
Red Hat
Training and
Certification
Red Hat

Copyright © 2019 Red Hat, Inc.

Red Hat
Training and
Certification

RPM Training for Verizon Media Group

Abstract:

In this lab, we'll learn best practices for packaging software using the Red Hat Enterprise Linux native packaging format, RPM. We'll cover how to properly build software from source code into RPM packages, create RPM packages from pre-compiled binaries, and to automate RPM builds from source code version control systems, such as git, for use in CI/DevOps environments. Also in this lab, we'll hear tips and tricks from lessons learned, such as how to set up and work with pristine build environments and why such things are important to software packaging.

Audience/Intro/Prerequisites:

This lab is geared towards Systems Administrators, DevOps Practitioners, and Software Developers who might be interested in learning how to create RPM Packages. Attendees, during this session, will learn:

- What is source code
- How software is made:
  - Natively Compiled
  - Interpreted Programming Languages
- Building software from source
- Patching software
- Installing arbitrary artifacts
- RPM Package Format
- How to setup an RPM Packaging Workspace
- What is an RPM SPEC file
  - Including various directives and sections
  - RPM Macros
- BuildRoots
- How to Build RPMs
- Sanity Checking RPMs
- PGP Signing RPMs With GPG
- Advanced RPM Packaging topics (Appendix)
  - Pristine Build Environments using mock
  - DevOps Workflows using Version Control Systems such as git
  - More on RPM Macros: Language Specific and Defining Your Own
  - Defining Package Epoch
  - Using RPM Scriptlets and Triggers
  - AppStreams and Modularity: The future of Packaging

To accomplish this, they will need a background or experience in at a minimum installing software on Red Hat Enterprise Linux 7 using rpm and yum.

Document Conventions

Code and command line output will be placed into a block similar to the following:

```
This is a block! We can do all sorts of cool code and command line stuff here!

Look, more lines!

$ echo "Here's some command line output!"
Here's some command line output!
```

Topics of interest or vocabulary terms will either be referred to as URLs to their respective documentation/website, as a **bold** item, or in *italics*. The first encounter of the term should be a reference to its respective documentation.
Command line utilities, commands, or things otherwise found in code that are used throughout paragraphs will be written in a monospace font.

Notes are marked as **Note:** and any files that are displayed in their entirety are marked as **File Listing:** FILENAME

Prerequisites

In order to perform the following examples you will need a few packages installed on your system:

Note: The inclusion of some of the packages below are not actually necessary because they are a part of the default installation of Red Hat Enterprise Linux but are listed explicitly for perspective of exactly the tools used within this document.

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

Beyond these preliminary packages you will also need a text editor of your choosing. We will not be discussing or recommending text editors in this document and we trust that everyone has at least one they are comfortable with at their disposal.

General Topics and Background

In this section we will walk through various topics about building software that are helpful background or otherwise general topics that are important for a good RPM Packager to be familiar with.

- What is Source Code?
- How Programs Are Made
- Building from source into an output artifact (what type of artifact will depend on the scenario and we will define what this means more specifically with examples).
- Patching Software
- Placing those output artifacts somewhere on the system that is useful within the Filesystem Hierarchy Standard.

What is Source Code?

Note: If you are familiar with what the following terms mean then feel free to skip this section: source code, programming, programming languages.

In the world of computer software, **source code** is the term used to the representation of instructions to the computer about how to perform a task in a way that is human readable, normally as simple text. This human readable format is expressed using a programming language which basically boils down to a set of rules about that programmers learn so that the text they write is meaningful to the computer.

Note: There are many thousands of programming languages in the world. In this document we will provide examples of only a couple, some finer points of various programming languages are going to vary but hopefully this guide will prove to be a good conceptual overview.

For example, the following three examples are all a very simple program that will display the text `Hello World` to the command line. The reason for three versions of the example will become apparent in the next section but this is three implementations of the same program written in different programming languages. The program is a very common starting place for newcomers to the programming world so it may appear familiar to some readers, but if it doesn't do not worry.

Note: In the first two examples below, the `#!` line is known as a shebang and is not technically part of the programming language source code.

This version of the example is written in the bash shell built in scripting language.

File listing: bello

```
#!/bin/bash

printf "Hello World\n"
```

This version of the example is written in a programming language named Python.

File Listing: pello.py

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

print("Hello World")
```

This version of the example is written in a programming language named C.

File Listing: cello.c

```
#include <stdio.h>

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

The finer points of how to write software isn't necessarily important at this time but if you felt so inclined to learn to program that would certainly be beneficial in your adventures as a software packager.

As mentioned before, the output of both examples to the command line will be simply, Hello World when the source code is built and run. The topic of how that happens is up next!

How Programs Are Made

Before we dive too far into how to actually build code it is best to first understand a few items about software source code and how it becomes instructions to the computer. Effectively, how programs are actually made. There are many ways in which a program can be executed but it boils down to effectively two common methods:

1. Natively Compiled
2. Interpreted (Byte Compiled and Raw Interpreted)

Natively Compiled Code

Software written in programming languages that compile to machines code or directly to a binary executable (i.e. - something that the computer natively understands without an help) that can be run stand alone is considered to be **Natively Compiled**. This is important for building RPM Packages because packages built this way are what is known as architecture specific, meaning that if you compile this particular piece of software on a computer that uses a 64-bit (x86_64) AMD or Intel processor, it will not execute on a (x86) 32-bit AMD or Intel processor. The method by which this happens will be covered in the next section.

Interpreted Code

There are certain programming languages that do not compile down to a representation of program that the computer natively understands. These programs are **Interpreted** and require a Language Interpreter or Language Virtual Machine(VM). The name *interpreter* comes from it's similarities with how human language interpreters convert between two representations of human speech to allow two people to talk, a programming language interpreter converts from a format that the computer doesn't "speak" to one that it does.

There are two types of Interpreted Languages, Byte Compiled and Raw Interpreted and the distinction between these is useful to keep in mind when packaging software because of the actual `%build` process is going to be very different and sometimes in the case of Raw Interpreted Languages there will be no series of steps required at all for the `%build`. (What `%build` means in detail will be explained later, but the short version is this is how we tell the RPM Packaging system to actually perform the *build*). Where as Byte Compiled programming languages will perform a build task that will "compile" or "translate" the code from the programming language source that is human readable to an intermediate representation of the program that is more efficient for the programming language interpreter to execute.

Software written entirely in programming languages such as bash shell script and Python (as used in our example) are *Interpreted* and therefore are not architecture specific which means the resulting RPM Package that is created will be considered `noarch`. This indicates that it does not have an architecture associated with it.

Building Software from Source

In this section we will discuss and provide examples of building software from its source code.

Note: If you are comfortable building software from source code please feel free to skip this section and move on. However, if you'd like to stick around and read it then please feel free and it will hopefully serve as a refresher or possibly contain something interesting that's new to you.

Source code must go through a **build** process and that process will vary based on specific programming language but most often this is referred to as **compiling** or **translating** the software. For software written in interpreted programming languages this step may not be necessary but sometimes it is desirable to perform what is known as **byte compiling** as it's build process. We will cover each scenario below. The resulting built software can then be **run** or "**executed**" which tells the computer to perform the task described to it in the source code provided by the programmer who authored the software.

Note: There are various methods by which software written in different programming languages can vary heavily. If the software you are interested in packaging doesn't follow the exact examples here, this will hopefully be an objective guideline.

Natively Compiled Code

Referencing the example previously used that is written in C (listed again below for the sake of those who may have skipped the previous section), we will build this source code into something the computer can execute.

File Listing: cello.c

```
#include <stdio.h>

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

Build Process

In the below example we are going to invoke the C compiler from the GNU Compiler Collection (GCC).

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

From here we can actually execute the resulting output binary.

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

That's it! You've built natively compiled software from source code!

Let's take this one step further and add a GNU make Makefile which will help automate the building of our code. This is an extremely common practice by real large scale software and is a good thing to become familiar with as an RPM Packager. Let's create a file named `Makefile` in the same directory as our example C source code file named `cello.c`.

File Listing: Makefile

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

clean:
        rm cello
```

Now to build our software we can simply run the command make, below you will see the command run more than once just for the sake of seeing what is expected behavior.

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

$ make clean
rm cello

$ make
gcc -o cello cello.c

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

$ ./cello
Hello World
```

Congratulations! You have now both compiled software manually and used a build tool!

Interpreted Code

For software written in interpreted programming languages we know that we don't need to compile it, but if it's a byte compiled language such as Python there may still be a step required. Referencing the two examples previously (listed again below for the sake of those who may have skipped the previous section), for Python we will build this source code into something the Python Language Interpreter (known as CPython) can execute.

Note: In the two examples below, the `#!` line is known as a shebang and is not technically part of the programming language source code.
The shebang allows us to use a text file as an executable and the system program loader will parse the line at the top of the file containing a `#!` character sequence looking a path to the binary executable to use as the programming language interpreter.

Byte Compiled Code

As mentioned previously, this version of the example is written in a programming language named Python and its default language virtual machine is one that executes *byte compiled* code. This will "compile" or "translate" the source code into an intermediate format that is optimized and will be much faster for the language virtual machine to execute.

File Listing: pello.py

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

print("Hello World")
```

The exact procedure to byte compile programs based on language will differ heavily based on the programming language, its language virtual machine, and the tools or processes that are common within that programming language's community. Below is an example using Python.

Note: The practice of byte compiling Python is common but the exact procedure shown here is not. This is meant to be a simple example. For more information, please reference the Software Packaging and Distribution documentation.

```
$ python -m compileall pello.py
$ python pello.pyc
Hello World

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

You can see here that after we byte-compiled the source `.py` file we now have a `.pyc` file which is of `python 2.7 byte-compiled` filetype. This file can be run with the python language virtual machine and is more efficient than passing in just the raw source file, which is a desired attribute of resulting software we as an RPM Packager will distribute out to systems.

Raw Interpreted

This version of the example is written in the bash shell built in scripting language.

File Listing: bello

```
#!/bin/bash

printf "Hello World\n"
```

UNIX-style shells have scripting languages, much like bash does, but programs written in these languages do not have any kind of byte compile procedure and are interpreted directly as they are written so the only procedure we have to do is make the file executable and then run it.

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

Patching Software

In software and computing a **patch** is the term given to source code that is meant to fix other code, this is similar to the way that someone will use a piece of cloth to patch another piece of cloth that is part of a shirt or a blanket. Patches in software are formatted as what is called a *diff* since it represents what is *different* between to pieces of source code. A *diff* is created using the `diff` command line utility that is provided by diffutils and then it is applied to the original source code using the tool patch.

Note: Software developer will often use “Version Control Systems” such as git to manage their code base. Tools like these provide their own methods of creating diffs or patching software but those are outside the scope of this document.

Let’s walk through an example where we create a patch from the original source code using `diff` and then apply it using the patch utility. We will revisit patching software in a later section when it comes to actually building RPMs and hopefully this exercise will prove it’s usefulness at that time. First step in patching software is to preserve the original source code because we want to keep the original source code in pristine condition as we will “patch it” instead of simply modifying it. A common practice for this is to copy it and append `.orig` to the filename. Let’s do that now.

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

Next, we want to make an edit to cello.c using our favorite text editor. Update your cello.c to match the output below.

File Listing: cello.c

```
#include <stdio.h>

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

Now that we have our original source code preserved and the updated source code written, we can generate a patch using the `diff` utility.

Note: Here we are using a handful of common arguments for the `diff` utility and their documentation is out of the scope of this document. Please reference the manual page on your local machine with: `man diff` for more information.

```
$ diff -Naur cello.c.orig cello.c
--- cello.c.orig        2016-05-26 17:21:30.478523360 -0500
+++ cello.c     2016-05-27 14:53:20.668588245 -0500
@@ -1,6 +1,6 @@
 #include<stdio.h>

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

In this output, you can see the line that starts with a `-` are being removed from the original source code and replaced by the line that starts with `+`. Let's now save that output to a file this time by redirecting the output so that we can use it later with the patch utility. It is not a requirement but it's good practice to use a meaningful filename when creating patches.

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

Now we want to restore the `cello.c` file to its original source code such that it is restored to its pristine state and we can patch it with our new patch file. The reason this particular process is important is because this is how it is done when building RPMs, the original source code is left in pristine condition and we patch it during the process that prepares to source code to be built.

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

Next up, let's go ahead and patch the source code by redirecting the patch file to the patch command.

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

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

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

From the output of the cat command we can see that the patch has been successfully applied, let's build and run it now.

```
$ make clean
rm cello

$ make
gcc -o cello cello.c

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

Congratulations, you have successfully created a patch, patched software, built the patched software and run it!

Next up, installing things!

Installing Arbitrary Artifacts

One of the many really nice things about Linux systems is the Filesystem Hierarchy Standard (FHS) which defines areas of the filesystem in which things should be placed. As an RPM Packager this is extremely useful because we will always know where to place things that come from our source code.

This section references the concept of an **Arbitrary Artifact** which in this context is anything you can imagine that is a file that you want to install somewhere on the system within the FHS. It could be a simple script, a pre-existing binary, the binary output of source code that you have created as a side effect of a build process, or anything else you can think up. We discuss it in such a vague vocabulary in order to demonstrate that neither the system nor RPM care what the *Artifact* in question is. To both RPM and the system, it is just a file that needs to exist in a predetermined place. The permissions and the type of file it is makes it special to the system but that is for us as an RPM Packager to decide.

For example, once we have built our software we can then place it on the system somewhere that will end up in the system $PATH so that they can be found and executed easily by users, developers, and sysadmins alike. We will explore two ways to accomplish this as they each are quite popular approaches used by RPM Packagers.

install command

When placing arbitrary artifacts onto the system without build automation tooling such as GNU make or because it is a simple script and such tooling would be seen as unnecessary overhead, it is a very common practice to use the `install` command (provided to the system by coreutils) to place the artifact in a correct location on the filesystem based on where it should exist in the FHS along with appropriate permissions on the target file or directory.

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.

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

At this point, we can execute `bello` from our shell no matter what our current working directory is because it has been installed into our $PATH.

```
$ cd ~/

$ bello
```

```
Hello World
```

make install

A very popular mechanism by which you will install software from source after it's built is by using a command called `make install` and in order to do that we need to enhance the `Makefile` we created previously just a little bit.

Note: The creation of Makefile is normally done by the developer who writes the original source code of the software in question and as an RPM Packager this is not generally something you will need to do. This is purely an exercise for background knowledge and we will expand upon this as it relates to RPM Packaging later.

Open the `Makefile` file up in your favorite text editor and make the appropriate edits needed so that it ends up looking exactly as the following.

Note: The use of $(DESTDIR) is a GNU make built-in and is commonly used to install into alternative destination directories.

File Listing: Makefile

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

clean:
	rm cello

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

Now we are able to use the make file to both build and install the software from source. Note that for the installation portion, like before when we ran the raw `install` command, you will need either sudo permissions or be the `root` user and omit the `sudo` portion of the command. The following will build and install the simple `cello.c` program that we had written previously.

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

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

Just as in the previous example, we can now execute `cello` from our shell no matter what our current working directory is because it has been installed into our $PATH.

```
$ cd ~/

$ cello
Hello World
```

Congratulations, you have now installed a build artifact into its proper location on the system!

Prepping our example upstream source code

Note: If you're familiar with how upstream software is distributed and would like to skip this, please feel free to download the example source code for our fake upstream project and skip this section. However if you are curious how the examples are created please feel free to read along.

Now that we have our RPM Packaging Workspace setup, we should create simulated upstream compressed archives of the example programs we have made. We will once again list them here just in case a previous section was skipped.

Note: What we are about to do here in this section is not normally something an RPM Packager has to do, this is normally what happens from an upstream software project, product, or developer who actually releases the software as source code. This is simply to setup the RPM Build example space and give some insight into where everything actually comes from.

We will also assume GPLv3 as the Software License for all of these simulated upstream software releases. As such, we will need a `LICENSE` file included with each source code release. We include this in our simulated upstream software release because encounters with a Software License when packaging RPMs is a very common occurrence for an RPM Packager and we should know how to properly handle them.

Note: The method used below to create the `LICENSE` file is known as a here document.

Let us go ahead and make a `LICENSE` file that can be included in the source code "release" for each example.

```
$ cat > /tmp/LICENSE <<EOF
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/>.
EOF
```

Each implementation of the `Hello World` example script will be created into a gzip compressed tarball which will be used to simulate what an upstream project might release as it's source code to then be consumed and packaged for distribution.

Below is an example procedure for each example implementation.

bello

For the bash example implementation we will have a fake project called *bello* and since the project named *bello* produces one thing and that's a shell script named bello then it will only contain that in it's resulting `tar.gz`. Let's pretend that this is version `0.1` of that software and we'll mark the tar.gz file as such. Below is the listing of the file as mentioned before.

File Listing: bello

```
#!/bin/bash

printf "Hello World\n"
```

Let's make a project tar.gz out of our source code.

```
$ mkdir /tmp/bello-0.1

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

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

$ 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

For the Python example implementation we will have a fake project called *pello* and since the project named *pello* produces one thing and that's a small program named `pello.py` then it will only contain that in it's resulting `tar.gz`. Let's pretend that this is version `0.1.1` of this software and we'll mark the `tar.gz` file as such.

Here is the listing of the file as mentioned before.

File Listing: pello.py

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

print("Hello World")
```

Let's make a project tar.gz out of our source code.

```
$ mkdir /tmp/pello-0.1.1

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

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

$ 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

For the C example implementation we will have a fake project called *cello* and since the project named *cello* produces two things, the source code to our program named `cello.c` and a `Makefile` we will need to make sure and include both of these in our `tar.gz`. Let's pretend that this is version `1.0` of the software and we'll mark the `tar.gz` file as such.

Here is the listing of the files involved as mentioned before.

You will notice the `patch` file is listed here, but it will not go in our project tarball because it is something that we as the RPM Packager will apply and not something that comes from the upstream source code. RPM Packages are built in such a way that the original upstream source code in preserved in its pristine form just as released by its creator. All patches required to the software happen at RPM Build time, not before. We will place that in the `~/rpmbuild/SOURCES/` directory along side the “upstream” source code that we are simulating here. (More on this later).

File Listing: cello.c

```
#include <stdio.h>

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

File Listing: cello-output-first-patch.patch

```
--- 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 1;
 }
```

File Listing: Makefile

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

clean:
        rm cello

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

Let’s make a project `tar.gz` out of our source code.

```
$ 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/

$ 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/

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

Great, now we have all of our upstream source code prep'd and ready to be turned into RPMs!

Red Hat
Training and
Certification

RPM Packaging Guide

Hello! Welcome to the RPM Packaging portion of From Source to RPM in 120 Minutes. Here you will find all of the information you need in order to start packaging RPMs for various Linux Distributions that use the RPM Packaging Format.

This guide assumes no previous knowledge about packaging software for any Operating System, Linux or otherwise. However, it should be noted that this guide is written to target the Red Hat "family" of Linux distributions, which are:

- Fedora
- CentOS
- Red Hat Enterprise Linux (often referred to as RHEL for short)

While these distros are the target environment, it should be noted that lessons learned here should be applicable across all distributions that are RPM based but the examples will need to be adapted for distribution specific items such as prerequisite installation items, guidelines, or macros. (More on macros later)

Note: If you have made it this far and don't know what a software package or a GNU/Linux distribution is, you might be best served by exploring some articles on the topics of Linux and Package Managers.

RPM Packages

In this section we are going to hopefully cover everything you ever wanted to know about the RPM Packaging format, and if not then hopefully the contents of the Appendix will satisfy the craving for knowledge that has been left out of this section.

What is an RPM?

To kick things off, let's first define what an RPM actually is. An RPM package is simply a file that contains some files as well as information the system needs to know about those files. More specifically, it is a file containing a cpio archive and metadata about itself. The cpio archive is the payload and the RPM Header contains the metadata. The package manager `rpm` uses this metadata to determine things like dependencies, where to install files, etc.
Conventionally speaking there are two different types of RPM, there is the Source RPM (SRPM) and the binary RPM. Both of these share a file format and tooling, but they represent very different things. The payload of a SRPM is a SPEC file (which describes how to build a binary RPM) and the actual source code that the resulting binary RPM will be built out of (including any patches that may be needed).

RPM Packaging Workspace

In the Prerequisites section we installed a package named `rpmdevtools` which provides a number of handy utilities for RPM Packagers.
Feel free to explore the output of the following command and check out the various utilities manual pages or help dialogs.

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

For the sake of setting up our RPM Packaging workspace let's use the rpmdev-setuptree utility to create our directory layout. We will then define what each directory in the directory structure is meant for.

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

Directory	Purpose
BUILD	Various `%buildroot` directories will be created here when packages are built. This is useful for inspecting a postmortem of a build that goes bad if the logs output don't provide enough information.
RPMS	Binary RPMs will land here in subdirectories of Architecture. For example: `noarch` and `x86_64`
SOURCES	Compressed source archives and any patches should go here, this is where the `rpmbuild` command will look for them.
SPECS	SPEC files live here.
SRPMS	When the correct arguments are passed to `rpmbuild` to build a Source RPM instead of a Binary RPM, the Source RPMs (SRPMS) will land in this directory.

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 between the *Preamble* and the *Body*. Within the *Preamble* we will define a series of metadata items that will be used throughout the *Body* and the *Body* is where the bulk of the work is accomplished.

Preamble Items

In the table below you will find the items that are used in RPM Spec files in the Preamble section.

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 initial value should normally be 1%{?dist}, this value should be incremented each new release of the package and 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 that are destined for 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 is ideally a listing of the upstream URL resting place and not just a local copy of the source. If needed, more SourceX directives can be added, incrementing the number each time such as: 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 such as: Patch1, Patch2, Patch3, and so on.

BuildArch	If the package is not architecture dependent, i.e. written entirely in an interpreted programming language, this should be BuildArch: noarch otherwise it will automatically inherit the Architecture of the machine it's being built on.
BuildRequires	A comma or whitespace separated list of packages required for building (compiling) the program. There can be multiple entries of BuildRequires each on it's own line in the SPEC file.
Requires	A comma or whitespace separated list of packages that are required by the software to run once installed. There can be multiple entries of Requires each on it's own line in the SPEC file.
ExcludeArch	In the event a piece of software can not operate on a specific processor architecture, you can exclude it here.

There are three "special" directives listed above which are `Name`, `Version`, and `Release` they are used to create the RPM package's filename. You will often see these referred to by other RPM Package Maintainers and Systems Administrators as **N-V-R** or just simply **NVR** as RPM package filenames are of `NAME-VERSION-RELEASE` format.

For example, if we were to query about a specific package:

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

Here `python` is our Package Name, `2.7.5` is our Version, and `34.el7` is our Release. The final marker is `x86_64` and is our architecture, which is not something we control as an RPM Packager (with the exception of `noarch`) but is a side effect of the `rpmbuild` build environment. We'll cover both of these later.

Body Items

In the table below you will find the items that are used in RPM Spec files in the body.

SPEC Directive	Definition
%description	A full description of the software packaged in the RPM, this can consume multiple lines and be broken into paragraphs.
%prep	Command or series of commands to prepare the software to be built. Example is to uncompress the archive in `Source0`. This can contain shell script.

%build	Command or series of commands used to actually perform the build procedure (compile) of the software.
%install	Command or series of commands used to actually install the various artifacts into a resulting location in the FHS. Something to note is that this is done within the relative context of the `%buildroot` (more on that later).
%check	Command or series of commands to “test” the software. This is normally things such as unit tests.
%files	The list of files that will be installed in their final resting place in the context of the target system.
%changelog	A record of changes that have happened to the package between different `Version` or `Release` builds.

Advanced items

There are a series of advanced items including what are known as *scriptlets* and *triggers* which take effect at different points throughout the installation process on the target machine (not the build process). These are out of the scope of this document, but there is plenty of information on them in the Appendix.

BuildRoots

The term “buildroot” is unfortunately ambiguous and you will often get various different definitions. However in the world of RPM Packages this is literally a chroot environment such that you are creating a filesystem hierarchy in a new “fake” root directory much in the way these contents can be laid down upon an actual system’s filesystem and not violate it’s integrity. Imagine this much in the same way that you would imagine creating the contents for a tarball such that it would be expanded at the root (/) directory of an existing system as this is effectively what RPM will do at a certain point during an installation transaction. Ultimately the payload of the resulting Binary RPM is extracted from this environment and put into the cpio archive.

RPM Macros

A 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 we can have RPM perform text substitutions for us so that we don’t have to.

An example of how this can be extremely useful for an RPM Packager is if we wanted to reference the Version of the software we are packaging multiple times throughout our SPEC file but only want to define it one time. We would then use the `%{version}` macro and it would be

substituted in place by whatever the actual version number is that was entered in the Version field of the SPEC.

Note: A handy utility of the `rpm` command for packagers is the `--eval` option which allows you to ask rpm to evaluate a macro. If you see one in a SPEC file that you're not familiar with, you can quickly find out what it evaluates to.

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

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

A common macro we will encounter as a packager is %{?dist} which signifies the "distribution tag" allowing for a short textual representation of the distribution used for the build to be injected into a text field.

For example:

```
$ rpm --eval %{?dist}
.el7
```

For more information, please reference the More on Macros section of the Appendix.

Working with SPEC files

As an RPM Packager, you will likely spend a large majority of your time, when packaging software, editing the SPEC file. The SPEC file is the recipe we use to tell `rpmbuild` how to actually perform a build. In this section we will discuss how to create and modify a spec file.

When it comes time to package new software, a new SPEC file must be created. We *could* write one from scratch from memory but that sounds boring and tedious, so let's not do that. The good news is that we're in luck and there's a utility called rpmdev-newspec. This utility will create a new SPEC file for us. We will just fill in the various directives or add new fields as needed. This provides us with a nice baseline template.

If you have not already done so by way of another section of the guide, download the example programs now and place them in your `~/rpmbuild/SOURCES` directory.

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

Let's go ahead and create a SPEC file for each of our three implementations of our example and then we will look at the SPEC files and make edits from there.

Note: Some programmer focused text editors will pre-populate a new file with the extension `.spec` with a SPEC template of their own but `rpmdev-newspec` is an editor-agnostic method which is why it is chosen here.

```
$ 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.
```

You will now find three SPEC files in your `~/rpmbuild/SPECS/` directory all matching the names you passed to `rpmdev-newspec` but with the `.spec` file extension. Take a moment to look at the files using your favorite text editor, the directives should look familiar from the What is a SPEC File? section. We will discuss the exact information we will input into these fields in the following sections that will focus specifically on each example.

Note: The `rpmdev-newspec` utility does not use Linux Distribution specific guidelines or conventions, however this document is targeted towards using conventions and guidelines for Fedora and RHEL so you will notice we remove the use of `rm $RPM_BUILD_ROOT` as it is no longer necessary to perform that task when building on RHEL 7.0 or newer or on Fedora version 18 or newer. We also will favor the use of `%{buildroot}` notation over `$RPM_BUILD_ROOT` when referencing RPM's Buildroot for consistency with all other defined or provided macros throughout the SPEC

There are three examples below, each one is meant to be self-sufficient in instruction such that you can jump to a specific one if it matches your needs for packaging. However, feel free to read them straight through for a full exploration of packaging different kinds of software.

Software Name	Explanation of example
bello	Software written in a raw interpreted programming language does doesn't require a build but only needs files installed. If a pre-compiled binary needs to be packaged, this method could also be used since the binary would also just be a file.

pello	Software written in a byte-compiled interpreted programming language used to demonstrate the installation of a byte compile process and the installation of the resulting pre-optimized files.
cello	Software written in a natively compiled programming language to demonstrate an common build and installation process using tooling for compiling native code.

bello

Our first SPEC file will be for our example written in bash shell script that you downloaded (or you created a simulated upstream release in the General Topics and Background Section) and placed it's source code into `~/rpmbuild/SOURCES/` earlier. Let's go ahead and open the file `~/rpmbuild/SPECS/bello.spec` and start filling in some fields.

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

File Listing: bello.spec

```
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>
-
```

Let us 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 *bello* source code is, which we can observe is `0.1` as set by the example code we downloaded (or we created in the General Topics and Background 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, bello version `0.2` 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:           bello
Version:        0.1
Release:        1%{?dist}
Summary:        Hello World example implemented in bash script
```

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/bello`. However, we will use the rpm macro variable of `%{name}` in its place for consistency and the resulting entry will be `https://example.com/%{name}`.

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/bello/releases/bello-0.1.tar.gz` and while we might want to, 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`. 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:           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
```

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 there is no **build** because the bash script is a raw interpreted programming language so we will only be installing files into locations on the system, but it does require the bash shell environment in order to execute so we will need to define `bash` as a requirement using the `Requires` directive.

Since we don't have a build step, we can simply omit the `BuildRequires` directive. There is no need to define is as "undefined" or otherwise, omitting its inclusion will suffice.

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:           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
```

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. However, since this software doesn’t need to be built we can simply leave this section blank (removing what was provided by the template).

The `%install` section is where we instruct `rpmbuild` how to install our previously built software (in the event of a build process) 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. Since the only thing we need to do in order to install `bello` into this environment is create the destination directory for the executable bash script file and then install the file into that directory, we can do so by using the same `install` command but we will make a slight modification since we are inside the SPEC file and we will use the macro variable of `%{name}` in it’s place for consistency.

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 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 `bello` file we are installing will be `%{_bindir}/%{name}` (this would be `/usr/bin/bello` if we weren't using the rpm macros).

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 bello package
- Example second item in the changelog for version-release 0.1-1
```

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 **bello**! In the next section we will cover how to build the RPM!

The full SPEC file should now look like the following:

File Listing: bello.spec

```
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
```

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 General Topics and Background Section) and placed it's 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`.

File Listing: pello.spec

```
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 General Topics and Background 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 it's 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
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 General Topics and Background 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. The option we're going to show in this example, however, 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 the Appendix, 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:

File Listing: pello.spec

```
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
- Example second item in the changelog for version-release 0.1.1-1
```

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 it's 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`.

File Listing: cello.spec

```
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 General Topics and Background 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 it's 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 `%setup` section. We will need to add 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> - 1.0-1
- 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:

File Listing: cello.spec

```
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
```

Building RPMS

When building RPMs there are is one main command, which is `rpmbuild` and we will use that throughout the guide. It has been eluded to in various sections in the guide but now we're actually going to dig in and get our hands dirty.

We will cover a couple different combinations of arguments we can pass to `rpmbuild` based on scenario and desired outcome but we will focus primarily on the two main targets of building an RPM and that is creating Source and Binary RPMs.

One of the things you may notice about `rpmbuild` is that it expects the directory structure created in a certain way and for various items such as source code to exist within the context of that directory structure. Luckily, this is the same directory structure that was setup by the `rpmdev-setuptree` utility that we used previously to setup our RPM workspace and we have been placing files in the correct place throughout the duration of the guide.

Source RPMs

Before we actually build a Source RPM, let's quickly address why we would want to do this. First, we might want to preserve the exact source of a `Name-Version-Release` of RPM that we deployed to our environment that included the exact SPEC file, the source code, and all relevant patches. This can be useful when looking back in history and/or debugging if something has gone wrong. Another reason is if we want to build a Binary RPM on a different hardware platform or architecture.

In order to create a Source RPM we need to pass the "build source" or `-bs` option to `rpmbuild` and we will provide a SPEC file as the argument. We will do so for each of our examples we've created above.

```
$ cd ~/rpmbuild/SPECS

$ rpmbuild -bs bello.spec
Wrote: /home/admiller/rpmbuild/SRPMS/bello-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
```

That's it! That's all there is to building a Source RPM or SRPM. Do note the directory that it was placed in though, this is also a part of the directory hierarchy that we covered previously.

Now it's time to move on to Binary RPMs!

Binary RPMS

When building Binary RPMs there are a few methods by which we could do this, we could "rebuild" a SRPM by passing the `--rebuild` option to `rpmbuild`. We could tell rpmbuild to "build binary" or `-bb` and pass a SPEC file as the argument similar to how we did for the Source RPMs.

Rebuild

Let's first rebuild each of our examples. Below you will see the example output generated from rebuilding each example SRPM. You will notice the output will vary differently based on the specific example you view and that the amount of detail provided is quite verbose. This maybe seem daunting at first but as you become a seasoned RPM Packager you will learn to appreciate and even welcome this level of detail as it can prove to be very valuable when diagnosing issues.

One important distinction to make about when `rpmbuild` is invoked with the `--rebuild` argument is that it actually installs the contents of the SRPM into your `~/rpmbuild` directory which will install the SPEC file and source code, then the build is performed and the SPEC file and Source code are removed. This might seem odd at first, but know that this is expected behavior and you can perform a `--recompile` which will not do the "clean up" operation at the end. We selected to use `--rebuild` in this guide to demonstrate how this happens and how you can "recover" from it to get the SPEC files and SOURCES back which is covered in the following section.

The commands required for each are as follows, with detailed output provided for each below:

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
```

Now you've built RPMs!

You will now find the resulting Binary RPMs in `~/rpmbuild/RPMS/` depending on your architecture and/or if the package was `noarch`.

At the end of each of these commands you will find that there are no longer SPEC files or contents in SOURCES for the specific SRPMs that you rebuilt because of how `--rebuild` cleans up after itself. We can resolve this by executing the following rpm commands which will perform an install of the SRPMs. You will want to do this after running a `--rebuild` if you want to continue to interact with the SPEC and SOURCES which we will want to do for the duration of this guide.

```
$ 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%]
```

Note: Some of the output below has been omitted for brevity and has been marked by an ellipsis (. . .).

bello

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
Installing /home/admiller/rpmbuild/SRPMS/bello-0.1-1.el7.src.rpm
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.GHTHCO
Wrote: /home/admiller/rpmbuild/RPMS/noarch/bello-0.1-1.el7.noarch.rpm
...
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.R9eRPW
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ cd bello-0.1
+ /usr/bin/rm -rf /home/admiller/rpmbuild/BUILDROOT/bello-0.1-1.el7.x86_64
+ exit 0
Executing(--clean): /bin/sh -e /var/tmp/rpm-tmp.S59sAf
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ rm -rf bello-0.1
+ exit 0
```

pello

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
Installing /home/admiller/rpmbuild/SRPMS/pello-0.1.1-1.el7.src.rpm
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.kRf2qV
...
```

```
Wrote: /home/admiller/rpmbuild/RPMS/noarch/pello-0.1.1-1.el7.noarch.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.kZTRbM
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ cd pello-0.1.1
+ /usr/bin/rm -rf /home/admiller/rpmbuild/BUILDROOT/pello-0.1.1-1.el7.x86_64
+ exit 0
Executing(--clean): /bin/sh -e /var/tmp/rpm-tmp.WChx3z
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ rm -rf pello-0.1.1
+ exit 0
```

cello

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
Installing /home/admiller/rpmbuild/SRPMS/cello-1.0-1.el7.src.rpm
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.ySAWzh
...
Wrote: /home/admiller/rpmbuild/RPMS/x86_64/cello-1.0-1.el7.x86_64.rpm
Wrote: /home/admiller/rpmbuild/RPMS/x86_64/cello-debuginfo-1.0-1.el7.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.oexkNU
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ cd cello-1.0
+ /usr/bin/rm -rf /home/admiller/rpmbuild/BUILDROOT/cello-1.0-1.el7.x86_64
+ exit 0
Executing(--clean): /bin/sh -e /var/tmp/rpm-tmp.ENKUE1
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ rm -rf cello-1.0
+ exit 0
```

Build Binary

Next up, let's "build binary" for each of our examples. Just as in the previous example, you will again see the example output generated from building each example. Similarly you will notice the output will vary differently based on the specific example you view and that the amount of detail provided is quite verbose.

The commands required for each are as follows, with detailed output provided for each below:

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

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

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

Now you've built RPMs!

You will now find the resulting Binary RPMs in `~/rpmbuild/RPMS/` depending on your architecture and/or if the package was `noarch`.

Note: Some of the output below has been omitted for brevity and has been marked by an ellipsis (`...`).

bello

```
$ rpmbuild -bb ~/rpmbuild/SPECS/bello.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.aaCBH0
...
Wrote: /home/admiller/rpmbuild/RPMS/noarch/bello-0.1-1.el7.noarch.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.74OMCd
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ cd bello-0.1
+ /usr/bin/rm -rf /home/admiller/rpmbuild/BUILDROOT/bello-0.1-1.el7.x86_64
+ exit 0
```

pello

```
$ rpmbuild -bb pello.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.dvOeYv
...
Wrote: /home/admiller/rpmbuild/RPMS/noarch/pello-0.1.1-1.el7.noarch.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.4tTJSw
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ cd pello-0.1.1
+ /usr/bin/rm -rf /home/admiller/rpmbuild/BUILDROOT/pello-0.1.1-1.el7.x86_64
+ exit 0
```

cello

```
$ rpmbuild -bb ~/rpmbuild/SPECS/cello.spec
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.FveYdS
...
Wrote: /home/admiller/rpmbuild/RPMS/x86_64/cello-1.0-1.el7.x86_64.rpm
Wrote: /home/admiller/rpmbuild/RPMS/x86_64/cello-debuginfo-1.0-1.el7.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.ZRORXv
+ umask 022
+ cd /home/admiller/rpmbuild/BUILD
+ cd cello-1.0
```

```
+ /usr/bin/rm -rf /home/admiller/rpmbuild/BUILDROOT/cello-1.0-1.el7.x86_64
+ exit 0
```

Checking RPMs For Sanity

Once we have created a package, we may desire to perform some sort of checks for quality on the package itself and not necessarily just the software we're delivering with the RPM.

For this the main tool of choice for RPM Packagers is rpmlint which performs many sanity and error checks that help assist with packaging in more maintainable and less error prone fashion. Something to keep in mind is that this is going to report things based on very strict guidelines and by way of static analysis. There is going to be lack of perspective by the rpmlint tool and what your primary objective is and thus it is sometimes alright to allow Errors or Warnings reported by rpmlint to persist in your packages, but the key is to understand **why** we would allow these to persist. In the following sections we will explore a couple examples of just that. Another really useful feature of rpmlint is that we can use it to check against Binary RPMs, Source RPMs, and SPEC files so that it can be used during all stages of packaging and not just after the fact. We will show examples of each below.

Note: For each example below we run rpmlint without any options, if you would like detailed explanations of what each Error or Warning means, then you can pass the -i option and run each command as rpmlint -i instead of just rpmlint. The shorter output is selected for brevity of the document.

bello

Let's get started by looking at some output and dive into each set of output.

```
$ 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.
```

When checking *bello*'s spec file we can see that we only have one warning and that is the URL listed in the `Source0` directive can not be reached which is something that we would expect given that example.com doesn't actually exist out in the real world and we've not setup a system with a local DNS entry to point to this URL. Since we know why the Warning was emitted and that it was expected, this can be safely ignored.

```
$ 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.
```

When checking *bello*'s SRPM we can see very similar output from the check against the spec file but we also see that the check against the SRPM looks for the `URL` directive as well as the `Source0` directive, neither can be reached but as we know is expected and these can also be safely ignored.

```
$ 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.
```

Now things will change a bit when looking at Binary RPMs as the rpmlint utility is going to check for other things that should be commonly found in Binary RPMs such as documentation and/or man pages as well as things like consistent use of the Filesystem Hierarchy Standard. As we can see, this is exactly what is being reported and we know that there are no man pages or other documentation because we didn't provide any. Also, once again our old friend the `HTTP Error 404: Not Found` is present but we're well aware as to why.

Other than our few items that we are carrying over because this is a simple example, our RPM is passing the rpmlint checks and all is well!

pello

Next up, let's look at some more output and dive into it one by one.

```
$ 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.
```

Now, I know you might be thinking "That's a lot of errors, this example must be really wrong" and you would be correct but it is wrong for good reason. The goal here is two fold, first to make a byte-compiled example that was not too complicated and allowed to demonstrate some

scripting in a SPEC file and second to show some examples of what we can expect rpmlint to report other than just a simple URL missing.

Looking at the output from the check on *pello*'s spec file we can see that we have a new Error entitled `hardcoded-library-path` and it was mentioned during the previous section that this was known to be incorrect but we were doing it anyways. The reality is that this is a half truth. Almost always, you should be using the `%{_libdir}` rpm macro or some other more sophisticated macro (more on this in the Appendix. The reason we do not use `%{_libdir}` in this instance is because that macro will expand to be either `/usr/lib/` or `/usr/lib64/` depending on a 32-bit or 64-bit architecture. Since we are packaging `noarch` that would have become problematic for one arch or the other in the event of a compile on one, run on the other. We also don't dive into more clever rpm macros as they are out of scope when trying to learn RPM Packaging at and introductory level, which is already a feat of it's own. For the sake of this example, we can ignore this Error but in a real packaging scenario you should either have a reasonable justification or find the appropriate rpm macro to use.

Once again, the `URL` listed in the `Source0` directive can not be reached which is something that we expect for the same reasons given in the previous example. Since we know why the Warning was emitted and that it was expected, this can be safely ignored also.

```
$ 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.
```

When checking *pello*'s SRPM we can see very similar output from the check against the spec file but we also see that the check against the SRPM looks for the `URL` directive as well as the `Source0` directive, neither can be reached but as we know this is expected and these can also be safely ignored.

Once again, the explanation for the `hardcoded-library-path` is the same as we covered previously in the `rpmlint` output for the SPEC file.

```
$ 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.
```

As with the previous example, things change a bit when looking at Binary RPMs as the rpmlint utility is now checking for other things that should be commonly found in Binary RPMs such as documentation and/or man pages as well as things like consistent use of the Filesystem Hierarchy Standard. As we can see, this is exactly what is being reported and we know that there are no man pages or other documentation because we didn't provide any. Also, once again our old friend the `HTTP Error 404: Not Found` is present but we're well aware as to why.

The two new ones are `non-executable-script` and `only-non-binary-in-usr-lib`.

First is `W: only-non-binary-in-usr-lib` which means that we've provided only non-binary artifacts in `/usr/lib/` which is normally reserved for shared object files which are binary data files and rpmlint therefore expects at least some of our files in `/usr/lib/` to be binary. This again rounds back to compliance with the Filesystem Hierarchy Standard as well as files ending up in incorrect or inconsistent locations because we are not using the appropriate rpm macros. This is of course by design *only* for the course of this example.

Next up is `E: non-executable-script /usr/lib/pello/pello.py 0644L /usr/bin/env` which is telling us that `rpmlint` has found a file with a shebang directive which would normally be an executable and have permissions more likely to be `0755` instead of `0644` (meaning it can not be executed), but since we're simply leaving it as an install artifact reference library because we used this as an example for doing byte-compilation at build time this can also be safely ignored.

Other than our items that we are carrying over for the purposes of the example, our RPM is passing the rpmlint checks and all is well!

cello

Next up, let's look at some more output and dive into each.

```
$ 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.
```

When checking *cello*'s spec file we can see that things appear much more as they did in our first example and we only have one warning. This is again that the `URL` listed in the `Source0` directive can not be reached which is something expected. Since we know why the `Warning` was emitted and that it was expected, this can be safely ignored.

```
$ 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.
```

When checking *cello*'s SRPM we can see very similar output from the check against the spec file but we also see that the check against the SRPM looks for the `URL` directive as well as the `Source0` directive, neither can be reached but as we know is expected and these can also be safely ignored.

```
$ 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.
```

As before, the output has changed when looking at Binary RPMs as the rpmlint utility is going to check for other things that should be commonly found in Binary RPMs such as documentation and/or man pages as well as things like consistent use of the Filesystem Hierarchy Standard. As we can see, this is exactly what is being reported just as in the previous examples and we know that there are no man pages or other documentation because we didn't provide any. Also, once again the `HTTP Error 404: Not Found` is present but we're well aware as to why.

Other than our few items that we are carrying over because this is a simple example, our RPM is passing the rpmlint checks and all is well!

That's it!

Our RPMs are sanitized (or we know and understand why they aren't) and it is now time to either go forth and Package RPMs or travel on into the Appendix.

Red Hat
Training and
Certification

Signing Packages

Signing a package is a way to secure the package for an end user. Secure transport can be achieved with the 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
```

Red Hat
Training and
Certification

Creating a PGP Key and Signing the Example RPMs

In this section we will go through the steps in order to create PGP keys with GNU Privacy Guard (GPG). First we need to create a PGP key, use the following steps to do so.

Create a PGP Key

```
$ gpg --gen-key
gpg (GnuPG) 2.0.22; Copyright (C) 2013 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

gpg: directory `/home/admiller/.gnupg' created
gpg: new configuration file `/home/admiller/.gnupg/gpg.conf' created
gpg: WARNING: options in `/home/admiller/.gnupg/gpg.conf' are not yet
active during this run
gpg: keyring `/home/admiller/.gnupg/secring.gpg' created
gpg: keyring `/home/admiller/.gnupg/pubring.gpg' created
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.
```

```
Real name: Testing User
Email address: testing@example.com
Comment: Testing RPM Signing Cert
You selected this USER-ID:
    "Testing User (Testing RPM Signing Cert) <testing@example.com>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.

We need to generate a lot of random bytes. It is a good idea to
perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to
perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /home/admiller/.gnupg/trustdb.gpg: trustdb created
gpg: key AABEF03C marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   2048R/AABEF03C 2019-07-11
      Key fingerprint = 7C0B 4EB4 3741 948C 3F80  8104 11C6 02C2 AABE
F03C
uid                  Testing User (Testing RPM Signing Cert)
<testing@example.com>
sub   2048R/13DB9F26 2019-07-11
```

Verify the GPG Key

```
$ gpg --list-keys
/home/admiller/.gnupg/pubring.gpg
---------------------------------
pub   2048R/AABEF03C 2019-07-11
uid                  Testing User (Testing RPM Signing Cert)
<testing@example.com>
sub   2048R/13DB9F26 2019-07-11
```

Export the public key from keyring

```
$ gpg --export -a 'Testing User' > RPM-GPG-KEY-testing
$ ls -l RPM-GPG-KEY-testing
-rw-rw-r--. 1 admiller admiller 1760 Jul 11 18:50 RPM-GPG-KEY-testing
```

Import public key into rpmdb (AS ROOT)

rpm --import RPM-GPG-KEY-faleman

Verify the gpg pubkeys

rpm -q gpg-pubkey --qf '%{name}-%{version}-%{release}: %{summary}\n'

Configure ~/.rpmmacros file for signing

The following RPM Macros are required for signing.

Macro	**Definition**
`%_signature`	Signature type (it's always `gpg`)
`%_gpg_path`	Full path to gnupg directory
`%_gpg_name`	Name to use when signing, typically an organization/department.
`%_gpgbin`	Path to gpg executable

```
$ cat >> ~/.rpmmacros <<EOF
%_signature gpg
%_gpg_path /home/student/.gnupg
%_gpg_name Testing User
%_gpgbin /usr/bin/gpg
EOF
```

Sign the rpms

You can sign RPMs files individually:

```
$ rpm --addsign ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el7.x86_64.rpm
Enter pass phrase:
Pass phrase is good.
cello-1.0-1.el7.x86_64.rpm:
```

Optionally you can sign multiple RPMs files at the same time with a shell glob:

```
$ rpm --addsign ~/rpmbuild/RPMS/*/*.rpm
```

Verify Signed RPMs

```
$ rpm --checksig ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el7.x86_64.rpm
cello-1.0-1.el7.x86_64.rpm: rsa sha1 (md5) pgp md5 OK
```

Notes

As mentioned in the previous section, you can sign packages at build time but this is often not the case in practice. It is common practice that packages be developed, built, iterated on, tested, and signed once verified and ready for distribution.

Appendix

Here you will find supplementary information that is very good to know and will likely prove to be helpful for anyone who is going to be building RPMs in any serious capacity but isn't necessarily a hard requirement to learn how to package RPMs,which is what the main goal of this document is.

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 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 has an effect on 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 the come with mock are targeted at Fedoran RPM Packagers and as such RHEL 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 it's 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 unneeded or unnecessary packages.

Note: Some of the output below has been omitted for brevity and has been marked by an ellipsis (`...`).

```
$ 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)
...
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 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 desirable 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 an 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 a 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.

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 bookkeeping, 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. upstream 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 rpm.org’s rpm macro official documentation.
There are also macros that are provided by your Linux Distribution, we will cover some of those provided by Fedora 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

You can define your own Macros, below is an excerpt from the RPM Official Documentation and I recommend anyone interested in an exhaustive explanation of the many possibilities of defining their own macros to visit that resource. It’s really quite good and there’s little reason to duplicate the bulk of that content here.

To define a macro use:

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

All whitespace surrounding `\<body\>` is removed. Name may be composed of alphanumeric characters, and the character _ and must be at least 3 characters in length. 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 (i.e. string between parentheses) is passed exactly as is to getopt(3) for argc/argv processing at the beginning of a macro invocation.

%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`
%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, however note that this will contain a *lot* of output so it’s often used in combination with a pipe to grep (or a clever shell Process Substitution).

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 command `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 Programming Language Specific Guidelines, which at the time of this writing has over 30 different sets of guidelines along with associated RPM Macro sets for subject matter specific RPM Packaging.

One example of these kinds 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), 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
```

Java Specific Macros

As mentioned previously, but with recap again in this section. 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 Programming Language Specific Guidelines, which at the time of this writing has over 30 different sets of guidelines along with associated RPM Macro sets for subject matter specific RPM Packaging.

One example of these kinds 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), we have a number of `python2` specific macros available to us.

```
$ rpm -ql javapackages-tools | grep macros
/etc/rpm/macros.fjava
/etc/rpm/macros.jpackage

$ rpm --showrc | grep java
  %{?_javaclasspath:CLASSPATH="%{_javaclasspath}"
-14: __docdir_path
%{_datadir}/doc:%{_datadir}/man:%{_datadir}/info:%{_datadir}/gtk-doc/html:%{?_d
ocdir}:%{?_mandir}:%{?_infodir}:%{?_javadocdir}:/usr/doc:/usr/man:/usr/info:/us
r/X11R6/man
-14: __javadoc_path     ^%{_javadocdir}/.*
-14: __javadoc_requires %{_rpmconfigdir}/javadoc.req
    %{!?__jar_repack:/usr/lib/rpm/redhat/brp-java-repack-jars}
-14: __pom_call . /usr/share/java-utils/pom_editor.sh; pom_
-14: _javaconfdir       %{_sysconfdir}/java
```

```
-14: _javadir   %{_datadir}/java
-14: _javadocdir        %{_datadir}/javadoc
-14: _jnidir    %{_prefix}/lib/java
-14: add_jvm_extension  JAVA_LIBDIR=%{buildroot}%{_javadir} %{_bindir}/jvmjar
-l
for _dir in %{_jnidir} %{_javajnidir} %{_javadir}; do
python -m /usr/share/java-utils/maven_depmap %{-a} %{-v*:-r %{-v*}} \
-14: ant        JAVA_HOME=%{java_home} ant
-14: jar        %{java_home}/bin/jar
-14: java       %(. %{_javadir}-utils/java-functions; set_javacmd; echo
$JAVACMD)
-14: java_home  %(. %{_javadir}-utils/java-functions; set_jvm; echo $JAVA_HOME)
-14: javac      %{java_home}/bin/javac
-14: javadoc    %{java_home}/bin/javadoc
. %{_javadir}-utils/java-functions
if [ -f %{_sysconfdir}/java/%{name}.conf ] ; then
  . %{_sysconfdir}/java/%{name}.conf
```

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 %java
/opt/ibm/java-x86_64-80/bin/java

$ rpm --eval %javac
/opt/ibm/java-x86_64-80/bin/javac
```

As you can see above, the example system this command on is running a non-standard Java SDK for a Red Hat Enterprise Linux install. This was done in order to show the advantages of using the macros instead of trying to hardcode to specific binaries or paths because as a packager this can impose added work as you build your SPRM on different versions of Java or on different releases of an operating system. By using the macros we are able to use or rebuild the same SPEC or SRPM to target different versions without any extra maintenance burden on use as the packager.

Advanced SPEC File Topics

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 it's 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.

Triggers and Scriptlets

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.

Is 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-4.*/triggers`

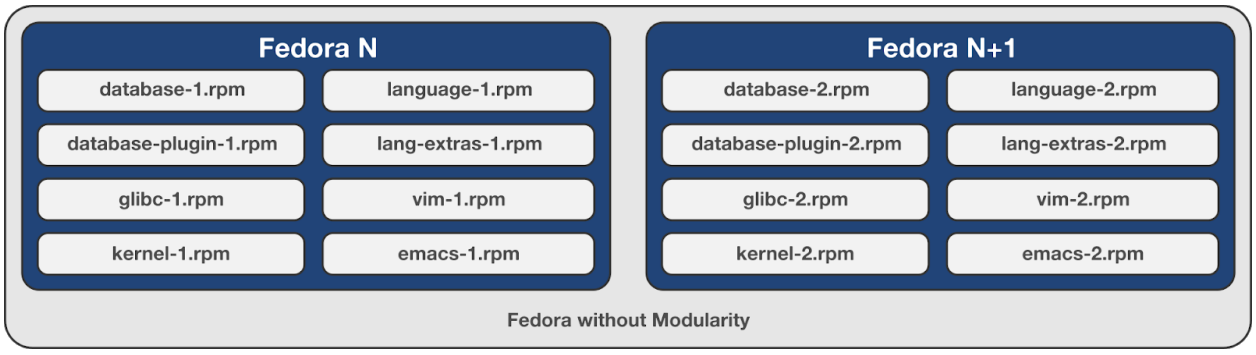
AppStreams and Modularity: The Future of Packaging

A common challenge of Enterprises is the desire to maintain stability at the operating system platform level, but also cater to differing lifecycle cadences of software and their dependencies. There have been many different attempts at solving this problem but from the lessons learned over time from various solutions in that past, the new concept of AppStreams and Modularity was born. Effectively this new technology allows many different packages (or sets of packages so the dependency chain can operate with the desired software as a single unit) can provide the same thing but be different versions and lifecycle managed independently. Below is an excerpt from the Fedora Modularity Documentation:

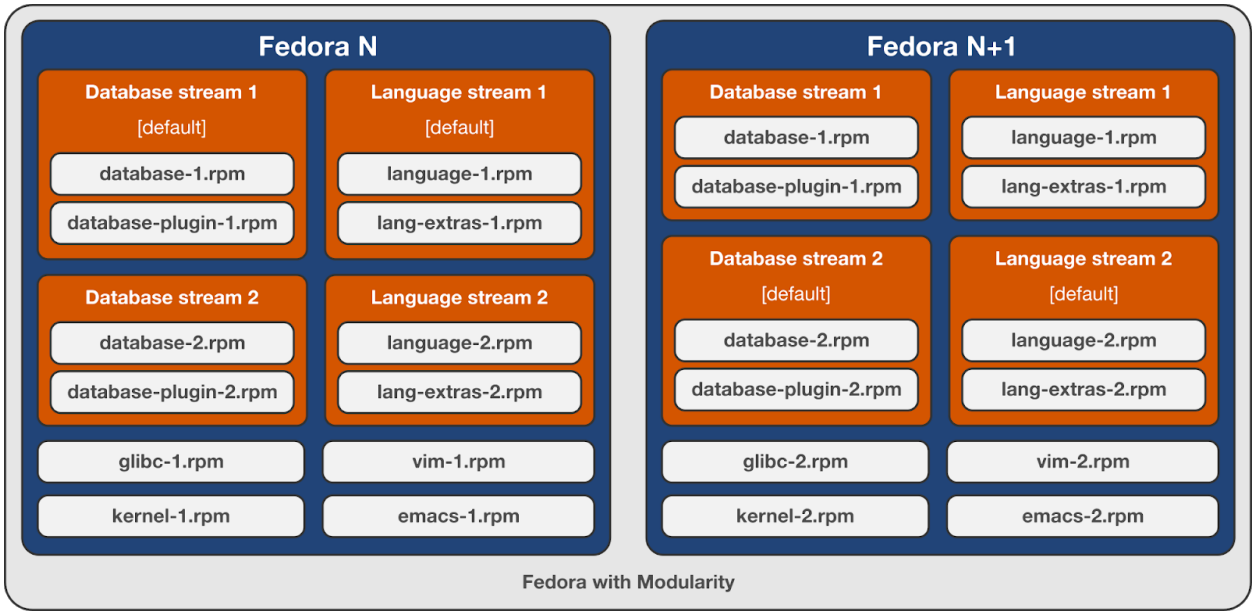
Modularity

Modularity enables you to choose a particular stream (major version) of content that has been natively built and tested for your system, and to receive the right updates for it.

Without Modularity



With modularity



That means you're no longer limited to a single version of each package for a given Fedora release. And because many streams are now available in multiple Fedora releases, you can install a specific version of software regardless of what Fedora release you're running.

Examples

Scenario 1: Some users install packages coming from a different Fedora release in order to consume a specific version of a database that is compatible with their application. But thanks to Modularity they might not need to do that anymore, because multiple versions of the database can available in each Fedora release. All they need to do is to consume the specific stream of that database right from the Fedora repositories for their system.

Scenaro 2: There were cases when users couldn't upgrade their system to a new Fedora release because their application wouldn't function with the new version of a language runtime coming with the upgrade. Modularity can fix this problem by providing the same language versions in both Fedora releases. With that, the user can consume a specific stream of the language and keep it even when they upgrade their system. And when the application is ready for the new language version, it can be upgraded later, independently from the OS, by switching to a different stream.

Compatibility

Modularity is built to be 100% compatible with existing expectations and workflows. The installation and update experience continues to work the same way — even when there are multiple versions of packages — thanks to default streams.

For example, the following two commands work the same way on systems with and without Modularity:

```
$ dnf install httpd
$ dnf update
```

On systems with multiple httpd streams available, the default stream is automatically enabled and consumed.

Building AppStream Modules

At the time of this writing, the only publicly available way to build AppStream Modules is through the upstream Fedora Project tooling which should not be considered production ready.

References

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

- Red Hat Enterprise Linux 7 RPM Packaging Guide
- Red Hat Enterprise Linux 8 Packaging and Distributing Software
- RPM Official Documentation
- Gurulabs CREATING RPMS (Student Version)
- Fedora How To Create An RPM Package Guide
- Fedora Packaging Guidelines
- OpenSUSE Packaging Guidelines
- IBM RPM Packaging Guide: Part 1, Part 2, Part 3
- Maximum RPM (Some material is dated, but this is still a great resource for advanced topics.)
- Fedora Modularity and AppStreams
- Fedora Java Packaging Tutorial

Red Hat
Training and
Certification

This Document

This Document was originally created for the Red Hat Summit 2016, but has been updated where applicable for both Red Hat Summit 2017 and 2018. There is an upstream document originally written by your presenter, Adam Miller, but is maintained on GitHub. Always feel free to check that document for newer versions and/or provide feedback about improvements you would like to see in the future.

- GitHub Project: https://github.com/redhat-developer/rpm-packaging-guide
- Read The Docs Pre-Rendered Guide: https://rpm-packaging-guide.github.io/