

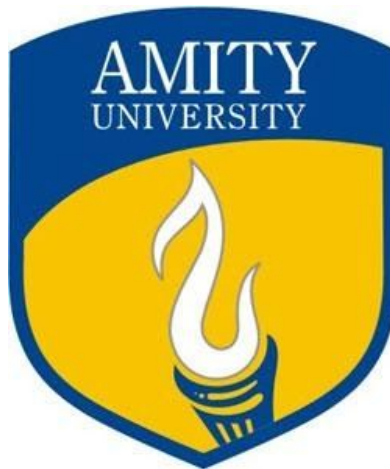
MINOR PROJECT

on

Linter for Enforcing Downstream Checks

Submitted to

Amity University Uttar Pradesh



In partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology in Computer Science and Engineering

By

Utkarsh Gupta, Samyak Jain

Enrolment No: A2305217557, A2305217638

Under the guidance of

Dr. Pooja Singh

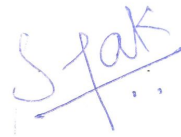
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY

AMITY UNIVERSITY, UTTAR PRADESH

DECLARATION

We, **Utkarsh Gupta** and **Samyak Jain**, students of B. Tech. (CSE) hereby declare that the project titled “**Linter for Enforcing Downstream Checks**”, which is submitted by us to the Department of Computer Science and Engineering, Amity School of Engineering and Technology, Amity University Uttar Pradesh in partial fulfillment of the requirement for the award of the degree of Bachelor of Technology in Computer Science and Engineering has not been previously formed the basis for the award of my degree, diploma or other similar title or recognition.



Date: 29th October 2020

Samyak Jain

Utkarsh Gupta

CERTIFICATE

On the basis of a declaration submitted by **Utkarsh Gupta** and **Samyak Jain**, students of B.Tech. CSE, We hereby certify that the project titled “**Linter for Enforcing Downstream Checks**” which is submitted to the Department of Computer Science and Engineering, Amity School of Engineering and Technology, Amity University, Uttar Pradesh in partial fulfillment of the requirement for the award of the degree of Bachelor of Technology in Computer Science and Engineering and as part of Industrial Training project is an original contribution with existing knowledge and faithful record of work carried out by her under my guidance and supervision.

Date: 29th October 2020

Dr. Pooja Singh

Department of Computer Science & Engineering

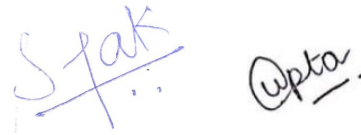
Amity School of Engineering and Technology

Amity University, Uttar Pradesh

ACKNOWLEDGEMENT

We, **Utkarsh Gupta** and **Samyak Jain** would like to express profound gratitude towards Dr. Sangeeta Rani for her undying guidance, continuous support, and cooperation through the course of my term paper titled “**Linter for Enforcing Downstream Checks**” without which this task would’ve been difficult to complete. My endeavor stands incomplete without dedicating my gratitude to her; as she has guided me a lot towards the successful completion of my project.

I would also like to express my gratitude to my family, friends for their support, and the tireless effort that kept me motivated throughout the completion of this project.

The image shows two handwritten signatures in blue ink. The first signature, on the left, is 'Syak' with a horizontal line underneath. The second signature, on the right, is 'Gupta' with a horizontal line underneath.

Date: 29th October 2020

Samyak Jain Utkarsh Gupta

Abstract

RuboCop::Packaging is an extension of [RuboCop](#), which is a Ruby static code analyzer (a.k.a. linter) and code formatter.

It helps to enforce some of the guidelines that are expected of upstream maintainers so that the downstream can build their packages in a clean environment without any problems.

Why Packaging Extension?

The Debian Ruby team has a lot of experience in packaging and maintaining Ruby libraries and applications for Debian. During this work, they identified several issues in upstream codebases that make it difficult to build a Debian package straight out of those Ruby gems (shipped via [RubyGems](#)).

The Debian developers (downstream maintainers) have been in touch with the RubyGems & Bundler and other upstream maintainers and we're collaborating to make things easier for OS packagers while not compromising the experience for upstream maintainers.

As a result, we're working on this RuboCop extension to enforce a set of best practices that upstream maintainers can follow to make the lives of packagers easier. And that is how rubocop-packaging is born!

[The Packaging Style Guide](#) lists the cops with some good and bad examples and the rationale behind these cops can be found in the [Cops Documentation](#) section.

Introduction:

Debian and Packaging

The basis of the open-source environment is sharing, which in turns acts as a bridge between everyone out there and makes things grow holistically, this idea has made very great things happen.

The package refers to an assemblage of files and data, Linux distros are generally installed as a unique package, each consisting of specific applications moreover a development environment. Every package comprises an archive of records about the software, like name, version, and description. We have different package managers for different distros like ubuntu has apt, while fedora has yum, which automates the process of installation.

The package in Debian involves a unique source package component and various binary package components. Debian Policy wants that the package files should be built on a particular structure and format but there are various approaches for the same.

The packaging is done in a suitable environment in which the system is upgraded to unstable. Also, the need for Debian or Linux is required for Debian packaging.

Ruby and its based application are also based on the same idea of sharing, a developer grows rapidly, the advantage of having various libraries and frameworks available.

Packaging a ruby gem makes them available to the package manager to directly install the gem in the environment using the specific package managers.

The most significant example of Debian packing one can see is Gitlab providing with web-based DevOps lifecycle instrument that offers a Git-repository manager providing wiki, issue-tracking, and CI/CD pipeline structures, via open-source license, established by GitLab Inc.

RuboCop

RuboCop is a Ruby static code analyzer (a.k.a. linter) and code formatter. Out of the box it will enforce many of the guidelines outlined in the community [Ruby Style Guide](#).

RuboCop packs a lot of features on top of what you'd normally expect from a linter:

- Works with every major Ruby implementation
- Auto-correction of many of the code offenses it detects
- Robust code formatting capabilities

- Multiple result formatters for both interactive use and for feeding data into other tools
- Ability to have different configuration for different parts of your codebase
- Ability to disable certain cops only for specific files or parts of files
- Extremely flexible configuration that allows you to adapt RuboCop to pretty much every style and preference
- It's easy to extend RuboCop with custom cops and formatters
- A vast number of ready-made extensions (e.g. rubocop-rails, rubocop-rspec, rubocop-performance and rubocop-minitest)
- Wide editor/IDE support
- Many online services use RuboCop internally (e.g. HoundCI, Sider and CodeClimate)
- Best logo/stickers ever

The project is closely tied to several efforts to document and promote the best practices of the Ruby community:

- [Ruby Style Guide](#)
- [Rails Style Guide](#)
- [RSpec Style Guide](#)
- [Minitest Style Guide](#)

A long-term goal of RuboCop (and its core extensions) is to cover with cops all the guidelines from the community style guides.

Philosophy

Early on RuboCop aimed to be an opinionated linter/formatter that adhered very closely to the Ruby Style Guide (think gofmt and the like). In those days cops supported just a single style and you couldn't even turn individual cops off. Eventually, we realized that in the Ruby community there were many competing styles and preferences that it was going to be really challenging to find one set of defaults that makes everyone happy. Part of this was Ruby's own culture and philosophy, part was the lack of common standards for almost 20 years. It's hard to undo any of those, but it's also not really necessary.

The early feedback we got led us to adopt the philosophy of (extreme) configurability and flexibility, and trying to account for every *common* style of programming in Ruby. While we still believe that there's a lot of merit to just sticking to the community style guides, we acknowledge that Ruby is all about diversity and doing things the way that makes you happy. Whatever style preferences you have RuboCop is there for you. That's our promises and our guarantee. Within the subjective limits of sanity that is.

Next Steps

So, what to do next? While you can peruse the documentation in whatever way you'd like, here are a few recommendations:

- See ["Basic Usage"](#) to get yourself familiar with RuboCop's capabilities.
- Adjust RuboCop to your style/preferences. RuboCop is an extremely flexible tool and most aspects of its behavior can be tweaked via various [configuration options](#). See ["Configuration"](#) for more details.
- See ["Versioning"](#) for information about RuboCop versioning, updates, and the process of introducing new cops.
- Explore the [existing extensions](#).

Literature Review:

- Getting ready with the environment
- A little brush on Unix command line and VCS (git and gbp)
- Setting up the environment
 - Installing the necessary packages and scripts
 - Configuring the environment
- Packaging
 - Identifying the gem
 - Filing the ITP
 - Gem: Creation and Fetching
 - Creation and management of git branches (upstream, master, Debian)
 - Package the gem
 - Error handling of issues
 - Request for sponsorship
 - Wait till it gets uploaded in Debian

Note: \$ is used to denote terminal prompt and is not part of the command

● **Getting ready with the environment**

Debian environment Like Linux distros moreover Debian itself, is the need for packaging, the environment is made unstable by changing the source.list file in the /etc/apt directory, this is the package manager.

Debian and other Linux distributions are freely available. Also, a virtual box can also be used in installing a virtual environment for Debian.

Basically, Debian has various distributions like stable, unstable, testing, experimental and oldstable. Every distribution has its own importance and classification of features.

Why unstable?

Unstable (a.k.a. Sid, a.k.a Still In Development)

unstable contains packages uploaded by the developers for the next release, but will never be released.

Instead, packages will usually migrate to testing if no release-critical bugs are found in 10 days (and there are no dependency problems).

So basically after the release, the package goes through various checks before going to testing.

How?

The source.list in /etc/apt/source.list is edited to :

```
$ nano /etc/apt/sources.list
```

The file should be like this:

```
deb http://deb.debian.org/debian squeeze main
deb http://deb.debian.org/ squeeze/updates main
```

For the first line, change "squeeze main" at the end of the line with "sid or unstable main", and replace the entire second line as shown:

After changing do

```
$ sudo apt-get update
$ sudo apt dist-upgrade
```

- A little brush on Unix command line and VCS (git and gbp)

General command needed:

```
mkdir directory_name creation of directory
cd directory_name use to go on the directory
mv new old use to change the name
cd .. used to go back to the previous directory
```

VCS a.k.a version control system is an arrangement that records various amendments made to a file or on its collection over time, in order to recall the specific version later. In packaging use of git and gbp is being used.

- **Setting up the environment**
 - o Installing the necessary packages and scripts

Following packages are prerequisite, hence installing it into the kernel:

```
$ sudo apt-get install gem2deb quilt git-buildpackage
```

So, gem2deb is basically a tool which converts our gem to a debian package.

quilt is used to add and modify patches made inside the Debian package.

git-buildpackage is used for construction for a build package.

- o Configuring the environment

- Depending upon the shell add following details to `~/.bashrc` or `~/.zshrc`

```
export DEBEMAIL=your@email.domain

export DEBFULLNAME='Your Name'

alias lintian='lintian -iIEcv --pedantic --color auto'
```

- To make the above command works, run the following command, this is done to make `.bashrc` file specific to your own shell.

```
$ source ~/.bashrc
```

- Change the command to `~/.quilt.rc`, and add the following details:

```
QUILT_PATCHES=debian/patches

QUILT_NO_DIFF_INDEX=1

QUILT_NO_DIFF_TIMESTAMPS=1

QUILT_REFRESH_ARGS="-p ab"

QUILT_DIFF_ARGS="--color=auto"
```

- Change the directory and open `~/.ssh/config` and add the following lines, replacing `username-guest` with your actual alioth (alioth user name is your gitlab account username):

```
Host git.debian.org alioth.debian.org

  User username-guest
```

- **Packaging**

- o Identifying the gem

Rubygems.org consists of almost all the gems (exceptions mainly include rails-assets-* gems), it is acting like a central repository for distribution. It contains most of the basic

information such as Runtime-Development dependencies, Homepage link, Upstreams Authors, Documentations, License etc.

Firstly check, whether someone is working on the package you took or not, at the very initial phase one has to ensure that no other person is working on the same because effort should not be wasted because of duplication. The <http://wnpp.debian.net> and <https://git.fosscommunity.in/debian-ruby/> and look for the package name in the search option, if any result comes up, then drop the package and switch to another one, as someone else is already working on it.

o Filing ITP

While constructing a Debian package, it is primarily necessary to file a bug generally known as ITP stands for 'Intend To Package'. Bug in Debian means a package. So, filing bugs will make your work global, and also it helps to minimize duplication of data. To file an ITP, either report a bug tool or send a mail to submit@bugs.debian.org. The mail should be properly structured.

****** For an instance let us take the ruby-safely-block package (I'm the uploader for the same package)

Mail for the gem was sent to submit@bugs.debian.org with the proper format.



ITP: ruby-safely-block -- Rescue and report exceptions in non-critical code
1 message

Samyak Jain <samyak.jn11@gmail.com>
To: submit@bugs.debian.org

Package: wnpp
Severity: wishlist
Owner: Samyak Jain <samyak.jn11@gmail.com>

* Package name : ruby-safely-block
Version : 0.2.1
Upstream Author : Andrew Kane <andrew@charkick.com>
* URL : <https://github.com/ankane/safely>
* License : Expat
Programming Lang: Ruby
Description : Rescue and report exceptions in non-critical code

Exceptions are rescued and automatically reported to your favorite reporting service. In development and test environments, exceptions are raised so you can fix them.

The Safely Pattern is a simple one. It allows you to tag non-critical code by wrapping it in a function. It's built on top of exception handling. Gem is created for efficient exceptional handling.

It is a dependency for loomio and hence needs to be packaged.

Thanks,
Samyak Jain

o Gem: Creation and Fetching

Firstly, create a directory with the same name, and now get inside the directory, there is no significance the same name of the directory is just given to remove confusion. All the operations are performed inside the directory.

Gem2deb is the preferred way of the Debian Ruby team, initiating the following command inside the directory in the terminal.

```
$ gem2deb <gem name>
```

Example: \$ gem2deb safely-block

The command is generally fetching the gem from the centralised repo i.e. rubygems.org page. After this, it generates the source tarball from the ruby gem, then on the third step it generates the Debian source package which is extracted from the tarball, and at last finally the package is built.

If there is any error which says **Unmet build dependencies**, just install the packages with the apt-get install <pkg-name> and run gem2deb again from scratch, and if any test failure comes, these failures need patches to resolve and can be done later.

ls commands in Linux gives you the list of files and directory. By checking the list various files with unique extensions can be spotted.

Such as:

- folder ruby-safely-block-<version number>
- safely-block_<version>.tar.gz file
- a ruby-safely-block-version.orig.tar.gz file, it is a symlink to the preceding tarball.

If the fourth step for gem2deb run successfully and the build was successful, following files are also constructed:

- File of _dsc
- File of _changes
- File of _deb and a .debian.tar.xz file, these are the consequence of the build.

If a specific version of a gem is needed to get packaged then:

```
$ gem fetch -v <gem name>

$ gem2deb <passthe file generated above>
```

- o Creation and management of git branches (upstream, master, Debian)

Debian packaging of ruby-gems trails a git-based packaging schema, which makes use of gbp and the salsa Debian(alioth) repositories. Therefore, the packaging is performed under a git directory consisting of three branches:

- master, upstream and pristine-tar
 - o master – This branch has the source code as well as all the packaging work is done and stored here
 - o upstream – This branch stores upstream data
 - o pristine-tar - This branch contains all the delta files that one needs in git for the regeneration of packages.

*Make all the changes specific to the master branch, command `$ git branch` will show the current and other git branches available.

1. The conversion of the package into the version-control directory is done via `git-buildpackage`, now run the given command it will create a directory with ruby-<...>
Example: ruby-safely-block (source directory)

```
$ gbp import-dsc --author-is-committer --pristine-tar <path to the  
dsc file created by gem2deb>
```

2. The source directory should be point towards the master branch, ensure about all the three-branch creation

```
$ git branch
```

3. Git tag is generally telling about all the tags applied to the git repository

```
$ git tag
```

4. From the above two tags (upstream and Debian), the Debian tag is removed to signify the state of the package in archives of Debian. Therefore, before uploading various changes and amends are made to the package, and hence after accepting the package in Debian, a new Debian tag is given to the package.

```
$ git tag -d debian/<version>--<revision>
```

- o Package the gem

After importing the .dsc file, a new directory with name `ruby-safely block` will be created, now we have to edit the basic information inside a Debian directory of the root package.

Inside Debian, we have `d/control`, `d/copyright`, `d/changelog`, `d/compat`, etc. All the new versions and the update vcs is updated when a Debian package is created using gem2deb, but the `d/copyright` files remains unchanged it needs to get edited suitable to the license of the package, we have various license like MIT(expat), Apache-2.0 and many more. Therefore the copyright should be edited as suitable.

Coming to the basic commits needed inside the Debian package:

- `debian/control` : In the control file, we can see the need to add a short and long description, actually these files tell about the nature of the package, like in safely-block it tells Rescue and report exceptions in non-critical code, and is followed by the long description. Proper indentation is to be followed when we edit the control file, the description should have not more than 80 words and a space should be added in front of each line, the difference between two paragraphs should be differentiated by add a new line with `.`.
- `debian/copyright` : In the copyright file, the upstream author is added according to the upstream, as well as the suitable information is added according to the license. We generally remove (fixme) from the expat, also the proper format is to be added for the year, name, and email-id.
Such as 2019, Samyak Jain < samyak.jn11@gmail.com >
A proper upstream author should be checked before packaging.
- `debian/changelog` : In the changelog file, the UNRELEASED should be changed to unstable, also the bug number is also replaced by ITP number, the bug number is mailed by the BTS system from Debian after submitting the ITP number to <submit@bug.debian.org>.
These are the basic commits needed to be followed.
- After editing the basic package information we run the commands for building the patch and check whether the build is free from bugs and errors, in case of errors we need to write patches for the same.
The errors may be because of various aspects such as git file error, internet usage error, use of installed libraries etc. Debian wiki has provided a proper guide for the management of usual errors.

- o **Error handling of issues**


After running the `dpkg-buildpackage -uc -us`, various errors in safely-blocking are seen. We need to add patches for the same for the successful packaging for the gem. So, the following errors are rectified in the same:

1. In the package safely-block, we see that we have bundler usage in `/test/test_helper.rb` and `rakefile`, we need to patch out these errors because bundler and ruby gems use the internet for fetching the related gems and files. But when we build the package there is no use of the internet and hence we need to remove these files from the source file directories from the package. So that all the changes are made via `dpkg-source --commit` or by using the quilt. Changes made are added in the `Debian/patches/space` directories and the order of patches is saved inside the `Debian/patches/series`.

```
001-remove-bundler 622 Bytes
1 Description: Remove usage of bundler from tests
2 .
3 ruby-safely-block (0.2.1-1) unstable; urgency=medium
4 .
5 * Initial release (Closes: #929135)
6 Author: Samyak Jain <samyak.jn11@gmail.com>
7 Bug-Debian: https://bugs.debian.org/929135
8
9 ---
10 --- a/test/test_helper.rb
11 +++ b/test/test_helper.rb
12 @@ -1,5 +1,5 @@
13 -require "bundler/setup"
14 -Bundler.require(:default)
15 +#require "bundler/setup"
16 +#Bundler.require(:default)
17 require "minitest/autorun"
18 require "minitest/pride"
19
20 --- a/Rakefile
21 +++ b/Rakefile
22 @@ -1,4 +1,4 @@
23 -require "bundler/gem_tasks"
24 +#require "bundler/gem_tasks"
25 require "rake/testtask"
26
27 task default: :test
```


2. The second patch added in the safely-block is meant to update the gemspec, we change the `ls-files` of the directories in order to drop the git usage so that proper linking of files can be passed in the directory.

 corrected patches
Samyak Jain authored 1 month ago

002-updating-git_files-in-gemspec 779 Bytes 


```
1 Description: Updating git files in gemspec
2 .
3 ruby-safely-block (0.2.1-1) unstable; urgency=medium
4 .
5 * Initial release (Closes: #929135)
6 Author: Samyak Jain <samyak.jn11@gmail.com>
7 Bug-Debian: https://bugs.debian.org/929135
8
9 ---
10 --- ruby-safely-block-0.2.1.orig/safely_block.gemspec
11 +++ ruby-safely-block-0.2.1/safely_block.gemspec
12 @@ -13,7 +13,7 @@ Gem::Specification.new do |spec|
13     spec.homepage     = "https://github.com/ankane/safely"
14     spec.license       = "MIT"
15
16 - spec.files          = `git ls-files -z`.split("\n")
17 + spec.files          = Dir.glob("**/*").select { |v| v !~ /^debian/ }
18     spec.executables   = spec.files.grep(%r{^bin/}) { |f| File.basename(f) }
19     spec.test_files    = spec.files.grep(%r{^(test|spec|features)/})
20     spec.require_paths = ["lib"]
```



3. After writing the first patch and then running the `dpkg -buildpackage` command again we saw that a ruby module was missing and hence showing ruby 2.5 test aborted!. So, we will add the required module from the file it's lacking. So finally we will add a safely blocked module in the gemspec.
For that, require “safely_block” is added in the `/test/test_helper.rb`.

003-adding-gemspec 426 Bytes 

```
1 Description: adding gemspec file
2 .
3 ruby-safely-block (0.2.1-1) unstable; urgency=medium
4 .
5 * Initial release (Closes: #929135)
6 Author: Samyak Jain <samyak.jn11@gmail.com>
7 Bug-Debian: https://bugs.debian.org/929135
8
9 ---
10 --- a/test/test_helper.rb
11 +++ b/test/test_helper.rb
12 @@ -1,5 +1,6 @@
13     #require "bundler/setup"
14     #Bundler.require(:default)
15 +require "safely_block"
16     require "minitest/autorun"
17     require "minitest/pride"
18
```

4. All the patches are added in the Debian `/patches/series` in a sequential manner when the `dpkg-buildpackage` runs then the patches are added in the build itself.

 **modified series**
Samyak Jain authored 1 month ago

 **series** 72 Bytes 

1

001-remove-bundler

2

002-updating-git_files-in-gemspec

3

003-adding-gemspec

There are different types of error in packaging, most of the common errors are documented in the Debian wiki, and by following them one can rectify the errors, also new packages are less probable for errors, and version updates are more prone to these. Some errors are even hard to find, for that we have Debian mailing list to contact core deb people.

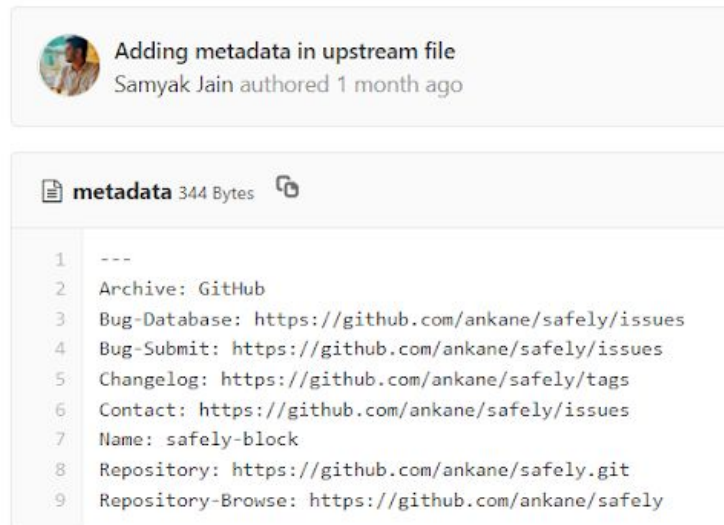
o Check for the lintian error

After the successful running of the package, it goes through the lintian check, we have to pass our package lintian clean, these involve proper formats, indentation checks, uploaders details, check all the formats in control, copyright and changelog files.

Also, check for metadata in upstream- we need to add the metadata in upstream directory for all the new packages.

```
$ dpkg-buildpackage -uc -us
# after successful build go for lintian check
$ lintian
#for adding metadata in upstream, create the directory
and add the metadata
$ mkdir -p debian/upstream
#now add the file
$ nano Debian/upstream/metadata
```

The following data is added in the upstream directory in the metadata file.



Finally, after clarifying the lintian warning and errors ,our mmain step for building the package comes for that inside the package we run sbuild command on the terminal .this is built have to be installed and configured before constructing the package,the .dotfile sbuild is ~/.sbuilddrc ,the configuration is done as follows:

Now if we have proper configuration we will construct using the same command .

Why?

Sbuild is the official build network which is used to construct various binary packages which is supported by almost all the architecture...sbuild is also used by Debian packager to test their packages which are being built in unstable environments.

Sbuild also comes with various flags ,let us suppose if a package needs a build dependency which is not yet accepted in the Debian then we can include it by passing --extra-package command and passing the .deb file of the required dependency.

Sbuild check is as follows:

We can see all these changes in the .amd.change binary files .If sbuild becomes successful the primary stage of building the package and lintian is successful. After building it successfully we can contact our Debian maintainer or developer to review our package.

One step forward is /meta/build it checks for all the packages of all the development architecture, autopkg test and checks for build dependency, reverse dependencies and all the related ruby tests included in the ruby-test.rake.

All the tests that include internet fetching and updates should be removed from the files. These tests are included in the autopkg section and at the time of the meta build,

these tests are reviewed and the errors are further reported. If autopkg is followed by an error, the test bed shows BAD pkg error or if the tests are clear then, it comes to the reverse dependency section resulting in an upbuild directory which in turn is a successful step for the package.

After the successful run, the packages are uploaded to the VCS control. The salsa.debian.org or gitlab is the proper VCS package manager for the packages. Now, the git link of the package is sent to the alioth Debian ruby team which reviews your package and runs various CI checks, these checks are made by Debian maintainer and after a successful review, these packages are uploaded by the Debian developers of the Ruby team.

o Request for sponsorship

For uploading the package, a mail for the sponsorship is sent to the Debian developer of the group, there is a proper format available in the documentation of the Debian wiki.

1. The RFS (Request for Sponsorship) for the gems are given as follows:
2. The RFS mail was sent to Praveen, which is a core Debian Developer.



RFS: ruby-heroku-deflater, ruby-discriminator, ruby-discourse-diff
2 messages

Samyak Jain <samyak.jn11@gmail.com>
To: debian-ruby@lists.debian.org

Hi,

I've prepared the Debian package of the Ruby gem heroku-deflater, discriminator, discourse-diff. The package was tested on sbuild and was successfully built. It is also lintian-clean. I've uploaded the package to the salsa repo which may be found at
<https://salsa.debian.org/ruby-team/ruby-heroku-deflater>
<https://salsa.debian.org/ruby-team/ruby-discriminator>
<https://salsa.debian.org/ruby-team/ruby-discourse-diff>

These packages are dependency for loomio and hence are required for completing the packaging.

Requesting you to please review and upload the same.

Samyak Jain

In approval,



RFS: ruby-heroku-deflater, ruby-discriminator, ruby-discourse-diff

Pirate Praveen <praveen@onenetbeyond.org>
To: Samyak Jain <samyak.jn11@gmail.com>
Cc: debian-ruby@lists.debian.org

On Thu, May 16, 2019 at 11:36 PM, Samyak Jain <samyak.jn11@gmail.com> wrote:
Hi,

I've prepared the Debian package of the Ruby gem heroku-deflater, discriminator, discourse-diff. The package was tested on sbuild and was successfully built. It is also lintian-clean. I've uploaded the package to the salsa repo which may be found at
<https://salsa.debian.org/ruby-team/ruby-heroku-deflater>
<https://salsa.debian.org/ruby-team/ruby-discriminator>
<https://salsa.debian.org/ruby-team/ruby-discourse-diff>

These packages are dependency for loomio and hence are required for completing the packaging.

Requesting you to please review and upload the same.

Uploaded! Congrats on your first 3 packages and welcome to the team!

Samyak Jain

Methodology:

Cops:

In RuboCop lingo, the various checks performed on the code are called "cops". Each cop is responsible for detecting one particular offense. RuboCop Packaging has only one department, called Packaging.

Packaging

Packaging cops helps both, upstream and downstream maintenance of your projects.

Department [Packaging](#)

- [Packaging/GemspecGit](#)

Packaging/GemspecGit

<u>Enabled</u> <u>default</u>	<u>by</u> <u>Safe</u>	<u>Supports</u> <u>autocorrection</u>	<u>VersionAdded</u>	<u>VersionChanged</u>
<u>Enabled</u>	<u>Yes</u>	<u>No</u>	<u>0.1</u>	<u>0.1</u>

Avoid using git ls-files to produce lists of files. Downstreams often need to build your package in an environment that does not have git (on purpose). Instead, use some pure Ruby alternatives, like Dir or Dir.glob.

Rationale

Packages in Debian are built in a clean environment ([sbuild](#), [schroot](#), et al) and whilst doing so, the build fails with:

```
Invalid gemspec in [<gem_name>.gemspec]: No such file or directory - git
```

And adding git as a dependency for each of the Ruby packaging is something that is not right and definitely not recommended. Besides, the source package consists of released tarballs (usually downloaded from GitHub/GitLab releases page or converted from the .gem file, obtained using `gem fetch foo`), which is extracted during build. So even if we add git as a build dependency, it would still fail as the Debian package source tree is not a git repository. Even when the package is maintained in git, it is uploaded as tarballs to the archive without any version control information.

Therefore, the only way forward here is to patch out the usage of git and use some plain Ruby alternatives like `Dir` or `Dir.glob` or even `Rake::FileList` whilst doing the Debian maintenance.

There's not only Debian or other OS packaging situation/examples, but also a couple of others, for instance:

- ruby-core as part of their CI system runs their test suite against an unpackaged Ruby tarball which doesn't have a `.git` directory. That means they needed to overwrite the bundler gemspec to not use git.
Actually, not anymore, [since git has been removed from bundler's gemspec](#).
- If you build your application on a bare docker image without git, and you are pointing to a git sourced gem that uses git on its gemspec, you'll get: `No such file or directory - git ls-files (Errno::ENOENT)` warnings all around. For example, if you use this in your Gemfile:

```
gem "foo", git: "https://github.com/has-git-in-gemspec/foo"
```

Originally, git ls-files inside the default gemspec template was designed so that users publishing their first gem wouldn't unintentionally publish artifacts to it. Recent versions of bundler won't let you release if you have uncommitted files in your working directory, so that risk is lower.

Examples

```
# bad
Gem::Specification.new do |spec|
  spec.files      = `git ls-files`.split("\n")
  spec.test_files = `git ls-files -- spec`.split("\n")
end

# good
Gem::Specification.new do |spec|
  spec.files      = Dir["lib/**/*", "LICENSE", "README.md"]
  spec.test_files = Dir["spec/**/*"]
end
```

```
# bad
Gem::Specification.new do |spec|
  spec.files = Dir.chdir(File.expand_path(__dir__)) do
    `git ls-files -z`.split("\x0").
      reject { |f| f.match(%r{^(test|spec|features)/}) }
  end
end

# good
require "rake/file_list"

Gem::Specification.new do |spec|
  spec.files      = Rake::FileList["**/*"]
                  .exclude(*File.read(".gitignore").split)
end
```

```

# bad
Gem::Specification.new do |spec|
  spec.files          = `git ls-files -- lib/`.split("\n")
  spec.test_files     = `git ls-files -- test/{unit}/*`
                        .split("\n")
  spec.executables    = `git ls-files -- bin/*`
                        .split("\n").map{ |f| File.basename(f) }
end

# good
Gem::Specification.new do |spec|
  spec.files          = Dir.glob("lib/**/*")
  spec.test_files     = Dir.glob("test/{functional,test}/*")
  spec.executables    = Dir.glob("bin/*")
                        .map{ |f| File.basename(f) }
end

```

Implementation

STEP 1: Determining the usage of `git ls-files` in the `gemspec` file using the AST (anti-pattern) on the entire source code.

```

class GemspecGit < Cop

  def_node_search :xstr, <<~PATTERN
    (block
      (send
        (const
          (const {cbase nil?} :Gem) :Specification) :new)
        (args
          (arg _)) `(xstr (str start_with('git')))))
    PATTERN
  end
end

```


STEP 2: The next thing is to process the AST formed against the source code to match problematic lines:

```
def investigate(processed_source)
  xstr(processed_source.ast).each do |node|
    add_offense(
      processed_source.ast,
      location: node.loc.expression,
      message: MSG
    )
  end
end
```

STEP 3: Write tests:

```
RSpec.describe RuboCop::Cop::Packaging::GemspecGit do
  subject(:cop) { described_class.new(config) }

  let(:config) { RuboCop::Config.new }
  let(:message) { RuboCop::Cop::Packaging::GemspecGit::MSG }

  it 'registers an offense when using `git` for :files=' do
    expect_offense(<<~RUBY)
      Gem::Specification.new do |spec|
        spec.files = `git ls-files`.split("\n")
                      ^^^^^^^^^^^^^^^^^^^ # {message}
      end
    RUBY
  end
end
```

Source Code: The source code for the entire library becomes something like:

```
module RuboCop
  module Cop
    module Packaging
      class GemspecGit < Base
        # This is the message that will be displayed when
        # RuboCop finds an offense of using `git ls-files`.
        MSG = "Avoid using git to produce lists of files. " \
              "Downstreams often need to build your " \
              "package in an environment " \
              "that does not have git (on purpose). " \
              "Use some pure Ruby alternative, like `Dir`."

        def_node_search :xstr, <<~PATTERN
          (block
            (send
              (const
                (const {cbase nil?} :Gem) :Specification) :new)
              (args
                (arg _)) `$(xstr (str #starts_with_git?)))
          PATTERN

          # Extended from the Cop class.
          # More about the `#investigate` method can be found in the docs.
          #
          # Processing of the AST happens here.
          def on_new_investigation
            return if processed_source.blank?

            xstr(processed_source.ast).each do |node|
              add_offense(
                node.loc.expression,
                message: MSG
              )
            end
          end
        end

        # This method is called from inside `#def_node_search`.
        # It is used to find strings which start with "git".
        def starts_with_git?(str)
          str.start_with?("git")
        end
      end
    end
  end
end
```

And the source code for testing becomes:

```
RSpec.describe RuboCop::Cop::Packaging::GemspecGit do
  subject(:cop) { described_class.new(config) }

  let(:config) { RuboCop::Config.new }

  let(:message) { RuboCop::Cop::Packaging::GemspecGit::MSG }

  it "registers an offense when using `git` for :files=" do
    expect_offense(<<~RUBY)
      Gem::Specification.new do |spec|
        spec.files = `git ls-files`.split("\n")
                      ^^^^^^^^^^^^^^^^^^ #{message}
      end
    RUBY
  end

  it "registers an offense when using `git ls-files filename` for :files=" do
    expect_offense(<<~RUBY)
      Gem::Specification.new do |s|
        s.files          = `git ls-files LICENSE docs lib`.split("\n")
                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ #{message}
      end
    RUBY
  end

  it "registers an offense when using `git` for :files= but differently" do
    expect_offense(<<~RUBY)
      Gem::Specification.new do |spec|
        spec.files = `git ls-files`.split + %(
          ^^^^^^^^^^^^^^^^^^ #{message}
          lib/parser/lexer.rb
          lib/parser/rubymotion.rb
        )
      end
    RUBY
  end

  it "does not register an offense when not using `git` for :files=" do
    expect_no_offenses(<<~RUBY)
      Gem::Specification.new do |spec|
        spec.files = Dir["lib/**/*"].reject { |f| File.directory?(f) }.sort
      end
    RUBY
  end
end
```

Conclusion:

Now that the library is up and running, let's install and use it in our production environment.

Installation

Add this line to your application's Gemfile:

```
gem 'rubocop-packaging', require: false
```

And then execute:

```
$ bundle install
```

Or install it yourself as:

```
$ gem install rubocop-packaging
```

Or on Debian based systems, you can also do:

```
$ apt install ruby-rubocop-packaging
```

Usage:

You need to tell RuboCop to load the Packaging extension. There are three ways to do this:

1. RuboCop Configuration File:

Put this into your .rubocop.yml file:

```
require: rubocop-packaging
```

Alternatively, use the following array notation when specifying multiple extensions:

```
require:
  - rubocop-other-extension
  - rubocop-packaging
```

Now you can run rubocop and it will automatically load the RuboCop Packaging cops together with the standard cops.

2. Command Line:

```
rubocop --require rubocop-packaging
```

3. Rake Task:

```
RuboCop::RakeTask.new do |task|
  task.requires << 'rubocop-packaging'
end
```

Running in production:

Now that the tool is ready to be deployed, let's use this in the other projects:

```
+ rubocop-packaging git:(master) rubocop --only Packaging
Inspecting 14 files
.....C.....

Offenses:

rubocop-packaging.gemspec:22:24: C: Packaging/GemspecGit: Avoid using git to produce l
ists of files. Downstreams often need to build your package in an environment that doe
s not have git (on purpose). Use some pure Ruby alternative, like Dir or Dir.glob.
  spec.executables  = `git ls-files`.split('\n')
                        ^^^^^^^^^^^^^^^^^
14 files inspected, 1 offense detected
+ rubocop-packaging git:(master) □
```

This correctly determines the usage of `git ls-files` in the `gemspec` file.

Future Goals:

This is just a start, there is a lot of scope of expansion of the project.

This project will have more cops that will be solving more such real-life problems that the downstream maintainers face in their daily work. It will help in the maintenance of the Ruby toolchain in all the distributions, like Debian, RedHat, OpenSUSE, Ubuntu, et al.

The cops that will be solving problems like (but not limited to):

1. Using ``require`` with relative path to ``lib`` directory.
2. Using ``require_relative`` from the test code to the ``lib`` directory.
3. Using ``require "bundler/setup"`` in the test code.

Supporting AutoCorrect:

The ``GemspecGit`` cop along with the other 3 mentioned above will be supporting an auto-corrector, which basically means that this project will not just be a simple linter but will also be able to auto-correct all the offenses that it catches.

This feature will need to be carefully implemented as it could result in a lot of false-positives. And so a thorough testing will need to be done prior to deployment and release.

There's also an ``AutoCorrector`` API that is already available in RuboCop which we'll use as per our convenience to enable this feature.

Releasing:

As and when a new cop is added, a release will be made to RubyGems and will be uploaded to the Debian archive. And each bug fix or a feature addition will warrant a point release, following the same principles for a new minor or major release. Those will be termed as patch releases.

For now, enjoy RuboCop::Packaging v0.1.0! :)

References

1. <https://rubocop.org/>
2. <https://github.com/rubocop-hq/rubocop>
3. <https://docs.rubocop.org/rubocop/>
4. <https://rubystyle.guide/>
5. <https://wiki.debian.org/Packaging>
6. <https://www.debian.org/doc/manuals/developers-reference>
7. https://www.debian.org/social_contract
8. <https://wiki.debian.org/DFSGLicenses>
9. <https://wiki.debian.org/Teams/Ruby/Packaging>
10. <https://wiki.debian.org/Teams/Ruby/Packaging/Tests>