Red Hat Enterprise Linux 10

Packaging and distributing software

Packaging software by using the RPM package management system

Last Updated: 2025-05-22

Red Hat Enterprise Linux 10 Packaging and distributing software

Packaging software by using the RPM package management system

Legal Notice

Abstract

Package software into an RPM package by using the RPM package manager. Prepare source code for packaging, package software, and investigate advanced packaging scenarios, such as packaging Python projects or RubyGems packages into RPM packages.

Table of Contents

PROVIDING FEEDBACK ON RED HAT DOCUMENTATION

We appreciate your feedback on our documentation. Let us know how we can improve it.

Submitting feedback through Jira (account required)

1. Log in to the Jira website.
2. Click **Create** in the top navigation bar
3. Enter a descriptive title in the **Summary** field.
4. Enter your suggestion for improvement in the **Description** field. Include links to the relevant parts of the documentation.
5. Click **Create** at the bottom of the dialogue.

CHAPTER 1. INTRODUCTION TO RPM

The RPM Package Manager (RPM) is a package management system that runs on Red Hat Enterprise Linux (RHEL), CentOS, and Fedora. You can use RPM to distribute, manage, and update software that you create for any of these operating systems.

The RPM package management system has the following advantages over distributing software in conventional archive files:

- RPM manages software in the form of packages that you can install, update, or remove independently of each other, which makes the maintenance of an operating system easier.
- RPM simplifies the distribution of software because RPM packages are standalone binary files, similar to compressed archives. These packages are built for a specific operating system and hardware architecture. RPMs contain files such as compiled executables and libraries that are placed into the appropriate paths on the filesystem when the package is installed.

With RPM, you can perform the following tasks:

- Install, upgrade, and remove packaged software.
- Query detailed information about packaged software.
- Verify the integrity of packaged software.
- Build your own packages from software sources and complete build instructions.
- Digitally sign your packages by using the GNU Privacy Guard (GPG) utility.
- Publish your packages in a DNF repository.

In Red Hat Enterprise Linux, RPM is fully integrated into the higher-level package management software, such as DNF or PackageKit. Although RPM provides its own command-line interface, most users need to interact with RPM only through this software. However, when building RPM packages, you must use the RPM utilities such as **rpmbuild(8)**.

1.1. RPM PACKAGES

An RPM package consists of an archive of files and metadata used to install and erase these files. Specifically, the RPM package contains the following parts:

- GPG signature
  The GPG signature is used to verify the integrity of the package.
- Header (package metadata)
  The RPM package manager uses this metadata to determine package dependencies, where to install files, and other information.
- Payload
  The payload is a **cpio** archive that contains files to install to the system.

There are two types of RPM packages. Both types share the file format and tooling, but have different contents and serve different purposes:

- Source RPM (SRPM)

An SRPM contains source code and a **spec** file, which describes how to build the source code into a binary RPM. Optionally, the SRPM can contain patches to source code.

- Binary RPM
  A binary RPM contains the binaries built from the sources and patches.

1.2. LISTING RPM PACKAGING UTILITIES

In addition to the **rpmbuild(8)** program for building packages, RPM provides other utilities to make the process of creating packages easier. You can find these programs in the **rpmdevtools** package.

Prerequisites

- The **rpmdevtools** package has been installed:

```
# dnf install rpmdevtools
```

Procedure

- Use one of the following methods to list RPM packaging utilities:
  - To list certain utilities provided by the **rpmdevtools** package and their short descriptions, enter:

```
$ rpm -qi rpmdevtools
```

  - To list all utilities, enter:

```
$ rpm -ql rpmdevtools | grep ^/usr/bin
```

Additional resources

- RPM utilities man pages on your system

CHAPTER 2. CREATING SOFTWARE FOR RPM PACKAGING

To prepare software for RPM packaging, you must understand what source code is and how to create software.

2.1. WHAT IS SOURCE CODE

Source code is human-readable instructions to the computer that describe how to perform a computation. Source code is expressed by using a programming language.

The following versions of the **Hello World** program written in three different programming languages cover major RPM Package Manager use cases:

- **Hello World** written in Bash
  The *bello* project implements **Hello World** in Bash. The implementation contains only the **bello** shell script. The purpose of this program is to output **Hello World** on the command line.

  The **bello** file has the following contents:

  ```
  #!/bin/bash

  printf "Hello World\n"
  ```

- **Hello World** written in Python
  The *pello* project implements **Hello World** in Python. The implementation contains only the **pello.py** program. The purpose of the program is to output **Hello World** on the command line.

  The **pello.py** file has the following contents:

  ```
  #!/usr/bin/python3

  print("Hello World")
  ```

- **Hello World** written in C
  The *cello* project implements **Hello World** in C. The implementation contains only the **cello.c** and **Makefile** files. The resulting **tar.gz** archive therefore has two files in addition to the **LICENSE** file. The purpose of the program is to output **Hello World** on the command line.

  The **cello.c** file has the following contents:

  ```
  #include <stdio.h>

  int main(void) {
      printf("Hello World\n");
      return 0;
  }
  ```

NOTE

The packaging process is different for each version of the **Hello World** program.

2.2. METHODS OF CREATING SOFTWARE

You can convert the human-readable source code into machine code by using one the following methods:

- Natively compile software.
- Interpret software by using a language interpreter or language virtual machine. You can either raw-interpret or byte-compile software.

2.2.1. Natively compiled software

Natively compiled software is software written in a programming language that compiles to machine code with a resulting binary executable file. Natively compiled software is standalone software.

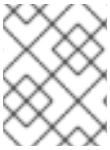
NOTE

Natively compiled RPM packages are architecture-specific.

If you compile such software on a computer that uses a 64-bit (x86_64) AMD or Intel processor, it does not run on a 32-bit (x86) AMD or Intel processor. The resulting package has the architecture specified in its name.

2.2.2. Interpreted software

Some programming languages, such as Bash or Python, do not compile to machine code. Instead, a language interpreter or a language virtual machine executes the programs' source code step-by-step without prior transformations.

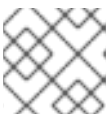


NOTE

Software written entirely in interpreted programming languages is not architecture-specific. Therefore, the resulting RPM package has the **noarch** string in its name.

You can either raw-interpret or byte-compile software written in interpreted languages:

- Raw-interpreted software
  You do not need to compile this type of software. Raw-interpreted software is directly executed by the interpreter.
- Byte-compiled software
  You must first compile this type of software into bytecode, which is then executed by the language virtual machine.



NOTE

Some byte-compiled languages can be either raw-interpreted or byte-compiled.

Note that the way you build and package software by using RPM is different for these two software types.

2.3. BUILDING SOFTWARE FROM SOURCE

During the software building process, the source code is turned into software artifacts that you can package by using RPM.

2.3.1. Building software from natively compiled code

You can build software written in a compiled language into an executable by using one of the following methods:

- Manual building
- Automated building

2.3.1.1. Manually building a sample C program

You can use manual building to build software written in a compiled language.

A sample **Hello World** program written in C (**cello.c**) has the following contents:

```
#include <stdio.h>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

Procedure

1. Invoke the C compiler from the GNU Compiler Collection to compile the source code into binary:

```
$ gcc -g -o cello cello.c
```

2. Run the resulting binary **cello**:

```
$ ./cello
Hello World
```

2.3.1.2. Setting automated building for a sample C program

Large-scale software commonly uses automated building. You can set up automated building by creating the **Makefile** file and then running the GNU **make** utility.

Procedure

1. Create the **Makefile** file with the following content in the same directory as **cello.c**:

```
cello:
 gcc -g -o cello cello.c
clean:
 rm cello
```

   Note that the lines under **cello:** and **clean:** must begin with a tabulation character (tab).

2. Build the software:

```
$ make
make: 'cello' is up to date.
```

3. Because a build is already available in the current directory, enter the **make clean** command, and then enter the **make** command again:

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c
```

Note that trying to build the program again at this point has no effect because the GNU **make** system detects the existing binary:

```
$ make
make: 'cello' is up to date.
```

4. Run the program:

```
$ ./cello
Hello World
```

2.3.2. Interpreting source code

You can convert the source code written in an interpreted programming language into machine code by using one of the following methods:

- Byte-compiling
  The procedure for byte-compiling software varies depending on the following factors:
  - Programming language
  - Language's virtual machine
  - Tools and processes used with that language

NOTE

You can byte-compile software written, for example, in Python. Python software intended for distribution is often byte-compiled, but not in the way described in this document. The described procedure aims not to conform to the community standards, but to be simple. For real-world Python guidelines, see Software Packaging and Distribution.

  You can also raw-interpret Python source code. However, the byte-compiled version is faster. Therefore, RPM packagers prefer to package the byte-compiled version for distribution to end users.

- Raw-interpreting
  Software written in shell scripting languages, such as Bash, is always executed by raw-interpreting.

2.3.2.1. Byte-compiling a sample Python program

By choosing byte-compiling over raw-interpreting of Python source code, you can create faster software.

A sample **Hello World** program written in the Python programming language (**pello.py**) has the following contents:

```
print("Hello World")
```

Procedure

1. Byte-compile the **pello.py** file:

```
$ python -m compileall pello.py
```

2. Verify that a byte-compiled version of the file is created:

```
$ ls __pycache__
pello.cpython-311.pyc
```

Note that the package version in the output might differ depending on which Python version is installed.

3. Run the program in **pello.py**:

```
$ python pello.py
Hello World
```

2.3.2.2. Raw-interpreting a sample Bash program

A sample **Hello World** program written in Bash shell built-in language (**bello**) has the following contents:

```
#!/bin/bash

printf "Hello World\n"
```

NOTE

The **shebang** (**#!**) sign at the top of the **bello** file is not part of the programming language source code.

Use the **shebang** to turn a text file into an executable. The system program loader parses the line containing the **shebang** to get a path to the binary executable, which is then used as the programming language interpreter.

Procedure

1. Make the file with source code executable:

```
$ chmod +x bello
```

2. Run the created file:

```
$ ./bello
Hello World
```

CHAPTER 3. PREPARING SOFTWARE FOR RPM PACKAGING

To prepare a piece of software for packaging with RPM, you can first patch the software, create a **LICENSE** file for it, and archive it as a tarball.

3.1. PATCHING SOFTWARE

When packaging software, you might need to make certain changes to the original source code, such as fixing a bug or changing a configuration file. In RPM packaging, you can instead leave the original source code intact and apply patches on it.

A patch is a piece of text that updates a source code file. The patch has a *diff* format, because it represents the difference between two versions of the text. You can create a patch by using the **diff** utility, and then apply the patch to the source code by using the **patch** utility.

NOTE

Software developers often use version control systems such as Git to manage their code base. Such tools offer their own methods of creating diffs or patching software.

3.1.1. Creating a patch file for a sample C program

You can create a patch from the original source code by using the **diff** utility. For example, to patch a **Hello world** program written in C (**cello.c**), complete the following steps.

Prerequisites

- You installed the **diff** utility on your system:

```
# dnf install diffutils
```

Procedure

1. Back up the original source code:

```
$ cp -p cello.c cello.c.orig
```

   The **-p** option preserves mode, ownership, and timestamps.

2. Modify **cello.c** as needed:

```
#include <stdio.h>

int main(void) {
    printf("Hello World from my very first patch!\n");
    return 0;
}
```

3. Generate a patch:

```
$ diff -Naur cello.c.orig cello.c
--- cello.c.orig        2016-05-26 17:21:30.478523360 -0500
+ cello.c     2016-05-27 14:53:20.668588245 -0500
```

```
@@ -1,6 +1,6 @@
 #include<stdio.h>

 int main(void){
-    printf("Hello World!\n");
+    printf("Hello World from my very first patch!\n");
     return 0;
 }
\ No newline at end of file
```

Lines that start with **+** replace the lines that start with **-**.

NOTE

Using the **Naur** options with the **diff** command is recommended because it fits the majority of use cases:

- **-N** (**--new-file**)
  The **-N** option handles absent files as empty files.
- **-a** (**--text**)
  The **-a** option treats all files as text. As a result, the **diff** utility does not ignore the files it classified as binaries.
- **-u** (**-U NUM** or **--unified[=NUM]**)
  The **-u** option returns output in the form of output NUM (default 3) lines of unified context. This is a compact and an easily readable format commonly used in patch files.
- **-r** (**--recursive**)
  The **-r** option recursively compares any subdirectories that the **diff** utility found.

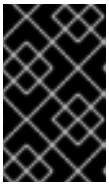
However, note that in this particular case, only the **-u** option is necessary.

4. Save the patch to a file:

```
$ diff -Naur cello.c.orig cello.c > cello.patch
```

5. Restore the original **cello.c**:

```
$ mv cello.c.orig cello.c
```



IMPORTANT

You must retain the original **cello.c** because the RPM package manager uses the original file, not the modified one, when building an RPM package. For more information, see Working with spec files.

Additional resources

- **diff(1)** man page on your system

3.1.2. Patching a sample C program

To apply code patches on your software, you can use the **patch** utility.

Prerequisites

- You installed the **patch** utility on your system:

```
# dnf install patch
```

- You created a patch from the original source code. For instructions, see Creating a patch file for a sample C program.

Procedure

The following steps apply a previously created **cello.patch** file on the **cello.c** file.

1. Redirect the patch file to the **patch** command:

```
$ patch < cello.patch
patching file cello.c
```

2. Check that the contents of **cello.c** now reflect the desired change:

```
$ cat cello.c
#include<stdio.h>

int main(void){
    printf("Hello World from my very first patch!\n");
    return 1;
}
```

Verification

1. Build the patched **cello.c** program:

```
$ make
gcc -g -o cello cello.c
```

2. Run the built **cello.c** program:

```
$ ./cello
Hello World from my very first patch!
```

3.2. CREATING A LICENSE FILE

It is recommended that you distribute your software with a software license.

A software license file informs users of what they can and cannot do with a source code. Having no license for your source code means that you retain all rights to this code and no one can reproduce, distribute, or create derivative works from your source code.

Procedure

- Create the **LICENSE** file with the required license statement:

```
$ vim LICENSE
```

Example 3.1. Example GPLv3 LICENSE file text

```
$ cat /tmp/LICENSE
This program is free software: you can redistribute it and/or modify it under the terms of
the GNU General Public License as published by the Free Software Foundation, either
version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this
program. If not, see http://www.gnu.org/licenses/.
```

Additional resources

- Source code examples

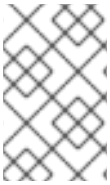
3.3. CREATING A SOURCE CODE ARCHIVE FOR DISTRIBUTION

An archive file is a file with the **.tar.gz** or **.tgz** suffix. Putting source code into the archive is a common way to release the software to be later packaged for distribution.

3.3.1. Creating a source code archive for a sample Bash program

The *bello* project is a **Hello World** file in Bash.

The following example contains only the **bello** shell script. Therefore, the resulting **tar.gz** archive has only one file in addition to the **LICENSE** file.



NOTE

The **patch** file is not distributed in the archive with the program. The RPM package manager applies the patch when the RPM is built. The patch will be placed into the **~/rpmbuild/SOURCES/** directory together with the **tar.gz** archive.

Prerequisites

- Assume that the **0.1** version of the **bello** program is used.
- You created a **LICENSE** file. For instructions, see Creating a LICENSE file.

Procedure

1. Move all required files into a single directory:

```
$ mkdir bello-0.1
```

```
$ mv ~/bello bello-0.1/

$ mv LICENSE bello-0.1/
```

2. Create the archive for distribution:

```
$ tar -cvzf bello-0.1.tar.gz bello-0.1
bello-0.1/
bello-0.1/LICENSE
bello-0.1/bello
```

3. Move the created archive to the **~/rpmbuild/SOURCES/** directory, which is the default directory where the **rpmbuild** command stores the files for building packages:

```
$ mv bello-0.1.tar.gz ~/rpmbuild/SOURCES/
```

Additional resources

- Hello World written in Bash

3.3.2. Creating a source code archive for a sample Python program

The *pello* project is a **Hello World** file in Python.

The following example contains only the **pello.py** program. Therefore, the resulting **tar.gz** archive has only one file in addition to the **LICENSE** file.

NOTE

The **patch** file is not distributed in the archive with the program. The RPM package manager applies the patch when the RPM is built. The patch will be placed into the **~/rpmbuild/SOURCES/** directory together with the **tar.gz** archive.

Prerequisites

- Assume that the **0.1.1** version of the **pello** program is used.
- You created a **LICENSE** file. For instructions, see Creating a LICENSE file .

Procedure

1. Move all required files into a single directory:

```
$ mkdir pello-0.1.1

$ mv pello.py pello-0.1.1/

$ mv LICENSE pello-0.1.1/
```

2. Create the archive for distribution:

```
$ tar -cvzf pello-0.1.1.tar.gz pello-0.1.1
pello-0.1.1/
```

```
pello-0.1.1/LICENSE
pello-0.1.1/pello.py
```

3. Move the created archive to the **~/rpmbuild/SOURCES/** directory, which is the default directory where the **rpmbuild** command stores the files for building packages:

```
$ mv pello-0.1.1.tar.gz ~/rpmbuild/SOURCES/
```

Additional resources

- Hello World written in Python

3.3.3. Creating a source code archive for a sample C program

The *cello* project is a **Hello World** file in C.

The following example contains only the **cello.c** and the **Makefile** files. Therefore, the resulting **tar.gz** archive has two files in addition to the **LICENSE** file.

NOTE

The **patch** file is not distributed in the archive with the program. The RPM package manager applies the patch when the RPM is built. The patch will be placed into the **~/rpmbuild/SOURCES/** directory together with the **tar.gz** archive.

Prerequisites

- Assume that the **1.0** version of the **cello** program is used.
- You created a **LICENSE** file. For instructions, see Creating a LICENSE file.

Procedure

1. Move all required files into a single directory:

```
$ mkdir cello-1.0

$ mv cello.c cello-1.0/

$ mv Makefile cello-1.0/

$ mv LICENSE cello-1.0/
```

2. Create the archive for distribution:

```
$ tar -cvzf cello-1.0.tar.gz cello-1.0
cello-1.0/
cello-1.0/Makefile
cello-1.0/cello.c
cello-1.0/LICENSE
```

3. Move the created archive to the **~/rpmbuild/SOURCES/** directory, which is the default directory where the **rpmbuild** command stores the files for building packages:

```
$ mv cello-1.0.tar.gz ~/rpmbuild/SOURCES/
```

Additional resources

- Hello World written in C

CHAPTER 4. PACKAGING SOFTWARE

In the following sections, learn the basics of the packaging process with the RPM package manager.

4.1. SETTING UP RPM PACKAGING WORKSPACE

To build RPM packages, you must first create a special workspace that consists of directories used for different packaging purposes.

4.1.1. Configuring RPM packaging workspace

To configure the RPM packaging workspace, you can set up a directory layout by using the **rpmdev-setuptree** utility.

Prerequisites

- You installed the **rpmdevtools** package, which provides utilities for packaging RPMs:

```
# dnf install rpmdevtools
```

Procedure

- Run the **rpmdev-setuptree** utility:

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/user/rpmbuild/
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS

5 directories, 0 files
```

Additional resources

- RPM packaging workspace directories

4.1.2. RPM packaging workspace directories

The following are the RPM packaging workspace directories created by using the **rpmdev-setuptree** utility:

Table 4.1. RPM packaging workspace directories

Directory	Purpose
BUILD	Contains build artifacts compiled from the source files from the **SOURCES** directory.

Directory	Purpose
RPMS	Binary RPMs are created under the **RPMS** directory in subdirectories for different architectures. For example, in the **x86_64** or **noarch** subdirectory.
SOURCES	Contains compressed source code archives and patches. The **rpmbuild** command then searches for these archives and patches in this directory.
SPECS	Contains **spec** files created by the packager. These files are then used for building packages.
SRPMS	When you use the **rpmbuild** command to build an SRPM instead of a binary RPM, the resulting SRPM is created under this directory.

4.2. ABOUT SPEC FILES

A **spec** file is a file with instructions that the **rpmbuild** utility uses to build an RPM package. This file provides necessary information to the build system by defining instructions in a series of sections. These sections are defined in the *Preamble* and the *Body* part of the **spec** file:

- The *Preamble* section contains a series of metadata items that are used in the *Body* section.
- The *Body* section represents the main part of the instructions.

4.2.1. Preamble items

The following are some of the directives that you can use in the *Preamble* section of the RPM **spec** file.

Table 4.2. The *Preamble* section directives

Directive	Definition
Name	A base name of the package that must match the **spec** file name.
Version	An upstream version number of the software.
Release	The number of times the version of the package was released. Set the initial value to **1%{?dist}** and increase it with each new release of the package. Reset to **1** when a new **Version** of the software is built.
Summary	A brief one-line summary of the package.

Directive	Definition
License	A license of the software being packaged. The exact format for how to label the **License** in your **spec** file varies depending on which RPM-based Linux distribution guidelines you are following, for example, GPLv3+.
URL	A full URL for more information about the software, for example, an upstream project website for the software being packaged.
Source	A path or URL to the compressed archive of the unpatched upstream source code. This link must point to an accessible and reliable storage of the archive, for example, the upstream page, not the packager's local storage. You can apply the **Source** directive either with or without numbers at the end of the directive name. If there is no number given, the number is assigned to the entry internally. You can also give the numbers explicitly, for example, **Source0**, **Source1**, **Source2**, **Source3**, and so on.
Patch	A name of the first patch to apply to the source code, if necessary. You can apply the **Patch** directive either with or without numbers at the end of the directive name. If there is no number given, the number is assigned to the entry internally. You can also give the numbers explicitly, for example, **Patch0**, **Patch1**, **Patch2**, **Patch3**, and so on. You can apply the patches individually by using the **%patch0**, **%patch1**, **%patch2** macro, and so on. Macros are applied within the**%prep** directive in the *Body* section of the RPM**spec** file. Alternatively, you can use the **%autopatch** macro that automatically applies all patches in the order they are given in the **spec** file.
BuildArch	An architecture that the software will be built for. If the software is not architecture-dependent, for example, if you wrote the software entirely in an interpreted programming language, set the value to **BuildArch: noarch**. If you do not set this value, the software automatically inherits the architecture of the machine on which it is built, for example, **x86_64**.
BuildRequires	A comma- or whitespace-separated list of packages required to build the program written in a compiled language. There can be multiple entries of **BuildRequires**, each on its own line in the SPEC file.
Requires	A comma- or whitespace-separated list of packages required by the software to run once installed. There can be multiple entries of **Requires**, each on its own line in the **spec** file.

Directive	Definition
ExcludeArch	If a piece of software cannot operate on a specific processor architecture, you can exclude this architecture in the **ExcludeArch** directive.
Conflicts	A comma- or whitespace-separated list of packages that must not be installed on the system in order for your software to function properly when installed. There can be multiple entries of **Conflicts**, each on its own line in the **spec** file.
Obsoletes	The **Obsoletes** directive changes the way updates work depending on the following factors: ● If you use the **rpm** command directly on a command line, it removes all packages that match obsoletes of packages being installed, or the update is performed by an updates or dependency solver. ● If you use the updates or dependency resolver (**DNF**), packages containing matching **Obsoletes:** are added as updates and replace the matching packages.
Provides	If you add the **Provides** directive to the package, this package can be referred to by dependencies other than its name.

The **Name**, **Version**, and **Release** (**NVR**) directives comprise the file name of the RPM package in the **name-version-release** format.

You can display the **NVR** information for a specific package by querying RPM database by using the **rpm** command, for example:

```
# rpm -q bash
bash-4.4.19-7.el8.x86_64
```

Here, **bash** is the package name, **4.4.19** is the version, and **7.el8** is the release. The **x86_64** marker is the package architecture. Unlike **NVR**, the architecture marker is not under direct control of the RPM packager, but is defined by the **rpmbuild** build environment. The exception to this is the architecture-independent **noarch** package.

4.2.2. Body items

The following are the items used in the *Body* section of the RPM **spec** file.

Table 4.3. The Body section items

Directive	Definition
%description	A full description of the software packaged in the RPM. This description can span multiple lines and can be broken into paragraphs.

Directive	Definition
%prep	A command or series of commands to prepare the software for building, for example, for unpacking the archive in the **Source** directive. The **%prep** directive can contain a shell script.
%build	A command or series of commands for building the software into machine code (for compiled languages) or bytecode (for some interpreted languages).
%install	A command or series of commands that the **rpmbuild** utility will use to install the software into the **BUILDROOT** directory once the software has been built. These commands copy the desired build artifacts from the **%_builddir** directory, where the build happens, to the **%buildroot** directory that contains the directory structure with the files to be packaged. This includes copying files from **~/rpmbuild/BUILD** to **~/rpmbuild/BUILDROOT** and creating the necessary directories in **~/rpmbuild/BUILDROOT**. The **%install** directory is an empty **chroot** base directory, which resembles the end user's **root** directory. Here you can create any directories that will contain the installed files. To create such directories, you can use RPM macros without having to hardcode the paths. Note that **%install** is only run when you create a package, not when you install it. For more information, see Working with spec files.
%check	A command or series of commands for testing the software, for example, unit tests.
%files	A list of files, provided by the RPM package, to be installed in the user's system and their full path location on the system. During the build, if there are files in the **%buildroot** directory that are not listed in **%files**, you will receive a warning about possible unpackaged files. Within the **%files** section, you can indicate the role of various files by using built-in macros. This is useful for querying the package file manifest metadata by using the **rpm** command. For example, to indicate that the **LICENSE** file is a software license file, use the **%license** macro.
%changelog	A record of changes that happened to the package between different **Version** or **Release** builds. These changes include a list of date-stamped entries for each Version-Release of the package. These entries log packaging changes, not software changes, for example, adding a patch or changing the build procedure in the **%build** section.

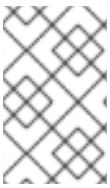
4.2.3. Advanced items

A **spec** file can contain advanced items, such as Scriptlets or Triggers. Scriptlets and Triggers take effect at different points during the installation process on the end user's system, not the build process.

4.3. BUILDROOTS

In the context of RPM packaging, **buildroot** is a **chroot** environment. The build artifacts are placed here by using the same file system hierarchy as the future hierarchy in the end user's system, with **buildroot** acting as the root directory. The placement of build artifacts must comply with the file system hierarchy standard of the end user's system.

The files in **buildroot** are later put into a **cpio** archive, which becomes the main part of the RPM. When RPM is installed on the end user's system, these files are extracted in the **root** directory, preserving the correct hierarchy.



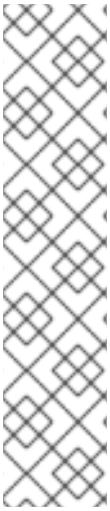
NOTE

The **rpmbuild** program has its own defaults. Overriding these defaults can cause certain issues. Therefore, avoid defining your own value of the **buildroot** macro. Use the default **%{buildroot}** macro instead.

4.4. RPM MACROS

An rpm macro is a straight text substitution that can be conditionally assigned based on the optional evaluation of a statement when certain built-in functionality is used. Therefore, RPM can perform text substitutions for you.

For example, you can define *Version* of the packaged software only once in the **%{version}** macro, and use this macro throughout the **spec** file. Every occurrence is automatically substituted by *Version* that you defined in the macro.



NOTE

If you see an unfamiliar macro, you can evaluate it with the following command:

```
$ rpm --eval %{MACRO}
```

For example, to evaluate the **%{_bindir}** and **%{_libexecdir}** macros, enter:

```
$ rpm --eval %{_bindir}
/usr/bin

$ rpm --eval %{_libexecdir}
/usr/libexec
```

4.5. WORKING WITH SPEC FILES

To package new software, you must create a **spec** file. You can create the **spec** file either of the following ways:

- Write the new **spec** file manually from scratch.
- Use the **rpmdev-newspec** utility. This utility creates an unpopulated **spec** file, where you fill the necessary directives and fields.

NOTE

Some programmer-focused text editors pre-populate a new **spec** file with their own **spec** template. The **rpmdev-newspec** utility provides an editor-agnostic method.

4.5.1. Creating a new spec file for sample Bash, Python, and C programs

You can create a **spec** file for each of the three implementations of the **Hello World!** program by using the **rpmdev-newspec** utility.

Prerequisites

- The following **Hello World!** program implementations were placed into the **~/rpmbuild/SOURCES** directory:
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)

Procedure

1. Navigate to the **~/rpmbuild/SPECS** directory:

```
$ cd ~/rpmbuild/SPECS
```

2. Create a **spec** file for each of the three implementations of the **Hello World!** program:

```
$ rpmdev-newspec bello
bello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec cello
cello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec pello
pello.spec created; type minimal, rpm version >= 4.11.
```

   The **~/rpmbuild/SPECS/** directory now contains three **spec** files named **bello.spec**, **cello.spec**, and **pello.spec**.

3. Examine the created files.
   The directives in the files represent those described in About spec files. In the following sections, you will populate a particular section in the output files of **rpmdev-newspec**.

4.5.2. Modifying an original spec file

The original output **spec** file generated by the **rpmdev-newspec** utility represents a template that you must modify to provide necessary instructions for the **rpmbuild** utility. **rpmbuild** then uses these instructions to build an RPM package.

Prerequisites

- The unpopulated **~/rpmbuild/SPECS/<name>.spec spec** file was created by using the **rpmdev-newspec** utility. For more information, see Creating a new spec file for sample Bash, Python, and C programs.

Procedure

1. Open the **~/rpmbuild/SPECS/<name>.spec** file provided by the **rpmdev-newspec** utility.

2. Populate the following directives of the **spec** file *Preamble* section:

Name

Name was already specified as an argument to **rpmdev-newspec**.

Version

Set **Version** to match the upstream release version of the source code.

Release

Release is automatically set to **1%{?dist}**, which is initially **1**.

Summary

Enter a one-line explanation of the package.

License

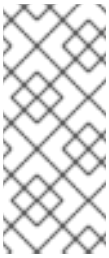
Enter the software license associated with the source code.

URL

Enter the URL to the upstream software website. For consistency, utilize the **%{name}** RPM macro variable and use the **https://example.com/%{name}** format.

Source

Enter the URL to the upstream software source code. Link directly to the software version being packaged.



NOTE

The example URLs in this documentation include hard-coded values that could possibly change in the future. Similarly, the release version can change as well. To simplify these potential future changes, use the **%{name}** and **%{version}** macros. By using these macros, you need to update only one field in the **spec** file.

BuildRequires

Specify build-time dependencies for the package.

Requires

Specify run-time dependencies for the package.

BuildArch

Specify the software architecture.

3. Populate the following directives of the **spec** file *Body* section. You can think of these directives as section headings, because these directives can define multi-line, multi-instruction, or scripted tasks to occur.

%description

Enter the full description of the software.

%prep

Enter a command or series of commands to prepare software for building.

%build

Enter a command or series of commands for building software.

%install

Enter a command or series of commands that instruct the **rpmbuild** command on how to install the software into the **BUILDROOT** directory.

%files

Specify the list of files, provided by the RPM package, to be installed on your system.

%changelog

Enter the list of datestamped entries for each **Version-Release** of the package.
Start the first line of the **%changelog** section with an asterisk (***) character followed by **Day-of-Week Month Day Year Name Surname <email> - Version-Release**.

For the actual change entry, follow these rules:

- Each change entry can contain multiple items, one for each change.
- Each item starts on a new line.
- Each item begins with a hyphen (**-**) character.

You have now written an entire **spec** file for the required program.

Additional resources

- Preamble items
- Body items

4.5.3. An example spec file for a sample Bash program

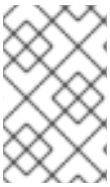
You can use the following example **spec** file for the **bello** program written in bash for your reference.

An example spec file for the bello program written in bash

```
Name:           bello
Version:        0.1
Release:        1%{?dist}
Summary:        Hello World example implemented in bash script

License:        GPLv3+
URL:            https://www.example.com/%{name}
Source0:        https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Requires:       bash

BuildArch:      noarch

%description
The long-tail description for our Hello World Example implemented in
bash script.

%prep
%setup -q

%build

%install

mkdir -p %{buildroot}/%{_bindir}
```

```
install -m 0755 %{name} %{buildroot}/%{_bindir}/%{name}

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First bello package
- Example second item in the changelog for version-release 0.1-1
```

- The **BuildRequires** directive, which specifies build-time dependencies for the package, was deleted because there is no building step for **bello**. Bash is a raw interpreted programming language, and the files are just installed to their location on the system.
- The **Requires** directive, which specifies run-time dependencies for the package, includes only **bash**, because the **bello** script requires only the **bash** shell environment to execute.
- The **%build** section, which specifies how to build the software, is blank, because the **bash** script does not need to be built.



NOTE

To install **bello**, you must create the destination directory and install the executable **bash** script file there. Therefore, you can use the **install** command in the **%install** section. You can use RPM macros to do this without hardcoding paths.

Additional resources

- What is source code

4.5.4. An example spec file for a sample Python program

You can use the following example **spec** file for the **pello** program written in the Python programming language for your reference.

An example spec file for the pello program written in Python

```
Name:           pello
Version:        0.1.1
Release:        1%{?dist}
Summary:        Hello World example implemented in Python

License:        GPLv3+
URL:            https://www.example.com/%{name}
Source0:        https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

BuildRequires:  python
Requires:       python
Requires:       bash

BuildArch:      noarch

%description
```

```
The long-tail description for our Hello World Example implemented in Python.

%prep
%setup -q

%build

python -m compileall %{name}.py

%install

mkdir -p %{buildroot}/%{_bindir}
mkdir -p %{buildroot}/usr/lib/%{name}

cat > %{buildroot}/%{_bindir}/%{name} <<EOF
#!/bin/bash
/usr/bin/python /usr/lib/%{name}/%{name}.pyc
EOF

chmod 0755 %{buildroot}/%{_bindir}/%{name}

install -m 0644 %{name}.py* %{buildroot}/usr/lib/%{name}/

%files
%license LICENSE
%dir /usr/lib/%{name}/
%{_bindir}/%{name}
/usr/lib/%{name}/%{name}.py*

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
  - First pello package
```

- The **Requires** directive, which specifies run-time dependencies for the package, includes two packages:
  - The **python** package required to execute the byte-compiled code at runtime.
  - The **bash** package required to execute the small entry-point script.
- The **BuildRequires** directive, which specifies build-time dependencies for the package, includes only the **python** package. The **pello** program requires **python** to perform the byte-compile build process.
- The **%build** section, which specifies how to build the software, creates a byte-compiled version of the script. Note that in real-world packaging, it is usually done automatically, depending on the distribution used.
- The **%install** section corresponds to the fact that you must install the byte-compiled file into a library directory on the system so that it can be accessed.

This example of creating a wrapper script in-line in the **spec** file shows that the **spec** file itself is scriptable. This wrapper script executes the Python byte-compiled code by using the **here document**.

Additional resources

- What is source code

4.5.5. An example spec file for a sample C program

You can use the following example **spec** file for the **cello** program that was written in the C programming language for your reference.

An example spec file for the cello program written in C

```
Name:          cello
Version:       1.0
Release:       1%{?dist}
Summary:       Hello World example implemented in C

License:       GPLv3+
URL:           https://www.example.com/%{name}
Source0:       https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Patch0:        cello-output-first-patch.patch

BuildRequires:  gcc
BuildRequires:  make

%description
The long-tail description for our Hello World Example implemented in
C.

%prep
%setup -q

%patch0

%build
make %{?_smp_mflags}

%install
%make_install

%files
%license LICENSE
%{_bindir}/%{name}

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 1.0-1
- First cello package
```

- The **BuildRequires** directive, which specifies build-time dependencies for the package, includes the following packages required to perform the compilation build process:
  - **gcc**
  - **make**
- The **Requires** directive, which specifies run-time dependencies for the package, is omitted in this example. All runtime requirements are handled by **rpmbuild**, and the **cello** program does not require anything outside of the core C standard libraries.

- The **%build** section reflects the fact that in this example the **Makefile** file for the **cello** program was written. Therefore, you can use the GNU make command. However, you must remove the call to **%configure** because you did not provide a configure script.

You can install the **cello** program by using the **%make_install** macro. This is possible because the **Makefile** file for the **cello** program is available.

Additional resources

- What is source code

4.6. BUILDING RPMS

You can build RPM packages by using the **rpmbuild** command. When using this command, a certain directory and file structure is expected, which is the same as the structure that was set up by the **rpmdev-setuptree** utility.

Different use cases and desired outcomes require different combinations of arguments to the **rpmbuild** command. The following are the main use cases:

- Building source RPMs.
- Building binary RPMs:
  - Rebuilding a binary RPM from a source RPM.
  - Building a binary RPM from the **spec** file.

4.6.1. Building a source RPM

Building a Source RPM (SRPM) has the following advantages:

- You can preserve the exact source of a certain **Name-Version-Release** of an RPM file that was deployed to an environment. This includes the exact **spec** file, the source code, and all relevant patches. This is useful for tracking and debugging purposes.
- You can build a binary RPM on a different hardware platform or architecture.

Prerequisites

- You have installed the **rpmbuild** utility on your system:

```
# dnf install rpm-build
```

- The following **Hello World!** implementations were placed into the **~/rpmbuild/SOURCES/** directory:
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)
- A **spec** file for the program that you want to package exists.

Procedure

1. Navigate to the **~/rpmbuild/SPECS/** directive, which contains the created **spec** file:

```
$ cd ~/rpmbuild/SPECS/
```

2. Build the source RPM by entering the **rpmbuild** command with the specified **spec** file:

```
$ rpmbuild -bs <specfile>
```

The **-bs** option stands for the *build source*.

For example, to build source RPMs for the **bello**, **pello**, and **cello** programs, enter:

```
$ rpmbuild -bs bello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
```

Verification

- Verify that the **rpmbuild/SRPMS** directory includes the resulting source RPMs. The directory is a part of the structure expected by **rpmbuild**.

Additional resources

- Working with spec files

4.6.2. Rebuilding a binary RPM from a source RPM

To rebuild a binary RPM from a source RPM (SRPM), use the **rpmbuild** command with the **--rebuild** option.

The output generated when creating the binary RPM is verbose, which is helpful for debugging. The output varies for different examples and corresponds to their **spec** files.

The resulting binary RPMs are located in either of the following directories:

- **~/rpmbuild/RPMS/YOURARCH**, where **YOURARCH** is your architecture.
- **~/rpmbuild/RPMS/noarch/**, if the package is not architecture-specific.

Prerequisites

- You have installed the **rpmbuild** utility on your system:

```
# dnf install rpm-build
```

Procedure

1. Navigate to the **~/rpmbuild/SRPMS/** directive, which contains the SRPM:

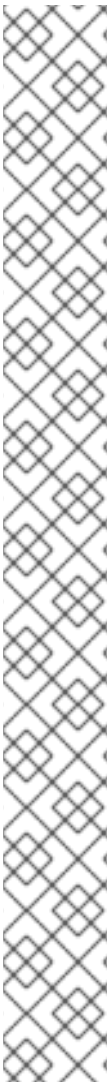
```
$ cd ~/rpmbuild/SRPMS/
```

2. Rebuild the binary RPM from the SRPM:

```
$ rpmbuild --rebuild <srpm>
```

For example, to rebuild **bello**, **pello**, and **cello** from their SRPMs, enter:

```
$ rpmbuild --rebuild bello-0.1-1.el8.src.rpm
[output truncated]

$ rpmbuild --rebuild pello-0.1.2-1.el8.src.rpm
[output truncated]

$ rpmbuild --rebuild cello-1.0-1.el8.src.rpm
[output truncated]
```



NOTE

Invoking **rpmbuild --rebuild** involves the following processes:

- Installing the contents of the SRPM (the **spec** file and the source code) into the **~/rpmbuild/** directory.
- Building an RPM by using the installed contents.
- Removing the **spec** file and the source code.

You can retain the **spec** file and the source code after building either of the following ways:

- When building the RPM, use the **rpmbuild** command with the **--recompile** option instead of the **--rebuild** option.
- Install SRPMs for **bello**, **pello**, and **cello**:

```
$ rpm -Uvh ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
Updating / installing…
   1:bello-0.1-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
Updating / installing…
…1:pello-0.1.2-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
Updating / installing…
…1:cello-1.0-1.el8              [100%]
```

4.6.3. Building a binary RPM from a spec file

To build a binary RPM from its **spec** file, use the **rpmbuild** command with the **-bb** option.

Prerequisites

- You have installed the **rpmbuild** utility on your system:

```
# dnf install rpm-build
```

Procedure

1. Navigate to the ~/**rpmbuild**/**SPECS**/ directive, which contains **spec** files:

```
$ cd ~/rpmbuild/SPECS/
```

2. Build the binary RPM from its **spec**:

```
$ rpmbuild -bb <spec_file>
```

For example, to build **bello**, **pello**, and **cello** binary RPMs from their **spec** files, enter:

```
$ rpmbuild -bb bello.spec
$ rpmbuild -bb pello.spec
$ rpmbuild -bb cello.spec
```

4.7. LOGGING RPM ACTIVITY TO SYSLOG

You can log any RPM activity or transaction by using the System Logging protocol (**syslog**).

Prerequisites

- The **syslog** plug-in is installed on the system:

```
# dnf install rpm-plugin-syslog
```

NOTE

The default location for the **syslog** messages is the /**var**/**log**/**messages** file. However, you can configure **syslog** to use another location to store the messages.

Procedure

1. Open the file that you configured to store the **syslog** messages.
   Alternatively, if you use the default **syslog** configuration, open the /**var**/**log**/**messages** file.
2. Search for new lines including the **[RPM]** string.

4.8. EXTRACTING RPM CONTENT

In some cases, for example, if a package required by RPM is damaged, you might need to extract the content of the package. In such cases, if an RPM installation is still working despite the damage, you can use the **rpm2archive** utility to convert an **.rpm** file to a tar archive to use the content of the package.

NOTE

If the RPM installation is severely damaged, you can use the **rpm2cpio** utility to convert the RPM package file to a **cpio** archive.

Procedure

- Convert the RPM file to the tar archive:

```
$ rpm2archive <filename>.rpm
```

The resulting file has the **.tgz** suffix. For example, to create an archive from the **bash** package, enter:

```
$ rpm2archive bash-4.4.19-6.el8.x86_64.rpm
$ ls bash-4.4.19-6.el8.x86_64.rpm.tgz
bash-4.4.19-6.el8.x86_64.rpm.tgz
```

4.9. SIGNING RPM PACKAGES

You can sign RPM packages to ensure no third party can alter their content by using either of the following software:

- Sequoia PGP supports the OpenPGP standard. RPM also uses Sequoia PGP to verify software signatures.
- GNU Privacy Guard (GnuPG) supports older OpenPGP standard versions, which makes GnuPG more compatible with RHEL 9 and earlier versions.

WARNING

New algorithms and signatures might not be compatible with earlier RHEL versions.

4.9.1. Signing RPM packages with GnuPG

You can sign RPM packages by using the GNU Privacy Guard (GnuPG) software.

4.9.1.1. Creating an OpenPGP key for signing packages with GnuPG

To sign an RPM package by using the GNU Privacy Guard (GnuPG) software, you must create an OpenPGP key first.

Prerequisites

- You have the **rpm-sign** and **pinentry** packages installed on your system.

Procedure

1. Generate an OpenPGP key pair:

```
$ gpg --gen-key
```

2. Check the generated key pair:

```
$ gpg --list-keys
```

3. Export the public key:

```
$ gpg --export -a '<public_key_name>' > RPM-GPG-KEY-pmanager
```

Next steps

1. Configure RPM to sign a package with GnuPG .
2. Add a signature to an RPM package .

4.9.1.2. Configuring RPM to sign a package with GnuPG

To sign an RPM package by using the GNU Privacy Guard (GnuPG) software, you must configure RPM by specifying the **%_gpg_name** RPM macro.

Prerequisites

- You created an OpenPGP key for GnuPG, For more information, see Creating an OpenPGP key for signing packages with GnuPG.

Procedure

- Define the **%_gpg_name** macro in your **$HOME/.rpmmacros** directory:

```
%_gpg_name <key-ID>
```

A valid key ID value for GnuPG can be a key fingerprint, full name, or email address you provided when creating the key.`

Next steps

- Add a signature to an RPM package .

4.9.1.3. Adding a signature to an RPM package

Packages are commonly built without signatures. You can add your signature before the package is released.

Prerequisites

- You created an OpenPGP key for GnuPG. For more information, see Creating an OpenPGP key for signing packages with GnuPG.
- You configured RPM for signing packages. For more information, see Configuring RPM to sign a package with GnuPG.
- You have the **rpm-sign** package installed on your system.

Procedure

- Add a signature to a package:

```
$ rpmsign --addsign <package-name>.rpm
```

Verification

1. Import the exported OpenPGP public key into the RPM keyring:

```
# rpmkeys --import RPM-GPG-KEY-pmanager
```

2. Display the key ID with GnuPG:

```
$ gpg --list-keys
[...]
pub   rsa3072 2025-05-13 [SC] [expires: 2028-05-12]
      A8AF1C39AC67A1501450734F6DE8FC866DE0394D
[...]
```

   The key ID is the 40-character string in the command output, for example, **A8AF1C39AC67A1501450734F6DE8FC866DE0394D**.

3. Verify that the RPM file has the corresponding signature:

```
$ rpm -Kv <package_name>.rpm
<package_name>.rpm:
   Header V4 RSA/SHA256 Signature, key ID 6de0394d: OK
   Header SHA256 digest: OK
   Header SHA1 digest: OK
   Payload SHA256 digest: OK
   MD5 digest: OK
```

   The signature key ID matches the last part of the OpenPGP key ID.

4.9.2. Signing RPM packages with Sequoia PGP

You can use Sequoia PGP to sign RPM packages and ensure no third party can alter their content.

4.9.2.1. Creating an OpenPGP key for signing packages with Sequoia PGP

To sign packages by using the Sequoia PGP software, you must create an OpenPGP key first.

Procedure

1. Install the Sequoia PGP tools:

```
# dnf install sequoia-sq
```

2. Generate an OpenPGP key pair:

```
$ sq key generate --own-key --userid <key_name>
```

3. Check the generated key pair:

```
$ sq key list
```

4. Export the public key:

```
$ sq cert export --cert-userid '<key_name>' > RPM-PGP-KEY-pmanager
```

Next steps

1. Configure RPM to sign a package with Sequoia PGP .
2. Add a signature to an RPM package .

4.9.2.2. Configuring RPM to sign a package with Sequoia PGP

To sign an RPM package with the Sequoia PGP software, you must configure the RPM to use Sequoia PGP and specify the **%_gpg_name** macro.

Prerequisites

- You have the **rpm-sign** package installed on your system.

Procedure

1. Copy the **macros.rpmsign-sequoia** file to the **/etc/rpm** directory:

```
# cp /usr/share/doc/rpm/macros.rpmsign-sequoia /etc/rpm/
```

2. Get a valid OpenPGP key fingerprint value from the output of key listing:

```
$ sq cert list --cert-userid '<key_name>'
 - 7E4B52101EB3DB08967A1E5EB595D12FDA65BA50
   - created 2025-05-13 10:33:29 UTC
   - will expire 2028-05-13T03:59:50Z

   - [  ✓  ] <key_name>
```

The key fingerprint is a 40-character string on the first line of the output, for example, **7E4B52101EB3DB08967A1E5EB595D12FDA65BA50**.

3. Define the **%_gpg_name** macro in your **$HOME/.rpmmacros** file as follows:

```
%_gpg_name <key_fingerprint>
```

Note that you can also use the full key ID instead of the fingerprint.

NOTE

Unlike GnuPG, Sequoia PGP accepts only the full key ID or fingerprint.

Next steps

- Add a signature to an RPM package .

4.9.2.3. Adding a signature to an RPM package

Packages are commonly built without signatures. You can add your signature before the package is released.

Prerequisites

- You created an OpenPGP key. For more information, see Creating an OpenPGP key for signing packages with Sequoia PGP.
- You configured RPM for signing packages. For more information, see Configuring RPM to sign a package with Sequoia PGP.
- You have the **rpm-sign** package installed on your system.

Procedure

- Add a signature to a package:

```
$ rpmsign --addsign <package-name>.rpm
```

Verification

1. Import the exported OpenPGP public key into the RPM keyring:

```
# rpmkeys --import RPM-PGP-KEY-pmanager
```

2. Display the key fingerprint of the signing key:

```
$ sq key list --cert-userid <key_name>
 - 7E4B52101EB3DB08967A1E5EB595D12FDA65BA50
   - user ID: <key_name> (authenticated)
   - created 2025-05-13 10:33:29 UTC
   - will expire 2028-05-13T03:59:50Z
   - usable for signing
   - @softkeys/7E4B52101EB3DB08967A1E5EB595D12FDA65BA50: available, unlocked

   - 78E56DD2E12E02CFEEA27F8B9FE57972D6BCEA6F
     - created 2025-05-13 10:33:29 UTC
     - will expire 2028-05-13T03:59:50Z
     - usable for decryption
     - @softkeys/7E4B52101EB3DB08967A1E5EB595D12FDA65BA50: available, unlocked
 - C06E45F8ABC3E59F44A9E811578DDDB66422E345
     - created 2025-05-13 10:33:29 UTC
     - will expire 2028-05-13T03:59:50Z
     - usable for signing
     - @softkeys/7E4B52101EB3DB08967A1E5EB595D12FDA65BA50: available, unlocked
 - E0BD231AB350AD6802D44C0A270E79FFC39C3B25
     - created 2025-05-13 10:33:29 UTC
     - will expire 2028-05-13T03:59:50Z
     - usable for signing
     - @softkeys/7E4B52101EB3DB08967A1E5EB595D12FDA65BA50: available, unlocked
```

The key fingerprint is usually a signing subkey in the **sq key list --cert-userid *<key_name>*** command output, for example, **E0BD231AB350AD6802D44C0A270E79FFC39C3B25**.

3. Verify that the RPM file has the corresponding signature, for example:

```
$ rpm -Kv <package_name>.rpm
<package_name>.rpm:
    Header V4 EdDSA/SHA512 Signature, key ID c39c3b25: OK
    Header SHA256 digest: OK
    Header SHA1 digest: OK
    Payload SHA256 digest: OK
    MD5 digest: OK
```

The signature key ID matches the last part of the key fingerprint.

CHAPTER 5. PACKAGING PYTHON 3 RPMS

You can install Python packages on your system by using the **DNF** package manager. **DNF** uses the RPM package format, which offers more downstream control over the software.

Packaging a Python project into an RPM package provides the following advantages compared to native Python packages:

- Dependencies on Python and non-Python packages are possible and strictly enforced by the **DNF** package manager.
- You can cryptographically sign the packages. With cryptographic signing, you can verify, integrate, and test contents of RPM packages with the rest of the operating system.
- You can execute tests during the build process.

The packaging format of native Python packages is defined by Python Packaging Authority (PyPA) Specifications. Historically, most Python projects used the **distutils** or **setuptools** utilities for packaging and defined package information in the **setup.py** file. However, possibilities of creating native Python packages have evolved over time:

- To package Python software that uses the **setup.py** file, follow this document.
- To package more modern packages with **pyproject.toml** files, see the **README** file in pyproject-rpm-macros. Note that **pyproject-rpm-macros** is included in the CodeReady Linux Builder (CRB) repository, which contains unsupported packages, and it can change over time to support newer Python packaging standards.

5.1. A SPEC FILE DESCRIPTION FOR AN EXAMPLE PYTHON PACKAGE

An RPM **spec** file for Python projects has some specifics compared to non-Python RPM **spec** files.

Note that it is recommended for any RPM package name of a Python library to include the **python3-** prefix.

See the notes about Python RPM **spec** files specifics in the following example of the **python3-pello** package.

An example SPEC file for the pello program written in Python

```
%global python3_pkgversion 3                                    1

Name:           python-pello                                    2
Version:        1.0.2
Release:        1%{?dist}
Summary:        Example Python library

License:        MIT
URL:            https://github.com/fedora-python/Pello
Source:         %{url}/archive/v%{version}/Pello-%{version}.tar.gz

BuildArch:      noarch
BuildRequires:  python%{python3_pkgversion}-devel               3

# Build dependencies need to be specified manually
```

```
BuildRequires:  python%{python3_pkgversion}-setuptools

# Test dependencies need to be specified manually
# Runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python%{python3_pkgversion}-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command line.}

%description %_description

%package -n python%{python3_pkgversion}-pello                    4
Summary:        %{summary}

%description -n python%{python3_pkgversion}-pello %_description


%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supports projects with setup.py
%py3_build                                                       5


%install
# The macro only supports projects with setup.py
%py3_install


%check                                                           6
%pytest


# Note that there is no %%files section for python-pello
%files -n python%{python3_pkgversion}-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

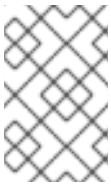
1. By defining the **python3_pkgversion** macro, you set which Python version this package will be built for. To build for the default Python version **3.12**, remove the line.

2. When packaging a Python project into RPM, always add the **python-** prefix to the original name of the project. The project name here is **Pello** and, therefore, the name of the Source RPM (SRPM) is **python-pello**.

3. **BuildRequires** specifies what packages are required to build and test this package. In **BuildRequires**, always include items providing tools necessary for building Python packages:

4. When choosing a name for the binary RPM (the package that users will be able to install), add a versioned Python prefix. Use the **python3-** prefix for the default Python 3.12. You can use the **%{python3_pkgversion}** macro, which evaluates to **3** for the default Python version **3.12** unless you set it to an explicit version, for example, when a later version of Python is available (see footnote 1).

5. The **%py3_build** and **%py3_install** macros run the **setup.py build** and **setup.py install** commands, respectively, with additional arguments to specify installation locations, the interpreter to use, and other details.



NOTE

Using the **setup.py build** and **setup.py install** commands from the **setuptools** package is deprecated and will be removed in the future major RHEL release. You can use pyproject-rpm-macros instead.

6. The **%check** section runs the tests of the packaged project. The exact command depends on the project itself, but you can use the **%pytest** macro to run the **pytest** command in an RPM-friendly way.

5.2. COMMON MACROS FOR PYTHON 3 RPMS

In a Python RPM **spec** file, always use the macros for Python 3 RPMs rather than hardcoding their values.

You can redefine which Python 3 version is used in these macros by defining the **python3_pkgversion** macro on top of your **spec** file. For more information, see A spec file description for an example Python package. If you define the **python3_pkgversion** macro, the values of the macros described in the following table will reflect the specified Python 3 version.

Table 5.1. Macros for Python 3 RPMs

Macro	Normal Definition	Description
%{python3_pkgversion}	3	The Python version that is used by all other macros. Can be redefined to any future Python versions that will be added.
%{python3}	/usr/bin/python3	The Python 3 interpreter.
%{python3_version}	3.12	The major.minor version of the Python 3 interpreter.
%{python3_sitelib}	/usr/lib/python3.12/site-packages	The location where pure-Python modules are installed.
%{python3_sitearch}	/usr/lib64/python3.12/site-packages	The location where modules containing architecture-specific extension modules are installed.

Macro	Normal Definition	Description
%py3_build		Expands to the **setup.py build** command with arguments suitable for an RPM package.
%py3_install		Expands to the **setup.py install** command with arguments suitable for an RPM package.
%{py3_shebang_flags}	sP	The default set of flags for the Python interpreter directives macro, **%py3_shebang_fix**.
%py3_shebang_fix		Changes Python interpreter directives to **#! %{python3}**, preserves any existing flags (if found), and adds flags defined in the **%{py3_shebang_flags}** macro.

Additional resources

- Python macros in upstream documentation

5.3. USING AUTOMATICALLY GENERATED DEPENDENCIES FOR PYTHON RPMS

You can automatically generate dependencies for Python RPMs by using upstream-provided metadata.

Prerequisites

- A **spec** file for the RPM exists. For more information, see A spec file description for an example Python package.

Procedure

1. Include one of the following directories in the resulting RPM:
   - **.dist-info**
   - **.egg-info**
     The RPM build process automatically generates virtual **pythonX.Ydist** provides from these directories, for example:

     ```
     python3.12dist(pello)
     ```

     The Python dependency generator then reads the upstream metadata and generates runtime requirements for each RPM package using the generated **pythonX.Ydist** virtual provides. Example of a generated requirements tag:

     ```
     Requires: python3.12dist(requests)
     ```

2. Inspect the generated **Requires**.

3. To remove some of the generated **Requires**, modify the upstream-provided metadata in the **%prep** section of the **spec** file.

4. To disable the automatic requirements generator, include the **%{?python_disable_dependency_generator}** macro above the main package's **%description** declaration.

Additional resources

- Automatically generated dependencies

CHAPTER 6. MODIFYING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS

In Red Hat Enterprise Linux 10, executable Python scripts are expected to use interpreter directives, also known as hashbangs or shebangs, that explicitly specify at a minimum the major Python version. For example:

```
#!/usr/bin/python3
#!/usr/bin/python3.12
```

The **/usr/lib/rpm/redhat/brp-mangle-shebangs buildroot** policy (BRP) script is run automatically when building any RPM package, and attempts to correct interpreter directives in all executable files. The BRP script generates errors when encountering a Python script with an ambiguous interpreter directive, for example, **#!/usr/bin/python** or **#!/usr/bin/env python**.

You can modify interpreter directives in the Python scripts to prevent build errors at RPM build time.

Prerequisites

- Some of the interpreter directives in your Python scripts cause a build error.

Procedure

- Depending on your scenario, modify interpreter directives by performing one of the following steps:
  - Use the following macro in the **%prep** section of your **spec** file:

    ```
    %py3_shebang_fix <SCRIPTNAME> …
    ```

    SCRIPTNAME can be any file, directory, or a list of files and directories.

    As a result, all listed files and all **.py** files in listed directories have their interpreter directives modified to point to **%{python3}**. Existing flags from the original interpreter directive will be preserved and additional flags defined in the **%{py3_shebang_flags}** macro will be added. You can redefine the **%{py3_shebang_flags}** macro in your **spec** file to change the flags that will be added.

  - Modify the packaged Python scripts so that they conform to the expected format.

Additional resources

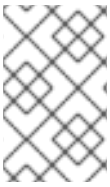
- Interpreter invocation

CHAPTER 7. PACKAGING RUBY GEMS

Ruby is a dynamic, interpreted, reflective, object-oriented, general-purpose programming language.

Programs written in Ruby are typically packaged by using the RubyGems software, which provides a specific Ruby packaging format.

Packages created by RubyGems are called gems and they can be re-packaged into RPM packages.



NOTE

This documentation refers to terms related to the RubyGems concept with the **gem** prefix, for example, **.gemspec** is used for the *gem specification*, and terms related to RPM are unqualified.

7.1. HOW RUBYGEMS RELATE TO RPM

RubyGems represent Ruby's own packaging format. However, RubyGems contain metadata similar to metadata required by RPM. This metadata streamlines packaging gems as RPMs. RPMs re-packaged from gems fit with the rest of the distribution. End users are also able to satisfy dependencies of a gem by installing the appropriate RPM-packaged gem and other system libraries.

RubyGems use terminology similar to RPM packages, such as **spec** files, package names, dependencies, and other items.

To conform with the rest of RHEL RPM distribution, packages created by RubyGems must comply with the following rules:

- Follow the **rubygem-%{gem_name}** pattern when naming your packages.
- Use the **#!/usr/bin/ruby** string as the interpreter directive.

7.2. RUBYGEMS SPEC FILE CONVENTIONS

A RubyGems **spec** file must meet the following conventions:

- The file contains a definition of **%{gem_name}**, which is the name from the gem's specification.
- The source of the package must be the full URL to the released gem archive.
- The version of the package must be the gem's version.
- The file contains the following **BuildRequires:** directive:

  ```
  BuildRequires: rubygems-devel
  ```

  The **rubygems-devel** package contains macros needed for a build.
- The file does not contain any additional **rubygem(foo) Requires** or **Provides** directives because these directives are autogenerated from the gem metadata.

7.2.1. RubyGems spec file example

The following is the RubyGems-specific part of an example **spec** file for building gems. The remaining part of the **spec** file follows the generic guidelines.

A RubyGems-specific part of an example spec file

```
%prep
%setup -q -n  %{gem_name}-%{version}

# Modify the gemspec if necessary
# Also apply patches to code if necessary
%patch 0 -p1

%build
# Create the gem as gem install only works on a gem file
gem build ../%{gem_name}-%{version}.gemspec

# %%gem_install compiles any C extensions and installs the gem into ./%%gem_dir
# by default, so that we can move it into the buildroot in %%install
%gem_install

%install
mkdir -p %{buildroot}%{gem_dir}
cp -a ./%{gem_dir}/* %{buildroot}%{gem_dir}/

# If there were programs installed:
mkdir -p %{buildroot}%{_bindir}
cp -a ./%{_bindir}/* %{buildroot}%{_bindir}

# If there are C extensions, copy them to the extdir.
mkdir -p %{buildroot}%{gem_extdir_mri}
cp -a .%{gem_extdir_mri}/{gem.build_complete,*.so} %{buildroot}%{gem_extdir_mri}/
```

7.2.2. RubyGems spec file directives

The following are the specifics of particular items in the RubyGems-specific part of the **spec** file.

Table 7.1. RubyGems' spec directives specifics

Directive	RubyGems specifics
%prep	RPM can directly unpack gem archives. The **%setup -n %{gem_name}-%{version}** macro provides the directory into which the gem is unpacked. At the same directory level, the **%{gem_name}-%{version}.gemspec** file is automatically created. You can use this file to perform the following actions: • Modify the **.gemspec** file • Apply patches to the code.

Directive	RubyGems specifics
%build	This section includes commands for building the software into machine code. The **%gem_install** macro operates only on gem archives. Therefore, you must first re-create the archive by using the **gem build ../%{gem_name}-%{version}.gemspec** command. The recreated gem file is then used by **%gem_install** to build and install the gem code into the default **./%{gem_dir}** temporary directory. Before being installed, the built sources are placed into a temporary directory that is created automatically.
%install	The installation is performed into the **%{buildroot}** hierarchy. You can create the necessary directories and then copy the installed code from the temporary directories into the **%{buildroot}** hierarchy. If the gem creates shared objects, they are moved into the architecture-specific **%{gem_extdir_mri}** path.

7.2.3. Additional resources

- Fedora packaging guidelines for Ruby

7.3. RUBYGEMS MACROS

The following are macros useful for packages created by RubyGems. These macros are provided by the **rubygems-devel** package.

Table 7.2. RubyGems' macros

Macro name	Extended path	Usage
%{gem_dir}	**/usr/share/gems**	Top directory for the gem structure.
%{gem_instdir}	**%{gem_dir}/gems/%{gem_name}-%{version}**	Directory with the actual content of the gem.
%{gem_libdir}	**%{gem_instdir}/lib**	The library directory of the gem.
%{gem_cache}	**%{gem_dir}/cache/%{gem_name}-%{version}.gem**	The cached gem.
%{gem_spec}	**%{gem_dir}/specifications/%{gem_name}-%{version}.gemspec**	The gem specification file.
%{gem_docdir}	**%{gem_dir}/doc/%{gem_name}-%{version}**	The RDoc documentation of the gem.
%{gem_extdir_mri}	**%{_libdir}/gems/ruby/%{gem_name}-%{version}**	The directory for gem extension.

7.4. USING GEM2RPM TO GENERATE A SPEC FILE

You can use the **gem2rpm** utility to create an RPM **spec** file.

7.4.1. Creating an RPM spec file for a Ruby gem

You can generate an RPM **spec** file for a RubyGems package by using the **gem2rpm** utility.

Prerequisites

- You have the **gem2rpm** utility installed on your system:

```
$ gem install gem2rpm
```

Procedure

1. Download a gem in its latest version and generate the RPM **spec** file for this gem:

```
$ gem2rpm --fetch <gem_name> > <gem_name>.spec
```

2. Edit the generated **spec** file to add the missing information, for example, a license and a changelog.

Additional resources

- Building RPMs

7.4.2. Using custom gem2rpm templates to generate a spec file

gem2rpm templates are standard Embedded Ruby (ERB) files that RPM **spec** files can be generated from. You can edit the template from which the RPM **spec** file is generated instead of editing the generated **spec** file.

Prerequisites

- You have the **gem2rpm** utility installed on your system:

```
$ gem install gem2rpm
```

Procedure

1. Display all **gem2rpm** built-in templates:

```
$ gem2rpm --templates
```

2. Select one of the built-in templates and save it as a custom template:

```
$ gem2rpm -t <template> -T > rubygem-<gem_name>.spec.template
```

   Note that for RHEL 10 Beta, the **fedora-27-rawhide** template is recommended.

3. Edit the template as needed. For more information, see gem2rpm template variables.

4. Generate the **spec** file by using the edited template:

```
$ gem2rpm -t rubygem-<gem_name>.spec.template
<gem_name>-<latest_version>.gem > <gem_name>-GEM.spec
```

Additional resources

- Building RPMs

7.4.3. gem2rpm template variables

The following are the variables included in the **gem2rpm** template for RPM **spec** file generation.

Table 7.3. Variables in the gem2rpm template

Variable	Explanation
package	The **Gem::Package** variable for the gem.
spec	The **Gem::Specification** variable for the gem (the same as **format.spec**).
config	The **Gem2Rpm::Configuration** variable that can redefine default macros or rules used in **spec** template helpers.
runtime_dependencies	The **Gem2Rpm::RpmDependencyList** variable that provides a list of package runtime dependencies.
development_dependencies	The **Gem2Rpm::RpmDependencyList** variable that provides a list of package development dependencies.
tests	The **Gem2Rpm::TestSuite** variable that provides a list of test frameworks allowing their execution.
files	The **Gem2Rpm::RpmFileList** variable that provides an unfiltered list of files in a package.
main_files	The **Gem2Rpm::RpmFileList** variable that provides a list of files suitable for the main package.
doc_files	The **Gem2Rpm::RpmFileList** variable that provides a list of files suitable for the **-doc** subpackage.