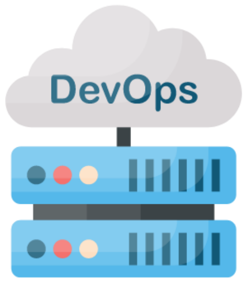
iNeuron

DevOps
Interview Questions

PART -5: Installing Software



Question 47: Red Hat Package manager

What is RPM?

A: RPM stands for Red Hat Package Manager. However, these days RPM isn't only Red Hat specific because many other Linux distros use RPM for managing their software. For example, both Mandriva and SuSE use RPM for software management. With RPM, you can install, upgrade and uninstall software on Linux, as well as keep track of already installed RPM packages on your system. This can be done because RPM keeps a database of all software that was installed with it.

RPM uses software packages that have the .rpm extension. An RPM package contains the actual software that gets installed, maybe some additional files for the software, information on where the software and its files get installed and a list of other files you need to have on your system in order to run this specific piece of software.

When you use RPM for installing the software package, RPM checks if your system is suitable for the software the RPM package contains, figures out where to install the files the package provides, installs them on your system and adds that piece of software into its database of installed RPM packages.

Different Linux distros may keep their software and the files related to that software in different directories. That's why it's important to use the RPM package that was made for your distribution. For example, if you install a SuSE specific software package on a Red Hat system, RPM may put the files from that package into wrong directories. In the worst case the result is that the program doesn't find all the files it needs and doesn't work properly.

Note that you need to be root when installing software in Linux. When you've got the root privileges, you use the rpm command with appropriate options to manage your RPM software packages.

Question 48: Installing and upgrading RPM packages

What is the proper procedure tin install and upgrade RPM packages?

A: Use the rpm command with -i option (which stands for "install") for installing a software package. For example, to install an RPM package called software-2.3.4.rpm:

```
# rpm -i software-2.3.4.rpm
```

If you already have some version installed on your system and want to upgrade it to the new version, use -U option (which stands for "upgrade"). For example, if you have software-2.3.3.rpm installed and wants to upgrade it:

```
# rpm -U software-2.3.4.rpm
```

If nothing goes wrong, the files in your package will get installed into your system and you can run your new program. Note that rpm doesn't usually create a special directory for the software package's files. Instead, the different files from the package get placed into appropriate existing directories on your Linux system. Executable programs go usually into *bin*, usr*bin*, usr/X11/bin, or *usr*X11R6/bin after installing with rpm.

Sometimes the program gets automatically added into your menu but usually you can just run the program by typing its name at the command prompt. In most cases you don't have to know where the program was installed because you don't have to type the whole path when running the program, only the program's name is needed.

Question 49: Error: failed dependencies

The RPM stopped installing and I’m

getting a dependency error. How do I

resume installation?

A: Many Linux programs need other files or programs in order to work properly. A piece of software depends on other software. When you try to install an RPM package, RPM automatically checks its database for other files that the software being installed needs. If RPM can't find those files in its database, it stops installing the software and complains about failed dependencies.

RPM gives out a list of files the program needs when you get a dependency error. The files in the list are probably ones you don't have on your system or files that you have but are the wrong versions. You will have to find the files RPM complains about, install or upgrade those files first and then try to install the package you were installing.

If the needed files are there and still get a failed dependency error, use the -- nodeps option. This tells RPM not to check any dependencies before installing the package:

```
# rpm -i software-2.3.4.rpm --nodeps
```

This forces RPM to ignore dependency errors and install software anyway but if the needed files are not there, the program won't work well or won't work at all. Use the --nodeps option only when you know that the needed files are there.

Question 50: Removing software installed with RPM

What is the correct method to remove software installed with RPM?

A: To remove software that was installed with RPM, use the -e option (erase): **`# rpm -e software-2.3.4.`**

You don't have to type the whole name of the package that contained the software and you don't have to type the .rpm extension when removing software. Probably, you don't have to type the version number either, so this would do exactly the same as the -e option: # rpm -e software

This rpm -e command uses the RPM database to check where all the files related to this software were installed and then automatically removes all of those files. After removing the program files, it also removes the program from the database of installed software.

It is very important that you never remove RPM software manually (for example, deleting single files with rm). If you just run around your system randomly deleting files that were installed with RPM, you'll get rid of the software, but RPM doesn't know it and doesn't remove the software package from its database. The result is that RPM still thinks the program is installed on your system and you may run into dependency problems later.

If you used RPM for installing a certain piece of software, also use RPM for removing that piece of software.

Question 51: Querying the RPM database

How can I the list of packages in the RPM database?

A: You can query the RPM database to get info of the packages on your Linux system. To query a single package, use the -q option. For example, to query a package whose name is "software": **`# rpm -q software`**

After issuing this command, rpm either tells you the version of the package or that the package isn't installed.

If you want a list of all packages installed on your system, you'll have to query all with -qa: **`# rpm -qa`**

If you are given a long list, you’ll need a way to scroll it. The best way is to pipe the list to less: **`# rpm -qa | less`**

If you're looking for packages whose names contain a specific word, you can use grep for finding those packages. For example, to get a list of all installed RPM packages whose names contain the word "kde", do something like this:

```
# rpm - qa | grep kde
```

The above command makes rpm list all packages in its database and pass the list to grep. Then grep checks every line for "kde" and finally shows you all the lines that contain the word "kde".

Question 52: Installing software from source in Linux

Is it easy to compile and install software from source in Linux?

A: Yes, it is. Compiling and installing software from source in Linux isn't as hard as it may sound.

The installation procedure for software that comes in tar.gz and tar.bz2 packages isn't always the same but usually it's like this:

```
# tar xvzf package.tar.gz (or tar
xvjf package.tar.bz2) # cd
package
#
./config
ure #
make
# make install
```

By issuing these simple commands you can unpack, configure, compile, and install the software package and you don't even have to know what you're doing. However, it’s best to have a clear understanding of the installation procedure and see what these steps stand for.

Step 1. Unpacking:
The package containing the source code of the program has a tar.gz or a tar.bz2 extension. This means that the package is a compressed tar archive also known as a tarball. When making the package, the source code and the other needed files were piled together in a single tar archive, hence the tar extension. After piling them all together in the tar archive, the archive was compressed with gzip, thus the gz extension.

Some people want to compress the tar archive with bzip2 instead of gzip. In these cases, the package has a tar.bz2 extension. You can install these packages exactly the same way as tar.gz packages but use a bit different command when unpacking.

It doesn't matter where you put the tarballs you download from the internet, but I suggest creating a special directory for downloaded tarballs. You can put your downloaded tar.gz or tar.bz2 software packages into any directory you want. In this example I assume your username is Larry and you've downloaded a package called pkg.tar.gz into the dls directory you've created (*home*larry/dls).

After downloading the package, you

unpack it with this command:

```
me@puter: ~/dls$ tar xvzf pkg.tar.gz
```

As you can see, you use the tar command with the appropriate options (xvzf) for unpacking the tarball. If you have a package with tar.bz2 extension instead, you must tell tar that this isn't a gzipped tar archive. You do so by using the j option instead of z like this:

```
me@puter: ~/dls$ tar xvjf pkg.tar.bz2
```

What happens after unpacking depends on the package. In most cases, a directory with the package's name is created. The newly created directory goes under the directory where you are right now. To be sure, you can give the ls command:

```
larry@puter:
~/dls$ ls pkg
pkg.tar.gz
larry@puter:
~/dls$
```

In our example unpacking our package pkg.tar.gz did what is expected and created a directory with the package's name. Now you must cd into that newly created directory:

```
larry@puter: ~/dls$
cd pkg
larry@puter:
~/dls/pkg$
```

It is very important to read any documentation you find in this directory like README or INSTALL files before continuing.

Step 2. Configuring:

After you've changed into the package's directory, it's time to configure the package. Usually but not always, it's done by running this configure script: larry@puter: ~/dls/pkg$ ./configure

Issuing the configure script doesn’t compile anything yet but checks your system and assigns values for system-dependent variables. These values are used for generating a Makefile. The Makefile in turn is used for generating the actual binary.

If configure finds an error, it complains about it and exits. However, if everything works like it should, configure doesn't complain about anything. If configure exited without errors, then it's time to move on to the next step.

Step 3. Building:
It's finally time to actually build the binary, the executable program, from the source code.
This is done by running the make command: larry@puter:

```
~/dls/pkg$ make
```

Note that make needs the Makefile for building the program. Otherwise it doesn't know what to do. This is why it's so important to run the configure script successfully or generate the Makefile in some other way.

When you run make, you'll see again a bunch of strange messages filling your screen. This is also perfectly normal and nothing you should worry about. This step may take some time, depending on how big the program is and how fast your computer is.

If all goes as it should, your executable is finished and ready to run after making has done its job. Now, the final step is to install the program.

Step 4: Installing

Now it's time to install the program. When doing this you must be root. If you've done things as a normal user, you can become root with the su command. It'll ask you the root password and then you're ready for the final step.

```
larry@puter: ~/dls/pkg$ su
```

```
Password:
root@puter: homelarry/dls/pkg#
```

Now when your root, you can install the program with the make install command:

```
root@puter: homelarry/dls/pkg# make install
```

Again, you'll get some weird messages scrolling on the screen. After it has stopped, you have successfully installed the software.

In this example, we didn't change the behavior of the configure script, the program was installed in the default place. In many cases it's *usr*local/bin. If *usr*local/bin (or whatever place your program was installed in) is already in your PATH, you can just run the program by typing its name.

If you became root with su, you'd better get back your normal user privileges before you do something else. Type exit to become a normal user again:

root@puter: *home*me/dls/pkg# exit

```
exit
larry@puter: ~/dls/pkg$
```

Question 53: Cleaning up the mess

How do I get rid of unwanted files after installing software from source in Linux?

A: When you ran “make”, it created all sorts of files that were needed during the build process but are useless now and are just taking up disk space. This is why you'll want to make clean:

```
me@puter: ~/dls/pkg$ make clean
```

However, make sure you keep your Makefile. It's needed if you later decide to uninstall the program and want to do it as less troublesome as possible.

Question 54: Uninstalling

I want to uninstall the programs I compiled by myself, what is the method for doing this?

A: Uninstalling the programs you've compiled yourself is not as easy as uninstalling programs you've installed with a package manager, like rpm.

The first step is to read the documentation that came with your software package and see if it says anything about uninstalling. If you didn't delete your Makefile, you may be able to remove the program by doing a make uninstall:

```
root@puter: homeme/dls/pkg# make uninstall
```

If you see weird text scrolling on your screen that's a good sign. If “make” starts complaining, that's a bad sign. Then you'll have to remove the program files manually.

If you know where the program was installed, you'll have to manually delete the installed files or the directory where your program is. If you have no idea where all the files are, you'll have to read the Makefile and see where all the files got installed and then delete them.

Source: From Internet

*********** THANK YOU ***********