rustic user documentation

the rustic authors

Contents

1 Introduction **3**

2 Contributing **4**

3 Frequently asked questions **6**
3.1 Comparison between `rustic` and `restic` 7

4 Installation **17**
4.1 Nightly builds 18

5 How to install shell completions **21**

6 Getting started **23**

7 Init - Preparing a new repository **24**
7.1 Local backend 25
7.2 REST Server 25
7.3 Other Services via rclone 25
7.4 Cold storage 26
7.5 Configuration file 26

8 Backup - Backing up data **28**
8.1 Creating snapshots 28
8.2 File change detection 29
8.3 Dry Run 30
8.4 Excluding Files 30
8.5 Comparing Snapshots 31
8.6 Backup special items and metadata 31
8.7 Reading data from StdIn 32
8.8 Tags for backup 32
8.9 Scheduling backups 32
8.10 Space requirements 32
8.11 Environment Variables 33

9 Miscellaneous - Working with repositories **34**
9.1 Listing snapshots 34
9.2 Copying snapshots between repositories 35
9.3 Filtering snapshots to copy 36
9.4 Ensuring deduplication for copied snapshots 36
9.5 Checking integrity and consistency 37
9.6 Key - Manage repository keys 38
9.7 Upgrading the repository format version 39

10 Restore - Restoring from backup **40**
10.1 Restore using mount 40
10.2 Printing files to stdout 41

11 Forget - Removing backup snapshots **42**
11.1 Remove a single snapshot 42
11.2 Removing snapshots according to a policy 44
11.3 Security considerations in append-only mode 47
11.4 Customize pruning 47
11.5 Recovering from ”no free space” errors 48

12 Stories **50**

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
- 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.

Developer's documentation

For more information about developing around `rustic`, see the developer's documentation.

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 questions

- Can I use rustic with my existing restic repositories?
- What are the differences between rustic and restic?
- Why is rustic written in Rust
- How does rustic work with cold storages like AWS Glacier?
- How does the lock-free prune work?
- You said ”rustic uses less resources than restic” but I'm observing the opposite

Can I use rustic with my existing restic repositories?

Yes, you can. Rustic uses the same repository format as restic, so you can use rustic and restic on the same repository. The only thing you have to take care of is that you don't run prune with restic and rustic at the same time.

What are the differences between rustic and restic?

- Written in Rust instead of golang
- Optimized for small resource usage (in particular memory usage, but also overall CPU usage)
- Philosophy of development (release new features early)
- New features (e.g. hot/cold repositories, lock-free pruning)
- Some commands or options act a bit different or have slightly different syntax

Why is rustic written in Rust

Rust is a powerful language designed to build reliable and efficient software. This is a very good fit for a backup tool.

How does rustic work with cold storages like AWS Glacier?

If you want to use cold storage, make sure you always specify an extra repository `--repo-hot` which contains the hot data. This repository acts like a cache for all metadata, i.e. config/key/snapshot/index files and tree packs. As all commands except `restore` only need to access the metadata, they are fully functional but only need the cold storage to list files while everything else is read from the ”hot

repo". Note that the "hot repo" on its own is not a valid rustic repository. The "cold repo", however, contains all files and is nothing but a standard rustic repository.

If you additionally use a cache, you effectively have a first level cache on your local disc and a second level cache with the "hot repo". Note that the "hot repo" can be also a remote repo, so hot/cold repositories also work for multiple rustic clients backing up to the same repository.

How does the lock-free prune work?

Like the prune within restic, rustic decides for each pack whether to keep it, remove it or repack it. Instead of removing packs, it however only marks the packs to remove in a separate index structure. Packs which are marked for removal are checked if they are really not needed and have been marked long enough ago. Depending on these checks they are either finally removed, recovered or kept in the state of being marked for removal.

This two-phase deletion is needed for rustic to work lock-free: If a `backup` runs parallel to a `prune` run (or `forget --prune`), it could be that prune decides that some blobs can be removed, but the parallel backup uses these blobs for the newly generated snapshot.

The time to hold marked packs should be long enough to guarantee that a possibly parallel backup run has finished in between. It can be set by the `--keep-delete` option and defaults to 23 hours. In any case, packs will be kept marked and only deleted by the next prune run.

Note that there is the option `--instant-delete` which circumvents this two-phase deletion. Only use this option, if you **REALLY KNOW** that there is no parallel access to your repo, else you risk losing data!

You said "rustic uses less resources than restic" but I'm observing the opposite

In general rustic uses less resources, but there may be some exceptions. For instance the crypto libraries of Rust and golang both have optimizations for some CPUs. But it might be that your CPU benefits from a golang optimization which is not present in the Rust implementation. If you observe some unexpected resource usage, please don't hesitate to submit an issue.

3.1 Comparison between `rustic` and `restic`

General differences

	`restic`	`rustic`
programming language	Go	Rust
test coverage	✓	✗ (only few tests implemented)
config profile support	✗ (wrapper tools available)	✓
locking	lock files in repository	lock-free operations, two-phase pruning
cold storage	✗ (no direct support, may work in special cases)	✓ (full support including warm-up of needed data)
in-repo config	✓	✓ (see below for details)
logging	`-v` or `--quiet`, no log-file support	`--log-level`, supports log-file output

	restic	rustic
returns error code	✅	✅
available as library	✅	✅ rustic_core

Storage backends

backend	restic	rustic
local	✅ (built-in)	✅ (built-in)
sftp	✅ (using external ssh command)	✅ (built-in using opendal, windows not supported)
rest	✅ (built-in)	✅ (built-in)
s3	✅ (built-in)	✅ (built-in using opendal)
swift	✅ (built-in)	✅ (built-in using opendal)
b2	✅ (built-in)	✅ (built-in using opendal)
azure	✅ (built-in)	✅ (built-in using opendal)
gs	✅ (built-in)	✅ (built-in using opendal)
dropbox	✅	✅ (built-in using opendal)
ftp	✅	✅ (built-in using opendal)
gdrive	✅	✅ (built-in using opendal)
onedrive	✅	✅ (built-in using opendal)
webdav	✅	✅ (built-in using opendal)
opendal (other services)	✅	✅ (built-in using opendal)
rclone	✅ (via stdin, using external rclone command)	✅ (via http on localhost, using external rclone command)

Commands

command	restic	rustic
backup	✅	✅
cache	✅	✅
cat	✅	✅
config	✅ (no in-repo config)	✅
check	✅	✅
copy	✅	✅
diff	✅	✅
dump	✅	✅
find	✅	✅
forget	✅	✅
generate	✅	✅ completions
init	✅	✅
key list	✅	✅
key add	✅	✅
key remove	✅	✅
key passwd	✅	✅
list	✅	✅
ls	✅	✅

command	restic	rustic
merge	✓	✓
migrate	✓	✓ (not needed; repo version migration via config)
mount	✓	✓ (WIP)
prune	✓	✓
recover	✓	✓
repair index	✓	✓
repair packs	✓	✓
repair snapshots	✓	✓
repoinfo	✓	✓
restore	✓	✓
rewrite	✓	✓
self-update	✓	✓
show-config	✓	✓
snapshots	✓	✓
stats	✓	✓ (but there is repoinfo)
tag	✓	✓
unlock	✓	lock-free
webdav	✓	✓

Information saved in snapshots

information	restic	rustic
from repo design info	✓	✓
program version used	✓	✓
summary (size,...)	✓	✓ (WIP)
used command	✓	✓
label	✓	✓
description	✓	✓
delete (protection)	✓	✓

General options

option	restic	rustic
--cacert	✓	✓
--cache-dir	✓	✓ (or in config profile)
--cleanup-cache	✓	✓
--compression	✓ (auto,max,off); needed in every call	✓ (-7..22) configure once in in-repo config
--dry-run	✓	✓ (or in config profile)
--json	✓	✓
--key-hint	✓	✓
--limit-download	✓	✓
--limit-upload	✓	✓
--log-file	✓	✓ (or in config profile)
--no-cache	✓	✓ (or in config profile)

option	`restic`	`rustic`
`--no-extra-verify`	✗ needed in every call	✓ configure once using `config`
`--no-lock`	✗	✓ all operations are lock-free
`--no-progress`	✗	✓ (or in config profile)
`--progress-intervall`	✓ (via env variable)	✓ (or in config profile)
`--option`	✓ as cmd arg or env variable	✓ via config profile or env variable
`--pack-size`	✓ fix limit; needed in every call	✓ fix or dynamic limit, configure once in in-repo config
`--password-command`	✗	✓ (or in config profile)
`--password-file`	✗	✓ (or in config profile)
`--quiet`	✗	✗
`--repo`	✗	✓ (or in config profile)
`--repo-hot`	✓ (no cold-storage support)	✓ (or in config profile)
`--repository-file`	✗	✓ (use `repository` in config profile instead)
`--retry-lock`	✗	not needed; lock-free
`--tls-client-cert`	✗	✗
`--use-profile`	✗ (no config profile support)	✓ (or in config profile for recursively using profiles)
`--verbose` (multiple times)	✗	✓ `--log-level`
`--warm-up`	✗ (no cold-storage support)	✓ (or in config profile)

`rustic` in-repo config options

option	`restic`	`rustic`
`append_only`	✗	✗
`compression`	✓ (by `--compression`)	✗
`treepack_size`	✓ (only all packs: `--pack-size`)	✗
`treepack_growfactor`	✗	✗
`treepack_size_limit`	✗	✗
`datapack_size`	✓ (only all packs: `--pack-size`)	✗
`datapack_growfactor`	✗	✗
`datapack_size_limit`	✗	✗
`min_packsize_tolerate_percent`	✓ (hardcoded 80% for `prune --repack-small`)	✗
`max_packsize_tolerate_percent`	✗	✗
`extra_verify`	✓ (default, can be unset using `--no-extra-verify`)	✓ (default)

Snapshot filtering

filter	`restic`	`rustic` (options also in config profile)
by host	✓ `--host`	✓ `--filter-host`
by label	✗	✓ `--filter-label`
by paths	✓ `--paths`	✓ `--filter-paths`
by tags	✓ `--tags`	✓ `--filter-tags`

filter	restic	rustic (options also in config profile)
custom	✗	✓ `--filter-fn` (using Rhai)

Comparison of important commands

init

option	restic	rustic
`--copy-chunker-params`	✗	✓ (not needed, use `copy --init`
`--from-*`	✗	✓ (not needed, see `copy` command)
`--hostname`	✓ (always sets hostname)	✗
`--repository-version`	✗	✓ (use `--set-version`)
`--set-*`	✓ (no in-repo config support)	✗
`--username`	✓ (always sets username)	✗
`--with-created`	✓ (always sets creation time)	✗

backup

general	restic	rustic
allow to backup relative paths	✗	✗

option	restic	rustic (options also in config profile)
`--as-path`	✗	✗
`--command`	✗	✗
`--custom-ignorefile`	✗	✗
`--description`	✗	✗
`--description-from`	✗	✗
`--delete-never`	✗	✗
`--delete-after`	✗	✗
`--exclude`	✗	✓ `--glob`
`--exclude-file`	✗	✓ `--glob-file`
`--exclude-caches`	✗	✓ (use `--exclude-if-present`)
`--exclude-if-present`	✓ (+ support for header parsing)	✓ (no header parsing)
`--exclude-larger-than`	✗	✗
`--files-from`	✗	✗
`--files-from-raw`	✗	✗
`--files-from-verbatim`	✗	✗
`--force`	✗	✗
`--git-ignore`	✓ (roadmap: 0.19)	✗
`--group-by`	✓ (host/paths/tags)	✓ (host/label/paths/tags)
`--host`	✗	✗

option	`restic`	`rustic` (options also in config profile)
`--iexclude`	✗	✓ `--iglob`
`--iexclude-file`	✗	✓ `--iglob-file`
`--ignore-ctime`	✗	✓
`--ignore-inode`	✗	✓
`--ignore-devid`	✗	✓
`--init`	✗	✓
`--label`	✗	✓
`--no-require-git`	✓ (no `--git-ignore`)	✓
`--no-scan`	✗	✓
`--one-file-system`	✗	✓
`--parent`	✗	✓
`--read-concurrency`	✗	✓ (hardcoded)
`--skip-identical-parent`	✗	✓
`--stdin`	✗	✓ (use `-` as backup source)
`--stdin-filename`	✗	✓
`--tag`	✗	✓
`--time`	✗	✓
`--with-atime`	✗	✓

`restore`

general	`restic`	`rustic`
scan and use already existing files	✗ (roadmap: 0.17)	✓
resumable restore	✗ (roadmap: 0.17)	✓
restore hard links	✓	✓
`<snapshotID>:<subfolder>` syntax	✓	✓
`<snapshotID>:<subfolder>/file` syntax	✓	✓

option	`restic`	`rustic`
filtering options for `latest`	✗	✓
`--delete`	✗	✓
`--exclude`	✓	✓ `--glob`
`--iexclude`	✗	✓ `--iglob`
`--iinclude`	✗	✓ `--iglob`
`--include`	✓	✓ `--glob`
`--no-ownership`	✗	✓
`--numeric-id`	✓	✓
`--sparse`	✓	✓
`--target`	✓	✓ (give target as second CLI argument)
`--verify`	✓	✓ (but `diff` can be used to verify after)
`--verify-existing`	✓ (no scanning of existing files)	✓

`dump`

general	restic	rustic
dump files	✓	✓
dump dirs	✓	✓

option	restic	rustic
snapshot filtering options for `latest`	✓	✓
`--archive`	✓	✓ (no dumping of dirs)

`forget`

general	restic	rustic
allow to keep all XXX	✓	✓
respect ”no delete” options in snapshot	✓	✓

option	restic	rustic (options also in config profile)
snapshot filtering options	✓	✓
`--keep-last`	✓	✓
`--keep-daily`	✓	✓
`--keep-weekly`	✓	✓
`--keep-monthly`	✓	✓
`--keep-quarter-yearly`	✓	✓
`--keep-half-yearly`	✓	✓
`--keep-yearly`	✓	✓
`--keep-within`	✓	✓
`--keep-within-hourly`	✓	✓
`--keep-within-daily`	✓	✓
`--keep-within-weekly`	✓	✓
`--keep-within-monthly`	✓	✓
`--keep-within-quarter-yearly`	✓	✓
`--keep-within-half-yearly`	✓	✓
`--keep-within-yearly`	✓	✓
`--keep-tag`	✓	✓
`--compact`	✓	✓
`--group-by`	✓ (host/paths/tags)	✓ (host/label/paths/tags)
`--prune`	✓	✓

`prune`

general	restic	rustic
prune plan without reading pack files	✓	✓
prune parallel to backup (two-phase prune)	✓ (roadmap: 0.19)	✓
different pack sizes for tree/data packs	✓	✓
resumable prune	✓ (roadmap: 0.17)	✓

general	restic	rustic
(option to) resize packs	✗	✓

option	restic	rustic
`--fast-repack`	✗	✓
`--instant-delete`	✓ (default, no two-phase)	✓
`--keep-pack`	✗	✓
`--keep-delete`	✓ (no two-phase)	✓
`--max-repack-size`	✗	✓ `--max-repack` (size/%/unlimited)
`--max-unused`	✗	✓
`--repack-all`	✗	✓
`--repack-cacheable-only`	✗	✓
`--repack-small`	✗	✓ (default behavior; to unset use `--no-resize`)
`--repack-uncompressed`	✗	✓
`--unsafe-recover-no-free-space`	✗	✓ `--early-delete-index`

check

general	restic	rustic
check index files	✓	✓
check index vs packs	✓	✓
check snapshot files	✓	✓
(optionally) check pack files	✓	✓
cache policy	create temporary (use existing: roadmap 0.18)	use existing
check cache integrity	✓	✓
check hot/cold integrity	✓ (no cold storage support)	✓

option	restic	rustic
`--read-data`	✓	✓
`--read-data-subset`	✓	✓
`--trust-cache`	✓ (no cache integrity check)	✓
`--with-cache`	✓	✓ (default behavior)

copy

general	restic	rustic
source/target given by	CLI options	in config profile
multiple targets	✓	✓
check for matching chunker parameters	✓	✓

option	restic	rustic
snapshot filtering options	✓	✓
`--from-*`	✓ (target is `--repository`)	✓ (source is `--repository`, target in config profile)
`--init`	✓ (extra run of `init --copy-chunker-params`)	✓

`snapshots`

general	restic	rustic
summarize identical snapshots (like +3)	✓	✓
show summary information (sizes)	✓ (WIP)	✓

option	restic	rustic
snapshot filtering options	✓	✓
`--all`	✓	✓
`--compact`	✓	✓
`--group-by`	✓ (host/paths/tags)	✓ (host/label/paths/tags)
`--latest`	✓	✓
`--long`	✓	✓

`ls`

option	restic	rustic
snapshot filtering options for `latest`	✓	✓
`--glob`	✓	✓
`--glob-file`	✓	✓
`--human-readable`	✓	✓
`--iglob`	✓	✓
`--iglob-file`	✓	✓
`--long`	✓	✓
`--numeric-uid-gid`	✓	✓
`--summary`	✓	✓
`--recursive`	✓	✓

`diff`

general	restic	rustic
allow `latest`	✓	✓
diff with local files	✓	✓
`<snapshotID>:<subfolder>` syntax	✓	✓
`<snapshotID>:<subfolder>/file` syntax	✓	✓

option	`restic`	`rustic`
snapshot filtering options for `latest`	✗ (no `latest` support)	✓
`--glob`	✓	✓
`--glob-file`	✓	✓
`--iglob`	✓	✓
`--iglob-file`	✓	✓
`--metadata`	✓	✓
`--no-content`	✓	✓
exclude options for local files	✗ (no diff with local files)	✓

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

How to install shell completions

- Bash
- Fish
- Powershell
  - Linux
  - Windows
  - macOs
- Zsh

All completion files are generated by invoking `rustic completions` command. So run:

Bash

```
rustic completions bash > /etc/bash_completion.d/rustic.bash
```

Fish

```
rustic completions fish > $HOME/.config/fish/completions/rustic.fish
```

Powershell

Linux

```
rustic completions powershell >>
 ↪  ~/.config/powershell/Microsoft.PowerShell_profile.ps1.
```

Windows

```
rustic completions powershell >>
 ↪  $HOME\Documents\PowerShell\Microsoft.PowerShell_profile.ps1.
```

macOs

```
rustic completions powershell >>
 ↪  ~/.config/powershell/Microsoft.PowerShell_profile.ps1
```

Zsh

ZSH completions are commonly stored in any directory listed in your `$fpath` variable. To use these completions, write completions script (`_rustic`) to one of those directories, or add your own to this list.

This list includes, for example, these directories:

- `/usr/local/share/zsh/site-functions`
- `/usr/share/zsh/site-functions`

So you can run:

```
rustic completions zsh > /usr/local/share/zsh/site-functions/_rustic
```

Chapter 6

Getting started

Contribution needed: We would like to add a written getting started guid here. If you are interested in helping, please check this issue.

Chapter 7

Init - Preparing a new repository

The place where your backups will be saved is called a ”repository”. This chapter explains how to create (”init”) such a repository. The repository can be stored locally, or on some remote server or service. We'll first cover using a local repository; the remaining sections of this chapter cover all the other options. You can skip to the next chapter once you've read the relevant section here.

For automated backups, rustic accepts the repository location in the environment variable `RUSTIC_REPOSITORY`.

For the password, several options exist:

- Setting the environment variable `RUSTIC_PASSWORD`
- Specifying the path to a file with the password via the option `--password-file` or the environment variable `RUSTIC_PASSWORD_FILE`
- Configuring a program to be called when the password is needed via the option `--password-command` or the environment variable `RUSTIC_PASSWORD_COMMAND`

The `init` command has an option called `--set-version` which can be used to explicitly set the version for the new repository.

The below table shows which rustic version is required to use a certain repository version and shows new features introduced by the repository format.

Repository version	Minimum rustic version	Major new features
1	any version	
2	>=0.2.0	Compression support

Moreover, there are different options which can be set when initializing a repository:

Options to specify the target pack size:

- `--set-treepack-size`, `--set-datapack-size` specify the default target pack size for tree and data pack files. Arguments can given using TODO For example, valid sizes are ”4048kiB”, ”2MB”, ”30MiB”, etc. If not specified, the default is 4 MiB for tree packs and 32 MiB for data packs.
- `--set-treepack-growfactor`, `--set-datapack-growfactor` specify how much the target pack size should be increased per square root of the total pack size in bytes of the given type. This equals to 32kiB per square root of the total pack size in GiB.

Note that larger pack sizes have advantages, especially for large repository or remote repositories. They lead to less packs in the repository and transfer larger datasets to the repository which can increase the throughput. But there are also disadvantages. Rustic keeps the whole pack in memory before writing it to the backend. As writes are parallelized, multiple packs are kept. So larger pack sizes increase the memory usage of the `backup` command. Moreover larger pack sizes lead to increased repack rates during `prune` or `forget --prune`.

7.1 Local backend

In order to create a repository at `/srv/rustic-repo`, run the following command and enter the same password twice:

```
$ rustic init -r /srv/rustic-repo
enter password for new repository:
created rustic repository 085b3c76b9 at /srv/rustic-repo
```

Warning: Remembering your password is important! If you lose it, you won't be able to access data stored in the repository.

7.2 REST Server

In order to backup data to the remote server via HTTP or HTTPS protocol, you must first set up a remote REST server instance. Once the server is configured, accessing it is achieved by changing the URL scheme like this:

```
rustic -r rest:http://host:8000/ init
```

Depending on your REST server setup, you can use HTTPS protocol, password protection, multiple repositories or any combination of those features. The TCP/IP port is also configurable. Here are some more examples:

```
rustic -r rest:https://host:8000/ init
rustic -r rest:https://user:pass@host:8000/ init
rustic -r rest:https://user:pass@host:8000/my_backup_repo/ init
```

If you use TLS, rustic will use the system's CA certificates to verify the server certificate. When the verification fails, rustic refuses to proceed and exits with an error. If you have your own self-signed certificate, or a custom CA certificate should be used for verification, you can pass rustic the certificate filename via the `--cacert` option. It will then verify that the server's certificate is contained in the file passed to this option, or signed by a CA certificate in the file. In this case, the system CA certificates are not considered at all.

REST server uses exactly the same directory structure as local backend, so you should be able to access it both locally and via HTTP, even simultaneously.

7.3 Other Services via rclone

The program `rclone` *can be used to access many other different services and store data there. First, you need to install and* `configure` rclone. The general backend specification format is `rclone:<remote>:<path>`, the `<remote>:<path>` component will be directly passed to rclone. When you configure a remote named `foo`, you can then call rustic as follows to initiate a new repository in the path `bar` in the repo:

```
rustic -r rclone:foo:bar init
```

rustic takes care of starting and stopping rclone.

As a more concrete example, suppose you have configured a remote named `b2prod` for Backblaze B2 with rclone, with a bucket called `yggdrasil`. You can then use rclone to list files in the bucket like this:

```
rclone ls b2prod:yggdrasil
```

In order to create a new repository in the root directory of the bucket, call rustic like this:

```
rustic -r rclone:b2prod:yggdrasil init
```

If you want to use the path `foo/bar/baz` in the bucket instead, pass this to rustic:

```
rustic -r rclone:b2prod:yggdrasil/foo/bar/baz init
```

Listing the files of an empty repository directly with rclone should return a listing similar to the following:

```
$ rclone ls b2prod:yggdrasil/foo/bar/baz
    155 bar/baz/config
    448
↪   bar/baz/keys/4bf9c78049de689d73a56ed0546f83b8416795295cda12ec7fb9465af3900b44
```

Rclone can be `configured with environment variables_`, so for instance configuring a bandwidth limit for rclone can be achieved by setting the `RCLONE_BWLIMIT` environment variable:

```
export RCLONE_BWLIMIT=1M
```

For debugging rclone, you can set the environment variable `RCLONE_VERBOSE=2`.

7.4 Cold storage

Rustic supports to store the repository in a so-called cold storage. These are storages which are designed for long-term storage and offer usually cheap storage for the price of retarded or expensive access. Examples are Amazon S3 Glacier or OVH Cloud Archive.

To use a cold storage and not access any data in the storage for every-day operations, rustic needs an extra repository to store hot data. This repository can be specified by the `--hot-repo` option or the `RUSTIC_REPO_HOT` environmental variable, e.g.:

```
rustic -r rclone:foo:bar --repo-hot rclone:foo:bar-hot init
```

In this example in the repository `rclone:foo:bar` all data is saved. In the repository `rclone:foo:bar-hot` only hot data is saved, i.e. this is not a complete repository.

Warning: You have to specify both the cold repository (using `-r`) and the hot repository (using `--repo-hot`) in the `init` command and all other commands which access and work with the repository.

7.5 Configuration file

Important: For always up-to-date information, please make sure to check the in-repository documentation for the config files available here.

Rustic supports configuration files in the TOML format. The files are searched in the following locations:

- the global rustic config dir (on unix typically `/etc/rustic`)
- the users' rustic config dir. On unix this is typically `$HOME/.config/rustic`, see https://docs.rs/directories/latest/directories/struct.ProjectDirs.html for more details about the config location.

- the current working dir

By default, rustic uses the file `rustic.toml`. This can be overwritten by the `-P <PROFILE>` option which tells rustic to search for a `<PROFILE>.toml` configuration file. For example, if you have a `local.toml` configuration for backing up to a local dir and a `remote.toml` configuration for a remote storage, you can use `rustic -P local <COMMAND>` and `rustic -P remote <COMMAND>`, respectively to switch between you two backup configurations.

Note that options in the config file can always be overwritten by ENV

In the configuration file, you can specify all global and repository-specific options as well as options/sources for the `backup` command and `forget` options. Using a config file like

```
# rustic config file to backup /home and /etc to a local repository

[repository]
repository = "/backup/rustic"
password-file = "/root/key-rustic"
no-cache = true # no cache needed for local repository

[forget]
keep-daily = 14
keep-weekly = 5

[backup]
exclude-if-present = [".nobackup", "CACHEDIR.TAG"]
glob-file = ["/root/rustic-local.glob"]

[[backup.sources]]
source = "/home"
git-ignore = true

[[backup.sources]]
source = "/etc"
```

allows you to use `rustic backup` and `rustic forget --prune` in your regularly backup/cleanup scripts.

For more config file examples check the config here

Chapter 8

Backup - Backing up data

Backing up your data is important. This guide will show you how to backup your data. And what else you can do with rustic.

8.1 Creating snapshots

Now we're ready to backup some data. The contents of a directory at a specific point in time is called a ”snapshot” in rustic. Run the following command and enter the repository password you chose above again:

```
$ rustic -r /srv/rustic-repo --verbose backup ~/work
open repository
enter password for repository:
password is correct
lock repository
load index files
start scan
start backup
scan finished in 1.837s
processed 1.720 GiB in 0:12
Files:        5307 new,     0 changed,     0 unmodified
Dirs:         1867 new,     0 changed,     0 unmodified
Added:      1.200 GiB
snapshot 40dc1520 saved
```

As you can see, rustic created a backup of the directory and was pretty fast! The specific snapshot just created is identified by a sequence of hexadecimal characters, `40dc1520` in this case.

You can see that rustic tells us it processed 1.720 GiB of data, this is the size of the files and directories in `~/work` on the local file system. It also tells us that only 1.200 GiB was added to the repository. This means that some of the data was duplicate and rustic was able to efficiently reduce it.

If you don't pass the `--verbose` option, rustic will print less data. You'll still get a nice live status display. Be aware that the live status shows the processed files and not the transferred data. Transferred volume might be lower (due to de-duplication) or higher.

If you run the backup command again, rustic will create another snapshot of your data, but this time it's even faster and no new data was added to the repository (since all data is already there). This is de-duplication at work!

```
$ rustic -r /srv/rustic-repo --verbose backup ~/work
open repository
enter password for repository:
password is correct
lock repository
load index files
using parent snapshot d875ae93
start scan
start backup
scan finished in 1.881s
processed 1.720 GiB in 0:03
Files:           0 new,     0 changed,  5307 unmodified
Dirs:            0 new,     0 changed,  1867 unmodified
Added:      0 B
snapshot 79766175 saved
```

You can even backup individual files in the same repository (not passing `--verbose` means less output):

```
$ rustic -r /srv/rustic-repo backup ~/work.txt
enter password for repository:
password is correct
snapshot 249d0210 saved
```

Now is a good time to run `rustic check` to verify that all data is properly stored in the repository. You should run this command regularly to make sure the internal structure of the repository is free of errors.

8.2 File change detection

When rustic encounters a file that has already been backed up, whether in the current backup or a previous one, it makes sure the file's contents are only stored once in the repository. To do so, it normally has to scan the entire contents of every file. Because this can be very expensive, rustic also uses a change detection rule based on file metadata to determine whether a file is likely unchanged since a previous backup. If it is, the file is not scanned again.

Change detection is only performed for regular files (not special files, symlinks or directories) that have the exact same path as they did in a previous backup of the same location. If a file or one of its containing directories was renamed, it is considered a different file and its entire contents will be scanned again.

Metadata changes (permissions, ownership, etc.) are always included in the backup, even if file contents are considered unchanged.

On **Unix** (including Linux and Mac), given that a file lives at the same location as a file in a previous backup, the following file metadata attributes have to match for its contents to be presumed unchanged:

- Modification timestamp (mtime).
- Metadata change timestamp (ctime).
- File size.
- Inode number (internal number used to reference a file in a filesystem).

The reason for requiring both mtime and ctime to match is that Unix programs can freely change mtime (and some do). In such cases, a ctime change may be the only hint that a file did change.

The following `rustic backup` command line flags modify the change detection rules:

- `--force`: turn off change detection and rescan all files.
- `--ignore-ctime`: require mtime to match, but allow ctime to differ.
- `--ignore-inode`: require mtime to match, but allow inode number and ctime to differ.

The option `--ignore-inode` exists to support FUSE-based filesystems and pCloud, which do not assign stable inodes to files.

Note that the device id of the containing mount point is never taken into account. Device numbers are not stable for removable devices and ZFS snapshots. If you want to force a re-scan in such a case, you can change the mountpoint.

8.3 Dry Run

You can perform a backup in dry run mode to see what would happen without modifying the repo.

- `--dry-run`/`-n` Report what would be done, without writing to the repository

8.4 Excluding Files

You can exclude folders and files by specifying exclude patterns, currently the exclude options are:

- `--git-ignore` Respect `.gitignore` files and exclude paths/files not handled by git.
- `--glob` include/exclue files and dirs based on given glob patterns
- `--iglob` Same as `--glob` but ignores the case of paths
- `--glob-file` Specified one or more times to exclude items listed in a given file
- `--iglob-file` Same as `--glob-file` but ignores cases like in `--iglob`
- `--exclude-if-present foo` Specified one or more times to exclude a folder's content if it contains a file called `foo`. For example, to exclude cache dirs, specify `--exclude-if-present CACHEDIR.TAG`.
- `--exclude-larger-than size` Specified once to excludes files larger than the given size

Please see `rustic help backup` for more specific information about each exclude option.

Let's say we have a file called `glob.txt` with the following content:

```
# exclude go-files
!*.go
# exclude foo/x/y/z/bar foo/x/bar foo/bar
!foo/**/bar
```

It can be used like this:

```
rustic -r /srv/rustic-repo backup ~/work --glob="!*.c" --glob-file=glob.txt
```

This instructs rustic to exclude files matching the following criteria:

- All files matching `*.c` (parameter `--glob`)
- All files matching `*.go` (second line in `glob.txt`)
- All files and sub-directories named `bar` which reside somewhere below a directory called `foo` (fourth line in `glob.txt`)

By specifying the option `--one-file-system` you can instruct rustic to only backup files from the file systems the initially specified files or directories reside on. In other words, it will prevent rustic from crossing filesystem boundaries and subvolumes when performing a backup.

For example, if you backup `/` with this option and you have external media mounted under `/media/usb` then rustic will not back up `/media/usb` at all because this is a different filesystem

than `/`. Virtual filesystems such as `/proc` are also considered different and thereby excluded when using `--one-file-system`:

```
rustic -r /srv/rustic-repo backup --one-file-system /
```

Please note that this does not prevent you from specifying multiple filesystems on the command line, e.g:

```
rustic -r /srv/rustic-repo backup --one-file-system / /media/usb
```

will back up both the `/` and `/media/usb` filesystems, but will not include other filesystems like `/sys` and `/proc`.

Note: `--one-file-system` is currently unsupported on Windows, and will cause the backup to immediately fail with an error.

Files larger than a given size can be excluded using the `--exclude-larger-than` option:

```
rustic -r /srv/rustic-repo backup ~/work --exclude-larger-than 1M
```

This excludes files in `~/work` which are larger than 1 MiB from the backup.

The default unit for the size value is bytes, so e.g. `--exclude-larger-than 2048` would exclude files larger than 2048 bytes (2 KiB). To specify other units, suffix the size value with one of `k`/`K` for KiB (1024 bytes), `m`/`M` for MiB (1024^2 bytes), `g`/`G` for GiB (1024^3 bytes) and `t`/`T` for TiB (1024^4 bytes), e.g. `1k`, `10K`, `20m`, `20M`, `30g`, `30G`, `2t` or `2T`).

8.5 Comparing Snapshots

Rustic has a `diff` command which shows the difference between two snapshots or a snapshot and a local path/dir

```
$ rustic -r /srv/rustic-repo diff 5845b002 2ab627a6
password is correct
comparing snapshot ea657ce5 to 2ab627a6:

C   /rustic/cmd_diff.go
+   /rustic/foo
C   /rustic/rustic
```

8.6 Backup special items and metadata

Symlinks are archived as symlinks, `rustic` does not follow them. When you restore, you get the same symlink again, with the same link target and the same timestamps.

If there is a **bind-mount** below a directory that is to be saved, rustic descends into it.

Device files are saved and restored as device files. This means that e.g. `/dev/sda` is archived as a block device file and restored as such. This also means that the content of the corresponding disk is not read, at least not from the device file.

By default, rustic does not save the access time (atime) for any files or other items, since it is not possible to reliably disable updating the access time by rustic itself. This means that for each new backup a lot of metadata is written, and the next backup needs to write new metadata again. If you really want to save the access time for files and directories, you can pass the `--with-atime` option to the `backup` command.

Note that `rustic` does not back up some metadata associated with files. Of particular note are::

- file creation date on Unix platforms
- inode flags on Unix platforms
- xattr information

8.7 Reading data from StdIn

Sometimes it can be nice to directly save the output of a program, e.g. `mysqldump` so that the SQL can later be restored. Rustic supports this mode of operation, just supply `-` as backup source to the `backup` command like this:

```
set -o pipefail
mysqldump [...] | rustic backup -
```

This creates a new snapshot of the output of `mysqldump`. You can then use e.g. the fuse mounting option (see below) to mount the repository and read the file.

By default, the file name `stdin` is used, a different name can be specified with `--stdin-filename`, e.g. like this:

```
mysqldump [...] | rustic --stdin-filename production.sql -
```

The option `pipefail` is highly recommended so that a non-zero exit code from one of the programs in the pipe (e.g. `mysqldump` here) makes the whole chain return a non-zero exit code. Refer to the `Use the Unofficial Bash Strict Mode <http://redsymbol.net/articles/unofficial-bash-strict-mode/>__` for more details on this.

8.8 Tags for backup

Snapshots can have one or more tags, short strings which add identifying information. Just specify the tags for a snapshot one by one with `--tag`:

```
$ rustic -r /srv/rustic-repo backup --tag projectX --tag foo --tag bar ~/work
[...]
```

The tags can later be used to keep (or forget) snapshots with the `forget` command. The command `tag` can be used to modify tags on an existing snapshot.

8.9 Scheduling backups

Rustic does not have a built-in way of scheduling backups, as it's a tool that runs when executed rather than a daemon. There are plenty of different ways to schedule backup runs on various different platforms, e.g. systemd and cron on Linux/BSD and Task Scheduler in Windows, depending on one's needs and requirements. When scheduling rustic to run recurringly, please make sure to detect already running instances before starting the backup.

8.10 Space requirements

Rustic currently assumes that your backup repository has sufficient space for the backup operation you are about to perform. This is a realistic assumption for many cloud providers, but may not be true when backing up to local disks.

Should you run out of space during the middle of a backup, there will be some additional data in the repository, but the snapshot will never be created as it would only be written at the very (successful) end of the backup operation. Previous snapshots will still be there and will still work.

8.11 Environment Variables

Important: For always up-to-date information, please make sure to check the in-repository documentation for the config files available here.

In addition to command-line options, rustic supports passing various options in environment variables. The following lists these environment variables:

```
RUSTIC_REPOSITORY                   Location of repository (replaces -r)
RUSTIC_REPO_HOT                     Location of hot repository (replaces
 ↪  -repo-hot)
RUSTIC_PASSWORD                     The actual password for the repository
 ↪  (replaces --password)
RUSTIC_PASSWORD_FILE                Location of password file (replaces
 ↪  --password-file)
RUSTIC_PASSWORD_COMMAND             Command printing the password for the
 ↪  repository to stdout (replaces --password-command)
RUSTIC_CACHE_DIR                    Location of the cache directory (replaces
 ↪  --cache-dir)
RUSTIC_NO_CACHE                     Use no cache (replaces --no-cache)
```

rustic may execute `rclone` (for rclone backends) which may respond to further environment variables and configuration files.

Chapter 9

Miscellaneous - Working with repositories

A repository is a storage location for all of your snapshots.

The repository is created with the `init` command:

```
$ rustic -r /srv/rustic-repo init
enter password for new repository:
enter password again:
created rustic repository 7a8c3b2a0c at /srv/rustic-repo
Please note that knowledge of your password is required to access
the repository. Losing your password means that your data is
irrecoverably lost.
```

The repository is now ready for use.

Note: In case you are using the rclone backend, please see the `rclone-backend` section for additional information.

9.1 Listing snapshots

To list all snapshots in the repository, use the `snapshots` command:

```
$ rustic -r /srv/rustic-repo snapshots
enter password for repository:
ID        Date                 Host    Tags   Directory
-----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir        /home/user/work
79766175  2015-05-08 21:40:19  kasimir        /home/user/work
bdbd3439  2015-05-08 21:45:17  luigi          /home/art
590c8fc8  2015-05-08 21:47:38  kazik          /srv
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

You can filter the listing by directory path:

```
$ rustic -r /srv/rustic-repo snapshots --path="/srv"
enter password for repository:
ID        Date                 Host    Tags   Directory
-----------------------------------------------------------------------
```

```
590c8fc8  2015-05-08 21:47:38  kazik          /srv
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

Or filter by host:

```
$ rustic -r /srv/rustic-repo snapshots --host luigi
enter password for repository:
ID        Date                Host    Tags   Directory
----------------------------------------------------------------------
bdbd3439  2015-05-08 21:45:17  luigi          /home/art
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

Combining filters is also possible.

Furthermore you can group the output by the same filters (host, paths, tags):

```
$ rustic -r /srv/rustic-repo snapshots --group-by host

enter password for repository:
snapshots for (host [kasimir])
ID        Date                Host    Tags   Directory
----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir        /home/user/work
79766175  2015-05-08 21:40:19  kasimir        /home/user/work
2 snapshots
snapshots for (host [luigi])
ID        Date                Host    Tags   Directory
----------------------------------------------------------------------
bdbd3439  2015-05-08 21:45:17  luigi          /home/art
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
2 snapshots
snapshots for (host [kazik])
ID        Date                Host    Tags   Directory
----------------------------------------------------------------------
590c8fc8  2015-05-08 21:47:38  kazik          /srv
1 snapshots
```

9.2 Copying snapshots between repositories

In case you want to transfer snapshots between two repositories, for example from a local to a remote repository, you can use the `copy` command:

```
$ rustic -r /srv/rustic-repo copy --repo2 /srv/rustic-repo-copy
repository d6504c63 opened successfully, password is correct
repository 3dd0878c opened successfully, password is correct

snapshot 410b18a2 of [/home/user/work] at 2020-06-09 23:15:57.305305 +0200
 ↪  CEST)
    copy started, this may take a while...
snapshot 7a746a07 saved

snapshot 4e5d5487 of [/home/user/work] at 2020-05-01 22:44:07.012113 +0200
 ↪  CEST)
skipping snapshot 4e5d5487, was already copied to snapshot 50eb62b7
```

The example command copies all snapshots from the source repository `/srv/rustic-repo` to the

destination repository `/srv/rustic-repo-copy`. Snapshots which have previously been copied between repositories will be skipped by later copy runs.

Important: This process will have to both download (read) and upload (write) the entire snapshot(s) due to the different encryption keys used in the source and destination repository. This *may incur higher bandwidth usage and costs* than expected during normal backup runs.

Important: The copying process does not re-chunk files, which may break deduplication between the files copied and files already stored in the destination repository. This means that copied files, which existed in both the source and destination repository, *may occupy up to twice their space* in the destination repository. See below for how to avoid this.

The destination repository is specified with `--repo` or can be read from a file specified via `--repository-file`. Both of these options can also set as environment variables `$RUSTIC_REPOSITORY` or `$RUSTIC_REPOSITORY_FILE` respectively. For the destination repository the password can be read from a file `--password-file` or from a command `--password-command`. Alternatively the environment variables `$RUSTIC_PASSWORD_COMMAND` and `$RUSTIC_PASSWORD_FILE` can be used. It is also possible to directly pass the password via `$RUSTIC_PASSWORD`. The key which should be used for decryption can be selected by passing its ID via the flag `--key-hint` or the environment variable `$RUSTIC_KEY_HINT`.

Note: In case the source and destination repository use the same backend, the configuration options and environment variables used to configure the backend may apply to both repositories – for example it might not be possible to specify different accounts for the source and destination repository. You can avoid this limitation by using the rclone backend along with remotes which are configured in rclone.

9.3 Filtering snapshots to copy

The list of snapshots to copy can be filtered by host, path in the backup and / or a comma-separated tag list:

```
rustic -r /srv/rustic-repo copy --repo2 /srv/rustic-repo-copy --host luigi
 ↪  --path /srv --tag foo,bar
```

It is also possible to explicitly specify the list of snapshots to copy, in which case only these instead of all snapshots will be copied:

```
rustic -r /srv/rustic-repo copy --repo2 /srv/rustic-repo-copy 410b18a2
 ↪  4e5d5487 latest
```

9.4 Ensuring deduplication for copied snapshots

Different repositories usually have different parameters for splitting larger files into smaller chunks. When copying snapshots between arbitrary repository, deduplication between snapshots from the source and destination repository doesn't work unless the repositories share the same parameters, the so-called chunker parameters.

Rustic enforces identical chunker parameters - if you try to copy to a repository with different chunker parameter, you get an error like

```
cannot copy to repository with different chunker parameter (re-chunking not
 ↪  implemented)!
```

Note: Currently it is not possible to change the chunker parameters of existing repositories (re-chunking is not yet implemented).

To create a repository with identical chunker parameters to a source repository, don't initialize the target repository, but instead run the first `copy` command with the `--init` option. This option initializes non-existing repositories with the correct chunker parameter:

```
rustic copy --init [SNAPSHOTS]
```

Note: The target repositories must be defined in the config file.

9.5 Checking integrity and consistency

Imagine your repository is saved on a server that has a faulty hard drive, or even worse, attackers get privileged access and modify the files in your repository with the intention to make you restore malicious data:

```
echo "boom" > /srv/rustic-
 ↪  repo/index/de30f3231ca2e6a59af4aa84216dfe2ef7339c549dc11b09b84000997b139628
```

Trying to restore a snapshot which has been modified as shown above will yield an error:

```
$ rustic -r /srv/rustic-repo --no-cache restore c23e491f --target
 ↪  /tmp/restore-work
...
Fatal: unable to load index de30f323: load <index/de30f3231c>: invalid data
 ↪  returned
```

In order to detect these things before they become a problem, it's a good idea to regularly use the `check` command to test whether your repository is healthy and consistent, and that your precious backup data is unharmed. There are two types of checks that can be performed:

- Structural consistency and integrity, e.g. snapshots, trees and pack files (default)
- Integrity of the actual data that you backed up (enabled with flags, see below)

To verify the structure of the repository, issue the `check` command. If the repository is damaged like in the example above, `check` will detect this and yield the same error as when you tried to restore:

```
$ rustic -r /srv/rustic-repo check
...
load indexes
error: error loading index de30f323: load <index/de30f3231c>: invalid data
 ↪  returned
Fatal: LoadIndex returned errors
```

If the repository structure is intact, rustic will show that no errors were found:

```
$ rustic -r /src/rustic-repo check
...
load indexes
check all packs
check snapshots, trees and blobs
no errors were found
```

By default, the `check` command does not verify that the actual pack files on disk in the repository are unmodified, because doing so requires reading a copy of every pack file in the repository. To tell rustic to also verify the integrity of the pack files in the repository, use the `--read-data` flag:

```
$ rustic -r /srv/rustic-repo check --read-data
...
load indexes
check all packs
```

```
check snapshots, trees and blobs
read all data
[0:00] 100.00%  3 / 3 items
duration: 0:00
no errors were found
```

Note: Since `--read-data` has to download all pack files in the repository, beware that it might incur higher bandwidth costs than usual and also that it takes more time than the default `check`.

Alternatively, use the `--read-data-subset` parameter to check only a subset of the repository pack files at a time. It supports three ways to select a subset. One selects a specific part of pack files, the second and third selects a random subset of the pack files by the given percentage or size.

Use `--read-data-subset=n/t` to check a specific part of the repository pack files at a time. The parameter takes two values, `n` and `t`. When the check command runs, all pack files in the repository are logically divided in `t` (roughly equal) groups, and only files that belong to group number `n` are checked. For example, the following commands check all repository pack files over 5 separate invocations:

```
rustic -r /srv/rustic-repo check --read-data-subset=1/5
rustic -r /srv/rustic-repo check --read-data-subset=2/5
rustic -r /srv/rustic-repo check --read-data-subset=3/5
rustic -r /srv/rustic-repo check --read-data-subset=4/5
rustic -r /srv/rustic-repo check --read-data-subset=5/5
```

Use `--read-data-subset=x%` to check a randomly choosen subset of the repository pack files. It takes one parameter, `x`, the percentage of pack files to check as an integer or floating point number. This will not guarantee to cover all available pack files after sufficient runs, but it is easy to automate checking a small subset of data after each backup. For a floating point value the following command may be used:

```
rustic -r /srv/rustic-repo check --read-data-subset=2.5%
```

When checking bigger subsets you most likely want to specify the percentage as an integer:

```
rustic -r /srv/rustic-repo check --read-data-subset=10%
```

Use `--read-data-subset=nS` to check a randomly chosen subset of the repository pack files. It takes one parameter, `nS`, where 'n' is a whole number representing file size and 'S' is the unit of file size (K/M/G/T) of pack files to check. Behind the scenes, the specified size will be converted to percentage of the total repository size. The behaviour of the check command following this conversion will be the same as the percentage option above. For a file size value the following command may be used:

```
rustic -r /srv/rustic-repo check --read-data-subset=50M
rustic -r /srv/rustic-repo check --read-data-subset=10G
```

9.6 Key - Manage repository keys

The `key` command allows you to set multiple access keys or passwords per repository. In fact, you can use the `list`, `add`, `remove`, and `passwd` (changes a password) sub-commands to manage these keys very precisely:

```
$ rustic -r /srv/rustic-repo key list
enter password for repository:
 ID          User        Host        Created
----------------------------------------------------------------------
*eb78040b    username    kasimir   2015-08-12 13:29:57
```

```
$ rustic -r /srv/rustic-repo key add
enter password for repository:
enter password for new key:
enter password again:
saved new key as <Key of username@kasimir, created on 2015-08-12
↪ 13:35:05.316831933 +0200 CEST>

$ rustic -r /srv/rustic-repo key list
enter password for repository:
 ID          User        Host        Created
------------------------------------------------------------------------
 5c657874    username    kasimir   2015-08-12 13:35:05
*eb78040b    username    kasimir   2015-08-12 13:29:57
```

Note: that the currently used key is indicated by an asterisk (*).

9.7 Upgrading the repository format version

Repositories created using earlier rustic versions use an older repository format version and have to be upgraded to allow using all new features. Upgrading must be done explicitly as a newer repository version increases the minimum rustic version required to access the repository. For example the repository format version 2 is only readable using rustic 0.2.0 or newer.

Upgrading to repo version 2 is a two step process: first run `migrate upgrade_repo_v2` which will check the repository integrity and then upgrade the repository version. Repository problems must be corrected before the migration will be possible. After the migration is complete, run `prune` to compress the repository metadata. To limit the amount of data rewritten in at once, you can use the `prune --max-repack-size size` parameter, see :ref:`customize-pruning` for more details.

File contents stored in the repository will not be rewritten, data from new backups will be compressed. Over time more and more of the repository will be compressed. To speed up this process and compress all not yet compressed data, you can run `prune --repack-uncompressed`.

Chapter 10

Restore - Restoring from backup

Restoring from a snapshot is as easy as it sounds, just use the following command to restore the contents of the latest snapshot to `/tmp/restore-work`:

```
$ rustic -r /srv/rustic-repo restore 79766175 /tmp/restore-work
enter password for repository:
restoring <Snapshot of [/home/user/work] at 2015-05-08 21:40:19.884408621
 ↪  +0200 CEST> to /tmp/restore-work
```

Use the word `latest` to restore the last backup. You can also combine `latest` with the `--filter-host` and `--filter-path` filters to choose the last backup for a specific host, path or both.

```
$ rustic -r /srv/rustic-repo restore latest /tmp/restore-art --filter-path
 ↪  "/home/art" --filter-host luigi
enter password for repository:
restoring <Snapshot of [/home/art] at 2015-05-08 21:45:17.884408621 +0200
 ↪  CEST> to /tmp/restore-art
```

Use `--glob` (pattern to exclude/include (can be specified multiple times)) to restrict the restore to a subset of files in the snapshot. For example, to restore a single file:

```
$ rustic -r /srv/rustic-repo restore 79766175 /tmp/restore-work --glob
 ↪  /work/foo
enter password for repository:
restoring <Snapshot of [/home/user/work] at 2015-05-08 21:40:19.884408621
 ↪  +0200 CEST> to /tmp/restore-work
```

This will restore the file `foo` to `/tmp/restore-work/work/foo`.

You can use the command `rustic ls latest`

the path to the file within the snapshot. This path you can then pass to `--glob` in verbatim to only restore the single file or directory.

There is case insensitive variants of `--glob` called `--iglob`. This option will behave the same way but ignore the casing of paths.

10.1 Restore using mount

NOTE: `rustic` doesn't support `mount` at this point, please use `restic` to invoke this operation for the time being. We are working on a `mount` implementation for `rustic`.

Browsing your backup as a regular file system is also very easy. First, create a mount point such as `/mnt/rustic` and then use the following command to serve the repository with FUSE:

```
$ mkdir /mnt/rustic
$ restic -r /srv/rustic-repo mount /mnt/rustic
enter password for repository:
Now serving /srv/rustic-repo at /mnt/rustic
Use another terminal or tool to browse the contents of this folder.
When finished, quit with Ctrl-c here or umount the mountpoint.
```

Mounting repositories via FUSE is only possible on Linux, macOS and FreeBSD. On Linux, the `fuse` kernel module needs to be loaded and the `fusermount` command needs to be in the `PATH`. On macOS, you need FUSE for macOS. On FreeBSD, you may need to install FUSE and load the kernel module (`kldload fuse`).

10.2 Printing files to stdout

Sometimes it's helpful to print files to stdout so that other programs can read the data directly. This can be achieved by using the dump command, like this:

```
rustic -r /srv/rustic-repo dump latest production.sql | mysql
```

If you have saved multiple different things into the same repo, the `latest` snapshot may not be the right one. For example, consider the following snapshots in a repo:

```
$ rustic -r /srv/rustic-repo snapshots
ID        Date                 Host        Tags        Directory
----------------------------------------------------------------------
562bfc5e  2018-07-14 20:18:01  mopped                  /home/user/file1
bbacb625  2018-07-14 20:18:07  mopped                  /home/other/work
e922c858  2018-07-14 20:18:10  mopped                  /home/other/work
098db9d5  2018-07-14 20:18:13  mopped                  /production.sql
b62f46ec  2018-07-14 20:18:16  mopped                  /home/user/file1
1541acae  2018-07-14 20:18:18  mopped                  /home/other/work
----------------------------------------------------------------------
```

Here, rustic would resolve `latest` to the snapshot `1541acae`, which does not contain the file we'd like to print at all (`production.sql`). In this case, you can pass rustic the snapshot ID of the snapshot you like to restore:

```
rustic -r /srv/rustic-repo dump 098db9d5 production.sql | mysql
```

Or you can pass rustic a path that should be used for selecting the latest snapshot. The path must match the patch printed in the ”Directory” column, e.g.:

```
rustic -r /srv/rustic-repo dump --path /production.sql latest production.sql |
 ↪  mysql
```

It is also possible to dump the contents of a whole folder structure to stdout. To retain the information about the files and folders rustic will output the contents in the tar (default) or zip format:

```
rustic -r /srv/rustic-repo dump latest /home/other/work > restore.tar
```

```
rustic -r /srv/rustic-repo dump -a zip latest /home/other/work > restore.zip
```

Chapter 11

Forget - Removing backup snapshots

All backup space is finite, so rustic allows removing old snapshots. This can be done either manually (by specifying a snapshot ID to remove) or by using a policy that describes which snapshots to forget. For all remove operations, two commands need to be called in sequence: `forget` to remove snapshots, and `prune` to remove the remaining data that was referenced only by the removed snapshots. The latter can be automated with the `--prune` option of `forget`, which runs `prune` automatically if any snapshots were actually removed.

Pruning snapshots can be a time-consuming process, depending on the number of snapshots and data to process. During a prune operation, the repository is locked and backups cannot be completed. Please plan your pruning so that there's time to complete it and it doesn't interfere with regular backup runs.

It is advisable to run `rustic check` after pruning, to make sure you are alerted, should the internal data structures of the repository be damaged.

11.1 Remove a single snapshot

The command `snapshots` can be used to list all snapshots in a repository like this:

```
$ rustic -r /srv/rustic-repo snapshots
enter password for repository:
ID        Date                 Host      Tags  Directory
-----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir         /home/user/work
79766175  2015-05-08 21:40:19  kasimir         /home/user/work
bdbd3439  2015-05-08 21:45:17  luigi           /home/art
590c8fc8  2015-05-08 21:47:38  kazik           /srv
9f0bc19e  2015-05-08 21:46:11  luigi           /srv
```

In order to remove the snapshot of `/home/art`, use the `forget` command and specify the snapshot ID on the command line:

```
$ rustic -r /srv/rustic-repo forget bdbd3439
enter password for repository:
removed snapshot bdbd3439
```

Afterwards this snapshot is removed:

```
$ rustic -r /srv/rustic-repo snapshots
enter password for repository:
```

```
ID        Date                 Host     Tags  Directory
----------------------------------------------------------------------
40dc1520  2015-05-08 21:38:30  kasimir        /home/user/work
79766175  2015-05-08 21:40:19  kasimir        /home/user/work
590c8fc8  2015-05-08 21:47:38  kazik          /srv
9f0bc19e  2015-05-08 21:46:11  luigi          /srv
```

But the data that was referenced by files in this snapshot is still stored in the repository. To cleanup unreferenced data, the `prune` command must be run:

```
$ rustic -r /srv/rustic-repo prune
enter password for repository:
repository 33002c5e opened successfully, password is correct
loading all snapshots...
loading indexes...
finding data that is still in use for 4 snapshots
[0:00] 100.00%  4 / 4 snapshots
searching used packs...
collecting packs for deletion and repacking
[0:00] 100.00%  5 / 5 packs processed

to repack:            69 blobs / 1.078 MiB
this removes:         67 blobs / 1.047 MiB
to delete:             7 blobs / 25.726 KiB
total prune:          74 blobs / 1.072 MiB
remaining:            16 blobs / 38.003 KiB
unused size after prune: 0 B (0.00% of remaining size)

repacking packs
[0:00] 100.00%  2 / 2 packs repacked
rebuilding index
[0:00] 100.00%  3 / 3 packs processed
deleting obsolete index files
[0:00] 100.00%  3 / 3 files deleted
removing 3 old packs
[0:00] 100.00%  3 / 3 files deleted
done
```

Afterwards the repository is smaller.

You can automate this two-step process by using the `--prune` switch to `forget`:

```
$ rustic forget --keep-last 1 --prune
snapshots for host mopped, directories /home/user/work:

keep 1 snapshots:
ID        Date                 Host        Tags        Directory
----------------------------------------------------------------------
4bba301e  2017-02-21 10:49:18  mopped                  /home/user/work

remove 1 snapshots:
ID        Date                 Host        Tags        Directory
----------------------------------------------------------------------
8c02b94b  2017-02-21 10:48:33  mopped                  /home/user/work

1 snapshots have been removed, running prune
```

```
loading all snapshots...
loading indexes...
finding data that is still in use for 1 snapshots
[0:00] 100.00%  1 / 1 snapshots
searching used packs...
collecting packs for deletion and repacking
[0:00] 100.00%  5 / 5 packs processed

to repack:           69 blobs / 1.078 MiB
this removes         67 blobs / 1.047 MiB
to delete:            7 blobs / 25.726 KiB
total prune:         74 blobs / 1.072 MiB
remaining:           16 blobs / 38.003 KiB
unused size after prune: 0 B (0.00% of remaining size)

repacking packs
[0:00] 100.00%  2 / 2 packs repacked
rebuilding index
[0:00] 100.00%  3 / 3 packs processed
deleting obsolete index files
[0:00] 100.00%  3 / 3 files deleted
removing 3 old packs
[0:00] 100.00%  3 / 3 files deleted
done
```

11.2 Removing snapshots according to a policy

Removing snapshots manually is tedious and error-prone, therefore rustic allows specifying a policy (one or more `--keep-*` options) for which snapshots to keep. You can for example define how many hourly, daily, weekly, monthly and yearly snapshots to keep, and any other snapshots will be removed.

Warning: If you use an append-only repository with policy-based snapshot removal, some security considerations are important. Please refer to the section below for more information.

Note: You can always use the `--dry-run` option of the `forget` command, which instructs rustic to not remove anything but instead just print what actions would be performed.

The `forget` command accepts the following policy options:

- `--keep-last n` keep the n last (most recent) snapshots.
- `--keep-hourly n` for the last n hours which have one or more snapshots, keep only the most recent one for each hour.
- `--keep-daily n` for the last n days which have one or more snapshots, keep only the most recent one for each day.
- `--keep-weekly n` for the last n weeks which have one or more snapshots, keep only the most recent one for each week.
- `--keep-monthly n` for the last n months which have one or more snapshots, keep only the most recent one for each month.
- `--keep-yearly n` for the last n years which have one or more snapshots, keep only the most recent one for each year.
- `--keep-tag` keep all snapshots which have all tags specified by this option (can be specified multiple times).
- `--keep-within duration` keep all snapshots having a timestamp within the specified duration of the latest snapshot, where `duration` is a number of years, months, days, and hours. E.g.

`2y5m7d3h` will keep all snapshots made in the two years, five months, seven days and three hours before the latest (most recent) snapshot.
- `--keep-within-hourly duration` keep all hourly snapshots made within the specified duration of the latest snapshot. The `duration` is specified in the same way as for `--keep-within` and the method for determining hourly snapshots is the same as for `--keep-hourly`.
- `--keep-within-daily duration` keep all daily snapshots made within the specified duration of the latest snapshot.
- `--keep-within-weekly duration` keep all weekly snapshots made within the specified duration of the latest snapshot.
- `--keep-within-monthly duration` keep all monthly snapshots made within the specified duration of the latest snapshot.
- `--keep-within-yearly duration` keep all yearly snapshots made within the specified duration of the latest snapshot.

Note: All calendar related options (`--keep-{hourly,daily,...}`) work on natural time boundaries and *not* relative to when you run `forget`. Weeks are Monday 00:00 to Sunday 23:59, days 00:00 to 23:59, hours :00 to :59, etc. They also only count hours/days/weeks/etc which have one or more snapshots.

Note: All duration related options (`--keep-{within,-*}`) ignore snapshots with a timestamp in the future (relative to when the `forget` command is run) and these snapshots will hence not be removed.

Note: Specifying `--keep-tag ''` will match untagged snapshots only.

When `forget` is run with a policy, rustic first loads the list of all snapshots and groups them by their host name and paths. The grouping options can be set with `--group-by`, e.g. using `--group-by paths,tags` to instead group snapshots by paths and tags. The policy is then applied to each group of snapshots individually. This is a safety feature to prevent accidental removal of unrelated backup sets. To disable grouping and apply the policy to all snapshots regardless of their host, paths and tags, use `--group-by ''` (that is, an empty value to `--group-by`).

Additionally, you can restrict the policy to only process snapshots which have a particular hostname with the `--host` parameter, or tags with the `--tag` option. When multiple tags are specified, only the snapshots which have all the tags are considered. For example, the following command removes all but the latest snapshot of all snapshots that have the tag `foo`:

```
rustic forget --tag foo --keep-last 1
```

This command removes all but the last snapshot of all snapshots that have either the `foo` or `bar` tag set:

```
rustic forget --tag foo --tag bar --keep-last 1
```

To only keep the last snapshot of all snapshots with both the tag `foo` and `bar` set use:

```
rustic forget --tag foo,bar --keep-last 1
```

To ensure only untagged snapshots are considered, specify the empty string '' as the tag.

```
rustic forget --tag '' --keep-last 1
```

Let's look at a simple example: Suppose you have only made one backup every Sunday for 12 weeks:

```
rustic snapshots repository f00c6e2a opened successfully, password is correct
↪  ID Time Host Tags Paths
```

```
## 0a1f9759 2019-09-01 11:00:00 mopped /home/user/work 46cfe4d5 2019-09-08
 ↪  11:00:00 mopped /home/user/work f6b1f037 2019-09-15 11:00:00 mopped
 ↪  /home/user/work eb430a5d 2019-09-22 11:00:00 mopped /home/user/work
 ↪  8cf1cb9a 2019-09-29 11:00:00 mopped /home/user/work 5d33b116 2019-10-06
 ↪  11:00:00 mopped /home/user/work b9553125 2019-10-13 11:00:00 mopped
 ↪  /home/user/work e1a7b58b 2019-10-20 11:00:00 mopped /home/user/work
 ↪  8f8018c0 2019-10-27 11:00:00 mopped /home/user/work 59403279 2019-11-03
 ↪  11:00:00 mopped /home/user/work dfee9fb4 2019-11-10 11:00:00 mopped
 ↪  /home/user/work e1ae2f40 2019-11-17 11:00:00 mopped /home/user/work

12 snapshots
```

Then `forget --keep-daily 4` will keep the last four snapshots, for the last four Sundays, and remove the other snapshots:

```
$ rustic forget --keep-daily 4 --dry-run repository f00c6e2a opened
 ↪  successfully, password is correct Applying Policy: keep the last 4 daily
 ↪  snapshots keep 4 snapshots: ID Time Host Tags Reasons Paths

8f8018c0 2019-10-27 11:00:00 mopped daily snapshot /home/user/work 59403279
 ↪  2019-11-03 11:00:00 mopped daily snapshot /home/user/work dfee9fb4
 ↪  2019-11-10 11:00:00 mopped daily snapshot /home/user/work e1ae2f40
 ↪  2019-11-17 11:00:00 mopped daily snapshot /home/user/work

4 snapshots

remove 8 snapshots: ID Time Host Tags Paths

0a1f9759 2019-09-01 11:00:00 mopped /home/user/work 46cfe4d5 2019-09-08
 ↪  11:00:00 mopped /home/user/work f6b1f037 2019-09-15 11:00:00 mopped
 ↪  /home/user/work eb430a5d 2019-09-22 11:00:00 mopped /home/user/work
 ↪  8cf1cb9a 2019-09-29 11:00:00 mopped /home/user/work 5d33b116 2019-10-06
 ↪  11:00:00 mopped /home/user/work b9553125 2019-10-13 11:00:00 mopped
 ↪  /home/user/work e1a7b58b 2019-10-20 11:00:00 mopped /home/user/work

8 snapshots
```

The processed snapshots are evaluated against all `--keep-*` options but a snapshot only need to match a single option to be kept (the results are ORed). This means that the most recent snapshot on a Sunday would match both hourly, daily and weekly `--keep-*` options, and possibly more depending on calendar.

For example, suppose you make one backup every day for 100 years. Then `forget --keep-daily 7 --keep-weekly 5 --keep-monthly 12 --keep-yearly 75` would keep the most recent 7 daily snapshots and 4 last-day-of-the-week ones (since the 7 dailies already include 1 weekly). Additionally, 12 or 11 last-day-of-the-month snapshots will be kept (depending on whether one of them ends up being the same as a daily or weekly). And finally 75 or 74 last-day-of-the-year snapshots are kept, depending on whether one of them ends up being the same as an already kept snapshot. All other snapshots are removed.

You might want to maintain the same policy as in the example above, but have irregular backups. For example, the 7 snapshots specified with `--keep-daily 7` might be spread over a longer period. If what you want is to keep daily snapshots for the last week, weekly for the last month, monthly for the last year and yearly for the last 75 years, you can instead specify `forget --keep-within-daily 7d --keep-within-weekly 1m --keep-within-monthly 1y --keep-within-yearly 75y` (note that `1w` is not a recognized duration, so you will have to specify `7d` instead).

For safety reasons, rustic refuses to act on an ”empty” policy. For example, if one were to specify `--keep-last 0` to forget *all* snapshots in the repository, rustic will respond that no snapshots will be removed. To delete all snapshots, use `--keep-last 1` and then finally remove the last snapshot manually (by passing the ID to `forget`).

11.3 Security considerations in append-only mode

Note: TL;DR: With append-only repositories, one should specifically use the `--keep-within` option of the `forget` command when removing snapshots.

To prevent a compromised backup client from deleting its backups (for example due to a ransomware infection), a repository service/backend can serve the repository in a so-called append-only mode. This means that the repository is served in such a way that it can only be written to and read from, while delete and overwrite operations are denied. rustic's `rest-server` *features an append-only mode, but few other standard backends do. To support append-only with such backends, one can use* `rclone` as a complement in between the backup client and the backend service.

- rust-server
- rclone

To remove snapshots and recover the corresponding disk space, the `forget` and `prune` commands require full read, write and delete access to the repository. If an attacker has this, the protection offered by append-only mode is naturally void. The usual and recommended setup with append-only repositories is therefore to use a separate and well-secured client whenever full access to the repository is needed, e.g. for administrative tasks such as running `forget`, `prune` and other maintenance commands.

However, even with append-only mode active and a separate, well-secured client used for administrative tasks, an attacker who is able to add garbage snapshots to the repository could bring the snapshot list into a state where all the legitimate snapshots risk being deleted by an unsuspecting administrator that runs the `forget` command with certain `--keep-*` options, leaving only the attacker's useless snapshots.

For example, if the `forget` policy is to keep three weekly snapshots, and the attacker adds an empty snapshot for each of the last three weeks, all with a timestamp (see the `backup` command's `--time` option) slightly more recent than the existing snapshots (but still within the target week), then the next time the repository administrator (or a scheduled job) runs the `forget` command with this policy, the legitimate snapshots will be removed (since the policy will keep only the most recent snapshot within each week). Even without running `prune`, recovering data would be messy and some metadata lost.

To avoid this, `forget` policies applied to append-only repositories should use the `--keep-within` option, as this will keep not only the attacker's snapshots but also the legitimate ones. Assuming the system time is correctly set when `forget` runs, this will allow the administrator to notice problems with the backup or the compromised host (e.g. by seeing more snapshots than usual or snapshots with suspicious timestamps). This is, of course, limited to the specified duration: if `forget --keep-within 7d` is run 8 days after the last good snapshot, then the attacker can still use that opportunity to remove all legitimate snapshots.

11.4 Customize pruning

To understand the custom options, we first explain how the pruning process works:

1. All snapshots and directories within snapshots are scanned to determine which data is still in use.

2. For all files in the repository, rustic finds out if the file is fully used, partly used or completely unused.
3. Completely unused files are marked for deletion. Fully used files are kept. A partially used file is either kept or marked for repacking depending on user options.

   Note that for repacking, rustic must download the file from the repository storage and re-upload the needed data in the repository. This can be very time-consuming for remote repositories.
4. After deciding what to do, `prune` will actually perform the repack, modify the index according to the changes and delete the obsolete files.

The `prune` command accepts the following options:

- `--max-unused limit` allow unused data up to the specified limit within the repository. This allows rustic to keep partly used files instead of repacking them.

  The limit can be specified in several ways:

  - As an absolute size (e.g. `200M`). If you want to minimize the space used by your repository, pass `0` to this option.
  - As a size relative to the total repo size (e.g. `10%`). This means that after prune, at most `10%` of the total data stored in the repo may be unused data. If the repo after prune has a size of 500MB, then at most 50MB may be unused.
  - If the string `unlimited` is passed, there is no limit for partly unused files. This means that as long as some data is still used within a file stored in the repo, rustic will just leave it there. Use this if you want to minimize the time and bandwidth used by the `prune` operation. Note that metadata will still be repacked. rustic tries to repack as little data as possible while still ensuring this limit for unused data. The default value is 5%.
- `--max-repack-size size` if set limits the total size of files to repack. As `prune` first stores all repacked files and deletes the obsolete files at the end, this option might be handy if you expect many files to be repacked and fear to run low on storage.
- `--repack-cacheable-only` if set to true only files which contain metadata and would be stored in the cache are repacked. Other pack files are not repacked if this option is set. This allows a very fast repacking using only cached data. It can, however, imply that the unused data in your repository exceeds the value given by `--max-unused`. The default value is false.
- `--dry-run` only show what `prune` would do.
- `--verbose` increased verbosity shows additional statistics for `prune`.

11.5 Recovering from ”no free space” errors

In some cases when a repository has grown large enough to fill up all disk space or the allocated quota, then `prune` might fail to free space. `prune` works in such a way that a repository remains usable no matter at which point the command is interrupted. However, this also means that `prune` requires some scratch space to work.

In most cases it is sufficient to instruct `prune` to remove all packs marked for removal and use as little scratch space as possible. Note that packs marked for removal are automatically removed by a prune run once they are old enough. If you can guarantee that the repository is not used by parallel processes, you can also use `rustic prune --instant-delete`.

To use as little scratch space as possibe, run `rustic prune --max-repack-size 0`. This removes all unneeded packs without repacking partly used packs. Obviously, this can only work if several snapshots have been removed using `forget` before. This then allows the `prune` command to actually

remove data from the repository. If the command succeeds, but there is still little free space, then remove a few more snapshots and run `prune` again.

Chapter 12

Stories

Stories of people using `rustic`:

- economic side of hosting rustic repo in AWS Glacier
- technical side of migrating to Glacier, restoring and such

Some talks about `restic` can be found here