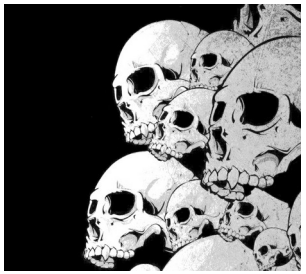
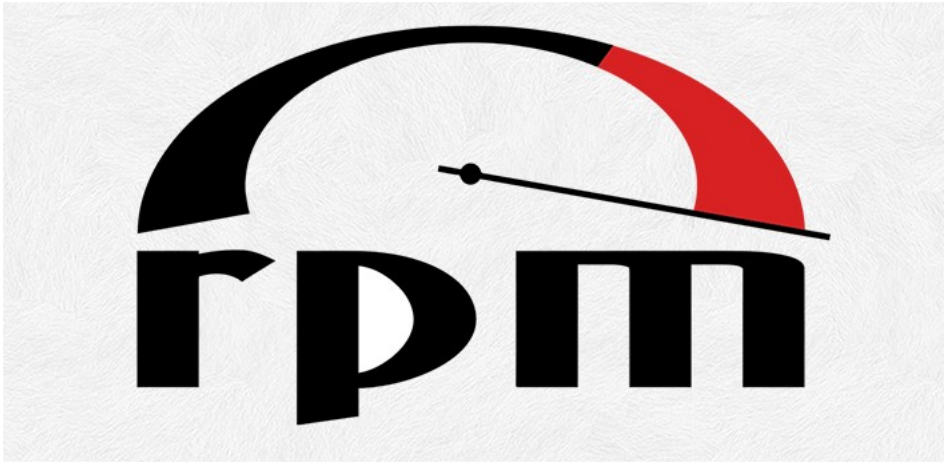
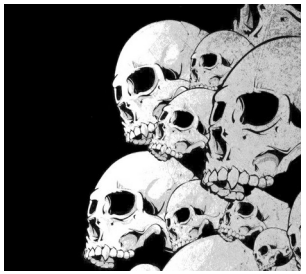


Packaging RPM



Introduction au packaging RPM

Y. Collette



Objectifs

L’objectif du packaging :

Préserver le code source du paquet original + patches correctifs pour assurer le build

Masquer les différentes étapes du build du paquet et des dépendances

Référencer les différentes modifications

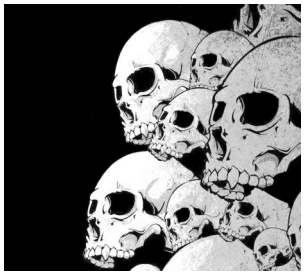
Faire des builds pour plusieurs architectures

Tout cela, encapsulé dans un simple fichier SPEC

Automatisation, répétabilité, documentation du build

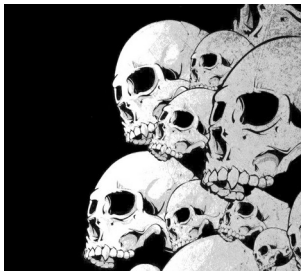
Définition :

Un container installable OS-spécifique ou distro-spécifique pour un software



Objectifs

Standardiser le déploiement	→	Savoir ce qu'on a installé
Simplifier l'environnement	→	Une seule source de logiciels
Conformité aux normes	→	Uniformisation de l'installation
Gestion des risques	→	Des mises à jour centralisées
Reproductible	→	Fiabilité des builds



DNF / YUM

Le problème de RPM: **La résolution des dépendences**

Les dépendances de paquets fonts que RPM est pratique, mais aussi compliqué

RHEL 5+ / Fedora utilisent YUM / DNF pour atténuer la peine

Les metadata sont générées à partir de l'arbre des paquets RPM

YUM / DNF utilisent les métadonnées pour résoudre les dépendances

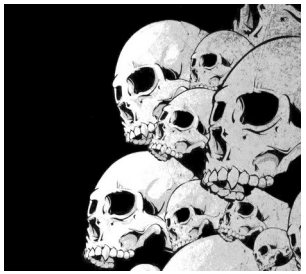
Red Hat Network utilisent YUM / DNF (RHEL 5 / RHEL 8)

Support des plugins, historique, rollbacks

Utilisé pour activer le preupgrade, anaconda

En cours de remplacement par DNF avec satsolver

yum
yellowdog updater modified



YUM / DNF
Qu'est ce que dnf ?

dnf est l'outil de gestion des paquets dans Fedora au delà de la version 22
dnf est utilisé dans un terminal et sert à installer, désinstaller des logiciels; mais aussi pour mettre à jour la distribution.

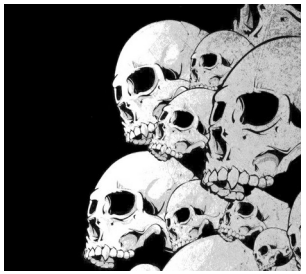
dnf a succédé à y**um**. DNF a été créé en 2015.

RPM est un logiciel libre créé à l'origine par Red Hat en 1995 pour la gestion de l'installation d'applications.
A l'origine : RPM → Redhat Package Manager → RPM Package Manager

YUM, pour Yellowdog Updater Modified est un gestionnaire de paquets RPM créé par YellowDog (qui à l'origine s'appelait YUP et avait été crée en 1999 / 2001)

Pour installer un paquet :

```
$ rpm --i package-name.rpm
$ dnf install package-name.rpm
$ yum install package-name.rpm
```



YUM / DNF
Qu'est ce que dnf ?

Liste un des paquets; le premier exemple liste tous les paquets présents dans le dépôt.

```
$ dnf list all
$ dnf list <package-name>
```

Liste les mises à jour du dépôt.

```
$ dnf check-update
$ dnf check-update kernel
```

La recherche d'un paquet peut se faire sur le nom du paquet ou sur une expression régulière.

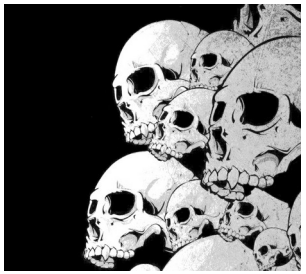
```
$ dnf search <package-name>
```

Liste tous les dépôts activés sur le système.

```
$ dnf repolist all
```

L'option `--recent` liste les paquets récemment ajoutés au dépôt. Les autres options sont `--extras`, `--upgrades`, et `–obsoletes`.

```
$ dnf list --recent
$ dnf list --recent <package-name>
```



YUM / DNF
Qu'est ce que dnf ?

Liste tous les avis diffusés dans le dépôt; l'ajout de l'option `sec` va lister les avis qui ont le label "security fix."

```
$ dnf updateinfo list available
$ dnf updateinfo list available sec
```

Liste tous les avis de sécurité diffusés dans le dépôt et marqués "critical".

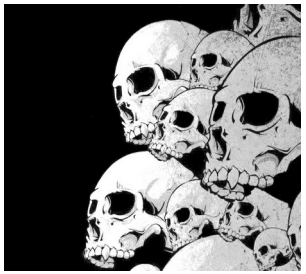
```
$ dnf updateinfo list available sec --sec-severity Critical
```

Vérifie l'information d'un avis via l'option `--info`.

```
$ dnf updateinfo FEDORA-2018-a86100a264 --info
```

Applique tous les avis de sécurité disponibles dans le dépôt. Avec l'option `--sec-severity`, vous pouvez include les paquets ayant une sévérité marquée `Critical`, `Important`, `Moderate`, ou `Low`.

```
$ dnf upgrade --security
$ dnf upgrade --sec-severity Critical
```



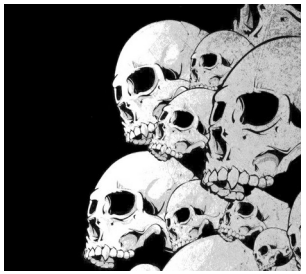
YUM / DNF
Qu'est ce que dnf ?

Pour installer / supprimer un paquet:

```
$ dnf install <package-name>
$ dnf remove <package-name>
```

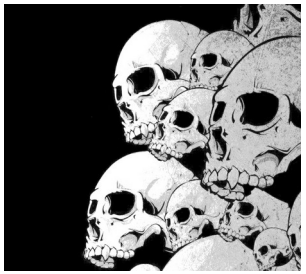
Pour supprimer un paquet et toutes les dépendances non utilisées :

```
$ dnf autoremove <package-name>
```



Premier exemple
Le fichier SPEC

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

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

%prep
# we have no source, so nothing here

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

%install
mkdir -p %{buildroot}/%{_bindir}/
install -m 755 hello-world.sh %{buildroot}/%{_bindir}/hello-world.sh

%files
%{_bindir}/hello-world.sh

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



Premier exemple
Structure du fichier SPEC

Préparation - %prep

%prep
%autosetup -q

Compilation - %build

%build
%configure
%make_build

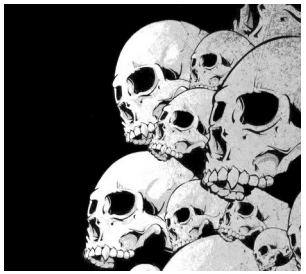
Installation - %install

%make_install

Optionnel - %check

Test de non régression.

Exemple :
make test



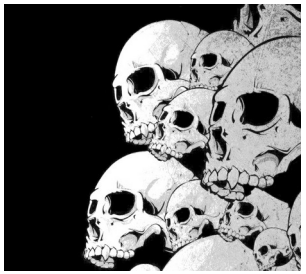
Premier exemple
Structure du fichier SPEC

%files
%doc
%license

À retenir :

%{_libdir}	→ bibliothèques (/usr/lib ou /usr/lib64)
%{_lib}	→ lib ou lib64
%{_bindir}	→ exécutables (/usr/bin)
%{_sysconfdir}	→ fichiers de configuration (/etc)
%{_datadir}	→ /usr/share
%{_localstatedir}	→ /var

%find_lang et %files -f %{name}.lang
%exclude
%ghost
%config → identifie un fichier comme fichier de configuration, permet de gérer le traitement des sauvegardes (.rpmsave et .rpmnew)



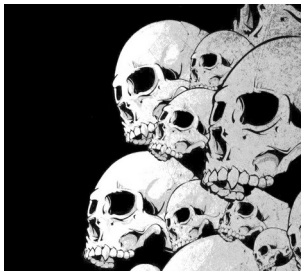
Premier exemple
Structure du fichier SPEC

%pre
Cette section contient tout ce qui doit être fait **avant l'installation** du paquetage

%post
Cette section contient tout ce qui doit être fait **après l'installation** du paquetage

%preun
Cette section contient tout ce qui doit être fait **avant** la **désinstallation** du paquetage

%postun
Cette section contient tout ce qui doit être fait **après** la **désinstallation** du paquetage



ChangeLog

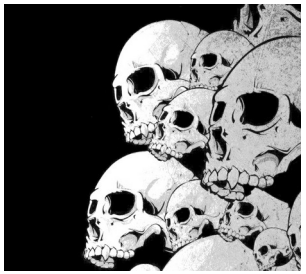
La section %changelog contient les informations sur les modifications apportées au paquet.

Le numéro de version d'un paquet :
A.B.C-D
A : version majeure
B : version mineure
C : version patche
D : version du fichier SPEC

Pour le %changelog, la structure d'une entrée est :
* date – email – version
- commentaire

Exemple :
* Sun Jun 10 2018 Yann Collette <ycollette.nospam@free.fr> - 4.16.12-rt5-2
- add 4.16.12-rt5 kernel

Pour obtenir la date :
$ LANG=C date



Premier exemple
Les commandes

$ dnf install rpmdevtools
$ dnf install rpm-build
$ dnf install rpmlint

$ rpmdev-setuptree
$ cd ~/rpmbuild/SPEC

$ spectool -g exemple.spec # permet de télécharger les sources

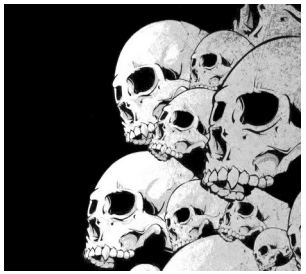
$ rpmlint exemple.spec
0 packages and 1 specfiles checked; 0 errors, 0 warnings

$ rpmbuild -ba hello-world.spec

rpmdev-setuptree : crée l'arborescence
~/rpmbuild/{SPEC,RPMS,SRPM,BUILD/SOURCE/BUILDROOT}

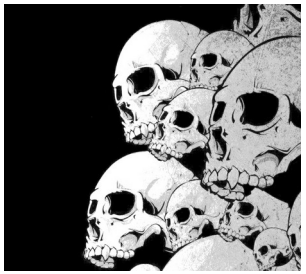
rpmlint : vérifie la syntaxe du fichier spec

rpmbuild : construit le paquet. Les fichiers rpm se trouveront dans :
RPMS/noarch, RPMS/x86_64, SRPMS



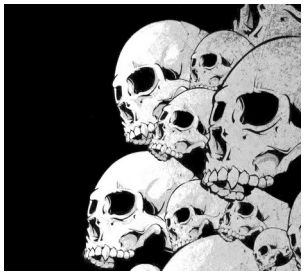
Premier exemple
Log 1/4

```
$ rpmbuild -ba exemple.spec
Exécution_de(%prep) : /bin/sh -e /var/tmp/rpm-tmp.iclYoJ
+ umask 022
+ cd /home/artelys/rpmbuild/BUILD
+ RPM_EC=0
++ jobs -p
+ exit 0
Exécution_de(%build) : /bin/sh -e /var/tmp/rpm-tmp.7GG1xI
+ umask 022
+ cd /home/artelys/rpmbuild/BUILD
+ cat
+ RPM_EC=0
++ jobs -p
+ exit 0
```



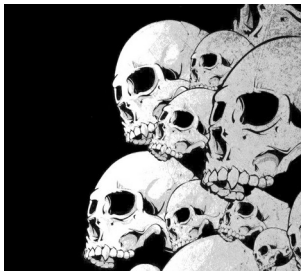
Premier exemple
Log 2/4

Exécution_de(%install) : /bin/sh -e /var/tmp/rpm-tmp.JfHAJK
+ umask 022
+ cd /home/artelys/rpmbuild/BUILD
+ '[' /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64 '!=' / ']'
+ rm -rf /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
++ dirname /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
+ mkdir -p /home/artelys/rpmbuild/BUILDROOT
+ mkdir /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
+ mkdir -p /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64/usr/bin/
+ install -m 755 hello-world.sh /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64/usr/bin/hello-world.sh



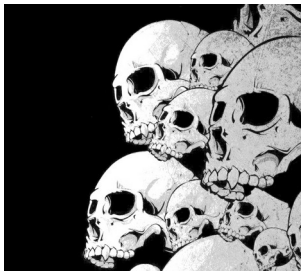
Premier exemple
Log 3/4

+ '[' '%{buildarch}' = noarch ']'
+ QA_CHECK_RPATHS=1
+ case "${QA_CHECK_RPATHS:-}" in
+ /usr/lib/rpm/**check-rpaths**
+ /usr/lib/rpm/**check-buildroot**
+ /usr/lib/rpm/redhat/**brp-ldconfig**
+ /usr/lib/rpm/**brp-compress**
+ /usr/lib/rpm/**brp-strip** /usr/bin/strip
+ /usr/lib/rpm/**brp-strip-comment-note** /usr/bin/strip /usr/bin/objdump
+ /usr/lib/rpm/**brp-strip-static-archive** /usr/bin/strip
+ /usr/lib/rpm/redhat/**brp-python-bytecompile** /usr/bin/python 1 0
+ /usr/lib/rpm/**brp-python-hardlink**
+ /usr/lib/rpm/redhat/**brp-mangle-shebangs**



Premier exemple
Log 4/4

Traitement des fichiers : hello-world-1-1.x86_64
Provides: hello-world = 1-1 hello-world(x86-64) = 1-1
Requires(rpmlib): rpmlib(CompressedFileNames) <= 3.0.4-1 rpmlib(FileDigests) <= 4.6.0-1 rpmlib(PayloadFilesHavePrefix) <= 4.0-1
Requires: /usr/bin/bash
Vérification des fichiers non empaquetés : /usr/lib/rpm/check-files
/home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
Écrit : /home/artelys/rpmbuild/SRPMS/hello-world-1-1.src.rpm
Écrit : /home/artelys/rpmbuild/RPMS/x86_64/hello-world-1-1.x86_64.rpm
Exécution_de(%clean) : /bin/sh -e /var/tmp/rpm-tmp.Kwk6qL
+ umask 022
+ cd /home/artelys/rpmbuild/BUILD
+ /usr/bin/rm -rf /home/artelys/rpmbuild/BUILDROOT/hello-world-1-1.x86_64
+ RPM_EC=0
++ jobs -p
+ exit 0



Second Example jm2cv

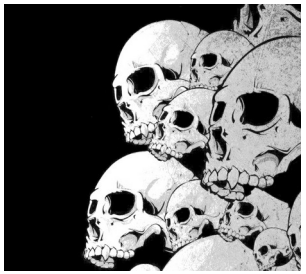
Nous allons package jm2cv (Jack Midi to Voltage Control).
Le lien de téléchargement de la bonne version sur github :

https://github.com/harryhaaren/jm2cv/archive/8bddbd13468b3d8497a9d8a19871293e3088f614.zip

Le site github :
https://github.com/harryhaaren/jm2cv

Ce programme utilise cmake pour la configuration de la compilation.
Il utilise aussi les paquets gcc, gcc-c++, jack-audio-connection-kit-devel.

Dans l’archive de ressources, il y a un fichier template.spec qui va servir de gabarit.



Builder étape par étape

Réglage des patches, des BR, ... (%prep)
`rpmbuild -bp <nomduspec>`

Compilation (%build)
`rpmbuild -bc --short-circuit <nomduspec>`

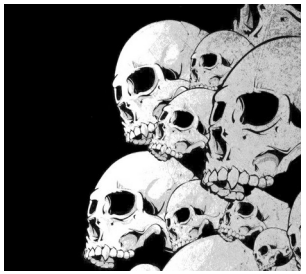
Empaquetage, contrôle des fichiers (%install et %files)
`rpmbuild -bi --short-circuit <nomduspec>`

Contrôle des fichiers (%files)
`rpmbuild -bl --short-circuit <nomduspec>`

Les RPM binaires (%prep, %build, %install...)
`rpmbuild -bb <nomduspec>`

Le RPM des sources (aucune compilation)
`rpmbuild -bs <nomduspec>`

Les RPM binaires et le RPM des sources
`rpmbuild -ba <nomduspec>`



Utiliser des macros consulter l’existant

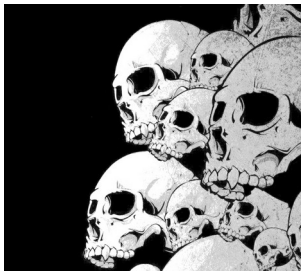
rpm –showrc :
Cette commande liste les macros disponibles

```
$ rpm --showrc | grep make_build
```
-13: make_build %{__make} %{_make_output_sync} %{?_smp_mflags} %{_make_verbose}

rpm –eval=’%{macroname}’ :
Cette commande permet d’évaluer le contenu d’une macro.

```
$ rpm --eval='%{make_build}'
```
/usr/bin/make -O -j4 V=1 VERBOSE=1

```
$ rpm --eval='%{?_smp_mflags}'
```
-j4



Utiliser des macros le fichier .rpmmacros

Un exemple de lignes l'on peut trouver dans un fichier ~/.rpmmacros

```
%_smp_mflags     -l3
```

Si un fichier SPEC a un "make %{?_smp_mflags}", alors la compilation se déroulera sur 3 CPUs.

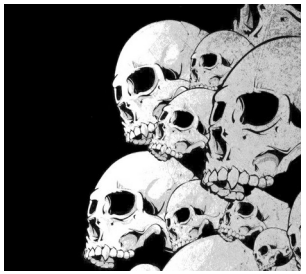
Pourquoi 3 ? Ce chiffre 3 dépend de votre CPU (s'il a 4 cœurs, 3, ça laisse un cœur pour effectuer d'autres tâches) et de la mémoire (lancer un build sur 4 cœurs va nécessiter plus de mémoire que sur 3).

%__arch_install_post   /usr/lib/rpm/check-rpaths /usr/lib/rpm/check-buildroot

Dans les paquets Fedora on trouve un paquet rpmdevtools qui fournit des macros rpm pour le développement.

check-rpaths : vérifiera qu'il n'y a pas de chemins codés en dur dans le code généré par le compilateur

check-buildroot : va vérifier que le chemin 'buildroot' n'apparaît pas dans les fichiers générés par le build



Patcher un paquet

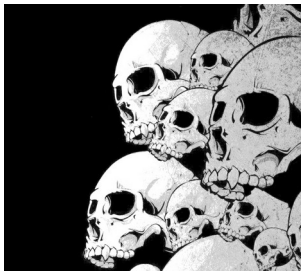
Tout se fait avec git.

```
$ tar xvfz mes-sources.tar.gz
$ cd mes-sources
$ git init .
$ git add .
$ git commit -m ‘initial commit’
```

Configure / Compiler / modifier

```
$ git add ‘modified files’
…
$ git commit -m ‘supress warnings’
$ git format-patch HEAD~1
$ mv 0001-suppress-warnings.patch ~/rpmbuild/SOURCE
```

On peut aussi effectuer ces opérations directement dans le répertoire : ~/rpmbuild/BUILD/mes-sources
Ce répertoire sera supprimé après le prochain appel à rpmbuild.



Patcher un paquet

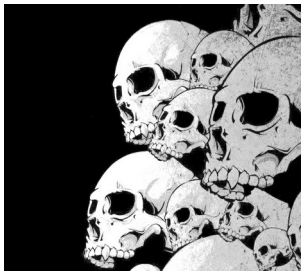
Ajouter les patches au paquet :

```
Patch0: enscribe_01-makefile.patch
Patch1: enscribe_02-FFTblocksizenorm.patch
Patch2: enscribe_03-fix-typo.patch
```

Et dans la section`%prep`, après la commande `%setup` :

```
%prep
%setup -q

%patch0 -p1
%patch1 -p1
%patch2 -p1
```

Aujourd'hui, il vaut mieux utiliser la macro %autosetup -q -p1 qui va charger et appliquer tous les patches sans avoir à utiliser de commande %patch<n>



SRPMS

Le paquet source → package-name-1.0.0-1.src.rpm

Le fichier SRPM contient les fichiers sources/composants/spec utilisés pour générer le paquet RPM binaire.

Installer un paquet SRPM avec rpm :

```
$ rpm -ivh package-name-1.0.0-1.src.rpm
```

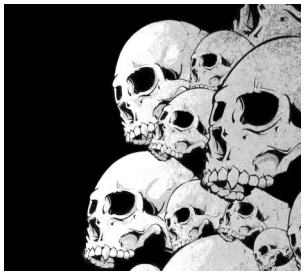
i pour install, **v** pour verbeux, **h** pour afficher la progression

La commande installe les fichiers dans la structure ~/rpmbuild/{SOURCES,SPEC}

Le fichier SRPM n'est pas directement installé. Il doit être rebuildé avant.

Pour supprimer un fichier SRPM installé :

```
$ rpmbuild --rmsource --rmspec package-name.spec
```



SRPMS

Construire un fichier RPM binaire à partir d'un fichier SRPM:

```
$ rpmbuild --rebuild package-name-1.0.0-1.src.rpm
```

Construire un fichier RPM binaire à partir d'un fichier spec (-b pour build, -a pour tous les paquets, source et binaire)

```
$ rpmbuild -ba package-name.spec
```

Construire les sources binaires patchés à partir d'une fichier spec (-b pour build, -p pour appliquer seulement les patches)

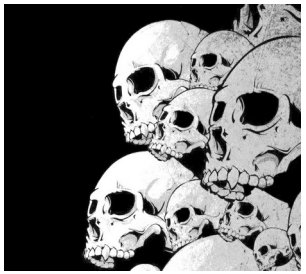
```
$ rpmbuild -bp package-name.spec
```

Les sources patchés vont dans le répertoire de build → Sous Red Hat, ce sera dans ~/rpmbuild/BUILD

Construire pour une architecture différente :

```
$ rpmbuild –target sparcv9 –rebuild package-name-1.0.0-1.src.rpm
```

Attention! Certains programmes détecte le type de système et configure le build sans prendre en compte l'option de rpm.



Multi-architectures

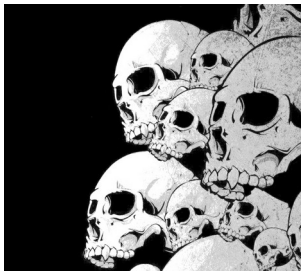
Actuellement, Fedora 32 supporte 4 architectures :

- aarch64
- armhfp
- i386
- x86_64

Comment faire pour gérer différents OS avec un seul fichier SPEC ?

En utilisant des macros et des conditions:

```
%if 0%{?rhel} == 6 || 0%{?fedora} < 17
Requires: ruby(abi) = 1.8
%else
Requires: ruby(release)
%endif
```



Les fichiers spec multi-paquets

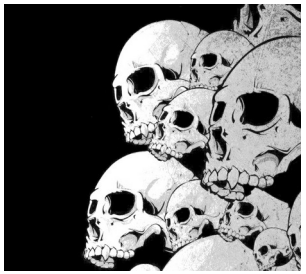
On prend l'exemple du paquet **libsmf**. La construction du fichier spec va générer un paquet **libsmf** (qui contient la librairie) et **libsmf-devel** (qui contient les headers).
Ce fichier est présent dans le répertoire Files de la formations.

Pour le paquet principal, on a :

```
%description
```

Et pour les paquets dépendants :

```
%package devel
Summary:  Development files for %{name}
Requires: %{name} = %{version}-%{release}

%description devel
The %{name}-devel package contains header files for %{name}.

%files devel
%{_includedir}/*
```



Mock
Tester le build du paquet

Mock est un outil pour construire des paquets. Il peut construire des paquets pour différentes architectures and différentes versions de Linux (Fedora, RHEL, et Mageia) que celle de l'environnement hôte. **Mock** crée un environnement chrooté et construit les paquets dedans. Sa seule tâche est d'installer des paquets dans un chroot et de builder des paquets dans ce chroot.

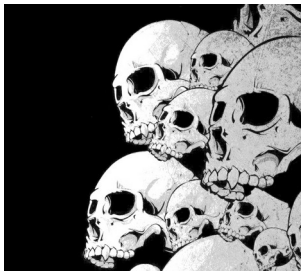
Mock offre aussi la commande multi-paquet (--chain), qui peut builder des chaînes de paquets qui dépendent les uns des autres.

Le principal avantage de construire des paquets via **mock** au lieu d'utiliser rpmbuild est que **mock** construit les paquets dans un environnement propre. **mock** fait cela en créant un environnement chrooté et en faisant le build dans cet environnement.

```
$ dnf install mock
$ sudo usermod -a -G mock joe
```

Les configurations **mock** se trouvent dans /etc/mock/.

```
$ mock -r epel-7-x86_64 –init
$ mock -r epel-7-x86_64 rebuild package-1.1-1.src.rpm
$ mock -r epel-7-x86_64 --clean
```



Mock
Tester l'appliquation

Une fois le build effectué, il est possible d'installer le paquet dans l'environnement chrooté de mock :

```
$ mock -r epel-7-x86_64 --init
$ mock -r epel-7-x86_64 --install package-name.rpm
```

Pour les applications graphique, il est nécessaire de donner les droits à l'environnement chrooté d'utiliser une application graphique :

```
$ xhost +
```

Puis il faut exporte l'interface graphique vers la machine hôte :

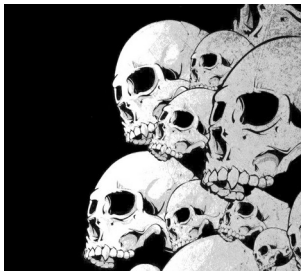
```
$ export DISPLAY=:0.0
```

Maintenant, il ne reste plus qu'à se connecter à l'environnement chrooté puis à tester l'application :

```
$ mock -r epel-7-x86_64 --shell
$ /usrbin/package-name
$ exit
```

Et éventuellement à supprimer l'environnement chrooté :

```
$ mock -r epel-7-x86_64 --clean
```



COPR-CLI

Copr est un serveur distant Fedora permettant de builder des packages et d'héberger des dépôts de paquets. Pour plus d'informations :
https://developer.fedoraproject.org/deployment/copr/copr-cli.html

```
$ sudo dnf install copr-cli
```

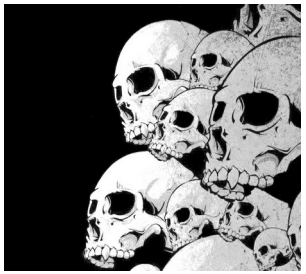
Ensuite, il faut se créer un compte / dépôt sur COPR :
https://copr.fedorainfracloud.org/

Il y a une clef API à installer et une fois cette opération effectuée, envoyer un build sur le serveur COPR est assez simple. Il faut dans un premier temps builder le paquet en local et ensuite, on envoie le fichier SRPM sur COPR :

```
$ copr-cli build test-project ~/packages/package-name.src.rpm
$ copr-cli build --chroot fedora-32-x86_64 --chroot epel-8-x86_64
test-project ~/packages/package-name.src.rpm
```

Pour utiliser le dépôt :

```
$ sudo dnf install dnf-plugins-core
$ sudo dnf copr enable your_name/test-project
```



fedora-review Usage

La commande fedora-review permet d'effectuer une revue de packaging complète.
La syntaxe :

```
$ fedora-review --verbose --mock-config fedora-32-x86_64 --rpm-spec --name <nom paquet>
```

Cette commande est à utiliser dans le répertoire rpmbuild/SRPMS.
Il faut qu'il n'y ait qu'un seul paquet <nom paquet>.src.rpm. S'il y a deux paquets avec le même nom (mais des numéros de version différents) la commande échouera.

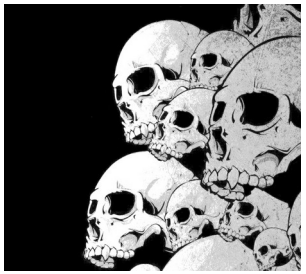
Cette commande peut être appliquée sur différentes sources :

-b <bug>, --bug <bug> : va chercher les informations dans un bug report bugzilla.

-n <name>, --name <name> : utilise un fichier local <name>.spec et / ou <name>*.src.rpm dans le répertoire courrant ou, avec le flag --rpm-spec, utilise <name> comme chemin vers le srpm.

-u <url>, --url <url> : utilise un autre bugzilla comme source.

--copr-build <build_descriptor> : utilise un build COPR



fedora-review
Le résultat

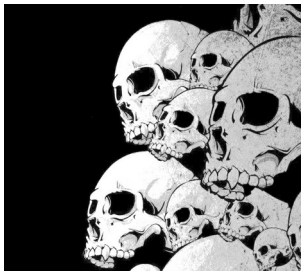
BUILD -> /var/lib/mock/fedora-32-x86_64/root/builddir/build/BUILD
build.log
dependencies
files.dir
licensecheck.txt
report.xml
results
review-env.sh
review.txt
rpmlint.txt
rpms-unpacked
srpm
srpm-unpacked
upstream
upstream-unpacked

Vérification des licences de tous les fichiers

Les paquets générés par mock

Le rapport de la revue de code

Le résultat de la commande rpmlint



Signer un paquet

Générer la clef :

```
$ gpg --gen-key
```

Vérifier que la clef a bien été créée :

```
$ gpg --list-keys
```

Exporter la clef public à partir de votre ring vers un fichier texte.

```
$ gpg --export --armor ycollette.nospam@free.fr > RGK-ycollette
```

Importer votre clef public dans la base de données RPM DB :

```
$ sudo rpm --import RGK-ycollette
```

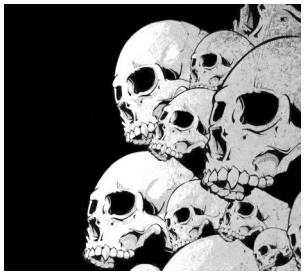
Vérifier la liste des clefs gpg publics dans la base de données RPM DB :

```
$ rpm -q gpg-pubkey --qf '%{name}-%{version}-%{release} --> %
{summary}\n'
```

Configurer votre fichier ~/.rpmmacros :

%_signature → On mettra toujours gpg
%_gpg_path → Mettre le chemin complet vers le répertoire .gnupg
%_gpg_name → Utilisez votre vrai nom utilisé pour créer la clef
%__gpg → exécutez `which gpg` (sans les marques `) pour récupérer le chemin complet

```
%_signature gpg
%_gpg_path /home/ycollette/.gnupg
%_gpg_name Yann Collette
%__gpg /usr/bin/gpg
```



Signer un paquet

Installer rpmsign :

```
$ sudo dnf install rpm-sign
```

Il est possible alors de signer des paquets individuellement:

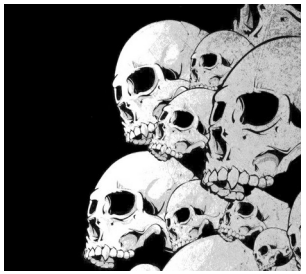
```
$ rpm --addsign package-name-1.0.0-1.el8.x86_64.rpm
```

Vérifier la signature d’un paquet :

```
$ rpm --checksig package-name-1.0.0-1.el8.x86_64.rpm
```

Signer un paquet durant le build :

```
$ rpmbuild -ba --sign package-name.spec
```



Créer un dépôt local

Installer le paquet createrepo puis générer la structure du dépôt :

```
$ dnf install createrepo
$ mkdir /opt/local-repo
$ createrepo /opt/local-repo
```

Créer un fichier de configuration du dépôt /etc/yum.repos.d/local.repo contenant :

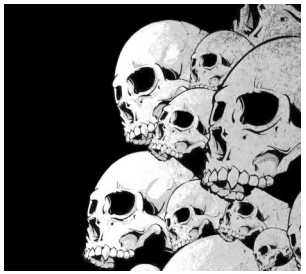
```
[local-repo]
name=Local Repo
baseurl=file:///opt/local-repo
enabled=1
gpgcheck=0
```

Copier les nouveaux paquets dans le dépôt local :

```
$ sudo cp ~/rpmbuild/RPMS/x86_64/mypackage.rpm /opt/local-repo/
$ …
```

Mettre à jour le dépôt :

```
$ createrepo --update /opt/local-repo
$ dnf makecache
```



Créer un dépôt local

Vérifier les paquets contenus dans le nouveau dépôt :

```
$ dnf --refresh repository-packages local-repo list
```

Pour avoir ce dépôt accessible à l'extérieure, il faudra configurer un serveur HTTP/HTTPS. On peut aussi installer repoview pour avoir une interface web navigable :

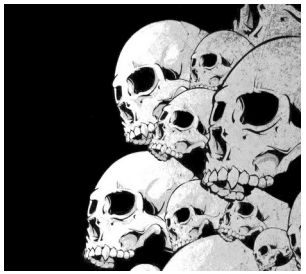
```
$ dnf install repoview
```

La commande repoview permet de regénérer des pages statiques HTML pour le dépôt.

```
$ repoview /opt/local-repo
```

Attention, ce paquet a été définitivement supprimé à partir de Fedora 31. Chaque nouveau paquet entrainait la génération de pas mal de pages HTML qui étaient ensuite synchronisées par pas mal de miroir. Et à cause de cette engorgement, repoview a été abandonné.

L'autre alternative consiste à configurer nginx pour générer automatiquement les fichiers index en utilisant l'arborescence des fichiers du dépôt.

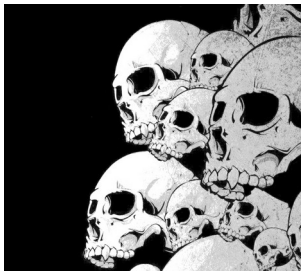


Créer un dépôt local

Le fichier de conf nginx /etc/nginx/conf.d/local-repo.conf :

```
server {
    listen       8081;          # a custom port

    # download
    autoindex on;               # enable directory listing output
    autoindex_exact_size off; # output file sizes rounded to
                              # kilobytes, megabytes,
                              # and gigabytes
    autoindex_localtime on;   # output local times in the
                              # directory
    location / {
        root /opt/local-repo/;
    }
}
```

Attention : sous Fedora, il va falloir gérer les autorisations SELinux.



Créer un dépôt local

Si la commande `getenforce` retourne « Enforcing » alors SELinux est activé.
Autoriser le port 8081 à être utilisé par nginx dans SELinux :

```
$ ausearch -c "nginx" --raw | audit2allow -M my-nginx
$ semodule -i my-nginx.pp
$ systemctl stop nginx.service
$ systemctl start nginx.service
```

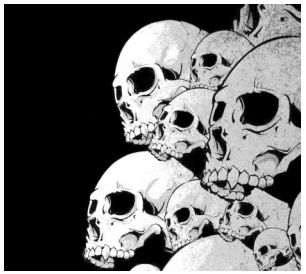
Et il est maintenant possible de visualiser les différents fichiers contenus dans le dépôt.

Pour utiliser le serveur nginx comme dépôt, il faut modifier /etc/yum.repos.d/local.repo :

```
[local-repo]
name=Local Repo
baseurl=http://localhost:8081
enabled=1
gpgcheck=0
```

Puis on test :

```
$ dnf makecache
$ dnf --refresh repository-packages local list
$ firefox http://locahost:8081
```



Ressources

IBM DeveloperWorks – good generic docs
http://docs.fedoraproject.org/drafts/rpm-guide-en/
http://www.vim.org/scripts/script.php?script_id=98

Fedora Packaging Guidelines:
https://fedoraproject.org/wiki/Packaging:Guidelines
https://fedoraproject.org/wiki/Packaging:ReviewGuidelines
https://fedoraproject.org/wiki/Licensing#Software_License_List

Maximum RPM:
http://www.rpm.org/max-rpm-snapshot/

Fedora GIT Tree (contains lots of example specs)
https://src.fedoraproject.org/

Fedora packaging mailing list
https://admin.fedoraproject.org/mailman/listinfo/packaging

Rpmlint website:
http://rpmlint.zarb.org/cgi-bin/trac.cgi