Fedora Draft Documentation

Packager's Guide

A guide to software packaging for Fedora 18

Petr Kovář

Fedora Draft Documentation Packager's Guide
A guide to software packaging for Fedora 18
Edition 18.0.1

Author Petr Kovář *pkovar@redhat.com*

The Packager's Guide provides basic information on creating, building, and testing RPM packages, and spec file writing. It also contains a spec file reference. The audience are developers and system administrators who have a basic understanding of software packaging and RPM.


DRAFT

Preface

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F2`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F1`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

[1] https://fedorahosted.org/liberation-fonts/

> **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **ssh *username*@*domain.name*** at a shell prompt. If the remote machine is **example.com** and your username on that machine is john, type **ssh john@example.com**.
>
> The **mount -o remount *file-system*** command remounts the named file system. For example, to remount the **/home** file system, the command is **mount -o remount /home**.
>
> To see the version of a currently installed package, use the **rpm -q *package*** command. It will return a result as follows: ***package-version-release***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **mono-spaced roman** and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in **mono-spaced roman** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;
```

```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in Bugzilla: *http://bugzilla.redhat.com/bugzilla/* against the product **Fedora Documentation.**

When submitting a bug report, be sure to mention the manual's identifier: *packager-guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

3. Acknowledgments

Certain portions of this text first appeared in the *Fedora Packaging Documentation*, copyright © 2012 Red Hat, Inc. and others, published by the Fedora Project at *http://fedoraproject.org/wiki/Category:Package_Maintainers*.

Certain portions of this text first appeared in the *Fedora Packaging Guidelines*, copyright © 2012 Red Hat, Inc. and others, published by the Fedora Project at *https://fedoraproject.org/wiki/Special:PrefixIndex/Packaging*.

Certain portions of this text first appeared in the *Fedora RPM Guide*, copyright © 2010 Red Hat, Inc. and others, published by the Fedora Project at *http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/index.html*.

The author of this book would like to thank the following people for their valuable contributions: Jindřich Nový, Vít Ondruch, Chris Negus, Richard Fontana, Abdel Gadiel Martínez Lassonde, Jaromír Hradílek, Douglas Silas, and Florian Nadge, among many others.

Exploring the Structure of Packages

This chapter describes the package design, defines the basic content of every package, and shows how to explore the package structure.

1.1. Packaging with RPM

The Red Hat Package Manager (RPM) makes it easier for you to distribute, manage, and update software that you create for Red Hat Enterprise Linux. This chapter provides basic information on how to package your software into an RPM.

1.1.1. Why Package Software with RPM?

Many software vendors distribute their software via a conventional archive file (such as a tarball). However, there are several advantages in packaging software into an RPM. These advantages include:

- Each RPM package includes metadata that describes the package's components, version, release, size, project URL, installation instructions, and so on.
- Users can use a set of standard package management tools (for example **yum** or **PackageKit**) to install, remove, and manage your software.
- With the standard package management tools, you can directly distribute your software, including any software updates, to your users.

1.2. Package Design

As a part of their design, RPM packages consist of the following three parts:

Metadata
: The metadata part defines the package itself, including its name, version, license, a list of changes, dependencies, and so on.

Files
: This part consists of an archive with files that are installed by the package on the system.

Scripts
: This part contains scripts, which are run when the package is installed, updated, or uninstalled.

Each of these three parts must be defined, provided, or included by the package in order to successfully build, install, or uninstall the package.

1.2.1. Package File Name

Every package file is labeled with a highly identifiable name. This name has four parts, which typically look something like:

- *kernel-smp-2.6.32.9-3.i686.rpm*
- *kernel-smp-2.6.32.9-3.x86_64.rpm*
- *rootfiles-7.2-1.noarch.rpm*

Here, the four parts of each name are separated from each other by dashes or periods. The structure of the package file name is as follows:

- *name-version-release.architecture.rpm*

1.2.1.1. Package Architecture

RPM supports various architectures. The following table presents some of the architectures available for different platforms that are supported by Fedora 18.

Table 1.1. Package Architecture

Platform	Architecture
Intel x86 or compatible	i386, i686
AMD64 / Intel 64	x86_64
Intel Itanium	ia64
IBM POWER	ppc64
IBM System z	s390x, s390
No architecture	noarch

A platform-independent package, which is identified with **noarch** in the architecture part of the file name, provides programs that are not dependent on any platform. Programs written in Perl, Python, or other scripting languages often do not depend on code compiled for a particular architecture. In addition, compiled Java applications are usually free of platform dependencies, thus are also distributed in the form of platform-independent packages.

1.2.1.2. Source RPM

A source RPM package (SRPM) contain all the commands, usually in scripts, necessary to recreate the binary RPM. Having a SRPM means that you can recreate the binary RPM at any time. A SRPM has a file name ending in **.src.rpm**, for example:

- *mlocate-0.22.2-2.src.rpm*

The SRPM distributes a source code for the corresponding software that is ready to be installed on the system with the binary RPM. It also includes a spec file, which describes the software and the package, and contains instructions on how to perform the installation on the system.

1.2.2. Format of the Archived Files

The archive with files, which are installed by the package, is stored in the **cpio** format inside the package and is compressed with the **xz** program.

To decompress the included archive and extract the archived files, run the command in the following format:

```
rpm2cpio package | cpio -id
```

For example, to decompress the archive included in the *eject-2.1.5-0.1.fc18.x86_64.rpm* package, run the following command:

```
rpm2cpio eject-2.1.5-0.1.fc18.x86_64.rpm | cpio -id
```

1.2.3. Querying Packages

The **rpm** allows you to query packages that are available on the system. This way, you can easily explore the basic structure of the packages.

Some of the useful commands for querying the packages include:

`rpm -qd` *`package`*
: The **`rpm -qd` *`package`*** command is used to get a list of included documentation files, which are defined by the **`%doc`** directive.

`rpm -qc` *`package`*
: The **`rpm -qc` *`package`*** command is used to get a list of included configuration files, which are defined by the **`%config`** directive.

`rpm -q --scripts` *`package`*
: The **`rpm -q --scripts` *`package`*** command is used to get a list of scripts, which are defined by the **`%pre`**, **`%post`**, **`%preun`**, and **`%postun`** directives.


DRAFT

Creating and Building Packages

This chapter shows you how to convert a source archive into an RPM package.

2.1. Preparing a Build Environment

This section shows how to prepare an environment for building RPM packages on your system.

2.1.1. Creating a Non-Root Buildroot

These steps show how to create a non-root buildroot environment. This non-root environment is used to build packages as a normal user, without the need of becoming the root user. Because some of the software source archives can contain code in a makefile or script that can possibly damage your system, it is highly recommended to build packages as a user that does not have full access to the system.

Procedure 2.1. Creating a non-root buildroot

1. As root, install the *rpmdevtools* package with the following command:

```
yum install -y rpmdevtools
```

2. As a normal user, run the following command to create the **~/rpmbuild/** directory where packages are built:

```
rpmdev-setuptree
```

2.2. Creating a Basic Spec File

This section shows how to write a basic spec file for your RPM package.

2.2.1. Creating an Example Spec File for eject

These steps show an example of creating a package from the source code archive of the **eject** utility. The **eject** utility is a simple software that ejects removable media using software control so it is an ideal candidate for creating a basic spec file.

Procedure 2.2. Creating an example package: eject

1. In a shell prompt, go into the buildroot and create a new spec file for your package.

   a. To create a spec file template with the **rpmdev-newspec** command, run the following commands:

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

```
rpmdev-newspec eject
```

   This creates a new spec file called **eject.spec** in the **~/rpmbuild/SPECS** directory.

   b. To specify a spec file template for a particular type of packages, refer to the contents of the **/etc/rpmdevtools/** directory, which includes spec file templates called **spectemplate-*type*.spec**. For example, to create a new spec file for a Python module, run the following commands:

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

```
rpmdev-newspec python-antigravity
```

c. For more information on creating a new spec file with **rpmdev-newspec**, run the **rpmdev-newspec --help** command.

d. Alternatively, you can use the **vim** editor to create a spec file template for you. Change to the buildroot and run **vim** with the name of the spec file you want to create:

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

```
vi eject.spec
```

2. Open the spec file in a text editor. The spec file should be similar to the following example:

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

Group:
License:
URL:
Source0:
BuildRoot:      %{_tmppath}/%{name}-%{version}-%{release}-root-%(%{__id_u} -n)

BuildRequires:
Requires:

%description


%prep
%setup -q


%build
%configure
make %{?_smp_mflags}


%install
rm -rf $RPM_BUILD_ROOT
make install DESTDIR=$RPM_BUILD_ROOT


%clean
rm -rf $RPM_BUILD_ROOT


%files
%defattr(-,root,root,-)
%doc



%changelog
```

The Group and BuildRoot tags are deprecated

Although the **Group** and **BuildRoot** tags are included in the spec file templates, **RPM** in Fedora 18 does not require the presence of these tags in the spec file and ignores them.

The %clean and %defattr directives are deprecated

Although the **%clean** and **%defattr** directives are included in the spec file templates, **RPM** in Fedora 18 does not require the presence of these directives in the spec file and ignores them.

3. Edit the **Release** tag to set the release value of the package. For example, set the value to **1%{?dist}** if you are creating the initial release of the package:

```
Release:        1%{?dist}
```

The **{?dist}** tag is used to mark the distribution revision of a package.

4. Fill in the version and add a summary of the software:

```
Version:        2.1.5
Release:        1%{?dist}
Summary:        A program that ejects removable media using software control
```

5. For the **License** tag, fill in the appropriate license for the software. In this case, **eject** uses the GNU General Public License v2.0 or later. The short name for this license is GPLv2+:

```
License: GPLv2+
```

6. From the **eject** project website, get the URL of the website and fill it in the **URL** tag:

```
URL:            http://www.pobox.com/~tranter
```

7. In the **Source** tag, fill in the URL of the source archive for the package:

```
Source0: http://www.ibiblio.org/pub/Linux/utils/disk-
management/%{name}-%{version}.tar.gz
```

8. Edit the **BuildRequires** tag with requirements that are needed to build the package. **BuildRequires** can contain either a list of required packages or files. For example, the *eject* package requires the *gettext* and *libtool* packages:

```
BuildRequires: gettext
BuildRequires: libtool
```

9. It is recommended to add a list of requirements that this package depends on to the **`Requires`** tag. **`Requires`** can contain either a list of required packages or files.

   In many cases, however, **rpmbuild** is able to detect the dependencies automatically for you so that you do not need to add them manually. For example, the **eject** package does not need any explicit requirements in the **`Requires`** tag, so do not include the tag in the spec file.

10. Add a description for the package. The lines of text in **`%description`** should be at most 79 characters long:

```
%description
The eject program allows the user to eject removable media (typically
CD-ROMs, floppy disks or Iomega Jaz or Zip disks) using software
control. Eject can also control some multi-disk CD changers and even
some devices' auto-eject features.

Install eject if you'd like to eject removable media using software
control.
```

11. Add the **`%check`** section between the sections **`%build`** and **`%install`** in the spec file. The **`%check`** section typically contains the **`make test`** or **`make check`** command that runs any self-tests distributed with the software:

```
%check
make check
```

12. Edit the **`%install`** section by adding the following installation instructions that are specific to *eject* the to the spec file:

```
install -m 755 -d $RPM_BUILD_ROOT/%{_sbindir}
ln -s ../bin/eject $RPM_BUILD_ROOT/%{_sbindir}
```

    This calls the **install** program and creates a symbolic link, which is necessary to properly build and install the *eject* package.

13. Because the **eject** utility includes translation files, you need to define a macro called **`%find_lang`**, which will locate all of the translation files that belong to the package, and put this list in a file called **`%{name}.lang`**.

    a. To define the **`%find_lang`** macro, add the following to the **`%install`** section:

```
%find_lang %{name}
```

    b. To include the **`%{name}.lang`** file with a list of translation files, add the file name with the **`-f`** option to **`%files`**:

```
%files -f %{name}.lang
```

14. Add the list of documentation files that are included in **eject** to the **`%doc`** directive:

```
%doc README TODO COPYING ChangeLog
```

15. Edit **%changelog** to describe the last change you have made to the package. Fill it in with the date, your name and email address, the version and release of the package, and a short description of what has changed in the package in the following format:

```
* date Packager's Name <packager's_email> version-revision
- Summary of changes
```

To get the changelog entry in the required format, you can use the **rpmdev-bumpspec** utility. Run the following command:

```
rpmdev-bumpspec --comment=summary of changes --userstring=Packager's
Name <packager's_email> spec file
```

Because this is the first release of the package, run the **rpmdev-bumpspec** command with the following options:

```
rpmdev-bumpspec --comment="Initial RPM release" --userstring="John Doe
<jdoe@example.com>" eject.spec
```

This will produce a changelog entry in the spec file similar to the following:

```
%changelog
* Wed Oct 20 2011 John Doe <jdoe@example.com> 2.1.5-0.1
- Initial RPM release
```

16. Now the spec file should look like in the following example:

```
Name: eject
Version: 2.1.5
Release: 1%{?dist}
Summary: A program that ejects removable media using software control

License: GPLv2+
URL: http://www.pobox.com/~tranter
Source0: http://www.ibiblio.org/pub/Linux/utils/disk-
management/%{name}-%{version}.tar.gz

BuildRequires: gettext
BuildRequires: libtool

%description
The eject program allows the user to eject removable media (typically
CD-ROMs, floppy disks or Iomega Jaz or Zip disks) using software
control. Eject can also control some multi-disk CD changers and even
some devices' auto-eject features.

Install eject if you'd like to eject removable media using software
control.

%prep
%setup -q -n

%build
%configure
make %{?_smp_mflags}

%check
make check

%install
```

```
rm -rf $RPM_BUILD_ROOT
make install DESTDIR=$RPM_BUILD_ROOT

install -m 755 -d $RPM_BUILD_ROOT/%{_sbindir}
ln -s ../bin/eject $RPM_BUILD_ROOT/%{_sbindir}

%find_lang %{name}

%files -f %{name}.lang
%doc README TODO COPYING ChangeLog
%{_bindir}/*
%{_sbindir}/*
%{_mandir}/man1/*

%changelog
* Wed Oct 20 2011 John Doe <jdoe@example.com> 0.8.18.1-0.1
- Initial RPM release
```

2.3. Building a Package

This section shows how to build your RPM package with the **rpmbuild** command.

Run rpmbuild as a non-root user

It is highly recommended to always run the **rpmbuild** command as a non-root, normal, user. If you build a package as the root user, possible mistakes in the spec file, for example in the **%install** section, can cause damage to your system.

2.3.1. Building an Example Package: eject

These steps show an example of building a package with a previously created spec file of the **eject** utility. For installation, **eject** only requires a minimum set of package dependencies. This makes it a good project to examine for package building.

Before starting with the package building process itself, make sure that you have created a non-root buildroot on your system.

Procedure 2.3. Building an example package: eject

1. Download the source code archive for **eject** and place it in the **~/rpmbuild/SOURCES/** directory.
2. Obtain the spec file for **eject** and place it in the **~/rpmbuild/SPECS/** directory.
3. In a shell prompt, change to the **~/rpmbuild/SPECS/** directory and run the **rpmbuild** command:

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

```
rpmbuild -ba eject.spec
```

4. If you end up with an error message similar to the following one, it means that you have not installed the package dependencies required by the *eject* package:

```
error: Failed build dependencies:
```

```
libtool is needed by eject-2.1.5-0.1.x86_64
```

- You can use *yum* to install the needed files or packages. Run the following command to install the required *libtool* package:

```
yum install -y libtool
```

5. In case you have received a build error related to the **%install** section, you may want to skip earlier stages of the build process with the **--short-circuit** option and restart the build process at the **%install** stage:

```
rpmbuild -bi --short-circuit eject.spec
```

6. Once you get a clean build, the last line of the **rpmbuild** output will be as follows:

```
+ exit 0
```

7. After a successful build with the **rpmbuild -ba eject.spec** command, the binary package will be placed in a subdirectory of the **~/rpmbuild/RPMS/** directory and the source package will be placed in **~/rpmbuild/SRPMS/**.

8. If you just want to create a source package (**.src.rpm**), run the following command:

```
rpmbuild -bs eject.spec
```

This will create the source package in the **~/rpmbuild/SRPMS/** directory, or recreate it if it has been previously created.

2.4. Testing a Package

This section shows how to use the **rpmlint** utility to test an RPM package. **rpmlint** can be used to test spec files, binary packages, and source packages. It is highly recommended to run **rpmlint** every time you make changes to the package.

2.4.1. Testing a Spec File

These steps show how to test a spec file of a package with the **rpmlint** utility.

Procedure 2.4. Testing a spec file with rpmlint

1. If you have not previously installed the *rpmlint* package, install it now with the following command:

```
yum install -y rpmlint
```

2. To test a spec file with **rpmlint**, run the following command:

```
rpmlint package.spec
```

By checking the spec file for correctness with the above command, **rpmlint** is able to catch many errors that can be often found in new or significantly changed spec files.

3. If the error messages reported with the above command are not clear enough, use the **-i** option, which provides more information on each error:

```
rpmlint -i package.spec
```

Refer to the *Fedora Packaging Guidelines*[1] for information on what **rpmlint** errors can be typically ignored.

2.4.2. Testing a Spec File, Binary, and Source Package

These steps show how to test a spec file, a binary package, and a source package with the **rpmlint** utility.

Procedure 2.5. Testing a spec file, binary, and source package with rpmlint

1. Change to the **~/rpmbuild/SPECS/** directory in the buildroot environment and run the following command:

   ```
   rpmlint package.spec ../RPMS/*/package*.rpm ../SRPMS/package*.rpm
   ```

   After running the command, **rpmlint** will build binary packages with debugging information.

2. Change to the **~/rpmbuild/RPMS/*architecture*/** directory to locate the binary packages that were built with **rpmlint**.

3. To check the files and their permissions included in the binary packages, use the **rpmls** command:

   ```
   rpmls *.rpm
   ```

4. If the included files and their permissions are correct, proceed with running the following command as root to install the included files:

   ```
   rpm -ivp package.rpm
   ```

   After a successful installation, you can test the installed files on your system.

5. After you have finished testing of the installed files, run the following command to uninstall the previously installed packages:

   ```
   rpm -e package.rpm
   ```

2.4.3. Testing a Package with Mock

These steps show how to test a package with the help of **Mock**. **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. Use **Mock** to test that you have accurate definitions of the build dependencies in your spec file.

Procedure 2.6. Testing a package with Mock

1. As root, run the following command to add your normal user that you intend to use with **Mock** to the `mock` group:

[1] http://fedoraproject.org/wiki/Packaging/Guidelines#Use_rpmlint

```
usermod -a G mock user_name && newgrp mock
```

2. Create a source package with the following command:

```
rpmbuild -bs package.spec
```

where *package.spec* is the name of the spec file for your package.

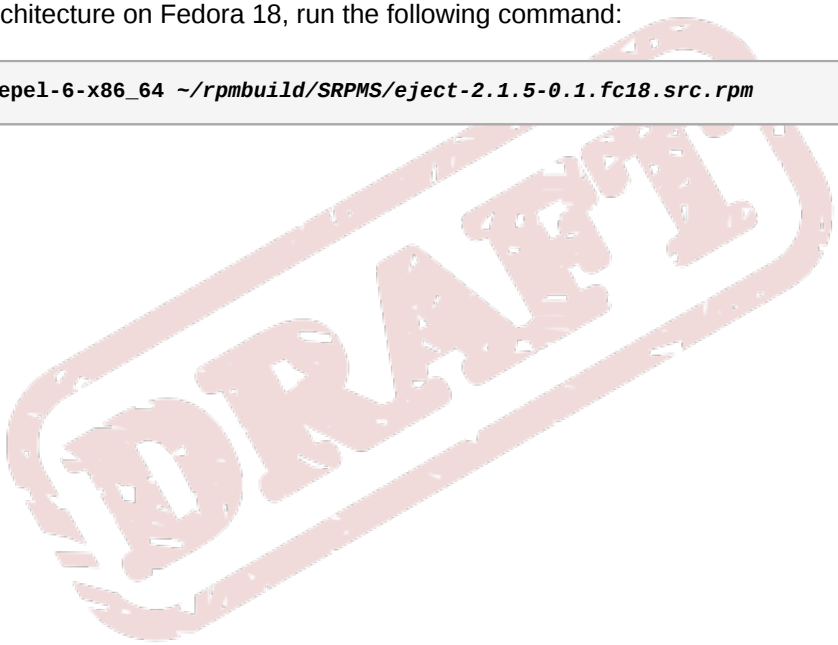
3. To test a package locally with **Mock**, run the following command:

```
mock -r config_name rebuild path_to_source_package
```

where *config_name* is the name of the configuration name. The configuration file name contains the name of the used system architecture and operating system (for example Fedora 18). Refer to the **/etc/mock/** directory for a list of available configuration files.

For example, to test the *~/rpmbuild/SRPMS/eject-2.1.5-0.1.fc18.src.rpm* package for the AMD64 (x86_64) architecture on Fedora 18, run the following command:

```
mock -r epel-6-x86_64 ~/rpmbuild/SRPMS/eject-2.1.5-0.1.fc18.src.rpm
```




DRAFT

Appendix A. Spec File Reference

Spec files are text files that contain RPM directives and macro definitions, which are used to build an RPM package. The RPM directives and macros are divided into a number of sections. Each of these sections is delimited with a **%** marker. For example, the build section starts with **%build**.

A typical spec file consists of approximately five sections:

Preamble
: The preamble describes the basic information on the package, for example name, version, license, and so on.

Build section
: The build section includes instructions, which are used to build and prepare the package for installation.

Scriptlets
: The scriptlets define commands used to install, upgrade, or uninstall the package.

Manifest
: The manifest section includes a list of packaged files and their permissions.

Changelog
: The changelog section consists of a list of changes made to the package.

A.1. Spec File Directives

Directives in a spec file are defined using a simple syntax of a tag name, a colon, and a value:

```
TagName: value
```

For example, the following directive sets the package version to 1.15:

```
Version: 1.15
```

The name of the item is not case sensitive, so tag names of **version**, **Version**, or **VERSION** all set the same value. This syntax works for most settings, including **Name**, **Release**, and so on.

A.2. Spec File Macros

In addition to the spec file directive syntax, you can define macros using the RPM **%global** syntax. For example:

```
%global major 2
```

The example above defines a macro named **major** with a value of **2**.

Once defined, you can access macros using the syntax **%{macro_name}** or just **%macro_name**. For example:

```
source: %{name}-%{version}.tar.gz
```

A.3. Spec File Comments

To include comments in the spec file, use a # character at the start of the line. That way, the line will be ignored by RPM.

Because macros are expanded first, do not insert any multiline macros in a comment. If you want to comment out a line with a macro, double the percent signs (%%) as in the following example:

```
# %%configure
```

Also, do not use inline comments ("#") on the same line after a script command.

A.4. Spec File Example

Below is an example of a spec file from the *eject* package in Fedora 18.

```
Summary: A program that ejects removable media using software control
Name: eject
Version: 2.1.5
Release: 11%{dist}
License: GPL
Source: http://metalab.unc.edu/pub/Linux/utils/disk-management/%{name}-%{version}.tar.gz
Source1: eject.pam
Patch1: eject-2.1.1-verbose.patch
Patch2: eject-timeout.patch
Patch3: eject-2.1.5-opendevice.patch
Patch4: eject-2.1.5-spaces.patch
Patch5: eject-2.1.5-lock.patch
Patch6: eject-2.1.5-umount.patch
URL: http://www.pobox.com/~tranter
ExcludeArch: s390 s390x
BuildRequires: gettext
BuildRequires: automake
BuildRequires: autoconf
BuildRequires: libtool

%description
The eject program allows the user to eject removable media (typically
CD-ROMs, floppy disks or Iomega Jaz or Zip disks) using software
control. Eject can also control some multi-disk CD changers and even
some devices' auto-eject features.

Install eject if you'd like to eject removable media using software
control.

%prep
%setup -q -n %{name}
%patch1 -p1 -b .verbose
%patch2 -p1 -b .timeout
%patch3 -p0 -b .opendevice
%patch4 -p0 -b .spaces
%patch5 -p0 -b .lock
%patch6 -p1 -b .umount

%build
%configure
make

%install
rm -rf %{buildroot}

make DESTDIR=%{buildroot} install
```

```
# pam stuff
install -m 755 -d %{buildroot}/%{_sysconfdir}/pam.d
install -m 644 %{SOURCE1} %{buildroot}/%{_sysconfdir}/pam.d/%{name}
install -m 755 -d %{buildroot}/%{_sysconfdir}/security/console.apps/
echo "FALLBACK=true" > %{buildroot}/%{_sysconfdir}/security/console.apps/%{name}

install -m 755 -d %{buildroot}/%{_sbindir}
pushd %{buildroot}/%{_bindir}
mv eject ../sbin
ln -s consolehelper eject
popd

%find_lang %{name}

%files -f %{name}.lang
%doc README TODO COPYING ChangeLog
%attr(644,root,root) %{_sysconfdir}/security/console.apps/*
%attr(644,root,root) %{_sysconfdir}/pam.d/*
%{_bindir}/*
%{_sbindir}/*
%{_mandir}/man1/*

%changelog
* Wed Apr 02 2008 Zdenek Prikryl <zprikryl at, redhat.com> 2.1.5-11
- Added check if device is hotpluggable
- Resolves #438610
```

The example above shows the usage of some of the directives and macros in a spec file. Refer to *Section A.5, "Spec File Preamble"* for a detailed description of the directives and macros.

A.5. Spec File Preamble

This section include a list with descriptions of some of the more frequently used directives in spec files.

`Name:` *`name`*

The **`Name:`** tag defines the (base) name of the package. This name must follow the *Fedora Package Naming Guidelines*[1]. Also, this name should match the spec file name. In many cases, the name will be in all lower case. For example:

```
Name: eject
```

Elsewhere in the spec file, you can refer to the name using the macro **`%{name}`**. That way, if the name changes, the new name will be used by those other locations.

`Version:` *`version`*

The **`Version`** tag defines the upstream version number. If the version is non-numeric (that is it contains tags that are not numbers or digits), you may need to include the additional non-numeric characters in the **`Release`** tag. If upstream uses full dates to distinguish versions, consider using version numbers of the form **`yy.mm[.dd]`** (so a **`2008-05-01`** release becomes **`8.05`**). See *Fedora Package Naming Guidelines*[2] for more information. An example of the **`Version`** tag:

```
Version: 2.1.5
```

Elsewhere in the spec file, refer to the tag value as %{version}. That way, if the tag value changes, the new value will be used by those other locations.

[1] http://fedoraproject.org/wiki/Packaging/NamingGuidelines
[2] http://fedoraproject.org/wiki/Packaging/NamingGuidelines#Package_Versioning

Release: *release*

The **Release** tag defines the value of the package's version. The initial release will typically be defined as **1%{?dist}**. After the initial release, increment the number every time a new package is released for the same version of software. If a new version of the packaged software is released, the **Version** tag should be changed to reflect the new software version, and the **Release** tag should be reset to **1**. See *Fedora Package Naming Guidelines*[3] for more information. Refer to the *Fedora Dist Tag Guidelines*[4] for a description of the **dist** tag, which is not required but can be useful. An example of the **Release** tag:

```
Release: 11%{dist}
```

Elsewhere in the spec file, refer to the tag value as %{release}. That way, if the tag value changes, the new value will be used by those other locations.

Summary: *summary*

The **Summary** tag defines a brief, one-line summary of the package. Do not use a period at the end of the summary. For example:

```
Summary: A program that ejects removable media using software control
```

Group: *group*

The **Group** tag defines a package group, which the package is a part of. The tag must define a previously existing group, for example **Applications/Engineering**. To display a complete list of existing groups, run the following command:

```
less /usr/share/doc/rpm-*/GROUPS
```

If you create a *package-doc* sub-package with documentation, use the group **Documentation**. An example of the **Group** tag:

```
Group: System Environment/Base
```

The Group tag is deprecated

RPM in Fedora 18 does not require the presence of the **Group** tag in the spec file. If the tag is defined, it will be ignored. The package groups of the **yum** application are used on a Fedora 18 system as the relevant source of information on which group is the package a part of.

License: *license*

The **License** tag defines the license of the packaged software. Use a standard abbreviation, for example **GPLv2+**. The definition of the license should be specific; for example do not use **GPL** or **GPLv2** when the license is in fact GPL version 2 or greater, that is **GPLv2+**. You can list multiple licenses in the tag by combining them with words **and** and **or**, for example **GPLv2 and BSD**.

[3] http://fedoraproject.org/wiki/Packaging/NamingGuidelines#Package_Versioning
[4] http://fedoraproject.org/wiki/Packaging/DistTag

Refer to the *Fedora Licensing Document*[5] and the *Licensing Guidelines*[6] for more information on this topic.

Do not use the tag **Copyright** in place of the **License** tag. An example of the **License** tag:

```
License: GPL
```

URL: *URL*

The **URL** tag defines the URL with more information about the program, for example the project website. This tag does not define where the original source code came from, use the the tag **Source** for that purpose. An example of the **URL** tag:

```
URL: http://www.pobox.com/~tranter
```

Source*0*: *URL*

The **Source** tag defines a URL for the compressed archive containing the (original) unmodified source code, as upstream released it. The tag **Source** is the same as the tag **Source0**. Because a full URL should be provided with this tag, its basename can be then used when looking in the **SOURCES/** directory. If possible, add **%{name}** and **%{version}** to the URL.

The tag **Source** and the tag **URL** are used for different purposes. Typically, they are both URLs, but the **URL** tag points to the project website, while the **Source** tag points to the actual file containing the source code.

When downloading the source code, consider using a client that preserves the upstream timestamps, such as **wget**. For example:

```
wget -N URL
```

If you are using **curl**, run the following command:

```
curl -R URL
```

If there is more than one source, name them **Source1**, **Source2**, and so on. If you are adding whole new files in addition to the unmodified sources, you can list each of them as sources as well, but list them *after* the unmodified sources. Remember to include a copy of each of these sources in any source package you create. For information on exceptions to this rule, such as when using revision control system, upstream using prohibited code, and so on, refer to the *Fedora Source URL Guidelines*[7]. An example of the **Source** tag:

```
Source1: eject.pam
```

Patch*0*: *file_name*

The **Patch0:** tag defines the name of the first patch that you will apply to the source code. If you need to patch the files after they have been uncompressed, edit the files, save their differences as a *patch* file in the **~/rpmbuild/SOURCES/** directory. Patches should make only one logical change, so it is likely to have multiple patch files. If there is more than one source, name them **Patch1**, **Patch2**, and so on. An example of the **Patch1** tag:

[5] http://fedoraproject.org/wiki/Licensing
[6] http://fedoraproject.org/wiki/Packaging/LicensingGuidelines
[7] http://fedoraproject.org/wiki/Packaging/SourceURL

```
Patch1: eject-2.1.1-verbose.patch
```

BuildArch: *architecture*

The **BuildArch** tag is used to define the build architecture of the package. If you are packaging files that are architecture-independent, for example shell scripts, data files, and so on, use **BuildArch: noarch**. In this case, the architecture for the binary RPM will be **noarch**. An example of the **BuildArch** tag:

```
BuildArch: noarch
```

ExcludeArch: *architecture*

The **ExcludeArch** tag defines any excluded build architecture. If the package does not successfully compile, build or work on an architecture, then the architecture should be defined in the **ExcludeArch** tag. An example of the **ExcludeArch** tag:

```
ExcludeArch: i386
```

BuildRoot: *build root*

The **BuildRoot** defines the location of the files *installed* during the **%install** process, which happens after the **%build** compilation process. In a typical situation, leave this line alone: under the usual Fedora 18 setup, a macro that will create a new special subdirectory in the **/var/tmp/** directory will be used. Newer versions of RPM ignore this value, and instead place the build root in **%{_topdir}/BUILDROOT/**. An example of the **BuildRoot** tag:

```
BuildRoot: %{_tmppath}/%{name}-%{version}-%{release}-root
```

The BuildRoot tag is deprecated

RPM in Fedora 18 does not require the presence of the **BuildRoot** tag in the spec file. If the tag is defined, it will be ignored. The provided buildroot will automatically be cleaned before commands in **%install** are called.

BuildRequires: *requirements*

The **BuildRequires** tag defines a comma-separated list of packages required for building (or compiling) the software. These are *not* automatically determined, so you must include every package needed to build the software.

There are a few packages that are so common in builds that you do not need to mention them, such as *gcc*. Refer to the *Fedora Packaging Guidelines*[8] for a complete list of the packages you may omit.

You can use more than one BuildRequires tag, in which case all BuildRequires tags they are all required for building. If necessary, you can also specify a minimum version of the package, for example:

[8] http://fedoraproject.org/wiki/Packaging/Guidelines

```
ocaml >= 3.08
```

Try to specify only the minimal set of packages necessary to properly build the package, since each one will slow down a *mock*-based build. An example of the **BuildRequires** tag:

```
BuildRequires: gettext
```

Requires: *requirements*

The **Requires** tag defines a comma-separate list of packages that are required when the software is installed. Remember that the list of packages for the **Requires** tag and the **BuildRequires** tag are independent: a package may be in one list but not the other, or it can be in both. The dependencies of binary packages are in many cases automatically detected by **pmbuild**, so it is often the case that you do not need to specify the **Requires** tag at all. But if you want to highlight some specific packages as being required, or require a package that RPM cannot detect should be required, then add it to the **Requires** tag. An example of the **Requires** tag:

```
Requires: gettext
```

%description *description text*

The **%description** directive defines a longer, multiline description of the package. All lines must be 80 characters long or less. Blank lines are assumed to separate paragraphs. Remember that some graphical user interface installation programs will reformat paragraphs. Lines that start with whitespace, such as a space or tab, will be treated as preformatted text and displayed as is, normally with a fixed-width font. An example of the **%description** directive:

```
%description
The eject program allows the user to eject removable media (typically
CD-ROMs, floppy disks or Iomega Jaz or Zip disks) using software
control. Eject can also control some multi-disk CD changers and even
some devices' auto-eject features.

Install eject if you'd like to eject removable media using software
control.
```

%prep

The **%prep** directive defines script commands to *prepare* the software before the start of the building process. In a typical situation, the definition is similar to **%setup -q**. For example, if the source file unpacks into **name**, the definition is **%setup -q -n name**. An example of the **%prep** directive:

```
%prep
%setup -q -n %{name}
%patch1 -p1 -b .verbose
```

%build

The **%build** directive defines script commands to build (compile) the software, that is, to get it ready for installing. An example of the %build directive:

```
%build
%configure
make
```

%check

The **%check** directive defines script commands to self-test the program. The self-tests are run after **%build** and before **%install**, so the **%check** directive should be placed accordingly between the directives mentioned above. Usually, the **%check** directive contains the **make test** or **make check** commands. These commands are separated from the **%build** directive so that users can skip the self-test process, if they desire.

```
%check
make check
```

%install

The **%install** directive defines script commands to *install* the software. The script commands copy the build files from the build directory **%{_builddir}** (usually a subdirectory of the directory **~/rpmbuild/BUILD/**) into the build root directory **%{buildroot}** (usually a subdirectory of the directory **/var/tmp/**). An example of the**%install** directive:

```
%install
rm -rf %{buildroot}

make DESTDIR=%{buildroot} install

install -m 755 -d %{buildroot}/%{_sysconfdir}/pam.d
install -m 644 %{SOURCE1} %{buildroot}/%{_sysconfdir}/pam.d/%{name}
install -m 755 -d %{buildroot}/%{_sysconfdir}/security/console.apps/
echo "FALLBACK=true" > %{buildroot}/%{_sysconfdir}/security/console.apps/%{name}

install -m 755 -d %{buildroot}/%{_sbindir}
pushd %{buildroot}/%{_bindir}
mv eject ../sbin
ln -s consolehelper eject
popd
```

%clean

The **%clean** directive defines instructions on how to clean out the build root. For example:

```
%clean
rm -rf %{buildroot}
```

The %clean directive is deprecated

RPM in Fedora 18 does not require the presence of the **%clean** directive in the spec file. If the tag is defined, it will be ignored.

%files

The **%files** directive contains a list of files in the package to be installed. For example:

```
%files -f %{name}.lang
```

%changelog

The **%changelog** directive defines changes in the package. For example:

```
%changelog
* Wed Apr 02 2008 John Due <jdoe at example.com> 2.1.5-11
- Added check if device is hotpluggable
- Resolves #438610
```


DRAFT

Appendix B. Getting More Information

For more information on RPM packaging, refer to the resources listed below.

Installed Documentation

- **rpm**(8) – The manual page for the **rpm** utility for building, installing, querying, verifying, updating, and erasing individual RPM packages.
- **rpmbuild**(8) – The manual page for the **rpmbuild** utility for building both binary and source RPM packages.
- **rpmlint**(1) – The manual page for the **rpmlint** utility for checking common errors in RPM packages.


DRAFT

Appendix C. Revision History

Revision 0-0 **Thu Aug 25 2011** **Petr Kovář** *pkovar@redhat.com*

Initial creation of book by publican


DRAFT