RPM Paketleme Rehberi

Adam Miller, Maxim Svistunov, Marie Doleželová, et al.

Çeviri: Emrecan Şuşter

Table of Contents

Tanıtım

RPM Paketleme açıklayıcı dokümanları:

Kaynak kodunu RPM paketlemek için hazırlamak.

Yazılım geliştirmek konusunda hiçbir fikriniz yoksa sizi şöyle alalım: Yazılımı Paketlemeye Hazırlamak.

Kaynak kodunu RPM olarak paketlemek.

Bu kısım paketlerini RPM olarak paketlemek isteyen yazılımcılar içindir. Buyrun: Yazılımı Paketlemek.

Gelişmiş paketleme senaryoları.

Bu kısım, RPM paketçilerinin karşısına çıkan zorlu RPM paketleme senaryoları için bir kaynakçadır. İleri Düzey Konular kısmını inceleyin.

PDF Versiyonu

Dilerseniz şu adresten PDF versiyonunu inceleyebilirsiniz: https://tarbetu.github.io/rpm_paketleme/rpm-packaging-guide.pdf

Doküman Anahtarları

Bu doküman anlatımı kolaylaştırmak üzere şöyle anahtarlar kullanır:

- Bloklar hâlindeki yerleştirilmiş kaynak kodlarını içeren, komut çıktısını ve içeriğini barındıran metin dosyaları:

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

[komut çıktısı sadeleştirildi]
```

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

[komut çıktısı sadeleştirildi]
```

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

print("Merhaba Dünya")
```

- Kendisine has anlamları olan konular veya başka bir sitedeki kaynakla ilişkili terimler **kalınla** ya da *eğik yazıyla* gösterilir.
- Bazı yazılımlar, komutlar ve komut içerisinde bulunabilecek şeyler `eş aralıklı` fontlarla gösterilir.

Ön Koşullar

Bu kılavuzu takip etmek için bazı paketlere ihtiyacınız olacak.

NOTE	Bazı paketler varsayılan olarak şu sistemlerde kuruludur: Fedora, CentOS ve RHEL. Bu paketler hangi araçların kullanıldığını açıkça göstermek için listelenmiştir.

Fedora, CentOS 8 ve RHEL 8 üzerinde:

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

CentOS 7 ve RHEL 7 üzerinde:

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

Neden Yazılımları RPM ile Paketlemeliyiz?

RPM Paket Yöneticisi (RPM), Red Hat Enterprise Linux, CentOS ve Fedora'da kullanılan bir paket yönetim sistemidir. RPM, bahsi edilen bu dağıtımlar üzerinde yazılımlarızı dağıtmaya, yönetmeye ve güncellemeye yardımcı olur. Pek çok yazılımcı bir gelenek olarak kendi yazılımlarını tarball olarak da bilinen arşiv dosyaları aracılığıyla dağıtır. Buna karşın, yazılımları RPM formatında paketlemenin pek çok faydası vardır. Bu faydalar, aşağıda şu şekilde anlatılmıştır.

RPM ile şunları yapabilirsiniz:

Paketleri kurma, yeniden kurma, kaldırma, güncelleme ve onaylama

Kullanıcılar, standart paket yönetim araçlarıyla (Yum veya PackageKit gibi) sizin RPM paketleriyle kurabilir, yeniden kurabilir, kaldırabilir ve onaylayabilir.

Paketleri onaylamak ve sıralamak için kurulmuş paketlerin bir veritabanını kurma

RPM'in kurulmuş paketler ve onların dosyaları için sağladığı veritabanı sayesinde kullanıcılar, kendi sistemlerindeki paketlerini rahatlıkla sıralayabilir ve onaylayabilir.

Paketlerinizi açıklamak, onların kurulum talimatlarını anlatmak ve buna benzer işler için üstverileri (metadata) kullanın

Her bir RPM paketi kendi paketinin öğeleri, versiyonu, yayın numarasını, büyüklüğünü, proje adresini ve kurulum talimatları gibi pek çok bilgiyi vermek için bir üstveri (metadata) içerir.

Saf yazılım kaynakları aracılığıyla paketinizi kaynak ve ikili hâlde paketleyin

RPM, saf yazılım kaynağına erişmenize ve onları ikili ve kaynak paketler hâlinde kullanıcılarınıza iletmenize izin verir. Kaynak paketlerde saf kaynaklara üzerine uygulanmış kodlarla ve derleme talimatlarına erişebilirsiniz. Bu tasarım, yazılımların yeni sürümleri yayınlandıkça yeni sürümlerin yayınlanabilirliğini kolaylaştırır.

Yum depolarına paketler ekleyin

Paketlerinizi alıcılarınızın kolaylıkla paketlerinizi bulup kurabileceği Yum depolarına ekleyebilirsiniz.

Paketlerinizi dijital olarak imzalayın

GPG imza anahtarı aracılığıyla, paketlerinizi dijital olarak imzalayabilirsiniz. Böylece kullanıcılar paketlerinizin güvenilirliğini onaylayabilir.

İlk RPM Paketiniz

Bir RPM paketi kurgulamak gayet karmaşık olabilir. Daha anlaşılır olması açısından burada sadeleştirilmiş ve bazı noktaları atlanmış bir RPM Spec dosyası hazırladık.

```
Name:       merhaba-dunya
Version:    1
Release:    1
Summary:    Epey sade bir RPM paketi
License:    FIXME

%description
Bu benim ilk RPM paketim. Süs bitkisi niyetine sisteminize kurabilirsiniz.

%prep
# bir kaynağımız yok, hâliyle buraya hiçbir şey eklemedik.
%build
cat > merhaba-dünya.sh <<EOF
#!/usr/bin/bash
echo Merhaba dünya
EOF

%install
mkdir -p %{buildroot}/usr/bin/
install -m 755 merhana-dünya.sh %{buildroot}/usr/bin/merhaba-dünya.sh

%files
/usr/bin/merhaba-dünya.sh

%changelog
# Şimdilik boş verin gitsin
```

Bu dosyayı `merhaba-dunya.spec` ismiyle kaydedin.

Ardından, şu komutları kullanın:

```
$ rpmdev-setuptree
$ rpmbuild -ba merhaba-dunya.spec
```

`rpmdev-setuptree` isimli komut çeşitli çalışma dizinleri hazırlar. Bu dizinler, ev dizininizde ($HOME) saklanacağı için bu komutun bir kere kullanılması yeterlidir.

`rpmbuild` komutu ise rpm paketini hazırlar. Bu komutun çıktı şöyle bir şeye benzer:

```
... [KESTİK!]
Wrote: /home/mirek/rpmbuild/SRPMS/merhaba-dunya-1-1.src.rpm
Wrote: /home/mirek/rpmbuild/RPMS/x86_64/merhaba-dunya-1-1.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.wgaJzv
+ umask 022
+ cd /home/mirek/rpmbuild/BUILD
+ /usr/bin/rm -rf /home/mirek/rpmbuild/BUILDROOT/merhaba-dunya-1-1.x86_64
+ exit 0
```

`/home/mirek/rpmbuild/RPMS/x86_64/merhaba-dunya-1-1.x86_64.rpm` sizin ilk RPM paketinizdir. Sisteminize kurabilir ve test edebilirsiniz.

Yazılımı Paketlemeye Hazırlamak

Bu bölüm bir RPM paketçisinin bilmesi gereken temel konular olan kaynak kodundan ve yazılım oluşturmaktan bahseder.

Kaynak kodu nedir?

Kaynak kodu bir dosyanın bilgisayara nasıl derlenmesi gerektiğinin anlatıldığı ve insanlar tarafından okunabilen talimatlardır. Kaynak kodları, bir programlama dili aracılığıyla hazırlanır. Bkz: Programlama Dili

Bu klavuz farklı programlama dillerinde yazılmış `Merhaba Dünya` programının üç sürümünü ele alır. Farklı programlama dilleriyle yazılmış programlar farklı şekilde paketlenir ve bir RPM paketçisinin karşılaşacağı üç farklı durumu özetler.

Not: Yüzden fazla programlama dili vardır. Bu belge, yalnızca üç tanesini ele alır fakat kavramsal bir gözden geçirme için yeterlidir.

Bash ile yazılmış bir `Merhaba Dünya`:

`belaba`

```
#!/bin/bash

printf "Merhaba Dünya\n"
```

Python ile yazılmış bir `Merhaba Dünya`:

`pelaba.py`

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

print("Merhaba Dünya")
```

C ile yazılmış bir `Merhaba Dünya`:

`celaba.c`

```
#include <stdio.h>

int main(void) {
    printf("Merhaba Dünya\n");
    return 0;
}
```

Yukarıdaki üç programın da amacı komut satırına "Merhaba Dünya" yazmaktır.

Not: Programlamayı bilmek yazılım paketlemek için gerekli değildir ama faydalıdır.

Programlar Nasıl Yapılır?

İnsan tarafından okunabilen kaynak kodunu bilgisayarın takip ettiği talimatları sunan makine koduna çevirmek için pek çok yöntem vardır. Bununla birlikte, bu yöntemler üç taneye indirgenebilir:

1. Program yerel olarak derlenebilir.
2. Program ham hâliyle yorumlanabilir.
3. Program bayt derleme ile yorumlanabilir.

Yerel Olarak Derlenmiş Kod

(Mimariye) Yerel derlenmiş yazılım, makine koduna dönüştürülen bir programlama dilinde yazılmış yazılımdır. Bu tarz yazılımlar tek başına, çalıştırılabilir ikili dosyalar olarak çalıştırılabilir.

Bu yolla derlenmiş RPM paketleri belirli bir mimariye yöneliktir. Yani, sizin 64 bit (x86_64) mimarisine yönelik hazırladığınız yazılım 32 bit (x86) mimarisinde çalıştırılamaz. Yazılımların paketlerinin isimlerinde hangi mimariyi hedef aldığı belirtilir.

Yorumlanan Kod

bash ve Python gibi bazı programlama dilleri doğrudan makine diline derlenmez. Bunun yerine, bu dillerin dillerde yazılan kaynak kodları herhangi bir ön dönüşüme tabii tutulmaksızın adım adım çalıştılır. Bu kodlar bir Dil Yorumlayıcısı tarafından veya Dil Sanal Makinesi tarafından işlenir.

Bu yazılımlar belirli bir mimariye yönelik değildir. Dolayısıyla, bu yazılımların RPM paketlerinin isminde `noarch` ibaresi bulunur.

Yorumlanan bir dil ya **bayt olarak derlenebilir** ya da **ham hâliyle yorumlanabilir**. Bu iki farklı tipin inşa süreci ve paketleme yöntemi birbirinden farklıdır.

Ham olarak yorumlanan programlar

Ham hâliyle yorumlanan dillerin programlarının derlenmeye ihtiyacı yoktur, direkt yorumlanabilirler.

Bayt derlenen programlar

Bayt derlenen dillerin bayt koduna derlenmeye ihtiyacı vardır. Bu programlar dil sanal makineleri tarafından çalıştırılırlar.

Not: Bazı diller ham olarak yorumlanmak veya bayt koduna derlenmek arasında bir seçenek sunar.

Yazılımı Kaynaktan İnşa Etmek

Bu kısım, kaynak kodundan bir yazılımı inşa etmeyi açıklar.

- Derlenen dillerde yazılan yazılımlar, bir **inşa** sürecinden geçer ve makine kodu üretilir. Bu süreç, yaygın olarak **derleme** veyahut **çeviri** olarak isimlendirilir, programlama dilleri açısından farklılık gösterir. Derleme sonrasında program **çalıştırılır** veyahut **yürütülür** ve yazılımcının belirlediği görevleri yerine getirir. (Ç.N: Orijinal belgede **run** ve **executed** ifadelerini sırasıyla **çalıştırma** ve **yürütme** olarak çevirdim.)
- Ham olarak yorumlanan dillerle yazılmış dillerde yazılım inşa edilmez, doğrudan çalıştırılır.
- Bayt derlenerek yorumlanan dillerle yazılmış dillerde ise kaynak kodu bayt kodu olarak derlenir ve dil sanal makinesi tarafından çalıştırılır.

Yerel Derlenen Kod

Bu örnekte C ile yazılmış `celaba.c`dosyasını çalıştırılabilir bir dosyaya dönüştüreceksiniz.

`celaba.c`

```
#include <stdio.h>

int main(void) {
    printf("Merhaba Dünya\n");
    return 0;
}
```

Elle İnşa

C kaynak kodlarını derlemek için kullanılan "GNU Derleyici Koleksiyonu"nu (GCC) çalıştıracağız.

```
gcc -g -o celaba celaba.c
```

Çıktı olarak gelen `celaba` dosyasını çalıştırın.

```
$ ./celaba
Merhaba Dünya
```

İşte, hepsi bu. Yerel olarak derlenen bir yazılımı kaynak kodundan çalıştırıp derlemiş oldunuz.

Otomatik İnşa

Kaynak kodunu elle inşa etmek yerine inşa sürecini otomatikleştirebilirsiniz. Büyük ölçekli projeler için yaygın bir uygulamadır. Otomatik inşa, `Makefile` isimli bir dosya oluşturup GNU `make` aracını çağırmakla gerçekleştirilebilir.

Otomatik bir inşa kurmak için, `celaba.c` ile aynı klasörde `Makefile` isimli bir dosya oluşturun:

`Makefile`

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

clean:
	rm celaba
```

Bu yazımlımı inşa etmek için, yalnızca `make` komutunu çalıştırın:

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

Daha önceden yapılmış bir inşayı temizlemek için `make clean` komutunu çalıştırın, ardından tekrar `make` komutunu çalıştırın:

```
$ make clean
rm celaba

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

Yine, hiçbir şey olmayacağını bile bile tekrar bir inşaya teşebbüs edelim:

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

Güzel, programı çalıştırabiliriz:

```
$ ./celaba
Merhaba Dünya
```

Hem elle, hem de otomatik olarak bir programı derlemiş bulunmaktasınız.

Yorumlanan Kod

İnceleyeceğimiz iki örnekten birisi Python ile yazılmış bayt olarak derlenen bir program, diğeri ise Bash ile yazılmış ve ham olarak yorumlanan başka bir program.

NOTE

İki örnekte de göreceğiniz, dosyanın başındaki `#!` satırı programlama dilinin bir parçası değildir ve mevzu (shebang) olarak anılır. (Bağlantı İngilizcedir.)

Mevzular bir yazı dosyasının çalıştırılabilir bir dosya olarak ele alınmasına izin verir. Sistemin program yükleyicisi bu satırı özellikle arar ve bu satır üzerinde bulunan ikili programla çalıştırır. Bu sonra ilgili programlama dilinin yorumlayıcısı olarak kullanılır.

Bayt Derlenen Kod

Bu örnekte, Python ile yazılmış `pelaba.py` isimli programı bayt koduna dönüştüreceğiz, bu bayt kodu Python sanal makinesi tarafından derlenecek. Python kodları istenirse ham yorumlanabilir ancak bayt derlenen sürümü daha hızlıdır. Dolayıcıyla, RPM paketçileri son kullanıcıya paketleri dağıtırken bayt derlenen sürümünü tercih eder.

`pello.py`

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

print("Merhaba Dünya")
```

Bayt derlenen programlar için prosedür dilden dile değişir. Bu yöntem dil üzerine kuruludur, dilin sanal makinesini gerektirir ve kullanılan araçlar/süreçler dile aittir.

NOTE Python çoğunlukla bayt olarak derlenir, ancak bu bahsettiğimiz şekilde değil. Bahsi geçen yöntem topluluk standarlarına uyumlu olmayı değil sade olmayı gerektirir. Gerçek dünyada kullanılan Python rehberleri için şu bağlantıya bakabilirsiniz: Software Packaging and Distribution (İngilizce).

Bayt derlenen `pelaba.py`:

```
$ python -m compileall pello.py

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

`pello.pyc` bayt kodunu çalıştırın:

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

Ham Hâlde Yorumlanan Kod

(Ç.N; Ham olarak yorumlanan kod (raw interpreted) hiçbir işleme tabii tutulmadan yazıldığı gibi yorumlanan kod demektir.)

Bu örneğimizde ise bash ile ham yorumlanan `belaba` programımızı inceleyeceğiz.

`belaba`

```
#!/bin/bash

printf "Merhaba Dünya\n"
```

Bash gibi kabuk betik dilleriyle yazılmış programlar ham yorumlanır. Hâliyle sadece dosyayı kaynak kodundan çalıştırılabilir yapmanız ve çalıştırmanız gerekmektedir:

```
$ chmod +x belaba
$ ./belaba
Merhaba Dünya
```

Yazılımı Yamalamak

Yama, başka bir kaynak kodunu güncelleyen bir kaynak kodudur. "*diff*" şeklinde formatlanmıştır zira bu format iki farklı versiyon arasındaki farkı gösterir. "*diff*" formatı, `diff` isimli bir araç kullanılarak oluşturulur ki daha sonra patch isimli bir araç kullanılarak kaynak koduna uygulanır.

Not: Yazılım geliştiricileri kendi kodlarını kontrol etmek için çoğunlukla git gibi Versiyon Kontrol Sistemlerini kullanır. Bu tür araçların *diff* yaratmak ve yamaları uygulamak için kendi yöntemleri vardır.

Aşağıdaki örnekte, orijinal kaynak kodundan `diff` kullanarak yeni bir yama oluşturuyoruz ve `patch` kullanarak uyguluyoruz. Yamalamadan, daha sonraları SPEC Dosyaları İle Çalışmak kısmında yararlanacağız.

Peki yamalamanın RPM paketlemekle ne alakası var? Paketlerken, orijinal kaynak kodunu olduğu değiştirmek yerine onu koruyup üzerine yama uygulamayı tercih ederiz.

`celaba.c` için bir yama hazırlayalım:

Esas kaynak kodunu koruyalım:

+

```
$ cp celaba.c celaba.c.orig
```

+ Bu, orijinal kaynak kodunu muhafaza etmek için yaygın bir yöntemdir.

+ . `celaba.c`'yi değiştirelim:

+

```
#include <stdio.h>

int main(void) {
    printf("Yeni yamamdan selam dünya!\n");
    return 0;
}
```

+ `diff` aracını kullanarak bir yama oluşturalım:

+ . Not: `diff` aracını kullanırken birden fazla argüman kullandık. Bu argümanlar hakkında bilgi almak için, `diff` ile ilgili belgeleri araştırınız.

+

```
$ diff -Naur celaba.c.orig celaba.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;
 }
```

+ . `-` ile başlayan satırlar, orijinal kaynak kodundan çıkarılmış kodlardır. `+` ile yer alan kodlar ise çıkarılan kodların yerine geçer.

+

1. Yamayı bir dosyaya kaydedelim:

   ```
   $ diff -Naur celaba.c.orig celaba.c > celaba-ciktisi-ilk-yama.patch
   ```

2. Orijinal `celaba.c` dosyasını geri getirelim:

   ```
   $ cp celaba.c.orig celaba.c
   ```

   Esas `celaba.c` dosyasını geri getirdik. Bunun nedeni, bir RPM paketi inşa edileceği zaman düzenlenmiş dosya yerine esas dosyanın kullanılmasıdır. Daha fazla bilgi için: SPEC Dosyaları İle Çalışmak

`celaba.c` dosyasını, `celaba-ciktisi-ilk-yama.patch` dosyasını kullanarak, yamalanmış dosyayı `patch` komutuna yönlendirin:

```
$ patch < celaba-ciktisi-ilk-yama.patch
patching file celaba.c
```

`celaba.c` dosyasının içeriği, gördüğünüz üzere yamayla değişti:

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

int main(void){
    printf("Yeni yamamdan selam dünya!\n");
    return 0;
}
```

Yamalanmış `celaba.c` dosyasını derleyip çalıştıralım:

```
$ make clean
rm celaba

$ make
gcc -g -o celaba celaba.c

$ ./celaba
Yeni yamamdan selam dünya!
```

Tebrikler! Bir yama oluşturdunuz, sonra programı yamaladınız, yamalı programı derlediniz ve çalıştırmış oldunuz!

İhtiyari Yapılar Kurma

Linux ve Unix benzeri işletim sistemlerinin büyük bir avantajı Dosya Sistemi Hiyerarşisi Standartlarıdır ifdef::rhel[Dosya Sistemi Hiyerarşisi Standartlarıdır]. Bu standartlar, hangi dizinde hangi dosyanın depolanacağını belirtir. RPM paketlerinden kurulan dosyalar ise Dosya Sistemi Hiyerarşisine uygun olmalıdır. Örneğin, çalıştırılabilir bir dosya PATH değişkeninde belirtilen bir dizin altında tutulmalıdır.

Bu belgenin bağlamında, bir *İhtiyari Yapı* RPM aracılığıyla sisteme kurulan herhangi bir şeydir. Bu RPM ve sistem için bir betik, paketin içerdiği kaynak kodundan derlenen ikili bir dosya, önceden derlenmiş ikili bir dosya veya başka bir dosya olabilir.

Burada, sisteme *İhtiyari Yapıları* yerleştirmenin iki popüler yolunu keşfedeceğiz: `install` veya `make install` kullanmak.

Install komutunu kullanmak

Program çok basit olduğunda ve fazladan ek yüke ihtiyaç duymadığında, GNU make gibi bir otomatik derleme aracı kullanmak pek mantıklı olmayabilir. Bu ve bunun gibi durumlarda paketçiler çoğunlukla coreutils tarafından sunulan `install` komutunu tercih ederler. Bu komut sözünü ettiğimiz yapıları dosya sisteminde belirli bir dizine, belirli izinlerle yerleştirir.

Aşağıdaki örnekte daha önce hazırladığımız `belaba` dosyasını ihtiyari yapı olarak sistemimize kuracağız. Yalnız dikkat etmeniz gereken bir şey, bu kurulum için sudo veya root yetkilerine sahip olmanız gerektiğidir.

Aşağıdaki örnekte, `belaba` dosyasını `/usr/bin` içerisine `install` komutuyla yerleştireceğiz, elbette ki çalıştırmak için gerekli izinlerle beraber:

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

`belaba` isimli dosyamız $PATH değişkeninde listelenmiş bir dizinde bulunmakta. Artık herhangi bir dizinde, `belaba` dosyasını bütün konumu belirtmeden çalıştırabilirsiniz.

```
$ cd ~

$ belaba
Merhaba Dünya!
```

Make Install komutunu kullanmak

Bir yazılımı otomatikleştirilmiş şekilde kurmak için `make install` komutunu kullanmak popüler bir yöntemdir. Bu yöntem, `Makefile` içerisinde ihtiyari yapıların sisteme nasıl kurulacağını belirtmenizi gerektirir.

Not: `Makefile` çoğunlukla paketçi tarafından değil geliştirici tarafından hazırlanır.

`Makefile` içerisine `install` kısmını ekleyin:

`Makefile`

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

clean:
	rm cello

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

$(DESTDIR) değişkeni GNU make içerisine yerleştirilmiş bir değişkendir ve çoğunlukla kurulum dizininin kök dizin dışında neresi olması gerektiğini belirtir.

Artık, `Makefile` dosyasını yalnızca dosyayı derlemek için değil, hedef sisteme kurmak için de kullanabilirsiniz.

`celaba.c`'yi derleyip kurmak için:

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

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

`celaba` programını `$PATH` değişkeni içerisinde tanımlanmış dizinlerden birisine eklemiş oldunuz. Artık, `celaba`'yı tam konumunu belirtmeden dilediğiniz gibi çalıştırabilirsiniz.

```
$ cd ~

$ celaba
Merhaba Dünya!
```

Sisteminize inşa edilmiş bir yapıyı, belirtilmiş bir konuma kurmuş bulunmaktasınız.

Kodu Paketlemek İçin Hazırlamak

Not: Bu bölümde hazırladığımız kodları burada bulabilirsiniz.

Geliştiriciler yazılımlarını çoğunlukla sıkıştırılmış arşivler içerisinde dağıtırlar ki bunlar paketleme için kullanılırlar. Bu bölümde sıkıştırılmış arşivler hazırlayacaksınız.

Not: Kaynak kodu arşivleme işi çoğunlukla RPM paketçisinin görevi değildir, geliştirici tarafından yapılır. Paketçi, hazır kaynak kodu arşivleriyle çalışır.

Yazılımlar yazılım lisansı ifdef::rhel[yazılım lisansı] aracılığıyla lisanslanmalıdır. Biz örnek olarak GPLv3 lisansını ele alacağız. Lisans metnini örnek programlarımızın `LICENCE` dosyasında sunacağız. Bir RPM paketçisi, paketlerken lisans dosyaları ile ilgilenmelidir.

Aşağıdaki örnekte bir lisans dosyası oluşturduk:

```
$ 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/>.
```

Tarball İçerisine Kaynak Kodu Eklemek

Ç.N: `Tarball`, kaynak kodlarını içinde bulunduran ve uzantısında "tar" bulunan (Örn: .tar.gz) arşiv dosyalarına verilen isim. Literatürde bu şekilde yer alır. Mot a mot çevirisi de "Tartopu" gibi anlamsız bir şeye tekabül ettiği için`tarball` kelimesini kabul ediyorum. Eğer `tarball` yerine geçecek iyi bir fikriniz varsa muhakkak bana bildirin.

Aşağıdaki örneklerde bulunan üç adet `Merhaba Dünya` programının her birini gzip ile sıkıştırılmış tarballlara ekliyoruz. Yazılımlar çoğunlukla paketlenmeden önce bu şekilde yayınlanırlar.

belaba

belaba yazılımı, bash ile `Merhaba Dünya` yazmamızı sağlıyor. Bu yazılım kendi içerisinde yalnızca `belaba` kabuk betiğini içeriyor ve oluşturacağımız `.tar.gz` arşivinde `LICENCE` dışında yalnızca bu betik var. Bu yazılımın sürüm numarasını `0.1` olarak düşünebiliriz.

belaba yazılımını dağıtıma hazırlayalım:

1. Dosyaları bir dizine yerleştirin:

```
$ mkdir /tmp/belaba-0.1

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

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

2. Arşiv dosyasını oluşturun ve `~/rpmbuild/SOURCES/` altına taşıyın:

```
$ cd /tmp/

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

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

pelaba

pelaba yazılımı, Python ile `Merhaba Dünya` yazmamızı sağlıyor. Bu yazılım kendi içerisinde yalnızca `pelaba` betiğini içeriyor ve oluşturacağımız `.tar.gz` arşivinde `LICENCE` dışında yalnızca bu betik var. Bu yazılımın sürüm numarasını `0.1.1` olarak düşünebiliriz.

pelaba yazılımını dağıtıma hazırlayalım:

1. Dosyaları bir dizine yerleştirin:

```
$ mkdir /tmp/pello-0.1.1

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

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

2. Arşiv dosyasını oluşturun ve `~/rpmbuild/SOURCES/` altına taşıyın:

```
$ cd /tmp/

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

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

celaba

celaba yazılımı, `C` ile `Merhaba Dünya` yazmamızı sağlıyor. Bu yazılım kendi içerisinde yalnızca `celaba.c`'yi ve `Makefile` dosyasını içeriyor ve oluşturacağımız `.tar.gz` arşivinde `LICENCE` dışında yalnızca iki dosya olacak. Bu yazılımın sürüm numarasını `1.0` olarak düşünebiliriz.

`patch` dosyasını arşiv ile beraber dağıtıma çıkartmadığımıza dikkat edin. RPM Paketleyicisi yamayı RPM derlenirken uygular. Yama, `.tar.gz` dosyası ile beraber `~/rpmbuild/SOURCES/` dizinine yerleştirilecek.

celaba yazılımını dağıtıma hazırlayalım:

1. Dosyaları bir dizine yerleştirin:

```
$ 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. Arşiv dosyasını oluşturun ve `~/rpmbuild/SOURCES/` altına taşıyın:

```
$ 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. Yamayı uygulayın

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

Ve kaynak kodlarınız RPM'e paketlenmek üzere hazır!

Yazılımı Paketlemek

Bu kaynakça, Red Hat bağlantılı dağıtımlar için paketlemeyi öncelikle ele alır. Bu dağıtımlar aşağıda listelenmiştir:

- Fedora
- CentOS
- Red Hat Enterprise Linux (RHEL)

Bu dağıtımlar, RPM paketleme usulünü kullanır.

Yine de, yazıda çoğunlukla bahsi geçen sistemler hedef alınsa da bu rehber çoğu RPM temelli dağıtıma uygulanabilir. Ancak dağıtımların kendisine özgü rehberleri, makroları ve önhazırlık maddeleri incelenmelidir.

Bu rehber, okuyucunun Linux veya başka bir işletim sistemiyle ilgili herhangi bir altyapısı olmadığını varsayar.

NOTE	Eğer yazılım paketleme ve Linux dağıtımları hakkında hiçbir fikriniz yoksa öncelikle Linux ve Paket Yöneticileri hakkında bir ön araştırma yapmayı düşünebilirsiniz.

RPM Paketleri

Bu bölüm, RPM hakkında basit bilgileri içerir. Eğer daha ileri düzey bilgiler görmek isterseniz [gelismis-konular] başlığını inceleyebilirsiniz.

RPM Nedir?

Bir RPM paketi, basitçe sistem tarafından gereken bilgileri ve dosyaları içeren bir dosyadır. Detaya inersek, bir RPM paketi cpio ismi verilen bir arşiv türüdür. `RPM` paket yöneticisi üstveri aracılığıyla bağımlılıkları saptar, nereye kurulması gerektiğini anlar ve diğer bilgileri ele alır.

İki tip RPM paketleri vardır:

- Kaynak RPM (SRPM, Source RPM'in kısaltmasıdır)
- İkili RPM

Kaynak ve ikili RPMler aynı dosya biçimini ve araçlarını kullanırlar ancak farklı içeriklere ve amaçlara sahiptirler. SRPM, kaynak kodunu ve isteğe bağlı olarak yamalarını ve SPEC dosyasını içerir ki SPEC, kaynak kodunun nasıl ikili RPM'e çevirilmesi gerektiğini içerir. İkili RPM ise kaynak kodunun ve yamalarının derlenmiş hâlidir.

RPM Paketleme Araçları

Ön Koşullar'da belirtilmiş `rpmdevtools` paketi pek RPM paketlemek için pek çok aracı içerir. Araçları listemek için şu komutu çalıştırın:

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

Araçlar hakkında bilgi edinmek için kullanım talimatlarını veya yardım yazılarını inceleyebilirsiniz.

RPM Paketlemesi İçin Çalışma Alanı

RPM için çalışma alanı oluşturacak bir dizin hazırlamak istiyorsanız `rpmdev-setuptree` aracını kullanın.

```
$ rpmdev-setuptree

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

5 directories, 0 files
```

Oluşturulan dizinler şu amaçlara hizmet eder:

Dizin	Amaç
BUILD	Bir paket inşa edildiği zaman gereken `%buildroot` dizinleri burada oluşturulur. Bu dizinleri incelemek, hata kayıtlarının yetersiz kaldığı zamanlar nerede hata çıktığını anlamaya yardımcı olur.
RPMS	İkili RPMler mimarilere göre ayrılmış altdizinlere göre burada oluşturulur. Bunlar `x86_64` veya `noarch` olabilir.
SOURCES	Burada paketçinin sıkıştırılmış kaynak kodu arşivleri ve yamaları bulunur. `rpmbuild` komutu burayı arar.
SPECS	Paketçi SPEC dosyalarını buraya yerleştirir.
SRPMS	Eğer `rpmbuild` paketi ikili RPM üretmek yerine SRPM inşa etmek için kullanılırsa, SRPMler burada oluşturulur.

Spec Dosyası Nedir?

Eğer RPM paketlerini yemeğe benzetirsek SPEC dosyasını, `rpmbuild` aracını RPM pişirmek için baz alınan yemek tarifi olarak düşünebiliriz. SPEC dosyası, inşa ortamına neler yapması gerektiğini çeşitli parçalar hâlinde bildirir. Bu parçalar, *Önsöz* (*Preamble*) ve *Gövde* (*Body*) olarak ikiye ayrılabilir. *Gövde* kısmı, talimatların esas parçasını içerir.

Önsöz Maddeleri

Aşağıdaki tablo RPM Spec dosyasının Önsöz kısmında kullanılan maddeleri listeler.

SPEC Yönergeleri	Açıklama
`Name`	Paketin esas adı, SPEC dosyasının adıyla eşleşmelidir.
`Version`	Yazılımın sürüm sayısını içerir.
`Release`	Bu sürümün kaç defa yayınlandığını içerir. Normalde ilk yayının değeri 1%{?dist} olarak belirlenir ve her yeni yayında sayı arttırılır. Yeni `Version` (sürüm) yayınlandığında tekrar 1 olur.
`Summary`	Paketin kısa ve tek satırlık özetini içerir.
`License`	Paketlenen yazılımın lisansını belirtir. Fedora gibi topluluk dağıtımlarında dağıtılan paketler, dağıtımın lisanslama rehberlerinde bahsedildiği şekilde özgür yazılım lisansına uygun olmak zorundadır.
`URL`	Dağıtım hakkında daha fazla bilginin alınabileceği tam adrestir. Çoğu zaman bu adres aynı zamanda yazılımın ana sayfasıdır.
`Source0`	Yazılımın kaynak kodunu barındıran internet adresi veyahut dosya konumudur. Her an erişilebilecek ve güvenilir bir yere atıfta bulunması tavsiye edilir. Güvenilirden kasıt, paketçinin yerel depolaması yerine yazılımın internet sayfasıdır. Eğer gerekliyse daha fazla SourceX yönergesi eklenebilir. (X yerine sayı gelir ve sayılar birer birer arttırılır. Örnek: Source1, Source2, Source3 gibi)
`Patch0`	Eğer gerekliyse kaynak koduna uygulanacak olan ilk yama burada belirtilir. İhtiyaç duyulması hâlinde, SourceX gibi fazladan PatchX eklenebilir. (Patch1, Patch2, Patch3 gibi)
`BuildArch`	Yazılım bir mimariye yönelik değilse, mesela tamamen yorumlanan bir dille yazılmışsa, bu yönerge `BuildArch: noarch` olarak ayarlanır. Eğer bu yönerge belirtilmezse, makinenin üzerinde çalıştığı mimariye ayarlanır. Örnek: `x86_64`
`BuildRequires`	Derlenen dilde yazılmış programı paketlemek için gereken programların virgülle veya boşluklarla ayrıldığı listelerdir. `BuildRequires` yönergesi, aynı satırda kalındığı müddetçe, birden fazla yazılım belirtebilir.
`Requires`	Yazılımların kurulması için gereken bağımlılıkların virgülle veya boşluklarlarla belirtildiği bir listedir. Tıpkı `BuildRequires` gibi, `Requires` yönergesi de satırına sadık kalınarak birden fazla yazılımı belirtebilir.
`ExcludeArch`	Eğer yazılım belirli bir işlemci mimarisinde çalışmayacaksa, o mimariyi burada dışlayabilirsiniz.

`Name`, `Version` ve `Release` yönergeleri RPM paketinin dosya ismini oluşturur. RPM Paket geliştiricileri ve sistem yöneticileri bu üç direktiften **N-V-R** ya da **NVR** olarak bahseder. Çünkü RPM paketleri `NAME-VERSION-RELEASE` biçimiyle isimlendirilir.

`NAME-VERSION-RELEASE` biçiminin örneklerini, `rpm` komutu aracılığıyla paketleri sorgulayarak bulabilirsiniz:

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

`python`, paket ismini gösteren `Name` direktifine, `2.7.5`, `Version` direktifine, `34.el7` ise `Release` direktifine atıfta bulunur. Son işaretçi olan `x86_64` ise mimariyi gösterir. **NVR** üçlüsünün aksine, mimari paketçinin insiyatifinde değildir fakat `rpmbuild` tarafından belirlenmiştir. Bu duruma bir istisna, paketin mimariden bağımsız olduğunu belirten `noarch` ibaresidir.

Gövde Maddeleri

Aşağıdaki tablo, RPM SPEC dosyasının *Gövde* kısmındaki maddeleri listeler:

SPEC Yönergesi	Açıklama
`%description`	RPM'e paketlenen yazılım için detaylı açıklamayı içerir. Bu açıklama birden fazla satırı, hatta paragrafı içerebilir.
`%prep`	Yazılımı kuruluma hazırlayan komut ya da komutları içerir. `Source0` içerisinde bulunan arşiv dosyasını çıkartmak buna bir örnek olabilir. Bu yönerge, bir kabuk betiğini de içerebilir.
`%build`	Yazılımı makine koduna veyahut bayt koduna derlemek için kullanılan komut veya komutları içerir.
`%install`	İstenen inşa yapılarını `%builddir` (Derleme konumunu içerir) içerisinden `%buildroot`a taşıyan (paketlenecek dizin yapısını barındıran dizin) komut veya komutları içerir. Bu işlem çoğu zaman dosyaları `~/rpmbuild/BUILD` konumundan `~/rpmbuild/BUILDROOT`'a taşımak anlamına gelir. Bu komutlar son kullanıcı paketi kurarken değil, yalnızca paket inşa edilirken çalıştırılır. Daha fazla bilgi için SPEC Dosyaları İle Çalışmak kısmını inceleyin.
`%check`	Yazılımı denetlemek için kullanılan komut ya da komutları içerir. Bu kısım çoğu zaman birim testlerinden oluşur.
`%files`	Bu kısımda listelenen dosyalar paketi kuran son kullanıcının sistemine yerleştirilir.
`%changelog`	Paketin `Version` ve `Release` numaraları değiştiğinde,neyin değiştiğini bildirmek için kullanılır.

İleri Düzey Maddeler

SPEC dosyaları ileri düzey maddeleri de içerebilir. Örneğin, bir SPEC dosyası *betikçi* ve *tetikleyici*leri içerebilir. Bu yöngergeler, son kullanıcı kendi sistemine kurarken farklı noktalarda tesir eder. (Paket inşa sürecini etkilemez)

Daha fazla bilgi için Betikçiler ve Tetikleyiciler kısmını okuyabilirsiniz.

BuildRoot

Ç.N: BuildRoot, Build (İnşa/Derleme) ve Root (Kök) kelimelerinden ortaya çıkan, mot a mot çevirisi `DerlemeKökü` olan bir kelimedir. Daha önce karşılaştığımız `Tarball` ve şimdi denk geleceğimiz `chroot` gibi, literatür içerisinde yer edindiğinden çevirmek istemedim. Bu yüzden, yazı boyunca `BuildRoot` kelimesini kullanmaya devam edeceğim. Eğer bu kelimeye karşılık gelen iyi bir karşılık bulursanız bana bildirebilirsiniz.

RPM paketleme bağlamında, "buildroot" bir chroot ortamıdır. Bu ortam, dosya sistemi içerisindeki paketi kuracak tarafın sisteminin kök dizinini temsil eden derleme yapıları dizinine yerleştirir. Yerleştirme düzeni son kullanıcının *Dosya Sistemi Hiyerarşisi* standartlarına uygun olmalıdır.

"Buildroot" içerisine eklenen dosyalar cpio arşivine dönüştürülür ki bu RPM'in temel parçasıdır. RPM paketi, son kullanıcının sistemine kurulduğu zaman dizindeki dosyalar hiyerarşiye uygun olarak kök dizin içerisine çıkartılır.

NOTE

Geçmişte, `%buildroot` makrosunun `%~/rpmmacros` içerisine tanımlanması veya SPEC dosyasında `BuildRoot` belirtilmesi tavsiye edilirdi. RedHat Enterprise Linux 6'dan itibaren `rpmbuild` yazılımı kendi varsayılanlarını benimsedi. Bu varsayılanları yeniden tanımlamak belli başlı sıkıntılar doğurabileceğinden, RedHat bu makroyu değiştirmenizi tavsiye etmez. `%{buildroot}` makrosunu `rpmbuild` dizinindeki varsayılanıyla kullanabilirsiniz.

RPM Makroları

Bir rpm makrosu, belirli gömülü işlevleri yerine getirmek için kullanılan ve koşula göre değişen değerleri tutan net ifadesidir. Bu demek oluyor ki, bilmek zorunda olmadığınız şeyleri RPM'e havale edebilirsiniz.

SPEC dosyasını hazırlarken yazılım sürümünü tekrar tekrar yazmak istemediğinizde makrolar size yardımcı olur. Daha önce tanımladığınız *Version* yönergesini, sonraları `%{version}` makrosunu kullanarak çağırabilirsiniz. Her `%{version}` makrosu, otomatik olarak *Version* yönergesiyle yer değiştirecektir.

NOTE

Eğer makronun ne iş yaptığını anlayamazsanız şu şekilde ne iş yaptığını bulabilirsiniz:

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

Örneğin:

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

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

%{?dist} makrosu derleme esasında hangi dağıtımın kullanıldığına işaret eder ve yaygın olarak kullanılır.

Örnek kullanım:

```
# RHEL 7.x üzerinde
$ rpm --eval %{?dist}
.el7

# Fedora 23 üzerinde
$ rpm --eval %{?dist}
.fc23
```

Makrolar hakkında daha fazla bilgi almak için: Makrolar Hakkında Daha Fazlası.

SPEC Dosyaları İle Çalışmak

Yazılımları paketlemenin büyük kısmı SPEC dosyalarını düzenlemekten oluşur. Bu bölümde, SPEC dosyaları oluşturmak ve düzenlemek üzerine konuşacağız.

Yeni bir yazılımı paketlemek için yeni bir SPEC dosyası oluşturmanız gerekir. Bütün dosyayı en temelden oluşturmak yerine, `rpmdev-newspec` aracını kullanabilirsiniz. Bu araç size doldurulmamış bir SPEC dosyası hazırlar ve ihtiyacınıza göre yönergeleri ve alanları kullanabilirsiniz.

Bu bölümde, daha önce Yazılımı Paketlemeye Hazırlamak kısmında gördüğümüz üç farklı Merhaba Dünya programını ele alacağız.

- belaba-0.1.tar.gz
- pelaba-0.1.1.tar.gz
- celaba-1.0.tar.gz
    - celaba-ciktisi-ilk-yama.patch

Hepsini `~/rpmbuild/SOURCES` içine yerleştirin.

Üç dosya için de bir SPEC dosyası hazırlayın:

NOTE	Bazı yazılımcı odaklı metin düzenleyicileri `.spec` dosyasını önceden hazırlayabilir. `rpmdev-newspec` aracı ise düzenleyiciden bağımsız bir yol sunar ki bu rehberde kullanmamızın nedeni budur.

```
$ cd ~/rpmbuild/SPECS

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

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

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

`~/rpmbuild/SPECS/` dizini artık üç farklı SPEC dosyasını içeriyor. Bunlar, `belaba.spec`, `celaba.spec` ve `pelaba.spec`'tir.

Dosyaları gözden geçirebilirsiniz. Spec Dosyası Nedir? kısmında gördüğünüz yönergeleri bu dosyalarda da göreceksiniz. Sonraki bölümlerde, bu SPEC dosyalarını kendiniz dolduracaksınız.

NOTE

`rpmdev-newspec` aracı herhangi bir dağıtıma özgü standartları ve eğilimleri göz önünde bulundurmaz. Fakat bu belge Fedora, CentOS ve RHEL'i hedeflediğinden bazı detayları fark edebilirsiniz:

- *CentOS* (7.0'dan önceki sürümler) veyahut Fedora (18'den önceki sürümler) üzerinde çalışırken `rm $RPM_BUILD_ROOT` komutuna denk gelebilirsiniz. Biz, diğer makrolarla tutarlılık sağlamak açısından `%{buildroot}` kullanmayı `$RPM_BUILD_ROOT` kullanmaya tercih ederiz.

Paketlenecek olan üç yazılımın özetleri aşağıda mevcuttur. Her biri detaylıca tarif edilmiştir. Paketleme için isterseniz sadece ihtiyacınıza özel hitap edenlere bakabilirsiniz veya farklı paketleme yöntemleri keşfetmek için hepsini birden okuyabilirsiniz.

Yazılım Adı	Açıklama
belaba	Ham olarak yorumlanan bir programlama dilinde yazılmış bir yazılım. Kaynak kodunun derlenmeye ihtiyacı olmadığı, yalnızca kurulmayı ihtiyaç olduğu bir durumda ne yapılması gerektiğine örnektir. Eğer önceden derlenmiş ikili bir paketin kurulması gerekiyorsa, ikili dosyalar yalnızca bir dosya olduğu için bu kısma göz atabilirsiniz.
pelaba	Bayt derlenen ve yorumlanan bir programlama dilinde yazılmış bir yazılım. Bayt derlenen kaynak kodunun nasıl derlenmesi ve kurulması gerektiğine örnektir.
celaba	Yerel olarak derlenen bir programlama dilinde yazılmış bir yazılım. Kaynak kodundan makine koduna programın nasıl derlenmesi gerektiğine ve çıktı olarak gelen çalıştırılabilir dosyaların nasıl kurulması gerektiğine örnektir.

belaba

İlk SPEC dosyamız, Yazılımı Paketlemeye Hazırlamak kısmında daha önce karşılaştığımız, bash kabuk betiği ile yazılmış olan belaba.

Şunları yaptığınızdan emin olun:

1. `belaba` kaynak kodunu `~/rpmbuild/SOURCES/` kısmına yerleştirdiğinize dikkat edin. Bilgi için: SPEC Dosyaları İle Çalışmak.
2. Doldurulmamış SPEC dosyasını `~/rpmbuild/SPECS/belaba.spec` konumunda olduğuna emin olun. Bu dosyanın içerisinde olması gerekenler:

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

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Şimdi, `belaba` RPMlerini oluşturmak için `~/rpmbuild/SPECS/belaba.spec` dosyasını düzenleyelim:

1. `Name`, `Version`, `Release`, ve `Summary` yöngergelerini doldurun:
   - `Name` yönergesi hâlihazırda `rpmdev-newspec` tarafından dolduruldu.
   - `Version` yönergesi yazılımının sürümünü belirtmelidir. Sevimli kaynak kodumuz `belaba` için bu, `0.1`.
   - `Release`, otomatik olarak ilk değer olan `1{%dist}` olarak ayarlandı. Eğer yazılımın sürümünde bir değişiklik olmadan yeni bir güncelleme olursa, mesela yeni bir yama, bu sayı tek tek

arttırılmalıdır. Yeni sürümlerde `Release` rakamı tekrar `1` olmalıdır. Örneğin eğer belaba'nın `0.2` sürümü yayınlanırsa, `Release` tekrardan ``1{%dist} olarak ayarlanmalıdır. *disttag* makrosu ise RPM Makroları kısmında detaylıca anlatılmıştır.

- `Summary`, yazılımın ne olduğu anlatan tek satırlık, kısa bir açıklamadır.

Düzenlemelerinizden sonra, SPEC dosyasının ilk kısmı şuna benzemelidir:

```
Name:           belaba
Version:        0.1
Release:        1%{?dist}
Summary:        Bash ile Yazılmış Bir Merhaba Dünya örneği
```

`License`, `URL` ve `Source0` yönergelerini doldurun:

- `License` kısmı, yazılımın kaynak kodundaki Software License ile ilişkili olmalıdır.

`License` bölümü için şu formatı takip ediniz: Fedora License Guidelines

Biz örnek olması için `GPLv3+` kullanacağız.

- `URL` bölümü, yazılımın yayınlandığı internet adresini gösterir. Örneğin: `https:://example.com/belaba`. Yine de tutarlı olması açısından, %{name} makrosunu tercih edin ve `https://example.com/%{name}` şeklinde kullanın.
- `Source0` yöngergesi, kaynak kodunun yayınlandığı internet adresini içerir. Doğrudan paketlenecek yazılımın indirilmek için yayınlandığı adresi içermelidir. Bu örnekte, temsilen `https://example.com/belaba/releases/belaba-0.1.tar.gz` adresini kullanacağız. Elbette ki %{name} makrosunu da kullanacağız. Ayrıca, %{version} makrosunu da sürüm değişikliklerine uyum sağlamak için kullacağız. Sonuç olarak girdimiz şu şekli alacaktır: `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz`

Değişikliklerinizden sonra, SPEC dosyasının ilk kısmı aşağıdaki gibi görünmelidir:

```
Name:           belaba
Version:        0.1
Release:        1%{?dist}
Summary:        Bash ile Yazılmış Bir Merhaba Dünya örneği

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

2. `BuildRequires` ve `Requires` yönergelerini doldurun ve `BuildArch` yönergesini ekleyin:
   - `BuildRequires`, paket için derleme zamanı bağımlılıklarını belirtir. `belaba`'nın derlenmesi için hiçbir adıma gerek yoktur, çünkü Bash ham hâlde yorumlanan bir programlama dilidir ve yalnızca dosyaların sisteme kurulması yeterlidir. Yapılacak tek şey, bu yönergeyi silmektir.
   - `Requires` ise paketlenmiş programın çalışması için gereken bağımlılıkları belirtir. `belaba` betiği için gereken tek bağımlılık, çalıştırılması için gereken `bash` kabuk ortamıdır. Bu

yüzden, bu yönergeye `bash` yazarak bunu belirteceğiz.

- Program yorumlanan bir dilde yazıldığından dolayı mimari açıdan herhangi bir bağlantı gerektirmeyecektir. `BuildArch` direktifini ekleyip buna `noarch` değeri vereceğiz. Bu değer, RPM paketinin herhangi bir işlemci mimarisi üzerinde çalışabileceğini belirtecektir.

  Değişikliklerinizden sonra, SPEC dosyasının ilk kısmı şu şekilde görünmelidir:

```
Name:           belaba
Version:        0.1
Release:        1%{?dist}
Summary:        Bash ile Yazılmış Bir Merhaba Dünya örneği

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

Requires:       bash

BuildArch:      noarch
```

3. `%description`, `%prep`, `%build`, `%install`, `%files`, ve `%license` kısımlarını doldurun. Bu yönergeler "Konu başlıkları" olarak da düşünülebilir, zira bu yönergeler çoklu satır hâlinde belirlenebilir, çalıştırılması gereken komutları belirleyebilir ve çoklu talimatlar bildirebilirler.
   - `%description` kısmı dosyanın uzun bir anlatımını içerir. `Summary` yönergesinden farkı, bir veya birden çok paragrafı barındırmasıdır. Bu örneğimizde kısa bir açıklamayla geçiştireceğiz.
   - `%prep` kısmı, derleme için uygun ortamı belirtir. Bu, arşivlenmiş kodun dışarı açılması, yamaların uygulanması ve SPEC dosyasının ileri safhalarında kullanmak üzere kaynak kodundaki belirli bir bilgiyi almak için taramak olabilir. Bu bölümde yalnızca `%setup -q` gömülü makrosunu kullanacağız.
   - `%build` kısmı ise, paketlediğimiz yazılımın nasıl derlenmesi gerektiğini belirtir. `bash` diliyle yazılan dosyaların derlenmeye ihtiyacı olmadığı için, basitçe bu kısmı silip boş bırakacağız.
   - `%install` kısmı ise `rpmbuild`'in dosyaları nasıl kurması gerektiğini bildiren yönergeleri içerir. Bu dizinde inşa edilen dosyalar `BUILDROOT` dizinine eklenir. Bu dizin, chroot temel dizinidir ve paketi kuran tarafın kök dizinini temsil eder. Burada, kurduğumuz dosyaların dizinlerini oluşturmamız gerekir.

     `belaba` için yalnızca hedef dizini kurmamız ve `bash` betiğini yerleştirmemiz gerektiği için `install` komutunu kullanacağız. RPM makroları bu işi kolay yoldan yapmamız için yardımcı olacaktır. .

     `%install` kısmı düzenlemeleri bitirdikten sonra şöyle görünecektir:

```
%install

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

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

- paketinizin `%files` kısmı, son kullanıcının sistemine yerleştireceği dosyaların tam konumunu belirtir. Yalnızca `belaba` dosyasını kuracağımız için, konumumuz `/usr/bin/belaba`'dır ki bunu RPM Makroları ile ``%{_bindir}/%{name}` olarak kısaltabiliriz.
- Aynı zamanda, gömülü makrolar aracılığıyla dosyaların ne gibi görevleri olduğunu belirtebilirsiniz. Örneğin LICENSE dosyasının yazılım lisansı olduğunu belirtmek için %license makrosunu kullanabilirsiniz. Bu,`rpm` komutunu aracılığıyla üstveriyi belirten paket dosyalarını sorgularken kolaylık sağlar. +o Değişikliklerinizden sonra, %files kısmı şu şekilde görünecektir:

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

4. Son kısım olan `%changelog`, her Sürüm-Yayın değişikliği için tarif damgalı girdileri listeler. Bu günlük, paketleme değişikliklerini içerir, yazılım değişikliğini değil. Paketleme değişiklikleri için örnekler: yama ekleme, inşa sürecini değiştirme vs.

Ç.N: `%changelog` içerisindeki değişiklikleri ve tarih damgasını İngilizce yazmanız paketi inceleyecek yabancı kullanıcılar için kolaylık sağlayacaktır.

+ İlk satır için şu biçimi takip ediniz:

+ `* HaftanınGünü Ay Gün Yıl İsim Soyisim <eposta> - Sürüm-Yayın`

+ Değişiklikleri bildirirken şu kuralları uygulayın:

+

- Her satır girdi birden fazla öğe içerebilir - Her madde için bir tane
- Her madde için yeni bir satıra geçilmelidir.
- Her madde `-` ile başlamalıdır.

+ Tarih damgalanmış örnek bir girdi:

+

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- İlk belaba paketi
- 0.1-1 sürüm - yayını için öylesine bir ikinci madde
```

Sonunda **belaba** için bütün SPEC dosyasını yazmış bulunmaktasınız. **belaba** için yazmış olduğunuz dosya buna benzemelidir:

```
Name:           belaba
Version:        0.1
Release:        1%{?dist}
Summary:        Bash ile Yazılmış Bir Merhaba Dünya örneği

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

Requires:       bash

BuildArch:      noarch

%description
Bash ile yazılmış Merhaba Dünya örneği için
satırlara sığmayan
epey uzun bir
tanıtım yazısı

%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 belaba package
```

Sonraki kısım, RPM'nin nasıl derlenmesi gerektiği hakkında bilgi verir.

pelaba

Hazırlayacağımız ikinci SPEC dosyası, Python ile hazırlanmış olan örnek programımız için. İndirdiğimiz (ya da Yazılımı Paketlemeye Hazırlamak kısmında hazırladığımız) dosyayı `~/rpmbuild/SOURCES/` dosyasına yerleştiriyoruz ve `~/rpmbuild/SPECS/pelaba.spec` dosyasını oluşturup

düzenliyoruz.

Bu düzenlemeye girişmeden önce, bayt derlenerek yorumlanan dillere dair önemli bir detaydan bahsetmemiz gerekiyor. Yazılımı bayt-derlediğimiz için mevzu (shebang) olarak isimlendirilen yapıyı kullanamıyoruz. Bu yapı, bayt derlenmeyen kabuk betikleri ve Python gibi dillerin ufak çaplı kodları için yaygın olarak tercih edilen bir yoldur. Bizim tek satırlık kodumuz için bayt derleme işi anlamsız görünebilir, fakat yüzlerce koddan oluşan büyük çaptaki yazılım projeleri için performans açısından oldukça faydalıdır.

NOTE

Bayt derlenmiş kodu çağıran betiği hazırlamak veyahut yazılıma bayt derlenmemiş koda giriş noktası oluşturmak, yazılımcıların paketi yayına çıkartmadan önce sıklıkla yaptıkları iştir. Ancak her zaman böyle bir giriş noktası hazırlanmamış olabilir ve yapacağımız alıştırma bunun gibi durumlarda ne yapmanız gerektiğini gösterir. Python kodunun normalde nasıl yayınlandığını ve dağıtıldığını öğrenmek istiyorsanız lütfen Yazılım Paketleme ve Dağıtma belgelerini inceleyiniz.

Bayt derlenmiş yazılıma bir giriş noktası hazırlamak için ufak bir kabuk betiği hazırlayacağız ve SPEC dosyasının içine dâhil ekleyeceğiz. Bu, aynı zamanda SPEC dosyası içinde nasıl betik kabuğu kodlarını çalıştırdığımıza dair bir örnek olmuş olacak. Daha sonra bunun nasıl yapılacağına dair detayları `%install` kısmında inceleyeceğiz.

Şimdi işi biraz daha ilerletelim ve `~/rpmbuild/SPECS/pelaba.spec` dosyasını incelemek üzere açalım.

Aşağıdaki örnek dosya `rpmdev-newspec` komutunun bize verdiği şablondur.

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

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Tıpkı ilk örnekte olduğu gibi, belgenin en tepesinde yer alan `Name`, `Version`, `Release`, `Summary` yönergeleriyle düzenlemeye başlayalım. `rpmdev-newspec` komutu gerekli bilgiyi önceden sağladığı için `Name` yönergesi önceden belirtilmiş oldu.

İlk iş, `Version` yönergesini *pelaba*'nın sürüm numarasına eşleştirmek olsun. Bu yazılım numarası, indirdiğimiz kod için (veya Yazılımı Paketlemeye Hazırlamak kısmında gördüğümüz üzere) `0.1.1`'dir.

`Release` hâlihazırda bizim için `1%{?dist}` olarak ayarlandı. Ayarlanmış değerin içindeki `1` sayısı paket her düzenlendiğinde bir arttırılmak zorundadır, ki bu düzenlemeler bir sorunun düzeltilmesi için yama eklemek olabilir. Fakat yeni bir `Version` yayınlandığında bu sayı tekrar `1`'e geri çekilmelidir. RPM Makroları bölümünü okuyanlar `%{?dist}` ile gösterilen *disttag* makrosunu hemen tanımış olmalılar.

`Summary`, yazılımın ne olduğunu açıklayan kısa, tek satırlık bir yönergedir.

SPEC dosyasının ilk kısmında yaptığımız değişikliklerden sonra bu kısım aşağıdakine benzemelidir:

```
Name:           pelaba
Version:        0.1.1
Release:        1%{?dist}
Summary:        Python ile Yazılmış Bir Merhaba Dünya örneği
```

Şimdi, `rpmdev-newspec` komutunun `License`, `URL`, `Source0` şeklinde gruplandığı ikinci kısma bakalım.

`License` kısmı, yazılımcının kaynak kodu için belirttiği yazılım lisansıdır . SPEC dosyasında kullanılan olan lisans etiketleri RPM tabanlı https://tr.wikipedia.org/wiki/Linux dağıtımları için farklılık gösterir. Biz, Fedora Lisanslama Rehberi için geçerli olan yazım standartlarını kullanacağız. Programımız için örnek olarak seçtiğimiz tarz `GPLv3+`'dır.

`URL` yönergesi kaynak kodun indirme linki değil, yazılımın anasayfasına giden bağlantıdır. İçeriğinde, yayındaki yazılım hakkında daha fazla bilgi sahibi olmak isteyenler için ürünün, projenin veyahut şirketin anasayfasına giden bağlantı olmalıdır. Şimdilik örnek olması açısından `https://example.com/pelaba` adresini seçiyoruz. Ancak, SPEC dosyasının diğer parçaları arasında tutarlılık olması için adresteki yazılım ismini `%{name}` RPM makrosu ile değiştireceğiz.

`Source0` kaynak kodu yazılım kodunun indirilebileceği adresi işaret eder. Bağlantı, paketlenecek olan kaynak kodun ilgili sürümüne yönlendirmelidir. Aynı şekilde, bu yalnızca bir örnek olduğu için aşağıdaki linki kullanacağız: `https://example.com/pelaba/releases/pelaba-0.1.1.tar.gz`

Tamamen sabit bir şekilde belirtilmiş bir bağlantı ileride yayınlanacak olan sürümler için sıkıntı yaratacaktır, bu yüzden adresteki `0.1.1` kısmı muhakkak değişecektir. SPEC dosyası yalnızca tek bir sürüm için hazırlanmaz, yeni sürümlerde de olabilecek en az değişiklikle tekrar kullanılır. Bu yüzden, kodu hazırlarken adres sabit olarak değil `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz` biçimiyle yazılır.

Değişikliklerin ardından, SPEC dosyanızın üst tarafı şuna benzemelidir:

```
Name:           pelaba
Version:        0.1.1
Release:        1%{?dist}
Summary:        Python ile Yazılmış Bir Merhaba Dünya örneği

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

Ardından, `BuildRequires` ve `Requires` isimli iki yönergeyi ele alacağız, bu iki yönerge de paketin bağımlılıklarını belirler. Fakat, `BuildRequires` paketin **inşa edilmesi** için gereken bağımlılıkları gösterirken `Requires` paketin düzgünce **çalışması** için gerekli olan bağımlılıklara işaret eder.

Örneğimizde, hem kaynak kodunu bayt olarak derlemek hem de bayt-derlenmiş kodu çalıştırmak için `python` paketine ihtiyacımız var. Bunu belirtmek için `Requires` direktifini kullanacağız. Aynı şekilde, paketimize hazırlayacağımız ufak bir giriş betiği çalıştırmak için `bash` paketine de ihtiyacımız var.

Eklememiz gereken bir diğer şey ise, yorumlanan bir programlama dili üzerinde çalıştığımız için `BuildArch` direktifini `noarch` olarak belirtmemiz gerekiyor, böylece RPM bu paketin bir işlemci mimarisine bağımlı olmadığını anlamış olacak.

Düzenlemlerin ardından, SPEC dosyanızın üst kısmı şuna benzemelidir:

```
Name:           pelaba
Version:        0.1.1
Release:        1%{?dist}
Summary:        Python ile Yazılmış Bir Merhaba Dünya örneği

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
```

Bundan sonraki yönergeler, "gövde başlıklar" olarak düşünülebilir. Çünkü bu yönergeler birden çok satırı, talimatı ve betiklendirilmiş görevleri kapsamakta. Bu yönergeleri de daha önce yaptıklarımızla aynı şekilde düzenleyeceğiz.

`%description`, `Summary` yönergesine kıyasla paketlenecek yazılım hakkında çok daha uzun bir açıklamayı içerir. Örneğimizde çok uzun bir tanıtım yazısı yazmayacağız, ancak bu kısım dilenirse tüm bir paragrafı ya da paragrafları kapsayabilir.

`%prep`, "Hazırlamak" kelimesinin İngilize karşılığı olan *prepare* kelimesinin kısaltmasıdır. Yazılımı derlemek için gerekli olan ortamı veya çalışma alanını inşa etmek için kullanılır. Çoğu zaman burada olan şey; arşivlerin açılması, yamaların uygulanması ve kaynak kodda gerekebilecek bilginin taranmasıdır ki bu bilgiler SPEC dosyasının sonraki bölümlerinde kullanılacaktır. Bu bölümde kısa yoldan işimizi göreceğinden `%setup -q` makrosunu kullanacağız.

`%build`, paketlerin derlenmesi için ne yapılması gerektiğini içerir. Örneğimizde kaynak kodunun nasıl derlenmesi gerektiğine dair komutları uygulayacağız. Yazılımı Paketlemeye Hazırlamak kısmını okumuş okuyucular bu kodu tanıyacaktır.

SPEC dosyamızdaki `%build` kısmı aşağıdakine benzemelidir:

```
%build

python -m compileall pelaba.py
```

`%install` kısmı ise `rpmbuild`'in dosyaları nasıl kurması gerektiğini bildiren yönergeleri içerir. Bu dizinde inşa edilen dosyalar `BUILDROOT` dizinine eklenir. Bu dizin, chroot temel dizinidir ve paketi kuran tarafın kök dizinini temsil eder. Burada, kurduğumuz dosyaların dizinlerini oluşturmamız

gerekir. Ancak, RPM Makroları burada yapılan işleri sabit bir şekilde kodlamadan yapmamıza yardımcı olur.

Daha önce, dosyanın mevzu isimli kısmını kaybettiğimizi söylemiştik, bundan dolayı bayt derlenmiş kodumuz için aynı işi yapan bir betik hazırlamamız gerekiyor. Bunu yapmak için birçok seçeneğimizden biri de bir betik hazırlayıp ayrı bir `SourceX` yönergesiyle bunu belirtmek ve bu örnekte tercih edeceğimiz şekilde SPEC dosyası içerisinde bir dosya oluşturmaktır. Bu örneği tercih etmemizin sebebi SPEC dosyasının da betik yazılabilir bir dosya olduğunu göstermektir. Yapacağımız şey, Python bayt-derlenmiş kodunu "here" belgesi ile çalıştırmak. Aynı zamanda bayt-derlenmiş dosyayı bir sistemin erişebileceği bir kütüphane dizinine kurmamız gerekmektedir.

NOTE	Fark ettiğiniz üzere burada kütüphane adresini elle belirtiyoruz. Bu durumdan kaçınmanın pek çok yolu var ve bunların önemli bir kısmı paketlenen yazılımın yazıldığı programlama diline uygun olarak [gelismis-konular] altındaki Makrolar Hakkında Daha Fazlası kısmında bahsediliyor. Bu örnekte kafa karışıklığı yaratmamak için kütüphane adresini elle yazmayı tercih ediyoruz.

`%install` kısmı düzenlemelerin ardından şu şekilde görünmelidir:

```
%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` kısmı, RPM paketinin içerdiği dosyaları ve paketin kurulacağı sisteme yerleştirilmesi planlanan dosyaları belirttiğimiz kısımdır.

Dikkat etmeniz gereken bir husus, bu kısmın `%{buildroot}`'a göre yazılmaması gerektiğidir. Bu kısımdaki dosyalar, paketi kuran kullanıcının sisteminde kurulumdan sonra belirlenecek tam konuma göre belirtilmelidir. Bu yüzden, `pelaba` dosyasının sistemdeki konumunu belirtirken `%{_bindir}/pelaba` olarak belirtmemiz gerekmektedir. Aynı zamanda, dosyaları içine yerleştirdiğimiz bir kütüphane dizinin bu paket tarafından "sahiplenildiğini" belirtmek üzere bir `%dir` tanımlamamız gerekmektedir.

Ayrıca bu kısımda, bir dosyanın içeriğinde ne olduğunu belirtmek üzere bazı gömülü makroları kullanmaya ihtiyacınız olacaktır. Bu, `rpm` komutunun çıktısını sorgulamak isteyen sistem yöneticileri ve son kullanıcılar için oldukça kullanışlı olabilir. Burada kullanacağımız `%license` gömülü makrosu, `rpmbuild`'e dosyanın yazılım lisansını içerdiğini bildirecektir.

`%files` kısmı, düzenlemenizin ardından şu şekilde görünmelidir:

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

Son kısım olan `%changelog`, tarih damgalı girdilerin Sürüm-Yayın değişikliklerinde ne olduğunu günlüklemek içindir. Bu günlükte her değişikliğin yazılması gerekmez ancak her önemli paketleme değişikliklerinin belirtilmesi gereklidir. Örneğin, bir paket içerisindeki yazılımın paketlemeye ihtiyacı varsa veya `%build` kısmında gösterilen derleme yönteminin değiştirilmesi gerekiyorsa burada bilgi verilebilir. Her girdi birden fazla maddeyi içerebilir, ve her madde `-` karakteriyle başlayan yeni bir satırla başlamalıdır. Aşağıda örnek bir girdi görmektesiniz:

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1.1-1
- İlk belaba paketi
- 0.1.1-1 sürüm - yayın için öylesine bir ikinci madde
```

Yukarıdaki örnekte dikkat etmelisiniz ki, tarih damgası `*` karakteriyle başlamalıdır ve günün adının ardından ay, gün, yıl şeklinde tarih atılmalı ve RPM paketçisi hakkındaki iletişim bilgileri formatıyla hazırlanmalıdır. Daha sonra, alışılageldiği üzre Sürüm (Version) - Yayın'dan (Release) önce `-` karakterini yerleştirdik ancak bu yaklaşım zorunlu değildir.

Ve hepsi bu kadar! **Pelaba** için bütün SPEC dosyasını yazmış bulunmaktayız! Bundan sonraki kısımda bir RPM dosyası nasıl inşa edilir, bunu okuyacaksınız!

Bütün SPEC dosyası değişikliklerden sonra aşağıdaki gibi görünmelidir:

```
Name:           pelaba
Version:        0.1.1
Release:        1%{?dist}
Summary:        Python ile Yazılmış Bir Merhaba Dünya örneği

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
Python ile yazılmış Merhaba Dünya örneği için
satırlara sığmayan
```

```
epey uzun bir
tanıtım yazısı

%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 pelaba package
```

celaba

Üçüncü SPEC dosyamız ise, C programlama dilinde yazılmış olan örneğimiz için. Daha önceden bu örnek programı hazırlamış (ya da indirmiş) ve kaynak kodunu `~/rpmbuild/SOURCES/` içerisine yerleştirmiş olmalısınız.

Şimdi biraz daha ileri gidelim ve `~/rpmbuild/SPECS/celaba.spec` dosyasını açıp boşluklarını doldurmaya başlayalım.

`rpmdev-newspec` komutu ile bu dosyayı oluşturabilirsiniz.

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

License:
URL:
Source0:

BuildRequires:
Requires:

%description

%prep
%setup -q

%build
%configure
make %{?_smp_mflags}

%install
rm -rf $RPM_BUILD_ROOT
%make_install

%files
%doc

%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org>
-
```

Tıpkı daha önceki örneklerimizde yaptığımız gibi, `rpmdev-newspec` komutunun dosyanın üst tarafına topladığı yönergelerlerle başlayacağız, bunlar: `Name`, `Version`, `Release`, `Summary`'dir. `Name` isimli yönerge, `rpmdev-newspec` tarafından hâlihazırda belirtildiği için düzenlememize gerek yok.

`Version` isimli yönerge *celaba*'nın kaynak kodu sürümüyle eşleşmelidir ki indirdiğimiz (ya da Yazılımı Paketlemeye Hazırlamak kısmında belirlediğimiz) üzere bu `1.0`'dır.

`Release` hâlihazırda bizim için `1%{?dist}` olarak ayarlandı. Ayarlanmış değerin içindeki `1` sayısı paket her düzenlendiğinde bir arttırılmak zorundadır, ki bu düzenlemeler bir sorunun düzeltilmesi için yama eklemek olabilir. Fakat yeni bir `Version` yayınlandığında bu sayı tekrar `1`'e geri çekilmelidir. RPM Makroları bölümünü okuyanlar `%{?dist}` ile gösterilen *disttag* makrosunu hemen tanımış olmalıdır.

`Summary` ise yazılımın ne olduğuna göre tek satırlık, kısa bir açıklamadır.

Düzenlemelerinizin ardından, SPEC dosyasının ilk bölümü şuna benzemelidir.

```
Name:           celaba
Version:        1.0
Release:        1%{?dist}
Summary:        C ile Yazılmış Bir Merhaba Dünya örneği
```

Şimdi, `rpmdev-newspec` komutunun SPEC dosyamızda grupladığı direktiflerin ikinci setine göz atalım: `License`, `URL` ve `Source0`. Ancak, bu direktiflerin içine `Source0` ile yakından alakalı, bizim hazırladığımız yamayı yazılıma dâhil edecek`Patch0` direktifini de ekleyeceğiz.

`License` kısmı, yazılımcının kaynak kodu için belirttiği Yazılım Lisansıdır. SPEC dosyasında kullanılan lisans etiketleri RPM tabanlı Linux dağıtımları için farklılık gösterir. Biz, Fedora Lisanslama Rehberi için geçerli olan yazım standartlarını kullanacağız ve bu programımız için örnek olarak seçtiğimiz tarz `GPLv3+`'dır.

`URL` yönergesi, yazılımın anasayfasına giden bağlantıyı içerir. İçeriğinde, yayındaki yazılım hakkında daha fazla bilgi sahibi olmak isteyenler için ürünün, projenin veyahut şirketin anasayfasına giden bağlantı olmalıdır. Şimdilik örnek olması açısından `https://example.com/celaba` adresini seçiyoruz. Ancak, SPEC dosyasının diğer parçaları arasında tutarlılık olması için adresteki yazılım ismini `%{name}` RPM makrosu ile değiştireceğiz.

`Source0` yöngergesi, kaynak kodunun yayınlandığı internet adresini içerir. Doğrudan paketlenecek yazılımın indirilmek için yayınlandığı adresi içermelidir. Bu örnekte, temsilen `https://example.com/belaba/releases/celaba-1.0.tar.gz` adresini kullanacağız. Elbette ki `%{name}` makrosunu ve buna ek olarak sürüm değişikliklerine uyum sağlamak için `%{version}` makrosunu da kullacağız. Sonuç olarak girdimiz şu şekli alacaktır: `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz`

Tamamen sabit bir şekilde belirtilmiş bir bağlantı ileride yayınlanacak olan sürümler sıkıntı yaratacaktır ve adresteki `1.0` kısmı muhakkak değişecektir. Bir SPEC dosyası yalnızca tek bir sürüm için hazırlanmaz, yeni sürümlerde de olabilecek en az değişiklikle tekrar kullanılır. Bu yüzden, kodu hazırlarken adres sabit olarak değil `https://example.com/%{name}/releases/%{name}-%{version}.tar.gz` biçimiyle yazılır.

Bir sonraki işimiz ise daha önceden hazırladığımız `.patch` dosyasını paketimize `%prep` kısmında kullanılmak üzere dahil etmek. Bunun için, `Patch0: celaba-ciktisi-ilk-yama.patch` satırını SPEC dosyamıza ekleyeceğiz.

Bütün değişikliklerden sonra, SPEC dosyanızın üst kısmı aşağıdakine benzemelidir:

```
Name:           celaba
Version:        1.0
Release:        1%{?dist}
Summary:        C ile Yazılmış Bir Merhaba Dünya örneği

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

Patch0:         celaba-ciktisi-ilk-yama.patch
```

Bundan sonraki yönergelerimiz, `BuildRequires` ve `Requires` yönergeleridir ki bu satırlar gerekli paketleri sıralar. Fakat `BuildRequires`, `rpmbuild`'e paketin **inşa edilmesi** için ne gerektiğini bildirirken `Requires` paketin **çalışması** için hangi paketlere ihtiyaç olduğunu bildirir.

Bu örnekte kaynak kodunu derlemek üzere `gcc` ve `make` paketine ihtiyacımız olacak. Yazılımın çalışması için gereken bağımlılıklar ise `rpmbuild` tarafından sağlanmakta, zira programımız standart `C` kütüphaneleri dışında hiçbir şeyi gerektirmemektedir. Bundan dolayı herhangi bir şeyi `Requires` yönergesi içerisinde belirtmemize gerek yok, bu yönergeyi SPEC içerisinden çıkartabiliriz.

Düzenlemenizin ardından, SPEC dosyanızın üst tarafı şuna benzemelidir:

```
Name:           celaba
Version:        1.0
Release:        1%{?dist}
Summary:        C ile Yazılmış Bir Merhaba Dünya örneği

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

Patch0:         celaba-ciktisi-ilk-yama.patch

BuildRequires:  gcc
BuildRequires:  make
```

Bundan sonraki yönergeler, "gövde başlıklar" olarak düşünülebilir. Çünkü bu yönergeler birden çok satırı, talimatı ve betiklendirilmiş görevleri kapsamakta. Bu yönergeleri de daha önceki yönergelerimizdekilerle aynı şekilde düzenleyeceğiz.

`%description`, `Summary` yönergesine kıyasla paketlenecek yazılım hakkında çok daha uzun bir açıklamayı içerir. Örneğimizde çok uzun bir tanıtım yazısı yazmayacağız, ancak bu kısım dilenirse tüm bir paragrafı ya da paragrafları kapsayabilir.

`%prep`, "Hazırlamak" kelimesinin İngilize karşılığı olan *prepare* kelimesinin kısaltmasıdır. Yazılımı derlemek için gerekli olan ortamı veya çalışma alanını inşa etmek için kullanılır. Çoğu zaman burada olan şey; arşivlerin açılması, yamaların uygulanması ve kaynak kodda gerekebilecek bilginin taranmasıdır ki bu bilgiler SPEC dosyasının sonraki bölümlerinde kullanılacaktır. Bu bölümde kısa yoldan işimizi göreceğinden `%setup -q` makrosunu kullanacağız.

`%build` ise paketlediğimiz yazılımın nasıl derlenmesi/inşa edilmesi gerektiğini bildiren kısımdır. `C` ile yazdığımız programımız için basit bir `Makefile` yazdığımızdan dolayı,`rpmdev-newspec` komutunun bizim için hazırladığı GNU make komutunu kolayca kullanabiliriz. Ancak, bir https://tr.wikipedia.org/wiki/Configure["configure" (yapılandırma) betiği hazırlamadığımız için `%configure` yönergesini kaldırmamız gerekmektedir.

Kodumuzun `%build` kısmı aşağıdakine benzemelidir:

```
%build
make %{?_smp_mflags}
```

`%install` kısmı ise `rpmbuild`'in dosyaları nasıl kurması gerektiğini bildiren yönergeleri içerir. Bu dizinde inşa edilen dosyalar `BUILDROOT` dizinine eklenir. Bu dizin, bir chroot temel dizinidir ve paketi kuran tarafın kök dizinini temsil eder. Yapmamız gereken şey kurduğumuz dosyaların dizinlerini oluşturmaktır. Buradaki RPM Makroları, bu işi gerçekleştirirken değişmez kalıp kodlardan kaçınmamıza yardımcı olur.

Tekrar bahsetmek gerekirse, elimizdeki `Makefile` dosyasındaki kurulum talimatları kolaylıkla `rpmdev-newspec` tarafından sağlanmış olan `%make_install` makrosu aracılığıyla kurulabilir.

Bütün değişikliklerden sonra `%install` yönergesi aşağıdaki şekilde görünmelidir:

```
%install
%make_install
```

`%files` kısmı, RPM paketinin içerdiği dosyaları ve paketin kurulacağı sisteme yerleştirilmesi planlanan dosyaları belirttiğimiz kısımdır.

Dikkat etmeniz gereken bir husus, bu kısmın `%{buildroot}`'a göre yazılmaması gerektiğidir çünkü buradaki dosyalar, paketi kuran kullanıcının sisteminde kurulumdan sonra belirlenecek tam konuma göre belirtilmelidir. Bu yüzden, `celaba` dosyasının sistemdeki konumunu belirtirken `%{_bindir}/celaba` olarak belirtmemiz gerekmektedir.

Ayrıca bu kısımda, bir dosyanın içeriğinde ne olduğunu belirtmek üzere bazı gömülü makroları kullanmaya ihtiyacınız olacaktır. Bu, `rpm` komutunun çıktısını sorgulamak isteyen sistem yöneticileri ve son kullanıcılar için oldukça kullanışlı olabilir. Burada kullanacağımız `%license` gömülü makrosu, `rpmbuild`'e bu dosyanın yazılım lisansını içerdiğini bildirecektir.

`%files` kısmı, son değişikliklerden sonra şu şekilde görünmelidir:

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

Son kısım olan `%changelog`'da, tarih damgalı girdiler Sürüm-Yayın değişikliklerinde ne olduğunun kaydını tutmak içindir. Bu günlükte her değişikliğin yazılması gerekmez ancak her önemli paketleme değişikliklerinin belirtilmesi gereklidir. Örneğin, bir paket içerisindeki yazılımın paketlemeye ihtiyacı varsa veya `%build` kısmında gösterilen derleme yönteminin değiştirilmesi gerekiyorsa burada bilgi verilebilir. Her bir girdi birden fazla maddeyi içerebilir, ve her madde `-` karakteriyle başlayan yeni bir satırla başlamalıdır. Aşağıda örnek bir girdi görmektesiniz:

```
%changelog
* Tue May 31 2016 Adam Miller <maxamillion@fedoraproject.org> - 0.1-1
- İlk celaba paketi
- 1.0-1 sürüm - yayın için öylesine bir ikinci madde
```

Yukarıdaki örnekte dikkat etmelisiniz ki, tarih damgası * karakteriyle başlamalıdır ve günün adının ardından ay, gün,yıl şeklinde tarih atılmalı ve RPM paketçisi hakkındaki iletişim bilgileri formatıyla hazırlanmalıdır. Ardından, Sürüm (Version) - Yayın'dan (Release) önce - karakterini yerleştirdik, bu alışılagelen bir yaklaşımdır ancak zorunlu değildir.

Ve hepsi bu kadar! **celaba** için bütün SPEC dosyasını yazmış bulunmaktayız. Bundan sonraki kısımda bir RPM dosyası nasıl inşa edilir, bunu okuyacaksınız.

Bütün SPEC dosyası aşağıdaki gibi olmalıdır:

```
Name:           celaba
Version:        1.0
Release:        1%{?dist}
Summary:        C ile Yazılmış Bir Merhaba Dünya örneği

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

Patch0:         celaba-ciktisi-ilk-yama.patch

BuildRequires:  gcc
BuildRequires:  make

%description
C ile yazılmış Merhaba Dünya örneği için
satırlara sığmayan
epey uzun bir
tanıtım yazısı

%prep
%setup -q

%patch0

%build
make %{?_smp_mflags}

%install
%make_install


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


%changelog
* Tue May 31 2016 Adam Miller <maxamillion@gmail.com> - 1.0-1
- İlk celaba paketi
```

`rpmdevtools` paketi aynı zamanda popüler programlama dilleri için pek çok SPEC dosyası örneği içerir.

RPMleri İnşa Etmek

RPMler `rpmbuild` komutu aracılığıyla inşa edilirler. Farklı senaryolara ve istenen sonuçlara göre `rpmbuild` komutu farklı parametreler gerektirir. Bu kısımda, iki ana senaryoyu ele alacağız:

1. kaynak RPM inşa etmek
2. ikili RPM inşa etmek

`rpmbuild` komutu belirli bir dizin ve dosya yapısını gerektirir ki bu yapı `rpmdev-setuptree` aracıyla hazırlanmış yapının birebir aynısıdır. Aynı şekilde daha önceki talimatlar da gerekli olan bu yapıya uygundur.

Kaynak RPMler

Neden kaynak RPM (SRPM) inşa etmeliyiz?

1. Yayını yapılmış bir RPM'in İsim-Sürüm-Yayın yapısına özgü gerçek kaynağı korumak için. Kaynak RPMler SPEC dosyasını, kaynak kodunu ve ilişkili yamaları barındırır. Bu tür paketler geri dönüp ne olduğunu incelemek ve hata ayıklamak için kullanışlıdır.
2. Çeşitli donanım platformları veya işlemci mimarileri için RPM inşa edebilmek için

Kaynak RPM inşa etmek:

```
$ rpmbuild -bs _SPECDOSYASİ_
```

SPECDOSYASİ ile SPEC dosyasının konumunu değiştirin. `-bs` ise kaynak kodu için kullanılan bir parametredir ve "build source" kelimelerinin kısaltmasından oluşur.

Şimdi `belaba`,`pelaba` ve `celaba` için kaynak RPM inşa edeceğiz:

```
$ cd ~/rpmbuild/SPECS/

$ rpmbuild -bs belaba.spec
Wrote: /home/admiller/rpmbuild/SRPMS/belaba-0.1-1.el7.src.rpm

$ rpmbuild -bs pelaba.spec
Wrote: /home/admiller/rpmbuild/SRPMS/pelaba-0.1.1-1.el7.src.rpm

$ rpmbuild -bs celaba.spec
Wrote: /home/admiller/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
```

İnşa edilecek olan kaynak RPMlerinin (İngilizce: Source RPM, SRPM) `rpmbuild/SRPM` dizinine yerleştirileceğine dikkat ediniz ki bu dizin `rpmbuild` komutunun gerektirdiği dizinlerden birisidir.

Kaynak RPM derlemek için gerekli olan her şey budur.

İkili RPMler

İkili RPM inşa etmek için iki yöntem vardır:

1. Kaynak RPM'i `rpmbuild --rebuild` komutuyla yeniden derlemek.
2. `rpmbuild -bb` komutunu kullanarak SPEC dosyasından inşa etmek. `-bb` parametresi, Türkçesi

"ikili inşa" olan "build binary" kelimelerinin kısaltmasıdır.

Kaynak RPM'i Yeniden İnşa Etmek

`belaba`, `pelaba`, ve `celaba`'yı yeniden inşa etmek için:

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/belaba-0.1-1.el7.src.rpm
[çıktı törpülendi]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/pelaba-0.1.1-1.el7.src.rpm
[çıktı törpülendi]

$ rpmbuild --rebuild ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
[çıktı törpülendi]
```

Ve RPM paketlerini inşa etmiş bulunmaktasınız. Şimdi, birkaç detay:

- İkili RPM'in inşasında üretilen çıktı gayet ayrıntılıdır ki bu hata ayıklamak için kullanışlıdır. Çıktı, SPEC dosyasına ve farklı örneklere göre farklılık gösterir.
- İkili RPMler `~/rpmbuild/RPMS/MIMARIADI` isimli bir dosyaya yerleştirilir. `MIMARIADI`, işlemci mimarisine tekabül eder. Eğer paket bir mimariye özgü değilse `~/rpmbuild/RPMS/noarch` dizini içerisindedir.
- `rpmbuild --rebuild` komutunu çalıştırınca adım adım şu olaylar gerçekleşir:

  1. `~/rpmbuild` dizini içerisine SRPM'in içerdiği SPEC dosyasını ve kaynak kodlarını yerleştirir.
  2. `~/rpmbuild` içerisindekiler derlenir.
  3. SPEC dosyası da kaynak kodu da temizlenir.

  Eğer SPEC dosyasını ve kaynak kodunu derlendikten sonra da saklamak isterseniz, iki seçeneğiniz var:

  - İnşa sırasında, `--rebuild` yerine `--recompile` parametresini kullanın.
  - Kaynak RPMleri şu komutlarla kurun:

  ```
  $ rpm -Uvh ~/rpmbuild/SRPMS/belaba-0.1-1.el7.src.rpm
  Updating / installing...
     1:belaba-0.1-1.el7                 ################################# [100%]

  $ rpm -Uvh ~/rpmbuild/SRPMS/pelaba-0.1.1-1.el7.src.rpm
  Updating / installing...
     1:pelaba-0.1.1-1.el7               ################################# [100%]

  $ rpm -Uvh ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
  Updating / installing...
     1:celaba-1.0-1.el7                 ################################# [100%]
  ```

SPEC dosyası ve kaynak kodlarıyla etkileşimi sürdürmek için, gördüğünüz gibi, `rpm -Uvh` komutunu kullanmanız gerekmektedir.

İkili Paketi SPEC dosyasından inşa etmek

`belaba`, `pelaba` ve `celaba`yı SPEC dosyalarından inşa edebilmek için, şu komutları çalıştırın:

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

$ rpmbuild -bb ~/rpmbuild/SPECS/pelaba.spec

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

Hepsi bu kadar! SPEC dosyalarından RPMleri inşa etmiş bulunmaktasınız.

Kaynak RPM'i Yeniden İnşa Etmek kısmındaki bilgilerin çoğunluğu burada da geçerlidir.

RPMlerin Geçerliliğini Denetleme

Bir paketi oluşturduktan sonra paket kalitesini test etmek hiç de fena bir fikir değildir. Paketin kalitesinden kasıt, paketin sunduğu yazılım değil paketin kendi kalitesidir. Bu denetimi yapmak üzere kullanılan ana araç rpmlint'dır. Bu araç sayesinde RPM'nin bakımı kolaylaşır, paketin geçerlilik denetimi ve statik hata analizi iyileşir.

`rpmlint`'in oldukça sıkı ilkelere sahip olduğa dikkat ediniz. Sonraki örneklerde göreceğiniz gibi, kimi zaman hata ve uyarı mesajlarını es geçmek sıkıntı yaratmaz.

NOTE	Göreceğiniz örneklerde `rpmlint` bize gayet sade bir çıktı sunmakta. Eğer hataların ve uyarıların detaylı bir çıktısını istiyorsanız, bu komut yerine `rpmlint -i` komutunu kullanabilirsiniz.

Belaba SPEC dosyasını Denetlemek

`belaba` için `rpmlint` çıktısı şöyledir:

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

Gözlemlenenler:

- `belaba.spec` için elimizde tek bir uyarı var ki bu da `Source0` yönergesindeki bağlantıya ulaşamadığından bahsediyor. Bu anlaşılmayacak bir şey değil, çünkü `example.com` adresinde belirttiğimiz dosya yok. Yine de belki gelecekte bağlantının geçerli olabileceğini varsayarak, bu uyarıyı görmezden gelebiliriz.

Bu da belaba kaynak RPM paketi için rpmlint çıktısı:

```
$ rpmlint ~/rpmbuild/SRPMS/belaba-0.1-1.el7.src.rpm
belaba.src: W: invalid-url URL: https://www.example.com/belaba HTTP Error 404: Not
Found
belaba.src: W: invalid-url Source0: https://www.example.com/belaba/releases/belaba-
0.1.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

Gözlemlenenler:

- Belaba kaynak RPM paketi için URL yönergesindeki bağlantının ulaşılamaz olmasıyla ilgili fazladan bir uyarımız daha var. Gelecekte bu bağlantının geçerli olacağını varsayarak, bu uyarıyı da gözardı edebiliriz.

Belaba'nın İkili RPM'ini Denetleme

İkili RPMleri denetlerken, rpmlint şu detayları da incelemekte:

1. Belgelendirme
2. Man (elkitabı) sayfaları
3. Dosya Sistemi Hiyerarşisi Standartı'nın doğru kullanımı

belaba ikili dosyası için rpmlint çıktısı şu şekildedir:

```
$ rpmlint ~/rpmbuild/RPMS/noarch/belaba-0.1-1.el7.noarch.rpm
belaba.noarch: W: invalid-url URL: https://www.example.com/belaba HTTP Error 404: Not
Found
belaba.noarch: W: no-documentation
belaba.noarch: W: no-manual-page-for-binary belaba
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

Gözlemlenenler:

- no-documentation (Tr. belgelendirme yok) ve no-manual-page-for-binary (Tr. ikili (paket) için man sayfası yok) uyarıları RPM paketimizin belgeleri olmadığını gösteriyor, çünkü böyle bir şey sağlamadık.

Uyarıları görmezden gelirsek, RPM paketimiz rpmlint denetiminden geçmiş sayılır.

Pelaba SPEC Dosyasını Denetlemek

Pelaba'nın SPEC dosyası için rpmlint çıktısı şöyledir:

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

Gözlemlenenler:

- `invalid-url Source0` (Tr. Source0 için geçerli adres) uyarısı Source0'da belirtilen adresin geçersiz olduğuna işaret etmekte. Bu beklenen bir durum, çünkü `example.com` adresindeki dosya yok. Gelecekte bu adresin çalışacağını varsayarak bu uyarıyı görmezden gelebiliriz.
- Pek çok hata var, çünkü kasıtlı olarak SPEC dosyasını gerekenden daha basit yazdık ve `rpmlint` bunu raporluyor.
- `hardcoded-library-path` (Tr. elle belirtilmiş konum) hatası, elle kütüphane konumu yerine `%{_libdir}` kullanmamızı öneriyor. Şimdilik bu örnek için bu hatayı görmezden gelebiliriz, ancak gelecekte hazırlayacağınız paketlerde bu hatayı görmezden gelmek için iyi bir nedene ihtiyacınız var.

`Pelaba`'nın kaynak RPM'inin `rpmlint` çıktısı şu şekildedir:

```
$ rpmlint ~/rpmbuild/SRPMS/pelaba-0.1.1-1.el7.src.rpm
pelaba.src: W: invalid-url URL: https://www.example.com/pelaba HTTP Error 404: Not
Found
pelaba.src:30: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}
pelaba.src:34: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.pyc
pelaba.src:39: E: hardcoded-library-path in %{buildroot}/usr/lib/%{name}/
pelaba.src:43: E: hardcoded-library-path in /usr/lib/%{name}/
pelaba.src:45: E: hardcoded-library-path in /usr/lib/%{name}/%{name}.py*
pelaba.src: W: invalid-url Source0: https://www.example.com/pelaba/releases/pelaba-
0.1.1.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 5 errors, 2 warnings.
```

Gözlemlenenler:

- Şimdi gördüğümüz `invalid-url URL` (Tr. geçersiz adres) hatası henüz ulaşılamayan `URL` yönergesiyle ilgilidir. Gelecekte bu adresin kullanılabilir olacağını düşünerek bu hatayı görmezden gelebiliriz.

İkili Pelaba RPM'ini Denetlemek

İkili RPMleri denetlerken, `rpmlint` şu detayları da incelemektedir:

1. Belgelendirme

2. Man (elkitabı) sayfaları
3. Dosya Sistemi Hiyerarşisi Standartı'nın doğru kullanımı

`Pelaba`'nın ikili RPM'inin `rpmlint` çıktısı şu şekildedir:

```
$ rpmlint ~/rpmbuild/RPMS/noarch/pelaba-0.1.1-1.el7.noarch.rpm
pelaba.noarch: W: invalid-url URL: https://www.example.com/pelaba HTTP Error 404: Not
Found
pelaba.noarch: W: only-non-binary-in-usr-lib
pelaba.noarch: W: no-documentation
pelaba.noarch: E: non-executable-script /usr/lib/pelaba/pelaba.py 0644L /usr/bin/env
pelaba.noarch: W: no-manual-page-for-binary pelaba
1 packages and 0 specfiles checked; 1 errors, 4 warnings.
```

Gözlemlenenler

- `no-documentation` (Tr. belgelendirme yok) ve `no-manual-page-for-binary` (Tr. ikili (paket) için man sayfası yok) uyarıları RPM paketimizin belgeleri olmadığını gösteriyor, çünkü böyle bir şey sağlamadık.
- `only-non-binary-in-usr-lib` uyarısı, `/usr/lib` dizini içerisinde ikili olmayan yapıları eklediğimizi belirtiyor. Bu dizin, ikili dosya olması gereken paylaşımlı nesne dosyaları içindir. Bundan dolayı, `rpmlint`, `/usr/lib` içerisine ekleyeceğimiz bir veya birden fazla dosyanın ikili olmasını bekler.

  Bu örnek, `rpmlint`in nasıl Dosya hiyerarşisi standartını koruduğunu gösterir. .

  Genellikle RPM makroları aracılığıyla dosyaları doğru konumlarına yerleştiririz. Yalnızca bu örnek için, bu uyarıyı görmezden geliyoruz.

- `non-executable-script` (Tr. Çalıştırılamayan betik) hatası `/usr/lib/pelaba/pelaba.py` dosyasının çalıştırma yetkileri olmadığını belirtiyor. Bu dosya, bir mevzu içerdiğinden dolayı `rpmlint` dosyanın çalıştırılabilir olmasını bekliyor. Şimdilik, bu dosyaya çalıştırma yetkileri vermiyoruz ve hatayı görmezden geliyoruz. , `rpmlint` expects

Yukarıdaki hatalar ve uyarıları görmezsek, RPM'imiz `rpmlint` denetimini geçti sayılır.

Celaba'nın SPEC Dosyasını Denetleme

`Celaba`'nın SPEC dosyası için `rpmlint` çıktısı şöyledir:

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

Gözlemlenenler:

- `celaba.spec` için elimizde tek bir uyarı var ki bu da `Source0` yönergesindeki bağlantıya

ulaşamadığından bahsediyor. Bu anlaşılmayacak bir şey değil, çünkü `example.com` adresinde belirttiğimiz dosya yok. Yine de belki gelecekte bağlantının geçerli olabileceğini varsayarak, bu uyarıyı görmezden gelebiliriz.

`Celaba`'nın kaynak RPM'inin `rpmlint` çıktısı şu şekildedir:

```
$ rpmlint ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
celaba.src: W: invalid-url URL: https://www.example.com/celaba HTTP Error 404: Not
Found
celaba.src: W: invalid-url Source0: https://www.example.com/celaba/releases/celaba-
1.0.tar.gz HTTP Error 404: Not Found
1 packages and 0 specfiles checked; 0 errors, 2 warnings.
```

Gözlemlenenler:

- Şimdi gördüğümüz `invalid-url URL` (Tr. geçersiz adres) hatası henüz ulaşılamayan `URL` yönergesiyle ilgilidir. Gelecekte bu adresin kullanılabilir olacağını düşünerek bu hatayı görmezden gelebiliriz.

Celaba İkili RPM'inin

İkili RPMleri denetlerken, `rpmlint` şu detayları da incelemektedir:

1. Belgelendirme
2. Man (elkitabı) sayfaları
3. Dosya Sistemi Hiyerarşisi Standartı'nın doğru kullanımı .

`Celaba`'nın ikili RPM'inin `rpmlint` çıktısı şu şekildedir:

```
$ rpmlint ~/rpmbuild/RPMS/x86_64/celaba-1.0-1.el7.x86_64.rpm
celaba.x86_64: W: invalid-url URL: https://www.example.com/celaba HTTP Error 404: Not
Found
celaba.x86_64: W: no-documentation
celaba.x86_64: W: no-manual-page-for-binary celaba
1 packages and 0 specfiles checked; 0 errors, 3 warnings.
```

Gözlemlenenler

- `no-documentation` (Tr. belgelendirme yok) ve `no-manual-page-for-binary` (Tr. ikili (paket) için man sayfası yok) uyarıları RPM paketimizin belgelerinin olmadığını gösteriyor, çünkü böyle bir şey sağlamadık.

Uyarıları görmezden gelirsek, RPM paketimiz `rpmlint` denetiminden geçmiş sayılır.

Ve RPMlerimiz hazır olduğu gibi `rpmlint` testinden de geçti. Bu kısım yalnızca nasıl RPM paketleneceğini anlatan bir rehberdi, daha fazla bilgi için İleri Düzey Konular kısmına göz atabilirsiniz.

İleri Düzey Konular

Bu bölüm, giriş seviyesi rehberlerden daha ileri konuları ele alır ancak bu konular RPM paketlemenin gerçek hayattaki kullanımı için yararlı bilgiler içerir.

Paketleri İmzalamak

Paketleri imzalamak, son kullanıcı için paketi güvenilir kılmanın bir yoludur. Kurulumdan önce paketi indirdiğiniz zaman HTTPS protokolü uygulamasıyla güvenli iletim sağlanabilir. Fakat kimi zaman paketler önceden indirilip daha sonra kullanılmadan önce yerel depolarda saklanırlar. Daha önceden üçüncü parti tarafından içeriğinin değiştirilmediğine emin olmak için bu paketler imzalanır.

Paketleri imzalamanın üç yolu vardır:

- Hazır bir pakete imza eklemek
- Hazır bir paketin imzasını değiştirmek
- İnşa zamanı paketi imzalamak

Pakete İmza Eklemek

Çoğu zaman paketler imzalanmadan inşa edilirler. Paketler yayınlanmadan önce imza eklenir.

Paketlere imza ekmek için, `--addsign` parametresini kullanın. Paketin birden fazla imzaya sahip olması, paketi inşa edenden son kullanıcıya kadar giden yolda kimden kime geçtiğinin kaydedilmesini mümkün kılar.

Mesela, bir şirketin bir bölümünün bir paketin oluşturduğunu ve bu bölümün anahtarıyla imzalandığını düşünün. Şirketin üst düzey yöneticileri bu paketi inceleyip onaylamak için kendi şirket imzalarını ekleyebilir.

Bu iki imza ile beraber, paket satıcıya ulaşır. Satıcı da imzaları ve paketi inceleyeyip kendi imzasını da üzerine ekleyebilir.

Şimdi paketi dağıtmayı düşünen başka bir firmaya gidebilir. Bu paketteki her bir imzanın incelenmesinin ardından, bu şirket paketin orijinal kopyası olduğunu ve ilk kez oluşturulduktan sonra değiştirilmediğine emin olabilir. Dağıtımcı firmanın iç deneyimleri sonrasında, onlar da kendi imzalarını, firmalarından onay aldığını göstermek üzere eklemeyi düşünebilirler.

`--addsign` çıktısı şu şekildedir:

```
$ rpm --addsign blather-7.9-1.i386.rpm
            Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
```

Birden çok imzaya sahip paketi incelemek için:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp md5 OK
```

`rpm --checksig` çıktısındaki `pgp` metni paketin daha önce iki kez imzalandığını belirtmektedir.

Aynı şekilde, RPM aynı imzanın birden fazla kez eklenebilmesine izin verir. `--addsign` parametresi imzaların birden fazla olup olmadığını denetlemez.

```
$ rpm --addsig blather-7.9-1.i386.rpm
              Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
              Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --addsig blather-7.9-1.i386.rpm
              Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp pgp pgp pgp md5 OK
```

`rpm --checksig` komutunun çıktısı dört imza sayacaktır.

The output of the `rpm --checksig` command displays four signatures.

Paket İmzasını Değiştirmek

Paketi yeniden inşa etmeden imzasını değiştirmek için, `--resign` parametresini kullanın.

```
$ rpm --resign blather-7.9-1.i386.rpm
            Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
```

`--resign` parametresini birden çok paket dosyası üzerinde kullanmak için:

```
$ rpm --resign b*.rpm
            Enter pass phrase:

Pass phrase is good.
blather-7.9-1.i386.rpm:
bother-3.5-1.i386.rpm:
```

İnşa zamanı imzalamak

İnşa zamanı esnasında bir paketi imzalamak için `rpmbuild` komutunu `--sign` parametresiyle kullanın. Bu, PGP parolasının (passphrase) girilmesini gerektirir.

Örnek:

```
$ rpmbuild -ba --sign blather-7.9.spec
            Enter pass phrase:

Pass phrase is good.
* Package: blather
…
Binary Packaging: blather-7.9-1
Finding dependencies...
…
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
…
Source Packaging: blather-7.9-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
```

Hem ikili hem kaynak paket derlerken ayrıca "Generating signature" (Tr. İmza üretiliyor) imzasına denk gelmekteyiz. Bu mesajın ardından gelen sayı ise PGP ile oluşturulmuş imzayı belirtir.

NOTE	`rpmbuild` için `sign` parametresi aracılığıyla imza eklerken paket derlemek için yalnızca `-bb` ve `-ba` kullanılmalıdır. `-ba`, ikili **ve** kaynak paket inşa etme anlamına gelir.

Bir paketin imzalasını doğrulamak için `rpm` komutunu `--checksig` parametresiyle birlikte kullanın. Örneğin:

```
$ rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm: size pgp md5 OK
```

Birden Çok Paket İnşa Etmek

Birden çok paketi inşa ederken, sürekli PGP parolarını girmekten kaçınmak için aşağıdaki sözdizimini kullanın. Mesela, `blather` ve `bother` paketlerini şu şekilde bir defa imzalayabilirsiniz:

```
$ rpmbuild -ba --sign b*.spec
              Enter pass phrase:

Pass phrase is good.
* Package: blather
…
Binary Packaging: blather-7.9-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
…
Source Packaging: blather-7.9-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
…
* Package: bother
…
Binary Packaging: bother-3.5-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/bother-3.5-1.i386.rpm
…
Source Packaging: bother-3.5-1
…
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/bother-3.5-1.src.rpm
```

Mock

Mock (Tr. Taklit) paketleri inşa etmek için bir araçtır. Farklı mimariler ve farklı Fedora/RHEL sürümleri için paket inşa edebilir. Mock, yeni chrootlar oluşturur ve paketleri içinde inşa eder. Tek görevi, chrootu eksiksiz bir şekilde doldurmak ve bu chroot paket inşa inşa etmeyi denemektir.

Mock aynı şekilde birbirine bağımlı paketler zincirini inşa edebilen `mockchain` isimli bir çoklu-paketleme aracı da sunar.

Mock, kaynak yapılandırmasına göre kaynak RPMler inşa edebilir, eğer `mock-scm` paketi mevcutsa kaynak RPMleri RPMlere de dönüştürebilir, yazılımın belgelerinden -scm-enable kısmını inceleyiniz.

NOTE	Mock'u bir RHEL veya CentOS sistemde kullanmak istiyorsanız "Extra Packages for Enterprise Linux" (Tr. Enterprise Linux için ek paketler) EPEL deposunu aktifleştirmeniz gerekmektedir. Bu depo Fedora topluluğu tarafından sağlanmaktadır ve sistem yöneticileri, geliştiriciler ve RPM paketçileri için faydalı araçlar sunar.

RPM paketçilerinin Mock'u kullanmalarının en yaygın sebeplerinden birisi "saf inşa ortamı" yaratabilmektir. Mock'u "saf inşa ortamı" olarak kullanırsanız sisteminizin mevcut durumu RPM paketinin hiçbir kısmını etkilenmez. Mock, sisteminizde inşa "hedef"ini belirtmek üzere çeşitli yapılandırmalar kullanır, bunları (mock paketini kurduysanız) `/etc/mock` dizininde bulabilirsiniz. Sadece komut satırında dağıtımı veya sürümü belirterek, o sistem için paket inşa edebilirsiniz. Unutulmaması gereken, `mock` ile beraber gelen yapılandırma dosyaları Fedora RPM paketçilerine yöneliktir ve RHEL, CentOS sürümlerinin versiyonları "epel" olarak isimlendirilir çünkü bu paketler bu depo için üretilmiştir. (`.cfg` dosya uzantısını saymazsak) dilediğiniz yapılandırmayı kullanabilirsiniz. Örneğin, hazırladığımız `celaba` örneğini hem RHEL 7 hem de Fedora 23 için tek bir makinede şu komutlarla inşa edebilirsiniz:

```
$ mock -r epel-7-x86_64 ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm

$ mock -r fedora-23-x86_64 ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
```

Neden `mock` kullanmayı isteyeceğinize dair bir örnek şudur: `BuildRequires`'da belirtmeniz gereken bir paketi (buna `hede` diyelim) belirtmeyi unuttunuz ve bu paket laptopunuzda kurulu. `Hede` sisteminizde kurulu olduğu için de paketiniz inşa edildi çünkü inşa için gerekli olan paket zaten sisteminizde kuruldu. Fakat siz bu kaynak RPM'i `hede` kurulu olmayan başka bir sisteme taşırsanız hatalı davranacak, bekleyen bir yan etki oluşturacaktır. Mock kaynak RPM'in içeriğini tarar ve `BuildRequires`'de bahsedilen paketleri chroot içine kurarak bu durumu önler. Yani, eğer siz `BuildRequires` girdisini girmeyi unuttuysanız, derleme başarısız olacaktır çünkü `mock` gerekli paketin nasıl kurulacağını bilemez ve buildroot içerisinde gerekli paket bulunmayacaktır.

Tam aksi bir örneği de düşünebiliriz. Diyelim ki `gcc`'ye bir paketi inşa etmek için ihtiyacınız var fakat sisteminizde kurulu değil. (Bir RPM paketçisinin olmazsa olmazıdır ama varsayalım ki bir şekilde oldu) Mock sayesinde `gcc`yi sisteminize kurmaya gerek yoktur çünkü `mock` işleminin bir parçası olarak chroot içerisine kurulmuş olacak.

Aşağıda sistemimde bulunmayan bir bağımlılığı gerektiren bir paketin yeniden inşa etme girişimimi görmektesiniz. İşin püf noktası şudur; `gcc` paketi bir RPM paketçisinin sisteminde hâlihazırda kurulu olabilir ancak bazı RPM paketleri bir düzineden fazla `BuildRequires` gerektirir. Bu yöntem, sizin belki bir daha ihtiyaç duymacağınız nice lüzumsuz paketten kurtulmanızı sağlar.

```
$ rpmbuild --rebuild ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
Installing /home/admiller/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
error: Failed build dependencies: gcc is needed by celaba-1.0-1.el7.x86_64

$ mock -r epel-7-x86_64 ~/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm
INFO: mock.py version 1.2.17 starting (python version = 2.7.5)...
Start: init plugins
```

```
INFO: selinux enabled
Finish: init plugins
Start: run
INFO: Start(/home/admiller/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm)  Config(epel-7-
x86_64)
Start: clean chroot
Finish: clean chroot
Start: chroot init
INFO: calling preinit hooks
INFO: enabled root cache
Start: unpacking root cache
Finish: unpacking root cache
INFO: enabled yum cache
Start: cleaning yum metadata
Finish: cleaning yum metadata
Mock Version: 1.2.17
INFO: Mock Version: 1.2.17
Start: yum update
base                                                                     | 3.6 kB
00:00:00
epel                                                                     | 4.3 kB
00:00:00
extras                                                                   | 3.4 kB
00:00:00
updates                                                                  | 3.4 kB
00:00:00
No packages marked for update
Finish: yum update
Finish: chroot init
Start: build phase for celaba-1.0-1.el7.src.rpm
Start: build setup for celaba-1.0-1.el7.src.rpm
warning: Could not canonicalize hostname: rhel7
Building target platforms: x86_64
Building for target x86_64
Wrote: /builddir/build/SRPMS/celaba-1.0-1.el7.centos.src.rpm
Getting requirements for celaba-1.0-1.el7.centos.src
 --> Already installed : gcc-4.8.5-4.el7.x86_64
 --> Already installed : 1:make-3.82-21.el7.x86_64
No uninstalled build requires
Finish: build setup for celaba-1.0-1.el7.src.rpm
Start: rpmbuild celaba-1.0-1.el7.src.rpm
Building target platforms: x86_64
Building for target x86_64
Executing(%prep): /bin/sh -e /var/tmp/rpm-tmp.v9rPOF
+ umask 022
+ cd /builddir/build/BUILD
+ cd /builddir/build/BUILD
+ rm -rf celaba-1.0
+ /usr/bin/gzip -dc /builddir/build/SOURCES/celaba-1.0.tar.gz
+ /usr/bin/tar -xf -
+ STATUS=0
```

```
+ '[' 0 -ne 0 ']'
+ cd celaba-1.0
+ /usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
Patch #0 (celaba-output-first-patch.patch):
+ echo 'Patch #0 (celaba-output-first-patch.patch):'
+ /usr/bin/cat /builddir/build/SOURCES/celaba-output-first-patch.patch
patching file celaba.c
+ /usr/bin/patch -p0 --fuzz=0
+ exit 0
Executing(%build): /bin/sh -e /var/tmp/rpm-tmp.UxRVtI
+ umask 022
+ cd /builddir/build/BUILD
+ cd celaba-1.0
+ make -j2
gcc -g -o celaba celaba.c
+ exit 0
Executing(%install): /bin/sh -e /var/tmp/rpm-tmp.K3i2dL
+ umask 022
+ cd /builddir/build/BUILD
+ '[' /builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64 '!=' / ']'
+ rm -rf /builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64
++ dirname /builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64
+ mkdir -p /builddir/build/BUILDROOT
+ mkdir /builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64
+ cd celaba-1.0
+ /usr/bin/make install DESTDIR=/builddir/build/BUILDROOT/celaba-1.0-
1.el7.centos.x86_64
mkdir -p /builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64/usr/bin
install -m 0755 celaba /builddir/build/BUILDROOT/celaba-1.0-
1.el7.centos.x86_64/usr/bin/celaba
+ /usr/lib/rpm/find-debuginfo.sh --strict-build-id -m --run-dwz --dwz-low-mem-die
-limit 10000000 --dwz-max-die-limit 110000000 /builddir/build/BUILD/celaba-1.0
extracting debug info from /builddir/build/BUILDROOT/celaba-1.0-
1.el7.centos.x86_64/usr/bin/celaba
dwz: Too few files for multifile optimization
/usr/lib/rpm/sepdebugcrcfix: Updated 0 CRC32s, 1 CRC32s did match.
+ /usr/lib/rpm/check-buildroot
+ /usr/lib/rpm/redhat/brp-compress
+ /usr/lib/rpm/redhat/brp-strip-static-archive /usr/bin/strip
+ /usr/lib/rpm/brp-python-bytecompile /usr/bin/python 1
+ /usr/lib/rpm/redhat/brp-python-hardlink
+ /usr/lib/rpm/redhat/brp-java-repack-jars
Processing files: celaba-1.0-1.el7.centos.x86_64
Executing(%license): /bin/sh -e /var/tmp/rpm-tmp.vxtAuO
+ umask 022
+ cd /builddir/build/BUILD
+ cd celaba-1.0
+ LICENSEDIR=/builddir/build/BUILDROOT/celaba-1.0-
1.el7.centos.x86_64/usr/share/licenses/celaba-1.0
+ export LICENSEDIR
+ /usr/bin/mkdir -p /builddir/build/BUILDROOT/celaba-1.0-
```

```
1.el7.centos.x86_64/usr/share/licenses/celaba-1.0
+ cp -pr LICENSE /builddir/build/BUILDROOT/celaba-1.0-
1.el7.centos.x86_64/usr/share/licenses/celaba-1.0
+ exit 0
Provides: celaba = 1.0-1.el7.centos celaba(x86-64) = 1.0-1.el7.centos
Requires(rpmlib): rpmlib(CompressedFileNames) <= 3.0.4-1 rpmlib(FileDigests) <= 4.6.0-
1 rpmlib(PayloadFilesHavePrefix) <= 4.0-1
Requires: libc.so.6()(64bit) libc.so.6(GLIBC_2.2.5)(64bit) rtld(GNU_HASH)
Processing files: celaba-debuginfo-1.0-1.el7.centos.x86_64
Provides: celaba-debuginfo = 1.0-1.el7.centos celaba-debuginfo(x86-64) = 1.0-
1.el7.centos
Requires(rpmlib): rpmlib(FileDigests) <= 4.6.0-1 rpmlib(PayloadFilesHavePrefix) <=
4.0-1 rpmlib(CompressedFileNames) <= 3.0.4-1
Checking for unpackaged file(s): /usr/lib/rpm/check-files
/builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64
Wrote: /builddir/build/RPMS/celaba-1.0-1.el7.centos.x86_64.rpm
warning: Could not canonicalize hostname: rhel7
Wrote: /builddir/build/RPMS/celaba-debuginfo-1.0-1.el7.centos.x86_64.rpm
Executing(%clean): /bin/sh -e /var/tmp/rpm-tmp.JuPOtY
+ umask 022
+ cd /builddir/build/BUILD
+ cd celaba-1.0
+ /usr/bin/rm -rf /builddir/build/BUILDROOT/celaba-1.0-1.el7.centos.x86_64
+ exit 0
Finish: rpmbuild celaba-1.0-1.el7.src.rpm
Finish: build phase for celaba-1.0-1.el7.src.rpm
INFO: Done(/home/admiller/rpmbuild/SRPMS/celaba-1.0-1.el7.src.rpm) Config(epel-7-
x86_64) 0 minutes 16 seconds
INFO: Results and/or logs in: /var/lib/mock/epel-7-x86_64/result
Finish: run
```

Göreceğiniz üzere, `mock` çıktısı biraz fazla "detaylı". Aynı şekilde pek çok (mock hedefinin RHEL7, CentOS7 veyahut Fedora olmasına göre) yum veya dnf çıktısını da görmektesiniz ki daha önce mock hedefi üzerinde önceden gerekli paketleri kuran, önbelleğe taşıyan ve hazırlayan `-init` parametresini çalıştırsaydınız (Örn: `mock -r epel-7-x86_64 --init`) bu çıktılarla karşılaşmayacaktınız.

Daha fazla için Mock hakkındaki güncel belgeleri inceleyebilirsiniz.

Versiyon Kontrol Sistemleri

Çeviri Notu: Bu bölüm çevirirken en çok zorlandığım bölüm. Eğer cümlelerin anlamsız göründüğünü düşünür, esas anlamını karşılamadığını düşünürseniz muhakkak haber verin.

RPMlerle çalışırken, çoğu zaman paketlediğimiz yazılımın içeriğini yönetebilmek için git gibi bir Versiyon Kontrol Sisteminden (İngilizce: Version Control System, VCS olarak kısaltılır ve çeviride bu kısaltma kullanılacaktır) yararlanmayı dileyebilirsiniz. Dikkat edilmesi gereken bir husus, bir VCS'i ikili dosyaları depolamak için kullanmak pek tercih edilen bir yöntem değildir. Çünkü bu araçlar dosyaların değişikliklerini ölçmek içindir (çoğunlukla metin dosyaları için optimize edilmiştir) ve

ikili dosyalar bu özelliği sunamazlar. Üstelik, ikili dosyalar kaynak deposunun büyüklüğünü hızla şişirirler.

Bu soruna karşı Açık kaynak projelerinin çözümleri şunlardır: ya SPEC dosyası VCS'teki kaynak kodunun içerisinde depolanır ya da VCS'e sadece SPEC dosyası ve yamalarla beraber kaynak kodlarının sıkıştırılmış arşivi yüklenir ki bu arşive "gözardı edilen önbellek" (look aside cache) denir.

Bu kısımda RPM paketlerine dönüştürecek içerikleri yönetmek için VCS (git) kullanmanın iki yolundan bahsedeceğiz: tito ve dist-git.

NOTE	Bu kısımdakileri takip edebilmek için sisteminize `git` isimli paketi kurmalısınız.

tito

Tito, paketlenecek yazılımın bütün kaynak kodunun git deposu içerisinde olduğunu varsayarak kullanılan bir araçtır. Tito, yazılım geliştiren bir ekibin normal Dallanma Akışını sürdürmesine izin verdiği için DevOps çalışanları için uygun bir tercihtir.

Tito yazılımın kademe kademe paketlenmesine izin verir. Paket süreçlerle otomatik olarak inşa edilir ve RPM tabanlı sistemler için olağan kurulum deneyimini sağlar.

NOTE	Tito paketi Fedora ve RHEL 7 ile CentOS 7 üzerine kullanılabilen EPEL deposu için mevcuttur.

Tito, git etiketleri aracılığıyla çalışır ve izin verirseniz sizin için yönetir. Bunun yanında isteğe bağlı olarak tercih ettiğiniz etiketleme şeması altında çalışması için de yapılandırılabilir.

Şimdi Tito hakkında bir keşfe çıkalım ve Tito'yu kullanan bir projeye göz atalım. Bu proje, gelecek kısımların birisinin konusu olacak olan dist-git'dir. Hazır bu proje GitHub'da yayınlamışken onu klonlayabiliriz.

```
$ git clone https://github.com/release-engineering/dist-git.git
Cloning into 'dist-git'...
remote: Counting objects: 425, done.
remote: Total 425 (delta 0), reused 0 (delta 0), pack-reused 425
Receiving objects: 100% (425/425), 268.76 KiB | 0 bytes/s, done.
Resolving deltas: 100% (184/184), done.
Checking connectivity... done.

$ cd dist-git/

$ ls *.spec
dist-git.spec

$ tree rel-eng/
rel-eng/
├─── packages
│    └─── dist-git
└─── tito.props

1 directory, 2 files
```

Görebileceğiniz üzere, SPEC dosyası git deposunun kök dizininde ve `rel-eng` dizini de Tito'nun veri sayımları, yapılandırması ve özel Tito modülleri gibi çeşitli detayları barındırır.

Aynı şekilde, `rel-eng` dizininde görüleceği gibi `packages` (Tr. paketler) isimli bir alt dizin vardır. Bu da her paket için bir dosya depolar ki tek bir git deposunda Tito bu işi halleder. Tek bir git deposunda bir çok RPM olabilir ve Tito bunu kolaylıkla halleder.

Ancak bu örneğinizde paketler listesinde yalnızca bir tek paketi görmekteyiz. Ayrıca dikkat edilmelidir ki bu paketin ismi de SPEC dosyasının adıyla örtüşmektedir. dist-git geliştiricilerinin git depolarını Tito ile yönetmeleri için tek gereken şey `tito init` komutunu kullanmalarıdır.

Eğer tam bir DevOpscu gibi davranmak istersek Sürekli Entegrasyon (Continuous Integration - CI olarak kısaltılır) ve Sürekli Teslimat (Continuous Delivery - CD olarak kısaltılır) sürecinin bir parçası olarak kullanmayı isteyebiliriz.

Bu örnekte yapacağımız şey Tito için bir "deneme inşası" gerçekleştirmek ki bunun için `Mock`'u dahi kullanabiliriz. Çıktıyı iş hattının (pipeline) diğer parçaları için bir kurulum noktası olarak da kullanabiliriz. Aşağıda bunu gerçekleştiren basit bir komut örneği verilmiştir ve çeşitli ortamlara uyarlanabilir.

```
$ tito build --test --srpm
Building package [dist-git-0.13-1]
Wrote: /tmp/tito/dist-git-git-0.efa5ab8.tar.gz

Wrote: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm

$ tito build --builder=mock --arg mock=epel-7-x86_64 --test --rpm
Building package [dist-git-0.13-1]
Creating rpms for dist-git-git-0.efa5ab8 in mock: epel-7-x86_64
Wrote: /tmp/tito/dist-git-git-0.efa5ab8.tar.gz

Wrote: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm

Using srpm: /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.fc23.src.rpm
Initializing mock...
Installing deps in mock...
Building RPMs in mock...
Wrote:
  /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm
  /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm

$ sudo yum localinstall /tmp/tito/dist-git-*.noarch.rpm
Loaded plugins: product-id, search-disabled-repos, subscription-manager
Examining /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm: dist-git-
0.13-1.git.0.efa5ab8.el7.centos.noarch
Marking /tmp/tito/dist-git-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm to be installed
Examining /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm: dist-
git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch
Marking /tmp/tito/dist-git-selinux-0.13-1.git.0.efa5ab8.el7.centos.noarch.rpm to be
installed
Resolving Dependencies
--> Running transaction check
---> Package dist-git.noarch 0:0.13-1.git.0.efa5ab8.el7.centos will be installed
```

Dikkat edilmelidir ki, son komutun sudo veyahut root yetkileriyle çalıştırılması gereklidir ve fazla uzamaması için çıktısı kırpılmıştır zira bağımlılıklar listesi oldukça uzun sayılabilir.

Bu kısa örneğimizde Tito kullanımından kısaca bahsettik, ancak geleneksel Sistem Yöneticileri, RPM paketçileri ve DevOpsçular için ne kadar faydalı özellikleri olduğunu göstermeye çalıştık. *Tito*'nun GitHub sitesinde bulunan belgelere göz atmanızı şiddetle tavsiye ederiz. Burada, bir projeye başlamak için ferekenlerden çeşitli ileri düzey konulara kadar pek çok şey bulabilirsiniz.

dist-git

dist-git aracı Tito'ya göre daha farklı bir yaklaşımı ele alır. Salt kaynak kodunu gitte tutmak yerine "gözardı edilen önbellek" olarak da anılan kaynak kodunu sıkıştırılmış bir arşivde spec dosyası ve yamalarıyla birlikte saklar.

Görmezden gelinen bellek (ya da look-aside-cache), RPM İnşa Sistemlerinin büyük boyutlu

dosyaları nasıl idare ettiğini belirten bir terimdir. Koji gibi RPM İnşa Sistemlerde sıklıkla kullanılan bir yaklaşımı ifade eder. İnşa sistemi, SPEC dosyasında belirtilen `SourceX` girdilerindeki verileri "çeker" (bkz: pull). Arşiv dosyası güncellenirken, versiyon kontrol sistemindeki SPEC dosyası ve yamaları sabit kalır. Aynı şekilde, bu yöntem için yardımcı olan bir komut satırı aracı da bulunmaktadır.

Başka bir belgelendirmeyi buraya kopyalamak yerine belgenin aslına yönlendirmeyi tercih ediyoruz. Bunun gibi bir sistemin nasıl kurulacağını öğrenmek istiyorsanız, dist-git belgelerini inceleyebilirsiniz.

Makrolar Hakkında Daha Fazlası

Gömülü pek çok RPM Makrosu vardır ve bunlardan birkaçını bu kısımda ele alacağız. Ancak dilenirse RPM Resmi Belgesinde kapsamlı bir listesiye ulaşabilirsiniz.

Sizin Linux dağıtımınızın da kendisine özgü bazı makroları da bulunmaktadır. Bu belgede Fedora, CentOS ve RHEL tarafından sağlanan pek çok makroyu göstereceğiz. Aynı zamanda makrolarla sisteminizi nasıl inceleyeceğinizi de bu bölümde alacağız. Ancak, diğer RPM tabanlı Linux dağıtımlarına özgü makroları tanıt**ma**yacağız.

Kendi Makronuzu Tanımlayın

Kendi makronuzu tanımlayabilirsiniz. RPM Resmi Belgesinden yapılan makroların yeteneklerini gösteren detaylı bir alıntıyı aşağıda görebilirsiniz.

Bir makro tanımlamak için:

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

`\` işaretinin ardından gelen boşluklar kaldırılacaktır. İsim, en üç harften oluşmalı ve yalnızca sayılardan, isimlerden ve `_` işaretinden oluşmalıdır. `(opts)` kısmı olmayan bir makro, yalnızca özyinelemeli makro genişletmesi yapabileceğinden "sade" bir makrodur. Parametre verilebilen bir makro `(opts)` kısmını barındırır. (Parantezlerle beraber bir metin olan) `(opts)` makro çağırmanın başlangıcındaki getopt(3) için argv/argc* işlemi gibidir.

*Argc, Argument Count'un kısaltmasıdır ve argüman sayısı anlamına gelir. Argv Argument Vector'ün kısaltmasıdır ve argümanın içeriğini gösterir.

NOTE

Eski RPM SPEC dosyaları `%define <name> <body>` makro örüntüsünü kullanabilir. `%define` ve `%global` atasındaki farklar şöyledir:

- `%define` daha yerel bir kapsam içindir. Yani, SPEC dosyasının belirli bir kısmına yöneliktir. Bunun yanısıra, `%define` kısmı tembelce değerlendirilir, ancak kullanıldığında genişletilir.
- `%global` ise SPEC dosyasının tamamını içine alan bir kapsam içindir. Derlenir derlenmez genişletilir.

Örnekler:

```
%global githash 0ec4e58
%global python_sitelib %(%{__python} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())")
```

NOTE Yorum içinde dahi olsa, makrolar muhakkak ki genişletilir. Kimi zaman bu zararsızdır, ancak ikinci örnekte göreceğiniz gibi için bir Python komutunu çalıştırmış bulunmaktayız. Bu komut siz yoruma alsanız da, %changelog içinde bahsetseniz de çalışacaktır. Bir makroyu yoruma almak için, `%%` kullanın. Mesela ki: `%%global`.

%setup

`%setup` makrosu tarball hâlindeki kaynak kodlarından paketleri inşa etmek için kullanılır. `%setup` makrosunun öntanımlı hareket tarzı `rpmbuild` çıktısında göreceğiniz gibidir. Her bir makro safhasının başlangıcında `Executing(%falancafilanca)` çıktısını verir. Mesela:

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

`set -x` aktifleştirilmiş bir kabuk çıktısı buna benzer. `/var/tmp/rpm-tmp.DhddsG` çıktısını merak ediyorsanız `-debug` kullanın çünkü `Rpmbuild` başarılı bir inşadan sonra geçici çıktıları siler. Aşağıdaki örnekte ortam değişkenlerinin yapılandırmasını görebilirsiniz:

```
cd '/builddir/build/BUILD'
rm -rf 'celaba-1.0'
/usr/bin/gzip -dc '/builddir/build/SOURCES/celaba-1.0.tar.gz' | /usr/bin/tar -xof -
STATUS=$?
if [ $STATUS -ne 0 ]; then
  exit $STATUS
fi
cd 'celaba-1.0'
/usr/bin/chmod -Rf a+rX,u+w,g-w,o-w .
```

`%setup` doğru dizinde çalıştığımıza emin olur, daha önceki inşalardan kalan kalıntıları siler, tarball kaynağını açar ve çeşitli yetkiler tanımlar. `%setup` makrosunun davranışını belirleyen çeşitli parametreler vardır.

%setup -q

`-q` parametresi `%setup` makrosunun çıktısını azaltır. `tar -xvvof` yerine `tar -xof` kullanılır. Bu parametre, ilk seferde kullanılmalıdır.

Option `-q` limits verbosity of `%setup` macro. Only `tar -xof` is executed instead of `tar -xvvof`. This option has to be used as first.

%setup -n

Bazı durumlarda, tarballdan gelen dizinin ismi `%{name}-%{version}` ikilisinden farklıdır. Bu, `%setup` makrosunun işleyişini sekteye uğratır. Dizin adı `-n dizin_adi` parametresiyle belirtilmelidir.

Örneğin, paketin adı `celaba`'dır ama kaynak kodunun arşivi `hello-1.0.tgz` ismindedir ve içeriğinde `hello/` dizinin barındırır. O zaman SPEC dosyasının içeriği şu şekilde olmalıdır:

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

%setup -c

`-c` parametresi kaynak kodunun arşivi açıldıktan sonra herhangi bir alt dizin içermiyorsa kullanılmalıdır. Arşivden gelecek dosyalar bütün dizini dolduracaktır ve `-c` parametresi bir dizin açıp arşivi içine genişletir. Bu makronun ne yaptığını tasvir eden bir örnek:

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

Dizin, arşiv genişletildikten sonra değişmeyecektir.

%setup -D ve -T

`-D` parametresi kaynak kodu dizinin silinmesini engeller. Tekrar tekrar `%setup` makrosunun kullanıldığı durumlar için faydalıdır. Aslında, `-D` parametresinin yap**ma**dığı iş şudur:

```
rm -rf 'celaba-1.0'
```

`-T` parametresi ise tarball'ın gelişletilmesine engel olur. Yani şu satırlar çalış**ma**yacaktır:

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

%setup -a ve -b

`-a` ve `-b` belirli kaynakları genişletir.

- `-b` (İngilizce "Önce" demek olan "B"efore kelimesinden gelir) belirlenmiş kaynağı çalışma dizinine girmeden önce genişletir.
- `-a` (İngilizce "Sonra" demek olan "A"fter kelimesinden gelir) belirlenmiş kaynağı çalışma dizinne girdikten sonra genişletir.

Argümanlarındaki kaynak numaraları SPEC dosyasındaki `SourceX` girdilerinden gelir.

Örneğin `celaba-1.0.tar.gz` içerisinde boş bir `örnekler` dizini var ve bu dizinin içeriği `örnekler.tar.gz` isimli başka bir arşivin içinde. O zaman, `Source1`'i çalışma dizinine girdikten sonra genişletmek için `-a 1` parametresini kullanmalıyız.

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

Fakat elimizde `celaba-1.0/örnekler` dizinine dosyaları çıkaran `celaba-1.0-örnekler.tar.gz` isimli bir arşiv varsa, `-b 1` parametresini kullanmalıyız. Çünkü çalışma dizinine girmeden önce `Source1` genişletilmelidir.

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

Aynı zamanda, üstteki parametreleri bir arada kullanabilirsiniz.

%files

`%files` yönergesinde kullanılan yaygın makrolar şu şekildedir:

Makro	Açıklama
%license	LICENCE dosyasını belirtir, sisteme uygun şekilde yükler ve RPM tarafından tanımlanır. Örnek kullanım: `%licence LICENCE`
%doc	Sisteme kurulmuş olan ve RPM tanımlanmış belgeleri işaretler. Çoğu zaman yalnızca yazılıma yönelik belgelendirmeyi içermez, belgelendirmeyi tamamlayan aynı zamanda örnek kullanımları ve farklı türden parçaları da içerir. Sözkonusu olan durumda kod örnekleri de dahildir. Dosyadan çalıştırılabilir kod çıkarılırken dikkatli olunmalıdır. Örnek kullanım: `%doc README`
%dir	Bu RPM paketi tarafından sahiplenilen konumu belirtilir. Paketin kaldırılması gerektiği zaman hangi dizinler ive dosyaların kaldırıldığını bilinmesi açısından oldukça önemlidir. Örnek kullanım: `%dir %{_libdir}/%{name}`
%config(noreplace)	Belirtilen dosyanın bir yapılandırma dosyası olduğunu belirtir. Eğer belirtilen dosyasının sağlama değeri (checksum) orijinal dosyadan ayrıysa RPM bu dosyaları koruyacaktır. Eğer bir değişim gerekecekse, sonu `.rpmnew` ile biten bir dosya oluşturulur ve son kullanıcının sisteminde daha önceden bulunan/değiştirilmiş dosya değiştirilmez. Örnek kullanım: `%config(noreplace) %{_sysconfdir}/%{name}/%{name}.conf`

Gömülü Makrolar

Sisteminizde pek gömülü makro bulunmaktadır ve bu makroların hepsini görmenin en kolay `rpm --showrc` komutunu çalıştırmaktır. Ancak ki çıktı fazlaca uzun olduğu için `grep` komutuyla kısaltmak, genellikle tercih edilen bir yöntemdir.

Aynı şekilde, `rpm -ql rpm` komutunun çıktısında, içinde `macros` dosyalardan da daha fazla bilgi alabilirsiniz. Bu dosyalar, sisteminizdeki RPM sürümüyle birlikte gelir.

RPM Dağıtım Makroları

Different distributions will supply different sets of recommended RPM Macros based on the language implementation of the software being packaged or the specific guidelines of the distribution in question.

Bu makrolar, RPM Paketleri tarafından sunulur ve yum veyahut dnf gibi dağıtımlara ait paket yöneticileri aracılığıyla kurulabilirler. Makro dosyaları kurulduktan sonra `/usr/lib/rpm/macros.d/` içerisinde bulunabilir ve `rpm --showrc` çıktısında görülebilir.

Bunun başlıca örneklerinden birisi, Fedora Paketleme Kılavuzunda ilişkili olan Uygulamaya Özgü Kılavuzlar bölümüdür. Bu rehberin yazıldığı tarihte, RPM Makrolarıyla birlikte 60'ın üzerinde farklı RPM paketleme kılavuzu bulunmaktadır.

Bunlara bir örnek, Python'un 2. versiyonu için hazırlanmış olan `python2-rpm-macros` paketidir. (RHEL 7 ve CentOS 7 için EPEL deposunda bulunmaktadır.) Python2'ye özgü pek çok makroya bu paketten ulaşabilirsiniz.

```
$ rpm -ql python2-rpm-macros
/usr/lib/rpm/macros.d/macros.python2

$ rpm --showrc | grep python2
-14: __python2  /usr/bin/python2
CFLAGS="%{optflags}" %{__python2} %{py_setup} %{?py_setup_args} build
--executable="%{__python2} %{py2_shbang_opts}" %{?1}
CFLAGS="%{optflags}" %{__python2} %{py_setup} %{?py_setup_args} install -O1 --skip
-build --root %{buildroot} %{?1}
-14: python2_sitearch   %(%{__python2} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib(1))")
-14: python2_sitelib    %(%{__python2} -c "from distutils.sysconfig import
get_python_lib; print(get_python_lib())")
-14: python2_version    %(%{__python2} -c "import sys;
sys.stdout.write('{0.major}.{0.minor}'.format(sys.version_info))")
-14: python2_version_nodots     %(%{__python2} -c "import sys;
sys.stdout.write('{0.major}{0.minor}'.format(sys.version_info))")
```

Üstteki örnekte ham RPM makro tanımlarını görmektesiniz. Eğer dilenirse `rpm --eval` komutunu kullanarak bu makroları daha anlaşılır bir biçimde tekrardan yorumlatabilirsiniz. Bu sayede RPM paketlerken hangi makronun ne işe yaracağını inceleyebilirsiniz.

```
$ rpm --eval %{__python2}
/usr/bin/python2

$ rpm --eval %{python2_sitearch}
/usr/lib64/python2.7/site-packages

$ rpm --eval %{python2_sitelib}
/usr/lib/python2.7/site-packages

$ rpm --eval %{python2_version}
2.7

$ rpm --eval %{python2_version_nodots}
27
```

Özel Makrolar

`~/.rpmmacros` dosyasını düzenleyerek dağıtım makrolarını geçersiz bırakabilirsiniz. Burada yapacağınız her bir değişiklik gelecekteki inşa süreçlerine etkide bulunacaktır.

Değiştirebileceğiniz bazı makrolar şu şekildedir:

`%_topdir /opt/falanca/calisma/dizini/rpmbuild`

: Bu dizini oluşturup oluşturabilir ve `rpmdev-setuptree` aracının oluşturduğu dizinleri ekleyebilirsiniz. Bu makronun değeri öntanımlı olarak `~/rpmbuild`'dir.

`%_smp_mflags -l3`

: Bu makro genellikle Makefile'a geçmek için kullanılırdı. Mesela: `make %{?_smp_mflags}` veya inşa sürecinde eşzamanlı işlikçilerin sayısını belirlemek. Öntanımlı olarak, `-jX`'i temsil eder ki `X` burada işlikçi sayısını belirtir. Bu numarayı değiştirerek paketlerin inşa sürecini hızlandırabilir ya da yavaşlatabilirsiniz.

`~/rpmmacros` içerisinde yeni makrolar belirleyebilirsiniz, ancak bunu yapmayın. Çünkü bu makrolar yalnızca sizin makinenizde tanımlanmış olacak. Diğer kullanıcılar paketleri tekrar inşa etmek istediği zaman bu makrolara erişemeyebilirler.

Epoch, Scriptlets, and Triggers

RPM SPEC dosyaları âlemindeki bazı konular daha ileri seviye sayılır çünkü bu konular yalnızca SPEC dosyasını değil, paketin nasıl derlendiğini ve aynı zamanda bu RPM paketi kurulacağı zaman nasıl tepki göstermesi gerektiğini de ilgilendirir.

Bu bölümde bu ileri düzey konulardan en bilinenleri olan dönemleri, betikçileri ve tetikleyicileri ele alacağız.

Dönem

İlk olarak `Dönem (Epoch)`'i ele alacağız. dönemler belirli bir versiyon numarasına yönelik bağımlılıkları tanımlar. Eğer `Epoch` yönergesi SPEC dosyasında belirtilmemişse öntanımlı değer 0'dır. Bu rehberin SPEC yazımıyla ilgili kısmında dönemleri ele almadık, çünkü versiyon kıyaslamak normalde RPM'in yaptığı bir iş olduğu için bunu belirtmek ayrıca kafa karıştırıcı olabilir.

`Epoch: 1` ve `Version: 1.0` yönergelerine sahip `hedehödö` diye bir paketin sisteme kurulduğunu farzedelim. Ve başka birisi de `hedehödö`'yü `Version: 2.0` yönergesiyle paketledi ancak dalgınlıkla `Epoch` yönergesini belirtmeyi unuttu. Yeni versiyon, RPM tarafından bir güncelleme olarak kabul edilmeyecektir çünkü RPM paketlerinin versiyonlarını belirtirken dönem (epoch) geleneksel İsim-Versiyon-Sürüm üçlemesinden daha öncelikli kabul edilir.

Bu yaklaşım ancak çok ama çok gerekli olduğu zaman son çare olarak kullanılır. Amaç, sürüm numaralandırmalarındaki değişikliklerden doğan yan etkileri telafi etmektir. Bir başka sebebi ise sürüm numalarında kullanılan kıyaslanması mümkün olmayan karakterleri telafi etmektir.

Betikçiler ve Tetikleyiciler

Paketin kurulması esnasında sistem üzerinde gereken veyahut istenen değişikliği gerçekleştirmek için kullanılan birtakım yönergeler vardır. Bu yönergeler **betikçiler** (İng: Scriptlet) olarak adlandırılır.

Neden böyle bir şeyi kullanmak isteyeceğinize dair bir örnek verelim. Diyelim ki sisteme yeni bir hizmet yerleştiren bir paket kurduk ve içeriğinde systemd'ye ait bir birim dosyası bulunmakta. Bu durumda systemd'ye `filanca` isimli hayali paketin kurulumundan sonra `systemctl start filanca.service` komutunun kullanılabileceğini bildirmemiz gerek. Aynı şekilde, kaldıracağımız zaman da tam aksini gerçekleştirebilmemiz gerekir, böylece birim dosyası Systemd'nin yapılandırmasına kendisini sürdürmeye devam ederken sözkonusu arkaplan programının kaldırılması başka hataları tetiklemez.

Sıklıkla kullanılan betikçilerin sayısı oldukça azdır ve kullanım şekilleri `%build` ile `%install` gibi "gövde başlık" yönergeleriyle oldukça benzeşirler. (Mesela birden çok satıra yayılabilirler.) Çoğu zaman bir POSIX uyumlu bir kabuk betiği ile yazılırlar ancak hedef dağıtımın izin verdiği programlama dillerinde de hazırlanabilirler. Hangi dillerle yazılabileceğinin kapsamlı bir listesini *RPM Resmi Belgesisinde* bulabilirsiniz.

Betikçi yönergeleri şunlardır:

Yönerge	Açıklama
`%pre`	Paket hedef sisteme kurulmadan hemen önce çalıştırılacak betikçi
`%post`	Paket hedef sisteme kurulduktan hemen sonra çalıştırılacak betikçi
`%preun`	Paket hedef sistemden kaldırılmadan hemen önce çalıştırılacak betikçi
`%postun`	Paket hedef sistemden kaldırıldıktan hemen sonra çalıştırılacak betikçi

Aynı zamanda bu işlev için sıklıkla kullanılan RPM makroları da vardır. Mesela daha önce bahsettiğimiz systemd'nin yeni bir birim dosyası için tetiklenmesi gerektiğini belirtmenin daha

kolay yolları da vardır. Alttaki çıktıdan inceleyebilirsiniz. Daha fazla bilgi için Fedora Systemd Paketleme Kılavuzuna göz atın.

```
$ 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
```

RPM işlemi üzerinde daha da incelikli bir kontrol arzuluyorsanız **tetikleyicilere** (İng. triggers) göz atabilirsiniz. Hemen hemen betikçilerle aynı işlevi görürler, ancak kurulum veya yükseltme işlemi içerisinde özel bir sıralamaya göre çalıştırılırlar, işlem üzerinde çok daha hassa bir kontrol

sağlarlar.

Tetikçiler, tetiklenme sıralarına göre detaylarıyla birlikte aşağıda listelenmişlerdir:

```
all-%pretrans
...
any-%triggerprein (%triggerprein - öteki paketler yeni bir kuruluma hazırlanırken)
new-%triggerprein
new-%pre      Paketin yeni versiyonu kurulurken
...           (bütün yeni dosyalar kurulur)
new-%post     Paketin yeni versiyou kurulurken

any-%triggerin (%triggerin - öteki paketler yeni bir kuruluma hazırlanırken)
new-%triggerin
old-%triggerun
any-%triggerun (%triggerun - öteki paketler kaldıırlmaya hazırlanırken)

old-%preun    Paketin eski versiyonu kaldırılırken
...           (bütün eski dosyalar kaldırılır)
old-%postun   Paketin eski versiyonu kaldırılırken

old-%triggerpostun
any-%triggerpostun (%triggerpostun - diğer paketler de kaldırılırken)
...
all-%posttrans
```

Yukarıdaki maddeler Fedora sistemlerde `/usr/share/doc/rpm/triggers`, RHEL 7 ve CentOS 7 sistemlerde ise `/usr/share/doc/rpm-4.*/triggers` adresinde bulunan RPM belgelerinden alıntıdır.

Kabuk Dışı Betikleri SPEC Dosyasında Kullanmak

`-p` isimli betikçi parametresi, SPEC dosyası içerisinde belirli bir yorumlayıcıyı çağırmamıza izin verir. Öntanımlı değer `-p /bin/sh`dir. Aşağıdaki açıklayıcı örnekte `pelaba.py` dosyasının kurulumundan sonra ekrana bir mesajı bastıran bir betiği görmektesiniz.

1. `pello.spec` dosyasını açın.
2. Şu satırı bulun:

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

   Bu satırın altına, şu kodu ekleyin:

   ```
   %post -p /usr/bin/python3
   print("Hey! Bu bir {} kodu!".format("python"))
   ```

3. RPMleri İnşa Etmek kısmına göre paketinizi inşa edin.
4. Paketinizi kurun:

```
# dnf install /home/<username>/rpmbuild/RPMS/noarch/pello-0.1.1-1.fc27.noarch.rpm
```

Paketleri kurduktan sonra çıktıda şu mesajı göreceksiniz:

```
Installing       : pello-0.1.1-1.fc27.noarch                                1/1
Running scriptlet: pello-0.1.1-1.fc27.noarch                                1/1
Hey! Bu bir python kodu!
```

NOTE

- Bir Python 3 betiği kullanmak için: SPEC dosyasındaki `install -m` satırı altında `%post -p /usr/bin/python3` satırını ekleyin.
- Bir Lua betiği kullanmak için: SPEC dosyasındaki `install -m` satırı altına `%post -p <lua>` satırını ekleyin.
- Bu yolla dilediğiniz yorumlayıcıyı SPEC dosyasında kullanabilirsiniz.

RPM'de Koşullu İfadeler

Koşullu ifadeler, SPEC dosyasının çeşitli kısımlarında kullanılabilir.

Koşullu ifadeler çoğunlukla şunlar için kullanılır:

- mimariye özgü durumlar için
- işletim sistemine özgü durumlar için
- işletim sisteminin sürümleri arasındaki uyumluluk sorunlarıyla başa çıkmak için
- makroların varlığını ve tanımını sorgulamak için

Koşullu İfadelerin Sözdizimi

Eğer *ifade* doğruysa bir şeyler yap:

```
%if ifade
...
%endif
```

Eğer *ifade* doğruysa bir şeyler yap. Değilse, başka bir şey yap:

```
%if ifade
...
%else
...
%endif
```

Koşullu İfadelerin Örnekleri:

`%if` koşulu

```
%if 0%{?rhel} == 6
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' configure.in
sed -i '/AS_FUNCTION_DESCRIBE/ s/^/#/' acinclude.m4
%endif
```

Bu koşul, RHEL6 ve diğer işletim sistemleri için AS_FUNCTION_DESCRIBE makrosunun desteğini düzenler. Eğer paket RHEL için inşa edilirse `%rhel` makrosu tanımlanmış olacaktır ve RHEL sürümünü verecektir. Eğer bu değer 6 ise, bu paket RHEL 6 için derlenecektir ve RHEL 6 tarafından desteklenmeyen AS_FUNCTION_DESCRIBE makrosunu otomatik yapılandırma betiklerinden silecektir.

```
%if 0%{?el6}
%global ruby_sitearch %(ruby -rrbconfig -e 'puts Config::CONFIG["sitearchdir"]')
%endif
```

Bu koşul ise Fedora 17 ile daha yeni versiyonlarını ve RHEL 6'nın `%ruby_sitearch` makro desteğini düzenler. Fedora 17 ve daha yeni sürümleri `%ruby_sitearch` makrosunu desteklerken RHEL6 desteklemez. Eğer çalışan işletim sistemi RHEL6 ise, `%ruby_sitearch` makrosunu tanımlar. Dikat edilmelidir ki `0%{?el6}` ile önceki örnekte gördüğümüz `0%{?rhel} == 6` ile aynıdır ve paketin RHEL 6 için derlenip derlenmediğini yoklar.

```
%if 0%{?fedora} >= 19
%global with_rubypick 1
%endif
```

Bu koşulda ise rubypick desteği yoklanır. Eğer işletim sistemi Fedora 19 veya daha yeniyse, rubypick destekleniyordur.

```
%define ruby_archive %{name}-%{ruby_version}
%if 0%{?milestone:1}%{?revision:1} != 0
%define ruby_archive %{ruby_archive}-
%{?milestone}%{?!milestone:%{?revision:r%{revision}}}
%endif
```

Bu koşulda da makro tanımları yoklanır. Eğer `%milestone` ve `%revision` makroları tanımlanmışsa tarball ismini belirten `%ruby_archive` makrosu tekrar tanımlanır.

`%if` Koşulunun Özel Türleri

`%ifarch`,`%ifnarch` ve `%ifos` koşulları `%if` koşulunun özelleştirilmiş türleridir. Bu türler sık sık kullanıldığı için kendi makroları vardır.

`%ifarch` koşulu

`%ifarch` koşulu genellikle belirli bir mimariye özgü kısmını tetiklemek için kullanılır. Hemen ardından bir veya birden çok mimari belirtir ve her biri virgüller ya da boşluklarla ayrılır.

```
%ifarch i386 sparc
...
%endif
```

`%ifarch` ve `endif` arasında kalan kısım yalnızca 32 bit AMD/Intel mimariler veya Sun SPARC temelli sistemlerde işlenir.

`%ifnarch` koşulu

`%ifnarch` koşulu ise `%ifarch` koşulunun tam tersidir.

```
%ifnarch alpha
...
%endif
```

`%ifnarch` ve `%endif` arasında kalan kısım Alfa işlemcilerini kullanan sistemler **dışında** bütün sistemlerde kullanılacaktır.

`%ifos` koşulu

`%ifos` koşulu da belirtilen işletim sistemi üzerinde uygulanacak işlemleri belirtir. Bir veya birden çok işletim sistemi belirtilebilir.

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

`%ifos` ve `%endif` arasında kalan kısım, eğer inşa bir Linux sistem üzerinde gerçekleştirilirse çalıştırılacak işlemleri içerir.

RHEL 7 ile Beraber Gelen Yeni RPM Özellikleri

Dokümanın bu kısmında Red Hat Enterprise Linux 6 ve 7. sürüm arasında dikkat edilmesi gereken farklar anlatılmaktadır.

- Anahtarlık ve imza denetimi için `rpmkeys` komutu eklendi.
- SPEC dosyalarının sorgulanması ve taranması için `rpmspec` komutu eklendi.
- Paketlerin imzalanması için `rpmsign` komutu eklendi.
- `%{lua:⋯}` içerisinde gömülmüş betiklerde çağrılan `posix.exec()` ve `os.exit()` eklentileri, `posix.fork()` betikçisinin altsüreci olarak çağırılmadıkları sürece hata vereceklerdir.
- `%pretrans` betikçi hatası, paket kurulumunun atlanmasına neden olur.
- Betikçiler, çalışma zamanında makro ve sorgu biçimi (queryformat) şeklinde genişletilebilir.
- İşlem öncesi ve işlem sonraki betikçilerinin bağımlılıkları artık `Requires(pretrans)` ve `Requires(posttrans)` betikçileriyle ifade edilebilir.
- Fazladan sıralama ipucularını ifade etmek için `OrderWithRequires` etiketi eklendi. Bu etiket, `Requires` etiketinin sözdizimini taklit eder ancak gerçek bağımlılıklar oluşturmaz. Sıralama ipuçlarında belirtilenler eğer işlem esnasında varsa `Requires`'da belirtilmiş gibi ele alınırlar.
- `%license` bayrağı `%files` kısmında kullanılabilirler. Tıpkı lisans dosyalarını işaretlerken tıpkı `%doc` gibi kullanılırlar, ancak `-nodocs` seçeneğinden etkilenmezler.
- Otomatik yama eklemek için kullanılan `%autosetup` makrosu, dilenirse bir versiyon kontrol sistemiyle beraber kullanılabilirler.
- Otomatik bağımlılık oluşturucusu gömülü bir filtrelemeyle beraber genişletilebilir ve özelleştirilebilir bir şekilde yeniden yazıldı.
- OpenPGP V3 genel anahtarları artık desteklenmeyecektir.

Referanslar

Aşağıda RPM paketlemek, inşa etmek ve RPM paketleri hakkında konuları ele alan bazı referansları görebilirsiniz. Bazıları, bu rehberde bahsedilenlerden çok daha ileri düzeyde konuları içerir.