Bash con ejemplos, Parte 3

1. Explorando el sistema de ebuilds

Entra en el sistema de ebuilds

He estado deseando que llegara este capítulo final de Bash con ejemplos, porque ahora que hemos cubierto los conceptos básicos de la programación en bash, Parte 1 y Parte 2, podemos centrarnos en temas más avanzados, como el desarrollo de aplicaciones en bash y diseño de programas. Te daré una buena dosis de programación práctica, del mundo real, presentando un proyecto en cuya codificación y refinamiento he invertido muchas horas: El sistema de Ebuilds de Gentoo.

Soy el arquitecto líder de Gentoo Linux, un sistema Linux de próxima generación, actualmente en estado beta. Una de mis principales responsabilidades es asegurarme de que todos los paquetes binarios (similares a los RPM) se crean correctamente, y funcionan bien en conjunto. Como probablemente sepas, un sistema Linux estándar no está compuesto de un solo árbol unificado de fuentes, como en el caso de BSD, sino que está compuesto de más de 25 paquetes críticos que funcionan juntos. Algunos de estos paquetes son:

Paquete	**Descripción**
linux	El kernel actual
util-linux	Una colección de programas variados relacionados con Linux
e2fsprogs	Una colección de utilidades relacionadas con ext2
glibc	La librería C de GNU

Cada paquete viene en su propio tarball, y es mantenido por desarrolladores o equipos de desarrollo distintos. Para crear una distribución, cada paquete debe ser descargado por separado, compilado, y empaquetado, cada vez que el paquete necesita ser reparado, actualizado o mejorado. Todo esto debe ser repetido (algunos paquetes quedan desfasados realmente rápido). Para ayudar en este proceso tan repetitivo, he creado el sistema de ebuilds. Escrito casi por completo en bash. Y para mejorar tu conocimiento de bash, te enseñaré como implementé las secciones de desempaquetado y compilado del sistema de ebuilds. Al explicar cada paso, explicaré también por qué se hicieron ciertas decisiones. Al final de este artículo, no solo tendrás una visión de proyectos en bash a mayor escala, sino que también habrás implementado una buena porción de un sistama de autocompilado.

¿Por qué bash?

Bash es un componente esencial del sistema de ebuilds de Gentoo Linux. Fue elegido como el lenguaje primario para los ebuilds por varias razones. En primer lugar, posee una sintaxis familiar y asequible, muy apropiada para el uso de programas externos. Un sistema de autocompilado es un código intermedio que automatiza la llamada a programas externos, y bash es un lenguaje particularmente apropiado para este tipo de aplicación. Segundo, el soporte de bash para funciones permite al sistema de ebuilds adoptar un diseño modular, fácil de entender. Tercero, el sistema de ebuilds saca provecho del soporte de bash para las variables de entorno, permitiendo a los mantenedores de paquetes y a los desarrolladores reconfigurarlo al vuelo.

Revista al proceso de construcción

Antes de entrar en el sistema de ebuilds de lleno, tendremos que conocer los pasos necesarios para compilar e instalar un paquete. Para nuestro ejemplo usaremos el paquete "sed", un editor estándar de flujos GNU que es parte integrante de todas las distribuciones de Linux. Primero descarga el archivo con las fuentes (sed-3.02.tar.gz) (ver Recursos). Almacenaremos este archivo en /usr/src/distfiles, un directorio al que nos referiremos usando la variable de entorno $DISTDIR. $DISTDIR es el directorio donde se guardarán todos los tarball de código fuente, será un gran almacén de código fuente.

Nuestro siguiente paso será crear un directorio temporal work, que aloje los fuentes descomprimidos. Nos referiremos a este directorio usando la variable $WORKDIR. Para ésto, cambia a un directorio sobre el que tengas permiso de escritura y escribe lo siguiente:

Listado de Código 1.1: Descomprimiendo sed en un directorio temporal

```
$ mkdir work
$ cd work
$ tar xzf /usr/src/distfiles/sed-3.02.tar.gz
```

Ahora el tarball está descomprimido, habrá creado un directorio llamado sed-3.02, que contiene las fuentes de sed. Nos referiremos a dicho directorio sed-3.02 más tarde usando la variable de entorno $SRCDIR. Para compilar el programa teclea lo siguiente:

Listado de Código 1.2: Uncompressing sed into a temporary directory

```
$ cd sed-3.02
$ ./configure --prefix=/usr
(autoconf generará los archivos make adecuados, esto puede tardar)

$ make

(el paquete se compila desde fuente, también tardará un poco)
```

Vamos a saltarnos el paso "make install", ya que solo estamos cubriendo los pasos de desempaquetado y compilación en este artículo. Si quisiéramos usar un guión de bash para realizar todos estos pasos por nosotros haríamos algo como:

Listado de Código 1.3: Script bash de ejemplo para desempaquetar y compilar

```
#!/usr/bin/env bash

if [ -d work ]
then
# remove old work directory if it exists
      rm -rf work
fi
mkdir work
cd work
tar xzf /usr/src/distfiles/sed-3.02.tar.gz
cd sed-3.02
./configure --prefix=/usr
make
```

Generalizando el código

Aunque este guión de autocompilado funciona, no es muy flexible. Básicamente, el guión contiene la lista de los comandos que han sido escritos anteriormente en línea de comandos. Esta solución funciona, pero sería mucho mejor tener un guión más genérico que pudiera configurar y desempaquetar cualquier paquete, quizás cambiando solo unas

pocas líneas. El trabajo para el mantenedor del paquete se ve así disminuido, y es más fácil añadir nuevos paquetes a la distribución. Podemos usar variables de entorno para hacer nuestro guión más genérico:

Listado de Código 1.4: Un guión nuevo, más genérico

```
#!/usr/bin/env bash

# P es el nombre del paquete

P=sed-3.02

# A es el nombre del archivo comprimido

A=${P}.tar.gz

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
export SRCDIR=${WORKDIR}/${P}

if [ -z "$DISTDIR" ]
then
# DISTDIR es /usr/src/distfiles si no ha sido definido ya
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

if [ -d ${WORKDIR} ]
then
# borra el directorio de trabajo antiguo si es que existe
        rm -rf ${WORKDIR}
fi

mkdir ${WORKDIR}
cd ${WORKDIR}
tar xzf ${DISTDIR}/${A}
cd ${SRCDIR}
./configure --prefix=/usr
make
```

Hemos añadido muchas variables al nuevo código, pero, básicamente, todavía hace lo mismo. Sin embargo, ahora podemos compilar cualquier paquete basado en GNU autoconf. Simplemente copiando este archivo con un nuevo nombre que refleje el nombre del paquete, y cambiando los valores de $A y $P, compilará. Las demás variables se ajustarán automáticamente. Si bien es útil, hay aún mejoras que podemos introducir en este código. Este código es bastante más largo que el original. Ya que una de las tareas principales de cualquier proyecto de programación es reducir la complejidad de cara al usuario, estaría bien reducir un poco la longitud del código, o, al menos, organizarlo un poco mejor. Podemos hacer esto con un ingenioso truco -- separaremos el código en dos ficheros separados, guarda lo siguiente como sed-3.02.ebuild:

Listado de Código 1.5: sed-3.02.ebuild

```
#fichero ebuild para sed - ¡simple!
P=sed-3.02
A=${P}.tar.gz
```

Nuestro primer fichero es trivial, y contiene solo variables de entorno, que han de ser configuradas paquete por paquete, el segundo fichero contiene el cerebro de la operación. Guárdalo como "ebuild" y hazlo ejecutable:

Listado de Código 1.6: El guión ebuild

```
#!/usr/bin/env bash


if [ $# -ne 1 ]
then
        echo "se esperaba un argumento."
        exit 1
fi

if [ -e "$1" ]
then
        source $1
else
        echo "ebuild $1 no encontrado."
        exit 1
fi

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
export SRCDIR=${WORKDIR}/${P}

if [ -z "$DISTDIR" ]
then
        # DISTDIR será /usr/src/distfiles si no está ya definido
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

if [ -d ${WORKDIR} ]
then
        # borra directorio antiguo si ya existía
        rm -rf ${WORKDIR}
fi

mkdir ${WORKDIR}
cd ${WORKDIR}
tar xzf ${DISTDIR}/${A}
cd ${SRCDIR}
./configure --prefix=/usr
make
```

Ahora que hemos dividido nuestro sistema en dos ficheros, apuesto a que te estarás preguntando como funciona. Fácil, para compilar sed, escribe:

Listado de Código 1.7: Probar ebuild

```
$ ./ebuild sed-3.02.ebuild
```

Cuando "ebuild" se ejecuta, primero intenta interpretar $1. ¿Que significa esto? Recuerda de mi anterior artículo, que $1 es el primer argumento de línea de comandos, en este caso sed-3.02.ebuild. En bash, el comando "source" lee instrucciones bash de un archivo y las ejecuta como si estuvieran dentro del fichero desde donde se usa el comando "source". Así que "source" ${1}" causa que el guión "ebuild" ejecute los comandos contenidos en sed-3.02.ebuild, de este modo, $P y $A son definidos. Este cambio de diseño es realmente conveniente, porque si queremos compilar otro programa, en lugar de sed, tan solo necesitamos un nuevo fichero .ebuild que pasar a nuestro guión "ebuild". De este modo, los ficheros .ebuild son realmente simples, mientras la parte complicada del sistema se almacena en el guión "ebuild". De esta forma, también se puede mejorar o actualizar el sistema ebuild simplemente editando el guión, manteniendo los detalles de la implementación fuera de los ficheros ebuild. Aquí hay un fichero ebuild de ejemplo

para gzip:

Listado de Código 1.8: gzip-1.2.4a.ebuild

```
#Otro guión ebuild realmente simple
P=gzip-1.2.4a
A=${P}.tar.gz
```

Añadiendo funcionalidad

Bien, ya hemos hecho algún progreso, pero hay funcionalidades adicionales que me gustaría añadir. Me gustaría que el guión ebuild aceptara un segundo parámetro que será compile, unpack, o all. Este segundo parámetro dirá al ebuild la operación que debe realizar. De esta forma, puedo decirle a ebuild que desempaquete el archivo pero sin compilarlo (por si necesito inspeccionar el código fuente antes de la compilación). Para hacer esto usaremos una estructura case, que comprobará la variable $2, y actuará de acuerdo con su valor. El código sería algo así:

Listado de Código 1.9: ebuild, revisión 2

```
#!/usr/bin/env bash

if [ $# -ne 2 ]
then
        echo "Por favor, especifique dos argumentos, el fichero .ebuild y"
        echo "unpack, compile or all"
        exit 1
fi


if [ -z "$DISTDIR" ]
then
# DISTDIR será /usr/src/distfiles si no está ya definido
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

ebuild_unpack() {
         #nos aseguramos de estar en el directorio correcto
        cd ${ORIGDIR}

        if [ -d ${WORKDIR} ]
        then
                rm -rf ${WORKDIR}
        fi

        mkdir ${WORKDIR}
        cd ${WORKDIR}
        if [ ! -e ${DISTDIR}/${A} ]
        then
            echo "${DISTDIR}/${A} no existe. Por favor, descárguelo primero."
            exit 1
        fi
        tar xzf ${DISTDIR}/${A}
        echo "Desempaquetado ${DISTDIR}/${A}."
        #el código fuente está descomprimido
}


ebuild_compile() {

         #nos aseguramos de estar en el directorio correcto
        cd ${SRCDIR}
```

```
        if [ ! -d "${SRCDIR}" ]
        then
                echo "${SRCDIR} no existe -- por favor, descomprima primero el paquete."
                exit 1
        fi
        ./configure --prefix=/usr
        make
}

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work

if [ -e "$1" ]
then
        source $1
else
        echo "Ebuild $1 no encontrado."
        exit 1
fi

export SRCDIR=${WORKDIR}/${P}

case "${2}" in
        unpack)
                ebuild_unpack
                ;;
        compile)
                ebuild_compile
                ;;
        all)
                ebuild_unpack
                ebuild_compile
                ;;
        *)
                echo "Por favor, especifique unpack, compile o All como segundo argumento"
                exit 1
                ;;
esac
```

Hemos hecho varios cambios, así que revisémoslos. Primero, hemos puesto las órdenes para desempaquetar y compilar los paquetes en su propia función. Las hemos llamado ebuild_compile() y ebuild_unpack(), respectivamente. Ha sido un buen movimiento, ya que el código se está complicando, y las funciones lo dotan de algo más de modularidad, lo que nos ayudará a mantener el guión ordenado. En la primera línea de cada función, se cambia de forma explícita, con cd, al directorio al que se quiere ir. Al complicarse nuestro código es muy probably que terminemos ejecutando algo en un directorio distinto del correcto, así, nos aseguramos de estar en el lugar correcto antes de hacer nada, con cd, y nos ahorraremos posible errores más adelante. Ésto es un paso importante, sobre todo, si se borran ficheros dentro de una función.

También he añadido un test al principio de la función ebuild_compile(). Ahora comprueba que el directorio $SRCDIR existe, y, si no, imprime un mensaje de error diciéndole al usuario que desempaquete el archivo y sale. Si lo prefieres, puedes cambiar el comportamiento de forma que, si $SRCDIR no existe, nuestro ebuild desempaquete automáticamente el archivo. Puedes hacerlo cambiando ebuild_compile() por esta nueva versión:

Listado de Código 1.10: Nueva versión de ebuild_compile()

```
ebuild_compile() {
         #nos aseguramos de estar en el directorio correcto
        if [ ! -d "${SRCDIR}" ]
```

```
        then
                ebuild_unpack
        fi
        cd ${SRCDIR}
        ./configure --prefix=/usr
        make
}
```

Uno de los cambios más obvios en nuestro guión ebuild es la estructura case añadida al final del mismo. Dicha estructura simplemente chequea el segundo argumento de línea de comandos, y, en base al valor del mismo, decide la acción a realizar. Si ahora ejecutamos esto:

Listado de Código 1.11: Acción predeterminada

```
$ ebuild sed-3.02.ebuild
```

Obtendremos un mensaje de error, porque ebuild ahora necesita que le digamos qué hacer, de esta forma:

Listado de Código 1.12: Descomprimir

```
$ ebuild sed-3.02.ebuild unpack
```

or:

Listado de Código 1.13: Compilar

```
$ ebuild sed-3.02.ebuild compile
```

or:

Listado de Código 1.14: Descomprimir y compilar

```
$ ebuild sed-3.02.ebuild all
```

Importante: Si se suministra un segundo parámetro distinto de los usados más arriba, se obtiene un mensaje de error (caso *), y el programa termina.

Modularizando el código

Ahora que el código es más avanzado y funcional, puede que estés pensando en crear varios ebuilds para desempaquetar y compilar tus programas favoritos. Si lo hicieras, tarde o temprano comprobarías que algunas fuentes no usan autoconf (./configure), sino que se valen de otros procesos de compilación no estándar. Tenemos que modificar el sistema de ebuilds para que se acomode a estos programas. Pero antes de hacerlo, es bueno pararse a pensar como conseguiremos esto.

Una de las grandes ventajas de usar siempre ./configure --prefix=/usr; make en la fase de compilación, es que, la mayoría de las veces funciona. Pero también debemos hacer que el sistema de ebuilds funcione con aquellos fuentes que no usan autoconf, o fichero Make normales. Propongo lo siguiente, como solución a este problema:

1. Si hay un guión configure en ${SRCDIR}, ejecutarlo de esta forma: ./configure --prefix=/usr. De otro modo, saltarse este paso.
2. Ejecutar el comando siguiente: make

Los ebuilds solo ejecutarán configure si dicho guión existe. Así hacemos que ebuild funcione con programas que no usan autoconf, y tienen un fichero Make estándar. Pero, ¿y si un simple "make" no funciona con algunos fuentes? Necesitamos una forma de saltarse esta funcionalidad predefinida, usando un código alternativo para manejar situaciones específicas. Para esto, convertiremos nuestra función ebuild_compile() en dos funciones. La primera de dichas funciones puede ser vista como "padre" de la segunda, y se llamará ebuild_compile(). La nueva función, llamada user_compile(), contendrá nuestras acciones predeterminadas:

Listado de Código 1.15: ebuild_compile() separada en dos funciones

```
user_compile() {
        #estamos en ${SRCDIR}
        if [ -e configure ]
        then
                #ejecuta el guión configure si éste existe
                ./configure --prefix=/usr
        fi
        #run make
        make
}

ebuild_compile() {
        if [ ! -d "${SRCDIR}" ]
        then
                echo "${SRCDIR} no existe -- por favor, descomprima primero."
                exit 1
        fi
        #se asegura de que estamos en el directorio correcto
        cd ${SRCDIR}
        user_compile
}
```

Puede que no parezca obvio el por qué de todo esto ahora mismo. Así que, por ahora, sigamos. Si bien el código de arriba funciona de forma idéntica a la anterior versión de ebuild, ahora podemos hacer algo que no podiamos hacer antes. Podemos redefinir la función user_compile() en sed-3.02.ebuild. Así, si la predeterminada user_compile() no sirve a nuestras necesidades, podemos redefinirla por completo en nuestro fichero .ebuild. Como ejemplo, un fichero .ebuild para e2fsprogs-1.18, que requiere una línea ./configure ligeramente modificada:

Listado de Código 1.16: e2fsprogs-1.18.ebuild

```
#este fichero ebuild redefine user_compile()
P=e2fsprogs-1.18
A=${P}.tar.gz

user_compile() {
       ./configure --enable-elf-shlibs
       make
}
```

Ahora, e2fsprogs será compilado de la forma correcta. Para la mayoría de los paquetes, esto no es necesario. Simplemente omitiendo la definición de user_compile() en nuestro fichero .ebuild, conseguiremos que se use la función user_compile() predeterminada.

¿Como sabe el guión ebuild qué función user_compile() debe usar? Muy sencillo: en el guión ebuild, la función user_compile() es definida antes de que el fichero .ebuild e2fsprogs-1.18.ebuild sea leído. Si hay una función user_compile() en e2fsprogs-1.18.ebuild, dicha función sobreescribe a la versión predeterminada, definida

previamente. Si no, la primera versión es usada.

Hemos añadido una gran funcionalidad sin requerir ningún tipo de codificación compleja. No lo explicaré aquí, pero se podría hacer algo similar con la función ebuild_unpack(), de forma que podamos reescribir el proceso de desempaquetado predeterminado. Esto podría ser práctico si se tiene que hacer algún tipo de parcheo o si los ficheros están contenido en múltiples archivos comprimidos. También sería una buena idea modificar el código de desempaquetado de forma que reconozca tarballs comprimidos con bzip2 por defecto.

Ficheros de configuración

Hemos cubierto varias técnicas interesantes de bash, y ahora es el momento de aprender una más. A menudo es práctico para un programa tener un fichero de configuración global que resida en /etc. Afortunadamente, esto es fácil cuando se usa bash. Simplemente crea este fichero y guárdalo como /etc/ebuild.conf:

Listado de Código 1.17: /ect/ebuild.conf

```
# /etc/ebuild.conf: configuraciones globales de ebuild

# MAKEOPTS son las opciones pasadas a make
MAKEOPTS="-j2"
```

En este ejemplo he incluido una sola opción de configuración, pero se podrían incluir muchas más. Una de las cosas interesantes de bash es que el fichero se puede interpretar simplemente usando el comando "source" sobre el mismo. Éste es un truco de diseño que funciona con la mayoría de lenguajes interpretados. Después de que /etc/ebuild.conf haya sido interpretado, $MAKEOPTS está definido en nuestro guión .ebuild, y le permite al usuario pasar dichas opciones a make. En este caso, la opción le dice al ebuild que lance una instancia paralela de make.

Nota: ¿Qué es una instancia paralela de make? Las instancias paralelas pueden servir para agilizar el proceso en sistema con varios procesadores. Make soporta la compilación en paralelo. Esto significa que, en lugar de compilar un fichero fuente en un momento dado, make puede compilar un número de ficheros (dado por el usuario) al mismo tiempo. En un sistema multiprocesador esto hace que se usen estos procesadores extra. Make en paralelo se activa al interpretar la opción -j # pasada a make, de esta forma: make -j4 MAKE="make -j4". Esto instruye a make para compilar cuatro programas de forma simultánea. El argumento MAKE="make -j4" le dice a make que pase la opción -j4 a cualquier proceso hijo que lance.

Y aquí tenemos la versión final de ebuild:

Listado de Código 1.18: ebuild, la versión final

```
#!/usr/bin/env bash

if [ $# -ne 2 ]
then
        echo "Por favor, especifique fichero ebuild file y unpack, compile u all"
        exit 1
fi

source /etc/ebuild.conf

if [ -z "$DISTDIR" ]
then
```

```
        # configura DISTDIR como /usr/src/distfiles si no está configurado ya
        DISTDIR=/usr/src/distfiles
fi
export DISTDIR

ebuild_unpack() {
        #se asegura de estar en el directorio correcto
        cd ${ORIGDIR}

        if [ -d ${WORKDIR} ]
        then
                rm -rf ${WORKDIR}
        fi

        mkdir ${WORKDIR}
        cd ${WORKDIR}
        if [ ! -e ${DISTDIR}/${A} ]
        then
                echo "${DISTDIR}/${A} no existe.  Por favor, descargue primero."
                exit 1
        fi
        tar xzf ${DISTDIR}/${A}
        echo "Unpacked ${DISTDIR}/${A}."
        #las fuentes han sido descomprimidas
}

user_compile() {
        #ya estamos en ${SRCDIR}
        if [ -e configure ]
        then
                #ejecuta el guión configure si existe
                ./configure --prefix=/usr
        fi
        #ejecuta make
        make $MAKEOPTS MAKE="make $MAKEOPTS"
}

ebuild_compile() {
        if [ ! -d "${SRCDIR}" ]
        then
                echo "${SRCDIR} no existe -- por favor, descomprima primero."
                exit 1
        fi
        #se asegura de estar en el directorio correcto
        cd ${SRCDIR}
        user_compile
}

export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work

if [ -e "$1" ]
then
        source $1
else
        echo "Fichero .ebuild $1 no encontrado."
        exit 1
fi

export SRCDIR=${WORKDIR}/${P}

case "${2}" in
        unpack)
                ebuild_unpack
                ;;
        compile)
                ebuild_compile
```

```
                ;;
        all)
                ebuild_unpack
                ebuild_compile
                ;;
        *)
                echo "Por favor, especifique unpack, compile u all como segundo argumento"
                exit 1
                ;;
esac
```

/etc/ebuild.conf es interpretado cerca del principio del fichero. Usamos $MAKEOPTS en nuestra user_compile() prefabricada. Puede que te preguntes como funcionará ésto -- después de todo, nos referimos a $MAKEOPTS antes de interpretar /etc/ebuild.conf, que es el encargado de definir $MAKEOPTS. Afortunadamente, ésto no es problema, porque la expansión de variables se produce al ejecutar user_compile(). Cuando eso sucede, /etc/ebuild.conf ha sido ya incorporado y $MAKEOPTS tiene un valor correcto.

Resumiendo

Hemos cubierto muchas técnicas de programación en bash en este artículo, pero en realidad solo hemos arañado la superficie de lo que el poder auténtico de bash representa. Por ejemplo, el sistema de ebuilds de Gentoo no solo puede desempaquetar y compilar de forma automática, sino que también:

- Descarga las fuentes de forma automática si no están en $DISTDIR
- Verifica que las fuentes no están corruptas, usando sumas MD5
- Si se especifica, instala el programa compilado en un sistema de archivos, en vivo, manteniendo un listado de los ficheros instalados de forma que el paquete pueda ser fácilmente desinstalado en cualquier momento.
- Si se especifica, puede empaquetar una aplicación instalada en un tarball (comprimido) de forma que pueda ser instalada después, en otro ordenador, o durante un proceso de instalación basado en CD (por ejemplo, si estás construyendo una distribución basada en dicho medio).

De forma adicional, el sistema ebuild en producción tiene otras opciones globales de configuración, que permiten al usuario establecer banderas de optimización que se usan en tiempo de compilación, o el soporte específico que se quiere en ciertas aplicaciones. Por ejemplo, los soportes para GNOME y slang se activan de forma predeterminada en los paquetes que lo soportan.

Bash puede hacer mucho más de lo que he tocado en esta serie de artículos. Espero que hayas aprendido mucho sobre esta increíble utilidad, y que estés deseando usar bash para acelerar y mejorar tus proyectos de desarrollo.

Sumario: En su artículo final de Bash con ejemplos, Daniel Robbins da un buen repaso al sistema de ebuilds de Gentoo Linux, un excelente ejemplo del poder verdadero de bash. Paso a paso te enseñará como fue implementado el sistema de ebuilds, entrando de lleno en técnicas de bash muy útiles y estrategias de diseño. Al final de este artículo tendrás una buena idea básica de todo lo que supone la creación de una aplicación completa basada en bash, así como un posible comienzo para la creación de tu propio sistema de compilación automática.