Linter for Enforcing Downstream Checks

Utkarsh Gupta
Amity University
Noida, India
utkarsh@debian.org

Samyak Jain
Amity University
Noida, India
samyak.jn11@gmail.com

Dr. Pooja Singh
Amity University
Noida, India
psingh22@amity.edu

***Abstract*—RuboCop::Packaging is a RuboCop extension that is a static code analyzer (a.k.a. linter) and code formatter for Ruby. This helps to incorporate some of the guidelines for upstream maintainers so that the downstream can produce their packages without any problems in a clean environment.**

In packaging and managing Ruby libraries and Debian software, the Debian Ruby team has a lot of experience. They have found many problems in upstream codebases during this work that make it difficult to create a Debian package directly from those Ruby gems (shipped through RubyGems). The developers of Debian (downstream maintainers) have been in contact with RubyGems & Bundler and other upstream maintainers and we are working together to make it simpler for OS packagers while not losing the experience of upstream maintainers. As a result, we are working on this extension of RuboCop to implement a collection of best practises that can be adopted by upstream maintainers to make the lives of packagers simpler. And that's how rubocop-packaging came into being! The Packaging Style Guide lists the cops with some good and poor examples, and in the Cops Documentation section, the reasoning behind these cops can be found.

***Index Terms*—ruby, linter, AST, parsing**

I. Introduction

Broadly, applications installed in our host systems is packages of relevant files and components which are built and customized according to the wishes and expectations of customers. These are classified under Application Packaging. Also, this software includes individual files and resources which are packed in a software collection providing certain functional- ity, are environment and architecture-dependent. This report includes software packaging in Debian, it's UNIX-like oper- ating system consisting of open source software for everyone. Debian is one of the oldest. To be specific, a package in Debian is a cluster of files allowing various software distributed over Debian management systems. The motive behind the packaging is to permit the computerization of install, upgrade, update, configure and to purge the computer application in a consistent manner. RubyGems are basically acting like a package man- ager for the Ruby Programming Language. Packaging Ruby gems will allow direct package libraries in the Linux kernel.

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

RELATED WORK

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.

II. Follow Up

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.

III. Methodology

A. 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.

B. Packaging

Packaging is an extension of RuboCop (like Performance, Rails, etcetera) that helps both. upstream and downstream maintenance of Ruby libraries and applications. So it deals with all such issues, like using git in the gemspec file, and so on.

C. Statement

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.

D. 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 rec- ommended. 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 situa- tion/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 any- more, 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.

E. AST Formation

To be able to work on this, we need to process the entire source code into its respective AST (Abstract Syntax Tree). This can be easily done using the existing RuboCop's classes and API. Something like:

```
def ofinvestigation
  return ifprocessedsource . blank?

  x s t r ( processessource . a s t ) . each  do
    node a d d o f f e n s e (
      node . l o c . e x p r e s s i o
      n , message : MSG
```

```
        )
    end
end
```

F. Writing Anti-Patterns

Now, the next step is to write the anti-pattern of the problem we're trying to solve, that is, the git-in-gemspec problem. That would look something like:

```
def node  sea r c h  : x s t r ,  << PATTERN
    ( b l o c k
        ( send
            ( const
                ( const  : Gem)   : Specification )   : new ) ( a r g s
            ( a r g ) ) ‘ $ ( x s t r ( s t r ” g i t ” ? ) ) )
PATTERN
```

G. Installation

Add this line to your application's Gem- file: 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
```

F. *Usage*

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.

Command line

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

Rake task

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

IV. Future Goals

This project is not limited to resolve one issue downstream. It is yet to be expanded to resolve major other issues. 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.

A. Future Cops

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

- Using ‘require‘ with relative path to ‘lib‘ directory.
- Using ‘require relative‘ from the test code to the ‘lib‘ directory.
- Using ‘require “bundler/setup”‘ in the test code.

B. 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.

C. 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.

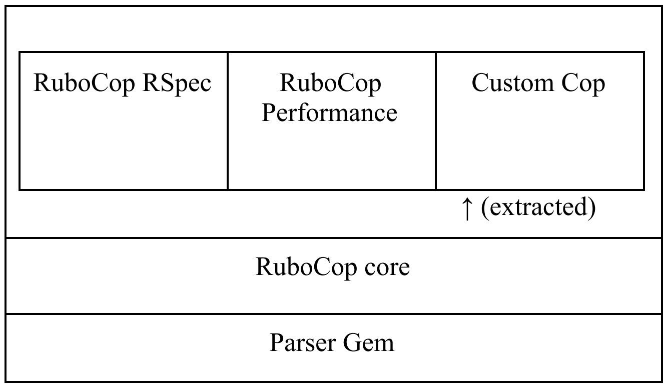
For now, enjoy RuboCop::Packaging v0.1.0!

D. Architecture

The principle architecture for RubCop is as follows (RuboCop Packaging is the extension to the same architecture).

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.

Here's how the customs cops are added in the main architecture:



Old Architecture:

RuboCop RSpec
RuboCop core
Parser Gem

New Architecture:

Custom Cops	RuboCop RSpec	RuboCop Performa nce	RuboCop Rails
Runtime engine and core cops	RuboCop core		
Node patterns and node extensions	RuboCop AST		
Ruby AST parser	Parser Gem		

E. *Comparison with ESLint*

ESLint and RuboCop can be categorized as "Code Review" tools.
"IDE Integration" is the primary reason why developers consider ESLint over the competitors, whereas "Open-source" was stated as the key factor in picking RuboCop.
ESLint and RuboCop are both open source tools. ESLint with 14.4K GitHub stars and 2.46K forks on GitHub appears to be more popular than RuboCop with 10.1K GitHub stars and 2.14K GitHub forks.
According to the StackShare community, ESLint has a broader approval, being mentioned in 541 company stacks & 592 developers stacks; compared to RuboCop, which is listed in 44 company stacks and 25 developer stacks.

References

[1] B. W. Boehm, Software Engineering Economics, Englewood Cliffs, NJ, USA:Prentice-Hall, 1981.

[2] C. Jaspan, I. Chen, A. Sharma et al., ”Understanding the value of program analysis tools”, Proc. 22nd ACM SIGPLAN Conf. Object- Oriented Program. Syst. Appl. Companion, pp. 963-970, 2007.

[3] M. Beller, R. Bholanath, S. McIntosh and A. Zaidman, ”Analyzing the state of static analysis: A large-scale evaluation in open source software”, Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng., pp. 470-481, 2016.

[4] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix and W. Pugh, ”Using static analysis to find bugs”, IEEE Softw., vol. 25, no. 5, pp. 22-29, Sep./Oct. 2008.

[5] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora and M. Di Penta, ”How open source projects use static code analysis tools in continuous integration pipelines”, Proc. 14th Int. Conf. Mining Softw. Repositories, pp. 334-344, 2017.

[6] ”Language trends on GitHub.”, Jun. 2015, [online] Available: https://github.com/blog/2047-language-trends-on-github.

[7] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[8] Popovici and M. Lalo, "Formal model and code verification in Model- Based Design", 2009 Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference, Jun. 2009.

[9] W. F. Lee, "Verilog Coding for Logic Synthesis", Apr. 2003.