

# **TUTORIAL DE CGI**

## INTRODUCCIÓN

El CGI (*Common Gateway Interface*) es un estandar para comunicar aplicaciones externas con los servidores de información, tales como servidores HTTP o *Web*. Un documento en HTML que el *daemon* del *Web* se trae es estático, es decir, se mantiene constante: un fichero de texto que no cambia. Un programa CGI, por otro lado es ejecutado en tiempo real, así que puede generar información dinámica.

Por ejemplo, supongamos que quieres enganchar tu base de datos de Unix al WWW, para permitir a gente de todo el mundo consultarla. Básicamente se necesitará un programa CGI que el *daemon* del *Web* ejecutará para transmitir la información al gestor de base de datos, y recibir los resultados para presentárselos al cliente. Este es un ejemplo de pasarela (*gateway*), y es lo que CGI, actualmente en su versión 1.1, tiene en sus orígenes.

El ejemplo de la base de datos es una idea sencilla, pero la mayoría de las veces difícil de implementar. Realmente no hay límite con lo que tu quieras enganchar al Web. La única cosa que debes recordar es que haga lo que haga tu programa CGI, no deberá tardar mucho tiempo en procesar. De otra manera, el usuario estaría esperando con su navegador a que algo pase.

### **Especificaciones**

Como un programa CGI es un ejecutable, es equivalente a dejar a el mundo ejecutar un programa en tu sistema, que no es lo mas seguro a hacer. Por ello existen una serie de precauciones de seguridad que son necesarias de implementar cuando se usan programas CGI. Probablemente la que afectará al usuario típico del *Web*, es que hecho de que los programas CGI necesitan residir en un directorio especial, así el servidor sabe que tiene que ejecutarlo, en vez de simplemente mostrarlo por pantalla. Este directorio está generalmente bajo el control del *webmaster*, prohibiendo al usuario medio crear programas CGI. Hay otros métodos para permitir el accesos a scripts CGI, pero depende del *webmaster* que se te de esta posibilidad. Así que deberás contactar con tu *webmaster* para consultar la factibilidad de permitirte un acceso a los CGI.

Si dispones de una versión del servidor *HTTPd NCSA*, verás un directorio denominado */cgi-bin*. Este es el directorio especial antes mencionado, donde todos los programas CGI residen. Un programa CGI se puede escribir en cualquier lenguaje que permita ser ejecutado en el sistema, como:

- C/C++
- Fortran
- PERL
- TCL
- Algún Shell de Unix
- Visual Basic

## AppleScript

Simplemente depende de lo que tengas en tu sistema. Si usas un lenguaje de programación como *C* o *Fortran*, como ya sabrás, debes compilar el programa antes de poder ejecutarlo. Si miras en el directorio */cgi-src*, encontrarás el código fuente de algunos programas CGI del directorio */cgi-bin*. Pero, si usas alguno de los lenguajes interpretados, como *PERL*, *TCL*, o un shell de *Unix*, el script simplemente necesita residir en el directorio */cgi-bin*, ya que no tiene un código fuente asociado. Mucha gente prefiere escribir scripts CGI en vez de programas, ya que son más fáciles de depurar, modificar y mantener que un programa típico compilado.

## ¿Qué es el directorio cgi-bin?

Este es un directorio especial, que contiene los scripts, configurado dentro del servidor http. El servidor conoce que este directorio contiene ejecutables que deberán ser ejecutados y su salida deberá ser enviada al navegador del cliente. No se puede simplemente crear un directorio *cgi-bin*, el administrador del servidor deberá configurarlo para su uso. Si no está configurado, los scripts serán cargados como simples ficheros de texto.

Algunos servidores están configurados de tal manera que los ficheros con una determinada extensión (generalmente *.cgi*) son reconocidos como scripts y serán ejecutados como si estuvieran en un directorio *cgi-bin*.

Nota: No deberemos confundirlo con *html analizado* (generalmente *.shtml*)

La configuración de los directorios, o de la extensión mencionada antes, depende únicamente del servidor. Comprueba la documentación sobre tu servidor, o pregunta a otro usuario que también lo use.

## ¿Qué parte es Perl, y qué parte es html?

El formulario que se presenta al usuario está escrito en *html*, y este llama al script en el servidor escrito en *perl*. El script devolverá en la mayoría de los casos código *html* para presentar al usuario.

## **VARIABLES DE ENTORNO EN CGI**

El servidor usa tanto de líneas de comando, como variables de entorno para pasar los datos del servidor al script.

Estas variables de entorno se activan cuando el programa ejecuta el programa cgi.

### **Especificación**

Las siguientes variables no dependen de la información enviada y son activadas en todos los casos:

#### **SERVER\_SOFTWARE**

Devuelve el nombre y la versión del software del servidor de información que contesta la petición de usuario (y ejecuta el programa cgi).

Formato: nombre/versión.

#### **SERVER\_NAME**

Devuelve nombre de host del servidor, el alias DNS, o la dirección IP como aparecería en las URL autoreferenciadas.

#### **GATEWAY\_INTERFACE**

Devuelve la revisión de la especificación CGI con que el servidor puede trabajar.  
Formato: CGI/revisión.

Las siguientes variables de entorno son específicas de la petición de usuario, y es el programa del *gateway* el que las da el valor:

#### **SERVER\_PROTOCOL**

Da el nombre y revisión del protocolo de información con el que la petición de usuario viene. Formato: protocolo/revisión.

#### **SERVER\_PORT**

Devuelve el número de puerto por el cual fue enviada la petición.

#### **REQUEST\_METHOD**

Devuelve el método por el cual la petición fue enviada. Para HTTP serán "GET", "HEAD", "POST", etc.

#### PATH\_INFO

La información extra sobre el *path*, tal como es dada por el cliente. En otras palabras, podemos acceder a los scripts por su *pathname* virtual, seguido de alguna información extra. Esa información extra es enviada como PATH\_INFO. La información será decodificada por el servidor si viene de una URL antes de pasarla al script CGI.

#### PATH\_TRANSLATED

El servidor proporciona una versión traducida del PATH\_INFO, que transforma el *path* virtual al físico.

#### SCRIPT\_NAME

Path virtual al script que va a ejecutar, usado para autoreferenciar URL.

#### QUERY\_STRING

La información que sigue al signo "?" en la URL que referencia al script. Es la información de la pregunta. No deberá ser decodificada de ningún modo. Esta variable será activada cuando hay una petición de información, sin hacer caso de la decodificación de la línea de comandos.

#### REMOTE\_HOST

El nombre de *host* que realiza la petición. Si el servidor no posee esta información activará REMOTE\_ADDR y dejará esta desactivada.

#### REMOTE\_ADDR

La dirección IP del *host* remoto que realiza la petición.

#### AUTH\_TYPE

Si el servidor soporta autenticación de usuario, y el script está protegido, esta es el método de autenticación específico del protocolo para validar el usuario.

#### REMOTE\_USER

Si el servidor soporta autenticación de usuario, y el script está protegido, este será el nombre de usuario con el que se ha autenticado.

#### REMOTE\_IDENT

Si el servidor HTTP soporta autenticación RFC 931 , entonces esta variable se activará con el nombre del usuario remoto obtenido por el servidor. Esta variable solo se utilizará durante el *login*.

#### CONTENT\_TYPE

Para peticiones que tienen información añadida, como HTTP POST y PUT, este será el tipo de datos contenido.

#### CONTENT\_LENGTH

La longitud del contenido tal como es dado por el cliente.

Además, las líneas de la cabecera recibidas por el cliente, si las hay, son colocadas en el entorno con el prefijo HTTP\_ seguido del nombre de la cabecera. Cada carácter de el nombre de la cabecera se cambia por caracteres \_. El servidor puede excluir algunos caracteres que ya haya procesado, como la autorización.

El tipo de contenido y la longitud de este, pueden ver suprimidas sus cabeceras si al incluirlos se excede el límite de entorno del sistema.

Un ejemplo de esto es la variable HTTP\_ACCEPT que se definió en CGI/1.0. Otro ejemplo es la cabecera USER\_AGENT.

#### HTTP\_ACCEPT

Los tipos MIME que el cliente aceptará, como son dados por las cabeceras HTTP. Otros protocolos pueden ser necesarios para obtener esa información de algún otro lugar. Cada elemento de esta lista deberá estar separado por comas por la especificación HTTP.

Formato: tipo/subtipo, tipo/subtipo

#### HTTP\_USER\_AGENT

El navegador que el cliente usa para mandar la petición.

Formato general: software/versión librería/versión.

## **LEYENDO EL FORMULARIO DE ENTRADA DE USUARIO**

Cuando el usuario envía el formulario, el script recibe los datos como pares nombre-valor. Los nombres son lo que definimos en las etiquetas INPUT (o las etiquetas SELECT o TEXTAREA), y los valores aquello que el usuario haya escrito o seleccionado. (Los usuarios también pueden enviar ficheros con los formularios, pero no nos ocuparemos de ello.)

Estos pares nombre-valor llegan como una larga cadena que necesitamos formatear. No es muy complicado, hay una gran cantidad de rutinas que lo hacen por tí. En el directorio CGI de Yahoo encontrarás unas cuantas en varios lenguajes.

Si aun así prefiere hacerlo usted mismo, aquí esta el formato de la cadena:

“nombre1=valor1&nombre2=valor2&nombre3=valor3”

Así que sólo hay que dividir donde están los signos ‘&’ y ‘=’, y luego hacer dos cosas a cada nombre y valor:

1. Convertir todos los signos ‘+’ a espacios.
2. Convertir todas las secuencias ‘%xx’ al valor del carácter cuyo valor ASCII sea ‘xx’ en hexadecimal. Por ejemplo convertir ‘%3d’ a ‘=’.

Esto se hace necesario porque la larga cadena original esta codificada según el código URL, para permitir los signos ‘&’, ‘=’, y todo lo que el usuario introduzca.

Pero, ¿de donde se obtiene la cadena de entrada? Esto dependerá del método por el cual el formulario haya sido enviado:

Para los envíos con GET, será la variable de entorno QUERY\_STRING.

Para los envíos con POST, habrá que leer delSTDIN. El número exacto de bytes a leer estará en la variable de entorno CONTENT\_LENGTH.

## **DEVOLVIENDO LA RESPUESTA AL USUARIO**

Primero, escribir la línea

*Content-Type: text/html*

más otra línea en blanco en el STDOUT. Después, escribiremos nuestra página de respuesta en HTML al STDOUT, y será enviada al usuario cuando el script esté ejecutado.

Sí, estamos generando código en HTML en tiempo real. No es difícil, si no más bien directo. El código HTML fue diseñado lo suficientemente simple para poder ser generado por este método.

## GUARDANDO LA ENTRADA DEL USUARIO EN UN ARCHIVO

En este apartado vamos a guardar los datos escritos por el usuario en un archivo para poder recuperarlos posteriormente. Como ejemplo utilizaremos un formulario para enviar chistes. Este formulario obtendrá la siguiente información y lo guardará en las variables indicadas entre paréntesis:

Nombre del Usuario (nombre)

Un Chiste (chiste)

Para ello creamos un formulario simple con el siguiente código:

```
<form method="POST" action="http://www.uva.es/cgi-bin/chiste-envio.pl">
<P>Tu nombre: <input name="nombre"></P>
<P>El chiste: <textarea cols=60 rows=5 name="chiste"></textarea></P>
<P><input type="submit" value="Enviar"><BR>
<input type="reset" value="Borrars"></P>
</form>
```

Nuestro programa colocará esta información para que la podamos dar el visto bueno. El programa devolverá una nota al usuario indicándole que su envío será revisado más tarde.

Por consistencia , llamaremos a este script chiste-envio.pl.

Necesitamos iniciar y comentar nuestro script. Utilizaremos la librería cgi-lib.pl de Steve Brenner's para facilitar la entrada de formularios..

Seguiremos con el script que coja la entrada y devuelva e siguiente mensaje:

Gracias \_nombre\_, tu envío será revisado en breve.

El script que realiza esto es el siguiente:

```
#!/usr/local/bin/perl
#chiste-envio.pl
# Por Javier Pérez Delgado (jperez@ctv.es)
#
# Este script añade el chiste al archivo de chistes.
# Devuelve una nota al usuario, indicando que su chiste será revisado.

push(@INC,"/p/www/httpd/cgi-bin");
require("cgi-lib.pl");

&ReadParse;

print &PrintHeader;
print "<HTML><HEAD>\n";
print "<TITLE>Reconocimiento de chiste<TITLE>\n";
```

```
print "</HEAD><BODY>\n";
print "Gracias ",$sin{'nombre'},",Tu envío será revisado lo antes posible. .<P>\n ";
print "</BODY></HTML>\n";
```

Ahora al tema central, añadir a un archivo. Para hacerlo necesitamos un archivo con permiso de escritura para el *daemon http*. La creación de este archivo variará entre distintos sistemas, y necesitaras consultar a tu webmaster. En este caso usaremos el archivo:

```
/users/jperez/www/tutor/chiste.txt
```

Para abrir el fichero para añadir al final usaremos el comando de perl para abrir ficheros, que tiene la siguiente sintaxis:

```
open(FILEHANDLE,">>nombearchivo");
```

FILEHANDLE es como llamaremos al archivo mientras esté abierto. Es similar a una variable. Nombearchivo es el archivo a abrir. Los signos ‘mayor’ (>>) son importantes, y como en el shell indican ‘añadir a un fichero’.

Así la orden de apertura quedará así:

```
open(CHISTE,">>/users/jperez/www/tutor/chiste.txt");
```

Nota: Es aconsejable usar letras mayúsculas para FILEHANDLE para ayudar a distinguirlo de las variables.

Escribir a un fichero es idéntico a como hicimos previamente. La única diferencia es que el FILEHANDLE es el primer argumento de la sentencia *print*, y está separado de lo que será imprimido por un espacio. Por ejemplo para escribir la línea “Las rosas son rojas” al FILEHANDLE CHISTE, escribiremos:

```
print CHISTE "Las rosas son rojas\n";
```

Escribir variables se realiza de la misma manera:

```
print CHISTE "Las rosas son ",$sin{'colorrosas'},"\n";
```

El código para nuestro ejemplo necesitará escribir al archivo:

1. El nombre del usuario precedido con la cadena “Nombre:”
2. Una línea en blanco.
3. El chiste enviado.
4. Una línea con 50 guiones.

El código es el siguiente:

```
print CHISTE "Nombre: ",$sin{'nombre'},"\n";
print CHISTE "\n";
print CHISTE $sin{'chiste'},"\n";
print CHISTE "-----\n";
```

Ahora solo nos queda cerrar el fichero, y esto lo hacemos con el comando:

```
close FILEHANDLE;
```

## DEVOLVIENDO UNA PÁGINA QUE NO HEMOS GENERADO

En este apartado crearemos un libro de visitas sencillito. Durante el proceso de realización, haremos bastantes cosas de forma ordenada.

Devolveremos una página html que nosotros no habremos creado en tiempo real. Esto se hace principalmente para demostrar la directiva de localización.

Hay que verificar que todos los campos tienen una entrada, y que la dirección de correo electrónico tiene un símbolo arroba (@).

Editar un documento html con un script, insertando información en el medio.

Vamos a empezar creando un formulario que coja la siguiente información en las variables indicadas entre paréntesis.

Nombre (nombre)  
email (email)  
comentarios (comentarios)

Colocamos un comentario en HTML que contenga la cadena "INSERTAR AQUÍ" en el formulario donde vayamos a insertar las nuevas entradas.

`<!--INSERT HERE-->`

Este formulario llamará al script, *guestbook.pl*. El formulario *guestbook.html* será el siguiente:

```
<!DOCTYPE HTML PUBLIC "-//IETF/DTD HTML//EN">
<html>
<!--Copyright 1996 Javier Pérez (jpereztv.es)-->

<head>
<title>Libro de visitas</title>
</head>

<body>

<h1>Firma el formulario</h1>

<P>Por favor, rellena todos los campos:</P>

<hr>

<form method=POST action="http://www.ctv.es/cgi-bin/guestbook.pl">

<P><input name="nombre"><BR>

Nombre</P>

<P><input name="email"><BR>
```

```
Dirección e-Mail [Debe tener una arroba @]</P>
<P><textarea rows=3 cols=70 name="comentarios"></textarea><BR>
Comentarios:</P>
<P><input type="submit" value="Añadir tus comentarios"><BR>
<input type="reset" value="Borrar formulario"></P>
</form>
<HR>
<!--INSERT HERE-->
<P>Última modificación 21/9/96
</body>
</html>
```

Como se puede ver, los nombres serán añadidos al final del formulario.

Estos son los pasos para realizar nuestra tarea:

1. Verificar la entrada, devolviendo un mensaje de error si hay un problema.
2. Uraremos un bloqueo de archivo denominado 'del pobre'. Creando un nuevo archivo denominado. Si este archivo existe el programa se detendrá durante un segundo y lo volverá a intentar.
3. Abriremos y leeremos el archivo del libro de firmas actual: *guestbook.html*.
4. Borraremos este archivo, y escribiremos un nuevo fichero que consistirá en la antigua información con toda la nueva en el sitio apropiado. Usaremos el comentario `<!--INSERTAR AQUÍ-->` como marcador donde añadir entradas.

Nota: Puede ser deseable hacer una copia de seguridad del archivo, antes de borrarlo. No lo hacemos en este ejemplo, pero lo recomiendo en cualquier desarrollo. Se podrá hacer por cualquier método, incluido una llamada al sistema para copiar (cp).

5. Retornar una directiva de lugar apuntando a una página de agradecimiento que ya hayamos preparado.
6. Desbloquear borrando el archivo `.guestlock`.

Obviamente, este no es el mejor libro de visitas, pero servirá como un buen ejemplo, y es fácilmente ampliable.

Empecemos:

1. Como es habitual nuestro script deberá empezar con una llamada al Perl, los comentarios apropiados, y una llamada a la rutina ReadParse. No lo detallaremos aquí, ya que se hizo anteriormente.
2. Para empezar debemos verificar los datos de entrada. Nos queremos asegurar de que no hay campos en blanco, y de que la dirección e-Mail contiene una arroba (@). Para ventaja nuestra, las cadenas nos vienen dadas sin los espacios en blanco, justo igual que los navegadores cuando presentan páginas html. Esto significa que solo debemos comprobar que ninguna de las cadenas es igual a la cadena vacía, "". En perl, como en muchos lenguajes, las cadenas se comparan de manera diferente a los números. Usaremos 'eq' para comprobar la igualdad de cadenas, y el signo '=' para comprobar la igualdad de números. Usaremos 'ne' y '!=' respectivamente para las comparaciones 'no igual'.

Podemos hacerlo de la siguiente manera:

```
if ($in{'nombre'} eq "") {  
    # Hay una cadena vacía, devolver mensaje de error  
  
    # salimos al final del script ahora, tenemos un error  
    exit;  
}
```

Como podrás adivinar, Ahora no ponemos el código para devolver el mensaje de error, en el ejemplo de arriba. Aún así muestra lo facil que es comprobar que un campo está vacío.

Nota: Los paréntesis () y las llaves ({} ) son importantes. Los paréntesis contienen lo que se está comprobando como verdadero o falso, y las llaves contienen las sentencias a ejecutar si es verdadero.

Como vamos a realizar varias comprobaciones de cadenas vacías, y vamos a devolver el mismo error para todas ellas, podemos incluirlas en la misma sentencia if-then uniéndolas por OR logicos ||, en perl. El OR significa que alguna de las comprobaciones tienen que ser verdad, para que toda la sentencia sea cierta.

Un ejemplo de comprobar si tres variables no están vacias podría ser este:

```
if (($in{'nombre'} eq "") || ($in{'email'} eq "") || ($in{'comentarios'} eq "")) {  
    # algun campo está en blanco, devolver mensaje de error  
  
    # salimos al final del script ahora, tenemos un error  
    exit;  
}
```

De nuevo he omitido, el código del mensaje de error, ya que es sencillo construir una pequeña tabla con los códigos para devolver un código de error útil para el usuario. El código que he elegido para hacerlo aparece más tarde en el código final de guestbook.pl.

La última verificación que necesitamos realizar es comprobar que la dirección eMail del usuario contiene un signo (@). Lo realizaremos fácilmente con las potentes expresiones regulares que el perl proporciona.

Una expresión regular sencilla para chequear esto sería `^\w*@\w*$`. Simplemente describe una situación donde una palabra aparece delante y detrás de un símbolo `@`. Las expresiones regulares son una herramienta extremadamente potente, pero no entran dentro del dominio de esta lección.

Para llevar a cabo la comprobación de que la expresión se cumple, simplemente comparamos. Funcionará como las comparaciones normales, pero usando el símbolo `'=~'` para igualdad y el símbolo `'!~'` para desigualdad. La expresión regular podría codificarse así:

```
if($in{'email'} !~ ^\w*@\w*$) {  
    # La expresión regular no es igual, devolver mensaje de error.  
}
```

De nuevo, he omitido el mensaje de error para este código.

3. Ahora comprobaremos la existencia de nuestro archivo de bloqueo. Si existe, nos detendremos por un segundo para un segundo intento. Si no existe deberemos crearlo, bloqueando de este modo el archivo. Para hacerlo necesitaremos un bucle continuo sobre una sentencia *if-then-else* que compruebe la existencia del archivo, y que terminará cuando hayamos añadido la entrada.

Para el bucle en perl usaremos la sentencia *while*. Esta sentencia ejecuta repetidamente lo que está entre llaves hasta que la condición entre paréntesis es falsa. Un ejemplo para clarificarlo:

```
while (condición) {  
    # Código a ejecutar mientras la condición sea cierta  
}
```

Las condiciones son del mismo tipo de las que pondríamos en una sentencia *if-then*. Vamos a usar de hecho una sentencia *if-then* con la cláusula *else* en este bucle. Como ya sabemos las sentencias *if-then-else* ejecuta el *then* cuando la condición es cierta y el *else* cuando es falsa.

```
if(condición) {  
    # Código a ejecutar si la condición es CIERTA  
}  
else {  
    # Código a ejecutar si la condición es FALSA  
}
```

Para comprobar la existencia de un archivo, usaremos el operador `'-e'` del nombre del archivo. No se debe olvidar que el archivo de bloqueo debe estar en un lugar donde el `httpd` pueda escribir. En este caso lo colocaremos en el directorio `users/jperez/www/tutor/`. Si el fichero existe, nos detendremos durante un segundo, y lo volveremos a intentar. El lenguaje perl tiene un comando de espera con la forma:

```
sleep(# segundos).
```

Aquí tenemos como debemos realizar el chequeo, con el comando de espera. Todo ello queda incluido en un bucle while que comprueba que la variable quitar vale 1 antes de salir.

```
$quitar = 0;
while ($quitar != 1) {
    if(-e "/users/jperez/www/tutor/.guestlock") {
        # El archivo existe. Esperaremos un segundo
        sleep(1);
    }
    else {
        # El archivo no existe, haremos el trabajo para añadir la entrada.
        # Aquí deberemos dar a $quitar valor 1.
    }
}
```

Para crear el archivo de bloqueo, simplemente deberemos abrir y cerrar el archivo. Así crearemos un archivo de 0 bytes de longitud, que servirá en nuestro chequeo, y bloqueará otros procesos que la gente pueda realizar en ese momento, hasta que este haya acabado. Para ello esaremos un trozo de código que aquí se detalla:

```
open(LOCK,">/users/jperez/www/tutor/.guestlock");
close LOCK;
```

4. Ahora tenemos un bloqueador del fichero. Ahora deberemos abrir y leer todos los contenidos del *guestbook.html*. Esto se hace fácilmente abriendo el archivo para lectura. En el comando open antes empleado usaremos el símbolo (<) donde antes usamos (>>). El resultado será:

```
open(FILEHANDLE,"<nombrefichero");
```

El código será::

```
open(GB,"<users/jperez/www/tutor/guestbook.html");
```

Leer del fichero es igual de fácil. Perl permite usar el FILEHANDLE entre los símbolos (<>) para sustituir a la próxima línea del fichero.

Así pues, podemos poner algo como esto:

```
$línea = <GB>;
```

Así leeremos una línea del fichero apuntado por GB, y avanzará el puntero a la siguiente línea automáticamente. Esto significa que la siguiente sentencia como esta leerá la siguiente línea. Podremos leer el archivo entero en un bucle, saliendo solo cuando no quede nada más. Pero hay un modo más fácil, podemos usar un vector. Un vector es una variable, con mucha información que puede ser accedida individualmente. Ya hemos usado arrays

asociativos en la construcción `$in{'variable'}`. Recuerda que se tiene el elemento etiquetado como variable en el vector asociativo `$in`. Usaremos un vector ordenado numericamente. Esto significa que la información es almacenada en el vector en el orden en la que la ponemos, y podemos referenciar el elemento n-ésimo elemento añadido, donde n es un número cualquiera. Estos vectores son referenciados con el signo @, y se usan de manera similar a los vectores asociativos cuando queremos recuperar un solo elemento de él. El siguiente ejemplo lo explica:

```
@pepe          # Este es el vector pepe entero, con todos sus elementos
$pepe[4]       # El el quinto elemento del vector pepe. Notar que
               # se empieza a contar desde 0, y que por tanto:
$pepe[0]       # es el primer elemento del vector. (Igual que en el lenguaje C)
```

Perl proporciona un modo rápido para llenar un vector con el contenido de un archivo. Podemos usar la sentencia `@vector = <FILEHANDLE>`. Se leerá cada línea del fichero apuntado por FILEHANDLE, y serán colocados secuencialmente en el vector. Para leer el libro de visitas entero, deberemos usar:

```
@lineas = <GB>;
```

No debemos olvidarnos de cerrar el fichero:

```
close GB;
```

5. Vaciar el archivo es muy sencillo. Simplemente debemos reabrirlo para escritura sin añadir. Esto significa usar un '>' en vez de dos. Una vez hecho esto debemos imprimir cada línea del fichero, sustituyendo lo que el usuario escribió justo delante del marcador.

Para llevar a cabo la sustitución usaremos la función `s/oldpattern/newpattern/`, que reemplaza el patrón antiguo con el nuevo. La buena noticia es que podemos usar metacaracteres como `\n` para reemplazar varias líneas. Para que esto afecte a una variable que contiene una cadena usaremos un operador. La sentencia quedará así:

```
$linea =~ s/oldpattern/newpattern/
```

Realmente haremos una sustitución en los comentarios, para convertir las nuevas líneas en comandos `<BR>`, de manera que las líneas queden tal y como el visitante las escribió. Para hacerlo pondremos:

```
$in{'comentarios'} =~ s/\n/<BR>\n/go;
```

La g que va detrás del patrón significa que se hará esto para cada línea en la variable, la o significa compilar esta expresión regular de manera que concuerde más rápido. Es bueno hacerlo.

Para realizar la sustitución y añadir los comentarios del usuario para cada línea del vector, usaremos un bucle `foreach`. Esto significa para cada elemento de la lista, coloca el

elemento en la variable `_de_bucle`, y ejecuta las sentencias que están entre las llaves (`{ }`). Esto sería:

```
foreach $variable_de_bucle (lista) {
    #cosas a hacer
}
```

Queremos insertar el nombre de la persona con su dirección e-Mail entre paréntesis, seguido de sus comentarios. Esto lo incluiremos en un párrafo HTML, seguido de un comando `<HR>`. No tenemos que olvidarnos de incluir un nuevo marcador `<!--INSERTAR AQUÍ-->`. Lo pondremos primero así las nuevas entradas serán añadidas al principio de la página. Mi código será:

```
open(GB,">/users/jperez/www/tutor/guestbook.html");
foreach $linea (@lineas) {
    $linea =~ s/<!--INSERTAR AQUÍ-->/<!--INSERTAR AQUÍ-->\n<P>Nombre:
    $in{'nombre'}($in{'email'})<BR>\nComentarios:<BR>\n$in{'comentarios'}</P>\n<HR>\n/o;
    print GB $linea;
}
close GB;
```

6. Ahora que hemos reescrito el archivo, debemos desbloquearlo, borrando el archivo `.guestlock`. Es sencillo, ya que perl tiene un buncion borrar:

```
unlink(lista of ficheros);
```

Así tendremos que poner:

```
unlink("/users/jperez/www/tutor/.guestlock");
```

7. El ultimo paso es apuntar a mi página de agradecimiento. Y poner la variable `$quitar = 1`; . HTTP nos permite hacer esto fácilmente con la directiva de localización. Simplemente devolvemos la línea:

```
Location: url
```

```
Instead of:
```

```
Content-Type: text/html
```

Esto lo resolvemos con la siguiente sentencia `print`:

```
print "Location: http://www.ctv.es/users/jperez/www/tutor/agradecimiento.html\n\n";
```

La nueva segunda línea es significativa. Los códigos de retorno Mime, de los cuales la localización es uno de ellos, requiere dos nuevas líneas siguiéndolo para trazar el fin del código de retorno. Olvidar la nueva segunda línea provocará un daño irreparable.

Poner la variable 'quitar' a 1 es trivial:

```
$quitar = 1;
```

El código final para el archivo guestbook.pl será:

```
#!/usr/local/bin/perl
# Por Javier Pérez Delgado (jperez@ctv.es) 6-9-96
#
# guestbook.pl
#     Procesa las entradas a un libro de firmas
#
# Usaremos la librería cgi-lib.pl para manejar la entrada

push(@INC,"p/www/httpd/cgi-bin");
require("cgi-lib.pl");

&ReadParse;

if ((${'nombre'} eq "") || ({'email'} eq "") || ({'comentarios'} eq "")) {
# algún campo en blanco, devolver mensaje de error
print &PrintHeader;
print "<HTML>\n";
print "<HEAD>\n";
print "<TITLE>Mensaje de error</TITLE>\n";
print "</HEAD>\n";
print "<BODY>\n";
print "\n";
print "<HI>Ha ocurrido un error</HI>\n";
print "\n";
print "<P>No has completado todos los campos Por favor vuelve a intentarlo.</P>\n";
print "</BODY>\n";
print "</HTML>\n";

# llamda para salir del script. Tenemos un error
exit;
}

if ({'email'} !~ /\w*\@\w*/) {
# No es igual a la expresión regular. Enviar mensaje de error
print &PrintHeader;
print "<HTML>\n";
print "<HEAD>\n";
print "<TITLE>Mensaje de error</TITLE>\n";
print "</HEAD>\n";
print "<BODY>\n";
print "\n";
print "<HI>Ha ocurrido un error</HI>\n";
print "\n";
print "<P>Tu dirección email no contiene una @.</P>\n";
print "</BODY>\n";
print "</HTML>\n";
}
```

```

# llamada para salir ahora del script, tenemos un error
exit;
}

$quitar = 0;
while ($quitar != 1) {
  if (-e "/users/jperez/www/tutor/guestlock") {
    # El fichero existe, esperamos un momento
    sleep(1);
  }
  else {

    # Creamos el fichero de bloqueo, de manera que el libro de firmas queda bloqueado.
    open(LOCK, ">/users/jperez/www/tutor/guestlock");
    close LOCK;

    # Abrimos y leemos el antiguo libro de firmas
    # Nota: Crear una copia de seguridad no sería una mala idea ...
    open(GB, "</users/jperez/www/tutor/guestbook.html");
    @lineas = <GB>;
    close GB;

    # Preparamos los comentarios para html
    $in{'comentarios'} =~ s/\n<BR>\n/go;

    # Vaciamos el libro de firmas antiguo, y lo volvemos a imprimir, añadiendo la nueva entrada
    open(GB, ">/users/jperez/www/tutor/guestbook.html");
    foreach $linea (@lineas) {
      $linea =~ s/<!--INSERTAR AQUÍ-->/<!--INSERTAR AQUÍ-->\n<P>Nombre:
      $in{'nombre'}($in{'email'})<BR>\nComentarios:<BR>\n$in{'comentarios'}</P>\n<HR>\n/o;
      print GB $linea;
    }
    close GB;

    # desbloquear el archivo
    unlink("/users/jperez/www/tutor/guestlock");

    # Devolver el nuevo libro de firmas, y poner $quitar = 1
    print "Location: http://www.ctv.es/users/jperez/www/tutor/thanks.html\n\n";
    $quitar = 1;
  }
}

```

Nota sobre el bloqueo de archivos:

El mecanismo de bloqueo de archivos usado arriba, no es perfecto ya que requiere varios pasos para bloquear un archivo. Por ello es posible obtener un bloqueo o desbloqueo falso, aunque es altamamente improbable. Una implementación mejot usara el sistema de ficheros nativo bloqueando, vinculando, o algún otro método atómico. Desconozco si perl simula el bloqueo en sistemas que no implementan el bloqueo de forma nativa.

Un método usando vínculos sin comentar sería:

```
$quit = 0;
```

```
while ($quit != 1) {
    /users/jperez/www/tutor/guestbook.html
    if (link(/users/jperez/www/tutor/guestbook.html,/users/jperez/www/tutor/.guestlock)) {
        # Archivo bloqueado, nos detenemos un momento
        sleep(1);
    }
    else {
        # Tenemos un bloqueo, hacemos lo que tengamos que hacer
        unlink("/users/jperez/www/tutor/.guestlock");
    }
}
}
```

## SCRIPTS CGI QUE ENVÍAN CORREO

En este apartado crearemos un script CGI que envíe correo, y devuelva una página que indique que el correo ha sido enviado. Sin embargo, los conceptos serán bastante generales para permitir que el script pueda ser adaptado a cualquier proyecto donde sea necesario enviar un correo desde un script.

También veremos brevemente como examinar las áreas de texto línea por línea.

Como siempre, cuando empezamos, necesitaremos un formulario.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

<html>

<!--Javier Pérez Delgado (jperez@ctv.es) All Rights Reserved -->

<head>

<link rev=made href="mailto:jperez@ctv.es">

<title>Enviar correo desde un script CGI</title>

</head>

<body>

<P>Este formulario encía correo a la dirección de la persona mostrada.</P>

<form method="POST" action="http://www.ctv.es/USERS/cgi-bin/mail.pl">

<P>Tu dirección: <INPUT NAME="De" SIZE=36></P>

<P>Tu URL: <INPUT NAME="xurl" SIZE=36></P>

<P>Subject: <INPUT NAME="subject" SIZE=40></P>

<P>Mensaje:</P>

<P><TEXTAREA name="body" rows=10 cols=60></TEXTAREA></P>
```

```
<P><input type="submit" value="Enviar mensaje">
<input type="reset" value="Borrar todo"></P>
</FORM>
</P>
</body>
</html>
```

Ahora pasaremos al script. Como siempre deberá empezar con una llamada al perl, algunos comentarios y una llamada a la rutina `ReadParse`, en la librería `cgi-lib.pl`. También haremos verificaciones de que el cuerpo del mensaje no está vacío, y de que la dirección contiene una arroba (@). Ya sabemos hacerlo, por que ya lo realizamos en el anterior apartado. Otra validación será para comprobar que se pasan argumentos, lo pondremos porque la gente tiene tendencia a llamar a los scripts de correo sin argumentos (eg: no desde un formulario), no entiendo por qué.

Para comprobar que no hay argumentos, simplemente examinaremos si el vector `%in` que la librería `cgi-lib.pl` nos devuelve tiene alguna tecla. Recuerda, que ningún argumento, significa que no nos han pasado nada, sin embargo, alguien puede pasar los argumentos en blanco, de aquí las dos comprobaciones. La manera de comprobar los valores de las teclas del vector es usar la función `keys()`. Esta función espera un array asociativo como argumento. Simplemente comprobaremos que devuelve algo, imprimiendo un mensaje de error si no hay teclas. El código puede ser este:

```
if(!keys(%in)){
  # ninguna tecla ha sido pasada, imprimir mensaje de error, y si es apropiado salir.
}
```

Date cuenta de que el signo '!' al principio de la llamada a `keys()`, significa 'no' o negación. Quiere decir que si `keys()` no devuelve nada (falso), será negado para que sea cierto y el `if` se ejecute.

Ahora que hemos validado las entradas, y se pueden realizar más comprobaciones, necesitaremos enviar la carta. Para ello necesitaremos un programa que acepte una carta del `stdin`. Luego usaremos `sendmail` en el ejemplo. Si no estás en un entorno Unix, necesitarás otro programa apropiado para hacerlo. Como no conozco otro tipo de sistemas, no puedo hacer ninguna recomendación.

Usar este programa será similar a escribir en un fichero. Necesitamos abrir el programa para aceptar la entrada, escribir al `FILEHANDLE`. Abrir un programa que espera entradas por el `stdin` es bastante fácil en Perl. Además, es muy fácil pasar el argumento de la línea de comandos. En este ejemplo, abriremos `sendmail`, y diremos que busque en la carta la dirección de destino.

```
open(MAIL, "/usr/lib/sendmail -t");
```

Como se puede observar, es igual que una llamada a fichero, pero los símbolos '>' o '<' han sido sustituidos por un '|' (tubería). Esto indica que lo que hay detrás de la tubería es un ejecutable, y que lo que imprimamos en este *FILEHANDLE* se deberá pasar como entrada a el programa ejecutable.

Nota: No he comprobado si la operación anterior terminó con éxito, pero deberíamos hacerlo. La manera más sencilla de hacerlo, es apoyarnos en que el comando *open* devuelve *true* cuando ha tenido éxito. Solo debemos hacer un *OR* entre el comando *open* y otro comando conociendo que ese comando será ejecutado si el *open* falla. El ejemplo de abajo realiza un *OR* con el comando *die*. Este comando imprime un mensaje de error y sale del programa.

```
open(MAIL, "/usr/lib/sendmail -t") || die "La llamada a sendmail ha fallado";
```

El imprimir la carta funciona como esperamos. Recuerda, sin embargo, que estamos imprimiendo un trozo de e-Mail, por lo que deberemos poner las cabeceras adecuadas, un línea en blanco y el cuerpo del mensaje. Puedes echar un vistazo a un e-Mail que hayas recibido y comprobar las cabeceras. Deciré abajo lo más importante. Los comentarios serán de ayuda.

```
# Esta es la línea 'para'. Si tenemos el nombre y la dirección, escribiremos
# primero el nombre y luego la dirección entre <>
print MAIL "To: ${De}\n";

# Esta es la línea 'De'. Pondremos el mismo nombre en la línea 'De' y en la 'para'
# Recuerda que no hay seguridad comprobando el email, por ello estas líneas
# pueden ser falsificadas. NO FALSIFIQUES EL EMAIL, no es divertido y en algunos lugares
# es un delito.
print MAIL "From: ${De}\n";

# Esta es la línea 'contestar a'. Esta línea se incluye debido a que algunos programas
# son bastante tonotos y no siempre responden a la línea 'De'.
# Al menos respetarán esta línea.
print MAIL "Reply-To: ${De}\n";

# Lo siguiente son cabeceras X. Son creadas por el usuario y pueden contener
# todo lo que desees. Incluyo una línea de descripción
# también he escrito las líneas REMOTE_HOST, REMOTE_ADDR, y REMOTE_USER para
# ayudar al seguimiento (tracking) de la carta.
# Solo escribo la línea X-URL si el usuario ha dado una url. Este tipo
# de comprobación probablemente hecha también en el X-Remote-Host y X-Remote-User

print MAIL "X-mailer: Mail.pl, a cgi-bin script at http://www.ctv.es/users/jperez/www/tutor/index.html\n";
print MAIL "X-Remote-Host: ${ENV{REMOTE_HOST}} (${ENV{REMOTE_ADDR}})\n";
print MAIL "X-Remote-User: ${ENV{REMOTE_USER}}\n";
print MAIL "X-disclaimer: La línea De: puede estar falsificada ";
print MAIL "No confiar en un 100% sobre la integridad de este mail. ";
print MAIL "No somos responsables de este correo de ninguna manera\n";
if (${xurl} ne "") {
    print MAIL "X-URL: ${xurl}\n";
}

# Finalmente escribimosly we print the famous subject line. I have appended a string
```

```
# identifying this as WWW generated email, this is far from a requirement.  
# Notice the second new line. This is the blank line that will separate the  
# headers from the body. All mail must have this line.  
print MAIL "Subject: ${subject} (WWW generated email)\n\n";
```

```
# Now we are going to print the body. Because it was input into a TEXTAREA  
# field, it has new lines after each line. We can just print it like any other  
# field, knowing that the new lines will expand it properly.  
print MAIL ${body};
```

Solo quedan dos cosas. Primero debemos cerrar la conexión con *sendmail*. Segundo debemos imprimir una página enseñando al usuario la carta que envía.

Para cerrar la conexión, usaremos el comando *close*, justo como en cualquier otro manejador:

```
close(MAIL);
```

Imprimir la respuesta no es diferente a las otras páginas generadas dinámicamente que ya hemos creado anteriormente. Debemos dividir el campo *TEXTAREA* en líneas colocando un `<BR>` al final de cada una. Esto se hace principalmente para demostrar como se divide un campo de texto. Recuerda sin embargo que *html* no respeta las nuevas líneas. Con ello se consigue el poder hacer los párrafos de manera más clara, pero el beneficio de hacerlo es cuestionable.

El valor del campo *body* (cuerpo del mensaje), no es más que un conjunto de frases separadas con un retorno de carro. Podemos usar la función `split()` para separarlas. Esta función necesita dos parámetros: la cadena o carácter para separar y la variable a separar, y retorna un array con los elementos separados. Usaremos *split* en un bucle *foreach*:

```
foreach $l (split('\n', ${body})) {  
  print "$l<BR>\n";  
}
```

Como se puede ver separa la variable `${body}` en partes cada nueva línea, y la imprime seguida de un `<BR>`.

## **ESCRIBIENDO SCRIPTS CGI SEGUROS**

Siempre que un programa interactúa con un cliente por red, existe la posibilidad de que el cliente atque al programa para conseguir un acceso. El script más inocente puede ser muy peligroso para la integridad de tu sistema.

Teniendo eso en cuenta, me gustaría comentar unos pequeños consejos para conseguir que tu programa no sea atacado.

### **Cuidado con la sentencia *eval***

Lenguajes como el *Perl* y el *Bourne shell* tienen un comando *eval* que permiten construir una cadena y dejar al intérprete que la ejecute.

Esto puede resultar peligroso. Observa la siguiente sentencia en *Bourne shell*:

```
eval `echo $QUERY_STRING | awk 'BEGIN{RS="&"} {printf "QS_%s\n",$1}'`
```

Esta sentencia coge la cadena de entrada y la convierte en un conjunto de comandos de declaración de variables. Desafortunadamente este script puede ser atacado mandando una cadena de entrada que empiece por `;`.

### **No confíes en que el cliente haga algo**

Un cliente correcto evitará todos los caracteres que tienen un significado especial para el *Bourne Shell* en una cadena de entrada, y que hega que tu *script* malinterprete los caracteres. Un cliente malintencionado usará esos caracteres para confundir a tu script y ganar acceso desautorizado.

### **Cuidado con *popen()* y *system()***

Si usas datos del cliente para construir una llamada a *popen()* o *system()*, asegurate de poner un backslash delante de cada carácter que tenga un significado especial para el *Bourne Shell* antes de llamar a la función. Lo podrás hacer con una sencilla función en C.

Si quieres más información sobre la seguridad en el WWW, consulta WWW Security FAQ en <http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html>.