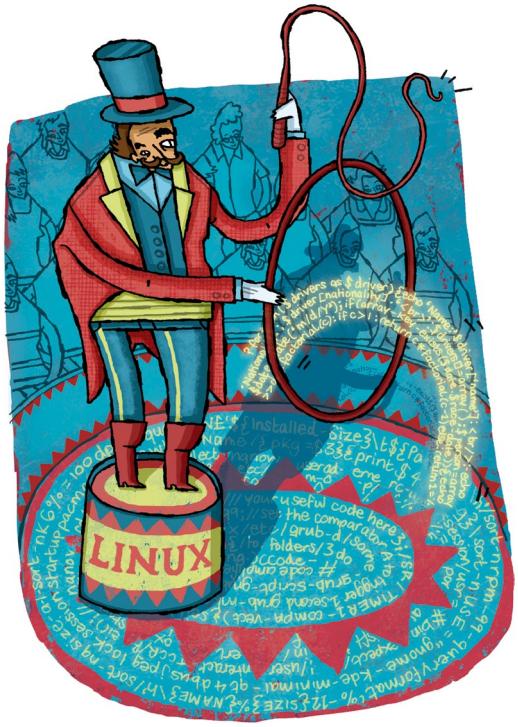
FPM: Building

Building packages has never been an easy job. Until now that is, thanks to FPM. **Mike Saunders** explains the change.

Our expert

Mike Saunders spends ages compiling things for the HotPicks section, so he hopes this program will encourage app developers to package up their work properly.



When it works, Linux package management is a marvellous thing to behold. Dependencies are resolved automatically, everything can be installed and easily removed with a few commands, and you can quickly find out which files belong to which packages. Sometimes it's not so pretty, when developers split up programs into 50 packages or you try to install an older package on a newer distribution release, but compared to the Windows world, where there are umpteen different installers in use and programs leave all sorts of crud in your Registry – well, it's world's apart.

```
Terminal - mike@mike-K52F: /tmp/packagedir
File  Edit  View  Terminal  Tabs  Help
mike@mike-K52F:~/Downloads/dateutils-0.2.5$ cd /tmp/packagedir/
mike@mike-K52F:/tmp/packagedir$ du
3.3M    ./usr/bin
8.0K    ./usr/lib/pkgconfig
1.1M    ./usr/lib
116K    ./usr/include/dateutils
120K    ./usr/include
16K     ./usr/share/doc/dateutils
20K     ./usr/share/doc
56K     ./usr/share/info
76K     ./usr/share/man/man1
80K     ./usr/share/man
160K    ./usr/share
4.7M    ./usr
4.7M    .
mike@mike-K52F:/tmp/packagedir$ find .
.
./usr
./usr/bin
./usr/bin/dround
./usr/bin/dgrep
./usr/bin/dtest
./usr/bin/strptime
./usr/bin/dadd
```

» Here, *Dateutils* hasn't been installed system-wide, but in a temporary directory instead.

Now, we all appreciate the work that distro developers do to package up software. If you've ever tried to make a Deb or *RPM* file by yourself, you might have run away screaming after a couple of hours of graft. It's not an especially simple process, and even when you've grokked it fully, it can be very time-consuming. *FPM* **(http://github.com/jordansissel/fpm)** aims to solve this, and its fundamental principle is: "If *FPM* is not helping you to make packages easily, then there is a bug in *FPM*." Great, but why would you want to make your own packages in the first place?

» You're writing your own program, you want to distribute it online and you want to give users something easier to install than the source code.

» You're an admin, you've built a custom version of some software, and you want to roll it out easily across other boxes.

» You just want to learn more about how packages work, boost your geek credentials and have something to natter about at your next LUG meeting.

Whatever the case, read on and we'll discover how *FPM* makes the job of creating and modifying packages much easier than you might expect.

First steps with FPM

FPM is written in Ruby, so first of all you'll need to get the latest version of the programming language, along with its development files. On a X/K/Ubuntu 13.04 box it's as simple as this:

```
sudo apt-get install ruby1.9.1 ruby1.9.1-dev
```

Almost every major distro has Ruby in its repositories though, so if you're running a different distro just search for it in your package manager. *FPM* is available as a Gem – that is, a Ruby package – so install it like this:

```
sudo gem install fpm
```

You'll see lots of output whizz by as RubyGems grabs and builds various dependencies. Once the process has finished, you're ready to use *FPM*.

Now, for *FPM* to do its job properly, it needs a bunch of files that it can wrap up into a package. These can be any kinds of files – after all, packages can contain executables, configuration files or images – so *FPM* doesn't ask for specifics. What it does need is a location containing the files, with the appropriate directory structure.

Let's look at the second scenario mentioned in the introduction: customising a program and making a package from it. In this case, we're going to use a simple program that's easy to build, *Dateutils* (**www.fresse.org/dateutils**) as covered in this month's HotPicks (*See p63*). Download **dateutils-0.2.5.tar.xz** and extract it like so:

```
tar xfv dateutils-0.2.5.tar.xz
```

At this stage you could make any customisations that you need, but for now we'll simply jump into the resulting directory and compile the program. Note that binary

packages

packages typically install into the **/usr** directory, as opposed to **/usr/local/**. This is just a convention, and it's not massively important. So we use the **--prefix=** option for the configure script:

```
cd dateutils-0.2.5/
./configure --prefix=/usr
make
```

From here, the usual command to install the program would be something like *sudo make install*. However, we don't want to scatter the files around the filesystem now; instead, we want to place them in a separate and distinct directory so that *FPM* can find them easily and bundle them into a package. This is possible with:

```
mkdir /tmp/packagedir
make install DESTDIR=/tmp/packagedir
```

If you look inside **/tmp/packagedir**, you'll see all of the files for a successful *Dateutils* installation, as per the first screenshot. The **DESTDIR** part should always work with programs that follow the usual *./configure*, *make* and *make install* procedure, but with other build systems you should check the documentation to find out how to install files into a temporary directory.

Creating a Deb package

So now we come to the big moment: using *FPM* to convert this directory into a package. The command you need is:

```
fpm -s dir -t deb -n dateutils -v 0.2.5 -C /tmp/packagedir/ .
```

Let's go through this carefully bit-by-bit. The first option, **-s**, tells FPM what we want to use as the source for the package; in this case it's a directory. *FPM* can use other sources as well, as we'll explore later.

Next is the **-t** option which describes the type of package that we want to create (a Deb in our case, but you can use **-t rpm** to build RPMs providing you have the relevant software installed (*see the Building RPMs box on page 72*). The **-n** switch provides the name for the package, while **-v** specifies the version. Finally, the **-C** part tells *FPM* to change into the specified directory before searching for files, and the '**.**' says that it should search from the base of the directory that its been switched-to.

Once this command has completed, you'll see something like this displayed:

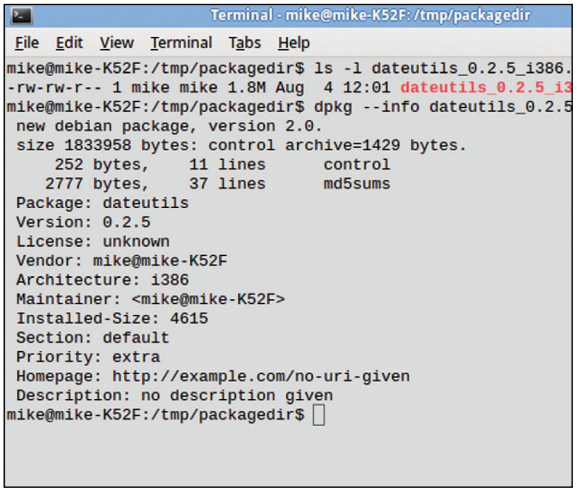
```
Created deb package {:path=>"dateutils_0.2.5_i386.deb"}
```

Take a look at the package details:

```
dpkg --info dateutils_0.2.5_i386.deb
```

You'll see that *FPM* has populated many of the package description fields automatically, eg using your login name and system hostname for the vendor and maintainer fields. Others are given generic text, like the description and homepage. You can add options to the *FPM* command to customise these, as we'll see in a moment, but for now you'll want to install the package to check that it's working:

```
sudo dpkg -i dateutils_0.2.5_i386.deb
```



Ta-da: our newly built package in all its glory. Some of the information fields need some more details, but we can easily fix that.

Enter one of the commands included in the package, such as **ddiff**, and you'll see that everything is in order. Great success! If you've ever tried to make a Deb package the traditional way, you're probably be jumping for joy at the simplicity of all this (or crying at the difficult memories it conjures up). And we're only just getting started...

During the package building phase, you can add extra information to the resulting file. This isn't a step which is necessary to produce a functional package, but if you're creating packages for others to use it makes your work look more professional. First of all, add a textual description via the **--description** flag: use single quotes to specify the text, which also lets you enter newline characters. A good description shouldn't be too long or meandering, and simply explain the core purpose of the program.

Customising the results

Next, use **--url** to add a website address for the program, which is typically its home page. It's also a good idea to use **--license** (put multiple words in single quotes if necessary) so that end users know whether they can redistribute the package, along with **--vendor** and **--maintainer** to provide a contact address if a user needs to get in touch.

The **--before-install**, **--after-install**, **--before-remove** and **--after-remove** options are especially useful. With these, you can provide scripts that should be run at the corresponding times during the (de)installation process.

Many packages make use of these scripts to perform initial setup operations before putting the files in place, or cleaning up old temporary files after the package is completely removed.

To see how this works, create a text file called **afterinstall** in **/tmp** with the following contents:

```
#!/bin/sh
  ls --color
```

› **Many distro-neutral packaging formats have come and gone. Do you remember this one?**

Tips for distro independence

Building a package that works across multiple distros is no mean feat. A few projects emerged over the years that aimed to create a distro-neutral packaging format, most notably the now defunct *Autopackage*, but none of them really took off. Still, there are a few things you can do to ensure that your packages work on as many distros as possible.

First, try to build the package on a slightly older version of your distro. Whether this is possible or not depends on the program's dependencies, but if you can get by with older versions of libraries, that helps a lot. For instance, if your distro has **libfoo 3.4**, but you can build the package on an earlier distro release that has **libfoo 3.1**, then your package (should!) work on a wider range of distros with varying **libfoo** versions (eg a spin-off of your distro that has **libfoo 3.3**). Generally, open source libraries take backward compatibility seriously, so you shouldn't run into major problems with this approach.

In your post-installation scripts, try to use vanilla tools that are available on every distro, and not distro-specific programs. If you're building a package on OpenSUSE and need to do some configuration work after the installation phase with a post-install script, it might be tempting to call a *Yast* module, but then the package definitely won't work on Fedora.

Also, take a look at the Linux Standards Base and Filesystem Hierarchy Standard **(www.linuxfoundation.org/collaborate/workgroups/lsb)**. These are projects that attempt to unify the common toolset, libraries and directory layouts across distros, and many distros include an **lsb_release** script for getting version information. You could use **lsb_release -a** in a script to find out which distro and version are being used.

You will need to make it executable (**chmod +x /tmp/afterinstall**) and build the package again, using the **--after-install** option like this:

```
fpm -s dir -t deb -n dateutils -v 0.2.5 --after-install /tmp/afterinstall -C /tmp/packagedir/ .
```

When you install the new package, you'll see the output of **ls --color** after the **Setting up dateutils (0.2.5)** line. This is a highly versatile system, in that you can print out messages during the installation phase or even ask the user questions with a bit of shell scripting.

Conflicts and dependencies

Some packages can't be installed if a certain other package is already present on the system. This isn't a common occurrence, but it helps to avoid clashes between packages that provide the same functionality in the same filesystem locations. Try building the package with **--conflicts xterm**, for instance, and then installing it again – you'll see an error saying that the package can't be installed because *Xterm* is already on the system. (Well, providing you have *Xterm* already installed, of course.)

While our *Dateutils* example has no requirements beyond the standard C library, most programs will need other libraries and packages installed to run. Yes, we're talking about dependencies here, but don't run away screaming as FPM handles them elegantly.

First off, you'll need to find out which libraries (and versions) are required by the program you're packaging up. If the software is well documented, you should be able to find this out from the **Readme** and **Install** files, but if not, there are some other tricks you can employ. Running l**dd** on the program's main binary after compilation will show you a detailed list of every library file that it uses, and with your usual packaging tools you can find out which library file belongs to which package.

Let's say that *Dateutils* needs at least version 2.17 of the C library. We can specify this as a dependency during the build phase like so:

```
fpm -s dir -t deb -n dateutils -v 0.2.5 -d 'libc6 (>= 2.17)' -C /tmp/packagedir/ .
```

The most important thing to note here is the **>=** part, which means 'greater than or equal to'. So, our *Dateutils* package won't be installed unless the current C library version is 2.17 or newer. You will need to change that to

Building RPMs

As mentioned earlier, to make *RPM*s you just need to pass the **-t rpm** option to FPM. This should work without a hitch on *RPM*-based systems, but if you're running another distro you'll need extra tools. On Ubuntu-based distros you can use **sudo apt-get install rpm** which provides the *rpmbuild* program that *FPM* needs. Then, by running the previously listed commands with **-t rpm**, you will end up with a package that's called **dateutils-0.2.5-1.i686.rpm**.

Note that there are some *FPM* options which only apply to *RPM*s. You can get a list of these by running **fpm --help** and looking for the lines which contain 'rpm only'. You should also take care with *RPM*s that are produced on Deb-based systems: in most cases they shouldn't pose any problems, but if you're distributing software online then it's worth trying them out on a genuine RPM-based distribution before open the doors and handing them out

something excessively large version number like 9.99, rebuild the package and try to install it again, and you'll see a message along the lines of:

```
dpkg: dependency problems prevent configuration of
dateutils:
 dateutils depends on libc6 (>= 9.99); however:
 Version of libc6:i386 on system is 2.17-0ubuntu5.
```

It's possible to specify multiple dependencies with a series of **-d** flags followed by the package names and versions as illustrated in the code above. If the program you're packaging up has a vast range of dependencies, check in your distro's package manager to see if there's a *meta-package* to cover them all. For instance, if you're creating a package for a Gnome app, instead of typing in endless lines of dependencies for the various parts of Gnome, you could simply make your package dependent on 'gnome' which itself pulls in all the major Gnome dependencies.

Advanced options

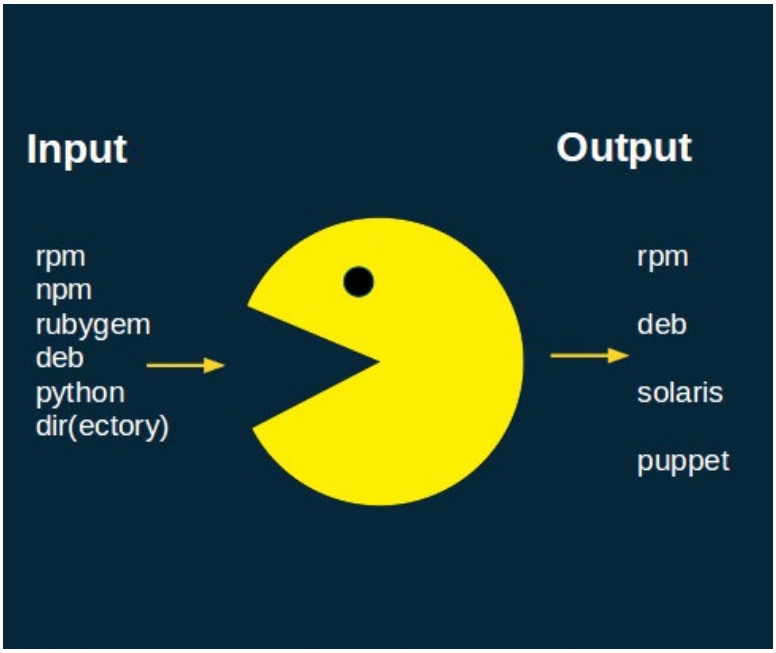
So far through this tutorial, we have looked at using directories as sources for *FPM*; the **-s dir** part. But *FPM* can generate packages from other files as well, such as tarballs. If you have **foo-1.0.tar.gz** which contains files and a directory structure that can be copied into the filesystem (eg **/usr/bin/foo, /usr/share/doc/foo/** and so forth), then you can convert it into a .deb or RPM using **-s tar** like this:

```
fpm -s tar -t deb -n foo -v 1.0 foo-1.0.tar.gz
```

Another source that *FPM* can use is Python modules. Thanks to *easy_install*, *FPM* can download modules and package them up automatically, rather than having to hunt around on websites. For instance, to build a package of *PyX*, a module for creating PostScript and PDF files:

```
fpm -s python -t deb pyx
```

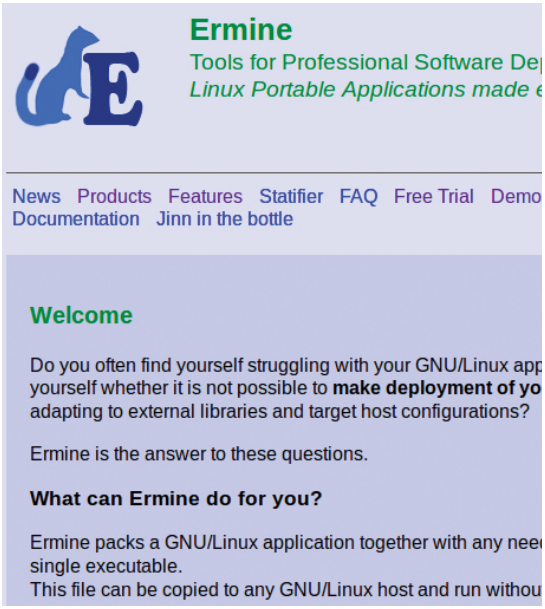
Last of all, with **-s empty** you can create completely empty packages. But why would you want to do that?



› See http://goo.gl/sWs3Z for a light-hearted presentation from FPM's author, explaining his motivation for writing the program.

Well, this feature is primarily useful for creating meta-packages. For instance, you might be setting up a bunch of desktop machines with a specific combination of programs (a certain window manager, web browser, editor etc.) Instead of installing the programs by hand on each machine, or fiddling around with a script, you could create a meta-package with the required programs as dependencies. Then you just need to install the package on each machine and the package manager will handle the rest. LXF

Building static binaries



› *Ermine* massively simplifies the job of making statically-linked binaries, but it's not open source.

One way to make your packages work across many distros is to statically link the executable files. Normally, programs make use of external code libraries via a system called dynamic linking, that is, they access the libraries when needed. Libraries typically live in **/lib** and **/usr/lib**, are provided in their own packages, and can be updated independently of the programs that use them.

This is a very sensible system: why should every *GTK*-based program include its own version of the *GTK* library, when they could all share the same version? And it's good from a security viewpoint as well, because when a vulnerability is found in *GTK*, you can update the shared version and all programs using it are automatically fixed.

Now, if you're willing to lose these benefits, you can statically link the executable file(s) inside your package. This rolls all of the shared library code into the executable, producing a much larger file but one that will work virtually everywhere. It doesn't matter which library versions a distro is using; all the code your program needs is included in the executable. Building static binaries is complicated, but a proprietary program called *Ermine* exists that makes it much simpler (**www.magicermine.com**). Download the **ErmineLightTrial.i386** (or .x86_64) file and make it executable. Now find the binary that you want to make static. For example, let's look at the *gedit* binary here: if we run **ldd /usr/bin/gedit** we see that it uses over 60 shared libraries on the system. But after doing this:

```
./ErmineLightTrial.i386 /usr/bin/gedit
--output static-gedit
```

We now have a complete binary called **static-gedit** which doesn't depend on any other libraries. It's much larger at -42MB, compared to 671K. In general, it's better to stick with dynamic libraries in most scenarios, but if you're using *FPM* to distribute your self-written software on the net, you could make an optional statically linked package for users who can't get the normal package to work.