Red Hat

Develop

Red Hat Developer Tools

November 11, 2026

Contents

1 Develop 3

Build and maintain Go applications 3

Compile Go programs 3

Prerequisites 3

Format Go source with `gofmt` 6

Prerequisites 7

Detect data races in Go programs 9

Prerequisites 9

Container images with Go Toolset 9

Build and maintain C and C++ applications with LLVM 12

Compile C and C++ with Clang 12

Prerequisites 13

Debug C and C++ with LLDB 15

Prerequisites 15

Container images with LLVM Toolset 19

Build and maintain Rust applications 20

Build and test Rust applications 20

Prerequisites 20

Format Rust source with `rustfmt` 25

Container images with Rust Toolset 27

Manage Rust project dependencies 28

Prerequisites 28

Generate Rust API documentation 30

Prerequisites 30

1 Develop

Build and maintain Go applications

Build and ship Go applications on Red Hat Enterprise Linux by using Go Toolset, a Red Hat-supported toolchain that provides the Go compiler, `gofmt` for consistent formatting, the race detector for concurrency testing, and container images for portable deployments.

Compile Go programs

Compile Go source code into executables and libraries on Red Hat Enterprise Linux with the Go compiler in Go Toolset by using a Red Hat-supported toolchain for building command-line tools, services, and shared libraries.

Prerequisites

- Go Toolset is installed.

Related information
Install Go Toolset
Official Go documentation

Configure a Go workspace

Modern Go projects are built using modules. You can start with a single module and then optionally group multiple modules into a workspace to work on them simultaneously.

Procedure

1. Create a root directory for your projects, for example:

```
$ mkdir ~/go-projects/
```

2. Change into the project directory:

```
$ cd ~/go-projects/
```

3. Initialize a module:
    a. Create a directory for your module:

    ```
    $ mkdir <module_name>
    ```

    b. Change into the module's directory:

    ```
    $ cd <module_name>
    ```

    c. Initialize the module:

    ```
    $ go mod init <module_name>
    ```

    This command creates a single-module project.

    If you want to create multiple modules, repeat this step for every module.
4. If you want to work on multiple modules at the same time, create a multi-module workspace:
    a. Change into the project directory:

    ```
    $ cd ~/go-projects/
    ```

    b. Initialize a workspace to include multiple modules:

    ```
    $ go work init <module_name_1><module_name_n> ...
    ```

Related information
[The official Go workspaces documentation]

Compile a Go program

You can compile Go sources in the current directory by running `go build` to produce an executable binary.

Before you begin

- A Go workspace with configured modules.
- Compile the Go sources in the current directory:

```
$ go build .
```

Related information
Configure a Go workspace

Run a Go program

The Go compiler creates an executable binary file as a result of compiling. Complete the following steps to run your program.

- Use one of the following options to run your Go program:
  - To run a compiled Go program, enter:

    ```
    $ ./<file_name>
    ```

    Replace `<file_name>` with the name of your executable file.
  - To compile the Go sources in the current directory and run the program in a single step, enter:

    ```
    $ go run .
    ```

Install compiled Go projects

You can download and install third-party Go projects from online resources to use their executable files and libraries in further Go projects. After installation, Go copies the project's executables and libraries to workspace directories and installs its dependencies as well.

Before you begin

- A Go workspace with configured modules.
- Install a Go project:

```
$ go install <go_project>
```

Related information
Configure a Go workspace

Download and install Go projects

Use `go install` with a third-party import path to download and install Go commands and libraries into your workspace. Optionally, run `go help importpath` for details on how to specify remote projects.

Before you begin

- A Go workspace with configured modules.

Procedure

1. To download and install a Go project, enter:

```
$ go install <third_party_go_project>
```

2. Optional: For information on possible values of third-party projects, enter:

```
$ go help importpath
```

Related information
Configure a Go workspace

Format Go source with `gofmt`

Enforce consistent Go code layout on Red Hat Enterprise Linux by using `gofmt`, the formatting tool in Go Toolset. The `gofmt` utility applies standard Go formatting rules so teams can share code without maintaining separate style guides.

Prerequisites

- Go Toolset is installed.

Related information
Install Go Toolset
Official documentation

Format code

You can use the `gofmt` formatting tool to format code in a given path. When the path leads to a single file, the changes apply only to the file. When the path leads to a directory, all `.go` files in the directory are processed.

- To format your code in a given path, enter:

```
$ gofmt -w <code_path>
```

Replace `<code_path>` with the path to the code you want to format.

NOTE:
To print the formatted code to standard output instead of writing it to the original file, omit the `-w` option.

Preview changes to code

You can use the `gofmt` formatting tool to preview changes done by formatting code in a given path. The output in unified diff format is printed to standard output.

- Show differences in your code in a given path:

```
$ gofmt -d <code_path>
```

Replace `<code_path>` with the path to the code you want to compare.

Simplify code

You can use the `gofmt` formatting tool to simplify your code.

Procedure

1. To simplify code in a given path, enter:

```
$ gofmt -s -w <code_path>
```

   Replace `<code_path>` with the path to the code you want to simplify.

2. To apply the changes, enter:

```
$ gofmt -w <code_path>
```

   Replace `<code_path>` with the path to the code you want to format.

Refactor code

You can use the `gofmt` formatting tool to refactor your code by applying arbitrary substitutions.

Procedure

1. To refactor your code in a given path, enter:

```
# gofmt -r -w <rewrite_rule><code_path>
```

   Replace `<code_path>` with the path to the code you want to refactor and `<rewrite_rule>` with the rule you want it to be rewritten by.

2. To apply the changes, enter:

```
# gofmt -w <code_path>
```

   Replace `<code_path>` with the path to the code you want to format.

Detect data races in Go programs

Find data race conditions in concurrent Go programs on Red Hat Enterprise Linux by using the Go race detector in Go Toolset. The race detector reports unsafe memory access between `goroutines` at runtime. Note that the race detector introduces significant runtime overhead.

Prerequisites

- Go Toolset is installed.

Related information
Install Go Toolset
Official Go race detector documentation

Use the Go race detector

Use the Go race detector to check your code for race conditions.

- Use the race detector:

```
# go build -race -o <output_file><go_main_package>
```

Replace `<output_file>` with the name of your executable file and `<go_main_package>` with the name of the package you want to test.

Container images with Go Toolset

Package Go Toolset into portable container images on Red Hat Enterprise Linux by building on Red Hat Enterprise Linux or Red Hat Universal Base Images (UBI), so you can compile and run Go applications in consistent environments across development, CI, and production.

Related information
Go Toolset Container Images in the Red Hat Container Registry
Working with Container Images
Universal Base Images (UBI): Images, repositories, packages, and source code (Red Hat Knowledgebase)

Go Toolset container image contents

The following table lists the Go component shipped in Red Hat Enterprise Linux Go Toolset container images, its version, and the package name.

Go Toolset component in container images

Component	Version	Package
`Go`	1.26.2	go-toolset-1.26.2

Pull the RHEL-based Go Toolset container image

Pull the container image from the Red Hat registry before running your container and performing actions.

- Pull the required image:
  - For an image based on RHEL 8, enter:

```
# podman pull registry.redhat.io/rhel8/go-toolset
```

  - For an image based on RHEL 9, enter:

```
# podman pull registry.redhat.io/rhel9/go-toolset
```

  - For an image based on RHEL 10, enter:

```
# podman pull registry.redhat.io/rhel10/go-toolset
```

Pull the UBI-based Go Toolset container image

Pull the container image from the Red Hat registry before running your container and performing actions.

- Pull the required image:
  - For an image based on RHEL 8, enter:

```
# podman pull registry.access.redhat.com/ubi8/go-toolset
```

  - For an image based on RHEL 9, enter:

```
# podman pull registry.access.redhat.com/ubi9/go-toolset
```

  - For an image based on RHEL 10, enter:

```
# podman pull registry.access.redhat.com/ubi10/go-toolset
```

Create a custom UBI-based container with Go Toolset

Go Toolset packages are in Red Hat Universal Base Images (UBI) repositories. You can add them to a base UBI container image. To keep the image small, install individual packages instead of the entire Go Toolset.

Before you begin

- An existing container file.

Alternatively, you can install the UBI Go Toolset container image to access Go Toolset.

- To create a container image containing Go Toolset, add the following to your container file.
  - For an image based on RHEL 8, enter:

```
FROM registry.access.redhat.com/ubi8/ubi:latest

RUN yum module install -y go-toolset
```

- For an image based on RHEL 9, enter:

```
FROM registry.access.redhat.com/ubi9/ubi:latest

RUN yum install -y go-toolset
```

- For an image based on RHEL 10, enter:

```
FROM registry.access.redhat.com/ubi10/ubi:latest

RUN yum install -y go-toolset
```

Related information
Pull the UBI-based Go Toolset container image
Dockerfile reference

Build and maintain C and C++ applications with LLVM

Build and ship native C and C++ applications on Red Hat Enterprise Linux by using LLVM Toolset, a Red Hat-supported toolchain that provides Clang for compilation, LLDB for debugging, and UBI-based container images for consistent builds across development and production environments.

Compile C and C++ with Clang

Compile C and C++ source code on Red Hat Enterprise Linux by using Clang, the LLVM compiler front end in LLVM Toolset. Clang 21.1.8 supports modern C and C++ standards and integrates with the LLVM build tools shipped on Red Hat Enterprise Linux.

NOTE:
To compile a C++ program, use `clang++` instead of `clang`.

Prerequisites

- LLVM Toolset is installed.

Related information
Install LLVM Toolset
The official Clang compiler documentation

Compile a source file

You can compile source files and assembly language files with the `clang` command to produce an executable binary. Add the `-g` flag to your `clang` command to include debug information.

NOTE:
To compile a C++ program, use `clang++` instead of `clang`.

- Compile your program:

```
$ clang -g -o <binary_file><source_file>
```

Replace `<binary_file>` with the name of your output file and `<source_file>` with the name of your source file.

Run a program

You produce an executable when you compile your program with `clang` command. Run the program from the directory that contains the executable file.

Before you begin

- Your program is compiled.
- To run your program, enter in the directory containing the executable file:

```
$ ./<binary_file>
```

Replace `<binary_file>` with the name of your executable file.

Related information
Compile a source file

Link object files together

By linking object files together, you can compile only source files that contain changes instead of your entire project.

When a project contains several source files, compile each one to an object file with `clang` command. Link the resulting object files with `clang` to produce an executable.

NOTE:
To compile a C++ program, use `clang++` instead of `clang`.

Procedure

1. Compile a source file to an object file:

```
$ clang -o <object_file> -c <source_file>
```

Replace `<object_file>` with the name of your object file and `<source_file>` with the name of your source file.

2. Link object files together:

```
$ clang -o <output_file><object_file_0><object_file_n>
```

Replace `<output_file>` with the name of your output file and `<object_file>` with the names of the object files you want to link.

IMPORTANT:
At the moment, certain library features are statically linked into applications built with LLVM Toolset to support their execution on multiple versions of Red Hat Enterprise Linux. This creates a small security risk. Red Hat will issue a security erratum in case you need to rebuild your applications due to this risk.

Do not statically link your entire application.

Debug C and C++ with LLDB

Debug C and C++ programs on Red Hat Enterprise Linux from the command line by using LLDB, the LLVM debugger in LLVM Toolset. Use LLDB 21.1.8 to inspect memory, control program execution, and set breakpoints when investigating application behavior.

Prerequisites

- LLVM Toolset is installed.
- Your compiler is configured to create debug information.

Related information
Install LLVM Toolset
Official Clang compiler documentation
Controlling Debug Information (Clang Compiler User's Manual)
Preparing a Program for Debugging (Red Hat Developer Toolset User Guide)

Start a debugging session

Use LLDB to start an interactive debugging session.

Procedure

1. Run LLDB on a program you want to debug:

```
$ lldb <binary_file>
```

Replace `<binary_file>` with the name of your compiled program.

You have started your LLDB debugging session in interactive mode. The command line now displays the default prompt (`lldb`).

2. To quit the debugging session and return to the shell prompt:

```
(lldb) quit
```

Run the program in a debugging session

Use LLDB to run your program during your debugging session. The execution of your program stops when the first breakpoint is reached, when an error occurs, or when the program terminates.

Before you begin

- You started an interactive debugging session with LLDB.
- Run the program you are debugging:

```
(lldb) run
```

  Alternatively, run the program you are debugging by using a specific argument:

```
(lldb) run <argument>
```

  Replace `<argument>` with the command-line argument you want to use.

Related information
Start a debugging session

Use breakpoints

Use breakpoints to pause the execution of your program at a set point in your source code.

Before you begin

- You started an interactive debugging session with LLDB.
- To set a new breakpoint on a specific line, enter:

```
(lldb) breakpoint set --file <source_file_name> --line <line_number>
```

  Replace `<source_file_name>` with the name of your source file and `<line_number>` with the line number you want to set your breakpoint at.
- To set a breakpoint on a specific function, enter:

```
(lldb) breakpoint set --name <function_name>
```

    - Replace `<function_name>` with the name of the function you want to set your breakpoint at.

- To display a list of currently set breakpoints, enter:

```
(lldb) breakpoint list
```

- To delete a breakpoint, run:

```
(lldb) breakpoint clear -f <source_file_name> -l <line_number>
```

    - Replace `<source_file_name>` with the name of your source file and `<line_number>` with line number of the breakpoint you want to delete.

- To resume the execution of your program after it reached a breakpoint, enter:

```
(lldb) continue
```

- To skip a specific number of breakpoints, enter:

```
(lldb) continue -i <breakpoints_to_skip>
```

    - Replace `<breakpoints_to_skip>` with the number of breakpoints you want to skip. To skip a loop, set the `<breakpoints_to_skip>` to match the loop iteration count.

Related information
Start a debugging session

Step through code

You can use LLDB to step through the code of your program to run only one line of code after the line pointer.

Before you begin

- You started an interactive debugging session with LLDB.

Procedure

1. Set your line pointer to the line you want to run.
2. Enter:

```
(lldb) step
```

Related information
Start a debugging session

List source code

Before you run the program you are debugging, the LLDB debugger automatically displays the first 10 lines of source code. Each time execution stops, LLDB displays the line where it stopped and its surrounding lines. You can use LLDB to manually display source code during your debugging session.

Before you begin

- You started an interactive debugging session with LLDB.
- To list the first 10 lines of the source code of the program you are debugging, enter:

```
(lldb) list
```

```
(lldb) list <source_file_name>:<line_number>
```

Replace `<source_file_name>` with the name of your source file and `<line_number>` with the number of the line you want to display.

Related information
Start a debugging session

Display current program data

The LLDB debugger provides data on variables of any complexity, any valid expressions, and function call return values. You can use LLDB to display data relevant to the program state.

Before you begin

- You started an interactive debugging session with LLDB.
- Display the current value of a certain variable, expression, or return value:

```
(lldb) print <data_name>
```

Replace `<data_name>` with data you want to display.

Related information
Start a debugging session

Container images with LLVM Toolset

Package LLVM Toolset into portable container images on Red Hat Enterprise Linux by building on Red Hat Universal Base Images (UBI), so you can compile and debug C and C++ applications in consistent environments across development, CI, and production.

Related information
Working with Container Images
Universal Base Images (UBI): Images, repositories, packages, and source code (Red Hat Knowledgebase)

Create a custom UBI-based container with LLVM Toolset

LLVM Toolset packages are part of the Red Hat Universal Base Images (UBIs) repositories. To keep the container image size small, install only individual packages instead of the entire LLVM Toolset.

Before you begin

- An existing container file. For information on creating Containerfiles, see the Dockerfile reference page.
- To create a container image containing LLVM Toolset, add the following to your container file:
  - For an image based on RHEL 8, enter:

```
FROM registry.access.redhat.com/ubi8/ubi:latest

RUN yum module install -y llvm-toolset
```

- For an image based on RHEL 9, enter:

```
FROM registry.access.redhat.com/ubi9/ubi:latest

RUN yum install -y llvm-toolset
```

- For an image based on RHEL 10, enter:

```
FROM registry.access.redhat.com/ubi10/ubi:latest

RUN yum install -y llvm-toolset
```

Build and maintain Rust applications

Build and ship Rust applications on Red Hat Enterprise Linux by using Rust Toolset, a Red Hat-supported toolchain that provides `rustc`, Cargo, `rustfmt`, dependency management, API documentation, and container images for portable deployments.

Build and test Rust applications

Use the Cargo build tool to create, build, test, and run Rust projects on Red Hat Enterprise Linux, including release builds and WebAssembly binaries.

Prerequisites

- Rust Toolset is installed.

Cargo project layout

The Cargo build tool uses conventions for directory structure and file placement within a Cargo package. Running the `cargo new` command generates the package structure and templates for a manifest and a project file. By default, it initializes a new Git repository in the package root directory.

For a binary program, Cargo creates a directory `<project_name>` containing a text file named `Cargo.toml` and a subdirectory `src` containing a text file named `main.rs`.

Related information
The Cargo Book - Package Layout
The Rust Programming Language - Managing Growing Projects with Packages, Crates, and Modules

Create a Rust project

Create a new Rust project that follows Cargo conventions. For more information on Cargo conventions, see Cargo project layout.

Procedure

1. Create a Rust project:

```
$ cargo new --bin <project_name>
```

   Replace `<project_name>` with your project name.
2. To edit the project code, edit the main executable file `main.rs` and add new source files to the `src` subdirectory.

Related information
Configure Rust project dependencies

Create a Rust library project

Create a Rust library project by using the Cargo build tool (`cargo new --lib`).

Procedure

1. Create a Rust library project:

```
$ cargo new --lib <project_name>
```

   Replace `<project_name>` with the name of your Rust project.
2. To edit the project code, edit the `src/lib.rs` source file.

Related information

Managing Growing Projects with Packages, Crates, and Modules

Build a Rust project

Build your Rust project by using the Cargo build tool. Cargo resolves all dependencies of your project, downloads missing dependencies, and compiles it using the `rustc` compiler.

Before you begin

- An existing Rust project.

By default, projects are built and compiled in debug mode. For information on compiling your project in release mode, see Build a Rust project in release mode.

Procedure

1. In the Rust project directory, build the project:

```
$ cargo build
```

2. To verify that your Rust program can be built when you do not need to build an executable file, enter:

```
$ cargo check
```

Build a Rust project in release mode

Build your Rust project in release mode by using the Cargo build tool. Release mode is optimizing your source code and can therefore increase compilation time while ensuring that the compiled binary will run faster. Use this mode to produce optimized artifacts suitable for release and production.

Before you begin

- An existing Rust project.

Cargo resolves all dependencies of your project, downloads missing dependencies, and compiles it by using the `rustc` compiler.

For information on compiling your project in debug mode, see Build a Rust project.

Procedure

1. In the Rust project directory, build the project in release mode:

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

2. To verify that your Rust program can be built when you do not need to build an executable file, enter:

```
$ cargo check
```

Run a Rust program

In a Cargo project directory, run `cargo run` command to compile when needed and start your binary. The executable you launch matches your active debug or release profile.

Before you begin

- A built Rust project.
- To run a Rust program managed as a project by Cargo, enter in the project directory:

```
$ cargo run
```

If your program has not been built yet, Cargo builds your program before running it.

Test a Rust project

Test your Rust program by using the Cargo build tool. Cargo rebuilds your project and then runs the tests found in the project. By default, Rust projects are tested in debug mode.

Before you begin

- A built Rust project.

By default, Rust projects are tested in debug mode.

Procedure

1. Add the `#[test]` attribute in front of your function.
2. Enter in the project directory:

```
$ cargo test
```

Related information
The Rust Reference – Testing attributes
Test a Rust project in release mode

Test a Rust project in release mode

Test your Rust program in release mode by using the Cargo build tool. Cargo rebuilds your project and then runs the tests found in the project. Release mode optimizes your source code, which can increase compilation time while producing a binary that runs faster.

Before you begin

- A built Rust project.

Cargo first rebuilds your project and then runs the tests found in the project. Test functions must be free, monomorphic, and take no arguments. The function return type must be either `()` or `Result<(), E> where E: Error`.

By default, Rust projects are tested in debug mode.

Procedure

1. Add the `#[test]` attribute in front of your function.
2. Enter in the project directory:

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

Related information
The Rust Reference - Testing attributes
Test a Rust project

Compile code into a WebAssembly binary with Rust

Install the WebAssembly standard library to compile Rust code into WebAssembly binaries.

Before you begin

- Rust Toolset is installed.

Procedure

1. Install the WebAssembly standard library:
   - On RHEL 8, enter:

```
# yum install rust-std-static-wasm32-unknown-unknown
```

   - On RHEL 9 and 10, enter:

```
# dnf install rust-std-static-wasm32-unknown-unknown
```

2. Use WebAssembly with Cargo:

```
$ cargo <command> --target wasm32-unknown-unknown
```

   Replace `<command>` with the Cargo command you want to run.

Related information
Rust and WebAssembly documentation
Rust and WebAssembly book

Format Rust source with `rustfmt`

Enforce consistent Rust code layout on Red Hat Enterprise Linux by using `rustfmt`, the formatting tool in Rust Toolset. Use `rustfmt` as a standalone tool or with Cargo to apply standard Rust formatting rules across your projects.

Install `rustfmt`

Install the `rustfmt` formatting tool by installing the `rustfmt` package (RHEL 8 module or RHEL 9 and 10 package).

Before you begin

- Rust Toolset is installed.

- Install the `rustfmt` package:
  - On RHEL 8, enter:

```
# yum install rustfmt
```

  - On RHEL 9 and 10, enter:

```
# dnf install rustfmt
```

Use `rustfmt` as a standalone tool

Use `rustfmt` as a standalone tool to format a Rust source file and its dependencies, or use it with the Cargo build tool.

Before you begin

- An existing Rust project.
- Format the Rust source code:

```
$ rustfmt <source_file>
```

Replace `<source_file>` with the name of your source file. If you use standard input instead, formatted output is printed to standard output.

IMPORTANT:
By default, `rustfmt` modifies the affected files without displaying details or creating backups. To display details and create backups, run `rustfmt` with the `--write-mode` option.

Use `rustfmt` with the Cargo build tool

Use the `rustfmt` tool with Cargo to format a Rust source file and all its dependencies. As an alternative, use `rustfmt` as a standalone tool. For more information, see Use `rustfmt` as a standalone tool.

Before you begin

- An existing Rust project.

Procedure

1. Optional: To change the `rustfmt` formatting options, create `rustfmt.toml` configuration file in the project directory and add your configurations to the file.
2. Format the Rust source code:

```
$ cargo fmt
```

Container images with Rust Toolset

Package Rust Toolset into portable container images on Red Hat Enterprise Linux by building on Red Hat Universal Base Images (UBI), so you can compile and run Rust applications in consistent environments across development, continuous integration (CI), and production.

Related information
Working with Container Images
Universal Base Images (UBI): Images, repositories, packages, and source code (Red Hat Knowledgebase)

Create a custom UBI-based container with Rust Toolset

Rust Toolset packages are part of the Red Hat Universal Base Images (UBIs) repositories. To keep the container size small, install only individual packages instead of the entire Rust Toolset.

Before you begin

- An existing container file. For information on creating Containerfiles, see the Dockerfile reference page.
- To create a container image containing Rust Toolset, add the following to your container file:
    - For an image based on RHEL 8, enter:

```
FROM registry.access.redhat.com/ubi8/ubi:latest

RUN yum module install -y rust-toolset
```

- For an image based on RHEL 9, enter:

```
FROM registry.access.redhat.com/ubi9/ubi:latest

RUN yum install -y rust-toolset
```

- For an image based on RHEL 10, enter:

```
FROM registry.access.redhat.com/ubi10/ubi:latest

RUN yum install -y rust-toolset
```

Manage Rust project dependencies

Configure and vendor dependencies for Rust projects on Red Hat Enterprise Linux by using the Cargo build tool to control crate versions and offline builds.

Prerequisites

- Rust Toolset is installed.

Configure Rust project dependencies

Configure the dependencies of your Rust project by using the Cargo build tool. Edit the `Cargo.toml` file in the project directory and rebuild your project. Cargo downloads packages and dependencies locally. Use `cargo build` or `cargo run` to compile your code and produce a runnable executable.

Before you begin

- A built Rust project.

Procedure

1. In your project directory, edit the `Cargo.toml` file, and list each dependency in the following format in the `[dependencies]` section:

```
<crate_name> = <version>
```

Rust code packages are called crates.

2. Rebuild your project:

```
$ cargo build
```

3. Run your project:

```
$ cargo run
```

Related information
The Cargo Book - Specifying Dependencies

Vendor Rust project dependencies

Create a local copy of the dependencies of your Rust project for offline redistribution by using the Cargo build tool (vendoring). Vendored sources live in the `vendor` directory, including any Windows-target crates you captured. Point Cargo at that tree to build offline without network access.

Before you begin

- A built Rust project.
- Configured dependencies.
- To vendor your Rust project with dependencies by using Cargo, enter in the project directory:

```
$ cargo vendor
```

Generate Rust API documentation

Use the Cargo tool on Red Hat Enterprise Linux to generate HTML API documentation from comments in your source code that are marked for extraction. Documentation comments are extracted only for public functions, variables, and members.

Prerequisites

- Rust Toolset is installed.

Build documentation for a Rust project

Use the Cargo tool to generate documentation from comments in your source code that are marked for extraction. Note that documentation comments are extracted only for public functions, variables, and members.

Before you begin

- A built Rust project.
- Configured dependencies.

Procedure

1. In your code, use three slashes `///` at the beginning of a line to mark the line for extracting the comment for documentation.
2. Build the documentation:

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

The command stores the generated documentation in the `target/doc/` directory.

Related information
The Rust Programming Language - Making Useful Documentation Comments