Obnam integration test suite

http://obnam.org/

Contents

Chapter 1

Introduction

This is the Obnam integration test suite. Obnam is a backup program. The test suite is implemented using yarn, which is a black box testing tool for Unix programs, inspired by the BDD tools used by the Ruby community (Cucumber, Gherkin).

Obnam has extensive unit tests, which ensure individual functions, classes, and method work in isolation. The goal of the integration test suite is to make sure all the pieces work together. Thus, a typical integration test is to run Obnam in a specific kind of way, or against a specific kind of data, and then verify that the data can be restored correctly and that the repository is correct.

With yarn, tests are written up as "scenarios", and each scenario may consist of several steps. Each scenario, or some group of steps within a scenario, may tests one aspect of Obnam, or one way to use Obnam, or one error situation.

This test suite is meant to be comprehensible to those who would use Obnam, but aren't programmers, and would not understand the quite low-level unit tests. Test scenarios written for yarn are written as a document (this document), and each scenario consists of two parts: the scenario itself, and the nitty-gritty implementation part. The scenario is for everyone to understand, while the implementation part is meant for programmers, and others who understand shell script. The scenario describes, to any Obnam user, what is being tested, and at a very high level how, without having to understand the implementation part.

For more information:

- Obnam: http://obnam.org/
- Yarn: http://liw.fi/cmdtest/

Chapter 2

FIXME: Outline of test suite

This chapter will be removed, later, when all the outlined parts have been implemented.

- De-duplication
  - single client vs multiple clients
  - across generations
  - with encryption
- Lock handling
  - force-lock
- Repository management
  - fsck
- System administration
  - nagios-last-backup-age

Open questions:

- test accessing live data over sftp
- what errors should I test? can I test?
- different filesystems? run test suite multiple times, and set TMPDIR to point at a particular filesystem each time

Chapter 3

Test environment and setup

This chapter describes the environment that is set up for the tests to run in: how live data is generated, where it is kept, where the repository is kept and accessed, how verification of results happens, etc. This chapter is required reading for anyone wanting to understand what happens in the test suite.

The DATADIR directory

Yarn provides a directory temporary test data, and sets the `DATADIR` environment variable to point at that directory. The Obnam test suite uses that completely.

Live data

Many Obnam tests require generating some data to be backed up, which we call *live data*. Some tests will require modifying that data, for testing multiple backup generations. We use the genbackupdata tool to generate bulk data: it supports modifying an existing data tree, as well as just generating data.

We care about not just the amount of data, but also how it is distributed, all kinds of file metadata, and all types of filesystem features that may affect backups. For example, sparse files verse dense ones; extended attributes; empty files; and symbolic links. We generate these using a small helper utility program that comes with Obnam, called `mkfunnyfarm`, which creates a small directory tree with all the interesting kind of filesystem objects we know about.

Some filesystem objects require root permission to create: device nodes, for example. For these, we assume that the unit tests are sufficient: they can inspect

the execution of the relevant code paths in much more detail than integration tests.

We store the live data we generate in `$DATADIR/live-data`. In a scenario that involves multiple clients, each client has its own set of live data at `$DATADIR/$CLIENT.live-data`.

Repositories

The backup repository is stored at `$DATADIR/repo`. In a scenario that involves multiple clients, each client has its own repository at `$DATADIR/$CLIENT.repo`.

The repository is accessed either via the local filesystem, using the directory name described above, or over the `sftp` protocol over `localhost`, using a URL of the form `sftp://localhost$DATADIR/repo`. For this to work, the user running the test suite needs to have ssh access over localhost, without requiring a password to be entered. The user may disable such tests when the test suite is running, by asking yarn to set the `OBNAM_TEST_SSH` to `no`.

Obnam configuration and multiple users/clients

In the tests, Obnam is run without a default configuration (`--no-default-config`), to avoid the user's settings affecting the test suite, or, indeed, having the test suite accidentally wreck the user's backup repository.

The shell library for the Obnam test suite provides a `run_obnam` shell function, which runs Obnam in the right way for the tests. All tests that run Obnam MUST use `run_obnam`. In addition to adding the `--no-default-config` option, it also tells Obnam to use the `$DATADIR/obnam.conf` configuration file, for single-client tests, or `$DATADIR/$CLIENT.obnam.conf` for multi-client tests.

We simulate multiple clients by providing each client with a different configuration, though on the command line instead of using configuration files. The relevant settings are `--client-name` and the backup roots (command line arguments to `obnam backup` or the `--root` setting).

Result verification

We verify backups by multiple methods:

1. Restore the data, and compare the restored data with the original.
2. Use the `obnam verify` command.

3. Run `obnam fsck` on the repository after any operation that may have change the repository.

The verification is done by using the summain checksum/manifest tool, written for this purpose. We run summain against the live data before making a backup, and again on the restored data, and then compare the two manifests (with diff). If they're identical, everything went well, otherwise there's a problem somewhere.

IMPLEMENTS sections

The IMPLEMENTS sections are where the nitty-gritty details is specified of what happens for each scenario test step. The sections fall into two classes: generic ones, and those specific to a scenario or small set of scenarios.

The generic ones are discussed and shown in their own chapter. The ones specific to some scenarios are kept with the scenarios using them.

Chapter 4

Basic operation: backup and restore

This chapter tests the basic operation of Obnam: backing up and restoring data. Tests in this chapter only concern themselves with a single generation; see later for tests for multiple generations.

The goal of this chapter is to test Obnam with every kind of data, every kind of file, and every kind of metadata.

Backup simple data

This is the simplest of all simple backup tests: generate a small amount of data in regular files, in a single directory, and backup that. No symlinks, no empty files, no extended attributes, no nothing. Just a few files with a bit of data in each. This is what every backup program must be able to handle.

```
SCENARIO backup simple data
GIVEN 100kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
AND user U can fsck the repository R
```

Backup sparse files

Sparse files present an interesting challenge to backup programs. Most people have none, but some people have lots, and theirs can have very large holes. For example, at work I often generate disk images as raw disk images in sparse files. The image may need to be, say 30 gigabytes in size, even though it only contains one or two gigabyte of data. The rest is a hole.

A backup program should restore a sparse file as a sparse file. Otherwise, the 30 gigabyte disk image file will, upon restore, use 30 gigabytes of disk space, rather than one. That might make restoring impossible.

Unfortunately, it is not easy to (portably) check whether a file is sparse. We'll settle for making sure the restored file does not use more disk space than the one in live data.

```
SCENARIO backup a sparse file
GIVEN a file S in L, with a hole, data, a hole
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
AND file S from L, restored in X doesn't use more disk
```

Backup all interesting file and metadata types

The Unix filesystem abstraction is surprisingly complicated. Indeed, it can come as a surprise to anyone who's not implemented a backup program with the intention of being able to restore the live data set exactly. To complicate things further, different filesystems have different features, and different Unix-like operating systems don't all implement all the features, and implement some features differently.

We need to ensure Obnam can handle anything it encounters, on any supported platform. That is the purpose of the scenarios in this section. There are some limitations, though: the test suite is not run as the `root` user, and thus we don't deal with filesystem objects that require privileged operations such as device node creation. We also don't, in these scenarios, handle multiple filesystem types: the test suite should, instead, be run multiple types, with `TMPDIR` set to point at a different filesystem type each time: we leave that to the user running the test suite.

We rely on a helper tool in the Obnam source tree, `mkfunnyfarm`, to create all the interesting filesystem objects, rather than spelling them out in the scenarios. This is because that helper tool is used by other parts of Obnam's test suite as well, and this reduces code duplication.

```
SCENARIO backup non-basic filesystem objects
ASSUMING extended attributes are allowed for users
GIVEN directory L with interesting filesystem objects
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

As a special case, Obnam needs to notice when only an extended attribute value changes.

```
SCENARIO backup notices when extended attribute value changes
ASSUMING extended attributes are allowed for users
GIVEN a file F in L, with data
AND file L/F has extended attribute user.foo set to foo
WHEN user U backs up directory L to repository R
GIVEN file L/F has extended attribute user.foo set to bar
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Backup without changes

If we run a backup, then a new one, then the new generation should match the first one, and no files should be have been backed up in the second generation.

```
SCENARIO no-op backup
GIVEN a file F in L, with data
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Remove the Obnam log file, so we only have the log from the next backup run.

```
WHEN user U removes file obnam.log
AND user U backs up directory L to repository R
THEN obnam.log matches INFO \* files backed up: 0$
AND L, restored to X, matches manifest M
```

Backup when a file or directory is unreadable

The backup shouldn't fail even if a file or directory is inaccessible.

```
SCENARIO unreadable live data file
```

We can't run this test as the `root` user, since then everything is readable.

```
ASSUMING not running as root
```

Create some live data, and a file that is unreadable.

```
GIVEN 1k of new data in directory L
AND file L/unreadable-file with permissions 000
WHEN user U attempts to back up directory L to repository R
THEN the error message matches "RCE08AX.*L/unreadable-file"
WHEN user U attempts to verify L against repository R
THEN the attempt succeeded
```

Next, let's do the same thing again, but with an unreadable directory instead of a file.

```
SCENARIO unreadable live data directory
ASSUMING not running as root
GIVEN 1k of new data in directory L
AND directory L/unreadable-dir with permissions 000
WHEN user U attempts to back up directory L to repository R
THEN the error message matches "RD5FA4X.*L/unreadable-dir"
WHEN user U attempts to verify L against repository R
THEN the attempt succeeded
```

Backup to roots at once

Often it's useful to backup more than one location at once. We'll assume that if we can backup two, then it'll all work well.

```
SCENARIO backup two roots
GIVEN 4kB of new data in directory L1
AND 16kB of new data in directory L2
AND a manifest of L1 in M1
AND a manifest of L2 in M2
WHEN user U backs up directories L1 and L2 to repository R
AND user U restores their latest generation in repository R into X
THEN L1, restored to X, matches manifest M1
THEN L2, restored to X, matches manifest M2
```

Checkpoint generations

Obnam is meant to remove checkpoint generations it created during a backup, if the backup finishes successfully.

```
SCENARIO checkpoint generations are removed
GIVEN 100kB of new data in directory L
AND user U sets configuration checkpoint to 1k
WHEN user U backs up directory L to repository R
THEN user U sees no checkpoint generations in repository R
```

Restore a single file

We need to be able to restore only a single file. Note that when restoring a single file, we do not set the parent directory's modification time according to the backup, so we need to manipulate the manifest to avoid getting an error.

```
SCENARIO restore a single file
GIVEN a file F in L, with data
AND a manifest of L/F in M
WHEN user U backs up directory L to repository R
AND user U restores file L/F to X from their latest generation in repository R
THEN L/F, restored to X, matches manifest M
```

Restores must happen to a non-existent or an empty directory

To avoid people doing unfortunate things such as `obnam restore --to=/` we make sure the target directory of restore either does not exist, or it's empty.

```
SCENARIO restore only to empty or new target
GIVEN 1kB of new data in directory L
AND a manifest of L in M
AND 0kB of new data in directory EMPTY
AND 2kB of new data in directory NOTEMPTY

WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into EMPTY
THEN L, restored to EMPTY, matches manifest M

WHEN user U restores their latest generation in repository R into NOTEXIST
THEN L, restored to NOTEXIST, matches manifest M

WHEN user U attempts to restore their latest generation
... in repository R into NOTEMPTY
THEN the attempt failed with exit code 1
```

Pretend backing up: the `--pretend` setting

The `--pretend` setting lets the user pretend they're doing a backup, without actually having anything backed up. This is useful for testing that the configuration is correct: the fake backup runs much faster than a real one.

```
SCENARIO a pretend backup
GIVEN 10kB of new data in directory L
WHEN user U backs up directory L to repository R
GIVEN a manifest of R in M1
WHEN user U pretends to back up directory L to repository R
GIVEN a manifest of R in M2
THEN manifests M1 and M2 match
```

Exclude cache directories

The Cache directory tagging standard provides an easy way to mark specific directories as cache directories, which means their data is easy to re-create (or re-download). Such data is often not worth backing up. The `--exclude-caches` option tells Obnam to exclude any directories tagged like that.

```
SCENARIO exclude cache directories
GIVEN 1k of new data in directory L/wanted
AND 1k of new data in directory L/cache
AND directory L/cache is tagged as a cache directory
```

We'll now create the manifest, but remove `L/cache` (and files in `L/cache`) so that it matches what we need. We do it this instead of creating the manifest before `L/cache`, because creating `L/cache` changes the timestamp of `L`.

```
AND a manifest of L in M
AND cache is removed from manifest M
```

Time to backup.

```
AND user U sets configuration exclude-caches to yes
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Excluded, already backed up files, are not included in next generation

Until Obnam version 1.7.4, but fixed after that, Obnam had a bug where a file that was not excluded became excluded was not removed from new backup

generations. In other words, if file `foo` exists and is backed up, and the user then makes a new backup with `--exclude=foo`, the new backup generation still contains `foo`. This is clearly a bug. This scenario verifies that the bug no longer exists, and prevents it from recurring.

```
SCENARIO new generation drops excluded, previously backed up files
GIVEN a file foo in L, with data
WHEN user U backs up directory L to repository R
GIVEN user U sets configuration exclude to foo
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, is empty
```

Changing backup roots

When we change the backup roots, i.e., the directories we want backed up, we do not want the any dropped backup roots to be included in the new backup.

```
SCENARIO replace backup root with new one
GIVEN 1k of new data in directory L1
AND 1k of new data in directory L2
WHEN user U backs up directory L1 to repository R
AND user U backs up directory L2 to repository R
AND user U lists latest generation in repository R into F
THEN nothing in F matches L1
```

Pre-epoch timestamps

It's possible to have timestamps before the epoch, i.e., negative ones. For example, in the UK during DST, `touch -t 197001010000` will create one. Test that such timestamps work.

```
SCENARIO pre-epoch timestamps
GIVEN file L/file has Unix timestamp -3600
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Change B-tree node size

The setting for B-tree node size (`--node-size`) only affects new B-trees. Thus, if we've backed up with one size, and change the setting to a new size, the backup should still work.

```
SCENARIO backup with changed B-tree node size
GIVEN 100kB of new data in directory L
AND user U sets configuration node-size to 65536
WHEN user U backs up directory L to repository R
GIVEN 100Kb of new data in directory L
AND a manifest of L in M
AND user U sets configuration node-size to 4096
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
AND user U can fsck the repository R
```

Chapter 5

Multiple backup generations

This chapter contains tests for Obnam's handling of multiple generations: making incremental backups, forgetting generations, and so on. We assume that backing up any individual directory tree works fine, regardless of whether it is for the initial generation or an incremental one. In the previous chapter for basic backups, we've already dealt with those. This chapter focuses on generation handling only.

Incremental backup generations (`obnam backup`)

First of all, most importantly, we must be able to make more than one backup generation, and restore them. The live data in each generation is different, but there are unchanged parts as well. For simplicity, we'll assume that if we can do two generations, we can do any number. It's possible that the 12765th generation might break, but that's unlikely, and it's even less likely we'll guess it. (If it turns out to actually happen, we'll add a regression test when we find the problem.)

```
SCENARIO backup two generations
GIVEN 1MB of new data in directory L
AND a manifest of L in G1
WHEN user U backs up directory L to repository R
GIVEN 2MB of new data in directory L
AND a manifest of L in G2
WHEN user U backs up directory L to repository R
AND user U restores generation 1 to R1 from repository R
AND user U restores generation 2 to R2 from repository R
THEN L, restored to R1, matches manifest G1
```

```
AND L, restored to R2, matches manifest G2
```

Listing generations (`obnam generations`, `obnam genids`)

When we make some number of generations, the Obnam generation listing commands should show that number of generations.

```
SCENARIO list generations
GIVEN 1MB of new data in directory L
WHEN user U backs up directory L to repository R
AND user U backs up directory L to repository R
AND user U backs up directory L to repository R
THEN user U sees 3 generations in repository R
AND user U sees 3 generation ids in repository R
```

Listing contents of a generation (`obnam ls`)

We'll assume the `obnam ls` command shows any generation. However, there's a couple of ways of using it: either listing everything, or only a specific directory to list.

```
SCENARIO list generation content
GIVEN 1MB of new data in directory D
WHEN user U backs up directory D to repository R
AND user U lists latest generation in repository R into all.txt
THEN all.txt matches /.*/D/.
WHEN user U lists D in latest generation in repository R into some.txt
THEN all lines in some.txt match (/D|Generation)
```

The first line of the generation listing contains the word "Generation". Every other line should contain the directory we requested as part of the pathname.

There was a bug in Obnam 1.5 (and possibly other versions) that listing contents of a directory that ends in a slash (but isn't the root directory) fails. The following is a test for that bug by requesting `D/` to be listed, and verifying that we get at least one line for that.

```
WHEN user U lists D/ in latest generation in repository R into bug.txt
THEN bug.txt matches /D
```

Comparing generations (`obnam diff`)

Once we've backed up two generations, we need to be able to see the difference. First of all, the diff should be empty when the generations are identical:

```
SCENARIO diff identical generations
GIVEN 1K of new data in directory L
WHEN user U backs up directory L to repository R
AND user U backs up directory L to repository R
AND user U diffs generations 1 and 2 in repository R into D
THEN file D is empty
```

`obnam diff` can be used with just one generation, and that compares it with the generation preceding the given one.

```
WHEN user U diffs latest generation in repository R into D
THEN file D is empty
```

If we make a change to the data, that should be reflected in the diff. We'll assume the diff works, we'll just check whether it's empty.

```
SCENARIO diff modified generations
GIVEN 1K of new data in directory L
WHEN user U backs up directory L to repository R
GIVEN 1K of new data in directory L
WHEN user U backs up directory L to repository R
AND user U diffs generations 1 and 2 in repository R into D
THEN file D is not empty
```

`obnam forget` does nothing by default

`obnam forget` is the command to remove backup generations from the repository. It can be used to remove specific generations, or to remove generations according to a schedule. If neither is specified, it should do nothing.

```
SCENARIO forget does nothing by default
GIVEN 1K of new data in directory L
AND a manifest of L in M

WHEN user U backs up directory L to repository R
AND user U runs obnam forget without generations or keep policy on repository R
THEN user U sees 1 generation in repository R

WHEN user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Forgetting a specific generation (`obnam forget`)

We need to be able to remove any generation. As a corner case, we should be able to remove the only generation. We'll test by making two generations, then removing both, and after removing the first one, checking that the remaining one is the one we want.

```
SCENARIO remove specific generations
GIVEN 1kB of new data in directory L
AND a manifest of L in M1
WHEN user U backs up directory L to repository R
GIVEN 1kB of new data in directory L
AND a manifest of L in M2
WHEN user U backs up directory L to repository R
AND user U forgets the oldest generation in repository R
THEN user U sees 1 generation in repository R
WHEN user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M2
WHEN user U forgets the oldest generation in repository R
THEN user U sees 0 generations in repository R
```

Forgetting generations according to a schedule (`obnam forget --keep`)

The normal way of forgetting generations is with the `obnam forget --keep` option.

```
SCENARIO remove generations according to schedule
GIVEN 1kB of new data in directory L
WHEN user U backs up directory L to repository R
GIVEN 1kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U forgets according to schedule 1y in repository R
THEN user U sees 1 generation in repository R
WHEN user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

There has been reports that the "keep N hourly backups" type of `--keep` policy doesn't work. Test this by creating several generations, pretending the time is something specific, and then check that the right ones get kept. For each calendar hour, we make two generations, and we create them for every other calendar hour, for four such hours (covering a total of eight hours). We then keep two hourly backups. This should result in the later of each backup during a calendar hour to be kept, for the last two calendar hours.

```
SCENARIO keep N hourly generations
```

The first generation of the first hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 01:00:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The second generation of the first hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 01:30:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The first generation of the second hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 02:00:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The second generation of the second hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 02:30:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The first generation of the third hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 03:00:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The second generation of the third hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 03:30:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The first generation of the fourth hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 04:00:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

The second generation of the fourth hour.

```
GIVEN user U sets configuration pretend-time to 2014-03-19 04:30:00
AND 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

Now run the forget and verify.

```
WHEN user U forgets according to schedule 2h in repository R
THEN user U sees 2 generations in repository R
AND user U has 1st generation timestamp 2014-03-19 03:30:00 in repository R
```

```
AND user U has 2nd generation timestamp 2014-03-19 04:30:00 in repository R
```

Dnn't really forget anything if pretending

The `--pretend` option prevents `obnam forget` from actually removing anything, but lets the user see what would be removed.

```
SCENARIO forget doesn't really, when pretending
GIVEN 1kB of new data in directory L
WHEN user U backs up directory L to repository R
GIVEN 1kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U pretends to forget according to schedule 1y in repository R
THEN user U sees 2 generations in repository R
WHEN user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Chapter 6

Multiple clients sharing a repository

An Obnam backup repository may be shared by multiple clients. There are a couple of aspects of this that need testing: whether it works at all, when each client is run in sequence, and whether it works concurrently, with locks used to exclude other clients from modifying the shared data.

The concurrency is really hard to test well. There is a non-yarn test for locking, which we assume will test that, and so in this yarn test suite we do not test concurrency at all.

```
SCENARIO two clients sharing a repository
GIVEN 64kB of new data in directory L1
AND 96kB of new data in directory L2
AND a manifest of L1 in M1
AND a manifest of L2 in M2
WHEN user U1 backs up directory L1 to repository R
WHEN user U2 backs up directory L2 to repository R
AND user U1 restores their latest generation in repository R into X1
AND user U2 restores their latest generation in repository R into X2
THEN L1, restored to X1, matches manifest M1
AND L2, restored to X2, matches manifest M2
```

In addition to backing up, we check for forget working. We first make a change to both sets of live data, and

```
GIVEN 1kB of new data in directory L1
AND 2kB of new data in directory L2
AND a manifest of L1 in M1A
AND a manifest of L2 in M2A
WHEN user U1 backs up directory L1 to repository R
```

```
AND user U2 backs up directory L2 to repository R
AND user U1 forgets the oldest generation in repository R
AND user U2 forgets the oldest generation in repository R
AND user U1 restores their latest generation in repository R into X1A
AND user U2 restores their latest generation in repository R into X2A
THEN L1, restored to X1A, matches manifest M1A
AND L2, restored to X2A, matches manifest M2A
```

Listing clients

In a repository shared by many clients, it may be necessary to list the names, and sometimes to do that without being one of the clients. For example, when restoring a machine that has been destroyed, you may need to list the clients to pick the right one to restore. This test verifies that you can list the clients even if you don't know an existing client's name.

```
SCENARIO list clients without being in the client list
GIVEN 1kB of new data in directory L
WHEN user U1 backs up directory L to repository R
THEN user U2 can see user U1 in repository R
```

Two clients sharing chunks, one forgets its generations

What happens when two clients share chunks and one of them forgets its generations? A problem was found in this scenario by Nemo Inis in 2014.

```
SCENARIO two clients share chunks and one forgets is generations
GIVEN 1k of new data in directory L
AND a manifest of L in M
WHEN user U1 backs up directory L to repository R
AND user U2 backs up directory L to repository R
AND user U1 forgets the oldest generation in repository R
AND user U2 restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Chapter 7

Encrypted repositories

Obnam repositories may be encrypted. The encryption is based on public keys, using GnuPG specifically. Internally, symmetric encryption is also used, but that is not visible, nor relevant, to the user. All encryption requires some level of key management, so the encryption plugin in Obnam provides a number of subcommands for that.

We need to test, at minimum, that key management works. Ideally, we'd also test that encryption works, but that's trickier to achieve without making assumptions about the repository format.

Test setup

We need two PGP keys for these tests, and they need to be independent of each other so that tests can meaningfully use the different keys to pretend they're different users. We have, in the Obnam source tree, two GnuPG keyrings (`test-data/keyring-1` and `test-data/keyring-2`), which we use for this purpose. We use pre-generated keys instead of generating new ones for each test run, since key generation is a fairly heavy operation that easily depletes the host of entropy.

However, to avoid inadvertent changes to the keys, keyrings, random data seeds, or other files, we make a copy of the data into `$DATADIR` for the duration of the test.

The keys have usernames `Test Key One` and `Test Key Two` (no e-mail addresses). They have no passphrase. Otherwise, they are generated using GnuPG defaults (as of 1.4.12 in Debian wheezy).

Encrypted backup and restore

We'll make a simple backup and restore using encryption. If this works, we can probably assume that any other normal repository operations (those not part of encryption management) also work, given that encryption is done at the I/O abstraction level.

```
SCENARIO basic encrypted backup and restore
GIVEN user U uses encryption key "Test Key One" from test-data/keyring-1
AND 128kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Keys provided by a custom directory

We'll make a simple backup and restore using encryption. If this works, we can probably assume that any other normal repository operations (those not part of encryption management) also work, given that encryption is done at the I/O abstraction level.

```
SCENARIO encrypted backup and restore with a separate keyring
GIVEN user U separately uses encryption key "Test Key One" from test-data/keyring-1
AND 128kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Adding and removing keys to clients

Each client specifies the key they want to use with the `--encrypt-with` setting. This is the primary key for the client. The client may additionally use other keys to encrypt to: this allows, for example, having a repository-wide encryption key that can run fsck or forget.

We test these by having two keys: one for the primary one, and a second one, and verifying that we can, or can't, access the backup with the second key, depending on whether it has or hasn't been added to the client.

First of all, we make a simple encrypted backup as the first client.

```
SCENARIO adding and removing encryption keys to clients
GIVEN user U1 uses encryption key "Test Key One" from test-data/keyring-1
```

```
AND 16kB of new data in directory L1
WHEN user U1 backs up directory L1 to repository R
THEN user U1 uses key "Test Key One" in repository R
```

Then we add the key of the second client to the repository. This is necessary, because by now the client list is already encrypted using only the first client's key, meaning the second client has no access to the client list, and thus can't add itself.

```
WHEN user U1 imports public key "Test Key Two" from test-data/keyring-2
AND user U1 adds key "Test Key Two" to repository R only
```

Then we make a backup as the second client.

```
GIVEN user U2 uses encryption key "Test Key Two" from test-data/keyring-2
AND 32kB of new data in directory L2
WHEN user U2 backs up directory L2 to repository R
THEN user U2 uses key "Test Key Two" in repository R
```

Let's make sure both clients can still restore their own data.

```
GIVEN a manifest of L1 in M1
WHEN user U1 restores their latest generation in repository R into X1
THEN L1, restored to X1, matches manifest M1

GIVEN a manifest of L2 in M2
WHEN user U2 restores their latest generation in repository R into X2
THEN L2, restored to X2, matches manifest M2
```

An unrelated client, which happens to use the same name as the first client, should not be able to access the data.

```
GIVEN a user U3 calling themselves U1
WHEN user U3 attempts to restore their latest generation in repository R into X3
THEN the attempt failed with exit code 1
AND the error message matches "R0C79EX: gpg failed"
AND the error message matches "secret key not available\|No secret key"
```

(The error message above indicates that there's a bug in Obnam, which is that the error message, when an encryption is not provided but the repository uses encryption, is not very clear. This should be FIXME'd some day.)

Likewise, even if a client has access to their own data, they should not have access to another client's data.

```
GIVEN a user U2 calling themselves U1
WHEN user U2 attempts to restore their latest generation in repository R into X4
THEN the attempt failed with exit code 1
AND the error message matches "secret key not available\|No secret key"
```

Replace a key for a client

If we replace the key for a client in a repository, and then the client gets rid of the old key, the new key should be able to restore old backups.

First, backup using the old key.

```
SCENARIO replace client key
GIVEN user U uses encryption key "Test Key One" from test-data/keyring-1
AND 1kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
```

Then, replace the old key with the new one and get rid of the old key.

```
GIVEN user U uses encryption key "Test Key Two" from test-data/keyring-2
WHEN user U adds key "Test Key Two" to repository R and self
AND user U removes key "Test Key One" from repository R
WHEN user U no longer has key "Test Key One"
```

Finally, verify that restores still work with the new key.

```
WHEN user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
```

Key queries

Obnam has a couple of commands to list the keys in the repository and what they have access to (`list-keys`, `list-toplevels`). These are primarily useful for debugging, and not not worth writing tests for (at least for now).

Removing a client

Obnam has a `obnam remove-client` command which currently only works when encryption is used. This is a wart, a bug, and a disgrace. However, it will be fixed some day, and until then the command is tested in this chapter.

First we make a backup as one client, then we add a second key to the repository. Finally, we remove the client and verify no clients remain.

```
SCENARIO remove a client
GIVEN user U1 uses encryption key "Test Key One" from test-data/keyring-1
AND user U2 uses encryption key "Test Key Two" from test-data/keyring-2
AND 48kB of new data in directory L
WHEN user U1 backs up directory L to repository R
THEN user U1 uses key "Test Key One" in repository R
```

```
WHEN user U1 imports public key "Test Key Two" from test-data/keyring-2
AND user U1 adds key "Test Key Two" to repository R only
AND user U2 removes user U1 from repository R
THEN user U2 can't see user U1 in repository R
```

Chapter 8

Compressed repositories

Backup and restore with compression

Compressed backups should work just like normal ones, except with all the data written to the repository being compressed. We thus repeat a basic backup test, but with compression.

```
SCENARIO backup simple data using compression
GIVEN 100kB of new data in directory L
AND a manifest of L in M
AND user U sets configuration compress-with to deflate
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
AND user U can fsck the repository R
```

If this works, everything else should work as well: by the time Obnam writes anything to the repository, it has lost its sparseness, or other special filesystem characteristics, and it's just B-tree nodes or chunk data.

It should be possible to restore from a compressed repository, even without turning compression on. We continue the test scenario accordingly.

```
GIVEN user U sets configuration compress-with to none
WHEN user U restores their latest generation in repository R into Y
THEN L, restored to Y, matches manifest M
AND user U can fsck the repository R
```

We would also like to turn compression on after not using it, without having to do any extra work. This requires a new scenario.

```
SCENARIO enable compression later
```

```
GIVEN 100kB of new data in directory L
WHEN user U backs up directory L to repository R
```

Then enable compression, add some more data, and backup again. The result should now be verifiable.

```
GIVEN user U sets configuration compress-with to deflate
AND 100kB of new data in directory L
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
AND user U can fsck the repository R
```

Using both compression and encryption

We need to be able to combine compression with encryption. Let's do the test again, with new options.

```
SCENARIO backup simple data using compression and encryption
GIVEN 100kB of new data in directory L
AND a manifest of L in M
AND user U sets configuration compress-with to deflate
GIVEN user U uses encryption key "Test Key One" from test-data/keyring-1
WHEN user U backs up directory L to repository R
AND user U restores their latest generation in repository R into X
THEN L, restored to X, matches manifest M
AND user U can fsck the repository R
```

Chapter 9

Verify backed up data

Verify notices modification time change

The user may want to verify that all their live data is still intact. This is done with `obnam verify`.

```
SCENARIO verify notices mtime change
GIVEN 100kB of new data in directory L
AND file L/foo has Unix timestamp 0
WHEN user U backs up directory L to repository R
AND user U attempts to verify L against repository R
THEN the attempt succeeded
```

However, if there have been any changes, such as for a timestamp, then the verification should find a problem.

```
GIVEN file L/foo has Unix timestamp 1
WHEN user U attempts to verify L against repository R
THEN the attempt failed with exit code 1
```

RDF30DX is the error code for metadata change, of which modification time is one.

```
THEN the error message matches "RDF30DX.*st_mtime_sec"
```

Verify one file randomly

`obnam verify` can pick files to verify randomly, for spot checks.

```
SCENARIO verify a random file
GIVEN 100kB of new data in directory L
```

```
WHEN user U backs up directory L to repository R
AND user U attempts to verify a random file in  L against repository R
THEN the attempt succeeded
```

Verify notices when live data file has been appended to

In March, 2015, Thomas Waldemann noticed that `obnam verify` would fail to notice if the live data file had been appended to. This regression test catches the problem.

```
SCENARIO verify file that has been appended to
GIVEN 0B of data in file L/foo
AND file L/foo has Unix timestamp 0
WHEN user U backs up directory L to repository R
GIVEN 1B of data in file L/foo
AND file L/foo has Unix timestamp 0
WHEN user U attempts to verify L against repository R
THEN the attempt failed with exit code 1
```

Chapter 10

Lock handling

This chapter contains scenarios for testing Obnam's lock handling, specifically the forcing of locks to become open when lock files have been left by Obnam for whatever reason.

Basic forcing of a lock

In this scenario, we force a repository to be locked, and force the lock open. To do this, we use an Obnam command that locks the desired parts of the repository, and does nothing else; this is a testing aid.

```
SCENARIO force repository open
```

We first create the repository and back up some data.

```
GIVEN 1kB of new data in directory L
WHEN user U backs up directory L to repository R
```

We then lock the repository, and verify that a backup now fails.

```
AND user U locks repository R
AND user U attempts to back up directory L to repository R
THEN the attempt failed with exit code 1
AND the error message matches "R681AEX"
```

Now we can force the lock open and verify that a backup now succeeds.

```
WHEN user U forces open the lock on repository R
AND user U attempts to back up directory L to repository R
THEN the attempt succeeded
```

Forcing of someone else's lock

We also need to force a lock by someone else. This is otherwise similar to the basic lock forcing scenario, but the lock is held by a different client. The lock is created before the second client even gets added to the client list, to maximise the difficulty.

```
SCENARIO force someone else's lock
```

We first create the repository and back up some data as the first client.

```
GIVEN 1kB of new data in directory L
WHEN user U1 backs up directory L to repository R
```

We then lock the repository as the first user, and verify that a backup now fails when run as the second client.

```
AND user U1 locks repository R
AND user U2 attempts to back up directory L to repository R
THEN the attempt failed with exit code 1
AND the error message matches "R681AEX"
```

The second client can force the lock open and successfully back up.

```
WHEN user U2 forces open the lock on repository R
AND user U2 attempts to back up directory L to repository R
THEN the attempt succeeded
```

Chapter 11

FUSE plugin

The FUSE plugin gives read-only access to a backup repository. There's a lot of potential corner cases here, but for now, this test suite concentrates on verifying that at least the basics work.

```
SCENARIO Browsing backups with FUSE plugin
ASSUMING user can use FUSE
AND extended attributes are allowed for users
GIVEN directory L with interesting filesystem objects
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U FUSE mounts the repository R at F
THEN L, restored to F/latest, matches manifest M
```

The FUSE view of the repository won't change while we have it mounted, even if we make a new backup.

```
GIVEN 100k of new data in directory L
AND a manifest of L in M2
WHEN user U backs up directory L to repository R
THEN L, restored to F/latest, matches manifest M
```

However, if we read the file `F/.pid`, the FUSE plugin refreshes the view and we can now see the new backup.

```
WHEN user U reads file F/.pid
THEN L, restored to F/latest, matches manifest M2
```

Clean up.

```
FINALLY unmount repository F
```

In 2014, for Obnam 1.7, a bug was reported that the FUSE plugin would only read the first 64 kilobytes of a file. Verify that this is no longer a problem.

```
SCENARIO restoring a big file with FUSE
ASSUMING user can use FUSE
GIVEN 1M of data in file L/big.dat
AND a manifest of L in M
WHEN user U backs up directory L to repository R
AND user U FUSE mounts the repository R at F
THEN L, restored to F/latest, matches manifest M
AND big.dat in L and in mounted F compare equally
FINALLY unmount repository F
```

We can only run this test if the user is in the `fuse` group. This may be a portability concern: this works in Debian GNU/Linux, but might be different in other Linux distros, or on non-Linux systems. (If it doesn't work for you, please report a bug.)

We do the backup, and verify that it can be accessed correctly, by doing a manifest of the live data before the backup, and then against the FUSE mount, and comparing the two manifests.

```
IMPLEMENTS WHEN user (\S+) FUSE mounts the repository (\S+) at (\S+)
mkdir "$DATADIR/$MATCH_3"
run_obnam "$MATCH_1" mount -r "$DATADIR/$MATCH_2" \
    --to "$DATADIR/$MATCH_3"
```

We also check a specific file by comparing it in the mount and in its original location. We do the comparison with cmp(1) instead of the usual way, because this triggered a bug.

```
IMPLEMENTS THEN (\S+) in (\S+) and in mounted (\S+) compare equally
cmp \
    "$DATADIR/$MATCH_2/$MATCH_1" \
    "$DATADIR/$MATCH_3/latest/$DATADIR/$MATCH_2/$MATCH_1"
```

If we did do the fuse mount, **always** unmount it, even when a step failed. We do not want failed test runs to leavo mounts lying around.

```
IMPLEMENTS FINALLY unmount repository (\S+)
if [ -e "$DATADIR/$MATCH_1" ]
then
    fusermount -u "$DATADIR/$MATCH_1"
fi
```

Chapter 12

System administration tasks

System administrators may want to do some tasks related to backups that normal users might not care about, such as monitoring and integrity checking.

Nagios monitoring support

Obnam has a command to help it be integrated into a monitoring system using Nagios plugins. First, setup a known configuration of the plugin to make things testable.

```
SCENARIO Nagios monitoring support
GIVEN user U sets configuration warn-age to 1h
AND user U sets configuration critical-age to 1d
```

Then make a backup at a known (pretended) time.

```
GIVEN 1kB of new data in directory L
AND user U sets configuration pretend-time to 1999-01-01 00:00:00
WHEN user U backs up directory L to repository R
```

Now, pretend that it's an hour and a second later. We should now be getting a warning.

```
GIVEN user U sets configuration pretend-time to 1999-01-01 01:00:01
WHEN user U attempts nagios-last-backup-age against repository R
THEN the attempt failed with exit code 1
AND the output matches "^WARNING:"
```

If it's more than a day later (just a second over), there should be an error.

```
GIVEN user U sets configuration pretend-time to 1999-01-02 00:00:01
WHEN user U attempts nagios-last-backup-age against repository R
```

```
THEN the attempt failed with exit code 2
AND the output matches "^CRITICAL:"
```

Chapter 13

Robustness: dealing with repository corruption

A repository may be corrupted in various ways, including due to bugs in Obnam itself. Obnam needs to be robust against this, and do as well as it can, even when the repository isn't quite as good as it might be. For example, it should be able to restore data that is still in the repository.

The scenario in this chapter handles a specific class of repository corruption: file data ("chunks") that have gone missing. As of Obnam 1.12, there are known to be bugs that cause that to happen. Hopefully, once these scenarios pass, the bugs will either be fixed, or at least are handled without crashing by later Obnam operations.

```
SCENARIO handle missing file chunks
```

First, let's create a repository that's OK. We'll make two backup generations, with some changes to live data in between.

```
GIVEN 10k of data in file L/foo
AND a manifest of L in M
WHEN user U backs up directory L to repository R
GIVEN a manifest of R in MR
AND a copy of R in R1

GIVEN 20k of data in file L/bar
AND a copy of L/foo in L/foocopy
WHEN user U backs up directory L to repository R
```

We now have the first generation that has just the file `L/foo`, and the second generation that has `L/bar` and `L/foocopy`, and the latter is identical to the `L/foo`. Because it is identical, it will re-use the file chunks of `L/foo`.

If we now remove the chunks that were created by the second backup, the first generation is intact, but the second generation's `L/bar` file is corrupt (it's chunks are missing).

```
WHEN repository R resets its chunks to those in R1
```

We should now be able to restore the first generation without problems.

```
WHEN user U restores generation 1 to X1 from repository R
THEN L, restored to X1, matches manifest M
```

Restoring the second generation should fail, partially.

```
WHEN user U attempts to restore their latest generation
... in repository R into X2
THEN the attempt failed with exit code 1
AND the error message matches "L/bar: R43272X"
AND file L/foo, restored to X2, matches live data
AND file L/foocopy, restored to X2, matches live data
```

We should be able to remove the second generation, despite the missing chunk.

```
WHEN user U forgets their latest generation in repository R
THEN user U sees 1 generation in repository R

WHEN user U restores their latest generation in repository R
... into X3
THEN L, restored to X3, matches manifest M
```

Chapter 14

Multiple repository format handling

Obnam supports (or, rather, will in the future; FIXME) several repository formats. As Obnam development progresses, there is sometimes a need to change the way data is stored in the backup repository, for example to allow speed optimisations, or to support more kinds of file metadata. However, it would be silly to invalidate all existing backups people have made with Obnam (never mind that until version 1.0 Obnam did exactly that). Thus, Obnam attempts to support every format any released version has been able to create since version 1.0.

The tests in this chapter verify that each such repository format still works. The source tree contains a directory of archived backup repositories (in tar archives) and the tests will unpack those, and verify that they can be restored from correctly. The verification is done using a `summain` manifest for each generation, stored in the tar archive with the repository.

Each tar archive will contain a directory `repo`, which is the backup repository, and `manifest-1` and `manifest-2`, which are the manifests for the first and second generation.

Repository format 6 (Obnam version 1.0)

The repository format 6 is the one used for the 1.0 release of Obnam. We have two variants of reference repositories: a normal one, and one using the miserable `--small-files-in-btree` option. It's miserable, because it complicates the code but doesn't actually make anything better.

First, the normal one reference repository.

```
SCENARIO use repository format 6
ASSUMING extended attributes are allowed for users
GIVEN unpacked test data from test-data/repo-format-6-encrypted-deflated.tar.xz in T
WHEN user havelock restores generation 1 in T/repo to R1
THEN restored data in R1 matches T/manifest-1
WHEN user havelock restores generation 2 in T/repo to R2
THEN restored data in R2 matches T/manifest-2
```

Then, the in-tree repository.

```
SCENARIO use repository format 6 with in-tree data
ASSUMING extended attributes are allowed for users
GIVEN unpacked test data from test-data/repo-format-6-in-tree-data.tar.xz in T
WHEN user havelock restores generation 1 in T/repo to R1
THEN restored data in R1 matches T/manifest-1
WHEN user havelock restores generation 2 in T/repo to R2
THEN restored data in R2 matches T/manifest-2
```

Implementations

The following scenario steps are only ever used by scenarios in this chapter, so we implement them here.

First, we unpack the test data into a known location.

```
IMPLEMENTS GIVEN unpacked test data from (\S+) in (\S+)
mkdir "$DATADIR/$MATCH_2"
tar -C "$DATADIR/$MATCH_2" -xf "$MATCH_1"
```

Then we restore the requested generation. Note the use of the `--always-restore-setuid` option. Without it, the setuid/setgid bits get restored only if the tests are being run by the `root` user, or a user with the same uid as recorded in the reference repository. That would almost always break the test for other people, including CI.

```
IMPLEMENTS WHEN user (\S+) restores generation (\d+) in (\S+) to (\S+)
# Copy the keyrings from source tree so they don't get modified
# by this test.
cp -a "$SRCDIR/test-gpghome" "$DATADIR/.gnupg"
export GNUPGHOME="$DATADIR/.gnupg"
genid=$(run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" \
    --encrypt-with=3B1802F81B321347 genids | sed -n "${MATCH_2}p")
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" \
    --encrypt-with=3B1802F81B321347 \
    restore --to "$DATADIR/$MATCH_4" --generation "$genid" \
    --always-restore-setuid
```

Finally, we verify the restored data against the manifest. We have one tricky bit here: there is no guarantee what the path to the root of the live data is in the repository, but we search downwards until we find a directory with more than one child. That's what we match against the manifest.

```
IMPLEMENTS THEN restored data in (\S+) matches (\S+)
cd "$DATADIR/$MATCH_1"
while true
do
    case $(ls | wc -l) in
        1) cd * ;;
        0) echo "No children, oops" 1>&2; exit 1 ;;
        *) break ;;
    esac
done
summain -r --exclude=Ino --exclude=Dev --exclude=Uid \
    --exclude=Username --exclude=Gid --exclude=Group \
    --checksum=SHA1 \
    . | normalise_manifest_times > "$DATADIR/restored-manifest"
normalise_manifest_times "$DATADIR/$MATCH_2" \
    > "$DATADIR/original-manifest"
diff -u "$DATADIR/original-manifest" "$DATADIR/restored-manifest"
```

Chapter 15

kdirstat integration: producing kdirstat cache files

Obnam implements an ls variant whose output format is compatible with the kdirstat cache format.

Create a simple cache file

Here we create a simple backup and dump the output as a kdirstat cache file.

```
SCENARIO create and do a simple check of a kdirstat cache
GIVEN 10kB of new data in directory L
WHEN user U backs up directory L to repository R
AND user U creates a kdirstat cache file C of repository R
THEN first line of C matches [kdirstat 4.0 cache file]
AND for each entry in directory L a line in the kdircache C matches it
```

Now a backup containing some interesting objects

```
SCENARIO create and do a check of a kdirstat cache of interesting objects
ASSUMING extended attributes are allowed for users
GIVEN directory L with interesting filesystem objects
WHEN user U backs up directory L to repository R
AND user U creates a kdirstat cache file C of repository R
THEN first line of C matches [kdirstat 4.0 cache file]
AND for each entry in directory L a line in the kdircache C matches it
```

Validating the cache file

Sadly there seems to be no CLI usable program to read or validate the produced cache files. If there were we would do a more thorough test of the syntax here.

Chapter 16

Test implementation

This chapter documents the generic, shared IMPLEMENTS sections for steps that are used in a variety of scenarios. It also discusses the shell library that may be used by all IMPLEMENTS sections.

The shell library

The shell library contains shell functions and sets some shell variables that can be used by any IMPLEMENTS sections.

Variables:

- `REPO`: the pathname of the backup repository.

Functions:

- `run_obnam`: run Obnam from the source tree, ignoring any system-wide or user configuration and using only the configuration specified by the test suite itself (`--no-default-config`). Run in quiet mode (`--quiet`). The first argument to `run_obnam` is the client name.
- `manifest`: run summain in a way that produces a useable manifest, which can be compared with diff with a later manifest.

Live data generation

The simplest way to generate test live data is to just generate the necessary number of bytes, split over some number of files. We have the user name the directory explicitly, to avoid hidden dependencies between steps.

```
IMPLEMENTS GIVEN (\S+) of new data in directory (\S+)
genbackupdata --quiet --create "$MATCH_1" "$DATADIR/$MATCH_2"
```

Sometimes we need an amount of data in a specific file.

```
IMPLEMENTS GIVEN (\S+) of data in file (\S+)
"$SRCDIR/mkdata" --size "$MATCH_1" "$DATADIR/$MATCH_2"
```

We also need to generate a sparse file. A sparse file has at least one hole in it, and it may matter where the hole is: at the beginning, middle, or end of the file. Thus, we provide a way for scenarios to specify that.

```
IMPLEMENTS GIVEN a file (\S+) in (\S+), with (.+)
mkdir -p "$DATADIR/$MATCH_2"
"$SRCDIR/mksparse" "$DATADIR/$MATCH_2/$MATCH_1" "$MATCH_3"
```

Create some *interesting* data, using the `mkfunnyfarm` utility. See the utility for details, but this is about creating files and other filesystem objects that are not the most common easy cases for backup programs (regular files with data and no holes).

```
IMPLEMENTS GIVEN directory (\S+) with interesting filesystem objects
"$SRCDIR/mkfunnyfarm" "$DATADIR/$MATCH_1"
```

Some directories will be tagged as cache directories (see Cache directory tagging).

```
IMPLEMENTS GIVEN directory (\S+) is tagged as a cache directory
printf 'Signature: 8a477f597d28d172789f06886806bc55' \
    > "$DATADIR/$MATCH_1/CACHEDIR.TAG"
```

Sometimes it is necessary to set the modification filestamp of a file. Actually, it's usually useful to set both `st_mtime` and `st_atime` to the same value. The timestamp is given in the "seconds after epoch" in UTC format, as is common in Unix.

```
IMPLEMENTS GIVEN file (\S+) has Unix timestamp (-?\d+)
parent=$(dirname "$MATCH_1")
if [ ! -e "$DATADIR/$parent" ]
then
    mkdir "$DATADIR/$parent"
fi
touch "$DATADIR/$MATCH_1"
python -c '
import os
filename = os.path.join(
    os.environ["DATADIR"],
    os.environ["MATCH_1"])
timestamp = int(os.environ["MATCH_2"])
os.utime(filename, (timestamp, timestamp))
'
```

Create a file with given permissions.

```
IMPLEMENTS GIVEN file (\S+) with permissions (\S+)
touch "$DATADIR/$MATCH_1"
chmod "$MATCH_2" "$DATADIR/$MATCH_1"
```

Create a directory with given permissions.

```
IMPLEMENTS GIVEN directory (\S+) with permissions (\S+)
install -d -m "$MATCH_2" "$DATADIR/$MATCH_1"
```

We need to manipulate extended attributes.

```
IMPLEMENTS GIVEN file (\S+) has extended attribute (\S+) set to (\S+)
mkdir -p $(dirname "$DATADIR/$MATCH_1")
setfattr --name="$MATCH_2" --value "$MATCH_3" "$DATADIR/$MATCH_1"
```

Create a symlink.

```
IMPLEMENTS GIVEN a symlink (\S+) pointing at (\S+)
ln -s "$MATCH_2" "$DATADIR/$MATCH_1"
```

Sometimes we need to remove a file.

```
IMPLEMENTS WHEN user (\S+) removes file (\S+)
rm -f "$DATADIR/$MATCH_2"
```

Copy a file.

```
IMPLEMENTS GIVEN a copy of (.+) in (.+)
mkdir -p "$DATADIR/$(dirname "$MATCH_2")"
cp -a "$DATADIR/$MATCH_1" "$DATADIR/$MATCH_2"
```

Reset a repository's chunk files.

```
IMPLEMENTS WHEN repository (.+) resets its chunks to those in (.+)
r1="$DATADIR/$MATCH_1"
r2="$DATADIR/$MATCH_2"
if [ -e "$r1/chunks" ]
then
    # format 6
    rm -rf "$r1/chunks"
    cp -a "$r2/chunks" "$r1/."
else
    rm -rf "$r1/chunk-store"
    cp -a "$r2/chunk-store" "$r1/."
fi
```

Manifest creation and checking

We make it explicit in the scenario when a manifest is generated, so that naming of the manifest is explicit. This reduces the need to debug weird test suite bugs, when an automatic or implicit manifest naming goes wrong.

```
IMPLEMENTS GIVEN a manifest of (\S+) in (\S+)
manifest "$DATADIR/$MATCH_1" > "$DATADIR/$MATCH_2"
```

We need to check a directory tree against an existing manifest. We do this by generating a temporary manifest and diffing against that. We store the temporary manifest in a file so that if things fail, we can see what the temporary manifest actually contains. Being able to look at the actual file is easier than trying to interpret complicated diffs.

We remove the restore directory prefix from the manifest (the `Name:` field that Summain outputs). This is necessary so that comparisons with **diff**(1) will work well.

```
IMPLEMENTS THEN (\S+), restored to (\S+), matches manifest (\S+)
manifest "$DATADIR/$MATCH_2/$DATADIR/$MATCH_1" |
    sed "s*$DATADIR/$MATCH_2/**" > "$DATADIR/temp-manifest"

diff -u "$DATADIR/$MATCH_3" "$DATADIR/temp-manifest"
rm -f "$DATADIR/temp-manifest"
```

We may also need to check two manifests against each other.

```
IMPLEMENTS THEN manifests (\S+) and (\S+) match
diff -u "$DATADIR/$MATCH_1" "$DATADIR/$MATCH_2"
```

In a special case, we may need to update the `Mtime` for the first entry in a manifest. This is used, at least, when testing cache directory exclusion: we create wanted data, then create a manifest, then add the cache directory. This invalidates the Mtime for the first entry.

```
IMPLEMENTS GIVEN manifest (\S+) has Mtime for the first entry set from (\S+)
x=$(date -r "$DATADIR/$MATCH_2" '+Mtime: %Y-%m-%d %H:%M:%S +0000')
awk -v "x=$x" '
    !x && /^Mtime:/ { print x; next }
    /^$/ { x = 1 }
    { print }
    ' "$DATADIR/$MATCH_1" > "$DATADIR/new-manifest"
mv "$DATADIR/new-manifest" "$DATADIR/$MATCH_1"
```

Sometimes we create manifests with extra stuff. This allows us to remove them afterwards.

```
IMPLEMENTS GIVEN (\S+) is removed from manifest (\S+)
awk -v skip1="$MATCH_1" -v skip2="$MATCH_1/" '
```

```
    $1 == "Name:" &&
        ($2 == skip1 || substr($2, 1, length(skip2)) == skip2) {
            paragraph = ""; ignore = 1; next }
    NF > 0 && !ignore { paragraph = paragraph $0 "\n" }
    NF == 0 && paragraph { printf "%s\n", paragraph; paragraph = "" }
    NF == 0 { ignore = 0 }
    END { if (paragraph) printf "%s", paragraph }
' "$DATADIR/$MATCH_2" > "$DATADIR/$MATCH_2.new"
mv "$DATADIR/$MATCH_2.new" "$DATADIR/$MATCH_2"
```

Obnam configuration management

In some scenarios, it is easier to maintain a configuration file than to pass in all the options to `run_obnam` every time. This section contains steps to do that.

Scenarios involving encryption need to specify the encryption key to use. We store that.

```
IMPLEMENTS GIVEN user (\S+) uses encryption key "(.*)" from (\S+)
if [ ! -e "$DATADIR/$MATCH_1.gnupg" ]
then
    mkdir "$DATADIR/$MATCH_1.gnupg"
    cp -a "$SRCDIR/$MATCH_3/." "$DATADIR/$MATCH_1.gnupg/."
    chmod -R 0700 "$DATADIR/$MATCH_1.gnupg"
    add_to_env "$MATCH_1" GNUPGHOME "$DATADIR/$MATCH_1.gnupg"
else
    # Export public and secret keys from new keyring.
    export GNUPGHOME="$SRCDIR/$MATCH_3"
    gpg --export "$MATCH_2" > "$DATADIR/public.key"
    gpg --export-secret-keys "$MATCH_2" > "$DATADIR/secret.key"

    # Import into the keyring uses for tests.
    export GNUPGHOME="$DATADIR/$MATCH_1.gnupg"
    gpg --import "$DATADIR/public.key"
    gpg --import "$DATADIR/secret.key"
fi

add_to_config "$MATCH_1" encrypt-with "$MATCH_2"
```

Scenarios involving encryption may also use a private keyring directory.

```
IMPLEMENTS GIVEN user (\S+) separately uses encryption key "(.*)" from (\S+)
if [ ! -e "$DATADIR/$MATCH_1.gnupg" ]
then
    mkdir "$DATADIR/$MATCH_1.gnupg"
    cp -a "$SRCDIR/$MATCH_3/." "$DATADIR/$MATCH_1.gnupg/."
```

```
    add_to_config "$MATCH_1" gnupghome "$DATADIR/$MATCH_1.gnupg"
else
    # Export public and secret keys from new keyring.
    export GNUPGHOME="$SRCDIR/$MATCH_3"
    gpg --export "$MATCH_2" > "$DATADIR/public.key"
    gpg --export-secret-keys "$MATCH_2" > "$DATADIR/secret.key"

    # Import into the keyring uses for tests.
    export GNUPGHOME="$DATADIR/$MATCH_1.gnupg"
    gpg --import "$DATADIR/public.key"
    gpg --import "$DATADIR/secret.key"

    # Use the configuration rather than the environment.
    add_to_config "$MATCH_1" gnupghome "$GNUPGHOME"
    unset GNUPGHOME
fi

add_to_config "$MATCH_1" encrypt-with "$MATCH_2"
```

Encryption scenarios, at least, also need users that pretend to be someone else.

```
IMPLEMENTS GIVEN a user (\S+) calling themselves (\S+)
add_to_config "$MATCH_1" client-name "$MATCH_2"
```

Add a setting to a client's configuration file.

```
IMPLEMENTS GIVEN user (\S+) sets configuration (\S+) to (.*)
add_to_config "$MATCH_1" "$MATCH_2" "$MATCH_3"
```

Backing up

The simplest way to run a backup, for single-client scenarios. In addition to backing up, this makes a manifest of the data.

```
IMPLEMENTS WHEN user (\S+) backs up directory (\S+) to repository (\S+)
run_obnam "$MATCH_1" backup -r "$DATADIR/$MATCH_3" "$DATADIR/$MATCH_2"
```

A test may make a backup fail. Run without failing the test so the failure may be inspected.

```
IMPLEMENTS WHEN user (\S+) attempts to back up directory (\S+) to repository (\S+)
attempt run_obnam "$MATCH_1" \
    backup -r "$DATADIR/$MATCH_3" "$DATADIR/$MATCH_2"
```

We may also need to backup two directories at once.

```
IMPLEMENTS WHEN user (\S+) backs up directories (\S+) and (\S+) to repository (\S+)
run_obnam "$MATCH_1" backup -r "$DATADIR/$MATCH_4" \
```

```
    "$DATADIR/$MATCH_2" "$DATADIR/$MATCH_3"
```

We can also just pretend to make a backup.

```
IMPLEMENTS WHEN user (\S+) pretends to back up directory (\S+) to repository (\S+)
run_obnam "$MATCH_1" backup --pretend -r "$DATADIR/$MATCH_3" "$DATADIR/$MATCH_2"
```

fsck'ing a repository

Verify that the repository itself is OK, by running `obnam fsck` on it.

```
IMPLEMENTS THEN user (\S+) can fsck the repository (\S+)
run_obnam "$MATCH_1" fsck -r "$DATADIR/$MATCH_2"
```

Restoring data

We need a way to restore data from a test backup repository.

```
IMPLEMENTS WHEN user (\S+) restores their latest generation in repository (\S+) into (\S+)
run_obnam "$MATCH_1" restore -r "$DATADIR/$MATCH_2" \
    --to "$DATADIR/$MATCH_3"
```

Restore a specific generation. The generation number is an ordinal in the list of generations, not the "generation id" Obnam assigns, as that is unpredictable.

```
IMPLEMENTS WHEN user (\S+) restores generation (\d+) to (\S+) from repository (\S+)
client="$MATCH_1"
gen="$MATCH_2"
to="$DATADIR/$MATCH_3"
repo="$DATADIR/$MATCH_4"
id=$(run_obnam "$client" -r "$repo" genids |
    awk -v "n=$gen" 'NR == n')
run_obnam "$client" restore -r "$repo" \
    --to "$to" --generation "$id"
```

We may also need to attempt a restore in a situation when we expect it to fail.

```
IMPLEMENTS WHEN user (\S+) attempts to restore their latest generation in repository (\S+) into (
attempt run_obnam "$MATCH_1" restore -r "$DATADIR/$MATCH_2" \
    --to "$DATADIR/$MATCH_3"
```

We may need to restore only a single file.

```
IMPLEMENTS WHEN user (\S+) restores file (\S+) to (\S+) from their latest generation in repositor
run_obnam "$MATCH_1" ls -r "$DATADIR/$MATCH_4"
run_obnam "$MATCH_1" restore -r "$DATADIR/$MATCH_4" \
    --to "$DATADIR/$MATCH_3" "$DATADIR/$MATCH_2"
```

Verifying live data

Run `obnam verify` and remember the output.

```
IMPLEMENTS WHEN user (\S+) attempts to verify (\S+) against repository (\S+)
attempt run_obnam "$MATCH_1" \
    verify -r "$DATADIR/$MATCH_3" "$DATADIR/$MATCH_2"
```

Verify a random file in live data.

```
IMPLEMENTS WHEN user (\S+) attempts to verify a random file in (\S+) against repository
attempt run_obnam "$MATCH_1" \
    verify -r "$DATADIR/$MATCH_3" --verify-randomly=1 "$DATADIR/$MATCH_2"
```

Removing (forgetting) generations

Run `obnam forget` with neither a policy of what to keep, nor a specific list of generations.

```
IMPLEMENTS WHEN user (\S+) runs obnam forget without generations or keep policy on repo
run_obnam "$MATCH_1" forget -r "$DATADIR/$MATCH_2"
```

Remove the oldest generation.

```
IMPLEMENTS WHEN user (\S+) forgets the oldest generation in repository (\S+)
# The grep below at the end of pipeline is there to make sure
# the pipeline fails if there were no generations.
id=$(run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_2" genids |
    head -n1 | grep .)
run_obnam "$MATCH_1" forget -r "$DATADIR/$MATCH_2" "$id"
```

Remove the newest generation.

```
IMPLEMENTS WHEN user (\S+) forgets their latest generation in repository (\S+)
# The grep below at the end of pipeline is there to make sure
# the pipeline fails if there were no generations.
id=$(run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_2" genids |
    tail -n1 | grep .)
run_obnam "$MATCH_1" forget -r "$DATADIR/$MATCH_2" "$id"
```

Remove according to a `--keep` schedule.

```
IMPLEMENTS WHEN user (\S+) forgets according to schedule (\S+) in repository (\S+)
run_obnam "$MATCH_1" forget -r "$DATADIR/$MATCH_3" --keep "$MATCH_2"
```

Pretend to forget.

```
IMPLEMENTS WHEN user (\S+) pretends to forget according to schedule (\S+) in repository
run_obnam "$MATCH_1" \
    forget --pretend -r "$DATADIR/$MATCH_3" --keep "$MATCH_2"
```

List generations

List everything in a generation. Capture the listing in a named file.

```
IMPLEMENTS WHEN user (\S+) lists latest generation in repository (\S+) into (\S+)
run_obnam "$MATCH_1" ls -r "$DATADIR/$MATCH_2" > "$DATADIR/$MATCH_3"
```

List only parts of a generation. Again, capture in a named file.

```
IMPLEMENTS WHEN user (\S+) lists (\S+) in latest generation in repository (\S+) into (\S+)
run_obnam "$MATCH_1" ls -r "$DATADIR/$MATCH_3" "$DATADIR/$MATCH_2" > "$DATADIR/$MATCH_4"
```

Checks on generations

Check that number of generations is correct.

```
IMPLEMENTS THEN user (\S+) sees (\d+) generation(s?) in repository (\S+)
run_obnam "$MATCH_1" generations -r "$DATADIR/$MATCH_4" \
    > "$DATADIR/generation.list"
n=$(wc -l < "$DATADIR/generation.list")
test "$MATCH_2" = "$n"
```

Ditto for generation ids.

```
IMPLEMENTS THEN user (\S+) sees (\d+) generation ids in repository (\S+)
run_obnam "$MATCH_1" generations -r "$DATADIR/$MATCH_3" \
    > "$DATADIR/generation-id.list"
n=$(wc -l < "$DATADIR/generation-id.list")
test "$MATCH_2" = "$n"
```

Check that there are no checkpoint generations.

```
IMPLEMENTS THEN user (\S+) sees no checkpoint generations in repository (\S+)
run_obnam "$MATCH_1" generations -r "$DATADIR/$MATCH_2" \
    > "$DATADIR/generation.list"
if grep checkpoint "$DATADIR/generation.list"
then
    echo "Can see checkpoint generations!" 1>&2
    exit 1
fi
```

Check timestamps of specific generations.

```
IMPLEMENTS THEN user (\S+) has (\d+)(st|nd|rd|th) generation timestamp (.*) in repository (\S+)
run_obnam "$MATCH_1" generations -r "$DATADIR/$MATCH_5" |
sed -n "${MATCH_2}p" |
awk -v "T=$MATCH_4" '
    ($2 " " $3) != T {
        print "Fail to match: " $0
```

```
        exit 1
    }
'
```

Diffs between generations

Compute the difference between two generations. The generations are identified by the ordinal, not generation id, since the ids are unpredictable.

```
IMPLEMENTS WHEN user (\S+) diffs generations (\d+) and (\d+) in repository (\S+) into
id1=$(run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_4" genids | awk -v "n=$MATCH_2" 'NR == r
id2=$(run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_4" genids | awk -v "n=$MATCH_3" 'NR == r
run_obnam "$MATCH_1" diff -r "$DATADIR/$MATCH_4" "$id1" "$id2" > "$DATADIR/$MATCH_5"
```

Show the diff between the latest generation and the generation before that.

```
IMPLEMENTS WHEN user (\S+) diffs latest generation in repository (\S+) into (\S+)
run_obnam "$MATCH_1" diff -r "$DATADIR/$MATCH_2" latest > "$DATADIR/$MATCH_3"
```

Encryption key management

List clients and the encryption keys they use.

```
IMPLEMENTS THEN user (\S+) uses key "(.+)" in repository (\S+)
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" client-keys |
    grep -x "$MATCH_1 $MATCH_2"
```

Import a key into one user's keyring from another keyring.

```
IMPLEMENTS WHEN user (\S+) imports public key "(.+)" from (\S+)
GNUPGHOME="$SRCDIR/$MATCH_3" gpg --export --armor "$MATCH_2" |
GNUPGHOME="$DATADIR/$MATCH_1.gnupg" gpg --import
```

Add a public key to a repository, but not to the calling user, only to the shared parts.

```
IMPLEMENTS WHEN user (\S+) adds key "(.+)" to repository (\S+) only
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" \
    add-key --keyid "$MATCH_2"
```

Add a public key to a repository, and the calling user.

```
IMPLEMENTS WHEN user (\S+) adds key "(.+)" to repository (\S+) and self
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" \
    add-key --keyid "$MATCH_2" "$MATCH_1"
```

Remove a public key from a repository.

```
IMPLEMENTS WHEN user (\S+) removes key "(.+)" from repository (\S+)
export GNUPGHOME="$DATADIR/$MATCH_1.gnupg"
keyid="$(
    get_fingerprint "$MATCH_2" |
    awk '{ print substr($0, length-16) }')"
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" \
    remove-key --keyid "$keyid"
```

Forget a key from the user's keyring.

```
IMPLEMENTS WHEN user (\S+) no longer has key "(.+)"
export GNUPGHOME="$DATADIR/$MATCH_1.gnupg"
ls -la "$GNUPGHOME"
echo fingerprints
gpg --fingerprint "$MATCH_2"
echo with colons
gpg --fingerprint --with-colons "$MATCH_2"
echo just fingeprint
gpg --fingerprint --with-colons "$MATCH_2" |
    awk -F: '{ print } /^fpr:/ { print $10; exit }'
fingerprint="$(get_fingerprint "$MATCH_2")"
gpg --batch --yes --delete-secret-key "$fingerprint"
```

Lock management

We need to lock parts of the repository, and force those locks open.

```
IMPLEMENTS WHEN user (\S+) locks repository (\S+)
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_2" _lock
```

Force it open.

```
IMPLEMENTS WHEN user (\S+) forces open the lock on repository (\S+)
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_2" force-lock
```

Client management

Sometimes, even if rarely, one wants to remove a complete client from a repository.

```
IMPLEMENTS WHEN user (\S+) removes user (\S+) from repository (\S+)
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" remove-client "$MATCH_2"
```

After that, one wants to make sure the removed client isn't in the repository anymore.

```
IMPLEMENTS THEN user (\S+) can't see user (\S+) in repository (\S+)
```

```
if run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" clients |
    grep -x "$MATCH_2"
then
    echo "Client $MATCH_2 still in repository $MATCH_3" 1>&2
    exit 1
fi
```

We may also want to make sure we do see a client.

```
IMPLEMENTS THEN user (\S+) can see user (\S+) in repository (\S+)
if ! run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" clients |
    grep -x "$MATCH_2"
then
    echo "Client $MATCH_2 is not in repository $MATCH_3" 1>&2
    exit 1
fi
```

Checks on results of an attempted operation

The `attempt` shell function in `obnam.sh` runs a command, then captures its stdout, stderr, and exit code. The scenarios can then test the results in various ways as separate steps.

```
IMPLEMENTS THEN the attempt succeeded
attempt_exit_was 0

IMPLEMENTS THEN the attempt failed with exit code (\d+)
attempt_exit_was "$MATCH_1"
```

We want to match the stdout against a regular expression.

```
IMPLEMENTS THEN the output matches "(.+)"
echo -----------------------
echo stdout being matched:
cat "$DATADIR/attempt.stdout"
echo -----------------------
attempt_matches stdout "$MATCH_1"
```

We can also match the stderr against a regular expression.

```
IMPLEMENTS THEN the error message matches "(.+)"
echo -----------------------
echo stderr being matched:
cat "$DATADIR/attempt.stderr"
echo -----------------------
attempt_matches stderr "$MATCH_1"
```

Checks on files

Is a file empty?

```
IMPLEMENTS THEN file (\S+) is empty
diff -u /dev/null "$DATADIR/$MATCH_1"
```

Is a file not empty?

```
IMPLEMENTS THEN file (\S+) is not empty
! diff -u /dev/null "$DATADIR/$MATCH_1"
```

Check that a restored file uses at most as much disk space as the original one in live data.

```
IMPLEMENTS THEN file (\S+) from (\S+), restored in (\S+) doesn't use more disk
old=$(stat -c %b "$DATADIR/$MATCH_2/$MATCH_1")
new=$(stat -c %b "$DATADIR/$MATCH_3/$DATADIR/$MATCH_2/$MATCH_1")
test "$old" -lt "$new"
```

Check that a restored directory is empty.

```
IMPLEMENTS THEN (\S+), restored to (\S+), is empty
if find "$DATADIR/$MATCH_2/$DATADIR/$MATCH_1" -mindepth 1 | grep .
then
    die "$DATADIR/$MATCH_2/$DATADIR/$MATCH_1 isn't empty"
fi
```

Checks on contents of files

Regular expressions are very powerful, and sometimes that power is warranted to use. It isn't always clear enough to the lay reader, so be careful. `grep -E` regular expressions are used here.

Does any line match?

```
IMPLEMENTS THEN (\S+) matches (.*)$
grep -E -e "$MATCH_2" -- "$DATADIR/$MATCH_1"
```

Does first line match?

```
IMPLEMENTS THEN first line of (\S+) matches (.*)$
head -n1 "$DATADIR/$MATCH_1" | grep -E -e "$MATCH_2" --
```

Do all lines match?

```
IMPLEMENTS THEN all lines in (\S+) match (\S+)
! grep -E -v -e "$MATCH_2" -- "$DATADIR/$MATCH_1"
```

Does no line match?

```
IMPLEMENTS THEN nothing in (\S+) matches (\S+)
if grep -E -e "$MATCH_2" -- "$DATADIR/$MATCH_1" | grep '.*'
then
    echo "At least one line matches, when none may!" 1>&2
    exit 1
fi
```

Merely read a file. This checks that the file exists and can be read by the user.

```
IMPLEMENTS WHEN user (\S+) reads file (\S+)
cat "$DATADIR/$MATCH_2"
```

Does a restored file match what's in live data?

```
IMPLEMENTS THEN file (.+), restored to (.+), matches live data
cmp "$DATADIR/$MATCH_1" "$DATADIR/$MATCH_2/$DATADIR/$MATCH_1"
```

Check on user running test suite

Some tests won't work correctly when `root` is running them.

```
IMPLEMENTS ASSUMING not running as root
test "$(id -u)" != 0
```

For testing FUSE stuff, we need to check that the relevant stuff is available. Previously, we did this by checking that the user running the test suite was in the group `fuse`, but that turns out to have been a Debianism (removed in the Debian `jessie` release). A better check is to check that the `fusermount` command is available.

However, if we're on Debian and on a version prior to 8.0, we need to also be in the `fuse` group. (This can be dropped after support for those versions of Debian is dropped from Obnam, probably around first or second quarter of 2015.)

```
IMPLEMENTS ASSUMING user can use FUSE

# We _must_ have fusermount in any case.
if ! command -v fusermount
then
    echo "No fusermount found. User cannot use FUSE without it." 1>&2
    exit 1
fi
echo "fusermount found"

# We also must be able to read from /dev/fuse. It might not exist,
# and if it does, the kernel module providing it might not be
# loaded. So we read 0 bytes from it, and if that works, it should
# be OK.
```

```
if ! dd if=/dev/fuse of=/dev/null bs=1 count=0
then
    echo "Can't read from /dev/null. User can't use FUSE." 1>&2
    exit 1
fi
echo "Can read from /dev/fuse."

# Are we on Debian? If so, /etc/debian_version exists.
# If it doesn't, we're done.
if [ ! -e /etc/debian_version ]
then
    echo "We are not on Debian. User can use FUSE."
    exit 0
fi

# Read /etc/debian_version, and interpret it as a floating point
# number, and compare it to 8.0. Prior to 8.0, we need to be in
# the fuse group as well.
if awk '($0 + 0.0) < 8.0 { exit 0 } END { exit 1 }' /etc/debian_version
then
    echo "We're on Debian prior to 8.0."
    if groups | tr ' ' '\n' | grep -Fx fuse
    then
        echo "User is in group fuse."
    else
        echo "User is NOT in group fuse. User can't use FUSE."
        exit 1
    fi
else
    echo "We're on Debian 8.0 or later."
fi

# We're good.
echo "User can use FUSE."
exit 0
```

Check on whether user extended attributes work

Extended attributes are, at least on some filesystems, a mount-time option which may be disabled. In fact, experience has shown that it often is disabled on build servers.

```
IMPLEMENTS ASSUMING extended attributes are allowed for users
touch "$DATADIR/xattr.test"
```

```
setfattr -n user.foo -v bar "$DATADIR/xattr.test"
```

Nagios

Run the Nagios monitoring subcommand.

```
IMPLEMENTS WHEN user (\S+) attempts nagios-last-backup-age against repository (\S+)
attempt run_obnam "$MATCH_1" nagios-last-backup-age \
    -r "$DATADIR/$MATCH_2"
```

kdirstat

Create a kdirstat cache.

```
IMPLEMENTS WHEN user (\S+) creates a kdirstat cache file (\S+) of repository (\S+)
run_obnam "$MATCH_1" -r "$DATADIR/$MATCH_3" kdirstat > "$DATADIR/$MATCH_2"
```

Check that the cache mentions each file in the repository. Since grep -E cannot specifically match a tab we check only for one character of whitespace.

```
IMPLEMENTS THEN for each entry in directory (\S+) a line in the kdircache (\S+) matches
find "$DATADIR/$MATCH_1" -type f | while read f ; do \
    grep -E -e "^F[[:space:]]$f" "$DATADIR/$MATCH_2" || exit 1; \
done
find "$DATADIR/$MATCH_1" -type p | while read p ; do \
    grep -E -e "^FIFO[[:space:]]$p" "$DATADIR/$MATCH_2" || exit 1; \
done
```