RPM Packaging Guide

Adam Miller, Maxim Svistunov, Marie Doleželová, et al.

Table of Contents

Introduction

The RPM Packaging Guide documents:

How to prepare source code for packaging into an RPM.

This is for people with no background in software development. See Preparing Software for Packaging.

How to package source code into an RPM.

This is for software developers who need to package software into RPMs. See Packaging Software.

Advanced packaging scenarios.

This is a reference material for RPM packagers dealing with advanced RPM Packaging scenarios. See Advanced Topics.

PDF Version

You can also download a PDF version of this document.

Document Conventions

The document uses the following conventions:

- Command output and contents of text files, including source code, are placed in blocks:

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

[command output trimmed]
```

```
Name:           bello
Version:
Release:        1%{?dist}
Summary:

[file contents trimmed]
```

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

print("Hello World")
```

- Topics of interest or vocabulary terms are referred to either as URLs to their respective

documentation or website, in **bold**, or in *italics*. The first occurrences of some terms link to their respective documentation.

- Names of utilities, commands, and things normally found in code are written in `monospace` font.

Contributing to this guide

You can contribute to this guide by submitting an issue or a pull request on the GitHub repository.

Both forms of contribution are greatly appreciated and welcome.

Feel free to file an issue ticket with feedback, submit a pull request on GitHub, or both!

Prerequisites

To follow this tutorial, you need the packages mentioned below.

NOTE	Some of these packages are installed by default on Fedora, CentOS and RHEL. They are listed explicitly to show which tools are used in this guide.

On Fedora, CentOS 8, and RHEL 8:

```
$ dnf install gcc rpm-build rpm-devel rpmlint make python bash coreutils diffutils
patch rpmdevtools
```

On CentOS 7 and RHEL 7:

```
$ yum install gcc rpm-build rpm-devel rpmlint make python bash coreutils diffutils
patch rpmdevtools
```

Why Package Software with RPM?

The RPM Package Manager (RPM) is a package management system that runs on Red Hat Enterprise Linux, CentOS, and Fedora. RPM makes it easier for you to distribute, manage, and update software that you create for Red Hat Enterprise Linux, CentOS, and Fedora. Many software vendors distribute their software via a conventional archive file (such as a tarball). However, there are several advantages in packaging software into RPM packages. These advantages are outlined below.

With RPM, you can:

Install, reinstall, remove, upgrade and verify packages

Users can use standard package management tools (for example Yum or PackageKit) to install, reinstall, remove, upgrade and verify your RPM packages.

Use a database of installed packages to query and verify packages

Because RPM maintains a database of installed packages and their files, users can easily query and verify packages on their system.

Use metadata to describe packages, their installation instructions, and so on

Each RPM package includes metadata that describes the package's components, version, release, size, project URL, installation instructions, and so on.

Package pristine software sources into source and binary packages

RPM allows you to take pristine software sources and package them into source and binary packages for your users. In source packages, you have the pristine sources along with any patches that were used, plus complete build instructions. This design eases the maintenance of the packages as new versions of your software are released.

Add packages to Yum repositories

You can add your package to a Yum repository that enables clients to easily find and deploy your software.

Digitally sign your packages

Using a GPG signing key, you can digitally sign your package so that users are able to verify the authenticity of the package.

Your First RPM Package

Creating an RPM package can be complicated. Here is a complete, working RPM Spec file with several things skipped and simplified.

```
Name:       hello-world
Version:    1
Release:    1
Summary:    Most simple RPM package
License:    FIXME

%description
This is my first RPM package, which does nothing.

%prep
# we have no source, so nothing here

%build
cat > hello-world.sh <<EOF
#!/usr/bin/bash
echo Hello world
EOF

%install
mkdir -p %{buildroot}/usr/bin/
install -m 755 hello-world.sh %{buildroot}/usr/bin/hello-world.sh

%files
/usr/bin/hello-world.sh

%changelog
# let's skip this for now
```

Save this file as `hello-world.spec`.

Now use these commands:

```
$ rpmdev-setuptree
$ rpmbuild -ba hello-world.spec
```

The command `rpmdev-setuptree` creates several working directories. As those directories are stored permanently in $HOME, this command does not need to be used again.

The command `rpmbuild` creates the actual rpm package. The output of this command can be similar to:

```
... [SNIP]
Wrote: /home/mirek/rpmbuild/SRPMS/hello-world-1-1.src.rpm
Wrote: /home/mirek/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.wgaJzv
+ umask 022
+ cd /home/mirek/rpmbuild/BUILD
+ /usr/bin/rm -rf /home/mirek/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
+ exit 0
```

The file `/home/mirek/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm` is your first RPM package. It can be installed in the system and tested.

Preparing Software for Packaging

This chapter is about source code and creating software, which are a necessary background for an RPM Packager.

What is Source Code?

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

This tutorial features three versions of the `Hello World` program, each written in a different programming language. Programs written in these three different languages are packaged differently, and cover three major use cases of an RPM packager.

NOTE	There are thousands of programming languages. This document features only three of them, but they are enough for a conceptual overview.

`Hello World` written in bash:

`bello`

```
#!/bin/bash

printf "Hello World\n"
```

`Hello World` written in Python:

`pello.py`

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

print("Hello World")
```

`Hello World` written in C :

`cello.c`

```
#include <stdio.h>

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

The purpose of every one of the three programs is to output `Hello World` on the command line.

NOTE	Knowing how to program is not necessary for a software packager, but is helpful.

How Programs Are Made

There are many methods by which human-readable source code becomes machine code - instructions the computer follows to actually execute the program. However, all methods can be reduced to these three:

1. The program is natively compiled.
2. The program is interpreted by raw interpreting.
3. The program is interpreted by byte compiling.

Natively Compiled Code

Natively compiled software is software written in a programming language that compiles to machine code, with a resulting binary executable file. Such software can be run stand-alone.

RPM packages built this way are architecture -specific. This means that if you compile such software on a computer that uses a 64-bit (x86_64) AMD or Intel processor, it will not execute on a 32-bit (x86) AMD or Intel processor. The resulting package will have architecture specified in its name.

Interpreted Code

Some programming languages, such as bash or Python, do not compile to machine code. Instead, their programs' source code is executed step by step, without prior transformations, by a Language Interpreter or a Language Virtual Machine.

Software written entirely in interpreted programming languages is not architecture -specific. Hence, the resulting RPM Package will have string `noarch` in its name.

Interpreted languages are either **byte-compiled** or **raw-interpreted**. These two types differ in program build process and in packaging procedure.

Raw-interpreted programs

Raw-interpreted language programs do not need to be compiled at all, they are directly executed by the interpreter.

Byte-compiled programs

Byte-compiled languages need to be compiled into byte code, which is then executed by the language virtual machine.

NOTE	Some languages give a choice: they can be raw-interpreted or byte-compiled.

Building Software from Source

This section explains building software from its source code.

- For software written in compiled languages, the source code goes through a **build** process, producing machine code. This process, commonly called **compiling** or **translating**, varies for different languages. The resulting built software can be **run** or "**executed**", which makes computer perform the task specified by the programmer.
- For software written in raw interpreted languages, the source code is not built, but executed directly.
- For software written in byte-compiled interpreted languages, the source code is compiled into byte code, which is then executed by the language virtual machine.

Natively Compiled Code

In this example, you will build the `cello.c` program written in the C language into an executable.

`cello.c`

```
#include <stdio.h>

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

Manual Building

Invoke the C compiler from the GNU Compiler Collection (GCC) to compile the source code into binary:

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

Execute the resulting output binary `cello`.

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

That is all. You have built and ran natively compiled software from source code.

Automated Building

Instead of building the source code manually, you can automate the building. This is a common practice used by large-scale software. Automating building is done by creating a `Makefile` and then running the GNU `make` utility.

To set up automated building, create a file named `Makefile` in the same directory as `cello.c`:

`Makefile`

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

clean:
	rm cello
```

Now to build the software, simply run `make`:

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

Since there is already a build present, `make clean` it and run `make` again:

```
$ make clean
rm cello

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

Again, trying to build after another build would do nothing:

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

Finally, execute the program:

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

You have now compiled a program both manually and using a build tool.

Interpreted Code

The next two examples showcase byte-compiling a program written in Python and raw-interpreting a program written in bash.

NOTE

In the two examples below, the `#!` line at the top of the file is known as a shebang and is not part of the programming language source code.

The shebang enables using a text file as 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.

Byte-Compiled Code

In this example, you will compile the `pello.py` program written in Python into byte code, which is then executed by the Python language virtual machine. Python source code can also be raw-interpreted, but the byte-compiled version is faster. Hence, RPM Packagers prefer to package the byte-compiled version for distribution to end users.

`pello.py`

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

print("Hello World")
```

Procedure for byte-compiling programs is different for different languages. It depends on the language, the language's virtual machine, and the tools and processes used with that language.

NOTE

Python is often byte-compiled, but not in the way described here. The following procedure aims not to conform to the community standards, but to be simple. For real-world Python guidelines, see Software Packaging and Distribution.

Byte-compile `pello.py`:

```
$ python -m compileall pello.py

$ file pello.pyc
pello.pyc: python 2.7 byte-compiled
```

Execute the byte code in `pello.pyc`:

```
$ python pello.pyc
Hello World
```

Raw Interpreted Code

In this example, you will raw-interpret the `bello` program written in the bash shell built-in language.

`bello`

```
#!/bin/bash

printf "Hello World\n"
```

Programs written in shell scripting languages, like *bash*, are raw-interpreted. Hence, you only need to make the file with source code executable and run it:

```
$ chmod +x bello
$ ./bello
Hello World
```

Patching Software

A **patch** is source code that updates other source code. It is formatted as a *diff*, because it represents what is different between two versions of text. A *diff* is created using the `diff` utility, which is then applied to the source code using the patch utility.

NOTE	Software developers often use Version Control Systems such as git to manage their code base. Such tools provide their own methods of creating diffs or patching software.

In the following example, we create a patch from the original source code using `diff` and then apply it using `patch`. Patching is used in a later section when creating an RPM, Working with SPEC files.

How is patching related to RPM packaging? In packaging, instead of simply modifying the original source code, we keep it, and use patches on it.

To create a patch for `cello.c`:

1. Preserve the original source code:

   ```
   $ cp cello.c cello.c.orig
   ```

   This is a common way to preserve the original source code file.

2. Change `cello.c`:

   ```
   #include <stdio.h>

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

3. Generate a patch using the `diff` utility:

NOTE	We use several common arguments for the `diff` utility. For more information on them, see the `diff` manual page.

```
$ 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;
 }
```

   Lines starting with a `-` are removed from the original source code and replaced with the lines that start with `+`.

4. Save the patch to a file:

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

5. Restore the original `cello.c`:

```
$ cp cello.c.orig cello.c
```

   We retain the original `cello.c`, because when an RPM is built, the original file is used, not a modified one. For more information, see Working with SPEC files.

To patch `cello.c` using `cello-output-first-patch.patch`, redirect the patch file to the `patch` command:

```
$ patch < cello-output-first-patch.patch
patching file cello.c
```

The contents of `cello.c` now reflect the patch:

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

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

To build and run the patched `cello.c`:

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

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

You have created a patch, patched a program, built the patched program, and run it.

Installing Arbitrary Artifacts

A big advantage of Linux and other Unix-like systems is the Filesystem Hierarchy Standard (FHS). It specifies in which directory which files should be located. Files installed from the RPM packages should be placed according to FHS. For example, an executable file should go into a directory that is in the system PATH variable.

In the context of this guide, an *Arbitrary Artifact* is anything installed from an RPM to the system. For RPM and for the system it can be a script, a binary compiled from the package's source code, a pre-compiled binary, or any other file.

We will explore two popular ways of placing *Arbitrary Artifacts* in the system: using the `install` command and using the `make install` command.

Using the install command

Sometimes using build automation tooling such as GNU make is not optimal - for example, if the packaged program is simple and does not need extra overhead. In these cases, packagers often use the `install` command (provided to the system by coreutils), which places the artifact to the specified directory in the filesystem with a specified set of permissions.

The example below is going to use the `bello` file that we had previously created as the arbitrary artifact subject to our installation method. Note that you will either need sudo permissions or run this command as root excluding the `sudo` portion of the command.

In this example, `install` places the `bello` file into `/usr/bin` with permissions common for executable

scripts:

```
$ sudo install -m 0755 bello /usr/bin/bello
```

Now `bello` is in a directory that is listed in the `$PATH` variable. Therefore, you can execute `bello` from any directory without specifying its full path:

```
$ cd ~

$ bello
Hello World
```

Using the make install command

A popular automated way to install built software to the system is to use the `make install` command. It requires you to specify how to install the arbitrary artifacts to the system in the `Makefile`.

> **NOTE** Usually `Makefile` is written by the developer and not by the packager.

Add the `install` section to the `Makefile`:

`Makefile`

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

clean:
	rm cello

install:
	mkdir -p $(DESTDIR)/usr/bin
	install -m 0755 cello $(DESTDIR)/usr/bin/cello
```

The `$(DESTDIR)` variable is a GNU make built-in and is commonly used to specify installation to a directory different than the root directory.

Now you can use `Makefile` not only to build software, but also to install it to the target system.

To build and install the `cello.c` program:

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

$ sudo make install
install -m 0755 cello /usr/bin/cello
```

Now `cello` is in a directory that is listed in the `$PATH` variable. Therefore, you can execute `cello` from any directory without specifying its full path:

```
$ cd ~

$ cello
Hello World
```

You have installed a build artifact into a chosen location on the system.

Preparing Source Code for Packaging

NOTE	The code created in this section can be found here.

Developers often distribute software as compressed archives of source code, which are then used to create packages. In this section, you will create such compressed archives.

NOTE	Creating source code archives is not normally done by the RPM Packager, but by the developer. The packager works with a ready source code archive.

Software should be distributed with a software license . For the examples, we will use the GPLv3 license. The license text goes into the `LICENSE` file for each of the example programs. An RPM packager needs to deal with license files when packaging.

For use with the following examples, create a `LICENSE` file:

```
$ 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/>.
```

Putting Source Code into Tarball

In the examples below, we put each of the three `Hello World` programs into a gzip-compressed tarball. Software is often released this way to be later packaged for distribution.

bello

The *bello* project implements `Hello World` in bash. The implementation only contains the `bello` shell script, so the resulting `tar.gz` archive will have only one file apart from the `LICENSE` file. Let us assume that this is version `0.1` of the program.

Prepare the *bello* project for distribution:

1. Put the files into a single directory:

```
$ mkdir /tmp/bello-0.1

$ mv ~/bello /tmp/bello-0.1/

$ cp /tmp/LICENSE /tmp/bello-0.1/
```

2. Create the archive for distribution and move it to `~/rpmbuild/SOURCES/`:

```
$ cd /tmp/

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

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

pello

The *pello* project implements `Hello World` in Python. The implementation only contains the `pello.py` program, so the resulting `tar.gz` archive will have only one file apart from the `LICENSE` file. Let us assume that this is version `0.1.1` of the program.

Prepare the *pello* project for distribution:

1. Put the files into a single directory:

```
$ mkdir /tmp/pello-0.1.1

$ mv ~/pello.py /tmp/pello-0.1.1/

$ cp /tmp/LICENSE /tmp/pello-0.1.1/
```

2. Create the archive for distribution and move it to `~/rpmbuild/SOURCES/`:

```
$ cd /tmp/

$ 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

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

cello

The *cello* project implements `Hello World` in `C` . The implementation only contains the `cello.c` and `Makefile` files, so the resulting `tar.gz` archive will have only two files apart from the `LICENSE` file. Let us assume that this is version `1.0` of the program.

Note that the `patch` file is not distributed in the archive with the program. The RPM Packager applies the patch when the RPM is built. The patch will be placed in the `~/rpmbuild/SOURCES/` directory alongside the `.tar.gz`.

Prepare the *cello* project for distribution:

1. Put the files into a single directory:

```
$ mkdir /tmp/cello-1.0

$ mv ~/cello.c /tmp/cello-1.0/

$ mv ~/Makefile /tmp/cello-1.0/

$ cp /tmp/LICENSE /tmp/cello-1.0/
```

2. Create the archive for distribution and move it to `~/rpmbuild/SOURCES/`:

```
$ cd /tmp/

$ 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

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

3. Add the patch:

```
$ mv ~/cello-output-first-patch.patch ~/rpmbuild/SOURCES/
```

Now the source code is ready for packaging into an RPM.

Packaging Software

This tutorial explains packaging RPMs for the Red Hat family of Linux distributions, primarily:

- Fedora
- CentOS
- Red Hat Enterprise Linux (RHEL)

These distributions use the RPM Packaging Format.

While these distributions are the target environment, this guide is mostly applicable to all RPM based distributions. However, the instructions need to be adapted for distribution-specific features, such as prerequisite installation items, guidelines, or macros.

This tutorial assumes no previous knowledge about packaging software for any Operating System, Linux or otherwise.

NOTE	If you do not know what a software package or a GNU/Linux distribution is, consider exploring some articles on the topics of Linux and Package Managers.

RPM Packages

This section covers the basics of the RPM packaging format. See Advanced Topics for more advanced information.

What is an RPM?

An RPM package is simply a file containing other files and information about them needed by the system. Specifically, an RPM package consists of the cpio archive, which contains the files, and the RPM header, which contains metadata about the package. The `rpm` package manager uses this metadata to determine dependencies, where to install files, and other information.

There are two types of RPM packages:

- source RPM (SRPM)
- binary RPM

SRPMs and binary RPMs share the file format and tooling, but have different contents and serve different purposes. An SRPM contains source code, optionally patches to it, and a SPEC file, which describes how to build the source code into a binary RPM. A binary RPM contains the binaries built from the sources and patches.

RPM Packaging Tools

The `rpmdevtools` package, installed in Prerequisites, provides several utilities for packaging RPMs. To list these utilities, run:

```
$ rpm -ql rpmdevtools | grep bin
```

For more information on the above utilities, see their manual pages or help dialogs.

RPM Packaging Workspace

To set up a directory layout that is the RPM packaging workspace, use the `rpmdev-setuptree` utility:

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

The created directories serve these 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.

What is a SPEC File?

A SPEC file can be thought of as the "recipe" that the `rpmbuild` utility uses to actually build an RPM. It tells the build system what to do by defining instructions in a series of sections. The sections are defined in the *Preamble* and the *Body*. The *Preamble* contains a series of metadata items that are used in the *Body*. The *Body* contains the main part of the instructions.

Preamble Items

This table lists the 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. For packages distributed in community distributions such as Fedora this must be an open source license abiding by the specific distribution's licensing guidelines.
`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.
`Patch0`	The name of the first patch to apply to the source code if necessary. If needed, more PatchX directives can be added, incrementing the number each time, for example: Patch1, Patch2, Patch3, and so on.
`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.

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

You can get an example of an `NAME-VERSION-RELEASE` by querying using `rpm` for a specific package:

```
$ rpm -q python
python-2.7.5-34.el7.x86_64
```

Here, `python` is the Package Name, `2.7.5` is the Version, and `34.el7` 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.

Body Items

This table lists the 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 actually building the software into machine code (for compiled languages) or byte code (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 installed in the end user's system.
`%changelog`	A record of changes that have happened to the package between different `Version` or `Release` builds.

Advanced items

The SPEC file can also contain advanced items. For example, a SPEC file can have *scriptlets* and *triggers*. They take effect at different points during the installation process on the end user's system (not the build process).

See the Scriptlets and Triggers for advanced topics.

BuildRoots

In the context of RPM packaging, "buildroot" is a chroot environment. This means that the build

artifacts are placed here using the same filesystem hierarchy as will be in the end user's system, with "buildroot" acting as the root directory. The placement of build artifacts should comply with the filesystem 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 release, the `rpmbuild` program has its own defaults. As overriding these defaults leads to several problems, Red Hat does not recommend to define your own value of this macro. You can use the `%{buildroot}` macro with the defaults from the `rpmbuild` directory.

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. What this means is that you can have RPM perform text substitutions for you so that you don't have to.

This is useful when, for example, referencing the packaged software *Version* multiple times in the SPEC file. You define *Version* only once - in the `%{version}` macro. Then use `%{version}` throughout the SPEC file. Every occurrence will be automatically substituted by *Version* you defined previously.

NOTE

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

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

For example:

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

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

A common macro is `%{?dist}`, which signifies the "distribution tag". It signals which distribution is used for the build.

For example:

```
# On a RHEL 7.x machine
$ rpm --eval %{?dist}
.el7

# On a Fedora 23 machine
$ rpm --eval %{?dist}
.fc23
```

For more information on macros, see More on Macros.

Working with SPEC files

A big part of packaging software into RPMs is editing the SPEC file. In this section we discuss how to create and modify a spec file.

To package new software, you need to create a new SPEC file. Instead of writing it manually from scratch, use the `rpmdev-newspec` utility. It creates an unpopulated SPEC file, and you fill in the necessary directives and fields.

For this tutorial, we use the three example implementations of the 'Hello World!' program created in Preparing Software for Packaging:

- bello-0.1.tar.gz
- pello-0.1.1.tar.gz
- cello-1.0.tar.gz
  - cello-output-first-patch.patch

Place them in `~/rpmbuild/SOURCES`.

Create a SPEC file for each of the three programs:

NOTE Some programmer-focused text editors pre-populate a new `.spec` file with their own SPEC template. The `rpmdev-newspec` provides an editor-agnostic method, which is why it is used in this guide.

```
$ cd ~/rpmbuild/SPECS

$ 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`.

Examine the files. The directives in them represent the ones described in the What is a SPEC File? section. In the following sections, you will populate these SPEC files.

NOTE

The `rpmdev-newspec` utility does not use guidelines or conventions specific to any particular Linux distribution. However, this document targets Fedora, CentOS, and RHEL, so you will notice that:

- Use `rm $RPM_BUILD_ROOT`when building on *CentOS* (versions previous to 7.0) or on Fedora (versions previous to 18).
- We favor the use of `%{buildroot}` notation over `$RPM_BUILD_ROOT` when referencing RPM's Buildroot for consistency with all other defined or provided macros throughout the SPEC file.

There are three examples below. Each one is fully described, so you can go to a specific one if it matches your packaging needs. Or, read them all to fully explore packaging different kinds of software.

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 since 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 installating 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.

bello

The first SPEC file is for the `bello` bash shell script from Preparing Software for Packaging.

Ensure that you have:

1. Placed `bello` source code into `~/rpmbuild/SOURCES/`. See Working with SPEC files.
2. Created the unpopulated SPEC file `~/rpmbuild/SPECS/bello.spec`. The file has these contents:

```
Name:           bello
Version:
Release:        1%{?dist}
Summary:

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Now, modify `~/rpmbuild/SPECS/bello.spec` for creating `bello` RPMs:

1. Populate the `Name`, `Version`, `Release`, and `Summary` directives:
    - The `Name` was already specified as an argument to `rpmdev-newspec`.
    - Set the `Version` to match the "upstream" release version of the `bello` source code, `0.1`.
    - The `Release` is automatically set to `1%{?dist}`, which is initially `1`. Increment the initial value whenever updating the package without a change in the upstream release `Version` - such as when including a patch. Reset `Release` to `1` when a new upstream release happens, for example, if bello version `0.2` is released. The *disttag* macro is covered in RPM Macros.
    - The `Summary` is a short, one-line explanation of what this software is.

    After your edits, the first section of the SPEC file should resemble:

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

2. Populate the `License`, `URL`, and `Source0` directives:
   - The `License` field is the Software License associated with the source code from the upstream release.

     Follow this format for the `License` field: Fedora License Guidelines

     For example, use `GPLv3+`.
   - The `URL` field provides URL to the upstream software page. For example, use `https://example.com/bello`. However, for consistency, utilize the %{name} macro and instead use `https://example.com/%{name}`.
   - The `Source0` field provides URL to the upstream software source code. It should link directly to the version of software that is being packaged. In this example, we can use `https://example.com/bello/releases/bello-0.1.tar.gz`. Instead, use the %{name} macro. Also, use the %{version} macro to accommodate for changes in version. The resulting entry is `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz`.

     After your edits, the first section of the SPEC file should resemble:

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

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

3. Populate the `BuildRequires` and `Requires` directives and include the `BuildArch` directive:
   - `BuildRequires` specifies build-time dependencies for the package. There is no building step for `bello`, because bash is a raw interpreted programming language, and the files are simply installed to their location on the system. Just delete this directive.
   - `Requires` specifies run-time dependencies for the package. The `bello` script requires only the `bash` shell environment to execute, so specify `bash` in this directive.
   - Since this is software written in an interpreted programming language with no natively compiled extensions, 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.

     After your edits, the first section of the SPEC file should resemble:

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

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

Requires:       bash

BuildArch:      noarch
```

4. Populate the `%description`, `%prep`, `%build`, `%install`, `%files`, and `%license` directives. These directives can be thought of as "section headings", because they are directives that can define multi-line, multi-instruction, or scripted tasks to occur.
   - The `%description` is a longer, fuller description of the software than `Summary`, containing one or more paragraphs. In our example we will use only a short description.
   - 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. In this section we simply use the built-in macro `%setup -q`.
   - The `%build` section specifies how to actually build the software we are packaging. However, since a `bash` does not need to be built, simply remove what was provided by the template and leave this section blank.
   - 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 we should create any directories that will contain the installed files.

     Since for installing `bello` we only need to create the destination directory and install the executable `bash` script file there, we will use the `install` command. RPM macros allow us to do this without hardcoding paths.

     The `%install` section should look like the following after your edits:

     ```
     %install

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

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

   - The `%files` section specifies the list of files provided by this RPM and their full path location on the end user's system. Therefore, the listing for the `bello` file we are installing is `/usr/bin/bello`, or, with RPM Macros, `%{_bindir}/%{name}`.

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 using the `rpm` command. For example, to indicate that the LICENSE file is a software license file, we use the %license macro.

After your edits, the `%files` section looks like this:

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

5. 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 `%build`.

   Follow this format for the first line:

   ```
   * Day-of-Week Month Day Year Name Surname <email> - Version-Release
   ```

   Follow this format for the actual change entry:

   - Each change entry can contain multiple items - one for each change
   - Each item starts on a new line.
   - Each item begins with a `-` character.

   An example datestamped entry:

   ```
   %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
   ```

You have now written an entire SPEC file for **bello**. The full SPEC file for **bello** now resembles:

```
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 next section covers how to build the RPM.

pello

Our second SPEC file will be for our example written in the Python programming language that you downloaded (or you created a simulated upstream release in the Preparing Software for Packaging section) and placed its source code into `~/rpmbuild/SOURCES/` earlier. Let's go ahead and open the file `~/rpmbuild/SPECS/pello.spec` and start filling in some fields.

Before we start down this path, we need to address something somewhat unique about byte-compiled interpreted software. Since we will be byte-compiling this program, the shebang is no longer applicable because the resulting file will not contain the entry. It is common practice to

either have a non-byte-compiled shell script that will call the executable or have a small bit of the Python code that isn't byte-compiled as the "entry point" into the program's execution. This might seem silly for our small example but for large software projects with many thousands of lines of code, the performance increase of pre-byte-compiled code is sizeable.

NOTE

The creation of a script to call the byte-compiled code or having a non-byte-compiled entry point into the software is something that upstream software developers most often address before doing a release of their software to the world, however this is not always the case and this exercise is meant to help address what to do in those situations. For more information on how Python code is normally released and distributed please reference the Software Packaging and Distribution documentation.

We will make a small shell script to call our byte compiled code to be the entry point into our software. We will do this as a part of our SPEC file itself in order to demonstrate how you can script actions inside the SPEC file. We will cover the specifics of this in the `%install` section later.

Let's go ahead and open the file `~/rpmbuild/SPECS/pello.spec` and start filling in some fields.

The following is the output template we were given from `rpmdev-newspec`.

```
Name:           pello
Version:
Release:        1%{?dist}
Summary:

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Just as with the first example, let's begin with the first set of directives that `rpmdev-newspec` has grouped together at the top of the file: `Name`, `Version`, `Release`, `Summary`. The `Name` is already specified because we provided that information to the command line for `rpmdev-newspec`.

Let's set the `Version` to match what the "upstream" release version of the *pello* source code is, which we can observe is `0.1.1` as set by the example code we downloaded (or we created in the Preparing Software for Packaging section).

The `Release` is already set to `1%{?dist}` for us, the numerical value which is initially `1` should be incremented every time the package is updated for any reason, such as including a new patch to fix an issue, but doesn't have a new upstream release `Version`. When a new upstream release happens (for example, pello version `0.1.2` were released) then the `Release` number should be reset to `1`. The *disttag* of `%{?dist}` should look familiar from the previous section's coverage of RPM Macros.

The `Summary` should be a short, one-line explanation of what this software is.

After your edits, the first section of the SPEC file should resemble the following:

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

Now, let's move on to the second set of directives that `rpmdev-newspec` has grouped together in our SPEC file: `License`, `URL`, `Source0`.

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 will vary depending on which specific RPM based Linux distribution guidelines you are following, we will use the notation standards in the Fedora License Guidelines for this document and as such this field will contain the text `GPLv3+`

The `URL` field is the upstream software's website, not the source code download link but the actual project, product, or company website where someone would find more information about this particular piece of software. Since we're just using an example, we will call this `https://example.com/pello`. However, we will use the RPM macro variable of `%{name}` in its place for consistency.

The `Source0` field is where the upstream software's source code should be able to be downloaded from. This URL should link directly to the specific version of the source code release that this RPM Package is packaging. Once again, since this is an example we will use an example value: `https://example.com/pello/releases/pello-0.1.1.tar.gz`

We should note that this example URL has hard coded values in it that are possible to change in the future and are potentially even likely to change such as the release version `0.1.1`. We can simplify this by only needing to update one field in the SPEC file and allowing it to be reused. we will use the value `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz` instead of the hard coded examples string previously listed.

After your edits, the top portion of your spec file should look like the following:

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

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

Next up we have `BuildRequires` and `Requires`, each of which define something that is required by the package. However, `BuildRequires` is to tell `rpmbuild` what is needed by your package at **build** time and `Requires` is what is needed by your package at **run** time.

In this example we will need the `python` package in order to perform the byte-compile build process. We will also need the `python` package in order to execute the byte-compiled code at runtime and

therefore need to define `python` as a requirement using the `Requires` directive. We will also need the `bash` package in order to execute the small entry-point script we will use here.

Something we need to add here since this is software written in an interpreted programming language with no natively compiled extensions is a `BuildArch` entry that is set to `noarch` in order to tell RPM that this package does not need to be bound to the processor architecture that it is built using.

After your edits, the top portion of your spec file should look like the following:

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

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

BuildRequires:  python
Requires:       python
Requires:       bash

BuildArch:      noarch
```

The following directives can be thought of as "section headings" because they are directives that can define multi-line, multi-instruction, or scripted tasks to occur. We will walk through them one by one just as we did with the previous items.

The `%description` should be a longer, more full length description of the software being packaged than what is found in the `Summary` directive. For the sake of our example, this isn't really going to contain much content but this section can be a full paragraph or more than one paragraph if desired.

The `%prep` section is where we *prepare* our build environment or workspace for building. Most often what happens here is the expansion of compressed archives of the source code, application of patches, and potentially parsing of information provided in the source code that is necessary in a later portion of the SPEC. In this section we will simply use the provided macro `%setup -q`.

The `%build` section is where we tell the system how to actually build the software we are packaging. Here we will perform a byte-compilation of our software. For those who read the Preparing Software for Packaging section, this portion of the example should look familiar.

The `%build` section of our SPEC file should look as follows.

```
%build

python -m compileall pello.py
```

The %install section is where we instruct rpmbuild how to install our previously built software into the BUILDROOT which is effectively a chroot base directory with nothing in it and we will have to construct any paths or directory hierarchies that we will need in order to install our software here in their specific locations. However, our RPM Macros help us accomplish this task without having to hardcode paths.

We had previously discussed that since we will lose the context of a file with the shebang line in it when we byte compile that we will need to create a simple wrapper script in order to accomplish that task. There are many options on how to accomplish this including, but not limited to, making a separate script and using that as a separate SourceX directive and the option we're going to show in this example which is to create the file in-line in the SPEC file. The reason for showing the example option that we are is simply to demonstrate that the SPEC file itself is scriptable. What we're going to do is create a small"wrapper script" which will execute the Python byte-compiled code by using a here document . We will also need to actually install the byte-compiled file into a library directory on the system such that it can be accessed.

NOTE

You will notice below that we are hard coding the library path. There are various methods to avoid needing to do this, many of which are addressed in Advanced Topics, under the More on Macros section, and are specific to the programming language in which the software that is being packaged was written in. In this example we hard code the path for simplicity as to not cover too many topics simultaneously.

The %install section should look like the following after your edits:

```
%install

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

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

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

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

The %files section is where we provide the list of files that this RPM provides and where it's intended for them to live on the system that the RPM is installed upon. Note here that this isn't relative to the %{buildroot} but the full path for the files as they are expected to exist on the end system after installation. Therefore, the listing for the pello file we are installing will be %{_bindir}/pello. We will also need to provide a %dir listing to define that this package "owns" the library directory we created as well as all the files we placed in it.

Also, within this section you will sometimes need a built-in macro to provide context on a file. This can be useful for Systems Administrators and end users who might want to query the system with

`rpm` about the resulting package. The built-in macro we will use here is `%license` which will tell `rpmbuild` that this is a software license file in the package file manifest metadata.

The `%files` section should look like the following after your edits:

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

The last section, `%changelog` is a list of date-stamped entries that correlate to a specific Version-Release of the package. This is not meant to be a log of what changed in the software from release to release, but specifically to packaging changes. For example, if software in a package needed patching or there was a change needed in the build procedure listed in the `%build` section that information would go here. Each change entry can contain multiple items and each item should start on a new line and begin with a `-` character. Below is our example entry:

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
- First pello package
- Example second item in the changelog for version-release 0.1.1-1
```

Note the format above, the date-stamp will begin with a `*` character, followed by the calendar day of the week, the month, the day of the month, the year, then the contact information for the RPM Packager. From there we have a `-` character before the Version-Release, which is an often used convention but not a requirement. Then finally the Version-Release.

That's it! We've written an entire SPEC file for **pello**! In the next section we will cover how to build the RPM!

The full SPEC file should now look like the following:

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

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

BuildRequires:  python
Requires:       python
Requires:       bash

BuildArch:      noarch

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

%prep
%setup -q

%build

python -m compileall %{name}.py

%install

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

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

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

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

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

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

cello

Our third SPEC file will be for our example written in the C programming language that we created a simulated upstream release of previously (or you downloaded) and placed its source code into `~/rpmbuild/SOURCES/` earlier.

Let's go ahead and open the file `~/rpmbuild/SPECS/cello.spec` and start filling in some fields.

The following is the output template we were given from `rpmdev-newspec`.

```
Name:           cello
Version:
Release:        1%{?dist}
Summary:

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Just as with the previous examples, let's begin with the first set of directives that `rpmdev-newspec` has grouped together at the top of the file: `Name`, `Version`, `Release`, `Summary`. The `Name` is already specified because we provided that information to the command line for `rpmdev-newspec`.

Let's set the `Version` to match what the "upstream" release version of the *cello* source code is, which we can observe is `1.0` as set by the example code we downloaded (or we created in the Preparing Software for Packaging section).

The `Release` is already set to `1%{?dist}` for us, the numerical value which is initially `1` should be incremented every time the package is updated for any reason, such as including a new patch to fix an issue, but doesn't have a new upstream release `Version`. When a new upstream release happens (for example, cello version `2.0` were released) then the `Release` number should be reset to `1`. The *disttag* of `%{?dist}` should look familiar from the previous section's coverage of RPM Macros.

The `Summary` should be a short, one-line explanation of what this software is.

After your edits, the first section of the SPEC file should resemble the following:

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

Now, let's move on to the second set of directives that `rpmdev-newspec` has grouped together in our SPEC file: `License`, `URL`, `Source0`. However, we will add one to this grouping as it is closely related to the `Source0` and that is our `Patch0` which will list the first patch we need against our software.

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 will vary depending on which specific RPM based Linux distribution guidelines you are following, we will use the notation standards in the Fedora License Guidelines for this document and as such this field will contain the text `GPLv3+`

The `URL` field is the upstream software's website, not the source code download link but the actual project, product, or company website where someone would find more information about this particular piece of software. Since we're just using an example, we will call this `https://example.com/cello`. However, we will use the rpm macro variable of `%{name}` in its place for consistency.

The `Source0` field is where the upstream software's source code should be able to be downloaded from. This URL should link directly to the specific version of the source code release that this RPM Package is packaging. Once again, since this is an example we will use an example value: `https://example.com/cello/releases/cello-1.0.tar.gz`

We should note that this example URL has hard coded values in it that are possible to change in the future and are potentially even likely to change such as the release version `1.0`. We can simplify this by only needing to update one field in the SPEC file and allowing it to be reused. we will use the value `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz` instead of the hard coded examples string previously listed.

The next item is to provide a listing for the `.patch` file we created earlier such that we can apply it to the code later in the `%prep` section. We will need a listing of `Patch0: cello-output-first-patch.patch`.

After your edits, the top portion of your spec file should look like the following:

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

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

Patch0:         cello-output-first-patch.patch
```

Next up we have `BuildRequires` and `Requires`, each of which define something that is required by the package. However, `BuildRequires` is to tell `rpmbuild` what is needed by your package at **build** time and `Requires` is what is needed by your package at **run** time.

In this example we will need the `gcc` and `make` packages in order to perform the compilation build process. Runtime requirements are fortunately handled for us by `rpmbuild` because this program does not require anything outside of the core C standard libraries and we therefore will not need to define anything by hand as a `Requires` and can omit that directive.

After your edits, the top portion of your spec file should look like the following:

```
Name:           cello
Version:        0.1
Release:        1%{?dist}
Summary:        Hello World example implemented in C

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

BuildRequires:  gcc
BuildRequires:  make
```

The following directives can be thought of as "section headings" because they are directives that can define multi-line, multi-instruction, or scripted tasks to occur. We will walk through them one by one just as we did with the previous items.

The `%description` should be a longer, more full length description of the software being packaged than what is found in the `Summary` directive. For the sake of our example, this isn't really going to contain much content but this section can be a full paragraph or more than one paragraph if desired.

The `%prep` section is where we *prepare* our build environment or workspace for building. Most often what happens here is the expansion of compressed archives of the source code, application of patches, and potentially parsing of information provided in the source code that is necessary in a later portion of the SPEC. In this section we will simply use the provided macro `%setup -q`.

The %build section is where we tell the system how to actually build the software we are packaging. Since wrote a simple Makefile for our C implementation, we can simply use the GNU make command provided by rpmdev-newspec. However, we need to remove the call to %configure because we did not provide a configure script . The %build section of our SPEC file should look as follows.

```
%build
make %{?_smp_mflags}
```

The %install section is where we instruct rpmbuild how to install our previously built software into the BUILDROOT which is effectively a chroot base directory with nothing in it and we will have to construct any paths or directory hierarchies that we will need in order to install our software here in their specific locations. However, our RPM Macros help us accomplish this task without having to hardcode paths.

Once again, since we have a simple Makefile the installation step can be accomplished easily by leaving in place the %make_install macro that was again provided for us by the rpmdev-newspec command.

The %install section should look like the following after your edits:

```
%install
%make_install
```

The %files section is where we provide the list of files that this RPM provides and where it's intended for them to live on the system that the RPM is installed upon. Note here that this isn't relative to the %{buildroot} but the full path for the files as they are expected to exist on the end system after installation. Therefore, the listing for the cello file we are installing will be %{_bindir}/cello.

Also, within this section you will sometimes need a built-in macro to provide context on a file. This can be useful for Systems Administrators and end users who might want to query the system with rpm about the resulting package. The built-in macro we will use here is %license which will tell rpmbuild that this is a software license file in the package file manifest metadata.

The %files section should look like the following after your edits:

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

The last section, %changelog is a list of date-stamped entries that correlate to a specific Version-Release of the package. This is not meant to be a log of what changed in the software from release to release, but specifically to packaging changes. For example, if software in a package needed patching or there was a change needed in the build procedure listed in the %build section that information would go here. Each change entry can contain multiple items and each item should start on a new line and begin with a - character. Below is our example entry:

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First cello package
```

Note the format above, the date-stamp will begin with a * character, followed by the calendar day of the week, the month, the day of the month, the year, then the contact information for the RPM Packager. From there we have a - character before the Version-Release, which is an often used convention but not a requirement. Then finally the Version-Release.

That's it! We've written an entire SPEC file for **cello**! In the next section we will cover how to build the RPM!

The full SPEC file should now look like the following:

```
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 `rpmdevtools` package provides a set of SPEC file templates for several popular languages in the `/etc/rpmdevtools/` directory.

Building RPMS

RPMs are built with the `rpmbuild` command. Different scenarios and desired outcomes require different combinations of arguments to `rpmbuild`. This section describes the two prime scenarios:

1. building a source RPM
2. building a binary RPM

The `rpmbuild` command expects a certain directory and file structure. This is the same structure as

set up by the `rpmdev-setuptree` utility. The previous instructions also confirmed to the required structure.

Source RPMs

Why build a Source RPM (SRPM)?

1. To preserve the exact source of a certain Name-Version-Release of the RPM that was deployed to an environment. This includes the exact SPEC file, the source code, and all relevant patches. This is useful for looking back in history and for debugging.
2. To be able to build a binary RPM on a different hardware platform or architecture.

To create a SRPM:

```
$ rpmbuild -bs _SPECFILE_
```

Substitute *SPECFILE* with the SPEC file. The `-bs` option stands for "build source".

Here we build SRPMs for `hello`, `pello`, and `cello`:

```
$ cd ~/rpmbuild/SPECS/

$ rpmbuild -bs hello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/hello-0.1-1.el7.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
```

Note that SRPMs were placed into the `rpmbuild/SRPMS` directory, which is part of the structure expected by `rpmbuild`.

This is all there is to building a SRPM.

Binary RPMS

There are two methods for building Binary RPMs:

1. Rebuilding it from a SRPM using the `rpmbuild --rebuild` command.
2. Building it from a SPEC file using the `rpmbuild -bb` command. The `-bb` option stands for "build binary".

Rebuilding from a Source RPM

To rebuild `hello`, `pello`, and `cello` from Source RPMs (SRPMs), run:

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
[output truncated]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
[output truncated]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
[output truncated]
```

Now you have built RPMs. A few notes:

- 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 in `~/rpmbuild/RPMS/YOURARCH` where `YOURARCH` is your architecture or in `~/rpmbuild/RPMS/noarch/`, if the package is not architecture-specific.
- Invoking `rpmbuild --rebuild` involves:

  1. Installing the contents of the SRPM - the SPEC file and the source code - into the `~/rpmbuild/` directory.
  2. Building using the installed contents.
  3. Removing the SPEC file and the source code.

  You can retain the SPEC file and the source code after building. For this, you have two options:

  - When building, use the `--recompile` option instead of `--rebuild`.
  - Install the SRPMs using these commands:

  ```
  $ rpm -Uvh ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
  Updating / installing...
     1:bello-0.1-1.el7                  ################################# [100%]

  $ rpm -Uvh ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
  Updating / installing...
     1:pello-0.1.1-1.el7                ################################# [100%]

  $ rpm -Uvh ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
  Updating / installing...
     1:cello-1.0-1.el7                  ################################# [100%]
  ```

  For this tutorial, execute the `rpm -Uvh` commands above to continue interacting with the SPEC files and sources.

Building Binary from the SPEC file

To build `bello`, `pello`, and `cello` from their SPEC files, run:

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

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

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

Now you have built RPMs from SPEC files.

Most of the information in Rebuilding from a Source RPM apply here.

Checking RPMs For Sanity

After creating a package, it is good to check its quality. Quality of the package, not of the software delivered within it. The main tool for this is rpmlint. It improves RPM maintainability and enables sanity and error checking by performing static analysis of the RPM. This utility can check Binary RPMs, Source RPMs (SRPMs), and SPEC files, so is useful for all stages of packaging, as shown in the following examples.

Note that `rpmlint` has very strict guidelines, and sometimes it is acceptable and necessary to skip some of its Errors and Warnings, as shown in the following examples.

NOTE	In the examples, we run `rpmlint` without any options, which produces non-verbose output. For detailed explanations of each Error or Warning, run `rpmlint -i` instead.

Checking the bello SPEC File

This is the output of running `rpmlint` on the SPEC file for `bello`:

```
$ 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.
```

Observations:

- For `bello.spec` there is only one warning. It says that the URL listed in the `Source0` directive is unreachable. This is expected, because the specified `example.com` URL does not exist. Presuming that we expect this URL to work in the future, we can ignore this warning.

This is the output of running `rpmlint` on the SRPM for `bello`:

```
$ rpmlint ~/rpmbuild/SRPMS/bello-0.1-1.el7.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.
```

Observations:

- For the `bello` SRPM there is a new warning, which says that the URL specified in the `URL` directive is unreachable. Assuming the link will be working in the future, we can ignore this warning.

Checking the bello Binary RPM

When checking Binary RPMs, `rpmlint` checks for more things, including:

1. documentation
2. manual pages
3. consistent use of the Filesystem Hierarchy Standard

This is the output of running `rpmlint` on the Binary RPM for `bello`:

```
$ rpmlint ~/rpmbuild/RPMS/noarch/bello-0.1-1.el7.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.
```

Observations:

- The `no-documentation` and `no-manual-page-for-binary` warnings say that the RPM has no documentation or manual pages, because we did not provide any.

Apart from the above warnings, our RPM is passing `rpmlint` checks.

Checking the pello SPEC File

This is the output of running `rpmlint` on the SPEC file for `pello`:

```
$ 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-
0.1.1.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 5 errors, 1 warnings.
```

Observations:

- The `invalid-url Source0` warning says that the URL listed in the Source0 directive is unreachable. This is expected, because the specified `example.com` URL does not exist. Presuming

that we expect this URL to work in the future, we can ignore this warning.

- There are many errors, because we intentionally wrote this SPEC file to be uncomplicated and to show what errors `rpmlint` can report.
- The `hardcoded-library-path` errors suggest to use the `%{_libdir}` macro instead of hard-coding the library path. For the sake of this example, we ignore these errors, but for packages going in production you need a good reason for ignoring this error.

This is the output of running `rpmlint` on the SRPM for `pello`:

```
$ rpmlint ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.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-
0.1.1.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 5 errors, 2 warnings.
```

Observations:

- The new `invalid-url URL` error here is about the `URL` directive, which is unreachable. Assuming that we expect the URL to become valid in the future, we can ignore this error.

Checking the pello Binary RPM

When checking Binary RPMs, `rpmlint` checks for more things, including:

1. documentation
2. manual pages
3. consistent use of the
4. Filesystem Hierarchy Standard

This is the output of running `rpmlint` on the Binary RPM for `pello`:

```
$ rpmlint ~/rpmbuild/RPMS/noarch/pello-0.1.1-1.el7.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.
```

Observations:

- The `no-documentation` and `no-manual-page-for-binary` warnings say that the RPM has no documentation or manual pages, because we did not provide any.
- The `only-non-binary-in-usr-lib` warning says that you provided only non-binary artifacts in `/usr/lib/`. 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, we can ignore this warning.

- The `non-executable-script` error warns that the `/usr/lib/pello/pello.py` file has no execute permissions. Since this file contains the shebang , `rpmlint` expects the file to be executable. For the purpose of the example, leave this file without execute permissions and ignore this error.

Apart from the above warnings and errors, our RPM is passing `rpmlint` checks.

Checking the cello SPEC File

This is the output of running `rpmlint` on the SPEC file for `cello`:

```
$ 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.
```

Observations:

- The only warning for `cello.spec` says that the URL listed in the `Source0` directive is unreachable. This is expected, because the specified `example.com` URL does not exist. Presuming that we expect this URL to work in the future, we can ignore this warning.

This is the output of running `rpmlint` on the SRPM file for `cello`:

```
$ rpmlint ~/rpmbuild/SRPMS/cello-1.0-1.el7.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.
```

Observations:

- For the `cello` SRPM there is a new warning, which says that the URL specified in the `URL` directive is unreachable. Assuming the link will be working in the future, we can ignore this warning.

Checking the cello Binary RPM

When checking Binary RPMs, `rpmlint` checks for more things, including:

1. documentation
2. manual pages
3. consistent use of the Filesystem Hierarchy Standard .

This is the output of running `rpmlint` on the Binary RPM for `cello`:

```
$ rpmlint ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el7.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.
```

Observations:

- The `no-documentation` and `no-manual-page-for-binary` warnings say that the RPM has no documentation or manual pages, because we did not provide any.

Apart from the above warnings and errors, our RPM is passing `rpmlint` checks.

Our RPMs are now ready and checked with `rpmlint`. This concludes the tutorial. For more information on packaging RPMs, proceed to Advanced Topics.

Advanced Topics

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

Signing Packages

Signing a package is a way to secure the package for an end user. Secure transport can be achieved with implementation of the HTTPS protocol, which can be done when the package is downloaded just before installing. However, the packages are often downloaded in advance and stored in local repositories before they are used. The packages are signed to make sure no third party can alter the content of a package.

There are three ways to sign a package:

- Adding a signature to an already existing package.
- Replacing the signature on an already existing package.
- Signing a package at build-time.

Adding a Signature to a Package

In most cases packages are built without a signature. The signature is added just before the release of the package.

In order to add another signature to the package package, use the `--addsign` option. Having more than one signature makes it possible to record the package's path of ownership from the package builder to the end-user.

As an example, a division of a company creates a package and signs it with the division's key. The company's headquarters then checks the package's signature and adds the corporate signature to the package, stating that the signed package is authentic.

With two signatures, the package makes its way to a retailer. The retailer checks the signatures and, if they check out, adds their signature as well.

The package now makes its way to a company that wishes to deploy the package. After checking every signature on the package, they know that it is an authentic copy, unchanged since it was first created. Depending on the deploying company's internal controls, they may choose to add their own signature, to reassure their employees that the package has received their corporate approval.

The output from the `--addsign` option:

```
$ rpm --addsign blather-7.9-1.i386.rpm
            Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
```

To check the signatures of a package with multiple signatures:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp md5 OK
```

The two `pgp` strings in the output of the `rpm --checksig` command show that the package has been signed twice.

RPM makes it possible to add the same signature multiple times. The `--addsign` option does not check for multiple identical signatures.

```
$ rpm --addsig blather-7.9-1.i386.rpm
              Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
              Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
              Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp pgp pgp md5 OK
```

The output of the `rpm --checksig` command displays four signatures.

Replacing a Package Signature

To change the public key without having to rebuild each package, use the `--resign` option.

```
$ rpm --resign blather-7.9-1.i386.rpm
            Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
```

To use the `--resign` option on multiple package files:

```
$ rpm --resign b*.rpm
            Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
bother-3.5-1.i386.rpm:
```

Build-time Signing

To sign a package at build-time, use the `rpmbuild` command with the `--sign` option. This requires entering the PGP passphrase.

For example:

```
$ rpmbuild -ba --sign blather-7.9.spec
            Enter pass phrase:

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
Finding dependencies...
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
```

The "Generating signature" message appears in both the binary and source packaging sections. The number following the message indicates that the signature added was created using PGP.

NOTE	When using the `--sign` option for `rpmbuild`, use only `-bb` or `-ba` options for package building. `-ba` option mean build binary **and** source packages.

To verify the signature of a package, use the `rpm` command with `--checksig` option. For example:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp md5 OK
```

Building Multiple Packages

When building multiple packages, use the following syntax to avoid entering the PGP passphrase multiple times. For example when building the `blather` and `bother` packages, sign them by using the

following:

```
$ rpmbuild -ba --sign b*.spec
              Enter pass phrase:

Pass phrase is good.
* Package: blather
…
Binary Packaging: blather-7.9-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
…
Source Packaging: blather-7.9-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
…
* Package: bother
…
Binary Packaging: bother-3.5-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/bother-3.5-1.i386.rpm
…
Source Packaging: bother-3.5-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/bother-3.5-1.src.rpm
```

Mock

Mock is a tool for building packages. It can build packages for different architectures and different Fedora or RHEL versions than the build host has. Mock creates chroots and builds packages in them. Its only task is to reliably populate a chroot and attempt to build a package in that chroot.

Mock also offers a multi-package tool, `mockchain`, that can build chains of packages that depend on each other.

Mock is capable of building SRPMs from source configuration management if the `mock-scm` package is present, then building the SRPM into RPMs. See –scm-enable in the documentation. (From the upstream documentation)

NOTE In order to use Mock on a RHEL or CentOS system, you will need to enable the "Extra Packages for Enterprise Linux" (EPEL) repository. This is a repository provided by the Fedora community and has many useful tools for RPM Packagers, systems administrators, and developers.

One of the most common use cases RPM Packagers have for Mock is to create what is known as a "pristine build environment". By using mock as a "pristine build environment", nothing about the current state of your system affects the RPM Package itself. Mock uses different configurations to specify what the build "target" is, these are found on your system in the `/etc/mock/` directory (once you've installed the `mock` package). You can build for different distributions or releases just by specifying it on the command line. Something to keep in mind is that the configuration files that come with `mock` are targeted at Fedora RPM Packagers, and as such RHEL and CentOS release versions are labeled as "epel" because that is the "target" repository these RPMs would be built for. You simply specify the configuration you want to use (minus the `.cfg` file extension). For example, you could build our `cello` example for both RHEL 7 and Fedora 23 using the following commands without ever having to use different machines.

```
$ mock -r epel-7-x86_64 ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm

$ mock -r fedora-23-x86_64 ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
```

One example of why you might want to use `mock` is if you were packaging RPMs on your laptop and you had a package installed (we'll call it `foo` for this example) that was a `BuildRequires` of that package you were creating but forgot to actually make the `BuildRequires: foo` entry. The build would succeed when you run `rpmbuild` because `foo` was needed to build and it was found on the system at build time. However, if you took the SRPM to another system that lacked `foo` it would fail, causing an unexpected side effect. Mock solves this by first parsing the contents of the SRPM and installing the `BuildRequires` into its chroot which means that if you were missing the `BuildRequires` entry, the build would fail because `mock` would not know to install it and it would therefore not be present in the buildroot.

Another example is the opposite scenario, let's say you need `gcc` to build a package but don't have it installed on your system (which is unlikely as an RPM Packager, but just for the sake of the example let us pretend that is true). With Mock, you don't have to install `gcc` on your system because it will get installed in the chroot as part of `mock`'s process.

Below is an example of attempting to rebuild a package that has a dependency that I'm missing on my system. The key thing to note is that while `gcc` is commonly on most RPM Packager's systems, some RPM Packages can have over a dozen `BuildRequires` and this allows you to not need to clutter up your workstation with otherwise un-needed or un-necessary packages.

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
Installing /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
error: Failed build dependencies: gcc is needed by cello-1.0-1.el7.x86_64

$ mock -r epel-7-x86_64 ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
INFO: mock.py version 1.2.17 starting (python version = 2.7.5)...
Start: init plugins
INFO: selinux enabled
Finish: init plugins
Start: run
INFO: Start(/home/admiller/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm)  Config(epel-7-
x86_64)
```

```
Start: clean chroot
Finish: clean chroot
Start: chroot init
INFO: calling preinit hooks
INFO: enabled root cache
Start: unpacking root cache
Finish: unpacking root cache
INFO: enabled yum cache
Start: cleaning yum metadata
Finish: cleaning yum metadata
Mock Version: 1.2.17
INFO: Mock Version: 1.2.17
Start: yum update
base                                                                    | 3.6 kB
00:00:00
epel                                                                    | 4.3 kB
00:00:00
extras                                                                  | 3.4 kB
00:00:00
updates                                                                 | 3.4 kB
00:00:00
No packages marked for update
Finish: yum update
Finish: chroot init
Start: build phase for cello-1.0-1.el7.src.rpm
Start: build setup for cello-1.0-1.el7.src.rpm
warning: Could not canonicalize hostname: rhel7
Building target platforms: x86_64
Building for target x86_64
Wrote: /builddir/build/SRPMS/cello-1.0-1.el7.centos.src.rpm
Getting requirements for cello-1.0-1.el7.centos.src
 --> Already installed : gcc-4.8.5-4.el7.x86_64
 --> Already installed : 1:make-3.82-21.el7.x86_64
No uninstalled build requires
Finish: build setup for cello-1.0-1.el7.src.rpm
Start: rpmbuild cello-1.0-1.el7.src.rpm
Building target platforms: x86_64
Building for target x86_64
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.v9rPOF
+ umask 022
+ cd /builddir/build/BUILD
+ cd /builddir/build/BUILD
+ rm -rf cello-1.0
+ /usr/bin/gzip -dc /builddir/build/SOURCES/cello-1.0.tar.gz
+ /usr/bin/tar -xf -
+ STATUS=0
+ '[' 0 -ne 0 ']'
+ cd cello-1.0
+ /usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
Patch #0 (cello-output-first-patch.patch):
+ echo 'Patch #0 (cello-output-first-patch.patch):'
```

```
+ /usr/bin/cat /builddir/build/SOURCES/cello-output-first-patch.patch
patching file cello.c
+ /usr/bin/patch -p0 --fuzz=0
+ exit 0
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.UxRVtI
+ umask 022
+ cd /builddir/build/BUILD
+ cd cello-1.0
+ make -j2
gcc -g -o cello cello.c
+ exit 0
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.K3i2dL
+ umask 022
+ cd /builddir/build/BUILD
+ '[' /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64 '!=' / ']'
+ rm -rf /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
++ dirname /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
+ mkdir -p /builddir/build/BUILDROOT
+ mkdir /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
+ cd cello-1.0
+ /usr/bin/make install DESTDIR=/builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64
mkdir -p /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64/usr/bin
install -m 0755 cello /builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/bin/cello
+ /usr/lib/rpm/find-debuginfo.sh --strict-build-id -m --run-dwz --dwz-low-mem-die
-limit 10000000 --dwz-max-die-limit 110000000 /builddir/build/BUILD/cello-1.0
extracting debug info from /builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/bin/cello
dwz: Too few files for multifile optimization
/usr/lib/rpm/sepdebugcrcfix: Updated 0 CRC32s, 1 CRC32s did match.
+ /usr/lib/rpm/check-buildroot
+ /usr/lib/rpm/redhat/brp-compress
+ /usr/lib/rpm/redhat/brp-strip-static-archive /usr/bin/strip
+ /usr/lib/rpm/brp-python-bytecompile /usr/bin/python 1
+ /usr/lib/rpm/redhat/brp-python-hardlink
+ /usr/lib/rpm/redhat/brp-java-repack-jars
Processing files: cello-1.0-1.el7.centos.x86_64
Executing(%license): /bin/sh -e /var/tmp/rpm-tmp.vxtAuO
+ umask 022
+ cd /builddir/build/BUILD
+ cd cello-1.0
+ LICENSEDIR=/builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/share/licenses/cello-1.0
+ export LICENSEDIR
+ /usr/bin/mkdir -p /builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/share/licenses/cello-1.0
+ cp -pr LICENSE /builddir/build/BUILDROOT/cello-1.0-
1.el7.centos.x86_64/usr/share/licenses/cello-1.0
+ exit 0
Provides: cello = 1.0-1.el7.centos cello(x86-64) = 1.0-1.el7.centos
```

```
Requires(rpmlib): rpmlib(CompressedFileNames) <= 3.0.4-1 rpmlib(FileDigests) <= 4.6.0-
1 rpmlib(PayloadFilesHavePrefix) <= 4.0-1
Requires: libc.so.6()(64bit) libc.so.6(GLIBC_2.2.5)(64bit) rtld(GNU_HASH)
Processing files: cello-debuginfo-1.0-1.el7.centos.x86_64
Provides: cello-debuginfo = 1.0-1.el7.centos cello-debuginfo(x86-64) = 1.0-
1.el7.centos
Requires(rpmlib): rpmlib(FileDigests) <= 4.6.0-1 rpmlib(PayloadFilesHavePrefix) <=
4.0-1 rpmlib(CompressedFileNames) <= 3.0.4-1
Checking for unpackaged file(s): /usr/lib/rpm/check-files
/builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
Wrote: /builddir/build/RPMS/cello-1.0-1.el7.centos.x86_64.rpm
warning: Could not canonicalize hostname: rhel7
Wrote: /builddir/build/RPMS/cello-debuginfo-1.0-1.el7.centos.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.JuPOtY
+ umask 022
+ cd /builddir/build/BUILD
+ cd cello-1.0
+ /usr/bin/rm -rf /builddir/build/BUILDROOT/cello-1.0-1.el7.centos.x86_64
+ exit 0
Finish: rpmbuild cello-1.0-1.el7.src.rpm
Finish: build phase for cello-1.0-1.el7.src.rpm
INFO: Done(/home/admiller/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm) Config(epel-7-
x86_64) 0 minutes 16 seconds
INFO: Results and/or logs in: /var/lib/mock/epel-7-x86_64/result
Finish: run
```

As you can see, `mock` is a fairly verbose tool. You will also notice a lot of yum or dnf output (depending on RHEL7, CentOS7, or Fedora mock target) that is not found in this output which was omitted for brevity and is often omitted after you have done an `--init` on a mock target, such as `mock -r epel-7-x86_64 --init` which will pre-download all the required packages, cache them, and pre-stage the build chroot.

For more information, please consult the Mock upstream documentation.

Version Control Systems

When working with RPMs, it is often desireable to utilize a Version Control System (VCS) such as git for managing components of the software we are packaging. Something to note is that storing binary files in a VCS is not favorable because it will drastically inflate the size of the source repository as these tools are engineered to handle differentials in files (often optimized for text files) and this is not something that binary files lend themselves to so normally each whole binary file is stored. As a side effect of this there are some clever utilities that are popular among upstream Open Source projects that work around this problem by either storing the SPEC file where the source code is in a VCS (i.e. - it is not in a compressed archive for redistribution) or place only the SPEC file and patches in the VCS and upload the compressed archive of the upstream release source to what is called a “look aside cache”.

In this section we will cover two different options for using a VCS system, git, for managing the contents that will ultimately be turned into a RPM package. One is called tito and the other is dist-

git.

NOTE	For the duration of this section you will need to install the `git` package on you system in order to follow along.

tito

Tito is an utility that assumes all the source code for the software that is going to be packaged is already in a git source control repository. This is good for those practicing a DevOps workflow as it allows for the team writing the software to maintain their normal Branching Workflow. Tito will then allow for the software to be incrementally packaged, built in an automated fashion, and still provide a native installation experience for RPM based systems.

NOTE	The tito package is available in Fedora as well as in the EPEL repository for use on RHEL 7 and CentOS 7.

Tito operates based on git tags and will manage tags for you if you elect to allow it, but can optionally operate under whatever tagging scheme you prefer as this functionality is configurable.

Let's explore a little bit about tito by looking at an upstream project already using it. We will actually be using the upstream git repository of the project that is our next section's subject, dist-git. Since this project is publicly hosted on GitHub, let's go ahead and clone the git repo.

```
$ git clone https://github.com/release-engineering/dist-git.git
Cloning into 'dist-git'...
remote: Counting objects: 425, done.
remote: Total 425 (delta 0), reused 0 (delta 0), pack-reused 425
Receiving objects: 100% (425/425), 268.76 KiB | 0 bytes/s, done.
Resolving deltas: 100% (184/184), done.
Checking connectivity... done.

$ cd dist-git/

$ ls *.spec
dist-git.spec

$ tree rel-eng/
rel-eng/
├── packages
│   └── dist-git
└── tito.props

1 directory, 2 files
```

As we can see here, the spec file is at the root of the git repository and there is a `rel-eng` directory in the repository which is used by tito for general book keeping, configuration, and various advanced topics like custom tito modules. We can see in the directory layout that there is a sub-directory entitled `packages` which will store a file per package that tito manages in the repository as you can

have many RPMs in a single git repository and tito will handle that just fine. In this scenario however, we see only a single package listing and it should be noted that it matches the name of our spec file. All of this is setup by the command `tito init` when the developers of dist-git first initialized their git repo to be managed by tito.

If we were to follow a common workflow of a DevOps Practitioner then we would likely want to use this as part of a Continuous Integration (CI) or Continuous Delivery (CD) process. What we can do in that scenario is perform what is known as a "test build" to tito, we can even use mock to do this. We could then use the output as the installation point for some other component in the pipeline. Below is a simple example of commands that could accomplish this and they could be adapted to other environments.

```
$ tito build --test --srpm
Building package [dist-git-0.13-1]
Wrote: /tmp/tito/dist-git-git-0.efa5ab8.tar.gz

Wrote: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm

$ tito build --builder=mock --arg mock=epel-7-x86_64 --test --rpm
Building package [dist-git-0.13-1]
Creating rpms for dist-git-git-0.efa5ab8 in mock: epel-7-x86_64
Wrote: /tmp/tito/dist-git-git-0.efa5ab8.tar.gz

Wrote: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm

Using srpm: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm
Initializing mock...
Installing deps in mock...
Building RPMs in mock...
Wrote:
  /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm
  /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm

$ sudo yum localinstall /tmp/tito/dist-git-*.noarch.rpm
Loaded plugins: product-id, search-disabled-repos, subscription-manager
Examining /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm: dist-git-
0.13-1.git.0.efa5ab8.el7.centos.noarch
Marking /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm to be installed
Examining /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm: dist-
git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch
Marking /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm to be
installed
Resolving Dependencies
--> Running transaction check
---> Package dist-git.noarch 0:0.13-1.git.0.efa5ab8.el7.centos will be installed
```

Note that the final command would need to be run with either sudo or root permissions and that much of the output has been omitted for brevity as the dependency list is quite long.

This concludes our simple example of how to use tito but it has many amazing features for traditional Systems Administrators, RPM Packagers, and DevOps Practitioners alike. I would highly recommend consulting the upstream documentation found at the *tito* GitHub site for more information on how to quickly get started using it for your project as well as various advanced features it offers.

dist-git

The dist-git utility takes a slightly different approach from that of tito such that instead of keeping the raw source code in git it instead will keep spec files and patches in a git repository and upload the compressed archive of the source code to what is known as a "look-aside cache". The "look-aside-cache"is a term that was coined by the use of RPM Build Systems storing large files like these "on the side". A system like this is generally tied to a proper RPM Build System such as Koji. The build system is then configured to pull the items that are listed as `SourceX` entries in the spec files in from this look-aside-cache, while the spec and patches remain in a version control system. There is also a helper command line tool to assist in this.

In an effort to not duplicate documentation, for more information on how to setup a system such as this please refer to the upstream dist-git docs.

More on Macros

There are many built-in RPM Macros and we will cover a few in the following section, however an exhaustive list can be found at the RPM Official Documentation.

There are also macros that are provided by your Linux Distribution, we will cover some of those provided by Fedora, CentOS and RHEL in this section as well as provide information on how to inspect your system to learn about others that we don't cover or for discovering them on other RPM-based Linux Distributions.

Defining Your Own Macros

You can define your own macros. Below is an excerpt from the RPM Official Documentation, which provides a comprehensive reference on macros capabilities.

To define a macro, use:

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

All whitespace surrounding `\` is removed. Name may be composed of alphanumeric characters, and the character `_` and must be at least 3 characters in length. A macro without an `(opts)` field is "simple" in that only recursive macro expansion is performed. A parameterized macro contains an `(opts)` field. The `opts` (the string between parentheses) is passed exactly as is to getopt(3) for argc/argv processing at the beginning of a macro invocation.

NOTE

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

- `%define` has local scope, which means that it applies only to a specified part of a SPEC file. In addition, the body of a `%define` macro is expanded when used—it is lazily evaluated.
- `%global` has global scope, which means that it applies to an entire SPEC file. In addition, the body of a `%global` macro is expanded at definition time.

Examples:

```
%global githash 0ec4e58
%global python_sitelib %(%{__python} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())")
```

NOTE

Macros are always evaluated, even in comments. Sometimes it is harmless. But in the second example, we are executing python command to get the content of a macro. This command will be executed even when you comment out the macro. Or when you put the name of the macro into %changelog. To comment out macro, use `%%`. For example: `%%global`.

%setup

Macro `%setup` can be used to build the package with source code tarballs. Standard behavior of the `%setup` macro can be seen in the `rpmbuild` output. At the beginning of each phase macro outputs `Executing(%something)`. For example:

```
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, since `rpmbuild` deletes temporary files after successful build. This displays the setup of environment variables, 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` ensures that we are working in the right directory, removes residues of previous builds, unpacks the source tarball, and sets up some default privileges. There are multiple options to adjust

the behavior of the `%setup` macro.

%setup -q

Option `-q` limits verbosity of `%setup` macro. Only `tar -xof` is executed instead of `tar -xvvof`. This option has to be used as first.

%setup -n

In some cases, the directory from expanded tarball has a different name than expected `%{name}-%{version}`. This can lead to an error of the `%setup` macro. The name of a directory has to be specified by `-n directory_name` option.

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

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

%setup -c

The `-c` option can be used if the source code tarball does not contain any subdirectories and after unpacking, files from an archive fill the current directory. The `-c` option creates the directory and steps into the archive expansion. An illustrative example:

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

The directory is not changed after archive expansion.

%setup -D and -T

`-D` option disables deleting of source code directory. This option is useful if `%setup` macro is used several times. Essentially, `-D` option means that 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 -
```

%setup -a and -b

Options `-a` and `-b` expand specific sources.

- Option `-b` (which stands for `before`) expands specific sources before entering the working directory.
- Option `-a` (which stands for `after`) expands those sources after entering. Their arguments are source numbers from the spec file preamble.

For example, let's say the `cello-1.0.tar.gz` archive contains empty `examples` directory, and the examples are shipped in separate `examples.tar.gz` tarball and they expand into the directory of the same name. In this case use `-a 1`, as we 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
```

But if the examples were in the separate `cello-1.0-examples.tar.gz` tarball, which expands into `cello-1.0/examples`, use `-b 1` options, since the `Source1` should be expanded 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
```

You can also use a combination of all these options.

%files

Common "advanced" RPM Macros needed in the `%files` section are as follows:

Macro	Definition
%license	This identifies the file listed as a LICENSE file and it will be installed and labeled as such by RPM. Example: `%license LICENSE`
%doc	This identifies the file listed as documentation and it will be installed and labeled as such by RPM. This is often used not only for documentation about the software being packaged but also code examples and various items that should accompany documentation. In the event code examples are included, care should be taken to remove executable mode from the file. Example: `%doc README`

%dir	Identifies that the path is a directory that should be 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)	Specifies 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. In the event that 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`

Built-In Macros

Your system has many built-in RPM Macros and the fastest way to view them all is to simply run the `rpm --showrc` command. Note that this will contain a lot of output so it is often used in combination with a pipe to `grep`.

You can also find information about the RPMs macros that come directly with your system's version of RPM by looking at the output of the `rpm -ql rpm` taking note of the files titled `macros` in the directory structure.

RPM Distribution Macros

Different distributions will supply different sets of recommended RPM Macros based on the language implementation of the software being packaged or the specific guidelines of the distribution in question.

These are often provided as RPM Packages themselves and can be installed with the distribution package manager, such as yum or dnf. The macro files themselves once installed can be found in `/usr/lib/rpm/macros.d/` and will be included in the `rpm --showrc` output by default once installed.

One primary example of this is the Fedora Packaging Guidelines section pertaining specifically to Application Specific Guidelines which at the time of this writing has over 60 different sets of guidelines along with associated RPM Macro sets for subject matter specific RPM Packaging.

One example of this kind of RPMs would be for Python version 2.x and if we have the `python2-rpm-macros` package installed (available in EPEL for RHEL 7 and CentOS 7), we have a number of python2 specific macros available to us.

```
$ rpm -ql python2-rpm-macros
/usr/lib/rpm/macros.d/macros.python2

$ rpm --showrc | grep python2
-14: __python2  /usr/bin/python2
CFLAGS="%{optflags}" %{__python2} %{py_setup} %{?py_setup_args} build
--executable="%{__python2} %{py2_shbang_opts}" %{?1}
CFLAGS="%{optflags}" %{__python2} %{py_setup} %{?py_setup_args} install -O1 --skip
-build --root %{buildroot} %{?1}
-14: python2_sitearch   %(%{__python2} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib(1))")
-14: python2_sitelib    %(%{__python2} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())")
-14: python2_version    %(%{__python2} -c "import sys;
sys.stdout.write('{0.major}.{0.minor}'.format(sys.version_info))")
-14: python2_version_nodots     %(%{__python2} -c "import sys;
sys.stdout.write('{0.major}{0.minor}'.format(sys.version_info))")
```

The above output displays the raw RPM Macro definitions, but we are likely more interested in what these will evaluate to which we can do with `rpm --eval` in order to determine what they do as well as how they may be helpful to us when packaging RPMs.

```
$ rpm --eval %{__python2}
/usr/bin/python2

$ rpm --eval %{python2_sitearch}
/usr/lib64/python2.7/site-packages

$ rpm --eval %{python2_sitelib}
/usr/lib/python2.7/site-packages

$ rpm --eval %{python2_version}
2.7

$ rpm --eval %{python2_version_nodots}
27
```

Custom Macros

You can override the distribution macros in the `~/.rpmmacros` file. Any changes you make will affect every build on your machine.

There are several macros you can use to override:

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

: You can create this directory, including all subdirectories using the `rpmdev-setuptree` utility. The value of this macro is by default `~/rpmbuild`.

`%_smp_mflags -l3`

This macro 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.

While you can define any new macros in the `~/.rpmmacros` file, this is discouraged, because those macros would not be present on other machines, where users may want to try to rebuild your package.

Epoch, Scriptlets, and Triggers

There are various topics in the world of RPM SPEC files that are considered advanced because they have implications on not only the SPEC file, how the package is built, but also on the end machine that the resulting RPM is installed upon. In this section we will cover the most common of these such as Epoch, Scriptlets, and Triggers.

Epoch

First on the list is `Epoch`, epoch is a way to define weighted dependencies based on version numbers. It's default value is 0 and this is assumed if an `Epoch` directive is not listed in the RPM SPEC file. This was not covered in the SPEC File section of this guide because it is almost always a bad idea to introduce an Epoch value as it will skew what you would normally otherwise expect RPM to do when comparing versions of packages.

For example if a package `foobar` with `Epoch: 1` and `Version: 1.0` was installed and someone else packaged `foobar` with `Version: 2.0` but simply omitted the `Epoch` directive either because they were unaware of its necessity or simply forgot, that new version would never be considered an update because the Epoch version would win out over the traditional Name-Version-Release marker that signifies versioning for RPM Packages.

This approach is generally only used when absolutely necessary (as a last resort) to resolve an upgrade ordering issue which can come up as a side effect of upstream software changing versioning number schemes or versions incorporating alphabetical characters that can not always be compared reliably based on encoding.

Scriptlets and Triggers

In RPM Packages, there are a series of directives that can be used to inflict necessary or desired change on a system during install time of the RPM. These are called **scriptlets**.

One primary example of when and why you'd want to do this is when a system service RPM is installed and it provides a systemd unit file. At install time we will need to notify systemd that there is a new unit so that the system administrator can run a command similar to `systemctl start foo.service` after the fictional RPM `foo` (which provides some service daemon in this example) has been installed. Similarly, we would need to inverse of this action upon uninstallation so that an administrator would not get errors due to the daemon's binary no longer being installed but the unit file still existing in systemd's running configuration.

There are a small handful of common scriptlet directives, they are similar to the “section headers” like `%build` or `%install` in that they are defined by multi-line segments of code, often written as standard POSIX shell script but can be a few different programming languages such that RPM for the target machine’s distribution is configured to allow them. An exhaustive list of these available languages can be found in the *RPM Official Documentation*.

Scriptlet directives are as follows:

Directive	Definition
`%pre`	Scriptlet that is executed just before the package is installed on the target system.
`%post`	Scriptlet that is executed just after the package is installed on the target system.
`%preun`	Scriptlet that is executed just before the package is uninstalled from the target system.
`%postun`	Scriptlet that is executed just after the package is uninstalled from the target system.

It is also common for RPM Macros to exist for this function. In our previous example we discussed systemd needing to be notified about a new unit file, this is easily handled by the systemd scriptlet macros as we can see from the below example output. More information on this can be found in the Fedora systemd Packaging Guidelines.

```
$ 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
```

Another item that provides even more fine grained control over the RPM Transaction as a whole is what is known as **triggers**. These are effectively the same thing as a scriptlet but are executed in a very specific order of operations during the RPM install or upgrade transaction allowing for a more fine grained control over the entire process.

The order in which each is executed and the details of which are provided 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 from the included RPM documentation found in `/usr/share/doc/rpm/triggers` on Fedora systems and `/usr/share/doc/rpm-4.*/triggers` on RHEL 7 and CentOS 7 systems.

Using Non-Shell Scripts in SPEC File

A scriptlet option, `-p`, in a SPEC file allows to invoke a specific interpreter instead of the default `-p /bin/sh`. An illustrative example is a script, which prints out a message after the installation of `pello.py`.

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

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

   Under this line, insert the following code:

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

3. Build your package according to the Building RPMS chapter.
4. Install your package:

```
# dnf install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.1-1.fc27.noarch.rpm
```

The output of this command is the following message after the installation:

```
Installing       : pello-0.1.1-1.fc27.noarch                                1/1
Running scriptlet: pello-0.1.1-1.fc27.noarch                                1/1
This is python code
```

NOTE

- To use a Python 3 script: Write a line `%post -p /usr/bin/python3` under the line `install -m` in a SPEC file.
- To use a Lua script: Write a line `%post -p <lua>` under the line `install -m` in a SPEC file.
- This way any interpreter can be specified in the SPEC file.

RPM Conditionals

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

Most commonly, conditional inclusions deal with:

- architecture-specific sections
- operating system-specific sections
- compatibility issues between various versions of operating systems
- existence and definition of macros

RPM Conditionals 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
```

RPM Conditionals Examples

The `%if` Conditional

```
%if 0%{?rhel} == 6
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

This conditional handles compatibility between RHEL6 and other operating systems in terms of support of the AS_FUNCTION_DESCRIBE macro. When the package is build for RHEL, the `%rhel` macro is defined and it is expanded to RHEL version. If its value is 6, meaning the package is build for RHEL 6, then the references to AS_FUNCTION_DESCRIBE, which is not supported by RHEL6, are deleted from autoconfig scripts.

```
%if 0%{?el6}
%global ruby_sitearch %(ruby -rrbconfig -e 'puts Config::CONFIG["sitearchdir"]')
%endif
```

This conditional handles compatibility between Fedora version 17 and newer and RHEL6 in terms of support of the `%ruby_sitearch` macro. Fedora version 17 and newer defines `%ruby_sitearch` by default, but RHEL6 does not support this macro. The conditional checks whether the operating system is RHEL6. If it is, `%ruby_sitearch` is defined explicitly. Note that `0%{?el6}` has the same meaning as `0%{?rhel} == 6` from the previous example, and it tests whether a package is built on RHEL6.

```
%if 0%{?fedora} >= 19
%global with_rubypick 1
%endif
```

This conditional handles support for the rubypick tool. If the operating system is Fedora version 19 or newer, rubypick is supported.

```
%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 the macros. If the `%milestone` or the `%revision` macros are set, the `%ruby_archive` macro, which defines the name of the upstream tarball, is redefined.

Specialized variants of `%if` Conditional

The `%ifarch` conditional, `%ifnarch` conditional and `%ifos` conditional are specialized variants of the

`%if` conditionals. These variants are commonly used, so 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.

```
%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.

```
%ifnarch alpha
...
%endif
```

All the contents of the SPEC file between `%ifnarch` and `%endif` are processed only if not being 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.

```
%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.

Appendix A: New features of RPM in RHEL 7

This list documents most noticable changes in RPM packaging between Red Hat Enterprise Linux 6 and 7.

- A new command, `rpmkeys`, used for keyring import and signature verification has been added.
- A new command, `rpmspec`, used for spec queries and parsed output has been added.
- A new command, `rpmsign`, used for package signing has been added.
- The `posix.exec()` and `os.exit()` extensions embedded in `%{lua:…}` scripts fail the script unless called from a child process created with the `posix.fork()` scriptlet.
- The `%pretrans` scriptlet failure causes the package installation to be skipped.
- Scriptlets can be macro-expanded and queryformat-expanded at runtime.
- Pre-transaction and post-transaction scriptlet dependencies can now be correctly expressed with `Requires(pretrans)` and `Requires(posttrans)` scriptlets.
- The `OrderWithRequires` tag for supplying additional ordering hints has been added. The tag follows `Requires` tag syntax, but does not generate actual dependencies. The ordering hints are treated as if they were `Requires` when calculating the transaction order, only if the involved packages are present in the same transaction.
- The `%license` flag can be used in the `%files` section. This flag can be used similar to the `%doc` flag to mark files as licenses, which need to be installed despite the `--nodocs` option.
- The `%autosetup` macro for automating patch application, with optional distributed version control system integration has been added.
- The automatic dependency generator has been rewritten into extensible and customizable rule based system with built-in filtering.
- The OpenPGP V3 public keys are no longer supported.

Appendix B: References

Below are references to various topics of interest around RPMs, RPM packaging, and RPM building. Some of these are advanced and extend far beyond the introductory material included in this guide.

Software Collections - SoftwareCollections.org is an open-source project for building and distributing community-supported Software Collections (SCLs) for Red Hat Enterprise Linux, Fedora, CentOS, and Scientific Linux.

Creating RPM package - Step-by-step guide for learning basics of RPM packaging.

Packaging software with RPM, Part 1, Part 2, Part 3 - IBM RPM packaging guide.

RPM Documentation - The official RPM documentation.

Fedora Packaging Guidelines - The official packaging guidelines for Fedora, useful for all RPM-based distributions.

rpmfluff - python library for building RPM packages, and sabotaging them so they are broken in controlled ways.

Appendix C: Acknowledgements

Certain portions of this text first appeared in the RPM Packaging Guide. Copyright © 2020 Adam Miller and others. Licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.