getFileType({ useMagicBytes: true });

// Get file size
const size = await metadata.getSizeInBytes();
```

Metadata use cases:

- Track file ownership and authorship
- Add tags and categories for search
- Store processing status or workflow state
- Enable rich querying and filtering
- Maintain audit trails

Trash & Recovery

SmartBucket includes an intelligent trash system for safe file deletion and recovery:

```
const baseDir = await bucket.getBaseDirectory();
const file = await baseDir.getFile({ path: 'important-data.xlsx' });

// Move to trash instead of permanent deletion
await file.delete({ mode: 'trash' });

// Permanent deletion (use with caution!)
await file.delete({ mode: 'permanent' });

// Access trash
const trash = await bucket.getTrash();
const trashDir = await trash.getTrashDir();
const trashedFiles = await trashDir.listFiles();

// Restore from trash
const trashedFile = await baseDir.getFile({
  path: 'important-data.xlsx',
  getFromTrash: true
});

await trashedFile.restore({ useOriginalPath: true });

// Or restore to a different location
await trashedFile.restore({
  toPath: 'recovered/important-data.xlsx'
});
```

Trash features:

- Recover accidentally deleted files
- Preserves original path in metadata
- Tracks deletion timestamp
- List and inspect trashed files

Advanced Features

File Statistics

```
// Get detailed file statistics
const stats = await bucket.fastStat({ path: 'document.pdf' });
console.log(`Size: ${stats.ContentLength} bytes`);
console.log(`Last modified: ${stats.LastModified}`);
console.log(`ETag: ${stats.ETag}`);
console.log(`Content type: ${stats.ContentType}`);
```

Magic Bytes Detection

Detect file types by examining the first bytes (useful for validation):

```
// Read first bytes for file type detection
const magicBytes = await bucket.getMagicBytes({
  path: 'mystery-file',
  length: 16
});

// Or from a File object
const baseDir = await bucket.getBaseDirectory();
const file = await baseDir.getFile({ path: 'image.jpg' });
const magic = await file.getMagicBytes({ length: 4 });

// Check file signatures
if (magic[0] === 0xFF && magic[1] === 0xD8) {
  console.log('This is a JPEG image');
} else if (magic[0] === 0x89 && magic[1] === 0x50) {
  console.log('This is a PNG image');
}
```

JSON Data Operations

```
const baseDir = await bucket.getBaseDirectory();
const file = await baseDir.getFile({ path: 'config.json' });

// Read JSON data
const config = await file.getJsonData();

// Update JSON data
config.version = '2.0';
config.updated = new Date().toISOString();
config.features.push('newFeature');

await file.writeJsonData(config);
```

Directory & File Type Detection

```
// Check if path is a directory
const isDir = await bucket.isDirectory({ path: 'uploads/' });

// Check if path is a file
const isFile = await bucket.isFile({ path: 'uploads/document.pdf' });
```

Clean Bucket Contents

```
// Remove all files and directories (use with caution!)
await bucket.cleanAllContents();
```

Cloud Provider Support

SmartBucket works seamlessly with all major S3-compatible providers:

Provider	Status	Notes
AWS S3	Supported	Native S3 API
MinIO	Supported	Self-hosted, perfect for development
DigitalOcean Spaces	Supported	Cost-effective S3-compatible
Backblaze B2	Supported	Very affordable storage
Wasabi	Supported	High-performance hot storage
Google Cloud Storage	Supported	Via S3-compatible API

Provider	Status	Notes
Cloudflare R2	Supported	Zero egress fees
Any S3-compatible	Supported	Works with any S3-compatible provider

Configuration examples:

```
// AWS S3
const awsStorage = new SmartBucket({
  accessKey: process.env.AWS_ACCESS_KEY_ID,
  accessSecret: process.env.AWS_SECRET_ACCESS_KEY,
  endpoint: 's3.amazonaws.com',
  region: 'us-east-1',
  useSsl: true
});

// MinIO (local development)
const minioStorage = new SmartBucket({
  accessKey: 'minioadmin',
  accessSecret: 'minioadmin',
  endpoint: 'localhost',
  port: 9000,
  useSsl: false
});

// DigitalOcean Spaces
const doStorage = new SmartBucket({
  accessKey: process.env.DO_SPACES_KEY,
  accessSecret: process.env.DO_SPACES_SECRET,
  endpoint: 'nyc3.digitaloceanspaces.com',
  region: 'nyc3',
  useSsl: true
});

// Backblaze B2
const b2Storage = new SmartBucket({
  accessKey: process.env.B2_KEY_ID,
  accessSecret: process.env.B2_APPLICATION_KEY,
  endpoint: 's3.us-west-002.backblazeb2.com',
  region: 'us-west-002',
```

```
  useSsl: true
});

// Cloudflare R2
const r2Storage = new SmartBucket({
  accessKey: process.env.R2_ACCESS_KEY_ID,
  accessSecret: process.env.R2_SECRET_ACCESS_KEY,
  endpoint: `${process.env.R2_ACCOUNT_ID}.r2.cloudflarestorage.com`,
  region: 'auto',
  useSsl: true
});
```

Advanced Configuration

```
// Environment-based configuration with @push.rocks/qenv
import { Qenv } from '@push.rocks/qenv';

const qenv = new Qenv('./', './.nogit/');

const smartBucket = new SmartBucket({
  accessKey: await qenv.getEnvVarOnDemand('S3_ACCESS_KEY'),
  accessSecret: await qenv.getEnvVarOnDemand('S3_SECRET'),
  endpoint: await qenv.getEnvVarOnDemand('S3_ENDPOINT'),
  port: parseInt(await qenv.getEnvVarOnDemand('S3_PORT')),
  useSsl: await qenv.getEnvVarOnDemand('S3_USE_SSL') === 'true',
  region: await qenv.getEnvVarOnDemand('S3_REGION')
});
```

Testing

```
# Run all tests
pnpm test

# Run specific test file
pnpm tstest test/test.watcher.node.ts --verbose

# Run tests with log file
```

```
pnpm test --logfile
```

Error Handling Best Practices

SmartBucket uses a **strict-by-default** approach - methods throw errors instead of returning null:

```
// Check existence first
if (await bucket.fastExists({ path: 'file.txt' })) {
  const content = await bucket.fastGet({ path: 'file.txt' });
  process(content);
}

// Try/catch for expected failures
try {
  const file = await bucket.fastGet({ path: 'might-not-exist.txt' });
  process(file);
} catch (error) {
  console.log('File not found, using default');
  useDefault();
}

// Explicit overwrite control
try {
  await bucket.fastPut({
    path: 'existing-file.txt',
    contents: 'new data',
    overwrite: false // Explicitly fail if exists
  });
} catch (error) {
  console.log('File already exists');
}
```

Best Practices

1. **Always use strict mode** for critical operations to catch errors early
2. **Check existence first** with `fastExists()`, `bucketExists()`, etc. before operations
3. **Implement proper error handling** for network and permission issues
4. **Use streaming** for large files (>100MB) to optimize memory usage
5. **Leverage metadata** for organizing and searching files

6. **Enable trash mode** for important data to prevent accidental loss
7. **Lock files** during critical operations to prevent race conditions
8. **Use async generators** for listing large buckets to avoid memory issues
9. **Set explicit overwrite flags** to prevent accidental file overwrites
10. **Use the watcher** for real-time synchronization and event-driven architectures

Performance Tips

- **Listing**: Use async generators or cursors for buckets with >10,000 objects
- **Uploads**: Use streams for files >100MB
- **Downloads**: Use streams for files you'll process incrementally
- **Metadata**: Cache metadata when reading frequently
- **Locking**: Keep lock durations as short as possible
- **Glob patterns**: Be specific to reduce objects scanned
- **Watching**: Use appropriate `pollIntervalMs` based on your change frequency

License and Legal Information

Trademarks

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 11:10:01 UTC by foss.global Team
Updated 2026-03-28 12:16:39 UTC by foss.global Team