Red Hat Developer Tools 2018.2

Using Rust Toolset

Installing and Using Rust Toolset

Last Updated: 2018-04-26

Red Hat Developer Tools 2018.2 Using Rust Toolset

Installing and Using Rust Toolset

Vladimír Slávik
vslavik@redhat.com

Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

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 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 8
2.5. RUNNING A PROGRAM 9
2.6. RUNNING PROJECT TESTS 10
2.7. CONFIGURING PROJECT DEPENDENCIES 11
Additional Resources 11
2.8. BUILDING PROJECT DOCUMENTATION 12
Additional Resources 13
2.9. VENDORING PROJECT DEPENDENCIES 14
2.10. ADDITIONAL RESOURCES 15
Installed Documentation 15
Online Documentation 16
See Also 16

CHAPTER 3. RUSTFMT 17
3.1. INSTALLING RUSTFMT 17
3.2. USING RUSTFMT AS A STANDALONE TOOL 17
3.3. USING RUSTFMT WITH CARGO 17
3.4. ADDITIONAL RESOURCES 17

CHAPTER 4. CONTAINER IMAGE 19
4.1. IMAGE CONTENTS 19
4.2. ACCESS TO THE IMAGE 19
4.3. ADDITIONAL RESOURCES 19

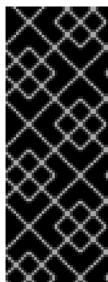
CHAPTER 5. CHANGES IN RUST TOOLSET IN RED HAT DEVELOPER TOOLS 2018.2 20
5.1. RUST 20
5.2. CARGO 20
5.3. CONTAINER IMAGE 21

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, available as a Technology Preview. It provides the Rust programming language compiler **rustc**, the **cargo** build tool and dependency manager, the **cargo-vendor** plugin, and required libraries.

Rust Toolset is distributed as a part of Red Hat Developer Tools for Red Hat Enterprise Linux 7.



IMPORTANT

Rust Toolset is available as a Technology Preview. See the Technology Preview Features Support Scope for more details.

Libraries in Rust Toolset provide no ABI compatibility with past or future releases.

Customers deploying Rust Toolset are encouraged to provide feedback to Red Hat.

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

Table 1.1. Rust Toolset Components

Package	Version	Description
rust	1.25.0	A Rust compiler front-end for LLVM.
cargo	0.26.0	A build system and dependency manager for Rust.
cargo-vendor	0.1.13	A cargo subcommand to vendor crates.io dependencies.

1.2. COMPATIBILITY

Rust Toolset is available for Red Hat Enterprise Linux versions 7.3 and 7.4 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

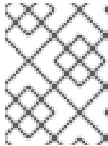
Rust Toolset is an offering that is distributed as a part of the Red Hat Developer Tools content set,

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 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” .

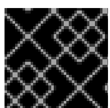
Additional Resources

- 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 your system. For detailed instructions on how to associate your system with an appropriate subscription and get access to Rust Toolset, see Section 1.3, “Getting Access to Rust Toolset”.



IMPORTANT

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

To install all components that are included in Rust Toolset, install the **rust-toolset-7** package:

```
# yum install rust-toolset-7
```

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-7-rust-doc** installs the *The Rust Programming Language* book and API documentation in HTML format to **`/opt/rh/rust-toolset-7/root/usr/share/doc/rust/html/index.html`**.

Online Documentation

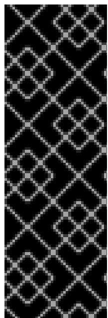
- Rust documentation — The upstream Rust documentation.
- Rust documentation overview — An extended overview of documentation related to Rust.

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 0.26.0**.



IMPORTANT

Rust Toolset and **cargo** are available as a Technology Preview. See the Technology Preview Features Support Scope for more details.

Libraries used by **cargo** and **rustc** in Rust Toolset provide no ABI compatibility with past or future releases.

Customers deploying Rust Toolset are encouraged to provide feedback to Red Hat.

2.1. INSTALLING CARGO

In Rust Toolset, **cargo** is provided by the **rust-toolset-7-cargo** package and is automatically installed with the **rust-toolset-7** package. 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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 'cargo new --lib project_name'
```

Note that you can execute any command using the **scl** utility, 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-7 'bash'
```

Example 2.1. Creating a Project using cargo

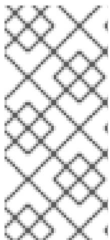
Create a new Rust project called **helloworld** and examine the result:

```
$ scl enable rust-toolset-7 'cargo new --bin helloworld'
     Created binary (application) helloworld project
$ 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:

```
$ scl enable rust-toolset-7 '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 build and compiled in debug mode. To build the project in release mode, run the **cargo** tool with the **--release** option as follows:

```
$ scl enable rust-toolset-7 'cargo build --release'
```

Example 2.2. Building a Project using cargo

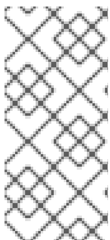
This example assumes that you have successfully created the Rust project **helloworld** according to Example 2.1, "Creating a Project using cargo" .

Change to the directory **helloworld**, build the project, and examine the result:

```
$ scl enable rust-toolset-7 'cargo build'
   Compiling helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.51 secs
$ 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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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 and examine the result:

```
$ scl enable rust-toolset-7 'cargo doc --no-deps'
 Documenting helloworld v0.1.0 (file:///home/vslavik/helloworld)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
$ 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
```

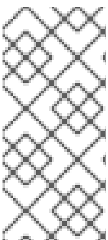
cargo builds the project documentation. To actually view the documentation, open the file **`target/doc/helloworld/index.html`** in your browser.

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**, and examine the result:

```
$ scl enable rust-toolset-7 '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
$ 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** plugin **cargo-vendor**. Rust Toolset is distributed with **cargo-vendor 0.1.13**. To install **cargo-vendor**:

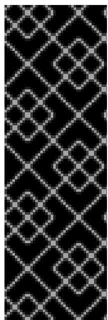
```
# yum install rust-toolset-7-cargo-vendor
```

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

```
$ scl enable rust-toolset-7 '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

Rust Toolset and **cargo** are available as a Technology Preview. See the Technology Preview Features Support Scope for more details.

Additionally to the Technology Preview status, the **cargo vendor** command is an experimental unofficial plugin for the **cargo** tool.

Customers deploying Rust Toolset are encouraged to provide feedback to Red Hat.

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**, run the **cargo vendor** command to vendor the project with dependencies and examine the result:

```
$ scl enable rust-toolset-7 '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
```

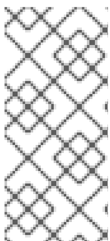
```
   Vendoring redox_syscall v0.1.31
(/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/redox_syscall-0.1.31) to vendor/redox_syscall
   Vendoring time v0.1.38 (/home/vslavik/.cargo/registry/src/github.com-
1ecc6299db9ec823/time-0.1.38) to vendor/time
   Vendoring winapi v0.2.8
(/home/vslavik/.cargo/registry/src/github.com-1ecc6299db9ec823/winapi-
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"

$ 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:

```
$ scl enable rust-toolset-7 'man cargo'
```

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

```
# yum install rust-toolset-7-cargo-doc
```

  Once installed, HTML documentation is available at **/opt/rh/rust-toolset-7/root/usr/share/doc/cargo/html/index.html**.

Online Documentation

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

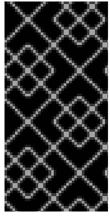
See Also

- 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 0.3.8**.



IMPORTANT

Rust Toolset and **rustfmt** are available as a Technology Preview. See the Technology Preview Features Support Scope for more details.

Customers deploying Rust Toolset are encouraged to provide feedback to Red Hat.

3.1. INSTALLING RUSTFMT

The `rustfmt` tool is provided by the **rust-toolset-7-rustfmt-preview** package. To install it:

```
# yum install rust-toolset-7-rustfmt-preview
```

3.2. USING RUSTFMT AS A STANDALONE TOOL

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

```
$ scl enable rust-toolset-7 '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`:

```
$ scl enable rust-toolset-7 '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:

```
$ scl enable rust-toolset-7 '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`:

```
$ scl enable rust-toolset-7 'rustfmt --config-help'
```

3.4. ADDITIONAL RESOURCES

- Help messages of **rustfmt**:

```
$ scl enable rust-toolset-7 'rustfmt --help'
$ scl enable rust-toolset-7 'rustfmt --config-help'
```

- The file **Configurations.md** installed under **/opt/rh/rust-toolset-7/root/usr/share/doc/rust-toolset-7-rustfmt-preview-0.3.8/Configurations.md**

CHAPTER 4. CONTAINER IMAGE

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

4.1. IMAGE CONTENTS

The **devtools/rust-toolset-7-rhel7** image provides content corresponding to the following packages:

Component	Version	Package
`Rust`	1.25.0	rust-toolset-7-rust
`Cargo`	0.26.0	rust-toolset-7-cargo
`Vendor plugin for Cargo`	0.1.13	rust-toolset-7-cargo-vendor

4.2. ACCESS TO THE IMAGE

To pull the **devtools/rust-toolset-7-rhel7** image, run the following command as **root**:

```
# docker pull registry.access.redhat.com/devtools/rust-toolset-7-rhel7
```

4.3. ADDITIONAL RESOURCES

- Rust Toolset 7 — 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 2018.2

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

5.1. RUST

Rust has been updated from version **1.22.1** to **1.25.0**. Notable changes include:

- Incremental compilation has been added to the Rust compiler. The compiler can reuse artifacts from previous builds and rebuild only the neccessary parts of code.
- The default amount of code generation units for compilation has been changed to 16.
- The methods from the `AsciiExt` trait are now implemented directly in the string and character types.
  Note that this change may cause `unused_imports` warnings to appear for code bases using the `AsciiExt` trait.
- Several library functions such as `mem::size_of()` can be used in const expressions.
- Structs can be aligned using the `#[repr(align(x))]` attribute.
- The `std::ptr::NonNull<T>` type has been added to the standard library for holding a non-null covariant pointer.
- The `rustfmt` code formatting tool has been added. For more information, see Chapter 3, *rustfmt*.

Additionally, the following bugs have been fixed:

- Previous changes in the **glibc** library caused Rust to not recognize stack guard code. As a consequence, when a stack overflow happened in a Rust thread, Rust terminated with a segmentation fault signal `SIGSEGV` instead of an abort signal `SIGABRT`. Additionally, the error message "*thread '{}' has overflowed its stack*" was not displayed. Rust has been updated to recognize the stack guard correctly and terminates again with the correct signal and error message. (BZ#1540329)

5.2. CARGO

The `cargo` tool has been updated from version **0.23.0** to **0.26.0**. Notable changes include:

- Support for unit tests has been added to the `cargo check` command.
- Support for selecting specific versions has been added to the `cargo install` command.
- Support for removing multiple packages at once has been added to the `cargo uninstall` command.
- Debug builds enable incremental compilation by default.
- The `cargo new` command now defaults to creating a project for a binary instead of a library. Additionally, it no longer mangles supplied project names when they include substrings related to Rust.

5.3. CONTAINER IMAGE

Notable changes include:

- Source-to-Image (S2I) support has been added to the **rust-toolset-7-rhel7** container image. As a result, S2I can be used to build Rust application containers. (BZ#1535050)