Red Hat Enterprise Linux 9

Packaging and distributing software

Packaging software by using the RPM package management system

Last Updated: 2024-03-20

Red Hat Enterprise Linux 9 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 into RPM packages.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

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

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

In the following sections, learn how to build the **Hello World** program written in the C programming language by using either manual or automated building.

2.3.1.1. Manually building the cello software

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

Hello World 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 up automated building for the cello 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.

In the following sections, learn how to byte-compile the **Hello World** program written in Python and how to raw-interpret the **Hello World** program written in Bash.

2.3.2.1. Byte-compiling the pello program

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

The **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 the bello program

The **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

In the following sections, learn how to prepare software for RPM packaging:

- Learn how to patch software.
- Create a **LICENSE** file.
- Learn how to put source code into 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 leave the original source code intact and just 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.

In the following sections, learn how to create a patch for the **Hello World** program written in the C programming language and how to patch this program.

3.1.1. Creating a patch file for the cello program

You can create a patch from the original source code by using the **diff** utility.

Procedure

1. Preserve 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
```

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

3.1.2. Patching the cello program

You can use the **patch** utility to patch software.

Prerequisites

- A patch from the original source code has been created.

Procedure

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;
}
```

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

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

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

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

3.2. CREATING A LICENSE FILE

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.

IMPORTANT

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

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

- Sorce code examples

3.3. PUTTING SOURCE CODE INTO A TARBALL

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

In the following sections, learn how to put each of the three **Hello World** program versions into a gzip-compressed tarball:

- Put the bello program into a tarball .
- Put the pello program into a tarball .
- Put the cello program into a tarball .

3.3.1. Putting the bello program into a tarball

The *bello* project implements **Hello World** in Bash. The implementation contains only the **bello** shell script. Therefore, the resulting **tar.gz** archive has only one file in addition to the **LICENSE** file.

Prerequisites

- Assume that the **0.1** version of the **bello** program is used.

Procedure

1. Put 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. Putting the pello program into a tarball

The *pello* project implements **Hello World** in Python. The implementation contains only the **pello.py** program. Therefore, the resulting **tar.gz** archive has only one file in addition to the **LICENSE** file.

Prerequisites

- Assume that the **0.1.1** version of the **pello** program is used.

Procedure

1. Put 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. Putting the cello program into a tarball

The *cello* project implements **Hello World** in C. The implementation 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.

Procedure

1. Put 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 /tmp/cello-1.0.tar.gz ~/rpmbuild/SOURCES/
```

Additional resources

- Hello World written in C

CHAPTER 4. PACKAGING SOFTWARE

4.1. SETTING UP RPM PACKAGING WORKSPACE

You can set up a directory layout that is the RPM packaging workspace by using the **rpmdev-setuptree** utility.

Prerequisites

- You installed the **rpmdevtools** package, which provides several 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
  ```

The created directories serve the following purposes:

Directory	Purpose
BUILD	When packages are built, various **%buildroot** directories are created here. This is useful for investigating a failed build if the logs output do not provide enough information.
RPMS	Binary RPMs are created here, in subdirectories for different architectures, for example in subdirectories **x86_64** and **noarch**.
SOURCES	Here, the packager puts compressed source code archives and patches. The **rpmbuild** command looks for them here.
SPECS	The packager puts SPEC files here.
SRPMS	When **rpmbuild** is used to build an SRPM instead of a binary RPM, the resulting SRPM is created here.

4.2. WHAT IS A SPEC FILE

You can understand a SPEC file as a recipe that the **rpmbuild** utility uses to build an RPM. A SPEC file provides necessary information to the build system by defining instructions in a series of sections. The sections are defined in the *Preamble* and the *Body* part. The *Preamble* part contains a series of metadata items that are used in the *Body* part. The *Body* part represents the main part of the instructions.

The following sections describe each section of a SPEC file.

4.2.1. Preamble items

The table below presents some of the directives that are used frequently in the *Preamble* section of the RPM SPEC file.

Table 4.1. Items used in the *Preamble* section of the RPM SPEC file

SPEC Directive	Definition
Name	The base name of the package, which should match the SPEC file name.
Version	The upstream version number of the software.
Release	The number of times this version of the software was released. Normally, set the initial value to 1%{?dist}, and increment 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.
License	The license of the software being packaged.
URL	The full URL for more information about the program. Most often this is the upstream project website for the software being packaged.
Source0	Path or URL to the compressed archive of the upstream source code (unpatched, patches are handled elsewhere). This should point to an accessible and reliable storage of the archive, for example, the upstream page and not the packager’s local storage. If needed, more SourceX directives can be added, incrementing the number each time, for example: Source1, Source2, Source3, and so on.
Patch	The name of the first patch to apply to the source code if necessary. The directive can be applied in two ways: with or without numbers at the end of Patch. If no number is given, one is assigned to the entry internally. It is also possible to give the numbers explicitly using Patch0, Patch1, Patch2, Patch3, and so on. These patches can be applied one by one using the %patch0, %patch1, %patch2 macro and so on. The macros are applied within the %prep directive in the *Body* section of the RPM SPEC file. Alternatively, you can use the %autopatch macro which automatically applies all patches in the order they are given in the SPEC file.

SPEC Directive	Definition
BuildArch	If the package is not architecture dependent, for example, if written entirely in an interpreted programming language, set this to **BuildArch: noarch**. If not set, the package 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 for building 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.
ExcludeArch	If a piece of software can not operate on a specific processor architecture, you can exclude that architecture here.
Conflicts	**Conflicts** are inverse to **Requires**. If there is a package matching **Conflicts**, the package cannot be installed independently on whether the **Conflict** tag is on the package that has already been installed or on a package that is going to be installed.
Obsoletes	This directive alters the way updates work depending on whether the **rpm** command is used directly on the command line or the update is performed by an updates or dependency solver. When used on a command line, RPM removes all packages matching obsoletes of packages being installed. When using an update or dependency resolver, packages containing matching **Obsoletes:** are added as updates and replace the matching packages.
Provides	If **Provides** is added to a package, the package can be referred to by dependencies other than its name.

The **Name**, **Version**, and **Release** directives comprise the file name of the RPM package. RPM package maintainers and system administrators often call these three directives **N-V-R** or **NVR**, because RPM package filenames have the **NAME-VERSION-RELEASE** format.

The following example shows how to obtain the **NVR** information for a specific package by querying the **rpm** command.

Example 4.1. Querying rpm to provide the NVR information for the bash package

```
# 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 final marker is **x86_64**, which signals the architecture. Unlike the **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 4.2. Items used in the Body section of the RPM SPEC file

SPEC 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.
%prep	Command or series of commands to prepare the software to be built, for example, unpacking the archive in **Source0**. This directive can contain a shell script.
%build	Command or series of commands for building the software into machine code (for compiled languages) or bytecode (for some interpreted languages).
%install	Command or series of commands for copying the desired build artifacts from the **%builddir** (where the build happens) to the **%buildroot** directory (which contains the directory structure with the files to be packaged). This usually means copying files from **~/rpmbuild/BUILD** to **~/rpmbuild/BUILDROOT** and creating the necessary directories in **~/rpmbuild/BUILDROOT**. This is only run when creating a package, not when the end-user installs the package. See Working with SPEC files for details.
%check	Command or series of commands to test the software. This normally includes things such as unit tests.
%files	The list of files that will be insstalled in the end user's system.
%changelog	A record of changes that have happened to the package between different **Version** or **Release** builds.

4.2.3. Advanced items

The SPEC file can also contain advanced items, such as Scriptlets or Triggers.

They 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

Starting from Red Hat Enterprise Linux 6, the **rpmbuild** program has its own defaults. Overriding these defaults leads to several problems. Therefore, it is not recommended to define your own value of this macro. You can use the **%{buildroot}** macro with the defaults from the **rpmbuild** directory.

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. Hence, RPM can perform text substitutions for you.

An example use is referencing the packaged software *Version* multiple times in a SPEC file. You define *Version* only once in the **%{version}** macro, and use this macro throughout the SPEC file. Every occurrence will be automatically substituted by *Version* that you defined previously.

NOTE

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

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

Evaluating the %{_bindir} and the %{_libexecdir} macros

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

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

One of the commonly-used macros is the **%{?dist}** macro, which signals which distribution is used for the build (distribution tag).

```
# On a RHEL 9.x machine
$ rpm --eval %{?dist}
.el9
```

4.5. WORKING WITH SPEC FILES

To package new software, you must create a SPEC file.

You can create a SPEC file 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 in 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.

The following sections use the three example implementations of the **Hello World!** program:

Software Name	Explanation of example
bello	A program written in a raw interpreted programming language. It demonstrates when the source code does not need to be built, but only needs to be installed. If a pre-compiled binary needs to be packaged, you can also use this method because the binary would also just be a file.
pello	A program written in a byte-compiled interpreted programming language. It demonstrates byte-compiling the source code and installing the bytecode - the resulting pre-optimized files.
cello	A program written in a natively compiled programming language. It demonstrates a common process of compiling the source code into machine code and installing the resulting executables.

The implementations of **Hello World!** are the following:

- bello-0.1.tar.gz
- pello-0.1.2.tar.gz
- cello-1.0.tar.gz (cello-output-first-patch.patch)

As a prerequisite, these implementations need to be placed into the **~/rpmbuild/SOURCES** directory.

For more information about the **Hello World!** program implementations, see What is source code.

In the following section, learn how to work with SPEC files:

- Create a new SPEC file with rpmdev-newspec .
- Modify an original SPEC file for creating RPMs .
- Examine example SPEC files for programs written in bash, Python, and C.

4.5.1. Creating a new SPEC file with rpmdev-newspec

To create a SPEC file for each of the three implementations of the **Hello World!** program by using the **rpmdev-newspec** utility, complete the following steps.

Procedure

1. Go to the **~/rpmbuild/SPECS** directory and use the **rpmdev-newspec** utility:

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

2. Create a SPEC file for each of the three implementations of the Hello World! Program by using the **rpmdev-newspec** utility:

```
$ 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 What is a SPEC file . In the following sections, you will populate particular section in the output files of **rpmdev-newspec**.

NOTE

The **rpmdev-newspec** utility does not use guidelines or conventions specific to any particular Linux distribution. However, this document targets Red Hat Enterprise Linux, therefore the **%{buildroot}** notation is preferred over the **$RPM_BUILD_ROOT** notation when referencing RPM's Buildroot for consistency with all other defined or provided macros throughout the SPEC file.

4.5.2. Modifying an original SPEC file for creating RPMs

To modify the output SPEC file provided by the **rpmdev-newspec** utility for creating RPM packages, complete the following steps.

Prerequisites

- The source code of the particular program has been placed into the **~/rpmbuild/SOURCES/** directory.
- The unpopulated **~/rpmbuild/SPECS/<name>.spec** SPEC file has been created by the **rpmdev-newspec** utility.

Procedure

1. Open the output template of the **~/rpmbuild/SPECS/<name>.spec** file provided by the **rpmdev-newspec** utility:
   a. Populate the first section of the SPEC file that includes the following directives that **rpmdev-newspec** grouped together:

      Name

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

      Version

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

      Release

The **Release** is automatically set to **1%{?dist}**, which is initially **1**. Increase the initial value whenever updating the package without a change in the upstream release **Version**, for example, when including a patch. Reset **Release** to **1** when a new upstream release happens.

Summary

The **Summary** is a short, one-line explanation of what this software is.

b. Populate the **License**, **URL**, and **Source0** directives:

License

The **License** field is the Software License associated with the source code from the upstream release. The exact format for how to label the **License** in your SPEC file varies depending on which specific RPM based Linux distribution guidelines you are following. For example, you can use GPLv3+.

URL

The **URL** field provides a URL to the upstream software website. For consistency, utilize the RPM macro variable of **%{name}**, and use **https://example.com/%{name}**.

Source

The **Source0** field provides URL to the upstream software source code. It must link directly to the specific version of software that is being packaged. Note that the example URLs given in this documentation include hard-coded values that are possible subject to 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, you need to update only one field in the SPEC file.

c. Populate the **BuildRequires**, **Requires** and **BuildArch** directives:

BuildRequires

BuildRequires specifies build-time dependencies for the package.

Requires

Requires specifies run-time dependencies for the package.

BuildArch

This is a software written in an interpreted programming language with no natively compiled extensions. Therefore, add the **BuildArch** directive with the **noarch** value. This tells RPM that this package does not need to be bound to the processor architecture on which it is built.

d. Populate the **%description**, **%prep**, **%build**, **%install**, **%files**, and **%license** directives. You can think of these directives as section headings, because they are directives that can define multi-line, multi-instruction, or scripted tasks to occur.

%description

The **%description** is a longer, fuller description of the software than **Summary**. This directive contains one or more paragraphs.

%prep

The **%prep** section specifies how to prepare the build environment. This usually involves expansion of compressed archives of the source code, application of patches, and, potentially, parsing of information provided in the source code for use in a later portion of the SPEC file. In this section you can use the built-in **%setup -q** macro.

%build

The **%build** section specifies how to build the software.

%install

The **%install** section contains instructions for **rpmbuild** on how to install the software, once it has been built, into the **BUILDROOT** directory.
This 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 the RPM macros without having to hardcode the paths.

%files

The **%files** section specifies the list of files provided by this RPM and their full path location on the end user's system.
Within this section, you can indicate the role of various files 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.

e. The last section, **%changelog**, is a list of datestamped entries for each Version-Release of the package. They log packaging changes, not software changes. Examples of packaging changes: adding a patch, changing the build procedure in the **%build** section.
Start the first line of the **%changelog** section with an ***** 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 **-** character.

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

Additional resources

- An example SPEC file for a program written in bash
- An example SPEC file for a program written in Python
- An example SPEC file for a program written in C
- Building RPMs

4.5.3. An example SPEC file for a program written in bash

You can use the following example SPEC file for the **bello** program that was 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, include 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 a **bash** does not need to be built.

NOTE

For installing **bello** you only need to 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 program written in Python

Example SPEC file for the **pello** program written in the Python programming language follows.

An example SPEC file for the pello program written in Python

```
%global python3_pkgversion 3.11                                  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 needed to be specified manually
BuildRequires:  python%{python3_pkgversion}-setuptools

# Test dependencies needed to be specified manually
# Also 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 supported projects with setup.py
%py3_build                                                   5


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


%check                                                       6
%{pytest}
```

```
# Note that there is no %%files section for the unversioned python module
%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 3.9, either set the macro to its default value **3** or remove the line entirely.

2 When packaging a Python project into RPM, always add the **python-** prefix to the original name of the project. The original 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: **python3-devel** (or **python3.11-devel**) and the relevant projects needed by the specific software that you package, for example, **python3-setuptools** (or **python3.11-setuptools**) or the runtime and testing dependencies needed to run the tests in the **%check** section.

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 3.9 or the **python3.11-** prefix for Python 3.11. You can use the **%{python3_pkgversion}** macro, which evaluates to **3** for the default Python version 3.9 unless you set it to an explicit version, for example, **3.11** (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.

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

4.5.5. An example SPEC file for a program written in C

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 two packages that are needed to perform the compilation build process:
  - The **gcc** package
  - The **make** package
- 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 a **Makefile** for the **cello** program was written, therefore, you can use the GNU make command provided by the **rpmdev-newspec** utility. However, you need to 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 that is provided by the **rpmdev-newspec** command. This is possible because the **Makefile** 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. This command expects a certain directory and file structure, 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 two main use cases are:

- Building source RPMs
- Building binary RPM
  - Rebuilding a binary RPM from a source RPM
  - Building a binary RPM from the SPEC file
  - Building a binary RPM from a source RPM

In the following sections, learn how to build an RPM after you have created a SPEC file for a program:

- Build source RPMs.
- Rebuild a binary RPM from a source RPM .
- Build a binary RPM from the SPEC file .
- Build binary RPMs from source RPMs .

4.6.1. Building source RPMs

To build a source RPM, complete the following steps.

Prerequisites

- A SPEC file for the program that you want to package must already exist.

Procedure

- Run the **rpmbuild** command with the specified SPEC file:

  ```
  $ rpmbuild -bs specfile
  ```

  Replace *specfile* with the name of the SPEC file. The **-bs** option stands for the build source.

Verification steps

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

Example 4.2. Building source RPMs for bello, pello, and cello.

The following is an example of building source RPMs for the **bello**, **pello**, and **cello** projects.

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

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

2. Run the **rpmbuild** command with the specified SPEC file:

   ```
   $ 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
   ```

Additional resources

- Working with SPEC files
- Creating a new SPEC file with rpmdev-newspec
- Modifying an original SPEC file for creating RPMs

4.6.2. Rebuilding a binary RPM from a source RPM

To rebuild a binary RPM from a source RPM (SRPM), complete the following steps.

Procedure

- To rebuild **bello**, **pello**, and **cello** from their SRPMs, run:

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

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

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

NOTE

Invoking **rpmbuild --rebuild** involves:

- Installing the contents of the SRPM (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.

To retain the SPEC file and the source code after building, complete either of the following steps:

- When building an RPM, use the **rpmbuild** command with the **--recompile** option instead of the **--rebuild** option.
- Install the 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%]
```

The output generated when creating a 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 the **~/rpmbuild/RPMS/YOURARCH** directory where **YOURARCH** is your architecture or in the **~/rpmbuild/RPMS/noarch/** directory, if the package is not architecture-specific.

4.6.3. Building a binary RPM from the SPEC file

To build **bello**, **pello**, and **cello** binary RPMs from their SPEC files, complete the following steps.

Procedure

- Run the **rpmbuild** command with the **bb** option:

```
$ rpmbuild -bb ~/rpmbuild/SPECS/bello.spec

$ rpmbuild -bb ~/rpmbuild/SPECS/pello.spec

$ rpmbuild -bb ~/rpmbuild/SPECS/cello.spec
```

4.6.4. Building binary RPMs from source RPMs

You build any kind of RPM from a source RPM. To do so, complete the following steps.

Procedure

- Run the **rpmbuild** command with one of the following options and with the source package specified:

  ```
  # rpmbuild {-ra|-rb|-rp|-rc|-ri|-rl|-rs} [rpmbuild-options] source-package
  ```

  Replace *source-package* with the name of the source RPM.

Additional resources

- **rpmbuild(8)** man page

4.7. CHECKING RPMS FOR SANITY

After creating a package, you must check the quality of the package.

The main tool for checking package quality is **rpmlint**.

The **rpmlint** tool does the following:

- Improves RPM maintainability.
- Enables sanity checking by performing static analysis of the RPM.
- Enables error checking by performing static analysis of the RPM.

The **rpmlint** tool can check binary RPMs, source RPMs (SRPMs), and SPEC files. Therefore, this tool is useful for all stages of packaging.

Note that **rpmlint** has strict guidelines. Therefore, it is sometimes acceptable to skip some of its errors and warnings, as shown in the following examples.

NOTE

In the examples described in the following sections, **rpmlint** is run without any options, which produces a non-verbose output. For detailed explanations of each error or warning, run **rpmlint -i** instead.

4.7.1. Checking bello for sanity

In the following sections, investigate possible warnings and errors that can occur when checking an RPM for sanity on the example of the **bello** SPEC file and **bello** binary RPM.

4.7.1.1. Checking the bello SPEC File for sanity

Inspect the outputs of the following examples to learn how to check a **bello** SPEC file for sanity.

Example 4.3. Output of running the rpmlint command on the SPEC file for bello

The following is an example output of the **rpmlint** command run on the **bello** SPEC file.

```
$ rpmlint bello.spec
bello.spec: W: invalid-url Source0: https://www.example.com/bello/releases/bello-0.1.tar.gz HTTP
Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

For **bello.spec**, there is only one warning. The **invalid-url Source0** warning means that the URL listed in the **Source0** directive is unreachable. This is expected, because the specified **example.com** URL does not exist. Assuming that this URL will be valid in the future, you can ignore this warning.

Example 4.4. Output of running therpmlint command on the SRPM for bello

The following is an example output of the **rpmlint** command run on the **bello** source RPM (SRPM).

```
$ rpmlint ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
bello.src: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.src: W: invalid-url Source0: https://www.example.com/bello/releases/bello-0.1.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

For the **bello** SRPM, there is a new **invalid-url URL** warning. This warning means that the URL specified in the **URL** directive is unreachable. Assuming that this URL will be valid in the future, you can ignore this warning.

4.7.1.2. Checking the bello binary RPM for sanity

When checking binary RPMs, the **rpmlint** command checks the following items:

- Documentation
- Manual pages
- Consistent use of the filesystem hierarchy standard

Inspect the outputs of the following example to learn how to check a **bello** binary RPM for sanity.

Example 4.5. Output of running therpmlint command on the binary RPM for bello

The following is an example output of the **rpmlint** command run on the **bello** binary RPM.

```
$ rpmlint ~/rpmbuild/RPMS/noarch/bello-0.1-1.el8.noarch.rpm
bello.noarch: W: invalid-url URL: https://www.example.com/bello HTTP Error 404: Not Found
bello.noarch: W: no-documentation
bello.noarch: W: no-manual-page-for-binary bello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

The **no-documentation** and **no-manual-page-for-binary** warnings mean that the RPM has no documentation or manual pages, because you did not provide any. Apart from the output warnings, the RPM passed **rpmlint** checks.

4.7.2. Checking pello for sanity

In the following sections, investigate possible warnings and errors that can occur when checking RPM sanity on the example of the **pello** SPEC file and **pello** binary RPM.

4.7.2.1. Checking the pello SPEC File for sanity

Inspect the outputs of the following examples to learn how to check a **pello** SPEC file for sanity.

Example 4.6. Output of running therpmlint command on the SPEC file for pello

The following is an example output of the **rpmlint** command run on the **pello** SPEC file.

```
$ rpmlint pello.spec
pello.spec:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.spec:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.spec:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.spec:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.spec:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.spec: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.2.tar.gz
HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 5 errors, 1 warnings.
```

- The **invalid-url Source0** warning means that the URL listed in the **Source0** directive is unreachable. This is expected, because the specified **example.com** URL does not exist. Assuming that this URL will be valid in the future, you can ignore this warning.
- The **hardcoded-library-path** errors suggest using the **%{_libdir}** macro instead of hard-coding the library path. For the sake of this example, you can safely ignore these errors. However, for packages going into production, check all errors carefully.

Example 4.7. Output of running therpmlint command on the SRPM for pello

The following is an example output of the **rpmlint** command run on the **pello** source RPM (SRPM).

```
$ rpmlint ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
pello.src: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.src:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pello.src:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pello.src:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pello.src:43: E: hardcoded-library-path in /usr/lib/%{name}/
pello.src:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pello.src: W: invalid-url Source0: https://www.example.com/pello/releases/pello-0.1.2.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 5 errors, 2 warnings.
```

The **invalid-url URL** error is about the **URL** directive, which is unreachable. Assuming that this URL will be valid in the future, you can ignore this warning.

4.7.2.2. Checking the pello binary RPM for sanity

When checking binary RPMs, the **rpmlint** command checks the following items:

- Documentation

- Manual pages
- Consistent use of the Filesystem Hierarchy Standard

Inspect the outputs of the following example to learn how to check a **pello** binary RPM for sanity.

Example 4.8. Output of running therpmlint command on the binary RPM for pello

The following is an example output of the **rpmlint** command run on the **pello** binary RPM.

```
$ rpmlint ~/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
pello.noarch: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not Found
pello.noarch: W: only-non-binary-in-usr-lib
pello.noarch: W: no-documentation
pello.noarch: E: non-executable-script /usr/lib/pello/pello.py 0644L /usr/bin/env
pello.noarch: W: no-manual-page-for-binary pello
1 packages and 0 specfiles checked; 1 errors, 4 warnings.
```

- The **no-documentation** and **no-manual-page-for-binary** warnings mean that the RPM has no documentation or manual pages, because you did not provide any.
- The **only-non-binary-in-usr-lib** warning means that you provided only non-binary artifacts in the **/usr/lib/** directory. This directory is normally reserved for shared object files, which are binary files. Therefore, **rpmlint** expects at least one or more files in **/usr/lib/** to be binary. This is an example of an **rpmlint** check for compliance with Filesystem Hierarchy Standard. Normally, use RPM macros to ensure the correct placement of files. For the sake of this example, you can safely ignore this warning.
- The **non-executable-script** error means that the **/usr/lib/pello/pello.py** file has no execute permissions. The **rpmlint** tool expects the file to be executable, because the file contains the shebang. For the purpose of this example, you can leave this file without executing permissions and ignore this error.

Apart from the output warnings and errors, the RPM passed **rpmlint** checks.

4.7.3. Checking cello for sanity

In the following sections, investigate possible warnings and errors that can occur when checking RPM sanity on the example of the **cello** SPEC file and **cello** binary RPM.

4.7.3.1. Checking the cello SPEC File for sanity

Inspect the outputs of the following examples to learn how to check a **cello** SPEC file for sanity.

Example 4.9. Output of running therpmlint command on the SPEC file for cello

The following is an example output of the **rpmlint** command run on the **cello** SPEC file.

```
$ rpmlint ~/rpmbuild/SPECS/cello.spec
/home/admiller/rpmbuild/SPECS/cello.spec: W: invalid-url Source0:
https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

For **cello.spec**, there is only one warning. The **invalid-url Source0** warning means that the URL listed in the **Source0** directive is unreachable. This is expected, because the specified **example.com** URL does not exist. Assuming that this URL will be valid in the future, you can ignore this warning.

Example 4.10. Output of running therpmlint command on the SRPM for cello

The following is an example output of the **rpmlint** command run on the **cello** source RPM (SRPM.)

```
$ rpmlint ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
cello.src: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.src: W: invalid-url Source0: https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP
Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

For the **cello** SRPM, there is a new **invalid-url URL** warning. This warning means that the URL specified in the **URL** directive is unreachable. Assuming that this URL will be valid in the future, you can ignore this warning.

4.7.3.2. Checking the cello binary RPM for sanity

When checking binary RPMs, the **rpmlint** command checks the following items:

- Documentation
- Manual pages
- Consistent use of the filesystem hierarchy standard

Inspect the outputs of the following example to learn how to check a **cello** binary RPM for sanity.

Example 4.11. Output of running therpmlint command on the binary RPM for cello

The following is an example output of the **rpmlint** command run on the **cello** binary RPM.

```
$ rpmlint ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el8.x86_64.rpm
cello.x86_64: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
cello.x86_64: W: no-documentation
cello.x86_64: W: no-manual-page-for-binary cello
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

The **no-documentation** and **no-manual-page-for-binary** warnings mean that the RPM has no documentation or manual pages, because you did not provide any.

Apart from the output warnings, the RPM passed **rpmlint** checks.

4.8. LOGGING RPM ACTIVITY TO SYSLOG

Any RPM activity or transaction can be logged by 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.

To see the updates on RPM activity, complete the following steps.

Procedure

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

4.9. EXTRACTING RPM CONTENT

In particular cases, for example, if a package required by RPM is damaged, it is necessary 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.

To convert an RPM payload to a tar archive by using the **rpm2archive** utility, complete the following steps.

Procedure

- Run the **rpm2archive** command on the RPM file:

  ```
  $ rpm2archive filename.rpm
  ```

  Replace *filename* with the name of the **.rpm** file.

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

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

CHAPTER 5. ADVANCED TOPICS

This section covers topics that are beyond the scope of the introductory tutorial but are useful in real-world RPM packaging.

5.1. SIGNING RPM PACKAGES

You can sign RPM packages to ensure that no third party can alter their content. To add an additional layer of security, use the HTTPS protocol when downloading the package.

You can sign a package by using the **--addsign** option provided by the **rpm-sign** package.

Prerequisites

- You have created a GNU Privacy Guard (GPG) key as described in Creating a GPG key.

5.1.1. Creating a GPG key

Use the following procedure to create a GNU Privacy Guard (GPG) key required for signing packages.

Procedure

1. Generate a GPG key pair:

```
# gpg --gen-key
```

2. Check the generated key pair:

```
# gpg --list-keys
```

3. Export the public key:

```
# gpg --export -a '<Key_name>' > RPM-GPG-KEY-pmanager
```

   Replace <Key_name> with the real key name that you have selected.

4. Import the exported public key into an RPM database:

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

5.1.2. Configuring RPM to sign a package

To be able to sign an RPM package, you need to specify the **%_gpg_name** RPM macro.

The following procedure describes how to configure RPM for signing a package.

Procedure

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

```
%_gpg_name Key ID
```

Replace *Key ID* with the GNU Privacy Guard (GPG) key ID that you will use to sign a package. A valid GPG key ID value is either a full name or email address of the user who created the key.

5.1.3. Adding a signature to an RPM package

The most usual case is when a package is built without a signature. The signature is added just before the release of the package.

To add a signature to an RPM package, use the **--addsign** option provided by the **rpm-sign** package.

Procedure

- Add a signature to a package:

  ```
  $ rpm --addsign package-name.rpm
  ```

  Replace *package-name* with the name of an RPM package you want to sign.

  NOTE

  You must enter the password to unlock the secret key for the signature.

5.2. MORE ON MACROS

This section covers selected built-in RPM Macros. For an exhaustive list of such macros, see RPM Documentation.

5.2.1. Defining your own macros

The following section describes how to create a custom macro.

Procedure

- Include the following line in the RPM SPEC file:

  ```
  %global <name>[(opts)] <body>
  ```

All whitespace surrounding **<body>** is removed. Name may be composed of alphanumeric characters, and the character _ and must be at least 3 characters in length. Inclusion of the **(opts)** field is optional:

- **Simple** macros do not contain the **(opts)** field. In this case, only recursive macro expansion is performed.
- **Parametrized** macros contain the **(opts)** field. The **opts** string between parentheses is passed to **getopt(3)** for **argc/argv** processing at the beginning of a macro invocation.

NOTE

Older RPM SPEC files use the **%define <name> <body>** macro pattern instead. The differences between **%define** and **%global** macros are as follows:

- **%define** has local scope. It applies to a specific part of a SPEC file. The body of a **%define** macro is expanded when used.
- **%global** has global scope. It applies to an entire SPEC file. The body of a **%global** macro is expanded at definition time.

IMPORTANT

Macros are evaluated even if they are commented out or the name of the macro is given into the **%changelog** section of the SPEC file. To comment out a macro, use **%%**. For example: **%%global**.

Additional resources

- Macro syntax

5.2.2. Using the %setup macro

This section describes how to build packages with source code tarballs using different variants of the **%setup** macro. Note that the macro variants can be combined. The **rpmbuild** output illustrates standard behavior of the **%setup** macro. At the beginning of each phase, the macro outputs **Executing(%…)**, as shown in the below example.

Example 5.1. Example %setup macro output

```
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.DhddsG
```

The shell output is set with **set -x** enabled. To see the content of **/var/tmp/rpm-tmp.DhddsG**, use the **--debug** option because **rpmbuild** deletes temporary files after a successful build. This displays the setup of environment variables followed by for example:

```
cd '/builddir/build/BUILD'
rm -rf 'cello-1.0'
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xof -
STATUS=$?
if [ $STATUS -ne 0 ]; then
  exit $STATUS
fi
cd 'cello-1.0'
/usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
```

The **%setup** macro:

- Ensures that we are working in the correct directory.
- Removes residues of previous builds.
- Unpacks the source tarball.

- Sets up some default privileges.

5.2.2.1. Using the %setup -q macro

The **-q** option limits the verbosity of the **%setup** macro. Only **tar -xof** is executed instead of **tar -xvvof**. Use this option as the first option.

5.2.2.2. Using the %setup -n macro

The **-n** option is used to specify the name of the directory from expanded tarball.

This is used in cases when the directory from expanded tarball has a different name from what is expected (**%{name}-%{version}**), which can lead to an error of the **%setup** macro.

For example, if the package name is **cello**, but the source code is archived in **hello-1.0.tgz** and contains the **hello/** directory, the SPEC file content needs to be as follows:

```
Name: cello
Source0: https://example.com/%{name}/release/hello-%{version}.tar.gz
…
%prep
%setup -n hello
```

5.2.2.3. Using the %setup -c macro

The **-c** option is used if the source code tarball does not contain any subdirectories and after unpacking, files from an archive fills the current directory.

The **-c** option then creates the directory and steps into the archive expansion as shown below:

```
/usr/bin/mkdir -p cello-1.0
cd 'cello-1.0'
```

The directory is not changed after archive expansion.

5.2.2.4. Using the %setup -D and %setup -T macros

The **-D** option disables deleting of source code directory, and is particularly useful if the **%setup** macro is used several times. With the **-D** option, the following lines are not used:

```
rm -rf 'cello-1.0'
```

The **-T** option disables expansion of the source code tarball by removing the following line from the script:

```
/usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xvvof -
```

5.2.2.5. Using the %setup -a and %setup -b macros

The **-a** and **-b** options expand specific sources:

- The **-b** option stands for **before**. This option expands specific sources before entering the working directory.

- The **-a** option stands for **after**. This option expands those sources after entering. Their arguments are source numbers from the SPEC file preamble.

In the following example, the **cello-1.0.tar.gz** archive contains an empty **examples** directory. The examples are shipped in a separate **examples.tar.gz** tarball and they expand into the directory of the same name. In this case, use **-a 1** if you want to expand **Source1** after entering the working directory:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: examples.tar.gz
…
%prep
%setup -a 1
```

In the following example, examples are provided in a separate **cello-1.0-examples.tar.gz** tarball, which expands into **cello-1.0/examples**. In this case, use **-b 1** to expand **Source1** before entering the working directory:

```
Source0: https://example.com/%{name}/release/%{name}-%{version}.tar.gz
Source1: %{name}-%{version}-examples.tar.gz
…
%prep
%setup -b 1
```

5.2.3. Common RPM macros in the %files section

The following table lists advanced RPM Macros that are needed in the **%files** section of a SPEC file.

Table 5.1. Advanced RPM Macros in the%files section

Macro	Definition
%license	The **%license** macro identifies the file listed as a**LICENSE** file and it will be installed and labeled as such by RPM. Example: **%license LICENSE**.
%doc	The **%doc** macro identifies a file listed as documentation and it will be installed and labeled as such by RPM. The %doc macro is used for documentation about the packaged software and also for code examples and various accompanying items. If code examples are included, care must be taken to remove executable mode from the file. Example: **%doc README**
%dir	The **%dir** macro ensures that the path is a directory owned by this RPM. This is important so that the RPM file manifest accurately knows what directories to clean up on uninstall. Example: **%dir %{_libdir}/%{name}**
%config(noreplace)	The **%config(noreplace)** macro ensures that the following file is a configuration file and therefore should not be overwritten (or replaced) on a package install or update if the file has been modified from the original installation checksum. If there is a change, the file will be created with **.rpmnew** appended to the end of the filename upon upgrade or install so that the pre-existing or modified file on the target system is not modified. Example: **%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

5.2.4. Displaying the built-in macros

Red Hat Enterprise Linux provides multiple built-in RPM macros.

Procedure

1. To display all built-in RPM macros, run:

   ```
   rpm --showrc
   ```

   NOTE

   The output is quite sizeable. To narrow the result, use the command above with the **grep** command.

2. To find information about the RPMs macros for your system's version of RPM, run:

   ```
   rpm -ql rpm
   ```

   NOTE

   RPM macros are the files titled **macros** in the output directory structure.

5.2.5. RPM distribution macros

Different distributions provide different sets of recommended RPM macros based on the language implementation of the software being packaged or the specific guidelines of the distribution.

The sets of recommended RPM macros are often provided as RPM packages, ready to be installed with the **dnf** package manager.

Once installed, the macro files can be found in the **/usr/lib/rpm/macros.d/** directory.

Procedure

- To display the raw RPM macro definitions, run:

  ```
  rpm --showrc
  ```

The above output displays the raw RPM macro definitions.

- To determine what a macro does and how it can be helpful when packaging RPMs, run the **rpm --eval** command with the name of the macro used as its argument:

  ```
  rpm --eval %{_MACRO}
  ```

Additional resources

- **rpm** man page

5.2.6. Creating custom macros

You can override the distribution macros in the **~/.rpmmacros** file with your custom macros. Any changes that you make affect every build on your machine.

WARNING

Defining any new macros in the **~/.rpmmacros** file is not recommended. Such macros would not be present on other machines, where users may want to try to rebuild your package.

Procedure

- To override a macro, run:

```
%_topdir /opt/some/working/directory/rpmbuild
```

You can create the directory from the example above, including all subdirectories through the **rpmdev-setuptree** utility. The value of this macro is by default **~/rpmbuild**.

```
%_smp_mflags -l3
```

The macro above is often used to pass to Makefile, for example **make %{?_smp_mflags}**, and to set a number of concurrent processes during the build phase. By default, it is set to **-jX**, where **X** is a number of cores. If you alter the number of cores, you can speed up or slow down a build of packages.

5.3. EPOCH, SCRIPTLETS AND TRIGGERS

This section covers **Epoch**, **Scriptlets**, and **Triggers**, which represent advanced directives for RMP SPEC files.

All these directives influence not only the SPEC file, but also the end machine on which the resulting RPM is installed.

5.3.1. The Epoch directive

The **Epoch** directive enables to define weighted dependencies based on version numbers.

If this directive is not listed in the RPM SPEC file, the **Epoch** directive is not set at all. This is contrary to common belief that not setting **Epoch** results in an **Epoch** of 0. However, the **dnf** utility treats an unset **Epoch** as the same as an **Epoch** of 0 for the purposes of depsolving.

However, listing **Epoch** in a SPEC file is usually omitted because in majority of cases introducing an **Epoch** value skews the expected RPM behavior when comparing versions of packages.

Example 5.2. Using Epoch

If you install the **foobar** package with **Epoch: 1** and **Version: 1.0**, and someone else packages **foobar** with **Version: 2.0** but without the **Epoch** directive, the new version will never be considered an update. The reason being that the **Epoch** version is preferred over the traditional **Name-Version-Release** marker that signifies versioning for RPM Packages.

■

Using of **Epoch** is thus quite rare. However, **Epoch** is typically used to resolve an upgrade ordering issue. The issue can appear as a side effect of upstream change in software version number schemes or versions incorporating alphabetical characters that cannot always be compared reliably based on encoding.

5.3.2. Scriptlets directives

Scriptlets are a series of RPM directives that are executed before or after packages are installed or deleted.

Use **Scriptlets** only for tasks that cannot be done at build time or in an start up script.

A set of common **Scriptlet** directives exists. They are similar to the SPEC file section headers, such as **%build** or **%install**. They are defined by multi-line segments of code, which are often written as a standard POSIX shell script. However, they can also be written in other programming languages that RPM for the target machine’s distribution accepts. RPM Documentation includes an exhaustive list of available languages.

The following table includes **Scriptlet** directives listed in their execution order. Note that a package containing the scripts is installed between the **%pre** and **%post** directive, and it is uninstalled between the **%preun** and **%postun** directive.

Table 5.2. Scriptlet directives

Directive	Definition
%pretrans	Scriptlet that is executed just before installing or removing any package.
%pre	Scriptlet that is executed just before installing the package on the target system.
%post	Scriptlet that is executed just after the package was installed on the target system.
%preun	Scriptlet that is executed just before uninstalling the package from the target system.
%postun	Scriptlet that is executed just after the package was uninstalled from the target system.
%posttrans	Scriptlet that is executed at the end of the transaction.

5.3.3. Turning off a scriptlet execution

The following procedure describes how to turn off the execution of any scriptlet using the **rpm** command together with the **--no_scriptlet_name_** option.

Procedure

- For example, to turn off the execution of the **%pretrans** scriptlets, run:

  ```
  # rpm --nopretrans
  ```

  You can also use the **-- noscripts** option, which is equivalent to all of the following:

- **--nopre**
- **--nopost**
- **--nopreun**
- **--nopostun**
- **--nopretrans**
- **--noposttrans**

Additional resources

- **rpm(8)** man page.

5.3.4. Scriptlets macros

The **Scriptlets** directives also work with RPM macros.

The following example shows the use of systemd scriptlet macro, which ensures that systemd is notified about a new unit file.

```
$ rpm --showrc | grep systemd
-14: __transaction_systemd_inhibit      %{__plugindir}/systemd_inhibit.so
-14: _journalcatalogdir /usr/lib/systemd/catalog
-14: _presetdir /usr/lib/systemd/system-preset
-14: _unitdir   /usr/lib/systemd/system
-14: _userunitdir       /usr/lib/systemd/user
/usr/lib/systemd/systemd-binfmt %{?*} >/dev/null 2>&1 || :
/usr/lib/systemd/systemd-sysctl %{?*} >/dev/null 2>&1 || :
-14: systemd_post
-14: systemd_postun
-14: systemd_postun_with_restart
-14: systemd_preun
-14: systemd_requires
Requires(post): systemd
Requires(preun): systemd
Requires(postun): systemd
-14: systemd_user_post  %systemd_post --user --global %{?*}
-14: systemd_user_postun        %{nil}
-14: systemd_user_postun_with_restart   %{nil}
-14: systemd_user_preun
systemd-sysusers %{?*} >/dev/null 2>&1 || :
echo %{?*} | systemd-sysusers - >/dev/null 2>&1 || :
systemd-tmpfiles --create %{?*} >/dev/null 2>&1 || :

$ rpm --eval %{systemd_post}

if [ $1 -eq 1 ] ; then
        # Initial installation
        systemctl preset  >/dev/null 2>&1 || :
fi

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

```
systemctl daemon-reload >/dev/null 2>&1 || :

$ rpm --eval %{systemd_preun}

if [ $1 -eq 0 ] ; then
        # Package removal, not upgrade
        systemctl --no-reload disable  > /dev/null 2>&1 || :
        systemctl stop  > /dev/null 2>&1 || :
fi
```

5.3.5. The Triggers directives

Triggers are RPM directives which provide a method for interaction during package installation and uninstallation.

WARNING

Triggers may be executed at an unexpected time, for example on update of the containing package. **Triggers** are difficult to debug, therefore they need to be implemented in a robust way so that they do not break anything when executed unexpectedly. For these reasons, Red Hat recommends to minimize the use of **Triggers**.

The order of execution on a single package upgrade and the details for each existing **Triggers** are listed below:

```
all-%pretrans
…
any-%triggerprein (%triggerprein from other packages set off by new install)
new-%triggerprein
new-%pre      for new version of package being installed
…           (all new files are installed)
new-%post     for new version of package being installed

any-%triggerin (%triggerin from other packages set off by new install)
new-%triggerin
old-%triggerun
any-%triggerun (%triggerun from other packages set off by old uninstall)

old-%preun    for old version of package being removed
…           (all old files are removed)
old-%postun   for old version of package being removed

old-%triggerpostun
any-%triggerpostun (%triggerpostun from other packages set off by old un
            install)
…
all-%posttrans
```

The above items are found in the **/usr/share/doc/rpm-4.*/triggers** file.

5.3.6. Using non-shell scripts in a SPEC file

The **-p** scriptlet option in a SPEC file enables the user to invoke a specific interpreter instead of the default shell scripts interpreter (**-p /bin/sh**).

The following procedure describes how to create a script, which prints out a message after installation of the **pello.py** program:

Procedure

1. Open the **pello.spec** file.
2. Find the following line:

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

3. Under the above line, insert:

```
%post -p /usr/bin/python3
print("This is {} code".format("python"))
```

4. Build your package as described in Building RPMs.
5. Install your package:

```
# dnf install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
```

6. Check the output message after the installation:

```
Installing       : pello-0.1.2-1.el8.noarch                           1/1
Running scriptlet: pello-0.1.2-1.el8.noarch                           1/1
This is python code
```

NOTE

To use a Python 3 script, include the following line under **install -m** in a SPEC file:

```
%post -p /usr/bin/python3
```

To use a Lua script, include the following line under **install -m** in a SPEC file:

```
%post -p <lua>
```

This way, you can specify any interpreter in a SPEC file.

5.4. RPM CONDITIONALS

RPM Conditionals enable conditional inclusion of various sections of the SPEC file.

Conditional inclusions usually deal with:

- Architecture-specific sections

- Operating system-specific sections
- Compatibility issues between various versions of operating systems
- Existence and definition of macros

5.4.1. RPM conditionals syntax

RPM conditionals use the following syntax:

If *expression* is true, then do some action:

```
%if expression
…
%endif
```

If *expression* is true, then do some action, in other case, do another action:

```
%if expression
…
%else
…
%endif
```

5.4.2. The %if conditionals

The following examples shows the usage of **%if** RPM conditionals.

Example 5.3. Using the %if conditional to handle compatibility between Red Hat Enterprise Linux 8 and other operating systems

```
%if 0%{?rhel} == 8
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

This conditional handles compatibility between RHEL 8 and other operating systems in terms of support of the AS_FUNCTION_DESCRIBE macro. If the package is built for RHEL, the **%rhel** macro is defined, and it is expanded to RHEL version. If its value is 8, meaning the package is build for RHEL 8, then the references to AS_FUNCTION_DESCRIBE, which is not supported by RHEL 8, are deleted from autoconfig scripts.

Example 5.4. Using the %if conditional to handle definition of macros

```
%define ruby_archive %{name}-%{ruby_version}
%if 0%{?milestone:1}%{?revision:1} != 0
%define ruby_archive %{ruby_archive}-%{?milestone}%{?!milestone:%{?revision:r%{revision}}}
%endif
```

This conditional handles definition of macros. If the **%milestone** or the **%revision** macros are set, the **%ruby_archive** macro, which defines the name of the upstream tarball, is redefined.

5.4.3. Specialized variants of %if conditionals

The **%ifarch** conditional, **%ifnarch** conditional and **%ifos** conditional are specialized variants of the **%if** conditionals. These variants are commonly used, hence they have their own macros.

The %ifarch conditional

The **%ifarch** conditional is used to begin a block of the SPEC file that is architecture-specific. It is followed by one or more architecture specifiers, each separated by commas or whitespace.

Example 5.5. An example use of the %ifarch conditional

```
%ifarch i386 sparc
…
%endif
```

All the contents of the SPEC file between **%ifarch** and **%endif** are processed only on the 32-bit AMD and Intel architectures or Sun SPARC-based systems.

The %ifnarch conditional

The **%ifnarch** conditional has a reverse logic than **%ifarch** conditional.

Example 5.6. An example use of the %ifnarch conditional

```
%ifnarch alpha
…
%endif
```

All the contents of the SPEC file between **%ifnarch** and **%endif** are processed only if not done on a Digital Alpha/AXP-based system.

The %ifos conditional

The **%ifos** conditional is used to control processing based on the operating system of the build. It can be followed by one or more operating system names.

Example 5.7. An example use of the %ifos conditional

```
%ifos linux
…
%endif
```

All the contents of the SPEC file between **%ifos** and **%endif** are processed only if the build was done on a Linux system.

5.5. PACKAGING PYTHON 3 RPMS

You can install Python packages on your system either from the upstream PyPI repository using the **pip** installer, or using the DNF package manager. DNF uses the RPM package format, which offers more downstream control over the software.

The packaging format of native Python packages is defined by Python Packaging Authority (PyPA) Specifications. Most Python projects use 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. For more information about emerging packaging standards, see pyproject-rpm-macros.

This chapter describes how to package a Python project that uses **setup.py** into an RPM package. This approach 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 content of RPM packages with the rest of the operating system.
- You can execute tests during the build process.

5.5.1. SPEC file description for a Python package

A SPEC file contains instructions that the **rpmbuild** utility uses to build an RPM. The instructions are included in a series of sections. A SPEC file has two main parts in which the sections are defined:

- Preamble (contains a series of metadata items that are used in the Body)
- Body (contains the main part of the instructions)

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

IMPORTANT

A name of any RPM package of a Python library must always include the **python3-** or **python3.11-** prefix.

Other specifics are shown in the following SPEC file example for the **python3*-pello** package. For description of such specifics, see the notes below the example.

An example SPEC file for the pello program written in Python

```
%global python3_pkgversion 3.11                         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 needed to be specified manually
BuildRequires:  python%{python3_pkgversion}-setuptools
```

```
# Test dependencies needed to be specified manually
# Also 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 supported projects with setup.py
%py3_build                                                5


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


%check                                                   6
%{pytest}


# Note that there is no %%files section for the unversioned python module
%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 3.9, either set the macro to its default value **3** or remove the line entirely.

2. When packaging a Python project into RPM, always add the **python-** prefix to the original name of the project. The original 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:

python3-devel (or **python3.11-devel**) and the relevant projects needed by the specific software that you package, for example, **python3-setuptools** (or **python3.11-setuptools**) or the runtime and testing dependencies needed to run the tests in the **%check** section.

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 3.9 or the **python3.11-** prefix for Python 3.11. You can use the **%{python3_pkgversion}** macro, which evaluates to **3** for the default Python version 3.9 unless you set it to an explicit version, for example, **3.11** (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.

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

5.5.2. Common macros for Python 3 RPMs

In a SPEC file, always use the macros that are described in the following *Macros for Python 3 RPMs* table 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 (see Section 5.5.1, "SPEC file description for a 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 5.3. 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 **3.11** to use Python 3.11
%{python3}	/usr/bin/python3	The Python 3 interpreter
%{python3_version}	3.9	The major.minor version of the Python 3 interpreter
%{python3_sitelib}	/usr/lib/python3.9/site-packages	The location where pure-Python modules are installed
%{python3_sitearch}	/usr/lib64/python3.9/site-packages	The location where modules containing architecture-specific extension modules are installed
%py3_build		Runs the **setup.py build** command with arguments suitable for an RPM package
%py3_install		Runs the **setup.py install** command with arguments suitable for an RPM package

Macro	Normal Definition	Description
%{py3_shebang_flags}	s	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

5.5.3. Using automatically generated dependencies for Python RPMs

The following procedure describes how to use automatically generated dependencies when packaging a Python project as an RPM.

Prerequisites

- A SPEC file for the RPM exists. For more information, see SPEC file description for a Python package.

Procedure

1. Make sure that one of the following directories containing upstream-provided metadata is included in the resulting RPM:
   - **.dist-info**
   - **.egg-info**
     The RPM build process automatically generates virtual **pythonX.Ydist** provides from these directories, for example:

     ```
     python3.9dist(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. For example, a generated requirements tag might look as follows:

     ```
     Requires: python3.9dist(requests)
     ```

2. Inspect the generated requires.
3. To remove some of the generated requires, use one of the following approaches:
   a. Modify the upstream-provided metadata in the **%prep** section of the SPEC file.

   b. Use automatic filtering of dependencies described in the upstream documentation.

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

Additional resources

- Automatically generated dependencies

5.6. HANDLING INTERPRETER DIRECTIVES IN PYTHON SCRIPTS

In Red Hat Enterprise Linux 9, 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/python3.9
#!/usr/bin/python3.11
```

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, such as:

```
#!/usr/bin/python
```

or

```
#!/usr/bin/env python
```

5.6.1. Modifying interpreter directives in Python scripts

Use the following procedure to modify interpreter directives in Python scripts that cause build errors at RPM build time.

Prerequisites

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

Procedure

- To modify interpreter directives, complete one of the following tasks:
    - 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 will 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.

- Apply the **pathfix.py** script from the **python3-devel** package:

  ```
  # pathfix.py -pn -i %{python3} PATH …
  ```

  You can specify multiple paths. If a ***PATH*** is a directory, **pathfix.py** recursively scans for any Python scripts matching the pattern **^[a-zA-Z0-9_]+\.py$**, not only those with an ambiguous interpreter directive. Add the command above to the **%prep** section or at the end of the **%install** section.

- Modify the packaged Python scripts so that they conform to the expected format. For this purpose, you can use the **pathfix.py** script outside the RPM build process, too. When running **pathfix.py** outside an RPM build, replace **%{python3}** from the preceding example with a path for the interpreter directive, such as **/usr/bin/python3** or **/usr/bin/python3.11**.

Additional resources

- Interpreter invocation

5.7. RUBYGEMS PACKAGES

This section explains what RubyGems packages are, and how to re-package them into RPM.

5.7.1. What RubyGems are

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

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

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

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.

5.7.2. How RubyGems relate to RPM

RubyGems represent Ruby's own packaging format. However, RubyGems contain metadata similar to those needed by RPM, which enables the conversion from RubyGems to RPM.

According to Ruby Packaging Guidelines, it is possible to re-package RubyGems packages into RPM in this way:

- Such RPMs fit with the rest of the distribution.
- End users are able to satisfy dependencies of a gem by installing the appropriate RPM-packaged gem.

RubyGems use similar terminology as RPM, such as SPEC files, package names, dependencies and other items.

To fit into the rest of RHEL RPM distribution, packages created by RubyGems must follow the conventions listed below:

- Names of gems must follow this pattern:

  ```
  rubygem-%{gem_name}
  ```

- To implement a shebang line, the following string must be used:

  ```
  #!/usr/bin/ruby
  ```

5.7.3. Creating RPM packages from RubyGems packages

To create a source RPM for a RubyGems package, the following files are needed:

- A gem file
- An RPM SPEC file

The following sections describe how to create RPM packages from packages created by RubyGems.

5.7.3.1. RubyGems SPEC file conventions

A RubyGems SPEC file must meet the following conventions:

- Contain 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.
- Contain the **BuildRequires:** a directive defined as follows to be able to pull in the macros needed to build.

  ```
  BuildRequires:rubygems-devel
  ```

- Not contain any RubyGems **Requires** or **Provides**, because those are autogenerated.
- Not contain the **BuildRequires:** directive defined as follows, unless you want to explicitly specify Ruby version compatibility:

  ```
  Requires: ruby(release)
  ```

  The automatically generated dependency on RubyGems (**Requires: ruby(rubygems)**) is sufficient.

5.7.3.2. RubyGems macros

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

Table 5.4. 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.

5.7.3.3. RubyGems SPEC file example

Example SPEC file for building gems together with an explanation of its particular sections follows.

An example RubyGems SPEC file

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

# Modify the gemspec if necessary
# Also apply patches to code if necessary
%patch0 -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}/
```

The following table explains the specifics of particular items in a RubyGems SPEC file:

Table 5.5. RubyGems' SPEC directives specifics

SPEC directive	RubyGems specifics
%prep	RPM can directly unpack gem archives, so you can run the **gem unpack** comamnd to extract the source from the gem. The **%setup -n %{gem_name}-%{version}** macro provides the directory into which the gem has been unpacked. At the same directory level, the **%{gem_name}-%{version}.gemspec** file is automatically created, which can be used to rebuild the gem later, to modify the **.gemspec**, or to apply patches to the code.
%build	This directive includes commands or series of commands for building the software into machine code. The **%gem_install** macro operates only on gem archives, and the gem is recreated with the next gem build. The gem file that is created is then used by **%gem_install** to build and install the code into the temporary directory, which is**./%{gem_dir}** by default. The**%gem_install** macro both builds and installs the code in one step. Before being installed, the built sources are placed into a temporary directory that is created automatically. The **%gem_install** macro accepts two additional options:**-n <gem_file>**, which allows to override gem used for installation, and **-d <install_dir>**, which might override the gem installation destination; using this option is not recommended. The **%gem_install** macro must not be used to install into the**%{buildroot}**.
%install	The installation is performed into the **%{buildroot}** hierarchy. You can create the directories that you need and then copy what was installed in the temporary directories into the **%{buildroot}** hierarchy. If this gem creates shared objects, they are moved into the architecture-specific **%{gem_extdir_mri}** path.

Additional resources

- Ruby Packaging Guidelines

5.7.3.4. Converting RubyGems packages to RPM SPEC files with gem2rpm

The **gem2rpm** utility converts RubyGems packages to RPM SPEC files.

The following sections describe how to:

- Install the **gem2rpm** utility

- Display all **gem2rpm** options
- Use **gem2rpm** to covert RubyGems packages to RPM SPEC files
- Edit **gem2rpm** templates

5.7.3.4.1. Installing gem2rpm

The following procedure describes how to install the **gem2rpm** utility.

Procedure

- To install **gem2rpm** from RubyGems.org, run:

```
$ gem install gem2rpm
```

5.7.3.4.2. Displaying all options of gem2rpm

The following procedure describes how to display all options of the **gem2rpm** utility.

Procedure

- To see all options of **gem2rpm**, run:

  ```
  $ gem2rpm --help
  ```

5.7.3.4.3. Using gem2rpm to covert RubyGems packages to RPM SPEC files

The following procedure describes how to use the **gem2rpm** utility to covert RubyGems packages to RPM SPEC files.

Procedure

- Download a gem in its latest version, and generate the RPM SPEC file for this gem:

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

The described procedure creates an RPM SPEC file based on the information provided in the gem's metadata. However, the gem misses some important information that is usually provided in RPMs, such as the license and the changelog. The generated SPEC file thus needs to be edited.

5.7.3.4.4. gem2rpm templates

The **gem2rpm** template is a standard Embedded Ruby (ERB) file, which includes variables listed in the following table.

Table 5.6. Variables in the gem2rpm template

Variable	Explanation
package	The **Gem::Package** variable for the gem.

Variable	Explanation
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 providing a list of package runtime dependencies.
development_dependencies	The **Gem2Rpm::RpmDependencyList** variable providing a list of package development dependencies.
tests	The **Gem2Rpm::TestSuite** variable providing a list of test frameworks allowing their execution.
files	The **Gem2Rpm::RpmFileList** variable providing an unfiltered list of files in a package.
main_files	The **Gem2Rpm::RpmFileList** variable providing a list of files suitable for the main package.
doc_files	The **Gem2Rpm::RpmFileList** variable providing a list of files suitable for the**-doc** subpackage.
format	The **Gem::Format** variable for the gem. Note that this variable is now deprecated.

5.7.3.4.5. Listing available gem2rpm templates

Use the following procedure describes to list all available **gem2rpm** templates.

Procedure

- To see all available templates, run:

```
$ gem2rpm --templates
```

5.7.3.4.6. Editing gem2rpm templates

You can edit the template from which the RPM SPEC file is generated instead of editing the generated SPEC file.

Use the following procedure to edit the **gem2rpm** templates.

Procedure

1. Save the default template:

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

2. Edit the template as needed.

3. 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
```

You can now build an RPM package by using the edited template as described in Building RPMs.

5.8. HOW TO HANDLE RPM PACKAGES WITH PERLS SCRIPTS

Since RHEL 8, the Perl programming language is not included in the default buildroot. Therefore, the RPM packages that include Perl scripts must explicitly indicate the dependency on Perl using the **BuildRequires:** directive in RPM SPEC file.

5.8.1. Common Perl-related dependencies

The most frequently occurring Perl-related build dependencies used in **BuildRequires:** are :

- **perl-generators**
  Automatically generates run-time **Requires** and **Provides** for installed Perl files. When you install a Perl script or a Perl module, you must include a build dependency on this package.

- **perl-interpreter**
  The Perl interpreter must be listed as a build dependency if it is called in any way, either explicitly via the **perl** package or the **%__perl** macro, or as a part of your package's build system.

- **perl-devel**
  Provides Perl header files. If building architecture-specific code which links to the **libperl.so** library, such as an XS Perl module, you must include **BuildRequires: perl-devel**.

5.8.2. Using a specific Perl module

If a specific Perl module is required at build time, use the following procedure:

Procedure

- Apply the following syntax in your RPM SPEC file:

```
BuildRequires: perl(MODULE)
```

NOTE

Apply this syntax to Perl core modules as well, because they can move in and out of the **perl** package over time.

5.8.3. Limiting a package to a specific Perl version

To limit your package to a specific Perl version, follow this procedure:

Procedure

- Use the **perl(:VERSION)** dependency with the desired version constraint in your RPM SPEC file:

For example, to limit a package to Perl version 5.30 and higher, use:

```
BuildRequires: perl(:VERSION) >= 5.30
```

> **WARNING**
>
> Do not use a comparison against the version of the **perl** package because it includes an epoch number.

5.8.4. Ensuring that a package uses the correct Perl interpreter

Red Hat provides multiple Perl interpreters, which are not fully compatible. Therefore, any package that delivers a Perl module must use at run time the same Perl interpreter that was used at build time.

To ensure this, follow the procedure below:

Procedure

- Include versioned **MODULE_COMPAT Requires** in RPM SPEC file for any package that delivers a Perl module:

  ```
  Requires:  perl(:MODULE_COMPAT_%(eval `perl -V:version`; echo $version))
  ```

CHAPTER 6. NEW FEATURES IN RHEL 9

This section documents the most notable changes in RPM packaging between Red Hat Enterprise Linux 8 and 9.

6.1. DYNAMIC BUILD DEPENDENCIES

Red Hat Enterprise Linux 9 introduces the **%generate_buildrequires** section that enables generating dynamic build dependencies.

Additional build dependencies can now be generated programmatically at RPM build time, using the newly available **%generate_buildrequires** script. This is useful when packaging software written in a language in which a specialized utility is commonly used to determine run-time or build-time dependencies, such as Rust, Golang, Node.js, Ruby, Python, or Haskell.

You can use the **%generate_buildrequires** script to dynamically determine which **BuildRequires** directives are added to a SPEC file at build-time. If present, **%generate_buildrequires** is executed after the **%prep** section and can access the unpacked and patched source files. The script must print the found build dependencies to standard output using the same syntax as a regular **BuildRequires** directive.

The **rpmbuild** utility then checks if the dependencies are met before continuing the build.

If some dependencies are missing, a package with the **.buildreqs.nosrc.rpm** suffix is created, which contains the found **BuildRequires** and no source files. You can use this package to install the missing build dependencies with the **dnf builddep** command before restarting the build.

For more information, see the **DYNAMIC BUILD DEPENDENCIES** section in the **rpmbuild(8)** man page.

Additional resources

- **rpmbuild(8)** man page
- **yum-builddep(1)** man page

6.2. IMPROVED PATCH DECLARATION

6.2.1. Optional automatic patch and source numbering

The **Patch:** and **Source:** tags without a number are now automatically numbered based on the order in which they are listed.

The numbering is run internally by the **rpmbuild** utility starting from the last manually numbered entry, or **0** if there is no such entry.

For example:

```
Patch: one.patch
Patch: another.patch
Patch: yet-another.patch
```

6.2.2. %patchlist and %sourcelist sections

It is now possible to list patch and source files without preceding each item with the respective **Patch:** and **Source:** tags by using the newly added **%patchlist** and **%sourcelist** sections.

For example, the following entries:

```
Patch0: one.patch
Patch1: another.patch
Patch2: yet-another.patch
```

can now be replaced with:

```
%patchlist
one.patch
another.patch
yet-another.patch
```

6.2.3. %autopatch now accepts patch ranges

The **%autopatch** macro now accepts the **-m** and **-M** parameters to limit the minimum and maximum patch number to apply, respectively:

- The **-m** parameter specifies the patch number (inclusive) to start at when applying patches.
- The **-M** parameter specifies the patch number (inclusive) to stop at when applying patches.

This feature can be useful when an action needs to be performed in between certain patch sets.

6.3. OTHER FEATURES

Other new features related to RPM packaging in Red Hat Enterprise Linux 9 include:

- Fast macro-based dependency generators
- Powerful macro and **%if** expressions, including ternary operator and native version comparison
- Meta (unordered) dependencies
- Caret version operator (**^**), which can be used to express a version that is higher than the base version. This operator complements the tilde (**~**) operator, which has the opposite semantics.
- **%elif**, **%elifos** and **%elifarch** statements

CHAPTER 7. ADDITIONAL RESOURCES

References to various topics related to RPMs, RPM packaging, and RPM building follows.

- Mock
- RPM Documentation
- RPM 4.15.0 Release Notes
- RPM 4.16.0 Release Notes
- Fedora Packaging Guidelines