Red Hat Enterprise Linux 7

RPM 패키징 가이드

RPM 패키지 관리자를 사용한 기본 및 고급 소프트웨어 패키지 시나리오

Last Updated: 2024-09-02

Red Hat Enterprise Linux 7 RPM 패키징 가이드

RPM 패키지 관리자를 사용한 기본 및 고급 소프트웨어 패키지 시나리오

Marie Doleželová
Red Hat Customer Content Services
mdolezel@redhat.com

Maxim Svistunov
Red Hat Customer Content Services

Adam Miller
Red Hat

Adam Kvítek
Red Hat Customer Content Services

Petr Kovář
Red Hat Customer Content Services

Miroslav Suchý
Red Hat

Customer Content Services rhel-notes@redhat.com

법적 공지

초록

RPM 패키징 가이드 문서는 소프트웨어 패키징 문서를 RPM에 추가합니다. 또한 패키징을 위해 소스 코드를 준비하는 방법도 설명합니다. 마지막으로 가이드에서는 선택한 고급 패키지 시나리오를 설명합니다.

차례

1장. RPM 패키지 시작하기

다음 섹션에서는 RPM 패키징 개념과 주요 이점에 대해 설명합니다.

1.1. RPM 패키지 소개

RPM(RPM)은 RHEL, CentOS 및 Fedora에서 실행되는 패키지 관리 시스템입니다. RPM을 사용하여 위에서 언급한 모든 운영 체제에 대해 생성한 소프트웨어를 배포, 관리 및 업데이트할 수 있습니다.

1.2. RPM 이점

RPM 패키지 관리 시스템은 기존 아카이브 파일에 소프트웨어를 배포하는 것보다 몇 가지 이점을 제공합니다.

RPM을 사용하면 다음을 수행할 수 있습니다.

- DestinationRule 또는 PackageKit과 같은 표준 패키지 관리 툴을 사용하여 패키지를 설치, 다시 설치, 제거, 업그레이드 및 확인합니다.
- 설치된 패키지의 데이터베이스를 사용하여 패키지를 쿼리하고 확인합니다.
- 메타데이터를 사용하여 패키지, 설치 명령 및 기타 패키지 매개 변수를 설명합니다.
- 패키지 소프트웨어 소스, 패치 및 빌드 명령을 소스 및 바이너리 패키지로 완료합니다.
- 4.6.1 리포지토리에 **패키지를** 추가합니다.
- GNU 개인 정보 보호 ECDHE (GPG) 서명 키를 사용하여 패키지에 디지털 서명합니다.

1.3. 첫 번째 RPM 패키지 만들기

RPM 패키지를 만드는 것은 복잡할 수 있습니다. 다음은 건너뛰고 단순화된 몇 가지 작업이 포함된 RPM 사양 파일을 완전히 작동하는 것입니다.

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

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

%prep
# we have no source, so nothing here

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

%install
mkdir -p %{buildroot}/usr/bin/
```

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

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

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

이 파일을 **hello-world.spec** 로 저장합니다.

이제 다음 명령을 사용합니다.

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

rpmdev-setuptree 명령은 여러 작업 디렉터리를 생성합니다. 이러한 디렉터리는 $HOME에 영구적으로 저장되므로 이 명령을 다시 사용할 필요가 없습니다.

rpmbuild 명령은 실제 rpm 패키지를 생성합니다. 이 명령의 출력은 다음과 유사할 수 있습니다.

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

/home/<username>/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm 파일은 첫 번째 RPM 패키지입니다. 시스템에 설치하고 테스트할 수 있습니다.

2장. RPM 패키지용 소프트웨어 준비

이 섹션에서는 RPM 패키징용 소프트웨어를 준비하는 방법에 대해 설명합니다. 이렇게하려면 코드가 필요하지 않습니다. 그러나 What source code is and how programs are made 와 같은 기본 개념을 이해해야 합니다.

2.1. 소스 코드는 무엇입니까?

이 부분에서는 소스 코드가 무엇인지 설명하고 세 가지 다른 프로그래밍 언어로 작성된 프로그램의 소스 코드를 보여줍니다.

소스 코드는 컴퓨터에 대해 사람이 읽을 수 있는 명령이며 계산을 수행하는 방법을 설명합니다. 소스 코드는 프로그래밍 언어를 사용하여 표현됩니다.

2.1.1. 소스 코드 예

이 문서에는 세 가지 다른 프로그래밍 언어로 작성된 **Hello World** 프로그램의 세 가지 버전이 포함되어 있습니다.

- 2.1.1.1절. “helloworld는 bash로 작성된 것입니다.”
- 2.1.1.2절. “helloworld는 Python으로 작성된 것입니다.”
- 2.1.1.3절. “C로 작성된 hello World”

각 버전은 다르게 패키지됩니다.

이러한 버전의 **Hello World** 프로그램은 RPM 패키지 관리자의 세 가지 주요 사용 사례를 다룹니다.

2.1.1.1. helloworld는 bash로 작성된 것입니다.

bello 프로젝트는 bash 에서 **Hello World** 를 구현합니다. 구현에는 **bello** 쉘 스크립트만 포함됩니다. 프로그램의 목적은 명령줄에서 **Hello World** 를 출력하는 것입니다.

bello 파일에는 다음 구문이 있습니다.

```
#!/bin/bash

printf "Hello World\n"
```

2.1.1.2. helloworld는 Python으로 작성된 것입니다.

pello 프로젝트는 Python 에서 **Hello World** 를 구현합니다. 구현에는 **pello.py** 프로그램만 포함됩니다. 프로그램의 목적은 명령줄에서 **Hello World** 를 출력하는 것입니다.

pello.py 파일에는 다음 구문이 있습니다.

```
#!/usr/bin/python3

print("Hello World")
```

2.1.1.3. C로 작성된 hello World

cello 프로젝트는 C에서 **Hello World** 를 구현합니다. 구현에는 **cello.c** 및 **Makefile** 파일만 포함되므로 결과 **tar.gz** 아카이브에는 **LICENSE** 파일과 별도로 두 개의 파일이 있습니다.

프로그램의 목적은 명령줄에서 **Hello World** 를 출력하는 것입니다.

cello.c 파일에는 다음과 같은 구문이 있습니다.

```
#include <stdio.h>

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

2.2. 프로그램 작성 방법

사람이 읽을 수 있는 소스 코드에서 머신 코드로 변환하는 방법(시스템이 프로그램을 실행하기 위해 따르는 구조)에는 다음이 포함됩니다.

- 이 프로그램은 기본적으로 컴파일됩니다.
- 이 프로그램은 raw interpreting으로 해석됩니다.
- 이 프로그램은 바이트 컴파일에 의해 해석됩니다.

2.2.1. 기본적으로 컴파일된 코드

기본적으로 컴파일된 소프트웨어는 결과 바이너리 실행 파일을 사용하여 머신 코드로 컴파일되는 프로그래밍 언어로 작성된 소프트웨어입니다. 이러한 소프트웨어는 단독으로 실행할 수 있습니다.

이러한 방식으로 구축된 RPM 패키지는 아키텍처에 따라 다릅니다.

64비트(x86_64) AMD 또는 Intel 프로세서를 사용하는 컴퓨터에서 이러한 소프트웨어를 컴파일하는 경우 32비트(x86) AMD 또는 Intel 프로세서에서 실행되지 않습니다. 결과 패키지에는 이름에 지정된 아키텍처가 있습니다.

2.2.2. 해석된 코드

bash 또는 Python 과 같은 일부 프로그래밍 언어는 머신 코드로 컴파일되지 않습니다. 대신, 프로그램의 소스 코드는 Language Interpreter 또는 Language Virtual Machine에 의해 사전 변환 없이 단계별로 실행됩니다.

전적으로 해석된 프로그래밍 언어로 작성된 소프트웨어는 아키텍처에 따라 다릅니다. 따라서 생성되는 RPM 패키지에는 이름에 **noarch** 문자열이 있습니다.

해석된 언어는 원시 해석 프로그램 또는 Byte-ECDHE 프로그램입니다. 이 두 가지 유형은 프로그램 빌드 프로세스 및 패키징 절차에서 다릅니다.

2.2.2.1. 원시 연결 프로그램

원시 해석 언어 프로그램은 컴파일할 필요가 없으며 인터프리터에 의해 직접 실행됩니다.

2.2.2.2. byte-ECDHE 프로그램

바이트 단위 언어를 바이트 코드로 컴파일해야 하며, 언어 가상 머신에서 실행됩니다.

참고

일부 언어에서는 옵션을 제공합니다: 원시 해석 또는 바이트로 사용할 수 있습니다.

2.3. 소스에서 소프트웨어 빌드

이 부분에서는 소스 코드에서 소프트웨어를 빌드하는 방법을 설명합니다.

컴파일된 언어로 작성된 소프트웨어의 경우 소스 코드는 빌드 프로세스를 통해 머신 코드를 생성합니다. 일반적으로 컴파일 또는 번역이라고 하는 이 프로세스는 언어에 따라 다릅니다. 결과적으로 구축된 소프트웨어를 실행할 수 있으므로 컴퓨터가 지정한 작업을 수행할 수 있습니다.

원시 해석 언어로 작성된 소프트웨어의 경우 소스 코드는 빌드되지 않지만 직접 실행됩니다.

바이트-ECDHE 해석 언어로 작성된 소프트웨어의 경우 소스 코드는 바이트 코드로 컴파일되며, 언어 가상 머신에서 실행됩니다.

2.3.1. 기본적으로 컴파일된 코드

이 섹션에서는 C 언어로 작성된 **cello.c** 프로그램을 실행 파일로 빌드하는 방법을 보여줍니다.

cello.c

```
#include <stdio.h>

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

2.3.1.1. 수동 빌드

cello.c 프로그램을 수동으로 빌드하려면 다음 절차를 사용하십시오.

절차

1. GNU 컴파일러에서 C 컴파일러를 호출하여 소스 코드를 바이너리로 컴파일합니다.

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

2. 결과 출력 바이너리 셀오를 실행합니다.

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

2.3.1.2. 자동화된 빌드

대규모 소프트웨어는 일반적으로 Makefile 파일을 만든 다음 GNU make 유틸리티를 실행하여 수행되는 자동화된 빌드를 사용합니다.

cello.c 프로그램을 구축하기 위해 자동화된 빌딩을 사용하려면 다음 절차를 사용하십시오.

절차

1. 자동 빌드를 설정하려면 **cello.c** 와 동일한 디렉터리에 다음 콘텐츠를 사용하여 **Makefile** 파일을 만듭니다.

Makefile

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

cello: 및 **clean:** 아래의 행은 탭으로 시작해야 합니다.

2. 소프트웨어를 빌드하려면 **make** 명령을 실행합니다.

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

3. 이미 사용 가능한 빌드가 있으므로 **make clean** 명령을 실행한 후 **make** 명령을 다시 실행합니다.

```
$ make clean
rm cello

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

참고

다른 빌드 후에 프로그램을 빌드하려고 하면 효과가 없습니다.

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

4. 프로그램을 실행합니다.

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

이제 프로그램을 수동으로 컴파일하고 빌드 도구를 사용합니다.

2.3.2. 코드 해석

이 섹션에서는 Python 으로 작성된 프로그램 및 bash 로 작성된 프로그램을 바이트 컴파일하는 방법을 보여줍니다.

참고

아래 두 가지 예에서 파일 맨 위에 있는 #! 행은 행 이라고 하며 프로그래밍 언어 소스 코드의 일부가 아닙니다.

he bang 을 사용하면 텍스트 파일을 실행 파일로 사용할 수 있습니다. 시스템 프로그램 로더는 바운드 가 포함된 행을 구문 분석하여 바이너리 실행 파일의 경로를 가져온 다음 프로그래밍 언어 인터프리터로 사용됩니다. 기능을 사용하려면 텍스트 파일이 실행 파일로 표시됩니다.

2.3.2.1. 바이트 컴파일 코드

이 섹션에서는 Python으로 작성된 pello.py 프로그램을 바이트 코드로 컴파일하는 방법을 보여주고 Python 언어 가상 머신에서 실행합니다.

Python 소스 코드도 원시 해석할 수 있지만 바이트-ECDHE 버전이 더 빠릅니다. 따라서 RPM Packagers는 최종 사용자에게 배포하기 위해 바이트-ECDHE 버전을 패키징하는 것을 선호합니다.

pello.py

```
#!/usr/bin/python3

print("Hello World")
```

바이트 컴파일 프로그램 절차는 다음 요인에 따라 다릅니다.

- 프로그래밍 언어
- 언어의 가상 머신
- 해당 언어와 함께 사용되는 툴 및 프로세스

참고

Python 은 종종 바이트-ECDHE이지만 여기에 설명된 방식으로는 아닙니다. 다음 절차는 커뮤니티 표준을 준수하는 것이 아니라 단순함을 목표로 합니다. 실제 Python 지침은 소프트웨어 패키징 및 배포를 참조하십시오.

pello.py 를 바이트 코드로 컴파일하려면 이 절차를 사용하십시오.

절차

1. byte-compile pello.py 파일:

```
$ python -m compileall pello.py

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

2. pello.pyc 에서 바이트 코드를 실행합니다.

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

2.3.2.2. Raw-interpreting 코드

이 섹션에서는 bash 쉘 내장 언어로 작성된 bello 프로그램을 raw-interpret하는 방법을 보여줍니다.

bello

```
#!/bin/bash

printf "Hello World\n"
```

bash 와 같이 쉘 스크립팅 언어로 작성된 프로그램은 raw가 해석됩니다.

절차

- 소스 코드를 사용하여 파일을 실행하고 실행합니다.

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

2.4. 패치 소프트웨어

이 섹션에서는 소프트웨어를 패치하는 방법을 설명합니다.

RPM 패키징에서는 원래 소스 코드를 수정하는 대신 해당 코드를 보관하고 패치를 사용합니다.

패치는 다른 소스 코드를 업데이트하는 소스 코드입니다. 두 버전의 텍스트 간에 다른 것을 나타내기 때문에 *diff* 로 포맷됩니다. *diff* 는 diff 유틸리티를 사용하여 생성된 다음 patch 유틸리티를 사용하여 소스 코드에 적용됩니다.

참고

소프트웨어 개발자는 종종 git 과 같은 버전 제어 시스템을 사용하여 코드 기반을 관리합니다. 이러한 도구는 diffs 또는 패치 소프트웨어를 만드는 자체 방법을 제공합니다.

다음 예제에서는 diff 를 사용하여 원래 소스 코드에서 패치를 생성하는 방법과 패치를 사용하여 패치를 적용하는 방법을 보여줍니다. 패치는 RPM을 생성할 때 이후 섹션에서 사용됩니다. 3.2절. “SPEC 파일 작업” 을 참조하십시오.

이 절차에서는 cello.c 의 원래 소스 코드에서 패치를 생성하는 방법을 설명합니다.

절차

1. 원본 소스 코드를 유지합니다.

```
$ cp -p cello.c cello.c.orig
```

-p 옵션은 모드, 소유권 및 타임스탬프를 유지하는 데 사용됩니다.

2. 필요에 따라 cello.c 를 수정합니다.

```
#include <stdio.h>

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

3. diff 유틸리티를 사용하여 패치를 생성합니다.

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

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

로 시작하는 행은 원래 소스 코드에서 제거되고 + 로 시작하는 행으로 교체됩니다.

diff 명령과 함께 Naur 옵션을 사용하는 것이 대부분의 일반적인 사용 사례에 적합하므로 사용하는 것이 좋습니다. 그러나 이 경우 -u 옵션만 필요합니다. 특정 옵션은 다음을 보장합니다.

- - n(또는 --new-file) - 빈 파일이 있는 것처럼 파일을 처리합니다.
- -a (또는 --text) - 모든 파일을 텍스트로 처리합니다. 결과적으로 바이너리로 분류되는 파일은 무시되지 않습니다.

- **-u (또는 -U NUM 또는 --unified[=NUM]) -** 출력 **NUM**(기본값 **3**) 통합 컨텍스트 라인으로 출력을 반환합니다. 이는 패치를 변경된 소스 트리에 적용할 때 유사할 수 있는 쉽게 읽을 수 있는 형식입니다.

- **-r (또는 --recursive) -** 발견된 하위 디렉토리를 다시 비교합니다.

diff 유틸리티에 대한 일반적인 인수에 대한 자세한 내용은 **diff** 매뉴얼 페이지를 참조하십시오.

4. 패치를 파일에 저장합니다.

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

5. 원래의 **cello.c** 를 복원 :

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

RPM이 빌드되면 수정된 파일이 아니라 원본 파일이 사용되므로 원래 **cello.c** 를 유지해야 합니다. 자세한 내용은 3.2절. “SPEC 파일 작업”의 내용을 참조하십시오.

다음 절차에서는 **cello-output-first-patch.patch** 를 사용하여 **cello.c** 패치를 패치하고 패치된 프로그램을 빌드한 다음 실행하는 방법을 보여줍니다.

1. 패치 파일을 **patch** 명령으로 리디렉션합니다.

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

2. **cello.c** 의 콘텐츠가 이제 패치를 반영하는지 확인합니다.

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

int main(void){
```

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

3. 패치된 **cello.c** 를 빌드하고 실행합니다.

```
$ make clean
rm cello

$ make
gcc -g -o cello cello.c

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

2.5. 임의의 아티팩트 설치

UNIX와 유사한 시스템은 파일 시스템 계층 구조 표준(**FHS**)을 사용하여 특정 파일에 적합한 디렉터리를 지정합니다.

RPM 패키지에서 설치된 파일은 **FHS**에 따라 배치됩니다. 예를 들어 실행 가능한 파일은 시스템 **$PATH** 변수에 있는 디렉터리로 이동해야 합니다.

이 설명서의 컨텍스트에서 ***Arbitrary Artifact*** 는 **RPM**에서 시스템에 설치된 모든 것입니다. **RPM** 및 시스템의 경우 이는 스크립트일 수 있으며 패키지의 소스 코드, **pre-ECDHE** 바이너리 또는 기타 파일에서 컴파일된 바이너리일 수 있습니다.

이 섹션에서는 시스템에서 ***Arbitrary Artifacts*** 를 배치하는 두 가지 일반적인 방법에 대해 설명합니다.

- 2.5.1절. “설치 명령 사용”
- 2.5.2절. “make install 명령 사용”

2.5.1. 설치 명령 사용

패키지 관리자는 종종 GNU make 와 같은 빌드 자동화 툴이 최적이 아닌 경우(예: 패키지 프로그램에 추가 오버헤드가 필요하지 않은 경우) **install** 명령을 사용합니다.

install 명령은 **coreutils** 를 통해 시스템에 제공되며, 이 명령은 지정된 권한 세트를 사용하여 파일 시스템에 지정된 디렉터리에 아티팩트를 배치합니다.

다음 절차에서는 이전에 생성된 **bello** 파일을 이 설치 방법의 대상으로 임의의 아티팩트로 사용합니다.

절차

1. 실행 가능한 스크립트에 공통된 권한이 있는 **bello** 파일을 **/usr/bin** 디렉터리에 배치하려면 **install** 명령을 실행합니다.

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

결과적으로 **bello** 는 이제 **$PATH** 변수에 나열된 디렉터리에 있습니다.

2. 전체 경로를 지정하지 않고 모든 디렉터리에서 **bello** 를 실행합니다.

```
$ cd ~

$ bello
Hello World
```

2.5.2. make install 명령 사용

make install 명령을 사용하는 것은 시스템에 빌드된 소프트웨어를 설치하는 자동화된 방법입니다. 이 경우 일반적으로 개발자가 작성한 **Makefile** 의 시스템에 임의의 아티팩트를 설치하는 방법을 지정해야 합니다.

이 절차에서는 빌드 아티팩트를 시스템의 선택한 위치에 설치하는 방법을 설명합니다.

절차

1. 설치 섹션을 **Makefile** 에 추가합니다.

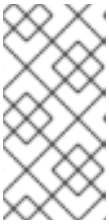
Makefile

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

```
clean:
 rm cello

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

cello 아래에 있는 행(예:, clean:, install:)은 탭 공간으로 시작해야 합니다.



참고

$(DESTDIR) 변수는 GNU make built-in이며 일반적으로 루트 디렉터리와 다른 디렉터리에 설치를 지정하는 데 사용됩니다.

이제 Makefile 을 사용하여 소프트웨어를 빌드할 뿐만 아니라 대상 시스템에 설치할 수도 있습니다.

2. cello.c 프로그램을 빌드하고 설치합니다.

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

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

결과적으로 cello 는 이제 $PATH 변수에 나열된 디렉터리에 있습니다.

3. 전체 경로를 지정하지 않고 모든 디렉터리에서 cello 를 실행합니다.

```
$ cd ~

$ cello
Hello World
```

2.6. 패키징을 위한 소스 코드 준비

개발자는 종종 소프트웨어를 소스 코드의 압축 아카이브로 배포하며 패키지를 만드는 데 사용됩니다. RPM 패키지는 준비된 소스 코드 아카이브와 함께 작동합니다.

소프트웨어는 소프트웨어 라이센스와 함께 배포되어야 합니다.

이 절차에서는 GPLv3 라이센스 텍스트를 LICENSE 파일의 예제 콘텐츠로 사용합니다.

절차

- LICENSE 파일을 만들고 다음 내용이 포함되어 있는지 확인합니다.

```
$ cat /tmp/LICENSE
This program is free software: you can redistribute it and/or modify it under the terms
of the GNU General Public License as published by the Free Software Foundation,
either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this
program. If not, see http://www.gnu.org/licenses/.
```

추가 리소스

- 이 섹션에서 생성된 코드는 여기에서 확인할 수 있습니다.

2.7. TARBALL에 소스 코드 배치

이 섹션에서는 2.1.1절. “소스 코드 예” 에 도입된 세 개의 Hello World 프로그램을 gzip-compressed tarball에 배치하는 방법을 설명합니다. 이는 나중에 배포를 위해 패키징할 소프트웨어를 릴리스하는 일반적인 방법입니다.

2.7.1. bello 프로젝트를 tarball에 배치

bello 프로젝트는 bash 에서 Hello World 를 구현합니다. 구현에는 bello 쉘 스크립트만 포함되므로 결과 tar.gz 아카이브에는 LICENSE 파일 외에도 하나의 파일만 포함됩니다.

이 절차에서는 배포를 위해 *bello* 프로젝트를 준비하는 방법을 설명합니다.

사전 요구 사항

이것은 프로그램의 버전 **0.1** 입니다.

절차

1. 필요한 모든 파일을 단일 디렉터리에 배치합니다.

```
$ mkdir /tmp/bello-0.1

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

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

2. 배포를 위한 아카이브를 생성하고 **rpmbuild/SOURCES/** 디렉터리로 이동합니다. 이 디렉터리는 **rpmbuild** 명령이 패키지를 빌드하는 파일을 저장하는 기본 디렉터리입니다.

```
$ cd /tmp/

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

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

bash로 작성된 소스 코드의 예제에 대한 자세한 내용은 2.1.1.1절. “helloworld는 bash로 작성된 것입니다.” 을 참조하십시오.

2.7.2. pello 프로젝트를 tarball에 배치

pello 프로젝트는 Python 에서 **Hello World** 를 구현합니다. 구현에는 **pello.py** 프로그램만 포함되므로 결과 **tar.gz** 아카이브에는 **LICENSE** 파일 외에도 하나의 파일만 포함됩니다.

이 절차에서는 배포를 위해 ***pello*** 프로젝트를 준비하는 방법을 설명합니다.

사전 요구 사항

이것은 프로그램의 버전 **0.1.1** 입니다.

절차

1. 필요한 모든 파일을 단일 디렉터리에 배치합니다.

```
$ mkdir /tmp/pello-0.1.2

$ mv ~/pello.py /tmp/pello-0.1.2/

$ cp /tmp/LICENSE /tmp/pello-0.1.2/
```

2. 배포를 위한 아카이브를 생성하고 **rpmbuild/SOURCES/** 디렉터리로 이동합니다. 이 디렉터리는 **rpmbuild** 명령이 패키지를 빌드하는 파일을 저장하는 기본 디렉터리입니다.

```
$ cd /tmp/

$ tar -cvzf pello-0.1.2.tar.gz pello-0.1.2
pello-0.1.2/
pello-0.1.2/LICENSE
pello-0.1.2/pello.py

$ mv /tmp/pello-0.1.2.tar.gz ~/rpmbuild/SOURCES/
```

Python으로 작성된 소스 코드에 대한 자세한 내용은 2.1.1.2절. “helloworld는 Python으로 작성된 것입니다.” 을 참조하십시오.

2.7.3. cello 프로젝트를 tarball에 배치

cello 프로젝트는 **C**에서 **Hello World** 를 구현합니다. 구현에는 **cello.c** 및 **Makefile** 파일만 포함되므로 결과 **tar.gz** 아카이브에는 **LICENSE** 파일과 별도로 두 개의 파일이 있습니다.

참고

패치 파일은 프로그램과 함께 아카이브에 배포되지 않습니다. **RPM Packager**는 **RPM**이 빌드될 때 패치를 적용합니다. 패치는 **.tar.gz** 아카이브와 함께 **~/rpmbuild/SOURCES/** 디렉토리에 배치됩니다.

다음 절차에서는 배포를 위해 ***cello*** 프로젝트를 준비하는 방법을 설명합니다.

사전 요구 사항

이는 프로그램의 버전 **1.0** 입니다.

절차

1. 필요한 모든 파일을 단일 디렉터리에 배치합니다.

```
$ mkdir /tmp/cello-1.0

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

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

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

2. 배포를 위한 아카이브를 생성하고 **rpmbuild/SOURCES/** 디렉터리로 이동합니다. 이 디렉터리는 **rpmbuild** 명령이 패키지를 빌드하는 파일을 저장하는 기본 디렉터리입니다.

```
$ cd /tmp/

$ tar -cvzf cello-1.0.tar.gz cello-1.0
cello-1.0/
cello-1.0/Makefile
cello-1.0/cello.c
cello-1.0/LICENSE

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

3. 패치를 추가합니다.

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

C로 작성된 소스 코드에 대한 자세한 내용은 2.1.1.3절. “C로 작성된 hello World” 을 참조하십시오.

3장. 패키지 소프트웨어

3.1. RPM 패키지

이 섹션에서는 RPM 패키지 형식의 기본 사항에 대해 설명합니다.

3.1.1. RPM이란 무엇입니까?

RPM 패키지는 다른 파일 및 메타데이터(시스템에 필요한 파일에 대한)를 포함하는 파일입니다.

특히 RPM 패키지는 cpio 아카이브로 구성됩니다.

cpio 아카이브에는 다음이 포함됩니다.

- Files
- RPM 헤더(패키지 메타데이터)

  rpm 패키지 관리자는 이 메타데이터를 사용하여 종속성, 파일 설치 위치 및 기타 정보를 결정합니다.

RPM 패키지 유형

RPM 패키지에는 두 가지 유형이 있습니다. 두 유형 모두 파일 형식과 툴링을 공유하지만 내용이 다르고 용도가 다릅니다.

- 소스 RPM(SRPM)

  SRPM에는 소스 코드와 SPEC 파일이 포함되어 있으며, 바이너리 RPM에 소스 코드를 빌드하는 방법을 설명합니다. 선택적으로 소스 코드에 대한 패치도 포함됩니다.

- 바이너리 RPM

  바이너리 RPM에는 소스 및 패치에서 빌드된 바이너리가 포함되어 있습니다.

3.1.2. RPM 패키징 툴의 유틸리티 나열

다음 절차에서는 **rpmdevtools** 패키지에서 제공하는 유틸리티를 나열하는 방법을 보여줍니다.

사전 요구 사항

RPM 패키지 툴을 사용하려면 **RPM** 패키징을 위한 여러 유틸리티를 제공하는 **rpmdevtools** 패키지를 설치해야 합니다.

```
# yum install rpmdevtools
```

절차

- **RPM** 패키지 도구의 유틸리티를 나열합니다.

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

추가 정보

- 위의 유틸리티에 대한 자세한 내용은 도움말 페이지 또는 도움말 대화 상자를 참조하십시오.

3.1.3. RPM 패키징 작업 공간 설정

이 섹션에서는 **rpmdev-setuptree** 유틸리티를 사용하여 **RPM** 패키징 작업 공간인 디렉터리 레이아웃을 설정하는 방법을 설명합니다.

사전 요구 사항

rpmdevtools 패키지가 시스템에 설치되어 있어야 합니다.

```
# yum install rpmdevtools
```

절차

- **rpmdev-setuptree** 유틸리티를 실행합니다.

```
$ rpmdev-setuptree

$ tree ~/rpmbuild/
/home/<username>/rpmbuild/
```

```
|-- BUILD
|-- RPMS
|-- SOURCES
|-- SPECS
`-- SRPMS

5 directories, 0 files
```

생성된 디렉터리는 다음 목적을 제공합니다.

디렉터리	목적
빌드	패키지가 빌드되면 다양한 **%buildroot** 디렉터리가 생성됩니다. 이는 로그 출력에서 충분한 정보를 제공하지 않는 경우 실패한 빌드를 조사하는 데 유용합니다.
RPMS	여기에서 바이너리 RPM은 다른 아키텍처를 위한 하위 디렉터리(예: 하위 디렉터리 **x86_64** 및 **noarch**)에서 생성됩니다.
소스	여기에서 packager는 압축 소스 코드 아카이브와 패치를 배치합니다. **rpmbuild** 명령은 여기에서 찾습니다.
SPECS	패키지러는 SPEC 파일을 여기에 배치합니다.
SRPMS	바이너리 RPM 대신 **rpmbuild** 를 사용하여 SRPM을 빌드하면 결과 SRPM이 생성됩니다.

3.1.4. SPEC 파일은 무엇입니까?

SPEC 파일을 rpmbuild 유틸리티에서 RPM을 빌드하는 데 사용하는 레시피로 이해할 수 있습니다. SPEC 파일은 일련의 섹션에서 명령을 정의하여 빌드 시스템에 필요한 정보를 제공합니다. 섹션은 *Preamble* 및 DestinationRule 부분에 *정의되어* 있습니다. *Preamble* 부분에는 DestinationRule 부분에서 사용되는 일련의 메타데이터 항목이 포함되어 *있습니다*. *DestinationRule* 부분은 지침의 주요 부분을 나타냅니다.

3.1.4.1. Preamble 항목

아래 표는 RPM SPEC 파일의 *Preamble* 섹션에서 자주 사용되는 몇 가지 지시문입니다.

표 3.1. RPM SPEC 파일의 *Preamble* 섹션에서 사용되는 항목

SPECECDHE	정의
이름	SPEC 파일 이름과 일치해야 하는 패키지의 기본 이름입니다.
버전	소프트웨어의 업스트림 버전 번호입니다.

SPECECDHE	정의
릴리스 버전	이 버전의 소프트웨어가 릴리스된 횟수입니다. 일반적으로 초기 값을 1%{?dist}로 설정하고 새 패키지 릴리스마다 늘립니다. 소프트웨어의 새 버전이 빌드된 경우 1로 재설정합니다.
요약	패키지에 대한 한 줄 요약입니다.
라이센스	소프트웨어 라이센스가 패키지됩니다.
URL	프로그램에 대한 자세한 내용은 전체 URL을 참조하십시오. 대부분의 경우 이것은 패키지화되는 소프트웨어의 업스트림 프로젝트 웹 사이트입니다.
Source0	업스트림 소스 코드의 압축 아카이브에 대한 경로 또는 URL(패치되지 않은 패치는 다른 위치에서 처리됨). 이는 아카이브의 로컬 스토리지가 아니라 액세스 가능하고 안정적인 아카이브(예: 업스트림 페이지)를 가리켜야 합니다. 필요하면 Source1, Source2, Source3 등과 같이 때마다 더 많은 SourceX 지시문을 추가할 수 있습니다.
패치	필요한 경우 소스 코드에 적용할 첫 번째 패치의 이름입니다. 이 지시문을 Patch 끝에 숫자와 함께 사용하거나 사용하지 않는 두 가지 방법으로 적용할 수 있습니다. 번호가 지정되지 않으면 내부적으로 항목이 할당됩니다. Patch0, Patch1, Patch2, Patch3 등을 사용하여 명시적으로 번호를 지정할 수도 있습니다. 이러한 패치는 %patch0, %patch1, %patch2 매크로 등을 사용하여 하나씩 적용할 수 있습니다. 매크로는 RPM SPEC 파일의 DestinationRule 섹션에 *있는* %prep 지시문에 적용됩니다. 또는 %autopatch 매크로를 사용하여 SPEC 파일에서 제공되는 모든 패치를 자동으로 적용할 수 있습니다.
BuildArch	패키지가 아키텍처 종속적이지 않은 경우 예를 들어 해석된 프로그래밍 언어로 완전히 작성된 경우 이 패키지를 **BuildArch: noarch** 로 설정합니다. 설정하지 않으면 패키지는 빌드된 시스템의 아키텍처를 자동으로 상속합니다(예: **x86_64**).
BuildRequires	컴파일된 언어로 작성된 프로그램을 빌드하는 데 필요한 패키지의 쉼표 또는 공백으로 구분된 목록입니다. **BuildRequires** 의 여러 항목이 있을 수 있습니다. 각각 SPEC 파일에 자체 행에 있습니다.
requires	설치된 한 번 실행하는 데 필요한 패키지의 쉼표 또는 공백으로 구분된 목록입니다. **Requires** 의 여러 항목이 있을 수 있습니다. 각각 SPEC 파일에 자체 행에 있습니다.
ExcludeArch	특정 프로세서 아키텍처에서 소프트웨어를 작동할 수 없는 경우 여기에서 해당 아키텍처를 제외할 수 있습니다.

SPECECDHE	정의
충돌	**충돌** 은 **Requires** 와 다릅니다. **충돌과 일치하는 패키지가 있는 경우 충돌** 태그가 이미 설치된 패키지에 있는지 아니면 설치하려는 패키지에 있는지 여부에 대해 패키지를 독립적으로 설치할 수 없습니다.
사용되지 않음	이 지시문에서는 **rpm** 명령을 명령줄에서 직접 사용하는지 아니면 업데이트 또는 종속성 솔버를 통해 업데이트를 수행하는지 여부에 따라 업데이트되는 방식을 변경합니다. 명령행에서 사용하는 경우 RPM은 설치 중인 패키지의 더 이상 사용되지 않는 것과 일치하는 모든 패키지를 제거합니다. 업데이트 또는 종속성 확인자를 사용하는 경우 일치하는 **Obsoletes:** 패키지가 업데이트로 추가되고 일치하는 패키지를 교체합니다.
provides	If **Provides** is added to a package, the package can be referred to by dependencies other than its name.

Name,Version, Release 지시문에는 RPM 패키지의 파일 이름이 포함됩니다. RPM 패키지 유지 관리자 및 시스템 관리자는 RPM 패키지 파일 이름에 NAME-VERSION-RELEASE 형식이 있기 때문에 이러한 세 가지 지시문인 N-V-R 또는 NVR 을 호출하는 경우가 많습니다.

다음 예제에서는 rpm 명령을 쿼리하여 특정 패키지에 대한 NVR 정보를 가져오는 방법을 보여줍니다.

예 3.1. bash 패키지에 대한 NVR 정보를 제공하도록 rpm 쿼리

```
$ rpm -q bash
bash-4.2.46-34.el7.x86_64
```

여기에서 bash 는 패키지 이름이고 4.2.46 은 버전이며 34.el7 은 릴리스입니다. 최종 마커는 x86_64 로 아키텍처 신호를 보냅니다. NVR 과 달리 아키텍처 마커는 RPM 패키지 관리자를 직접 제어하지 않지만 rpmbuild 빌드 환경에서 정의합니다. 이에 대한 예외는 아키텍처 독립적인 noarch 패키지입니다.

3.1.4.2. 본문 항목

RPM SPEC 파일 의 DestinationRule 섹션에 사용된 항목은 아래 표에 나열되어 있습니다.

표 3.2. RPM SPEC 파일의 DestinationRule 섹션에서 사용되는 항목

SPECECDHE	정의
%description	RPM에 패키지된 소프트웨어에 대한 전체 설명입니다. 이 설명은 여러 줄에 걸쳐 있을 수 있으며 단락으로 나눌 수 있습니다.

SPECECDHE	정의
%prep	소프트웨어를 빌드할 명령 또는 일련의 명령(예: **Source0** 에서 아카이브 압축을 풉니다.) 이 지시문에는 쉘 스크립트가 포함될 수 있습니다.
%build	소프트웨어를 머신 코드(컴파일 언어용) 또는 바이트 코드(일부 해석된 언어의 경우)로 빌드하기 위한 명령 또는 일련의 명령
%install	**%builddir** 에서 원하는 빌드 아티팩트를 복사하기 위한 명령 또는 일련의 명령(빌드이 발생하는 경우)은 패키징할 파일과 함께 디렉터리 구조를 포함하는 **%buildroot** 디렉토리로 복사합니다. 일반적으로 ~/rpmbuild/BUILD에서 **~/rpmbuild/BUILD ROOT** 로 파일을 복사하고 **~/rpmbuild/BUILDROOT** 에 필요한 디렉터리를 만드는 것을 의미합니다. 이는 최종 사용자가 패키지를 설치하는 경우가 아니라 패키지를 만들 때만 실행됩니다. 자세한 내용은 3.2절. "SPEC 파일 작업" 을 참조하십시오.
%check	소프트웨어를 테스트하기 위한 명령 또는 일련의 명령 여기에는 일반적으로 단위 테스트와 같은 사항이 포함됩니다.
%files	최종 사용자의 시스템에 설치될 파일 목록입니다.
%changelog	다양한 **버전** 또는 **릴리스** 빌드 간 패키지에 발생한 변경 사항 레코드입니다.

3.1.4.3. 고급 항목

SPEC 파일에는 또한 Scriptlets 또는 Triggers 와 같은 고급 항목을 포함할 수 있습니다. 빌드 프로세스가 아닌 최종 사용자 시스템의 설치 프로세스 중에 다른 시점에서 적용됩니다.

3.1.5. BuildRoots

RPM 패키징 컨텍스트에서 buildroot 는 chroot 환경입니다. 즉, 빌드 아티팩트는 최종 사용자의 시스템에서 향후 계층 구조와 동일한 파일 시스템 계층을 사용하여 여기에 배치되며 buildroot 가 루트 디렉터리로 작동합니다. 빌드 아티팩트 배치는 최종 사용자 시스템의 파일 시스템 계층 표준을 준수해야 합니다.

나중에 buildroot 의 파일은 RPM의 주요 부분이 되는 cpio 아카이브에 배치됩니다. RPM이 최종 사용자 시스템에 설치되면 이러한 파일이 루트 디렉터리에 추출되어 올바른 계층을 유지합니다.

참고

6부터 rpmbuild 프로그램에는 자체 기본값이 있습니다. 이러한 기본값을 재정의하면 여러 문제가 발생합니다. 따라서 {RH}는 이 매크로의 고유 값을 정의하지 않는 것이 좋습니다. rpmbuild 디렉터리의 기본값과 함께 %{buildroot} 매크로를 사용할 수 있습니다.

3.1.6. RPM 매크로

rpm 매크로 는 특정 기본 제공 기능이 사용될 때 문의 선택적 평가를 기반으로 조건부로 할당할 수 있는 간단한 텍스트 대체입니다. 따라서 **RPM**은 텍스트 대체를 수행할 수 있습니다.

예제 사용은 패키지 소프트웨어 버전을 **SPEC** *파일에서* 여러 번 참조하는 것입니다. **%{version}** 매크로에서 한 번만 ***Version*** 을 정의하고 **SPEC** 파일 전체에서 이 매크로를 사용합니다. 모든 이벤트는 이전에 정의한 버전으로 자동으로 대체됩니다.

참고

예기치 않은 매크로가 표시되면 다음 명령을 사용하여 평가할 수 있습니다.

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

%{_bindir} 및 %{_libexecdir} 매크로 평가

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

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

일반적으로 사용되는 매크로는 %{?dist} 매크로로, 빌드에 사용되는 배포(distribution 태그)를 나타냅니다.

```
# On a RHEL 8.x machine
$ rpm --eval %{?dist}
.el8
```

3.2. SPEC 파일 작업

이 섹션에서는 SPEC 파일을 만들고 수정하는 방법에 대해 설명합니다.

사전 요구 사항

이 섹션에서는 2.1.1절. "소스 코드 예" 에 설명된 Hello World! 프로그램의 세 가지 예제 구현을 사용합니다.

각 프로그램은 아래 표에 설명되어 있습니다.

소프트웨어 이름	예를 들어 설명
Bello	원시 해석 프로그래밍 언어로 작성된 프로그램입니다. 소스 코드를 빌드할 필요가 없는 경우를 나타내지만 설치만 하면 됩니다. pre-ECDHE 바이너리를 패키징해야 하는 경우 바이너리도 파일이므로 이 방법을 사용할 수도 있습니다.
pello	바이트로 해석되는 프로그래밍 언어로 작성된 프로그램입니다. 소스 코드를 바이트로 컴파일하고 바이트 코드를 설치하며, 그 결과 미리 최적화된 파일을 보여줍니다.
cello	기본적으로 컴파일된 프로그래밍 언어로 작성된 프로그램입니다. 소스 코드를 머신 코드로 컴파일하고 결과 실행 파일을 설치하는 일반적인 프로세스를 보여줍니다.

Hello World! 의 구현은 다음과 같습니다.

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

사전 요구 사항으로 이러한 구현을 ~/rpmbuild/SOURCES 디렉터리에 배치해야 합니다.

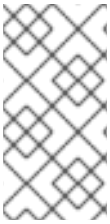
3.2.1. 새로운 SPEC 파일을 만드는 방법

새 소프트웨어를 패키지하려면 새로운 SPEC 파일을 만들어야 합니다.

이를 위해서는 두 가지가 있습니다.

- 새 **SPEC** 파일을 처음부터 수동으로 작성
- **rpmdev-newspec** 유틸리티 사용

  이 유틸리티는 채워지지 않은 **SPEC** 파일을 만들고 필요한 지시문과 필드를 작성합니다.



참고

일부것에 중점을 둔 텍스트 편집기는 새로운 **.spec** 파일을 자체 **SPEC** 템플릿으로 미리 채웁니다. **rpmdev-newspec** 유틸리티는 편집기와 무관한 방법을 제공합니다.

3.2.2. rpmdev-newspec을 사용하여 새 SPEC 파일 생성

다음 절차에서는 **rpmdev-newspec** 유틸리티를 사용하여 앞서 언급한 세 가지 **Hello World!** 프로그램 각각에 대해 **SPEC** 파일을 생성하는 방법을 보여줍니다.

절차

1. **~/rpmbuild/SPECS** 디렉터리로 변경하고 **rpmdev-newspec** 유틸리티를 사용합니다.

```
$ cd ~/rpmbuild/SPECS

$ rpmdev-newspec bello
bello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec cello
cello.spec created; type minimal, rpm version >= 4.11.

$ rpmdev-newspec pello
pello.spec created; type minimal, rpm version >= 4.11.
```

   ~/rpmbuild/SPECS/ 디렉토리에는 이제 **bello.spec,cello.spec** 및 **pello.spec** 이라는 세 개의 **SPEC** 파일이 포함되어 있습니다.

fd. 파일을 확인합니다.

+

참고

rpmdev-newspec 유틸리티는 특정 Linux 배포판과 관련된 지침이나 규칙을 사용하지 않습니다. 그러나 이 문서를 대상으로 하는 이 문서는 이므로 RPM의 Build root를 참조하는 경우 $RPM_BUILD_ROOT 표기법을 통해 %{build root 표기법을 우선하여 다른 모든 정의된 또는 제공된 매크로와의 일관성을 참조합니다.

3.2.3. RPM을 생성하기 위한 원본 SPEC 파일 수정

다음 절차에서는 RPM을 생성하기 위해 rpmdev-newspec 에서 제공하는 output SPEC 파일을 수정하는 방법을 보여줍니다.

사전 요구 사항

다음을 확인하십시오.

- 특정 프로그램의 소스 코드가 ~/rpmbuild/SOURCES/ 디렉터리에 배치됩니다.
- 채워지지 않은 SPEC 파일 ~/rpmbuild/SPECS/<name>.spec 파일은 rpmdev-newspec 유틸리티에 의해 생성되었습니다.

절차

1. rpmdev-newspec 유틸리티에서 제공하는 ~/rpmbuild/SPECS/<name>.spec 파일의 출력 템플릿을 엽니다.
2. SPEC 파일의 첫 번째 섹션을 채웁니다.

   첫 번째 섹션에는 rpmdev-newspec 이 함께 그룹화된 다음 지시문이 포함되어 있습니다.

   - 이름
   - 버전

- *릴리스 버전*

- *요약*

  *이름은 이미 **rpmdev-newspec** 에 인수로 지정되어 있습니다.*

  *소스 코드의 업스트림 릴리스 버전과 일치하도록 **Version** 을 설정합니다.*

  *릴리스 는 자동으로 **1** %인 **1%{?dist}** 로 설정됩니다. 업스트림 릴리스 버전이 변경되지 않고 패키지를 업데이트할 때마다(예: 패치 포함 시) 초기 값을 늘립니다. 새로운 업스트림 릴리스가 발생하면 릴리스 를 **1** 로 재설정합니다.*

  요약 은 이 소프트웨어가 무엇인지에 대한 간단한 한 줄 설명입니다.

3. *라이센스,**URL**, **Source0** 지시문을 채웁니다.*

   ***License** 필드는 업스트림 릴리스의 소스 코드와 관련된 소프트웨어 라이센스입니다. **SPEC** 파일에서 라이센스 레이블을 지정하는 방법에 대한 정확한 형식은 다음과 같은 특정 **RPM** 기반 **Linux** 배포 지침에 따라 다릅니다.*

   예를 들어, GPLv3+ 를 사용할 수 있습니다.

   ***URL** 필드는 업스트림 소프트웨어 웹 사이트에 **URL**을 제공합니다. 일관성을 위해 **%{name}** 의 **RPM** 매크로 변수를 사용하고 **https://example.com/%{name}** 을 사용합니다.*

   ***Source0** 필드는 업스트림 소프트웨어 소스 코드에 **URL**을 제공합니다. 패키지화 중인 특정 버전의 소프트웨어에 직접 연결되어야 합니다. 이 문서에 제공된 예제 **URL**에는 나중에 변경될 수 있는 하드 코드된 값이 포함되어 있습니다. 마찬가지로 릴리스 버전도 변경될 수 있습니다. 이러한 잠재적인 향후 변경 사항을 단순화하려면 **%{name}** 및 **%{version}** 매크로를 사용합니다. 이 값을 사용하면 **SPEC** 파일에서 필드 하나만 업데이트해야 합니다.*

4. ***BuildRequires**,**Requires** 및 **BuildArch** 지시문을 채웁니다.*

   ***BuildRequires** 는 패키지에 대한 빌드 타임 종속 항목을 지정합니다.*

패키지에 대한 런타임 종속성을 지정합니다.

*이는 기본적으로 컴파일된 확장없이 해석된 프로그래밍 언어로 작성된 소프트웨어입니다. 따라서 **noarch** 값을 사용하여 **BuildArch** 지시문을 추가합니다. **RPM**에 이 패키지가 빌드된 프로세서 아키텍처에 바인딩할 필요가 없음을 나타냅니다.*

5.

***%description,%prep,%build,%install,%files, %license** 지시문을 채우십시오.*

이러한 지시문은 다중 줄, 다중 구조 또는 스크립트 작업을 수행할 수 있는 지시문이므로 섹션 제목으로 간주할 수 있습니다.

***%description** 은 하나 이상의 단락을 포함하는 **Summary** 보다 소프트웨어에 대한 더 긴 전체 설명입니다.*

***%prep** 섹션에서는 빌드 환경을 준비하는 방법을 지정합니다. 일반적으로 소스 코드의 압축 아카이브 확장, 패치 적용, 소스 코드에 제공된 정보의 구문 분석을 통해 **SPEC** 파일의 뒷부분에서 사용할 수 있습니다. 이 섹션에서는 내장된 **%setup -q** 매크로를 사용할 수 있습니다.*

***%build** 섹션은 소프트웨어 빌드 방법을 지정합니다.*

***%install** 섹션에는 소프트웨어를 빌드한 후 **BUILDROOT** 디렉터리에 설치하는 방법에 대한 **rpmbuild** 지침이 포함되어 있습니다.*

*이 디렉토리는 최종 사용자의 **root** 디렉토리와 유사한 빈 **chroot** 기본 디렉터리입니다. 여기에서 설치된 파일이 포함될 모든 디렉터리를 만들 수 있습니다. 이러한 디렉터리를 생성하려면 경로를 하드 코딩하지 않고도 **RPM** 매크로를 사용할 수 있습니다.*

***%files** 섹션은 이 **RPM**에서 제공하는 파일 목록과 최종 사용자의 시스템의 전체 경로 위치를 지정합니다.*

*이 섹션에서는 기본 제공 매크로를 사용하여 다양한 파일의 역할을 나타낼 수 있습니다. 이 명령은 **command[]rpm** 명령을 사용하여 패키지 파일 매니페스트 메타데이터를 쿼리하는 데 유용합니다. 예를 들어 **LICENSE** 파일이 소프트웨어 라이센스 파일임을 나타내려면 **%license** 매크로를 사용합니다.*

6.

마지막 섹션인 **%changelog** 는 패키지의 각 버전 릴리스에 대한 **datetamped** 항목 목록입니다. 소프트웨어 변경 사항이 아닌 패키지 변경 사항을 기록합니다. 패키징 변경 사항의 예: 패치 추가, **%build** 섹션의 빌드 절차를 변경합니다.

첫 번째 줄에 대해 다음 형식을 따르십시오.

* 문자로 시작한 다음 **Day-of-Week Month Name Surname <email> - Version-Release**로 시작합니다.

실제 변경 항목에 대해서는 다음 형식을 따릅니다.

- 각 변경 항목은 변경할 때마다 하나씩 여러 항목을 포함할 수 있습니다.
- 각 항목은 새 줄에서 시작됩니다.
- 각 항목은 **-** 문자로 시작합니다.

이제 필요한 프로그램에 대한 전체 **SPEC** 파일을 작성했습니다.

다른 프로그래밍 언어로 작성된 **SPEC** 파일의 예는 다음을 참조하십시오.

3.2.4. bash로 작성된 프로그램에 대한 SPEC 파일의 예

이 섹션에서는 **bash**로 작성된 **bello** 프로그램에 대한 **SPEC** 파일의 예를 보여줍니다. **bello** 에 대한 자세한 내용은 2.1.1절. “소스 코드 예” 을 참조하십시오.

bash로 작성된 **bello** 프로그램에 대한 **SPEC** 파일의 예

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

License:        GPLv3+
URL:            https://www.example.com/%{name}
```

```
Source0:        https://www.example.com/%{name}/releases/%{name}-%{version}.tar.gz

Requires:       bash

BuildArch:      noarch

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

%prep
%setup -q

%build

%install

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

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

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

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- First bello package
- Example second item in the changelog for version-release 0.1-1
```

bello 에 대한 빌드 단계가 없기 때문에 패키지에 대한 빌드 타임 종속 항목을 지정하는 **BuildRequires** 지시문이 삭제되었습니다. **Bash**는 원시 해석 프로그래밍 언어이며 파일은 시스템의 위치에만 설치됩니다.

패키지에 대한 런타임 종속성을 지정하는 **Requires** 지시문에는 **bello** 스크립트만 실행할 **bash** 쉘 환경만 포함하므로 **bash** 만 포함합니다.

bash 를 빌드할 필요가 없으므로 소프트웨어 빌드 방법을 지정하는 **%build** 섹션이 비어 있습니다.

bello 를 설치하려면 대상 디렉터리를 생성하고 실행 가능한 **bash** 스크립트 파일만 설치해야 합니다. 따라서 **%install** 섹션에서 **install** 명령을 사용할 수 있습니다. **RPM** 매크로를 사용하면 하드 코딩 경로 없이 이 작업을 수행할 수 있습니다.

3.2.5. Python으로 작성된 프로그램의 SPEC 파일 예

이 섹션에서는 Python 프로그래밍 언어로 작성된 pello 프로그램에 대한 SPEC 파일의 예를 보여줍니다. pello 에 대한 자세한 내용은 2.1.1절. “소스 코드 예” 을 참조하십시오.

Python으로 작성된 pello 프로그램에 대한 SPEC 파일의 예

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

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

BuildRequires:  python
Requires:      python
Requires:      bash

BuildArch:     noarch

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

%prep
%setup -q

%build

python -m compileall %{name}.py

%install

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

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

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

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

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

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

중요

pello 프로그램은 바이트로 해석되는 언어로 작성됩니다. 따라서 결과 파일에 항목이 포함되어 있지 않기 때문에**hebang**이 적용되지 않습니다.

Beanbang은 적용되지 않기 때문에 다음 방법 중 하나를 적용하는 것이 좋습니다.

- 실행 파일을 호출할 바이트가 아닌 쉘 스크립트를 생성합니다.
- 프로그램의 실행 진입점으로 바이트가 아닌 **Python** 코드의 작은 비트를 제공합니다.

이러한 접근 방식은 수천 줄의 코드가 있는 대규모 소프트웨어 프로젝트에 특히 유용하며, 이 경우 사전 바이트 단위 코드의 성능 향상이 가능합니다.

패키지에 대한 빌드 타임 종속 항목을 지정하는 **BuildRequires** 지시문에는 다음 두 가지 패키지가 포함됩니다.

- 바이트-컴파일 빌드 프로세스를 수행하려면 **python** 패키지가 필요합니다.
- 작은 진입점 스크립트를 실행하려면 **bash** 패키지가 필요합니다.

패키지에 대한 런타임 종속성을 지정하는 **Requires** 지시문에는 **python** 패키지만 포함됩니다. **pello** 프로그램을 사용하려면 런타임 시 **python** 패키지가 바이트-**ECDHE** 코드를 실행해야 합니다.

소프트웨어를 빌드하는 방법을 지정하는 **%build** 섹션은 소프트웨어가 바이트 단위라는 사실에 해당합니다.

pello 를 설치하려면hebang이 바이트-ECDHE 언어로 적용되지 않기 때문에 래퍼 스크립트를 생성해야 합니다. 다음과 같은 다양한 옵션을 사용할 수 있습니다.

- *별도의 스크립트를 작성하고 이를 별도의 SourceX 지시문으로 사용합니다.*
- *SPEC 파일에서 인라인으로 파일 만들기.*

이 예제에서는 SPEC 파일 자체를 스크립팅할 수 있음을 설명하기 위해 SPEC 파일에서 래퍼 스크립트 생성을 보여줍니다. 이 래퍼 스크립트는 여기에 있는 문서를 사용하여 Python 바이트-ECDHE 코드를 실행합니다.

이 예제의 %install 섹션은 바이트 파일 을 액세스할 수 있도록 시스템의 라이브러리 디렉터리에 설치해야 한다는 사실에도 해당합니다.

3.2.6. C로 작성된 프로그램에 대한 SPEC 파일의 예

이 섹션에서는 C 프로그래밍 언어로 작성된 cello 프로그램에 대한 SPEC 파일의 예를 보여줍니다. cello 에 대한 자세한 내용은 2.1.1절. “소스 코드 예” 을 참조하십시오.

C로 작성된 cello 프로그램에 대한 SPEC 파일의 예

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

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

Patch0:       cello-output-first-patch.patch

BuildRequires:  gcc
BuildRequires:  make

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

%prep
%setup -q
```

```
%patch0

%build
make %{?_smp_mflags}

%install
%make_install

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

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 1.0-1
- First cello package
```

*패키지에 대한 빌드 타임 종속 항목을 지정하는 **BuildRequires** 지시문에는 컴파일 빌드 프로세스를 수행하는 데 필요한 두 개의 패키지가 포함되어 있습니다.*

- *gcc 패키지*
- *make 패키지*

*패키지에 대한 런타임 종속성을 지정하는 **Requires** 지시문이 이 예제에서는 생략됩니다. 모든 런타임 요구 사항은 **rpmbuild** 에서 처리하며 **cello** 프로그램에는 핵심 **C** 표준 라이브러리 이외의 항목이 필요하지 않습니다.*

*%build 섹션은 이 예에서 **cello** 프로그램에 대한 **Makefile** 이 작성되었으므로 **rpmdev-newspec** 유틸리티에서 제공하는 GNU make 명령을 사용할 수 있다는 사실을 반영합니다. 그러나 구성 스크립트를 제공하지 않았기 때문에 **%configure** 에 대한 호출을 제거해야 합니다.*

***cello** 프로그램의 설치는 **rpmdev-newspec** 명령에서 제공한 **%make_install** 매크로를 사용하여 수행할 수 있습니다. 이는 **cello** 프로그램의 **Makefile** 을 사용할 수 있기 때문에 가능합니다.*

3.3. RPM 빌드

*이 섹션에서는 프로그램에 대한 **SPEC** 파일이 생성된 후 **RPM**을 빌드하는 방법을 설명합니다.*

RPM은 rpmbuild 명령을 사용하여 빌드됩니다. 이 명령은 rpmdev-setuptree 유틸리티에 의해 설정된 구조와 동일한 특정 디렉토리 및 파일 구조를 예상합니다.

다른 사용 사례와 원하는 결과에는 rpmbuild 명령에 다양한 인수가 필요합니다. 이 섹션에서는 다음 두 가지 주요 사용 사례에 대해 설명합니다.

- 소스 RPM 빌드
- 바이너리 RPM 빌드

3.3.1. 소스 RPM 빌드

이 단락은 프로시저 모듈 소개입니다. 프로시저에 대한 간단한 설명입니다.

사전 요구 사항

패키지하려는 프로그램에 대한 SPEC 파일이 이미 있어야 합니다. SPEC 파일 생성에 대한 자세한 내용은 SPEC 파일 작업을 참조하십시오.

절차

다음 절차에서는 소스 RPM을 빌드하는 방법을 설명합니다.

- 지정된 SPEC 파일을 사용하여 rpmbuild 명령을 실행합니다.

```
$ rpmbuild -bs SPECFILE
```

SPECFILE 을 SPEC 파일로 대체합니다. b s 옵션은 빌드 소스를 나타냅니다.

다음 예제에서는 bello,pello 및 cello 프로젝트에 대한 소스 RPM을 빌드하는 방법을 보여줍니다.

bello, pello 및 cello에 대한 소스 RPM 빌드.

```
$ cd ~/rpmbuild/SPECS/

8$ rpmbuild -bs bello.spec
Wrote: /home/<username>/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm

$ rpmbuild -bs pello.spec
Wrote: /home/<username>/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm

$ rpmbuild -bs cello.spec
Wrote: /home/<username>/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
```

검증 단계

- *rpmbuild/SRPMS 디렉토리에 결과 소스 RPM이 포함되어 있는지 확인합니다. 디렉터리는 rpmbuild 에서 예상되는 구조의 일부입니다.*

3.3.2. 바이너리 RPM 빌드

바이너리 RPM을 빌드하는 데 사용할 수 있는 방법은 다음과 같습니다.

- *소스 RPM에서 바이너리 RPM 재구축*
- *SPEC 파일에서 바이너리 RPM 빌드*
- *소스 RPM에서 바이너리 RPM 빌드*

3.3.2.1. 소스 RPM에서 바이너리 RPM 재구축

다음 절차에서는 소스 RPM(SRPM)에서 바이너리 RPM을 다시 빌드하는 방법을 보여줍니다.

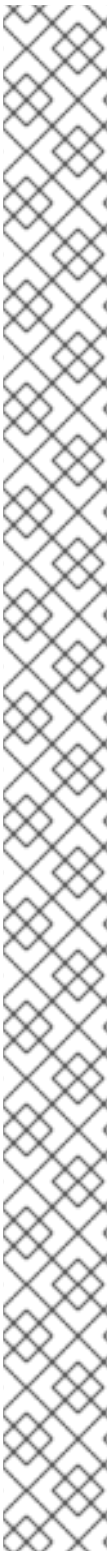
절차

- *bello,pello, cello 를 SRPM에서 다시 빌드하려면 다음을 실행합니다.*

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

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

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



참고

rpmbuild --rebuild 호출에는 다음이 포함됩니다.

- SRPM의 콘텐츠 설치 - SPEC 파일 및 소스 코드를 ~/rpmbuild/ 디렉터리에 설치합니다.
- 설치된 콘텐츠를 사용하여 빌드합니다.
- SPEC 파일 및 소스 코드 제거.

빌드 후 SPEC 파일 및 소스 코드를 유지하려면 다음을 수행할 수 있습니다.

- 빌드할 때 --rebuild 옵션 대신 --recompile 옵션과 함께 rpmbuild 명령을 사용합니다.
- 다음 명령을 사용하여 SRPM을 설치합니다.

```
$ rpm -Uvh ~/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm
Updating / installing…
   1:bello-0.1-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/pello-0.1.2-1.el8.src.rpm
Updating / installing…
…1:pello-0.1.2-1.el8              [100%]

$ rpm -Uvh ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
Updating / installing…
…1:cello-1.0-1.el8              [100%]
```

바이너리 RPM을 생성할 때 생성되는 출력은 상세 정보이며 이는 디버깅에 유용합니다. 출력은 다양한 예에 따라 다르며 SPEC 파일에 해당합니다.

생성된 바이너리 RPM은 ~/rpmbuild/RPMS/YOURARCH 디렉터리에 있습니다. 여기서R ARCH 는 아키텍처별로 고유하지 않은 경우 해당 아키텍처 또는 ~/rpmbuild/RPMS/noarch/ 디렉터리에 있습니다.

3.3.2.2. SPEC 파일에서 바이너리 RPM 빌드

다음 절차에서는 SPEC 파일에서 bello,pello, cello 바이너리 RPM을 빌드하는 방법을 보여줍니다.

절차

- bb 옵션과 함께 rpmbuild 명령을 실행합니다.

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

3.3.2.3. 소스 RPM에서 RPM 빌드

소스 RPM에서 모든 종류의 RPM을 빌드할 수도 있습니다. 이를 수행하려면 다음 절차를 사용하십시오.

절차

- 아래 옵션 중 하나와 소스 패키지가 지정된 상태에서 rpmbuild 명령을 실행합니다.

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

추가 리소스

소스 RPM에서 RPM을 빌드하는 방법에 대한 자세한 내용은 rpmbuild(8) 매뉴얼 페이지의 BUILDING PACKAGES 섹션을 참조하십시오.

3.4. 심각도를 위한 RPM 확인

패키지를 만든 후 패키지의 품질을 확인합니다.

패키지 품질을 확인하는 주요 도구는 rpmlint 입니다.

rpmlint 툴은 다음을 수행합니다.

- *RPM 유지 관리 기능 개선.*
- *RPM의 정적 분석을 수행하여 온전성 검사를 활성화합니다.*
- *RPM의 정적 분석을 수행하여 오류 검사를 활성화합니다.*

rpmlint 툴은 바이너리 RPM, 소스 RPM(SRPM) 및 SPEC 파일을 확인할 수 있으므로 다음 예와 같이 패키지의 모든 단계에 유용합니다.

rpmlint 에는 매우 엄격한 지침이 있으므로 다음 예제와 같이 일부 오류 및 경고를 건너뛸 수 있습니다.

참고

다음 예에서 rpmlint 는 옵션 없이 실행되며, 이는 비점적 출력을 생성합니다. 각 오류 또는 경고에 대한 자세한 설명을 위해 rpmlint -i 를 대신 실행할 수 있습니다.

3.4.1. sanity에 대한 bello 확인

이 섹션에서는 bello SPEC 파일 예와 bello 바이너리 RPM에서 RPM의 심각도를 확인할 때 발생할 수 있는 경고 및 오류를 보여줍니다.

3.4.1.1. bello SPEC 파일 확인

예 3.2. bello용 SPEC 파일에서 rpmlint 명령을 실행하는 출력

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

bello.spec 의 경우 하나의 경고만 있습니다. 이 경고는 Source0 지시문에 나열된 URL에 연결할 수 없음을 나타냅니다. 지정된 example.com URL이 존재하지 않기 때문에 이 문제가 예상됩니다. 나중에 이 URL이 작동할 것으로 가정하면 이 경고를 무시할 수 있습니다.

예 3.3. bello SRPM에서 rpmlint 명령을 실행하는 출력

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

bello SRPM의 경우 새로운 경고가 있습니다. 이 경고는 URL 지시문에 지정된 URL 에 연결할 수 없음을 나타냅니다. 향후에 링크가 작동할 것으로 가정하면 이 경고를 무시할 수 있습니다.

3.4.1.2. bello 바이너리 RPM 확인

바이너리 RPM을 확인할 때 rpmlint 는 다음 항목을 확인합니다.

- *문서*
- *수동 페이지*
- *파일 시스템 계층 구조 표준을 일관되게 사용*

예 3.4. bello의 바이너리 RPM에서 rpmlint 명령을 실행하는 출력

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

no-documentation 및 no-manual-page-for-binary 경고는 RPM에 문서 또는 수동 페이지가 없음을 나타냅니다. 위의 경고 외에 RPM은 rpmlint 검사를 통과했습니다.

3.4.2. sanity에 대한 pello 확인

이 섹션에서는 pello SPEC 파일 및 pello 바이너리 RPM의 예에서 RPM의 심각도를 확인할 때 발생할 수 있는 경고 및 오류를 보여줍니다.

3.4.2.1. pello SPEC 파일 확인

예 3.5. pello용 SPEC 파일에서 rpmlint 명령을 실행하는 출력

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

invalid-url Source0 경고에 따라 **Source0** 지시문에 나열된 **URL**에 연결할 수 없습니다. 지정된 **example.com URL**이 존재하지 않기 때문에 이 문제가 예상됩니다. 이 **URL**이 나중에 작동할 것으로 가정하면 이 경고를 무시할 수 있습니다.

hardcoded-library-path 오류는 라이브러리 경로를 하드 코딩하는 대신 **%{_libdir}** 매크로를 사용하도록 제안합니다. 이 예제에서는 이러한 오류를 무시해도 됩니다. 그러나 프로덕션으로 전환하는 패키지의 경우 모든 오류를 주의 깊게 확인하십시오.

예 3.6. pello SRPM에서 rpmlint 명령을 실행하는 출력

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

여기에 새로운 **invalid-url URL** 오류는 연결할 수 없는 **URL** 지시문에 대한 것입니다. 나중에 **URL**이 유효한 것으로 가정하면 이 오류를 무시해도 됩니다.

3.4.2.2. pello 바이너리 RPM 확인

바이너리 **RPM**을 확인할 때 **rpmlint** 는 다음 항목을 확인합니다.

- 문서
- 수동 페이지
- 파일 시스템 계층 구조 표준을 일관되게 사용

> 예 3.7. pello의 바이너리 RPM에서 rpmlint 명령을 실행하는 출력
>
> ```
> $ rpmlint ~/rpmbuild/RPMS/noarch/pello-0.1.2-1.el8.noarch.rpm
> pello.noarch: W: invalid-url URL: https://www.example.com/pello HTTP Error 404: Not
> Found
> pello.noarch: W: only-non-binary-in-usr-lib
> pello.noarch: W: no-documentation
> pello.noarch: E: non-executable-script /usr/lib/pello/pello.py 0644L /usr/bin/env
> pello.noarch: W: no-manual-page-for-binary pello
> 1 packages and 0 specfiles checked; 1 errors, 4 warnings.
> ```

no-documentation 및 **no-ECDHE-page-for-binary** 경고는 **RPM**에 문서 또는 수동 페이지가 없음을 나타냅니다.

only-non-binary-in-usr-lib 경고는 **/usr/lib/** 에 바이너리가 아닌 아티팩트만 제공했음을 나타냅니다. 이 디렉터리는 일반적으로 바이너리 파일인 공유 개체 파일용으로 예약되어 있습니다. 따라서 **rpmlint** 는 **/usr/lib/** 디렉토리에 있는 하나 이상의 파일이 바이너리여야 합니다.

이는 **rpmlint** 검사에서 파일 시스템 계층 구조 표준을 준수하는 예입니다. 일반적으로 **RPM** 매크로를 사용하여 파일을 올바르게 배치하십시오. 이 예제에서는 이 경고를 무시해도 됩니다.

실행 불가능한**-script** 오류는 **/usr/lib/pello/pello.py** 파일에 실행 권한이 없음을 경고합니다. 파일에 **hebang**이 포함되어 있으므로 **rpmlint** 툴에서 파일을 실행할 수 있어야 합니다. 이 예제에서는 실행 권한 없이 이 파일을 그대로 두고 이 오류를 무시할 수 있습니다.

위의 경고와 오류 외에 **RPM**은 **rpmlint** 검사를 통과했습니다.

3.4.3. 유일성을 위해 cello 확인

이 섹션에서는 **cello SPEC** 파일 및 **pello** 바이너리 **RPM**의 예에서 **RPM sanity**를 확인할 때 발생할 수 있는 경고 및 오류를 보여줍니다.

3.4.3.1. cello SPEC 파일 확인

> 예 3.8. cello에 대한 SPEC 파일에서 rpmlint 명령을 실행하는 출력
>
> ```
> $ rpmlint ~/rpmbuild/SPECS/cello.spec
> /home/<username>/rpmbuild/SPECS/cello.spec: W: invalid-url Source0:
> https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP Error 404: Not Found
> 0 packages and 1 specfiles checked; 0 errors, 1 warnings.
> ```

cello.spec 의 경우 하나의 경고만 있습니다. 이 경고는 **Source0** 지시문에 나열된 **URL**에 연결할 수 없음을 나타냅니다. 지정된 **example.com URL**이 존재하지 않기 때문에 이 문제가 예상됩니다. 이 **URL**이 나중에 작동할 것으로 가정하면 이 경고를 무시할 수 있습니다.

> 예 3.9. cello용 SRPM에서 rpmlint 명령을 실행하는 출력
>
> ```
> $ rpmlint ~/rpmbuild/SRPMS/cello-1.0-1.el8.src.rpm
> cello.src: W: invalid-url URL: https://www.example.com/cello HTTP Error 404: Not Found
> cello.src: W: invalid-url Source0: https://www.example.com/cello/releases/cello-1.0.tar.gz
> HTTP Error 404: Not Found
> 1 packages and 0 specfiles checked; 0 errors, 2 warnings.
> ```

cello SRPM의 경우 새로운 경고가 있습니다. 이는 **URL** 지시문에 지정된 **URL** 에 연결할 수 없음을 나타냅니다. 나중에 링크가 작동한다고 가정하면 이 경고를 무시할 수 있습니다.

3.4.3.2. cello 바이너리 RPM 확인

바이너리 **RPM**을 확인할 때 **rpmlint** 는 다음 항목을 확인합니다.

- 문서
- 수동 페이지
- 파일 시스템 계층 구조 표준을 일관되게 사용

> 예 3.10. cello의 바이너리 RPM에서 rpmlint 명령을 실행하는 출력
>
> ```
> $ rpmlint ~/rpmbuild/RPMS/x86_64/cello-1.0-1.el8.x86_64.rpm
> ```

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

no-documentation 및 **no-ECDHE-page-for-binary** 경고는 해당 **RPM**에 문서 또는 수동 페이지가 없음을 나타냅니다. 위의 경고 외에 **RPM**은 **rpmlint** 검사를 통과했습니다.

4장. 고급 주제

이 섹션에서는 소개 튜토리얼의 범위를 벗어나 실제 RPM 패키징에서 유용한 주제를 설명합니다.

4.1. 패키지 서명

패키지는 타사가 콘텐츠를 변경할 수 없도록하기 위해 서명됩니다. 사용자는 패키지를 다운로드할 때 HTTPS 프로토콜을 사용하여 추가 보안 계층을 추가할 수 있습니다.

패키지에 서명하는 세 가지 방법이 있습니다.

4.1.1. GPG 키 생성

절차

1. GPG(GNU 개인 정보 보호 ECDHE) 키 쌍을 생성합니다.

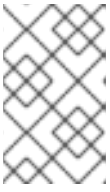
```
# gpg --gen-key
```

2. 생성된 키를 확인하고 확인합니다.

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

3. 공개 키를 내보냅니다.

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



참고

<Key_name> 대신 키에 대해 선택한 실제 이름을 포함합니다.

4. 내보낸 공개 키를 RPM 데이터베이스로 가져옵니다.

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

4.1.2. 기존 패키지에 서명 추가

이 섹션에서는 서명 없이 패키지를 빌드할 때 가장 일반적인 경우를 설명합니다. 서명은 패키지 릴리스 바로 앞에 추가됩니다.

패키지에 서명을 추가하려면 **rpm-sign** 패키지에서 제공하는 **--addsign** 옵션을 사용합니다.

둘 이상의 서명이 있으면 패키지 빌더에서 최종 사용자까지 패키지의 소유권 경로를 기록할 수 있습니다.

절차

- 패키지에 서명을 추가합니다.

```
$ rpm --addsign blather-7.9-1.x86_64.rpm
```

참고

서명의 시크릿 키의 잠금을 해제하려면 암호를 입력해야 합니다.

4.1.3. 여러 서명이 포함된 패키지 서명 확인

절차

- 여러 서명이 있는 패키지 서명을 확인하려면 다음을 실행합니다.

```
$ rpm --checksig blather-7.9-1.x86_64.rpm
blather-7.9-1.x86_64.rpm: size pgp pgp md5 OK
```

rpm --checksig 명령의 출력에 있는 두 **pgp** 문자열은 패키지가 두 번 서명되었음을 표시합니다.

4.1.4. 이미 존재하는 패키지에 서명을 추가하는 실제 예

이 섹션에서는 기존 패키지에 서명을 추가하는 것이 유용할 수 있는 예를 설명합니다.

회사의 분할은 패키지를 만들고 분할의 키를 사용하여 서명합니다. 회사의 서명은 패키지의 서명을 확인하고 패키지에 회사 서명을 추가하여 서명된 패키지가 정품임을 나타냅니다.

두 개의 서명을 사용하면 패키지는 소매 업체로 이동합니다. 리포터는 서명을 확인하고 일치하는 경우 서명을 추가합니다.

이제 패키지를 통해 패키지를 배포하려는 회사로 전환할 수 있습니다. 패키지의 모든 서명을 확인한 후 실제 사본임을 확인합니다. 배포 회사의 내부 제어에 따라 자체 서명을 추가하여 패키지가 회사의 승인을 수신했음을 알릴 수 있습니다.

4.1.5. 기존 패키지에서 서명 교체

다음 절차에서는 각 패키지를 다시 빌드하지 않고도 공개 키를 변경하는 방법을 설명합니다.

절차

- 공개 키를 변경하려면 다음을 실행합니다.

```
$ rpm --resign blather-7.9-1.x86_64.rpm
```

참고

서명의 시크릿 키의 잠금을 해제하려면 암호를 입력해야 합니다.

--resign 옵션을 사용하면 다음 절차에 표시된 대로 여러 패키지의 공개 키를 변경할 수도 있습니다.

절차

- 여러 패키지의 공개 키를 변경하려면 다음을 실행합니다.

```
$ rpm --resign b*.rpm
```

참고

서명의 시크릿 키의 잠금을 해제하려면 암호를 입력해야 합니다.

4.1.6. 빌드 시 패키지에 서명

절차

1. **rpmbuild** 명령을 사용하여 패키지를 빌드합니다.

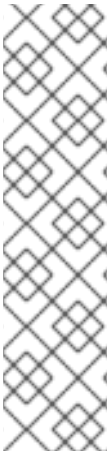
```
$ rpmbuild blather-7.9.spec
```

2. **--addsign** 옵션을 사용하여 **rpmsign** 명령으로 패키지에 서명합니다.

```
$ rpmsign --addsign blather-7.9-1.x86_64.rpm
```

3. 선택적으로 패키지의 서명을 확인합니다.

```
$ rpm --checksig blather-7.9-1.x86_64.rpm
blather-7.9-1.x86_64.rpm: size pgp md5 OK
```



참고

여러 패키지를 빌드하고 서명할 때 다음 구문을 사용하여 **Pretty privacy (PGP)** 암호를 여러 번 입력하지 마십시오.

```
$ rpmbuild -ba --sign b*.spec
```

서명의 시크릿 키 잠금을 해제하려면 암호를 입력해야 합니다.

4.2. 매크로에 대한 추가 정보

이 섹션에서는 선택한 내장 **RPM** 매크로에 대해 설명합니다. 이러한 매크로의 전체 목록은 **RPM** 문서를 참조하십시오.

4.2.1. 사용자 고유의 매크로 정의

다음 섹션에서는 사용자 지정 매크로를 만드는 방법을 설명합니다.

절차

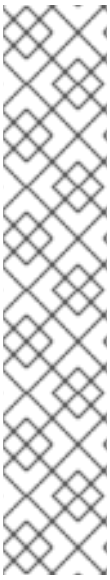
- RPM SPEC 파일에 다음 행을 추가합니다.

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

\ 에 대한 모든 공백이 제거됩니다. name은 영숫자 문자 및 문자 _ 로 구성될 수 있으며 길이가 3자 이상이어야 합니다. (opts) 필드의 포함은 선택 사항입니다.

- 간단한 매크로에는 (opts) 필드가 포함되어 있지 않습니다. 이 경우 재귀 매크로 확장만 수행됩니다.

- 매개 변수가 있는 매크로 에는 (opts) 필드가 포함되어 있습니다. macro 호출 시작 시 argc/argv 처리를 위해 getopt(3) 간 opts 문자열이 전달됩니다.



참고

이전 RPM SPEC 파일은 %define <name> <body> 매크로 패턴을 대신 사용합니다. %define 과 %global 매크로의 차이점은 다음과 같습니다.

- %define 에는 로컬 범위가 있습니다. 이는 SPEC 파일의 특정 부분에 적용됩니다. %define 매크로의 본문이 사용될 때 확장됩니다.

- %global 에는 글로벌 범위가 있습니다. 이는 전체 SPEC 파일에 적용됩니다. %global 매크로의 본문은 정의 시간에 확장됩니다.

중요

매크로는 주석 처리되거나 매크로의 이름이 SPEC 파일의 %changelog 섹션에 제공되더라도 평가됩니다. 매크로를 주석으로 처리하려면 %% 를 사용합니다. 예: %%global.

추가 리소스

매크로 기능에 대한 포괄적인 정보는 RPM 문서를 참조하십시오.

4.2.2. %setup 매크로 사용

이 섹션에서는 %setup 매크로의 다양한 변형을 사용하여 소스 코드 tarballs로 패키지를 빌드하는 방

*법을 설명합니다. 매크로 변형을 결합할 수 있습니다. **rpmbuild** 출력은 **%setup** 매크로의 표준 동작을 보여줍니다. 각 단계의 시작 부분에서 매크로는 아래 예제와 같이 **Executing(%...)** 을 출력합니다.*

> *예 4.1. **%setup** 매크로 출력 예*
>
> ```
> Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.DhddsG
> ```
>
> *쉘 출력은 **set -x** 가 활성화된 상태로 설정됩니다. **/var/tmp/rpm-tmp.DhddsG** 의 내용을 보려면 **rpmbuild** 가 빌드 후 임시 파일을 삭제하므로 **--debug** 옵션을 사용합니다. 그러면 환경 변수 설정이 표시되고 그 뒤에 예를 들면 다음과 같습니다.*
>
> ```
> cd '/builddir/build/BUILD'
> rm -rf 'cello-1.0'
> /usr/bin/gzip -dc '/builddir/build/SOURCES/cello-1.0.tar.gz' | /usr/bin/tar -xof -
> STATUS=$?
> if [ $STATUS -ne 0 ]; then
>   exit $STATUS
> fi
> cd 'cello-1.0'
> /usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
> ```

%setup 매크로:

- *올바른 디렉터리에서 작업하고 있는지 확인합니다.*
- *이전 빌드의 남은 항목을 제거합니다.*
- *소스 **tarball**의 압축을 풉니다.*
- *일부 기본 권한을 설정합니다.*

4.2.2.1. %setup -q 매크로 사용

*-q 옵션은 **%setup** 매크로의 상세 정보를 제한합니다. **tar -xvvof** 대신 **tar -xvof**만 실행됩니다. 첫 번째 옵션으로 이 옵션을 사용합니다.*

4.2.2.2. %setup -n 매크로 사용

n 옵션은 확장된 tarball의 디렉터리 이름을 지정하는 데 사용됩니다.

이는 확장 tarball의 디렉터리에 예상되는 이름과 다른 이름(%{name}-%{version})이 있는 경우, %setup 매크로 오류가 발생할 수 있습니다.

예를 들어 패키지 이름이 cello 이지만 소스 코드가 hello-1.0.tgz 에 보관되고 hello/ 디렉터리가 포함된 경우 SPEC 파일 콘텐츠는 다음과 같아야 합니다.

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

4.2.2.3. %setup -c 매크로 사용

소스 코드 tarball에 하위 디렉터리가 포함되어 있지 않은 경우 -c 옵션이 사용되며 압축을 푼 후 아카이브의 파일이 현재 디렉터리를 채웁니다.

그런 다음 -c 옵션은 아래와 같이 아카이브 확장에 디렉토리와 단계를 생성합니다.

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

아카이브 확장 후에는 디렉터리가 변경되지 않습니다.

4.2.2.4. %setup -D 및 %setup -T 매크로 사용

-D 옵션은 소스 코드 디렉터리 삭제를 비활성화하고 %setup 매크로가 여러 번 사용되는 경우 특히 유용합니다. -D 옵션을 사용하면 다음 행이 사용되지 않습니다.

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

-T 옵션은 스크립트에서 다음 행을 제거하여 소스 코드 tarball의 확장을 비활성화합니다.

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

4.2.2.5. %setup -a 및 %setup -b 매크로 사용

-a 및 -b 옵션은 특정 소스를 확장합니다.

-b 옵션은 이전을 나타내며 작업 디렉터리를 입력하기 전에 특정 소스를 확장합니다. a 옵션은 후를 나타내며 입력 후 해당 소스를 확장합니다. 해당 인수는 SPEC 파일 preamble의 소스 번호입니다.

다음 예에서 cello-1.0.tar.gz 아카이브에는 빈 examples 디렉터리가 포함되어 있습니다. 예제는 별도의 examples.tar.gz tarball에 포함되어 있으며 동일한 이름의 디렉터리로 확장됩니다. 이 경우 작업 디렉터리를 입력한 후 Source1 을 확장하려면 -a 1 을 사용합니다.

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

다음 예에서는 cello-1.0/examples 로 확장되는 별도의 cello-1.0-examples.tar.gz tarball로 예제를 제공합니다. 이 경우 작업 디렉터리를 입력하기 전에 -b 1, 를 사용하여 Source1 을 확장합니다.

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

4.2.3. %files 섹션의 일반적인 RPM 매크로

이 섹션에는 SPEC 파일의 %files 섹션에 필요한 고급 RPM 매크로가 나열되어 있습니다.

표 4.1. %files 섹션의 고급 RPM Macros

Macro	정의
%license	매크로는 LICENSE 파일로 나열된 파일을 식별하며 RPM과 같이 설치되고 레이블이 지정됩니다. 예: **%license LICENSE**
%doc	매크로는 문서로 나열된 파일을 식별하며 RPM과 같이 설치되고 레이블이 지정됩니다. 매크로는 패키지 소프트웨어에 대한 문서와 코드 예제 및 다양한 관련 항목에 대해 사용됩니다. 이벤트 코드 예제가 포함된 경우 파일에서 실행 모드를 제거하려면 주의해야 합니다. 예: **%doc README**

Macro	정의
%dir	매크로를 사용하면 경로가 이 RPM이 소유한 디렉터리임을 확인할 수 있습니다. RPM 파일 매니페스트가 제거 시 정리할 디렉토리를 정확하게 알고 있도록 하는 것이 중요합니다. 예: **%dir %{_libdir}/%{name}**
%config(noreplace)	매크로는 다음 파일이 구성 파일이므로 원래 설치 체크섬에서 파일이 수정된 경우 패키지 설치 또는 업데이트에서 덮어쓰거나 교체해서는 안 됩니다. 변경이 있는 경우 대상 시스템의 기존 또는 수정된 파일이 변경되지 않도록 업그레이드 또는 설치 시 파일 이름 끝에 **.rpmnew** 가 추가됩니다. 예:**%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

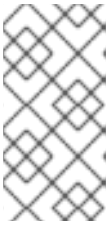
4.2.4. 기본 제공 매크로 표시

*여러 기본 제공되는 **RPM** 매크로를 제공합니다.*

절차

1. *기본 제공 **RPM** 매크로를 모두 표시하려면 다음을 실행합니다.*

```
rpm --showrc
```



참고

*출력은 크기가 매우 적습니다. 결과를 좁히려면 위의 명령을 **grep** 명령과 함께 사용합니다.*

2. *시스템 버전의 **RPM**에 대한 **RPM**에 대한 정보를 찾으려면 다음을 실행합니다.*

```
rpm -ql rpm
```

참고

***RPM** 매크로는 출력 디렉터리 구조에서 매크로 라는 파일입니다.*

4.2.5. RPM 배포 매크로

다른 배포는 패키지화된 소프트웨어의 언어 구현 또는 배포의 특정 지침에 따라 다양한 권장 **RPM** 매크로 세트를 제공합니다.

권장되는 **RPM** 매크로 세트는 종종 **RPM** 패키지로 제공되며 **yum** 패키지 관리자를 사용하여 설치할 준비가 되어 있습니다.

설치가 완료되면 **/usr/lib/rpm/macros.d/** 디렉터리에 매크로 파일을 찾을 수 있습니다.

원시 **RPM** 매크로 정의를 표시하려면 다음을 실행합니다.

```
rpm --showrc
```

위의 출력에는 원시 **RPM** 매크로 정의가 표시됩니다.

매크로의 기능과 **RPM**을 패키징할 때 유용할 수 있는 방법을 확인하려면 인수로 사용되는 매크로의 이름으로 **rpm --eval** 명령을 실행합니다.

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

자세한 내용은 **rpm man** 페이지를 참조하십시오.

4.2.5.1. 사용자 정의 매크로 생성

사용자 지정 매크로를 사용하여 **~/.rpmmacros** 파일에서 배포 매크로를 덮어쓸 수 있습니다. 변경한 모든 변경 사항은 머신의 모든 빌드에 영향을 미칩니다.

주의

~/.rpmmacros 파일에 새 매크로를 정의하는 것은 권장되지 않습니다. 이러한 매크로는 다른 시스템에는 존재하지 않으며 사용자가 패키지를 다시 빌드하려고 할 수 있습니다.

매크로를 재정의하려면 다음을 실행합니다.

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

rpmdev-setuptree 유틸리티를 통한 모든 하위 디렉토리를 포함하여 위의 예제에서 디렉토리를 만들 수 있습니다. 이 매크로의 값은 기본적으로 **~/rpmbuild** 입니다.

```
%_smp_mflags -l3
```

위의 매크로는 종종 **Makefile**에 전달하는 데 사용됩니다(예: **%{?_smp_mflags}**) 빌드 단계에서 여러 동시 프로세스를 설정합니다. 기본적으로 **-jX** 로 설정됩니다. 여기서 **X** 는 코어 수입니다. 코어 수를 변경하면 패키지 빌드의 속도를 높이거나 느려질 수 있습니다.

4.3. EPOCH, SCRIPTLETS 및 TRIGGERS

이 섹션에서는 **RMP SPEC** 파일의 고급 지시문을 나타내는 **Epoch,Scriptlets,** 및 **Trigger** 에 대해 설명합니다.

이러한 모든 지시문은 **SPEC** 파일뿐만 아니라 결과 **RPM**이 설치된 최종 시스템에도 영향을 미칩니다.

4.3.1. Epoch 지시문

Epoch 지시문을 사용하면 버전 번호에 따라 가중치가 지정된 종속성을 정의할 수 있습니다.

RPM SPEC 파일에 이 지시문이 나열되지 않으면 **Epoch** 지시문이 전혀 설정되지 않습니다. 이는 **Epoch** 를 **0**으로 설정하지 않는 일반적인 추측 입니다. 그러나 **YUM** 유틸리티는 해독 목적으로 설정되지 않은 **Epoch** 를 **0**의 **Epoch** 와 동일하게 처리합니다.

그러나 대부분의 경우 **Epoch** 값이 패키지 버전을 비교할 때 예상되는 **RPM** 동작을 도입하기 때문에 **SPEC** 파일에 **Epoch** 를 나열하는 것은 일반적으로 생략됩니다.

예 **4.2. Epoch** 사용

Epoch: 1 및 버전**: 1.0** 과 함께 **foobar** 패키지를 설치하고 다른 사람이 버전**: 2.0** 을 사용하여 **foobar** 패키지인 경우 **Epoch** 지시문이 없는 경우 새 버전은 업데이트로 간주되지 않습니다. **Epoch**

> *버전이 RPM 패키지에 대한 버전 관리를 나타내는 기존 Name-Version-Release 마커보다 우선하기 때문입니다.*

따라서 Epoch 를 사용하는 것은 매우 드문 경우입니다. 그러나 일반적으로 Epoch 는 업그레이드 순서 문제를 해결하는 데 사용됩니다. 이 문제는 인코딩을 기반으로 항상 안정적으로 비교할 수 없는 알파벳 순 문자를 포함하는 소프트웨어 버전 번호 체계 또는 버전을 포함하는 업스트림 변경의 부작용을 나타낼 수 있습니다.

4.3.2. scriptlets

***scriptlets** 는 패키지 설치 또는 삭제 전이나 후에 실행되는 일련의 RPM 지시문입니다.*

***Scriptlets** 는 빌드 시 또는 시작 스크립트에서는 수행할 수 없는 작업에만 사용합니다.*

4.3.2.1. scriptlets 지시문

일반적인 Scriptlet 지시문 세트가 있습니다. SPEC 파일 섹션 헤더(예: %build 또는 %install)와 유사합니다. 이는 여러 줄 코드 세그먼트에 의해 정의되며 종종 표준 POSIX 쉘 스크립트로 작성됩니다. 그러나 대상 머신의 배포에 대한 RPM이 허용하는 다른 프로그래밍 언어로도 작성할 수 있습니다. RPM 설명서에는 사용 가능한 언어의 전체 목록이 포함되어 있습니다.

다음 표에는 실행 순서에 나열된 Scriptlet 지시문이 포함되어 있습니다. 스크립트가 포함된 패키지는 %pre 및 %post 지시문 간에 설치되며 %preun 과 %postun 지시문 간에 제거됩니다.

표 4.2. Scriptlet 지시문

directive	정의
%pretrans	패키지를 설치하거나 제거하기 직전에 실행되는 Scriptlet입니다.
%pre	대상 시스템에 패키지를 설치하기 직전에 실행되는 Scriptlet입니다.
%post	대상 시스템에 패키지가 설치된 직후에 실행되는 Scriptlet입니다.
%preun	대상 시스템에서 패키지를 제거하기 직전에 실행되는 Scriptlet입니다.
%postun	대상 시스템에서 패키지가 제거된 직후에 실행되는 Scriptlet입니다.
%posttrans	트랜잭션 마지막에 실행되는 Scriptlet입니다.

4.3.2.2. scriptlet 실행 끄기

스크립트릿 실행을 끄려면 **rpm** 명령을 **--no_scriptlet_name_** 옵션과 함께 사용합니다.

절차

- 예를 들어 **%pretrans** 스크립트릿의 실행을 끄려면 다음을 실행합니다.

```
# rpm --nopretrans
```

  다음과 같은 **--noscripts** 옵션을 사용할 수도 있습니다.

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

추가 리소스

- 자세한 내용은 **rpm(8)** 매뉴얼 페이지를 참조하십시오.

4.3.2.3. scriptlets 매크로

Scriptlets 지시문도 **RPM** 매크로에서 작동합니다.

다음 예제에서는 **systemd scriptlet** 매크로를 사용하여 새 장치 파일에 대해 **systemd**에 대한 알림을

받는 방법을 보여줍니다.

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

$ rpm --eval %{systemd_post}

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

$ rpm --eval %{systemd_postun}

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

$ rpm --eval %{systemd_preun}

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

4.3.3. Triggers 지시문

*트리거 는 패키지 설치 및 제거 중 상호 작용을 위한 방법을 제공하는 **RPM** 지시문입니다.*

주의

트리거 는 예기치 않은 시간에 실행될 수 있습니다(예: 포함 패키지의 업데이트). 트리거 는 디버그하기 어렵기 때문에 예기치 않게 실행될 때 아무것도 중단하지 않도록 강력한 방식으로 구현해야 합니다. 이러한 이유로 **{RH}**는 트리거 사용을 최소화할 것을 권장합니다.

실행 순서 및 각 기존 **Trigger** 의 세부 정보는 다음과 같습니다.

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

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

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

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

위의 항목은 **/usr/share/doc/rpm-4.*/triggers** 파일에 있습니다.

4.3.4. SPEC 파일에서 쉘 이외의 스크립트 사용

SPEC 파일의 **-p scriptlet** 옵션을 사용하면 사용자가 기본 쉘 스크립트 인터프리터**(-p /bin/sh)** 대신 특정 인터프리터를 호출할 수 있습니다.

다음 절차에서는 **pello.py** 프로그램을 설치한 후 메시지를 출력하는 스크립트를 생성하는 방법을 설명합니다.

절차

1. ***pello.spec** 파일을 엽니다.*

2. *다음 행을 찾습니다.*

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

3. *위의 줄에서 다음을 삽입합니다.*

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

4. *패키지를 설치합니다.*

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

5. *설치 후 출력 메시지를 확인합니다.*

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

참고

*Python 3 스크립트를 사용하려면 **install -m** 아래에 **SPEC** 파일에 다음 행을 포함합니다.*

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

*Lua 스크립트를 사용하려면 **install -m** 아래에 **SPEC** 파일에 다음 행을 포함합니다.*

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

*이렇게 하면 **SPEC** 파일에서 모든 인터프리터를 지정할 수 있습니다.*

4.4. RPM 조건

RPM 조건부를 사용하면 SPEC 파일의 다양한 섹션을 조건부로 포함할 수 있습니다.

일반적으로 조건부 포함은 다음을 처리합니다.

- 아키텍처별 섹션
- 운영 체제별 섹션
- 다양한 운영 체제 버전 간의 호환성 문제
- 매크로의 존재 및 정의

4.4.1. RPM 조건 구문

RPM 조건의 경우 다음 구문을 사용합니다.

expression 이 true이면 몇 가지 작업을 수행합니다.

```
%if expression
…
%endif
```

expression 이 true인 경우 다른 경우에는 다른 작업을 수행합니다.

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

4.4.2. RPM 조건 예

*이 섹션에서는 **RPM** 조건의 여러 예를 제공합니다.*

4.4.2.1. %if 조건

> *예 4.3. 8 및 기타 운영 체제 간의 호환성을 처리하기 위해 %if 조건을 사용*
>
> ```
> %if 0%{?rhel} == 8
> sed -i '/AS_FUNCTION_DESCRIBE/ s/^//' configure.in sed -i '/AS_FUNCTION_DESCRIBE/
> s/^//' acinclude.m4
> %endif
> ```

*이 조건은 **AS_FUNCTION_DESCRIBE** 매크로 지원 측면에서 **RHEL 8**과 기타 운영 체제 간의 호환성을 처리합니다. **RHEL**용으로 패키지를 빌드하면 **%rhel** 매크로가 정의되고 **RHEL** 버전으로 확장됩니다. 값이 **8**이면 **RHEL 8**에 대한 패키지가 빌드되는 것을 나타내는 경우 **RHEL 8**에서 지원하지 않는 **AS_FUNCTION_DESCRIBE**에 대한 참조가 **autoconfig** 스크립트에서 삭제됩니다.*

> *예 4.4. 매크로 정의를 처리하기 위해 %if 조건을 사용*
>
> ```
> %define ruby_archive %{name}-%{ruby_version}
> %if 0%{?milestone:1}%{?revision:1} != 0
> %define ruby_archive %{ruby_archive}-%{?milestone}%{?!milestone:%{?revision:r%
> {revision}}}
> %endif
> ```

*이 조건부에서는 매크로에 대한 정의를 처리합니다. **%milestone** 또는 **%revision** 매크로가 설정되면 업스트림 **tarball**의 이름을 정의하는 **%ruby_archive** 매크로가 다시 정의됩니다.*

4.4.2.2. %if 조건의 특수 변형

***%ifarch** 조건, **%ifnarch** 조건 및 **%ifos** 조건부는 **%if** 조건의 특수 변형입니다. 이러한 변형은 일반적으로 사용되며, 따라서 자체 매크로가 있습니다.*

4.4.2.2.1. %ifarch 조건

***%ifarch** 조건은 아키텍처별 **SPEC** 파일 블록을 시작하는 데 사용됩니다. 그런 다음 각각 쉼표 또는 공백으로 구분된 하나 이상의 아키텍처 **licenses**가 있습니다.*

> *예 4.5. %ifarch 조건 사용 예*

```
%ifarch i386 sparc
…
%endif
```

%ifarch 와 %endif 사이의 SPEC 파일의 모든 내용은 32비트 AMD 및 Intel 아키텍처 또는 SunECDHE 기반 시스템에서만 처리됩니다.

4.4.2.2.2. %ifnarch 조건

%ifnarch 조건에는 %ifarch 조건보다 역방향 논리가 있습니다.

예 4.6. %ifnarch 조건 사용 예

```
%ifnarch alpha
…
%endif
```

%ifnarch 와 %endif 사이의 SPEC 파일의 모든 내용은 디지털 알파/AXP 기반 시스템에서 수행되지 않은 경우에만 처리됩니다.

4.4.2.2.3. %ifos 조건

%ifos 조건은 빌드의 운영 체제에 따라 처리를 제어하는 데 사용됩니다. 그런 다음 하나 이상의 운영 체제 이름을 지정할 수 있습니다.

예 4.7. %ifos 조건 사용 예

```
%ifos linux
…
%endif
```

%ifos 와 %endif 사이의 SPEC 파일의 모든 내용은 Linux 시스템에서 빌드가 수행된 경우에만 처리됩니다.

부록 A. RHEL 7의 RPM 새로운 기능

이 목록에는 **Red Hat Enterprise Linux 6**와 **7** 간의 **RPM** 패키징에서 가장 눈에 띄는 변경 사항이 설명되어 있습니다.

- 인증 키 가져오기 및 서명 확인에 사용되는 새로운 명령 **rpmkeys** 가 추가되었습니다.
- 사양 쿼리 및 구문 분석 출력에 사용되는 새로운 명령 **rpmspec** 이 추가되었습니다.
- 패키지 서명에 사용되는 새로운 명령 **rpmsign** 이 추가되었습니다.
- **%{lua:...}** 스크립트에 포함된 **posix.exec()** 및 **os.exit()** 확장은 **posix.fork()** 스크립트로 생성된 하위 프로세스에서 호출하지 않는 한 스크립트에 실패합니다.
- **%pretrans scriptlet** 실패로 인해 패키지를 건너뜁니다.
- **scriptlets**는 런타임에 매크로 확장 및 **queryformat-expanded**일 수 있습니다.
- 이제 **Requires(pretrans)** 및 **Requires(posttrans) scriptlets**를 사용하여 **pre-ECDHE** 및 **post-ECDHE scriptlet** 종속성을 올바르게 표시할 수 있습니다.
- 주문 추가 힌트를 제공하기 위한 **OrderWithRequires** 태그가 추가되었습니다. 태그는 **Requires** 태그 구문을 따르지만 실제 종속성은 생성하지 않습니다. 순서 지정 힌트는 관련 패키지가 동일한 트랜잭션에 있는 경우에만 트랜잭션 순서를 계산할 때 필요한 것처럼 취급됩니다.
- **%license** 플래그는 **%files** 섹션에서 사용할 수 있습니다. 이 플래그는 **%doc** 플래그와 유사하게 사용하여 파일을 라이센스로 표시할 수 있으며 **--nodocs** 옵션에도 불구하고 설치해야 합니다.
- 선택적 분산 버전 제어 시스템 통합과 함께 패치 애플리케이션을 자동화하는 **%autosetup** 매크로가 추가되었습니다.
- 자동 종속성 생성기는 기본 제공 필터링이 포함된 확장 가능하고 사용자 정의 가능한 규칙 기반 시스템으로 다시 작성되었습니다.

- ***OpenPGP V3** 공개 키는 더 이상 지원되지 않습니다.*

5장. RPM 패키지에 대한 추가 리소스

이 섹션에서는 RPM, RPM 패키징 및 RPM 빌드와 관련된 다양한 주제에 대한 참조를 제공합니다. 이 중 일부는 이 문서에 포함된 소개 자료의 고급 기능을 제공합니다.

Red Hat Software Collections 개요 - Red Hat Software Collections 오퍼링은 안정적인 최신 버전에서 지속적으로 업데이트된 개발 툴을 제공합니다.

Red Hat 소프트웨어 컬렉션 - 패키징 가이드에서는 소프트웨어 컬렉션에 대한 설명과 이를 빌드하고 패키징하는 방법에 대해 자세히 설명합니다. RPM으로 소프트웨어 패키징에 대한 기본적인 이해가 있는 개발자 및 시스템 관리자는 이 가이드를 사용하여 Software Collections를 시작할 수 있습니다.

모크 - **Mock** 은 빌드 호스트보다 다양한 아키텍처 및 다른 Fedora 또는 RHEL 버전을 위한 커뮤니티 지원 패키지 구축 솔루션을 제공합니다.

RPM 문서 - 공식 RPM 설명서.

Fedora 패키징 지침 - Fedora의 공식 패키징 지침으로, 모든 RPM 기반 배포에 유용합니다.