Red Hat Developer Tools 2019.1

Using Rust Toolset

Installing and using Rust Toolset 1.31.1

Last Updated: 2019-09-16

Red Hat Developer Tools 2019.1 Using Rust Toolset

Installing and using Rust Toolset 1.31.1

Peter Macko
pmacko@redhat.com

Kevin Owen

Vladimir Slavik

Legal Notice

Abstract

Rust Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. The Rust Toolset User Guide provides an overview of this product, explains how to invoke and use the Rust Toolset versions of the tools, and links to resources with more in-depth information.

Table of Contents

CHAPTER 1. RUST TOOLSET 3
1.1. ABOUT RUST TOOLSET 3
1.2. COMPATIBILITY 3
1.3. GETTING ACCESS TO RUST TOOLSET ON RED HAT ENTERPRISE LINUX 7 3
Additional Resources 4
1.4. INSTALLING RUST TOOLSET 4
1.5. ADDITIONAL RESOURCES 5
Installed Documentation 5
Online Documentation 5

CHAPTER 2. CARGO 6
2.1. INSTALLING CARGO 6
2.2. CREATING A NEW PROJECT 6
2.3. BUILDING A PROJECT 7
2.4. CHECKING A PROGRAM 9
2.5. RUNNING A PROGRAM 10
2.6. RUNNING PROJECT TESTS 11
2.7. CONFIGURING PROJECT DEPENDENCIES 12
Additional Resources 14
2.8. BUILDING PROJECT DOCUMENTATION 14
Additional Resources 16
2.9. VENDORING PROJECT DEPENDENCIES 16
2.10. ADDITIONAL RESOURCES 18
Installed Documentation 18
Online Documentation 19
See Also 19

CHAPTER 3. RUSTFMT 20
3.1. INSTALLING RUSTFMT 20
3.2. USING RUSTFMT AS A STANDALONE TOOL 20
3.3. USING RUSTFMT WITH CARGO 20
3.4. ADDITIONAL RESOURCES 21

CHAPTER 4. CONTAINER IMAGE WITH RUST TOOLSET FOR RHEL 7 22
4.1. IMAGE CONTENTS 22
4.2. ACCESS TO THE IMAGE 22
4.3. ADDITIONAL RESOURCES 22

CHAPTER 5. CHANGES IN RUST TOOLSET IN RED HAT DEVELOPER TOOLS 2019.1 23
5.1. RUST 23
5.2. CARGO 24

CHAPTER 1. RUST TOOLSET

1.1. ABOUT RUST TOOLSET

Rust Toolset is a Red Hat offering for developers on the Red Hat Enterprise Linux platform. It provides the Rust programming language compiler **rustc**, the **cargo** build tool and dependency manager, the **cargo-vendor** plug-in, the **rustfmt** tool, and required libraries.

Rust Toolset is distributed as a part of Red Hat Developer Tools for Red Hat Enterprise Linux 7 and is available as a module in Red Hat Enterprise Linux 8.

The following components are available as a part of Rust Toolset:

Table 1.1. Rust Toolset Components

Package	Version	Description
rust	1.31.1	A Rust compiler front-end for LLVM.
cargo	1.31.0	A build system and dependency manager for Rust.
cargo-vendor	0.1.22	A cargo subcommand to vendor crates.io dependencies.
rustfmt	1.0.0	A tool for automatic formatting of Rust code.

1.2. COMPATIBILITY

Rust Toolset is available for Red Hat Enterprise Linux 7 and Red Hat Enterprise Linux 8 on the following architectures:

- The 64-bit Intel and AMD architectures
- The 64-bit ARM architecture
- The IBM Power Systems architecture
- The little-endian variant of IBM Power Systems architecture
- The IBM Z Systems architecture

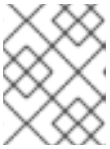
1.3. GETTING ACCESS TO RUST TOOLSET ON RED HAT ENTERPRISE LINUX 7

Rust Toolset is an offering that is distributed as a part of the Red Hat Developer Tools content set, which is available to customers with deployments of Red Hat Enterprise Linux 7. In order to install Rust Toolset, enable the Red Hat Developer Tools and Red Hat Software Collections repositories by using the Red Hat Subscription Management and add the Red Hat Developer Tools GPG key to your system.

1. Enable the **rhel-7-*variant*-devtools-rpms** repository:

```
# subscription-manager repos --enable rhel-7-variant-devtools-rpms
```

Replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**).



NOTE

We recommend developers to use Red Hat Enterprise Linux Server for access to the widest range of development tools.

2. Enable the **rhel-*variant*-rhscl-7-rpms** repository:

```
# subscription-manager repos --enable rhel-variant-rhscl-7-rpms
```

Replace *variant* with the Red Hat Enterprise Linux system variant (**server** or **workstation**).

3. Add the Red Hat Developer Tools key to your system:

```
# cd /etc/pki/rpm-gpg
# wget -O RPM-GPG-KEY-redhat-devel https://www.redhat.com/security/data/a5787476.txt
# rpm --import RPM-GPG-KEY-redhat-devel
```

Once the subscription is attached to the system and repositories enabled, you can install Red Hat Rust Toolset as described in Section 1.4, “Installing Rust Toolset”.

Additional Resources

- For more information on how to register your system using Red Hat Subscription Management and associate it with subscriptions, see the Red Hat Subscription Management collection of guides.
- For detailed instructions on subscription to Red Hat Software Collections, see the *Red Hat Developer Toolset User Guide*, Section 1.4. Getting Access to Red Hat Developer Toolset .

1.4. INSTALLING RUST TOOLSET

Rust Toolset is distributed as a collection of RPM packages that can be installed, updated, uninstalled, and inspected by using the standard package management tools that are included in Red Hat Enterprise Linux. Note that a valid subscription that provides access to the Red Hat Developer Tools content set is required in order to install Rust Toolset on a Red Hat Enterprise Linux 7 system. For detailed instructions on how to associate your Red Hat Enterprise Linux 7 system with an appropriate subscription and get access to Rust Toolset, see Section 1.3, “Getting Access to Rust Toolset on Red Hat Enterprise Linux 7”.

IMPORTANT

Before installing Rust Toolset, install all available Red Hat Enterprise Linux updates.

1. Install all of the components included in Rust Toolset for your operating system:
   - On Red Hat Enterprise Linux 7, install the **rust-toolset-1.31** package:

```
# yum install rust-toolset-1.31
```

- On Red Hat Enterprise Linux 8, install the **rust-toolset** module:

```
# yum module install rust-toolset
```

This installs all development and debugging tools, and other dependent packages to the system. Notably, Rust Toolset has a dependency on Clang and LLVM Toolset.

1.5. ADDITIONAL RESOURCES

A detailed description of the Rust programming language and all its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- The package **rust-toolset-1.31-rust-doc** installs the *The Rust Programming Language* book and API documentation in HTML format to **/opt/rh/rust-toolset-1.31/root/usr/share/doc/rust/html/index.html**.

Online Documentation

- Rust documentation – The upstream Rust documentation.
- Rust documentation overview – An extended overview of documentation related to Rust.

CHAPTER 2. CARGO

cargo is a tool for development using the Rust programming language. **cargo** fulfills the following roles:

- Build tool and frontend for the Rust compiler **rustc**.
  Use of **cargo** is preferred to using the **rustc** compiler directly.

- Package and dependency manager.
  cargo allows Rust projects to declare dependencies with specific version requirement. **cargo** will resolve the full dependency graph, download packages as needed, and build and test the entire project.

Rust Toolset is distributed with **cargo 1.31.0**.

2.1. INSTALLING CARGO

In Rust Toolset on Red Hat Enterprise Linux 7, **cargo** is provided by the **rust-toolset-1.31-cargo** package and is automatically installed with the **rust-toolset-1.31** package. On Red Hat Enterprise Linux 8, **cargo** is provided by the **rust-toolset** module. See Section 1.4, "Installing Rust Toolset".

2.2. CREATING A NEW PROJECT

To create a Rust program on the command line, run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo new --bin project_name'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo new --bin project_name
```

This creates a directory ***project_name*** containing a text file named **Cargo.toml** and a subdirectory **src** containing a text file named **main.rs**.

To configure the project and add dependencies, edit the file **Cargo.toml**. See Section 2.7, "Configuring Project Dependencies".

To edit the project code, edit the file **main.rs** and add new source files in the **src** subdirectory as needed.

To create a project for a cargo package instead of a program, run the **cargo** tool on the command line as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo new --lib project_name'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo new --lib project_name
```

Note that you can execute any command using the **scl** utility on Red Hat Enterprise Linux 7, causing it to be run with the Rust Toolset binaries available. This allows you to run a shell session with Rust Toolset **cargo** command directly available:

```
$ scl enable rust-toolset-1.31 'bash'
```

Example 2.1. Creating a Project using cargo

Create a new Rust project called **helloworld**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo new --bin helloworld'
    Created binary (application) helloworld project
```

- For Red Hat Enterprise Linux 8:

```
$ cargo new --bin helloworld
    Created binary (application) helloworld project
```

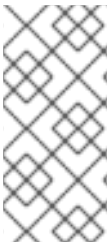
Examine the result:

```
$ cd helloworld
$ tree
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
$ cat src/main.rs
fn main() {
    println!("Hello, world!");
}
```

A directory **helloworld** is created for the project, with a file **Cargo.toml** for tracking project metadata, and a subdirectory **src** containing the main source code file **main.rs**.

The source code file **main.rs** has been initialized by **cargo** to a sample hello world program.



NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

2.3. BUILDING A PROJECT

To build a Rust project on the command line, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo build'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo build
```

This resolves all dependencies of the project, downloads the missing dependencies, and compiles the project using the **rustc** compiler.

By default, the project is built and compiled in debug mode. To build the project in release mode, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo build --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo build --release
```

Example 2.2. Building a Project using cargo

This example assumes that you have successfully created the Rust project **helloworld** according to Example 2.1, “Creating a Project using cargo” .

Change to the directory **helloworld** and build the project:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo build'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo build
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
```

Examine the result:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
    └── debug
        ├── build
        ├── deps
```

```
│   └── helloworld-b7c6fab39c2d17a7
├── examples
├── helloworld
├── helloworld.d
├── incremental
└── native

8 directories, 6 files
```

A subdirectory structure has been created, starting with the directory **target**. Since the project was built in debug mode, the actual build output is contained in a further subdirectory **debug**. The actual resulting executable file is **target/debug/helloworld**.

NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

2.4. CHECKING A PROGRAM

To verify that a Rust program managed by **cargo** can be built, on the command line change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo check'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo check
```

The **cargo check** command is faster than a full project build using the **cargo build** command, because it does not generate the executable code. Therefore, prefer using **cargo check** for verification of Rust program validity when you do not need the executable code.

By default, the project is checked in debug mode. To check the project in release mode, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo check --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo check --release
```

Example 2.3. Checking a Program with cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, “Building a Project using cargo” .

Change to the directory **helloworld** and check the project:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo check'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.5 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo check
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.5 secs
```

The project is checked, with output similar to that of the **cargo build** command. However, the executable file is not generated. You can verify this by comparing the current time with the time stamp of the executable file:

```
$ date
Fri Oct 13 08:53:21 CEST 2017
$ ls -l target/debug/helloworld
-rwxrwxr-x. 2 vslavik vslavik 252624 Oct 13 08:48 target/debug/helloworld
```

2.5. RUNNING A PROGRAM

To run a Rust program managed as a project by **cargo** on the command line, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo run'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run
```

If the program has not been built yet, **cargo** will run a build before running the program.

Using **cargo** to run a Rust program during development is preferred, because it will correctly resolve the output path independent of the build mode.

By default, the project is built in debug mode. To build the project in release mode before running, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo run --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run --release
```

Example 2.4. Running a Program with cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, "Building a Project using cargo" .

Change to the directory **helloworld** and run the project:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo run'
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/helloworld
Hello, world!
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running target/debug/helloworld
Hello, world!
```

cargo first rebuilds the project, and then runs the resulting executable file.

Note that in this example, there were no changes to the source code since last build. As a result, **cargo** did not have to rebuild the executable file, but merely accepted it as current.

2.6. RUNNING PROJECT TESTS

To run tests for a **cargo** project on the command line, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo test'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo test
```

By default, the project is tested in debug mode. To test the project in release mode, run the **cargo** tool with the **--release** option as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo test --release'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo test --release
```

Example 2.5. Testing a Project with cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, “Building a Project using cargo” .

Change to the directory **helloworld**, and edit the file **src/main.rs** so that it contains the following source code:

```
fn main() {
    println!("Hello, world!");
}

#[test]
fn my_test() {
    assert_eq!(21+21, 42);
}
```

The function **my_test** marked as a test has been added.

Save the file, and run the test:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo test'
   Compiling helloworld v0.1.0 (file:///home/vslavik/Documentation/rusttest/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
     Running target/debug/deps/helloworld-9dd6b83647b49aec

running 1 test
test my_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

- For Red Hat Enterprise Linux 8:

```
$ cargo test
   Compiling helloworld v0.1.0 (file:///home/vslavik/Documentation/rusttest/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.26 secs
     Running target/debug/deps/helloworld-9dd6b83647b49aec

running 1 test
test my_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

cargo first rebuilds the project, and then runs the tests found in the project. The test **my_test** has been succesfully passed.

2.7. CONFIGURING PROJECT DEPENDENCIES

To specify dependencies for a **cargo** project, edit the file **Cargo.toml** in the project directory. The section **[dependencies]** contains a list of the project’s dependencies. Each dependency is listed on a new line in the following format:

```
crate_name = version
```

Rust code packages are called crates.

Example 2.6. Adding Dependency to a Project and Building it with cargo

This example assumes that you have successfully built the Rust project **helloworld** according to Example 2.2, "Building a Project using cargo" .

Change to the directory **helloworld** and edit the file **src/main.rs** so that it contains the following source code:

```
extern crate time;

fn main() {
    println!("Hello, world!");
    println!("Time is: {}", time::now().rfc822());
}
```

The code now requires an external crate **time**. Add this dependency to project configuration by editing the file **Cargo.toml** so that it contains the following code:

```
[package]
name = "helloworld"
version = "0.1.0"
authors = ["Your Name <yourname@example.com>"]

[dependencies]
time = "0.1"
```

Finally, run the **cargo run** command to build the project and run the resulting executable file:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo run'
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading time v0.1.38
 Downloading libc v0.2.32
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/helloworld`
Hello, world!
Time is: Fri, 13 Oct 2017 11:08:57
```

- For Red Hat Enterprise Linux 8:

```
$ cargo run
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading time v0.1.38
 Downloading libc v0.2.32
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/helloworld`
Hello, world!
Time is: Fri, 13 Oct 2017 11:08:57
```

cargo downloads the **time** crate and its dependencies (crate **libc**), stores them locally, builds all of the project source code including the dependency crates, and finally runs the resulting executable.

Additional Resources

- Specifying Dependencies – official **cargo** documentation.

2.8. BUILDING PROJECT DOCUMENTATION

Rust code can contain comments marked for extraction into documentation. These comments support the Markdown language. To build project documentation using the **cargo** tool, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo doc --no-deps'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo doc --no-deps
```

This extracts documentation stored from the special comments in the source code of your project and writes the documentation in HTML format.

Note that the **cargo doc** command extracts documentation comments only for public functions, variables and members.

- To include dependencies in the generated documentation, including third party libraries, omit the **--no-deps** option.
- To show the generated documentation in your browser, add the **--open** option.

The command **cargo doc** uses the **rustdoc** utility. Using **cargo doc** is preferred to **rustdoc**.

Example 2.7. Building Project Documentation

This example assumes that you have successfully built the Rust project **helloworld** with dependencies, according to Example 2.6, "Adding Dependency to a Project and Building it with cargo".

Change to the directory **helloworld** and edit the file **src/main.rs** so that it contains the following source code:

```
//! This is a hello-world program.
extern crate time;

/// Prints a greeting to `stdout`.
pub fn print_output() {
    println!("Hello, world!");
    println!("Time is: {}", time::now().rfc822());
}

/// The program entry point.
fn main() {
    print_output();
}
```

The code now contains a public function **print_output()**. The whole **helloworld** program, the **print_output()** function, and the **main()** function have documentation comments.

Run the **cargo doc** command to build the project documentation:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo doc --no-deps'
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo doc --no-deps
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
```

Examine the result:

```
$ tree
.
├── Cargo.lock
├── Cargo.toml
├── src
│   └── main.rs
└── target
...
    └── doc
...
        ├── helloworld
        │   ├── fn.print_output.html
        │   ├── index.html
        │   ├── print_output.v.html
        │   └── sidebar-items.js
...
        └── src
            └── helloworld
                └── main.rs.html

12 directories, 32 files
```

cargo builds the project documentation. To actually view the documentation, open the file **target/doc/helloworld/index.html** in your browser.

Note that the generated documentation does not contain any mention of the **main()** function, because it is not public.

Finally, run the **cargo doc** command without the **--no-deps** option to build the project documentation, including the dependency libraries **time** and **libc**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo doc'
 Documenting libc v0.2.32
 Documenting time v0.1.38
```

```
   Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
     Finished dev [unoptimized + debuginfo] target(s) in 3.41 secs
```

- For Red Hat Enterprise Linux 8:

```
$ cargo doc
 Documenting libc v0.2.32
 Documenting time v0.1.38
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
   Finished dev [unoptimized + debuginfo] target(s) in 3.41 secs
```

Examine the result:

```
$ tree
...
92 directories, 11804 files
$ ls -d target/doc/*/
target/doc/helloworld/  target/doc/implementors/  target/doc/libc/  target/doc/src/  target/doc/time/
```

The resulting documentation now covers also the dependency libraries **time** and **libc**, with each present as another subdirectory in the **target/doc/** directory.

NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

Additional Resources

A detailed description of the **cargo doc** tool and its features is beyond the scope of this book. For more information, see the resources listed below.

- Documentation – The official book The Rust Programming Language has a section on documentationin the first edition.

2.9. VENDORING PROJECT DEPENDENCIES

Vendoring project dependencies means creating a local copy of the dependencies for offline redistribution and reuse. Vendored dependencies can be used by the **cargo** build tool without any connection to the internet.

The **cargo vendor** command for vendoring dependencies is supplied by the **cargo** plug-in **cargo-vendor**.

To install **cargo-vendor 0.1.22**:

- For Red Hat Enterprise Linux 7:

```
# yum install rust-toolset-1.31-cargo-vendor
```

- For Red Hat Enterprise Linux 8:

```
# dnf install cargo-vendor
```

To vendor dependencies for a **cargo** project, change to the project directory and run the **cargo** tool as follows:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo vendor'
```

- For Red Hat Enterprise Linux 8:

```
$ cargo vendor
```

This creates a directory **vendor** and downloads sources of all dependencies to this directory. Additional configuration steps are printed to command line.

The **cargo vendor** command gathers the dependencies for a platform-independent result. Dependency crates for all potential target platforms are downloaded.

IMPORTANT

The **cargo vendor** command is an experimental, unofficial plug-in for the **cargo** tool.

Example 2.8. Vendoring Project Dependencies

This example assumes that you have successfully built the Rust project **helloworld** with dependencies, according to Example 2.6, "Adding Dependency to a Project and Building it with cargo".

Change to the directory **helloworld** and run the **cargo vendor** command to vendor the project with dependencies:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'cargo vendor'
 Downloading kernel32-sys v0.2.2
 Downloading redox_syscall v0.1.31
 Downloading winapi-build v0.1.1
 Downloading winapi v0.2.8
   Vendoring kernel32-sys v0.2.2 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/kernel32-sys-0.2.2) to vendor/kernel32-sys
   Vendoring libc v0.2.32 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/libc-0.2.32) to vendor/libc
   Vendoring redox_syscall v0.1.31 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/redox_syscall-0.1.31) to vendor/redox_syscall
   Vendoring time v0.1.38 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/time-0.1.38) to vendor/time
   Vendoring winapi v0.2.8 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/winapi-0.2.8) to vendor/winapi
   Vendoring winapi-build v0.1.1 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/winapi-build-0.1.1) to vendor/winapi-build
To use vendored sources, add this to your .cargo/config for this project:

    [source.crates-io]
```

```
replace-with = "vendored-sources"

[source.vendored-sources]
directory = "/home/vslavik/helloworld/vendor"
```

- For Red Hat Enterprise Linux 8:

```
$ cargo vendor
```

Examine the result:

```
$ ls
Cargo.lock  Cargo.toml  src  target  vendor
$ tree vendor
vendor
├── kernel32-sys
│   ├── build.rs
│   ├── Cargo.toml
│   ├── README.md
│   └── src
│       └── lib.rs
├── libc
│   ├── appveyor.yml
│   ├── Cargo.toml
...
75 directories, 319 files
```

The **vendor** directory contains copies of all the dependency crates needed to build the **helloworld** program. Note that the crates for building the project on the Windows operating system have been vendored, too, despite running this command on Red Hat Enterprise Linux.

NOTE

The **tree** tool is available from the default Red Hat Enterprise Linux repositories. To install it:

```
# yum install tree
```

2.10. ADDITIONAL RESOURCES

A detailed description of the **cargo** tool and its features is beyond the scope of this book. For more information, see the resources listed below.

Installed Documentation

- *cargo*(1) — The manual page for the **cargo** tool provides detailed information on its usage. To display the manual page for the version included in Rust Toolset:
  - For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'man cargo'
```

- For Red Hat Enterprise Linux 8:

```
$ man cargo
```

- *Cargo, Rust's Package Manager* — Documentation on the **cargo** tool can be optionally installed:

```
# yum install rust-toolset-1.31-cargo-doc
```

Once installed, HTML documentation is available at **/opt/rh/rust-toolset-1.31/root/usr/share/doc/cargo/html/index.html**.

Online Documentation

- Cargo Guide — The cargo tool documentation provides detailed information on **cargo**'s usage.

See Also

- Chapter 1, *Rust Toolset* — An overview of Rust Toolset and more information on how to install it on your system.

CHAPTER 3. RUSTFMT

The **rustfmt** tool provides automatic formatting of Rust source code.

Rust Toolset is distributed with **rustfmt 1.0.0**.

3.1. INSTALLING RUSTFMT

On Red Hat Enterprise Linux 7, the **rustfmt** tool is provided by the **rust-toolset-1.31-rustfmt-preview** package. To install it:

```
# yum install rust-toolset-1.31-rustfmt-preview
```

3.2. USING RUSTFMT AS A STANDALONE TOOL

To format a rust source file and all its dependencies with the **rustfmt** tool:

- For Red Hat Enterprise Linux 7:

  ```
  $ scl enable rust-toolset-1.31 'rustfmt source-file'
  ```

- For Red Hat Enterprise Linux 8:

  ```
  $ rustfmt source-file
  ```

Replace *source-file* with path to the source file.

By default, **rustfmt** modifies the affected files in place without displaying details or creating backups. To change the behavior, use the **--write-mode *value*** option. For further details see the help mesage of **rustfmt**:

- For Red Hat Enterprise Linux 7:

  ```
  $ scl enable rust-toolset-1.31 'rustfmt --help'
  ```

- For Red Hat Enterprise Linux 8:

  ```
  $ rustfmt --help
  ```

Additionally, **rustfmt** accepts standard input instead of a file and provides its output in standard output.

3.3. USING RUSTFMT WITH CARGO

To format all source files in a cargo crate:

- For Red Hat Enterprise Linux 7:

  ```
  $ scl enable rust-toolset-1.31 'cargo fmt'
  ```

- For Red Hat Enterprise Linux 8:

```
$ cargo fmt
```

To change the **rustfmt** formatting options, create the configuration file **rustfmt.toml** in the project directory and supply the configuration there. For further details see the help message of **rustfmt**:

- For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'rustfmt --config-help'
```

- For Red Hat Enterprise Linux 8:

```
$ rustfmt --config-help
```

3.4. ADDITIONAL RESOURCES

- Help messages of **rustfmt**:
  - For Red Hat Enterprise Linux 7:

```
$ scl enable rust-toolset-1.31 'rustfmt --help'
$ scl enable rust-toolset-1.31 'rustfmt --config-help'
```

  - For Red Hat Enterprise Linux 8:

```
$ rustfmt --help
$ rustfmt --config-help
```

- The file **Configurations.md** installed under **/opt/rh/rust-toolset-1.31/root/usr/share/doc/rust-toolset-1.31-rustfmt-preview-0.8.2/Configurations.md**

CHAPTER 4. CONTAINER IMAGE WITH RUST TOOLSET FOR RHEL 7

The Rust Toolset is available as a container image which can be downloaded from Red Hat Container Registry.

4.1. IMAGE CONTENTS

The **devtools/rust-toolset-1.31-rhel7** image provides content corresponding to the following packages:

Component	Version	Package
Rust	1.31.1	rust-toolset-1.31-rust
Cargo	1.31.0	rust-toolset-1.31-cargo
Vendor plug-in for Cargo	0.1.22	rust-toolset-1.31-cargo-vendor

4.2. ACCESS TO THE IMAGE

To pull the **devtools/rust-toolset-1.31-rhel7** image, run the following command as **root**:

```
# podman pull registry.access.redhat.com/devtools/rust-toolset-1.31-rhel7
```

4.3. ADDITIONAL RESOURCES

- Rust Toolset 1.31.1 – entry in the Red Hat Container Catalog
- Using Red Hat Software Collections Container Images

CHAPTER 5. CHANGES IN RUST TOOLSET IN RED HAT DEVELOPER TOOLS 2019.1

This chapter lists some notable changes in Rust Toolset since its previous release.

5.1. RUST

Rust has been updated from version **1.29.0** to **1.31.1**. Notable changes include:

- New capabilities with defining procedural macros
  - Attribute macros let you define custom **#[name]** annotations.
  - Function macros work like those defined by **macro_rules!**, but have more flexibility being implemented in Rust.
  - Macros can now be imported in **use** statements, removing the need for the **#[macro_use]** crate attribute.
  - The **proc_macro** crate is now stable, to help write these new macros.
- Module improvements
  - External crates are now in the prelude, which allows a crate name to serve as the root of a path from anywhere.
  - The **crate** keyword now serves as the root of your own crate in paths and **use** statements.
- 2018 edition
  - The new 2018 edition marks a collective milestone of the last 3 years of Rust development, while also making a few **opt-in** breaking changes. Existing code will default to 2015 edition, with no breaking changes, and crates from different editions are fully interoperable. **cargo new** will specify **edition = "2018"** in **Cargo.toml** for new projects.
  - **async**, **await**, and **try** are reserved keywords in 2018, and **dyn** is now a strict keyword.
  - Non-lexical lifetimes are a refinement of the previous block-based lifetime system, allowing borrowed values to be released sooner in many cases to be reused elsewhere. This is initially exclusive to the 2018 edition, but planned for 2015 as well.
  - Module changes: Explicit **extern crate** declarations are unnecessary in most cases in 2018. **use** paths can now be relative from the current scope, rather than always starting from the root scope as in 2015.
- Lifetimes can now be left implicit in more cases, especially using the new **'_** placeholder.
- **const fn** – Functions can be declared constant, which allows them to be used in restricted contexts, like the initialization of a **const** or **static** value.
- Stable tools: **clippy**, **rls**, and **rustfmt**. We have been shipping these tools as preview already, but now they are officially supported.
  - **clippy** adds extra lints for code/style issues.
  - **rls** implements the Language Server protocol for IDE integration.

- **rustfmt** formats your code, also integrated with the **cargo fmt** subcommand.

- Tool lints allow you to add warning annotations for custom lints, especially for those added by **clippy**. For example, **#[allow(clippy::bool_comparison)]** will silence that warning on an item for which you deem it acceptable.

5.2. CARGO

The **cargo** tool has been updated from version 1.29.0 to 1.31.0. Notable changes include:

- Cargo now shows a progress bar as it builds your crates and dependencies.

- Cargo now allows renaming dependencies in **Cargo.toml**, affecting how they are referenced in your sources. Previously, you could only rename in source like **extern crate published_name as new_name;**.