NAME

Lintian::Tutorial::WritingChecks −− Writing checks for Lintian

SYNOPSIS

Warning: This tutorial may be outdated.

This guide will quickly guide you through the basics of writing a Lintian check. Most of the work is in writing the two files:

```
checks/<my-check>.pm
checks/<my-check>.desc
```

And then either adding a Lintian profile or extending an existing one.

DESCRIPTION

The basics of writing a check are outlined in the Lintian User Manual (X3.3). This tutorial will focus on the act of writing the actual check. In this tutorial, we will assume the name of the check to be written is ''deb/pkg−check''.

The tutorial will work with a ''binary'' and ''udeb'' check. Checking source packages works in a similar fashion.

Create a check *.desc* file

As mentioned, this tutorial will focus on the writing of a check. Please see the Lintian User Manual (X3.3) for how to do this part.

Create the Perl check module

Start with the template:

```
# deb/pkg-check is loaded as Lintian::deb::pkg_check
# - See Lintian User Manual X3.3 for more info
package Lintian::deb::pkg_check;

use strict;
use warnings;

sub run {
    my ($pkg, $type, $info, $proc, $group) = @_;
    return;
}
```

The snippet above is a simple valid check that does ''nothing at all''. We will extend it in just a moment, but first let us have a look at the arguments at the setup.

The *run* sub is the entry point of our ''deb/pkg−check'' check; it will be invoked once per package it should process. In our case, that will be once per ''binary'' (.deb) and once per udeb package processed.

It is given 5 arguments (in the future, possibly more), which are:

`$pkg` − The name of the package being processed.
(Same as `$proc`−>pkg_name)

`$type` − The type of the package being processed.
At the moment, `$type` is one of ''binary'' (.deb), ''udeb'', ''source'' (.dsc) or ''changes''. This argument is mostly useful if certain checks do not apply equally to all package types being processed.

Generally it is advisable to check only binaries (''binary'' and ''udeb''), sources or changes in a given check. But in rare cases, it makes sense to lump multiple types together in the same check and this argument helps you do that.

(Current it is always identical to `$proc`−>pkg_type)

`$info` – Accessor to the data Lintian has extracted
: Basically all information you want about a given package comes from the `$info` object. Sometimes referred to as either the ''info object'' or (an instance of) Lintian::Collect.

  This object (together with a properly set Needs-Info in the *.desc* file) will grant you access to all of the data Lintian has extracted about this package.

  Based on the value of the `$type` argument, it will be one of Lintian::Collect::Binary, Lintian::Collect::Changes or Lintian::Collect::Source.

  (Currently it is the same as `$proc->`info)

`$proc` – Basic metadata about the package
: This is an instance of Lintian::Processable and is useful for trivially obtaining very basic package metadata. Particularly, the name of source package and version of source package are readily available through this object.

`$group` – Group of processables from the same source
: If you want to do a cross-check between different packages built from the same source, `$group` helps you access those other packages (if they are available).

  This is an instance of Lintian::ProcessableGroup.

Now back to the coding.

Accessing fields

Let's do a slightly harder example. Assume we wanted to emit a tag for all packages without a (valid) Multi-Arch field. This requires us to A) identify if the package has a Multi-Arch field and B) identify if the content of the field was valid.

Starting from the top. All `$info` objects have a method called field, which gives you access to a (raw) field from the control file of the package. It returns `undef` if said field is not present or the content of said field otherwise. Note that field names must be given in all lowercase letters (i.e. use ''multi-arch'', not ''Multi-Arch'').

This was the first half. Let's look at checking the value. Multi-arch fields can (currently) be one of ''no'', ''same'', ''foreign'' or ''allowed''. One way of checking this would be using the regex:

Notice that Lintian automatically strips leading and trailing spaces on the *first* line in a field. It also strips trailing spaces from all other lines, but leading spaces and the '' .''–continuation markers are kept as is.

Checking dependencies

Lintian can do some checking of dependencies. For most cases it works similar to a normal dependency check, but keep in mind that Lintian uses *pure* logic to determine if dependencies are satisfied (i.e. it will not look up relations like Provides for you).

Suppose you wanted all packages with a multi-arch ''same'' field to pre-depend on the package ''multiarch-support''. Well, we could use the `$info->`relation method for this.

`$info->`relation returns an instance of Lintian::Relation. This object has an ''implies'' method that can be used to check if a package has an explicit dependency. Note that ''implies'' actually checks if one relation ''implies'' another (i.e. if you satisfied relationA then you definitely also satisfied relationB).

As with the ''field''–method, field names have to be given in all lowercase. However ''relation'' will never return `undef` (not even if the field is missing).

Using static data files

Currently our check mixes data and code. Namely all the valid values for the Multi-Arch field are currently hard-coded in our check. We can move those out of the check by using a data file.

Lintian natively supports data files that are either ''sets'' or ''tables'' via Lintian::Data (i.e. ''unordered'' collections). As an added bonus, Lintian::Data transparently supports vendor specific data files for us.

First we need to make a data file containing the values. Which could be:

```
# A table of all the valid values for the multi-arch field.
no
same
foreign
allowed
```

This can then be stored in the data directory as *data/deb/pkg-check/multiarch-values*.

Now we can load it by using:

```
use Lintian::Data;

my $VALID_MULTI_ARCH_VALUES =
    Lintian::Data->new('deb/pkg-check/multiarch-values');
```

Actually, this is not quite true. Lintian::Data is lazy, so it will not load anything before we force it to do so. Most of the time this is just an added bonus. However, if you ever have to force it to load something immediately, you can do so by invoking its ''known'' method (with an arbitrary defined string and ignore the result).

Data files work with 3 access methods, ''all'', ''known'' and ''value''.

all ''all'' (i.e. `$data->all`) returns a list of all the entries in the data file (for key/value tables, all returns the keys). The list is not sorted in any order (not even input order).

known
''known'' (i.e. `$data->known('item')`) returns a truth value if a given item or key is known (present) in the data set or table. For key/pair tables, the value associated with the key can be retrieved with ''value'' (see below).

value
''value'' (i.e. `$data->value('key')`) returns a value associated with a key for key/value tables. For unknown keys, it returns `undef`. If the data file is not a key/value table but just a set, value returns a truth value for known keys.

While we could use both ''value'' and ''known'', we will use the latter for readability (and to remind ourselves that this is a data set and not a data table).

Basically we will be replacing:

```
unless exists $VALID_MULTI_ARCH_VALUES{$multiarch};
```

with

```
unless $VALID_MULTI_ARCH_VALUES->known($multiarch);
```

Accessing contents of the package

Another heavily used mechanism is to check for the presence (or absence) of a given file. Generally this is what the `$info->index` and `$info->sorted_index` methods are for. The ''index'' method returns instances of Lintian::Path, which has a number of utility methods.

If you want to loop over all files in a package, the sorted_index will do this for you. If you are looking for a specific file (or directory), a call to ''index'' will be much faster. For the contents of a specific directory, you can use something like:

```
if (my $dir = $info->index('path/to/dir/')) {
    foreach my $elem ($dir->children) {
        print $elem->name . " is a file" if $elem->is_file;
        # ...
    }
}
```

Keep in mind that using the ''index'' or ''sorted_index'' method will require that you put ''unpacked'' in Needs-Info. See ''Keeping Needs-Info up to date''.

There are also a pair of methods for accessing the control files of a binary package. These are `$info->control_index` and `$info->sorted_control_index`.

Accessing contents of a file in a package

When you actually want to see the contents of a file, you can use open (or open_gz) on an object returned by e.g. `$info->index`. These methods will open the underlying file for reading (the latter applying a gzip decompression).

However, please do assert that the file is safe to read by calling is_open_ok first. Generally, it will only be true for files or safely resolvable symlinks pointing to files. Should you attempt to open a path that does not satisfy those criteria, Lintian::Path will raise a trappable error at runtime.

Alternatively, if you access the underlying file object, you can use the fs_path method. Usually, you will want to test either is_open_ok or is_valid_path first to ensure you do not follow unsafe symlinks. The "is_open_ok" check will also assert that it is not (e.g.) a named pipe or such.

Should you call fs_path on a symlink that escapes the package root, the method will throw a trappable error at runtime. Once the path is returned, there are no more built-in fail-safes. When you use the returned path, keep things like "../../../../../etc/passwd"-symlink and "fifo" pipes in mind.

In some cases, you may even need to access the file system objects *without* using Lintian::Path. This is, of course, discouraged and suffers from the same issues above (all checking must be done manually by you). Here you have to use the "unpacked", "debfiles" or "control" methods from Lintian::Collect or its subclasses.

The following snippet may be useful for testing that a given path does not escape the root.

```
use Lintian::Util qw(is_ancestor_of);

my $path = ...;
# The snippet applies equally well to $info->debfiles and
# $info->control (just remember to subst all occurrences of
# $info->unpacked).
my $unpacked_file = $info->unpacked($path);
if ( -f $unpacked_file && is_ancestor_of($info->unpacked, $unpacked_file)) {
   # a file and contained within the package root.
} else {
   # not a file or an unsafe path
}
```

Keeping Needs-Info up to date

Keeping the "Needs-Info" field of your *.desc* file is a bit of manual work. In the API description for the method there will generally be a line looking something like:

```
Needs-Info requirements for using methodx: Y
```

Which means that the methodx requires Y to work. Here Y is a comma separated list and each element of Y basically falls into 3 cases.

- The element is the word *none*

  In this case, the method has no "external" requirements and can be used without any changes to your Needs-Info. The "field" method is an example of this.

  This only makes sense if it is the only element in the list.

- The element is a link to a method

  In this case, the method uses another method to do its job. An example is the sorted_control_index method, which uses the control_index method. So using *sorted_control_index* has the same requirements as using *control_index*.

- The element is the name of a collection (e.g. ‘‘control_index’’).

  In this case, the method needs the given collection to be run. So to use (e.g.) control_index, you have to put ‘‘bin-pkg-control’’ in your Needs-Info.

CAVEAT: Methods can have different requirements based on the type of package! An example of this ‘‘changelog’’, which requires ‘‘changelog-file’’ in binary packages and ‘‘Same as debfiles’’ in source packages.

Avoiding security issues

Over the years a couple of security issues have been discovered in Lintian. The problem is that people can in theory create some really nasty packages. Please keep the following in mind when writing a check:

- Avoid 2−arg open, system/exec($shellcmd), ‘$shellcmd‘ like the plague.

  When you get any one of those wrong you introduce ‘‘arbitrary code execution’’ vulnerabilities (we learned this the hard way via CVE−2009−4014).

  Usually 3−arg open and the non-shell variant of system/exec are enough. When you actually need a shell pipeline, consider using Lintian::Command. It also provides a *safe_qx* command to assist with capturing stdout as an alternative to ‘$cmd‘ (or qx/$cmd/).

- Do not trust field values.

  This is especially true if you intend to use the value as part of a file name. Verify that the field contains what you expect before you use it.

- Use Lintian::Path (or, failing that, is_ancestor_of)

  You might be tempted to think that the following code is safe:

```
use autodie;

my $filename = 'some/file';
my $ufile = $info->unpacked($filename);
if ( ! -l $ufile) {
   # Looks safe, but isn't in general
   open(my $fd, '<', $ufile);
   ...;
}
```

  This is definitely unsafe if ‘‘$filename’’ contains at least one directory segment. So, if in doubt, use is_ancestor_of to verify that the requested file is indeed the file you think it is. A better version of the above would be:

```
use autodie,
use Lintian::Util qw(is_ancestor_of);
[...]
my $filename = 'some/file';
my $ufile = $info->unpacked($filename);
if ( ! -l $ufile && -f $ufile && is_ancestor_of($info->unpacked, $ufile)) {
   # $ufile is a file and it is contained within the package root.
   open(m $fd, '<', $ufile);
   ...;
}
```

  In some cases you can even drop the ‘‘! −l `$ufile`’’ part.

  Of course, it is much easier to use the Lintian::Path object (whenever possible).

```
my $filename = 'some/file';
my $ufile = $info->index($filename);
if ( $ufile && $ufile->is_file && $ufile->is_open_ok) {
    my $fd = $ufile->open;
    ...;
}
```

Here you can drop the " && `$ufile->`is_file" if you want to permit safe symlinks.

For more information on the is_ancestor_of check, see is_ancestor_of

SEE ALSO

Lintian::Tutorial::WritingTests, Lintian::Tutorial::TestSuite