Rust Packaging Tutorial @ Nest with Fedora 2022

Fabio Valentini
Rust SIG / FPC / FESCo

decathorpe@gmail.com
decathorpe@mastodon.social
@decathorpe
decathorpe
@decathorpe:fedora.im
decathorpe.com

Roadmap (Pt. 1/2): Introduction

- Anatomy of Rust crates
- Mapping Cargo metadata to equivalent RPM concepts:
  - Name, Version, Summary, License, %description
  - Features & Optional Dependencies
  - BuildRequires, Requires, Provides
- Mapping Cargo SemVer to RPM Version requirements

Roadmap (Pt. 2/2): Real-world examples

- Creating a new Rust package from scratch
- Updating an existing Rust package that can be updated independently
- Packaging bindings for system libraries
(for example, bindings for system “libcurl”)
- Handling of multi-crate / multi-package updates
- Creating “Compat” packages for older crate versions
- etc. (coverage depends on time constraints)

Prerequisites

The standard Fedora packaging toolkit plus some additional tools will be required to follow the tutorial:

- `fedora-packager`: rpm-build, mock, fedpkg, etc.
- `rust2rpm` (v22 is recommended):
  Our tool for generating RPM spec files for Rust crates.
- `rpmautospec`:
  New Rust packages default to using rpmautospec, and existing packages are being converted.

Introduction to Rust Packaging

What happens to Rust crates in RPM's build phases?

- `%prep`
  unpack .crate file (`%autosetup`) and set up local build environment for cargo (`%cargo_prep`)

- `%generate_buildrequires`
  generate `BuildRequires` from `Cargo.toml` according to enabled feature flags and taking into account whether tests are enabled (`%cargo_generate_buildrequires`)

- `%build`
  compile crate with specified feature flags (`%cargo_build`)

- `%install`
  install crate sources to the buildroot and install binaries to `%{buildroot}%{_bindir}` (`%cargo_install`)

- `%check`
  compile unit tests, integration tests, and doctests, and run them if tests are enabled (`%cargo_test`)

Contents of .crate files

Mandatory contents:

- `Cargo.toml` (project metadata, dependencies, etc.)
- crate source code
- `build.rs` (build script – if necessary)

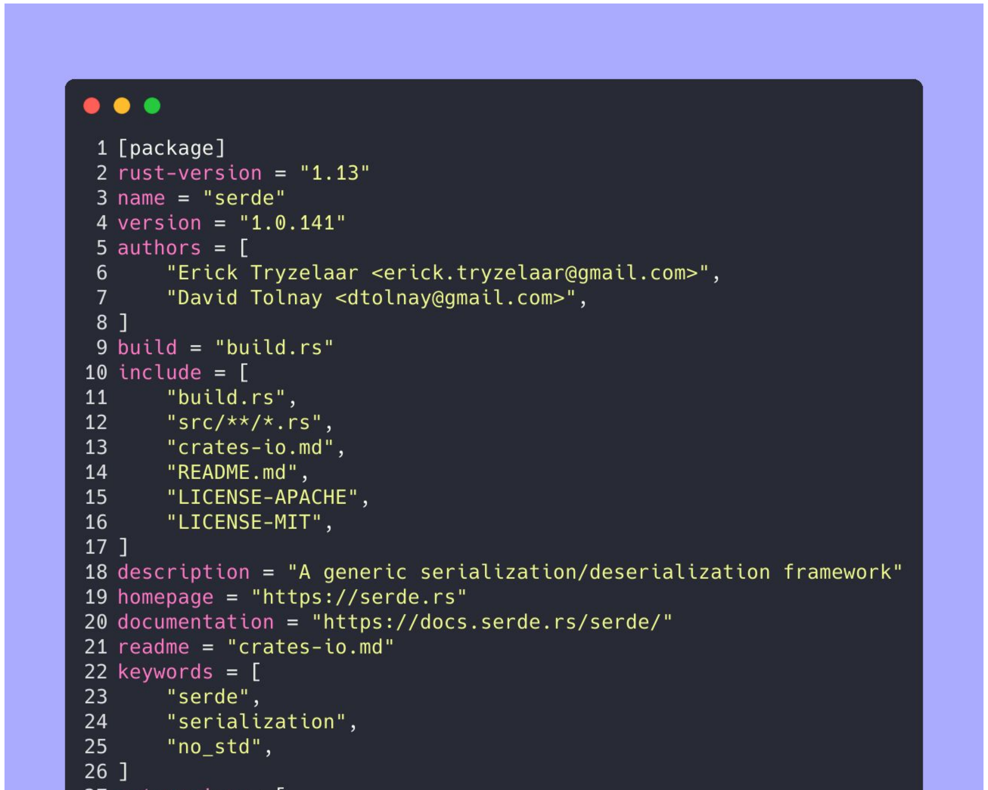
“Technically” optional contents:

- README file
- LICENSE files
- source code for tests, examples, benchmarks
- files that contain data for tests

Cargo.toml

The `[package]` table contains project metadata. The following values are interesting for generating RPM packages:

- `name`: maps to `rust-$name`
- `version`: Translated to RPM-compatible `Version` string.
- `description`: Used for `Summary` and `%description`.
- `license` (SPDX): Used to populate the `License` tag.



```toml
[package]
rust-version = "1.13"
name = "serde"
version = "1.0.141"
authors = [
    "Erick Tryzelaar <erick.tryzelaar@gmail.com>",
    "David Tolnay <dtolnay@gmail.com>",
]
build = "build.rs"
include = [
    "build.rs",
    "src/**/*.rs",
    "crates-io.md",
    "README.md",
    "LICENSE-APACHE",
    "LICENSE-MIT",
]
description = "A generic serialization/deserialization framework"
homepage = "https://serde.rs"
documentation = "https://docs.serde.rs/serde/"
readme = "crates-io.md"
keywords = [
    "serde",
    "serialization",
    "no_std",
]
```

Source code organization

The default (!) structure of a Rust crate looks like this:

- `src/**.rs`: source code and unit tests (can use private APIs; compiled into a single test runner binary)
- `src/lib.rs`: entry point for libraries (if present)
- `src/main.rs` and `src/bin/*.rs`: applications (if present)

- `tests/*.rs`: integration tests (can only use public APIs; compiled into separate test runners)
- `examples/*.rs`: example code (only compiled, but not executed, when running tests)
- `benches/*.rs`: benchmark code (untouched except when explicitly running benchmarks with `cargo bench`)

All aspects of this default layout can be overridden by explicit settings in the project's `Cargo.toml` file!

Mapping crate version to RPM Version

cargo uses semantic versioning (SemVer) as the format for its version strings, which can contain characters that are not valid in RPM version strings. For example, pre-releases will be normalized by rust2rpm and RPM generators:

    1.0.0-alpha.1 (SemVer) ←→ 1.0.0~alpha.1 (RPM)

Additionally, SemVer allows arbitrary suffixes after a "+" character, which should be stripped for RPM packages. This is often the case for bindings to C libraries, where this suffix usually contains the version of the bundled library:

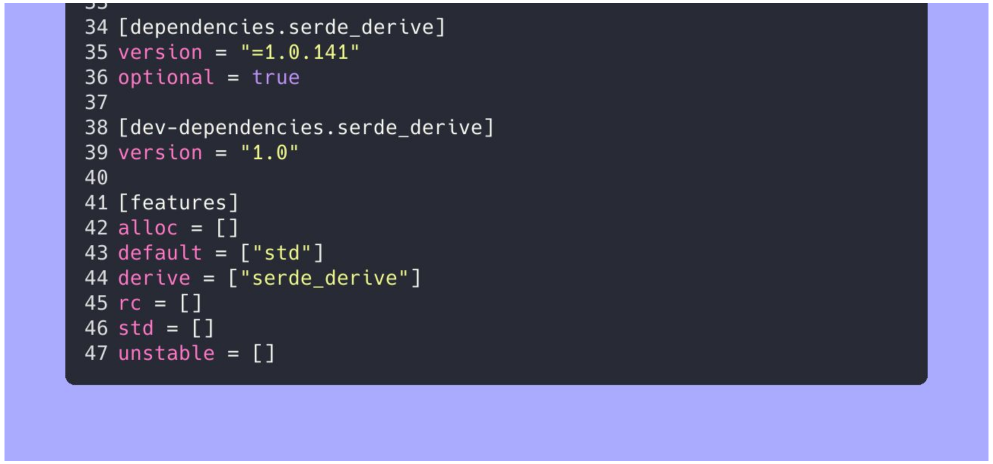
    0.4.56+curl-7.83.1 → 0.4.56

This information is not relevant for RPM packages, since we do not build against the bundled version of those libraries, but instead dynamically link to the shared library provided by the system.

Features and optional dependencies (1/2)

Rust crates can define a set of “features” and declare some of their dependencies as “optional” (optional dependencies implicitly also define a feature of the same name).

Features can, in turn, have a list of dependencies on other features, or on optional dependencies. All crates implicitly define an empty “default” feature unless it is explicitly specified to have dependencies.



```
[dependencies.serde_derive]
version = "=1.0.141"
optional = true

[dev-dependencies.serde_derive]
version = "1.0"

[features]
alloc = []
default = ["std"]
derive = ["serde_derive"]
rc = []
std = []
unstable = []
```

Features and optional dependencies (2/2)

Which features and / or optional dependencies are enabled can affect functionality and behaviour (i.e. with conditional compilation)!

Features and optional dependencies are mapped to RPM subpackages by rust2rpm, and RPM dependency generators ensure that dependencies between subpackages map to feature dependencies from `Cargo.toml`.

Since the set of features and optional dependencies can change with any new version of a crate, it is important to re-run rust2rpm for new versions - to ensure the mapping between optional dependencies / features and RPM subpackages stays in sync!

Mapping crate dependencies to RPM (Build)Requires

rust2rpm generates RPM spec files which run a “test build” during the `%build` phase to ensure that code that does not compile will fail RPM builds.

For this purpose, all crate `[dependencies]` and `[build-dependencies]` need to be available during the build, as well. These are generated automatically by the `%cargo_generate_buildrequires` macro, and also by the dependency generator for the `rust-$crate-devel` package.

Mapping crate dependencies to RPM (Build)Requires

By default, packages generated by rust2rpm also run the test suite of the packaged crate.

In this case, the `BuildRequires` generator will also include `[dev-dependencies]`, which define dependencies that are needed to compile and run unit and integration tests.

These dependencies are <u>not</u> generated for the `rust-$crate-devel` package, since they are not used at build-time or runtime, but only when compiling and running tests.

Mapping RPM packages to virtual Provides

In addition to dependencies (RPM `Requires`), the RPM generators for Rust crates also generate virtual `Provides` for all valid subpackages (i.e. they map to a feature or optional dependency).

subpackage	Provides
`rust-foo-devel`	`crate(foo) = %{version}-%{release}`
`rust-foo+default-devel`	`crate(foo/default) = %{version}-%{release}`
`rust-foo+bar-devel`	`crate(foo/bar) = %{version}-%{release}`

These virtual `Provides` are what is referenced by the `Requires` that are generated by dependency generators.

Mapping SemVer requirements to RPM (1/2)

Semantic Versioning provides syntax for “ranges” of versions when specifying which version of a dependency is required, and these also need to be mapped to RPM semantics.

```
foo = "0.1"
foo = "^0.1"
foo = "~0.1"
```

These expressions are all equivalent, and translate to this RPM dependency expression:

```
(crate(foo/default) >= 0.1.0 with crate(foo/default) < 0.2.0~)
```

Mapping SemVer requirements to RPM (2/2)

For post-1.0-releases, the “~”-style is no longer equivalent, and should not be used when building RPM packages in Fedora (because it is a stronger requirement than what is provided by SemVer compatibility guarantees).

```
foo = “1.1”
foo = “^1.1”
```

These expressions remain equivalent, and translate to this RPM dependency:

```
(crate(foo/default) >= 1.1.0 with crate(foo/default) < 2.0.0~)
```

However, `foo = “~1.1”` maps to the following dependency:

```
(crate(foo/default) >= 1.1.0 with crate(foo/default) < 1.2.0~)
```

Putting everything together (1/2)

For this example crate, the generated BuildRequires will contain:

- rust-packaging (RPM generators, etc.)
- cargo + rustc

If tests are enabled, an additional BuildRequires on `foo-test-data` will be generated:

```
[package]
name = “foo”
version = “1.0.1”

[dependencies.bar]
version = “1.0.2”
optional = true

[dev-dependencies.foo-test-data]
version = “0.1”

[features]
default = []
foobar = [“bar”]
```

```
BuildRequires: (crate(foo-test-data/default) >= 0.1.0 with crate(foo-test-data/default) < 0.2.0~)
```

Putting everything together (2/2)

For subpackages, RPM generator output will look something like this:

- rust-foo-devel

```
Provides: crate(foo) = 1.0.1
```

- rust-foo+default-devel

```
Requires: crate(foo) = 1.0.1
Provides: crate(foo/default) = 1.0.1
```

- rust-foo+bar-devel

```
Requires: crate(foo) = 1.0.1
Requires: (crate(bar) > 1.0.2 with crate(bar) < 2.0.0~)
Provides: crate(foo/bar) = 1.0.1
```

- rust-foo+foobar-devel

```
Requires: crate(foo/bar) = 1.0.1
Provides: crate(foo/foobar) = 1.0.1
```

```
[package]
name = “foo”
version = “1.0.1”

[dependencies.bar]
version = “1.0.2”
optional = true

[dev-dependencies.foo-test-data]
version = “0.1”

[features]
default = []
foobar = [“bar”]
```

Real-World Examples (finally!)

Getting help with Rust Packaging

If there are questions regarding Rust Packaging for Fedora, many Rust SIG members hang out on Matrix (or IRC).

#rust:fedoraproject.im

#fedora-rust

We also have a dedicated mailing list.

rust@lists.fedoraproject.org

The source code for rust2rpm and RPM macros for Rust packaging are hosted on pagure.io, where issues and feature requests can be filed.

https://pagure.io/fedora-rust/rust2rpm