rustic dev documentation

the rustic authors

Contents

1 Introduction **2**

2 Contributing **3**

3 Frequently asked technical questions **8**

4 Installation **9**
4.1 Nightly builds . . . 10

5 Design **13**
5.1 Terminology . . . 13
5.2 Repository Format . . . 13
5.2.1 Repository Layout . . . 14
5.2.2 S3 Legacy Layout . . . 15
5.3 Pack Format . . . 15
5.3.1 Unpacked Data Format . . . 16
5.4 Indexing . . . 16
5.5 Keys, Encryption and MAC . . . 17
5.6 Snapshots . . . 19
5.7 Trees and Data . . . 20
5.8 Locks . . . 21
5.9 Read and Write Ordering . . . 22
5.10 Backups and Deduplication . . . 23
5.11 Threat Model . . . 23

6 UX-Design **25**
6.1 Personas . . . 25
6.2 User stories . . . 27
6.3 Use cases . . . 29
6.4 Requirements . . . 32

7 Development guide **35**
7.1 Glossary . . . 36
7.2 Release Process . . . 40
7.3 Testing . . . 42

8 Talks **43**

Chapter 1

Introduction

Rustic is a fast and secure backup program. In the following sections, we will present typical workflows, starting with installing, preparing a new repository, and making the first backup.

Note: Parts of this documentation are shamelessly copied from the restic documentation and then adapted to rustic as most workflows work in rustic exactly like restic. See also the restic documentation for more information about restic.

Contact

You can ask questions in the Discussions or have a look at the FAQ.

Contact	Where?
Issue Tracker	GitHub Issues
Discord	
Discussions	GitHub Discussions

Chapter 2

Contributing

Thank you for your interest in contributing to the `rustic` ecosystem!

We appreciate your help in making this project better.

Table of Contents

- Code of Conduct
- How to Contribute
  - Reporting Bugs
  - Issue and Pull Request Labels
  - Suggesting Enhancements
  - Code Style and Formatting
  - Testing
  - Submitting Pull Requests
    - Rebasing and other workflows
- Development Setup
- License

Code of Conduct

Please review and abide by the general Rust Community Code of Conduct when contributing to this project. In the future, we might create our own Code of Conduct and supplement it at this location.

How to Contribute

Reporting Bugs

If you find a bug, please open an issue on GitHub and provide as much detail as possible. Include steps to reproduce the bug and the expected behavior.

Issue and Pull Request labels

Our Issues and Pull Request labels follow the official Rust style:

```
A - Area
C - Category
D - Diagnostic
E - Call for participation
F - Feature
I - Issue e.g. I-crash
M - Meta
O - Operating systems
P - priorities e.g. P-{low, medium, high, critical}
PG - Project Group
perf - Performance
S - Status e.g. S-{blocked, experimental, inactive}
T - Team relevancy
WG - Working group
```

Suggesting Enhancements

If you have an idea for an enhancement or a new feature, we'd love to hear it! Open an issue on GitHub and describe your suggestion in detail.

Code Style and Formatting

We follow the Rust community's best practices for code style and formatting. Before submitting code changes, please ensure your code adheres to these guidelines:

- Use `rustfmt` to format your code. You can run it with the following command:

  ```
  cargo fmt --all
  ```

- Use `dprint` to format text in markdown, toml, and json. You can install it with `cargo install dprint`/`scoop install dprint` and run it with the following command in the repository root:

  ```
  dprint fmt
  ```

- Use `Clippy` — a collection of lints to catch common mistakes. If you haven't installed your Rust toolchain with `rustup`, please install Clippy first. Then run Clippy with the following command in the crate root:

  ```
  cargo clippy --all -- -D warnings
  ```

- Write clear and concise code with meaningful, self-describing variable and function names. This tells the reader **what** the code does.
- Write clear and consise comments to tell the reader **why** you chose to implement it that way and **which** problem it solves.

Testing

We value code quality and maintainability. If you are adding new features or making changes, please include relevant unit tests. Run the test suite with:

```
cargo test --workspace
```

or check the testing guide for more information which tools we provide for making developing `rustic` easier.

Make sure all tests pass before submitting your changes. PRs containing tests have a much higher probability of getting merged (fast).

We expect PRs especially ones that introduce new features to contain tests for the new code.

Besides that, we welcome PRs which increase the general test coverage of the project. You can check the testing guide for more information.

We appreciate tests in every form: be it *unit*, *doc* or *integration* tests (chose depending on your use case).

If you want to implement some *fuzzing* or *benchmarking*, that is also highly appreciated.

Submitting Pull Requests

To contribute code changes, follow these steps:

1. **Fork** the repository.
2. **Create** a new branch with a descriptive name:

   ```
   git checkout -b feature/your-feature-name
   ```

3. **Check** and **Commit** your changes:

   ```
   just pre-commit
   git commit -m "Add your meaningful commit message here"
   ```

4. **Push** your branch to your forked repository:

   ```
   git push origin feature/your-feature-name
   ```

5. **Open** a Pull Request (PR) to our repository. Please include a detailed description of the changes and reference any related issues.

`Release early and often!` also applies to pull requests

Consider drafting a Pull request early in the development process, so we can follow your progress and can give early feedback.

Once your PR is submitted, it will be reviewed by the maintainers. We may suggest changes or ask for clarifications before merging.

IMPORTANT NOTE Please don't force push commits in your branch, in order to keep commit history and make it easier for us to see changes between reviews.

Make sure to Allow edits of maintainers (under the text box) in the PR so people can actually collaborate on things or fix smaller issues themselves.

Rebasing and other workflows

(taken from: openage on rebasing)

Rebasing is 'moving' your commits to a different parent commit.

In other words: *Cut off* your branch from its tree, and *attach it* somewhere else.

There's two main applications:

- If you based your work on a older main (so old that stuff can't be automatically merged), you can rebase to move your commits to the current upstream main:

```
# update the upstream remote to receive new commits
git fetch upstream

# be on your feature branch (you probably are)
git checkout my-awesome-feature

# make backup (you never know, you know?)
git branch my-awesome-feature-backup

# rebase: put your commits on top of upstream's main
git rebase -m upstream/main
```

- If you want to fix an older commit of yours, or merge several commits into a single one (**squash** them), rebase interactively. We ***don't*** want to have a commit history like this:
  - `add stuff`
  - `fix typo in stuff`
  - `fix compilation`
  - `change stuff a bit`
  - and so on...

rebase in practice `git log --graph --oneline` shows your commit history as graph. To make some changes in that graph, you do an **interactive rebase**:

```
git rebase -i -m upstream/main
```

With this command, your new ”base” is `upstream/main` and you can then change any of your branch's commits.

`-i` will open an interactive editor where you can choose actions for each individual commit:

- re-order commits
- drop commits by deleting their line
- squash/fixup (”meld”) your commits
- reword a commit message
- stop rebasing at a commit to edit (`--amend`) it manually

Just follow the messages on screen.

Changing commits with `amend` and `fixup` There's also `git commit --amend` which is a ”mini-rebase” that modifies just the last commit with your current changes by `git add`. It just skips the creation of a new commit and instead melds the changes into the last one you made.

If you want to update a single commit in the range `[upstream/main, current HEAD]` which is not the last commit:

- `edit stuff you wanna change in some previous commit`
- `git add changed_stuff`
- `git commit --fixup $hash_of_commit_to_be_fixed`
- `git rebase --autosquash -i -m upstream/main`

Pushing changes After you have rebased stuff (”rewritten history”) that had already been pushed, git will not accept your pushes because they're not simple fast-forwards:

- The commit contents and the parent commit have changed as you updated the commit, therefore the commit hash changed, too.

  - If somebody used those commits, they will keep a copy and have a hard time updating to your updated version (because they ”use” the old hashes).
  - Update your pull request branch with your re-written history!
- **force push** is the standard way of overwriting your development work with the fixed and mergeable version of your contribution!
  - Why? You changed the commits, so you want the old ones to be deleted! You can use any of:
  - `git push origin +my-awesome-feature`
  - `git push origin -f my-awesome-feature`
  - `git push origin --force my-awesome-feature`

Some extra tutorials on `git rebase`:

- Atlassian's Git Tutorial
- Pro Git book
- `man git-rebase`

Development Setup

If you want to set up a local development environment, follow the steps in the development guide file - which is currently being worked on.

License

By contributing to `rustic` or any crates contained in this repository, you agree that your contributions will be licensed under:

- Apache License, Version 2.0
- MIT license.

Unless you explicitly state otherwise, any contribution intentionally submitted for inclusion in the work by you, as defined in the Apache-2.0 license, shall be dual licensed as above, without any additional terms or conditions.

Chapter 3

Frequently asked technical questions

Memory Requirements

> **Question**: *You mention a "huge decrease in memory requirement"; are you streaming blobs to the data store without keeping them in memory and just keeping the hashes in memory? How does the memory footprint look like?*

The largest memory consuming part is the index which is kept in-memory by restic and rustic. For restic, however, the index took about 3-4 times the theoretical size - maybe due to Golangs garbage collector additions or other Golang-specific stuff.

In rustic, there are two basic improvements:

- the memory consumption is exactly as expected as Rust allows to directly define the in-memory structures
- For some operations, not all index information is needed. Rustic allows several kinds of index types to be loaded into memory (only trees, only ids, full index) and uses only the type needed for a given command. For instance with backup you need the list of all present blob IDs, but not the position within the repository - rustic thus only saves the IDs in memory for the backup command and therefore uses less memory.

Chapter 4

Installation

- Official Binaries
  - Stable Releases
  - Unstable Builds
- From Source

Official Binaries

Stable Releases

cargo-binstall

```
cargo binstall rustic-rs
```

Windows

Scoop

```
scoop install rustic
```

You can download the latest stable release versions of rustic from the rustic release page. These builds are considered stable and releases are made regularly in a controlled manner.

There's both pre-compiled binaries for different platforms as well as the source code available for download. Just download and run the one matching your system.

Once downloaded, the official binaries can be updated in place using the `rustic self-update` command (needs rustic 0.3.1 or later):

```
$ rustic  self-update
Checking target-arch... x86_64-unknown-linux-musl
Checking current version... v0.3.0-dev
Checking latest released version... v0.3.1
New release found! v0.3.0-dev --> v0.3.1
New release is *NOT* compatible

rustic release status:
    * Current exe: "/usr/local/bin/rustic"
    * New exe release: "rustic-v0.3.1-x86_64-unknown-linux-musl.tar.gz"
    * New exe download url:
↪   "https://api.github.com/repos/rustic/rustic/releases/assets/75146490"
```

```
The new release will be downloaded/extracted and the existing binary will be
 ↪  replaced.
Do you want to continue? [Y/n] Y
Downloading...
[00:00:00] [========================================] 4.29MiB/4.29MiB (0s)
 ↪  Done
Extracting archive... Done
Replacing binary file... Done
Update status: `0.3.1`!
```

Note: Please be aware that the user executing the `rustic self-update` command must have the permission to replace the rustic binary.

Unstable Builds

Another option is to use the nightly builds for the main branch, available on the nightly download page. These too are pre-compiled and ready to run, and a new version is built every night from the main branch of various repositories.

From Source

Beware: This installs the latest development version, which might be unstable.

rustic is written in Rust and you need a current Rust version.

In order to build rustic from source, execute the following steps:

Github

```
cargo install --git https://github.com/rustic-rs/rustic.git rustic-rs
```

crates.io

You can also directly install the latest crate from crates.io.

```
cargo install rustic-rs
```

Cross-compile

You can easily cross-compile rustic for all supported platforms, make sure that the cross-compile toolchain is installed for your target. Then run the build for your chosen target like this

```
cargo build --target aarch64-unknown-linux-gnu --release
```

4.1 Nightly builds

Nightly builds of rustic's, rustic_server's, and rustic_scheduler's main branch are available here for download.

WARNING: These builds are not guaranteed to be stable, and may contain bugs. Use at your own risk.

Verification

Minisign/Rsign2

Install

- `rsign2` with `cargo install rsign2` or
- `minisign` with `scoop install minisign` (on Windows, check other installation instructions here).

Run

```
rsign verify <filename>.tar.gz \
  -x <filename>.tar.gz.sig \
  -P RWSWSCEJEEacVeCy0va71hlrVtiW8YzMzOyJeso0Bfy/ZXq5OryWi/8T
```

PGP

Download our public key or copy and paste it from below:

`wget https://github.com/rustic-rs/nightly/raw/main/pub/pgp.pub`

Check the fingerprint:

`12B7166D9FD59124416952E34018C5DE3BF8C081`

against the output of: `gpg --show-keys <PUBLIC_KEY_FILE>`

Import the key with `gpg --import <PUBLIC_KEY_FILE>`

Verify the signature with `gpg --verify <filename>.tar.gz.asc <filename>.tar.gz`

The output should say "Good Signature".

Note: We use the `.asc` extension for the files because `.sig` was already taken for supporting `minisign` used by `cargo-binstall`.

Status

Download matrix

Platform	rustic	rustic_scheduler	rustic_server
Linux x86_64 / gnu	□ #□ □	□ #□ □	□ #□ □
Linux x86_64 / musl (static)	□ #□ □	□ #□ □	□ #□ □
Linux i686 / gnu	□ #□ □	□ #□ □	□ #□ □

Platform	rustic	rustic_scheduler	rustic_server
Linux aarch64 / gnu	□ #□ □	□ #□ □	□ #□ □
Linux armv7 / raspberry pi	□ #□ □	□ #□ □	□ #□ □
MacOS x86_64	□ #□ □	□ #□ □	□ #□ □
MacOS aarch64	□ #□ □	□ #□ □	n.a., #6
Windows x86_64 / msvc (exp)	□ #□ □	□ #□ □	□ #□ □
Windows x86_64 / gnu (exp)	□ #□ □	□ #□ □	□ #□ □

Chapter 5

Design

This chapter describes the original design of restic. It is intended for people who want to understand how restic works internally, or who want to contribute to restic or rustic.

5.1 Terminology

This section introduces terminology used in this document.

Repository: All data produced during a backup is sent to and stored in a repository in a structured form, for example in a file system hierarchy with several subdirectories. A repository implementation must be able to fulfill a number of operations, e.g. list the contents.

Blob: A Blob combines a number of data bytes with identifying information like the SHA-256 hash of the data and its length.

Pack: A Pack combines one or more Blobs, e.g. in a single file.

Snapshot: A Snapshot stands for the state of a file or directory that has been backed up at some point in time. The state here means the content and metadata like the name and modification time for the file or the directory and its contents.

Storage ID: A storage ID is the SHA-256 hash of the content stored in the repository. This ID is required in order to load the file from the repository.

5.2 Repository Format

All data is stored in a restic repository. A repository is able to store data of several different types, which can later be requested based on an ID. This so-called ”storage ID” is the SHA-256 hash of the content of a file. All files in a repository are only written once and never modified afterwards. Writing should occur atomically to prevent concurrent operations from reading incomplete files. This allows accessing and even writing to the repository with multiple clients in parallel. Only the `prune` operation removes data from the repository.

Repositories consist of several directories and a top-level file called `config`. For all other files stored in the repository, the name for the file is the lower case hexadecimal representation of the storage ID, which is the SHA-256 hash of the file's contents. This allows for easy verification of files for accidental modifications, like disk read errors, by simply running the program `sha256sum` on the file and comparing its output to the file name. If the prefix of a filename is unique amongst all the other files in the same directory, the prefix may be used instead of the complete filename.

Apart from the files stored within the `keys` directory, all files are encrypted with AES-256 in counter mode (CTR). The integrity of the encrypted data is secured by a Poly1305-AES message authentication code (sometimes also referred to as a "signature").

In the first 16 bytes of each encrypted file the initialisation vector (IV) is stored. It is followed by the encrypted data and completed by the 16 byte MAC. The format is: `IV || CIPHERTEXT || MAC`. The complete encryption overhead is 32 bytes. For each file, a new random IV is selected.

The file `config` is encrypted this way and contains a JSON document like the following:

```
{
    "version": 2,
    "id": "5956a3f67a6230d4a92cefb29529f10196c7d92582ec305fd71ff6d331d6271b",
    "chunker_polynomial": "25b468838dcb75"
}
```

After decryption, restic first checks that the version field contains a version number that it understands, otherwise it aborts. At the moment, the version is expected to be 1 or 2. The list of changes in the repository format is contained in the section "Changes" below.

The field `id` holds a unique ID which consists of 32 random bytes, encoded in hexadecimal. This uniquely identifies the repository, regardless if it is accessed via a remote storage backend or locally. The field `chunker_polynomial` contains a parameter that is used for splitting large files into smaller chunks (see below).

5.2.1 Repository Layout

The `local` and `sftp` backends are implemented using files and directories stored in a file system. The directory layout is the same for both backend types and is also used for all other remote backends.

The basic layout of a repository is shown here:

```
/tmp/restic-repo
├── config
├── data
│   ├── 21
│   │   └── 2159dd48f8a24f33c307b750592773f8b71ff8d11452132a7b2e2a6a01611be1
│   ├── 32
│   │   └── 32ea976bc30771cebad8285cd99120ac8786f9ffd42141d452458089985043a5
│   ├── 59
│   │   └── 59fe4bcde59bd6222eba87795e35a90d82cd2f138a27b6835032b7b58173a426
│   ├── 73
│   │   └── 73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c
│   [...]
├── index
│   ├── c38f5fb68307c6a3e3aa945d556e325dc38f5fb68307c6a3e3aa945d556e325d
│   └── ca171b1b7394d90d330b265d90f506f9984043b342525f019788f97e745c71fd
├── keys
│   └── b02de829beeb3c01a63e6b25cbd421a98fef144f03b9a02e46eff9e2ca3f0bd7
├── locks
├── snapshots
│   └── 22a5af1bdc6e616f8a29579458c49627e01b32210d09adb288d1ecda7c5711ec
└── tmp
```

A local repository can be initialized with the `restic init` command, e.g.:

```
restic -r /tmp/restic-repo init
```

The local and sftp backends will auto-detect and accept all layouts described in the following sections, so that remote repositories mounted locally e.g. via fuse can be accessed. The layout auto-detection can be overridden by specifying the option `-o local.layout=default`, valid values are `default` and `s3legacy`. The option for the sftp backend is named `sftp.layout`, for the s3 backend `s3.layout`.

5.2.2 S3 Legacy Layout

Unfortunately during development the Amazon S3 backend uses slightly different paths (directory names use singular instead of plural for `key`, `lock`, and `snapshot` files), and the pack files are stored directly below the `data` directory. The S3 Legacy repository layout looks like this:

```
/config
/data
  ├── 2159dd48f8a24f33c307b750592773f8b71ff8d11452132a7b2e2a6a01611be1
  ├── 32ea976bc30771cebad8285cd99120ac8786f9ffd42141d452458089985043a5
  ├── 59fe4bcde59bd6222eba87795e35a90d82cd2f138a27b6835032b7b58173a426
  ├── 73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c
[...]
/index
  ├── c38f5fb68307c6a3e3aa945d556e325dc38f5fb68307c6a3e3aa945d556e325d
  └── ca171b1b7394d90d330b265d90f506f9984043b342525f019788f97e745c71fd
/key
  └── b02de829beeb3c01a63e6b25cbd421a98fef144f03b9a02e46eff9e2ca3f0bd7
/lock
/snapshot
  └── 22a5af1bdc6e616f8a29579458c49627e01b32210d09adb288d1ecda7c5711ec
```

The S3 backend understands and accepts both forms, new backends are always created with the default layout for compatibility reasons.

5.3 Pack Format

All files in the repository except Key and Pack files just contain raw data, stored as `IV || Ciphertext || MAC`. Pack files may contain one or more Blobs of data.

A Pack's structure is as follows:

```
EncryptedBlob1 || ... || EncryptedBlobN || EncryptedHeader || Header_Length
```

At the end of the Pack file is a header, which describes the content. The header is encrypted and authenticated. `Header_Length` is the length of the encrypted header encoded as a four byte integer in little-endian encoding. Placing the header at the end of a file allows writing the blobs in a continuous stream as soon as they are read during the backup phase. This reduces code complexity and avoids having to re-write a file once the pack is complete and the content and length of the header is known.

All the blobs (`EncryptedBlob1`, `EncryptedBlobN` etc.) are authenticated and encrypted independently. This enables repository reorganisation without having to touch the encrypted Blobs. In addition it also allows efficient indexing, for only the header needs to be read in order to find out which Blobs are contained in the Pack. Since the header is authenticated, authenticity of the header can be checked without having to read the complete Pack.

After decryption, a Pack's header consists of the following elements:

```
Type_Blob1 || Data_Blob1 ||
[...]
Type_BlobN || Data_BlobN ||
```

The Blob type field is a single byte. What follows it depends on the type. The following Blob types are defined:

Type	Meaning	Data
0b00	data blob	Length(encrypted_blob) or Hash(plaintext_blob)
0b01	tree blob	Length(encrypted_blob) or Hash(plaintext_blob)
0b10	compressed data blob	Length(encrypted_blob) or Length(plaintext_blob) or Hash(plaintext_blob)
0b11	compressed tree blob	Length(encrypted_blob) or Length(plaintext_blob) or Hash(plaintext_blob)

This is enough to calculate the offsets for all the Blobs in the Pack. The length fields are encoded as four byte integers in little-endian format. In the Data column, `Length(plaintext_blob)` means the length of the decrypted and uncompressed data a blob consists of.

All other types are invalid, more types may be added in the future. The compressed types are only valid for repository format version 2. Data and tree blobs may be compressed with the zstandard compression algorithm.

In repository format version 1, data and tree blobs should be stored in separate pack files. In version 2, they must be stored in separate files. Compressed and non-compress blobs of the same type may be mixed in a pack file.

For reconstructing the index or parsing a pack without an index, first the last four bytes must be read in order to find the length of the header. Afterwards, the header can be read and parsed, which yields all plaintext hashes, types, offsets and lengths of all included blobs.

5.3.1 Unpacked Data Format

Individual files for the index, locks or snapshots are encrypted and authenticated like Data and Tree Blobs, so the outer structure is `IV || Ciphertext || MAC` again. In repository format version 1 the plaintext always consists of a JSON document which must either be an object or an array.

Repository format version 2 adds support for compression. The plaintext now starts with a header to indicate the encoding version to distinguish it from plain JSON and to allow for further evolution of the storage format: `encoding_version || data` The `encoding_version` field is encoded as one byte. For backwards compatibility the encoding versions '[' (0x5b) and '{' (0x7b) are used to mark that the whole plaintext (including the encoding version byte) should treated as JSON document.

For new data the encoding version is currently always 2. For that version `data` contains a JSON document compressed using the zstandard compression algorithm.

5.4 Indexing

Index files contain information about Data and Tree Blobs and the Packs they are contained in and store this information in the repository. When the local cached index is not accessible any more, the index files can be downloaded and used to reconstruct the index. The file encoding is described in the ”Unpacked Data Format” section. The plaintext consists of a JSON document like the following:

```
{
  "supersedes": [
```

```
    "ed54ae36197f4745ebc4b54d10e0f623eaaaedd03013eb7ae90df881b7781452"
  ],
  "packs": [
    {
      "id":
       ↪  "73d04e6125cf3c28a299cc2f3cca3b78ceac396e4fcf9575e34536b26782413c",
      "blobs": [
        {
          "id":
           ↪  "3ec79977ef0cf5de7b08cd12b874cd0f62bbaf7f07f3497a5b1bbcc8cb39b1ce",
          "type": "data",
          "offset": 0,
          "length": 38,
          // no 'uncompressed_length' as blob is not compressed
        },
        {
          "id":
           ↪  "9ccb846e60d90d4eb915848add7aa7ea1e4bbabfc60e573db9f7bfb2789afbae",
          "type": "tree",
          "offset": 38,
          "length": 112,
          "uncompressed_length": 511,
        },
        {
          "id":
           ↪  "d3dc577b4ffd38cc4b32122cabf8655a0223ed22edfd93b353dc0c3f2b0fdf66",
          "type": "data",
          "offset": 150,
          "length": 123,
          "uncompressed_length": 234,
        }
      ]
    }, [...]
  ]
}
```

This JSON document lists Packs and the blobs contained therein. In this example, the Pack `73d04e61` contains two data Blobs and one Tree blob, the plaintext hashes are listed afterwards. The `length` field corresponds to `Length(encrypted_blob)` in the pack file header. Field `uncompressed_length` is only present for compressed blobs and therefore is never present in version 1. It is set to the value of `Length(blob)`.

The field `supersedes` lists the storage IDs of index files that have been replaced with the current index file. This happens when index files are repacked, for example when old snapshots are removed and Packs are recombined.

There may be an arbitrary number of index files, containing information on non-disjoint sets of Packs. The number of packs described in a single file is chosen so that the file size is kept below 8 MiB.

5.5 Keys, Encryption and MAC

All data stored by restic in the repository is encrypted with AES-256 in counter mode and authenticated using Poly1305-AES. For encrypting new data first 16 bytes are read from a cryptographically secure

pseudo-random number generator as a random nonce. This is used both as the IV for counter mode and the nonce for Poly1305. This operation needs three keys: A 32 byte for AES-256 for encryption, a 16 byte AES key and a 16 byte key for Poly1305. For details see the original paper The Poly1305-AES message-authentication code by Dan Bernstein. The data is then encrypted with AES-256 and afterwards a message authentication code (MAC) is computed over the ciphertext, everything is then stored as IV || CIPHERTEXT || MAC.

The directory `keys` contains key files. These are simple JSON documents which contain all data that is needed to derive the repository's master encryption and message authentication keys from a user's password. The JSON document from the repository can be pretty-printed for example by using the Python module `json` (shortened to increase readability):

```
python -mjson.tool /tmp/restic-repo/keys/b02de82*
```

```
{
    "hostname": "kasimir",
    "username": "fd0",
    "kdf": "scrypt",
    "N": 65536,
    "r": 8,
    "p": 1,
    "created": "2015-01-02T18:10:13.48307196+01:00",
    "data":
     ↪  "tGwYeKoM0C4j4/9DFrVEmMGAldvEn/+iKC3te/QE/6ox/V4qz58FUOgMa0Bb1cIJ6asrypCx/Ti/pRXCPH
    "salt":
     ↪  "uW4fEI1+IOzj7ED9mVor+yTSJFd68DGlGOeLgJELYsTU5ikhG/83/+jGd4KKAaQdSrsfzrdOhAMftTSih5
}
```

When the repository is opened by restic, the user is prompted for the repository password. This is then used with `scrypt`, a key derivation function (KDF), and the supplied parameters (`N`, `r`, `p` and `salt`) to derive 64 key bytes. The first 32 bytes are used as the encryption key (for AES-256) and the last 32 bytes are used as the message authentication key (for Poly1305-AES). These last 32 bytes are divided into a 16 byte AES key `k` followed by 16 bytes of secret key `r`. The key `r` is then masked for use with Poly1305 (see the paper for details).

Those keys are used to authenticate and decrypt the bytes contained in the JSON field `data` with AES-256 and Poly1305-AES as if they were any other blob (after removing the Base64 encoding). If the password is incorrect or the key file has been tampered with, the computed MAC will not match the last 16 bytes of the data, and restic exits with an error. Otherwise, the data yields a JSON document which contains the master encryption and message authentication keys for this repository (encoded in Base64). The command `restic cat masterkey` can be used as follows to decrypt and pretty-print the master key:

```
restic -r /tmp/restic-repo cat masterkey
```

```
{
    "mac": {
        "k": "evFWd9wWlndL9jc501268g==",
        "r": "E9eEDnSJZgqwTOkDtOp+Dw=="
    },
    "encrypt": "UQCqa0lKZ94PygPxMRqkePTZnHRYh1k1pX2k2lM2v3Q="
}
```

All data in the repository is encrypted and authenticated with these master keys. For encryption, the AES-256 algorithm in Counter mode is used. For message authentication, Poly1305-AES is used as described above.

A repository can have several different passwords, with a key file for each. This way, the password

can be changed without having to re-encrypt all data.

5.6 Snapshots

A snapshot represents a directory with all files and sub-directories at a given point in time. For each backup that is made, a new snapshot is created. A snapshot is a JSON document that is stored in a file below the directory `snapshots` in the repository. It uses the file encoding described in the "Unpacked Data Format" section. The filename is the storage ID. This string is unique and used within restic to uniquely identify a snapshot.

The command `restic cat snapshot` can be used as follows to decrypt and pretty-print the contents of a snapshot file:

```
restic -r /tmp/restic-repo cat snapshot 251c2e58
enter password for repository:

{
    "time": "2015-01-02T18:10:50.895208559+01:00",
    "tree":
     ↪  "2da81727b6585232894cfbb8f8bdab8d1eccd3d8f7c92bc934d62e62e618ffdf",
    "dir": "/tmp/testdata",
    "hostname": "kasimir",
    "username": "fd0",
    "uid": 1000,
    "gid": 100,
    "tags": [
        "NL"
    ]
}
```

Here it can be seen that this snapshot represents the contents of the directory `/tmp/testdata`. The most important field is `tree`. When the meta data (e.g. the tags) of a snapshot change, the snapshot needs to be re-encrypted and saved. This will change the storage ID, so in order to relate these seemingly different snapshots, a field `original` is introduced which contains the ID of the original snapshot, e.g. after adding the tag `DE` to the snapshot above it becomes:

```
$ restic -r /tmp/restic-repo cat snapshot 22a5af1b
enter password for repository:

{
    "time": "2015-01-02T18:10:50.895208559+01:00",
    "tree":
     ↪  "2da81727b6585232894cfbb8f8bdab8d1eccd3d8f7c92bc934d62e62e618ffdf",
    "dir": "/tmp/testdata",
    "hostname": "kasimir",
    "username": "fd0",
    "uid": 1000,
    "gid": 100,
    "tags": [
        "NL",
        "DE"
    ],
    "original":
     ↪  "251c2e5841355f743f9d4ffd3260bee765acee40a6229857e32b60446991b837"
}
```

Once introduced, the `original` field is not modified when the snapshot's meta data is changed again.

All content within a restic repository is referenced according to its SHA-256 hash. Before saving, each file is split into variable sized Blobs of data. The SHA-256 hashes of all Blobs are saved in an ordered list which then represents the content of the file.

In order to relate these plaintext hashes to the actual location within a Pack file, an index is used. If the index is not available, the header of all data Blobs can be read.

5.7 Trees and Data

A snapshot references a tree by the SHA-256 hash of the JSON string representation of its contents. Trees and data are saved in pack files in a subdirectory of the directory `data`.

The command `restic cat blob` can be used to inspect the tree referenced above (piping the output of the command to `jq .` so that the JSON is indented):

```
$ restic -r /tmp/restic-repo cat blob
 ↪  2da81727b6585232894cfbb8f8bdab8d1eccd3d8f7c92bc934d62e62e618ffdf | jq .
enter password for repository:
```


```
{
    "nodes": [
        {
            "name": "testdata",
            "type": "dir",
            "mode": 493,
            "mtime": "2014-12-22T14:47:59.912418701+01:00",
            "atime": "2014-12-06T17:49:21.748468803+01:00",
            "ctime": "2014-12-22T14:47:59.912418701+01:00",
            "uid": 1000,
            "gid": 100,
            "user": "fd0",
            "inode": 409704562,
            "content": null,
            "subtree":
             ↪  "b26e315b0988ddcd1cee64c351d13a100fedbc9fdbb144a67d1b765ab280b4dc"
        }
    ]
}
```


A tree contains a list of entries (in the field `nodes`) which contain meta data like a name and timestamps. When the entry references a directory, the field `subtree` contains the plain text ID of another tree object.

When the command `restic cat blob` is used, the plaintext ID is needed to print a tree. The tree referenced above can be dumped as follows:

```
$ restic -r /tmp/restic-repo cat blob
 ↪  b26e315b0988ddcd1cee64c351d13a100fedbc9fdbb144a67d1b765ab280b4dc
enter password for repository:
```


```
{
    "nodes": [
      {
        "name": "testfile",
        "type": "file",
```

```
        "mode": 420,
        "mtime": "2014-12-06T17:50:23.34513538+01:00",
        "atime": "2014-12-06T17:50:23.338468713+01:00",
        "ctime": "2014-12-06T17:50:23.34513538+01:00",
        "uid": 1000,
        "gid": 100,
        "user": "fd0",
        "inode": 416863351,
        "size": 1234,
        "links": 1,
        "content": [
          "50f77b3b4291e8411a027b9f9b9e64658181cc676ce6ba9958b95f268cb1109d"
        ]
      },
      [...]
    ]
}
```

This tree contains a file entry. This time, the `subtree` field is not present and the `content` field contains a list with one plain text SHA-256 hash.

The command `restic cat blob` can also be used to extract and decrypt data given a plaintext ID, e.g. for the data mentioned above:

```
$ restic -r /tmp/restic-repo cat blob
 ↪  50f77b3b4291e8411a027b9f9b9e64658181cc676ce6ba9958b95f268cb1109d |
 ↪  sha256sum
enter password for repository:
50f77b3b4291e8411a027b9f9b9e64658181cc676ce6ba9958b95f268cb1109d  -
```

As can be seen from the output of the program `sha256sum`, the hash matches the plaintext hash from the map included in the tree above, so the correct data has been returned.

5.8 Locks

The restic repository structure is designed in a way that allows parallel access of multiple instance of restic and even parallel writes. However, there are some functions that work more efficient or even require exclusive access of the repository. In order to implement these functions, restic processes are required to create a lock on the repository before doing anything.

Locks come in two types: Exclusive and non-exclusive locks. At most one process can have an exclusive lock on the repository, and during that time there must not be any other locks (exclusive and non-exclusive). There may be multiple non-exclusive locks in parallel.

A lock is a file in the subdir `locks` whose filename is the storage ID of the contents. It is stored in the file encoding described in the "Unpacked Data Format" section and contains the following JSON structure:

```
{
    "time": "2015-06-27T12:18:51.759239612+02:00",
    "exclusive": false,
    "hostname": "kasimir",
    "username": "fd0",
    "pid": 13607,
    "uid": 1000,
```

```
    "gid": 100
}
```

The field `exclusive` defines the type of lock. When a new lock is to be created, restic checks all locks in the repository. When a lock is found, it is tested if the lock is stale, which is the case for locks with timestamps older than 30 minutes. If the lock was created on the same machine, even for younger locks it is tested whether the process is still alive by sending a signal to it. If that fails, restic assumes that the process is dead and considers the lock to be stale.

When a new lock is to be created and no other conflicting locks are detected, restic creates a new lock, waits, and checks if other locks appeared in the repository. Depending on the type of the other locks and the lock to be created, restic either continues or fails.

5.9 Read and Write Ordering

The repository format allows writing (e.g. backup) and reading (e.g. restore) to happen concurrently. As the data for each snapshot in a repository spans multiple files (snapshot, index and packs), it is necessary to follow certain rules regarding the order in which files are read and written. These ordering rules also guarantee that repository modifications always maintain a correct repository even if the client or the storage backend crashes for example due to a power cut or the (network) connection between both is interrupted.

The correct order to access data in a repository is derived from the following set of invariants that must be maintained at **any time** in a correct repository. *Must* in the following is a strict requirement and will lead to data loss if not followed. *Should* will require steps to fix a repository (e.g. rebuilding the index) if not followed, but should not cause data loss. *existing* means that the referenced data is **durably** stored in the repository.

- A snapshot *must* only reference an existing tree blob.
- A reachable tree blob *must* only reference tree and data blobs that exist (recursively). *Reachable* means that the tree blob is reachable starting from a snapshot.
- An index *must* only reference valid blobs in existing packs.
- All blobs referenced by a snapshot *should* be listed in an index.

This leads to the following recommended order to store data in a repository. First, pack files, which contain data and tree blobs, must be written. Then the indexes which reference blobs in these already written pack files. And finally the corresponding snapshots.

Note that there is no need for a specific write order of data and tree blobs during a backup as the blobs only become referenced once the corresponding snapshot is uploaded.

Reading data should follow the opposite order compared to writing. Only once a snapshot was written, it is guaranteed that all required data exists in the repository. This especially means that the list of snapshots to read should be collected before loading the repository index. The other way round can lead to a race condition where a recently written snapshot is loaded but not its accompanying index, which results in a failure to access the snapshot's tree blob.

For removing or rewriting data from a repository the following rules must be followed, which are derived from the above invariants.

- A client removing data *must* acquire an exclusive lock first to prevent conflicts with other clients.
- A pack *must* be removed from the referencing index before it is deleted.
- Rewriting a pack *must* write the new pack, update the index (add an updated index and delete the old one) and only then delete the old pack.

5.10 Backups and Deduplication

For creating a backup, restic scans the source directory for all files, sub-directories and other entries. The data from each file is split into variable length Blobs cut at offsets defined by a sliding window of 64 bytes. The implementation uses Rabin Fingerprints for implementing this Content Defined Chunking (CDC). An irreducible polynomial is selected at random and saved in the file `config` when a repository is initialized, so that watermark attacks are much harder.

Files smaller than 512 KiB are not split, Blobs are of 512 KiB to 8 MiB in size. The implementation aims for 1 MiB Blob size on average.

For modified files, only modified Blobs have to be saved in a subsequent backup. This even works if bytes are inserted or removed at arbitrary positions within the file.

5.11 Threat Model

The design goals for restic include being able to securely store backups in a location that is not completely trusted (e.g., a shared system where others can potentially access the files) or even modify or delete them in the case of the system administrator.

General assumptions:

- The host system a backup is created on is trusted. This is the most basic requirement, and it is essential for creating trustworthy backups.
- The user uses an authentic copy of restic.
- The user does not share the repository password with an attacker.
- The restic backup program is not designed to protect against attackers deleting files at the storage location. There is nothing that can be done about this. If this needs to be guaranteed, get a secure location without any access from third parties.
- The whole repository is re-encrypted if a key is leaked. With the current key management design, it is impossible to securely revoke a leaked key without re-encrypting the whole repository.
- Advances in cryptography attacks against the cryptographic primitives used by restic (i.e., AES-256-CTR-Poly1305-AES and SHA-256) have not occurred. Such advances could render the confidentiality or integrity protections provided by restic useless.
- Sufficient advances in computing have not occurred to make brute-force attacks against restic's cryptographic protections feasible.

The restic backup program guarantees the following:

- Unencrypted content of stored files and metadata cannot be accessed without a password for the repository. Everything except the metadata included for informational purposes in the key files is encrypted and authenticated. The cache is also encrypted to prevent metadata leaks.
- Modifications to data stored in the repository (due to bad RAM, broken harddisk, etc.) can be detected.
- Data that has been tampered will not be decrypted.

With the aforementioned assumptions and guarantees in mind, the following are examples of things an adversary could achieve in various circumstances.

An adversary with read access to your backup storage location could:

- Attempt a brute force password guessing attack against a copy of the repository (please use strong passwords with sufficient entropy).
- Infer which packs probably contain trees via file access patterns.
- Infer the size of backups by using creation timestamps of repository objects.

An adversary with network access could:

- Attempt to DoS the server storing the backup repository or the network connection between client and server.
- Determine from where you create your backups (i.e., the location where the requests originate).
- Determine where you store your backups (i.e., which provider/target system).
- Infer the size of backups by observing network traffic.

The following are examples of the implications associated with violating some of the aforementioned assumptions.

An adversary who compromises (via malware, physical access, etc.) the host system making backups could:

- Render the entire backup process untrustworthy (e.g., intercept password, copy files, manipulate data).
- Create snapshots (containing garbage data) which cover all modified files and wait until a trusted host has used `forget` often enough to remove all correct snapshots.
- Create a garbage snapshot for every existing snapshot with a slightly different timestamp and wait until certain `forget` configurations have been run, thereby removing all correct snapshots at once.

An adversary with write access to your files at the storage location could:

- Delete or manipulate your backups, thereby impairing your ability to restore files from the compromised storage location.
- Determine which files belong to what snapshot (e.g., based on the timestamps of the stored files). When only these files are deleted, the particular snapshot vanishes and all snapshots depending on data that has been added in the snapshot cannot be restored completely. Restic is not designed to detect this attack.

An adversary who compromises a host system with append-only (read+write allowed, delete+overwrite denied) access to the backup repository could:

- Capture the password and decrypt backups from the past and in the future (see the ”leaked key” example below for related information).
- Render new backups untrustworthy *after* the host has been compromised (due to having complete control over new backups). An attacker cannot delete or manipulate old backups. As such, restoring old snapshots created *before* a host compromise remains possible.
- Potentially manipulate the use of the `forget` command into deleting all legitimate snapshots, keeping only bogus snapshots added by the attacker. Ransomware might try this in order to leave only one option to get your data back: paying the ransom. For safe use of `forget`, please see the corresponding documentation on removing backup snapshots and append-only mode.

An adversary who has a leaked (decrypted) key for a repository could:

- Decrypt existing and future backup data. If multiple hosts backup into the same repository, an attacker will get access to the backup data of every host. Note that since the local encryption key gives access to the master key, a password change will not prevent this. Changing the master key can currently only be done using the `copy` command, which moves the data into a new repository with a new master key, or by making a completely new repository and new backup.

Chapter 6

UX-Design

To have a clearer picture of the UX-Design for rustic, we have collected some of the most important use cases and user stories. These are living documents and will be updated as we go along.

6.1 Personas

1. Sophie - The Small Business Owner

- **Age:** 32
- **Background:** Owns a small e-commerce website.
- **Tech-savviness:** Moderate; familiar with basic software and e-commerce tools.
- **Needs:** A reliable backup system, easy-to-use interface, and affordable rates.
- **Pain Points:** Previous backup solutions were too complicated or pricey. Lost data once due to a failed backup.
- **Goals:** Ensure that all her business data is backed up securely, easily retrieve data when needed.

2. Arjun - The IT Manager

- **Age:** 41
- **Background:** IT Manager for a mid-size company with 200+ employees.
- **Tech-savviness:** High; knows ins and outs of most enterprise software.
- **Needs:** A scalable backup solution, deduplication to save space, advanced security features.
- **Pain Points:** Dealing with multiple backup tools for different types of data, storage limitations.
- **Goals:** Centralize the backup process, ensure quick data restoration, and streamline IT operations.

3. Elena - The Freelancer

- **Age:** 29
- **Background:** Freelance graphic designer.
- **Tech-savviness:** Moderate; uses specific tools for her design work.
- **Needs:** A simple backup solution that can handle large files, automatic backups.
- **Pain Points:** Lost a project once due to a hard drive crash, cloud storage subscriptions becoming too expensive.
- **Goals:** Protect her portfolio and client projects, ensure she doesn't lose time redoing lost work.

4. Martin - The Retiree

- **Age:** 67
- **Background:** Retired school principal, now writes memoirs and short stories.
- **Tech-savviness:** Basic; uses a computer primarily for word processing and web browsing.
- **Needs:** A straightforward backup tool, clear instructions, preferably with visual guides.
- **Pain Points:** Complicated tech jargon, fears of losing his writings.
- **Goals:** Keep his digital memories and writings safe for future generations.

5. Nadia - The Academic Researcher

- **Age:** 35
- **Background:** PhD in Biology, conducts extensive research with large datasets.
- **Tech-savviness:** Moderate-to-High; adept with research tools and databases.
- **Needs:** A backup solution that can handle diverse file types, easy data retrieval, metadata preservation.
- **Pain Points:** Managing diverse datasets, ensuring long-term data preservation.
- **Goals:** Secure her research, share datasets without loss of fidelity, compliance with academic data storage norms.

6. Liam - The Digital Artist

- **Age:** 23
- **Background:** Digital artist and animator who regularly works on large projects.
- **Tech-savviness:** High; familiar with advanced design tools and software.
- **Needs:** A backup system that supports large file formats, auto-syncing, and version control.
- **Pain Points:** Slow upload speeds, backup solutions not supporting all file formats.
- **Goals:** Protect his art, easily retrieve older versions of his work.

7. Maria - The Non-Profit Organizer

- **Age:** 45
- **Background:** Runs a non-profit focused on education for underprivileged children.
- **Tech-savviness:** Moderate; basic office software and social media.
- **Needs:** Affordable backup solutions, secure storage for donor data, easy collaboration tools.
- **Pain Points:** Limited budget, ensuring data privacy.
- **Goals:** Keep organizational data safe, access backups from remote locations.

8. Takumi - The Independent Game Developer

- **Age:** 28
- **Background:** Develops indie games, works from home.
- **Tech-savviness:** Very high; proficient in various programming languages and game engines.
- **Needs:** Large storage capacities, high-speed backups, integration with development environments.
- **Pain Points:** Version conflicts, loss of game assets or code.
- **Goals:** Streamlined development process, protection against data loss.

9. Fatima - The College Student

- **Age:** 20
- **Background:** Studying Computer Science.
- **Tech-savviness:** Moderate-to-high; regularly uses coding platforms and academic tools.

- **Needs:** Affordable backup solutions, mobile access to backups.
- **Pain Points:** Previous backup solutions were too complicated or unreliable.
- **Goals:** Secure academic projects, quick data restoration during exam times.

10. Albert - The Travel Blogger

- **Age:** 38
- **Background:** Travels the world and documents his experiences through photos and writing.
- **Tech-savviness:** Moderate; skilled in content creation tools.
- **Needs:** Backup solutions accessible from different parts of the world, mobile-friendly.
- **Pain Points:** Lost photos/videos from past trips, connectivity issues.
- **Goals:** Safeguard memories, quick uploads even from remote areas.

11. Aisha - The Integrative Developer

- **Age**: 28
- **Background**: Software Developer at a mid-sized tech company
- **Tech-savviness**: Advanced
- **Needs**: Robust and reliable library, clear documentation, integration examples, extensive customization.
- **Pain Points**: poor documentation, integration conflicts, performance issues.
- **Goals**: Aims to develop scalable and feature-rich applications. Seeks reliable and efficient external libraries to enhance her software's capabilities without reinventing the wheel.

6.2 User stories

Sophie - The Small Business Owner

- **As** Sophie,
  - **I want** to schedule daily backups of my e-commerce site automatically,
  - **So that** I don't have to remember to do it manually.
- **As** Sophie,
  - **I want** a one-click restoration feature,
  - **So that** I can quickly retrieve lost or corrupted data.

Arjun - The IT Manager

- **As** Arjun,
  - **I want** to initiate backups for multiple systems in my company simultaneously,
  - **So that** I can ensure all systems are backed up efficiently.
- **As** Arjun,
  - **I want** a centralized dashboard to monitor backup health,
  - **So that** I can ensure all backups are completed successfully and address any issues promptly.

Elena - The Freelancer

- **As** Elena,

  - **I want** to backup large design files without compression losses,
  - **So that** my designs retain their quality.
- **As** Elena,
  - **I want** the backup solution to auto-sync with my design directory,
  - **So that** all changes are automatically backed up.

Martin - The Retiree

- **As** Martin,
  - **I want** step-by-step guidance when backing up data,
  - **So that** I am confident I'm doing it right.
- **As** Martin,
  - **I want** a deduplicated backup of my photo collection,
  - **So that** I don't waste storage space on repeated photos.

Nadia - The Academic Researcher

- **As** Nadia,
  - **I want** to backup large research datasets efficiently,
  - **So that** I don't waste time and resources.
- **As** Nadia,
  - **I want** to generate shareable links for specific backup datasets,
  - **So that** I can share data with fellow researchers easily.

Liam - The Digital Artist

- **As** Liam,
  - **I want** to backup entire project folders including all assets,
  - **So that** I have a comprehensive backup of my work.
- **As** Liam,
  - **I want** high-speed uploads for backups,
  - **So that** I don't waste time waiting for large files to upload.

Maria - The Non-Profit Organizer

- **As** Maria,
  - **I want** a secure backup for donor data,
  - **So that** the privacy and trust of our donors are maintained.
- **As** Maria,
  - **I want** to access the backup system remotely during field visits,
  - **So that** I can manage backups even when I'm away from the office.

Takumi - The Independent Game Developer

- **As** Takumi,
  - **I want** the backup solution to integrate with my code repositories,
  - **So that** code backups are streamlined.
- **As** Takumi,
  - **I want** to store different versions of my game builds,
  - **So that** I can easily revert or compare between versions.

Fatima - The College Student

- **As** Fatima,
  - **I want** to backup my academic projects at the end of each semester,
  - **So that** I have a cumulative record of my academic progress.
- **As** Fatima,
  - **I want** a rapid restore feature,
  - **So that** I can quickly get back crucial data during exam times or project submissions.

Albert - The Travel Blogger

- **As** Albert,
  - **I want** to quickly backup photos and videos from my travels,
  - **So that** my memories are preserved irrespective of device failures.
- **As** Albert,
  - **I want** to schedule regular backups of my blog content,
  - **So that** my written content, along with user comments, are always safe.

Aisha - The Integrative Developer

- **As** a software developer,
  - **I want** to easily integrate the backup library.
  - **So that** I can provide backup functionalities in my software without building from scratch.
- **As** a software developer,
  - **I want** to customize the backup settings using the library
  - **So that** I can ensure my software's users have flexibility in their backup preferences.

6.3 Use cases

Sophie - The Small Business Owner

- **Use-Case 1: Automated Backup Scheduling**
  - **Scenario:** Sophie wants to ensure that her e-commerce site's data is backed up regularly without manual intervention.
  - **Actions:** Sophie sets up an automated backup schedule for daily backups at midnight.
- **Use-Case 2: Transactional Data Recovery**

  - **Scenario:** There's a glitch on her e-commerce platform and she loses a day's worth of transactional data.
  - **Actions:** Sophie uses the backup solution to quickly recover the specific day's transaction data.

Arjun - The IT Manager

- **Use-Case 1: Centralized Management**
  - **Scenario:** Arjun wants to oversee all backups from one central dashboard.
  - **Actions:** Arjun logs into the central dashboard to monitor and manage backups for the entire company.
- **Use-Case 2: Access Level Management**
  - **Scenario:** Arjun needs to give specific team members permission to initiate but not delete backups.
  - **Actions:** Arjun sets role-based permissions for various team members.

Elena - The Freelancer

- **Use-Case 1: Large File Handling**
  - **Scenario:** Elena has designed a high-resolution poster for a client.
  - **Actions:** Elena uses the backup solution to securely store the large design file without quality loss.
- **Use-Case 2: Portfolio Protection**
  - **Scenario:** Elena's local hard drive crashes.
  - **Actions:** Elena retrieves her entire portfolio from the backup solution, ensuring she doesn’t lose her work.

Martin - The Retiree

- **Use-Case 1: Photo Album Backup**
  - **Scenario:** Martin wants to preserve old family photos digitally.
  - **Actions:** Martin scans and backs up photo albums, relying on deduplication to avoid redundant backups of similar photos.
- **Use-Case 2: Guided Backup Wizard**
  - **Scenario:** Martin is unsure of how to backup his new writings.
  - **Actions:** Martin uses a step-by-step backup guide to securely store his documents.

Nadia - The Academic Researcher

- **Use-Case 1: Research Data Deduplication**
  - **Scenario:** Nadia has multiple versions of a large dataset with small differences.
  - **Actions:** Nadia backs up her datasets, with the solution ensuring deduplication to save space.
- **Use-Case 2: Data Collaboration**
  - **Scenario:** Nadia wants to share her dataset backup with a fellow researcher.
  - **Actions:** Nadia generates a secure, shareable link to her backup dataset.

Liam - The Digital Artist

- **Use-Case 1: Version Control for Art**
  - **Scenario:** Liam is working on a digital art piece and makes significant changes daily.
  - **Actions:** Liam uses the backup solution's version control to save daily iterations of his art.
- **Use-Case 2: Bulk Backup for Projects**
  - **Scenario:** Liam completes a big project with multiple art assets.
  - **Actions:** Liam uses the backup solution to bulk-backup the entire project folder.

Maria - The Non-Profit Organizer

- **Use-Case 1: Donor Data Protection**
  - **Scenario:** Maria collects sensitive donor data during a fundraising event.
  - **Actions:** Maria uses the backup solution to securely store this data with strong encryption.
- **Use-Case 2: Field Data Backup**
  - **Scenario:** Maria collects data during a field visit in a remote area.
  - **Actions:** Maria initiates an offline backup, which gets uploaded once she's back in connectivity.

Takumi - The Independent Game Developer

- **Use-Case 1: Codebase Backup with Deduplication**
  - **Scenario:** Takumi has different game versions with shared assets.
  - **Actions:** Takumi backs up his games, with the solution deduplicating shared assets across versions.
- **Use-Case 2: Rapid Recovery after a Bug**
  - **Scenario:** A new code update introduces a major bug.
  - **Actions:** Takumi quickly recovers the last stable version of his game from the backup.

Fatima - The College Student

- **Use-Case 1: Semester-wise Academic Backup**
  - **Scenario:** At the end of every semester, Fatima wants to backup all her academic projects and notes.
  - **Actions:** Fatima organizes her data semester-wise and initiates backups accordingly.
- **Use-Case 2: Emergency Data Restoration**
  - **Scenario:** Fatima accidentally deletes her thesis a week before submission.
  - **Actions:** Fatima quickly restores the deleted thesis from her backup solution.

Albert - The Travel Blogger

- **Use-Case 1: Backup from Remote Locations**
  - **Scenario:** Albert is traveling in a remote location with limited connectivity but wants to backup his new blogs and photos.
  - **Actions:** Albert initiates an offline backup, which queues up and uploads once he finds a stable connection.

- **Use-Case 2: Blog Platform Integration**
  - **Scenario:** Albert wants his blog platform to auto-backup every week.
  - **Actions:** Albert integrates his blogging platform with the backup solution for automated weekly backups.

Aisha - The Integrative Developer

- **Use Case 1: Integrate Backup Library**
  - **Scenario**: Aisha wants to integrate the deduplicated backup solution library into her software.
  - **Actions**:
    1. Search and find the library in a package repository.
    2. Review the documentation.
    3. Install the library.
    4. Implement initial backup functionalities in her software.
    5. Test the integration.
- **Use Case 2: Customize Backup Settings**
  - **Scenario**: Aisha aims to customize the backup settings to suit her software's needs.
  - **Actions**:
    1. Review library documentation for customization options.
    2. Modify settings like backup frequency, destination, and deduplication method.
    3. Run tests to ensure the settings are applied correctly.

6.4 Requirements

1. **Automated Backup Scheduling**
   - The system should allow for automated backups at specified intervals (daily, weekly, monthly, etc.).
2. **Easy Restoration**
   - Users should be able to quickly and easily restore data from the backups, preferably with a one-click solution.
3. **Selective Backup**
   - The solution should allow users to select specific data sets or files for backup.
4. **Centralized Management Dashboard**
   - A unified dashboard where users (especially those managing multiple systems) can oversee all backups.
5. **User Access Control & Role-based Permissions**
   - Different levels of access and permissions to ensure that data integrity and security are maintained.
6. **Large File Handling without Compression Loss**
   - The system should handle the backup of large files while maintaining their original quality.
7. **Auto-sync Feature**
   - A feature that automatically syncs and backs up directories or specified data.
8. **Guided Backup Process**

- For users who are not tech-savvy, a step-by-step guide to backing up data.

9. **Efficient Deduplication Process**
    - To ensure storage is used efficiently by avoiding duplicate backups of identical files.
10. **Shareable Backup Links**
    - Generation of secure links to backup datasets/files for sharing.
11. **Version Control**
    - The system should keep track of different versions of backed-up files, allowing users to revert if needed.
12. **Bulk Backup**
    - Allow users to backup large sets of data or entire directories at once.
13. **Strong Encryption and Data Security**
    - All backed-up data, especially sensitive information, should be encrypted and stored securely.
14. **Remote Access Capability**
    - Users should be able to access the backup solution from different locations or devices.
15. **Integration with Other Platforms**
    - E.g., Code repositories, blogging platforms, etc.
16. **Rapid Data Recovery**
    - Fast restore features for urgent data recovery needs.
17. **Offline Backup Queuing**
    - Allow backups to be initiated offline and processed once connectivity is available.
18. **High-speed Data Uploads**
    - Efficiently handle and quickly upload large files or data sets.
19. **Documentation Clarity**
    - The backup solution library should have clear and comprehensive documentation. This will enable developers like Aisha to integrate the library seamlessly.
20. **Customization Options**
    - The library should provide a variety of settings and configurations that developers can adjust, ensuring it fits various software's unique requirements.
21. **Test Environment**
    - The backup solution should offer a sandbox or test mode. This helps developers to test the library's functionalities without affecting real data.
22. **Active Support Community**
    - An active forum or community where developers can post questions, share experiences, and offer solutions. Quick response times are crucial.
23. **Error Handling**
    - The library should be designed with robust error handling and should provide descriptive error messages. This aids developers in diagnosing integration or functionality issues.

24. **Compatibility**
    - Ensure the core library is compatible with a range of programming languages and frameworks to cater to a diverse developer base.
25. **Scalability**
    - As software may grow and handle more data, the backup library should be scalable to accommodate increasing data without performance bottlenecks.

Chapter 7

Development guide

Work in progress ...

`cargo xtask`

We utilize `cargo xtask` to provide additional functionality for the development process.

Usage

Currently it supports the following functionalities:

Command	Description
`cargo xtask help`	Show an overview over all or the help of the given subcommand(s).
`cargo xtask bloat-deps`	Show biggest crates in release build using cargo-bloat.
`cargo xtask bloat-time`	Show longest times taken in release build using cargo-bloat.
`cargo xtask coverage`	Generate code coverage report.
`cargo xtask install-deps`	Install dependencies for the development process.
`cargo xtask timings`	Show longest times taken in release build using cargo-bloat.

Maskfile

We utilize `mask` to provide additional functionality for the development process.

Installation

Install `mask` with:

```
cargo install mask
```

or by using `scoop`:

```
scoop install mask
```

Usage

A maskfile is self-documenting, because it is written in Markdown. You can view the maskfile here.

7.1 Glossary

Table of Contents

- Archiver
- Backend
- Beta Builds
- Blob
- Cache
- Chunker
- Cold Repository
- Completions
- Compression
- Data Pack
- Description
- Environmental Variables
- Filter
- Glob
- Hot Repository
- Library (rustic)
- Labels and Tags
- Nodes
- One-File-System
- Pack and Pack Size
- Packer
- Parallelization
- Parent
- Profiles and Configs
- Repository
- Repack
- Retry
- Scripting
- Snapshot
- Source
- Sub-trees
- Storage ID
- Trees
- Warm-up

Archiver

Note: This content is still under review/in progress.

An archiver is a tool or software that creates archives, which are collections of files and data bundled together.

Backend

Note: This content is still under review/in progress.

The backend refers to the server-side of a software application. It is responsible for processing requests from the frontend, handling data storage, and executing complex tasks.

Beta Builds

Beta builds are pre-release versions of the software made available to a limited audience for testing and feedback before the official release. They are used to identify and fix bugs and gather user input. Beta-builds of `rustic` can be found in the rustic-beta repository.

Blob

A Blob combines a number of data bytes with identifying information like the SHA-256 hash of the data and its length.

Cache

Note: This content is still under review/in progress.

A cache is a temporary storage mechanism used to store frequently accessed data. It helps to speed up operations and reduce the need for repetitive computations.

Chunker

Note: This content is still under review/in progress.

The chunker is a component responsible for breaking down data into smaller, manageable pieces or chunks, typically for more efficient processing and storage.

Cold Repository

Note: This content is still under review/in progress.

The term cold repository refers to a storage location for data that is less frequently accessed and may require extra steps to retrieve.

Completions

Completions refer to the suggestions or options provided by the terminal to help users complete commands or input based on what they have typed so far. You can generate completions using `rustic completions <SH>` where `<SH>` is one of `bash, fish, zsh, powershell`. More information on shell completions you can find in the FAQ.

Compression

Compression is the process of reducing the size of data or files to save storage space and improve transfer speeds.

Data Pack

Note: This content is still under review/in progress.

A data pack is a collection of data files bundled together for distribution or deployment.

Description

Note: This content is still under review/in progress.

A description is metadata that provides additional information or context about an item within the project.

Environmental Variables

Environmental variables are settings or values that affect the behavior of software running on a particular system. They provide configuration and runtime information. Available settings that are influenced by environment variables can be found here.

Filter

A filter is used to selectively process or display certain data while excluding the rest based on specific criteria. It helps to narrow down relevant information. `rustic` integrates the Rhai scripting language for snapshot filtering. ‘--filter-fn“ allows to implement your own snapshot filter in a 'Rust-like' syntax (Rust closure).

Glob

A glob is a pattern used to match and select files or directories based on specific rules. It is commonly used for file manipulation and searching.

Hot Repository

Note: This content is still under review/in progress.

The term hot repository refers to a storage location for data that is readily available for access and immediate use.

Library (rustic)

In the context of `rustic`, a library refers to a set of reusable code components that can be used by other parts of the project or by external projects. The main library being used is `rustic_core`. The crate resides in `crates/rustic_core/`. You can use it in your project with `cargo add rustic_core`.

Labels and Tags

Labels, tags, and descriptions are metadata elements used to categorize and annotate items within the project. They provide context and make it easier to search and manage repositories.

Nodes

Note: This content is still under review/in progress.

Nodes are individual elements within a hierarchical structure, such as trees or sub-trees.

One-File-System

Note: This content is still under review/in progress.

One-File-System is a mode or option that restricts file operations to a single filesystem, preventing access to other filesystems on the system.

Pack and Pack Size

A Pack combines one or more Blobs, e.g. in a single file. In this project, packing refers to the process of combining multiple files or data into a single pack. Pack size refers to the size of such packages.

Packer

The packer is a component responsible for combining multiple Blobs into packs.

Parallelization

Parallelization is the technique of dividing tasks into smaller units and executing them simultaneously on multiple processors or cores to achieve faster performance.

Parent

Note: This content is still under review/in progress.

In this project, a parent refers to a higher-level element or entity that has one or more subordinate components or children associated with it.

Profiles and Configs

Profiles and configurations are settings that determine the behavior of the application based on different scenarios or environments. They allow the project to adapt to varying conditions. A profile consists of a set of configurations. More information on profiles and configuration options can be found here.

Repository

A repository is often abbreviated as ”repo”. All data produced during a backup is sent to and stored in a repository in a structured form, for example in a file system hierarchy with several subdirectories. It is a version-controlled storage location for backups. It allows multiple snapshots to be stored, tracks changes, manages the history of files and directories, and is able to list its contents.

Repack

Note: This content is still under review/in progress.

Repack refers to the process of recreating a package or archive, often with some changes or updates.

Retry

Retry is the act of attempting an operation again if it previously failed. It is a common approach to handle temporary errors or network issues.

Scripting

Scripting refers to the use of scripts, which are small programs written in a scripting language, to automate tasks or execute specific operations. `rustic` integrates the Rhai scripting language for snapshot filtering. ‘--filter-fn“ allows to implement your own snapshot filter in a 'Rust-like' syntax (Rust closure).

Snapshot

A Snapshot stands for the state of a file or directory that has been backed up at some point in time. The state here means the content and metadata like the name and modification time for the file or the directory and its contents.

Source

Note: This content is still under review/in progress.

In the context of this project, the source refers to the backup sources `rustic` is taking snapshots of.

Sub-trees

Note: This content is still under review/in progress.

Sub-trees are lower-level hierarchical structures that are part of a larger hierarchical structure, such as trees.

Storage ID

A storage ID is the SHA-256 hash of the content stored in the repository. This ID is required in order to load the file from the repository.

Trees

Note: This content is still under review/in progress.

Trees are hierarchical structures that organize data with a root node and branches (sub-trees) leading to leaves.

Warm-up

Note: This content is still under review/in progress.

Warm-up is the process of preparing a system or a component for optimal performance. It involves loading necessary data into memory or initializing resources beforehand to reduce startup time.

7.2 Release Process

CLI Releases: `rustic-rs` / `rustic_server` / `rustic_scheduler`

1. **Open a Release PR**: Open a release PR workflow by opening the drop down on the top right `Run workflow`. For `Crate to release` select `rustic-rs` for `Version to release` select the version you want to release with the format `X.Y.Z` (strip the 'v'). This will create a PR with the changes needed to release the version.
2. **Auto-generate completions test fixtures**: If the CLI API tests fail, it means this is definitely a breaking change and needs a major/minor version bump. To update the fixtures, go to the top of the just created PR and copy the branch name, it should be in the form `release/rustic-rs/X.Y.Z`. Then go to the update completions workflow. On the top right under `Run workflow`, run the workflow from the `main` branch and paste the just copied branch name to `PR branch to push to` and click `Run workflow`. This will auto-generate the completions test fixtures for the new version, to make the tests run through.

3. **Generate changelog**: To update the changelog, you need `git-cliff` installed. Run: `git-cliff -u -p .\CHANGELOG.md -t {new_version}`. This will update the changelog with the changes since the last tag.
4. **Documentation**: Check if the documentation needs to be updated somewhere. Take notes if there are any changes needed in the release process and update this document.
5. **Review and Merge the PR**: Review the PR and make sure all checks have passed. Then merge the PR.
6. **Tag the release**: After the PR has been merged, tag the commit on the `main` branch with the version number and push the tag to GitHub. This should make the release workflow run and crate a release for the tag and attach built artifacts to it. It will also copy the changelog to the release notes.
7. **Publishing to crates.io**: After pushing the tag to the `main` branch run `cargo publish` in the repository root.
8. **Publishing to GitHub**: After the release workflow has finished, go to the releases page and find your release draft with the attached artifacts. Edit the release draft to your liking and publish it.
9. **Write an announcement**: Write an announcement for the release and post it on the rustic-rs/rustic discussions.

Library Releases: `rustic_core` / `rustic_backend`

1. **Open a Release PR**: Open a release PR workflow by opening the drop down on the top right `Run workflow`. For `Crate to release` select `rustic-rs` for `Version to release` select the version you want to release with the format `X.Y.Z` (strip the 'v'). This will create a PR with the changes needed to release the version.
2. **Auto-generate Public API test fixtures**: If the API tests fail, it means this is definitely a breaking change and needs a major/minor version bump. To update the fixtures, go to the top of the just created PR and copy the branch name, it should be in the form `release/rustic_core/X.Y.Z`. Then go to the update completions workflow. On the top right under `Run workflow`, run the workflow from the `main` branch and paste the just copied branch name to `PR branch to push to` and click `Run workflow`. This will auto-generate the Public API test fixtures for the new version, to make the tests run through.
3. **Examples**: Check if the examples need to be updated. Run documentation tests to check if the examples are still working. You can run them by running `cargo check --examples` in the `rustic_core` directory.
4. **Documentation**: Check if the documentation needs to be updated somewhere. Take notes if there are any changes needed in the release process and update this document. Run documentation tests to check if the documentation is still working. You can run them by running `cargo test --doc` in the `rustic_core` directory. You can also view the documentation with: `cargo doc --no-deps --all-features --document-private-items --open -p rustic_core`
5. **Generate changelog**: To update the changelog, you need `git-cliff` installed. Run: `git-cliff -u -p .\CHANGELOG.md -t {new_version}`. This will update the changelog with the changes since the last tag.
6. **Review and Merge the PR**: Review the PR and make sure all checks have passed. Then merge the PR.
7. **Tag the release**: After the PR has been merged, tag the commit on the `main` branch with the version number and push the tag to GitHub. This should make the release workflow run and crate a release for the tag. It will also copy the changelog to the release notes.

8. **Publishing to crates.io**: After pushing the tag to the `main` branch run `cargo publish` in the repository root.
9. **Review the Draft Release**: After the release workflow has finished, go to the releases page and find your release draft. Edit the release draft to your liking and publish it.
10. **Write an announcement**: Write an announcement for the release and post it on the rustic-rs/rustic discussions.

7.3 Testing

Completions (CLI-API)

Generate completions fixtures

In the repository root run:

```
cargo run -- completions {SHELL} > tests/completions-fixtures/{SHELL}.txt
```

with `{SHELL}` being one of `bash`, `fish`, `zsh`, or `powershell`.

Test completions

Run the test with:

```
cargo test --test completions -- --ignored
```

Code coverage

VS-Code/VS-Codium Extension

You can live preview the coverage on your code with this extension:

```
Name: Coverage Gutters
Id: ryanluker.vscode-coverage-gutters
Description: Display test coverage generated by lcov or xml - works with many
 ↪  languages
Version: 2.11.0 (at the time of writing)
Publisher: ryanluker
VS Marketplace Link:
 ↪  <https://marketplace.visualstudio.com/items?itemName=ryanluker.vscode-
 ↪  coverage-gutters>
```

Generate with `cargo-tarpaulin`

Install cargo tarpaulin `cargo install cargo-tarpaulin` and run in the repository root:

```
cargo tarpaulin --all -o Lcov --output-dir ./coverage
```

Generate with `grcov`

Install the dependencies for code coverage:

```
cargo xtask install-deps code-coverage
```

Generate a code coverage report into the `/coverage` directory:

```
cargo xtask coverage
```

Chapter 8

Talks

The following talks will be or have been given about `restic`:

- 2021-04-02: The Changelog: Restic has your backup (Podcast)
- 2016-01-31: Lightning Talk at the Go Devroom at FOSDEM 2016
  Brussels, Belgium
- 2016-01-29: **(GER)** restic - Backups mal richtig
  Public lecture at CCC Cologne e.V. in Cologne, Germany
- 2015-08-23: A Solution to the Backup Inconvenience
  Lecture at FROSCON 2015 in Bonn, Germany
- 2015-02-01: Lightning Talk at FOSDEM 2015
  A short introduction (with slightly outdated command line)
- 2015-01-27: **(GER)** Talk about restic at CCC Aachen