Red Hat Software Collections 2.x

Packaging Guide

A guide to packaging Software Collections for Red Hat Enterprise Linux

Last Updated: 2017-10-17

Red Hat Software Collections 2.x Packaging Guide

A guide to packaging Software Collections for Red Hat Enterprise Linux

Petr Kovář
Red Hat Customer Content Services
pkovar@redhat.com

Legal Notice

Abstract

The Packaging Guide provides an explanation of Software Collections and details how to build and package them. Developers and system administrators who have a basic understanding of software packaging with RPM packages, but who are new to the concept of Software Collections, can use this Guide to get started with Software Collections.

Table of Contents

CHAPTER 1. INTRODUCING SOFTWARE COLLECTIONS

This chapter introduces you to the concept and usage of Software Collections or SCLs for short.

1.1. WHY PACKAGE SOFTWARE WITH RPM?

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

With RPM, you can:

Install, reinstall, remove, upgrade and verify packages.

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

Use a database of installed packages to query and verify packages.

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

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

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

Package pristine software sources into source and binary packages.

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

Add packages to Yum repositories.

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

Digitally sign your packages.

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

For in-depth information on what is RPM and how to use it, see the Red Hat Enterprise Linux 7 System Administrator's Guide, or the Red Hat Enterprise Linux 6 Deployment Guide.

1.2. WHAT ARE SOFTWARE COLLECTIONS?

With Software Collections, you can build and concurrently install multiple versions of the same software components on your system. Software Collections have no impact on the system versions of the packages installed by any of the conventional RPM package management utilities.

Software Collections:

Do not overwrite system files

Software Collections are distributed as a set of several components, which provide their full functionality without overwriting system files.

Are designed to avoid conflicts with system files

Software Collections make use of a special file system hierarchy to avoid possible conflicts between a single Software Collection and the base system installation.

Require no changes to the RPM package manager

Software Collections require no changes to the RPM package manager present on the host system.

Need only minor changes to the spec file

To convert a conventional package to a single Software Collection, you only need to make minor changes to the package spec file.

Allow you to build a conventional package and a Software Collection package with a single spec file

With a single spec file, you can build both the conventional package and the Software Collection package.

Uniquely name all included packages

With Software Collection's namespace, all packages included in the Software Collection are uniquely named.

Do not conflict with updated packages

Software Collection's namespace ensures that updating packages on your system causes no conflicts.

Can depend on other Software Collections

Because one Software Collection can depend on another, you can define multiple levels of dependencies.

1.3. ENABLING SUPPORT FOR SOFTWARE COLLECTIONS

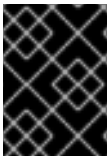
To enable support for Software Collections on your system so that you can enable and build Software Collections, you need to have installed the packages scl-utils and scl-utils-build.

If the packages scl-utils and scl-utils-build are not already installed on your system, you can install them by typing the following at a shell prompt as root:

```
# yum install scl-utils scl-utils-build
```

The scl-utils package provides the **scl** tool that lets you enable Software Collections on your system. For more information on enabling Software Collections, see Section 1.6, “Enabling a Software Collection”.

The scl-utils-build package provides macros that are essential for building Software Collections. For more information on building Software Collections, see Section 2.12, “Building a Software Collection”.



IMPORTANT

Depending on the subscriptions available to your Red Hat Enterprise Linux system, you may need to enable the **Optional** channel to install the scl-utils-build package.

1.4. INSTALLING A SOFTWARE COLLECTION

To ensure that a Software Collection is on your system, install the so-called metapackage of the Software Collection. Thanks to Software Collections being fully compatible with the RPM Package Manager, you can use conventional tools like **Yum** or **PackageKit** for this task.

For example, to install a Software Collection with the metapackage named **software_collection_1**, run the following command:

```
# yum install software_collection_1
```

This command will automatically install all the packages in the Software Collection that are essential for the user to perform most common tasks with the Software Collection.

Software Collections allow you to only install a subset of packages you intend to use. For example, to use the Ruby interpreter from the rh-ruby23 Software Collection, you only need to install a package rh-ruby23-ruby from that Software Collection.

If you install an application that depends on a Software Collection, that Software Collection will be installed along with the rest of the application's dependencies.

For detailed information on Software Collection metapackages, see Section 2.7.1, “Metapackage”.

For detailed information on **Yum** and **PackageKit** usage, see the Red Hat Enterprise Linux 7 System Administrator's Guide, or the Red Hat Enterprise Linux 6 Deployment Guide.

1.5. LISTING INSTALLED SOFTWARE COLLECTIONS

To get a list of Software Collections that are installed on the system, run the following command:

```
scl --list
```

To get a list of installed packages contained within a specified Software Collection, run the following command:

```
scl --list software_collection_1
```

1.6. ENABLING A SOFTWARE COLLECTION

The **scl** tool is used to enable a Software Collection and to run applications in the Software Collection environment.

General usage of the **scl** tool can be described using the following syntax:

```
scl action software_collection_1 software_collection_2 command
```

If you are running a ***command*** with multiple arguments, remember to enclose the command and its arguments in quotes:

```
scl action software_collection_1 software_collection_2 'command --
argument'
```

Alternatively, use a `--` command separator to run a ***command*** with multiple arguments:

```
scl action software_collection_1 software_collection_2 -- command --
argument
```

Remember that:

- When you run the **scl** tool, it creates a child process (subshell) of the current shell. Running the command again then creates a subshell of the subshell.
- You can list enabled Software Collections for the current subshell. See Section 1.7, “Listing Enabled Software Collections” for more information.
- You have to disable an enabled Software Collection first to be able to enable it again. To disable the Software Collection, exit the subshell created when enabling the Software Collections.
- When using the **scl** tool to enable a Software Collection, you can only perform one action with the enabled Software Collection at a time. The enabled Software Collection must be disabled first before performing another action.

1.6.1. Running an Application Directly

For example, to directly run **Perl** with the **`--version`** option in the Software Collection named **software_collection_1**, execute the following command:

```
scl enable software_collection_1 'perl --version'
```

Alternatively, you can create a wrapper script that shortens the commands for running applications in the Software Collection environment. For more information on wrappers, see Section 3.3, “Packaging Wrappers for Software Collections”.

1.6.2. Running a Shell with Multiple Software Collections Enabled

To run the **Bash** shell in the environment with multiple Software Collections enabled, execute the following command:

```
scl enable software_collection_1 software_collection_2 bash
```

The command above enables two Software Collections, named **software_collection_1** and **software_collection_2**.

1.6.3. Running Commands Stored in a File

To execute a number of commands, which are stored in a file, in the Software Collection environment, run the following command:

```
cat cmd | scl enable software_collection_1 -
```

The command above executes commands, which are stored in the **cmd** file, in the environment of the Software Collection named **software_collection_1**.

1.7. LISTING ENABLED SOFTWARE COLLECTIONS

To get a list of Software Collections that are enabled in the current session, print the **$X_SCLS** environment variable by running the following command:

```
echo $X_SCLS
```

1.8. UNINSTALLING A SOFTWARE COLLECTION

You can use conventional tools like **Yum** or **PackageKit** when uninstalling a Software Collection because Software Collections are fully compatible with the RPM Package Manager. For example, to uninstall all packages and subpackages that are part of a Software Collection named **software_collection_1**, run the following command:

```
yum remove software_collection_1\*
```

You can also use the **yum remove** command to remove the **scl** utility.

For detailed information on **Yum** and **PackageKit** usage, see the Red Hat Enterprise Linux 7 System Administrator's Guide, or the Red Hat Enterprise Linux 6 Deployment Guide.

CHAPTER 2. PACKAGING SOFTWARE COLLECTIONS

This chapter introduces you to packaging Software Collections.

2.1. CREATING YOUR OWN SOFTWARE COLLECTIONS

In general, you can use one of the following two approaches to deploy an application that depends on an existing Software Collection:

- install all required Software Collections and packages manually and then deploy your application, or
- create a new Software Collection for your application.

When creating a new Software Collection for your application:

Create a Software Collection metapackage

Each Software Collection includes a metapackage, which installs a subset of the Software Collection's packages that are essential for the user to perform most common tasks with the Software Collection. See Section 2.7.1, “Metapackage” for more information on creating metapackages.

Consider specifying the location of the Software Collection root directory

You are advised to specify the location of the Software Collection root directory by setting the `%_scl_prefix` macro in the Software Collection spec file. For more information, see Section 2.3, “The Software Collection Root Directory”.

Consider prefixing the name of your Software Collection packages

You are advised to prefix the name of your Software Collection packages with the vendor and Software Collection's name. For more information, see Section 2.4, “The Software Collection Prefix”.

Specify all Software Collections and other packages required by your application as dependencies

Ensure that all Software Collections and other packages required by your application are specified as dependencies of your Software Collection. For more information, see Section 2.10.8, “Making a Software Collection Depend on Another Software Collection”.

Convert existing conventional packages or create new Software Collection packages

Ensure that all macros in your Software Collection package spec files use conditionals. See Section 2.10, “Converting a Conventional Spec File” for more information on how to convert an existing package spec file.

Build your Software Collection

After you create the Software Collection metapackage and convert or create packages for your Software Collection, you can build the Software Collection with the **rpmbuild** utility. For more information, see Section 2.12, “Building a Software Collection”.

2.2. THE FILE SYSTEM HIERARCHY

The root directory of Software Collections is normally located in the **/opt/** directory to avoid possible conflicts between Software Collections and the base system installation. The use of the **/opt/** directory is recommended by the Filesystem Hierarchy Standard (FHS).

Below is an example of the file system hierarchy layout with two Software Collections, *software_collection_1* and *software_collection_2*:

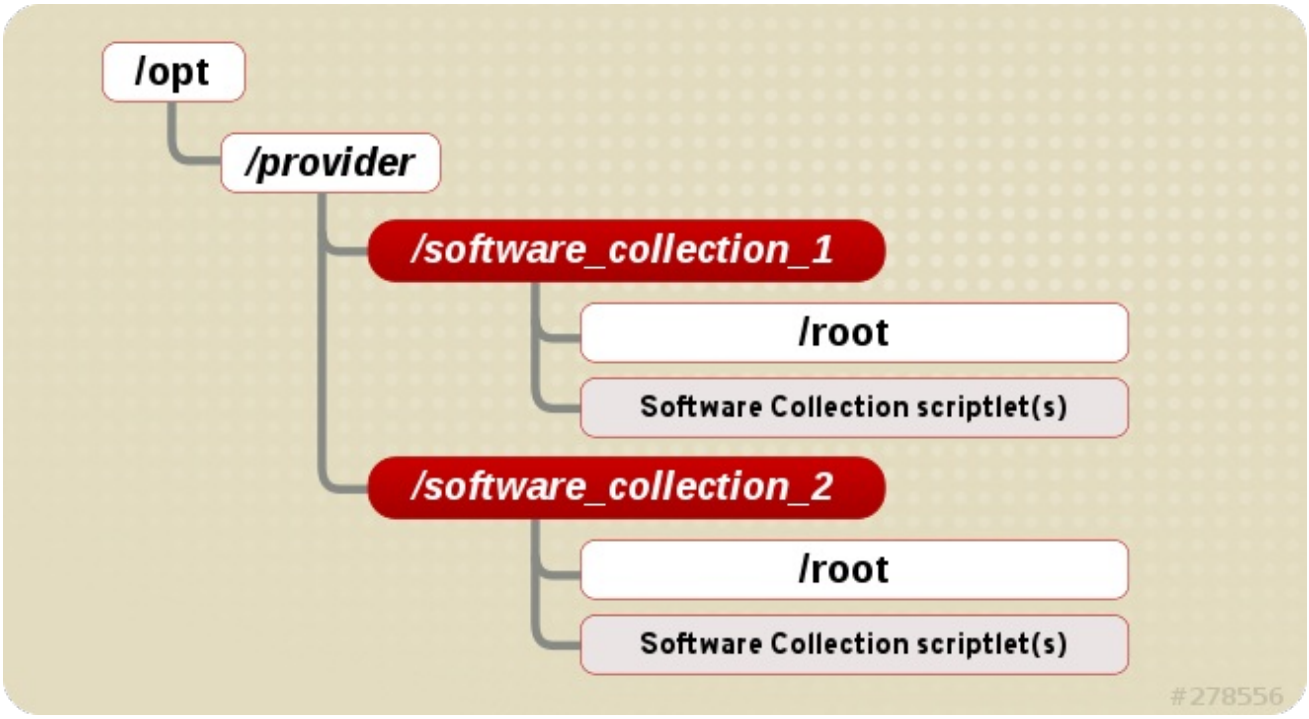


Figure 2.1. The Software Collection File System Hierarchy

As you can see above, each of the Software Collections directories contains the Software Collection root directory, and one or more Software Collection scriptlets. For more information on the Software Collection scriptlets, refer to Section 2.6, “Software Collection Scriptlets”.

2.3. THE SOFTWARE COLLECTION ROOT DIRECTORY

You can change the location of the root directory by setting the **%_scl_prefix** macro in the spec file, as in the following example:

```
%global _scl_prefix /opt/provider
```

where *provider* is the provider (vendor) name registered, where applicable, with the Linux Foundation and the subordinated Linux Assigned Names and Numbers Authority (LANANA), in conformance with the Filesystem Hierarchy Standard.

Each organization or project that builds and distributes Software Collections should use its own provider name, which conforms to the Filesystem Hierarchy Standard (FHS) and avoids possible conflicts between Software Collections and the base system installation.

You are advised to make the file system hierarchy conform to the following layout:

```
/opt/provider/prefix-application-version/
```

For more information on the Filesystem Hierarchy Standard, see http://www.pathname.com/fhs/.

For more information on the Linux Assigned Names and Numbers Authority, see http://www.lanana.org/.

2.4. THE SOFTWARE COLLECTION PREFIX

When naming your Software Collection, you are advised to prefix the name of your Software Collection as described below in order to avoid possible name conflicts with the system versions of the packages that are part of your Software Collection.

The Software Collection prefix consists of two parts:

- the *provider* part, which defines the provider's name, and
- the name of the Software Collection itself.

These two parts of the Software Collection prefix are separated by a dash (`-`), as in the following example:

```
myorganization-ruby193
```

In this example, *myorganization* is the provider's name, and *ruby193* is the name of the Software Collection.

While it is ultimately a vendor's or distributor's decision whether to specify the provider's name in the prefix or not, specifying it is highly recommended.

A notable exception are Software Collections which were first shipped with Red Hat Software Collections 1.x, they do not specify the provider's name in their prefixes. Newer Software Collections added in Red Hat Software Collections 2.0 and later use **`rh`** as the provider's name. For example:

```
rh-ruby23
```

2.5. SOFTWARE COLLECTION PACKAGE NAMES

The Software Collection package name consists of two parts:

- the *prefix* part, discussed in Section 2.4, “The Software Collection Prefix”, and
- the name and version number of the application that is a part of the Software Collection.

These two parts of the Software Collection package name are separated by a dash (`-`), as in the following example:

```
myorganization-ruby193-foreman-1.1
```

In this example, *myorganization-ruby193* is the prefix, and *foreman-1.1* is the name and version number of the application.

2.6. SOFTWARE COLLECTION SCRIPTLETS

The Software Collection scriptlets are simple shell scripts that change the current system environment so that the group of packages in the Software Collection is preferred over the corresponding group of conventional packages installed on the system.

To utilize the Software Collection scriptlets, use the **scl** tool that is part of the scl-utils package. For more information on **scl**, refer to Section 1.6, “Enabling a Software Collection”.

A single Software Collection can include multiple Software Collection scriptlets. These scriptlets are located in the **/opt/provider/software_collection/** directory in your Software Collection package. If you only need to distribute a single scriptlet in your Software Collection, it is highly recommended that you use **enable** as the name for that scriptlet. When the user runs a command in the Software Collection environment by executing **scl enable *software_collection command***, the **/opt/provider/software_collection/enable** scriptlet is then used to update search paths, and so on.

Note that Software Collection scriptlets can only set the system environment in a subshell that is created by running the **scl enable** command. The subshell is only active for the time the command is being performed.

2.7. PACKAGE LAYOUT

Each Software Collection's layout consists of the metapackage, which installs a subset of other packages, and a number of the Software Collection's packages, which are installed within the Software Collection namespace.

2.7.1. Metapackage

Each Software Collection includes a metapackage, which installs a subset of the Software Collection's packages that are essential for the user to perform most common tasks with the Software Collection. For example, the essential packages can provide the Perl language interpreter, but no Perl extension modules. The metapackage contains a basic file system hierarchy and delivers a number of the Software Collection's scriptlets.

The purpose of the metapackage is to make sure that all essential packages in the Software Collection are properly installed and that it is possible to enable the Software Collection.

The metapackage produces the following packages that are also part of the Software Collection:

The main package: *%name*

The main package in the Software Collection contains dependencies of the base packages, which are included in the Software Collection. The main package does not contain any files.

When specifying dependencies for your Software Collection's packages, ensure that no other package in your Software Collection depends on the main package. The purpose of the main package is to install only those packages that are essential for the user to perform most common tasks with the Software Collection.

Normally, the main package does not specify any build time dependencies (for instance, packages that are only build time dependencies of another Software Collection's packages).

For example, if the name of the Software Collection is **myorganization-ruby193**, then the main package macro is expanded to:

```
myorganization-ruby193
```

The runtime subpackage: *%name*-runtime

The runtime subpackage in the Software Collection owns the Software Collection's file system and delivers the Software Collection's scriptlets. This package needs to be installed for the user to be able to use the Software Collection.

For example, if the name of the Software Collection is **myorganization-ruby193**, then the runtime subpackage macro is expanded to:

```
myorganization-ruby193-runtime
```

The build subpackage: *%name*-build

The build subpackage in the Software Collection delivers the Software Collection's build configuration. It contains RPM macros needed for building packages into the Software Collection. The build subpackage is optional and can be excluded from the Software Collection.

For example, if the name of the Software Collection is **myorganization-ruby193**, then the build subpackage macro is expanded to:

```
myorganization-ruby193-build
```

The contents of the **myorganization-ruby193-build** subpackage are shown below:

```
$ cat /etc/rpm/macros.ruby193-config
%scl myorganization-ruby193
```

The scldevel subpackage: *%name*-scldevel

The scldevel subpackage in the %name Software Collection contains development files, which are useful when developing packages of another Software Collection that depends on the %name Software Collection. The scldevel subpackage is optional and can be excluded from the %name Software Collection.

For example, if the name of the Software Collection is **myorganization-ruby193**, then the scldevel subpackage macro is expanded to:

```
myorganization-ruby193-scldevel
```

For more information about the scldevel subpackage, see Section 4.1, “Providing an scldevel Subpackage”.

2.7.2. Creating a Metapackage

When creating a new metapackage:

- It is recommended to define the following macros at the top of the metapackage spec file:
  - **scl_name_prefix** that specifies the provider's name to be used as a prefix in your Software Collection's name, for example, *myorganization-*. This is different from **_scl_prefix**, which specifies the root of your Software Collection but also uses the provider's name. See Section 2.4, “The Software Collection Prefix” for more information.
  - **scl_name_base** that specifies the base name of your Software Collection, for example, *ruby*.

    - **scl_name_version** that specifies the version of your Software Collection, for example, *193*.

- You are advised to define a Software Collection macro **nfsmountable** that changes the location of configuration and state files and makes your Software Collection usable over NFS. For more information, see Section 3.1, “Using Software Collections over NFS”.
- Consider specifying all packages in your Software Collection that are essential for the Software Collection run time as dependencies of the metapackage. That way you can ensure that the packages are installed with the Software Collection metapackage.
- You are advised to add **Requires: scl-utils-build** to the build subpackage.
- You are not required to use conditionals for Software Collection-specific macros in the metapackage.
- Include any path redefinition that the packages in your Software Collection may require in the **enable** scriptlet.

  For information on commonly used path redefinitions, see Section 2.9, “Commonly Used Path Redefinitions”.
- Always make sure that the metapackage contains the **%setup** macro in the **%prep** section, otherwise building the Software Collection will fail. If you do not need to use a particular option with the **%setup** macro, add the **%setup -c -T** command to the **%prep** section.

  This is because the **%setup** macro defines and creates the **%buildsubdir** directory, which is normally used for storing temporary files at build time. If you do not define **%setup** in your Software Collection packages, files in the **%buildsubdir** directory will be overwritten, causing the build to fail.
- Add any macros you need to use to the **macros.%{scl}-config** file in the build subpackage.

Example of the Metapackage

To get an idea of what a typical metapackage for a Software Collection named *myorganization-ruby193* looks like, see the following example:

```
%global scl_name_prefix myorganization-
%global scl_name_base ruby
%global scl_name_version 193

%global scl %{scl_name_prefix}%{scl_name_base}%{scl_name_version}

# Optional but recommended: define nfsmountable
%global nfsmountable 1

%scl_package %scl
%global _scl_prefix /opt/myorganization

Summary: Package that installs %scl
Name: %scl_name
Version: 1
Release: 1%{?dist}
License: GPLv2+
Requires: %{scl_prefix}less
BuildRequires: scl-utils-build
```

```
%description
This is the main package for %scl Software Collection.

%package runtime
Summary: Package that handles %scl Software Collection.
Requires: scl-utils

%description runtime
Package shipping essential scripts to work with %scl Software Collection.

%package build
Summary: Package shipping basic build configuration
Requires: scl-utils-build

%description build
Package shipping essential configuration macros to build %scl Software
Collection.

# This is only needed when you want to provide an optional scldevel
subpackage
%package scldevel
Summary: Package shipping development files for %scl

%description scldevel
Package shipping development files, especially useful for development of
packages depending on %scl Software Collection.

%prep
%setup -c -T

%install
%scl_install

cat >> %{buildroot}%{_scl_scripts}/enable << EOF
export PATH="%{_bindir}:%{_sbindir}\${PATH:+:\${PATH}}"
export LD_LIBRARY_PATH="%
{_libdir}\${LD_LIBRARY_PATH:+:\${LD_LIBRARY_PATH}}"
export MANPATH="%{_mandir}:\${MANPATH:-}"
export PKG_CONFIG_PATH="%
{_libdir}/pkgconfig\${PKG_CONFIG_PATH:+:\${PKG_CONFIG_PATH}}"
EOF

# This is only needed when you want to provide an optional scldevel
subpackage
cat >> %{buildroot}%{_root_sysconfdir}/rpm/macros.%{scl_name_base}-
scldevel << EOF
%%scl_%{scl_name_base} %{scl}
%%scl_prefix_%{scl_name_base} %{scl_prefix}
EOF

# Install the generated man page
mkdir -p %{buildroot}%{_mandir}/man7/
install -p -m 644 %{scl_name}.7 %{buildroot}%{_mandir}/man7/

%files
```

```
%files runtime -f filelist
%scl_files

%files build
%{_root_sysconfdir}/rpm/macros.%{scl}-config

%files scldevel
%{_root_sysconfdir}/rpm/macros.%{scl_name_base}-scldevel

%changelog
* Fri Aug 30 2013 John Doe &lt;jdoe@example.com&gt; 1-1
- Initial package
```

2.8. SOFTWARE COLLECTION MACROS

The Software Collection packaging macro **scl** defines where to relocate the Software Collection's file structure. The relocated file structure is a file system used exclusively by the Software Collection.

The **%scl_package** macro defines files ownership for the Software Collection's metapackage and provides additional packaging macros to use in the Software Collection environment.

To be able to build a conventional package and a Software Collection package with a single spec file, prefix the Software Collection macros with **%{?scl:*macro*}**, as in the following example:

```
%{?scl:Requires: %scl_runtime}
```

In the example above, the **%scl_runtime** macro is the value of the **Requires** tag. Both the macro and the tag use the **%{?scl:** prefix.

2.8.1. Macros Specific to a Software Collection

The table below shows a list of all macros specific to a particular Software Collection. All the macros have default values that you will not need to change in most cases.

Table 2.1. Software Collection Specific Macros

Macro	Description	Example value
%scl_name	name of the Software Collection	**software_collection_1**
%scl_prefix	name of the Software Collection with a dash appended at the end	**software_collection_1-**
%pkg_name	name of the original package	**perl**
%_scl_prefix	root of the Software Collection (not package's root)	**/opt/provider/**
%_scl_scripts	location of Software Collection's scriptlets	**/opt/provider/software_collection_1/**

Macro	Description	Example value
%_scl_root	installation root (install-root) of the package	**/opt/provider/software_collection_1/root/**
%scl_require_package software_collection_1 package_2	depend on a particular package from a specific Software Collection	**software_collection_1-package_2**

2.8.2. Macros Not Specific to a Software Collection

The table below shows a list of macros that are not specific to a particular Software Collection. Because these macros are not relocated and do not point to the Software Collection file system, they allow you to point to the system root file system. These macros use **_root** as a prefix.

All the macros have default values that you will not need to change in most cases.

Table 2.2. Software Collection Non-Specific Macros

Macro	Description	Relocated	Example value
%_root_prefix	Software Collection's **%_prefix** macro	no	**/usr/**
%_root_exec_prefix	Software Collection's **%_exec_prefix** macro	no	**/usr/**
%_root_bindir	Software Collection's **%_bindir** macro	no	**/usr/bin/**
%_root_sbindir	Software Collection's **%_sbindir** macro	no	**/usr/sbin/**
%_root_datadir	Software Collection's **%_datadir** macro	no	**/usr/share/**
%_root_sysconfdir	Software Collection's **%_sysconfdir** macro	no	**/etc/**
%_root_libexecdir	Software Collection's **%_libexecdir** macro	no	**/usr/libexec/**
%_root_sharedstatedir	Software Collection's **%_sharedstatedir** macro	no	**/usr/com/**

Macro	Description	Relocated	Example value
%_root_localstatedir	Software Collection's **%_localstatedir** macro	no	**/usr/var/**
%_root_includedir	Software Collection's **%_includedir** macro	no	**/usr/include/**
%_root_infodir	Software Collection's **%_infodir** macro	no	**/usr/share/info/**
%_root_mandir	Software Collection's **%_mandir** macro	no	**/usr/share/man/**
%_root_initddir	Software Collection's **%_initddir** macro	no	**/etc/rc.d/init.d/**
%**_root_libdir**	Software Collection's **%_libdir** macro, this macro does not work if Software Collection's metapackage is platform-independent	no	**/usr/lib/**

2.8.3. The nfsmountable Macro

Using a Software Collection macro **nfsmountable** allows you to change values for the **_sysconfdir**, **_sharedstatedir**, and **_localstatedir** macros so that your Software Collection can have its state files and configuration files located outside the Software Collection's **/opt** file system hierarchy. This makes the files easier to manage and is also required when using your Software Collection over NFS.

If you do not need support for Software Collections over NFS, using **nfsmountable** is optional but recommended. For more information, see Section 3.1, “Using Software Collections over NFS”.

2.9. COMMONLY USED PATH REDEFINITIONS

This section lists environment variables commonly used to redefine paths in the **enable** scriptlet to set up the Software Collection environment. They are also used to specify the location of the Software Collection components in the Software Collection file system hierarchy.

Whether you need to specify a path redefinition in the **enable** scriptlet depends on the packages you choose to include in your Software Collection. The environment variables normally follow this pattern:

```
$ENV_VAR=$SCL_ENV_VAR:$ENV_VAR
```

2.9.1. Language-specific Path Redefinitions

GEM_PATH

The **GEM_PATH** environment variable specifies the location of Ruby gems. As such, it is also used in those Software Collections that extend the rh-ruby23 Software Collection. For more information, see Section 4.3, “Extending the rh-ruby23 Software Collection”.

Include the following in the **enable** scriptlet to redefine the environment variable:

```
export GEM_PATH="\${GEM_PATH:=%{gem_dir}:\`scl enable %{scl_ruby} --
ruby -e "print Gem.path.join(':')"\`}"
```

GOPATH

The **GOPATH** environment variable specifies the location of Go source and binary files. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export GOPATH="%{gopath}\${GOPATH:+:\${GOPATH}}"
```

JAVACONFDIRS

The **JAVACONFDIRS** environment variable is used to specify the location of the **java.conf** configuration file. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export JAVACONFDIRS="%
{_sysconfdir}/java\${JAVACONFDIRS:+:}\${JAVACONFDIRS:-}"
```

PERL5LIB

The **PERL5LIB** environment variable is used to specify the location of custom Perl modules so that they can be installed with the **%{?_scl_root}** prefix. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export PERL5LIB="%{_scl_root}%
{perl_vendorlib}\${PERL5LIB:+:\${PERL5LIB}}"
```

PYTHONPATH

The **PYTHONPATH** environment variable specifies the location of custom Python libraries. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export PYTHONPATH="%{_scl_root}%{python_sitearch}:%{_scl_root}%
{python_sitelib}\${PYTHONPATH:+:}\${PYTHONPATH:-}"
```

2.9.2. Other Path Redefinitions

CPATH

The **CPATH** environment variable specifies include paths for the **GCC** compiler to use. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export CPATH="%{_includedir}\${CPATH:+:\${CPATH}}"
```

INFOPATH

The **INFOPATH** environment variable specifies directories that contain Info files. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export INFOPATH="%{_infodir}\${INFOPATH:+:\${INFOPATH}}"
```

LD_LIBRARY_PATH

The **LD_LIBRARY_PATH** environment variable specifies the location of libraries. For more information, see Section 3.5, “Software Collection Library Support”.

Include the following in the **enable** scriptlet to redefine the environment variable:

```
export LD_LIBRARY_PATH="%
{_libdir}\${LD_LIBRARY_PATH:+:\${LD_LIBRARY_PATH}}"
```

LIBRARY_PATH

The **LIBRARY_PATH** environment variable specifies the location of special linker files or ordinary libraries for **GCC** to use. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export LIBRARY_PATH="%{_libdir}\${LIBRARY_PATH:+:\${LIBRARY_PATH}}"
```

MANPATH

The **MANPATH** environment variable specifies the location of man pages. For more information, see Section 3.7, “Software Collection MANPATH Support”.

Include the following in the **enable** scriptlet to redefine the environment variable:

```
export MANPATH="%{_mandir}:\${MANPATH:-}"
```

PATH

The **PATH** environment variable specifies the location of binary files. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export PATH="%{_bindir}:%{_sbindir}\${PATH:+:\${PATH}}"
```

PCP_DIR

The **PCP_DIR** environment variable specifies the location of files and directories used by **PCP**. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export PCP_DIR="%{_scl_root}"
```

PKG_CONFIG_PATH

The **PKG_CONFIG_PATH** environment variable specifies the location of **.pc** files used by the **pkg-config** program. For more information, see Section 3.6, “Software Collection .pc Files Support”.

Include the following in the **enable** scriptlet to redefine the environment variable:

```
export PKG_CONFIG_PATH="%
{_libdir}/pkgconfig\${PKG_CONFIG_PATH:+:\${PKG_CONFIG_PATH}}"
```

XDG_CONFIG_DIRS

The **XDG_CONFIG_DIRS** environment variable specifies the location of desktop configuration files according to the freedesktop.org specification. Include the following in the **enable** scriptlet to redefine the environment variable:

```
export XDG_CONFIG_DIRS="%{_sysconfdir}/xdg:\${XDG_CONFIG_DIRS:-
/etc/xdg}"
```

XDG_DATA_DIRS

The **XDG_DATA_DIRS** environment variable specifies the location of desktop data files according to the freedesktop.org specification. It is used in some Software Collections to locate the Software Collection-specific scripts or to enable bash completion.

Include the following in the **enable** scriptlet to redefine the environment variable:

```
export XDG_DATA_DIRS="%{_datadir}:\${XDG_DATA_DIRS:-/usr/local/share:%
{_root_datadir}}"
```

2.10. CONVERTING A CONVENTIONAL SPEC FILE

This section discusses converting a conventional spec file into a Software Collection spec file so that the converted spec file can be used in both the conventional package and the Software Collection.

2.10.1. Example of the Converted Spec File

To see what the diff file comparing a conventional spec file with a converted spec file looks like, refer to the following example:

```
--- a/less.spec
+++ b/less.spec
@@ -1,10 +1,13 @@
+%{?scl:%scl_package less}
+%{!?scl:%global pkg_name %{name}}
+
 Summary: A text file browser similar to more, but better
-Name: less
+Name: %{?scl_prefix}less
 Version: 444
 Release: 7%{?dist}
 License: GPLv3+
 Group: Applications/Text
-Source: http://www.greenwoodsoftware.com/less/%{name}-%{version}.tar.gz
+Source: http://www.greenwoodsoftware.com/less/%{pkg_name}-%
{version}.tar.gz
 Source1: lesspipe.sh
 Source2: less.sh
 Source3: less.csh
@@ -19,6 +22,7 @@ URL: http://www.greenwoodsoftware.com/less/
```

```
 Requires: groff
 BuildRequires: ncurses-devel
 BuildRequires: autoconf automake libtool
-Obsoletes: lesspipe < 1.0
+Obsoletes: %{?scl_prefix}lesspipe < 1.0
+%{?scl:Requires: %scl_runtime}

 %description
 The less utility is a text file browser that resembles more, but has
@@ -31,7 +35,7 @@ You should install less because it is a basic utility
for viewing text
 files, and you'll use it frequently.

 %prep
-%setup -q
+%setup -q -n %{pkg_name}-%{version}
 %patch1 -p1 -b .Foption
 %patch2 -p1 -b .search
 %patch4 -p1 -b .time
@@ -51,16 +55,16 @@ make CC="gcc $RPM_OPT_FLAGS -D_GNU_SOURCE -
D_LARGEFILE_SOURCE -D_LARGEFILE64_SOU
 %install
 rm -rf $RPM_BUILD_ROOT
 make DESTDIR=$RPM_BUILD_ROOT install
-mkdir -p $RPM_BUILD_ROOT/etc/profile.d
+mkdir -p $RPM_BUILD_ROOT%{_sysconfdir}/profile.d
 install -p -c -m 755 %{SOURCE1} $RPM_BUILD_ROOT/%{_bindir}
-install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT/etc/profile.d
-install -p -c -m 644 %{SOURCE3} $RPM_BUILD_ROOT/etc/profile.d
-ls -la $RPM_BUILD_ROOT/etc/profile.d
+install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{_sysconfdir}/profile.d
+install -p -c -m 644 %{SOURCE3} $RPM_BUILD_ROOT%{_sysconfdir}/profile.d
+ls -la $RPM_BUILD_ROOT%{_sysconfdir}/profile.d

 %files
 %defattr(-,root,root,-)
 %doc LICENSE
-/etc/profile.d/*
+%{_sysconfdir}/profile.d/*
 %{_bindir}/*
 %{_mandir}/man1/*
```

2.10.2. Converting Tags and Macro Definitions

The following steps show how to convert tags and macro definitions in a conventional spec file into a Software Collection spec file.

Procedure 2.1. Converting tags and macro definitions

1. Add the **%scl_package** macro to the spec file. Place the macro in front of the spec file preamble as follows:

   ```
   %{?scl:%scl_package package_name}
   ```

2. You are advised to define the **%pkg_name** macro in the spec file preamble in case the package is not built for the Software Collection:

```
%{!?scl:%global pkg_name %{name}}
```

   Consequently, you can use the **%pkg_name** macro to define the original name of the package wherever it is needed in the spec file that you can then use for building both the conventional package and the Software Collection.

3. Change the **Name** tag in the spec file preamble as follows:

```
Name: %{?scl_prefix}package_name
```

4. If you are building or linking with other Software Collection packages, then prefix the names of those Software Collection packages in the **Requires** and **BuildRequires** tags with **%{?scl_prefix}** as follows:

```
Requires: %{?scl_prefix}ifconfig
```

   When depending on the system versions of packages, you should avoid using versioned **Requires** or **BuildRequires**. If you need to depend on a package that could be updated by the system, consider including that package in your Software Collection, or remember to rebuild your Software Collection when the system package updates.

5. To check that all essential Software Collection's packages are dependencies of the main metapackage, add the following macro after the **BuildRequires** or **Requires** tags in the spec file:

```
%{?scl:Requires: %scl_runtime}
```

6. Prefix the **Obsoletes**, **Conflicts** and **BuildConflicts** tags with **%{?scl_prefix}**. This is to ensure that the Software Collection can be used to deploy new packages to older systems without having the packages specified, for example, by **Obsolete** removed from the base system installation. For example:

```
Obsoletes: %{?scl_prefix}lesspipe < 1.0
```

7. Prefix the **Provides** tag with **%{?scl_prefix}**, as in the following example:

```
Provides: %{?scl_prefix}more
```

2.10.3. Converting Subpackages

For any subpackages that define their name with the **-n** option, prefix their name with **%{?scl_prefix}**, as in the following example:

```
%package -n %{?scl_prefix}more
```

Prefixing applies not only to the **%package** macro, but also for **%description** and **%files**. For example:

```
%description -n %{?scl_prefix}rubygems
RubyGems is the Ruby standard for publishing and managing third party
libraries.
```

In case the subpackage requires the main package, make sure to also adjust the **Requires** tag in that subpackage so that the tag uses **%{?scl_prefix}%{pkg_name}**. For example:

```
Requires: %{?scl_prefix}%{pkg_name} = %{version}-%{release}
```

2.10.4. Converting RPM Scripts

This section describes general rules for converting RPM scripts that can often be found in the **%prep**, **%build**, **%install**, **%check**, **%pre**, and **%post** sections of a conventional spec file.

- Replace all occurrences of **%name** with **%pkg_name**. Most importantly, this includes adjusting the **%setup** macro.
  - Adjust the **%setup** macro in the **%prep** section of the spec file so that the macro can deal with a different package name in the Software Collection environment:

    ```
    %setup -q -n %{pkg_name}-%{version}
    ```

    Note that the **%setup** macro is required and that you must always use the macro with the **-n** option to successfully build your Software Collection.
- If you are using any of the **%_root_** macros to point to the system file system hierarchy, you must use conditionals for these macros so that you can then use the spec file for building both the conventional package and the Software Collection. Edit the macros as in the following example:

  ```
  mkdir -p %{?scl:%_root_sysconfdir}%{?!scl:%_sysconfdir}
  ```
- When building Software Collection packages that depend on other Software Collection packages, it is often important to ensure that the **scl enable** functionality links properly or run proper binaries, and so on. One of the examples where this is needed is compiling against a Software Collection library or running an interpreted script with the interpreter in the Software Collection.

  Wrap the script using the **%{?scl:** prefix, as in the following example:

  ```
  %{?scl:scl enable %scl - << \EOF}
   set -e
   ruby example.rb
   RUBYOPT="-Ilib" ruby bar.rb
   # The rest of the script contents goes here.
  %{?scl:EOF}
  ```

  It is important to specify **set -e** in the script so that the script behavior is consistent regardless of whether the script is executed in the **rpm** shell or the **scl** environment.
- Pay attention to any scripts that are executed during the Software Collection package installation, such as:

- **%pretrans**, **%pre**,
- **%post**, **%postun**, **%posttrans**,
- **%triggerin**, **%triggerun**, and **%triggerpostun**.

If you use the **scl enable** functionality in those scripts, you are advised to start with an empty environment to avoid any unintentional collisions with the base system installation.

To do so, use **env -i -** before enabling the Software Collection, as in the following example:

```
%posttrans
%{?scl:env -i - scl enable %{scl} - << \EOF}
%vagrant_plugin_register %{vagrant_plugin_name}
%{?scl:EOF}
```

- All hardcoded paths found in RPM scripts must be replaced with proper macros. For example, replace all occurrences of **/usr/share** with **%{_datadir}**. This is needed because the **$RPM_BUILD_ROOT** variable and the **%{build_root}** macro are not relocated by the **scl** macro.

2.10.5. Software Collection Automatic Provides and Requires and Filtering Support

IMPORTANT

The functionality described in this section is not available in Red Hat Enterprise Linux 6.

RPM in Red Hat Enterprise Linux 7 features support for automatic **Provides** and **Requires** and filtering. For example, for all Python libraries, RPM automatically adds the following **Requires**:

```
Requires: python(abi) = (version)
```

As explained in Section 2.10, “Converting a Conventional Spec File”, you should prefix this **Requires** with **%{?scl_prefix}** when converting your conventional RPM package:

```
Requires: %{?scl_prefix}python(abi) = (version))
```

Keep in mind that the scripts searching for these dependencies must sometimes be rewritten for your Software Collection, as the original RPM scripts are not extensible enough, and, in some cases, filtering is not usable. For example, to rewrite automatic Python **Provides** and **Requires**, add the following lines in the **macros.%{scl}-config** macro file:

```
%__python_provides /usr/lib/rpm/pythondeps-scl.sh --provides %{_scl_root}
%{scl_prefix}
%__python_requires /usr/lib/rpm/pythondeps-scl.sh --requires %{_scl_root}
%{scl_prefix}
```

The **/usr/lib/rpm/pythondeps-scl.sh** file is based on a **pythondeps.sh** file from the conventional package and adjusts search paths.

If there are **Provides** or **Requires** that you need to adjust, for example, a **pkg_config Provides**, there are two ways to do it:

- Add the following lines in the **macros.%{scl}-config** macro file so that it applies to all packages in the Software Collection:

```
%_use_internal_dependency_generator 0
%__deploop() while read FILE; do /usr/lib/rpm/rpmdeps -%{1} ${FILE};
done | /bin/sort -u
%__find_provides /bin/sh -c "%{?__filter_prov_cmd} %{__deploop P} %
{?__filter_from_prov}"
%__find_requires /bin/sh -c "%{?__filter_req_cmd}  %{__deploop R} %
{?__filter_from_req}"

# Handle pkgconfig's virtual Provides and Requires
%__filter_from_req | %{__sed} -e 's|pkgconfig|%{?
scl_prefix}pkgconfig|g'
%__filter_from_prov | %{__sed} -e 's|pkgconfig|%{?
scl_prefix}pkgconfig|g'
```

- Or, alternatively, add the following lines after tag definitions in every spec file for which you want to filter **Provides** or **Requires**:

```
%{?scl:%filter_from_provides s|pkgconfig|%{?scl_prefix}pkgconfig|g}
%{?scl:%filter_from_requires s|pkgconfig|%{?scl_prefix}pkgconfig|g}
%{?scl:%filter_setup}
```

IMPORTANT

When using filters, you need to pay attention to the automatic dependencies you change. For example, if the conventional package contains **Requires: pkgconfig(package_1)** and **Requires: pkgconfig(package_2)**, and only package_2 is included in the Software Collection, ensure that you do not filter the **Requires** tag for package_1.

2.10.6. Software Collection Macro Files Support

In some cases, you may need to ship macro files with your Software Collection packages. They are located in the **%{?scl:%{_root_sysconfdir}}%{!?scl:%{_sysconfdir}}/rpm/** directory, which corresponds to the **/etc/rpm/** directory for conventional packages. When shipping macro files, ensure that:

- You rename the macro files by appending **.%{scl}** to their names so that they do not conflict with the files from the base system installation.
- The macros in the macro files are either not expanded, or they are using conditionals, as in the following example:

```
%__python2 %{_bindir}/python
%python2_sitelib %(%{?scl:scl enable %scl '}%{__python2} -c "from
distutils.sysconfig import get_python_lib; print(get_python_lib())"%
{?scl:'})
```

As another example, there may be a situation where you need to create a Software Collection mypython that depends on a Software Collection python26. The python26 Software Collection defines the **%{__python2}** macro as in the above sample. This macro will evaluate to **/opt/provider/mypython/root/usr/bin/python2**, but the **python2** binary is only available in the python26 Software Collection (**/opt/provider/python26/root/usr/bin/python2**).

To be able to build software in the mypython Software Collection environment, ensure that:

- The **macros.python.python26** macro file, which is a part of the python26-python-devel package, contains the following line:

```
%__python26_python2 /opt/provider/python26/root/usr/bin/python2
```

- And the macro file in the python26-build subpackage, and also the build subpackage in any depending Software Collection, contains the following line:

```
%scl_package_override() {%global __python2 %__python26_python2}
```

This will redefine the **%{__python2}** macro only if the build subpackage from a corresponding Software Collection is present, which usually means that you want to build software for that Software Collection.

2.10.7. Software Collection Shebang Support

A shebang is a sequence of characters at the beginning of a script that is used as an interpreter directive. The shebang is processed by the automatic dependency generator and it points to a certain location, possibly in the system root file system.

When the automatic dependency generator processes the shebang, it adds dependencies according to the interpreters they point to. From the Software Collection point of view, there are two types of shebangs:

#!/usr/bin/env *example*

This shebang instructs the **/usr/bin/env** program to run the interpreter.

The automatic dependency generator will create a dependency on the **/usr/bin/env** program, as expected.

If the **$PATH** environment variable is redefined properly in the **enable** scriptlet, the *example* interpreter is found in the Software Collection file system hierarchy, as expected.

You are advised to rewrite the shebang in your Software Collection package so that the shebang specifies the full path to the interpreter located in the Software Collection file system hierarchy.

#!/usr/bin/*example*

This shebang specifies the direct path to the interpreter.

The automatic dependency generator will create a dependency on the **/usr/bin/example** interpreter located outside the Software Collection file system hierarchy. However, when building a package for your Software Collection, you often want to create a dependency on the **%{?_scl_root}/usr/bin/example** interpreter located in the Software Collection file system hierarchy.

Keep in mind that even when you properly redefine the **$PATH** environment variable, this has no effect on what interpreter is used. The system version of the interpreter located outside the Software Collection file system hierarchy is always used. In most cases, this is not desired.

If you are using this type of shebang and you want the shebang to point to the Software Collection file system hierarchy when building your Software Collection package, use a command like the following:

```
find %{buildroot} -type f | \
  xargs sed -i -e '1 s"^#!/usr/bin/example"#!%{?
_scl_root}/usr/bin/example"'
```

where */usr/bin/example* is the interpreter you want to use.

2.10.8. Making a Software Collection Depend on Another Software Collection

To make one Software Collection depend on a package from another Software Collection, you need to adjust the **BuildRequires** and **Requires** tags in the dependent Software Collection's spec file so that these tags properly define the dependency.

For example, to define dependencies on two Software Collections named **software_collection_1** and **software_collection_2**, add the following three lines to your application's spec file:

```
BuildRequires: scl-utils-build
Requires: %scl_require software_collection_1
Requires: %scl_require software_collection_2
```

Ensure that the spec file also contains the **%scl_package** macro in front of the spec file preamble, for example:

```
%{?scl:%scl_package less}
```

Note that the **%scl_package** macro must be included in every spec file of your Software Collection.

You can also use the **%scl_require_package** macro to define dependencies on a particular package from a specific Software Collection, as in the following example:

```
BuildRequires: scl-utils-build
Requires: %scl_require_package software_collection_1 package_name
```

2.11. UNINSTALLING ALL SOFTWARE COLLECTION DIRECTORIES

Keep in mind that the **yum remove** command does not uninstall directories provided by those Software Collection packages and subpackages that are removed after the Software Collection runtime subpackage is removed.

To ensure that all directories are uninstalled, make those packages and subpackages depend on the runtime subpackage. To do so, add the following line with the **%scl_runtime** macro to the spec file of each of those packages and subpackages:

```
%{?scl:Requires: %scl_runtime}
```

Adding the above line ensures that all directories provided by those packages and subpackages are removed correctly as long as the runtime subpackage does not depend on any of those packages and subpackages.

2.12. BUILDING A SOFTWARE COLLECTION

If you have correctly converted a conventional spec file for your Software Collection as documented in Section 2.10, “Converting a Conventional Spec File”, you will be able to build the resulting package in both the Software Collection and conventional build roots. Building the converted package in a conventional build root will produce a conventional base system RPM package, while building in a Software Collection build root that contains %{scl}-build will produce a Software Collection package.

To build a Software Collection on your system, run the following command:

```
rpmbuild -ba package.spec --define 'scl name'
```

The difference between the command shown above and the standard command to build conventional packages (**rpmbuild -ba *package*.spec**) is that you have to append the **--define** option to the **rpmbuild** command when building a Software Collection.

The **--define** option defines the **scl** macro, which uses the Software Collection configured in the Software Collection spec file (***package*.spec**).

Alternatively, to be able to use the standard command **rpmbuild -ba *package*.spec** to build the Software Collection, specify the following in the **package.spec** file:

```
BuildRequires: software_collection-build
```

where *software_collection* is the name of the Software Collection.

2.12.1. Rebuilding a Software Collection without build Subpackages

When you want to rebuild a Software Collection that comes with no build subpackage (*software_collection*-build), you can create the build subpackage by rebuilding the Software Collection metapackage, and thus avoid using the **rpmbuild -ba *package*.spec --define 'scl *name*'** command.

Note that you need to have the scl-utils-build package installed on your system, otherwise rebuilding the Software Collection metapackage with the **rpmbuild** command will fail.

For more information about the scl-utils-build package, see Section 1.3, “Enabling Support for Software Collections”.

2.12.2. Avoiding debuginfo File Conflicts

When you build two Software Collection packages (or a conventional RPM package and a Software Collection package) that specify the same **Source** tag, and thus unpack source files into the same directory underneath the **%_builddir** directory, their **debuginfo** packages will have file conflicts. Due to these conflicts, the user will be unable to install both packages on the same system at the same time.

To avoid these file conflicts, the spec file of one of the packages has to be altered to unpack its upstream source into a uniquely named top directory. This adds one more directory level to the build tree underneath the **%_builddir** directory. By doing so, the **debuginfo** package generation script produces **debuginfo** files that do not conflict with files from the other **debuginfo** package.

To see what the diff file comparing an original spec file with an altered spec file looks like, refer to the following example:

```
--- a/tbb.spec
+++ b/tbb.spec
@@ -66,11 +66,13 @@ PDF documentation for the user of the Threading
Building Block (TBB)
 C++ library.

 %prep
-%setup -q -n %{sourcebasename}
+%setup -q -c -n %{name}
+cd %{sourcebasename}
 %patch1 -p1
 %patch2 -p1

 %build
+cd %{sourcebasename}
 %{?scl:scl enable %{scl} - << \EOF}
 make %{?_smp_mflags} CXXFLAGS="$RPM_OPT_FLAGS" tbb_build_prefix=obj
 %{?scl:EOF}
@@ -81,6 +83,7 @@ done

 %install
 rm -rf $RPM_BUILD_ROOT
+cd %{sourcebasename}
 mkdir -p $RPM_BUILD_ROOT/%{_libdir}
 mkdir -p $RPM_BUILD_ROOT/%{_includedir}

@@ -108,20 +111,20 @@ done

 %files
 %defattr(-,root,root,-)
-%doc COPYING doc/Release_Notes.txt
+%doc %{sourcebasename}/COPYING %{sourcebasename}/doc/Release_Notes.txt
 %{_libdir}/*.so.2

 %files devel
 %defattr(-,root,root,-)
-%doc CHANGES
+%doc %{sourcebasename}/CHANGES
 %{_includedir}/tbb
 %{_libdir}/*.so
 %{_libdir}/pkgconfig/*.pc

 %files doc
 %defattr(-,root,root,-)
-%doc doc/Release_Notes.txt
-%doc doc/html
+%doc %{sourcebasename}/doc/Release_Notes.txt
+%doc %{sourcebasename}/doc/html

 %changelog
 * Wed Nov 13 2013 John Doe <jdoe@example.com> - 4.1-5.20130314
```

CHAPTER 3. ADVANCED TOPICS

This chapter discusses advanced topics on packaging Software Collections.

3.1. USING SOFTWARE COLLECTIONS OVER NFS

In some environments, the requirement is often to have a centralized model for how applications and tools are distributed rather than allowing users to install the application or tool version they prefer. In this way, NFS is the common method of mounting centrally managed software.

You need to define a Software Collection macro **`nfsmountable`** to use a Software Collection over NFS. If the macro is defined when building a Software Collection, the resulting Software Collection has its state files and configuration files located outside the Software Collection's **`/opt`** file system hierarchy. This enables you to mount the **`/opt`** file system hierarchy over NFS as read-only. It also makes state files and configuration files easier to manage.

If you do not need support for Software Collections over NFS, using **`nfsmountable`** is optional but recommended.

To define the **`nfsmountable`** macro, ensure that the Software Collection metapackage spec file contains the following lines:

```
%global nfsmountable 1

%scl_package %scl
```

As shown above, the **`nfsmountable`** macro must be defined before defining the **`%scl_package`** macro. This is because the **`%scl_package`** macro redefines the **`_sysconfdir`**, **`_sharedstatedir`**, and **`_localstatedir`** macros depending on whether the **`nfsmountable`** macro has been defined or not. The values that **`nfsmountable`** changes for the redefined macros are detailed in the following table.

Table 3.1. Changed Values for Software Collection Macros

Macro	Original definition	Expanded value for the original definition	Changed definition	Expanded value for the changed definition
`_sysconfdir`	%{_scl_root}/etc	/opt/provider/%{scl}/root/etc	%{_root_sysconfdir}%{_scl_prefix}/%{scl}	/etc/opt/provider/%{scl}
`_sharedstatedir`	%{_scl_root}/var/lib	/opt/provider/%{scl}/root/var/lib	%{_root_localstatedir}%{_scl_prefix}/%{scl}/lib	/var/opt/provider/%{scl}/lib
`_localstatedir`	%{_scl_root}/var	/opt/provider/%{scl}/root/var	%{_root_localstatedir}%{_scl_prefix}/%{scl}	/var/opt/provider/%{scl}

3.1.1. Changed Directory Structure and File Ownership

The **nfsmountable** macro also has an impact on how the **scl_install** and **scl_files** macros create a directory structure and set the file ownership when you run the **rpmbuild** command.

For example, a directory structure of a Software Collection named **software_collection** with the **nfsmountable** macro defined looks as follows:

```
$ rpmbuild -ba software_collection.spec --define 'scl software_collection'
...
$ rpm -qlp software_collection-runtime-1-1.el6.x86_64
/etc/opt/provider/software_collection
/etc/opt/provider/software_collection/X11
/etc/opt/provider/software_collection/X11/applnk
/etc/opt/provider/software_collection/X11/fontpath.d
...
/opt/provider/software_collection/root/usr/src
/opt/provider/software_collection/root/usr/src/debug
/opt/provider/software_collection/root/usr/src/kernels
/opt/provider/software_collection/root/usr/tmp
/var/opt/provider/software_collection
/var/opt/provider/software_collection/cache
/var/opt/provider/software_collection/db
/var/opt/provider/software_collection/empty
...
```

3.1.2. Registering and Deregistering Software Collections

In case a Software Collection is shared over NFS but not locally installed on your system, you need to make the **scl** tool aware of it by registering that Software Collection.

Registering a Software Collection is done by running the **scl register** command:

```
$ scl register /opt/provider/software_collection
```

where */opt/provider/software_collection* is the absolute path to the file system hierarchy of the Software Collection you want to register. The path's directory must contain the **enable** scriptlet and the **root/** directory to be considered a valid Software Collection file system hierarchy.

Deregistering a Software Collection is a reverse operation that you perform when you no longer want the **scl** tool to be aware of a registered Software Collection.

Deregistering a Software Collection is done by calling a **deregister** scriptet when running the **scl** command:

```
$ scl deregister software_collection
```

where *software_collection* is the name of the Software Collection you want to deregister.

3.1.2.1. Using (de)register Scriptlets in a Software Collection Metapackage

You can specify (de)register scriptlets in a Software Collection metapackage similarly to how enable scriptlets are specified. When specifying the scriptets, remember to explicitly include them in the **%file** section of the metapackage spec file.

See the following sample code for an example of specifying (de)register scriptets:

```
%install
%scl_install

cat >> %{buildroot}%{_scl_scripts}/enable << EOF
# Contents of the enable scriptlet goes here
...
EOF

cat >> %{buildroot}%{_scl_scripts}/register << EOF
# Contents of the register scriptlet goes here
...
EOF

cat >> %{buildroot}%{_scl_scripts}/deregister << EOF
# Contents of the deregister scriptlet goes here
...
EOF
...
%files runtime -f filelist
%scl_files
%{_scl_scripts}/register
%{_scl_scripts}/deregister
```

In the register scriptlet, you can optionally specify the commands you want to run when registering the Software Collection, for example, commands to create files in **/etc/opt/** or **/var/opt/**.

3.2. CONVERTING SOFTWARE COLLECTION SCRIPTLETS INTO ENVIRONMENT MODULES

Environment modules allow you to manage, for example, different versions of applications by dynamically modifying your shell environment. To use your Software Collection with the environment module system, convert the Software Collection's **enable** scriptlet into an environment module with a script **/usr/share/Modules/bin/createmodule.sh**.

Procedure 3.1. Converting an enable scriptlet into an environment module

1. Ensure that an environment-modules package is installed on your system:

   ```
   # yum install environment-modules
   ```

2. Run the **/usr/share/Modules/bin/createmodule.sh** script to convert your Software Collection's **enable** scriptlet into an environment module:

   ```
   /usr/share/Modules/bin/createmodule.sh /path/to/enable/scriptlet
   ```

   Replace */path/to/enable/scriptlet* with the file path of the **enable** scriptlet you want to convert.

3. Add the same command **/usr/share/Modules/bin/createmodule.sh** ***/path/to/enable/scriptlet*** in the **%pre** section of your Software Collection metapackage, below the code generating your **enable** scriptlet.

In case you have the **enable** scriptlet packaged as a file in one of your Software Collection packages, add the command **/usr/share/Modules/bin/createmodule.sh *_/path/to/enable/scriptlet_*** in the **%post** section.

See the **module**(1) man page for more information about environment modules.

3.3. PACKAGING WRAPPERS FOR SOFTWARE COLLECTIONS

Using wrappers is an easy way to shorten commands that the user runs in the Software Collection environment.

The following is an example of a wrapper from a Ruby-based Software Collection named rubyscl that is installed as **/usr/bin/rubyscl-ruby** and allows the user to run **rubyscl-ruby *_command_*** instead of **scl enable rubyscl 'ruby *_command_*'**:

```
#!/bin/bash

COMMAND="ruby $@"
scl enable rubyscl "$COMMAND"
```

It is important to package these wrappers as subpackages of the Software Collection package that will use them. That way, you can make installation of these wrappers optional, allowing the user not to install them, for example, on systems with read-only access to the **/usr/bin/** directory where the wrappers would otherwise be installed.

3.4. MANAGING SERVICES IN SOFTWARE COLLECTIONS

When packaging your Software Collection, ensure that users can directly manage any services (daemons) provided by the Software Collection or one of the associated applications with the system default tools, like **service** or **chkconfig** on Red Hat Enterprise Linux 6, or **systemctl** on Red Hat Enterprise Linux 7.

For Software Collections on Red Hat Enterprise Linux 6, make sure to adjust the **%install** section of the spec file as follows to avoid possible name conflicts with the system versions of the services that are part of the Software Collection:

```
%install
install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?scl:%_root_sysconfdir}%
{!?scl:%_sysconfdir}/rc.d/init.d/%{?scl_prefix}service_name
```

Replace *service_name* with the actual name of the service.

For Software Collections on Red Hat Enterprise Linux 7, adjust the **%install** section of the spec file as follows:

```
%install
install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{_unitdir}/%{?
scl_prefix}service_name.service
```

With this configuration in place, you can then refer to the version of the service included in the Software Collection as follows:

```
%{?scl_prefix}service_name
```

Keep in mind that no environment variables are propagated from the user's environment to a SysV init script (or a systemd service file on Red Hat Enterprise Linux 7). This is expected and ensures that services are always started in a clean environment. However, this requires you to properly set up a Software Collection environment for processes that are to be run by the SysV init scripts (or systemd service files).

3.4.1. Configuring an Environment for Services

It is recommended to make the Software Collection you want to enable for services configurable. The directions in this section show how to make a Software Collection named *software_collection* configurable.

Procedure 3.2. Configuring an environment for services on Red Hat Enterprise Linux 6

1. Create a configuration file in **/opt/provider/*software_collection*/service-environment** with the following content:

   ```
   [SCLNAME]_SCLS_ENABLED="software_collection"
   ```

   Replace *SCLNAME* with a unique identifier for your Software Collection, for instance, your Software Collection's name written in capital letters.

   Replace *software_collection* with the name of your Software Collection as defined by the **%scl_name** macro.

2. Add the following line at the beginning of the SysV init script:

   ```
   source /opt/provider/software_collection/service-environment
   ```

3. In the SysV init script, determine commands that run binaries located in the **/opt/provider/** file system hierarchy. Prefix these commands with **scl enable $[SCLNAME]_SCLS_ENABLED**, similarly to when you run a command in the Software Collection environment.

   For example, replace the following line:

   ```
   /usr/bin/daemon_binary --argument-1 --argument-2
   ```

   with:

   ```
   scl enable $[SCLNAME]_SCLS_ENABLED -- /usr/bin/daemon_binary --
   argument-1 --argument-2
   ```

4. Some commands, like **su** or **runuser**, also clear environment variables. Thus, if these commands are used in the SysV init script, enable your Software Collection again after running these commands.

   For instance, replace the following line:

   ```
   su - user_name -c '/usr/bin/daemon_binary --argument-1 --argument-2'
   ```

   with:

```
su - user_name -c '\
  source /opt/provider/software_collection/service-environment \
  scl enable $SCLNAME_SCLS_ENABLED -- /usr/bin/daemon_binary --
argument-1 --argument-2'
```

Procedure 3.3. Configuring an environment for services on Red Hat Enterprise Linux 7

1. Create a configuration file in **/opt/provider/*software_collection*/service-environment** with the following content:

   ```
   [SCLNAME]_SCLS_ENABLED="software_collection"
   ```

   Replace *SCLNAME* with a unique identifier for your Software Collection, for instance, your Software Collection's name written in capital letters.

   Replace *software_collection* with the name of your Software Collection as defined by the **%scl_name** macro.

2. Add the following line in the systemd service file to load the configuration file:

   ```
   EnvironmentFile=/opt/provider/software_collection/service-
   environment
   ```

3. In the systemd service file, prefix all commands specified in **ExecStartPre**, **ExecStart**, and similar directives with **scl enable $[SCLNAME]_SCLS_ENABLED**, similarly to when you run a command in the Software Collection environment:

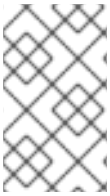
   ```
   ExecStartPre=/usr/bin/scl enable $[SCLNAME]_SCLS_ENABLED --
   /opt/provider/software_collection/root/usr/bin/daemon_helper_binary
   --argument-1 --argument-2
   ExecStart=/usr/bin/scl enable $[SCLNAME]_SCLS_ENABLED --
   /opt/provider/software_collection/root/usr/bin/daemon_binary --
   argument-1 --argument-2
   ```

3.5. SOFTWARE COLLECTION LIBRARY SUPPORT

In case you distribute libraries that you intend to use only in the Software Collection environment or in addition to the libraries available on the system, update the **LD_LIBRARY_PATH** environment variable in the **enable** scriptlet as follows:

```
export LD_LIBRARY_PATH="%
{_libdir}\${LD_LIBRARY_PATH:+:\${LD_LIBRARY_PATH}}"
```

The configuration ensures that the version of the library in the Software Collection is preferred over the version of the library available on the system if the Software Collection is enabled.



NOTE

In case you distribute a private shared library in the Software Collection, consider using the **DT_RUNPATH** attribute instead of the **LD_LIBRARY_PATH** environment variable to make the private shared library accessible in the Software Collection environment.

3.5.1. Using a Library Outside of the Software Collection

If you distribute libraries that you intend to use outside of the Software Collection environment, you can use the directory **/etc/ld.so.conf.d/** for this purpose.

WARNING

Do not use **/etc/ld.so.conf.d/** for libraries already available on the system. Using **/etc/ld.so.conf.d/** is only recommended for a library that is not available on the system, as otherwise the version of the library in the Software Collection might get preference over the system version of the library. That could lead to undesired behavior of the system versions of the applications, including unexpected termination and data loss.

Procedure 3.4. Using /etc/ld.so.conf.d/ for libraries in the Software Collection

1. Create a file named **%{?scl_prefix}libs.conf** and adjust the spec file configuration accordingly:

   ```
   SOURCE2: %{?scl_prefix}libs.conf
   ```

2. In the **%{?scl_prefix}libs.conf** file, include a list of directories where the versions of the libraries associated with the Software Collection are located. For example:

   ```
   /opt/provider/software_collection_1/root/usr/lib64/
   ```

   In the example above, the **/usr/lib64/** directory that is part of the Software Collection **software_collection_1** is included in the list.

3. Edit the **%install** section of the spec file, so the **%{?scl_prefix}libs.conf** file is installed as follows:

   ```
   %install
   install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?
   scl:%_root_sysconfdir}%{!?scl:%_sysconfdir}/ld.so.conf.d/
   ```

3.5.2. Prefixing the Library Major soname with the Software Collection Name

When using libraries included in the Software Collection, always remember that a library with the same major soname can already be available on the system as a part of the base system installation. It is thus important not to forget to use the **scl enable** command when building an application against a library included in the Software Collection. Failing to do so may result in the application being executed in an incorrect environment, linked against the incorrect system version of the library.

WARNING

Keep in mind that executing your application in an incorrect environment (for example in the system environment instead of the Software Collection environment) as well as linking your application against an incorrect library can lead to undesired behavior of your application, including unexpected termination and data loss.

To ensure that your application is not linked against an incorrect library even if the **LD_LIBRARY_PATH** environment variable has not been set properly, change the major soname of the library included in the Software Collection. The recommended way to change the major soname is to prefix the major soname version number with the Software Collection name.

Below is an example of the MySQL client library with the **mysql55-** prefix:

```
$ rpm -ql mysql55-mysql-libs | grep 'lib.*so'
/opt/provider/mysql55/root/usr/lib64/mysql/libmysqlclient.so.mysql55-18
/opt/provider/mysql55/root/usr/lib64/mysql/libmysqlclient.so.mysql55-
18.0.0
```

On the same system, the system version of the MySQL client library is listed below:

```
$ rpm -ql mysql-libs | grep 'lib.*so'
/usr/lib64/mysql/libmysqlclient.so.18
/usr/lib64/mysql/libmysqlclient.so.18.0.0
```

The **rpmbuild** utility generates an automatic **Provides** tag for packages that include a versioned shared library. If you do not prefix the soname as described above, then an example of the **Provides** in case of the mysql package is **libmysqlclient.so.18()(64bit)**. With this **Provides**, RPM can choose the incorrect RPM package, resulting in the application missing the requirement.

If you prefix the soname as described above, then an example of the generated **Provides** in case of mysql is **libmysqlclient.so.mysql55-18()(64bit)**. With this **Provides**, RPM chooses the correct RPM dependencies and the application's requirements are satisfied.

In general, unless absolutely necessary, Software Collection packages should not provide any symbols that are already provided by packages from the base system installation. One exception to that rule is when you want to use the symbols in the packages from the base system installation.

3.5.3. Software Collection Library Support in Red Hat Enterprise Linux 7

When building your Software Collection for Red Hat Enterprise Linux 7, use the **%__provides_exclude_from** macro to prevent scanning certain files for automatically generated RPM symbols.

For example, to prevent scanning **.so** files in the **%{_libdir}** directory, add the following lines before the **BuildRequires** or **Requires** tags in your Software Collection spec file:

```
%if %{?scl:1}%{!?scl:0}
  # Do not scan .so files in %{_libdir}
```

```
  %global __provides_exclude_from ^%{_libdir}/.*.so.*$
%endif
```

The functionality is part of RPM support for automatic **Provides** and **Requires**, see Section 2.10.5, "Software Collection Automatic Provides and Requires and Filtering Support" for more information.

3.6. SOFTWARE COLLECTION .PC FILES SUPPORT

The .pc files are special metadata files used by the **pkg-config** program to store information about libraries available on the system.

In case you distribute .pc files that you intend to use only in the Software Collection environment or in addition to the .pc files installed on the system, update the **PKG_CONFIG_PATH** environment variable. Depending on what is defined in your .pc files, update the **PKG_CONFIG_PATH** environment variable for the **%{_libdir}** macro (which expands to the library directory, typically **/usr/lib/** or **/usr/lib64/**), or for the **%{_datadir}** macro (which expands to the share directory, typically **/usr/share/**).

If the library directory is defined in your .pc files, update the **PKG_CONFIG_PATH** environment variable by adjusting the **%install** section of the Software Collection spec file as follows:

```
%install
cat >> %{buildroot}%{_scl_scripts}/enable << EOF
export PKG_CONFIG_PATH="%
{_libdir}/pkgconfig\${PKG_CONFIG_PATH:+:\${PKG_CONFIG_PATH}}"
EOF
```

If the share directory is defined in your .pc files, update the **PKG_CONFIG_PATH** environment variable by adjusting the **%install** section of the Software Collection spec file as follows:

```
%install
cat >> %{buildroot}%{_scl_scripts}/enable << EOF
export PKG_CONFIG_PATH="%
{_datadir}/pkgconfig\${PKG_CONFIG_PATH:+:\${PKG_CONFIG_PATH}}"
EOF
```

The two examples above both configure the **enable** scriptlet so that it ensures that the .pc files in the Software Collection are preferred over the .pc files available on the system if the Software Collection is enabled.

The Software Collection can provide a wrapper script that is visible to the system to enable the Software Collection, for example in the **/usr/bin/** directory. In this case, ensure that the .pc files are visible to the system even if the Software Collection is disabled.

To allow your system to use .pc files from the disabled Software Collection, update the **PKG_CONFIG_PATH** environment variable with the paths to the .pc files associated with the Software Collection. Depending on what is defined in your .pc files, update the **PKG_CONFIG_PATH** environment variable for the **%{_libdir}** macro (which expands to the library directory), or for the **%{_datadir}** macro (which expands to the share directory).

Procedure 3.5. Updating the PKG_CONFIG_PATH environment variable for %{_libdir}

1. To update the **PKG_CONFIG_PATH** environment variable for the **%{_libdir}** macro, create a custom script **/etc/profile.d/*name.sh***. The script is preloaded when a shell is started on the system.

For example, create the following file:

```
%{?scl_prefix}pc-libdir.sh
```

2. Use the **pc-libdir.sh** short script that modifies the **PKG_CONFIG_PATH** variable to refer to your .pc files:

```
export PKG_CONFIG_PATH="%
{_libdir}/pkgconfig:/opt/provider/software_collection/path/to/your/p
c_files"
```

3. Add the file to your Software Collection package's spec file:

```
SOURCE2: %{?scl_prefix}pc-libdir.sh
```

4. Install this file into the system **/etc/profile.d/** directory by adjusting the **%install** section of the Software Collection package's spec file:

```
%install
install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?
scl:%_root_sysconfdir}%{!?scl:%_sysconfdir}/profile.d/
```

Procedure 3.6. Updating the PKG_CONFIG_PATH environment variable for %{_datadir}

1. To update the **PKG_CONFIG_PATH** environment variable for the **%{_datadir}** macro, create a custom script **/etc/profile.d/*name.sh***. The script is preloaded when a shell is started on the system.

   For example, create the following file:

```
%{?scl_prefix}pc-datadir.sh
```

2. Use the **pc-datadir.sh** short script that modifies the **PKG_CONFIG_PATH** variable to refer to your .pc files:

```
export PKG_CONFIG_PATH="%
{_datadir}/pkgconfig:/opt/provider/software_collection/path/to/your/
pc_files"
```

3. Add the file to your Software Collection package's spec file:

```
SOURCE2: %{?scl_prefix}pc-datadir.sh
```

4. Install this file into the system **/etc/profile.d/** directory by adjusting the **%install** section of the Software Collection package's spec file:

```
%install
install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?
scl:%_root_sysconfdir}%{!?scl:%_sysconfdir}/profile.d/
```

3.7. SOFTWARE COLLECTION MANPATH SUPPORT

To allow the **man** command on the system to display man pages from the enabled Software Collection, update the **MANPATH** environment variable with the paths to the man pages that are associated with the Software Collection.

To update the **MANPATH** environment variable, add the following to the **%install** section of the Software Collection spec file:

```
%install
cat >> %{buildroot}%{_scl_scripts}/enable << EOF
export MANPATH="%{_mandir}:\${MANPATH:-}"
EOF
```

This configures the **enable** scriptlet to update the **MANPATH** environment variable. The man pages associated with the Software Collection are then not visible as long as the Software Collection is not enabled.

The Software Collection can provide a wrapper script that is visible to the system to enable the Software Collection, for example in the **/usr/bin/** directory. In this case, ensure that the man pages are visible to the system even if the Software Collection is disabled.

To allow the **man** command on the system to display man pages from the disabled Software Collection, update the **MANPATH** environment variable with the paths to the man pages associated with the Software Collection.

Procedure 3.7. Updating the MANPATH environment variable for the disabled Software Collection

1. To update the **MANPATH** environment variable, create a custom script **/etc/profile.d/*name.sh***. The script is preloaded when a shell is started on the system.

   For example, create the following file:

   ```
   %{?scl_prefix}manpage.sh
   ```

2. Use the **manpage.sh** short script that modifies the **MANPATH** variable to refer to your man path directory:

   ```
   export
   MANPATH="/opt/provider/software_collection/path/to/your/man_pages:${
   MANPATH}"
   ```

3. Add the file to your Software Collection package's spec file:

   ```
   SOURCE2: %{?scl_prefix}manpage.sh
   ```

4. Install this file into the system **/etc/profile.d/** directory by adjusting the **%install** section of the Software Collection package's spec file:

   ```
   %install
   install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?
   scl:%_root_sysconfdir}%{!?scl:%_sysconfdir}/profile.d/
   ```

3.8. SOFTWARE COLLECTION CRONJOB SUPPORT

With your Software Collection, you can run periodic tasks on the system either with a dedicated service or with cronjobs. If you intend to use a dedicated service, refer to Section 3.4, “Managing Services in Software Collections” on how to work with initscripts in the Software Collection environment.

Procedure 3.8. Running periodic tasks with cronjobs

1. To use cronjobs for running periodic tasks, place a **crontab** file for your Software Collection in the **/etc/cron.d/** directory with the Software Collection's name.

   For example, create the following file:

   ```
   %{?scl_prefix}crontab
   ```

2. Ensure that the contents of the **crontab** file follow the standard **crontab** file format, as in the following example:

   ```
   0 1 * * Sun root scl enable software_collection
   '/opt/provider/software_collection/root/usr/bin/cron_job_name'
   ```

   where *software_collection* is the name of your Software Collection, and **/opt/*provider*/*software_collection*/root/usr/bin/*cron_job_name*** is the command you want to periodically run.

3. Add the file to your spec file of the Software Collection package:

   ```
   SOURCE2: %{?scl_prefix}crontab
   ```

4. Install the file into the system directory **/etc/cron.d/** by adjusting the **%install** section of the Software Collection package's spec file:

   ```
   %install
   install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?
   scl:%_root_sysconfdir}%{!?scl:%_sysconfdir}/cron.d/
   ```

3.9. SOFTWARE COLLECTION LOG FILE SUPPORT

By default, programs packaged in a Software Collection create log files in the **/opt/*provider*/%{scl}/root/var/log/** directory.

To make log files more accessible and easier to manage, you are advised to use the **nfsmountable** macro that redefines the **_localstatedir** macro. This results in log files being created underneath the **/var/opt/*provider*/%{scl}/log/** directory, outside of the **/opt/*provider*/%{scl}** file system hierarchy.

For example, a service *mydaemon* normally stores its log file in **/var/log/mydaemon/mydaemond.log** in the base system installation. When *mydaemon* is packaged as a *software_collection* Software Collection and the **nfsmountable** macro is defined, the path to the log file in *software_collection* is as follows:

```
/var/opt/provider/software_collection/log/mydaemon/mydaemond.log
```

For more information on using the **nfsmountable** macro, see Section 3.1, “Using Software Collections over NFS”.

3.10. SOFTWARE COLLECTION LOGROTATE SUPPORT

With your Software Collection or an application associated with your Software Collection, you can manage log files with the **logrotate** program.

Procedure 3.9. Managing log files with logrotate

1. To manage your log files with **logrotate**, place a custom **logrotate** file for your Software Collection in the system directory for the **logrotate** jobs **/etc/logrotate.d/**.

   For example, create the following file:

   ```
   %{?scl_prefix}logrotate
   ```

2. Ensure that the contents of the **logrotate** file follow the standard **logrotate** file format as follows:

   ```
   /opt/provider/software_collection/var/log/your_application_name.log
   {
         missingok
         notifempty
         size 30k
         yearly
         create 0600 root root
     }
   ```

3. Add the file to your spec file of the Software Collection package:

   ```
   SOURCE2: %{?scl_prefix}logrotate
   ```

4. Install the file into the system directory **/etc/logrotate.d/** by adjusting the **%install** section of the Software Collection package's spec file:

   ```
   %install
   install -p -c -m 644 %{SOURCE2} $RPM_BUILD_ROOT%{?
   scl:%_root_sysconfdir}%{!?scl:%_sysconfdir}/logrotate.d/
   ```

3.11. SOFTWARE COLLECTION /VAR/RUN/ FILES SUPPORT

PID files are one example of files usually located underneath the **/var/run/*package_name*/** directory. When packaging PID files into your Software Collection, you are advised to use the **nfsmountable** macro and store the PID files in the following directory:

```
/var/run/software_collection-package_name/
```

where *software_collection* is the name of your Software Collection and *package_name* is the name of the package included in your Software Collection.

Following this naming convention avoids file conflicts with the base system installation, while it makes it possible for your Software Collection to use **/var/run/** features, for example the **tmpfs** file system for PID files.

For more information on using the **nfsmountable** macro, see Section 3.1, “Using Software Collections over NFS”.

3.12. SOFTWARE COLLECTION LOCK FILE SUPPORT

By default, programs packaged into a Software Collection create lock files in the **/opt/*provider*/%{scl}/root/var/lock/** directory.

To make lock files more accessible and easier to manage, you are advised to use the **nfsmountable** macro that redefines the **_localstatedir** macro. This results in lock files being created underneath the **/var/opt/*provider*/%{scl}/lock/** directory, outside of the **/opt/*provider*/%{scl}** file system hierarchy.

If applications or services packaged into your Software Collection write the lock underneath the **/var/opt/*provider*/%{scl}/lock/** directory, then those applications and services can run concurrently with the system versions (when the resources of your Software Collection's applications and services will not conflict with the system versions' resources).

For example, a lock file **mylockfile.lock** is normally created in the **/var/lock/** directory in the base system installation. If the lock file is a part of a *software_collection* Software Collection and the **nfsmountable** macro is defined, the path to the lock file in *software_collection* is as follows:

```
/var/opt/provider/software_collection/lock/mylockfile.lock
```

For more information on using the **nfsmountable** macro, see Section 3.1, “Using Software Collections over NFS”.

Preventing Programs from Running Concurrently

If you want to prevent your Software Collection's applications or services from running while the system version of the respective application or service is running, make sure that your applications or services, which require a lock, write the lock to the system directory **/var/lock/**. In this way, your applications or services' lock file will not be overwritten. The lock file will not be renamed and the name stays the same as the system version.

3.12.1. Software Collection SysV init Lock File Support

When a service is started by an init script, a lock file is touched in the **/var/lock/subsys/** directory with the same name as the init script. As discussed in Section 3.4, “Managing Services in Software Collections”, service names include a Software Collection prefix. Use the same naming convention for files underneath **/var/lock/subsys/** to ensure that the lock file names do not conflict with the base system installation.

3.13. SOFTWARE COLLECTION CONFIGURATION FILES SUPPORT

By default, configuration files in a Software Collection are stored within the **/opt/*provider*/%{scl}** file system hierarchy.

To make configuration files more accessible and easier to manage, you are advised to use the **nfsmountable** macro that redefines the **_sysconfdir** macro. This results in configuration files being

created underneath the **/etc/opt/*provider*/%{scl}/** directory, outside of the **/opt/*provider*/%{scl}** file system hierarchy.

For example, a configuration file *example.conf* is normally stored in the **/etc** directory in the base system installation. If the configuration file is a part of a *software_collection* Software Collection and the **nfsmountable** macro is defined, the path to the configuration file in *software_collection* is as follows:

```
/etc/opt/provider/software_collection/example.conf
```

For more information about using the **nfsmountable** macro, see Section 3.1, “Using Software Collections over NFS”.

3.14. SOFTWARE COLLECTION KERNEL MODULE SUPPORT

Because Linux kernel modules are normally tied to a particular version of the Linux kernel, you must be careful when you package kernel modules into a Software Collection. This is because the package management system on Red Hat Enterprise Linux does not automatically update or install an updated version of the kernel module if an updated version of the Linux kernel is installed. To make packaging the kernel modules into the Software Collection easier, see the following recommendations. Ensure that:

1. the name of your kernel module package includes the kernel version,
2. the tag **Requires**, which can be found in your kernel module spec file, includes the kernel version and revision (in the format **kernel-*version*-*revision***).

3.15. SOFTWARE COLLECTION SELINUX SUPPORT

Because Software Collections are designed to install the Software Collection packages in an alternate directory, set up the necessary SELinux labels so that SELinux is aware of the alternate directory.

If the file system hierarchy of your Software Collection package imitates the file system hierarchy of the corresponding conventional package, you can run the **semanage fcontext** and **restorecon** commands to set up the SELinux labels.

For example, if the **/opt/provider/software_collection_1/root/usr/** directory in your Software Collection package imitates the **/usr/** directory of your conventional package, set up the SELinux labels as follows:

```
semanage fcontext -a -e /usr /opt/provider/software_collection_1/root/usr
```

```
restorecon -R -v /opt/provider/software_collection_1/root/usr
```

The commands above ensure that all directories and files in the **/opt/provider/software_collection_1/root/usr/** directory are labeled by SELinux as if they were located in the **/usr/** directory.

3.15.1. SELinux Support in Red Hat Enterprise Linux 7

When packaging a Software Collection for Red Hat Enterprise Linux 7, add the following commands to the **%post** section in the Software Collection metapackage to set up the SELinux labels:

```
semanage fcontext -a -e /usr /opt/provider/software_collection_1/root/usr
```

```
restorecon -R -v /opt/provider/software_collection_1/root/usr
```

```
selinuxenabled && load_policy || :
```

The last command ensures that the newly created SELinux policy is properly loaded, and that the files installed by a package in the Software Collection are created with the correct SELinux context. By using this command in the metapackage, you do not need to include the **restorecon** command in all packages in the Software Collection.

Note that the **semanage fcontext** command is provided by the policycoreutils-python package, therefore it is important that you include **policycoreutils-python** in **Requires** for the Software Collection metapackage.

3.16. DIFFERENCES BETWEEN RED HAT ENTERPRISE LINUX 6 AND 7

The RPM Package Manager in Red Hat Enterprise Linux 7 ships with a number of feature changes that are not available in the older version of the RPM Package Manager shipped with Red Hat Enterprise Linux 6.

This section provides more details on the changes that may affect you when building your Software Collection packages for both systems.

Differences in library support are detailed in Section 3.5.3, “Software Collection Library Support in Red Hat Enterprise Linux 7”. Differences in SELinux support are documented in Section 3.15.1, “SELinux Support in Red Hat Enterprise Linux 7”.

3.16.1. The %license Macro

The **%license** macro allows you to specify the license file to be installed by your package. The macro is only supported by the RPM Package Manager in Red Hat Enterprise Linux 7. When building your Software Collection package on both Red Hat Enterprise Linux 6 and 7, declare the **%license** macro for Red Hat Enterprise Linux 6 as follows:

```
%{!?_licensedir:%global license %%doc}
```

3.16.2. Missing runtime Subpackage Dependencies

On Red Hat Enterprise Linux 7, the **scl** tool automatically generates the needed **Requires** on the Software Collection runtime subpackage. This does not work on Red Hat Enterprise Linux 6. When building your Software Collection for that system, you need to explicitly specify the dependency on the runtime subpackage in each Software Collection package:

```
Requires: %{?scl_prefix}runtime
```

3.16.3. The scl-package() Provides

By design, building a Software Collection package generates a number of **Provide: scl-package()** tags. The purpose of these is to internally identify the built package as belonging to a specific Software Collection. The tags are detailed in the following table.

Table 3.2. Provides in Red Hat Enterprise Linux 7

Software Collection package	Provide
`${software_collection_1}`	`scl-package(software_collection_1)`
`${software_collection_1}-build`	`scl-package(software_collection_1)`
`${software_collection_1}-runtime`	`scl-package(software_collection_1)`

Red Hat Enterprise Linux 6 ships with an older version of the RPM Package Manager, so as an exception, building the same package on Red Hat Enterprise Linux 6 only generates a single **`Provide: scl-package()`** tag, as detailed in the following table. This is an expected behavior and the differences are handled internally by the **scl** tool.

Table 3.3. Provide in Red Hat Enterprise Linux 6

Software Collection package	Provide
`${software_collection_1}`	`scl-package(software_collection_1)`

Do not use these internally generated dependencies to list packages that belong to a particular Software Collection. For information on how to properly list Software Collection packages, see Section 1.5, “Listing Installed Software Collections”.

CHAPTER 4. EXTENDING RED HAT SOFTWARE COLLECTIONS

This chapter describes extending some of the Software Collections that are part of the Red Hat Software Collections offering.

4.1. PROVIDING AN SCLDEVEL SUBPACKAGE

The purpose of an scldevel subpackage is to make the process of creating dependent Software Collections easier by providing a number of generic macro files. Packagers then use these macro files when they are extending existing Software Collections. scldevel is provided as a subpackage of your Software Collection's metapackage.

The following section describes creating an scldevel subpackage for two examples of Ruby Software Collections, ruby193 and ruby200.

Procedure 4.1. Providing your own scldevel subpackage

1. In your Software Collection's metapackage, add the scldevel subpackage by defining its name, summary, and description:

```
%package scldevel
Summary: Package shipping development files for %scl
Provides: scldevel(%{scl_name_base})

%description scldevel
Package shipping development files, especially useful for
development of
packages depending on %scl Software Collection.
```

   You are advised to use the virtual **`Provides: scldevel(%{scl_name_base})`** during the build of packages of dependent Software Collections. This will ensure availability of a version of the **`%{scl_name_base}`** Software Collection and its macros, as specified in the following step.

2. In the **`%install`** section of your Software Collection's metapackage, create the **`macros.%{scl_name_base}-scldevel`** file that is part of the scldevel subpackage and contains:

```
cat >> %{buildroot}%{_root_sysconfdir}/rpm/macros.%{scl_name_base}-
scldevel << EOF
%%scl_%{scl_name_base} %{scl}
%%scl_prefix_%{scl_name_base} %{scl_prefix}
EOF
```

   Note that between all Software Collections that share the same **`%{scl_name_base}`** name, the provided **`macros.%{scl_name_base}-scldevel`** files must conflict. This is to disallow installing multiple versions of the **`%{scl_name_base}`** Software Collections. For example, the ruby193-scldevel subpackage cannot be installed when there is the ruby200-scldevel subpackage installed.

4.1.1. Using an scldevel Subpackage in a Dependent Software Collection

To use your scldevel subpackage in a Software Collection that depends on the ruby200 Software Collection, update the metapackage of the dependent Software Collection as described below.

Procedure 4.2. Using your own scldevel subpackage in a dependent Software Collection

1. Consider adding the following at the beginning of the metapackage's spec file:

```
%{!?scl_ruby:%global scl_ruby ruby200}
%{!?scl_prefix_ruby:%global scl_prefix_ruby %{scl_ruby}-}
```

   These two lines are optional. They are only meant as a visual hint that the dependent Software Collection has been designed to depend on the ruby200 Software Collection. If there is no other scldevel subpackage available in the build root, then the ruby200-scldevel subpackage is used as a build requirement.

   You can substitute these lines with the following line:

```
%{?scl_prefix_ruby}
```

2. Add the following build requirement to the metapackage:

```
BuildRequires: %{scl_prefix_ruby}scldevel
```

   By specifying this build requirement, you ensure that the scldevel subpackage is in the build root and that the default values are not in use. Omitting this package could result in broken requires at the subsequent packages' build time.

3. Ensure that the **`%package runtime`** part of the metapackage's spec file includes the following lines:

```
%package runtime
Summary: Package that handles %scl Software Collection.
Requires: scl-utils
Requires: %{scl_prefix_ruby}runtime
```

4. Ensure that the **`%package build`** part of the metapackage's spec file includes the following lines:

```
%package build
Summary: Package shipping basic build configuration
Requires: %{scl_prefix_ruby}scldevel
```

   Specifying **`Requires: %{scl_prefix_ruby}scldevel`** ensures that macros are available in all packages of the Software Collection.

4.2. EXTENDING THE PYTHON27 AND RH-PYTHON35 SOFTWARE COLLECTIONS

This section describes extending the python27 and rh-python35 Software Collections by creating a dependent Software Collection.

In Red Hat Software Collections 2.4, the **scl** tool is extended to support a macro **`%scl_package_override()`**, which allows for easier packaging of your own dependent Software Collection.

4.2.1. The vt191 Software Collection

Below is a commented example of building a dependent Software Collection. The Software Collection is named **vt191** and contains the versiontools Python package version 1.9.1.

Note the following in the vt191 Software Collection metapackage:

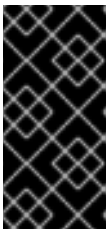
- The vt191 Software Collection metapackage has the following build dependency set:

  ```
  BuildRequires: %{scl_prefix_python}scldevel
  ```

  This expands to, for example, python27-scldevel.

  The python27-scldevel subpackage ships two important macros, **%scl_python** and **%scl_prefix_python**. Note that these macros are defined at the top of the metapackage spec file. Although the definitions are not required, they provide a visual hint that the vt191 Software Collection has been designed to be built on top of the python27 Software Collection. They also serve as a fallback value.

- To have a **site-packages** directory set up properly, use the value of the **%python27python_sitelib** macro and replace **python27** with **vt191**. Note that if you are building the Software Collection with a different provider (for example, **/opt/myorganization/** instead of **/opt/rh/**), you will need to change these, too.

  IMPORTANT

  Because the **/opt/rh/** provider is used to install Software Collections provided by Red Hat, it is strongly recommended to use a different provider to avoid possible conflicts. See Section 2.3, “The Software Collection Root Directory” for more information.

- The vt191-build subpackage has the following dependency set:

  ```
  Requires: %{scl_prefix_python}scldevel
  ```

  This expands to, for example, python27-scldevel. The purpose of this dependency is to ensure that the macros are always present when building packages for the vt191 Software Collection.

- The **enable** scriptlet for the vt191 Software Collection uses the following line:

  ```
  . scl_source enable %{scl_python}
  ```

  Note the dot at the beginning of the line. This line makes the Python Software Collection start implicitly when the vt191 Software Collection is started so that the user can only type **scl enable vt191 *command*** instead of **scl enable python27 vt191 *command*** to run *command* in the Software Collection environment.

- The macro file **macros.vt191-config** calls the **%scl_package_override** function to properly override **%__os_install_post**, Python dependency generators, and certain Python-specific macros used in other packages' spec files.

```
# define name of the scl
%global scl vt191
```

```
%scl_package %scl

# Defaults for the values for the python27/rh-python35 Software Collection.
These
# will be used when python27-scldevel (or rh-python35-scldevel) is not in
the
# build root
%{!?scl_python:%global scl_python python27}
%{!?scl_no_vendor:%global scl_no_vendor python27}
%{!?scl_prefix_python:%global scl_prefix_python %{scl_python}-}

# Only for this build, you need to override default __os_install_post,
# because the default one would find /opt/.../lib/python2.7/ and try
# to bytecompile with the system /usr/bin/python2.7
%global __os_install_post %{%{scl_no_vendor}_os_install_post}
# Similarly, override __python_requires for automatic dependency generator
%global __python_requires %{%{scl_no_vendor}_python_requires}

# The directory for site packages for this Software Collection
%global vt191_sitelib %(echo %{python27python_sitelib} | sed 's|%
{scl_python}|%{scl}|')

Summary: Package that installs %scl
Name: %scl_name
Version: 1
Release: 1%{?dist}
License: GPLv2+
BuildRequires: scl-utils-build
# Always make sure that there is the python27-sclbuild (or rh-python35-
sclbuild)
# package in the build root
BuildRequires: %{scl_prefix_python}scldevel
# Require python27-python-devel, you will need macros from that package
BuildRequires: %{scl_prefix_python}python-devel
Requires: %{scl_prefix}python-versiontools

%description
This is the main package for %scl Software Collection.

%package runtime
Summary: Package that handles %scl Software Collection.
Requires: scl-utils
Requires: %{scl_prefix_python}runtime

%description runtime
Package shipping essential scripts to work with %scl Software Collection.

%package build
Summary: Package shipping basic build configuration
Requires: scl-utils-build
# Require python27-scldevel (or rh-python35-scldevel) so that there is
always access
# to the %%scl_python and %%scl_prefix_python macros in builds for this
Software
# Collection
Requires: %{scl_prefix_python}scldevel
```

```
%description build
Package shipping essential configuration macros to build %scl Software
Collection.

%prep
%setup -c -T

%install
%scl_install

# Create the enable scriptlet that:
# - Adds an additional load path for the Python interpreter.
# - Runs scl_source so that you can run:
#     scl enable vt191 "bash"
#   instead of:
#     scl enable python27 vt191 "bash"

cat >> %{buildroot}%{_scl_scripts}/enable << EOF
. scl_source enable %{scl_python}
export PYTHONPATH="%{vt191_sitelib}\${PYTHONPATH:+:\${PYTHONPATH}}"
EOF

mkdir -p %{buildroot}%{vt191_sitelib}

# - Enable Software Collection-specific bytecompilation macros from
#   the python27-python-devel package.
# - Also override the %%python_sitelib macro to point to the vt191
Software
#   Collection.
# - If you have architecture-dependent packages, you will also need to
override
#   the %%python_sitearch macro.

cat >> %{buildroot}%{_root_sysconfdir}/rpm/macros.%{scl}-config << EOF
%%scl_package_override() %%{expand:%{?python27_os_install_post:%%global
__os_install_post %%python27_os_install_post}
%%global __python_requires %%python27_python_requires
%%global __python_provides %%python27_python_provides
%%global __python %python27__python
%%global python_sitelib %vt191_sitelib
%%global python2_sitelib %vt191_sitelib
}
EOF

%files

%files runtime -f filelist
%scl_files
%vt191_sitelib

%files build
%{_root_sysconfdir}/rpm/macros.%{scl}-config
```

```
%changelog
* Wed Jan 22 2014 John Doe <jdoe@example.com> - 1-1
- Initial package.
```

4.2.2. The python-versiontools Package

Below is a commented example of the python-versiontools package spec file. Note the following in the spec file:

- The **BuildRequires** tags are prefixed with **%{?scl_prefix_python}** instead of **%{scl_prefix}**.
- The **%install** section explictly specifies **--install-purelib**.

```
%{?scl:%scl_package python-versiontools}
%{!?scl:%global pkg_name %{name}}

%global pypi_name versiontools

Name:           %{?scl_prefix}python-versiontools
Version:        1.9.1
Release:        1%{?dist}
Summary:        Smart replacement for plain tuple used in __version__

License:        LGPLv3
URL:            https://launchpad.net/versiontools
Source0:
http://pypi.python.org/packages/source/v/versiontools/versiontools-
1.9.1.tar.gz

BuildArch:      noarch
BuildRequires:  %{?scl_prefix_python}python-devel
BuildRequires:  %{?scl_prefix_python}python-setuptools
%{?scl:BuildRequires: %{scl}-build %{scl}-runtime}
%{?scl:Requires: %{scl}-runtime}

%description
Smart replacement for plain tuple used in __version__

%prep
%setup -q -n %{pypi_name}-%{version}

%build
%{?scl:scl enable %{scl} "}
%{__python} setup.py build
%{?scl:"}

%install
# Explicitly specify --install-purelib %{python_sitelib}, which is now
overriden
# to point to vt191, otherwise Python will try to install into the
python27
# Software Collection site-packages directory
%{?scl:scl enable %{scl} "}
%{__python} setup.py install -O1 --skip-build --root %{buildroot} --
```

```
install-purelib %{python_sitelib}
%{?scl:"}

%files
%{python_sitelib}/%{pypi_name}*

%changelog
* Wed Jan 22 2014 John Doe <jdoe@example.com> - 1.9.1-1
- Built for vt191 SCL.
```

4.2.3. Building the vt191 Software Collection

To build the vt191 Software Collection:

1. Install the python27-scldevel and python27-python-devel subpackages that are part of the python27 Software Collection.
2. Build **`vt191.spec`** and install the vt191-runtime and vt191-build packages.
3. Install the python27-python-setuptools package, which is a build requirement for versiontools.
4. Build **`python-versiontools.spec`**.

4.2.4. Testing the vt191 Software Collection

To test the vt191 Software Collection:

1. Install the vt191-python-versiontools package.
2. Run the following command:

   ```
   $ scl enable vt191 "python -c 'import versiontools;
   print(versiontools.__file__)'"
   ```

3. Verify that the output contains the following line:

   ```
   /opt/rh/vt191/root/usr/lib/python2.7/site-
   packages/versiontools/__init__.pyc
   ```

   Note that the provider **`rh`** in the path may vary depending on your redefinition of the **`%_scl_prefix`** macro. See Section 2.3, “The Software Collection Root Directory” for more information.

4.3. EXTENDING THE RH-RUBY23 SOFTWARE COLLECTION

In Red Hat Software Collections 2.4, it is possible to extend the rh-ruby23 Software Collection by adding dependent packages. The Ruby on Rails 4.2 (rh-ror42) Software Collection, which is built on top of Ruby 2.3 provided by the rh-ruby23 Software Collection, is one example of such an extension.

This section provides detailed information about the rh-ror42 metapackage and the rh-ror42-rubygem-bcrypt package, which are both part of the rh-ror42 Software Collection.

4.3.1. The rh-ror42 Software Collection

This section contains a commented example of the Ruby on Rails 4.2 metapackage for the rh-ror42 Software Collection. The rh-ror42 Software Collection depends on the rh-ruby23 Software Collection.

Note the following in the rh-ror42 Software Collection metapackage example:

- The rh-ror42 Software Collection spec file has the following build dependencies set:

```
BuildRequires: %{scl_prefix_ruby}scldevel
BuildRequires: %{scl_prefix_ruby}rubygems-devel
```

  This expands to, for example, rh-ruby23-scldevel and rh-ruby23-rubygems-devel.

  The rh-ruby23-scldevel subpackage contains two important macros, **%scl_ruby** and **%scl_prefix_ruby**. The rh-ruby23-scldevel subpackage should be available in the build root. In case there are multiple Ruby Software Collections available, rh-ruby23-scldevel determines which of the available Software Collections should be used.

  Note that the **%scl_ruby** and **%scl_prefix_ruby** macros are also defined at the top of the spec file. Although the definitions are not required, they provide a visual hint that the rh-ror42 Software Collection has been designed to be built on top of the rh-ruby23 Software Collection. They also serve as a fallback value.

- The rh-ror42-runtime subpackage must depend on the runtime subpackage of the Software Collection it depends on. This dependency is specified as follows:

```
%package runtime
Requires: %{scl_prefix_ruby}runtime
```

  When the package is built against the rh-ruby23 Software Collection, this expands to rh-ruby23-runtime.

- The rh-ror42-build subpackage must depend on the scldevel subpackage of the Software Collection it depends on. This is to ensure that all other packages of this Software Collection will have the same macros defined, thus it is built against the same Ruby version.

```
%package build
Requires: %{scl_prefix_ruby}scldevel
```

  In the case of the rh-ruby23 Software Collection, this expands to rh-ruby23-scldevel.

- The **enable** scriptlet for the rh-ror42 Software Collection contains the following line:

```
. scl_source enable %{scl_ruby}
```

  Note the dot at the beginning of the line. This line makes the Ruby Software Collection start implicitly when the rh-ror42 Software Collection is started so that the user can only type **scl enable rh-ror42 *command*** instead of **scl enable rh-ruby23 rh-ror42 *command*** to run *command* in the Software Collection environment.

- The rh-ror42-scldevel subpackage is provided so that it is available in case you need it to build a Software Collection which extends the rh-ror42 Software Collection. The package provides the **%{scl_ror}** and **%{scl_prefix_ror}** macros, which can be used to extend the rh-ror42 Software Collection.

- Because the rh-ror42 Software Collection's gems are installed in a separate root directory structure, you need to ensure that the correct ownership for the rubygems directories is set. This is done by using a snippet to generate a file list rubygems_filesystem.list.

  You are advised to set the runtime package to own all directories which would, if located in the root file system, be owned by another package. One example of such directories in the case of the rh-ror42 Software Collection is the Rubygem directory structure.

```
%global scl_name_prefix rh-
%global scl_name_base ror
%global scl_name_version 41

%global scl %{scl_name_prefix}%{scl_name_base}%{scl_name_version}

# Fallback to rh-ruby23. rh-ruby23-scldevel is unlikely to be available in
# the build root.
%{!?scl_ruby:%global scl_ruby rh-ruby23}
%{!?scl_prefix_ruby:%global scl_prefix_ruby %{scl_ruby}-}

# Do not produce empty debuginfo package.
%global debug_package %{nil}

# Support SCL over NFS.
%global nfsmountable 1

%{!?install_scl: %global install_scl 1}

%scl_package %scl

Summary: Package that installs %scl
Name: %scl_name
Version: 2.0
Release: 5%{?dist}
License: GPLv2+

%if 0%{?install_scl}
Requires: %{scl_prefix}rubygem-therubyracer
Requires: %{scl_prefix}rubygem-sqlite3
Requires: %{scl_prefix}rubygem-rails
Requires: %{scl_prefix}rubygem-sass-rails
Requires: %{scl_prefix}rubygem-coffee-rails
Requires: %{scl_prefix}rubygem-jquery-rails
Requires: %{scl_prefix}rubygem-sdoc
Requires: %{scl_prefix}rubygem-turbolinks
Requires: %{scl_prefix}rubygem-bcrypt
Requires: %{scl_prefix}rubygem-uglifier
Requires: %{scl_prefix}rubygem-jbuilder
Requires: %{scl_prefix}rubygem-spring
%endif
BuildRequires: help2man
BuildRequires: scl-utils-build
BuildRequires: %{scl_prefix_ruby}scldevel
BuildRequires: %{scl_prefix_ruby}rubygems-devel

%description
This is the main package for %scl Software Collection.
```

```
%package runtime
Summary: Package that handles %scl Software Collection.
Requires: scl-utils
# The enable scriptlet depends on the ruby executable.
Requires: %{scl_prefix_ruby}ruby

%description runtime
Package shipping essential scripts to work with %scl Software Collection.

%package build
Summary: Package shipping basic build configuration
Requires: scl-utils-build
Requires: %{scl_runtime}
Requires: %{scl_prefix_ruby}scldevel

%description build
Package shipping essential configuration macros to build %scl Software
Collection.

%package scldevel
Summary: Package shipping development files for %scl
Provides: scldevel(%{scl_name_base})

%description scldevel
Package shipping development files, especially usefull for development of
packages depending on %scl Software Collection.

%prep
%setup -c -T

%install
%scl_install

cat >> %{buildroot}%{_scl_scripts}/enable << EOF
export PATH="%{_bindir}:%{_sbindir}\${PATH:+:\${PATH}}"
export LD_LIBRARY_PATH="%
{_libdir}\${LD_LIBRARY_PATH:+:\${LD_LIBRARY_PATH}}"
export MANPATH="%{_mandir}:\${MANPATH:-}"
export PKG_CONFIG_PATH="%
{_libdir}/pkgconfig\${PKG_CONFIG_PATH:+:\${PKG_CONFIG_PATH}}"
export GEM_PATH="\${GEM_PATH:=%{gem_dir}:\`scl enable %{scl_ruby} -- ruby
-e "print Gem.path.join(':')"\`}"

. scl_source enable %{scl_ruby}
EOF

cat >> %{buildroot}%{_root_sysconfdir}/rpm/macros.%{scl_name_base}-
scldevel << EOF
%%scl_%{scl_name_base} %{scl}
%%scl_prefix_%{scl_name_base} %{scl_prefix}
EOF


scl enable %{scl_ruby} - << \EOF
set -e
```

```
# Fake rh-ror42 Software Collection environment.
GEM_PATH=%{gem_dir}:`ruby -e "print Gem.path.join(':')"` \
X_SCLS=%{scl} \
ruby -rfileutils > rubygems_filesystem.list << \EOR
  # Create the RubyGems file system.
  Gem.ensure_gem_subdirectories '%{buildroot}%{gem_dir}'
  FileUtils.mkdir_p File.join '%{buildroot}', Gem.default_ext_dir_for('%
{gem_dir}')

  # Output the relevant directories.
  Gem.default_dirs['%{scl}_system'.to_sym].each { |k, p| puts p }
EOR
EOF

%files

%files runtime -f rubygems_filesystem.list
%scl_files

%files build
%{_root_sysconfdir}/rpm/macros.%{scl}-config

%files scldevel
%{_root_sysconfdir}/rpm/macros.%{scl_name_base}-scldevel


%changelog
* Thu Jan 16 2015 John Doe <jdoe@example.com> - 1-1
- Initial package.
```

4.3.2. The rh-ror42-rubygem-bcrypt Package

Below is a commented example of the rh-ror42-rubygem-bcrypt package spec file. This package provides the bcrypt Ruby gem. For more information on bcrypt, see the following website:

- http://rubygems.org/gems/bcrypt-ruby

Note that the only significant difference between the rh-ror42-rubygem-bcrypt package spec file and a normal Software Collection package spec file is the following:

- The **BuildRequires** tags are prefixed with **%{?scl_prefix_ruby}** instead of **%{scl_prefix}**.

```
%{?scl:%scl_package rubygem-%{gem_name}}
%{!?scl:%global pkg_name %{name}}

%global gem_name bcrypt

Summary: Wrapper around bcrypt() password hashing algorithm
Name: %{?scl_prefix}rubygem-%{gem_name}
Version: 3.1.9
Release: 2%{?dist}
Group: Development/Languages
# ext/* - Public Domain
```

```
# spec/TestBCrypt.java - ISC
License: MIT and Public Domain and ISC
URL: https://github.com/codahale/bcrypt-ruby
Source0: http://rubygems.org/downloads/%{gem_name}-%{version}.gem
Requires: %{?scl_prefix_ruby}ruby(release)
Requires: %{?scl_prefix_ruby}ruby(rubygems)
BuildRequires: %{?scl_prefix_ruby}rubygems-devel
BuildRequires: %{?scl_prefix_ruby}ruby-devel
BuildRequires: %{?scl_prefix}rubygem(rspec)
Provides: %{?scl_prefix}rubygem(bcrypt) = %{version}

%description
bcrypt() is a sophisticated and secure hash algorithm designed by The
OpenBSD project for hashing passwords. bcrypt provides a simple,
humane wrapper for safely handling passwords.

%package doc
Summary: Documentation for %{pkg_name}
Group: Documentation
Requires: %{?scl_prefix}%{pkg_name} = %{version}-%{release}

%description doc
Documentation for %{pkg_name}.

%prep
%setup -n %{pkg_name}-%{version} -q -c -T
%{?scl:scl enable %{scl} - << \EOF}
%gem_install -n %{SOURCE0}
%{?scl:EOF}

%build

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

mkdir -p %{buildroot}%{gem_extdir_mri}
cp -pa .%{gem_extdir_mri}/* %{buildroot}%{gem_extdir_mri}/

# Prevent a symlink with an invalid target in -debuginfo (BZ#878863).
rm -rf %{buildroot}%{gem_instdir}/ext/

%check
%{?scl:scl enable %{scl} - << \EOF}
pushd .%{gem_instdir}
# 2 failutes due to old RSpec
# https://github.com/rspec/rspec-expectations/pull/284
rspec -I$(dirs +1)%{gem_extdir_mri} spec |grep '34 examples, 2 failures'
|| exit 1
popd
%{?scl:EOF}

%files
%dir %{gem_instdir}
%exclude %{gem_instdir}/.*
```

```
%{gem_libdir}
%{gem_extdir_mri}
%exclude %{gem_cache}
%{gem_spec}
%doc %{gem_instdir}/COPYING

%files doc
%doc %{gem_docdir}
%doc %{gem_instdir}/README.md
%doc %{gem_instdir}/CHANGELOG
%{gem_instdir}/Rakefile
%{gem_instdir}/Gemfile*
%{gem_instdir}/%{gem_name}.gemspec
%{gem_instdir}/spec


%changelog
* Fri Mar 21 2015 John Doe <jdoe@example.com> - 3.1.2-4
- Initial package.
```

4.3.3. Building the rh-ror42 Software Collection

To build the rh-ror42 Software Collection:

1. Install the rh-ruby23-scldevel subpackage which is a part of the rh-ruby23 Software Collection.
2. Build **`rh-ror42.spec`** and install the ror42-runtime and ror42-build packages.
3. Build **`rubygem-bcrypt.spec`**.

4.3.4. Testing the rh-ror42 Software Collection

To test the rh-ror42 Software Collection:

1. Install the rh-ror42-rubygem-bcrypt package.
2. Run the following command:

```
$ scl enable rh-ror42 -- ruby -r bcrypt -e "puts
BCrypt::Password.create('my password')"
```

3. Verify that the output contains the following line:

```
$2a$10$s./ReniLY.wXPHVBQ9npoeyZf5KzywfpvI5lhjG6Ams3u0hKqwVbW
```

4.4. EXTENDING THE RH-PERL524 SOFTWARE COLLECTION

This section describes extending the rh-perl524 Software Collection by building your own dependent Software Collection.

IMPORTANT

Examples described in this section only work as expected when extending the rh-perl524 Software Collection with packages that:

- do not provide any Perl modules, and
- only depend on Perl modules provided by the rh-perl524 Software Collection.

4.4.1. The h2m144 Software Collection

This section contains a commented example of a dependent Software Collection's metapackage. The dependent Software Collection is named h2m144 and contains the help2man Perl package version 1.44.1. The h2m144 Software Collection depends on the rh-perl524 Software Collection.

Note the following in the h2m144 Software Collection metapackage:

- The h2m144 Software Collection metapackage has the following build dependency set:

  ```
  BuildRequires: %{scl_prefix_perl}scldevel
  ```

  This expands to rh-perl524-scldevel.

  The rh-perl524-scldevel subpackage contains two important macros, **%scl_perl** and **%scl_prefix_perl**, and also provides Perl dependency generators. Note that the macros are defined at the top of the metapackage spec file. Although the definitions are not required, they provide a visual hint that the h2m144 Software Collection has been designed to be built on top of the rh-perl524 Software Collection. They also serve as a fallback value.

- The h2m144-build subpackage has the following dependency set:

  ```
  Requires: %{scl_prefix_perl}scldevel
  ```

  This expands to rh-perl524-scldevel. The purpose of this dependency is to ensure that the macros and dependency generators are always present when building packages for the h2m144 Software Collection.

- The **enable** scriptlet for the h2m144 Software Collection contains the following line:

  ```
  . scl_source enable %{scl_perl}
  ```

  Note the dot at the beginning of the line. This line makes the Perl Software Collection start implicitly when the h2m144 Software Collection is started so that the user can only type **scl enable h2m144 *command*** instead of **scl enable rh-perl524 h2m144 *command*** to run *command* in the Software Collection environment.

- The macro file **macros.h2m144-config** calls the Perl dependency generators, and certain Perl-specific macros used in other packages' spec files.

```
%global scl h2m144
%scl_package %scl

# Default values for the rh-perl524 Software Collection. These
# will be used when rh-perl524-scldevel is not in the build root.
```

```
%{!?scl_perl:%global scl_perl rh-perl524}
%{!?scl_prefix_perl:%global scl_prefix_perl %{scl_perl}-}

# Only for this build, override __perl_requires for the automatic
dependency
# generator.
%global __perl_requires /usr/lib/rpm/perl.req.stack

Summary: Package that installs %scl
Name:    %scl_name
Version: 1
Release: 1%{?dist}
License: GPLv2+
BuildRequires: scl-utils-build
# Always make sure that there is the rh-perl524-scldevel
# package in the build root.
BuildRequires: %{scl_prefix_perl}scldevel
# Require rh-perl524-perl-macros; you will need macros from that package.
BuildRequires: %{scl_prefix_perl}perl-macros
Requires: %{scl_prefix}help2man

%description
This is the main package for %scl Software Collection.

%package runtime
Summary: Package that handles %scl Software Collection.
Requires: scl-utils
Requires: %{scl_prefix_perl}runtime

%description runtime
Package shipping essential scripts to work with %scl Software Collection.

%package build
Summary: Package shipping basic build configuration
Requires: scl-utils-build
# Require rh-perl524-scldevel so that there is always access to the
%%scl_perl
# and %%scl_prefix_perl macros in builds for this Software Collection.
Requires: %{scl_prefix_perl}scldevel

%description build
Package shipping essential configuration macros to build %scl Software
Collection.

%prep
%setup -c -T

%build

%install
%scl_install

# Create the enable scriptlet that:
# - Adds an additional load path for the Perl interpreter.
# - Runs scl_source so that you can run:
#     scl enable h2m144 'bash'
```

```
#   instead of:
#     scl enable rh-perl524 h2m144 'bash'

cat >> %{buildroot}%{_scl_scripts}/enable << EOF
. scl_source enable %{scl_perl}
export PATH="%{_bindir}:%{_sbindir}\${PATH:+:\${PATH}}"
export MANPATH="%{_mandir}:\${MANPATH:-}"
EOF

cat >> %{buildroot}%{_root_sysconfdir}/rpm/macros.%{scl}-config << EOF
%%scl_package_override() %%{expand:%%global __perl_requires
/usr/lib/rpm/perl.req.stack
%%global __perl_provides /usr/lib/rpm/perl.prov.stack
%%global __perl %{_scl_prefix}/%{scl_perl}/root/usr/bin/perl
}
EOF

%files

%files runtime -f filelist
%scl_files

%files build
%{_root_sysconfdir}/rpm/macros.%{scl}-config

%changelog
* Tue Apr 22 2014 John Doe <jdoe@example.com> - 1-1
- Initial package.
```

4.4.2. The help2man Package

Below is a commented example of the help2man package spec file. Note the following in the spec file:

- The **BuildRequires** tags are prefixed with **%{?scl_prefix_perl}** instead of **%{scl_prefix}**.

```
%{?scl:%scl_package help2man}
%{!?scl:%global pkg_name %{name}}

# Supported build option:
#
# --with nls ... build this package with --enable-nls
%bcond_with nls

Name:           %{?scl_prefix}help2man
Summary:        Create simple man pages from --help output
Version:        1.44.1
Release:        1%{?dist}
Group:          Development/Tools
License:        GPLv3+
URL:            http://www.gnu.org/software/help2man
Source:         ftp://ftp.gnu.org/gnu/help2man/help2man-%{version}.tar.xz
%{!?with_nls:BuildArch: noarch}

BuildRequires:  %{?scl_prefix_perl}perl(Getopt::Long)
```

```
BuildRequires:  %{?scl_prefix_perl}perl(POSIX)
BuildRequires:  %{?scl_prefix_perl}perl(Text::ParseWords)
BuildRequires:  %{?scl_prefix_perl}perl(Text::Tabs)
BuildRequires:  %{?scl_prefix_perl}perl(strict)
%{?with_nls:BuildRequires: %{?scl_prefix_perl}perl(Locale::gettext)
/usr/bin/msgfmt}
%{?with_nls:BuildRequires: %{?scl_prefix_perl}perl(Encode)}
%{?with_nls:BuildRequires: %{?scl_prefix_perl}perl(I18N::Langinfo)}
Requires:   %{?scl_prefix_perl}perl(:MODULE_COMPAT_%(%{?scl:scl enable %
{scl_perl} '}eval "`perl -V:version`"; echo $version%{?scl:'}))

Requires(post): /sbin/install-info
Requires(preun): /sbin/install-info

%description
help2man is a script to create simple man pages from the --help and
--version output of programs.

Since most GNU documentation is now in info format, this provides a
way to generate a placeholder man page pointing to that resource while
still providing some useful information.

%prep
%setup -q -n help2man-%{version}

%build
%configure --%{!?with_nls:disable}%{?with_nls:enable}-nls --libdir=%
{_libdir}/help2man
%{?scl:scl enable %{scl} "}
make %{?_smp_mflags}
%{?scl:"}

%install
%{?scl:scl enable %{scl} "}
make install_l10n DESTDIR=$RPM_BUILD_ROOT
%{?scl:"}
%{?scl:scl enable %{scl} "}
make install DESTDIR=$RPM_BUILD_ROOT
%{?scl:"}
%find_lang %pkg_name --with-man

%post
/sbin/install-info %{_infodir}/help2man.info %{_infodir}/dir 2>/dev/null ||
:

%preun
if [ $1 -eq 0 ]; then
  /sbin/install-info --delete %{_infodir}/help2man.info \
    %{_infodir}/dir 2>/dev/null || :
fi

%files -f %pkg_name.lang
%doc README NEWS THANKS COPYING
%{_bindir}/help2man
%{_infodir}/*
%{_mandir}/man1/*
```

```
%if %{with nls}
%{_libdir}/help2man
%endif

%changelog
* Tue Apr 22 2014 John Doe <jdoe@example.com> - 1.44.1-1
- Built for h2m144 SCL.
```

4.4.3. Building the h2m144 Software Collection

To build the h2m144 Software Collection:

1. Install the rh-perl524-scldevel and rh-perl524-perl-macros packages that are part of the perl524 Software Collection.
2. Build h2m144.spec and install the h2m144-runtime and h2m144-build packages.
3. Install the rh-perl524-perl, rh-perl524-perl-Text-ParseWords and rh-perl524-perl-Getopt-Long packages, which are all build requirements for help2man.
4. Build **`help2man.spec`**.

4.4.4. Testing the h2m144 Software Collection

To test the h2m144 Software Collection:

1. Install the h2m144-help2man package.
2. Run the following command:

   ```
   $ scl enable h2m144 'help2man bash'
   ```

3. Verify that the output is similar to the following lines:

   ```
   .\" DO NOT MODIFY THIS FILE!  It was generated by help2man 1.44.1.
   .TH BASH, "1" "April 2014" "bash, version 4.1.2(1)-release (x86_64-
   redhat-linux-gnu)" "User Commands"
   .SH NAME
   bash, \- manual page for bash, version 4.1.2(1)-release (x86_64-
   redhat-linux-gnu)
   .SH SYNOPSIS
   .B bash
   [\fIGNU long option\fR] [\fIoption\fR] ...
   .SH DESCRIPTION
   GNU bash, version 4.1.2(1)\-release\-(x86_64\-redhat\-linux\-gnu)
   .IP
   bash [GNU long option] [option] script\-file ...
   .SS "GNU long options:"
   .HP
   \fB\-\-debug\fR
   ```

CHAPTER 5. TROUBLESHOOTING SOFTWARE COLLECTIONS

This chapter helps you troubleshoot some of the common issues you can encounter when building your Software Collections.

5.1. ERROR: LINE *XX*: UNKNOWN TAG: %SCL_PACKAGE *SOFTWARE_COLLECTION_NAME*

You can encounter this error message when building a Software Collection package. It is usually caused by a missing package scl-utils-build. To install the scl-utils-build package, run the following command:

```
# yum install scl-utils-build
```

For more information, see Section 1.3, “Enabling Support for Software Collections”.

5.2. SCL COMMAND DOES NOT EXIST

This error message is usually caused by a missing package scl-utils. To install the scl-utils package, run the following command:

```
# yum install scl-utils
```

For more information, see Section 1.3, “Enabling Support for Software Collections”.

5.3. UNABLE TO OPEN /ETC/SCL/PREFIXES/*SOFTWARE_COLLECTION_NAME*

This error message can be caused by using incorrect arguments with the **scl** command you are calling. Check the **scl** command is correct and that you have not mistyped any of the arguments.

The same error message can also be caused by a missing Software Collection. Ensure that the *software_collection_name* Software Collection is properly installed on the system. For more information, see Section 1.5, “Listing Installed Software Collections”.

5.4. SCL_SOURCE: COMMAND NOT FOUND

This error message is usually caused by having an old version of the scl-utils package installed. To update the scl-utils package, run the following command:

```
# yum update scl-utils
```

APPENDIX A. GETTING MORE INFORMATION

For more information on Software Collection packaging, Red Hat Developers, the Red Hat Software Collections and Red Hat Developer Toolset offerings, and Red Hat Enterprise Linux, see the resources listed below.

A.1. RED HAT DEVELOPERS

- Overview of Red Hat Software Collections on Red Hat Developers – The *Red Hat Developers* portal provides a number of tutorials to get you started with developing code using different development technologies. This includes the Node.js, Perl, PHP, Python, and Ruby Software Collection.
- Red Hat Enterprise Linux Developer Program – The *Red Hat Enterprise Linux Developer Program* delivers industry-leading developer tools, instructional resources, and an ecosystem of experts to help programmers maximize productivity in building Linux applications.
- Red Hat Developer Blog – The *Red Hat Developer Blog* contains up-to-date information, best practices, opinion, product and program announcements as well as pointers to sample code and other resources for those who are designing and developing applications based on Red Hat technologies.

A.2. INSTALLED DOCUMENTATION

- **scl**(1) – The man page for the **scl** tool for enabling Software Collections and running programs in Software Collection's environment.
- **scl --help** – General usage information for the **scl** tool for enabling Software Collections and running programs in Software Collection's environment.
- **rpmbuild**(8) – The man page for the **rpmbuild** utility for building both binary and source packages.

A.3. ACCESSING RED HAT DOCUMENTATION

Red Hat Product Documentation located at https://access.redhat.com/documentation/ serves as a central source of information. It is currently translated in 22 languages and for each product, it provides different kinds of books from release and technical notes to installation, user, and reference guides in HTML, PDF, and EPUB formats.

The following is a brief list of documents that are directly or indirectly relevant to this book:

- Red Hat Software Collections 2.4 Release Notes – The *Release Notes* for Red Hat Software Collections 2.4 document the major features and contains other information about Red Hat Software Collections, a Red Hat offering that provides a set of dynamic programming languages, database servers, and various related packages.
- Red Hat Developer Toolset 6.1 User Guide – The *User Guide* for Red Hat Developer Toolset 6.1 contains information about Red Hat Developer Toolset, a Red Hat offering for developers on the Red Hat Enterprise Linux platform. Using Software Collections, Red Hat Developer Toolset provides current versions of the **GCC** compiler, **GDB** debugger and other binary utilities.
- Using Red Hat Software Collections Container Images – This article provides information on how to use container images based on Red Hat Software Collections. The available container images include applications, daemons, and databases. The images can be run on Red Hat Enterprise

Linux 7 Server and Red Hat Enterprise Linux Atomic Host.

- Red Hat Enterprise Linux 7 Developer Guide – The *Developer Guide* for Red Hat Enterprise Linux 7 provides detailed description of Red Hat Developer Toolset features, as well as an introduction to Red Hat Software Collections, and information on libraries and runtime support, compiling and building, debugging, and profiling.

- Red Hat Enterprise Linux 7 System Administrator's Guide – The *System Administrator's Guide* for Red Hat Enterprise Linux 7 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 7.

- Red Hat Enterprise Linux 6 Developer Guide – The *Developer Guide* for Red Hat Enterprise Linux 6 provides detailed description of Red Hat Developer Toolset features, as well as an introduction to Red Hat Software Collections, and information on libraries and runtime support, compiling and building, debugging, and profiling.

- Red Hat Enterprise Linux 6 Deployment Guide – The *Deployment Guide* for Red Hat Enterprise Linux 6 documents relevant information regarding the deployment, configuration, and administration of Red Hat Enterprise Linux 6.

APPENDIX B. REVISION HISTORY

Revision 3.10-0 **Mon Jun 5 2017** **Petr Kovář**
Republished to fix BZ#1458821.

Revision 3.9-0 **Thu Apr 20 2017** **Petr Kovář**
Red Hat Software Collections 2.4 release of the Packaging Guide.

Revision 3.8-0 **Wed Apr 05 2017** **Petr Kovář**
Red Hat Software Collections 2.4 Beta release of the Packaging Guide.

Revision 3.7-0 **Wed Jan 25 2017** **Petr Kovář**
Republished to fix BZ#1263733.

Revision 3.6-0 **Wed Nov 02 2016** **Petr Kovář**
Red Hat Software Collections 2.3 release of the Packaging Guide.

Revision 3.5-0 **Wed Oct 12 2016** **Petr Kovář**
Red Hat Software Collections 2.3 Beta release of the Packaging Guide.

Revision 3.4-0 **Mon May 23 2016** **Petr Kovář**
Red Hat Software Collections 2.2 release of the Packaging Guide.

Revision 3.3-0 **Tue Apr 26 2016** **Petr Kovář**
Red Hat Software Collections 2.2 Beta release of the Packaging Guide.

Revision 3.2-0 **Wed Nov 04 2015** **Petr Kovář**
Red Hat Software Collections 2.1 release of the Packaging Guide.

Revision 3.1-0 **Tue Oct 06 2015** **Petr Kovář**
Red Hat Software Collections 2.1 Beta release of the Packaging Guide.

Revision 3.0-2 **Tue May 19 2015** **Petr Kovář**
Red Hat Software Collections 2.0 release of the Packaging Guide.

Revision 3.0-1 **Wed Apr 22 2015** **Petr Kovář**
Red Hat Software Collections 2.0 Beta release of the Packaging Guide.

Revision 2.2-4 **Fri Nov 21 2014** **Petr Kovář**
Republished to fix BZ#1150573, BZ#1022023, and BZ#1149650.

Revision 2.2-2 **Thu Oct 30 2014** **Petr Kovář**
Red Hat Software Collections 1.2 release of the Packaging Guide.

Revision 2.2-1 **Tue Oct 07 2014** **Petr Kovář**
Red Hat Software Collections 1.2 Beta refresh release of the Packaging Guide.

Revision 2.2-0 **Tue Sep 09 2014** **Petr Kovář**
The Software Collections Guide renamed to Packaging Guide.
Red Hat Software Collections 1.2 Beta release of the Packaging Guide.

Revision 2.1-29 **Wed Jun 04 2014** **Petr Kovář**
Red Hat Software Collections 1.1 release of the Software Collections Guide.

Revision 2.1-21 **Thu Mar 20 2014** **Petr Kovář**
Red Hat Software Collections 1.1 Beta release of the Software Collections Guide.

Revision 2.1-18 **Tue Mar 11 2014** **Petr Kovář**
Red Hat Developer Toolset 2.1 release of the Software Collections Guide.

Revision 2.1-8 **Tue Feb 11 2014** **Petr Kovář**
Red Hat Developer Toolset 2.1 Beta release of the Software Collections Guide.

Revision 2.0-12 **Tue Sep 10 2013** **Petr Kovář**
Red Hat Developer Toolset 2.0 release of the Software Collections Guide.

Revision 2.0-8 **Tue Aug 06 2013** **Petr Kovář**
Red Hat Developer Toolset 2.0 Beta-2 release of the Software Collections Guide.

Revision 2.0-3 **Tue May 28 2013** **Petr Kovář**
Red Hat Developer Toolset 2.0 Beta-1 release of the Software Collections Guide.

Revision 1.0-2 **Tue Apr 23 2013** **Petr Kovář**
Republished to fix BZ#949000.

Revision 1.0-1 **Tue Jan 22 2013** **Petr Kovář**
Red Hat Developer Toolset 1.1 release of the Software Collections Guide.

Revision 1.0-2 **Thu Nov 08 2012** **Petr Kovář**
Red Hat Developer Toolset 1.1 Beta-2 release of the Software Collections Guide.

Revision 1.0-1 **Wed Oct 10 2012** **Petr Kovář**
Red Hat Developer Toolset 1.1 Beta-1 release of the Software Collections Guide.

Revision 1.0-0 **Tue Jun 26 2012** **Petr Kovář**
Red Hat Developer Toolset 1.0 release of the Software Collections Guide.

Revision 0.0-2 **Tue Apr 10 2012** **Petr Kovář**
Red Hat Developer Toolset 1.0 Alpha-2 release of the Software Collections Guide.

Revision 0.0-1 **Tue Mar 06 2012** **Petr Kovář**
Red Hat Developer Toolset 1.0 Alpha-1 release of the Software Collections Guide.

B.1. ACKNOWLEDGMENTS

The author of this book would like to thank the following people for their valuable contributions: Jindřich Nový, Marcela Mašláňová, Bohuslav Kabrda, Honza Horák, Jan Zelený, Martin Čermák, Jitka Plesníková, Langdon White, Florian Nadge, Stephen Wadeley, Douglas Silas, Tomáš Čapek, and Vít Ondruch, among many others.