Red Hat Enterprise Linux 9

소프트웨어 패키징 및 배포

RPM 패키지 관리 시스템을 사용하여 소프트웨어 패키지 패키지

Last Updated: 2024-08-20

Red Hat Enterprise Linux 9 소프트웨어 패키징 및 배포

RPM 패키지 관리 시스템을 사용하여 소프트웨어 패키지 패키지

법적 공지

초록

RPM 패키지 관리자를 사용하여 소프트웨어를 RPM 패키지로 패키징합니다. 패키징, 패키지 소프트웨어용으로 소스 코드를 준비하고, Python 프로젝트 패키징 또는 RubyGems와 같은 고급 패키징 시나리오를 RPM 패키지로 조사합니다.

차례

RED HAT 문서에 관한 피드백 제공

문서에 대한 피드백에 감사드립니다. 어떻게 개선할 수 있는지 알려주십시오.

Jira를 통해 피드백 제출 (등록 필요)

1. Jira 웹 사이트에 로그인합니다.
2. 상단 탐색 모음에서 **생성** 을 클릭합니다.
3. **Summary** (요약) 필드에 설명 제목을 입력합니다.
4. **Description** (설명) 필드에 개선을 위한 제안을 입력합니다. 문서의 관련 부분에 대한 링크를 포함합니다.
5. 대화 상자 하단에서 **생성** 을 클릭합니다.

1장. RPM 소개

RPM(RPM)은 RHEL(Red Hat Enterprise Linux), CentOS 및 Fedora에서 실행되는 패키지 관리 시스템입니다. RPM을 사용하여 이러한 운영 체제에 대해 생성하는 소프트웨어를 배포, 관리 및 업데이트할 수 있습니다.

RPM 패키지 관리 시스템은 기존 아카이브 파일에 소프트웨어를 배포하는 것보다 다음과 같은 이점이 있습니다.

- RPM은 서로 독립적으로 설치, 업데이트 또는 제거할 수 있는 패키지 형태로 소프트웨어를 관리하므로 운영 체제를 보다 쉽게 유지 관리할 수 있습니다.
- RPM 패키지는 압축된 아카이브와 유사하게 독립 실행형 바이너리 파일이므로 RPM을 단순화합니다. 이러한 패키지는 특정 운영 체제 및 하드웨어 아키텍처를 위해 빌드됩니다. RPM에는 패키지가 설치될 때 파일 시스템의 적절한 경로에 배치되는 컴파일된 실행 파일 및 라이브러리와 같은 파일이 포함되어 있습니다.

RPM을 사용하면 다음 작업을 수행할 수 있습니다.

- 패키지 소프트웨어를 설치, 업그레이드 및 제거합니다.
- 패키지 소프트웨어에 대한 자세한 정보를 쿼리합니다.
- 패키지 소프트웨어의 무결성을 확인합니다.
- 소프트웨어 소스에서 자체 패키지를 빌드하고 빌드 지침을 완료합니다.
- GPG(GNU Privacy Guard) 유틸리티를 사용하여 패키지에 디지털 서명합니다.
- DNF 리포지토리에 패키지를 게시합니다.

Red Hat Enterprise Linux에서 RPM은 DNF 또는 PackageKit과 같은 고급 패키지 관리 소프트웨어에 완전히 통합되어 있습니다. RPM은 자체 명령줄 인터페이스를 제공하지만 대부분의 사용자는 이 소프트웨어를 통해서만 RPM과 상호 작용해야 합니다. 그러나 RPM 패키지를 빌드할 때는 **rpmbuild(8)** 와 같은 RPM 유틸리티를 사용해야 합니다.

1.1. RPM 패키지

RPM 패키지는 이러한 파일을 설치 및 삭제하는 데 사용되는 파일 및 메타데이터의 아카이브로 구성됩니다. 특히 RPM 패키지에는 다음 부분이 포함되어 있습니다.

GPG 서명

GPG 서명은 패키지의 무결성을 확인하는 데 사용됩니다.

헤더(패키지 메타데이터)

RPM 패키지 관리자는 이 메타데이터를 사용하여 패키지 종속성, 파일 설치 위치 및 기타 정보를 확인합니다.

페이로드

페이로드는 시스템에 설치할 파일이 포함된 **cpio** 아카이브입니다.

RPM 패키지에는 두 가지 유형이 있습니다. 두 유형 모두 파일 형식과 툴링을 공유하지만 콘텐츠가 다르며 다른 용도로 사용됩니다.

- 소스 RPM(SRPM)

SRPM에는 소스 코드와 **사양** 파일이 포함되어 있으며, 바이너리 RPM에 소스 코드를 빌드하는 방법을 설명합니다. 선택적으로 SRPM은 소스 코드에 대한 패치를 포함할 수 있습니다.

바이너리 RPM

바이너리 RPM에는 소스 및 패치에서 빌드된 바이너리가 포함되어 있습니다.

1.2. RPM 패키징 유틸리티 나열

RPM은 패키지 빌드를 위한 **rpmbuild(8)** 프로그램 외에도 다른 유틸리티를 제공하여 패키지 생성 프로세스를 보다 쉽게 수행할 수 있도록 합니다. 이러한 프로그램은 **rpmdevtools** 패키지에서 찾을 수 있습니다.

사전 요구 사항

- **rpmdevtools** 패키지가 설치되었습니다.

```
# dnf install rpmdevtools
```

절차

- 다음 방법 중 하나를 사용하여 RPM 패키징 유틸리티를 나열합니다.
  - **rpmdevtools** 패키지에서 제공하는 특정 유틸리티와 짧은 설명을 나열하려면 다음을 입력합니다.

    ```
    $ rpm -qi rpmdevtools
    ```

  - 모든 유틸리티를 나열하려면 다음을 입력합니다.

    ```
    $ rpm -ql rpmdevtools | grep ^/usr/bin
    ```

추가 리소스

- RPM 유틸리티 도움말 페이지

2장. RPM 패키징 소프트웨어 생성

RPM 패키징용 소프트웨어를 준비하려면 소스 코드가 무엇이며 소프트웨어 생성 방법을 이해해야 합니다.

2.1. 소스 코드란

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

세 가지 프로그래밍 언어로 작성된 **Hello World** 프로그램의 다음 버전은 주요 RPM 패키지 관리자 사용 사례를 다룹니다.

- **bash로 작성된 hello world**
  bello 프로젝트는 Bash 에서 **Hello World** 를 구현합니다. 구현에는 **bello** 쉘 스크립트만 포함됩니다. 이 프로그램의 목적은 명령줄에서 **Hello World** 를 출력하는 것입니다.

  벨로 **파일에는** 다음 내용이 있습니다.

  ```
  #!/bin/bash

  printf "Hello World\n"
  ```

- **Hello World** 는 Python으로 작성됩니다.
  pello 프로젝트는 Python 에서 **Hello World** 를 구현합니다. 구현에는 **pello.py** 프로그램만 포함됩니다. 프로그램의 목적은 명령행에 **Hello World** 를 출력하는 것입니다.

  pello.py 파일에는 다음과 같은 내용이 있습니다.

  ```
  #!/usr/bin/python3

  print("Hello World")
  ```

- **Hello World** 에서 C로 작성됩니다.
  cello 프로젝트는 **Hello World** in C를 구현합니다. 구현에는 **cello.c** 및 **Makefile** 파일만 포함됩니다. 따라서 생성된 **tar.gz** 아카이브에는 **LICENSE** 파일 외에 두 개의 파일이 있습니다. 프로그램의 목적은 명령줄에서 **Hello World** 를 출력하는 것입니다.

  cello.c 파일에는 다음과 같은 내용이 있습니다.

  ```
  #include <stdio.h>

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

참고

패키징 프로세스는 **Hello World** 프로그램의 각 버전에 따라 다릅니다.

2.2. 소프트웨어 생성 방법

다음 방법 중 하나를 사용하여 사람이 읽을 수 있는 소스 코드를 머신 코드로 변환할 수 있습니다.

- 기본적으로 소프트웨어를 컴파일합니다.
- 언어 인터프리터 또는 언어 가상 머신을 사용하여 소프트웨어를 해석합니다. raw-interpret 또는 byte-compile 소프트웨어 중 하나를 사용할 수 있습니다.

2.2.1. 기본적으로 컴파일된 소프트웨어

기본적으로 컴파일된 소프트웨어는 결과 바이너리 실행 파일을 사용하여 머신 코드로 컴파일되는 프로그래밍 언어로 작성된 소프트웨어입니다. 기본적으로 컴파일된 소프트웨어는 독립 실행형 소프트웨어입니다.

참고

기본적으로 컴파일된 RPM 패키지는 아키텍처에 따라 다릅니다.

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

2.2.2. 해석된 소프트웨어

Bash 또는 Python 과 같은 일부 프로그래밍 언어는 머신 코드로 컴파일되지 않습니다. 대신 언어 인터프리터 또는 언어 가상 머신은 사전 변환 없이 프로그램의 소스 코드를 단계별로 실행합니다.

참고

해석된 프로그래밍 언어로 전체적으로 작성된 소프트웨어는 아키텍처에 국한되지 않습니다. 따라서 결과 RPM 패키지에는 이름에 **noarch** 문자열이 있습니다.

해석된 언어로 작성된 raw-interpret 또는 byte-compile 소프트웨어 중 하나를 사용할 수 있습니다.

- 원시 중단 소프트웨어
  이러한 유형의 소프트웨어를 컴파일할 필요는 없습니다. 원시 해석 소프트웨어는 인터프리터에 의해 직접 실행됩니다.
- 바이트로 컴파일된 소프트웨어
  먼저 이 유형의 소프트웨어를 바이트 코드로 컴파일해야 하며, 그런 다음 언어 가상 머신에서 실행해야 합니다.

  참고

  일부 바이트로 컴파일된 언어는 원시 해석 또는 바이트 컴파일될 수 있습니다.

RPM을 사용하여 소프트웨어를 빌드하고 패키지하는 방법은 두 가지 소프트웨어 유형에 따라 다릅니다.

2.3. 소스에서 소프트웨어 빌드

소프트웨어 빌드 프로세스 중에 소스 코드는 RPM을 사용하여 패키징할 수 있는 소프트웨어 아티팩트로 변환됩니다.

2.3.1. 기본적으로 컴파일된 코드에서 소프트웨어 빌드

다음 방법 중 하나를 사용하여 컴파일된 언어로 작성된 소프트웨어를 실행 파일로 빌드할 수 있습니다.

- 수동 빌딩
- 자동화된 빌드

2.3.1.1. 수동으로 샘플 C 프로그램 빌드

수동 빌드를 사용하여 컴파일된 언어로 작성된 소프트웨어를 빌드할 수 있습니다.

C (**cello.c**)로 작성된 샘플 **Hello World** 프로그램에는 다음과 같은 내용이 있습니다.

```
#include <stdio.h>

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

절차

1. GNU 컴파일러 컬렉션에서 C 컴파일러를 호출하여 소스 코드를 바이너리로 컴파일합니다.

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

2. 결과 바이너리 셀로를 실행합니다.

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

2.3.1.2. 샘플 C 프로그램에 대한 자동화된 빌드 설정

대규모 소프트웨어는 일반적으로 자동화된 빌드를 사용합니다. Makefile 파일을 만든 다음 GNU make 유틸리티를 실행하여 자동화된 빌드를 설정할 수 있습니다.

절차

1. **cello.c 와 동일한 디렉토리에 다음 콘텐츠를 사용하여 Makefile 파일을 만듭니다.**

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

cello: 및 clean 아래의 줄은 탭 문자(tab)로 시작해야 합니다.

2. 소프트웨어를 빌드합니다.

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

3. 빌드는 현재 디렉터리에서 이미 사용할 수 있으므로 **make clean** 명령을 입력한 다음 **make** 명령을 다시 입력합니다.

```
$ make clean
rm cello

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

이 시점에서 프로그램을 다시 빌드하려는 경우 **GNU make** 시스템이 기존 바이너리를 감지하므로 영향을 미치지 않습니다.

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

4. 프로그램을 실행합니다.

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

2.3.2. 소스 코드 해석

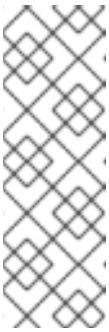
해석된 프로그래밍 언어로 작성된 소스 코드를 다음 방법 중 하나를 사용하여 머신 코드로 변환할 수 있습니다.

- 바이트 컴파일

  바이트 컴파일 소프트웨어의 절차는 다음 요인에 따라 다릅니다.

  - 프로그래밍 언어
  - 언어의 가상 머신

- 해당 언어로 사용되는 툴 및 프로세스



참고

예를 들어 Python 에서 작성된 소프트웨어를 바이트로 작성할 수 있습니다. 배포를 위해 설계된 Python 소프트웨어는 종종 바이트로 컴파일되지만 이 문서에서 설명하는 방식에는 포함되지 않습니다. 설명된 절차는 커뮤니티 표준을 준수하는 것이 아니라 간단하게 하는 것을 목표로 합니다. 실제 Python 지침은 소프트웨어 패키징 및 배포를 참조하십시오.

또한 원시 해석 Python 소스 코드도 사용할 수 있습니다. 그러나 바이트로 컴파일된 버전이 더 빠릅니다. 따라서 RPM 패키지는 최종 사용자에게 배포하기 위해 바이트로 컴파일된 버전을 패키징하는 것을 선호합니다.

- **Raw-interpreting**

  Bash 와 같은 쉘 스크립팅 언어로 작성된 소프트웨어는 항상 원시 해석에 의해 실행됩니다.

2.3.2.1. 샘플 Python 프로그램 바이트 컴파일

Python 소스 코드의 원시 해석을 통해 바이트 컴파일을 선택하면 더 빠른 소프트웨어를 만들 수 있습니다.

Python 프로그래밍 언어(pello.py)로 작성된 샘플 Hello World 프로그램에는 다음과 같은 내용이 있습니다.

```
print("Hello World")
```

절차

1. pello.py 파일을 바이트 압축합니다.

```
$ python -m compileall pello.py
```

2. 바이트로 컴파일된 파일 버전이 생성되었는지 확인합니다.

```
$ ls __pycache__
pello.cpython-311.pyc
```

출력의 패키지 버전은 설치된 **Python** 버전에 따라 다를 수 있습니다.

3. **pello.py** 에서 프로그램을 실행합니다.

```
$ python pello.py
Hello World
```

2.3.2.2. 샘플 Bash 프로그램 원시 해석

Bash 쉘로 내장된 언어(**llo**)로 작성된 샘플 **Hello World** 프로그램에는 다음과 같은 내용이 있습니다.

```
#!/bin/bash

printf "Hello World\n"
```

참고

벨로 파일의 상단에 있는 **shebang (#!)**기호는 프로그래밍 언어 소스 코드의 일부가 아닙니다.

shebang 을 사용하여 텍스트 파일을 실행 파일로 전환합니다. 시스템 프로그램 로더는 **shebang** 이 포함된 행을 구문 분석하여 바이너리 실행 파일의 경로를 가져온 다음 프로그래밍 언어 인터프리터로 사용됩니다.

절차

1. 소스 코드로 파일을 실행 가능하게 만듭니다.

```
$ chmod +x bello
```

2. 생성된 파일을 실행합니다.

```
$ ./bello
Hello World
```

3장. RPM 패키지 소프트웨어 준비

RPM을 사용하여 패키징할 소프트웨어를 준비하려면 먼저 소프트웨어를 패치하고 LICENSE 파일을 만들고 tarball로 보관하면 됩니다.

3.1. 패치 소프트웨어

소프트웨어를 패키징할 때는 버그 수정 또는 구성 파일 변경과 같은 원래 소스 코드를 변경해야 할 수 있습니다. RPM 패키징에서는 원래 소스 코드를 그대로 두고 패치를 적용할 수 있습니다.

패치는 소스 코드 파일을 업데이트하는 텍스트입니다. 패치는 두 버전의 텍스트의 차이를 나타내기 때문에 *diff* 형식이 있습니다. diff 유틸리티를 사용하여 패치를 생성한 다음 patch 유틸리티를 사용하여 소스 코드에 패치를 적용할 수 있습니다.

참고

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

3.1.1. 샘플 C 프로그램에 대한 패치 파일 생성

diff 유틸리티를 사용하여 원래 소스 코드에서 패치를 생성할 수 있습니다. 예를 들어 C (cello.c)로 작성된 Hello world 프로그램을 패치하려면 다음 단계를 완료합니다.

사전 요구 사항

- 시스템에 diff 유틸리티를 설치했습니다.

```
# dnf install diffutils
```

절차

1. 원래 소스 코드를 백업합니다.

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

p 옵션은 모드, 소유권 및 타임스탬프를 유지합니다.

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. 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 줄은 **-** 로 시작하는 행을 바꿉니다.

참고

대부분의 사용 사례에 적합하므로 **diff** 명령과 함께 **n ur** 옵션을 사용하는 것이 좋습니다.

- **- n (--new-file)**

  N 옵션은 없는 파일을 빈 파일로 처리합니다.

- **-a (--text)**

  a 옵션은 모든 파일을 텍스트로 처리합니다. 결과적으로 **diff** 유틸리티는 바이너리로 분류된 파일을 무시하지 않습니다.

- **-u (-U NUM 또는 --unified[=NUM])**

  -u 옵션은 통합 컨텍스트의 출력 **NUM**(기본값 **3**) 줄로 출력을 반환합니다. 이는 패치 파일에서 일반적으로 사용되는 컴팩트하고 쉽게 읽을 수 있는 형식입니다.

- **-r (--recursive)**

  r 옵션은 **diff** 유틸리티에서 발견된 하위 디렉토리를 재귀적으로 비교합니다.

그러나 이 특별한 경우에는 **-u** 옵션만 필요합니다.

4. 패치를 파일에 저장합니다.

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

5. 원본 **cello.c** 를 복원 :

```
$ mv cello.c.orig cello.c
```

중요

RPM 패키지를 빌드할 때 RPM 패키지 관리자가 수정된 파일이 아닌 원본 파일을 사용하므로 원래 cello.c 를 유지해야 합니다. 자세한 내용은 사양 파일 작업을 참조하십시오.

추가 리소스

- diff(1) 도움말 페이지

3.1.2. 샘플 C 프로그램 패치

소프트웨어에 코드 패치를 적용하려면 patch 유틸리티를 사용할 수 있습니다.

사전 요구 사항

- 시스템에 패치 유틸리티를 설치했습니다.

```
# dnf install patch
```

- 원본 소스 코드에서 패치를 생성했습니다. 자세한 내용은 샘플 C 프로그램에 대한 패치 파일 생성을 참조하십시오.

절차

다음 단계는 cello.c 파일에 이전에 생성된 cello.patch 파일을 적용합니다.

1. 패치 파일을 패치 명령으로 리디렉션합니다.

```
$ patch < cello.patch
patching file cello.c
```

2. cello.c 의 내용이 이제 원하는 변경 사항을 반영하는지 확인합니다.

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

int main(void){
```

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

검증

1. 패치된 cello.c 프로그램을 빌드합니다.

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

2. 빌드된 cello.c 프로그램을 실행합니다.

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

3.2. LICENSE 파일 만들기

소프트웨어 라이센스와 함께 소프트웨어를 배포하는 것이 좋습니다.

소프트웨어 라이센스 파일은 사용자에게 소스 코드로 수행할 수 있는 작업을 사용자에게 알립니다. 소스 코드에 대한 라이센스가 없으면 이 코드에 대한 모든 권한을 보유하고 소스 코드에서 파생 작업을 재현, 배포 또는 생성할 수 없습니다.

절차

- 필요한 라이센스 문을 사용하여 LICENSE 파일을 생성합니다.

```
$ vim LICENSE
```

예 3.1. GPLv3 LICENSE 파일 텍스트 예

```
$ 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

- sorce 코드 예

3.3. 배포를 위한 소스 코드 아카이브 생성

아카이브 파일은 **.tar.gz** 또는 **.tgz** 접미사가 있는 파일입니다. 소스 코드를 아카이브에 배치하는 것은 나중에 배포를 위해 패키징할 소프트웨어를 릴리스하는 일반적인 방법입니다.

3.3.1. 샘플 Bash 프로그램의 소스 코드 아카이브 생성

벨로 *프로젝트*는 Bash 의 **Hello World** 파일입니다.

다음 예제에는 **bello** 쉘 스크립트만 포함되어 있습니다. 따라서 생성된 **tar.gz** 아카이브에는 **LICENSE** 파일 외에 하나의 파일만 있습니다.

참고

패치 파일은 프로그램과 함께 아카이브에 배포되지 않습니다. **RPM** 패키지 관리자는 **RPM**을 빌드할 때 패치를 적용합니다. 패치는 **tar.gz** 아카이브와 함께 **~/rpmbuild/SOURCES/** 디렉터리에 배치됩니다.

사전 요구 사항

- **bello** 프로그램의 **0.1** 버전이 사용된다고 가정합니다.
- **LICENSE** 파일을 생성하셨습니다. 자세한 내용은 LICENSE 파일 만들기를 참조하십시오.

절차

1. 필요한 모든 파일을 단일 디렉터리로 이동합니다.

```
$ mkdir bello-0.1

$ mv ~/bello bello-0.1/

$ mv LICENSE bello-0.1/
```

2. 배포를 위한 아카이브를 생성합니다.

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

3. 생성된 아카이브를 ~/rpmbuild/SOURCES/ 디렉터리로 이동합니다. 이 디렉터리는 rpmbuild 명령이 패키지 빌드용 파일을 저장하는 기본 디렉터리입니다.

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

추가 리소스

- bash로 작성된 hello world

3.3.2. 샘플 Python 프로그램에 대한 소스 코드 아카이브 생성

pello 프로젝트는 Python 의 Hello World 파일입니다.

다음 예제는 pello.py 프로그램만 포함합니다. 따라서 생성된 tar.gz 아카이브에는 LICENSE 파일 외에 하나의 파일만 있습니다.

참고

패치 파일은 프로그램과 함께 아카이브에 배포되지 않습니다. RPM 패키지 관리자는 RPM을 빌드할 때 패치를 적용합니다. 패치는 tar.gz 아카이브와 함께 ~/rpmbuild/SOURCES/ 디렉터리에 배치됩니다.

사전 요구 사항

- pello 프로그램의 0.1.1 버전이 사용되었다고 가정합니다.

- LICENSE 파일을 생성하셨습니다. 자세한 내용은 LICENSE 파일 만들기를 참조하십시오.

절차

1. 필요한 모든 파일을 단일 디렉터리로 이동합니다.

```
$ mkdir pello-0.1.1

$ mv pello.py pello-0.1.1/

$ mv LICENSE pello-0.1.1/
```

2. 배포를 위한 아카이브를 생성합니다.

```
$ tar -cvzf pello-0.1.1.tar.gz pello-0.1.1
pello-0.1.1/
pello-0.1.1/LICENSE
pello-0.1.1/pello.py
```

3. 생성된 아카이브를 ~/rpmbuild/SOURCES/ 디렉터리로 이동합니다. 이 디렉터리는 rpmbuild 명령이 패키지 빌드용 파일을 저장하는 기본 디렉터리입니다.

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

추가 리소스

- Hello World는 Python으로 작성됩니다.

3.3.3. 샘플 C 프로그램에 대한 소스 코드 아카이브 생성

cello 프로젝트는 C의 Hello World 파일입니다.

다음 예제에는 cello.c 및 Makefile 파일만 포함되어 있습니다. 따라서 생성된 tar.gz 아카이브에는 LICENSE 파일 외에도 두 개의 파일이 있습니다.

참고

패치 파일은 프로그램과 함께 아카이브에 배포되지 않습니다. RPM 패키지 관리자는 RPM을 빌드할 때 패치를 적용합니다. 패치는 tar.gz 아카이브와 함께 ~/rpmbuild/SOURCES/ 디렉터리에 배치됩니다.

사전 요구 사항

- cello 프로그램의 1.0 버전이 사용되었다고 가정합니다.
- LICENSE 파일을 생성하셨습니다. 자세한 내용은 LICENSE 파일 만들기를 참조하십시오.

절차

1. 필요한 모든 파일을 단일 디렉터리로 이동합니다.

```
$ mkdir cello-1.0

$ mv cello.c cello-1.0/

$ mv Makefile cello-1.0/

$ mv LICENSE cello-1.0/
```

2. 배포를 위한 아카이브를 생성합니다.

```
$ 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
```

3. 생성된 아카이브를 ~/rpmbuild/SOURCES/ 디렉터리로 이동합니다. 이 디렉터리는 rpmbuild 명령이 패키지 빌드용 파일을 저장하는 기본 디렉터리입니다.

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

추가 리소스

- Hello World에서 C로 작성됩니다.

4장. 패키지 소프트웨어

다음 섹션에서는 RPM 패키지 관리자를 사용하여 패키징 프로세스의 기본 사항을 알아봅니다.

4.1. RPM 패키지 작업 공간 설정

RPM 패키지를 빌드하려면 먼저 다른 패키징 목적으로 사용되는 디렉터리로 구성된 특수 작업 공간을 생성해야 합니다.

4.1.1. RPM 패키지 작업 공간 구성

RPM 패키징 작업 공간을 구성하려면 rpmdev-setuptree 유틸리티를 사용하여 디렉터리 레이아웃을 설정할 수 있습니다.

사전 요구 사항

- RPM 패키징 유틸리티를 제공하는 rpmdevtools 패키지가 설치되어 있습니다.

```
# dnf install rpmdevtools
```

절차

- rpmdev-setuptree 유틸리티를 실행합니다.

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

추가 리소스

- RPM 패키징 작업 공간 디렉터리

4.1.2. RPM 패키징 작업 공간 디렉터리

다음은 **rpmdev-setuptree** 유틸리티를 사용하여 생성된 **RPM** 패키징 작업 공간 디렉터리입니다.

표 **4.1. RPM** 패키징 작업 공간 디렉터리

디렉터리	목적
BUILD	**SOURCES** 디렉터리의 소스 파일에서 컴파일된 빌드 아티팩트를 포함합니다.
RPMS	바이너리 RPM은 다양한 아키텍처의 하위 디렉토리에 있는 **RPMS** 디렉토리 아래에 생성됩니다. 예를 들어 **x86_64** 또는 **noarch** 하위 디렉터리에서.
소스	압축된 소스 코드 아카이브 및 패치를 포함합니다. 그런 다음 **rpmbuild** 명령은 이 디렉토리에서 이러한 아카이브 및 패치를 검색합니다.
SPECS	패키지 관리자에서 생성한 **사양** 파일을 포함합니다. 그런 다음 이러한 파일은 패키지를 빌드하는 데 사용됩니다.
SRPMS	**rpmbuild** 명령을 사용하여 바이너리 RPM 대신 SRPM을 빌드하면 결과 SRPM이 이 디렉터리에 생성됩니다.

4.2. 사양 파일 정보

사양 파일은 **rpmbuild** 유틸리티에서 **RPM** 패키지를 빌드하는 데 사용하는 지침이 포함된 파일입니다. 이 파일은 일련의 섹션에 지침을 정의하여 빌드 시스템에 필요한 정보를 제공합니다. 이러한 섹션은 ***Preamble*** 및 **spec** 파일의 ***Body*** 부분에 정의되어 있습니다.

- ***Preamble*** 섹션에는 본문 섹션에서 사용되는 일련의 메타데이터 항목이 포함되어 있습니다.
- ***Body*** 섹션은 지침의 주요 부분을 나타냅니다.

4.2.1. 사전 항목

다음은 **RPM** 사양 파일의 ***Preamble*** 섹션에서 사용할 수 있는 몇 가지 지시문입니다.

표 **4.2.** ***Preamble*** 섹션 지시문

directive	정의
이름	**사양** 파일 이름과 일치해야 하는 패키지의 기본 이름입니다.
버전	소프트웨어의 업스트림 버전 번호입니다.
릴리스 버전	패키지 버전이 릴리스된 횟수입니다. 초기 값을 **1%{?dist}** 로 설정하고 패키지의 새 릴리스마다 늘립니다. 새 버전의 소프트웨어가 빌드되면 **1** 로 재설정합니다.
요약	패키지에 대한 간단한 한 줄 요약입니다.
라이센스	패키지되는 소프트웨어의 라이센스입니다. **사양** 파일의 **라이센스** 레이블을 지정하는 방법에 대한 정확한 형식은 다음 RPM 기반 Linux 배포 지침(예: GPLv3+)에 따라 달라집니다.
URL	소프트웨어에 대한 자세한 내용은 전체 URL(예: 패키징 중인 소프트웨어에 대한 업스트림 프로젝트 웹 사이트)입니다.
소스	패치되지 않은 업스트림 소스 코드의 압축된 아카이브의 경로 또는 URL입니다. 이 링크는 아카이브의 액세스 가능하고 안정적인 스토리지(예: 패키지 관리자의 로컬 스토리지)가 아닌 업스트림 페이지(예: 업스트림 페이지)를 가리켜야 합니다. 지시문 이름 끝에 숫자 또는 숫자 없이 **Source** 지시문을 적용할 수 있습니다. 번호가 지정되지 않은 경우 번호가 내부적으로 항목에 할당됩니다. 값을 명시적으로 지정할 수도 있습니다(예: **Source0**,**Source1**,**Source2**,**Source3** 등).
패치	필요한 경우 소스 코드에 적용할 첫 번째 패치의 이름입니다. 지시문 이름 끝에 숫자 또는 숫자 없이 **Patch** 지시문을 적용할 수 있습니다. 번호가 지정되지 않은 경우 번호가 내부적으로 항목에 할당됩니다. 또한 **Patch0**,**Patch1**,**Patch2**,**Patch3** 등과 같은 번호를 명시적으로 지정할 수도 있습니다. **%patch0**,**%patch1**,**%patch2** 매크로 등을 사용하여 패치를 개별적으로 적용할 수 있습니다. 매크로는 RPM **사양** 파일의 *Body* 섹션에 있는 **%prep** 지시문 내에 적용됩니다. 또는 모든 패치를 **사양** 파일에서 제공되는 순서대로 자동으로 적용하는 **%autopatch** 매크로를 사용할 수 있습니다.
BuildArch	소프트웨어가 빌드될 아키텍처입니다. 예를 들어 소프트웨어를 해석된 프로그래밍 언어로 완전히 작성한 경우 해당 값을 **BuildArch: noarch** 로 설정합니다. 이 값을 설정하지 않으면 소프트웨어가 빌드된 시스템의 아키텍처(예: **x86_64**)를 자동으로 상속합니다.

directive	정의
BuildRequires	컴파일된 언어로 작성된 프로그램을 빌드하는 데 필요한 쉼표 또는 공백으로 구분된 패키지 목록입니다. **BuildRequires** 의 여러 항목이 있을 수 있으며, 각 항목은 SPEC 파일의 자체 행에 있습니다.
필수 항목	소프트웨어를 설치한 후 실행하는 데 필요한 쉼표 또는 공백으로 구분된 패키지 목록입니다. **Requires** 의 여러 항목이 있을 수 있으며 각각**사양** 파일의 자체 행에 있을 수 있습니다.
ExcludeArch	특정 프로세서 아키텍처에서 소프트웨어를 작동할 수 없는 경우 **ExcludeArch** 지시문에서 이 아키텍처를 제외할 수 있습니다.
충돌	설치된 경우 소프트웨어가 제대로 작동하려면 시스템에 설치해서는 안 되는 쉼표 또는 공백으로 구분된 패키지 목록입니다. **Conflicts** 의 여러 항목이 있을 수 있으며 각각 **spec** 파일의 자체 행에 있습니다.
사용되지 않음	**Obsoletes** 지시문은 다음 요인에 따라 업데이트 작동 방식을 변경합니다. ● 명령행에서 **rpm** 명령을 직접 사용하는 경우 설치 중인 패키지의 더 이상 사용되지 않는 패키지와 일치하는 모든 패키지를 제거하거나 업데이트 또는 종속성 해결기에 의해 업데이트가 수행됩니다. ● 업데이트 또는DNF(종속성 확인자)를 사용하는 경우 일치하는 **Obsoletes가 포함된 패키지:** 업데이트가 추가되고 일치하는 패키지를 교체합니다.
제공	패키지에 **Provides** 지시문을 추가하면 이 패키지를 이름 이외의 종속성에서 참조할 수 있습니다.

이름,버전 및 릴리스 (NVR) 지시문은 name-version-release 형식으로 RPM 패키지의 파일 이름을 구성합니다.

rpm 명령을 사용하여 RPM 데이터베이스를 쿼리하여 특정 패키지에 대한 NVR 정보를 표시할 수 있습니다. 예를 들면 다음과 같습니다.

```
# rpm -q bash
bash-4.4.19-7.el8.x86_64
```

bash 는 패키지 이름이고 4.4.19 는 버전이며 7.el8 은 릴리스입니다. x86_64 마커는 패키지 아키텍처입니다. NVR 과 달리 아키텍처 마커는 RPM 패키지러를 직접 제어하지 않지만 rpmbuild 빌드 환경에 의해 정의됩니다. 이에 대한 예외는 아키텍처 독립적인 noarch 패키지입니다.

4.2.2. 본문 항목

다음은 RPM 사양 파일의 *Body* 섹션에 사용된 항목입니다.

표 4.3. 본문 섹션 항목

directive	정의
%Description	RPM에 패키지된 소프트웨어에 대한 전체 설명입니다. 이 설명은 여러 행에 걸쳐 있을 수 있으며 단락으로 나눌 수 있습니다.
%Prep	빌드하기 위해 소프트웨어를 준비하는 명령 또는 일련의 명령(예: **Source** 지시문에서 아카이브 압축을 풀기 위한 명령)입니다. **%prep** 지시문에는 쉘 스크립트가 포함될 수 있습니다.
%build	소프트웨어를 머신 코드(컴파일된 언어의 경우) 또는 바이트 코드(일부 해석 언어의 경우)로 빌드하는 명령 또는 일련의 명령입니다.
%install	소프트웨어를 빌드한 후 **rpmbuild** 유틸리티에서 **BUILDROOT** 디렉토리에 소프트웨어를 설치하는 데 사용할 명령 또는 일련의 명령입니다. 이러한 명령은 빌드가 수행되는 **%_builddir** 디렉터리의 원하는 빌드 아티팩트를 패키징할 파일과 디렉터리 구조가 포함된 **%buildroot** 디렉터리에 복사합니다. 여기에는 ~/rpmbuild/BUILD에서 **~/rpmbuild/BUILD ROOT** 로 파일을 복사하고 **~/rpmbuild/BUILDROOT** 에 필요한 디렉토리를 만드는 작업이 포함됩니다. **%install** 디렉터리는 최종 사용자의 **루트** 디렉터리와 유사한 빈 **chroot** 기본 디렉터리입니다. 여기에서 설치된 파일을 포함할 디렉토리를 만들 수 있습니다. 이러한 디렉터리를 만들려면 경로를 하드 코딩하지 않고도 RPM 매크로를 사용할 수 있습니다. **%install** 은 패키지를 설치할 때가 아니라 패키지를 만들 때만 실행됩니다. 자세한 내용은 사양 파일 작업을 참조하십시오.
%check	소프트웨어를 테스트하는 명령 또는 일련의 명령(예: 단위 테스트)입니다.
%files	RPM 패키지에서 제공하는 파일 목록(사용자 시스템 및 시스템의 전체 경로 위치)을 설치합니다. 빌드 중에 **%files** 에 나열되지 않은 **%buildroot** 디렉터리에 파일이 있는 경우 패키징되지 않은 파일에 대한 경고가 표시됩니다. **%files** 섹션에서 기본 제공 매크로를 사용하여 다양한 파일의 역할을 나타낼 수 있습니다. **rpm** 명령을 사용하여 패키지 파일 매니페스트 메타데이터를 쿼리하는 데 유용합니다. 예를 들어 **LICENSE** 파일이 소프트웨어 라이센스 파일임을 나타내기 위해 **%license** 매크로를 사용합니다.
%changelog	다른 **버전** 또는 **릴리스** 빌드 간에 패키지에 발생한 변경 사항 레코드입니다. 이러한 변경 사항에는 패키지의 각 Version-Release에 대한 날짜-amped 항목 목록이 포함됩니다. 이러한 항목은 소프트웨어 변경이 아닌 패키징 변경 사항을 기록합니다(예: **%build** 섹션에서 패치 추가 또는 빌드 절차 변경).

4.2.3. 고급 항목

사양 파일은 Scriptlets 또는 Trigger 와 같은 고급 항목을 포함할 수 있습니다.

스크립트릿 및 트리거는 빌드 프로세스가 아닌 최종 사용자의 시스템에서 설치 프로세스 중에 다른 지점에서 적용됩니다.

4.3. BUILDROOTS

RPM 패키징 컨텍스트에서 **buildroot** 는 **chroot** 환경입니다. 빌드 아티팩트는 최종 사용자 시스템에서 향후 계층 구조와 동일한 파일 시스템 계층 구조를 사용하고 **buildroot** 는 루트 디렉터리 역할을 합니다. 빌드 아티팩트 배치는 최종 사용자 시스템의 파일 시스템 계층 구조 표준을 준수해야 합니다.

buildroot 의 파일은 나중에 **RPM**의 주요 부분이 되는 **cpio** 아카이브에 배치됩니다. **RPM**이 최종 사용자의 시스템에 설치되면 이러한 파일이 루트 디렉터리에 추출되어 올바른 계층 구조를 유지합니다.

참고

rpmbuild 프로그램에는 자체 기본값이 있습니다. 이러한 기본값을 재정의하면 특정 문제가 발생할 수 있습니다. 따라서 **buildroot** 매크로의 고유한 값을 정의하지 마십시오. 대신 기본 **%{buildroot}** 매크로를 사용합니다.

4.4. RPM 매크로

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

예를 들어 패키지된 소프트웨어 *의* 버전은 **%{version}** 매크로에서 한 번만 정의하고 사양 파일 전체에서 이 매크로를 사용할 수 있습니다. 모든 발생은 매크로에서 정의한 ***Version*** 으로 자동 대체됩니다.

참고

익숙하지 않은 매크로가 표시되면 다음 명령을 사용하여 평가할 수 있습니다.

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

예를 들어 %{_bindir} 및 %{_libexecdir} 매크로를 평가하려면 다음을 입력합니다.

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

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

추가 리소스

- 매크로에 대한 자세한 내용

4.5. 사양 파일 작업

새 소프트웨어를 패키징하려면 사양 파일을 생성해야 합니다. 다음 방법 중 하나를 사양 파일을 생성할 수 있습니다.

- 새 사양 파일을 처음부터 수동으로 작성합니다.
- **rpmdev-newspec** 유틸리티를 사용합니다. 이 유틸리티는 필요한 지시문 및 필드를 채우는 채워지지 않은 사양 파일을 생성합니다.

참고

일부 자격증에 중점을 둔 텍스트 편집기는 자체 사양 템플릿으로 새 사양 파일을 미리 채웁니다. **rpmdev-newspec** 유틸리티는 편집기와 무관한 방법을 제공합니다.

4.5.1. 샘플 Bash, Python 및 C 프로그램에 대한 새 사양 파일 생성

rpmdev-new spec 유틸리티를 사용하여 **Hello World!** 프로그램의 세 가지 구현 각각에 대해 사양 파일을 생성할 수 있습니다.

사전 요구 사항

- 다음 **Hello World!** 프로그램 구현은 **~/rpmbuild/SOURCES** 디렉터리에 배치되었습니다.
  - bello-0.1.tar.gz
  - pello-0.1.2.tar.gz
  - cello-1.0.tar.gz (cello-output-first-patch.patch)

절차

1. **~/rpmbuild/SPECS** 디렉터리로 이동합니다.

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

2. **Hello World!** 프로그램의 세 가지 구현 각각에 대한 사양 파일을 생성합니다.

```
$ 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/ 디렉터리에는 이제 **bello. spec ,cello.spec, pello.spec** 이라는 세 개의 사양 파일이 포함되어 있습니다.

3. 생성된 파일을 검사합니다.

   파일의 지시문은 사양 파일 정보를 나타냅니다. 다음 섹션에서는 **rpmdev-newspec** 의 출력 파일에 특정 섹션을 채웁니다.

4.5.2. 원래 사양 파일 수정

rpmdev-new spec 유틸리티로 생성된 원본 출력 사양 파일은 rpmbuild utility. rpmbuild에 필요한 지침을 제공하도록 수정해야 하는 템플릿을 나타냅니다. rpmbuild 는 이러한 지침을 사용하여 RPM 패키지를 빌드합니다.

사전 요구 사항

- rpm dev-new spec 유틸리티를 사용하여 채워지지 않은 ~/rpmbuild/SPECS/<name>. spec 사양 파일이 생성되었습니다. 자세한 내용은 샘플 Bash, Python 및 C 프로그램에 대한 새 사양 파일 생성 을 참조하십시오.

절차

1. rpmdev-newspec 유틸리티에서 제공하는 ~/rpmbuild/SPECS/<name>.spec 파일을 엽니다.

2. spec 파일 *Preamble* 섹션의 다음 지시문을 채웁니다.

이름

name 이 이미 rpmdev-newspec 에 인수로 지정되었습니다.

버전

소스 코드의 업스트림 릴리스 버전과 일치하도록 Version 을 설정합니다.

릴리스 버전

릴리스 는 자동으로 1%{?dist} 로 설정되며 처음 1 입니다.

요약

패키지에 대한 한 줄 설명을 입력합니다.

라이센스

소스 코드와 관련된 소프트웨어 라이센스를 입력합니다.

URL

업스트림 소프트웨어 웹 사이트의 URL을 입력합니다. 일관성을 위해 %{name} RPM 매크로 변수를 사용하고 https://example.com/%{name} 형식을 사용합니다.

소스

업스트림 소프트웨어 소스 코드에 대한 **URL**을 입력합니다. 패키지되는 소프트웨어 버전에 직접 연결합니다.

참고

이 문서의 예제 **URL**에는 향후 변경될 수 있는 하드 코드된 값이 포함되어 있습니다. 마찬가지로 릴리스 버전도 변경될 수 있습니다. 이러한 잠재적인 변경 사항을 단순화하려면 **%{name}** 및 **%{version}** 매크로를 사용합니다. 이러한 매크로를 사용하면 사양 파일에서 하나의 필드만 업데이트해야 합니다.

BuildRequires

패키지에 대한 빌드 시간 종속 항목을 지정합니다.

필수 항목

패키지에 대한 런타임 종속 항목을 지정합니다.

BuildArch

소프트웨어 아키텍처를 지정합니다.

3. **spec** 파일 ***Body*** 섹션의 다음 지시문을 채웁니다. 이러한 지시문은 여러 줄, 다중 설명 또는 스크립팅된 작업을 정의할 수 있으므로 이러한 지시문을 섹션 제목으로 간주할 수 있습니다.

%Description

소프트웨어에 대한 전체 설명을 입력합니다.

%Prep

빌드용 소프트웨어를 준비하려면 명령 또는 일련의 명령을 입력합니다.

%build

소프트웨어 빌드를 위한 명령 또는 일련의 명령을 입력합니다.

%install

rpmbuild 명령에 소프트웨어를 **BUILDROOT** 디렉터리에 설치하는 방법에 대한 명령 또는 일련의 명령을 입력합니다.

%files

RPM 패키지에서 제공하는 파일 목록을 시스템에 설치합니다.

%changelog

패키지의 각 Version-Release 에 대한 datestamped 항목 목록을 입력합니다.

별표(*) 문자 뒤에 Day -of-Week Day Name Surname <email> - Version-Release 를 사용하여 %changelog 섹션의 첫 번째 행을 시작합니다.

실제 변경 항목은 다음 규칙을 따릅니다.

- 각 변경 항목에는 각 변경 사항에 대해 하나씩 여러 항목이 포함될 수 있습니다.
- 각 항목은 새 줄에서 시작됩니다.
- 각 항목은 하이픈(-) 문자로 시작합니다.

이제 필요한 프로그램에 대한 전체 사양 파일을 작성했습니다.

추가 리소스

- Preamble 항목
- 본문 항목
- 샘플 Bash 프로그램의 사양 파일 예
- 샘플 Python 프로그램에 대한 사양 파일의 예
- 샘플 C 프로그램에 대한 사양 파일 예

- RPM 빌드

4.5.3. 샘플 Bash 프로그램의 사양 파일 예

참조를 위해 bash 로 작성된 벨로 프로그램에 다음 예제 사양 파일을 사용할 수 있습니다.

bash로 작성된 bello 프로그램의 사양 파일 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
```

- 패키지에 대한 빌드 시간 종속성을 지정하는 **BuildRequires** 지시문이 **bello** 빌드 단계가 없기 때문에 삭제되었습니다. **Bash**는 원시 해석된 프로그래밍 언어이며 파일은 시스템의 위치에만 설치됩니다.

- 패키지에 대한 런타임 종속성을 지정하는 **Requires** 지시문에는 **bello** 스크립트를 실행하기 위해 **bash** 쉘 환경만 필요하므로 **bash** 만 포함됩니다.

- **bash** 스크립트를 빌드할 필요가 없기 때문에 소프트웨어 빌드 방법을 지정하는 **%build** 섹션은 비어 있습니다.

참고

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

추가 리소스

- 소스 코드란

4.5.4. Python으로 작성된 프로그램의 SPEC 파일 예제

참조용으로 **Python** 프로그래밍 언어로 작성된 **pello** 프로그램에 다음 예제 사양 파일을 사용할 수 있습니다.

Python으로 작성된 pello 프로그램의 사양 파일 예

```
%global python3_pkgversion 3.11                    (1)

Name:        python-pello                          (2)
Version:     1.0.2
Release:     1%{?dist}
Summary:     Example Python library

License:     MIT
URL:         https://github.com/fedora-python/Pello
Source:      %{url}/archive/v%{version}/Pello-%{version}.tar.gz

BuildArch:     noarch
BuildRequires:  python%{python3_pkgversion}-devel   (3)
```

```
# Build dependencies needed to be specified manually
BuildRequires:  python%{python3_pkgversion}-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python%{python3_pkgversion}-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command
line.}

%description %_description

%package -n python%{python3_pkgversion}-pello                    4
Summary:        %{summary}

%description -n python%{python3_pkgversion}-pello %_description


%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supported projects with setup.py
%py3_build                                                       5


%install
# The macro only supported projects with setup.py
%py3_install


%check                                                           6
%{pytest}


# Note that there is no %%files section for the unversioned python module
%files -n python%{python3_pkgversion}-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

1

2

Python 프로젝트를 RPM에 패키징할 때 항상 프로젝트의 원래 이름에 python- 접두사를 추가합니다. 원래 이름은 pello 이므로 소스 RPM(SRPM)의 이름은 python-pello 입니다.

3

BuildRequires 지시문은 이 패키지를 빌드하고 테스트하는 데 필요한 패키지를 지정합니다. BuildRequires 에서 항상 Python 패키지를 빌드하는 데 필요한 도구를 제공하는 항목이 포함되어 있습니다. python3-devel (또는 python3.11-devel 또는 python3.12-devel)과 같은 특정 소프트웨어에 필요한 관련 프로젝트 (예: python3-setuptools 또는 python3.11-setuptools) 또는 %check 섹션에서 테스트를 실행하는 데 필요한 런타임 및 테스트 종속 항목을 포함합니다.

4

바이너리 RPM의 이름(사용자가 설치할 수 있는 패키지)을 선택할 때 버전 지정된 Python 접두사를 추가합니다. 기본 Python 3.9, Python 3.11용 python3.11- 접두사 또는 Python 3.12용 python3.12- 접두사에 python3- 접두사를 사용합니다. 명시적 버전으로 설정하지 않는 한 기본 Python 버전 3.9에 대해 3 으로 평가되는 %{python3_pkgversion} 매크로를 사용할 수 있습니다 (예: 3.11 참조).

5

%py3_build 및 %py3_install 매크로는 설치 위치, 사용할 인터프리터 및 기타 세부 정보를 지정하는 추가 인수와 함께 setup.py build 및 setup.py install 명령을 각각 실행합니다.

6

%check 섹션은 패키지된 프로젝트의 테스트를 실행해야 합니다. 정확한 명령은 프로젝트 자체에 따라 다르지만 %pytest 매크로를 사용하여 RPM에 친숙한 방식으로 pytest 명령을 실행할 수 있습니다.

4.5.5. 샘플 C 프로그램에 대한 사양 파일 예

참조용으로 C 프로그래밍 언어로 작성된 cello 프로그램에 다음 예제 사양 파일을 사용할 수 있습니다.

C로 작성된 셀오 프로그램에 대한 사양 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 시간 종속성을 지정하는 **BuildRequires** 지시문에는 컴파일 빌드 프로세스를 수행하는 데 필요한 다음 패키지가 포함됩니다.
  - **gcc**
  - **make**
- 패키지에 대한 런타임 종속성을 지정하는 **Requires** 지시문은 이 예에서 생략됩니다. 모든 런타임 요구 사항은 **rpmbuild** 에 의해 처리되며 **cello** 프로그램에는 코어 **C** 표준 라이브러리 외부에 아무것도 필요하지 않습니다.
- **%build** 섹션은 이 예에서 **cello** 프로그램의 **Makefile** 파일이 작성되었다는 사실을 반영합니

다. 따라서 GNU make 명령을 사용할 수 있습니다. 그러나 구성 스크립트를 제공하지 않았기 때문에 %configure 에 대한 호출을 제거해야 합니다.

%make_install 매크로를 사용하여 cello 프로그램을 설치할 수 있습니다. 이는 cello 프로그램에 대한 Makefile 파일을 사용할 수 있기 때문에 가능합니다.

추가 리소스

- 소스 코드란

4.6. RPM 빌드

rpmbuild 명령을 사용하여 RPM 패키지를 빌드할 수 있습니다. 이 명령을 사용하는 경우 rpmdev-setuptree 유틸리티에서 설정한 구조와 동일한 특정 디렉터리 및 파일 구조가 예상됩니다.

다른 사용 사례와 원하는 결과에는 rpmbuild 명령에 다양한 인수 조합이 필요합니다. 다음은 주요 사용 사례입니다.

- 소스 RPM 빌드.
- 바이너리 RPM 빌드:
  - 소스 RPM에서 바이너리 RPM 다시 빌드.
  - 사양 파일에서 바이너리 RPM 빌드.

4.6.1. 소스 RPM 빌드

SRPM(Source RPM)을 구축하면 다음과 같은 이점이 있습니다.

- 환경에 배포된 RPM 파일의 특정 Name-Version-Release 의 정확한 소스를 유지할 수 있습니다. 여기에는 정확한 사양 파일, 소스 코드 및 모든 관련 패치가 포함됩니다. 이는 추적 및 디버깅 목적에 유용합니다.

- 다른 하드웨어 플랫폼 또는 아키텍처에 바이너리 **RPM**을 빌드할 수 있습니다.

사전 요구 사항

- 시스템에 **rpmbuild** 유틸리티를 설치했습니다.

```
# dnf install rpm-build
```

- 다음 **Hello World!** 구현은 **~/rpmbuild/SOURCES/** 디렉터리에 배치되었습니다.
  - **bello-0.1.tar.gz**
  - **pello-0.1.2.tar.gz**
  - **cello-1.0.tar.gz** (**cello-output-first-patch.patch**)
- 패키징하려는 프로그램의 사양 파일이 있습니다.

절차

1. 생성된 사양 파일이 포함된 **~/rpmbuild/SPECS/** 지시문으로 이동합니다.

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

2. 지정된 사양 파일을 사용하여 **rpmbuild** 명령을 입력하여 **source RPM**을 빌드합니다.

```
$ rpmbuild -bs <specfile>
```

-bs 옵션은 *빌드 소스* 를 나타냅니다.

예를 들어 벨로 ,**pello** 및 **cello** 프로그램에 대한 소스 **RPM**을 빌드하려면 다음을 입력합니다.

```
$ rpmbuild -bs bello.spec
```

```
Wrote: /home/admiller/rpmbuild/SRPMS/bello-0.1-1.el8.src.rpm

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

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

검증

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

추가 리소스

- 사양 파일 작업
- 샘플 Bash, C 및 Python 프로그램에 대한 새 사양 파일 생성
- 원래 사양 파일 수정

4.6.2. 소스 RPM에서 바이너리 RPM 재빌드

소스 RPM(SRPM)에서 바이너리 RPM을 다시 빌드하려면 --rebuild 옵션과 함께 rpmbuild 명령을 사용합니다.

바이너리 RPM을 생성할 때 생성되는 출력은 상세화되어 디버깅에 유용합니다. 출력은 다양한 예에 따라 다르며 사양 파일에 해당합니다.

생성된 바이너리 RPM은 ~/rpmbuild/RPMS/ YOURARCH 디렉터리에 있습니다. 여기서ARCH는 아키텍처별이 아닌 경우 ~/rpmbuild/RPMS/noarch/ 디렉터리에 있습니다.

사전 요구 사항

- 시스템에 rpmbuild 유틸리티를 설치했습니다.

```
# dnf install rpm-build
```

절차

1. 소스 **RPM**이 포함된 **~/rpmbuild/SRPMS/** 지시문으로 이동합니다.

```
$ cd ~/rpmbuild/SRPMS/
```

2. 소스 **RPM**에서 바이너리 **RPM**을 다시 빌드합니다.

```
$ rpmbuild --rebuild <srpm>
```

srpm 을 소스 **RPM** 파일의 이름으로 바꿉니다.

예를 들어 **SRPM**에서 **p ello,pello, cello** 를 다시 빌드하려면 다음을 입력합니다.

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

$ rpmbuild --rebuild pello-0.1.2-1.el8.src.rpm
[output truncated]

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

참고

rpmbuild --rebuild 를 호출하려면 다음 프로세스가 포함됩니다.

- SRPM의 콘텐츠(spec 파일 및 소스 코드)를 ~/rpmbuild/ 디렉터리에 설치합니다.
- 설치된 콘텐츠를 사용하여 RPM 빌드.
- 사양 파일 및 소스 코드 제거

다음 방법 중 하나를 빌드한 후 사양 파일 및 소스 코드를 유지할 수 있습니다.

- RPM을 빌드할 때 --rebuild 옵션 대신 --recompile 옵션과 함께 rpmbuild 명령을 사용합니다.
- 벨로 ,pello , 셀오 용 SRPM 설치:

```
$ 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%]
```

4.6.3. 사양 파일에서 바이너리 RPM 빌드

사양 파일에서 바이너리 RPM을 빌드하려면 rpmbuild 명령을 -bb 옵션과 함께 사용합니다.

사전 요구 사항

- 시스템에 rpmbuild 유틸리티를 설치했습니다.

-

```
# dnf install rpm-build
```

절차

1. 사양 파일이 포함된 ~/rpmbuild/SPECS/ 지시문으로 이동합니다.

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

2. 사양에서 바이너리 RPM을 빌드합니다.

```
$ rpmbuild -bb <spec_file>
```

예를 들어 사양 파일에서 벨로,pello 및 cello 바이너리 RPM을 빌드하려면 다음을 입력합니다.

```
$ rpmbuild -bb bello.spec

$ rpmbuild -bb pello.spec

$ rpmbuild -bb cello.spec
```

4.7. 일반적인 오류가 있는지 RPM 확인

패키지를 만든 후 패키지의 품질을 확인하는 것이 좋습니다. 패키지 품질을 확인하는 주요 도구는 rpmlint 입니다.

rpmlint 툴을 사용하면 다음 작업을 수행할 수 있습니다.

- RPM 유지 관리 기능 개선.
- RPM의 정적 분석을 수행하여 콘텐츠 검증을 활성화합니다.
- RPM의 정적 분석을 수행하여 오류 검사를 활성화합니다.

rpmlint 를 사용하여 바이너리 RPM, 소스 RPM(SRPM) 및 사양 파일을 확인할 수 있습니다. 따라서 이 도구는 패키징의 모든 단계에 유용합니다.

rpmlint 에는 엄격한 지침이 있습니다. 따라서 다음 섹션에 표시된 것처럼 일부 오류 및 경고를 건너뛸 수 있는 경우가 있습니다.

참고

다음 섹션에 설명된 예에서 rpmlint 는 검증되지 않은 출력을 생성하는 옵션 없이 실행됩니다. 각 오류 또는 경고에 대한 자세한 설명은 대신 rpmlint -i 를 실행합니다.

4.7.1. 일반적인 오류가 있는지 샘플 Bash 프로그램 확인

다음 섹션에서 bello 사양 파일 및 bello 바이너리 RPM의 예에서 RPM에서 일반적인 오류가 있는지 확인할 때 발생할 수 있는 경고 및 오류를 조사합니다.

4.7.1.1. 일반적인 오류가 있는지 벨로 사양 파일 확인

다음 예제의 출력을 검사하여 벨로 사양 파일에서 일반적인 오류가 있는지 확인하는 방법을 알아봅니다.

bello 사양 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 invalid-url Source0 경고 하나만 있습니다. 이 경고는 Source0 지시문에 나열된 URL에 연결할 수 없음을 의미합니다. 지정된 example.com URL이 없기 때문에 이는 예상되는 것입니다. 이 URL이 나중에 유효하다고 가정하면 이 경고를 무시할 수 있습니다.

bello SRPM에서 rpmlint 명령 실행의 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 invalid-url URL 경고가 있습니다. 즉, URL 지시문에 지정된 URL 에 연결할 수 없습니다. 이 URL이 나중에 유효하다고 가정하면 이 경고를 무시할 수 있습니다.

4.7.1.2. 일반적인 오류가 있는지 벨로 바이너리 RPM 확인

바이너리 RPM을 확인할 때 rpmlint 명령은 다음 항목을 확인합니다.

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

다음 예제의 출력을 검사하여 벨로 바이너리 RPM에서 일반적인 오류가 있는지 확인하는 방법을 알아봅니다.

벨로 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 다음을 제공하지 않았기 때문에 RPM에 문서 또는 수동 페이지가 없음을 의미합니다. 출력 경고 외에도 RPM은 rpmlint 검사를 전달했습니다.

4.7.2. 일반적인 오류가 있는지 샘플 Python 프로그램 확인

다음 섹션에서 pello 사양 파일 및 pello 바이너리 RPM 예제의 RPM 콘텐츠를 검증할 때 발생할 수 있는 경고 및 오류를 조사합니다.

4.7.2.1. 일반적인 오류가 있는지 pello 사양 파일 확인

다음 예제의 출력을 검사하여 pello 사양 파일에서 일반적인 오류가 있는지 확인하는 방법을 알아봅니다.

pello 사양 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 경고는 Source0 지시문에 나열된 URL에 연결할 수 없음을 의미합니다. 지정된 example.com URL이 없기 때문에 이 작업이 예상됩니다. 이 URL이 나중에 유효하다고 가정하면 이 경고를 무시할 수 있습니다.

- hardcoded-ctlplane-path 오류는 라이브러리 경로를 하드 코딩하는 대신 %{_libdir} 매크로를 사용하는 것이 좋습니다. 이 예제에서는 이러한 오류를 무시해도 됩니다. 그러나 프로덕션 환경으로 들어오는 패키지의 경우 모든 오류를 주의 깊게 확인합니다.

SRPM for pello에서 rpmlint 명령 실행의 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.
```

invalid-url URL 오류는 **URL** 지시문에 언급된 **URL** 에 연결할 수 없음을 의미합니다. 이 **URL**이 나중에 유효하다고 가정하면 이 경고를 무시할 수 있습니다.

4.7.2.2. 일반적인 오류가 있는지 pello 바이너리 RPM 확인

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

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

다음 예제의 출력을 검사하여 **pello** 바이너리 **RPM**에서 일반적인 오류가 있는지 확인하는 방법을 알아봅니다.

pello 바이너리 **RPM**에서 **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-manual-page-for-binary** 경고는 제공하지 않았기 때문에 **RPM**에

문서 또는 수동 페이지가 없음을 의미합니다.

- **only-binary-in-usr-lib** 경고는 **/usr/lib/** 디렉터리에 바이너리가 아닌 아티팩트만 제공했음을 의미합니다. 이 디렉터리는 일반적으로 바이너리 파일인 공유 오브젝트 파일에 사용됩니다. 따라서 **rpmlint** 는 **/usr/lib/** 에 있는 하나 이상의 파일이 바이너리가 될 것으로 예상합니다.

  다음은 **Filesystem Hierarchy Standard**의 준수를 확인하는 **rpmlint** 검사의 예입니다. 파일이 올바르게 배치되도록 하려면 **RPM** 매크로를 사용합니다. 이 예제에서는 이 경고를 무시해도 됩니다.

- **non-executable-script** 오류는 **/usr/lib/pello/pello.py** 파일에 실행 권한이 없음을 의미합니다. 파일에 **shebang (#!)**이 포함되어 있으므로 **rpmlint** 툴에서 파일을 실행할 수 있을 것으로 예상합니다. 이 예제의 목적을 위해 실행 권한 없이 이 파일을 그대로 두고 이 오류를 무시할 수 있습니다.

출력 경고 및 오류 외에도 **RPM**은 **rpmlint** 검사를 통과했습니다.

4.7.3. 일반적인 오류가 있는지 샘플 C 프로그램 확인

다음 섹션에서 **cello** 사양 파일 및 셀오 바이너리 **RPM** 예제의 **RPM** 콘텐츠를 검증할 때 발생할 수 있는 경고 및 오류를 조사합니다.

4.7.3.1. 일반적인 오류가 있는지 cello 사양 파일 확인

다음 예제의 출력을 검사하여 **cello** 사양 파일에서 일반적인 오류를 확인하는 방법을 알아봅니다.

cello 사양 파일에서 **rpmlint** 명령을 실행하는 출력

```
$ rpmlint ~/rpmbuild/SPECS/cello.spec
/home/admiller/rpmbuild/SPECS/cello.spec: W: invalid-url Source0:
https://www.example.com/cello/releases/cello-1.0.tar.gz HTTP Error 404: Not Found
0 packages and 1 specfiles checked; 0 errors, 1 warnings.
```

cello.spec 의 경우 **invalid-url Source0** 경고 하나만 있습니다. 이 경고는 **Source0** 지시문에 나열된 **URL**에 연결할 수 없음을 의미합니다. 이는 지정된 **example.com URL**이 없기 때문에 예상됩니다. 이

URL이 나중에 유효하다고 가정하면 이 경고를 무시할 수 있습니다.

cello SRPM에서 rpmlint 명령 실행의 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 invalid-url URL 경고가 있습니다. 이 경고는 URL 지시문에 지정된 URL에 연결할 수 없음을 의미합니다. 이 URL이 나중에 유효하다고 가정하면 이 경고를 무시할 수 있습니다.

4.7.3.2. 일반적인 오류가 있는지 cello 바이너리 RPM 확인

바이너리 RPM을 확인할 때 rpmlint 명령은 다음 항목을 확인합니다.

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

다음 예제의 출력을 검사하여 cello 바이너리 RPM에서 일반적인 오류가 있는지 확인하는 방법을 알아봅니다.

cello 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-manual-page-for-binary** 경고는 제공하지 않았기 때문에 **RPM**에 문서 또는 수동 페이지가 없음을 의미합니다.

출력 경고 외에도 **RPM**은 **rpmlint** 검사를 전달했습니다.

4.8. SYSLOG에 RPM 활동 기록

syslog(System Logging Protocol)를 사용하여 **RPM** 활동 또는 트랜잭션을 기록할 수 있습니다.

사전 요구 사항

- **syslog** 플러그인은 시스템에 설치됩니다.

```
# dnf install rpm-plugin-syslog
```

참고

syslog 메시지의 기본 위치는 **/var/log/messages** 파일입니다. 그러나 다른 위치를 사용하여 메시지를 저장하도록 **syslog** 를 구성할 수 있습니다.

절차

1. **syslog** 메시지를 저장하도록 구성한 파일을 엽니다.

   또는 기본 **syslog** 구성을 사용하는 경우 **/var/log/messages** 파일을 엽니다.

2. **[RPM]** 문자열을 포함하여 새 행을 검색합니다.

4.9. RPM 콘텐츠 추출

예를 들어 **RPM**에 필요한 패키지가 손상된 경우 패키지의 내용을 추출해야 할 수 있습니다. 이러한 경우 **RPM** 설치가 손상에도 불구하고 여전히 작동하는 경우 **rpm2archive** 유틸리티를 사용하여 **.rpm** 파일

을 **tar** 아카이브로 변환하여 패키지 내용을 사용할 수 있습니다.

참고

RPM 설치가 심각한 경우 **rpm2cpio** 유틸리티를 사용하여 **RPM** 패키지 파일을 **cpio** 아카이브로 변환할 수 있습니다.

절차

- **RPM** 파일을 **tar** 아카이브로 변환합니다.

```
$ rpm2archive <filename>.rpm
```

결과 파일에는 **.tgz** 접미사 가 있습니다. 예를 들어 **bash** 패키지에서 아카이브를 생성하려면 다음을 입력합니다.

```
$ rpm2archive bash-4.4.19-6.el8.x86_64.rpm
$ ls bash-4.4.19-6.el8.x86_64.rpm.tgz
bash-4.4.19-6.el8.x86_64.rpm.tgz
```

5장. 고급 주제

이 섹션에서는 소개 튜토리얼의 범위를 벗어나지만 실제적인 RPM 패키지에 유용한 주제를 다룹니다.

5.1. RPM 패키지 서명

RPM 패키지에 서명하여 타사가 콘텐츠를 변경할 수 없도록 할 수 있습니다. 추가 보안 계층을 추가하려면 패키지를 다운로드할 때 HTTPS 프로토콜을 사용합니다.

rpm-sign 패키지에서 제공하는 --addsign 옵션을 사용하여 패키지에 서명할 수 있습니다.

사전 요구 사항

- GPG 키 생성에 설명된 대로 GPG(GG) 키를 생성했습니다.

5.1.1. GPG 키 생성

다음 절차에 따라 패키지에 서명하는 데 필요한 GNU 개인 정보 보호 보호 (GPG) 키를 만듭니다.

절차

1. GPG 키 쌍을 생성합니다.

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

2. 생성된 키 쌍을 확인합니다.

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

3. 공개 키를 내보냅니다.

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

<Key_name>을 선택한 실제 키 이름으로 바꿉니다.

4. 내보낸 공개 키를 **RPM** 데이터베이스로 가져옵니다.

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

5.1.2. 패키지에 서명하도록 RPM 구성

RPM 패키지에 서명하려면 **%_gpg_name RPM** 매크로를 지정해야 합니다.

다음 절차에서는 패키지에 서명하기 위해 **RPM**을 구성하는 방법을 설명합니다.

절차

- **$HOME/.rpmmacros** 파일에서 **%_gpg_name** 매크로를 다음과 같이 정의합니다.

```
%_gpg_name Key ID
```

키 ID 를 패키지에 서명하는 데 사용할 **GNU GPG(GPG)** 키 **ID**로 바꿉니다. 유효한 **GPG** 키 **ID** 값은 키를 생성한 사용자의 전체 이름 또는 이메일 주소입니다.

5.1.3. RPM 패키지에 서명 추가

가장 일반적인 경우는 서명 없이 패키지를 빌드하는 경우입니다. 서명은 패키지를 릴리스하기 직전에 추가됩니다.

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

절차

- 패키지에 서명을 추가합니다.

```
$ rpm --addsign package-name.rpm
```

package-name 을 서명하려는 **RPM** 패키지의 이름으로 바꿉니다.

참고

서명의 시크릿 키 잠금을 해제하려면 암호를 입력해야 합니다.

5.2. 매크로에 대한 추가 정보

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

5.2.1. 자체 매크로 정의

다음 섹션에서는 사용자 지정 매크로를 만드는 방법을 설명합니다.

절차

- RPM 사양 파일에 다음 행을 포함합니다.

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

<body> 주변의 모든 공백이 제거됩니다. 이름은 영숫자로 구성될 수 있으며 문자 _ 문자는 3자 이상이어야 합니다. (opts) 필드를 포함하는 것은 선택 사항입니다.

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

- Parametrized 매크로에는 (opts) 필드가 포함되어 있습니다. 괄호 사이에 선택 문자열은 매크로 호출 시작 시 argc/argv 처리를 위해 getopt(3) 에 전달됩니다.

참고

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

- %define 에는 지역 범위가 있습니다. 이는 사양 파일의 특정 부분에 적용됩니다. %define 매크로의 본문은 사용 시 확장됩니다.
- %global 에는 전역 범위가 있습니다. 이는 전체 사양 파일에 적용됩니다. %global 매크로의 본문은 정의 시 확장됩니다.

중요

매크로는 주석 처리되거나 매크로의 이름이 사양 파일의 %changelog 섹션에 지정되더라도 평가됩니다. 매크로를 주석 처리하려면 %% 를 사용합니다. 예: %%global.

추가 리소스

- 매크로 구문

5.2.2. %setup 매크로 사용

이 섹션에서는 %setup 매크로의 다양한 변형을 사용하여 소스 코드 tarball을 사용하여 패키지를 빌드하는 방법을 설명합니다. 매크로 변형을 결합할 수 있습니다. rpmbuild 출력은 %setup 매크로의 표준 동작을 보여줍니다. 각 단계가 시작될 때 매크로는 Executing(%...)을 출력합니다.) 는 아래 예와 같이 입니다.

예 5.1. %setup 매크로 출력 예

```
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.DhddsG
```

쉘 출력은 set -x enabled로 설정됩니다. /var/tmp/rpm-tmp.DhddsG 의 내용을 보려면 rpmbuild 가 빌드 후 임시 파일을 삭제하므로 --debug 옵션을 사용합니다. 그러면 환경 변수 설정이 표시된 다음 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 작동하는지 확인합니다.
- 이전 빌드의 남은 부분을 제거합니다.
- 소스 tarball의 압축을 풉니다.
- 몇 가지 기본 권한을 설정합니다.

5.2.2.1. %setup -q 매크로 사용

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

5.2.2.2. %setup -n 매크로 사용

n 옵션은 확장된 tarball의 디렉터리 이름을 지정하는 데 사용됩니다.

이는 확장된 tarball의 디렉터리가 예상 것과 다른 이름(%{name}-%{version})과 다른 이름이 있어 %setup 매크로의 오류로 이어질 수 있는 경우에 사용됩니다.

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
```

5.2.2.3. %setup -c 매크로 사용

소스 코드 tarball에 하위 디렉터리가 포함되어 있지 않고 압축을 푼 후 아카이브의 파일이 현재 디렉터리를 채우는 경우 -c 옵션이 사용됩니다.

그런 다음 -c 옵션은 다음과 같이 디렉터리 및 단계를 아카이브 확장에 생성합니다.

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

아카이브 확장 후에는 디렉터리가 변경되지 않습니다.

5.2.2.4. %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 tarball 확장을 비활성화합니다.

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

5.2.2.5. %setup -a 및 %setup -b 매크로 사용

a 및 - b 옵션은 특정 소스를 확장합니다.

- b 옵션은 이전 을 나타냅니다. 이 옵션은 작업 디렉터리를 입력하기 전에 특정 소스를 확장합니다.
- a 옵션은 이후 를 나타냅니다. 이 옵션은 입력 후 해당 소스를 확장합니다. 해당 인수는 spec 파일 preamble의 소스 번호입니다.

다음 예에서 cello-1.0.tar.gz 아카이브에는 빈 예제 디렉터리가 포함되어 있습니다. 예제는 별도의 examples.tar.gz tarball에 포함되어 있으며 동일한 이름의 디렉토리로 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 예에서 예제는 cello-1.0/examples 로 확장되는 별도의 cello-1.0-examples.tar.gz tarball에 제공되어 있습니다. 이 경우 작업 디렉터리를 입력하기 전에 -b 1 을 사용하여 Source1 을 확장합니다.

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

5.2.3. %files 섹션의 공통 RPM 매크로

다음 표에는 사양 파일의 %files 섹션에 필요한 고급 RPM Macros가 나열되어 있습니다.

표 5.1. %files 섹션의 고급 RPM 매크로

macro	정의
% license	**%license** 매크로는 **LICENSE** 파일로 나열된 파일을 식별하고 RPM과 같이 설치 및 레이블이 지정됩니다. 예: **%license LICENSE**.
%doc	**%doc** 매크로는 문서로 나열된 파일을 식별하고 RPM에 따라 설치 및 레이블이 지정됩니다. %doc 매크로는 패키지 소프트웨어에 대한 문서와 코드 예제 및 다양한 관련 항목에도 사용됩니다. 코드 예제가 포함된 경우 파일에서 실행 모드를 제거하려면 주의해야 합니다. 예: **%doc README**
%dir	**%dir** 매크로는 경로가 이 RPM이 소유한 디렉터리인지 확인합니다. RPM 파일 매니페스트에서 설치 제거할 디렉토리를 정확하게 파악할 수 있도록 하는 것이 중요합니다. 예: **%dir %{_libdir}/%{name}**
%config(noreplace)	**%config(noreplace)** 매크로는 다음 파일이 구성 파일임을 확인하므로 원래 설치 체크섬에서 파일을 수정한 경우 패키지 설치 또는 업데이트에서 덮어쓰거나 교체해서는 안 됩니다. 변경 사항이 있는 경우 대상 시스템의 기존 또는 수정된 파일이 수정되지 않도록 업그레이드 또는 설치 시 파일 이름 끝에 **.rpmnew** 가 추가됩니다. 예:**%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf**

5.2.4. 내장 매크로 표시

Red Hat Enterprise Linux는 여러 개의 내장 RPM 매크로를 제공합니다.

절차

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

```
rpm --showrc
```

참고

출력은 매우 크기 조정 가능합니다. 결과를 좁히려면 위의 명령을 **grep** 명령과 함께 사용합니다.

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

```
rpm -ql rpm
```

참고

RPM 매크로는 출력 디렉토리 구조에서 매크로 라는 이름의 파일입니다.

5.2.5. RPM 배포 매크로

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

권장 **RPM** 매크로 세트는 종종 **RPM** 패키지로 제공되며 **dnf** 패키지 관리자를 사용하여 설치할 수 있습니다.

설치한 후 매크로 파일은 **/usr/lib/rpm/macros.d/** 디렉토리에서 찾을 수 있습니다.

절차

- 원시 **RPM** 매크로 정의를 표시하려면 다음을 실행합니다.

```
rpm --showrc
```

위의 출력에는 원시 **RPM** 매크로 정의가 표시됩니다.

- 매크로가 수행하는 작업과 **RPM**을 패키징할 때 도움이 되는 방법을 확인하려면 인수로 사용되는 매크로 이름으로 **rpm --eval** 명령을 실행합니다.

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

추가 리소스

- **RPM** 도움말 페이지

5.2.6. 사용자 정의 매크로 생성

사용자 지정 매크로를 사용하여 **~/.rpmmacros** 파일에서 배포 매크로를 재정의할 수 있습니다. 변경 사항은 시스템의 모든 빌드에 영향을 미칩니다.

> 주의
>
> **~/.rpmmacros** 파일에서 새 매크로를 정의하는 것은 권장되지 않습니다. 이러한 매크로는 다른 시스템에는 존재하지 않으며, 사용자는 패키지를 다시 빌드하려고 할 수 있습니다.

절차

- 매크로를 재정의하려면 다음을 실행합니다.

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

rpmdev-setuptree 유틸리티를 통해 모든 하위 디렉터리를 포함하여 위 예제에서 디렉터리를 생성할 수 있습니다. 이 매크로의 값은 기본적으로 **~/rpmbuild** 입니다.

```
%_smp_mflags -l3
```

위의 매크로는 대부분 **Makefile**에 전달하는 데 사용됩니다(예: **%{?_smp_mflags}**, 빌드 단계에서 여

러 개의 동시 프로세스 설정). 기본적으로 -jX 로 설정됩니다. 여기서 X 는 여러 코어입니다. 코어 수를 변경하면 패키지 빌드 속도를 높이거나 느려질 수 있습니다.

5.3. EPOCH, SCRIPTLETS 및 TRIGGERS

이 섹션에서는 RMP 사양 파일의 고급 지시문을 나타내는 Epoch,Scriptlets 및 Triggers 에 대해 설명합니다.

이러한 모든 지시문은 사양 파일뿐만 아니라 결과 RPM이 설치된 최종 시스템에도 영향을 미칩니다.

5.3.1. Epoch 지시문

Epoch 지시문을 사용하면 버전 번호에 따라 가중치 종속 항목을 정의할 수 있습니다.

이 지시문이 RPM 사양 파일에 나열되지 않으면 Epoch 지시문이 전혀 설정되지 않습니다. 이는 Epoch 가 0인 Epoch 를 설정하지 않는다는 일반적인 신념과는 대조적입니다. 그러나 dnf 유틸리티는 해독 목적으로 0의 Epoch 와 동일한 설정되지 않은 Epoch를 처리합니다.

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

> **예 5.2. Epoch 사용**
>
> Epoch를 사용하여 foobar 패키지를 설치하는 경우: 1 및 Version: 1.0, 및 다른 사용자가 Version 을 사용하여 foobar 를 패키징합니다. 2.0 하지만 Epoch 지시문이 없으면 새 버전이 업데이트되지 않습니다. Epoch 버전이 RPM 패키지에 대한 버전 관리를 나타내는 기존 Name-Version-Release 마커보다 우선하기 때문입니다.

따라서 Epoch 를 사용하는 것은 매우 드물다. 그러나 Epoch 는 일반적으로 업그레이드 순서 문제를 해결하는 데 사용됩니다. 이 문제는 인코딩에 따라 항상 안정적으로 비교할 수 없는 알파벳 문자를 포함하는 소프트웨어 버전 번호 체계 또는 버전의 업스트림 변경의 측면으로 나타날 수 있습니다.

5.3.2. scriptlets 지시문

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

빌드 시 또는 시작 스크립트에서 수행할 수 없는 작업에만 Scriptlets 를 사용합니다.

공통 scriptlet 지시문 세트가 있습니다. 이는 spec file 섹션 헤더(예: %build 또는 %install)와 유사합니다. 이는 종종 표준 POSIX 쉘 스크립트로 작성되는 코드의 여러 줄 세그먼트에 의해 정의됩니다. 그러나 대상 시스템의 배포를 위해 RPM이 허용하는 다른 프로그래밍 언어로도 작성할 수 있습니다. RPM 문서에는 사용 가능한 언어의 전체 목록이 포함되어 있습니다.

다음 표에는 실행 순서에 나열된 Scriptlet 지시문이 포함되어 있습니다. 스크립트가 포함된 패키지가 %pre 및 %post un 지시문 사이에 설치되고 %preun 과 %postun 지시문 사이에 제거됩니다.

표 5.2. Scriptlet 지시문

directive	정의
%pretrans	Scriptlet은 패키지를 설치하거나 제거하기 직전에 실행됩니다.
%pre	대상 시스템에 패키지를 설치하기 직전에 실행되는 Scriptlet입니다.
%post	대상 시스템에 패키지를 설치한 후에만 실행되는 Scriptlet입니다.
%preun	대상 시스템에서 패키지를 설치 제거하기 직전에 실행되는 Scriptlet입니다.
%postun	대상 시스템에서 패키지를 제거한 후에 실행되는 Scriptlet입니다.
%posttrans	트랜잭션 종료 시 실행되는 Scriptlet입니다.

5.3.3. scriptlet 실행 비활성화

다음 절차에서는 rpm 명령을 --no_scriptlet_name_ 옵션과 함께 사용하여 스크립트 파일의 실행을 끄는 방법을 설명합니다.

절차

- 예를 들어 %pretrans scriptlet의 실행을 끄려면 다음을 실행합니다.

```
# rpm --nopretrans
```

다음 모든 항목과 동일한 --noscripts 옵션을 사용할 수도 있습니다.

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

추가 리소스

- RPM(8) 도움말 페이지.

5.3.4. scriptlets 매크로

Scriptlets 지시문도 RPM 매크로에서 작동합니다.

다음 예제에서는 systemd에 새 장치 파일에 대해 알림을 받는 systemd scriptlet 매크로를 사용하는 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
```

5.3.5. Triggers 지시문

Trigger 는 패키지 설치 및 제거 중에 상호 작용 방법을 제공하는 RPM 지시문입니다.

주의

트리거 는 예기치 않은 시간에 실행될 수 있습니다(예: 포함된 패키지의 업데이트). 트리거 는 디버그하기 어렵기 때문에 예기치 않게 실행될 때 아무 것도 손상시키지 않도록 강력한 방식으로 구현해야 합니다. 이러한 이유로 Red Hat은 트리거 사용을 최소화할 것을 권장합니다.

단일 패키지 업그레이드에 대한 실행 순서와 각 기존 트리거에 대한 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** 파일에서 찾을 수 있습니다.

5.3.6. 사양 파일에서 쉘이 아닌 스크립트 사용

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

다음 절차에서는 **pello.py** 프로그램 설치 후 메시지를 출력하는 스크립트를 생성하는 방법을 설명합니다.

절차

1. **pello.spec** 파일을 엽니다.

2. 다음 행을 찾으십시오.

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

3. 위 줄 아래에 다음을 삽입합니다.

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

4. RPM 빌드에 설명된 대로 패키지를 빌드합니다.

5. 패키지를 설치합니다.

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

6. 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 spec 파일에 install -m 아래에 다음 행을 포함합니다.

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

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

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

이렇게 하면 사양 파일에서 인터프리터를 지정할 수 있습니다.

5.4. RPM 조건

RPM Conditionals는 사양 파일의 다양한 섹션을 조건부로 포함할 수 있습니다.

조건부 포함은 일반적으로 다음을 처리합니다.

- 아키텍처별 섹션
- 운영 체제별 섹션

- 다양한 버전의 운영 체제 간 호환성 문제
- 매크로의 존재 및 정의

5.4.1. RPM 조건 구문

RPM 조건에서는 다음 구문을 사용합니다.

expression 이 true이면 몇 가지 작업을 수행합니다.

```
%if expression
…
%endif
```

expression 이 true이면 다른 작업을 수행하는 경우 다른 작업을 수행합니다.

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

5.4.2. %if 조건

다음 예제에서는 %if RPM 조건을 사용하는 방법을 보여줍니다.

예 5.3. %if 조건부를 사용하여 Red Hat Enterprise Linux 8과 기타 운영 체제 간의 호환성 처리

```
%if 0%{?rhel} == 8
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

이 조건부는 AS_FUNCTION_DESCRIBE 매크로를 지원하는 측면에서 RHEL 8과 기타 운영 체제 간의 호환성을 처리합니다. RHEL용으로 패키지를 빌드하면 %rhel 매크로가 정의되고 RHEL 버전으로 확장됩니다. 해당 값이 8이므로 패키지가 RHEL 8에 대한 빌드인 경우 RHEL 8에서 지원하지 않는 AS_FUNCTION_DESCRIBE에 대한 참조가 autoconfig 스크립트에서 삭제됩니다.

예 5.4. %if 조건부를 사용하여 매크로 정의 처리

```
%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 조건부는 매크로의 정의를 처리합니다. %milestone 또는 %revision 매크로가 설정된 경우 업스트림 tarball의 이름을 정의하는 %ruby_archive 매크로가 다시 정의됩니다.

5.4.3. %if 조건의 특수 변형

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

%ifarch 조건

%ifarch 조건은 아키텍처별로 지정된 사양 파일의 블록을 시작하는 데 사용됩니다. 그런 다음 각각 쉼표 또는 공백으로 구분된 하나 이상의 아키텍처 분류자가 뒤에 옵니다.

예 5.5. %ifarch 조건을 사용하는 예

```
%ifarch i386 sparc
…
%endif
```

%ifarch 와 %endif 사이의 사양 파일의 모든 내용은 32비트 AMD 및 Intel 아키텍처 또는 Sun Cryostat 기반 시스템에서만 처리됩니다.

%ifnarch 조건

%ifnarch 조건부에는 %ifarch 조건보다 역방향 논리가 있습니다.

예 5.6. %ifnarch 조건을 사용하는 예

```
%ifnarch alpha
…
%endif
```

%ifnarch 와 %endif 사이의 사양 파일의 모든 내용은 Digital Alpha/AXP 기반 시스템에서 수행되지 않는 경우에만 처리됩니다.

%ifos 조건

%ifos 조건부는 빌드의 운영 체제에 따라 처리를 제어하는 데 사용됩니다. 한 개 이상의 운영 체제 이름을 뒤에 넣을 수 있습니다.

예 5.7. %ifos 조건 사용 예

```
%ifos linux
...
%endif
```

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

5.5. PYTHON 3 RPM 패키징

pip 설치 프로그램을 사용하거나 DNF 패키지 관리자를 사용하여 업스트림 PyPI 리포지토리에서 시스템에 Python 패키지를 설치할 수 있습니다. DNF는 소프트웨어에 대한 다운스트림 제어를 제공하는 RPM 패키지 형식을 사용합니다.

네이티브 Python 패키지의 패키징 형식은 Python Packaging Authority (PyPA) 사양으로 정의됩니다. 대부분의 Python 프로젝트는 패키징에 distutils 또는 setuptools 유틸리티를 사용하고 setup.py 파일에서 정의된 패키지 정보를 사용합니다. 그러나 기본 Python 패키지를 만들 가능성이 시간이 지남에 따라 발전했습니다. 새로운 패키징 표준에 대한 자세한 내용은 pyproject-rpm-macros 를 참조하십시오.

이 장에서는 setup.py 를 RPM 패키지로 사용하는 Python 프로젝트를 패키징하는 방법을 설명합니다. 이 방법은 네이티브 Python 패키지와 비교하여 다음과 같은 이점을 제공합니다.

- Python 및 비 Python 패키지에 대한 종속 항목을 사용할 수 있으며 DNF 패키지 관리자가 엄격하게 적용할 수 있습니다.
- 패키지를 암호화 방식으로 서명할 수 있습니다. 암호화 서명을 사용하면 RPM 패키지의 콘텐츠를 나머지 운영 체제와 검증, 통합 및 테스트할 수 있습니다.

- 빌드 프로세스 중에 테스트를 실행할 수 있습니다.

5.5.1. Python 패키지에 대한 SPEC 파일 설명

SPEC 파일에는 rpmbuild 유틸리티가 RPM을 빌드하는 데 사용하는 지침이 포함되어 있습니다. 지침은 여러 섹션에 포함되어 있습니다. SPEC 파일에는 섹션이 정의된 두 가지 주요 부분이 있습니다.

- Preamble (본문에 사용되는 일련의 메타데이터 항목 포함)
- 본문(명령의 주요 부분 포함)

Python 프로젝트의 RPM SPEC 파일에는 Python이 아닌 SPEC 파일에 비해 몇 가지 구체적인 내용이 있습니다.

중요

Python 라이브러리의 RPM 패키지 이름에는 항상 python3-, python3.11- 또는 python3.12- 접두사가 포함되어야 합니다.

기타 세부 사항은 python3*-pello 패키지에 대한 다음 SPEC 파일 예제에 표시되어 있습니다. 이러한 세부 사항에 대한 설명은 예제 아래의 노트를 참조하십시오.

Python으로 작성된 pello 프로그램의 사양 파일 예

```
%global python3_pkgversion 3.11                    (1)

Name:        python-pello                          (2)
Version:     1.0.2
Release:     1%{?dist}
Summary:     Example Python library

License:     MIT
URL:         https://github.com/fedora-python/Pello
Source:      %{url}/archive/v%{version}/Pello-%{version}.tar.gz

BuildArch:     noarch
BuildRequires:  python%{python3_pkgversion}-devel   (3)
```

```
# Build dependencies needed to be specified manually
BuildRequires:  python%{python3_pkgversion}-setuptools

# Test dependencies needed to be specified manually
# Also runtime dependencies need to be BuildRequired manually to run tests during build
BuildRequires:  python%{python3_pkgversion}-pytest >= 3


%global _description %{expand:
Pello is an example package with an executable that prints Hello World! on the command
line.}

%description %_description

%package -n python%{python3_pkgversion}-pello                    4
Summary:        %{summary}

%description -n python%{python3_pkgversion}-pello %_description


%prep
%autosetup -p1 -n Pello-%{version}


%build
# The macro only supported projects with setup.py
%py3_build                                                       5


%install
# The macro only supported projects with setup.py
%py3_install


%check                                                           6
%{pytest}


# Note that there is no %%files section for the unversioned python module
%files -n python%{python3_pkgversion}-pello
%doc README.md
%license LICENSE.txt
%{_bindir}/pello_greeting

# The library files needed to be listed manually
%{python3_sitelib}/pello/

# The metadata files needed to be listed manually
%{python3_sitelib}/Pello-*.egg-info/
```

1

python3_pkgversion 매크로를 정의하면 이 패키지가 빌드할 Python 버전을 설정합니다. 기본 Python 버전 3.9를 빌드하려면 매크로를 기본값 3 으로 설정하거나 행을 완전히 제거합니다.

2

Python 프로젝트를 RPM에 패키징할 때 항상 프로젝트의 원래 이름에 python- 접두사를 추가합니다. 원래 이름은 pello 이므로 소스 RPM(SRPM)의 이름은 python-pello 입니다.

3

BuildRequires 지시문은 이 패키지를 빌드하고 테스트하는 데 필요한 패키지를 지정합니다. BuildRequires 에서 항상 Python 패키지를 빌드하는 데 필요한 도구를 제공하는 항목이 포함되어 있습니다. python3-devel (또는 python3.11-devel 또는 python3.12-devel)과 같은 특정 소프트웨어에 필요한 관련 프로젝트 (예: python3-setuptools 또는 python3.11-setuptools) 또는 %check 섹션에서 테스트를 실행하는 데 필요한 런타임 및 테스트 종속 항목을 포함합니다.

4

바이너리 RPM의 이름(사용자가 설치할 수 있는 패키지)을 선택할 때 버전 지정된 Python 접두사를 추가합니다. 기본 Python 3.9, Python 3.11용 python3.11- 접두사 또는 Python 3.12용 python3.12- 접두사에 python3- 접두사를 사용합니다. 명시적 버전으로 설정하지 않는 한 기본 Python 버전 3.9에 대해 3 으로 평가되는 %{python3_pkgversion} 매크로를 사용할 수 있습니다 (예: 3.11 참조).

5

%py3_build 및 %py3_install 매크로는 설치 위치, 사용할 인터프리터 및 기타 세부 정보를 지정하는 추가 인수와 함께 setup.py build 및 setup.py install 명령을 각각 실행합니다.

6

%check 섹션은 패키지된 프로젝트의 테스트를 실행해야 합니다. 정확한 명령은 프로젝트 자체에 따라 다르지만 %pytest 매크로를 사용하여 RPM에 친숙한 방식으로 pytest 명령을 실행할 수 있습니다.

5.5.2. Python 3 RPM의 일반적인 매크로

SPEC 파일에서는 항상 값을 하드 코딩하지 않고 다음 *Macros for Python 3 RPMs* 테이블에 설명된 매크로를 사용합니다. SPEC 파일 상단에 python3_pkgversion 매크로를 정의하여 이러한 매크로에서 어떤 Python 3 버전이 사용되는지 확인할 수 있습니다(5.5.1절. “Python 패키지에 대한 SPEC 파일 설명”참조). python3_pkgversion 매크로를 정의하는 경우 다음 표에 설명된 매크로 값은 지정된 Python 3 버전을 반영합니다.

표 5.3. Python 3 RPM용 매크로

macro	일반 정의	설명
%{python3_pkgversion}	3	다른 모든 매크로에서 사용하는 Python 버전입니다. Python **3.11** 을 사용하거나 3.12로 Python **3.12** 를 사용하도록 3.11로 다시 정의할 수 있습니다.
%{python3}	/usr/bin/python3	Python 3 인터프리터
%{python3_version}	3.9	Python 3 인터프리터의 major.minor 버전
%{python3_sitelib}	/usr/lib/python3.9/site-packages	pure-Python 모듈이 설치된 위치
%{python3_sitearch}	/usr/lib64/python3.9/site-packages	아키텍처별 확장 모듈을 포함하는 모듈이 설치된 위치
%py3_build		RPM 패키지에 적합한 인수와 함께 **setup.py build** 명령 실행
%py3_install		RPM 패키지에 적합한 인수와 함께 **setup.py install** 명령 실행
%{py3_shebang_flags}	s	Python 인터프리터 지시문 매크로, **%py3_shebang_fix**의 기본 플래그 세트
%py3_shebang_fix		Python 인터프리터 지시문을 **#! %{python3}** 로 변경하고 기존 플래그(있는 경우)를 유지하고 **%{py3_shebang_flags}** 매크로에 정의된 플래그를 추가합니다.

추가 리소스

- 업스트림 문서의 **Python** 매크로

5.5.3. Python RPM에 자동 생성된 종속 항목 사용

다음 절차에서는 **Python** 프로젝트를 **RPM**으로 패키징할 때 자동으로 생성된 종속 항목을 사용하는 방법을 설명합니다.

사전 요구 사항

- **RPM**용 **SPEC** 파일이 있습니다. 자세한 내용은 **Python** 패키지에 대한 **SPEC** 파일 설명을 참조하십시오.

절차

1. 업스트림 제공 메타데이터가 포함된 다음 디렉터리 중 하나가 결과 **RPM**에 포함되어 있는지 확인합니다.

   - **.dist-info**
   - **.egg-info**

   RPM 빌드 프로세스는 다음과 같이 이러한 디렉터리에서 제공하는 가상 **pythonX.Ydist**를 자동으로 생성합니다.

   ```
   python3.9dist(pello)
   ```

   그런 다음 **Python** 종속성 생성기는 업스트림 메타데이터를 읽고 생성된 **pythonX.Ydist** 가상 기능을 사용하여 각 **RPM** 패키지에 대한 런타임 요구 사항을 생성합니다. 예를 들어 생성된 요구 사항 태그는 다음과 같을 수 있습니다.

   ```
   Requires: python3.9dist(requests)
   ```

2. 생성된 요구 사항을 검사합니다.

3. 생성된 요구 중 일부를 제거하려면 다음 방법 중 하나를 사용합니다.

   a. **SPEC** 파일의 **%prep** 섹션에서 업스트림 제공 메타데이터를 수정합니다.

   b. 업스트림 설명서에 설명된 종속성을 자동 필터링하여 사용합니다.

4. 자동 종속성 생성기를 비활성화하려면 기본 패키지의 **%description** 선언 위에 **%{?python_disable_dependency_generator}** 매크로를 포함합니다.

추가 리소스

- 자동 생성된 종속 항목

5.6. PYTHON 스크립트에서 인터프리터 지시문 처리

Red Hat Enterprise Linux 9에서는 실행 가능한 Python 스크립트는 최소 주요 Python 버전에서 명시적으로 지정하는 인터프리터 지시문(hashbangs 또는 shebangs라고도 함)을 사용해야 합니다. 예를 들어 다음과 같습니다.

```
#!/usr/bin/python3
#!/usr/bin/python3.9
#!/usr/bin/python3.11
#!/usr/bin/python3.12
```

RPM 패키지를 빌드할 때 /usr/lib/rpm/redhat/brp-mangle-shebangs BRP(Buildroot 정책) 스크립트를 자동으로 실행하고 모든 실행 파일에서 인터프리터 지시문을 수정하려고 시도합니다.

BRP 스크립트는 다음과 같은 모호한 인터프리터 지시문을 사용하여 Python 스크립트를 시작할 때 오류를 생성합니다.

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

또는

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

5.6.1. Python 스크립트에서 인터프리터 지시문 수정

다음 절차에 따라 RPM 빌드 시 빌드 오류가 발생하는 Python 스크립트에서 인터프리터 지시문을 수정합니다.

사전 요구 사항

- Python 스크립트의 인터프리터 지시문 중 일부는 빌드 오류가 발생합니다.

절차

- 인터프리터 지시문을 수정하려면 다음 작업 중 하나를 완료합니다.

- SPEC 파일의 %prep 섹션에서 다음 매크로를 사용합니다.

```
# %py3_shebang_fix SCRIPTNAME ...
```

SCRIPTNAME 은 모든 파일, 디렉터리 또는 파일 및 디렉터리 목록일 수 있습니다.

결과적으로 나열된 디렉터리의 모든 파일 및 모든 .py 파일은 %{python3} 을 가리키도록 인터프리터 지시문을 수정합니다. 원래 인터프리터 지시문의 기존 플래그는 보존되고 %{py3_shebang_flags} 매크로에 정의된 추가 플래그가 추가됩니다. SPEC 파일에서 %{py3_shebang_flags} 매크로를 재정의하여 추가할 플래그를 변경할 수 있습니다.

- python3-devel 패키지에서 pathfix.py 스크립트를 적용합니다.

```
# pathfix.py -pn -i %{python3} PATH ...
```

여러 경로를 지정할 수 있습니다. *PATH* 가 디렉터리인 경우 pathfix.py 는 모호한 인터프리터 지시문이 있는 것뿐만 아니라 ^[a-zA-Z0-9_]+\.py$ 패턴과 일치하는 Python 스크립트를 반복적으로 검사합니다. 위의 명령을 %prep 섹션에 추가하거나 %install 섹션의 끝에 추가합니다.

- 패키지된 Python 스크립트를 수정하여 예상 형식을 준수하도록 합니다. 이를 위해 RPM 빌드 프로세스 외부의 pathfix.py 스크립트도 사용할 수 있습니다. RPM 빌드 외부에서 pathfix.py 를 실행하는 경우 이전 예제의 %{python3} 을 /usr/bin/python3 또는 /usr/bin/python3.11 과 같은 인터프리터 지시문의 경로로 바꿉니다.

추가 리소스

- 인터프리터 호출

5.7. RUBYGEMS 패키지

이 섹션에서는 RubyGems 패키지가 무엇인지, 그리고 RPM으로 다시 패키징하는 방법에 대해 설명합니다.

5.7.1. RubyGems의 정의

Ruby는 동적, 해석, 반사, 객체 지향, 일반 목적의 프로그래밍 언어입니다.

Ruby로 작성된 프로그램은 일반적으로 특정 Ruby 패키징 형식을 제공하는 RubyGems 프로젝트를 사용하여 패키징됩니다.

RubyGems에서 만든 패키지는 gems라고 하며 RPM으로도 다시 패키징할 수 있습니다.

참고

이 문서는 gem 접두사와 함께 RubyGems 개념과 관련된 용어(예: .gemspec)는 gem 사양에 사용되며 RPM 관련 용어는 정규화되지 않습니다.

5.7.2. RubyGems의 RPM 관련 방법

RubyGems는 Ruby의 자체 패키징 형식을 나타냅니다. 그러나 RubyGems에는 RPM에 필요한 메타데이터가 포함되어 있으며 RubyGems에서 RPM으로 변환할 수 있습니다.

Ruby Packaging guidelines 에 따르면 RubyGems 패키지를 RPM으로 다시 패키징할 수 있습니다.

- 이러한 RPM은 나머지 배포에 적합합니다.
- 최종 사용자는 적절한 RPM 패키지 gem을 설치하여 gem의 종속성을 충족할 수 있습니다.

RubyGems는 사양 파일, 패키지 이름, 종속성 및 기타 항목과 같은 RPM과 유사한 용어를 사용합니다.

나머지 RHEL RPM 배포에 적합하려면 RubyGems에서 생성한 패키지는 아래 나열된 규칙을 따라야 합니다.

- gems의 이름은 다음 패턴을 따라야 합니다.

```
rubygem-%{gem_name}
```

- shebang 라인을 구현하려면 다음 문자열을 사용해야 합니다.

```
#!/usr/bin/ruby
```

5.7.3. RubyGems 패키지에서 RPM 패키지 생성

RubyGems 패키지에 대한 소스 RPM을 만들려면 다음 파일이 필요합니다.

- gem 파일
- RPM 사양 파일

다음 섹션에서는 RubyGems에서 만든 패키지에서 RPM 패키지를 만드는 방법에 대해 설명합니다.

5.7.3.1. RubyGems 사양 파일 규칙

RubyGems 사양 파일은 다음 규칙을 충족해야 합니다.

- gem의 사양의 이름인 %{gem_name} 를 포함합니다.
- 패키지 소스는 릴리스된 gem 아카이브에 대한 전체 URL이어야 합니다. 패키지 버전은 gem의 버전이어야 합니다.
- 빌드에 필요한 매크로를 가져올 수 있도록 다음과 같이 정의된 지시문인 BuildRequires 가 포함되어 있습니다.

```
BuildRequires:rubygems-devel
```

- RubyGems Requires 또는 Provides 가 자동으로 생성되기 때문에 포함되어 있지 않습니다.
- Ruby 버전 호환성을 명시적으로 지정하지 않으려면 다음과 같이 정의된 BuildRequires: 지시문을 포함하지 않습니다.

> **Requires: ruby(release)**

RubyGems에 자동으로 생성된 종속성(필수: ruby(rubygems))이면 충분합니다.

5.7.3.2. RubyGems 매크로

다음 표에는 RubyGems에서 만든 패키지에 유용한 매크로가 나열되어 있습니다. 이러한 매크로는 rubygems-devel 패키지에서 제공합니다.

표 5.4. RubyGems의 매크로

매크로 이름	확장 경로	사용법
%{gem_dir}	/usr/share/gems	gem 구조의 최상위 디렉터리입니다.
%{gem_instdir}	%{gem_dir}/gems/%{gem_name}-%{version}	gem의 실제 콘텐츠가 있는 디렉터리입니다.
%{gem_libdir}	%{gem_instdir}/lib	gem의 라이브러리 디렉터리입니다.
%{gem_cache}	%{gem_dir}/cache/%{gem_name}-%{version}.gem	캐시된 gem.
%{gem_spec}	%{gem_dir}/specifications/%{gem_name}-%{version}.gemspec	gem 사양 파일입니다.
%{gem_docdir}	%{gem_dir}/doc/%{gem_name}-%{version}	gem의 RDoc 문서입니다.
%{gem_extdir_mri}	%{_libdir}/gems/ruby/%{gem_name}-%{version}	gem 확장을 위한 디렉터리입니다.

5.7.3.3. RubyGems 사양 파일 예

특정 섹션에 대한 설명과 함께 gem을 빌드하는 사양 파일의 예는 다음과 같습니다.

예제 RubyGems 사양 파일

```
%prep
%setup -q -n  %{gem_name}-%{version}

# Modify the gemspec if necessary
# Also apply patches to code if necessary
%patch0 -p1

%build
# Create the gem as gem install only works on a gem file
gem build ../%{gem_name}-%{version}.gemspec

# %%gem_install compiles any C extensions and installs the gem into ./%%gem_dir
# by default, so that we can move it into the buildroot in %%install
%gem_install

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

# If there were programs installed:
mkdir -p %{buildroot}%{_bindir}
cp -a ./%{_bindir}/* %{buildroot}%{_bindir}

# If there are C extensions, copy them to the extdir.
mkdir -p %{buildroot}%{gem_extdir_mri}
cp -a .%{gem_extdir_mri}/{gem.build_complete,*.so} %{buildroot}%{gem_extdir_mri}/
```

다음 표에서는 **RubyGems** 사양 파일의 특정 항목에 대한 세부 사항을 설명합니다.

표 5.5. RubyGems의 spec 지시문 특정

directive	RubyGems 세부 정보
%Prep	RPM은 gem 아카이브의 직접 압축을 풀 수 있으므로 **gem에서 소스를 추출하기 위해 gem 압축** 을 풀 수 있습니다.**%setup -n %{gem_name}-%{version}** 매크로는 gem의 압축을 풀 수 있는 디렉터리를 제공합니다. 동일한 디렉터리 수준에서 **%{gem_name}-%{version} .gemspec** 파일이 자동으로 생성됩니다. 이 파일은 나중에 gem을 다시 빌드하거나 .gemspec을 수정하거나 .gemspec을 코드에 패치를 적용하는 데 사용할 수 있습니다.

directive	RubyGems 세부 정보
%build	이 지시문에는 소프트웨어를 머신 코드로 빌드하는 명령 또는 일련의 명령이 포함되어 있습니다. **%gem_install** 매크로는 gem 아카이브에서만 작동하며 gem은 다음 gem 빌드로 다시 생성됩니다. 그런 다음 **%gem_install** 에서 생성된 gem 파일은 기본적으로 **./%{gem_dir}** 인 임시 디렉터리에 코드를 빌드하고 설치하는 데 사용됩니다. **%gem_install** 매크로는 한 단계로 코드를 빌드하고 설치합니다. 설치하기 전에 빌드된 소스는 자동으로 생성된 임시 디렉터리에 배치됩니다. **%gem_install** 매크로는 설치에 사용되는 gem을 재정의할 수 있는 **-n <gem_file>**, gem 설치 대상을 재정의할 수 있는 **-d <install_dir>**; 이 옵션을 사용하는 것은 권장되지 않습니다. **%gem_install** 매크로는 **%{buildroot}** 에 설치하는 데 사용해서는 안 됩니다.
%install	설치는 **%{buildroot}** 계층 구조로 수행됩니다. 필요한 디렉터리를 생성한 다음 임시 디렉터리에 설치된 디렉터리를 **%{buildroot}** 계층 구조로 복사할 수 있습니다. 이 gem이 공유 오브젝트를 생성하는 경우 아키텍처별 **%{gem_extdir_mri}** 경로로 이동합니다.

추가 리소스

- Ruby 패키징 지침

5.7.3.4. gem2rpm을 사용하여 RubyGems 패키지를 RPM 사양 파일로 변환

gem2rpm 유틸리티는 RubyGems 패키지를 RPM 사양 파일로 변환합니다.

다음 섹션에서는 다음 방법을 설명합니다.

- **gem2rpm 유틸리티 설치**
- 모든 **gem2rpm 옵션 표시**
- **gem2rpm** 을 사용하여 **RubyGems** 패키지를 **RPM** 사양 파일로 변환
- **gem2rpm 템플릿 편집**

5.7.3.4.1. gem2rpm 설치

다음 절차에서는 gem2rpm 유틸리티를 설치하는 방법을 설명합니다.

절차

- RubyGems.org 에서 gem2rpm 을 설치하려면 다음을 실행합니다.

```
$ gem install gem2rpm
```

5.7.3.4.2. gem2rpm의 모든 옵션 표시

다음 절차에서는 gem2rpm 유틸리티의 모든 옵션을 표시하는 방법을 설명합니다.

절차

- gem2rpm 의 모든 옵션을 보려면 다음을 실행합니다.

```
$ gem2rpm --help
```

5.7.3.4.3. gem2rpm을 사용하여 RubyGems 패키지를 RPM 사양 파일로 변환

다음 절차에서는 gem2rpm 유틸리티를 사용하여 RubyGems 패키지를 RPM 사양 파일로 변환하는 방법을 설명합니다.

절차

- 최신 버전에서 gem을 다운로드하고 이 gem에 대한 RPM 사양 파일을 생성합니다.

```
$ gem2rpm --fetch <gem_name> > <gem_name>.spec
```

설명된 절차에서는 gem의 메타데이터에 제공된 정보를 기반으로 RPM 사양 파일을 생성합니다. 그러나 gem은 라이센스 및 변경 로그와 같은 RPM에서 일반적으로 제공되는 몇 가지 중요한 정보를 놓치고 있습니다. 따라서 생성된 사양 파일을 편집해야 합니다.

5.7.3.4.4. gem2rpm 템플릿

gem2rpm 템플릿은 다음 표에 나열된 변수를 포함하는 표준 임베디드 Ruby(ERB) 파일입니다.

표 5.6. gem2rpm 템플릿의 변수

Variable	설명
패키지	gem의 **Gem::Package** 변수.
spec	gem의 **Gem::Specification** 변수(format.spec과 동일).
config	사양 템플릿 도우미에 사용되는 기본 매크로 또는 규칙을 재정의할 수 있는 **Gem2Rpm::Configuration** 변수.
runtime_dependencies	**Gem2Rpm::RpmDependencyList** 변수는 패키지 런타임 종속 항목 목록을 제공합니다.
development_dependencies	**Gem2Rpm::RpmDependencyList** 변수는 패키지 개발 종속 항목 목록을 제공합니다.
테스트	**Gem2Rpm::TestSuite** 변수는 실행할 수 있는 테스트 프레임워크 목록을 제공합니다.
파일	**Gem2Rpm::RpmFileList** 변수는 패키지에서 필터링되지 않은 파일 목록을 제공합니다.
main_files	**Gem2Rpm::RpmFileList** 변수는 기본 패키지에 적합한 파일 목록을 제공합니다.
doc_files	**Gem2Rpm::RpmFileList** 변수는 **-doc** 하위 패키지에 적합한 파일 목록을 제공합니다.
형식	gem의 **Gem::Format** 변수. 이 변수는 더 이상 사용되지 않습니다.

5.7.3.4.5. 사용 가능한 gem2rpm 템플릿 나열

다음 절차에서는 사용 가능한 모든 gem2rpm 템플릿을 나열하도록 설명합니다.

절차

- 사용 가능한 모든 템플릿을 보려면 다음을 실행합니다.

```
$ gem2rpm --templates
```

5.7.3.4.6. gem2rpm 템플릿 편집

생성된 사양 파일을 편집하는 대신 RPM 사양 파일이 생성되는 템플릿을 편집할 수 있습니다.

gem2rpm 템플릿을 편집하려면 다음 절차를 사용하십시오.

절차

1. 기본 템플릿을 저장합니다.

```
$ gem2rpm -T > rubygem-<gem_name>.spec.template
```

2. 필요에 따라 템플릿을 편집합니다.

3. 편집된 템플릿을 사용하여 사양 파일을 생성합니다.

```
$ gem2rpm -t rubygem-<gem_name>.spec.template <gem_name>-<latest_version.gem
> <gem_name>-GEM.spec
```

이제 RPM 빌드에 설명된 대로 편집된 템플릿을 사용하여 RPM 패키지를 빌드할 수 있습니다.

5.8. PERLS 스크립트를 사용하여 RPM 패키지를 처리하는 방법

RHEL 8부터 Perl 프로그래밍 언어는 기본 buildroot에 포함되지 않습니다. 따라서 Perl 스크립트가 포함된 RPM 패키지는 RPM 사양 파일에서 BuildRequires: 지시문을 사용하여 Perl에 대한 종속성을 명시적으로 표시해야 합니다.

5.8.1. 공통 Perl 관련 종속 항목

BuildRequires 에 사용되는 Perl 관련 빌드 종속 항목은 다음과 같습니다.

- perl-generators

  런타임 Requires 를 자동으로 생성하고 설치된 Perl 파일을 제공합니다. Perl 스크립트 또는 Perl 모듈을 설치할 때 이 패키지에 대한 빌드 종속성을 포함해야 합니다.

- perl-interpreter

  Perl 인터프리터는 perl 패키지 또는 %_perl 매크로를 통해 명시적으로 또는 패키지 빌드 시

스템의 일부로 명시적으로 호출되는 경우 빌드 종속성으로 나열되어야 합니다.

- **perl-devel**

  Perl 헤더 파일을 제공합니다. **XS Perl** 모듈과 같은 **libperl.so** 라이브러리에 연결하는 아키텍처별 코드를 빌드하는 경우, **BuildRequires: perl-devel.**

5.8.2. 특정 Perl 모듈 사용

빌드 시 특정 **Perl** 모듈이 필요한 경우 다음 절차를 사용하십시오.

절차

- **RPM** 사양 파일에 다음 구문을 적용합니다.

```
BuildRequires: perl(MODULE)
```

참고

시간이 지남에 따라 **perl** 패키지로 이동할 수 있기 때문에 **Perl core** 모듈에도 이 구문을 적용합니다.

5.8.3. 패키지를 특정 Perl 버전으로 제한

패키지를 특정 **Perl** 버전으로 제한하려면 다음 절차를 따르십시오.

절차

- **RPM** 사양 파일에서 원하는 버전 제약 조건과 함께 **perl(:VERSION)** 종속성을 사용합니다.

  예를 들어 패키지를 **Perl** 버전 **5.30** 이상으로 제한하려면 다음을 사용하십시오.

```
BuildRequires: perl(:VERSION) >= 5.30
```

주의

epoch 번호가 포함되어 있기 때문에 **perl** 패키지 버전에 대한 비교를 사용하지 마십시오.

5.8.4. 패키지가 올바른 Perl 인터프리터를 사용하는지 확인

Red Hat은 완전히 호환되지 않는 여러 **Perl** 인터프리터를 제공합니다. 따라서 **Perl** 모듈을 제공하는 모든 패키지는 빌드 시 사용된 것과 동일한 **Perl** 인터프리터를 런타임에 사용해야 합니다.

이를 확인하려면 다음 절차를 따르십시오.

절차

- **Perl** 모듈을 제공하는 모든 패키지에 대해 버전이 지정된 **MODULE_COMPAT** 요구 사항을 **RPM** 사양 파일에 포함합니다.

```
Requires:  perl(:MODULE_COMPAT_%(eval `perl -V:version`; echo $version))
```

6장. RHEL 9의 새로운 기능

이 섹션에서는 Red Hat Enterprise Linux 8과 9 사이의 RPM 패키지의 주요 변경 사항을 문서화합니다.

6.1. 동적 빌드 종속 항목

Red Hat Enterprise Linux 9에는 동적 빌드 종속 항목을 생성할 수 있는 %generate_buildrequires 섹션이 도입되었습니다.

이제 새로 사용 가능한 %generate_buildrequires 스크립트를 사용하여 RPM 빌드 시 추가 빌드 종속 항목을 프로그래밍 방식으로 생성할 수 있습니다. 이 기능은 특수 유틸리티가 일반적으로 러스트, Golang, Node.js, Ruby, Python 또는 Haskell과 같은 런타임 종속성을 결정하는 데 사용되는 언어로 작성된 소프트웨어를 패키징할 때 유용합니다.

%generate_buildrequires 스크립트를 사용하여 빌드 시 SPEC 파일에 추가된 BuildRequires 지시문을 동적으로 확인할 수 있습니다. 존재하는 경우 %generate_buildrequires 는 %prep 섹션 다음에 실행되며 압축 풀기 및 패치된 소스 파일에 액세스할 수 있습니다. 스크립트는 일반 BuildRequires 지시문과 동일한 구문을 사용하여 확인된 빌드 종속 항목을 표준 출력에 출력해야 합니다.

그런 다음 rpmbuild 유틸리티는 빌드를 계속하기 전에 종속성이 충족되는지 확인합니다.

일부 종속 항목이 누락된 경우 .buildreqs.nosrc.rpm 접미사가 있는 패키지가 생성되어 있으며 여기에는 확인된 BuildRequires 및 소스 파일이 포함되어 있습니다. 이 패키지를 사용하여 빌드를 다시 시작하기 전에 dnf builddep 명령으로 누락된 빌드 종속 항목을 설치할 수 있습니다.

자세한 내용은 rpmbuild(8) 도움말 페이지의 DYNAMIC BUILD DEPENDENCIES 섹션을 참조하십시오.

추가 리소스

- rpmbuild(8) 도움말 페이지
- yum-builddep(1) 매뉴얼 페이지

6.2. 개선된 패치 선언

6.2.1. 자동 패치 및 소스 번호 지정

Patch: 및 **Source:** 숫자가 없는 태그는 이제 나열된 순서에 따라 자동으로 번호가 지정됩니다.

번호 매기기는 마지막 수동으로 번호가 매겨진 항목부터 **rpmbuild** 유틸리티에 의해 내부적으로 실행되거나 이러한 항목이 없는 경우 **0** 이 실행됩니다.

예를 들어 다음과 같습니다.

```
Patch: one.patch
Patch: another.patch
Patch: yet-another.patch
```

6.2.2. %patchlist 및 %sourcelist 섹션

이제 새로 추가된 **%patchlist** 및 **%sourcelist** 섹션을 사용하여 각 항목 앞에 각 항목 없이 패치 및 소스 파일을 나열할 수 있습니다.

예를 들어 다음 항목은 다음과 같습니다.

```
Patch0: one.patch
Patch1: another.patch
Patch2: yet-another.patch
```

이제 다음과 같이 교체할 수 있습니다.

```
%patchlist
one.patch
another.patch
yet-another.patch
```

6.2.3. %autopatch 이제 패치 범위를 수락

이제 **%autopatch** 매크로는 **-m** 및 **-M** 매개변수를 수락하여 적용할 최소 및 최대 패치 번호를 제한합니다.

- **m** 매개변수는 패치를 적용할 때 시작할 패치 번호(포함)를 지정합니다.

- **-M** 매개변수는 패치 적용 시 중지할 패치 번호(포함)를 지정합니다.

이 기능은 특정 패치 세트 간에 작업을 수행해야 하는 경우에 유용할 수 있습니다.

6.3. 기타 기능

Red Hat Enterprise Linux 9의 **RPM** 패키징과 관련된 기타 새로운 기능은 다음과 같습니다.

- 빠른 매크로 기반 종속성 생성기
- 다이어리 연산자 및 네이티브 버전 비교를 포함한 강력한 매크로 및 **%if** 표현식
- 메타 종속성(**unordered**) 종속 항목
- 캐럿 버전 운영자(**^**)는 기본 버전보다 큰 버전을 표시하는 데 사용할 수 있습니다. 이 연산자는 반대 의미를 갖는 틸드(**~**) 연산자를 보완합니다.
- **%elif**,**%elifos** 및 **%elifarch** 문

7장. 추가 리소스

RPM, RPM 패키징 및 RPM 빌드와 관련된 다양한 주제에 대한 참조는 다음과 같습니다.

- mock
- RPM 문서
- RPM 4.15.0 릴리스 정보
- RPM 4.16.0 릴리스 노트
- Fedora 패키징 지침