

PROGRAMACIÓN WEB

PARA PROGRAMADORES

HTTP HTML CSS JAVASCRIPT JQUERY PHP MYSQL

MAURO GULLINO

PROGRAMACION WEB PARA PROGRAMADORES

Gullino, Mauro

Programación web para programadores. - 1a ed. - Buenos Aires : el autor, 2014.
192 p. ; 25x18 cm.

ISBN 978-987-33-4505-0

1. Programación. 2. Internet. I. Título

CDD 004

Fecha de catalogación: 20/02/2014

© Mauro Gullino, 2014.

maurogullino.com.ar/pwpp

Las familias tipográficas utilizadas en el interior son Poly (*Nicolás Silva*), Exo 2 (*Natanael Gama*) y Envy Code R (*Damien Guard*).

Hecho el depósito que prevé la ley 11.723.
Reservados todos los derechos.

Impreso en La Imprenta Digital SRL
Melo 3711, Florida, Provincia de Buenos Aires

PROGRAMACION WEB

PARA PROGRAMADORES

MAURO GULLINO

OBJETIVOS

Este material fue pensado para introducir los conceptos fundamentales de la programación web. Estos conceptos *básicos* constituyen justamente la *base* para entender la práctica de la vida profesional real en el ámbito de la programación web y su complejidad inherente. Con esto queremos anticipar que no es nuestro objetivo ser *exhaustivos*. Ya existe muy buena bibliografía sobre cada uno de los temas. Lo que no hemos encontrado es, justamente, una compilación introductoria que ofrezca un **panorama general**. Y esa tarea es la que nos hemos propuesto con esta obra.

El público principal que se tiene en mente es alumnos universitarios, pero este libro también puede ser utilizado por profesionales con experiencia en programación y que deseen incorporarse al mundo web. Con esto queremos decir que damos por descontada cierta experiencia en la informática en general y la programación en particular. No haremos aquí introducción de, por ejemplo, planteo de algoritmos para resolución de problemas. Nos focalizaremos en personas con experiencia previa *mínima* en lenguajes como C o Pascal.

Nos atendremos a las particularidades de la web y a sentar las bases para que cada uno pueda continuar aprendiendo por sí mismo, un desafío que sin dudas marca la actualidad y el futuro de los profesionales del software.

Este material es fundamentalmente apoyo bibliográfico para los cursos que el autor desarrolla en su actividad docente, pero fue pensado para funcionar también en su ausencia, por medio de la utilización de un lenguaje claro, directo, incluyendo ejercicios y referencias para tener un camino por donde continuar.

La presentación y el desarrollo de los temas toma en consideración que simultáneamente el lector irá experimentando frente a una computadora, a medida que avanza con la lectura del material. Es muy importante realizar la práctica, ya que *leer* y comprender código es una actividad cognitiva muy distinta a *escribirlo*.

HERRAMIENTAS

Las distintas tecnologías que se utilizan son sencillamente vehículos para concretar de manera práctica los ejercicios y programas pero, desde ya, no representan las únicas posibles y en parte están influenciadas por las preferencias personales. Es en este sentido que decimos que nuestro objetivo principal es presentar los fundamentos de la programación web, es decir, “*cómo funciona*” y no, en lo posible, tecnologías ni versiones particulares de algún producto o programa. Para presentar esos fundamentos hemos elegido las siguientes tecnologías:

- **Navegador web:** es una de las herramientas que más utilizaremos a lo largo de los diferentes temas, por lo que es muy importante contar con un navegador con el que nos sintamos a gusto y que nos provea de las herramientas y ayudas apropiadas. Elegimos Google Chrome o Mozilla Firefox.
- **Editor de texto:** es definitivamente necesario contar con un buen editor de código fuente. No es imprescindible para nuestros objetivos usar un IDE completo como

Eclipse o NetBeans. Programas como SublimeText, Notepad++ o gEdit son muy buenas opciones, sin complejidades inútiles por el momento.

- **Servidor web:** para los capítulos de PHP y MySQL necesariamente tendremos que hacer uso del software pertinente al lado servidor. Personalmente preferimos el paquete preconfigurado WampServer para instalaciones permanentes, o su equivalente Server2Go como opción portable. La elección misma de este lenguaje y esta base de datos se debe a su nivel de popularidad y a la mínima configuración necesaria para comenzar a trabajar. Debe decirse que existen otros lenguajes y bases de datos también muy utilizados en la práctica y que pueden ser perfectamente utilizados para enseñanza. Este es el punto donde más diversidad de tecnologías se puede encontrar y, por lo tanto, se debe realizar una elección y esforzarnos por concentrar la atención en los *conceptos* y no en las *particularidades* de estas tecnologías, en definitiva circunstanciales.

ORGANIZACIÓN

Cada capítulo se extiende sobre una tecnología en particular y corresponde a un concepto del mundo de la programación web. Si bien esta forma de tratar los temas es, entendemos, la más conveniente para *aprender*, es verdad que la web no funciona compartimentada de esta forma. Justamente, la dificultad inherente a la programación web tiene que ver con la cantidad de tecnologías que funcionan *a la vez*, tanto en los navegadores como en los servidores.

Por esta razón, es necesario aclarar que al principio se puede tener la sensación de estar haciendo cosas muy primitivas, feas o inútiles... y esto es totalmente verdad. No creemos que sea posible hacer ejemplos que se *parezcan* a las aplicaciones web reales sin antes emprender un pequeño viaje por cada una de las tecnologías involucradas. Es decir, no nos parece realmente productivo a largo plazo hacerlo de otra manera. Este esfuerzo por compartimentar los temas es fruto de varios años de experiencia en la enseñanza de la programación web.

Si no se comprenden las bases de cada tecnología, las razones de su diseño y los problemas que intenta abordar creemos que será muy difícil encarar luego de manera eficiente los problemas de las aplicaciones y clientes reales. Tal vez sea un poco frustrante al principio, pero en el mediano plazo es muchísimo más eficaz abordar con cierto detenimiento cada tema para *luego* integrarlos en una aplicación real y, lo más importante, aplicar el conocimiento a situaciones nuevas y distintas.

Trataremos de utilizar los términos en castellano siempre que sea posible. Cuando los términos o nombres en inglés sean imprescindibles, serán mencionados en ese idioma. Muchas palabras propias de los protocolos y estándares serán repetidas y utilizadas en distintos contextos y lenguajes de programación, por lo que se hace indispensable su mención en inglés ya que no podrán ser traducidas en el código.

Sin más introducción comencemos, obviamente, por el principio. ¡Bienvenidos!

1. HTTP

El protocolo HTTP se creó a principios de la década de los '90 en los laboratorios del CERN, en Francia. Su objetivo es definir el **transporte** de documentos web a través de las entonces nacientes redes de computadoras. Su sigla proviene de *HyperText Transfer Protocol* (protocolo para transferencia de hipertexto), en donde *hipertexto* puede ser simplemente entendido por documento web, más puntualmente **documentos HTML**, o como los llamamos habitualmente: **páginas web**. Por lo tanto nos detendremos a observar cómo funciona este protocolo que se utiliza para controlar el viaje que cada página web emprende cada vez que navegamos.

El conocimiento de este protocolo es sumamente importante. Sin temor a exagerar, este sea quizás el tema más importante de este libro. El protocolo HTTP es el que determina el funcionamiento de la web y es el que funda las diferencias con el resto de la programación. HTTP es lo que nos define como *programadores web*. El conocimiento que implica este protocolo es de muy largo plazo porque es una tecnología que ha cambiado muy poco y que demuestra ser absolutamente funcional: la versión que utilizamos hoy en día es la 1.1, correspondiente a la última revisión aprobada en 1999.

1.1 LA WEB COMO TRÁFICO

Una de las cuestiones más radicales que implica trabajar programando en web es pensar en términos de **tráfico de red**. Es tan radical como fundamental porque, como dijimos, es lo que diferencia la programación web del resto de la programación. Tradicionalmente utilizamos los flujos de entrada y salida estándar para comunicarnos con el usuario, por ejemplo, en lenguajes como C, Pascal o Java en cursos introductorios de programación. Habiendo avanzado un poco más suelen introducirse las interfaces gráficas para escritorio, por ejemplo en VB.net, que un usuario utilizará en su computadora por medio del teclado y el mouse.

Pero al llegar a la programación web nos encontramos con sistemas que son necesariamente multiplataforma (todos tenemos distintos navegadores y distintas computadoras), con gran cantidad de asincronismo (programaremos tanto para el navegador del usuario como en el servidor) e ineludiblemente multiusuario y concurrentes (muchas personas están ejecutando nuestro programa simultáneamente). Estas son características que surgen del tráfico web, de la misma arquitectura, por lo que son conceptos modulares a la programación web.

Comprender este tráfico es fundamental para programar sistemas web. No habrá aplicación web sin tráfico de red, aunque se trate de una intranet muy pequeña o una aplicación pensada para nosotros mismos. Siempre tendremos tráfico cuando trabajemos en web, y este tráfico será HTTP.

Es importante destacar el uso que hacemos de la palabra **web**. Siempre que hablemos de web estaremos haciendo referencia al tráfico HTTP. Este tráfico HTTP es tráfico que

tiene como destino el **puerto 80** del protocolo TCP. Los servidores web están escuchando en este puerto 80. Este tráfico HTTP es entonces lo que llamamos *web*. Este tráfico web es encausado por la red de routers, empresas y subredes que llamamos ampliamente **Internet**.

Destacamos esto porque puede haber más ramas de la programación que trabajen sobre tráfico de Internet pero que no sean *web* específicamente. Por ejemplo, el protocolo FTP, los servicios de voz sobre IP, toda la infraestructura de correo electrónico o la resolución de DNS constituyen muestras de tráfico de red que no es *web*, ya que no es tráfico HTTP sino de otros protocolos con otros puertos. Decimos entonces que la *web* “corre” sobre Internet, utilizándola como infraestructura de comunicación para llegar de un punto al otro del planeta. El tráfico web es entonces una *parte* del tráfico total de Internet.

1.2 ARQUITECTURA CLIENTE-SERVIDOR

El protocolo HTTP responde a la arquitectura **cliente-servidor**. Esto quiere decir que el protocolo define dos roles diferentes que serán cumplidos por dos piezas de software distintas. Tendremos entonces los **clientes web** y los **servidores web**. Estos dos roles tienen distintas responsabilidades. Cuando un programa cliente dialoga con un programa servidor tenemos una comunicación HTTP, y toda esta comunicación y conjunto de responsabilidades es la que está definida en el protocolo.



Figura 1.1: HTTP es un protocolo cliente-servidor

En la figura 1.1 observamos que el cliente y el servidor están separados por una línea punteada. Esto intenta poner el foco en la distancia geográfica. Es decir que estos dos programas, el cliente y el servidor, están ejecutándose en computadoras distintas. Llamaremos a estas dos computadoras como **lado cliente** y **lado servidor**, o sencillamente **cliente** y **servidor**. Es importante visualizar que son dos computadoras diferentes. La única manera de intercambiar información que tienen estos dos programas es a través de mensajes HTTP definidos en el protocolo.

Es muy importante tener en cuenta que los roles no son intercambiables. Los clientes solo pueden (y deben) hacer las cosas que el protocolo define para los clientes, y lo mismo ocurre con los servidores. Cada rol tiene sus responsabilidades.

Hay distintos fabricantes de software que producen y comercializan clientes y servidores web. Los clientes web son muy conocidos por nosotros porque los utilizamos todos los días. Son llamados comúnmente **navegadores**. Por lo tanto, los navegadores que usamos en nuestras computadoras son, técnicamente, clientes web. Es decir que son programas que fueron creados para cumplir con lo que el protocolo HTTP exige para los

clientes web. Todas las empresas que crean navegadores deberán leer el protocolo y hacer un programa que cumpla las responsabilidades.

Lo mismo sucede con los servidores web que, salvo que trabajemos como programadores web, no son programas con los que estemos familiarizados normalmente. Las personas que navegan no trabajan de manera directa con estos programas, por lo que parecen no existir desde el lado cliente. Esto se debe a que el navegador hace el trabajo por nosotros, es decir, traduce nuestras acciones (teclas y clicks) en mensajes del protocolo HTTP que envía hacia un servidor web. Con el correr de nuestro estudio iremos “perdiendo la inocencia”, y es uno de nuestros objetivos entender que la web es esencialmente HTTP, aunque “no lo veamos” de manera directa en nuestro uso cotidiano. Profundizaremos en el funcionamiento de HTTP para entender el gran trabajo que realizan los navegadores y servidores que luego programaremos.

Si todos los fabricantes de software se atienen a lo convenido en el protocolo, se tiene el gran beneficio de que los clientes y los servidores son **interoperables**. Esto es una realidad hoy en día, ya que prácticamente no encontramos diferencias al navegar con distintos clientes y no notamos cambios en los distintos sitios web que navegamos, aunque cada uno utilice un software servidor de distinto fabricante.

1.3 TIPOS DE PAQUETES

El tráfico HTTP se compone de **paquetes**. Estos paquetes o mensajes se dividen en dos grupos: **peticiones** (*requests*) y **respuestas** (*responses*). No hay otros elementos en el tráfico HTTP más que estos paquetes que viajan entre el cliente y el servidor.

Las peticiones son los paquetes que el cliente envía al servidor. Las respuestas son los paquetes que el servidor envía al cliente como consecuencia. Siempre la comunicación HTTP comienza en el lado cliente con una petición y concluye con una respuesta que proviene del lado servidor. Cada petición, si no hay problemas de comunicación, tendrá siempre una respuesta, y cada respuesta existirá porque antes hubo una petición. A este par petición-respuesta se lo llama **transacción HTTP**.

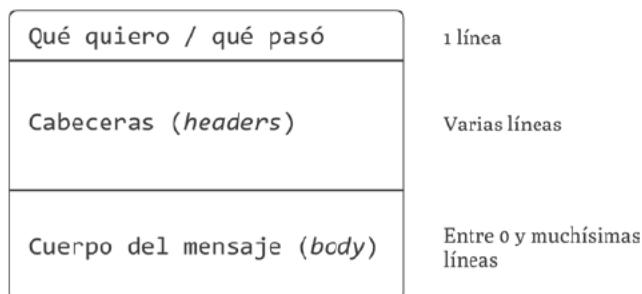


Figura 1.2: Estructura de los paquetes HTTP

En la figura 1.2 podemos observar un esquema que ilustra la estructura de los paquetes HTTP. Los paquetes, tanto peticiones como respuestas, tienen la misma estructura y se componen de tres partes.

La primera parte está constituida por una sola línea que indica en el caso de las peticiones qué es lo que estamos pidiendo al servidor. En el caso de las respuestas tenemos información sobre qué sucedió con nuestro pedido.

La segunda parte está compuesta de varias líneas que son llamadas **cabeceras** (*headers*). Estas cabeceras son información sobre el paquete (ya sea petición o respuesta) y contienen, por ejemplo, la fecha en la que se generó el paquete, el sitio web al que pertenece, el navegador que estamos usando, etc. Existen muchas cabeceras definidas en el protocolo HTTP. Las cabeceras que encontramos en las peticiones son distintas de las que encontramos en las respuestas. Por el momento no entraremos en más detalle; todas las cabeceras pueden consultarse en el protocolo.

La tercera y última parte del paquete, que llamamos **cuerpo** (*body*) es la que contiene efectivamente la información que está siendo transportada. Hemos visto que el protocolo HTTP fue creado para transportar páginas web. En el cuerpo de las respuestas están entonces las páginas web que solicitamos con las peticiones. Más adelante encontraremos que son muchos los recursos que deben transportarse entre clientes y servidores. Toda esta información es transportada en el cuerpo de las respuestas.

Veamos un ejemplo con paquetes reales. Estamos navegando en un sitio web que contiene recetas de cocina. Nos concentraremos en el lado cliente, es decir, en el navegador. En el menú lateral del sitio web encontramos categorías de recetas. Presionamos sobre un *link* para ir a la página con recetas que pertenecen al grupo “pastas”.

```
GET /pastas.html HTTP/1.1
-----
Host: www.todorecetas.net
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; es-ES; rv:1.9.....
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-es,es;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Cache-Control: max-age=0
-----
```

Figura 1.3: Petición real enviada por un cliente

En la figura 1.3 observamos la petición que generó nuestro navegador. Aquí vemos en funcionamiento una **responsabilidad** que tienen los clientes: a partir de nuestros clicks, generar peticiones para enviar al servidor. Las líneas punteadas no son parte del paquete: las hemos incluido sólo para visualizar las partes más claramente.

La primera línea contiene la palabra “GET”, el nombre de la página que queremos ver (hemos hecho click en pastas) y la versión del protocolo que el cliente prefiere utilizar. Decimos entonces que nuestro click sobre un link fue traducido por el navegador en una *petición tipo GET*.

En la segunda parte del paquete encontramos las cabeceras. Allí vemos que nuestro navegador es un Mozilla Firefox corriendo en un sistema Windows (*User-Agent*). Además vemos que el lenguaje preferido por el usuario es castellano (*Accept-Language*). Estas

cabeceras fueron armadas por nuestro navegador en función de nuestras preferencias y de su propia información. Existen muchas más cabeceras y puede profundizarse mucho en cada una de ellas, pero por el momento es suficiente.

La tercera parte (el cuerpo) de este paquete está vacía. Recordemos que esta parte de los paquetes se utiliza para enviar los datos que están siendo transportados. Por definición, el cuerpo de las peticiones GET siempre está vacío. Razonemos de esta forma: si estamos enviando un mensaje al servidor para que éste nos devuelva una página web, ¿qué incluiremos en el cuerpo del mensaje? La respuesta es: nada. *El cuerpo de las peticiones GET siempre está vacío*. A continuación veremos que hay más peticiones además de GET, en donde esto no siempre es así.

Una vez que la petición haya llegado al servidor, será procesada y, luego de otro tiempo de tránsito el navegador recibirá una respuesta. Pensemos en que estos paquetes tardan cierto tiempo en llegar al servidor y viceversa. Es importante pensar en este tiempo para estar atentos del **asincronismo** y de que son computadoras diferentes.

```
HTTP/1.1 200 OK
-----
Date: Fri, 05 Aug 2013 11:23:24 GMT
Server: Apache
X-Powered-By: PHP/5.2.13
Keep-Alive: timeout=1, max=64
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
-----
(en el cuerpo está La página que veo en el navegador, bytes...)
```

Figura 1.4: Respuesta real contestada por el servidor

En la figura 1.4 observamos la respuesta que corresponde a la petición anterior. Esta respuesta fue generada por el servidor, enviada y recibida en el lado cliente. Aquí tenemos una responsabilidad que el protocolo HTTP marca para los servidores web: contestar a las peticiones que envían los clientes.

En la primera parte encontramos la versión del protocolo, el número “200” y un breve mensaje “OK”. Esto quiere decir que lo que hemos solicitado fue encontrado y nos está siendo enviado con esta respuesta.

En la segunda parte podemos observar las cabeceras, que son distintas a las cabeceras de la petición, y que tienen información sobre el paquete y sobre el servidor en sí. Podemos ver que el software del lado servidor se llama Apache (*Server*), la fecha en la que el servidor generó esta respuesta (*Date*) y el tipo de información que nos está enviando (*Content-Type*).

En el cuerpo del paquete, que en la figura fue recortado porque era muy extenso, encontramos lo que hemos pedido, es decir, la página web que se corresponde a las recetas de pastas. La página web está escrita siguiendo un estándar llamado HTML, que estudiaremos en el próximo capítulo.

El protocolo HTTP determina entonces de qué forma una persona que utiliza un cliente web puede solicitar a un servidor cierto documento. El protocolo también establece la forma en la que el servidor web enviará de vuelta ese documento. Nótese que el protocolo fue diseñado para ser leído directamente por seres humanos.

Enumeremos entonces las responsabilidades de los programas *cliente*:

- armar y enviar las peticiones al servidor
- entender las respuestas que vuelven del servidor
- atender la interfaz del usuario (notar la complejidad de este ítem)

Y las responsabilidades de los programas *servidores*:

- esperar peticiones escuchando el puerto 80
- concretar la acción que se solicita (por ejemplo, buscar una página web)
- armar la respuesta y enviarla al cliente (por ejemplo, con la página web)

1.4 MÉTODOS HTTP

El protocolo HTTP define ciertos **métodos**, también llamados comandos o verbos, que son los utilizados en la primera línea de las peticiones para indicar al servidor cuál es nuestro pedido. Estos métodos son los que cargan a la petición de un significado, porque según qué método utilicemos será el resultado que se obtenga del lado servidor.

Hemos hablado ya del método **GET**. Pero definamos ahora más precisamente: el método GET se utiliza cuando queremos que el servidor nos envíe cierto **recurso** (*resource*). Para el protocolo HTTP, un recurso puede ser una página web, una imagen, una canción, un video, o cualquier otra cosa. Sólo para visualizarlo, podemos pensar en un recurso como en un archivo. Por ejemplo, cada vez que vemos en nuestro navegador una página web que contiene imágenes, el cliente tuvo que pedir esas imágenes, una por una, al servidor. Para cada imagen construyó un paquete GET solicitándola. Hemos visto también que, cada vez que nos movemos de una página a la otra a través de los *links*, el cliente construye paquetes GET para pedir la página siguiente que queremos navegar.

Las peticiones GET están definidas en el protocolo HTTP como *seguras*. Esto es porque por definición deben ser **idempotentes**, es decir, siempre generar las mismas respuestas. Si solicitamos a un servidor una página web diez veces, utilizando diez peticiones GET, el servidor nos enviará diez respuestas con la misma página web. Otra forma de observarlo es que las peticiones GET no deben cambiar el estado interno del servidor, por lo que a idéntica petición debe corresponder idéntica respuesta. El método GET debe ser idempotente según marca el protocolo. Recordemos que más adelante seremos nosotros quienes programemos el software que está en el servidor, por lo que tendremos responsabilidad en cumplir con este requisito.

Hablemos ahora de un método HTTP que no es seguro, porque no es idempotente: el método **POST**. Las peticiones POST se utilizan cuando es el cliente el que está enviando información al servidor. Por lo general el funcionamiento de la web es con peticiones GET, ya que la mayor parte del tiempo estamos navegando entre páginas y, por ende, es

el servidor el que nos envía la información a los clientes. Pero cada tanto somos nosotros, desde el lado cliente, quienes enviamos información hacia el servidor. El servidor tomará la información y realizará alguna acción, por lo que probablemente cambiará su estado interno. De aquí que no sea *seguro* como GET.

Veamos un ejemplo clásico de paquete POST: un formulario de contacto. Es conocido por todos el formato en donde se nos pide nuestro nombre, email y mensaje de contacto. Este mensaje se envía al dueño del sitio web, que más tarde nos contesta.

¿Cómo se implementa el formulario de contacto? Cuando introducimos nuestros datos y el mensaje que queremos enviar en el formulario, y presionamos “Enviar”, el navegador construye una petición POST.

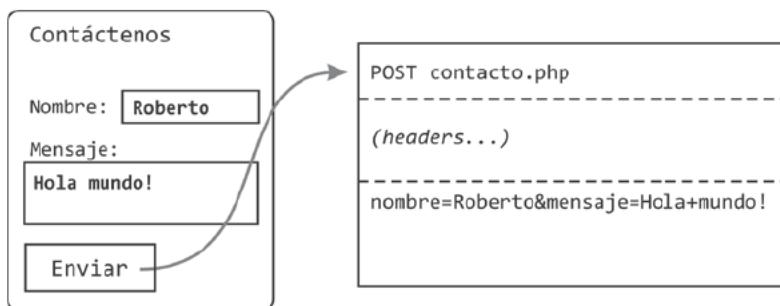


Figura 1.5: Esquema de paquete POST generado con un formulario

Vemos en la figura que los paquetes, como dijimos, tienen siempre la misma estructura de tres partes. En la petición que se construyó observamos que el método es POST y no GET. En segundo lugar tenemos las cabeceras y, a diferencia del paquete GET que estudiámos antes, encontramos que en el cuerpo sí hay información.

La información que encontramos en el cuerpo de los paquetes POST es la información que, en efecto, estamos enviando al servidor. Una petición POST le indica al servidor que es el cliente quien le está enviando algo para procesar. En el cuerpo está dicha información; en nuestro caso se trata de los datos para realizar el contacto.

El formato que tienen los datos presentes en el cuerpo se denomina **URL encoding**, y lo estudiaremos más adelante junto a HTML.

Los navegadores tienen una función muy conocida por todos: al presionar **F5** hacen una **recarga de página**. Por definición, lo que los navegadores realizan al presionar **F5** es, en realidad, volver a enviar el último paquete que hayan enviado al servidor. Si el último paquete enviado fue de tipo GET, se producirá la conocida “recarga” de página. Pero si el último paquete corresponde a un POST el navegador nos consultará si realmente queremos volver a enviarlo. Esto es así, justamente, porque el método POST no es idempotente. Traducido a nuestro ejemplo del formulario de contacto, si enviamos nuevamente el POST (porque presionamos **F5**) juntaremos generando otro contacto en el sitio! Para ponerlo en mayor perspectiva, pensemos que los formularios de compra online (cuando ingresamos los datos de nuestra tarjeta de crédito) también se envían con POST, por lo que si presionamos **F5** juntaremos comprando varias veces lo mismo! De esta manera observamos que, con acierto, se denomina al método POST como no-seguro (*unsafe*).

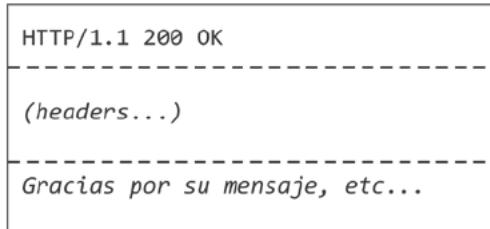


Figura 1.6: Ejemplo de posible respuesta a la petición POST

En la figura 1.6 observamos que la respuesta que nos envía el servidor a nuestra petición POST es como todos los paquetes. Tiene tres partes e indica qué sucedió con nuestra petición precedente, en nuestro ejemplo, el POST del formulario de contacto.

Además de los métodos GET y POST, el protocolo define algunos más y permite la extensión a través de métodos personalizados. Algunos métodos usados, aunque muy escasamente en comparación a los dos que mencionamos son HEAD, PUT y DELETE. Estos métodos exóticos son utilizados en algunos diseños modernos de software denominados *RESTful*.

1.5 CÓDIGOS DE ESTADO

Las respuestas que envía el servidor contienen en la primera línea un número de tres dígitos que llamamos **código de estado**. Este código es una de las partes más importantes de la respuesta ya que indica claramente al cliente qué ocurrió con su petición al llegar al servidor.

Los códigos son números que se clasifican en grupos según su primer dígito. Los códigos que comienzan con 2 son los que indican que la operación se concretó con éxito. Este es el código que nunca vemos al navegar, ya que nuestro cliente no nos indica si una página web solicitada se pudo conseguir con éxito: sencillamente se limita a mostrarla en pantalla.

Los códigos que comienzan con 4 son los errores del cliente. El célebre error **404** es el más conocido. Lo que estos errores agrupan son las situaciones de error que el servidor reconoce como problema del cliente. Por ejemplo, solicitar una página web que no existe generará que el servidor nos devuelva una respuesta 404, que indica justamente esto. El servidor está funcionando bien: somos nosotros quienes nos estamos equivocando al pedir algo que no existe. Otro error de este tipo puede ser solicitar una página para la cual no tenemos permisos de acceso.

Los códigos que comienzan con 5 son los errores del servidor. El más conocido es el error **500**, que quiere decir, de manera difusa, que hay un problema en el servidor y no se ha podido completar nuestro pedido. Si estamos solicitando una página web y obtenemos una respuesta 500 tendremos que intentar más tarde. Se trata de una indicación de que no es nuestra culpa (*¡salvo que seamos nosotros los programadores del servidor!*).

Los códigos que comienzan con 3 son llamados **redirecciones**. Las redirecciones son respuestas que llegan al cliente y que indican que debe generar una nueva petición. De

alguna manera, es una instrucción que el servidor envía al cliente, indicando que para completar la petición debe dirigirse hacia otro lado. Se utiliza cuando un recurso cambia de dirección. Si enviamos una petición, por ejemplo GET, para solicitar una página que ha cambiado de nombre, es posible que el servidor nos devuelva una respuesta 301, indicando una nueva dirección para la página. Nuestro navegador identificará la respuesta 301 y, automáticamente, se dirigirá a la nueva dirección realizando un nuevo GET, comenzando el diálogo nuevamente. Este mecanismo de redirección es fácilmente reconocible al realizar *login* en los correos electrónicos. El efecto visual que tiene en el navegador es que la barra de direcciones cambia varias veces hasta que llegamos a ver nuestros correos. Esas constituyen varias redirecciones en el diálogo HTTP que nuestro navegador ha tenido con el servidor.

Código	Texto que acompaña	Significado
200	OK	Se pudo completar lo solicitado
301	Moved permanently	Redirección permanente
302	Found	Redirección temporal
403	Forbidden	Se debe proveer una contraseña
404	Not Found	Recurso no existente en el servidor
500	Internal Server Error	El servidor está fuera de servicio

Figura 1.7: Códigos de estado HTTP más utilizados

1.6 INSPECCIÓN DE TRÁFICO

A continuación utilizaremos el navegador Chrome para realizar inspecciones de tráfico HTTP. Esta inspección también se puede realizar en Firefox y otros, además de existir diversas extensiones para los navegadores que nos permiten distintos análisis. Nuestro objetivo es verificar las propiedades del tráfico que hemos estudiado y familiarizarnos con esta herramienta tan útil en la práctica profesional.

Primero desplegaremos la herramienta para desarrolladores presionando las teclas **CTRL+MAYUS+J** o eligiendo “Herramientas del desarrollador” en el menú. Luego seleccionaremos la pestaña “Network” o “Red”, según el idioma.

En principio no visualizaremos nada porque recién hemos activado la herramienta. Pediremos al navegador que envíe un paquete GET. Haremos esto escribiendo en la barra de direcciones la siguiente URL: es.wikipedia.org/wiki/HTTP.

Al presionar **ENTER**, el navegador genera el paquete GET y espera la respuesta del servidor. Si todo funciona bien, luego de un breve tiempo veremos en pantalla la entrada de Wikipedia para el protocolo HTTP. Lo más interesante es que nuestra herramienta de inspección de tráfico de red se ha llenado de información.

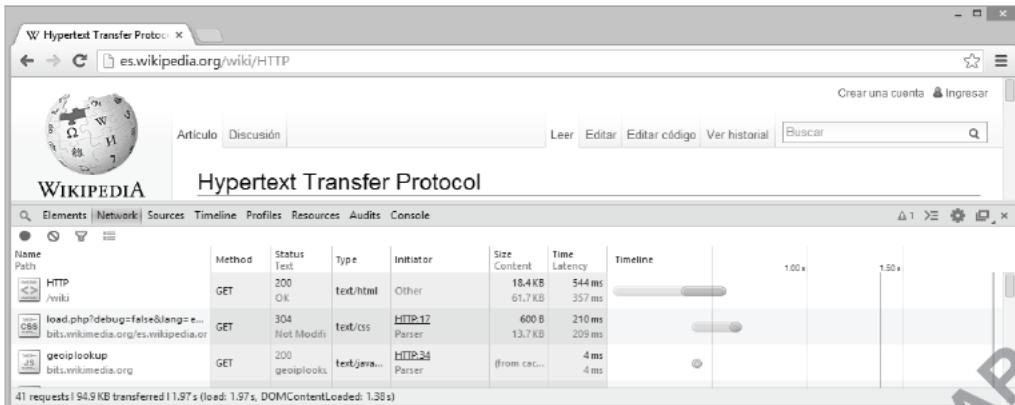


Figura 1.8: Inspección de tráfico HTTP con Chrome

Cada línea corresponde a un paquete enviado al servidor, con su indicación de método, código de estado de la respuesta, tiempos, pesos y demás información. Es evidente la potencia que tiene esta herramienta de análisis.

Si hacemos click en un paquete accedemos a información más detallada, además de poder ver las cabeceras tanto de la petición como de la respuesta correspondiente. El navegador nos está mostrando el tráfico generado a partir de nuestra solicitud original, que se inició con un simple **ENTER**. Estamos viendo el tráfico HTTP concreto.



Figura 1.9: Detalle de petición HTTP

Es interesante notar la cantidad de solicitudes que el navegador debe realizar para cargar una sola página web. Muchas de las peticiones no tendrán sentido en este momento, pero es de destacar que, por ejemplo, cuantas más imágenes posea una página web, más solicitudes deberá realizar el cliente. Ello se traduce en ancho de banda y en tiempo de carga. El tráfico de red es fundamental para comprender cómo funciona la navegación a través de la web.

La mayoría de los paquetes enviados serán GET y la mayoría de las respuestas serán “200 OK”. Esto quiere decir, básicamente, que estamos navegando y que no hay problemas. Al final del capítulo se podrán encontrar ejercicios para probar distintas situaciones y continuar trabajando con esta importante herramienta.

1.7 PRODUCTOS MÁS UTILIZADOS

La existencia de un protocolo, aceptado mundialmente, permite que distintos fabricantes de software creen clientes y servidores web, y que todos ellos sean interoperables. Si bien en las primeras épocas de Internet esto todavía era un desafío, hoy es prácticamente una realidad.

Los clientes HTTP más utilizados son Google Chrome, Microsoft Internet Explorer, Mozilla Firefox, Opera y Apple Safari. Los servidores HTTP más utilizados son Apache, Nginx y Microsoft IIS (*Internet Information Server*).

Mes a mes se modifica la tasa de participación de cada uno en el mercado y los distintos fabricantes van desarrollando sus productos de acuerdo a las tendencias tecnológicas que surgen cada vez más rápidamente. Otro factor de decisión importante entre las distintas opciones es el tiempo que toma cada organización para corregir las vulnerabilidades de seguridad que se descubren en todo software.

1.8 UBICACIÓN DE RECURSOS

Hemos dicho que HTTP trabaja con recursos. Cuando visitamos una página web, nuestro navegador solicitará al servidor correspondiente esa página, a través de una petición GET. Decimos que ese servidor web **aloja** (*hosts*) a ese recurso. Para indicar qué recurso solicitamos se necesita una URL, sigla de *Uniform Resource Locator*, que indica qué recurso (*resource*) buscar y en dónde está alojado.

protocolo://máquina:puerto/directorio/archivo

Figura 1.10: Partes de una URL

En la figura 1.10 observamos las partes que componen una URL. Analicemos algunos ejemplos:

- `http://es.wikipedia.org` El dominio donde se aloja el recurso es `es.wikipedia.org`, es decir, ese es el “nombre” de la máquina servidora. Como no se indica ninguna página en especial, se nos responderá con el **índice** (*index*) de la carpeta raíz. El índice es una página web creada con un nombre especial, que el servidor envía cuando un cliente accede a una carpeta sin especificar un archivo.
- `ftp://ftp.servidor.com/imagenes/hola.jpg` Se indica otro protocolo, el FTP. El nombre de la máquina es `ftp.servidor.com` y el recurso, que es una imagen, se encuentra dentro de una carpeta llamada “*imagenes*” dentro de la raíz.

- <https://seguro.afip.gob.ar/monotributo> Se indica una variante de HTTP que es el **HTTP seguro** (de allí la "S"). Esta variante se utiliza, por ejemplo, en las comunicaciones bancarias y basa su funcionamiento en la criptografía asimétrica. No se indica el nombre de un archivo, por lo que se accederá, como en el primer ejemplo, al **índice**, pero de la carpeta "monotributo".
- <http://secreto.com:81/ingreso.php> Aquí encontramos una URL que señala que el puerto de escucha no es el 80 sino el 81. Es decir que tenemos un servidor HTTP que fue configurado para escuchar un puerto que no es el estándar.

Para indicar cuál es el servidor que aloja el recurso se suele utilizar un **dominio** (*domain*). El dominio es un nombre, generalmente palabras en cierto idioma, fácilmente recordable para las personas, que está asociado a un **número IP**. Dentro de toda red de computadoras, incluyendo a la gran red Internet, las computadoras se identifican con un número específico para cada una, que es su IP. Sin embargo, este número es muy difícil de recordar, por lo que el dominio nos provee de un atajo mucho más fácil en el uso cotidiano.

1.9 DOMINIOS Y SU TRADUCCIÓN A IP



Figura 1.11: Partes de un dominio

En la figura 1.11 se detallan las distintas partes que componen un dominio de los regularmente utilizados por nosotros. El **nombre de dominio** propiamente dicho es el que aparece justo antes del **top level domain**, que es regulado internacionalmente. Cada país cuenta además con un código adicional que también está regulado. El organismo que coordina y mantiene el sistema es IANA (*Internet Assigned Numbers Authority*).

Pero hemos dicho que los dominios son en realidad nombres más memorizables por nosotros, cuando en realidad las computadoras de una red se identifican y comunican utilizando su número IP. Cuando indicamos al navegador qué página queremos ver lo hacemos a través de una URL que expresa el dominio. Evidentemente debe existir una forma de traducir los dominios en números IP para localizar los recursos en las computadoras correspondientes. Esta traducción se llama **resolución de nombre**, y es llevada a cabo por los **servidores DNS** (*domain name server*, servidor de nombre de dominio).

La infraestructura de DNS es muy importante para el funcionamiento de Internet y de todas las redes, no sólo de los servicios web. Por cierto, su complejidad escapa a nuestro foco de atención, pero haremos una breve reseña de su funcionamiento.

Los servidores DNS básicamente reciben preguntas del tipo ¿qué IP corresponde a este dominio? Estas preguntas se envían con cierto formato, para lo cual existe también un

protocolo. El servidor *resuelve* el nombre buscándolo en una tabla y devuelve la IP correspondiente. La infraestructura es muy compleja porque todo el sistema está diseñado para ser tolerante a fallas, y ese objetivo se consigue con una fuerte redundancia de la información. Cada servidor de DNS tiene un “padre” a quien puede consultar, y cada vez que resuelve un nombre lo almacena en una **caché**. Cuando se le pide resolver un dominio que no encuentra en su tabla, el servidor le consulta a su padre y guarda el resultado para próximas preguntas. El padre sigue la misma lógica. De esta forma, todos los servidores tienen a quién preguntarle en caso de “no saber” qué IP corresponde a un dominio. El último eslabón de esta cadena de servidores DNS se denomina **root server**. Estos servidores están divididos por continente, y son el último parentesco regular al que se le puede pedir que convierta un dominio a una IP. ¿Qué sucede si el root server tampoco puede convertir un dominio en IP?

La fuente última de conocimiento sobre qué IP corresponde a un dominio la tiene un servidor especial denominado **servidor autoritativo**. Este servidor es el “dueño” del dominio y es el que tiene la información real y más actualizada. Cuando un root server encuentra que no conoce la IP de cierto dominio, lo que hace es analizar a qué zona geográfica corresponde. Si es un dominio con código de país “.ar” lo que hará es delegar la resolución del dominio en el sistema argentino. Argentina, como todos los países, administra un registro que indica cuál es el servidor autoritativo de cada dominio. De esa manera se llega al servidor autoritativo, que conoce el número IP que corresponde a un dominio. Y todo el proceso continúa en sentido inverso, hasta que el navegador que originó toda esta cascada recibe cuál es la IP del dominio que posee en su URL. Todos los servidores DNS intermedios mantendrán en caché este número, para que este largo proceso no se tenga que repetir.

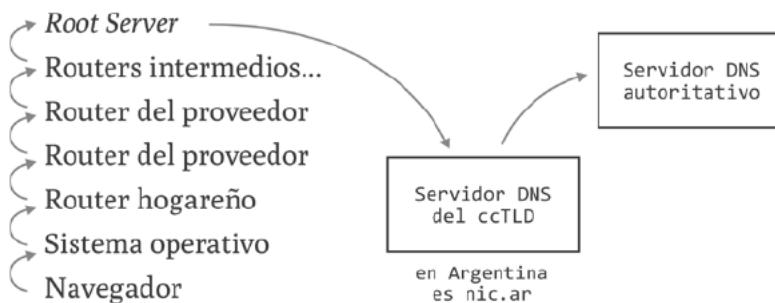


Figura 1.12: Procedimiento de consulta entre servidores DNS

Analicemos un posible caso de resolución de dominio y sigamos los pasos en cada servidor de DNS. Supongamos que hemos creado un sitio y su dominio es `prueba.com.ar`. Abrimos nuestro navegador e intentamos navegar por primera vez.

1. El navegador intenta enviar una petición GET hacia `prueba.com.ar`, pero no conoce la IP del servidor con el cual debe dialogar.
2. El navegador le consulta al sistema operativo cuál es la IP de `prueba.com.ar`, pero éste tampoco lo sabe porque nunca se lo han preguntado y, por ende, no lo tiene en su caché.

3. El sistema operativo le pregunta al router que la empresa de internet ha dejado en nuestra casa cuando realizó la instalación (es decir, su padre). Este aparato tampoco sabe la IP, por lo que deberá preguntar a su propio padre.
4. En la empresa que nos provee Internet hay más routers, que tampoco saben y deben seguir preguntando a sus padres.
5. La empresa que nos provee de Internet tiene también sus propios proveedores, que son mayoristas de telecomunicaciones. Estas empresas, a las que los usuarios finales no tenemos acceso directo, proveen de ancho de banda a las empresas minoristas que nosotros contratamos. Ellos también tienen routers que, igualmente, tampoco conocen en qué IP está nuestro sitio web.
6. Siguiendo hacia niveles superiores llegaremos al root server del continente que, naturalmente, tampoco sabe la IP de nuestro sitio.
7. El root server evaluará que nuestro dominio es de Argentina y consultará al servidor **nic.ar**, que es la entidad que registra los dominios en nuestro país.
8. **Nic.ar** dirá que el *servidor autoritativo* para nuestro dominio es 200.10.10.10 (ejemplo). Esto lo hemos configurado nosotros cuando creamos el dominio.
9. El root server le consultará al servidor DNS localizable en la IP 200.10.10.10 cuál es la IP del dominio **prueba.com.ar**.
10. Como este servidor es el *autoritativo* debe conocer la IP de nuestro dominio, por lo que contesta, por ejemplo, 200.40.40.40.
11. El diálogo inicia el camino inverso, almacenando el resultado en las cachés de todos los participantes intermedios, hasta llegar a nuestro navegador.
12. El navegador envía el paquete GET a la IP 200.40.40.40.
13. Nuestro servidor web atiende la petición y devuelve una respuesta.
14. Vemos el índice en nuestro navegador.

1.10 PASOS PARA PUBLICAR UN SITIO WEB

Si bien todavía no hemos incursionado en las tecnologías necesarias para crear las páginas web, ya hemos introducido los conceptos mínimos para comprender cómo funcionará un sitio web nuevo, desde el punto de vista de la infraestructura de red.

El primer paso será contratar un servicio de **hosting**. Este servicio nos proveerá de **alojamiento** para nuestras páginas. Alojamiento quiere decir, básicamente, espacio en el disco rígido de un servidor HTTP. La empresa de hosting nos alquilará espacio en su servidor y, la mayoría de las veces, nos brindará además el servicio de DNS autoritativo. Esta empresa es la que conoce la IP que tendrá el servidor (porque el servidor les pertenece) y mantendrán el servidor DNS autoritativo que indica cuál es la IP del dominio. Al momento de contratar el servicio, nos preguntarán cuál es el dominio de nuestro sitio web.

Por otro lado, para registrar un dominio debemos hacer una solicitud en una **entidad registrante**. Las entidades registrantes dependen del país al que pertenece el dominio. Como mencionamos, en el caso de Argentina los dominios son los **.ar** y se registran en **nic.ar**. El registro puede conllevar un costo, que varía de país en país y según el tipo de dominio que registremos.

Cuando realicemos el registro del dominio en la entidad correspondiente, uno de los datos solicitados será el servidor DNS autoritativo. Nosotros indicaremos el que nuestro hosting nos informe. Este paso se denomina **delegación de dominio**.

Revisemos los pasos con un ejemplo:

1. Nos acercamos a una empresa de hosting y contratamos espacio de alojamiento para nuestras páginas.
2. La empresa nos da acceso al disco rígido de uno de sus servidores HTTP. Allí subimos nuestras páginas. La IP de este servidor es 200.40.40.40.
3. La empresa nos pregunta cuál será nuestro dominio. Supongamos nuevamente **prueba.com.ar**
4. La empresa posee un servidor de DNS escuchando en la IP 200.10.10.10. En ese servidor crea un registro en la tabla de nombres, que dice que el dominio **prueba.com.ar** corresponde a la IP 200.40.40.40 y nos informa que el servidor autoritativo de nuestro dominio está en 200.10.10.10
5. Nosotros registramos nuestro dominio ingresando a **nic.ar**. Allí debemos ingresar datos personales e indicar la delegación, o sea, que el servidor autoritativo de DNS es 200.10.10.10
6. Cuando cualquier persona quiera navegar **prueba.com.ar** verá nuestras páginas web, alojadas en el servidor HTTP del hosting que hemos contratado.

Es importante notar que toda esta complejidad está diseñada para ser tolerante a fallas y para permitir una modificación sencilla de las IP sin que los usuarios finales vean cambios en los dominios.

Si en algún momento precisamos cambiar de empresa de hosting, el dominio no se verá afectado, porque las modificaciones serán en el servidor autoritativo, que no afectan al usuario de manera directa. El servidor HTTP, el servidor DNS autoritativo y la entidad registrante son los tres componentes de este diseño flexible.

1.11 CONSERVACIÓN DE ESTADO (COOKIES)

Como último punto a considerar sobre el protocolo HTTP, observaremos una de sus características de diseño y una técnica muy difundida denominada **cookie**.

El protocolo HTTP es un **protocolo sin estado**. Esto quiere decir que el protocolo no obliga al servidor a realizar ningún seguimiento sobre las peticiones que contesta. En otras palabras, no es una de sus responsabilidades. Esto resulta muy conveniente desde el punto de vista de la performance, pero es problemático para ciertas implementaciones. Por ejemplo, si solicitamos a un usuario su contraseña de acceso ¿cómo recordaremos que este cliente es quien está autorizado si perdemos su rastro en la próxima petición? ¿cómo saber que el cliente que nos envía la contraseña también es el que, dentro de un minuto, nos pide una página protegida?

Este problema surgió muy temprano en la historia de la web y los primeros desarrolladores de software lo solucionaron con una técnica llamada **cookie**.

Las cookies son porciones de información que se transmiten cada vez que un cliente realiza una petición al servidor. Esta información viaja en los paquetes en forma de cabecera. Es correcto decir entonces: las cookies son cabeceras.

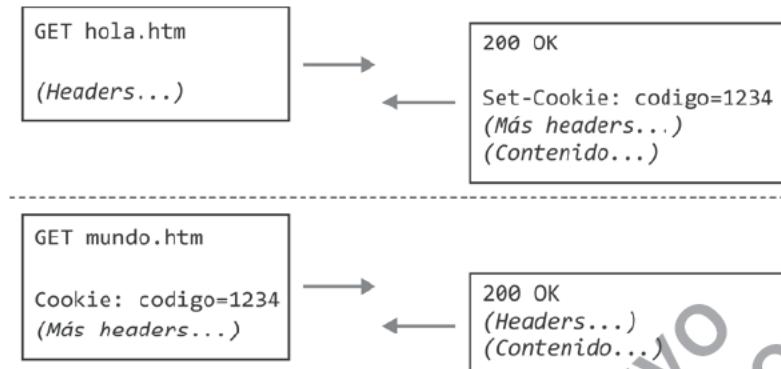


Figura 1.13: Proceso de creación de cookies con la primera respuesta y posterior envío con peticiones

Supongamos que ingresamos a un sitio web por primera vez. El servidor verificará que “no nos conoce”, por lo que en la respuesta nos enviará un cabecera llamada *Set-Cookie*, que le solicita al navegador que cree una cookie. La cookie es, en definitiva, una **variable**, que tiene un nombre y un valor.

La primera vez que ingresemos a un sitio, lo más probable es que el servidor nos asigne un número. Este identificador se utilizará para, en la primera respuesta, crear esta cabecera *Set-Cookie*. Al recibir esta cabecera, el navegador creará un registro y almacenará el nombre y el valor de la cookie.

Cada vez que el navegador envíe una nueva petición a este mismo servidor enviará una cabecera *Cookie* con el nombre y valor almacenados previamente. De esta forma el servidor, en la segunda petición y sucesivas, podrá deducir que se trata del mismo usuario. De esta forma se logra conservar el estado ya que cada usuario tendrá un número distinto y el servidor podrá ofrecerles páginas personalizadas para cada uno.

Es importante destacar que la cabecera *Set-Cookie* es sólo un pedido para el navegador y que éste no está obligado a guardar la información ni retransmitirla. Los navegadores pueden configurarse para no almacenar cookies, aunque esto generará hoy en día que prácticamente no podamos utilizar de forma razonable ningún sitio.

Existen muchos mitos sobre las cookies, relacionados con la publicidad, la privacidad y el *malware*. Durante mucho tiempo han estado en boca de los usuarios de Internet y en las noticias. Lo concreto es que son mecanismos utilizados para conservar estados en las aplicaciones que utilizan el protocolo HTTP.

1.12 EL FUTURO

Actualmente se encuentra en desarrollo la **versión 2.0** del protocolo HTTP, 15 años después de la última revisión que ha tenido lugar, la actual 1.1.

Esta nueva versión está basada en desarrollos de la industria, que ya habían puesto manos a la obra en resolver ciertos cuellos de botella. Para reducir el tráfico y los tiempos de latencia, el nuevo protocolo indica comprimir las cabeceras en lugar de trabajar con texto plano. Además establece mecanismos para eliminar la repetición de información idéntica entre sucesivas peticiones y busca un mejor uso de las conexiones TCP, que son costosas de establecer.

En definitiva, la versión 2 supone una mejora que no altera la esencia del protocolo, por lo que las aplicaciones de versión 1.1 seguirán funcionando de la misma manera, tunelizadas a través del nuevo protocolo. Los navegadores más modernos ya soportan funcionalidad que acabará siendo el protocolo versión 2, y en algunos servicios como Google Gmail está en funcionamiento efectivo.

1.13 BIBLIOGRAFÍA

Un excelente libro, a la vez directo, completo y ameno es **Gourley y Totty**, “*HTTP, The Definitive Guide*”, O'Reilly, Sebastopol: 2002.

El protocolo 1.1 propiamente dicho está disponible en tools.ietf.org/html/rfc2616

1.14 EJERCICIOS

1. Nombre dos clientes y dos servidores HTTP que no estén listados en este capítulo entre los más usados.
2. Utilice la herramienta de inspección de tráfico HTTP de su navegador e ingrese a dos de sus sitios favoritos. ¿Cuántas peticiones se realizan para la carga completa del índice o página principal?
3. ¿Cuántas de las peticiones encontradas en el ejercicio 2 corresponden a imágenes y cuántas a los demás recursos?
4. Al realizar la experiencia del punto 2, ¿existe alguna respuesta con código distinto de 200? En ese caso ¿qué código se indica y qué significa según el protocolo?
5. Ingrese al índice de 4 sitios distintos e indique qué contiene la cabecera *Server* de las respuestas. ¿Son iguales?
6. Encuentre algún paquete POST utilizando sus sitios favoritos.
7. ¿Qué es la criptografía asimétrica?
8. ¿Para qué se utiliza la herramienta `nslookup` de la línea de comandos de Windows?

2. HTML

El estándar HTML es una creación del mismo grupo de investigadores que iniciaron la web con HTTP, en la misma época. HTML es la sigla para *HyperText Markup Language*, lenguaje de marcas para hipertexto. El **hipertexto** es lo que conocemos comúnmente como multimedia, es decir, el conjunto de información en distintos formatos (textual, imágenes, audio, video, animaciones) y que posee información que *superá* al texto, principalmente, vínculos entre los documentos (*links*). HTML será el lenguaje que utilizaremos para escribir y describir las **páginas web**.

Si bien se denomina “lenguaje”, HTML no es un lenguaje en el sentido de los *lenguajes de programación*. No encontraremos variables, estructuras de control o llamadas a función. En cambio, HTML es un lenguaje de **marcas**, y son estas marcas las que, conjuntamente con la **sintaxis**, conforman la parte principal del estándar.

La importancia de HTML para un programador web no sólo radica en su uso directo, es decir, para producir páginas web reales. La práctica de trabajar con un lenguaje de marcado favorece la conceptualización de estructuras anidadas como los árboles y ayuda a acelerar la comprensión de tecnologías con el mismo espíritu, como XML. Además, HTML está presente de manera ineludible en el desarrollo web, cualquiera sea el resto de las tecnologías utilizadas.

2.1 LENGUAJE DE MARCAS

HTML es un **lenguaje de marcas**. Las marcas son metainformación que acompaña al contenido propiamente dicho del **documento**. Utilizaremos la palabra “documento” de manera intercambiable con “página web”.

La metainformación que intercalaremos en el documento es necesaria para dar *significado* a la información. Es decir que las marcas HTML tienen una carga **semántica**: definirán la semántica de cada parte del documento. Esta definición explícita del significado es necesaria porque las computadoras no tienen la capacidad simbólica y cultural que tenemos los seres humanos, esto es, la capacidad de deducir el significado a partir de la visualidad de un documento.

Para poner un ejemplo claro, pensemos en la portada de un diario. Nuestra habilidad simbólica nos permite saber inmediatamente cuál es el título principal de esta portada. Además entendemos que algunos subtítulos se acompañan de ciertos textos y no de otros. Entendemos que los epígrafes acompañan a ciertas imágenes y no a otras. De la misma forma reconocemos cuál es el nombre del diario aunque no comprendamos el sentido específico de lo que se está diciendo. Incluso podemos identificar qué partes del contenido corresponden a publicidades y conjeturar cuál es la orientación política del diario, basados en nuestros prejuicios y en información previa de la que disponemos, ya sea consciente o inconscientemente.

La información visual es transformada por nosotros automáticamente en significados, y es justamente esta tarea la que no puede ser realizada por las computadoras. Por esta razón, cuando creemos nuestros documentos HTML, la mayor parte del tiempo estaremos cargando de significado, o de ayudas de significado, a la información concreta de cada documento. Estas indicaciones de significado son las marcas del lenguaje HTML.

Para tener una perspectiva histórica, y para comprender mejor por qué HTML tiene la forma actual, es útil comparar con artículos de Wikipedia. En sus principios, el uso de Internet fue principalmente universitario, en donde se publicaban *papers* y se compartían trabajos entre docentes de distintas universidades del mundo. Esta red de contenido dio origen a la web de hoy. Inicialmente, este contenido estaba muy relacionado con el tipo de documentos que podemos encontrar hoy en día en Wikipedia, esto es, material académico. Este material precisa fundamentalmente de una semántica editorial: títulos y subtítulos, imágenes con sus epígrafes, párrafos, destacados con negritas e itálicas, citas bibliográficas, viñetas, etc. Por eso encontramos útil pensar en los artículos de Wikipedia, porque ese tipo de documentos es el que tenían en mente, por necesidad, quienes delinearon el lenguaje HTML original alrededor de 1990.

Esta forma de comprender HTML puede ser frustrante en un primer momento, ya que las aplicaciones web que estamos acostumbrados a utilizar hoy en día están muy lejos de estos criterios iniciales. Pero entendemos que la comprensión de la lógica fundamental del lenguaje nos permitirá luego avanzar rápido y, lo que es muy importante, poder aprender más por nosotros mismos en el futuro.

2.2 SINTAXIS DE LAS MARCAS

De manera intercambiable utilizaremos los términos marca, etiqueta y **H4**: . A continuación examinaremos cuál es la sintaxis de HTML, y de qué forma las etiquetas comenzarán a marcar el contenido de los documentos para expresar su semántica.

Cuando coloquemos el **título** de un documento utilizaremos la etiqueta **h1**:

```
<h1>Introducción a la biología marina</h1>
```

Princess's travel plans may have been given to paparazzi

Continued on page K

Josh Halliday

Transcript of phone conversations between Virgin Atlantic and its PR agency over the last year show the airline was fully aware of the scale of the problem before it emerged, but it is a mystery of affects whether the company's handling of the crisis was appropriate.

Morgan Stanley and its legal team at the legal firm of DLA Piper, which "extremely seriously" and "urgently" advised the airline to take action, are now under investigation by the City of London Police. The transcript also shows that Virgin Atlantic's PR agency, Ogilvy PR Worldwide, was asked to "keep quiet" about the issue, despite the fact that the airline had already issued a statement to the press.

Details of a flight booking to the US were made available to the press agency

More... Media headache. Your director and co-founder of the The Big Off radio show, Mark 'Boris' Borisovitch, has been charged with a criminal offence related to his role in the show. Borisovitch is accused of being involved in the distribution of child pornography. The 43-year-old has been charged with "distribution of child pornography". He has been held in custody since April 10, 2010, and is due to appear in court on May 11. Borisovitch has denied the charges. His lawyer, Mark 'Kerr' Kerrigan, said: "He is innocent and he will fight to clear his name."

Las etiquetas se delimitan por los signos menor y mayor, dentro de los cuales se coloca el **nombre** de la etiqueta. Vemos que el título del documento se está marcando haciendo uso de una **apertura** (`<h1>`) y un **cierre** (`</h1>`). El cierre del título se indica con una marca similar a la de apertura pero con una barra delante del nombre de etiqueta.

La etiqueta `h1` quiere decir entonces “*título principal*”. Proviene del inglés *heading* (título), y el número 1 hace referencia a que es un título de jerarquía 1, o sea, el más importante.

Por supuesto, los nombres de etiquetas no pueden inventarse. Las etiquetas están definidas en el estándar HTML y son la razón por la cual estudiamos este lenguaje. En la terminología del estándar también se dice que este `h1` crea un **elemento**.

Luego del título principal es probable que encontremos un **subtítulo** y, a continuación, un **párrafo** con texto. Hasta aquí, nuestro documento se vería así:

```
<h1>Introducción a la biología marina</h1>
<h2>Científicos participantes</h2>
<p>En este estudio han participado colegas provenientes de...</p>
```

Observamos un documento que tiene un título principal, un subtítulo (o sea, un título que es de una jerarquía menor que el principal) y un párrafo de texto. Las etiquetas están indicando *qué* es cada parte del contenido, definiendo tres elementos.

Si creamos un documento nuevo con estas tres líneas y lo guardamos con el nombre `ejemplo1.html`, al hacer doble click sobre él se abrirá en nuestro navegador web.

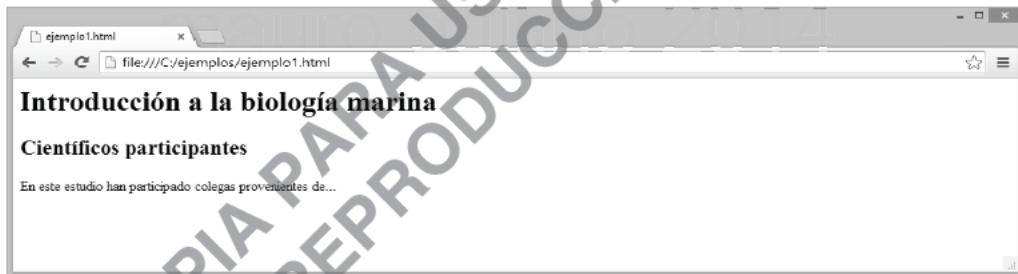


Figura 2.1: Navegador Chrome interpretando las etiquetas

2.3 SEMÁNTICA VS. GRÁFICA

En la figura 2.1 encontramos el resultado visual que genera el navegador al interpretar el documento que creamos. Debemos poner mucho énfasis para evitar un error conceptual muy habitual: que el subtítulo (`h2`) se vea más pequeño que el título principal (`h1`) es una decisión que ha tomado el navegador, pero esa decisión **gráfica** no es el objetivo de nuestras etiquetas. Para decirlo con otras palabras: las marcas HTML no serán colocadas para determinar la gráfica del documento, *nunca*.

El objetivo de HTML no es determinar la gráfica, o sea, la forma **visual** del documento, sino sus significados: su **semántica**. Las marcas determinan *qué* es el elemento y no *qué forma* debe tener. Elegir las etiquetas de acuerdo al tamaño que queremos que tenga un texto es un error conceptual.

Para determinar la forma gráfica de un documento web haremos uso de la tecnología creada para ese fin: las hojas de estilo CSS. Ese será el tema del próximo capítulo. Por el momento, la consecuencia visual que tengan las etiquetas que incorporamos serán definidas por el navegador y no podrá ser modificada sino hasta el próximo capítulo. Si una parte del texto es el título principal del documento procederemos a marcarlo con `h1`. Luego modificaremos su tamaño visual (con CSS) si no nos agrada, pero esto no debe alterar la semántica que hemos marcado en HTML previamente.

La asociación entre el *significado* de las distintas partes del documento y su *representación* gráfica está muy arraigada en nuestras competencias simbólicas, tal como exploramos con la portada del diario. Para nosotros, como seres culturales, es muy difícil disociar el significado de la visualidad. Debemos realizar este esfuerzo conceptual para trabajar con las dos tecnologías (HTML y CSS) de la manera correcta y sacar el mejor provecho del diseño de ambas.

2.4 ESTÁNDARES

El lenguaje HTML tiene varias versiones y en los últimos años ha evolucionado bastante y a un ritmo cada vez más rápido. Las distintas versiones y estándares son, en definitiva, documentos y libros que, como profesionales, nos obligamos a entender y cumplir. Este compromiso redunda en un beneficio para todos los interesados.

El organismo encargado de proponer, revisar, crear y mantener el estándar es el **W3 Consortium** (abreviado **W3C**). Este consorcio está conformado por profesionales, empresas tecnológicas, investigadores y personas interesadas de diversos sectores. El sitio web [w3.org](http://www.w3.org) contiene muy valiosa información sobre todos los estándares de los cuales son responsables. Su director es Tim Berners-Lee, creador de HTTP y HTML.

Tenemos hoy en día tres versiones de HTML en circulación. La primera es **HTML 4**, ya casi en desuso, que representa el estándar más parecido al original de 1990. Se lo reconoce fácilmente por ser un lenguaje escrito en mayúsculas. Las etiquetas se ven de la forma `<H1>`, `<H2>`, `<P>`. En este estándar se observan algunas etiquetas para determinar la gráfica del documento. Esto ha sido eliminado a medida que avanzó el tiempo, de manera que el estándar sólo hable de la semántica.

En el año 2000 se aprueba el estándar **XHTML 1**. Este estándar resulta de una evolución de HTML 4 y la introducción de reglas de sintaxis que provienen del estándar **XML**. De aquí la "X" que se antepone al nombre. La incorporación de la sintaxis de XML supone un código mucho más limpio y más fácil de interpretar por parte de los programas. En términos prácticos, una de los cambios más notorios es que las etiquetas pasan a escribirse en minúscula. Además, se pone gran esfuerzo en separar la información de su representación. Como hemos dicho, la semántica se separa fuertemente de la visualidad a partir de este estándar.

En su momento, el estándar XHTML representó una gran diferencia práctica para los profesionales, por lo que se lanzó con dos formatos: el **Strict** y el **Transitional**. La idea fue que el pasaje sea gradual, ya que la mayoría de los desarrolladores web no provenían de la informática sino del diseño gráfico, y algunas modificaciones eran difíciles de

asimilar o implementar, de tal manera que el cambio de estándar resultaba muy resistido por quienes ya estaban trabajando con esta tecnología. La versión transicional es un intermedio entre el anterior HTML 4 y el XHTML estricto.

El estándar más reciente es **HTML 5**, que ha supuesto grandes incorporaciones. Las nuevas etiquetas responden a los nuevos desafíos que propone la web. Como mencionamos, el diseño inicial de las etiquetas vino dado por las necesidades de los universitarios que dieron comienzo a la web. Tres décadas después, las aplicaciones que hoy en día utilizamos en la web son muy distintas en cuanto a objetivos y funcionalidades. Por esto, el W3C ha introducido cambios que responden a estos nuevos requisitos de las aplicaciones web. En este estándar se habla de documentos interactivos para diferentes soportes más que de documentos editoriales textuales. Para fines de 2013, sin embargo, HTML 5 todavía no es una recomendación oficial del consorcio aunque el borrador está muy cerca de serlo. Sin perjuicio de esto, los navegadores más usados ya implementan prácticamente todo el estándar candidato.

El W3C posee una herramienta online que nos permite conocer si un documento cumple con determinado estándar. Esto es fundamental a la hora de verificar nuestro trabajo. Esta herramienta de verificación se encuentra en validator.w3.org. Permite indicar una URL o subir un archivo, para saber si las etiquetas responden a las reglas de un estándar en particular o si se trata de un **documento mal formado**.

2.5 ESTRUCTURA BASE

En el ejemplo anterior hemos pedido a nuestro navegador que abra un documento HTML y lo muestre. Aquí ocurre un hecho interesante, porque el documento de tres líneas que hemos visto no es **válido**. Esto quiere decir que no cumple el estándar; podemos averiguarlo con la herramienta de validación que acabamos de mencionar.

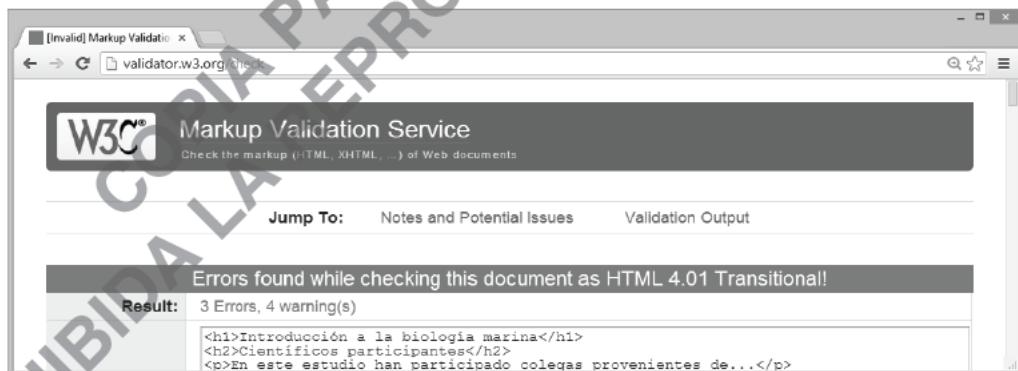


Figura 2.2: La herramienta de validación de W3C señalando un documento mal formado

¿Por qué el navegador no nos comunicó nada al respecto? Este es el punto interesante que precisamos comprender: los clientes web no están diseñados principalmente para nosotros, los desarrolladores, sino para el usuario final. Nunca un navegador va a dejar de mostrar una página web porque el documento no cumpla con el estándar de W3C. Esto es así porque el criterio de diseño responde a la necesidad de los usuarios finales

de navegar. Con otras palabras, el navegador tratará de hacer “lo mejor posible” al abrir cada documento HTML, e intentará mostrar el contenido aunque este sea **inválido** desde el punto de vista del estándar oficial. Históricamente los navegadores han trabajado de esta manera, tal vez, laxa.

Pero, ¿para qué preocuparnos tanto por lo que dice el estándar si los navegadores de todas formas mostrarán nuestro documento al usuario? Y aquí tocamos un punto importante, que es el de la **interoperabilidad**. Para que nuestros documentos se vean igual en todos los clientes web (o sea, los distintos navegadores) que existen hoy y los que se creen también mañana, es preciso que sigamos la recomendación del estándar. Esta es la razón principal de esforzarnos en crear estándares en la industria: garantizar la interoperabilidad. Si nuestros documentos no son válidos estamos dejando librado a la decisión de cada navegador la forma en la que se muestra la página web. Ningún programador quiere ser molestado porque el sitio web que hizo para un cliente no funciona en el navegador del teléfono celular que este cliente se acaba de comprar.

Dicho esto, examinemos por qué nuestro primer ejemplo no era un documento válido. Todos los documentos HTML, esto es, toda página web, tiene una estructura de base que debe respetar. Es la siguiente:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Título del documento</title>
    </head>
    <body>
        <!-- esto es un comentario, aquí estará el contenido de la página -->
    </body>
</html>
```

Cada vez que comencemos una página web lo haremos a partir de esta estructura básica que todo documento debe cumplir para ser HTML 5 válido.

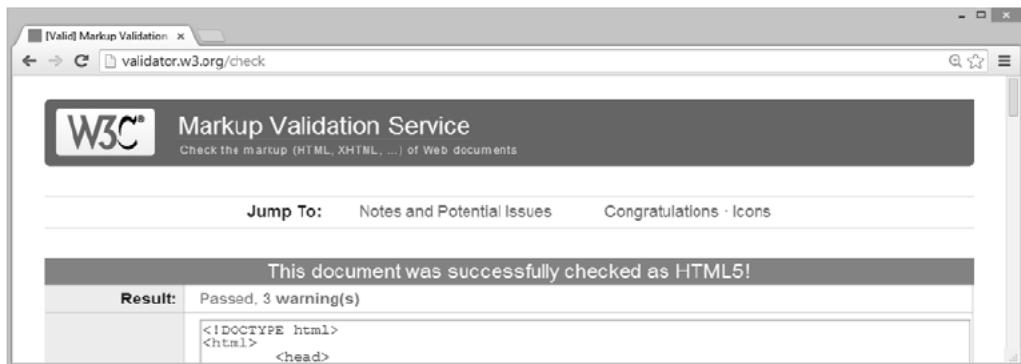


Figura 2.3: Un documento correcto siendo validado por la herramienta validator.w3.org

Un hecho destacable que puede observarse en la estructura básica de un documento, incluso cuando todavía no hemos colocado ningún contenido a la página, es que los documentos HTML son básicamente estructuras anidadas, donde unas etiquetas van a

ir colocadas dentro de otras. Siguiendo las buenas prácticas de estilo, colocaremos tabulaciones a medida que ingresemos en niveles más profundos.

Examinemos la estructura de base. Observamos una etiqueta `<html>` que engloba a todas las demás. Esta etiqueta envuelve a todo el documento. Dentro de ella, tenemos dos grandes secciones, que son `<head>` y `<body>`. Dentro de `<body>` irá todo el contenido de la página web y dentro de `<head>` se colocará la metainformación sobre el documento en sí, por ejemplo, su título descriptivo y otros elementos que encontraremos luego.

En `<body>` tendremos nuestros títulos, párrafos, imágenes y demás, es decir, lo que verá el usuario. Por lo tanto, siempre habrá unas etiquetas dentro de otras, y finalmente texto marcado por alguna etiqueta, por ejemplo `<p>`. Siempre encontraremos anidamiento.

La primera línea `<!DOCTYPE html>` es una etiqueta especial (nótese el signo de exclamación) que le indica al navegador con cuál estándar debe procesar este documento. El `267F-5` (*document type*) que se ve en este ejemplo corresponde a HTML 5 y será el que utilicemos de aquí en adelante.

2.6 ÁRBOL DEL DOCUMENTO

Los documentos HTML pueden representarse como árboles.

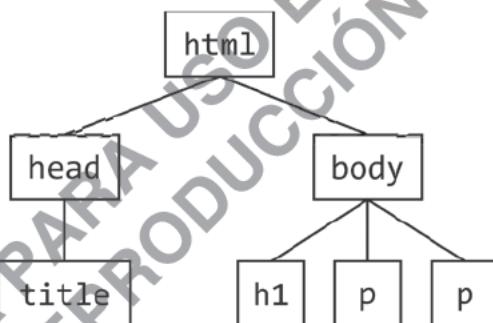


Figura 2.4: Un documento representado como un árbol

Los árboles y las etiquetas son dos formas distintas de representar la naturaleza jerárquica de la información que contiene nuestro documento. Cada elemento denotado con un par de marcas conforma un nodo del árbol. La raíz del árbol es la etiqueta `html`.

Al observar la representación del árbol es fácil encontrar relaciones entre los distintos nodos: relaciones padre-hijo, nodos hermanos, nodos que no tienen hijos, etc. Estas relaciones serán utilizadas más adelante cuando desde un lenguaje de programación podamos acceder a este árbol.

La representación en memoria de este árbol se denomina **Document Object Model**, modelo de objetos del documento, o sencillamente **DOM**. El DOM es accesible y nos permite agregar, modificar y quitar elementos del documento en tiempo real. El DOM es generado por el navegador cuando recibe el documento desde el servidor.

Los árboles son, junto con las listas enlazadas, pilas y colas, estructuras de datos muy utilizadas en el desarrollo de software y en la práctica de la programación.

2.7 ETIQUETAS PARA MARCADO DE TEXTO

Haremos un recorrido por las etiquetas más utilizadas de HTML. Comenzaremos por las etiquetas que más se utilizan para marcar el contenido de texto del documento.

Ya hemos visto los títulos, que se marcan con las etiquetas `<h1>` hasta `<h6>`, representando los números más pequeños una mayor jerarquía del título. `<h1>` es entonces el título principal del documento. Si pensamos en un artículo de Wikipedia, `<h1>` sería el nombre del artículo que estamos leyendo.

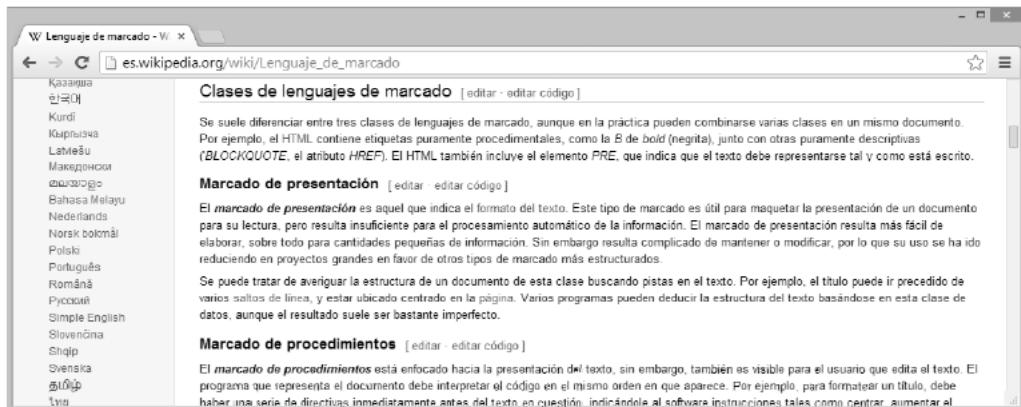


Figura 2.5: Títulos de distinta jerarquía y párrafos de texto

Recordemos que las etiquetas están definidas en el estándar. Si marcamos algún texto con la etiqueta inexistente `<h7>` el navegador no nos presentará ningún error, pero estaremos ante un documento mal formado.

También hemos utilizado la etiqueta `<p>`, que define párrafos (*paragraph*) de texto.

Veamos ahora cómo se definen los vínculos entre las páginas:

```
<!DOCTYPE html>
<html>
<head>
    <title>Prueba</title>
</head>
<body>
    <h1>Prueba de links</h1>
    <p>Este texto tiene
        <a href="pagina2.html">un link</a> a otra página
    </p>
</body>
</html>
```

La etiqueta `<a>`, de *anchor* (ancla), se utiliza para demarcar un **vínculo** o *link* a otra página. Si abrimos este documento con el navegador veremos que, por defecto, los links aparecen en color azul y con su texto subrayado.



Figura 2.6: Un documento con título, párrafo y link

Para volver sobre nuestros conceptos de semántica es importante, y hasta obvio, observar que no colocaremos links cuando precisemos tener un texto azul y subrayado. La gráfica es de esta manera porque el navegador coloca ese estilo por defecto. Si deseamos otra visualidad para los vínculos faremos los cambios empleando CSS. Cuando necesitemos un vínculo a otra página colocaremos un link, y luego evaluaremos si su forma gráfica nos satisface.

En la etiqueta `<a>` encontramos un componente nuevo llamado **atributo**. En este caso observamos un **atributo obligatorio** de las etiquetas `<a>`, que es el atributo `href` (*hyper reference*). El atributo `href` indica el destino del link.

Los atributos tienen un **nombre** y un **valor**, separados por un signo "`=`". Los atributos se utilizan para especificar propiedades especiales de las etiquetas, y cada etiqueta tiene un conjunto de atributos posibles. Algunos atributos son obligatorios, como el caso de `href` en los links. Siempre que coloquemos un link tendremos que indicar su `href`. Además existen **atributos opcionales**, que utilizaremos sólo en caso de necesitarlos. Los atributos siempre se colocan dentro de la etiqueta de apertura y su orden es indistinto. Deben estar separados entre sí con al menos un espacio.

Bajo ciertas circunstancias pueden omitirse las comillas utilizadas en el valor del atributo, pero como recomendación general y para evitar problemas, utilizaremos siempre comillas dobles para encerrar los valores. Algunos autores prefieren las comillas simples (apóstrofes). Ambas están permitidas por el estándar HTML 5.

Desde ahora, además de conocer las etiquetas principales de HTML también debemos conocer sus correspondientes atributos obligatorios y opcionales. Tanto las etiquetas como los atributos indican cosas muy puntuales, y ése es el conocimiento que debemos tener del estándar.

Si hacemos un click sobre el texto del link, es decir, las palabras que han quedado marcadas por la etiqueta `<a>`, notaremos que el navegador envía una petición GET al servidor, utilizando la URL conformada con el `href` que indicamos. El valor del atributo `href` que se muestra en el ejemplo contiene una **URL relativa**, es decir que, tal como las rutas relativas, se conforma de manera *relativa a la URL actual*. El servidor, como se trata simplemente de un nombre de archivo (`pagina2.html`), al recibir la petición buscará ese

documento en la carpeta actual. En caso de existir el documento lo recibiremos y visualizaremos en el navegador. Caso contrario, el servidor nos enviará una respuesta con código 404 (no encontrado). Este diálogo HTTP fue tratado con detalle en el capítulo anterior. En el apéndice D se explican con detenimiento las rutas absolutas y relativas.

Exploraremos ahora la marca utilizada para insertar una imagen en el documento:

```
<!DOCTYPE html>
<html>
<head>
    <title>Prueba</title>
</head>
<body>
    <h1>Prueba de imagen</h1>
    <p>Este texto tiene una imagen:<br/>
        
    </p>
</body>
</html>
```

Cuando el navegador encuentra una etiqueta ``, realiza una petición GET al servidor para descargar la imagen. Si todo es correcto, veremos la imagen en el documento y utilizando la herramienta de inspección podemos comprobar el tráfico HTTP que se ha generado (figura 2.6).

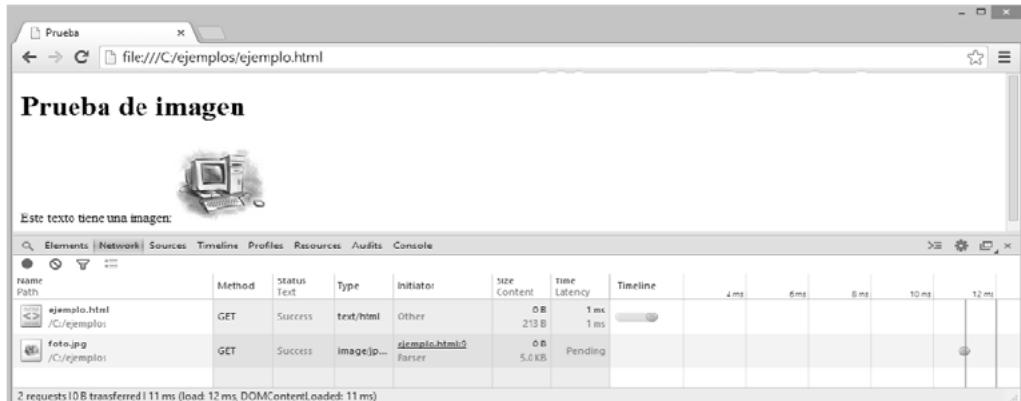


Figura 2.7: Tráfico HTTP generado por un documento con una imagen

La etiqueta `img` tiene dos atributos obligatorios. El atributo `src` (*source*) es la URL de la imagen a descargar. Puede ser una URL relativa o absoluta. El atributo `alt` (*alternative*) es un texto que puede ser usado cuando la imagen no está disponible; debe poder ser un *reemplazo* coherente de la imagen. Una forma de pensar en este atributo es imaginando una comunicación telefónica: dada una imagen ¿cómo se la describiríamos a alguien por teléfono? Este **texto alternativo** es muy importante, porque hace a la accesibilidad de nuestros documentos y al tratamiento del mismo por parte de computadoras. La **accesibilidad** se refiere al grado de funcionalidad que puede utilizar una persona con capacidades perceptivas diferentes, por ejemplo, no videntes. Estas personas navegan la web utilizando software que sintetiza una voz y lee los sitios web. Al encontrarse una imagen, estos programas leerán el `alt`. De la misma manera, los programas que utilizan

motores de búsqueda como Google para indexar el contenido web utilizan el `alt` para recolectar más información de nuestros documentos.

Si nuestro documento incluye, por ejemplo, un gráfico de barras, es muy importante que en el atributo `alt` expresemos de manera textual la misma información que una persona vidente puede deducir visualmente del gráfico. De esta manera el gráfico será accesible y estaremos utilizando el atributo con su intención según el estándar. En cambio, si tenemos imágenes que son meramente decorativas, y que no aportan al contenido del documento, dejaremos el atributo `alt` vacío, de esta manera: `alt=""` (dos comillas pegadas, simples o dobles, denotan un *atributo sin valor*).

Otra característica nueva que encontramos observando la etiqueta `img` es que no posee una marca de apertura y otra de cierre, como observamos en las marcas que exploramos hasta aquí. La etiqueta `img` pertenece a un grupo de *etiquetas vacías* (*void*). Las etiquetas vacías son aquellas que no poseen contenido textual.

En el estándar HTML, todas las etiquetas se dividen en dos grandes grupos: aquellas que poseen contenido (como `h1`, `p`, `a`) y aquellas que son vacías (como `img` y otras). Es natural que `img` no posea contenido ya que es una marca que indica que allí debe colocarse una imagen; no está demarcando ningún texto y por lo tanto no tiene "comienzo" ni "fin", sencillamente está ubicada en esa posición del documento.

Obsérvese que las etiquetas que sean vacías llevan una **barra** al final de la marca, indicando al navegador que no debe buscar una etiqueta de cierre, porque no la tiene. Esta característica de las barras en las etiquetas vacías proviene también de XML.

Hagamos mención de dos etiquetas más, utilizadas para el marcado de texto:

```
<!DOCTYPE html>
<html>
<head>
    <title>Prueba</title>
</head>
<body>
    <h1>Prueba de marcas de texto</h1>
    <p>Este texto <strong>es importante</strong>
        y éste tiene algún <em>énfasis particular</em>
    </p>
</body>
</html>
```

Presentamos estas dos etiquetas porque ilustran de manera clara el objetivo semántico del marcado HTML. La etiqueta `strong` se utiliza para delimitar texto que sea **importante**. En los diarios, por ejemplo, se utiliza para marcar las ideas principales de un documento. En este libro lo utilizamos para marcar las **palabras clave**. La consecuencia visual por defecto es que el texto es mostrado en negrita. Por supuesto, este aspecto gráfico puede ser modificado y aplicaremos el mismo razonamiento semántico que venimos sosteniendo: no colocaremos `strong` simplemente cuando el texto vaya en negritas, sino que nos preguntaremos por qué está en negrita. Si la respuesta es que el texto *es importante*, entonces lo marcaremos con `strong`. Si el texto está en negrita por otra razón, por ejemplo porque es el nombre de una marca que se coloca en negritas para

resaltarla visualmente, buscaremos otra etiqueta. Este criterio semántico puede ser difícil de aplicar al principio ya que nuestra capacidad simbólica ata fuertemente la visualidad con el significado, pero con el tiempo se hace una práctica más sencilla y, además, generalmente los casos se repetirán de documento en documento.

La etiqueta `` indica un **énfasis** (*emphasis*) en el tono del contenido. De acuerdo al estándar, es una etiqueta que modifica el sentido de lo que se dice, a diferencia de `strong` que no modifica el sentido sino que indica relevancia de ciertas partes. Nótese el cambio de tono que las itálicas sugieren en estos ejemplos:

- La casa tiene ventanas
- La casa *tiene* ventanas (sugiriendo una corrección a un interlocutor anterior que haya sostenido que la casa no tenía ventanas)
- La casa tiene *ventanas* (sugiriendo una corrección a un interlocutor anterior que haya sostenido que la casa tenía ventanales)
- La *casa* tiene ventanas (sugiriendo que está en duda que sea una casa)
- **¡Cuidado!** La casa tiene ventanas (aquí se usa `strong`; nótese que el sentido no cambia respecto del primer ejemplo de la lista)

La etiqueta `em` es traducida por defecto en itálicas, pero esto puede ser modificado con CSS como todos los aspectos visuales.

2.8 ETIQUETAS DE AGRUPAMIENTO

Algunas etiquetas no se utilizan para marcar semántica de texto directamente, sino para construir grupos que adentro contendrán más etiquetas. Ya hemos visto el uso de las etiquetas `html`, `head` y `body`. Estas etiquetas no tienen contenido textual por sí mismas, sino que se utilizan para agrupar otras etiquetas que en última instancia contendrán el texto en sí.

A partir de ahora omitiremos todas las marcas de base para ahorrar espacio, y nos limitaremos a mostrar el contenido dentro de `body` que es, en definitiva, el contenido de la página web. Si se prueban los ejemplos con la herramienta `validator.w3.org` es imprescindible utilizar el documento completo ya que de otra manera nos indicará errores por ser un documento inválido (incompleto).

Veamos el caso de las marcas que se utilizan para construir **listas**.

```
<ul>
  <li>300 gr. de harina</li>
  <li>medio litro de leche</li>
</ul>
```

La etiqueta ``, *unordered list*, se utiliza para indicar una lista de ítems cuyo orden no es importante, es decir que alterar el orden de los ítems no modificará el significado. Por ejemplo, los ingredientes para preparar una torta no tienen un orden específico: la torta resultante será la misma sin importar el orden en el que compraremos los ingredientes en el supermercado.

Consideremos otro tipo de lista:

```
<ol>
    <li>Encender el horno</li>
    <li>Cocinar la preparación 30 minutos</li>
</ol>
```

En la siguiente figura observamos el resultado de ambas listas en el navegador:



Figura 2.8: Lista `` y lista ``

La etiqueta ``, *ordered list*, marca por el contrario una lista cuyos elementos sí tienen un orden particular que, de modificarse, altera el significado de la información. Naturalmente, los pasos para preparar la torta deben ser seguidos en el orden correcto. El navegador coloca automáticamente números para los ítems en este caso. Nótese que los números no están en el texto del contenido sino que son creados por defecto por el navegador. Por supuesto este comportamiento puede modificarse, también con CSS.

En ambos casos, tanto para listas ordenadas como no ordenadas, la etiqueta para marcar cada uno de los elementos de la lista es ``, *list item*.

Dentro de las listas (`ul` ó `ol`) solo pueden colocarse etiquetas `li`, y las etiquetas `li` siempre deben ir dentro de alguna lista. Es decir que estas etiquetas siempre aparecerán en conjunto. Dentro de la etiqueta `li`, sin embargo, podemos tener texto, links, imágenes, y cualquier otro contenido según necesite el documento.

Existen más marcas para definir otros agrupamientos, pero no es común encontrarlas en la práctica.

2.9 ETIQUETAS PARA DATOS TABULARES

Veamos ahora cómo se marcan en HTML los datos tabulares, o sea, las **tablas**:

```
<table border="1">
    <tr> <th>Año</th> <th>Población</th> </tr>
    <tr> <td>2001</td> <td>34 millones</td> </tr>
    <tr> <td>2011</td> <td>41 millones</td> </tr>
</table>
```

En este ejemplo encontramos cuatro etiquetas que siempre trabajan en conjunto para marcar los datos de una tabla. Observemos el resultado en el navegador y describamos el significado de cada marca.

Año	Población
2001	34 millones
2011	41 millones

Figura 2.9: Una tabla sencilla

¡Atención! Hemos utilizado el atributo `border` en la etiqueta `table` ya que, en caso contrario, el navegador por defecto no dibujará los bordes de cada una de las celdas. **Esta práctica no es correcta** porque estaríamos determinando aspectos gráficos en el documento HTML, algo que tratamos de evitar sistemáticamente para aprovechar la flexibilidad de separar los datos de la presentación. Nos tomamos la licencia, con la correspondiente aclaración y advertencia, para trabajar más fácilmente en este momento. Las características visuales deben ser indicadas con CSS, estándar creado especialmente para este fin.

La marca `<table>` agrupa a las demás para conformar una tabla, es decir, una estructura de datos que deben trabajarse de manera tabulada, o encolumnada. El sentido de la información se modifica si alteramos la estructura de filas y columnas.

Las **filas** de la tabla se marcan con `<tr>`, *table row*. Cada fila posee **celdas**, que pueden ser `<th>` (*table heading*) para los títulos de cada columna, o `<td>` (*table data*) para las celdas que poseen efectivamente los datos de la tabla. Si un programa lee nuestra tabla, y la misma está marcada correctamente, podrá calcular promedios sabiendo que la primera fila contiene textos (`th`) que no deben ser, obviamente, promediados porque no forman parte del conjunto de datos (`td`).

Es importante notar que en HTML las tablas se definen primeramente a partir de sus filas. Luego se colocan las celdas dentro de cada fila, por lo que no existe una indicación explícita de **columnas**. Es nuestro trabajo mantener la cantidad de celdas de manera homogénea en todas las filas para que la tabla no se "desarme" visualmente.

Existen dos atributos que nos permiten alterar la manera en que las celdas se combinan, para lograr tablas más complejas:

```
<table border="1">
    <tr> <td colspan="2"> A </td> <td> B </td> </tr>
    <tr> <td> C </td> <td> D </td> <td> E </td> </tr>
</table>
```

A	B	
C	D	E
C		

Figura 2.10: Efecto del atributo `colspan`

El atributo `colspan` es un **atributo opcional** de las etiquetas `td`. Indica al navegador la cantidad de **columnas** (horizontalmente) a través de las cuales se *extiende* cierta celda. Por defecto, cada celda constituye una columna. El valor del atributo es un número entero positivo.

De la misma manera, el atributo `rowspan` indica la cantidad de **filas** (verticalmente) sobre las cuales se debe extender una celda.

```
<table border="1">
    <tr> <td rowspan="2"> A </td> <td> B </td> </tr>
        <tr> <td> C </td> </tr>
    </table>
```



Figura 2.11: Efecto del atributo `rowspan`

Una celda puede poseer simultáneamente los atributos `rowspan` y `colspan`. La clave para comprender el resultado de aplicar estos atributos está en recorrer la tabla de izquierda a derecha y de arriba hacia abajo.

Veamos cómo funciona este marcado:

```
<table border="1">
    <tr> <td>A</td> <td colspan="2" rowspan="2">B</td> <td>C</td> </tr>
        <tr> <td>D</td> <td>E</td> </tr>
        <tr> <td>F</td> <td>G</td> <td>H</td> <td>I</td> </tr>
    </table>
```



Figura 2.12: Atributos `colspan` y `rowspan` aplicados en la misma celda

La primera fila tiene, de izquierda a derecha, una celda normal A, una celda B que ocupa dos columnas y dos filas, y una celda normal C. En la segunda fila encontramos primariamente la celda normal D. Además observamos que la celda B está *ocupando lugar* en esta segunda fila, porque "cae" de arriba debido a la indicación de `rowspan`, es decir, B se extiende *descendiendo* por dos filas. La celda E nos queda a la derecha de B y encolumnada con la celda C de la primera fila. En la tercera fila encontramos cuatro celdas normales. Las celdas G y H determinan dos columnas, que son las que ocupa la celda B

debido a su `colspan` de 2. Si recorremos las filas podemos encontrar que todas tienen cuatro celdas, reales o "heredadas" de otra fila:

- la primera fila tiene dos celdas normales más una celda con `colspan` 2 ($1+1+2=4$)
- la segunda fila tiene dos celdas normales más una celda que "cae" desde arriba y con `colspan` 2 ($1+1+2=4$)
- la tercera fila tiene cuatro celdas normales ($1+1+1+1=4$)

Al final de este capítulo pueden encontrarse ejercicios sobre marcado de tablas.

2.10 ETIQUETAS DE SECCIÓN

En el estándar HTML existen etiquetas que se utilizan meramente para contener a otras, como hemos visto en el caso del agrupamiento, por ejemplo con listas. Existe otro grupo de etiquetas que también contiene a otras pero con el objetivo de demarcar **secciones** del documento, por ejemplo los encabezados y los pies de la página.

Este grupo de etiquetas es tal vez el que más cambios tiene entre la versión HTML 4 histórica y el HTML 5 actual. Se han agregado nuevas marcas que corresponden a la semántica de, como las llamamos hoy, *aplicaciones web*. Las interfaces de estos documentos no son del estilo "artículo de Wikipedia". Quien utilice Facebook sabrá cuánto se aleja ese tipo de interacción respecto de la página de una enciclopedia. Son nuevos usos de la tecnología que influyen en el diseño del estándar.

Un hecho interesante de estas etiquetas es que no tienen una representación gráfica predeterminada en los navegadores. Exploraremos algunas de ellas:

```
<header>Este es el encabezado del documento</header>
<section>En esta sección tenemos las entradas del blog:
    <article>Primer artículo</article>
    <article>Segundo artículo</article>
</section>
<aside>
    Aquí tenemos las publicidades
</aside>
<footer>Todos los derechos reservados</footer>
```



Figura 2.13: Ausencia de gráfica específica en las etiquetas de sección

Los elementos `header` y `footer` se utilizan para marcar el encabezado y pie del documento. También pueden utilizarse para encabezados o pies de secciones.

Los elementos `section` determinan grupos de contenido que está relacionado temáticamente. Por ejemplo, en una página podemos tener un conjunto de noticias por un lado y la información de contacto por otro. Esas serían dos secciones de la misma página.

Los `article` son utilizados para delimitar porciones del contenido que pueden ser distribuidas por separado. Por ejemplo, las entradas de un blog pueden leerse por separado porque su significado no está asociado al contexto en el que se lee. Lo mismo ocurre con los artículos en un diario: son entes de significado independientes. En cambio los `section` forman parte de un documento mayor y pierden sentido si los sacamos del contexto original.

La etiqueta `aside` (literalmente, *al lado*) se utiliza cuando parte del contenido está relacionado pero de manera *tangencial* al contenido principal de la página. Este tipo de información se suele ubicar en una columna al lado del texto principal y es utilizado, típicamente, para publicidades o para frases y destacados del contenido principal. Si bien este contenido es útil, la marca sugiere que se puede omitir sin alterar el significado del texto principal al que acompaña.

La utilización de estas marcas de sección, como hemos dicho, es lo más novedoso del estándar. La decisión de una u otra marca puede quedar a consideración del autor y algunos casos dependerán del criterio de cada uno. Esto es así porque la comprensión de la semántica de cada parte del contenido es una cualidad simbólica humana, como hemos visto, y difícilmente pueda ser validada por una computadora.

2.11 CONTENEDORES GENÉRICOS

No siempre es posible encontrar, entre todas las propuestas del estándar, una marca que tenga la carga semántica exacta de algún contenido o parte de la página web que queremos construir. Existen para estos casos dos etiquetas que no cargan significado, y que se pueden entender como **marcas genéricas**.

También es común encontrar porciones de páginas web que no tienen un significado específico ni relación con el contenido principal, sino que son utilizadas para materializar algún proceso mediante un lenguaje de programación. Por ejemplo, ¿qué marca utilizaríamos para delimitar el *chat* de Facebook? ¿Es una sección? ¿un aside? ¿tal vez una tabla?

Experimentemos estas dos opciones de marcas genéricas presentes en HTML:

```
<div>Este es un contenedor que <span>no tiene</span> significado</div>
<div>Esta es otra caja</div>
```



Figura 2.14: Utilización de marcas genéricas `<div>` y ``

Las etiquetas genéricas `<div>` y `` están fuertemente relacionadas con la tarea de **maquetación**, es decir, la determinación visual de una página web y cómo debe verse en pantalla. La determinación visual es **ortogonal** a la semántica. Dos conceptos son ortogonales cuando se puede pensar en uno de ellos sin involucrar al otro. Por eso necesitamos marcas que no introduzcan semántica donde no la hay.

Un ejemplo claro de contenedor genérico es el formateo de texto en varias columnas. Aunque se está trabajando en una solución nativa del navegador, hoy en día no existe una manera sencilla y portable de hacerlo. Por lo tanto, separaremos el texto en varios elementos `div` y utilizaremos CSS para ubicar una "columna" al lado de la otra. Esta etiqueta `div` está siendo utilizada solamente para dar formato gráfico a los datos y no forma parte del significado.

La etiqueta `span` se utiliza en el mismo sentido, pero para marcar texto. Para mencionar un ejemplo, puede ocurrir que una marca comercial deba ser utilizada siempre con su color distintivo, imaginemos, azul. En todo texto en donde aparezca mencionada la marca deberá escribirse en color azul. De esta manera, cada vez que se mencione esta palabra se hará de esta forma:

```
<p>Los productos de la marca <span>Acme</span> son los mejores.</p>
```

No tenemos una etiqueta mejor para realizar este marcado, por lo que utilizamos `span` y mediante CSS indicaremos el color azul requerido.

La diferencia entre `div` y `span` tiene que ver con su comportamiento visual en relación al estándar CSS, que lo omitiremos hasta el próximo capítulo. Como primer acercamiento, es suficiente considerar a `div` como una *caja* que contendrá más elementos y `span` como una etiqueta para marcar algunas *palabras* de un texto.

Estas dos etiquetas genéricas son caracterizadas en el estándar como "última opción", es decir, como último recurso cuando no encontramos una marca más adecuada para nuestros significados. Antes de la aparición de HTML 5, la etiqueta `div` fue una de las más utilizadas, ya que la transición de páginas web a interfaces de aplicaciones exige muchos contenedores y comportamientos por fuera de las intenciones del estándar original. Con la introducción de `section`, `article`, `aside` y demás etiquetas de sección se han creado marcas específicas que cubren esta falta. Esto hace que `div` se utilice, con acierto, cada vez menos. Otra forma de verlo es que las nuevas secciones son las viejas `div` más refinadas y especializadas.

2.12 CÓDIGO FUENTE, RENDERIZACIÓN Y ESPACIO BLANCO

Hasta el momento hemos utilizado rápidamente el navegador para ver en pantalla el resultado de nuestros documentos HTML. Profundicemos un poco en el proceso que se está llevando a cabo.

Cuando escribimos nuestras etiquetas en un archivo con extensión `.html` estamos creando el **código fuente (source)** de un documento web. Para escribir el código fuente, lo más recomendable es utilizar un editor de texto pensado para programadores y que,

al menos, nos provea de coloreado automático del código. De esta manera trabajaremos más rápidamente y con mayor chance de hacerlo bien en el primer intento.

Al hacer doble click sobre este archivo nuestro sistema operativo está reconociendo que se trata de un documento HTML, por lo que buscará cuál es el programa que se encuentra asociado a este tipo de documentos. Lo más probable, y lo que hemos supuesto hasta aquí, es que el documento se abrirá en un navegador web. El navegador en cuestión dependerá de las particularidades de la computadora y las preferencias de usuario.

Cuando el navegador reconoce que está abriendo un documento HTML procede a interpretarlo. En primer lugar determinará a qué estándar corresponden las marcas y luego armará el árbol del documento a partir del código fuente. Si el código fuente cumple con el estándar que marca W3C será mucho más fácil para el navegador lograr su tarea. En caso contrario deberá decidir cómo corregir los problemas, algo que no está normado por el estándar y queda a criterio del fabricante del navegador.

Como resultado, hemos visto que el navegador crea en pantalla una representación gráfica del documento, tomando decisiones como colocar negritas, itálicas, agrandar ciertos textos o colocar viñetas. Algo que nos parece muy natural, pero que también es una decisión predeterminada, es el hecho de que el texto sea negro y el fondo de la página sea blanco. Este proceso de crear un "dibujo" del documento HTML a partir de la interpretación del código fuente se denomina **renderización**.

Observemos un ejemplo para precisar un punto importante de la renderización: el tratamiento que se hace del **espacio blanco** en el código fuente.

```
<p>Este es un párrafo
```

```
que tiene saltos de línea y espacios de más</p>
```

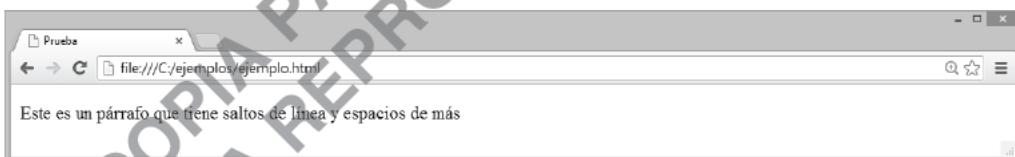


Figura 2.15: Los espacios del código fuente no se trasladan a la renderización

Es claro que el navegador realiza un tratamiento especial de los espacios blancos que encuentra en el código fuente. Al renderizar el contenido, el navegador no traslada literalmente los **saltos de línea**, las **tabulaciones** ni los **espacios**. Esto es, en principio, beneficioso para los programadores porque permite tabular el código fuente para mostrar el anidamiento de las etiquetas. Este tabulado del código nos permite visualizar más fácilmente la estructura del documento, y hace que escribir y leer el código fuente sea mucho más agradable y veloz. Podemos imaginar que, si al renderizar el documento se trasladaran las tabulaciones y saltos al resultado visual, no sería posible tabular el código ya que modificaría el aspecto de la página vista por el usuario.

El navegador está obligado por el estándar a *recolectar* las secuencias de caracteres consecutivos que sean espacios blancos (saltos de línea, tabulaciones y espacios) y *reducirlos*

a un sólo espacio. Por esta razón, el párrafo del ejemplo anterior se renderiza como una sola línea de texto, sin saltos ni espacios múltiples.

Ahora bien, ¿qué sucede si nuestro documento *tiene* que mostrar esos espacios o saltos de más? Puede existir el caso en que sea necesario tener varios espacios juntos o que sea pertinente forzar saltos de línea.

Veamos primero una alternativa para los saltos de línea:

```
<p>Este párrafo tiene <br /><br /> saltos de linea forzados</p>
```

La etiqueta `
`, *break*, indica al navegador que realice un salto de línea. De acuerdo al estándar, debe utilizarse cuando el salto de línea forme parte del contenido, y no con propósitos de formato visual. Algunos datos, como las direcciones postales o las poesías, están compuestas esencialmente de varias *líneas*. Para estos casos se utiliza `br`, es decir, cuando los saltos forman parte del contenido mismo. No utilizaremos `br` simplemente porque necesitamos trasladarnos hacia abajo. Ese tratamiento visual debe realizarse con CSS. Nótese que `br` pertenece, al igual que `img`, al grupo de las etiquetas vacías.

Haciendo uso de CSS también puede modificarse la visualidad de `br`, algo que quizá no es evidente a simple vista por tratarse de un simple salto. La etiqueta `br` plantea un dilema inherentemente complejo, porque se utiliza para marcar saltos semánticos y no saltos visuales, algo que nos resulta muy difícil de comprender. Muchas veces esos saltos semánticos se representan gráficamente como saltos visuales, pero no siempre. Si existen restricciones de espacio, podemos optar por modificar el comportamiento de los saltos con CSS, para mostrar un guion con cierto color y evitar el salto de línea. Algunas poesías muestran el cambio de verso con una barra.

También es interesante observar el comportamiento de otra etiqueta, la marca `<pre>`:

```
<pre>Este párrafo  
ha sido  
preformatado</pre>
```

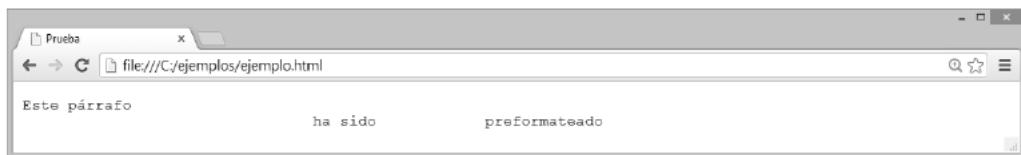


Figura 2.16: Renderización de etiqueta `<pre>`

La etiqueta `<pre>`, si bien es muy poco utilizada en la práctica, es interesante porque le indica al navegador que el texto marcado está **preformatado** y, como tal, no se debe modificar. El navegador respetará saltos, espacios y tabulaciones dentro de esta marca. Un uso común de esta marca aparece cuando precisamos mostrar al usuario código fuente de algún lenguaje, de manera que no se altere su formato ya establecido.

Es muy importante tener presente la diferencia entre **código fuente del documento** y **representación gráfica renderizada**. Nuestro trabajo al escribir HTML es generar el código fuente necesario para que el *resultado* de la renderización logre la visualidad

deseada. Puede verse el código fuente de cualquier página web presionando **CTRL+U**, o buscando la opción "Ver código fuente" en nuestro navegador preferido. Encomendamos y recomendamos al lector la tarea de inspeccionar el código fuente de sus sitios web preferidos, y fomentamos la curiosidad sobre las formas en que están hechas las páginas web que utilizamos todos los días. Puede resultar intimidante al principio, pero luego de un tiempo descubriremos que no es más, ni menos, que HTML.

2.13 ENTIDADES HTML

El estándar HTML provee un mecanismo para colocar en el documento un carácter tipográfico (un signo o símbolo) que no pueda ser representado fácilmente. Por ejemplo, podemos necesitar colocar un signo de *copyright* (©), el signo monetario Euro (€) o cualquier signo matemático (π), entre muchos otros. Estos caracteres especiales se indican con **referencias de carácter**, construcciones especiales del código fuente históricamente llamadas **entidades**.

Si bien las entidades HTML se pueden indicar con ciertos códigos de carácter, o sea números, lo más común es que se utilicen las **entidades con nombre**. Veamos algunos ejemplos para visualizarlo más claramente:

```
<p>Este párrafo tiene un símbolo de copyright &copy; y cuesta &euro; 10</p>
<p>Colocaremos varios espacios &nbsp;&nbsp;&nbsp;&nbsp; así</p>
<p>También tenemos acentos: &aacute; &eacute; &iacute; </p>
<p>Esta es una entidad definida con su código: &#8482;</p>
```

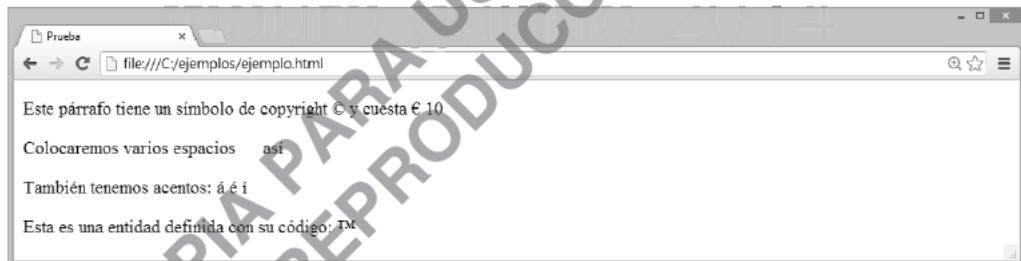


Figura 2.17: Renderización de entidades por parte del navegador

Todas las entidades respetan el mismo formato: comienzan con un signo **ampersand** (&) y finalizan con un punto y coma (;). La lista de entidades existentes puede encontrarse en el estándar y rápidamente en muchos sitios de Internet.

Las entidades son una manera de indicar al navegador que en cierto lugar debe dibujarse un carácter especial. Este carácter es, básicamente, todo signo que no pueda escribirse con un teclado en inglés. En el ejemplo encontramos algunas entidades que representan los caracteres acentuados que utilizamos en el castellano, aunque esto no quiere decir que nuestros documentos no puedan tener acentos escritos con nuestros teclados. Si la configuración del documento y del navegador es adecuada podemos escribir acentos en el código fuente. Estas consideraciones están relacionadas con los llamados **encodings**. El apéndice C aporta más información sobre los encodings.

En el ejemplo anterior proveemos una solución al inconveniente de varios espacios seguidos. Hemos visto que el navegador recolecta y compacta el espacio en blanco al renderizar, por lo que si separamos palabras con varios espacios sólo veremos renderizado uno de ellos. Si colocamos la entidad que *representa* al espacio blanco (de *non-breaking space*) lograremos el efecto deseado ya que el navegador no suprime nunca las entidades: se limita a reemplazarlas por el signo que corresponde.

Observemos ahora dos entidades interesantes:

```
<h1>Bienvenidos a A & F</h1>
<p>Para crear títulos utilice la etiqueta &lt;h1&gt;</p>
```



Figura 2.18: Entidad para el ampersand y para los signos menor y mayor

La entidad & representará al signo ampersand (&). Esta entidad es necesaria porque el signo ampersand es para HTML un **caracter de control**. Es decir que el navegador, cada vez que encuentra un ampersand, supondrá que se está comenzando una entidad. Si lo que sigue a este ampersand no es un nombre o código de entidad válido estaremos frente a un error de sintaxis en el documento, que lo convertirá en un documento inválido desde el punto de vista del estándar. La entidad para el ampersand garantiza que se dibujará el signo adecuado, en lugar de procesar una supuesta entidad que no es tal.

Algo similar ocurre con los signos < y >. Ambos son caracteres de control que, como sabemos, indican el comienzo de una etiqueta HTML. Cuando precisemos mostrar al usuario estos signos, y para no generar un error de sintaxis en el documento, será necesario utilizar la entidad que representa a estos caracteres: < para el signo < (less than) y > para el signo > (greater than). El navegador ve a la entidad como una orden para posicionar en ese lugar el *dibujo* del signo correspondiente.

En el ejemplo precedente, la utilización de < y > genera que se dibujen los signos menor y mayor. Si hubiéramos escrito en el código fuente los *caracteres* en lugar de sus *entidades*, el navegador hubiera interpretado un título h1 dentro del párrafo.

Es muy importante comprender la diferencia de significado que existe en ese párrafo para el navegador, o sea, entre la utilización de los signos versus sus entidades. La importancia radica en que, aunque parezca inverosímil ahora, estas posibles confusiones dan lugar a serios problemas de seguridad conocidos como **inyecciones**.

2.14 FORMULARIOS HTML

En el capítulo anterior mencionamos que en algunos casos es necesario enviar información desde el cliente hacia el servidor. Concretamente ilustramos con un formulario de contacto que generaba un paquete POST. Veamos ahora cómo se implementan en HTML los campos necesarios para que un usuario ingrese información y pueda enviarla hacia el servidor. El estándar define los llamados **formularios HTML**:

```
<form action="http://localhost/contacto.php" method="post">
    <input type="text" name="nombre" />
    <input type="submit" />
</form>
```



Figura 2.19: Formulario con campo de texto y botón de submit

Para delimitar un formulario HTML se utiliza la etiqueta `<form>`. Esta marca tiene dos atributos obligatorios: `action` indica cuál es la URL hacia la cual deben enviarse los datos del formulario y `method` indica qué método de HTTP debe indicarse en la petición que generará el navegador.

Dentro de la etiqueta `form` encontramos los **campos** que conforman el formulario, es decir los diferentes elementos que utilizará el usuario para ingresar la información que será enviada. El formulario del ejemplo anterior sólo tiene un campo para ingresar texto, marcado con la etiqueta vacía `<input>`. La etiqueta `input` tiene un atributo `type` que determina el tipo de campo para mostrar. El `type` por defecto es `text`. El atributo `name` del `input` es muy importante, porque indica el nombre que tendrá este dato al llegar del lado servidor. Los campos del formulario se transforman en **variables**, que del lado servidor se utilizan para realizar alguna acción. El **valor** de la variable es el que el usuario escribe en el campo, y su **nombre** es el `name` del `input`.

Encontramos también otro `input`, que no tiene `name`, y cuyo `type` es `submit`. Cuando el navegador encuentra un campo de este tipo, dibuja un botón, que es el responsable de desencadenar el envío del formulario al ser presionado. Ingresaremos un nombre en el campo de texto y presionaremos el botón de `submit`. Para inspeccionar el paquete enviado al servidor abriremos la herramienta de inspección de tráfico.



Figura 2.20: Formulario enviando petición POST con un campo "nombre"

Ocurren aquí dos cosas importantes. En primer lugar, vemos que el navegador generó una petición POST y la envió al servidor. En el cuerpo del paquete enviado encontramos el nombre que escribimos junto al `name` del `input`. Es decir que se ha conformado una variable, con nombre y valor. En segundo lugar, hemos recibido una respuesta 404, con justa razón ya que el destino del formulario que indicamos fue la URL absoluta `http://localhost/contacto.php`, que no hemos creado.

Realicemos ahora la experiencia con algunas modificaciones:

```
<form action="http://localhost/contacto.php" method="get">
    <input type="text" name="nombre" />
    <input type="text" name="apellido" />
    <input type="submit" />
</form>
```

Completemos ambos campos en el navegador y presionemos el submit, con el objetivo de inspeccionar el tráfico.

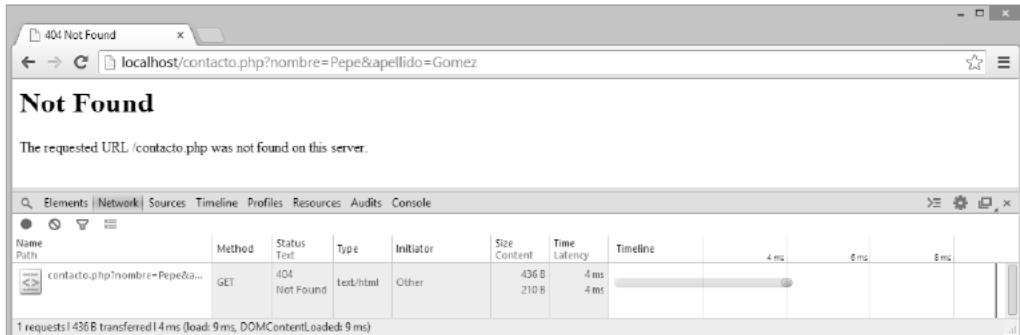


Figura 2.21: Formulario enviando por GET dos campos de texto

Nuevamente tenemos una respuesta 404, con la misma justa razón. Por el momento no es importante ya que estamos analizando solamente el envío del formulario por parte del cliente, y no su procesamiento en el lado servidor.

Hemos cambiado el **método** HTTP por GET, y la consecuencia directa y observable de esta modificación es que las variables (o sea, los datos de los campos) ya no se encuentran en el **cuerpo** de la petición sino en la **URL**. Esta es la única diferencia entre los formularios enviados por GET o por POST. Al utilizar POST, los datos del formulario viajan en el cuerpo de la petición, mientras que al utilizar GET, viajarán en la URL de la petición que se envía, tal como vimos en el capítulo anterior.

La URL ha quedado del siguiente modo: `contacto.php?nombre=Pepe&apellido=Gomez`. El navegador arma las variables (con la forma `nombre=valor`) y las concatena separándolas con ampersands (&). Finalmente, coloca un signo de pregunta (?) entre el `action` que indicamos y el conjunto de las variables, y envía la petición al servidor. Al utilizar el método POST, el mecanismo es casi idéntico, con la diferencia de que las variables armadas no se colocan en la URL del paquete sino en el cuerpo del mismo. En el capítulo anterior exploramos las tres partes que componen un paquete HTTP y hemos visto que la URL de destino se encuentra en la primera parte y el cuerpo en la tercera.

Otra modificación que hemos incorporado en este último ejemplo es el agregado de un segundo campo de texto. Los formularios tendrán varios `input`, uno para cada dato que precisemos recibir del usuario. Cada `input` debe tener su `name` para llegar al lado servidor. Es importante notar que al `input` de tipo `submit` no le hemos asignado `name` y, como consecuencia de esto, es omitido por el navegador cuando prepara las variables para enviar.

Además de nombre, las variables también tienen un valor, que en el ejemplo anterior se toma del ingreso que el usuario realiza en el campo. El valor del `input` se puede definir de antemano, de la siguiente manera:

```
<input type="text" name="nombre" value="Pepe" />
```

Los atributos `name` y `value` se corresponden con el nombre y valor de la variable que se enviará al lado servidor. Si observamos este `input` en el navegador encontraremos que ya está precargado con la palabra "Pepe". Existen otros tipos de campo dentro de los cuales el usuario no puede escribir, por lo que el `value` resulta muy importante.

Es importante notar que el atributo `value` debe utilizarse para *precargar* ciertos valores a los `input` que, como hemos dicho, finalmente serán variables del lado servidor. Esto es conceptualmente muy distinto a indicar *ejemplos* al usuario, para guiarlo en la tarea de completar el formulario. Para este fin existe otro atributo, denominado `placeholder`.

```
<input type="text" name="nombre" placeholder="Indique sus nombres" />
```

Este atributo es utilizado por el navegador para mostrar un texto dentro del `input` que se visualiza como una sugerencia. Se verá en color gris y desaparecerá cuando el usuario escriba dentro del campo, ingresando efectivamente datos para el `value` del `input`. Este texto de ayuda nunca será enviado con el formulario, por lo que es meramente de lado cliente.

2.15 CAMPOS DE FORMULARIO

Veamos más tipos de `input` y campos que pueden utilizarse en los formularios:

```
<form action="procesar.php" method="get">
    <label for="mayorsi">Mayor de edad</label>
    <input type="radio" name="edad" value="si" id="mayorsi" />
    <br />
    <label for="mayorno">Menor de edad</label>
    <input type="radio" name="edad" value="no" id="mayorno" />
</form>
```



Figura 2.22: Formulario con `input` de tipo `radio`

En este ejemplo observamos un `input` de tipo `radio`, que permite la selección mutuamente excluyente de **opciones**. Para que el navegador genere opciones excluyentes es necesario que ambos `input` tengan el mismo `name`. De acuerdo a cuál sea la opción elegida, el navegador enviará un `value` o el otro en la variable llamada "edad". Si precisamos indicar una opción por defecto, podemos hacerlo con el **atributo vacío checked**:

```
<input type="radio" name="edad" value="no" id="mayorno" checked />
```

Además se han incorporado textos descriptivos con la etiqueta `label`, que asocia una **descripción** con un elemento de formulario. La relación se da por la correspondencia del atributo `for` del `label` con el `id` de cada `input`.

El atributo `id` es un atributo opcional de todas las etiquetas HTML, que permite asignarles un **identificador** único que no puede repetirse en el documento. De alguna forma, `id` asigna a este nodo del árbol un nombre único por el cual podemos referenciarlo directamente. En los próximos capítulos este atributo `id` será muy utilizado.

Otro campo muy utilizado es el **tilde**. Examinemos un ejemplo:

```
<form action="procesar.php" method="get">
    <p>Indique las comidas que prefiere:</p>
    <label for="asado">Asado</label>
    <input type="checkbox" name="asado" value="1" id="asado" checked />
    <br />
    <label for="empanadas">Empanadas</label>
    <input type="checkbox" name="empanadas" value="1" id="empanadas" />
    <br />
    <label for="milasoja">Milanesa de soja</label>
    <input type="checkbox" name="milasoja" value="1" id="milasoja" />
    <br />
    <input type="submit" />
</form>
```

Figura 2.23: Formulario con input de tipo checkbox

Los `input` de tipo `checkbox` se renderizan como rectángulos que pueden ser marcados al hacer un click sobre ellos. Estas opciones son independientes entre sí, a diferencia de los campos de tipo `radio`. Es interesante notar que los `name` y `id` de estos `input` son coincidentes en el ejemplo, pero ésta es una situación meramente circunstancial. El `name` se transformará en el nombre de la variable enviada mientras que el `id` es la identificación usada para relacionar con el `label` descriptivo. En otras palabras, el `name` tiene sentido del lado servidor mientras que el `id` se utiliza del lado cliente.

La utilización correcta de los `label`, tal como se muestra en estos ejemplos, genera un documento correctamente marcado. Esto redunda en un beneficio de accesibilidad por dos razones. Primero, se puede comprobar que al hacer click sobre la descripción se genera el tilde en el campo, tal como si presionáramos directamente el pequeño cuadrado en sí. Esto es muy deseable porque, aunque nos parezca extraño, muchas personas tienen dificultades para posicionar el mouse en una superficie tan pequeña de la pantalla. En segundo lugar, las personas no videntes podrán escuchar una referencia clara sobre qué significa cada opción del formulario, cosa que no sucedería si el texto no tuviera su marca `label` correcta.

Es importante notar que el navegador incorpora o elimina de las variables a enviar los datos de los `checkbox` según cada uno esté tildado o no. Este comportamiento de los `checkbox` puede ser desconcertante. Si elegimos, por ejemplo "asado" y "milanesa de soja", podemos comprobar que la URL generada es:

```
procesar.php?asado=1&milasoja=1
```

Como señalamos anteriormente, este es un caso de `input` en donde, a diferencia del `input` de texto, el `value` no puede ser modificado por el usuario directamente sino que somos nosotros quienes lo indicamos en la etiqueta.

Exploraremos ahora el funcionamiento de las **listas desplegables**:

```
<form action="procesar.php" method="get">
    <label for="paises">Pais de residencia</label>
    <select name="paisresidencia" id="paises">
        <option value="54">Argentina</option>
        <option value="55">Brasil</option>
        <option value="56">Chile</option>
    </select>
    <input type="submit" />
</form>
```

A screenshot of a web browser window titled "Formulario". The address bar shows "file:///C:/ejemplos/ejemplo.html". The page content contains a form field labeled "País de residencia" with a dropdown menu open, showing "Argentina" as the selected option. Below the dropdown is a button labeled "Enviar".

Figura 2.24: Formulario con lista desplegable

Aquí vemos en acción un campo `select`, que presenta una lista desplegable con opciones fijas. Es importante notar que, al enviar el formulario, la variable se conforma tomando el `name` del `select` y el `value` del `option` que esté seleccionado. El texto marcado por las etiquetas `option` (en el ejemplo, los nombres de los países) es el que visualiza el usuario en la lista desplegable, pero el valor real enviado al lado servidor es el señalado en el atributo `value`, generalmente un número o código. Nuevamente, el atributo `for` del `label` se relaciona con el `id` del `select`. Hemos utilizado un `name` y un `id` intencionalmente diferentes en el `select`.

Otro campo comúnmente utilizado es el **área de texto**:

```
<form action="procesar.php" method="get">
    <label for="mensaje">Mensaje de contacto</label>
    <br />
    <textarea name="mensaje" id="mensaje"></textarea>
    <br />
    <input type="submit" />
</form>
```

A screenshot of a web browser window titled "Formulario". The address bar shows "file:///C:/ejemplos/ejemplo.html". The page content contains a form field labeled "Mensaje de contacto" with a text area below it. The text area is empty. Below the text area is a button labeled "Enviar".

Figura 2.25: Formulario con área de texto

El `textarea` nos permite ingresar varias líneas de texto, algo que no es posible con los `input` de tipo `text`. Es el tipo de campo que utilizaremos para el ejemplo del formulario de contacto. Cuando enviamos el formulario, el navegador toma los datos del campo `textarea` y los codifica de una manera especial, ya que según el protocolo HTTP, las URL no pueden contener caracteres como espacios o acentos.

Es interesante notar que si bien los `input` pertenecen al grupo de las etiquetas vacías, como `img` o `br`, tanto `select` como `textarea` son etiquetas del otro grupo, con apertura y cierre.

Por último, existe un input especial, que no tiene representación visual en el formulario, que llamamos **campo oculto**:

```
<form action="procesar.php" method="get">
    <input type="hidden" name="usuario" value="333" />
    <input type="submit" />
</form>
```

Estos campos `hidden` se envían con el formulario como cualquier otro `input`. La única diferencia es que el usuario no obtiene de él una representación gráfica en pantalla y, por esta razón, suponemos que no lo modificará. Estos campos se utilizan para colocar en los formularios datos que precisamos recibir del lado servidor cuando este formulario regrese, pero sin interacción del usuario. Si bien pueden parecer poco útiles ahora, son fundamentales para la programación de lado servidor que abordaremos más adelante. Por lo demás, podemos encontrarlos en la URL (o el cuerpo de la petición si es método POST) como cualquier otro `input` enviado.

Para concluir nuestra recorrida por los formularios HTML debemos destacar que, si observamos las URL generadas por los formularios con método GET, encontraremos las variables que se arman con los `name` y `value` de los campos, pero no hallaremos ninguna referencia a qué tipo de campo correspondía originalmente esa variable. La misma observación cuenta para las variables en formularios con método POST ya que el proceso de envío es el mismo, cambiando solamente el lugar en donde las variables se adjuntan y no la manera en la que se empaquetan para su envío. En otras palabras, si observamos desde el lado servidor, al llegar las variables con los datos del formulario, no podremos decir con certeza si surgieron de un `input` de texto o un `select` o cualquier otro tipo de campo. Por lo tanto, hay *muchos* formularios posibles que pueden generar el *mismo* conjunto de variables.

Por ejemplo, estos tres formularios generarán la misma URL cuando el usuario presione el botón de `submit`:

```
<form action="procesar.php" method="get">
    <input type="text" name="nombre" value="Pepe" />
    <input type="submit" />
</form>

<form action="procesar.php" method="get">
    <input type="radio" name="nombre" value="Pepe" checked />
    <input type="submit" />
</form>

<form action="procesar.php" method="get">
    <input type="hidden" name="nombre" value="Pepe" />
    <input type="submit" />
</form>
```

Cuando se reciba cualquiera de estos tres formularios en el servidor, la consecuencia será que hallaremos una variable llamada "nombre" que tendrá la string "Pepe".

2.16 RESUMEN DE ETIQUETAS Y ATRIBUTOS MÁS UTILIZADOS

Etiqueta	Significado	Atributos obligatorios	Atributos opcionales más usados
h1	Título de jerarquía máxima		
h6	Título de jerarquía mínima		
p	Párrafo		
a	Link a otro documento	href	target
img	Imagen incrustada	src, alt	
strong	Texto importante		
em	Texto con énfasis		
ul	Lista no ordenada		
ol	Lista ordenada		
li	Elemento de lista		
table	Tabla		
tr	Fila de una tabla		
th	Celda con título de una tabla		colspan, rowspan
td	Celda de datos de una tabla		colspan, rowspan
div	Contenedor genérico		id
span	Contenedor de texto genérico		id
br	Salto de línea		
pre	Texto preformatado		
form	Formulario	method, action	id
label	Etiqueta descriptiva		for
input	Campo de formulario		type, name, id, value
select	Lista desplegable		name, id
option	Opción de lista desplegable		value
textarea	Campo de texto multilínea		name, id

2.17 BIBLIOGRAFÍA

Consideramos que no existe mejor recurso que el sitio oficial de W3C: [w3.org](http://www.w3.org). Allí están disponibles todos los estándares y todos los trabajos actuales del consorcio, incluidas las discusiones en curso y las propuestas preliminares.

Existe traducción al castellano de material de divulgación que, si bien no es profundo, puede ser un buen punto de partida: www.w3c.es/Divulgacion. En los últimos tiempos el W3C pone mucho empeño en la creación de material introductorio, de divulgación y de aprendizaje, con un tono diferente al de los estándares propiamente dichos. Gran parte de ese material sólo está disponible en inglés.

Nos parece importante acostumbrarse a leer de las fuentes originales, en este caso los estándares, aún con su lenguaje a veces críptico. Existen muchos recursos "sueltos" en Internet que son antiguos o directamente erróneos.

2.18 EJERCICIOS

1. Busque en el sitio w3.org qué indica el atributo `target` de las etiquetas `a` en el estandar HTML 5. Verifique el comportamiento utilizando el navegador.
2. ¿Qué sucede si se coloca una lista `ul` dentro de un `li` perteneciente a una `ol`?
3. ¿Qué sucede si se coloca una imagen dentro de una etiqueta `a`?
4. ¿Cuál es la diferencia entre la marca `title` que se utiliza en `head` y los títulos como `h1` que se utilizan en el `body`?
5. Arme una tabla con la siguiente estructura:

F	A	J
B	C	H E
N		I
G	K	M

6. Arme tres formularios distintos que, al hacer submit, generen la siguiente URL:

`add.php?name=Pepe&age=30&country=ARG`

7. Arme tres formularios distintos que, al hacer submit, generen la siguiente URL:

`contacto.php?id=30&code=TRR&text=Hola%20mundo`

8. Cree un formulario con un `input` de tipo `checkbox` que no tenga atributo `value`. En el navegador, tilde el checkbox y envíe el formulario. ¿Qué valor se asigna a la variable?
9. ¿Qué efecto tiene el atributo `selected` en los `option` de los campos `select`? Busque en el sitio de la W3C y compruébelo construyendo un documento de prueba.
10. Ingrese a Google y realice una búsqueda cualquiera. ¿Cuántos y cuáles parámetros encuentra en la URL de los resultados? ¿Qué pasa si alteramos o quitamos manualmente alguno de esos parámetros? ¿Puede realizarse una búsqueda tipeando directamente esta URL?
11. ¿Es posible crear un documento HTML que contenga un formulario que, al completarse un `input` de texto y enviarse, genere una petición GET con la URL de resultados de Google, de forma que sea procesada por sus servidores?
12. Arme un documento HTML que tenga cinco imágenes distintas. Utilizando la herramienta de inspección de tráfico, determine: ¿cuántas peticiones en total son necesarias para cargar el documento? ¿Cuánto tiempo total tarda la carga de página?

13. Acceda a un artículo cualquiera de Wikipedia, preferentemente uno que no sea demasiado extenso. Visualice el código fuente de la página. Explore el `<body>` del documento e identifique las etiquetas que se mencionaron en este capítulo. ¿Existen etiquetas que no se hayan demostrado? ¿Cuántas y cuáles son? ¿Con qué estándar está escrito el documento?
14. Cree un formulario de contacto. El usuario debe poder completar su nombre, su email y escribir un texto multilínea con un mensaje. Además, deberá indicar si es hombre o mujer. Utilice la herramienta validator.w3.org para verificar si su documento es HTML 5 válido.
15. Con el formulario de contacto creado en el ejercicio anterior, y usando la misma herramienta de validación, verifique si el documento cumple también el estándar "XHTML 1 Strict".
16. ¿Qué sucede si un documento posee más de un formulario? ¿Qué pasa con las variables de un formulario cuando se utiliza el botón `submit` del otro?

3. CSS

El estándar CSS es un proyecto dependiente del W3C, al igual que HTML y muchos otros. Su sigla proviene de *Cascading Style Sheets*, y los documentos que creamos con este estándar son comúnmente llamados **hojas de estilo**. Las hojas de estilo acompañan a los documentos HTML para determinar la forma en que este contenido es representado en ciertos **soportes**. El soporte más habitual es una pantalla, pero la idea poderosa detrás de CSS es que un *mismo* documento HTML pueda hacer uso de distintas hojas de estilo para representarse visualmente en una pantalla de escritorio, en un proyector, en un celular, ser impreso, o ser leído por un sintetizador de voz. Estos constituyen distintos **media** posibles para nuestros documentos.

Para aprovechar esta ventaja de diseño del estándar es que necesitamos garantizar una fuerte separación entre nuestros datos y su presentación. Si el documento HTML posee estrictamente la información de contenido con su semántica, podremos adaptar la forma visual para muchos soportes diferentes sin realizarle cambios. Simplemente modificaremos su CSS asociado. Esto permite además dividir el trabajo sin que ocurran colisiones, ya que los autores se concentrarán en generar contenido de calidad, mientras que los diseñadores trabajarán para lograr una excelente presentación. Aunque durante muchos años esta ventaja de CSS fue subestimada, hoy en día se vuelve imprescindible su comprensión a raíz de la cantidad de tamaños de pantalla distintos y a la tendencia a crear un diseño adaptado a cada dispositivo en lugar de exigir al usuario navegar con tal o cual hardware.

Entendemos que la importancia de CSS desde la perspectiva del programador viene dada por dos aspectos. En primer lugar, CSS es el único estándar disponible para el formato gráfico de todo el contenido web. Por ende, es una *lengua franca* del lado cliente, al igual que HTML. En segundo lugar, su diseño hace uso de conceptos informáticos interesantes, como el **traversing** de árboles y la **herencia** de propiedades, que aparecen utilizados luego en otras tecnologías, por lo que su comprensión se amortiza a mayor plazo. Abordaremos estos temas a lo largo del capítulo, conjuntamente con el uso práctico de CSS.

3.1 ESTÁNDARES, VERSIONES Y NIVELES

El sistema de versiones de CSS es bastante complejo comparado con otros productos de software. CSS es un estándar que va siendo extendido con el tiempo, pero también es una tecnología que los fabricantes de software están empujando constantemente hacia adelante, en su intento de tener navegadores más potentes y que faciliten el trabajo al desarrollador. Un claro ejemplo es el que mencionamos cuando explicamos el uso de las etiquetas `div`. Hicimos referencia a la división de texto en columnas. Los navegadores fueron imponiendo algunas implementaciones que, discusión más o menos, tarde o temprano terminará normada en el estándar previo acuerdo en el W3C. A partir de ese momento se podrá indicar que un texto debe disponerse en columnas utilizando sólo

CSS y sin más artilugios. El tiempo que pasa desde que hay propuestas de implementación hasta que se aprueban las recomendaciones en W3C es en el que ocurren los "infiernos" de versionado que describimos.

En primer lugar debe decirse que tradicionalmente CSS no tiene versiones sino **niveles** (*levels*). Por lo tanto hablamos de CSS nivel 1, CSS nivel 2, etc. Cada nivel está construido partiendo del anterior, agregando funcionalidad, de manera que todos los navegadores que cumplan el estándar nivel 2 también cumplen el nivel 1 por definición. Se puede decir que cada nivel es un subconjunto del que le sigue. Esto quiere decir que CSS es totalmente **compatible hacia atrás** por su mismo diseño.

El primer estándar CSS, nivel 1, data de 1996. Hasta el nivel 2, que fue aprobado en 1998, todo el estándar CSS se trabajó como una unidad. En 2008 se aprobó una revisión de este nivel, llamado **CSS 2.1** que constituye básicamente el más utilizado hoy en día.

Posterior a la aprobación de este nivel, sin embargo, el W3C comenzó a separar las distintas áreas del estándar en **módulos** más o menos independientes. El trabajo en módulos, en lugar de un estándar monolítico, permite trabajar de manera parcial para generar más acuerdo entre los participantes de forma más eficiente. Esto tiene el efecto de generar estandarización más rápidamente, lo que redunda en beneficio para todos. Nombremos, traducidos al castellano, algunos módulos para entender mejor este trabajo: colores, fuentes, marcos, fondos y bordes, cascada y herencia, transiciones, animaciones, paginación, grilla, tablas, valores y unidades, impresión.

Cada módulo tiene ahora su propio nivel. Como los módulos avanzan de manera independiente, el W3C crea **snapshots** (instantáneas) cada cierto tiempo, para fijar un estado al que se llegó con la estandarización en cada módulo. El último snapshot es el 2010 e incluye el módulo principal de CSS 2.1 y algunos módulos nivel 3. Si fuéramos a construir un navegador, éste sería el estándar que deberíamos implementar.

Para los desarrolladores el asunto es más complejo. Si bien se van logrando acuerdos y creando especificaciones para cada módulo, esto no quiere decir que todas las implementaciones en los navegadores sean capaces de cumplirlo. Con las características más modernas, incluso aquellas experimentales, será necesario un proceso de testeo en los navegadores. Este puede ser un proceso bastante frustrante.

En términos coloquiales hablamos de **CSS 3**, aunque no sabemos a ciencia cierta cuándo se convertirá en un estándar ni qué módulos concretos incorporará. Esta forma de llamar a la tecnología es más bien producto del marketing que del proceso de estandarización. Algunos módulos ya comenzaron el proceso de nivel 4, mientras otros todavía no han concluido el nivel 3. Por lo tanto, hablar de CSS 3, aunque suene muy moderno, es hoy algo impreciso.

Decir que desarrollamos con HTML 5 y CSS 3 es, más que una precisión sobre estándares, una declaración de propósitos. En los últimos años se ha avanzado en la definición de **estándares abiertos**, en detrimento de muchas tecnologías propietarias que lentamente van quedando en desuso, como Adobe Flash. Cuando decimos que trabajamos con HTML 5 y CSS 3 estamos expresando nuestra ideología, abogando por los estándares abiertos.

3.2 FORMAS DE INCLUSIÓN EN HTML

Existen dos formas de relacionar un documento HTML con una hoja de estilos CSS. La primera forma es creando un archivo separado para el CSS y colocando una etiqueta `<link>` en el head del documento, de esta forma:

```
<!DOCTYPE html>
<html>
  <head>
    <link href="estilos.css" rel="stylesheet" type="text/css" />
    <!-- el documento HTML continúa -->
```

La etiqueta `link` indica un recurso externo que está asociado por alguna razón a este documento. Cuando el navegador encuentre esta etiqueta procederá a descargar la hoja de estilos, utilizando la URL provista en el atributo obligatorio `href`, por medio de una petición GET. El atributo obligatorio `rel` (*relationship*) indica cuál es la relación con el documento actual y el valor de este atributo debe ser uno de entre una serie posible de valores. Si especificamos `rel="stylesheet"` estamos diciendo, justamente, que el documento externo descargado *es* una hoja de estilos para este HTML. El atributo opcional `type` es meramente indicativo del tipo de archivo que se encontrará el navegador al descargar el recurso. Es habitual encontrarlo aunque el estándar no lo exige. El ejemplo mostrado, que además es el que se utiliza siempre, corresponde a un **MIME type** que indica un archivo de texto plano (*text*) que contiene lenguaje CSS. Algunos autores omiten este atributo, ya que CSS es el único lenguaje que tenemos, por ahora, para escribir las hojas de estilo.

La otra forma de incluir código CSS es **embebiendo** la hoja de estilos en el documento HTML. Esto se realiza con la etiqueta `<style>` de la siguiente manera:

```
<!DOCTYPE html>
<html>
  <head>
    <style>
      /* aquí colocaremos el código CSS */
      /* estos son los comentarios de CSS, iguales a C */
    </style>
    <!-- el documento continúa -->
```

Ambas marcas, `link` y `style` se utilizan en el `head` del documento. Como exploramos en el capítulo anterior, el estándar HTML exige que la etiqueta `head` agrupe la *metainformación* sobre el documento, dejando el *contenido* concreto de la página web dentro de `body`. Por esta razón, la inclusión de CSS se realiza en el `head`, ya que no es contenido *del* documento sino información *sobre* él. El estándar HTML permite la inclusión de estas etiquetas en otros sectores, pero bajo circunstancias exóticas muy especiales que no hacen a nuestros objetivos por el momento.

Ahora que sabemos dónde incorporar el CSS a nuestras páginas web, incursionemos en la sintaxis específica de este lenguaje creado para definir la gráfica de los documentos.

3.3 SINTAXIS

El lenguaje CSS tiene una sintaxis muy sencilla. Lo difícil de CSS no está en comprender la sintaxis sino en la complejidad gráfica que tenga el documento. En general, CSS es un lenguaje que genera archivos muy largos, ya que cada porción de su código fuente hace modificaciones gráficas muy pequeñas y, si bien esto otorga mucha precisión y detalle, también hace que sea necesario escribir mucho código para lograr el nivel de visualidad al que estamos acostumbrados.

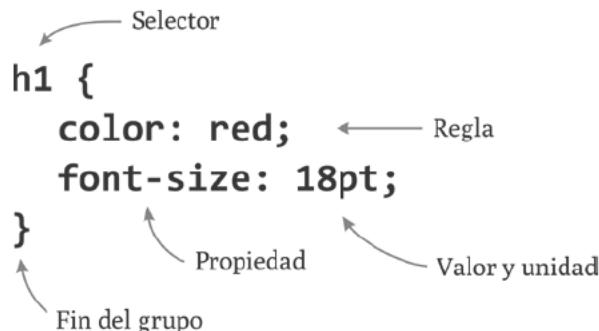


Figura 3.1: Elementos de la sintaxis de CSS

El código CSS se compone de **bloques**. Los bloques se encierran entre llaves y su misión es agrupar **reglas** o sentencias que especifican distintos **propiedades** o aspectos visuales y son el principal contenido del estándar.

Las **reglas** tienen siempre la misma forma: están conformadas por una **propiedad**, y uno o más **valores** separados del nombre con un signo dos puntos (:) . Cada regla debe terminarse con un punto y coma (;). Si bien los saltos de línea y tabulaciones son opcionales, los recomendamos para que el código sea más legible. Las propiedades no pueden inventarse, y tampoco puede colocarse cualquier valor a cualquier regla. Algunas propiedades admiten valores que son **palabras** enumeradas (por ejemplo, los colores), otras admiten **medidas** (por ejemplo, el tamaño de la tipografía) y otras admiten una combinación. Las medidas se acompañan *siempre* con una **unidad**. Son muy pocos los casos válidos en los que podemos utilizar valores numéricos sin expresar una unidad.

En la figura 3.1 observamos un bloque que tiene dos reglas. La primera regla (*color*) indica que el **color del texto** debe ser rojo. El estándar define algunos colores por su nombre pero también puede indicarse el valor RGB. La segunda regla (*font-size*) define el **tamaño de la tipografía**, en el ejemplo, en 18 puntos. El punto es una unidad de medida tradicionalmente utilizada para la tipografía; es la misma unidad que se utiliza en los procesadores de texto. Las reglas no tienen un **orden** particular por el momento: dentro del mismo bloque pueden alterarse sin afectar al resultado.

En el caso de entregarle al navegador una hoja de estilos que tenga **errores de sintaxis**, el intérprete de CSS omitirá las líneas con errores, pasando a la siguiente regla. No habrá un mensaje de error a la vista, como ya hemos visto también con HTML, salvo que ingresemos a las herramientas para desarrolladores.

Otro componente de la sintaxis es el **selector**. Los selectores indican a qué elementos se les debe aplicar el bloque de reglas que acompañan. El selector es, desde nuestro punto de vista, el concepto más importante y complejo de CSS. Le dedicaremos la sección siguiente a su estudio.

Confeccionemos un documento HTML con el CSS embebido, a los efectos de probar en el navegador cuál es el efecto de la hoja de estilos:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Prueba de CSS</title>
    <style>
      h1 {
        color: red;
        font-size: 18pt;
      }
    </style>
  </head>
  <body>
    <h1>Este es el titulo principal</h1>
  </body>
</html>
```



Figura 3.2: Aplicación de los estilos CSS al <h1> del documento

Si hacemos un click derecho sobre el `h1` y elegimos la opción "Inspeccionar elemento", el navegador nos mostrará a la izquierda el árbol del documento y, a la derecha, los estilos que están siendo aplicados al elemento. Esta herramienta es indispensable.

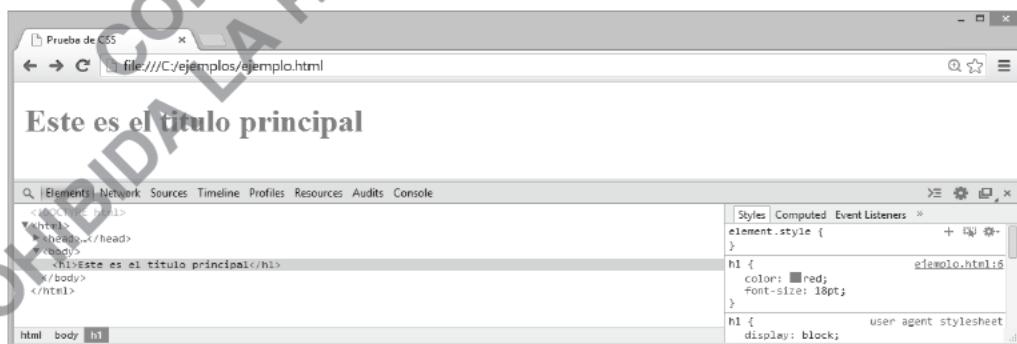


Figura 3.3: Herramienta "Inspeccionar elemento" mostrando el árbol a la izquierda y el CSS a la derecha

3.4 LOS SELECTORES

El **selector** *selecciona* los elementos a los cuales el navegador debe aplicar las reglas visuales que se especifican en el grupo. Los selectores se colocan siempre precediendo el bloque que dominan y pueden confeccionarse con estos tres componentes:

- nombres de etiquetas
- identificadores (*id*)
- clases (*class*)

En nuestro primer ejemplo hemos usado un selector que indicaba un **nombre de etiqueta**: `h1`. Cuando un selector menciona un nombre de etiqueta, las reglas deben aplicarse a *todos* los elementos que sean de ese tipo. Si nuestro documento tuviera muchos `h1`, todos se verán rojos y con el texto en 18 puntos. Este tipo de selector es *muy poco específico*, porque selecciona a todas las etiquetas de un mismo tipo sin distinguirlas. Consideremos otro ejemplo:

```
h1 { color: red; }
h2 { color: blue; }

<h1>Titulo A</h1> <h1>Titulo B</h1>
<h2>Titulo C</h2> <h2>Titulo D</h2> <h2>Titulo E</h2>
```

Intencionadamente omitiremos el código base. El lector está en condiciones de armar un documento HTML válido para probar el ejemplo, insertando las etiquetas HTML en el `body` y los estilos CSS en `style`, como se ha ilustrado antes. El navegador debe mostrar los dos `h1` en rojo y los tres `h2` en azul, ya que estamos seleccionando todas las etiquetas `h1` y todas las `h2` del documento, respectivamente.

Otro componente posible de los selectores son los **identificadores**. Observemos un ejemplo:

```
#titulo-destacado {
    background: yellow; /* color del fondo, por defecto transparente */
}

<h1>Titulo A</h1>
<h1 id="titulo-destacado">Titulo B</h1>
```

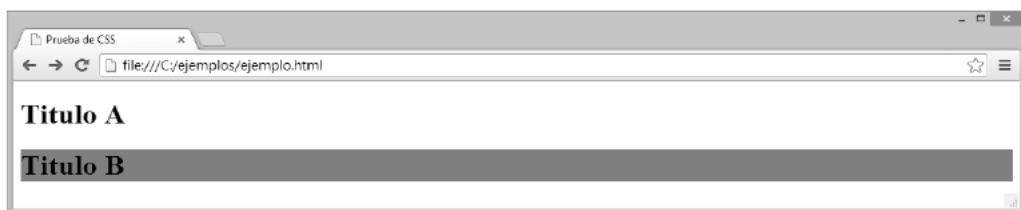


Figura 3.4: Cambio del color de fondo

Cuando un selector especifica un **identificador**, el bloque de reglas se aplica *sólo* al elemento que tenga ese `id`. El `id`, como ya hemos visto en el capítulo anterior, es un nombre que identifica de manera única a un y sólo un elemento del árbol, y que se

asigna con el atributo opcional `id` de cualquier etiqueta. En el selector, el carácter **numeral (#)** indica que *a la derecha* encontramos un `id`. Nótese que el numeral es un **caracter de control** de CSS: no forma parte del identificador. Este tipo de selector, contrariamente al anterior, es *muy específico*, porque se refiere a un solo elemento preciso de todo el documento.

Recordemos que los `id` no pueden repetirse. Es de destacar también que el selector del ejemplo sólo especifica un `id` y no incluye ningún nombre de etiqueta, por lo que es indistinto si se trata de un `h1` o cualquier otro elemento. También es destacable que CSS permite utilizar el **guion medio (-)** en los nombres de identificadores y clases. Esto no suele ser posible en otros lenguajes ya que este signo representa casi siempre la operación de resta aritmética.

En tercer lugar, un selector puede nombrar una **clase**. Podemos imaginar las clases como conjuntos que se usan para agrupar a varias etiquetas con misma visualidad.

```
.poesia {  
    text-align: center; /* alinear al centro */  
}  
  
<p class="poesia">Línea 1</p>  
<p class="poesia otra-clase">Línea 2</p>  
<p>Línea 3</p>  
<h1 class="poesia">Un título</h1>
```

El carácter de control **punto (.)** indica que *a la derecha* encontramos el nombre de una **clase** y no forma parte de este nombre. El grupo de reglas se aplicará a *todas* las etiquetas que pertenezcan a esa clase. Dentro de una clase podemos agrupar, de ser necesario, etiquetas de distinto tipo, y una etiqueta puede pertenecer simultáneamente a varias clases. Para indicar que un elemento pertenece a cierta clase utilizamos el atributo opcional `class`, cuyo valor es el nombre de las clases a las que pertenece, separadas con un espacio si fueran varias. Tanto `class` como `id` son **atributos globales**, es decir que se pueden utilizar en todas las etiquetas. El selector de clase es intermedio respecto de cuán específico resulta, si comparamos con los dos tipos anteriores que eran muy generales o muy específicos.

Podemos resumir estos tres tipos de selectores de la siguiente manera:

- `h1` selecciona a todos los elementos que sean `h1`
- `#titulo-destacado` selecciona al elemento con `id="titulo-destacado"`
- `.poesia` selecciona a todos los elementos que sean de la clase "poesia"

Los selectores indican al navegador un criterio de búsqueda. El navegador explorará el árbol del documento, recorriéndolo en busca de elementos que cumplan con el selector que estamos especificando, ya sean estos sencillos selectores con los que hemos comenzado, o los complejos selectores que abordaremos a medida que progresemos en el capítulo. Es posible que no hallemos ninguna etiqueta que cumpla el selector, y esto no es necesariamente un error. Podemos tener una hoja de estilos que se aplica a muchas

páginas web de nuestra aplicación, por medio de la etiqueta `<link>` en todos los documentos, por lo que es posible que algunos grupos de reglas sólo encuentren aplicación en ciertas páginas y no en otras. Esto es habitual.

Los selectores se empiezan a complejizar cuando descubrimos que también pueden especificarse nombres de etiqueta *a la izquierda* de los caracteres de control:

- `p#principal { color: gray; }` pondrá en gris el texto *del elemento con id="principal", pero sólamente si es un párrafo.*
- `a.noticias { color: green; }` pondrá en verde el color de *los elementos de la clase "noticias", pero sólo aquellos que sean links.*

Es decir que si utilizamos nombres de etiqueta a la izquierda de los caracteres de control podremos hacer que el selector sea más específico.

Otro carácter de control que puede aparecer en los selectores es el **espacio**. Esto puede resultar sorprendente, ya que el espacio no suele ser un carácter importante en los lenguajes, pero para CSS es determinante en muchas circunstancias.

```
p .destacados { /* notar el espacio antes del punto */
    text-decoration: underline; /* subrayar el texto */
}

<span class="destacados"> A </span>
<p> B <span class="destacados"> C </span> </p>
<p> D <strong> E <span class="destacados"> F </span> </strong> </p>
```



Figura 3.5: Textos subrayados sólamente si poseen la clase y están dentro de un `<p>`

El espacio indica **descendencia**, es decir, que permite seleccionar elementos solamente si son descendientes (hijos, nietos, etc.) de otros elementos, como si se tratara de una *ruta* para buscar elementos. Lo que recomendamos es leer el selector de derecha a izquierda, de esta forma: `p .destacados` selecciona a todos los elementos que sean de clase "destacados" y que estén *dentro de* un párrafo. En el ejemplo estamos agregando un subrayado al texto por medio de la regla `text-decoration`. Es importante notar que el texto que se modifica es el que está presente dentro de la etiqueta con clase "destacados" y no se afecta al `p`, que está presente sólo para indicar dónde buscar el elemento. Esto puede comprobarse observando que las letras "B", "D" y "E" no están subrayadas. De la misma forma, la letra "A" no está subrayada porque no cumple el selector: es un elemento de clase "destacados" pero no está dentro de un párrafo. Asimismo, la letra "F" está subrayada porque está efectivamente dentro de un párrafo, o más precisamente, es *descendiente* de algún párrafo. Esto quiere decir que "descendiente" no es sólo *hijo directo*.

Es importante considerar cómo cambia el significado si se omite el espacio:

- el selector `p.destacados` (todo junto, sin espacios) selecciona los párrafos que sean de clase "destacados" como ya hemos visto
- el selector `p .destacados`, en cambio, selecciona los elementos de clase destacados que estén dentro de un párrafo.

En el primer caso estamos pidiendo al navegador que busque párrafos de *cierta* clase en *cualquier* lado y en el segundo caso no estamos especificando qué tipos de etiquetas son sino *cierta* clase y *dónde* están.

Otra posibilidad que nos brindan los selectores es realizar **agrupamiento** para poder escribir menos líneas de código. Esto se realiza con el carácter de control **coma** (,):

```
a#contacto, a#inicio {  
    color: red;  
}
```

Con este selector estamos pidiendo que se muestre en rojo el texto para el link con `id` "contacto" y *lo mismo* para el link con `id` "inicio". No se establece ninguna conexión a ambos lados de la coma. Con otras palabras, no estamos precisando cuál es la relación entre estos dos links: pueden encontrarse en cualquier lugar del documento. La coma nos permite agrupar selectores que seleccionen elementos con la misma visualidad, para no repetir el grupo de reglas varias veces. Nuevamente, no indica que un elemento esté adentro del otro ni mucho menos.

Hemos mencionado que el navegador busca en el árbol del documento los elementos que coinciden con el selector que indicamos. Para realizar esta búsqueda debe realizar un **recorrido** del árbol. Este recorrido se conoce como *tree traversal* o *traversing* y es una operación genérica muy importante que se realiza sobre los árboles. Más adelante exploraremos otras posibilidades que CSS nos ofrece para hacer algunos *traversing* más complejos, que serían bastante trabajosos de no contar con el lenguaje de los selectores. Este concepto de *traversing* también se aplica a todos los documentos que pueden representarse con árboles, como los documentos XML. Para ilustrar con un ejemplo, consideremos esta tarea: buscar todas las palabras de este libro que sean conceptos en negrita y cambiarles el tamaño a 15 puntos, pero solamente la primera vez que aparecen mencionadas en cada párrafo. Realizar esta transformación exige hacer *traversing* sobre el árbol del contenido.

3.5 ESPECIFICIDAD DE LOS SELECTORES

Hasta aquí hemos utilizado bloques de reglas que no presentaban colisiones, es decir, cada selector indicaba elementos diferentes y ningún grupo se aplicó en simultáneo. Cada grupo modificaba algunos elementos por su lado sin interferir con los demás. Pero el estándar permite que distintos grupos apliquen de manera simultánea varias reglas al *mismo* conjunto de etiquetas, y de hecho es una de las características más importantes de esta tecnología. Este aspecto de CSS es lo que llamamos **cascading**. Observemos algunos ejemplos para comprender cómo se leen y aplican las hojas de estilo:

```

a { font-size: 30pt; }
a { color: red; }
a { color: green; }

<a href="nada.html">Link 1</a> y <a href="nada.html">Link 2</a>

```

Como efecto de aplicar estos tres grupos a un documento veremos que todos los links quedarán con tamaño 30 puntos y en color verde. Es decir que el navegador lee la hoja de estilos en orden, de arriba hacia abajo, y aplica las reglas a medida que las encuentra. La ultima regla **color** (*green*) está sobreescritiendo a la anterior (*red*). Por lo tanto, el navegador determina cuál regla vale en función del **orden**. Nótese que esto se realiza *regla por regla, y no grupo por grupo*; esto queda demostrado porque la regla **font-size** se ha aplicado en el primer grupo y no fue eliminada con los subsiguientes. Si utilizamos la función "Inspeccionar elemento" sobre un link veremos este interesante resultado:



Figura 3.6: Predominio de la última regla (los links se dibujan de color verde)

El navegador nos muestra que una de las reglas ha sido sobreescrita por otra, y por eso aparece tachada. En este caso, la sobreescritura se da porque la regla que indica "verde" está *después* de la que indica "rojo".

Pero ¿qué ocurre si realizamos la experiencia de la siguiente manera?

```

a#contacto { color: red; }
a { color: green; font-size:9pt; }
a { color: black; }

<a href="nada.html" id="contacto">Link 1</a> y <a href="nada.html">Link 2</a>

```

La situación clave para observar en esta experiencia es que el "Link 1" ha quedado de color rojo, mientras que el otro ha quedado negro. Hemos visto que el navegador aplica las reglas leyendo los estilos en orden, pero esto no fue del todo completo. El navegador aplica las nuevas reglas que encuentra para las etiquetas, solamente si el selector es de *la misma o mayor especificidad* que el anteriormente aplicado. Detallemos el proceso siguiendo paso a paso la aplicación de los estilos:

- El primer grupo aplica color rojo al link que tiene `id="contacto"`. Este selector tiene especificidad 101. Este cálculo se detalla más abajo.

- El segundo grupo aplica color verde a todos los links de la página. El "Link 2" queda entonces en color verde, pero no el "Link 1", porque hay una regla anterior `color` que se ha aplicado con un selector de especificidad 101 y el selector actual es de especificidad 1. El `font-size` se aplica a ambos links, porque no hay aplicaciones anteriores que considerar.
- El tercer grupo aplica color negro a todos los links. El "Link 2" termina en color negro porque hay una aplicación de color anterior con especificidad 1, y el selector actual también tiene especificidad 1, por lo que gana este último (el negro). El "Link 1" no cambia de color por la misma razón anterior, el color que tiene aplicado fue aplicado con especificidad 101. Sólo un selector igual o mayor puede cambiar el color rojo que tiene el "Link 1".

Es claro entonces que el navegador no aplica simplemente las reglas que va encontrando, sino que *para cada regla* que aplica recuerda cuál fue la especificidad del selector que dominaba el *grupo* al que pertenece esta regla. Posteriormente, aplica las reglas solamente cuando provienen de grupos con selectores de igual o mayor especificidad que las ya aplicadas.

Verifiquemos qué indica el navegador al realizar "Inspeccionar elemento" sobre el link rojo ("Link 1"):



Figura 3.7: Predominio de la regla más específica

Efectivamente nos muestra la regla que está "ganando" debido a que es más específico su selector. La regla que vale es mostrada arriba de las que han "perdido", que son tachadas.

Es interesante conocer cómo define el estándar el cálculo de la especificidad. Es evidente que no puede estar librado a consideraciones azarosas, sino que es un número que debe surgir de algún cálculo con los componentes del selector. A continuación se indican algunos selectores de ejemplo y se menciona la especificidad de cada uno.

<code>p { }</code>	<code>/* especificidad = 1 */</code>
<code>div p { }</code>	<code>/* especificidad = 2 */</code>
<code>p.resumenes { }</code>	<code>/* especificidad = 11 */</code>
<code>p#primero { }</code>	<code>/* especificidad = 101 */</code>
<code>div p#primero { }</code>	<code>/* especificidad = 102 */</code>
<code>div#contenido p#primero { }</code>	<code>/* especificidad = 202 */</code>

```
<div id="contenido">
    <p id="primero" class="resumenes"> texto </p>
</div>
```

Todos estos selectores pueden utilizarse para alterar, por ejemplo, el color de la palabra "texto" que aparece en el párrafo. La diferencia que existe entre ellos es la especificidad, es decir, con qué grado de precisión se refieren a este párrafo y no a otro.

El navegador descompone el selector y hace el siguiente cálculo:

- por cada nombre de etiqueta, sumar una unidad
- por cada nombre de clase, sumar una decena
- por cada identificador, sumar una centena

Si bien no es habitual calcular la especificidad de los selectores que escribimos, algunas veces el conocimiento de esta forma de cálculo puede ahorrarnos muchos dolores de cabeza. Principalmente cuando ha pasado un tiempo del desarrollo y las hojas de estilo empiezan a tener tamaños considerables y se aplican varias de ellas a la vez.

El estándar CSS llama **cascada** al proceso de tomar una serie de valores declarados para cierta propiedad en cierta etiqueta y ordenarlos según su importancia. El valor más importante será el que se aplique finalmente. Uno de los factores que afecta a la importancia es la especificidad, como hemos ilustrado. Otro de los factores que afecta este orden de prioridades es el origen de los valores. Las declaraciones de valores pueden tener tres orígenes según el diseño del estándar CSS:

- Las declaraciones del **autor** son las que hemos visto. Se refieren al autor del documento y son las reglas que especificamos con CSS cuando creamos los estilos.
- Las declaraciones del **usuario** son las que los navegadores permiten crear a los usuarios según sus preferencias personales. Por ejemplo, el usuario puede necesitar aumentar el tamaño de todos los textos en un 50% o alterar ciertos colores porque tiene dificultades para distinguir algunos de ellos. Estas declaraciones son las de mayor importancia para la cascada.
- Las declaraciones del **cliente** son las que vienen predeterminadas en el navegador. Por ejemplo, que los links se vean azules y subrayados se debe a que el navegador aplica estas reglas CSS incorporadas. Estas declaraciones tienen la menor importancia de las tres.

Son importantes dos consideraciones sobre la existencia de estos tres orígenes de estilos. En primer lugar, el estándar le asigna al usuario final del documento una gran importancia. Las reglas de usuario son más importantes que las reglas de autor y siempre le ganan a éstas. Esto puede ser chocante para profesionales del diseño, que fueron educados para controlar cada milímetro del aspecto visual. La filosofía de la W3C pone en boca del usuario final la última palabra sobre la presentación de los documentos.

En segundo lugar, debemos notar que siempre que aplicamos estilos estamos trabajando inevitablemente con la cascada de CSS. Siempre que aplicamos una regla nueva a algún elemento lo que estamos logrando en realidad es "pisar" alguna regla predeter-

minada del navegador. Cambiar el color a un texto, definir un fondo, establecer un tamaño de texto... todos estos ejemplos han sobreescrito los valores provenientes de la hoja de estilos predeterminada en el navegador, por lo que siempre hemos estado utilizando la sobreescritura de reglas aun sin saberlo. Esto puede comprobarse con la herramienta de inspección, ya que también nos muestra las reglas provenientes del navegador por defecto (en Chrome figuran como *user agent stylesheets*) y cómo son superadas en importancia por las reglas de autor.

3.6 PSEUDOCLASES

Existen unas construcciones de CSS llamadas **pseudoclases** que pueden utilizarse en los selectores para especificar algunas situaciones especiales. Veamos tres ejemplos:

```
a:hover { background: gray; }  
p.cuerpo:first-letter { font-size: 25pt; }  
#nombre:focus { border: 2px solid red; }
```

Las pseudoclases se indican con el carácter de control **dos puntos (:)** a continuación del selector que queremos refinar. Las pseudoclases nos permiten seleccionar algunos aspectos especiales de ciertas etiquetas.

En el primer caso, el fondo de todos los links del documento cambiará a color gris cuando pasemos el mouse sobre ellos. La pseudoclase `hover` indica esta situación. Cuando quitamos el mouse se deja de cumplir el selector, por lo que el fondo vuelve a ser transparente.

En el segundo ejemplo vemos una pseudoclase `first-letter` que nos permite seleccionar la primera letra de los párrafos de clase "cuerpo". Esto permite realizar las llamadas *letras capitales* en los párrafos, como se utilizaba antiguamente en los manuscritos.

En el tercer ejemplo tenemos un selector que habla de un elemento con `id` "nombre" y la pseudoclase `focus` especifica que la regla se aplicará sólo cuando el elemento tenga el foco. Un elemento tiene el **foco** cuando el ingreso por teclado se dirige a ese elemento. Normalmente los sistemas operativos nos indican el elemento que tiene el foco con un cursor que parpadea. La forma más habitual de darle el foco a un elemento es haciendo click en él, aunque no la única. Por acción de esta pseudoclase, cuando el elemento (imaginemos un `input` de texto) tenga el foco, su borde se verá rojo. La regla `border` es un ejemplo de **regla múltiple**, que nos permite definir varias propiedades en una sola declaración. En este caso se está definiendo, en orden: el grosor de la línea (2 pixels), el estilo de la línea (una línea sólida) y el color (rojo). Si quisieramos modificar sólamente el color de la línea podemos utilizar la **regla simple** `border-color`. Cuando quitemos el foco, el selector se dejará de cumplir y el elemento volverá a tener su borde normal.

Las pseudoclases `hover` y `focus` se pueden considerar **eventos** del navegador a los efectos de visualizar su funcionamiento. El estándar las califica como pseudoclases **dinámicas**, porque una etiqueta adquiere o pierde cierto estado a partir de la interacción del usuario con la interfaz.

3.7 SELECTORES AVANZADOS

Exploraremos algunos selectores más complejos que, si bien no se utilizan constantemente, cada vez son más apreciados e ilustran la potencia de los selectores de CSS. El módulo de selectores es el más desarrollado de todo el estándar.

Existe un carácter de control que nos permite indicar **descendencia directa**:

```
p > em { color: red; }

<p> <em>Este texto será rojo</em>, <strong>pero <em>este
no</em></strong></p>
```

El selector `p > em` puede leerse como *todos los elementos em que sean hijos directos de un párrafo*. El `em` que se encuentra dentro de la marca `strong` no es hijo directo. Sí es descendiente, pero por no ser directo no recibe la aplicación del estilo que es dominado por este selector.

Encontramos también una pseudoclase que nos da la posibilidad de reducir un conjunto de etiquetas que responden a un selector. Veamos dos ejemplos:

```
ul.ingredientes li:nth-child(3) { color: maroon; }
table#poblacion tr:nth-child(even) { background: yellow; }
```

La pseudoclase `nth-child` nos permite indicar cuáles nodos, del conjunto total seleccionado, queremos acotar. Podemos leer el primer selector de esta manera: *el tercer li que está dentro de las ul de la clase "ingredientes"*. Nótese que las listas con clase "ingredientes" pueden ser cero, una o muchas en el documento. De las `ul` que existan con esa clase estamos tomando los ítems (`li`) y de ellos, cambiando el color sólo al *tercero* de cada lista. Este índice comienza en 1 y no en 0 como puede suponerse.

El segundo ejemplo de esta pseudoclase muestra cómo se puede utilizar para cambiar fácilmente el fondo de las filas **pares** (*even*) de una tabla. Podemos leer el selector así: *todas las filas en posición par dentro de una tabla con id "población"*. Las **impares** se indican con la palabra *odd*.

Existe otro carácter de control interesante, el **corchete** (`[]`), que es cada día más usado:

```
input.contacto[type="text"] { border: 3px dashed blue; }
select#paises option[value] { font-weight: bold; }
```

El corchete permite reducir, de un conjunto de etiquetas seleccionadas, según sus atributos. En el primer caso puede leerse el selector del siguiente modo: *todos los input de clase "contacto" que tengan su atributo type definido como "text"*. Visualmente, este selector está indicando que los campos de texto que posean esa clase, y sólo los de texto, deben verse con un borde azul de trazo interrumpido ("a rayas").

El segundo ejemplo utiliza la misma mecánica pero sin especificar el valor del atributo. En este caso el selector indica que se debe acotar la búsqueda a los elementos que posean ese atributo en su etiqueta, *cualquiera sea su valor*. En este caso puede leerse así: *todos los option que tengan definido algún value, dentro de un select con id "paises"*. Recordemos

que el atributo `value` es opcional, por lo que existe la posibilidad de que algún `option` no tenga `value` asociado. Los option que sí *tengan* `value` declarado se verán en negrita (debido al `font-weight`).

Por último, mencionaremos otra pseudoclase interesante, la **negación**:

```
#noticias p:not(.destacados) { font-size: 9pt; }
```

Este selector puede leerse así: *todos los párrafos que no pertenezcan a la clase "destacados", que se encuentran dentro de un elemento con id "noticias"*. No resulta difícil comprender esta negación. Dentro de los paréntesis de la pseudoclase `not` puede indicarse cualquier selector, menos la negación misma. Para ilustrar con otro ejemplo, podríamos necesitar seleccionar los `input` de un formulario, *menos* los que son de texto:

```
input.contacto:not([type="text"]) { color: blue; }
```

El significado del selector interior es el mismo al que ya mencionamos al estudiar los corchetes, resumidamente: *que tenga el atributo type declarado como "text"*. Por lo tanto este selector tomará los `input` de clase "contacto", pero que no sean de tipo texto, ya que esa cualidad viene determinada por este atributo `type`.

3.8 HERENCIA DE PROPIEDADES

Hemos hablado extensamente sobre los selectores. A nuestro criterio, los selectores son la ciencia más compleja de todo el estándar y también lo más difícil de la práctica diaria con este lenguaje. Como contrapartida, también creemos que es un aporte sumamente interesante a nuestra caja de herramientas como programadores.

Pero volvamos ahora sobre las reglas visuales, que son la razón de ser última de esta tecnología. Exploramos ya algunas reglas básicas, como cambio de color de texto, algunas propiedades para alterar los fondos, los bordes, entre otras.

Observemos que algunas reglas tienen la cualidad de **heredarse**, de los padres a los hijos, mientras que otras no:

```
p { color: red; border: 2px solid blue; }  
#borde { border: 2px solid blue; }  
  
<p> Este texto será rojo y con borde, y <span>éste será solo rojo</span></p>  
<p> Este texto es rojo y con borde, y <span id="borde">este sí tiene  
borde</span></p>
```

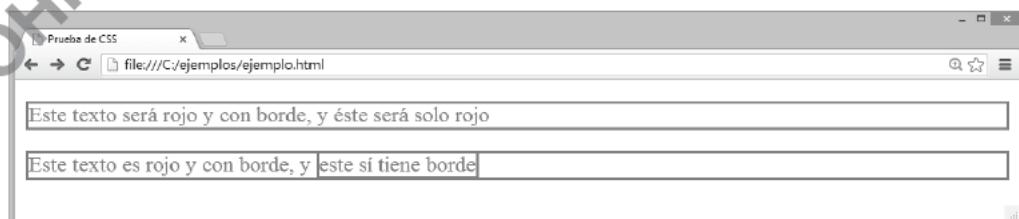


Figura 3.8: La propiedad "color" se hereda de padres a hijos, mientras que "border" no

El estándar define cuáles reglas se heredan y cuáles no. La herencia funciona de manera bastante lógica. Por ejemplo, si definimos un `background` a un elemento, sus hijos *no* heredan el mismo fondo. Tal comportamiento sería caótico porque deberíamos ir hijo por hijo forzando a que tengan fondo transparente. El mismo razonamiento se puede aplicar a la regla `border`. Si los hijos heredaran las propiedades de borde de su padre se generaría un caos gráfico, que implicaría mayor CSS a escribir. Otras reglas es natural que *sí* se hereden, por ejemplo, la tipografía a utilizar. Si definimos que cierto elemento se muestre con cierta tipografía, tiene sentido que todos sus hijos se muestren también de esa forma. Si el sentido común no nos aporta la respuesta, siempre podemos consultar la tabla de propiedades presente en el estándar en el sitio [W3.org](http://www.w3.org).

Las propiedades heredadas influyen entonces en el proceso de cascada. Utilizando la herramienta de inspección de estilos del navegador podemos comprobarlo. Veamos el siguiente ejemplo:

```
p { color: red; }
strong { color: blue; }

<p> Este párrafo tiene un <strong>texto importante</strong> </p>
```



Figura 3.9: Herencia y cascada se aplican simultáneamente

Si realizamos la inspección del elemento `strong` encontraremos que la propiedad `color` se está heredando (*Inherited*) del párrafo padre, y a su vez está siendo sobreescrita por la misma propiedad indicada con el segundo selector. La sobreescritura se denota con la regla `color: red` tachada. Ambos selectores tienen especificidad 1. ¿Qué sucede si invertimos el orden del CSS, colocando el bloque para `strong` primero y el bloque para `p` después?

3.9 INTRODUCCIÓN A LA MAQUETACIÓN

Con el correr de los años se ha consolidado cierta división del trabajo en el desarrollo de sitios web. Típicamente, este proceso puede ser analizado a la luz de **tres roles**, que constituyen los pasos mínimos para construir un sitio web.

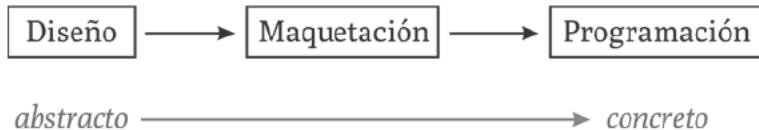


Figura 3.10: Los tres roles mínimos en el desarrollo web

El **diseño gráfico** involucra la definición de los aspectos visuales y tangibles del sitio web. De acuerdo a la magnitud del proyecto puede también involucrar el diseño de la *identidad*, como ser los logotipos, los esquemas de color y hasta el mismo tono de la comunicación. Además puede involucrarse el diseño de la *interacción*, el estudio de la *usabilidad* y diversos aspectos que hacen a la profesión de los diseñadores gráficos, entre otras profesiones relacionadas. En proyectos extremadamente grandes puede necesitarse el aporte de psicólogos y antropólogos. La salida de esta etapa es un conjunto de **imágenes**, que demuestran el producto del proceso de diseño. El diseño es una *idea*.

La **maquetación** es el proceso por el cual se convierten estas imágenes o ideas en tecnologías concretas capaces de funcionar en los navegadores. Es decir que *maquetar* es construir los documentos utilizando los estándares HTML y CSS para que se vean de la manera en que fueron diseñados. La salida de esta etapa es un conjunto de páginas web que se *ven* como el producto final.

La **programación** abarca el conjunto de tareas que más nos interesan. Involucra la programación de las distintas funcionalidades que debe soportar el sitio web o aplicación web. De alguna forma, podemos verlo como el proceso en el que "se le da vida" a las maquetas generadas. Desde esta perspectiva es claro que los programadores necesitan conocer las tecnologías de maquetación, ya que trabajan directamente "sobre" ellas, o mejor, "partiendo" de ellas para definir uno o más programas a su alrededor. La salida de esta etapa es el sitio web concreto, de manera que se *ve* y se *comporta* de la manera en que fue diseñado.

Estos tres roles mínimos son, desde ya, desafiados a medida que el proyecto crece en **escala**. Los sitios y aplicaciones web de un tamaño considerable necesitan el trabajo de muchas más profesiones. Podemos pensar en administradores de red, administradores de bases de datos, consultores en seguridad informática, especialistas en usabilidad de interfaces, traductores y lingüistas, publicistas, especialistas en marketing, especialistas en formatos de video, tipógrafos, abogados, entre muchos otros.

En el caso de un sitio pequeño, los tres roles suelen desempeñarse por un mínimo de **dos personas**. Esto es así porque el rol de maquetación suele ser llevado a cabo por los diseñadores ó por los programadores. Muy pocos profesionales, en comparación, se dedican *solamente* a maquetar lo diseñado por otros, para que sea programado por otros. Desde otro punto de vista, esto también quiere decir que es muy difícil que un diseñador gráfico se dedique a programar o que un programador se dedique a diseñar. De alguna manera, históricamente estos dos roles *parecen* ser mutuamente excluyentes. Sin embargo, también es cierto que la tendencia de la evolución tecnológica genera que cada vez más diseñadores *tengan que* programar. Esto supone un gran desafío vocacional.

Hemos descripto brevemente el ciclo de trabajo habitual para desarrollos web de pequeña y mediana escala. Además, en este capítulo exploramos la sintaxis de CSS y el

funcionamiento general de esta tecnología, luego de haber conocido el estándar HTML en el capítulo anterior. En el camino hemos encontrado algunas propiedades gráficas de los elementos de una página, como el color o los fondos, pero no hemos incursionado en la combinación más compleja de reglas que se necesitan para maquetar un sitio web real. Para ello necesitamos entender ciertos conceptos clave de CSS que son más complejos que un sencillo color o tamaño de texto.

El **modelo de caja** (*box model*) es la parte del estándar CSS que especifica cómo debe dimensionarse y medirse la superficie de un elemento. Antes de hablar específicamente de las dimensiones, examinemos este ejemplo:

```
h1, p, div { border: 1px solid blue; }
strong, em, span { border: 1px solid red; }

<h1>Este es el título</h1> <p>Esto es un párrafo</p> <div>Esto es un
div</div>
<p>
    Este texto <strong>es importante</strong>, este <em>tiene énfasis</em>
    y este tiene <span>algo genérico</span>
</p>
```

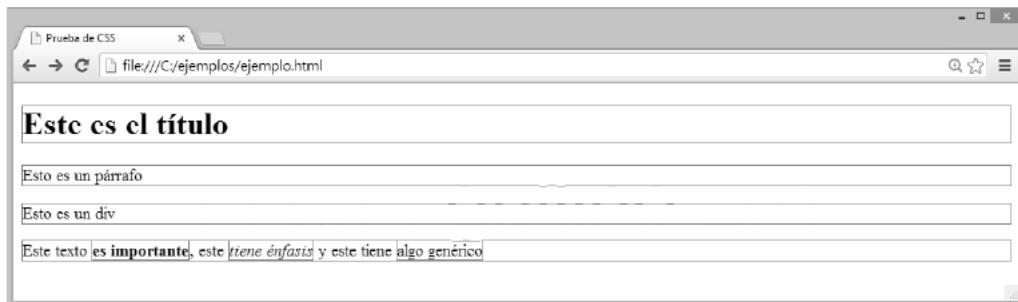


Figura 3.11: Elementos de bloque y de línea

Lo que se observa en el navegador es que hay dos grandes grupos de etiquetas. Las etiquetas que son **de bloque** (*block*), como `h1`, `p` y `div`, ocupan siempre todo el **ancho** disponible y realizan un **salto de línea**. Hemos elegido dibujar el borde de los elementos para ver claramente la superficie que ocupan. Es quizás sorprendente que los elementos que contienen las primeras tres frases lleguen hasta el borde derecho del navegador aún teniendo tan poco texto en su interior. Este comportamiento se debe, justamente, a que son etiquetas de bloque. Si alteramos el ancho de la ventana del navegador veremos que siempre llegan hasta el borde. Podemos decir que los elementos de bloque quedan naturalmente uno *debajo* del otro, como las cajas en una pila.

El otro grupo es el de las etiquetas **de línea** (*inline*). Estos elementos, como `strong`, `em` y `span`, se mantienen *en línea* con el texto. Las etiquetas de línea ocupan la superficie definida por su contenido; cuando más texto incluyan, más ancho ocuparán. Si la tipografía tiene un tamaño mayor, crecerán en altura, y así. Además no producen saltos de línea sino que "corren" junto al renglón. Los elementos de línea permanecen uno *al lado* del otro, como las letras en un renglón.

Los tamaños de los elementos que son *de bloque* se pueden alterar con estas reglas:

```
div { width: 70px; height: 70px; border: 2px dotted blue; }

<div>A</div> <div>B</div> <div>C</div>
```



Figura 3.12: Definición del alto y ancho de elementos de bloque

Los elementos de bloque ocupan siempre todo el ancho y, para determinar su altura, se toma el contenido que posean. Sin embargo, también podemos forzar el tamaño de un elemento con `width` (ancho) y `height` (alto). Si establecemos una o ambas dimensiones, el navegador las respetará. Los elementos de línea, en cambio, no pueden ser dimensionados porque su tamaño depende sólo de su contenido, por definición.

Si bien hemos podido reducir el ancho de los elementos haciendo uso de `width`, los `div` siguen, de algún modo, ocupando todo el ancho. Con otras palabras, se sigue produciendo el salto de línea ya que son elementos de bloque. ¿Cómo podríamos ubicar un `div` al lado de otro? Recordemos que no podemos cambiar la marca sólo porque no nos gusta su comportamiento gráfico.

La regla `float` (**flotado**) se utiliza para permitir que el contenido subsiguiente fluya por el lateral de un elemento. Decimos que el `float` transforma el espacio que un elemento tiene a su lado en un espacio que *puede* ser llenado por los elementos que están debajo.

```
#caja { width: 70px; height: 70px; border: 1px solid red; float: left; }

p { border: 1px dotted blue; }

<div id="caja">CAJA</div>
<p>Este es un párrafo de texto. Como el elemento anterior (div) es de bloque, este párrafo debería estar debajo. Como hemos "flotado" al div, este párrafo ha "subido" hasta aquí.</p>
```

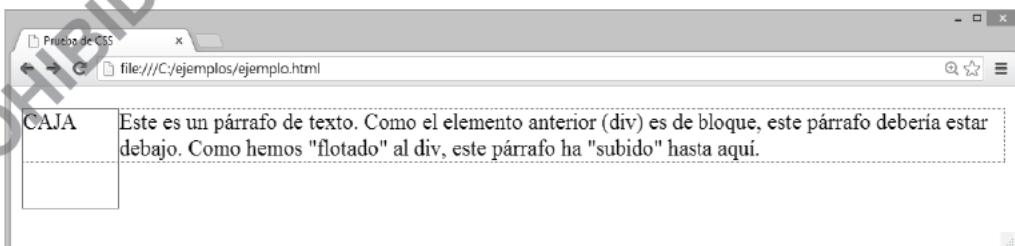


Figura 3.13: Efecto de la propiedad "float"

La propiedad `float` le ha indicado al `div` que se ubique en el margen izquierdo (*left*) y que permita que los elementos de abajo "suban" en el espacio que tiene a su lado. Eso es lo que el estándar define como *flotar*. Es interesante notar que la propiedad `float` se le aplica al `div`, pero el contenido que efectivamente "se mueve" es el párrafo.

El párrafo, por ser un elemento de bloque, sigue ocupando todo el ancho disponible, y por eso llega hasta el extremo derecho del navegador.

Agreguemos dos reglas más, también pertenecientes al modelo de caja:

```
#caja { width: 70px; height: 70px; border: 1px solid red;
         float: left; padding: 20px; margin-right: 30px; }
p { border: 1px dotted blue; }

<div id="caja">CAJA</div>
<p>Este es un párrafo de texto. Como el elemento anterior (div) es de bloque, este párrafo debería estar debajo. Como hemos "flotado" al div, este párrafo ha "subido" hasta aquí.</p>
```

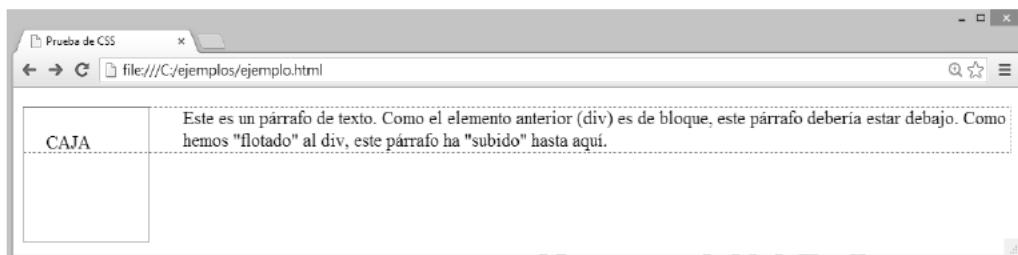


Figura 3.14: Efecto de "padding" y "margin"

El `padding` es el espacio que separa el contenido de un elemento de su borde (es decir que es un espacio *interno*). El `borde` del elemento es el que observamos con la propiedad `border` y pone de manifiesto la superficie que ocupa y la frontera hasta donde llega un elemento. En nuestro ejemplo, el `padding` generó que la palabra "CAJA" se aleje del borde del `div`. El `margin` es el espacio que separa los bordes de los elementos circundantes, por lo que lo consideramos un espacio *externo*. En el ejemplo, el `margin` aplicado sobre el lateral derecho (*right*) generó que el `p` se desplace hacia la derecha, alejándose del elemento que tiene a su izquierda, el `div`. Por lo tanto, el `margin` *mueve* a los elementos en sí, mientras que el `padding` *mueve* al contenido interno sin alterar la posición.

En el ejemplo, se está utilizando la **regla abreviada** `padding`, que modifica los espacios internos de los cuatro laterales simultáneamente (arriba, derecha, abajo e izquierda). Para el `margin` hemos optado por una regla más específica, que sólo altera el margen del lateral derecho (*right*). Ambas propiedades pueden utilizarse de las dos maneras, de acuerdo a la necesidad. En el siguiente gráfico se indica cada espacio según el modelo de caja:

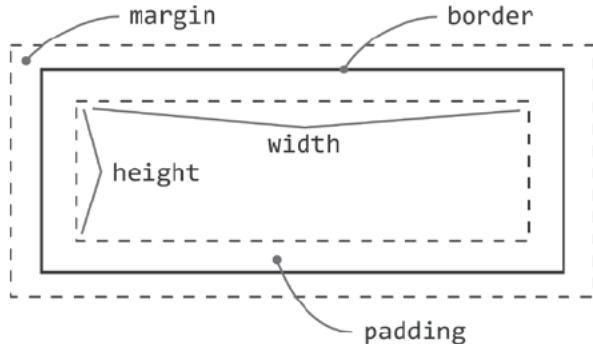


Figura 3.15: Elementos más importantes del modelo de caja

Es importante notar que el modelo de caja define al `width` y `height` como los tamaños para el *contenido*. Por lo tanto, el tamaño *final* exterior del elemento es el que declaramos con `width` y `height` sumado al `padding` que posea. De hecho, si colocamos un `border` muy grueso también estaremos influyendo en el tamaño exterior final.

Volvamos a plantearnos el desafío de ubicar elementos `div`, que son de bloque, uno al lado del otro. Hemos visto que reducirles el ancho no basta ya que el salto de línea los sigue disponiendo uno encima del otro. Apliquemos entonces el `float`:

```
div { width: 90px; height: 90px; border: 2px dotted blue; float: left; }

<div>A</div> <div>B</div>
```



Figura 3.16: Aplicación de regla "float" a dos `<div>`

¡Bien! Ahora, según el selector que hemos utilizado, *todos* los `div` permiten que, a su derecha, se coloquen los elementos subsiguientes. El elemento A se ubicó a la izquierda (`left`) y permitió que el elemento B "suba" quedando, por lo tanto, a su derecha. El elemento B también está flotado, por lo que si hubiera más elementos debajo ocurriría el mismo comportamiento: "subiendo" a la derecha de B. Le recomendamos al lector hacer la prueba y testear diferentes situaciones.

Para consolidar la comprensión de la propiedad `float`, observemos que pasa si utilizamos `right` en lugar de `left`:

```
div { width: 90px; height: 90px; border: 2px dotted blue; float: right; }

<div>A</div> <div>B</div>
```



Figura 3.17: Aplicación de "float" hacia la derecha

Nótese que los `div` no fueron cambiados de orden en el documento. Cuando aplicamos un flotado *hacia derecha*, los elementos se ubican en ese lateral y la mecánica es idéntica a la desarrollada. El `div` A está ubicándose en el margen derecho y el elemento B está "subiendo" a su izquierda. De manera idéntica, si hubiera más marcas luego de B, estos elementos intentarían ubicarse a su izquierda.

Hasta aquí hemos realizado las pruebas directamente dentro de la etiqueta `body` sin más marcas que la estructura base del documento y nuestras propias etiquetas a experimentar. A continuación exploraremos cómo se puede construir, con las reglas que describimos hasta aquí, una página web de **ancho fijo**. Los sitios diseñados con ancho fijo son la mayoría e implican que todo el contenido está ubicado dentro de una gran caja que, generalmente, se centra horizontalmente en el navegador.

Veamos entonces las marcas por las que se suele comenzar al crear una página:

```
#main { width:900px; margin: auto; border: 1px solid red; }
#col1 { width: 250px; float: left; border: 1px dashed blue; }
#col2 { margin-left: 10px; min-height: 500px; float:left; }
footer { clear: both; }

<div id="main">
    <div id="col1">Esta es la columna izquierda</div>
    <div id="col2">Esta es la columna para el contenido principal</div>
    <footer>Pie de página</footer>
</div>
```

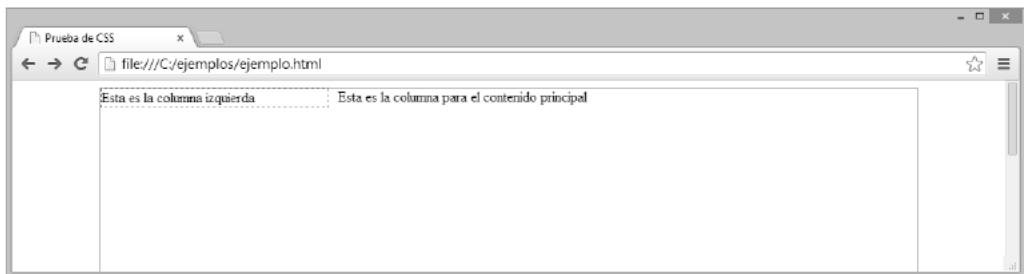


Figura 3.18: Maqueta básica para sitio web a dos columnas

Observamos que la caja principal (`main`) se utiliza para determinar el ancho del diseño y envolver a toda la página para poder centrarla. El centrado horizontal de un elemento *de bloque* se logra con la regla `margin: auto`. Al encontrar esta declaración, el navegador

iguala los márgenes a izquierda y derecha, lo que tiene como consecuencia que el elemento se centra en el espacio horizontal de su padre. Para que esto tenga sentido, es necesario que el elemento a centrar tenga un ancho determinado ya que, caso contrario, los elementos de bloque ocupan siempre todo el ancho del padre como hemos visto. El padre de `#main` es `body`.

Los dos `div` interiores marcan las dos columnas. Es decir que estamos realizando la maqueta de un diseño de *ancho fijo* diagramado a *dos columnas*. Por medio del `float` se está colocando la columna `#col1` a la izquierda. Para el elemento `#col2` aplicamos la regla `min-height`, que indica al navegador un *alto mínimo* para el elemento. Esto permite que el elemento crezca, pero reservando este alto mínimo si el contenido fuera muy poco. Recordemos que la *altura* de los elementos de bloque viene dado por el contenido que tengan, salvo que esté indicado con `height`. Nosotros no podemos indicar el `height` porque no podemos saber la cantidad de texto que nuestro cliente ubicará en esta página y además queremos que automáticamente se ajuste al cambiar el contenido. Por este requisito de que la maqueta se adapte al contenido actual y futuro de manera dinámica, las *alturas* de los elementos generalmente no pueden determinarse.

Recomendamos siempre utilizar la propiedad `border` para ver la superficie que están ocupando los elementos. A veces puede sorprendernos la posición y el tamaño de algunas etiquetas ya que esta parte del estándar no es, por cierto, de aplicación intuitiva. La maquetación de la estructura subyacente, mayoritariamente dominada por el modelo de caja, es uno de los temas más difíciles para quien comienza a trabajar con CSS.

El elemento `footer` hace uso de la propiedad `clear`, que le indica al navegador que desde ese punto ya no considere los `float` anteriores. Si el pie de página no tuviera `clear`, se ubicaría a la derecha de la columna principal, puesto que `#col2` tiene un `float` a la izquierda, de manera que los elementos que le siguen intentar colocarse a su derecha. Para evitar que esto suceda se utiliza la regla `clear`.

Sin intención de ser exhaustivos hemos presentado ciertas secciones del estándar CSS, a los fines de demostrar cuál es la idea general de esta tecnología, y para aportar conocimiento específico a quienes trabajen en programación web. La maquetación se consolida, como muchas cosas, básicamente con experiencia y práctica en distintos proyectos. Se aprende haciendo, y por eso es importante contar con buenos recursos bibliográficos y un buen punto de partida. Eso es lo que hemos intentado construir. Si bien nuestro capítulo concluye aquí, el camino en realidad está comenzando.

3.10 PROPIEDADES MÁS UTILIZADAS

Propiedad	Significado sintético	Valores comunes	Heredada
<code>background</code>	Color o imagen de fondo	<code>múltiples</code>	no
<code>border</code>	Estilo del borde	<code>múltiples</code>	no
<code>color</code>	Color del texto	<code>un color</code>	sí
<code>cursor</code>	Puntero del mouse	<code>pointer</code>	sí
<code>font-family</code>	Nombre de tipografía	Arial, Tahoma...	sí

<code>font-size</code>	Tamaño del texto	<i>número y unidad</i>	sí
<code>font-style</code>	Variable tipográfica (estilo)	<code>italic</code>	sí
<code>font-weight</code>	Variable tipográfica (peso)	<code>bold</code>	sí
<code>line-height</code>	Espacio entre líneas de texto	<i>porcentaje</i>	sí
<code>list-style-type</code>	Tipo de viñeta en listas	<code>none, disc</code>	sí
<code>opacity</code>	Opacidad del contenido	0 (transparente) a 1 (opaco)	no
<code>text-align</code>	Alineación del texto	<code>left, right, center, justify</code>	sí
<code>text-decoration</code>	Decoración del texto	<code>none, underline</code>	no
<code>width</code>	Ancho del contenido	<i>número y unidad</i>	no
<code>height</code>	Alto del contenido	<i>número y unidad</i>	no
<code>margin</code>	Espacio exterior	<i>número y unidad</i>	no
<code>padding</code>	Espacio interior	<i>número y unidad</i>	no
<code>display</code>	Tipo de contenedor	<code>block, none</code>	no
<code>float</code>	Flotación	<code>left, right</code>	no
<code>clear</code>	Reseteo de la flotación	<code>left, right, both</code>	no

3.11 BIBLIOGRAFÍA

De la misma forma que recomendamos para HTML, creemos que la mejor opción es recurrir al W3C: www.w3.org/Style/CSS. Allí encontraremos el estándar propiamente dicho, además de material educativo y las discusiones sobre el progreso de los distintos módulos, nuevamente, en su mayor parte sólo en inglés. Es interesante ver que las tecnologías son cuerpos en evolución. A veces perdemos de vista ese aspecto.

Recomendamos también tener a mano un "machete". Hay muchos de ellos disponibles en Internet, confeccionados por autores independientes. Estos resúmenes de reglas serán muy útiles al principio, cuando no recordemos todas las propiedades de memoria. Pueden buscarse en Google con los términos *CSS cheatsheet*. Algunos están más actualizados que otros, pero para empezar casi cualquiera estará bien.

3.12 EJERCICIOS

1. Construya un documento HTML que haga referencia a dos hojas de estilo externas (con `<link>`). Utilizando la inspección de tráfico del navegador, verifique que se realicen las dos peticiones GET que descargan cada CSS.
2. ¿Qué son los *MIME types*? ¿Para qué y cuándo fueron definidos?
3. ¿Qué son las tecnologías LESS y SASS?
4. Investigue para qué sirve el atributo `media` de la etiqueta `style` y los valores posibles.

5. ¿Qué elementos seleccionan estos selectores?

- a#b {}
- strong.d {}
- p a {}
- p.a {}
- p .a {}
- #cuerpo #col_izq table {}
- strong.ofertas {}
- a strong, strong a {}
- p#resumen a em, p#resumen strong {}
- p, a, em, strong {}

6. Busque en el estándar dos pseudoclases y compruebe su funcionamiento.

7. Busque en el estándar cómo se definen los colores en valores RGB. Cree un documento con dos elementos cualquiera y especifique un `background` negro para uno y gris para el otro, utilizando RGB en lugar de los *colores con nombre*.

8. Utilizando la herramienta de inspección de elementos, encuentre en algún sitio web un elemento que tenga propiedades sobreescritas. Calcule la especificidad de los selectores involucrados mirando el código fuente del CSS. Compruebe si las reglas aplicadas en último orden son las que corresponden al grupo con un selector más específico.

9. En un documento, coloque un `div` de 300px de ancho y alto y colóquelo un borde. Dentro del `div` ubique mucho texto, de manera que no entre en el tamaño asignado. ¿Qué hace el navegador con el texto que cae "fuera" del `div`? Busque en el estándar el significado de la propiedad `overflow` y verifique su uso.

10. En un documento coloque dos `div` y dos párrafos. Dentro de los cuatro elementos coloque una línea de texto y dibuje sus bordes. Si bien los cuatro elementos quedan uno debajo del otro por ser elementos de bloque, los `div` quedan pegados mientras que los párrafos no. ¿Por qué?

11. Partiendo del ejemplo de maqueta de ancho fijo y dos columnas del final del capítulo, agregue los elementos necesarios para convertirla en una maqueta a tres columnas.

12. ¿De qué manera se puede agregar un `header` al ejercicio anterior, que ocupe las tres columnas al igual que el pie de página?

4. JavaScript

JavaScript es un **lenguaje de programación** utilizado para crear programas de **lado cliente** que se incorporó en navegadores comerciales en el año 1995. Su objetivo inicial era posibilitar pequeñas animaciones y comportamiento dinámico en las páginas web. Hoy en día es uno de los lenguajes más extendidos, ya que prácticamente cualquier navegador del mundo puede ejecutar un programa escrito en JavaScript, lo que lo hace sumamente **portable** por definición. En el año 1997 fue convertido en un estándar, mantenido por una organización llamada ECMA.

Aunque su nombre lo sugiera, JavaScript no está relacionado de ninguna manera con el lenguaje Java. Ambos fueron introducidos al mundo web en la misma época y, por alguna razón, quienes realizaron la campaña de lanzamiento de JavaScript pensaron que sería una buena idea copiar el nombre de un producto como Java, que estaba causando un gran impacto en el mundo informático y en la prensa.

JavaScript es un lenguaje **interpretado** y orientado a **objetos**. Las empresas fabricantes de los navegadores más conocidos tienen sus propias implementaciones del **intérprete** de JavaScript. Algunos de estos intérpretes realizan traducción a *bytecode*, o **código intermedio**, que es ejecutado por una **máquina virtual**, de la misma manera que Java, C# o PHP. Otros intérpretes realizan **compilación JIT (just in time)**, es decir que convierten el código JavaScript a código de máquina ejecutado por el microprocesador.

Las primeras versiones de JavaScript fueron utilizadas por los pioneros de la web en la década de los noventa. Por circunstancias históricas, los profesionales que más participaron en estas primeras épocas provenían del área del diseño. Esto influyó, de alguna manera, en la definición inicial del lenguaje. Originalmente los autores se propusieron que la curva de aprendizaje creciera rápidamente, es decir, que la barrera de entrada al lenguaje no sea tan elevada. Esto tiene como consecuencia que, hoy en día, muchas características del lenguaje tengan que ser revisadas porque impactan negativamente en la escalabilidad de las aplicaciones creadas con él. Por esto, JavaScript es cuestionado por muchas empresas que lo ven como un obstáculo para el desarrollo del lado cliente.

La utilización de navegadores es cada vez mayor, ya que prácticamente tenemos un navegador por cada dispositivo que pueda comunicarse con Internet. Esto quiere decir que tenemos navegadores prácticamente en todos lados, e intérpretes JavaScript en todos ellos. Las interfaces de las aplicaciones y sitios web son cada vez más complejas, en cuanto a funcionalidad y en cuanto a escala, por lo que el diseño del lenguaje utilizado para soportar todos estos desarrollos se ve fuertemente desafiado. Empresas como Google proponen reemplazarlo con lenguajes que han desarrollado ellos mismos, pero creamos que esto es muy difícil en la práctica. La siguiente versión del estándar a ser aprobada incluye cambios de fondo en el lenguaje, que serán beneficiosos para su futuro y su permanencia en el mercado.

4.1 FORMAS DE INCLUSIÓN EN HTML

Así como CSS proveía de definiciones gráficas a los documentos HTML, JavaScript permite definir **comportamientos** en estos documentos. Es decir que tanto CSS como JavaScript son tecnologías relacionadas a HTML que deben, de algún modo, ser referenciadas en el cuerpo de los documentos HTML a los cuales afectan. Para el caso de CSS hemos visto que disponemos de las etiquetas `<link>` y `<style>`, que nos permiten incorporar el código CSS desde un archivo separado o bien embeberlo en la misma página, respectivamente. Lo mismo sucede con JavaScript. Veamos un ejemplo:

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="programa1.js"></script>

        <script type="text/javascript">
            //aquí colocaremos el código fuente JavaScript
        </script>
    <!-- el documento continúa ... -->
```

Para la inclusión de JavaScript se utiliza una sola etiqueta: `<script>`. Esta misma marca nos permite incorporar archivos externos, mediante el atributo `src` (*source*), o escribir el código fuente directamente en la página web, es decir, embebido (*embedded*). Si utilizamos un archivo separado, debemos crear un documento con extensión `.js` y dentro del mismo escribir nuestro código fuente, por supuesto, con la **sintaxis** correcta del lenguaje JavaScript. Al código fuente de los lenguajes interpretados se lo suele denominar **script** (guion) en lugar de *programa*, aunque ambos términos hacen referencia a lo mismo: al texto que escribe el programador al realizar su trabajo. La diferencia es que el *script* no se compila sino que se entrega al intérprete en el mismo formato en que se produce: texto plano.

La etiqueta `script` tiene apertura y cierre tanto si utilizamos un archivo externo como si embebemos el código en la página. En el caso de archivos externos, dentro de la etiqueta no se permite colocar ningún texto, es decir que se abre y se cierra directamente. Ambos usos de la marca `script` involucran un atributo `type`, que indica cuál es el lenguaje que estamos utilizando para programar del lado cliente. Hace unos años existía otro lenguaje conjuntamente con JavaScript, llamado VBScript, que ha caído en desuso y ya no es soportado. De la misma forma, es probable que en los próximos años se incorpore algún lenguaje más, por lo que este atributo puede comenzar a tener sentido nuevamente.

Al igual que con CSS, ambas formas de inclusión pueden utilizarse en conjunto y, de hecho, esto es lo que sucede habitualmente. Típicamente incluiremos algunos archivos JavaScript externos con funciones de biblioteca (librerías de funciones) y agregaremos código escrito por nosotros de manera embebida. El navegador descargará los archivos externos mediante peticiones GET y los ejecutará, además de ejecutar nuestras porciones de código. Puede preverse que controlar este orden de ejecución es importante.

4.2 CLIENTE / SERVIDOR

Utilizaremos JavaScript para crear programas que serán ejecutados en el **lado cliente**. Con otras palabras, el intérprete JavaScript es el navegador. Es importante tener en cuenta que, desde el lado servidor, el código JavaScript no es más que texto plano que es enviado hacia el cliente, como todo contenido. En este sentido, HTML, CSS, JavaScript, imágenes, PDF y cualquier otro contenido que forma parte de los sitios web es simplemente un conjunto de *bytes* desde el punto de vista del servidor. JavaScript viajará entonces hacia los clientes como cualquier otro archivo y no merece de parte del servidor ningún tratamiento especial, aunque nosotros lo identifiquemos visualmente como *código ejecutable*.

Esta consideración es importante porque en un capítulo posterior exploraremos otro lenguaje cuyo intérprete reside, contrariamente, en el lado servidor, y la comprensión de estos dos *lados* es fundamental para entender cómo funcionan las aplicaciones reales cuando la complejidad comienza a crecer. Una aplicación web está compuesta por el software ejecutado en *ambos lados* y que, en conjunto, hacen que la aplicación funcione.

En el lado cliente la complejidad de los programas suele darse por el gran nivel de **asincronismo**. Esto es así, en gran medida, porque la programación de lado cliente web es un caso de programación **orientada a eventos**. El usuario realizará ciertas acciones y en base a ello el programa reaccionará. Esto ocurre en todas las interfaces gráficas de usuario (GUI, *graphical user interface*), de las cuales la web es un caso. El lenguaje JavaScript no es difícil en sí, sino que es utilizado para resolver problemas que pueden complejizarse bastante.

En los últimos años han surgido tecnologías que permiten programar con JavaScript del lado servidor, siendo la más conocida el proyecto *node.js*. Sin embargo, creemos que esta tecnología es más bien una prueba de concepto y habrá que esperar un tiempo para evaluar qué impacto real tendrá en el futuro del desarrollo web.

4.3 SINTAXIS BÁSICA

Crearemos un documento HTML e incluiremos código para experimentar la sintaxis y el ambiente de trabajo con JavaScript. Utilizaremos la forma de inclusión embebida porque nos facilita las pruebas sin tener que modificar varios archivos a la vez.

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript">
        alert("Hola mundo");
    </script>
</head>
<body>
    <h1>Prueba de Javascript</h1>
</body>
</html>
```

Este documento HTML posee un bloque `script` en el `head`. Cuando abrimos el documento en un navegador observamos que se ejecuta el programa, que consiste en una sola llamada a función con un parámetro:

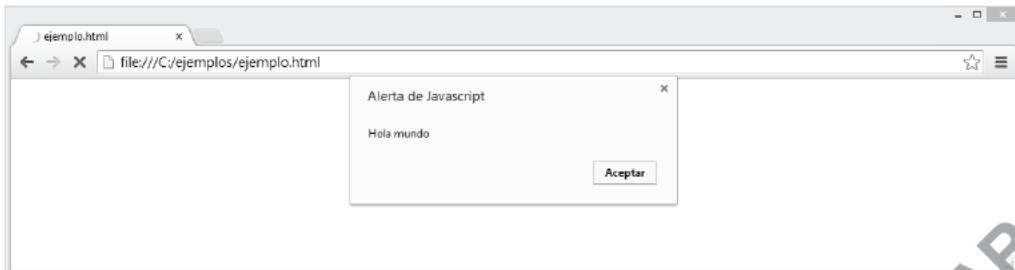


Figura 4.1: <script> ubicado en el <head>

Primero, es importante notar que el programa se ejecuta inmediatamente cuando el navegador encuentra la etiqueta `script`. Por utilizar la etiqueta `script` en el `head` podemos ver que todavía no se ha renderizado el `h1`. Con otras palabras, el árbol del documento todavía no ha sido completado cuando el programa es ejecutado.

En segundo lugar, la función `alert` de JavaScript abre una ventana de diálogo con el texto que le pasamos como parámetro. La ventana es de tipo **modal**, es decir que la ejecución no continúa hasta que no aceptamos el mensaje. Si bien esta función `alert` nos permite rápidamente inspeccionar variables al realizar pruebas, no es habitual utilizarla en aplicaciones reales.

Cambiamos el bloque `script` de lugar y veamos qué sucede con el orden de ejecución:

```
<!DOCTYPE html>
<html>
<head></head>
<body>
    <h1>Prueba de Javascript</h1>
    <script type="text/javascript">
        alert("Hola mundo");
    </script>
</body>
</html>
```

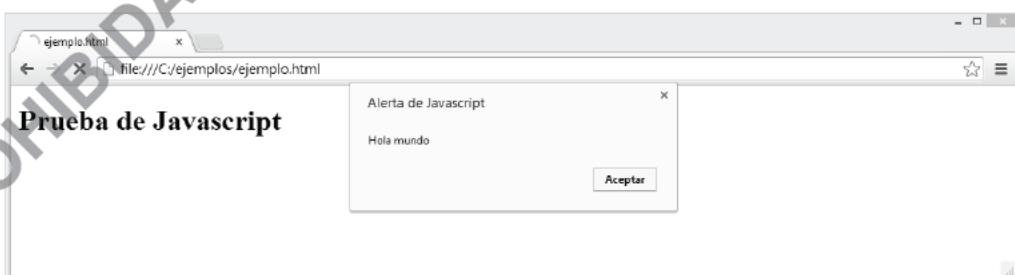


Figura 4.2: <script> colocado en el <body> a continuación del título

Si realizamos la experiencia de esta manera corroboramos que el navegador ejecutará el programa apenas se lo encuentre en el código fuente de la página. El `alert` entra en acción cuando el `h1` ya fue renderizado.

Con este programa de una sola línea podemos notar que la sintaxis de JavaScript es muy similar a la de C. Esto puede concluirse a partir de la utilización del punto y coma (`;`) para delimitar las sentencias, los paréntesis para indicar los llamados a función y las comillas para delimitar las *string*.

Es bueno notar que la sintaxis del lenguaje Java, al igual que JavaScript, también es descendiente de C, por lo que esto puede sumar más confusión a la relación entre estos dos lenguajes. Aunque tengan nombres parecidos y ambos hereden una sintaxis de un lenguaje en común, presentan características de fondo muy distintas. Avancemos sobre más herramientas y aspectos de JavaScript.

4.4 UTILIZACIÓN DE LA CONSOLA

Hemos dejado claro que el intérprete de JavaScript reside en nuestro navegador. Trabajar con un lenguaje interpretado nos permite hacer **debugging** de manera mucho más directa que al utilizar lenguajes compilados, ya que el intérprete nos indica directamente los errores que encuentra al intentar ejecutar el programa. Entre otras cosas, también podemos darle un texto al intérprete y pedirle que lo ejecute y nos muestre los resultados. Utilizaremos para ello una herramienta que se denomina normalmente **consola de JavaScript**, a la que accedemos presionando `CTRL+MAYUS+J` ó bien `F12`.

Creemos un documento con un programa erróneo y veamos qué sucede en la consola:

```
<!DOCTYPE html>
<html>
<head></head>
<body>
    <h1>Prueba de Javascript</h1>

    <script type="text/javascript">
        funcion_inexistente();
    </script>
</body>
</html>
```



Figura 4.3: Consola de JavaScript mostrando un error en el programa

En la consola encontramos un mensaje que señala, correctamente, que la función que estamos intentando llamar no ha sido definida. Se puede percibir que el uso de la consola es fundamental al crear nuestros programas. El intérprete de JavaScript *detiene* toda la ejecución del programa cuando encuentra un error. Recordemos que esto es distinto a CSS, en donde el intérprete simplemente *omite* las líneas con errores de sintaxis.

En la consola también se puede escribir código fuente. El intérprete lo analizará y lo ejecutará. Por ejemplo:

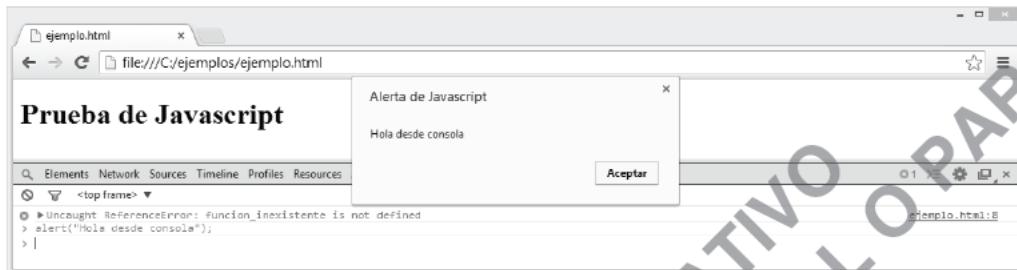


Figura 4.4: Ejecución de código Javascript directamente desde la consola

Al presionar **Enter** veremos el diálogo modal con el mensaje.

Podemos pedirle al intérprete que ejecute cualquier código JavaScript válido. Veamos un ejemplo más para dejar claro su utilidad. Ingresemos una expresión:



Figura 4.5: Evaluación de una expresión mediante la consola

4.5 VARIABLES Y TIPOS

JavaScript es un lenguaje **dinámicamente tipado**. Esto quiere decir que no podemos forzar, al momento de escribir el programa, cuál será el **tipo** de una variable. Lenguajes como C ó Java son **estáticamente tipados**, o sea que el tipo de la variable se determina al compilar y no se puede modificar. Los lenguajes con **tipado** dinámico definen el tipo de una variable a partir de los valores que se le van asignando. Veamos un ejemplo:

```
var i = 10;
i = true;
i = "hola";
alert(i);
```

Este programa define una **variable** `i` por medio de la **palabra reservada** `var`, que es inicializada con un número entero. En la línea siguiente se le asigna un valor booleano y luego una string. Finalmente se llama a la función `alert` que ya experimentamos. En pantalla se verá el mensaje modal con el texto que contiene la variable ("hola"). A lo largo del programa, el tipo de `i` se fue modificando a medida que le asignamos **valores literales** de distintos tipos. Estas líneas se pueden ejecutar en la consola, como ya hemos visto.

Este concepto es claramente distinto a lenguajes como C, en donde el programador declara las variables indicando su *nombre* y su *tipo*. Nótese que hemos declarado una variable con el **operador** `var`, que se utiliza para todas las variables, y que no se observa ninguna indicación acerca del tipo. Esto quiere decir que los lenguajes de **tipado dinámico** no nos permiten forzar los tipos de las variables. Con otras palabras, los tipos de las variables pueden ir cambiando en tiempo de ejecución, y sólo dependen de los valores que se les asignen.

Sin perjuicio de esto, JavaScript es un lenguaje que sí *tiene* tipos. Exploraremos cuáles son los tipos de JavaScript utilizando `alert` y el operador `typeof`:

```
var i;  
i = 10; alert(typeof i); //number  
  
i = 3.5; alert(typeof i); //number  
  
i = false; alert(typeof i); //boolean  
  
i = "cadena"; alert(typeof i); //string  
  
i = new String; alert(typeof i); //object
```

El operador `typeof` devuelve una string con el *nombre del tipo* del **operando**. Si utilizamos la consola para ejecutar esta prueba, veremos que los dos primeros tipos son `number`, el tercero es `boolean`, el siguiente es `string` y el último es `object`. Aquí notamos una característica distintiva de JavaScript y que trae muchos dolores de cabeza: los **números** se representan con un solo tipo `number`, ya sean enteros o flotantes.

En los lenguajes de tipado estático, la información del tipo define cuánta memoria se reserva para almacenar cada variable. Además, los tipos se utilizan para determinar el significado de algunos operadores, por ejemplo, los aritméticos.

Algunos lenguajes permiten el uso de variables de distinto tipo como operandos en expresiones. Esto se denomina **tipado débil**. Podemos mencionar al lenguaje C, en donde es posible, por ejemplo, sumar un entero con un flotante o asignar un flotante en una variable `char`. En C existen reglas que determinan qué sucede si se operan variables de distinto tipo. En cambio, otros lenguajes son de **tipado fuerte**, y la operación entre variables de distinto tipo no está permitida aunque los tipos sean relativamente parecidos. Podemos nombrar al lenguaje Ruby como un ejemplo de lenguaje fuertemente (y dinámicamente) tipado.

JavaScript es un lenguaje de tipado débil. Exploraremos entonces qué sucede cuando intentamos operar con distintos tipos:

```
var i, j;
i = 10;      j = "5";      alert(i*j); //50 number
i = "10";    j = 5;       alert(i-j); //5 number
i = 10;      j = 5;       alert(i+j); //15 number
i = "10";    j = "5";     alert(i+j); //105 string
i = "10";    j = 5;       alert(i+j); //105 string
```

Los operadores aritméticos tratarán de realizar la operación, convirtiendo los operandos en números. Es muy importante notar que el operador `+` está **sobrecargado**. Actúa como **suma aritmética** si enviamos números o como operador de **concatenación** si *al menos uno* de los operandos es una string. Olvidar este comportamiento puede acarrearnos muchos problemas difíciles de diagnosticar. En el caso de tener una variable tipo string y querer sumar aritméticamente con otra, debemos *forzar* el tipo a `number` para que no ocurra la concatenación:

```
var i, j, k;
i = 10;
j = "5";
k = parseInt(j);

alert(i+j); //105 string
alert(i+k); //15 number
```

La función `parseInt` intenta convertir lo que le envíamos en un entero. Nos retornará un `number` con el resultado de su conversión. También disponemos de `parseFloat`.

Observemos qué sucede si la operación de "conversión" a número no se puede realizar:

```
var i, j;

i = 10; j = "a";
alert(i*j); //la string "a" no se puede convertir a número

i = "b"; j = 5;
alert(i-j); //idem

i = parseInt("fff"); //no es posible
```

JavaScript incorpora un valor especial llamado **NaN (not a number)**. Este valor se retorna cuando las operaciones que esperan números fallan al no poder **evaluar** los argumentos como números. Ejecutemos estas dos líneas utilizando la consola del navegador:

```
var i = "zzz"; alert("i no es un numero: " + isNaN(i) ); //true
var j = "24"; alert("j no es un numero: " + isNaN(j) ); //false
```

La función `isNaN` nos permite conocer si una variable *puede* utilizarse como número. Es importante notar que esto va más allá del tipo, ya que estamos enviando strings.

La naturaleza del tipado dinámico y débil de JavaScript hace que la codificación de los programas se lleve adelante muy rápidamente. Sin embargo, cuando la escala comienza a crecer, la ventaja que representa no estar obligado a declarar las variables ni sus tipos se va convirtiendo en una desventaja. A medida que tenemos más y más líneas, con más y más variables, las operaciones entre datos de distinto tipo y la alteración de los tipos

a medida que avanza la ejecución puede crear un caos de variables que lleven a resultados inesperados. Las últimas versiones incorporan modos más estrictos que imponen algunas restricciones. Avancemos con otras características del lenguaje antes de mencionarlo más detalladamente.

4.6 FUNCIONES

JavaScript, al igual que C, incorpora el concepto de **función**: un subprograma que puede ser "llamado" desde otra parte del programa, que posee parámetros y retorna un valor. Las funciones realizan pequeñas operaciones y permiten dividir un programa en partes, más fáciles de mantener. A diferencia de C, las funciones en JavaScript son **objetos**. Para ser precisos, *todas* las variables de JavaScript son en realidad objetos. Las funciones también son objetos y pueden almacenarse en variables, pasarse como argumento y retornarse como cualquier otro valor. Al principio, esto puede ser desconcertante. Comencemos declarando una función:

```
function prueba() { //declaración
    alert("función prueba");
}

prueba(); //llamada
```

Aquí tenemos la declaración de una función, de manera muy similar a C pero con las siguientes diferencias:

- la palabra reservada `function` indica que se está declarando el cuerpo de una función
- no se indica el tipo de retorno
- no se indica el tipo de los argumentos

La llamada a la función se materializa en la ejecución del `alert`, por lo que estamos ante una declaración y una llamada, que además se ven muy parecidas al lenguaje C.

Veamos cómo podemos ilustrar que las funciones son, en realidad, objetos:

```
function prueba() {
    alert("función prueba");
}

var b = prueba; //prueba sin paréntesis!
b(); //se hace el alert
```

Puede verse que esto es *absolutamente* distinto a C. Intentemos explicarlo detalladamente. Cuando se declara una función en JavaScript, el intérprete crea un objeto para almacenar el *cuerpo* de la función (o sea, el código a ejecutar) y lo relaciona con un *nombre*. Ese nombre es una variable que almacena una **referencia** al objeto de tipo función. Al asignar de la forma `b=prueba` estamos creando una variable `b` en donde se está *copiando* la referencia a la función (objeto) que declaramos. De la misma manera que asignamos números o strings, también podemos asignar funciones (objetos) de esta manera. Es muy importante notar que en la asignación, el nombre de la función no va seguido

de paréntesis, puesto que esa expresión indicaría que se debe *llamar* a la función y aquí no se está invocando sino que se está *copiando* una referencia. Cuando utilicemos la variable **b** para hacer una llamada a función, es decir, colocando paréntesis, veremos el **alert**. Lo que sucede en estas pocas líneas es que tenemos dos referencias al mismo objeto y ese objeto es de tipo función. Con otras palabras, tenemos dos nombres para lo mismo.

Veamos otra sintaxis para declarar funciones, en donde queda totalmente claro que en JavaScript las funciones se pueden "almacenar" en variables como cualquier otro valor:

```
var prueba2 = function() { alert("prueba2"); } //función anónima  
var prueba3 = prueba2;  
  
prueba2();  
prueba3();
```

Lo que estamos viendo en la primera línea es lo que se llama **función anónima**. Las funciones anónimas son construcciones que permiten ciertos lenguajes y que, básicamente, representan una *porción de código* que puede ser ejecutado. Típicamente, las funciones anónimas se utilizan para enviar código a otras funciones, o para que alguna función *retorne* código. Si "almacenamos" una función anónima en una variable, podemos utilizar el nombre de la variable para llamar a la función. Decimos "almacenar" entre comillas porque, como ya dijimos, en realidad la variable contiene una referencia a la función y no la función en sí.

Por su propio concepto, las funciones anónimas no pueden ser llamadas directamente, ya que no tienen un *nombre* que podamos utilizar para construir una llamada. Pero podremos llamar a la función si tenemos una referencia a ella en alguna variable, como mostramos en el ejemplo anterior (**prueba2**). Si la pasamos como argumento, la función se almacenará en uno de los parámetros de la función que la recibe. Veamos este caso para ilustrar más claramente:

```
function prueba4( param1, param2) { //función que recibe 2 parámetros  
    alert("param1 vale " + param1);  
    param2(); //lo que haya en param2 tiene que ser "llamable"  
}  
  
var prueba5 = function() { alert("soy la función anónima"); }  
  
prueba4(222, prueba5);
```

Si ejecutamos este código veremos que se realiza el **alert** de la función **prueba4** y luego el **alert** de la función anónima. Lo que está sucediendo es que declaramos una **función con nombre** **prueba4** utilizando la construcción tradicional. Luego declaramos una función anónima que *referenciamos* con una variable **prueba5**. Finalmente estamos llamando a la función **prueba4** y le enviamos un entero y *una función* como parámetros.

También se puede prescindir de la variable **prueba5** y escribir el mismo programa así:

```

function prueba4( param1, param2) { //función que recibe 2 parámetros
    alert("param1 vale " + param1);
    param2(); //lo que haya en param2 tiene que ser "llamable"
}
prueba4(222, function() { alert("soy la función anónima") } );

```

El hecho de que las funciones puedan pasarse y retornarse como cualquier otro valor permite trabajar con **programación funcional**. Si bien este paradigma tiene un interés en sí mismo, para el desarrollo web este tipo de herramientas es importante porque permite simplificar muchísimo el trabajo con eventos que, como mencionamos, es una gran parte del esfuerzo del lado cliente. En este momento nuestro objetivo es mostrar cómo funciona el lenguaje JavaScript y compararlo con algún otro lenguaje imperativo conocido, como C, para encontrar sus parecidos y diferencias. Dejamos sentadas las bases sintácticas para abordar nuevamente este complejo tema más adelante.

Mencionamos antes que JavaScript tiene algunas características que, en sus orígenes, los autores consideraron que facilitaría su uso por personas que no habían sido formalmente educadas en informática. Veamos dos de estas características y por qué es una mala idea confiar en ellas.

En primer lugar debemos decir que, en rigor, los puntos y coma (;) que separan las sentencias son *opcionales*. Esto puede resultar sorprendente, pero para quienes no tienen experiencia en programación o para quienes comienzan a estudiar, determinar cuándo colocar y cuándo no este carácter de fin de sentencia es bastante confuso. Por esta razón, en el diseño de JavaScript se incluyó un paso previo automático antes de la ejecución que hace que el intérprete coloque los puntos y coma por nosotros. Por supuesto, responde a una serie de reglas que determinan cuándo estamos ante un final de sentencia y cuándo no. Creemos que, en lugar de recordar estas reglas de memoria, lo mejor es indicar explícitamente los cambios de sentencia como en todos los otros lenguajes.

La otra característica que se diseñó como un "facilitador" es que la declaración de variables también es opcional. Sorprendente, ¿verdad? Este pequeño programa es totalmente válido, y podemos probarlo con la consola del navegador:

```

i = 2;
j = 1;
alert(i+j); //3

```

El resultado es aparentemente obvio, y el programa se nos ocurre también muy inocente. De alguna manera, el intérprete está declarando las variables cuando las ve definidas por primera vez. Recordemos que, antes, hemos declarado variables con el operador **var**. Este es el primer programa que realizamos sin utilizar **var**, sencillamente definiendo las variables. El resultado es aparentemente el mismo, por lo que tranquilamente podemos preguntarnos qué sentido tiene declarar explícitamente las variables antes de usarlas. La respuesta está en este ejemplo:

```

function Func1() {
    i=0;
}

```

```
function Func2() {  
    i++;  
}  
  
function Func3() {  
    alert(i);  
}  
  
Func1();  
Func2(); Func2();  
Func3(); //vemos un 2
```

Cuando utilicemos una variable, y el intérprete determine que es la primera aparición de la misma, procederá a declararla como **variable global**. Esto tiene efectos muy indeseables, ya que distintas funciones que utilicen una variable con el mismo nombre, aún creyendo que se trata de variables **locales** auxiliares, en realidad se estarán refiriendo a una global, lo que creará problemas muy difíciles de corregir.

Si utilizamos el operador **var**, el resultado es totalmente diferente:

```
function Func1() {  
    var i=0;  
}  
  
function Func2() {  
    var i;  
    i++;  
}  
  
function Func3() {  
    var i;  
    alert(i);  
}  
  
Func1();  
Func2(); Func2();  
Func3(); //vemos "undefined"
```

Ahora cada función tiene una **variable local** llamada **i**, que obviamente resulta independiente entre las distintas funciones. En cuanto utilicemos una variable por primera vez, y no hagamos una declaración explícita con **var**, el intérprete creará por nosotros la variable pero siempre con ámbito global, algo que es una muy mala idea.

Este comportamiento del intérprete resulta ser una idea tan mala, a medida que los programas van creciendo en escala, que las nuevas versiones del estándar incluyen una manera de exigir la declaración explícita de las variables, convirtiendo un posible descuido en un error de interpretación. Para utilizar este **modo estricto**, se debe incluir una string "**use strict**" al comienzo de la función o el bloque **<script>**, de esta manera:

```
<script type="text/javascript">  
    "use strict"  
  
    //... aquí colocamos el programa  
</script>
```

Esta string es omitida por intérpretes antiguos, ya que se trata sencillamente de una expresión literal cuyo valor es descartado. Si tenemos un documento con varios bloques `script`, cada uno debe incluir esta string. La misma string debe incluirse como primera línea si utilizamos archivos externos. Recomendamos la utilización del modo estricto a los efectos de evitar problemas. En alguna versión futura éste será el modo por defecto.

4.7 ACCESO AL DOM

Cuando exploramos la tecnología HTML hemos visto que el navegador compone un árbol del documento al leer cada página web. Este árbol puede ser accedido desde JavaScript para operar sobre él. El navegador nos provee acceso a este árbol a través de objetos y una serie de funciones. Se suelen llamar **funciones de DOM** a estas operaciones. **DOM** (*document object model*) es, justamente, una forma de representar el documento con objetos, es decir *un modelo* de acceso al documento.

Hasta aquí exploramos características básicas de JavaScript y su sintaxis. Veamos ahora cómo interactuar con los elementos que, colocados en el documento a través de etiquetas, acompañan a estos programas JavaScript.

```
<!DOCTYPE html>
<html>
<head> <title>Prueba de DOM</title> </head>
<body>
    Su nombre: <input type="text" id="nombre" />

    <script type="text/javascript">
        document.getElementById("nombre").value="Pepe";
    </script>
</body>
</html>
```

Tenemos aquí un ejemplo muy sencillo para ilustrar el acceso al DOM. Mediante el objeto `document`, que representa a todo el documento HTML podemos acceder a las funciones para manipular el DOM. La función `getElementById` nos devuelve una **referencia** a un elemento del árbol dado su `id`. Una vez que tenemos la referencia podemos acceder a sus métodos y propiedades. Este programa escribe el `value` del `input` por programa. Podemos imaginar la referencia como un *teléfono*, una vía de comunicación con el elemento del documento. Pudimos escribir su propiedad `value` porque se trata de una referencia a un elemento `input`, que posee tal propiedad.

Supongamos ahora que el usuario ingresará su nombre en el `input` y queremos utilizarlo en nuestro programa. Evidentemente necesitamos *esperar* a que el usuario escriba en el campo para poder leer su contenido. Lo que necesitamos es un **evento**.

```
<!DOCTYPE html>
<html>
<head> <title>Prueba de DOM</title> </head>
<body>
    Su nombre: <input type="text" id="nombre" />
    <input type="button" id="ejecutar" value="Ejecutar" />
```

```

<script type="text/javascript">
    "use strict"

    function mostrar() {
        var campo = document.getElementById("nombre"); //(2)
        var nombre = campo.value; //(3)
        alert("Su nombre es " + nombre); //(4)
    }

    document.getElementById("ejecutar").onclick = mostrar; //(1)
</script>
</body>
</html>

```

Hemos incorporado varias cosas en el ejemplo. Se agregó un botón (`input` de tipo `button`) al solo efecto de escuchar su evento `click`. Esto se realiza (1) tomando una referencia, con `getElementById` y enviando a la propiedad `onclick` la función que debe ejecutarse cuando eso suceda. Es importante notar que la función `mostrar` está siendo *escrita* en el miembro `onclick`, tal como examinamos con las funciones en el apartado anterior. En sentido estricto, se está copiando una referencia a la función. Esto indica al interprete que, cuando suceda el click, y sólo si este sucede alguna vez, el código a ejecutar es el de la función. La función `mostrar` (2) toma una referencia al `input` donde el usuario indicó su nombre, (3) con esa referencia lee el `value` y (4) realiza un simple `alert` concatenando el nombre con un mensaje fijo.

Anteriormente mencionamos que las funciones anónimas son una gran herramienta para el tratamiento de eventos. Algunas librerías como jQuery, que exploraremos en el capítulo siguiente, están diseñadas para aprovechar al máximo las ventajas de las funciones anónimas. En el siguiente ejemplo, aunque quizá más difícil de leer para un principiante, se demuestra cómo el tratamiento de las funciones como objetos de primera clase redonda en código más compacto al momento de trabajar con eventos:

```

document.getElementById("ejecutar").onclick = function() {
    alert("Su nombre es " + document.getElementById("nombre").value);
}

```

Avancemos un poco más e incorporemos al ejemplo la funcionalidad de mostrar el nombre en el mismo documento, en lugar de realizar un `alert` (mostraremos sólo el `body`):

```

<body>
    Su nombre: <input type="text" id="nombre" />
    <input type="button" id="ejecutar" value="Ejecutar" />

    <div id="resultado">Ingrrese su nombre y presione Ejecutar</div>

    <script type="text/javascript">
        "use strict"
        function mostrar() {
            var campo = document.getElementById("nombre");
            var nombre = campo.value;

            var caja = document.getElementById("resultado");
            caja.innerHTML = "Su nombre es " + nombre;
        }

        document.getElementById("ejecutar").onclick = mostrar;
    </script>

```

```
</script>
</body>
```

Hemos incluido un `div` simplemente para escribir en su interior el mensaje resultante. En la función, en lugar de realizar un `alert` hemos tomado una referencia a este `div` y utilizado su propiedad `innerHTML`, que nos permite leer y escribir el *contenido* del elemento referido, o sea, lo que se encuentre *entre* su etiquetas de apertura y cierre.

En estos últimos ejemplos queda claro además que JavaScript es un lenguaje **orientado a objetos**. En este lenguaje el **operador miembro** es el **punto** `(.)`, de la misma manera que en Java ó C#. Hemos operado con funciones miembro (`getElementById`) y con variables miembro (`onClick`, `value`, `innerHTML`). A diferencia de la mayoría de los lenguajes orientados a objetos, sin embargo, JavaScript no posee el concepto de **clase** al cual estamos tan acostumbrados. Los objetos se construyen en realidad *duplicando* otros objetos existentes. Esta técnica es conocida como **prototipado**. El hecho de que JavaScript no tenga la noción de clase genera mucha controversia entre la comunidad web. La próxima versión del lenguaje probablemente incorpore esta característica, ya que su falta lo hace complejo de aprender, complejo de utilizar y es una de las razones de peso por las cuales algunas compañías pretenden reemplazarlo por otros lenguajes.

Exploraremos ahora más funciones de DOM. Supongamos que necesitamos construir una página web para un salón de eventos, en donde la persona que contrata el evento precisa enviar una lista con los nombres de los invitados. Armaremos un documento HTML con un input en donde escribir y un botón para agregar un nombre a la lista.

```
<input type="text" id="nombre" />
<input type="button" id="agregar" value="Aregar" />
<div id="caja"></div>
```

Hemos creado un contenedor con un `div`, para colocar dentro de él los nombres de las personas que invitamos. El problema radica en que no sabemos cuántas personas se van a invitar y que probablemente también debamos permitir que se eliminan invitados una vez añadidos a la lista. Es decir que la cantidad de invitados posibles es *variable* en lugar de ser un número *constante*. Esto nos obliga a que las etiquetas que contienen a los nombres deban construirse por programa, de manera dinámica. Teniendo esto en mente, veamos ahora el código necesario para atender el evento del botón.

```
document.getElementById("agregar").onclick = function() { // (1)
    var nombre = document.getElementById("nombre").value;
    var caja = document.getElementById("caja");

    var nuevo = document.createElement("p"); // (2)
    nuevo.innerHTML = nombre; // (3)
    caja.appendChild(nuevo); // (4)
}
```

El ejemplo introduce dos nuevas funciones de DOM: `createElement` crea un elemento del tipo que le pasemos por parámetro, en este caso, un párrafo `<p>`; y `appendChild`, que *agrega* un nodo al árbol del documento, pasando el nodo hijo como parámetro al método que es invocado en el nodo padre.

Es importante destacar que al crearse un elemento con `createElement`, este nodo *no* se agrega al documento de manera automática sino que queda en memoria. Lo que obtenemos es una *referencia* a ese nodo nuevo. Con esa referencia debemos utilizar la función `appendChild`, que efectivamente lo relaciona con algún parente ya existente en el árbol. Si creamos un elemento con `createElement` y no lo agregamos al documento, al terminar la ejecución de la función se elimina la referencia, porque es una variable local, y el recolector de basura eliminará el nodo creado de la memoria.

Este pequeño programa, entonces, (1) espera el click del botón y, cuando sucede este evento, (2) crea un elemento `p`, (3) pone dentro de él el texto que copia del `input` y (4) agrega el nuevo párrafo dentro del `div`. El efecto visual es que los nombres se irán "apilando" a medida que sean ingresados en la lista.

Supongamos ahora que necesitamos que los párrafos pertenezcan a cierta clase, por ejemplo, debido a las reglas de estilo que se aplican con CSS. Hemos visto que para agrupar las etiquetas en clases debemos utilizar el atributo `class`. Modifiquemos el programa para que, al momento de crear cada párrafo, se le coloque la clase:

```
document.getElementById("agregar").onclick = function() {
    var nombre = document.getElementById("nombre").value;
    var caja = document.getElementById("caja");

    var nuevo = document.createElement("p");
    nuevo.innerHTML = nombre;
    nuevo.setAttribute("class", "invitados");
    caja.appendChild(nuevo);
}
```

La función de DOM `setAttribute`, y su hermana `getAttribute`, nos permite acceder a los **atributos** de los elementos, tanto los creados por programa como los que se encuentran originalmente en el documento. Recordemos que en HTML, los atributos se escriben en las etiquetas de apertura. Estamos utilizando `setAttribute` para indicar el `class` de cada párrafo que vamos creando. Claramente, los dos parámetros de la función `setAttribute` son strings que indican el nombre del atributo y el valor a escribir en él. Ejecutemos el programa en el navegador, agreguemos algunos nombres y utilicemos la herramienta "Inspeccionar elemento" haciendo click derecho sobre algún nombre:



Figura 4.6: Párrafos creados con JavaScript con atributo "class"

Encontramos ahora en el `class` de los párrafos un valor conocido, que podemos utilizar para modificar la presentación gráfica utilizando CSS. El uso de `setAttribute` es idéntico para todos los atributos que pueden tener los elementos, como exploramos en el capítulo sobre HTML.

Veamos ahora cómo podemos incluir la funcionalidad de borrado. Al crear un párrafo para el nombre del invitado, crearemos además un botón nuevo, que agregaremos también al párrafo, y que servirá para borrarlo. El código resultante completo es el siguiente:

```
document.getElementById("agregar").onclick = function() {
    var nombre = document.getElementById("nombre").value;
    var caja = document.getElementById("caja");

    var nuevo = document.createElement("p");
    nuevo.innerHTML = nombre;
    nuevo.setAttribute("class", "invitados");

    var borrar = document.createElement("input");
    borrar.setAttribute("type", "button");
    borrar.setAttribute("value", "Borrar");
    borrar.onclick = function() { caja.removeChild(nuevo); } // !!
    nuevo.appendChild(borrar);

    caja.appendChild(nuevo);
}
```

Estamos creando un nuevo `input` y usando `setAttribute` para colocar los atributos del botón que ya conocemos. Es muy interesante, sin embargo, lo que sucede con el evento click de este nuevo botón. Nótese que hemos declarado una función *dentro* de otra función. Esto es completamente válido en JavaScript y es consecuencia de que las funciones sean objetos de primer nivel. Las funciones tienen **ámbito** de la misma manera que las variables. Podemos decir que las funciones, al igual que las variables, *viven* en cierto **contexto**. Aquí estamos asociando una función anónima al evento click del botón, y es el primer caso que encontramos de una función que no está en el **ámbito global**, como estamos acostumbrados. Lo que es más asombroso aun es que la función anónima que se ejecuta al presionar el botón de borrado utilice variables que están *afuera*, es decir, en el contexto de la función superior. Nótese que tanto la variable `caja` como `nuevo` son accesibles desde la función interna. De la misma manera que las funciones tienen acceso a las variables globales, una función interna tiene acceso a las variables que están "afuera" de ella.

Es decir que, en Javascript, las funciones acceden a las variables globales por un mecanismo distinto que en lenguajes como C. Cuando programamos en C, todas las funciones son globales, ya que no puede haber funciones dentro de funciones. Las variables, en cambio, pueden ser globales o locales. Las variables globales son accesibles dentro de todas las funciones.

En JavaScript, en cambio, hablamos de **contextos**. Puede haber funciones dentro de funciones. Existe el **contexto global**, que es el que encontramos ni bien abrimos la etiqueta `<script>`, pero cada función además genera un contexto propio. Las funciones tienen

acceso a sus variables locales, es decir las variables declaradas con `var` en su contexto, y también a las variables de **contextos superiores**. Veamos este ejemplo:

```
var i = 10;
function uno() {
    var j = 20;
    dos();
    tres();

    function dos() {
        var k = 30;
        alert(i); //10
        alert(j); //20
        alert(k); //30
    }

    function tres() {
        alert(i); //10
        alert(j); //20
        alert(k); //error: k is not defined
    }
}

uno();
```

Si ejecutamos el programa en el navegador, podremos ver que las funciones de JavaScript tienen acceso a sus variables locales, es decir de su contexto, y a las variables de contextos superiores. En C tendríamos un sólo contexto superior, el global, pero en JavaScript los contextos superiores pueden ser múltiples.

Esta característica de trabajar con *contextos* es la que nos permite programar el borrado del invitado de la manera que lo hemos hecho. Cuando el intérprete ejecute la función asignada para el evento click, hará un **cambio de contexto** y volverá a encontrar las variables `caja` y `nuevo`, con referencias respectivamente a la `div` y el párrafo creados en su momento. Es importante notar que, además, *cada vez* que se ejecuta esta función de borrado, la variable `nuevo` evidentemente tendrá una referencia a un párrafo *diferente*. En caso contrario siempre se borraría el mismo. Por lo tanto, cuando la función se asocia con el evento, el intérprete está agendando también cuál era el contenido de las variables. Cuando se realiza el cambio de contexto, no sólo las variables vuelven a ser accesibles sino que tienen el valor que deben tener.

Esta característica del lenguaje se denomina *lexical scoping* y quiere decir que cuando se ejecutan las funciones, el ámbito se determina por el contexto en el cual la función fue *declarada* y no el contexto desde el cual la función es *invocada*. Por eso, cuando volvemos a la función de borrado, las variables tienen el valor "que tenían" cuando se asoció la función con el evento, es decir, cuando se la declaró. La combinación de una función más sus variables asociadas se denomina **closure**. Todas las funciones de JavaScript son en realidad *closures*. Muchos lenguajes de programación soportan este concepto.

En JavaScript, la declaración de las variables puede ocurrir en cualquier parte de la función, y no solamente en las primeras líneas, como en el C tradicional.

```

var prueba = "afuera";

function probando() {
    alert(prueba);
    var prueba = "adentro";
    alert(prueba);
}
probando();

```

Dejamos al lector la inquietud de ejecutar el código. Estamos seguros de que el resultado del primer `alert` lo sorprenderá, ya que no es "afuera". Ejecute también el mismo ejemplo pero eliminando la línea de la declaración `var` dentro de la función.

Todos estos conceptos, difíciles de comprender en sí mismos, se conjugan con la complejidad de las interfaces cada vez más repletas de eventos y asincronismo. En este sentido, el lenguaje JavaScript está llamado a traernos muchos dolores de cabeza.

4.8 EVENTOS

Para los ejemplos anteriores hemos introducido ya el evento `click`. Veamos ahora algunos eventos más, que nos servirán en la programación de las interfaces de todos los días. Los elementos de HTML generan muchos eventos, todos factibles de atraparse con JavaScript. Examinemos los más utilizados.

Cuando presentamos las pseudoclases de CSS hablamos del *foco*. Un elemento de la interfaz tiene el foco cada vez, lo que indica que es el receptor de, por ejemplo, los eventos del teclado. Hay dos eventos interesantes que podemos utilizar para trabajar con formularios y que se relacionan con el foco. Cuando un elemento recibe el foco, se produce el evento `focus`. Contrariamente, cuando pierde el foco se produce el evento `blur`. Veamos un ejemplo en donde se utilizan ambos:

```

<style> p { display: none; } </style>

Ingrese su código postal:
<input type="text" id="cp" />
<p id="mensaje1">CP debe ser un entero de 4 dígitos</p>
<p id="mensaje2">CP no es válido</p>

<script type="text/javascript">
    var cp = document.getElementById("cp");
    var mens1 = document.getElementById("mensaje1");
    var mens2 = document.getElementById("mensaje2");

    cp.onfocus = function() {
        mens2.style.display = "none";
        mens1.style.display = "block";
    }

    cp.onblur = function() {
        mens1.style.display = "none";

        if(parseInt(cp.value) != cp.value || cp.value > 9999)
            mens2.style.display = "block";
    }
</script>

```

En la primera línea se utiliza la propiedad `display` de CSS para ocultar los mensajes, presentes en la página en forma de párrafos. La propiedad `display` puede tomar tres valores:

- `none`: el elemento *no* se renderiza en la página
- `block`: el elemento se renderiza como elemento *de bloque*
- `inline`: el elemento se renderiza como elemento *de línea*

El usuario ingresará su código postal en el `input` de texto. Cuando el usuario hace click en el campo, para escribir en él, le estará dando el foco, por lo que se disparará el evento `focus`. La función asociada utiliza la propiedad `style` para modificar el `display`, que sobrescribirá el asignado por CSS. Estamos encontrando un programa que modifica el CSS de los elementos mediante JavaScript. El acceso a las reglas CSS se realiza siempre a través de esta propiedad `style` que nos ofrecen las referencias al DOM.

Cuando el usuario quita el foco al campo se llamará a la otra función, asociada al evento `blur`. Esta función realiza la verificación del dato y, si es incorrecto, muestra el párrafo con el mensaje de error, por medio de la misma propiedad `display`.

Recordemos que la función `parseInt` toma un valor e intenta convertirlo a entero. Si comparamos una variable con el resultado de enviarla a `parseInt` y no hay coincidencia, es claro que estamos ante un valor *distinto* de entero. Si la conversión a entero de un valor y el valor mismo son iguales, entonces sí estamos ante un entero.

Además de los dos eventos, es interesante ver en el ejemplo cómo se relacionan las tres tecnologías, HTML, CSS y Javascript, para construir una interface interactiva.

Examinemos ahora el evento `submit` de los formularios. Este evento es muy importante porque nos permite intervenir en el momento en que un usuario *envía* un formulario, con el objetivo de validar sus datos. Además, el intérprete nos permite *detener* el envío, de manera que podamos avisar al usuario que el formulario no está completado correctamente y evitarle pérdidas de tiempo.

Utilicemos un ejemplo parecido al anterior, en donde tenemos un formulario para ingresar el código postal, y que validaremos antes de permitir el envío:

```
<form action="noimporta" method="get" id="formulario">
    Ingrese su código postal:
    <input type="text" id="cp" />
    <input type="submit" />
</form>

<script type="text/javascript">
    var formu = document.getElementById("formulario");

    formu.onsubmit = function() {
        var cp = document.getElementById("cp");
        if(parseInt(cp.value) != cp.value || cp.value > 9999) {
            alert("El CP ingresado no es correcto"); return false;
        }
        return true;
    }
</script>
```

Hemos incluido un formulario HTML, con su campo de texto y su botón de `submit`. En el programa JavaScript relacionamos el evento submit con una función anónima. La clave de la función está en su retorno. Por convención, si una función asociada con un evento retorna un valor booleano **falso**, quiere decir que el evento debe *cancelarse*. En nuestro caso, esto tiene el efecto de que el envío del formulario se detiene. Si el retorno es **verdadero**, el evento seguirá su curso normal. Esta convención funciona para todos los eventos. Por ejemplo, podemos incluso detener un link si escuchamos su evento click y retornamos falso.

Esta funcionalidad nos permite validar los formularios antes de enviarlos al servidor. Realizamos este trabajo para evitarle pérdidas de tiempo al usuario. Más adelante veremos que también es necesario validar la información para evitar problemas de **seguridad**, pero ese tipo de validaciones serán hechas del lado servidor. Del lado cliente, las validaciones de datos nos sirven solamente para mejorar la **usabilidad** de la interfaz.

4.9 EL OBJETO STRING

Ya hemos mencionado que todas las variables de JavaScript en realidad son objetos. De hecho, incluso las funciones son objetos. Hemos trabajado también con strings, comúnmente traducido al castellano como *cadenas de caracteres*, y que nos sirven para incluir información de texto en nuestros programas. Examinemos ahora algunos **miembros** de las cadenas que nos permiten, entre otras cosas, validar los datos de formulario.

```
var lenguaje = "JavaScript";
alert(typeof lenguaje); //string
alert(lenguaje.length); //10
```

Al tratarse de objetos, las variables que son de tipo string poseen miembros propios de ese tipo. El miembro **length** (largo) nos devuelve un entero con la cantidad de caracteres de la cadena. En todos los lenguajes existe una función que recibe una string y devuelve su largo, pero en JavaScript lo obtenemos a través de una **propiedad** del objeto.

```
var lenguaje = "JavaScript";
alert(lenguaje.toUpperCase()); //JAVASCRIPT
alert(lenguaje.toLowerCase()); //javascript

"también funciona".toUpperCase(); //TAMBIÉN FUNCIONA
```

Aquí vemos dos miembros de las strings que son **métodos**. Realizan la conversión a mayúsculas y minúsculas. Es importante notar que son métodos, es decir, *funciones miembro*, y que por esa razón llevan paréntesis. De no llevar paréntesis serían propiedades o *variables miembro*. Incluimos además un ejemplo en donde se observa que un valor literal también se constituye en objeto y pueden llamarse a los métodos sobre él.

Otra función muy útil es la que nos devuelve *una parte* de una cadena:

```
var lenguaje = "JavaScript";
alert(lenguaje.substr(2,3)); //vaS
```

El método `substr` toma dos enteros que indican la posición inicial y la cantidad de caracteres a recortar, respectivamente. Devuelve una string que resulta un subconjunto de la inicial. Es muy importante notar que el primer carácter vale como posición *cero*, y que el segundo parámetro es la *cantidad* de letras y no la posición de finalización.

Hagamos un sencillo ejercicio para demostrar estas funciones:

```
<div id="caja"></div>

<script type="text/javascript">

var caja = document.getElementById("caja");
var palabra = "JavaScript";

for(var i=0; i < palabra.length; i++) {
    var letra = palabra.substr(i, 1);

    var p = document.createElement("p");
    p.innerHTML = letra;
    caja.appendChild(p);
}

</script>
```



Figura 4.7: Recorrido letra por letra de una palabra y creación de párrafos

El mecanismo de recorrer una string letra por letra nos abre un poderoso panorama para validar los datos del usuario. Por ejemplo, podríamos verificar si un email ingresado es correcto realizando recorridos sobre los caracteres, para comprobar si el dato contiene uno y sólo un signo arroba (@), la cantidad de puntos, asegurar que no contenga espacios, etcétera.

Exploraremos otro método de las strings y, a la vez, veamos cómo se puede utilizar la **consola de JavaScript** para evaluar expresiones:

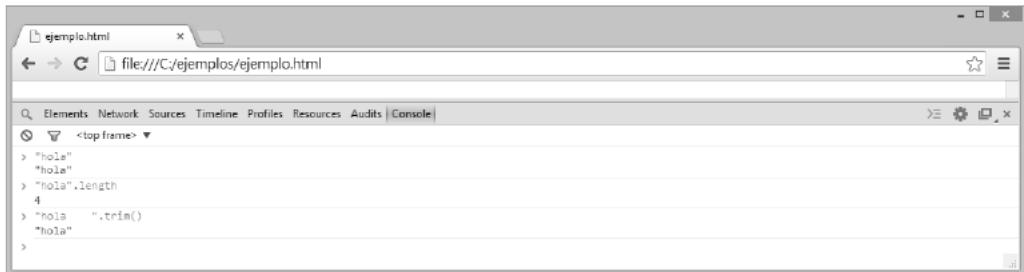


Figura 4.8: Prueba de funciones en la consola de JavaScript

El método `trim` elimina los espacios en blanco al comienzo y al final de la cadena. Es muy común que los usuarios, por descuido, agreguen espacios a los datos que envían en los formularios. Este método nos permite tratarlo de manera adecuada. Sorprendentemente, el método `trim` es una de las características agregadas con la última versión del estándar y no existía previamente. Durante dos décadas los programadores crearon sus propias funciones para realizar esta tarea de recorte.

Es importante comprender que la consola evalúa en tiempo real cada expresión que le indicamos. Si se trata de una expresión nos indicará su resultado, si se trata de una llamada a función veremos el retorno, y si se trata de una declaración de función la misma estará disponible para ser llamada a partir de ese momento. También es interesante notar que con la consola de JavaScript que nos provee el navegador podemos intervenir en la ejecución de los programas de los sitios web que visitamos. Esto nos deja claro por qué no podemos realizar las validaciones de *seguridad* del lado cliente, ya que la ejecución no puede ser controlada o garantizada, por definición.

4.10 EL OBJETO DATE

Veamos ahora un objeto muy interesante de JavaScript. El objeto Date se utiliza para la representación de fechas. Construyamos una fecha y veamos sus métodos:

```
var fechahoy = new Date();
alert(fechahoy.getFullYear());
```

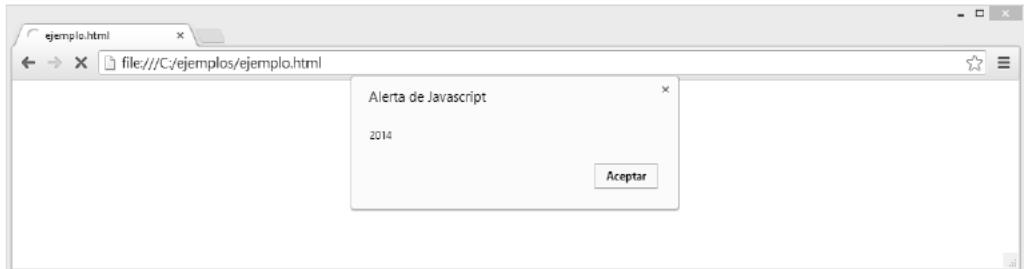


Figura 4.9: Obtención del año actual mediante el objeto Date

El objeto Date, al ser construido sin parámetros, representa la **fecha actual** de la computadora en donde se está ejecutando el programa. Desde ya, la fecha de sistema puede no

ser correcta, ya que depende de la configuración de la computadora. En algunos casos puede no ser útil si precisamos de cierta seguridad sobre su validez.

Pero el objeto Date también nos permite construir fechas arbitrarias, de este modo:

```
var fecha = new Date(2011, 0, 20);
alert(fecha.getMonth()); //0 (enero)
alert(fecha.getDate()); //20
alert(fecha.getDay()); //4 (jueves)
```

Si utilizamos los **parámetros del constructor** podemos construir fechas según nuestra necesidad. En este caso estamos creando un objeto que representa al *"20 de enero de 2011"*. Es muy importante notar que los números de los meses comienzan en 0, por lo que enero es el mes 0 y diciembre el mes 11.

Dos métodos que suelen confundirse son `getDate`, que devuelve el *día del mes* (1 a 31) y `getDay` que devuelve el *día de la semana*, siendo el domingo el día cero y sábado el seis.

Otra característica muy importante de los objetos Date es que nunca, por definición, pueden representar fechas inválidas. Veamos este caso:

```
var fecha = new Date(2013, 1, 29);
alert(fecha.getMonth()); //2 (marzo)
alert(fecha.getDate()); //1
```

Hemos construido la fecha *"29 de febrero de 2013"*, y sin embargo la fecha representada es *"1ro de marzo"*. Esto es así porque el año 2013 no fue un año bisiesto, por lo que la fecha que quisimos construir no fue una fecha existente en ese año.

Los objetos Date no solamente evitan representar fechas incorrectas, sino que emplean una **corrección** hacia adelante, de manera de representar siempre una fecha correcta. Esta corrección puede ser útil para ciertas aplicaciones. Por ejemplo, supongamos que queremos construir un programa que nos informe la distancia en días entre una fecha y otra:

```
var distancia = 0;
var desde_a = 2013; var desde_m = 3; var desde_d = 1; //desde 1/abr/2013
var hasta_a = 2013; var hasta_m = 11; var hasta_d = 10; //hasta 10/dic/2013

var f = new Date(desde_a, desde_m, desde_d + distancia);
while(f.getFullYear() != hasta_a ||
      f.getMonth() != hasta_m ||
      f.getDate() != hasta_d) {
    distancia++;
    f = new Date(desde_a, desde_m, desde_d + distancia);
}

alert("La distancia en días es " + distancia); //253
```

Claramente, la expresión `desde_d + distancia` superará el número 31, que es el máximo para un *día* del mes. Como el objeto Date se irá corrigiendo, en algún momento llegare-

mos a que el mismo representa una fecha idéntica a la fecha "de llegada". En ese momento se termina el bucle y se informa el resultado. Con otras palabras, una fecha *alcanzó* a la otra y esto es posible por la corrección que implementa el objeto Date.

4.11 VALIDACIÓN DE FORMULARIOS Y HTML5

El estándar HTML en su versión 5 ha incorporado nuevos atributos que nos permiten indicar al navegador cómo debe realizarse la validación de un formulario. Esto supone un gran avance, ya que es posible indicar sólo con atributos situaciones que antes debían programarse con JavaScript de manera explícita. Por esta razón, muchas veces se necesitaba la participación de un programador para realizar tareas rutinarias que hoy son triviales gracias a estos nuevos atributos. Exploraremos algunos de ellos:

```
<form action="noimporta" method="post">
    <input type="text" name="nombre" required />
    <input type="submit" />
</form>
```

El atributo `required` le indica al navegador que, antes de realizar el envío, compruebe que el usuario haya completado el campo. Los navegadores tienen libertad para adecuar el estilo visual del mensaje a su propia interfaz o a la del sistema operativo.



Figura 4.10: Mensaje de validación de Chrome para un campo con atributo "required"

Antes, esta validación se realizaba con JavaScript, con algún código parecido a esto:

```
if(document.getElementById("nombre").value.length < 1) {
    //error de validación, avisar al usuario
    return false;
}
```

El estándar versión 5 también introduce nuevos tipos de `input`, que permiten que los navegadores muestren controles más adecuados a cada tipo de información solicitada al usuario. Esto es muy importante en los navegadores de los teléfonos móviles, donde el teclado táctil se puede adecuar al tipo de información que el usuario debe ingresar y, de esta manera, ahorrar mucho tiempo y ganar en satisfacción:

```
<form action="noimporta" method="post">
    <input type="email" name="correo" required />
    <input type="submit" />
</form>
```

El atributo `required`, en conjunción con el `type="email"` garantiza que el navegador detenga el envío del formulario si el campo no fue completado o si contiene una string que no es un email válido. Validar un email puede ser una tarea bastante trabajosa. Si se trata de un teléfono móvil, el teclado táctil que verá el usuario tendrá una tecla para el carácter @ ya que se trata de un campo para ingresar un email.

Estas incorporaciones facilitan el trabajo a los maquetadores y permite que los programadores dediquen su tiempo a tareas más propias de su oficio.

4.12 EL FUTURO

Hemos hecho un recorrido por las características principales de JavaScript, poniendo atención en las características que lo diferencian de otros lenguajes. Algunas de estas diferencias surgen de su propio diseño, y por tratarse de un lenguaje interpretado. Otras se deben a decisiones históricas en relación al primer tipo de público que tuvo la tecnología, en los orígenes de la web.

Algunas características de JavaScript no agradan a grandes compañías. Esto motiva a que se propongan algunos lenguajes para ser sus sucesores. Por el momento, JavaScript es cada vez más usado, aunque también es cierto que las alternativas están cada vez más maduras.

El surgimiento de tecnologías que permiten programar con JavaScript del lado servidor es también una bocanada de aire fresco para el lenguaje que, además, prueba que es lo suficientemente robusto como para ser utilizado también en el lado servidor y con exigencias productivas. Además, también existen *frameworks* como Apache Cordova, que permiten desarrollar aplicaciones móviles con JavaScript, lo que representa una gran ventaja competitiva porque el código es portable, sin cambios, a múltiples plataformas. En contraposición a esto, desarrollar aplicaciones móviles con tecnología propietaria (Java u Objective-C) es sumamente costoso y no resulta ni siquiera mínimamente portable entre plataformas de distintos fabricantes de teléfonos.

Sin poder saber qué le deparará el futuro a JavaScript seguimos sosteniendo que nuestra mejor inversión a largo plazo es estudiar conceptos. En este sentido creemos mucho más conveniente dedicar nuestro tiempo a comprender qué es un *closure* antes que memorizar los nombres de métodos que cambiarán en la próxima revisión del lenguaje.

4.13 BIBLIOGRAFÍA

Un excelente recurso, que además está escrito con la pretensión de ser una *referencia total* es **David Flanagan**, “*JavaScript, The Definitive Guide*”, sexta edición, O'Reilly, Sebastopol: 2011.

Otro destacado recurso, con bastante material en castellano, es la sección *JavaScript* de la *Mozilla Developer Network*, online en: developer.mozilla.org/es/docs/JavaScript

4.14 EJERCICIOS

- 1) En un input de texto, el usuario ingresará su año de nacimiento y presionará un botón. Mostrar en la consola de JavaScript cuál es el año en el que cumplirá 18.
- 2) ¿Qué indica el operador `typeof` si lo utilizamos con el nombre de una función?
- 3) Cree distintos elementos con la función `createElement`. ¿Qué informa `typeof`?
- 4) En un formulario el usuario ingresará su nombre, su apellido y su email. Se pide detener el envío del submit si alguno de los campos no tiene datos. También se debe verificar que al menos tengan tres caracteres, y que los mismos no sean espacios. En cuanto al email, verificar que posee al menos un arroba y un punto. Si el formulario está incorrecto, debe detenerse el evento e informar la situación. Evite el `alert`.
- 5) Determine qué día de la semana (lunes, martes, etc.) fue su nacimiento.
- 6) Haciendo uso de objetos Date, crear una función que determine la cantidad de años bisiestos que hay en el intervalo entre dos años dados. Los años del intervalo se escriben en dos inputs `text` y se presiona un botón. La cantidad de bisiestos se muestra simplemente en un `alert`. Los años que se ingresan también son incluidos en el cálculo. Ejemplos para verificar: 1903 a 1941 = 10; 2092 a 2109 = 4.
- 7) Coloque dos inputs de texto y un botón. Se ingresarán dos fechas, con formato `dd/mm/yyyy`. El programa informará dentro de un `div` cuál es la distancia, en días, entre una fecha y la otra. Realizar el conteo utilizando objeto/s Date.
- 8) Hacer un formulario que permita al usuario ingresar sus datos: nombre (`text`), fecha de nacimiento (`text`), país de nacimiento (`select`). Detener el submit si el usuario es menor de edad, considerando no sólo el año. Al detener el submit se muestra un `div` que explica la situación y el input de la fecha debe destacarse con un borde rojo.
- 9) En un documento HTML coloque un `select` con los nombres de los meses, un botón y un `div`. Realice una función que "dibuje" dentro del `div` el "calendario" (una tabla) del mes seleccionado, para el año en curso. La tabla debe crearse con funciones de DOM.
- 10) Un documento posee un input `text` en donde se escribirá una fecha con el formato `dd/mm/yyyy`. Por razones de migración de software se desea que, justo antes de hacerse el submit del formulario, la fecha se transforme automáticamente en el formato `yyyy-mm-dd`. Realice el programa que altera la fecha y verifique observando la URL generada. El método del formulario debe ser GET para permitir esto último.
- 11) Se le solicitará al usuario ingresar una contraseña dos veces, en dos input `text`. Al hacer submit debe validarse:
 - que ambas contraseñas coincidan y tengan más de 6 caracteres
 - que la contraseña tenga por lo menos un signo "_" y no tenga espacios

Si hay errores debe detenerse el submit e informar al usuario el problema encontrado.

5. jQuery

jQuery (pronunciado [shei kuéri] en inglés, o [jota kuéri] en castellano) es una librería programada con el lenguaje JavaScript. Con el correr de los años se ha vuelto un aliado casi indispensable de todo desarrollador. jQuery es, básicamente, una serie de funciones que facilitan mucho el trabajo del día a día, reduciendo drásticamente la cantidad de código en nuestros programas. Como reza su slogan: "*escribir menos y hacer más*".

El interés de incluir jQuery aquí es doble. Por un lado, porque es muy útil desde un punto de vista práctico, es decir, en la práctica profesional real. Pero, y como hemos insistido varias veces, el diseño mismo de esta librería nos permite introducir temas conceptuales que formarán parte de nuestro bagaje como programadores.

Es evidente que jQuery ha suplido grandes faltas de JavaScript. Esta librería fue presentada en 2006 y en muy pocos años se ha vuelto una herramienta ineludible. El vacío que ha llenado es tan importante que incluso algunas características originales de la librería están hoy en día incluidas en los navegadores más modernos como JavaScript nativo.

Para trabajar con jQuery, lo primero que debemos hacer es descargar el código fuente desde jquery.com/download e incluirlo en nuestro documento, de la manera habitual. El código se distribuye de dos maneras: comprimido o sin comprimir. El archivo comprimido tiene el código fuente de las funciones que componen la librería y ha sido procesado con un programa para *minimizarlo*. **Minimizar** implica eliminar todos los espacios sobrantes y quitar los comentarios. Esta es la versión que recomendamos, ya que nosotros no modificaremos el funcionamiento interno de la biblioteca, por lo que nos resulta más eficiente esta versión que implica menor tiempo de descarga.

```
<!DOCTYPE html>
<html>
<head>
    <title>Prueba de jQuery</title>
    <script type="text/javascript" src="jquery-1.10.2.min.js"></script>
</head>
<!-- el documento continúa -->
```

Una vez que hemos incluido la librería jQuery, podemos empezar a hacer uso de sus funciones y disfrutar de sus beneficios.

5.1 USO DE SELECTORES

Comenzaremos demostrando uno de los usos más habituales de jQuery:

```
<script type="text/javascript">
    $("#error").hide();
</script>
```

jQuery se accede a través de una función con un nombre muy particular: la función `$`. En JavaScript, el signo **pesos** (`$`) es un carácter válido para nombre de identificadores, es decir, funciones y variables. El inventor de jQuery ha elegido este carácter inteligentemente. Cuando encontramos código JavaScript con este signo `$`, es altamente probable que estemos ante código que hace uso de la librería jQuery.

La función `$` (es decir, jQuery) recibe un parámetro `string`. La librería encuentra su razón de ser en el siguiente hecho: recorrerá el árbol del documento y utilizará esta `string` como si se tratara de un **selector** de CSS, para encontrar los elementos que coinciden con él. De esta forma, jQuery nos devuelve una colección de elementos que cumplen el selector. Esta colección de elementos es un objeto propiamente de jQuery por lo que posee métodos y propiedades que podemos acceder y que están documentados en este proyecto. Es importante notar que el intérprete de CSS no tiene nada que ver aquí: simplemente se copia la sintaxis de los selectores para permitirnos expresar un patrón de elementos a buscar. Por eso hemos hecho tanto hincapié en los selectores cuando hablamos sobre CSS. Los selectores aparecen ahora aquí en jQuery como una herramienta muy poderosa.

El método `hide` de jQuery produce que los elementos *se oculten*. En términos concretos, lo que hará jQuery es cambiar el atributo `display` al valor `none`, que como ya hemos visto tiene el efecto de anular la renderización de las etiquetas. El efecto contrario se logra con el método `show`.

Como consecuencia de esta sencilla línea hemos buscado y ocultado un elemento de la página. Comparemos las versiones con y sin jQuery:

```
<script type="text/javascript">
    //con jQuery:
    $("#error").hide();

    //sin jQuery:
    document.getElementById("error").style.display = "none";
</script>
```

Claramente, la versión con jQuery es más sintética. Sin embargo, esta sola línea puede no resultar convincente para muchos. Analizar las **dependencias** que tiene nuestra aplicación es un asunto muy serio, que debe estudiarse con cuidado antes de comenzar el desarrollo. Cuantas más dependencias externas tenga nuestro proyecto, más difícil será todo el proceso de mantenimiento, ya que nos estamos comprometiendo con los ciclos de desarrollo de todas las librerías de las que dependemos. Cada vez que se descubra y corrija un error en alguna librería tendremos que, de alguna forma, modificar nuestro propio trabajo.

Veamos un ejemplo un poco más complejo, de manera de seguir evaluando si jQuery nos conviene o no:

```
<script type="text/javascript">
    //con jQuery:
    $("#form1 p.campos").addClass("con-error");
```

```
//sin jQuery ??  
</script>
```

Esta línea de jQuery toma todos los párrafos de clase "campos" que estén adentro de un elemento con id "form1" y los *agrega* dentro de una clase "con-error". Para visualizar más claramente qué elementos estaría modificando esta línea veamos un posible caso:

```
<style>  
    .con-error { border: 1px dashed red; color:red; }  
</style>  
  
<form action="enviar.php" method="post" id="form1">  
    <p class="campos">Nombre de usuario  
        <input type="text" name="nombre" />  
    </p>  
  
    <p class="campos">Email  
        <input type="email" name="email" />  
    </p>  
  
    <p><input type="checkbox" />Acepto los términos y condiciones</p>  
</form>
```

Este ejemplo de formulario nos permite ver lo trabajoso que sería realizar el cambio sin disponer de jQuery. En primer lugar, encontrar los `<p>` que debemos afectar es en sí complejo. Deberíamos tomar todos los párrafos de la página, descartar los que no estén dentro del formulario y descartar los que no posean la clase "campos".

Una vez encontrado los dos párrafos a los que debemos modificar, tenemos que concatenar la palabra "con-error" al atributo `class` que tengan actualmente los elementos, separando con un espacio si los párrafos ya tuvieran otras clases. El resultado de aplicar el jQuery es que las etiquetas quedan de este modo:

```
<p class="campos con-error">Nombre de usuario  
    <input type="text" name="nombre" />  
</p>
```

Recordemos que una etiqueta puede estar en más de una clase a la vez y que esto se indica colocando los nombres de las clases en el atributo `class`, separados con espacios. Si no tuviéramos jQuery, todo este trabajo sería una odisea.

5.2 MANEJO DE EVENTOS

La potencia de los selectores de jQuery se pone realmente en contexto cuando examinamos las facilidades que nos provee esta librería para el tratamiento de eventos. Veamos un ejemplo muy sencillo:

```
<input type="text" id="nombre" />  
<input type="button" class="saludar" value="Saludar 1" />  
<input type="button" class="saludar" value="Saludar 2" />
```

```

<script type="text/javascript">
    $(".saludar").click( function() {
        var nombre = $("#nombre").val();

        if(nombre=="") alert("Por favor ingrese su nombre");
        else alert("Hola, " + nombre);
    });
</script>

```

El método `val` de los objetos jQuery nos permite leer y escribir el `value` de los `input`. El método `click` nos permite asociar una función con el evento nativo `click` y está preparado para recibir una función como parámetro, donde naturalmente escribimos una función anónima. Todos los elementos que sean seleccionados por el selector responderán al `click` de la manera que indiquemos en la función. Si quisieramos lograr el mismo manejo de eventos sin utilizar jQuery tendríamos que asociar la función con cada uno de los botones.

Un evento muy utilizado con jQuery es el evento `ready` del documento. El navegador dispara el evento `ready` cuando el árbol del documento se ha terminado de crear y, por lo tanto, todos los elementos del documento HTML ya se han convertido en nodos. Este evento es muy importante, porque si creamos nuestras funciones y estas funciones se ejecutan *antes* de que los nodos sean creados, las llamadas a `getElementById` fallarán. Para lograr portabilidad del código se utiliza justamente esta característica de jQuery:

```

$(document).ready( function() {
    // se ejecutará cuando el documento haya sido leído por completo

    $("#form1").submit( function() {
        // se ejecutará cuando se envie el formulario
    });
});

```

Es importante notar que el primer parámetro enviado aquí a jQuery no es una string sino una referencia: el objeto `document`. Ya hemos trabajado con este objeto al utilizar varios de sus métodos, como `getElementById`. jQuery asociará la función anónima con el evento `ready` del documento, que corresponde a la generación completa del árbol, como hemos mencionado.

También es importante observar que en el ejemplo hay varios niveles de **asincronismo**. Por un lado, el navegador esperará al evento `ready` para ejecutar la primera función. Cuando esta función se ejecute lo que sucederá es que se programará el evento `submit` del elemento con id "form1". Justamente hemos esperado a que el documento cargara por completo para asegurar que este elemento esté disponible en el árbol. Recién cuando el formulario sea enviado por el usuario se ejecutará la función anónima que corresponde al evento `submit`. Cuando se programa un evento, la función se *agenda* para ejecutarse más tarde y esto es a lo que llamamos *asincronismo*. El intérprete no detiene la ejecución de manera sincrónica, sino que declara la función y continúa con el código que sigue. La función se ejecuta recién cuando el evento se produce. Si bien esto puede parecer sencillo, no serán pocas las veces en las que sea difícil contestar *cuándo* se ejecuta alguna porción de código.

Al utilizar `ready` garantizamos que el código JavaScript se puede colocar en cualquier parte de la página, tanto en el `head` como en el `body`, sin importar si los elementos afectados por nuestro código ya han sido leídos por el navegador o no.

Hablemos ahora de otra característica de los eventos en JavaScript. Por convención, las funciones que son llamadas producto de un evento reciben un parámetro que corresponde a un objeto que representa al evento producido. Este objeto tiene varios miembros que podemos utilizar para manejar el evento, siendo el más usado `Event`. Veamos este ejemplo:

```
<input type="button" value="Hola" />
<input type="button" value="Hola" />
<input type="button" value="Hola" />

<script type="text/javascript">
    $("input[type='button']").click( function(evento) {
        $(evento.target).val("Chau");
    });
</script>
```

La propiedad `target` contiene una **referencia** al elemento que desencadenó el evento, en este caso, el botón. Es importante notar que la referencia es del tipo nativo de JavaScript y no un objeto de los retornados por jQuery. Por esta razón debemos convertirlo, enviando la referencia a jQuery para que nos devuelva un objeto sobre el cual sí podamos utilizar el método `val`. Por lo tanto, jQuery puede recibir tanto *strings* con selectores como *referencias* a nodos. En ambos casos nos retornará un objeto sobre el cual podemos llamar métodos propios de esta librería.

Con otras palabras, las referencias obtenidas con `getElementById` son las correspondientes a las funciones de DOM, y no son lo mismo que los objetos y colecciones devueltos por jQuery al utilizar un selector. A veces nos encontraremos que es necesario realizar estas conversiones.

5.3 TRAVERSING

Hemos hablado de la operación de *tree traversal* cuando desarrollamos los selectores de CSS en un capítulo anterior. Veamos ahora las posibilidades que añade jQuery a las necesidades de recorrer el árbol del documento en busca de los elementos que queremos modificar. Tomemos este ejemplo que ya hemos probado:

```
<input type="text" id="nombre" />
<input type="button" class="saludar" value="Saludar 1" />
<input type="button" class="saludar" value="Saludar 2" />

<script type="text/javascript">
    $(".saludar").click( function() {
        var nombre = $("#nombre").val();

        if(nombre=="") alert("Por favor ingrese su nombre");
        else alert("Hola, " + nombre);
    });
</script>
```

Vamos a intentar rescribir el código para eliminar el id "nombre". Es común que las interfaces terminen teniendo muchísimos `id` definidos sólo para trabajar con el comportamiento de la interfaz y no porque sean realmente necesarios. Cuantos más `id` tengamos que utilizar mayor será la cantidad de nombres que debamos recordar y finalmente no podremos mantenerlos a todos en nuestra propia memoria. Es entonces cuando el desarrollo se va a empezar a entorpecer. Además, las posibilidades de equivocarse son muy grandes. Veamos como se puede modificar para hacer uso del *traversing* en lugar de los `id`:

```
<input type="text" />
<input type="button" class="saludar" value="Saludar 1" />
<input type="button" class="saludar" value="Saludar 2" />

<script type="text/javascript">
    $(".saludar").click( function(ev) {
        var nombre = $(ev.target).siblings("input[type='text']").val();

        if(nombre=="") alert("Por favor ingrese su nombre");
        else alert("Hola, " + nombre);
    });
</script>
```

El método `siblings` (hermanos) nos devuelve los elementos hermanos de lo que hemos seleccionado y nos permite simultáneamente filtrarlos con un selector que le enviamos como parámetro. En el ejemplo tomamos el `target` del evento (es decir, una referencia al botón presionado) y buscamos un hermano que sea un `input` de tipo texto. Con este método estamos prescindiendo del `id` y haciendo *traversing*.

Si bien el efecto del programa es el mismo y la forma de lograrlo puede ser casi idéntica, hay diferencias estructurales importantes. La consecuencia más importante es la eliminación del atributo `id` del `input` de texto. Esto quiere decir dos cosas. Primero, que para que el programa funcione no es necesario que el `input` tenga ningún `id` en particular; es decir que nuestro programa es portable y no exige cambios en las marcas HTML sobre las que se aplica. En segundo lugar, el `input` es libre de utilizar el `id` para otros fines, lo que añade flexibilidad al diseño de nuestro sistema.

Veamos un ejemplo con una estructura de marcas más compleja:

```
<style> td { border: 1px solid #bbbbbb; } </style>
<table>
    <tr><td>1</td><td>Primero</td> <td><a href="#">Eliminar</a></td></tr>
    <tr><td>2</td><td>Segundo</td> <td><a href="#">Eliminar</a></td></tr>
    <tr><td>3</td><td>Tercero</td> <td><a href="#">Eliminar</a></td></tr>
    <tr><td>4</td><td>Cuarto</td> <td><a href="#">Eliminar</a></td></tr>
</table>
```



Figura 5.1: Una tabla con un link que permite eliminar cada fila

El objetivo es eliminar una fila de la tabla (es decir, un `tr`) cuando se presione el link de la última columna. Si utilizáramos `id` deberíamos *ensuciar* toda la marca HTML, asignando uno a cada fila. Intentemos utilizar el *traversing* que nos facilita jQuery:

```
<script type="text/javascript">
    $("td a").click( function(evento) {
        $(evento.target).parent().parent().remove();
    });
</script>
```

Esta solución resulta muy concisa y elegante. El `target` del evento es una referencia al link, ya que es quien produce el evento. El `parent` (padre) del link es la celda que lo contiene (`td`) y el `parent` de esa celda es la fila (`tr`) que debemos eliminar. El método `remove` quita del árbol del documento el nodo sobre cuya referencia se opera. Sencillo ¿verdad?

jQuery nos provee de potentes métodos para hacer *traversing*. Estos métodos permiten, a partir de una referencia, obtener los hijos, el padre, los hermanos, el elemento anterior y siguiente, el primero y el último... En definitiva, métodos que nos permiten recorrer la estructura del árbol para encontrar los elementos a afectar, en lugar de asignarle `id` a *todas* las etiquetas sólo porque necesitamos tomar una referencia a ellas. Muchas veces las referencias se pueden encontrar recorriendo el árbol, y partiendo de otro elemento para llegar al que precisamos.

Este trabajo con métodos de *traversing* es fuertemente utilizado por los diseñadores de *plugins*. Existen muchos programas hechos sobre la base de jQuery denominados *plugins* de jQuery. Son, en realidad, programas JavaScript que precisan de esta librería y que cumplen alguna función muy habitual como crear paginadores, *sliders* de imágenes, menús desplegables, efectos visuales, etcétera. Los programadores de estos *plugins* se basan en el *traversing* para garantizar que su código funciona sobre cualquier página donde se aplique, evitando que el autor de esta página tenga que incorporar `id` y `class` a sus elementos sólo porque el *plugin* lo requiere. De esta forma, se depende de la **estructura** del documento y no de los **atributos** de los elementos que lo componen.

5.4 OTROS USOS DE JQUERY

Hemos descripto hasta aquí las funcionalidades del núcleo de jQuery: aquellas características originales que la convirtieron en la librería JavaScript más utilizada. Con el correr de los años, jQuery fue incorporando más funcionalidades e incluso se ha bifurcado en otros proyectos que la toman como base.

Una de las características más usadas de jQuery es su conjunto de **efectos**, que pueden aplicarse a los elementos de la página. Veamos un ejemplo:

```
<style>
    div {
        border: 1px solid red;
        background: yellow;
        width: 200px;
        height: 200px;
    }
</style>

<div>Haga click aquí</div>
<script type="text/javascript">
    $('div').click( function() { $(this).fadeOut('slow'); } );
</script>
```

Este ejemplo muestra el uso del efecto `fadeOut`, que altera la opacidad de los elementos hasta ocultarlos por completo. Además hemos utilizado el atajo `$(this)` para tomar una referencia al elemento que generó el evento, en reemplazo de `event.target`. Estas dos formas de captar el origen del evento son similares.

Otros efectos incluidos en jQuery, que el lector podrá experimentar, incluyen `fadeIn`, `slideDown`, `slideUp`, `delay` y `animate`. Una característica interesante de los eventos es que se pueden *encadenar*, para que suceda uno detrás del otro. jQuery también nos permite elegir el tipo de *easing* a utilizar para el cálculo del efecto, es decir, qué curva de suavizado se utilizará para llegar de un extremo al otro de la animación (lineal, acelerado al principio, acelerado al final, exponencial, logarítmico, etcétera).

Es importante notar que estos efectos, si bien parecen sencillos, serían muy difíciles de lograr por nuestra cuenta sin contar con jQuery. Realizar estos efectos visuales implica trabajar con temporizadores, que además no son de precisión confiable en el contexto de un navegador web. Finalmente, jQuery nos garantiza portabilidad entre distintos navegadores y versiones, algo realmente muy trabajoso de conseguir. A medida que necesitamos agregar soporte para navegadores más antiguos, la complejidad (y el costo) de la solución crece desproporcionadamente.

Otro uso muy habitual de jQuery es para realizar **AJAX** (*Asynchronous JavaScript and XML*). Sintéticamente, AJAX es una técnica para recuperar información del lado servidor sin tener que recargar la página completa. El ejemplo más fácil de visualizar es el caso de la selección simultánea de país y ciudad: al elegir el país en un `select` debe poder elegirse una ciudad de ese país en otro `select`. Lo habitual es aplicar AJAX para traer las ciudades del país elegido desde el servidor, sin necesitar el envío de un formulario. Esta técnica busca ahorrar tiempo, ya que este tipo de peticiones AJAX es mucho

más veloz que una recarga completa de página porque no implica, entre otras cosas, armar el árbol del documento completo nuevamente. jQuery incluye métodos que hacen esta comunicación AJAX mucho más fácil de implementar. Exploramos estas características en el apéndice B.

Existe un proyecto que surge de jQuery pero que se conforma como una librería por separado, llamada **jQuery UI** (*user interface*, interface de usuario). Esta librería define elementos llamados *widgets* (componentes), que podemos utilizar en nuestras páginas. Por ejemplo, la creación de diálogos para reemplazar los `alert` se hace muy sencilla utilizando el componente *Dialog*. Además nos permite modificar la estética según nuestras necesidades, y de una manera uniforme para todos los componentes, con una herramienta visual muy sencilla que genera directamente el archivo CSS. Otros *widgets* y funcionalidades que podemos encontrar son: pestanas, barras de progreso, `input` con autocompletar automático, *drag and drop*, implementaciones que permiten al usuario reordenar los elementos de la página, entre otros. En definitiva, se nos propone no "reinventar la rueda" en cada interfaz que desarrollemos.

Por último, también existe un proyecto llamado **jQuery Mobile**, que incorpora funcionalidades muy utilizadas en el desarrollo para móviles como eventos táctiles en la pantalla, *widgets* propios de las interfaces móviles, transiciones entre pantallas, entre otras.

5.5 BIBLIOGRAFÍA

El proyecto jQuery está muy bien documentado, lo que definitivamente lo convierte en una buena elección para nuestras aplicaciones. Toda la documentación puede encontrarse online en jquery.com, además del acceso a los proyectos relacionados que hemos comentado, como jQuery UI.

En la documentación online se incluyen además numerosos ejemplos que, casi con seguridad, son el punto de partida (a veces también de llegada) para implementar la funcionalidad que necesitamos.

El libro sobre JavaScript de la editorial O'Reilly, que hemos mencionado en el capítulo anterior, incluye información sobre jQuery. Pero, como suele ocurrir con los libros en soporte papel, en tan solo dos años ya se atrasó siete versiones respecto de la última rama del proyecto. También es justo decir que los libros en papel tienen muchas ventajas, como la posibilidad de escribir sobre ellos, además de ser muy superiores en usabilidad a los ebooks. Por supuesto, estas son características de todos los libros en soporte papel y no sólo de éste. Como nos gusta decir: el saber a veces *tiene* que ocupar lugar.

6. PHP

PHP es un lenguaje de programación inventado en 1994 específicamente para el procesamiento de formularios del **lado servidor**. Su nombre es una sigla recursiva que significa *PHP Hypertext Preprocessor*. Es un lenguaje **interpretado** cuyo intérprete se ejecuta en los servidores, generalmente como un **módulo** del servidor HTTP Apache.

PHP es desarrollado por un equipo de programadores cuyas decisiones se eligen por votación. Está desarrollado en lenguaje C y el código fuente es abierto, por lo que puede descargarse del sitio web oficial del proyecto: php.net.

El lenguaje PHP se encuentra actualmente en la versión 5.5 e incluye soporte completo para programación orientada a objetos aunque también permite la programación imperativa y funcional. De naturaleza modular, se suele utilizar junto con bases de datos para dar vida a las aplicaciones web como las conocemos hoy en día. Aunque está fuertemente orientado a la programación web, se lo puede utilizar incluso para desarrollar aplicaciones de escritorio o para tareas en línea de comandos.

Si bien utilizaremos este lenguaje para ilustrar las operaciones que se pueden realizar del lado servidor, existen otros lenguajes también ampliamente utilizados para el mismo fin. Podemos nombrar a Java, ASP, Python y Ruby como sus principales competidores. Aunque cada lenguaje tiene su propio objetivo, ecosistema y criterio de diseño, todos ellos pueden utilizarse para crear programas que se ejecutan en el servidor web. Ejemplos de aplicaciones que se desarrollan en PHP son Facebook, Wikipedia, Wordpress, Drupal y Moodle.

6.1 PROGRAMACIÓN DEL LADO SERVIDOR

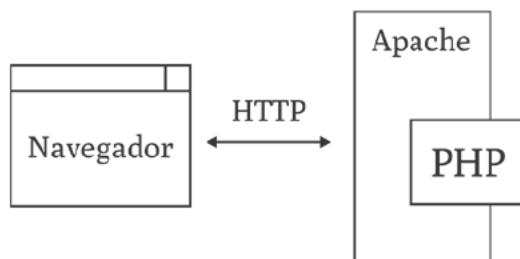


Figura 6.1: Esquema cliente-servidor

En capítulos anteriores hemos trabajado con el lenguaje JavaScript, y hemos dicho que los programas creados con ese lenguaje se ejecutan en el lado *cliente (client-side)*, es decir, que el intérprete es el navegador web. También hicimos hincapié en que el código JavaScript no tiene ningún significado particular para el servidor, y que el mismo lo entrega a los clientes de la misma manera que entrega las imágenes, los estilos CSS o las propias páginas web maquetadas con HTML.

Ahora daremos un salto y pasaremos a trabajar del lado *servidor (server-side)*. Nuestros programas serán ejecutados por el servidor, ya que el **intérprete PHP** reside allí. Es importante considerar que, a partir de este momento, tendremos varios programas destinados a ejecutarse en distintas computadoras y en distinto momento. Cuando un navegador pida un documento al servidor mediante una petición HTTP, se ejecutará un programa PHP en el servidor. Cuando este programa finalice, su resultado será devuelto como respuesta al cliente. Si dentro de la respuesta tuviéramos JavaScript, recién será interpretado cuando llegue al navegador.

Para poder trabajar del lado servidor, naturalmente, necesitamos tener un servidor. Existen múltiples posibilidades, pero atentos a nuestros fines educativos recomendaremos la instalación de un paquete preconfigurado, con el objetivo de reducir la complejidad y poder rápidamente ingresar al mundo PHP, que es lo que nos interesa. Muchas veces, cuando se trabaja en proyectos pequeños, llevados adelante por un solo programador, ésta es también la plataforma de trabajo utilizada.

Descargaremos un paquete tipo **WAMP** (*Windows, Apache, MySQL y PHP*) llamado WampServer desde wampserver.com/en. Luego de instalarlo encontraremos un nuevo ícono en la barra de notificaciones de nuestro sistema, que nos permite acceder a las configuraciones del paquete. Si bien existen varios paquetes del mismo estilo, creados por distintos autores, hemos elegido WampServer por su facilidad de instalación y configuración casi nula. Es importante saber que no se trata de un software que *simula* ser un servidor sino que realmente es el *mismo* software utilizado en los servidores que se encuentran en producción. Los autores de estos paquetes descargan el código fuente de Apache y PHP y realizan la compilación y preconfiguración para facilitarnos el trabajo. No se trata entonces de programas *parecidos* a los reales.

Ahora, nuestra computadora puede funcionar como servidor HTTP. Esto quiere decir que tenemos un programa escuchando en el puerto 80, y que contestará las peticiones HTTP que le envíemos, tal como vimos en el primer capítulo. Para probar esto, abrimos un navegador y nos dirigimos a <http://localhost>



Figura 6.2: Acceso a localhost luego de una instalación nueva de WampServer

Si realizamos una inspección del tráfico de red veremos que, efectivamente, existe **tráfico HTTP** entre el navegador y el servidor. La palabra **localhost** es un sinónimo para el *loopback* de red definido en **127.0.0.1**. En resumen, nuestra propia computadora.

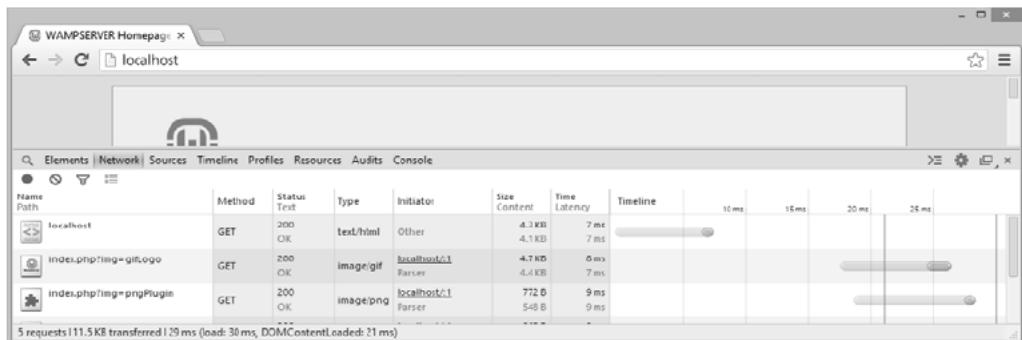


Figura 6.3: Tráfico HTTP entre el navegador y el servidor

Es muy importante tener en mente que, si bien ambos programas se están ejecutando en la misma computadora, estamos ante los dos roles de HTTP que hemos estudiado: cliente y servidor. Además verificamos que hay tráfico entre ambos. Este contexto de trabajo se suele denominar **ambiente de desarrollo local**, en donde cada programador trabaja *localmente* en su computadora antes de subir los cambios del código fuente al **repositorio** central de la empresa (si lo hubiera). Trabajar *localmente* implica que cada programador tiene su propio servidor, replicando el servidor real que se encuentra en otra parte. El servidor definitivo es el que llamamos *hosting* en el primer capítulo.

Dejamos al lector la inquietud de experimentar si es posible enviarle peticiones a este servidor local desde otra computadora de la red, por medio del número IP que tenga asignado. Si esto fuera posible, implicaría que podemos *hostear* un sitio en nuestra casa.

6.2 RELACIÓN CON HTML

Comenzaremos creando un archivo con el nombre `prueba1.php`. Es importante que el archivo tenga esa extensión y no otra. Este archivo debe estar localizado en una carpeta que llamamos **raíz de documentos** (*document root*). La carpeta raíz es el lugar donde el servidor HTTP comienza la búsqueda de los archivos que se mencionan en las URL de los peticiones. Por defecto, en Windows y con la instalación normal de WampServer este directorio raíz será `C:\wamp\www`. Diferentes servidores pueden ubicar esta raíz en otros lugares, por lo que es muy importante que detectemos dónde se encuentra. Por definición, los servidores no pueden acceder a archivos que se encuentren por fuera de su raíz. Esto se debe también a consideraciones de seguridad. Por ejemplo, no podemos pedirle al servidor que nos entregue vía HTTP el archivo `C:\Windows\win.ini`.

Una vez hayamos creado el archivo `prueba1.php` dentro de `C:\wamp\www`, pasaremos a editarlo y escribir las siguientes líneas en su interior:

```
<!DOCTYPE html>
<html>
    <head><title>Prueba de PHP</title></head>
    <body>

        <?php
            echo "Este es un programa";
        ?>
```

```
</body>  
</html>
```

Navegaremos hacia la URL <http://localhost/prueba1.php>

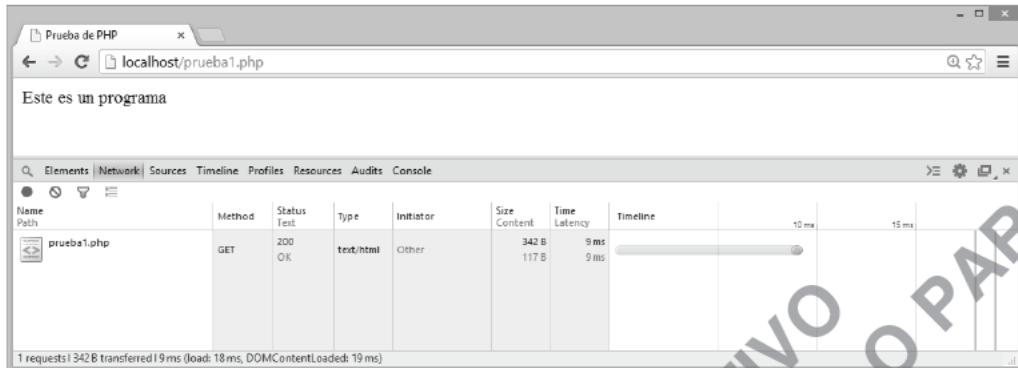


Figura 6.4: Ejecución del primer programa PHP

Con esto hemos ejecutado nuestro primer programa del lado servidor, escrito en PHP. Detallemos cómo es el funcionamiento.

Cuando escribimos la URL en el navegador y hacemos **Enter**, éste envía al servidor un paquete GET. El servidor recibe la petición y busca el archivo **prueba1.php** en el lugar donde se indica. En nuestra URL no hemos indicado carpetas, por lo que la búsqueda se realiza en la raíz. El servidor está configurado para enviar todos los archivos con extensión **.php** al **intérprete** de PHP. El intérprete procesa el programa y devuelve el resultado. Ese resultado es acomodado por Apache en el **cuerpo** de una respuesta, que se envía al cliente completando la transacción HTTP.

Cuando el intérprete de PHP procesa un archivo, lo primero que debe hacer es buscar las etiquetas **<?php** y **?>**, que indican exactamente qué partes del archivo corresponden a código que debe *ejecutar* y qué partes debe *omitar*. A las partes que se ejecutan las llamamos **contenido dinámico**, porque generarán una salida que depende de la ejecución del programa. A las partes del archivo que quedan por fuera de estas **etiquetas de PHP** las llamamos **contenido estático**, porque el intérprete no las modifica en absoluto.

Es muy importante comprender cómo se realiza la interpretación del código, porque casi siempre tendremos esta combinación de HTML y PHP en un mismo archivo. El intérprete analiza los archivos con dos ópticas:

- los caracteres que estén *fueras* de las etiquetas de PHP serán enviados intactos por el intérprete al *buffer* de salida.
- las líneas que estén *dentro* de las etiquetas de PHP serán ejecutadas de acuerdo a las reglas del lenguaje.

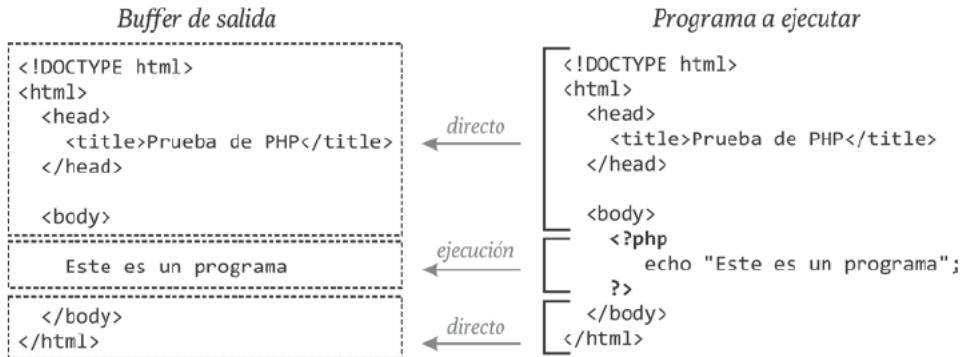


Figura 6.5: Envío de partes estáticas y dinámicas al buffer de salida

La sentencia `echo` envía un conjunto de caracteres al *buffer* de salida. Nótese que, si inspeccionamos el código fuente en el navegador (`CTRL+U`), no hay forma de determinar qué parte del contenido es estático y cuál es dinámico. Con otras palabras, no podemos decir con certeza si la frase "*Este es un programa*" estaba escrita en el archivo o fue enviada a la salida como producto de la ejecución de un programa.

A medida que nuestros programas vayan creciendo en extensión y complejidad, lo más probable es que, aunque no queramos, cometamos **errores** de sintaxis. Cuando el intérprete encuentre un problema nos informará enviando texto al *buffer* de salida, es decir, al mismo lugar que la ejecución correcta del programa. Como el intérprete se ejecuta en el servidor no tiene otra forma de comunicarse con nosotros sino a través de las respuestas HTTP. Creemos un programa de una sola línea, intencionadamente erróneo, llamado `error.php` y veamos qué genera el intérprete:

```
<?php  funcion_inexistente() ?>
```

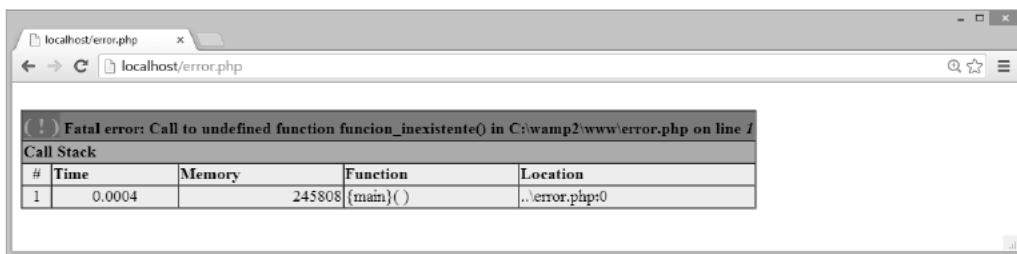


Figura 6.6: El intérprete comunicándonos un error en el programa

PHP nos informa el archivo y la línea en la que ocurre el error y nos dice "*Fatal error: Call to undefined function*" (llamada a una función sin definir). Cuando encontramos mensajes de este tipo se trata de información que el intérprete coloca en la salida para el programador. El lenguaje PHP incluye varios tipos de errores (*notice, warning, error, fatal error*) con distintos grados de severidad. En principio, todos deben corregirse.

Volviendo a la cuestión de contenido estático versus dinámico, incorporemos algo de código que, aunque totalmente inútil, es claramente un programa ejecutable:

```

<!DOCTYPE html>
<html>

<?php
    $a = 10; //asignamos algunas variables...
    $b = 2;
    $c = $a + $b; //operación aritmética
?>

<head><title>Prueba de PHP</title></head>
<body>
<h1><?php echo $c; ?></h1>
</body>
</html>

```

Con este ejemplo queremos ilustrar dos cuestiones. Por un lado, como hemos dicho, desde el lado cliente no se puede distinguir el código estático del código dinámico. Si miramos el código fuente de esta página cuando llega al navegador, encontraremos un `<h1>` con un número 12 en su interior. ¿Cómo podríamos saber que ese "12" es producto de una operación si no viéramos el código del programa *server-side* que lo generó?

```

1 <!DOCTYPE html>
2 <html>
3
4
5     <head><title>Prueba de PHP</title></head>
6     <body>
7         <h1>12</h1>
8     </body>
9 </html>
10

```

Figura 6.7: Código fuente generado por el programa

Por otro lado, es importante ver cómo el intérprete considera a todo el archivo como un *mismo* programa, aunque ejecute cada bloque en orden a medida que los encuentra. Esto podemos deducirlo del hecho de que la variable `$c` que se calculó en el primer bloque sigue existiendo en el bloque siguiente, cuando se hace un `echo` mandando al *buffer* de salida su valor. El siguiente programa es absolutamente equivalente y da el mismo resultado:

```

<?php $a=10; ?>
<!DOCTYPE html>
<?php $b=2; ?>
<html>
<?php $c = $a + $b;      ?>
<head><title>Prueba de PHP</title></head>
<body>
<h1><?php echo $c; ?></h1>
</body>
</html>

```

Para insistir con la diferencia estático/dinámico, incluso el siguiente código tiene el mismo resultado desde el punto de vista del cliente:

```

<!DOCTYPE html>
<html>
    <head><title>Prueba de PHP</title></head>
    <body>
        <h1><?php echo 12; ?></h1>
    </body>
</html>

```

Con otras palabras, que el contenido estático y dinámico no sea distinguible del lado cliente quiere decir que nadie notaría la diferencia entre estos últimos tres ejemplos.

PHP es un lenguaje con una sintaxis similar a C, por lo que utilizaremos llaves para marcar bloques, puntos y coma para finalizar sentencias, corchetes en los vectores y paréntesis para indicar llamadas a función.

Veamos ahora cómo se comportan los **bloques** y su relación con el contenido estático. Omitimos la estructura base del documento HTML que el lector ya conoce.

```

<?php for($i = 0; $i < 5; $i++) { ?>
    <p>Hola</p>
<?php } ?>

```



Figura 6.8: Página renderizada a la izquierda y su código fuente a la derecha

Este sencillo ejemplo demuestra cómo las llaves se extienden *más allá* de cada bloque de PHP. El intérprete buscará más abajo la llave que cierra el bucle `for`, de manera que el contenido estático que ha quedado encerrado será enviado, como todo contenido estático, intacto al *buffer* de salida. Como el bucle realiza cinco vueltas, el contenido estático es enviado a la salida cinco veces.

Este comportamiento de los bloques puede ser desconcertante al principio, pero es la base de toda la programación en PHP para generar páginas dinámicas. Las llaves funcionando de esta manera nos permiten realizar cosas como estas:

```

Seleccione un año:
<select name="anio">
<?php for($i = 2013; $i >= 1900; $i--) { ?>
    <option><?php echo $i; ?></option>
<?php } ?>
</select>

```



Figura 6.9: Lista desplegable renderizada a la izquierda y su código fuente a la derecha

De hecho, esta forma de trabajar con PHP es tan habitual que existe una forma abreviada del `echo`. El programa anterior también puede ser escrito así:

```
Seleccione un año:  
<select name="anio">  
  <?php for($i = 2013; $i >= 1900; $i--) { ?>  
    <option><?= $i ?></option>  
  <?php } ?>  
</select>
```

Para que esta **etiqueta abreviada** funcione en intérpretes PHP anteriores a la versión 5.4, es necesario que habilitemos una opción de configuración. Haciendo click izquierdo sobre el ícono del WampServer se desplegarán las opciones de los distintos programas. Dentro de "PHP Settings" es preciso activar la directiva "*short open tag*".

Nótese que cuando el bloque PHP tiene una sola sentencia, como el `echo` que se realiza dentro del `option`, se puede omitir el `;` terminal.

En resumen:

- `<?php` indica el comienzo de un bloque de código PHP
- `<?=` es la versión abreviada de `<?php echo`
- `?>` indica el fin de un bloque PHP

Queremos hacer hincapié en que el intérprete deja intactos los caracteres que se encuentran *fueras* de los bloques que debe interpretar. Con otras palabras: *PHP no sabe nada de HTML*. El siguiente programa es totalmente válido:

```
<?php $link = "pagina2.html" ?>  
<a href="<?= $link ?>"> Ir a página 2</a>
```

El resultado ejecutado, visto desde el navegador, es un link de esta forma:

```
<a href="pagina2.html"> Ir a página 2</a>
```

El primer bloque asignó una variable `$link` y el segundo bloque realizó un `echo`. La circunstancia de que el `echo` justo se produce dentro del atributo del link es totalmente ajena a PHP. Es por esta razón que nuestro trabajo como programadores es *siempre* chequear el **código fuente generado** que llega al navegador y no focalizar solamente en el resultado renderizado en pantalla.

Esta es la causa por la cual PHP se llama *preprocesador de HTML*: porque nos permite generar código HTML de manera dinámica, de la misma forma que el preprocesador de los compiladores de C son un paso previo a la compilación definitiva. PHP nos permite, del lado servidor, ejecutar un programa que escribirá *otro* programa. De alguna manera, HTML es un lenguaje que transporta una serie de órdenes para el navegador y esas órdenes tienen una sintaxis específica que ya hemos estudiado. Nuestra tarea como programadores de lado servidor es, justamente, crear programas que escribirán a su vez otro lenguaje: HTML. Como consecuencia debemos atender a la sintaxis de cada uno y generar código válido para todos los intérpretes involucrados en la cadena.

Es importante notar, en este último ejemplo, la diferencia entre las **comillas** que se encuentran *dentro* y *fuerza* de los bloques de PHP. Dentro del primer bloque, las comillas delimitan la string `"pagina2.html"`. Como en muchísimos lenguajes, las strings de PHP se delimitan con comillas dobles o simples. Pero en el caso del atributo `href`, las comillas dobles se encuentran fuera del bloque de PHP y están allí para que, como contenido estático, lleguen intactas al navegador. Esto es así porque el estándar HTML exige que los valores de los atributos estén entrecomillados. Al principio puede ser confusa la finalidad de cada comilla, pero con el tiempo se hará más fácil visualizar la razón de ser de cada par. Ahora es importante distinguir, básicamente, qué comillas son exigidas por el lenguaje PHP y cuáles corresponden al estándar HTML.

6.3 VARIABLES Y TIPOS

Al igual que JavaScript, PHP es un lenguaje **interpretado y dinámicamente tipado**. Es decir que el tipo de las variables se va determinando a medida que les asignamos valores en el transcurso de la ejecución. A diferencia de JavaScript, sin embargo, las variables no se pueden **declarar**. No existe un operador como `var`: las variables sencillamente se empiezan a utilizar. Todas las variables comienzan con el signo **pesos** (`$`).

```
<?php
    $i = 0; //tipo entero
    $i = "Cadena"; //tipo string
?>
```

Para examinar cuáles son los **tipos** de PHP utilizaremos una función muy útil, llamada `var_dump`. Esta función genera un **volumen** de la variable, es decir, envía al *buffer* de salida toda la información que el intérprete conoce sobre esa variable en esa línea del programa. Veamos un ejemplo sencillo:

```
<?php
    $i = 0;
    var_dump($i);

    $i = "Cadena";
    var_dump($i);
?>
```

A screenshot of a Microsoft Edge browser window. The address bar shows 'localhost/pruebas.php'. The page content displays the output of the PHP var_dump function:

```
int 0
string 'Cadena' (length=6)
```

Figura 6.10: Salida de la función var_dump

En este caso, las dos ejecuciones de `var_dump` demuestran que ha cambiado el tipo de la variable. En el primer caso es tipo `int` y luego es `string`. Examinemos más tipos del lenguaje PHP:

```
<?php
    $i = 10.6; var_dump($i); //float
    $i = true; var_dump($i); //bool
    $i = array(1,2,3); var_dump($i); //array
    $i = new SplStack; var_dump($i); //object
?>
```

Es interesante ver que `var_dump` nos muestra mucha información útil sobre vectores e instancias de objetos que, de otro modo, sería trabajoso conseguir. Esta información es vital al momento de solucionar errores en nuestros programas. Cuando necesitemos inspeccionar una variable utilizaremos `var_dump`. Recordemos que, tal como vimos con JavaScript, los lenguajes dinámicamente tipados pueden ser muy desafiantes a la hora de solucionar problemas relacionados con valores y tipos inesperados en las variables.

Igual que JavaScript, los tipos se convierten implícitamente a otros tipos para realizar ciertas operaciones. Por ejemplo:

```
<?php
    $i = 20; $j = 4.4;
    var_dump($i + $j); //(float) 24.4

    $i = 10; $j = "a";
    var_dump($i + $j); // (int) 10

    $i = 10; $j = "5a";
    var_dump($i + $j); // (int) 15

    var_dump($i . $j); // (string) 105a
?>
```

Observamos aquí varias diferencias con JavaScript. Primero, tenemos dos tipos diferentes para enteros y flotantes. Segundo, una string se puede sumar con un número. Tercero, la conversión de string a número se realiza carácter por carácter y, en el peor caso, produce un `(int) 0` si la conversión no es posible, en lugar del `Nan` de JavaScript. Por último, el operador `+` siempre es **suma aritmética** y el punto `.` es siempre el operador de **concatenación**.

Los operadores de comparación también realizan conversiones:

```
<?php
    $a = 10; $b = "10";
    if($a == $b) echo '$a es igual a $b <br />'; //verdadero

    $c = 0; $d = false;
    if($c == $d) echo '$c es igual a $d <br />'; //verdadero

    if($c === $d) echo '$c es idéntico a $d <br />'; //falso
?>
```

La primera comparación es verdadera porque se convierte el "10" string en 10 entero. La segunda comparación también es verdadera porque el valor booleano falso se convierte al 0 entero. Lo interesante es que la tercera comparación es falsa, porque no utilizamos el operador de **igualdad** (`==`) sino el de **identidad** (`===`) que, a diferencia del otro, verifica no sólamente el valor de las variables sino también su tipo. Este operador de identidad es importante porque algunas funciones tienen un retorno `false` y también `0`, por lo que el operador de igualdad no serviría para distinguir entre ambos retornos.

Nótese que las strings que se envían a la salida por medio de los `echo` están delimitadas con **comillas simples** en lugar de dobles. Esto es así porque las **comillas dobles** en PHP producen una **interpolación** de las variables. La interpolación es el proceso de reemplazar el nombre de la variable por su valor dentro de una string. La interpolación no sucede si la string se delimita con comillas simples. Experimentemos con un ejemplo:

```
<?php $hola = "chau";
      echo "hola $hola <br />";           //comillas dobles
      echo 'hola $hola';                   //comillas simples
?>
```

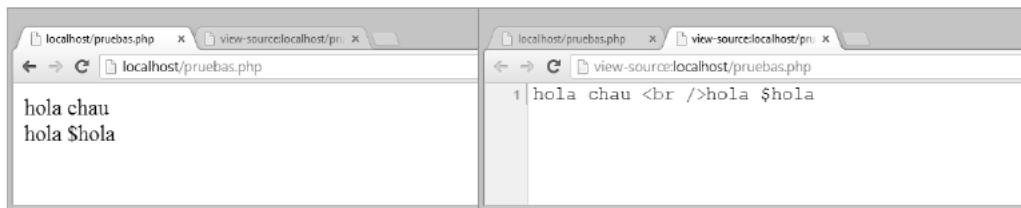


Figura 6.11: Diferencia entre comillas simples y dobles

Por lo demás, ambas comillas son totalmente intercambiables. La mayoría de las veces la elección entre una y otra es meramente azarosa o responde a gustos personales.

6.4 ARRAYS

Encontramos en el lenguaje PHP un tipo nativo muy versátil para representar **colecciones** de datos. Si bien este tipo es llamado **array**, se trata en realidad de un **mapa**. Los mapas son estructuras de datos que almacenan pares, conocidos como *claves* y *valores*. Veamos un ejemplo y profundicemos en su estudio.

```
<?php $uno = array(1,2,3);
      var_dump($uno);
?>
```

The screenshot shows a browser window with two tabs: 'localhost/pruebas.php' and 'view-source:localhost/pruebas.php'. The main content area displays the output of the `var_dump` function on a simple array. The output is:

```
array (size=3)
  0 => int 1
  1 => int 2
  2 => int 3
```

Figura 6.12: Volcado de un vector a través de la función `var_dump`

Observamos que el array tiene tres elementos y, entre corchetes, han surgido los números 0, 1 y 2. Estos números *parecen* los **subíndices** del vector, tal como ocurre en el lenguaje C. Nótese que `var_dump` utiliza un símbolo `=>` para relacionar esta especie de subíndice con los valores que hemos colocado en el vector.

El acceso a los valores del array se realiza con corchetes, como en C.

```
<?php $uno = array(1,2,3);
      echo "el primer valor vale " . $uno[0]; //1
?>
```

La construcción `array()` nos permite crear un array e incorporarle datos. Como hemos dicho, en PHP no se pueden declarar las variables, por lo que debemos utilizar esta especie de función para crear arrays. Lo interesante es que esta función nos permite no sólo determinar los datos sino también los "subíndices":

```
<?php $dos = array(3=>1, 5=>2, 20=>3);
      var_dump($dos);
?>
```

The screenshot shows a browser window with two tabs: 'localhost/pruebas.php' and 'view-source:localhost/pruebas.php'. The main content area displays the output of the `var_dump` function on an associative array. The output is:

```
array (size=3)
  3 => int 1
  5 => int 2
  20 => int 3
```

Figura 6.13: Un vector con claves arbitrarias

Lo que podemos observar entonces es que no se trata justamente de *subíndices*. Los **mapas** almacenan un conjunto de **pares de datos**. Estos datos se denominan **clave** (*key*) y **valor** (*value*). Las claves son las que utilizamos dentro de los corchetes. Cuando accedemos a un array utilizando una clave, el intérprete nos devuelve el valor asociado. Por lo tanto, las claves no nos sirven para, por ejemplo, determinar cuántos elementos tiene un array. En C, los subíndices siempre son consecutivos, algo no obligatorio en PHP.

```
<?php $tres = array(3000 => 10);
      $cant = count($tres);
      echo "El array tiene $cant elementos"; // 1
?>
```

La función `count` nos retorna un entero con el tamaño que tiene un array. En este ejemplo vemos que, si bien la clave es 3000, el array sólo tiene un elemento, más concretamente, un solo elemento entero con clave 3000.

Las claves no se pueden repetir, por lo que si accedemos al array en una clave ya existente lo que haremos será sobrescribir el valor:

```
<?php $cuatro = array(2=>10, 3=>20, 4=>30);

    echo $cuatro[3]; //muestra 20
    $cuatro[5] = 100; //crea un par
    $cuatro[3] = 200; //sobrescribe el 20
    echo $cuatro[5]; //muestra 100
    echo $cuatro[3]; //muestra 200
?>
```

Algo que es más sorprendente es que los valores no tienen que ser necesariamente del mismo tipo. Los arrays de PHP pueden almacenar tipos heterogéneos:

```
<?php $cinco = array(20, "hola", true);
    var_dump($cinco);
?>
```

De hecho, los arrays pueden almacenar otros arrays como valores:

```
<?php $seis = array(4, 5, "hola");
    $siete = array(1, $seis, 3);

    var_dump($seis[0]); // (int) 4
    var_dump($siete[0]); // (int) 1
    var_dump($siete[1][1]); // (int) 5
    var_dump($siete[1][2]); // (string) "hola"
    echo count($seis); // 3
    echo count($siete); // 3
?>
```

Podemos ilustrar el array `$siete` de la siguiente forma:

0	1
1	0 4
	1 5
	2 "hola"
2	3

Figura 6.14: Esquema del array `$siete`, conteniendo una copia de `$seis` en su interior

Cuando se utilice la función `array` para construir mapas, el intérprete asignará las claves automáticamente si no fueran declaradas, partiendo del entero 0 e incrementando consecutivamente, ya que las claves no pueden ser vacías.

```
<?php $ocho = array("a", "b");
    var_dump($ocho); // [0]=>"a"      [1]=>"b"
?>
```

Ya hemos visto que el operador `=>` se utiliza para indicar la relación entre claves y valores, y que puede utilizarse al construir el array para forzar las claves:

```
<?php $nueve = array(20=>"a", 30=>"b");
    var_dump($nueve); // [20]=>"a"      [30]=>"b"
    echo count($nueve); // 2
?>
```

Podemos, además, forzar las claves de algunos pares pero dejar librado a la determinación del intérprete las claves de otros:

```
<?php $diez = array(20=>"a", 30=>"b", "c", "d");
    var_dump($diez); // [20]=>"a"  [30]=>"b"  [31]=>"c"  [32]=>"d"
    echo count($diez); // 4
?>
```

Siempre que encuentre un valor cuya clave no ha sido definida, el intérprete tomará la clave más grande del array y le sumará una unidad para construir la clave faltante. Dejamos al lector la tarea de ejecutar el siguiente programa:

```
<?php $once = array(20=>"a", 30=>"b", "c", "d", 100=>"e", "f");
    var_dump($once);
?>
```

Otra diferencia evidente con los vectores clásicos de C es que el tamaño es dinámico, es decir que la cantidad de elementos que almacena puede variar durante la ejecución del programa. Una vez que hemos declarado el vector se pueden agregar y quitar pares. Veamos un ejemplo:

```
<?php $doce = array(3=>"a", "b");
    echo $doce[4]; // b

    $doce[10] = "c";
    echo count($doce); // 3

    unset($doce[3]);
    echo count($doce); // 2

    $doce[] = "e";
    $doce[] = "f";
    var_dump($doce); 4=>b 10=>c 11=>e 12=>f
    echo count($doce); // 4
?>
```

La función `unset` elimina un par del array, dada su clave. Además, observamos que se pueden agregar valores al array con la utilización de los **corchetes vacíos**. El comportamiento del intérprete respecto de completar la clave faltante es el mismo: tomar la clave más grande de las existentes y sumarle uno.

Veamos ahora una característica que suele ser aún más desconcertante: las claves también pueden ser strings y, de hecho, pueden convivir con las claves enteras:

```
<?php $trece = array("uno"=>10, "dos"=>20, 3=30);
    echo $trece["uno"]; //10
    $trece[] = "nuevo";
    echo $trece[4]; //nuevo
?>
```

Las **claves**, entonces, pueden ser enteros o strings, y sirven para identificar a cada parte de la colección, por lo que no se pueden repetir. Los **valores** pueden ser de cualquier tipo, incluso otro array con sus propios pares.

Luego de explorar las características de los arrays de PHP, podemos concluir que no son la misma estructura de datos que los vectores tradicionales de C, aunque sintácticamente se vean muy similares. Enumeremos las diferencias:

- Los vectores de C tienen *subíndices* numéricos, mientras que las *claves* de los arrays de PHP pueden ser enteros o strings.
- Los subíndices de C siempre son consecutivos, mientras que las claves en PHP pueden ser cualquier número o string arbitrarios.
- Los vectores almacenan *datos* del mismo tipo, mientras que los arrays de PHP pueden almacenar *valores* de tipos diferentes.
- Los vectores de C tienen un tamaño constante definido al momento de compilar, mientras que los arrays de PHP son dinámicos en tiempo de ejecución.
- Los vectores no se pueden pasar ni retornar a una función más que por su dirección en memoria, mientras que en PHP sí se pueden copiar, pasar y retornar arrays como cualquier valor escalar. En PHP no existen los punteros.
- El lenguaje C no puede informarnos en tiempo de ejecución el tamaño de un vector, mientras que PHP dispone de la función `count`.
- El lenguaje C es compilado, mientras que PHP es interpretado; en esta dicotomía radica la mayor parte de sus diferencias.

Para finalizar nuestra introducción a los arrays de PHP, presentemos una estructura de control que nos permitirá recorrer esta estructura de datos. En general no podremos utilizar un bucle `for` con un contador ya que, como vimos, las claves son muy distintas a los subíndices del lenguaje C y pueden no ser consecutivas. De hecho, la mayor parte de las veces ni siquiera serán números sino strings, lo que hecha por tierra la posibilidad de iterar con un `for`. Veamos este ejemplo:

```
<?php $prueba = array("uno"=>1, "dos"=>2, "tres"=>3);
    foreach($prueba as $valor) {
        echo '$valor es ' . $valor . '<br />';
    }
?>
```

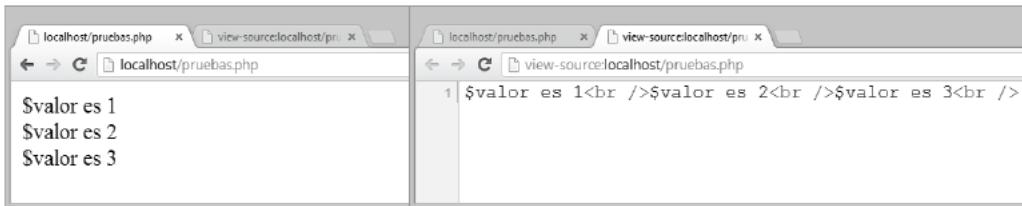


Figura 6.15: Recorrido de un array con la estructura de control foreach

El bucle `foreach` nos garantiza que recorrerá todo el array, y nos entregará en una variable auxiliar cada *valor*, vuelta a vuelta. La variable auxiliar `$valor` tiene sentido sólo dentro del bucle y puede tener, por supuesto, cualquier otro nombre.

Si además de los *valores*, precisáramos las *claves* del array, podemos utilizar esta forma:

```
<?php $prueba = array("uno"=>1, "dos"=>2, "tres"=>3);
    foreach($prueba as $clave => $valor) {
        echo 'Clave ' . $clave . ' tiene el valor ' . $valor . '<br />';
    }
?>
```



Figura 6.16: Utilización de `foreach` para recorrer las claves y valores de un array

6.5 DATOS DE FORMULARIO

Hasta aquí hemos experimentado con la sintaxis y las características básicas de PHP. Pero los programas necesitan de algo más para convertirse en entidades útiles: **datos externos**. En el caso del lado servidor, los datos externos provienen típicamente del lado cliente y, principalmente, de los formularios HTML.

Construyamos un formulario en un archivo, y un programa PHP en otro.

```
<!-- primero.html mostramos sólo el body -->
<form action="segundo.php" method="get">
    Su nombre: <input type="text" name="nombre" /> <br />
    Su apellido: <input type="text" name="apellido" /> <br />
    <input type="submit">
</form>

<?php //segundo.php
echo "Su nombre es " . $_GET['nombre'] . " " . $_GET['apellido'];
?>
```

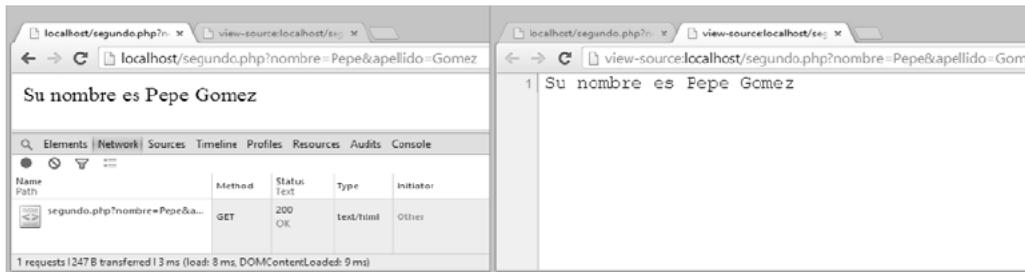


Figura 6.17: Utilización en PHP de los datos enviados por GET. Nótese la URL.

Al llenar el formulario y hacer submit, el navegador construyó un paquete GET y colocó las variables en la URL. Estos datos son llamados a veces **parámetros GET**. Ya hemos estudiado este mecanismo, además de los paquetes POST, en el capítulo sobre HTML.

La petición GET llegó al servidor, que buscó y encontró el archivo señalado en la URL **segundo.php**. La particularidad de este archivo es que se trata de un programa PHP, de manera que corresponde al intérprete su ejecución. El resultado de la ejecución es simplemente enviar a la salida una string, que surge de la concatenación de algunas variables. Estas variables son las que contienen los datos de formulario.

Es importante notar que la variable **`$_GET`** es un array. Lo que ha hecho el intérprete es tomar los datos que han llegado por el método GET y cargarlos, *automáticamente*, en estas variables con nombre extraño. Los **value** de los **input** se convierten en **valores** del array y los **name** se convierten en las **claves**.

Si editamos el código fuente del formulario para que el método sea POST, el archivo PHP debería tomar los datos de esta forma:

```
<?php //segundo.php
echo "Su nombre es " . $_POST['nombre'] . " " . $_POST['apellido'];
?>
```

El funcionamiento es exactamente el mismo. La única diferencia radica en que la variable donde se colocan los datos se llama **`$_POST`** en lugar de **`$_GET`**. Esto es así porque el intérprete reconoce que la ejecución comienza con una solicitud POST. Ya hemos señalado que la única diferencia entre GET y POST estriba en el *lugar* donde se colocan los datos de formulario, esto es, en la URL o en el cuerpo del paquete. Desde el punto de vista de PHP, todo este manejo es automático. Si el formulario fue enviado como GET debemos buscar los datos en **`$_GET`**, caso contrario en **`$_POST`**.

Recordemos que si escribimos una URL en la barra de direcciones del navegador y presionamos **Enter**, se construirá una solicitud de tipo GET y se enviará al servidor. Desde el punto de vista del servidor, es *imposible* diferenciar si la variable **`$_GET`** se ha rellenado con datos que surgen de un formulario o bien con datos que fueron tipeados manualmente en la barra de direcciones.

Ahora que sabemos dónde buscar los datos de formulario, modifiquemos nuestro programa para que todo el código quede en el mismo archivo. En las aplicaciones reales no

se suele separar el formulario en un documento y el *proceso* de ese formulario en otro, sino que se intenta unificar en un solo programa que se ejecutará dos veces.

```
<!-- tercero.php mostramos sólo el body -->

<body>
<?php if(isset($_GET['nombre'])) {
    echo "Su nombre es " . $_GET['nombre'] . " " . $_GET['apellido'];
}
?>

<form action="tercero.php" method="get">
    Su nombre: <input type="text" name="nombre" /> <br />
    Su apellido: <input type="text" name="apellido" /> <br />
    <input type="submit">
</form>
</body>
```

Aquí encontramos un formulario que realiza el `submit` sobre si mismo. Es decir que, este programa PHP se ejecuta **dos veces**: cuando el usuario accede a completar el formulario y cuando recibe los datos que se han completado.

¿Cómo distinguir entre estas dos ejecuciones del programa? La función `isset` nos devuelve verdadero o falso, dependiendo de si una variable está seteada a esa altura del programa o no. Esto puede parecer extraño ya que somos los programadores quienes declaramos las variables de los programas, de manera que parece un sinsentido preguntar si una variable existe o no. Pero en PHP, como las variables no pueden declararse, existe esta construcción que nos indica si una variable existe. Es un error intentar evaluar una variable que no existe. Recordemos que, al estudiar los arrays, hemos visto que las variables se pueden "desetear" con la función `unset`.

El caso de que una variable *no existe* es, justamente, el caso de los datos de formulario. En la primera ejecución, el parámetro por GET llamado "nombre" no existe, dado que estamos entrando por primera vez. Como esta variable no existe, se omite el cuerpo del `if`. En la segunda ejecución esta variable sí está seteada, porque forma parte de los datos del formulario y el intérprete automáticamente crea esa clave en el array. Es importante notar que esta variable `$_GET['nombre']` está siendo utilizada dentro del `if` como una **bandera**, para distinguir entre una ejecución y otra. Lo extraño, si se quiere, es que no utilicemos el *valor* de la variable sino su mera *existencia* o *inexistencia*.

Podemos imaginar que el intérprete mantiene una tabla con las variables que existen en un programa. Esto se llama **tabla de símbolos**. Cuando asignamos una variable por primera vez, el intérprete crea una entrada en la tabla de símbolos. Cuando escribimos una expresión que tiene variables, el intérprete busca en la tabla los valores de las variables y calcula la expresión. Si la variable no existe en la tabla de símbolos, se produce un error. La función `isset` busca un nombre en la tabla de símbolos y nos retorna un valor booleano que indica si tal variable está o no en la tabla. Hacemos hincapié en que no nos dice nada sobre el *valor* ni el *tipo* de la variable, simplemente si está o no en la tabla de símbolos que se conoce en ese momento. Recordemos además que la tabla de símbolos se va a ir alterando en el transcurso de la ejecución, no solamente los valores

de las variables, sino la cantidad de variables existentes, ya que en PHP no se declaran las variables antes de ejecutarse el programa.

Modifiquemos ahora el programa para que, además del nombre y apellido, se solicite al usuario su año de nacimiento. Con esto calcularemos su edad aproximada.

```
<!-- cuarto.php mostramos sólo el body -->

<body>
<?php if(isset($_GET['nombre'])) {
    $anios = date('Y') - $_GET['anio'];
    ?>

    <p>Su nombre es <?=$_GET['nombre']?> <?=$_GET['apellido']?>
    y tiene más o menos <?=$anios?> años.</p>

    <a href="cuarto.php">Volver</a>
<?php } else { ?>

    <form action="cuarto.php" method="get">
        Su nombre: <input type="text" name="nombre" /> <br />
        Su apellido: <input type="text" name="apellido" /> <br />
        Su año de nacimiento: <input type="text" name="anio" />
        <br />
        <input type="submit">
    </form>
<?php } ?>
</body>
```

Hemos realizado dos cambios estructurales. En primer lugar, el texto que se indica como resultado ("Su nombre es...") ya no es un `echo` sino que hemos *salido* de PHP para colocar el texto como contenido estático. Volvemos a *entrar* a PHP para hacer `echo` de las variables en los tres lugares que corresponde al contenido dinámico, con la etiqueta abreviada `<?=`. Esta forma de trabajo es preferible a la anterior, es decir, a dejar todos los textos (incluso etiquetas) dentro de strings literales que luego se hacen `echo`. De la forma que lo hemos modificado, si un diseñador abre nuestro programa en un editor gráfico podrá ver el párrafo y podrá cambiarle los estilos o lo que necesite. Si dejamos el párrafo dentro de una string, el editor gráfico no reconocerá la etiqueta porque no "sabe" ejecutar código PHP. Además, nuestros editores colorean el código fuente. El coloreo será más significativo de esta forma que si dejamos todo dentro de una string.

En segundo lugar, hemos utilizado un `else` para encerrar el formulario en un bloque mediante llaves. Esto genera que el formulario "no se vea" en pantalla cuando mostramos el resultado. Esta consecuencia visual surge del funcionamiento de las llaves de PHP que ya hemos explorado con algunos bucles. Al mostrar el resultado el intérprete saltará el bloque `else`, por lo que el contenido estático, directamente, nunca llegará al navegador en la segunda ejecución de nuestro programa. Nótese que en la "pantalla" de resultados hemos incluido un link que vincula al *mismo* programa, pero en su URL no incluimos ningún parámetro por GET. El efecto de hacer click sobre el link es que volveremos a ver el formulario, debido a que el programa se ejecutará nuevamente y no se cumplirá la condición del `if`. Hacer click sobre este link es totalmente idéntico a escribir `cuarto.php` en la barra de direcciones y presionar `Enter`.

Para el cálculo de la edad hemos optado, sencillamente, por restar el año actual con el año de nacimiento aunque esto no sea del todo verdadero. Para obtener el año actual utilizamos la función `date` de PHP con un parámetro `string` que indica que debe retornar el año en cuatro dígitos. Esto nos retornará un entero con el año que marca la fecha del sistema operativo del servidor. Nótese que este entero se está *restando* con el dato que proviene del formulario. Por definición, los datos de formulario son siempre `strings`, por lo que está ocurriendo una conversión implícita a número. Dejamos al lector la inquietud de verificar el tipo, por ejemplo, mediante `var_dump($_GET)` para obtener un volcado del array.

6.6 CONSERVACIÓN DE VARIABLES

Avanzaremos un poco más, modificando el programa para que el nombre y apellido se soliciten en una "pantalla", luego se pida el año de nacimiento en otra y luego se informen los datos en una tercera. Hasta aquí hemos tenido dos "pantallas", es decir: (1) pedimos los datos y (2) mostramos los resultados. Veamos si podemos lograrlo en tres pasos en lugar de dos. Utilizamos "pantalla" entre comillas porque visualmente el usuario las percibe como tales, aunque en realidad se trata de un programa que genera distintas porciones de un documento HTML a partir de ciertos parámetros GET o POST. Más allá de la importancia del detalle técnico, reconocemos que "pantalla" es más directo además de entendible.

Veamos una primera aproximación:

```
<!-- quinto.php (incorrecto) -->

<body>
<?php if(isset($_GET['nombre']) && isset($_GET['anio'])) { //pantalla 3
    $anios = date('Y') - $_GET['anio']; ?>

    <p>Su nombre es <?=$_GET['nombre']?> <?=$_GET['apellido']?>
    y tiene más o menos <?=$anios?> años.</p>

    <a href="cuarto.php">Volver</a>
<?php } ?>

<?php if(!isset($_GET['nombre'])) { // pantalla 1 ?>
    <form action="quinto.php" method="get">
        Su nombre: <input type="text" name="nombre" /> <br />
        Su apellido: <input type="text" name="apellido" /> <br />
        <input type="submit">
    </form>
<?php } ?>

<?php if(isset($_GET['nombre']) && !isset($_GET['anio'])) { //pant. 2 ?>
    <form action="quinto.php" method="get">
        Su año de nacimiento: <input type="text" name="anio" />
        <br />
        <input type="submit">
    </form>
<?php } ?>
</body>
```

Esta primera aproximación no es correcta. Hay una razón sencilla por la cual esta forma de resolución no funcionará: las variables de PHP se destruyen al terminar el programa. Veamos los pasos de ejecución, señalando cuáles ocurren en el cliente (C) y cuáles en el servidor (S):

1. (C) El usuario ingresa `quinto.php` en el navegador y presiona `Enter`. Esto envía un GET al servidor. Este paquete GET no contiene parámetros.
2. (S) Se ejecuta el programa e ingresa al segundo `if`, ya que se cumple que la variable `$_GET['nombre']` no existe (dijimos que la petición no tiene parámetros). No existe ésa ni ninguna otra clave en `$_GET`. Se envía un formulario a la salida (correspondiente a pantalla 1).
3. (C) El navegador recibe el formulario y lo muestra. El usuario completa los datos y presiona el botón, que genera el envío hacia `quinto.php?nombre=Juan&ape-llido=Perez`
4. (S) El servidor ejecuta otra vez el programa. Esta vez se cumple el tercer `if`, porque el parámetro por GET `nombre` sí existe (se ve en la URL) pero el parámetro `anio` no. Se envía el otro formulario a la salida (pantalla 2).
5. (C) El navegador recibe el segundo formulario y lo muestra. El usuario completa su año de nacimiento y lo envía, generando `quinto.php?anio=1970` (¡atención!)
6. (S) Se ejecuta nuevamente el programa PHP... y hemos perdido el parámetro `nombre` por lo que entramos de nuevo en el segundo `if` que muestra el primer formulario...

En el paso 4 ocurre un problema. El paso 4 corresponde a la segunda ejecución de nuestro programa en el lado servidor, cuando estamos recibiendo el nombre y apellido del usuario. Al enviar el formulario que sigue, para el año, estamos *perdiendo* el nombre y apellido, ya que no lo conservamos de ninguna forma. Como hemos dicho, las variables se destruyen cuando termina la ejecución, por lo que si no almacenamos el nombre y apellido en algún lado, el fin de la ejecución terminará destruyendo estas strings. Las variables `$_GET` y `$_POST` son como cualquier otra variable y corren esta misma suerte: son destruidas. En todos los programas hechos con cualquier lenguaje sucede esto, es decir, que las variables se destruyen cuando el programa termina. El inconveniente aquí está en que el navegador crea una *ilusión* de ejecución continua. Queremos decir que, estar frente al navegador, crea la ilusión de que nuestro programa nos está "presentando pantallas" de acuerdo a nuestros ingresos, cuando en realidad el programa de lado servidor se *ejecuta y finaliza* cada vez que hay una petición HTTP que atender. Por esta razón hemos hecho mucho énfasis en el tráfico HTTP, para poder desarticular esta ilusión y entender claramente los **momentos de ejecución** de cada programa.

Debemos entonces incorporar alguna solución en el paso 4. La solución viene de la mano de unas etiquetas que hemos mencionado antes, y que abandonamos hasta este momento: los `input hidden`.

```
<!-- quinto.php solución correcta -->
<body>
<?php if(isset($_GET['nombre']) && isset($_GET['anio'])) { // pantalla 3
    $anios = date('Y') - $_GET['anio'];
    ?>
    <p>Su nombre es <?=$_GET['nombre']?> <?=$_GET['apellido']?>
```

```

y tiene más o menos <?=$anios?> años.</p>

<a href="cuarto.php">Volver</a>
<?php } ?>

<?php if (!isset($_GET['nombre'])) { // pantalla 1 ?>
    <form action="quinto.php" method="get">
        Su nombre: <input type="text" name="nombre" /> <br />
        Su apellido: <input type="text" name="apellido" /> <br />
        <input type="submit">
    </form>
<?php } ?>

<?php if (isset($_GET['nombre']) && !isset($_GET[anio])) { // pantalla 2 ?>
    <form action="quinto.php" method="get">
        Su año de nacimiento: <input type="text" name="anio" /> <br />
        <input type="submit">

        <input type="hidden" name="nombre" value="<?=$_GET['nombre']?>" />
        <input type="hidden" name="apellido" value="<?=$_GET[apellido]?>" />
    </form>
<?php } ?>
</body>

```

Ahora el funcionamiento es el esperado. La inclusión de los `input hidden` garantiza que, al momento de enviar hacia el servidor el año (paso 5), el nombre y apellido también se reciban desde el paso anterior. Recordemos que los `input` de tipo `hidden` no tienen representación gráfica, por lo que el usuario no percibirá su presencia. De esta forma hemos encontrado una manera de persistir, entre pasos, los datos ingresados por el usuario en los pasos anteriores. Se puede verificar que el nombre y apellido figuran en la URL junto con el año de nacimiento que indica el usuario, ya que son enviados de vuelta al servidor como todo `input`.

6.7 ALGUNAS FUNCIONES ÚTILES

Si bien PHP está construido de manera modular, incorpora en su núcleo muchas funciones que se utilizan día a día. Repasemos algunas de ellas.

Para el trabajo con strings podemos contar con las siguientes funciones:

```

<?php
    $prueba = "Hola";
    echo strlen($prueba); // 4
    echo strtoupper($prueba); // HOLA
    echo strtolower($prueba); // hola
    echo substr($prueba, 2, 2); // la
?>

```

Estas funciones de string son utilizadas cotidianamente y no presentan sorpresas. De hecho, muchos nombres son replicados de la biblioteca estándar de C (`string.h`).

Es importante notar que, a diferencia de JavaScript y otros lenguajes modernos, las funcionalidades para tratamiento de **strings multibyte** se hallan en un módulo dedicado a ello y no en el núcleo del intérprete. Las **strings multibyte** son, en breve, las que se utilizan para lenguajes fuera del inglés, ocupando más de un byte por carácter.

```

<meta charset="utf-8" />
<?php
    $prueba = "aeíáéíñ";
    echo strtoupper($prueba); //AEIÁÉÍÑ
    echo mb_strtoupper($prueba, 'UTF-8'); //AEIÁÉÍÑ
?>

```

El módulo **Multibyte**, cuyas funciones están prefijadas con `mb` maneja correctamente las codificaciones de caracteres internacionales. No abordaremos en este momento el problema en toda su magnitud. Nos limitaremos a mencionar que la etiqueta `<meta>` está siendo utilizada para indicar al navegador con qué codificación trabajar y, de la misma forma, estamos mostrando que la función `mb_strlen` hace un tratamiento correcto de todos los caracteres. Contrariamente, la función del núcleo `strtoupper` no reconoce los caracteres acentuados y, por lo tanto, no los modifica. En el apéndice C tratamos más aspectos de los *encodings*.

Veamos otra función de string perteneciente al núcleo:

```

<?php
    $origen = "Programación";
    $posicion = strpos($origen, "gra");
    echo "El texto buscado se encuentra en la posición $posicion"; //3
?>

```

La función `strpos` nos permite saber si una cadena se encuentra dentro de otra y, en tal caso, en qué **posición** fue encontrada. La posición del primer carácter es 0. Pero ocurre una situación llamativa: la función `strpos` nos devuelve un entero con la posición de una cadena dentro de otra, pero retorna un `false` si no se encuentra. Experimentemos:

```

<?php
    $origen = "Programación";
    $buscar = "Pr";
    $posicion = strpos($origen, $buscar);

    if($posicion != false) echo "El texto $buscar está dentro de $origen";
    else echo "El texto $buscar NO está dentro de $origen";
?>

```



Figura 6.18: Resultado erróneo por comparar con el operador equivocado

El programa no funciona adecuadamente. La razón es que la cadena "Pr" está en la posición *cero* de la cadena "Programación", es decir, está al comienzo. Como se realiza una conversión al momento de comparar, el cero es *igualable* al falso y el `if` ejecuta el bloque `else`, que informa erróneamente que la string no se encuentra. La solución es utilizar el operador de comparación que tiene en cuenta también el **tipo** de los operandos:

```
<?php  
    $origen = "Programación";  
    $buscar = "Pr";  
    $posición = strpos($origen, $buscar);  
  
    if($posición !== false) echo "El texto $buscar está dentro de $origen";  
    else echo "El texto $buscar NO está dentro de $origen";  
?>
```

Hemos mencionado estos operadores de identidad (`==`), en relación a los de igualdad (`=`), anteriormente en este capítulo al examinar los tipos de PHP. Cuando sea necesario distinguir entre un valor *convertible a falso* y el falso mismo, será necesario el operador que también considera el tipo.

Otra función que puede resultar útil, aunque no lo parezca a primera vista, se encarga de generar **números aleatorios**:

```
<?php  
    echo rand(1,999); // retorna un entero entre 1 y 999 inclusive  
?>
```

Esta función, si bien no es confiable desde el punto de vista de la seguridad, puede sacarnos de algún que otro apuro. Decimos que no es *confiable* porque no devuelve valores *realmente aleatorios*. Esta cuestión es muy profunda y se relaciona con la seguridad informática muy estrechamente.

Otra función que suele utilizarse a menudo es la que se encarga del **formato de números**:

```
<?php  
    $num = 30.458;  
    echo number_format($num, 2); // 30.46  
?>
```

Por último, presentamos una función que, si bien no se utiliza asiduamente, es una fuente invaluable de información en ciertas etapas de los proyectos. Generalmente al principio, es utilizada cuando se precisa verificar el estado de los servidores, la versión del intérprete que se está ejecutando y los módulos que están disponibles para su uso:

```
<?php  
    phpinfo();  
?>
```

La función `phpinfo()` envía a la salida esta gran tabla (figura 6.19). Nos permite conocer la configuración del intérprete y de todos los módulos que estén habilitados en este momento. También nos informa sobre la relación existente entre el intérprete de PHP y el servidor HTTP, en nuestro caso Apache, además de las versiones.

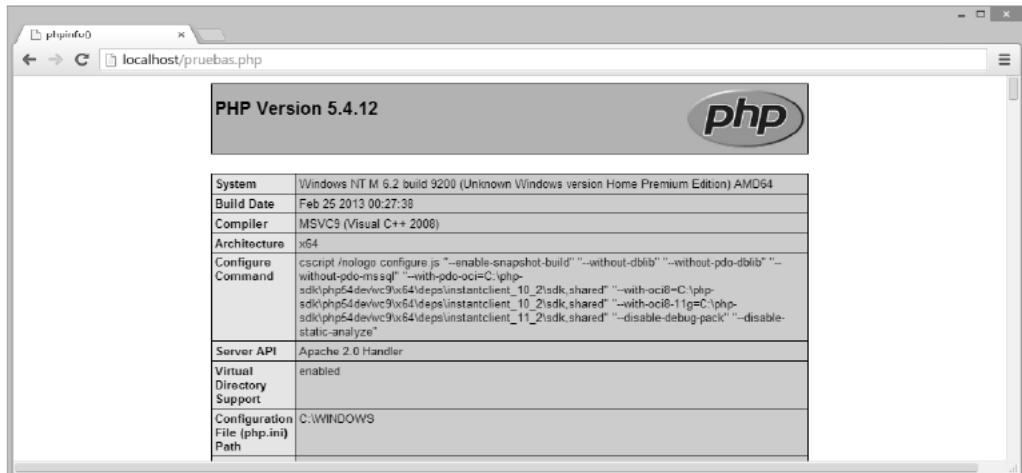


Figura 6.19: Salida de la función *phpinfo*

6.8 DIVISIÓN DEL CÓDIGO EN PARTES

Si bien no es necesario para la realización de los ejercicios propuestos para el presente capítulo, puede ser útil conocer de qué manera pueden separarse las partes de un programa PHP a medida que crece la extensión y se transforma en una aplicación web.

Una forma históricamente probada de mantener fácilmente distintas partes de una aplicación es la utilización de **funciones**. Su declaración en PHP es muy similar a JavaScript con la salvedad de que siempre las variables se indican con \$, siendo los parámetros de la función un caso más.

```
<?php
    function prueba($param1, $param2) {
        return $param1 + $param2;
    }

    var_dump( prueba(2,3) ); // (int) 5
?>
```

Es importante notar que, a diferencia de JavaScript y de C, las variables globales no son directamente accesibles dentro de las funciones. Recordemos que en PHP no se realiza una declaración explícita de las variables sino que se setean con su mera definición. Dentro de las funciones, la primera aparición de una variable la declarará como local. Esto puede ser extraño para quienes provienen de otros lenguajes.

```
<?php
    function prueba() {
        global $b;
        $a++; // crea una $a local!
        $b++; // afecta la $b global
    }

    $a = 3;  $b = 3;

    prueba();
```

```
    var_dump($a); //3  
    var_dump($b); //4  
?>
```

El operador `global` le solicita al intérprete que no cree una variable local sino que asuma la declaración global. De no indicar esto, todas las variables de la función serán locales. Existe una gran excepción a esta regla que es el conjunto de las variables **superglobales**. Son aquellas variables que son globales siempre, aunque no se indique el `global`. Estas variables superglobales son `$_GET`, `$_POST` y similares, es decir, las variables automáticas que inicializa el intérprete con información proveniente del entorno.

Es habitual que, a medida que nuestra aplicación crece en escala tengamos cada vez más funciones, de manera que se va conformando una librería de funciones. También es normal que ciertas líneas de código deban incluirse en todos los programas de una misma aplicación antes de comenzar el procesamiento, algo comúnmente llamado *bootstrapping*. Existen cuatro construcciones que realizan inclusión de código externo, de la misma manera que el preprocesador de C con su directiva `#include`. Estas construcciones del lenguaje nos permiten separar el código que se repite en un archivo externo e incluirlo en cada programa que escribimos.

```
<?php  
    include 'misfunciones.php';  
    //... resto del programa  
?>
```

Podemos imaginar que la construcción `include` hace "copiar/pegar" del código que se encuentra en otro archivo, causando el mismo efecto que si ese código estuviese escrito aquí. De hecho, éste puede tener partes interpretables y partes estáticas, y el intérprete ejecutará o mandará al *buffer* de salida de la misma forma que hemos visto hasta aquí. Lo siguiente es muy habitual en la práctica profesional para evitar la repetición:

```
<body>  
    <!-- documento, etc... -->  
    <?php include 'pie.html' ?>  
</body>
```

Existe también la construcción `require`, que realiza exactamente la misma tarea pero que se diferencia en el error. Si el archivo externo no se encuentra detendrá la ejecución. En cambio, `include` emitirá un error pero continuará la ejecución. También contamos con `include_once` y `require_once`, que tienen la misma finalidad pero garantizan una sola inclusión aunque la repitamos, de manera de no redefinir funciones varias veces, lo que es un error fatal.

6.9 BIBLIOGRAFÍA

Una vez más recomendaremos acudir a las fuentes originales. El manual oficial de PHP se puede encontrar en php.net/manual. La gran mayoría se encuentra también en castellano, aunque la versión en inglés siempre se encuentra más actualizada.

PHP es un lenguaje que ha evolucionado muchísimo si comparamos con versiones anteriores. En los últimos tiempos, además, la velocidad a la que se incorporan características se ha ido acelerando. Debido a esto, mucho material que se encuentra en Internet está, probablemente, desactualizado. En función de esto es que recomendamos acudir al manual oficial siempre que sea posible.

Además de esto, la documentación oficial es amena de leer e incluye muchos ejemplos y artículos pensados específicamente para el aprendizaje.

6.10 EJERCICIOS

1) Dados estos arrays:

```
$marcas = array(1=>"Ford", 2=>"Renault", 5=>"Ferrari");
$provincia = array(1=>"CABA", 7=>"Cordoba", 2=>"Misiones");
$detalles = array (20=>"Aire", 30=>"ABS", 33=>"Gas");

$autos = array();
$autos[25] = array( "marca"=>1, "prov" => 2, "det" => array(20) );
$autos[4] = array( "marca"=>2, "prov" => 1, "det" => array(20,30) );
$autos[14] = array( "marca"=>1, "prov" => 2, "det" => array(30) );
$autos[1] = array( "marca"=>5, "prov" => 7, "det" => array(20,30,33) );
```

En un formulario se seleccionará una marca y una provincia (dos `select`), que deben surgir de los arrays. Al hacer submit se deben ver los vehículos que pertenecen a la marca y provincia elegidos. Para los vehículos que coincidan con este criterio de búsqueda se muestran sus datos (marca y provincia), y se listarán las características que poseen (nótense la clave `det` y el array `$detalles`).

2) Agregar al ejercicio anterior una opción “indistinto”, que se puede elegir en ambos `select`. Si el usuario elige esa opción, el criterio (marca y/o provincia) no será tenido en cuenta para buscar los autos. Notar que si seleccionamos “indistinto” en ambos `select` deben verse todos los vehículos que existan. Esta opción “indistinto” no está en el array de datos.

3) Un formulario tiene tres inputs de texto para: pesos, dólares y tasa de cambio. El usuario completa dos de estos y envía el formulario. Con PHP debe calcularse el número faltante. El usuario siempre indicará dos de los tres datos.

4) Se desea presentar al usuario una operación matemática al azar, formada por dos números y un operador. Los números irán del 1 al 9 y la operación podrá ser suma, resta o multiplicación. Utilizando la función `rand`, mostrar al usuario una operación tipo `1+3`, `2-4`, `3*7`, etc., al azar. El usuario completará el resultado en un input de texto y enviará el formulario. Con PHP se verificará si el resultado de la operación es correcto o no.

7. MySQL

MySQL es un motor para la gestión de **bases de datos relacionales** que se utiliza muy a menudo en conjunto con PHP. Si bien son dos productos desarrollados por distintos equipos, históricamente se ha constituido una sinergia que continúa en el día de hoy.

En sus orígenes, MySQL es un producto de código abierto que comenzó a desarrollarse en Suecia y cuyas primeras versiones surgen en el año 2000. Hace unos años fue adquirida por Oracle, una corporación cuyo producto estrella es también un motor de base de datos, lo que generó muchas dudas acerca de su continuidad como software libre. La distribución estable actual es la rama 5.6.

MySQL está disponible con licencias libres y también privativas, dependiendo de su uso. Existen muchos motores de bases de datos con las mismas características, e incluso algunos de ellos han sido superiores en características durante varios años. Podemos mencionar PostgreSQL y el desprendimiento reciente MariaDB.

7.1 PERSISTENCIA DE INFORMACIÓN

Las **bases de datos** son soluciones tecnológicas al problema de la **persistencia** de los datos de una aplicación. Mencionemos otras formas de almacenar información:

- Memoria RAM: son extremadamente veloces pero volátiles, por lo que se descartan como almacenamiento de largo plazo. Además, suelen ser un recurso escaso y el control de acceso es muy difícil de implementar. Algunas soluciones de almacenamiento, como *memcache*, trabajan sobre esta base.
- Archivos de disco: es una tecnología bien conocida y con buen desarrollo a lo largo de varias décadas. Es un recurso relativamente barato y su disponibilidad no suele ser un problema. Su principal contra es la escasa velocidad.
- *Storage* de terceros: algunas empresas como Amazon o Dropbox poseen unidades de negocios dedicadas a comercializar espacio de almacenamiento. Si bien los costos pueden ser reducidos frente a otras opciones, el tiempo de latencia del acceso por red puede ser un factor de descarte.

Veamos ahora cuáles son los criterios de diseño de un motor de bases de datos moderno, en función de comparar y elegir la mejor opción para nuestros proyectos:

- Grandes volúmenes de datos: las bases de datos están diseñadas para manejar tablas con millones de filas, que se traducen en un volumen de información del orden de los gigabytes y terabytes.
- Guardar y recuperar eficientemente: por supuesto, de nada serviría un producto que nos garantice un volumen de almacenamiento enorme si luego los tiempos de acceso son muy extensos. No sólo el tiempo de almacenamiento tiene que ser aceptable, sino también el de recuperación. Muchas veces se piensa en el almacenamiento de información pasando por alto que el objetivo último de este almacenamiento es recuperar los datos posteriormente. No es

- raro que el diseñador de la base de datos deba elegir entre velocidad de escritura y velocidad de lectura.
- Concurrencia: los motores de bases de datos están diseñados para soportar el acceso concurrente a la información, tanto en lectura como en escritura. En el contexto del desarrollo web ésta es una consideración clave ya que la concurrencia es algo inevitable.
- Control de acceso: es importante poder controlar los usuarios y roles en relación a la información. El control de permisos está incorporado directamente en las bases de datos.

Por supuesto que estas ventajas vienen acompañadas de una contra: es mucho más difícil trabajar con bases de datos que sin ellas. Este capítulo es testimonio de ello.

En otras palabras, si nuestro desarrollo no posee la escala para las cuales las bases de datos fueron pensadas, estaremos trabajando con un producto sobredimensionado que nos traerá más problemas que beneficios. Existe una base de datos que posee características en común con los motores como MySQL y también con los archivos de disco, llamada **SQLite**. Este producto es muy adecuado para sitios web de pequeña escala y es muy recomendable interiorizarse en sus particularidades.

7.2 CLIENTE / SERVIDOR

El motor MySQL está diseñado aplicando la arquitectura **cliente-servidor**, al igual que HTTP. Esto quiere decir, de la misma forma, que tendremos dos programas cumpliendo los roles de **cliente** y de **servidor**, y entre ellos cierto **tráfico**. El servidor escucha por defecto en el puerto TCP 3306.

A diferencia de HTTP, no estudiaremos los detalles de este tráfico ya que no nos aporta conceptos útiles para entender el funcionamiento de la tecnología. Solamente diremos que se trata de datos binarios comprimidos. Este tráfico es propio de cada motor y no forma parte de un protocolo, como en el caso de HTTP.

El servidor de base de datos que utilizaremos es MySQL, que se encuentra en el mismo paquete preconfigurado que hemos utilizado para trabajar con PHP: WampServer. Este paquete, al igual que hemos considerado con Apache, nos permite instalar fácilmente el motor de bases de datos sin necesidad de complejas configuraciones. Existe un grupo de profesionales que se dedica exclusivamente a configurar y calibrar servidores de bases de datos para mejorar la performance, la seguridad y la confiabilidad.

En cuanto a los clientes, existen bastantes empresas que los producen, e incluso algunos de ellos funcionan con motores de distintos fabricantes simultáneamente. Como hemos dicho, el tráfico está definido por cada servidor de base de datos, por lo que debemos encontrar un cliente específico para el motor con el que vayamos a conectar. Existen clientes en forma de **programas de escritorio**, es decir, que se instalan en el sistema operativo y se utilizan para conectar y trabajar sobre una base de datos remota.

Otra posibilidad, que será la que elegiremos, es utilizar un **cliente web**, en nuestro caso, llamado **phpMyAdmin**. Este cliente está programado con un lenguaje de lado servidor,

puntualmente PHP. Este tipo de programas, como ya hemos visto, se ejecutan en el *servidor* y en el caso de phpMyAdmin, a su vez, cumplirá el papel de *cliente* de base de datos. Por lo tanto, tenemos un programa de lado servidor (HTTP) que, en su rol de cliente, conecta con otro servidor (MySQL). Esquemáticamente, podríamos ilustrarlo así:

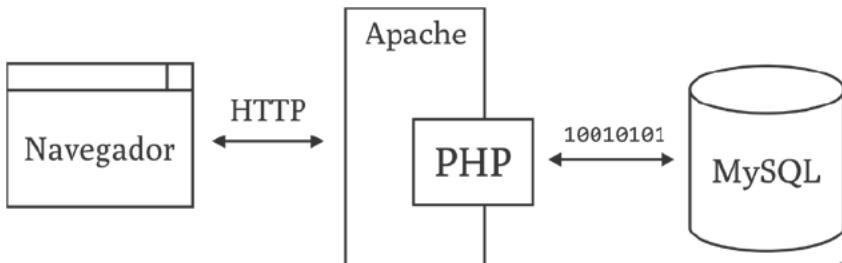


Figura 7.1: Esquema de clientes y servidores al utilizar phpMyAdmin

Si accedemos a la URL <http://localhost/phpmyadmin> debemos ver la pantalla inicial de phpMyAdmin, de esta forma:



Figura 7.2: Pantalla inicial de phpMyAdmin

El software empaquetado en WampServer ya está configurado para que tal URL ejecute el programa phpMyAdmin y que éste conecte con el servidor MySQL que trae el mismo paquete. Es interesante notar que, dentro de nuestra carpeta raíz de Apache (`C:\wamp\www`) no encontramos ninguna carpeta llamada `phpmyadmin`. Esto pone de manifiesto que Apache está configurado, de alguna manera, para que tal URL ejecute un programa que se encuentra en otro lado, fuera de la raíz. Esta configuración de Apache se denomina *alias*.

7.3 USUARIOS Y PRIVILEGIOS

Una gran diferencia que existe entre la arquitectura cliente-servidor de HTTP y de MySQL es que, en este último, no existe el acceso anónimo. Cuando utilizamos un navegador para enviar peticiones a un servidor HTTP, por lo general no precisamos enviar ninguna información sobre quiénes somos. Salvo casos puntuales, navegar por la web

no es una tarea que precise que nos autentiquemos: sencillamente navegamos. En cambio, el servidor de MySQL *siempre* nos pedirá nuestras **credenciales** para aceptar nuestras peticiones. Con otras palabras, no hay operaciones anónimas sobre el servidor de base de datos.

Cuando se establezca la conexión con MySQL siempre se deberá proveer de un usuario y contraseña. Estos usuarios son creados en el servidor por el usuario administrador y poseen distintos privilegios para realizar acciones en el servidor. Un usuario, por ejemplo, puede tener permisos para leer información, otro para guardar información y un tercero puede tener ambos. Existe un superusuario especial que tiene todos los permisos: se denomina usuario **root**.

Es común que en el entorno de desarrollo se conecte a la base directamente con este usuario *root*. Esto quiere decir que tendremos todos los privilegios sobre la base de datos. En el ambiente de **desarrollo** esto es deseable porque acelera el propio tiempo de desarrollo, donde es común realizar modificaciones y operaciones que en el entorno de **producción** serán sólo accesibles para los administradores. Así como mencionamos que el programador suele trabajar en una copia local del proyecto, de la misma manera conecta con un servidor de base de datos local, sobre el que tiene todos los privilegios.

El usuario *root* de MySQL tiene por defecto una contraseña vacía, por lo que si phpMyAdmin nos pregunta el usuario y contraseña para establecer la conexión le indicaremos *root* y ninguna contraseña.

7.4 DISEÑO RELACIONAL

MySQL es un motor de bases de datos relacionales. Detallemos un poco de qué se trata este aspecto **relacional**. No es nuestra intención presentar toda la teoría del modelo relacional, sino los conceptos fundamentales que nos permitirán continuar de una manera práctica con el uso de la tecnología.

El **modelo relacional** es una teoría matemática desarrollada alrededor de 1960. Está muy relacionada con la lógica y la teoría de conjuntos, y su objetivo original es aportar a la informática principios sólidos que surjan de una teoría matemática rigurosa. Las bases de datos relacionales son piezas de software que aplican muchos de los conceptos provenientes de este modelo, aunque no son materializaciones perfectas de él. En lugar de detallar la teoría detrás de este modelo, examinemos las características que sí tienen las bases de datos concretas que, además, resultan fáciles de comprender y aplicar.

El primer paso para *diseñar* la base datos es pensar en las **entidades y relaciones** de nuestro **dominio**. El dominio es el conjunto de conceptos y hechos que definen el problema que queremos solucionar con nuestro programa. Las entidades y sus relaciones conforman el **modelo de datos**, que representa la información de la realidad que, de alguna manera, necesitamos en nuestra aplicación. El mundo real posee infinita cantidad de información que, de acuerdo a la necesidad, seleccionaremos e incluiremos en nuestro modelo de datos. Para ilustrar más claramente, supongamos el dominio de las *veterinarias* y que el problema a resolver es el almacenamiento de las mascotas que se atienden en dicho lugar y la información de sus dueños. Diseñaremos un modelo de datos que contiene dos entidades: "*mascota*" y "*dueño*" que se encuentran relacionadas

de la forma: cada mascota "*tiene un*" dueño. Podemos ilustrar este diseño con el siguiente **diagrama entidad-relación**:



Figura 7.3: Diagrama Entidad-Relación con dos entidades del modelo

Cada entidad tendrá además **atributos**. Por ejemplo, las mascotas tendrán un nombre, una raza, una edad; los dueños tendrán un DNI, un saldo adeudado, una dirección, etcétera. La información que se incluye o se deja fuera del modelo de datos está directamente relacionada con los problemas que tiene que resolver nuestro sistema.

Las bases de datos son agrupaciones de **tablas**. Las tablas poseen **filas y columnas**, de una manera similar a una hoja de cálculo. Básicamente, lo que haremos será crear una tabla para cada entidad. Esta tabla tendrá una columna por cada atributo de esa entidad. El conjunto de tablas es llamado base de datos. Veamos el ejemplo sencillo de la veterinaria:

Mascotas		Dueños	
Nombre	Dueño	Dueño	Nombre
Bobby	1	1	Juan
Fatiga	2	2	Pedro
Doggy	2		

Figura 7.4: Dos tablas relacionadas, con datos de dos entidades

La flecha que hemos dibujado representa la *relación* que hay entre las dos entidades. Es claro que el dueño identificado con el número 2 tiene dos perros, mientras que la otra persona tiene uno solo. Esto, que nos resulta inmediatamente claro al verlo, quiere decir que las dos columnas llamadas "Dueño" en las dos tablas son la *misma* información.

El resultado es un **diseño relacional**. Esta forma de estructurar los datos tiene muchas ventajas. Por ejemplo, es muy fácil saber cuántas mascotas y cuántas personas acuden a la veterinaria. También es muy fácil cambiar el nombre a un dueño. Si esta información fuera almacenada en una sola tabla gigante, estas operaciones exigirían recorrer la tabla fila por fila. El objetivo del diseño relacional es evitar este recorrido de la tabla completa, además de garantizar que la información no se repita ni pierda coherencia.

Mascotas		Dueños		Veterinaria	
Nombre	Dueño	Dueño	Nombre	Mascota	Dueño
Bobby	1	1	Juan	Bobby	Juan
Fatiga	2	2	Pedro	Fatiga	Pedro
Doggy	2			Doggy	Pedro

Figura 7.5: Diseño relacional a la izquierda, diseño no relacional a la derecha

El motor de base de datos puede configurarse para que no nos permita eliminar un dueño si hay mascotas que le pertenecen, ya que esto dejaría a las mascotas en un estado erróneo. Esta precondición de las bases de datos se llama **integridad referencial**. Otro ejemplo de pérdida de la integridad sería ingresar una mascota indicando un dueño con un número que no existe en la tabla de dueños.

Diremos que las filas de la tabla son sus **datos** y las columnas su **estructura**. Las filas son también llamadas **registros** y las columnas, **campos**. Concretamente, el motor MySQL almacena la información en archivos de disco, por lo que las tablas son solamente una representación gráfica que nos sirve a las personas para pensar el diseño. Por lo tanto, es importante notar que esta visualización de *datos en filas y estructura en columnas* no es más que una convención. Algunas partes de la interfaz de phpMyAdmin están diagramadas en el otro sentido, por lo que es muy importante distinguir correctamente entre el concepto de *dato* y *estructura*. Los datos, por ejemplo las mascotas, se irán acumulando en la tabla, pero su estructura será siempre la misma: ya sean mil o un millón, *todos* los perros tendrán *siempre* nombre y dueño.

Otro concepto fundamental de las bases de datos es el de las **claves**. Sin entrar en detalles teóricos, diremos que la **clave primaria** es una columna que identifica de manera única a cada fila de la tabla. El ejemplo más claro es el DNI que tiene cada ciudadano. El DNI es el número que lo identifica de manera que nadie tiene dos DNI, todos tenemos exactamente uno y el mismo DNI no es asignado nunca más de una vez. Podemos decir entonces que el DNI nos identifica de manera única. De la misma forma, cada tabla tendrá siempre una clave primaria. En realidad, la clave primaria puede estar conformada por *varias* columnas, pero esto no sucede habitualmente en la práctica.

Además, las tablas pueden tener (o no) **claves foráneas**. En resumidas cuentas, las claves foráneas son los atributos que, en otra tabla, son claves primarias. En nuestro ejemplo de la veterinaria, las mascotas poseen una clave foránea que indica quién es su dueño. Este número es la clave primaria de la tabla de dueños. La flecha que hemos dibujado como la relación entre ambas entidades tiene las claves en sus extremos: de un lado la primaria y del otro la foránea. Es muy común abreviar con las iniciales **PK** (*primary key*) y **FK** (*foreign key*). La tarea de definir la estructura de las tablas y sus relación por medio de claves se denomina **normalización**.

Mascotas

Nombre	Dueño
Bobby	1
Fatiga	2
Doggy	2

PK

FK

Dueños

Dueño	Nombre
1	Juan
2	Pedro

Figura 7.6: Claves primaria y foránea

7.5 TIPOS DE DATOS

Un aspecto importante sobre el diseño de las tablas es que sus columnas tienen **tipo**. Los tipos son propios de cada motor. Los más utilizados en MySQL son:

- **INT**, **FLOAT**, **DECIMAL**: datos numéricos enteros, de punto flotante y de punto fijo.
- **VARCHAR**: datos alfanuméricos de largo variable con un máximo de 255 caracteres.
- **TEXT**: datos alfanuméricos que no se almacenan físicamente con cada fila. No tienen un límite, por lo que se utilizan para textos extensos.
- **DATE**, **DATETIME**: tipos para fechas y horas. El motor es muy eficiente para realizar estos cálculos y posee muchas funciones útiles para ellas.
- **BLOB** (*binary large object*): para almacenar datos binarios, por ejemplo, una imagen.

La mayoría de los tipos tienen además la posibilidad de indicar su **tamaño**. Por ejemplo, los tipos **TINYINT**, **SMALLINT**, **MEDIUMINT**, **INT** y **BIGINT** establecen todos un tipo entero, pero de 1, 2, 3, 4 y 8 bytes respectivamente. Estas indicaciones son importantes para la optimización del espacio de almacenamiento que realizan los motores. Por lo demás, este es un claro ejemplo de aspectos de la implementación de las bases relationales que no guardan relación con el modelo matemático original.

Las columnas numéricas además pueden indicarse como **signadas** o no-signadas. Como en la mayoría de los lenguajes de programación, esto modifica el rango de números representables y dependerá de los datos que necesite nuestro programa. Otro aspecto interesante de las columnas con datos numéricos enteros es que pueden marcarse como **autoincrementales**. Esto tendrá como consecuencia que, al crearse una fila en la tabla, será el motor quien asigne el número automáticamente incrementando en una unidad cada vez. Este comportamiento es muy conveniente y lo utilizaremos para casi todas las claves primarias en nuestras entidades.

Existe en las bases de datos un valor especial que representa la *ausencia* de información. Es el tipo **NULL**. Se utiliza para indicar que no se conoce el dato y su presencia altera el comportamiento del motor cuando realizamos ciertas consultas. Es importante notar que un valor NULL no es lo mismo que un *cero* o una *string vacía*. El NULL no tiene tipo e indica, sencillamente, que el dato es *desconocido*. Los campos de una tabla, es decir, sus atributos, pueden declararse como **NULL** o **NOT NULL**, lo que indica a la base si es *possible*

que ese dato sea desconocido o si es preciso que el dato exista al crear la fila. Las claves primarias son un ejemplo claro de campo `NOT NULL`, ya que siempre debe existir cuando creamos un registro. Esta propiedad del campo se determina de acuerdo a las necesidades de la aplicación. En nuestro ejemplo podríamos preguntarnos ¿puede crearse una mascota desconociendo su dueño? Si la respuesta es sí, el campo se marca `NULL`. En caso contrario será `NOT NULL`.

7.6 LENGUAJE SQL

Las bases de datos utilizan un lenguaje llamado SQL (*Structured Query Language*, lenguaje estructurado de consultas) para recibir **consultas**. Todas las operaciones que pueden solicitarse a la base, sean de lectura de datos, modificación, creación de tablas, creación de bases o *cualquier* otra operación, deben realizarse utilizando un cliente y, por medio de éste, enviar una consulta al servidor. Incluso cuando se trabaja de manera local es necesario utilizar un cliente para conectar a un servidor local y enviar una consulta.

Todas las consultas se escriben en lenguaje SQL, por lo que este lenguaje es ineludible al trabajar con bases de datos. De hecho, los comandos que realizan tareas administrativas como reiniciar el servidor, configurar los tipos de motor disponibles, alterar el funcionamiento interno del caché, etc., también se ejecutan de esta forma.

Existen **estándares** denominados ANSI SQL, y diferenciados por el año en que fueron aprobados. El estándar más aceptado actualmente es el SQL99, que define no solamente el lenguaje de consultas sino también aspectos avanzados de los servidores como *triggers* (disparadores) y *stored procedures* (procedimientos almacenados). Si bien, idealmente, una consulta escrita para un motor de base de datos debería ser totalmente portable a otro motor, esto es realmente difícil de lograr. Algunas cláusulas muy utilizadas en el SQL de la práctica real son propietarias de cada fabricante, y esto se relaciona con las demandas de optimización constante que enfrentan estas aplicaciones. Incluso algunas consultas pueden escribirse de cierta forma para ser más eficientes en versiones específicas del mismo motor. Todo esto contribuye a que las consultas portables sean un objetivo que directamente se deja de perseguir, para intentar la portabilidad en capas superiores de la aplicación. Estas capas generan consultas SQL de manera automática, partiendo de otro lenguaje intermedio. Un ejemplo de esto es *Doctrine*, una capa de abstracción de base de datos para PHP que se basa en el producto *Hibernate* para la plataforma Java.

El lenguaje SQL es un lenguaje orientado a **resultados**. La idea detrás de esto es que la consulta exprese el conjunto de datos que queremos obtener y no *cómo* obtenerlo. El motor de base de datos recibirá la consulta y armará un **plan de ejecución**. Este plan determina cómo obtener la información que se solicita. Utilicemos un ejemplo para ilustrar la diferencia con un lenguaje imperativo:

- Lenguaje imperativo: "*inicializar una colección vacía, recorrer fila por fila, agregar a la colección las personas mayores de 18 años, retornar la colección*"
- Lenguaje SQL: "*obtener las personas con edad mayor a 18 años*"

Como programadores estamos acostumbrados a pensar los pasos necesarios para obtener un resultado. Este lenguaje, que nos promete poder expresar los resultados deseados y delegar en la base la tarea de "pensar" la forma de obtención, se torna bastante complejo a medida que la consulta involucra a más tablas. Cuando la consulta se complejiza de esa manera, los límites entre orientación a resultados e imperativo se tornan cada vez más grises.

El lenguaje SQL distingue tres tipos de consultas:

1. DML, *Data Manipulation Language* (manipulación de datos): recuperación, creación, modificación y borrado de filas de una tabla. Son las más habituales.
2. DDL, *Data Definition Language* (definición de datos): creación de tablas, definición de columnas, agrupamiento en bases.
3. DCL, *Data Control Language* (control de datos): definición de usuarios y privilegios sobre los datos de la base.

7.7 USO DE PHPMYADMIN

Antes de explorar la utilización del lenguaje SQL, preparamos una base de datos de ejemplo para poder realizar consultas sobre sus tablas. Como ya hemos dicho, utilizaremos phpMyAdmin como cliente.

El primer paso será crear la base de datos que, recordemos, es un conjunto de tablas. Cada motor de base de datos puede almacenar muchas bases de datos, cada una con muchas tablas en su interior. Ingresando a <http://localhost/phpmyadmin>, nos dirigiremos a la pestaña "Bases de datos" y crearemos una base llamada "empresa":



Figura 7.7: Lista de bases de datos existentes y formulario para creación de una nueva

Una vez creada la base de datos estamos listos para crear sus tablas, haciendo click en el nombre "empresa" que se acaba de añadir a la lista de bases de datos existentes. Es importante observar la parte superior de la pantalla, donde phpMyAdmin nos indica el lugar en donde estamos ubicados. El camino `localhost >> empresa` nos permite saber cuál es el servidor y sobre qué base estamos trabajando. Crearemos una tabla llamada "cargos", que tiene dos columnas:

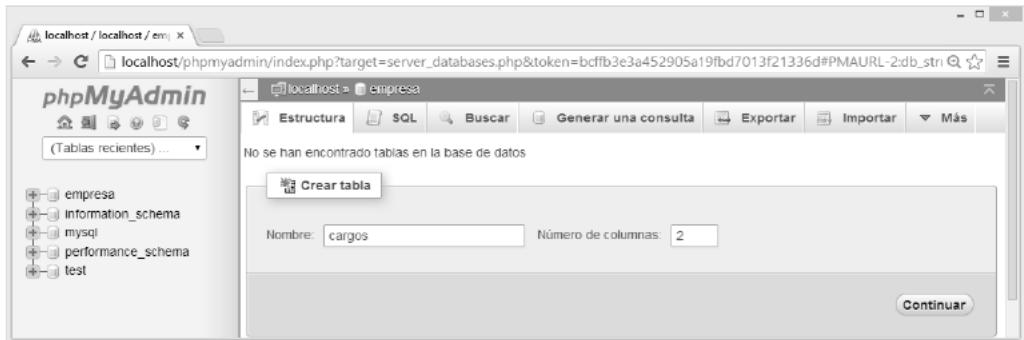


Figura 7.8: Primer paso para creación de una tabla

La indicación "Número de columnas" es meramente orientativa para el armado del formulario siguiente. No hay inconveniente en indicar columnas de más ya que podemos dejar en blanco las que no utilicemos. Asimismo, si nos faltan columnas podemos agregarlas en la pantalla que sigue. Veamos cómo se indican las columnas de la tabla:

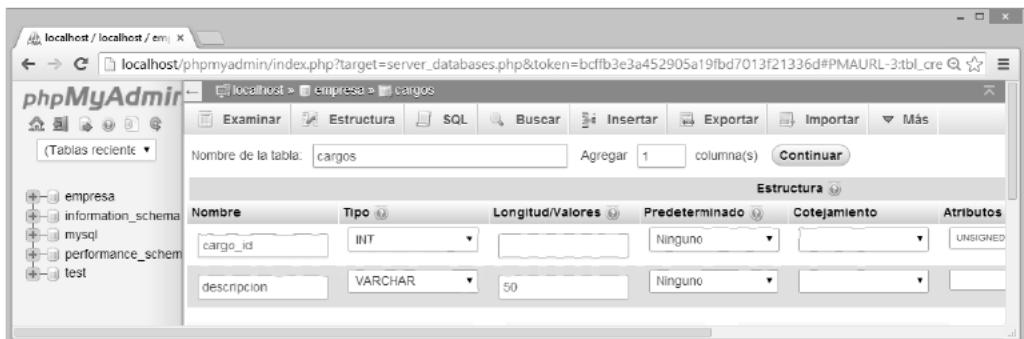


Figura 7.9: Formulario para creación de una tabla

Aquí es importante recordar que estamos creando una tabla y, para ello, necesitamos indicar qué columnas tiene y qué características tienen estas columnas. Sin embargo, phpMyAdmin nos presenta una interfaz con filas. Por eso hemos dicho antes que es importante tener clara la distinción entre datos y estructura. En este momento estamos determinando la *estructura*, aunque la interfaz gráfica esté materializada en forma de filas. Creemos los siguientes campos:

- **cargo_id**: la clave primaria. Utilizamos tipo **INT**, atributo **UNSIGNED**, dejamos sin marcar la casilla "NULL" para que sea **NOT NULL** y marcamos la opción **AUTO_INCREMENT**. De la lista "Índice" elegimos **PRIMARY**.
- **descripcion**: Declaramos tipo **VARCHAR** y en longitud indicamos **50**

Es recomendable, para evitar problemas innecesarios, evitar los caracteres extraños en los nombres de los campos, tales como espacios, acentos y eñes. Distintas versiones de MySQL tratan de diferente forma estas situaciones, incluso variando el tratamiento de mayúsculas y minúsculas en algunos casos. Para no generar situaciones inesperadas y por preferencia personal, utilizaremos siempre minúsculas.

Cuando presionemos "Guardar", el cliente enviará una consulta de creación de tabla hacia el servidor. Luego de esto, el cliente nos mostrará la información sobre la tabla que se acaba de crear que, obviamente, no tendrá todavía filas pero estará lista para poder recibirlas.

The screenshot shows the 'Estructura' (Structure) tab of the 'cargos' table in phpMyAdmin. The table has two columns: 'cargo_id' (int(10)) and 'descripcion' (varchar(50)). The 'cargo_id' column is marked as 'Primaria' (Primary). The 'descripcion' column has a value 'Gerente' entered in its input field. There are buttons for 'Continuar' (Continue) and 'Añadir' (Add).

Figura 7.10: Estructura de la tabla recién creada

Para crear filas, es decir *datos*, en esta nueva tabla, utilizamos la pestaña "Insertar", que nos permite dar de alta dos filas por vez. Crearemos dos cargos, uno llamado "Gerente" y otro "Empleado". El campo `cargo_id` lo dejaremos en blanco, ya que lo hemos marcado como autoincremental y será el motor quien determine los sucesivos números. Este campo es la clave primaria y no tiene una representación en la realidad. Con otras palabras, es un dato *inventado* para el tratamiento de la fila en la base de datos.

The screenshot shows the 'Insertar' (Insert) tab of the 'cargos' table in phpMyAdmin. Two rows are being inserted. The first row has 'descripcion' set to 'Gerente'. The second row has 'descripcion' set to 'Empleado'. Both rows have their 'cargo_id' fields left empty. There are buttons for 'Continuar' (Continue) and 'Añadir' (Add).

Figura 7.11: Formulario para inserción de datos (filas) en la tabla

La tabla `cargos` tendrá entonces solamente dos filas. Esto cambiará solamente si la empresa desea crear nuevos cargos en el futuro. La clave primaria se ha cargado de manera automática con "1" y "2" debido al campo autoincremental:

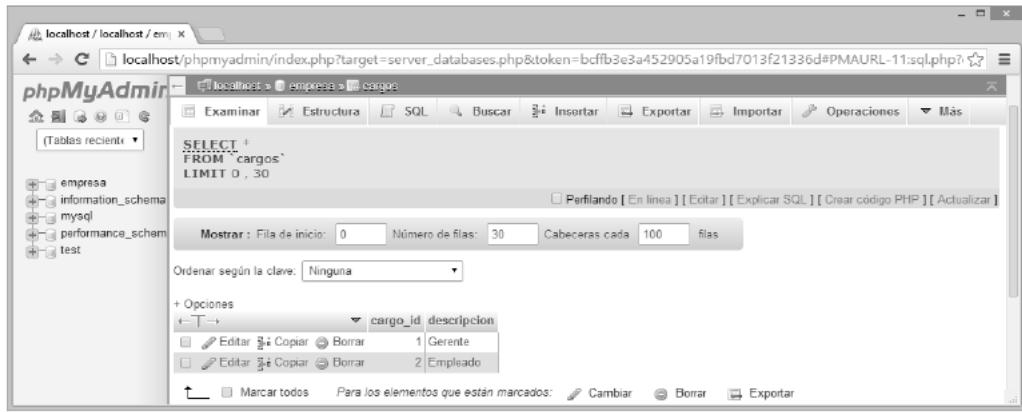


Figura 7.12: Clave primaria (`cargo_id`) asignada automáticamente por ser autoincremental

Siempre que estemos viendo una tabla podemos utilizar las pestañas "Examinar" y "Estructura" para acceder a los *datos* y la *estructura*, respectivamente. Tanto los datos como la estructura pueden cambiarse. La información que vemos en el cliente está almacenada en el servidor de base de datos y, por lo tanto, es persistente.

Si hacemos click en el nombre de la base (`empresa`) en la parte superior volveremos al listado de tablas, donde encontraremos nuestra recientemente creada `cargos`.

Nuestro ejemplo representa los datos de una supuesta aplicación para administrar los adelantos de sueldo que piden los empleados de una empresa. Los empleados se acercan a una oficina y solicitan un adelanto, de manera que se necesita almacenar quién pidió cuánto dinero y en qué fecha. Crearemos dos tablas más para poder realizar consultas más completas:

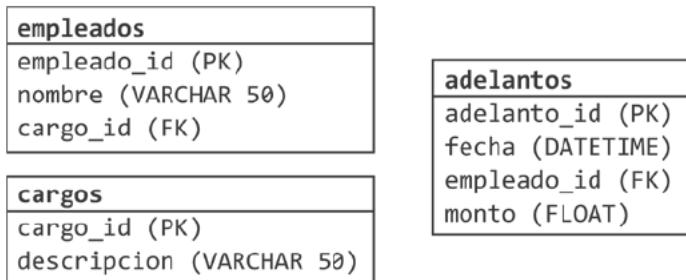


Figura 7.13: Base de datos "empresa"

Y colocaremos los siguientes datos mediante "Insertar":

empleados			adelantos			
empleado_id	nombre	cargo_id	adelanto_id	fecha	empleado_id	monto
1	Pepe	1	1	2013-5-1	1	100.0
2	Juan	2	2	2013-6-1	2	50.0
3	Maria	1	3	2012-5-1	3	300.0
4	Ana	2	4	2012-5-5	4	100.0
			5	2013-1-2	1	1000.0
			6	2013-8-5	2	200.5
			7	2012-7-1	3	10.0
			8	2013-9-1	4	650.0
			9	2013-9-2	4	80.0

cargos	
cargo_id	descripcion
1	Gerente
2	Empleado

Figura 7.14: Datos de ejemplo para insertar en la base "empresa"

Es importante verificar adecuadamente que las claves primarias posean el atributo **AUTO_INCREMENT**, ya que si no lo tuvieran fallarían las inserciones de las filas al no poder determinarse la clave primaria. Recordemos que las claves primarias deben ser únicas para cada fila.

7.8 CONSULTAS DE SELECCIÓN

El tipo de consulta más realizado es el de **selección**, perteneciente al grupo DML. La selección nos devuelve filas existentes en la base de datos. Para la ejecución de estas consultas utilizaremos la pestaña "SQL".

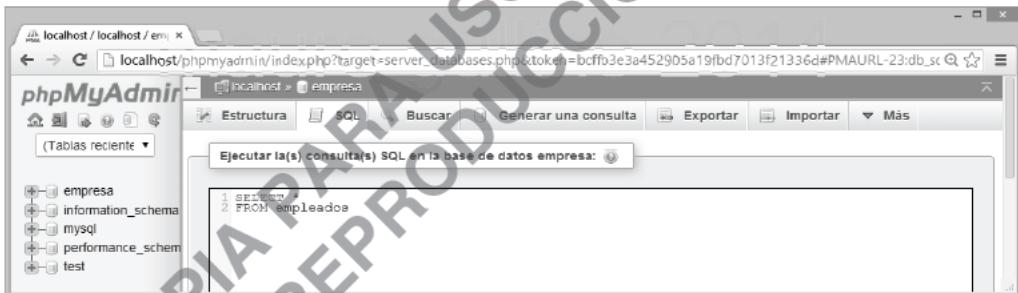


Figura 7.15: Pestaña para ejecución de consultas SQL

```
SELECT *
FROM empleados
```

Al presionar "Continuar" veremos el resultado de ejecutar la consulta en el servidor de bases de datos (figura. 7.16).

Esta consulta SQL tiene dos **cláusulas**:

- **SELECT**: indica las columnas que queremos en el resultado
- **FROM**: indica de dónde se deben tomar las filas

Es importante conocer el significado específico de cada cláusula para poder escribir y leer consultas cada vez más complejas. Las cláusulas de SQL tienen un orden específico y se acostumbra escribirlas en distintas líneas, aunque esto es meramente un estilo.

	empleado_id	nombre	cargo_id
<input type="checkbox"/> Editar	1	Pepe	1
<input type="checkbox"/> Editar	2	Juan	2
<input type="checkbox"/> Editar	3	Maria	1
<input type="checkbox"/> Editar	4	Ana	2

Figura 7.16: Resultado de ejecutar una consulta SQL de selección

Esta consulta trivial selecciona *todas* las columnas y *todas* las filas de una tabla. Aunque parece inocente, es una consulta peligrosa en las aplicaciones reales porque puede implicar un resultado con muchísima información. Recordemos que las tablas pueden almacenar gigabytes de datos.

Agreguemos y combinemos distintas cláusulas para explorar cómo funciona SQL:

```
SELECT *
FROM empleados
WHERE cargo_id = 1
```

La cláusula **WHERE** nos permite filtrar filas. Las filas para las cuales se *cumpla* la condición serán las que se anexen al resultado. Por lo tanto, esta consulta nos mostrará todos los datos de los gerentes.

```
SELECT descripcion
FROM cargos
```

Esta consulta nos muestra todos los cargos de la empresa, pero seleccionando solamente el campo **descripcion**. Si utilizamos un asterisco (*) estaremos indicando *todas* las columnas de la tabla. En caso de querer sólo algunas columnas las enumeramos, separando con comas (,).

```
SELECT adelanto_id, monto
FROM adelantos
WHERE MONTH(fecha) = 5
```

En esta consulta hemos pedido los adelantos, pero sólamente los que fueron hechos en el mes de mayo. Las columnas seleccionadas son la clave primaria del adelanto y su monto. En la cláusula **WHERE** aparece una función. Los distintos motores tienen incorporadas ciertas funciones que nos permiten manipular los datos directamente en la consulta. Como el campo **fecha** de la tabla **adelantos** es de tipo **DATETIME**, es preciso utilizar una función para extraer el mes y poder comparar. El nombre de la columna opera como si se tratara de una variable, que estamos pasando como parámetro a la función. La base de datos aplicará la función fila por fila y descartará las que no correspondan con el criterio pedido.

```
SELECT empleado_id, monto  
FROM adelantos  
ORDER BY monto DESC  
LIMIT 1
```

Esta consulta tiene dos cláusulas nuevas:

- **ORDER BY**: indica cómo deben ordenarse las filas resultantes. Por defecto se realiza ascendente. En caso contrario debe indicarse con **DESC**.
- **LIMIT**: es una cláusula propietaria de MySQL, que impone un límite a la cantidad de filas que queremos recibir del resultado calculado

Como resultado, obtendremos la fila con el adelanto mayor, ya que ordenamos por **monto** de manera descendente (los más grandes primero). Al limitar a una sola fila, encontraremos el adelanto de monto mayor, del cual estamos seleccionando solamente los campos **empleado_id** y **monto**.

La cláusula **ORDER BY** es la última que puede aparecer en una consulta según el estándar ANSI. La cláusula **LIMIT** es propia de MySQL y debe aparecer luego del **ORDER BY**. Es importante notar que esta consulta no tiene **WHERE** y que, de tenerlo, debería ubicarse entre el **FROM** y el **ORDER BY**. Recordemos que las cláusulas tienen un orden específico que debe respetarse. Si no utilizamos el orden correcto o si cometemos cualquier error de sintaxis el servidor nos devolverá un error que, a su vez, será mostrado por el cliente en pantalla. Dejamos al lector la responsabilidad de experimentar qué sucede al generar un error. Los mensajes que nos envía el servidor son muy útiles para corregir el problema por lo que resultará productivo dedicar un momento a examinarlos.

```
SELECT empleado_id, monto  
FROM adelantos  
WHERE MONTH(fecha) = 5  
ORDER BY monto  
LIMIT 1
```

En esta consulta hemos intercalado la cláusula **WHERE**, por lo que ahora obtenemos el adelanto más chico, pero sólamente considerando los que se hayan realizado en el mes de mayo. Nótese que no indicamos **DESC** en el ordenamiento, por lo que se ordena de forma ascendente. Podemos imaginar que la ejecución de esta consulta se realiza en el mismo orden que se escriben las cláusulas: primero se toman las filas de la tabla **adelantos** (**FROM**), luego se quitan las filas que no sean de mayo (**WHERE**), se las ordena (**ORDER BY**) y se devuelve sólo la primera de ellas (**LIMIT**).

Hasta aquí hemos tomado datos de una sola tabla y, mediante algunas cláusulas, obtenido distintos resultados filtrados u ordenados. Pero nuestro modelo de datos tiene tres entidades que, evidentemente, están relacionados entre sí. Veamos cómo podemos escribir consultas que tomen datos de distintas tablas o, con otras palabras, utilicen las relaciones que existen entre las entidades:

```
SELECT nombre, descripcion  
FROM empleados  
LEFT JOIN cargos ON cargos.cargo_id = empleados.cargo_id
```

En esta consulta hemos incorporado una cláusula **JOIN**. Esta cláusula indica que se deben **combinar** los datos provenientes de distintas tablas. En el **SELECT** indicamos que el resultado debe incorporar la columna **nombre**, que surge de la tabla **empleados** mencionada en el **FROM**. Pero además se indica la columna **descripcion**, que no pertenece a esa tabla sino a **cargos**. Lo solicitamos de esta forma porque no queremos ver los *números* sino las *palabras* que *describen* el cargo de cada empleado. Para que la base de datos pueda devolvernos esta columna es necesario que le indiquemos cuál es la relación entre esas dos tablas.

La cláusula **JOIN** pone de manifiesto lo que hemos dibujado antes con una flecha, es decir, relaciona las claves foráneas con las primarias. Podríamos leer el **JOIN** utilizado en el ejemplo de esta forma: "relacionar **empleados** con **cargos**, siendo la columna **cargo_id** de **cargos** *lo mismo* que **cargo_id** de **empleados**". Si las tablas han sido diseñadas a conciencia, es muy probable que estos nombres de campo coincidan ya que se tratan de la misma cosa.

Es importante notar que el resultado no es simplemente un agregado de una tabla con la otra. La base no ha tomado la columna **descripcion** y la ha "pegado" al lado de los empleados. Por el contrario, ha puesto en juego la *relación* y colocado a cada persona el nombre del cargo que le corresponde. Cuando realizamos el diseño relacional de la base de datos descomponemos la información en entidades y relaciones, que luego volvemos a reconstruir al hacer las consultas. Las bases de datos relacionales están diseñadas para ser muy eficientes en este trabajo y es, justamente, su razón de ser.

Cuando exista **ambigüedad** en los nombres de campos o cuando se quiera ser más claro en cuanto a qué tabla corresponde cada columna del **SELECT**, se puede indicar de esta forma por medio del **punto** (.):

```
SELECT empleados.nombre, cargos.descripcion  
FROM empleados  
LEFT JOIN cargos ON cargos.cargo_id = empleados.cargo_id
```

Es muy común utilizar **alias** para reducir la extensión de la consulta y evitar errores de tipeo, de esta forma:

```
SELECT e.nombre, c.descripcion  
FROM empleados e  
LEFT JOIN cargos c ON c.cargo_id = e.cargo_id
```

Cuando se nombra la tabla por primera vez, se indica su alias por medio de un simple espacio. Este alias, por ejemplo una letra, se puede utilizar luego en lugar del nombre completo. Los alias no se almacenan en ningún lado y sólo funcionan dentro de cada consulta.

En nuestros ejemplos utilizaremos siempre **JOIN** de tipo **LEFT**, que es el utilizado casi siempre. También existe el **RIGHT**, **NATURAL**, **INNER**, **OUTER** y algunas variantes más. Estas indicaciones modifican el comportamiento respecto de los **NULL** que hemos mencionado antes. En el caso del **LEFT**, el motor tomará las filas de la tabla indicada en el **FROM** y

luego buscará en la tabla indicada en **JOIN** los campos que les corresponden. Si utilizáramos **RIGHT** la búsqueda se hace al revés: comenzando por **cargos**. Podríamos ilustrar esta diferencia como "a los empleados agregarles los cargos" (**LEFT**) versus "tomar los cargos y agregarles los empleados" (**RIGHT**). El cambio de dirección afectará el resultado si hay cargos que no tienen ningún empleado. El **LEFT JOIN** desde **empleados** no mostrará los cargos *sin* empleados. Esto puede ilustrarse así:

empleados LEFT JOIN cargos

empleados		
nombre	cargo_id	descripcion
Pepe	1	Gerente
Juan	2	Empleado
Maria	1	Gerente
Ana	2	Empleado
Roberto	5	NULL

cargos	
cargo_id	descripcion
1	Gerente
2	Empleado
3	Supervisor

empleados RIGHT JOIN cargos

empleados	
nombre	cargo_id
Pepe	1
Juan	2
Maria	1
Ana	2
Roberto	5

cargos	descripcion	nombre
1	Gerente	Pepe
1	Gerente	Maria
2	Empleado	Juan
2	Empleado	Ana
3	Supervisor	NULL

Figura 7.17: Comparación entre **LEFT JOIN** y **RIGHT JOIN**

La cláusula **JOIN** se ubica siempre después del **FROM**. La siguiente consulta contiene además un **WHERE**:

```
SELECT e.nombre, c.descripcion
FROM empleados e
LEFT JOIN cargos c ON c.cargo_id = e.cargo_id
WHERE e.cargo_id=1
```

Obtendremos una lista de empleados gerentes.

Hasta aquí hemos presentado las cláusulas de SQL que nos permiten seleccionar filas y relacionar las entidades entre sí para obtener resultados filtrados y combinados. Pero los motores de bases de datos son capaces de muchas más operaciones. Entre ellas, podemos realizar **agrupamientos** para calcular resultados. Por ejemplo:

```
SELECT empleado_id, SUM(monto)
FROM adelantos
GROUP BY empleado_id
```

La cláusula **GROUP BY**, que se coloca en orden luego del **WHERE**, indica al motor que agrupe las filas que tengan la misma información en cierto campo. La consecuencia es que varias filas se "apilan" y pasan a ser una sola. Además, en el **SELECT** utilizamos una **función de agregación**. Las funciones de agregación operan sobre los datos que se han *apilado*, en este caso, calculando la suma aritmética. Esta consulta, entonces, tomará las filas de

la tabla `adelantos` y agrupará las filas tomando como criterio el número de empleado, por lo que obtendremos un total de dinero pedido por cada empleado. Por supuesto, si el agrupamiento se realiza según otro criterio obtendremos distintas sumas. Si agrupamos por año, por ejemplo, obtendremos los totales por año.

Si utilizamos una función de agregación sin especificar un `GROUP BY`, se tomarán todas las filas de la tabla. Algunas versiones antiguas, sin embargo, no permiten realizar algunas operaciones de esta manera.

```
SELECT SUM(monto)
FROM adelantos
```

Con esta consulta obtendremos el total de dinero pedido, en toda la historia de la base de datos.

Casi siempre que utilicemos estas funciones será necesario colocar un alias a la columna calculada. De no hacerlo, el nombre de la columna quedará con un formato muy complicado, con paréntesis incluidos, que puede traer problemas al utilizar el resultado en un lenguaje de programación como PHP. Indicamos un alias de esta forma:

```
SELECT SUM(monto) total
FROM adelantos
```

Otras funciones de agregación utilizadas comúnmente son:

- `AVG (average)`: promedio
- `COUNT`: cuenta la cantidad de filas que se apilaron
- `MAX` y `MIN`: devuelven el valor máximo y mínimo de los que se agruparon
- `GROUP_CONCAT`: concatena los valores apilados

La siguiente consulta nos dirá cuál es el promedio que pidió cada empleado:

```
SELECT empleado_id, AVG(monto) promedio
FROM adelantos
GROUP BY empleado_id
```

Si quisiéramos obtener además el *nombre* del empleado deberíamos realizar un `JOIN`, como ya hemos visto:

```
SELECT a.empleado_id, AVG(a.monto) promedio, e.nombre
FROM adelantos a
LEFT JOIN empleados e ON e.empleado_id = a.empleado_id
GROUP BY a.empleado_id
```

Si además quisiéramos indicar el *cargo* que tiene este empleado deberíamos agregar otro `JOIN` más, para relacionar también con la tabla `cargos`:

```
SELECT a.empleado_id, AVG(a.monto) promedio, e.nombre, c.descripcion
FROM adelantos a
LEFT JOIN empleados e ON e.empleado_id = a.empleado_id
LEFT JOIN cargos c ON e.cargo_id = c.cargo_id
GROUP BY a.empleado_id
```

Las consultas, a medida que necesitamos resultados con mayor cantidad de información, se tornarán cada vez más extensas. Cuando la cantidad de tablas en el modelo de datos crece, también se hacen necesarios más y más **JOIN**. Esto es habitual en la práctica cotidiana con el lenguaje SQL.

Veamos de qué forma podemos saber cuánto pide en promedio cada empleado, pero sólamente quienes sean gerentes:

```
SELECT e.nombre, AVG(a.monto) promedio
FROM adelantos a
LEFT JOIN empleados e ON e.empleado_id = a.empleado_id
WHERE e.cargo_id=1
GROUP BY a.empleado_id
```

Supongamos ahora que nos piden la misma información, pero se requiere eliminar a los gerentes que hayan pedido menos de \$300 en promedio. Es muy importante notar que este filtro no se puede indicar en el **WHERE**, porque esta cláusula se ejecuta *antes* de llegar al **GROUP BY**. Con otras palabras, cuando se filtran con **WHERE** las filas de la tabla indicada en el **FROM**, todavía no se ha hecho ningún agrupamiento. Es *después* del agrupamiento cuando se sabe cuál fue el promedio por empleado y recién entonces podríamos filtrar. Veamos cómo se resuelve entonces esa consulta:

```
SELECT e.nombre, AVG(a.monto) promedio
FROM adelantos a
LEFT JOIN empleados e ON e.empleado_id = a.empleado_id
WHERE e.cargo_id=1
GROUP BY a.empleado_id
HAVING promedio > 300;
```

La cláusula **HAVING** constituye un filtro que se puede aplicar *luego* de haber agrupado. Cuando el agrupamiento involucre algún cálculo con funciones de agregación, y necesitemos filtrar las filas con datos resultantes de este cálculo, utilizaremos **HAVING**.

Tenemos entonces un panorama completo de las principales cláusulas que se utilizan habitualmente para recuperar filas de la base de datos. En resumen y ordenadamente:

- **SELECT**: qué campos/columnas forman el resultado
- **FROM**: de qué tabla tomar las filas
- **LEFT JOIN**: con qué otras tablas relacionar
- **WHERE**: condiciones para que las filas entren en el resultado
- **GROUP BY**: criterios para agrupar filas
- **HAVING**: condiciones para que las filas ya agrupadas entren en el resultado
- **ORDER BY**: criterios para ordenar el resultado
- **LIMIT**: cantidad de filas a devolver (no ANSI)

Existen más operadores y características que pueden utilizarse para expresar consultas de selección, siendo algunos de ellos irremplazables en algunos casos que, sin embargo, no son los habituales. De todas formas, no es nuestro objetivo realizar una presentación exhaustiva sobre SQL en este momento sino ofrecer un panorama de la idea que persigue esta tecnología y cuáles son sus posibilidades. Además, recordemos que cada motor y cada versión tiene sus particularidades y posibilidades.

7.9 CONSULTAS DE MODIFICACIÓN

Las otras consultas de manipulación de datos son, a diferencia de la selección, las que **modifican** los datos existentes en la base. Estas consultas son mucho más fáciles porque se utilizan casi siempre de la misma manera y no suelen presentar sorpresas.

La siguiente consulta **crea** una fila en una tabla existente, cada vez que se ejecuta:

```
INSERT INTO adelantos (empleado_id, monto, fecha)
VALUES (4, 300.50, "2013-08-01")
```

Se indica el nombre de la tabla en donde queremos crear la fila y los nombres de los campos para los cuales indicaremos datos. Recordemos que algunos campos *pueden* tener valores nulos: son los que hemos llamado campos **NULL**. Estos campos quedarán con un valor nulo si creamos una fila y no indicamos ningún dato para ellos. Además, el orden en que se indican los campos es indistinto y no guarda relación con el orden que tienen las columnas en la tabla real. De hecho, el modelo relacional no soporta este concepto de *orden* en las columnas ni tampoco, sorprendentemente, el orden en las filas. Siendo estrictos, el modelo no habla de filas ni de columnas sino de *conjuntos*.

Por medio de **VALUES** se indica cuáles son los datos a insertar en la nueva fila, manteniendo por supuesto el orden de las columnas indicadas anteriormente. Nótese que las fechas se indican entre comillas, al igual que las strings.

Antes, hemos creado filas utilizando la funcionalidad "Insertar" que nos facilita el cliente. En realidad, phpMyAdmin ha tomado los datos que escribimos en el formulario y envió una consulta **INSERT** al servidor concatenándolos con las partes fijas de esta consulta. El cliente nos facilita ciertas operaciones por medio de su interfaz gráfica pero, como hemos dicho, lo único que se puede hacer con el servidor de bases de datos es enviarle consultas SQL. El cliente gráfico traduce nuestras acciones en consultas.

La siguiente consulta permite **modificar** filas existentes:

```
UPDATE empleados
SET nombre = "Juan"
WHERE empleado_id = 1
```

Se indica sencillamente el nombre de la tabla y los campos a modificar, seguidos de sus nuevos valores. En este caso se modificará el nombre de un empleado.

Es muy importante recalcar el uso del **WHERE** en estas consultas de modificación. Si no indicamos la fila que debe ser afectada, el servidor **modificará el nombre de todos** los empleados. Lo mismo ocurre al realizar un **SELECT**, en donde la ausencia de **WHERE** indica que queremos obtener *todas* las filas. Olvidar el **WHERE** en un **UPDATE** no es algo inusual. Lamentablemente esta omisión es **catastrófica**.

Para eliminar filas ejecutamos la siguiente consulta:

```
DELETE
FROM adelantos
WHERE monto < 10
```

Lo mismo cabe aclarar enfáticamente para esta consulta. Si no indicamos el `WHERE`, el servidor **borrará todas las filas** de la tabla `adelantos`. Esta consulta, en cambio, borra sólamente las filas que tengan un monto inferior a \$10, es decir, aquellas para las cuales se cumple la condición del `WHERE`.

Es interesante comparar con la siguiente consulta, casi idéntica:

```
SELECT *
FROM adelantos
WHERE monto < 10
```

Si reemplazamos el `DELETE` por un `SELECT *` podremos ver las filas que serán afectadas, es decir, eliminadas **sin posibilidad de volver atrás** más que recuperando algún *backup*, si es que lo tenemos.

Si la consulta a ejecutar no es un `SELECT`, es imperiosamente recomendable revisarla con mucha atención antes de enviarla al servidor. Olvidar un `WHERE` puede ser trágico.

7.10 MANEJO DE TEXTO

El lenguaje SQL incluye un operador de comparación de texto que permite la utilización de **comodines**: el operador `LIKE`. Estos comodines hacen que la búsqueda de cadenas dentro de una tabla pueda realizarse de manera más flexible, por medio de patrones. Veamos la siguiente consulta:

```
SELECT *
FROM empleados
WHERE nombre LIKE "% Perez"
```

Esta consulta nos devolverá todos los empleados que tengan apellido "Perez", ya que el comodín **porcentaje (%)** genera coincidencia con cero, uno o más caracteres cualquiera. El espacio se tomará literal, por lo que antes de "Perez" deberá existir un espacio. Antes de ese espacio podrá haber cualquier cadena, incluso nada. Las personas que cumplen estas condiciones en su nombre serán las enviadas al resultado.

```
SELECT *
FROM empleados
WHERE nombre LIKE "Antoni_ %"
```

Esta consulta devolverá las personas llamadas "Antonia" y "Antonio", debido a que el comodín **guión bajo (\)** genera coincidencia con exactamente un carácter cualquiera. Utilizamos un comodín `%` para coincidir con cualquier apellido.

Existe una función avanzada de búsqueda de texto llamada `MATCH`, que se utiliza para realizar las llamadas búsquedas de **texto completo**. Esta búsqueda se realiza teniendo en cuenta el lenguaje natural, de la misma manera que lo hacen motores como Google. La búsqueda se realiza por *similaridad* de palabras, y no por patrones o coincidencia exacta de caracteres. Los textos son ordenados por grado de similitud, permitiendo ofrecer los resultados más significativos primero. Si bien es una funcionalidad interesante, sólo se puede utilizar con cierto tipo de bases y es preciso configurar algunos

aspectos de la tabla antes de poder ejecutar el `MATCH`. Dejamos al lector la posibilidad de investigarlo con más profundidad.

Finalmente, SQL también incluye las clásicas funciones de string que podemos encontrar en cualquier lenguaje de programación. Mencionaremos sólo algunas:

```
SELECT nombre, UPPER(nombre) mayusculas, LOWER(nombre) minusculas,  
       TRIM(nombre) limpio  
  FROM empleados
```

En esta consulta podemos observar la conversión a mayúsculas, a minúsculas y la eliminación de espacios blancos a izquierda y derecha. También es posible realizar recortes, contar la cantidad de caracteres, invertir las letras, etcétera.

7.11 FORMATO DE FECHAS

Es muy común trabajar con datos temporales en las bases de datos. De hecho, los motores están preparados muy eficientemente para realizar cálculos con estos tipos de datos e incluyen muchas funciones útiles para su tratamiento.

El formato que utiliza MySQL para las fechas es año-mes-día. Normalmente necesitaremos mostrar las fechas en un formato más adecuado a nuestras costumbres. Podemos cambiar el formato de esta manera:

```
SELECT DATE_FORMAT(fecha, "%d/%m/%Y") fecha_castellano  
      FROM adelantos  
     ORDER BY fecha DESC
```

La función `DATE_FORMAT` recibe un nombre de campo y convierte al formato que especifiquemos mediante una cadena con indicadores de formato. Lo habitual será que necesitemos convertir las fechas al momento de recuperarlas. Contrariamente, al momento de enviarlas al servidor, ya sea para almacenarlas o para realizar comparaciones en una selección, debemos utilizar el formato propio del motor.

7.12 EXPORTACIÓN E IMPORTACIÓN

Uno de los aspectos que hemos destacado de los motores de bases de datos es que la única forma de interacción es a través de un cliente y por medio de consultas SQL. Cuando pedimos a un servidor que cree una tabla, que almacene datos y luego procedemos a preguntar por ellos, evidentemente será necesario que el motor realice acciones para persistir la información y eventualmente recuperarla. La tarea del servidor de bases de datos es, justamente, llevar a cabo esas tareas por nosotros y, no menos importante, aislarnos de esas tareas necesarias de manera que no sean nuestro problema.

Podemos preguntarnos qué sucede en el caso que necesitemos migrar la información de un servidor a otro. Si bien podemos suponer que el servidor almacena los datos en archivos de disco, por el sólo hecho de estar usando un motor de bases de datos nos hemos obligado a no intervenir directamente en estos archivos. Esta es la contracara

del aislamiento que mencionábamos: nos libra de un problema, pero en algún punto nos hace perder el control físico de la información.

Para trasladar la información de un motor a otro, por ejemplo para hacer un *backup*, se deben realizar dos operaciones llamadas **exportación** e **importación**. La exportación genera un archivo con extensión **.sql** que contendrá muchas consultas en lenguaje SQL. La importación es la acción de ejecutar dichas consultas sucesivamente, lo que tendrá como consecuencia la creación de una base de datos idéntica a la de origen, de modo que se conserva la estructura y los datos. Por lo general este es el comportamiento deseado, aunque la estructura y los datos también pueden exportarse por separado dependiendo del caso.

Para generar una exportación, se utiliza la pestaña "Exportar" de phpMyAdmin.



Figura 7.18: Pestaña para exportación de base de datos

Esta operación tiene distinto significado según dónde la utilicemos. Si estamos ubicados en el servidor (*localhost*) se exportarán las bases, si estamos en una base de datos se exportan sus tablas y si estamos en una tabla se exportan sus filas. Por esta razón es importante tener en cuenta desde qué pantalla se accede a la opción de exportación. El formato "Personalizado" nos permite tener todo el control sobre las consultas que se volcarán definitivamente al archivo de exportación.

La importación es mucho más simple, y se reduce a entrar en la pestaña "Importar". Si el archivo de exportación está completo, es decir que no se trata de exportaciones de datos parciales sino de bases enteras, podemos acceder a esta opción desde cualquier lado de la interfaz ya que las consultas crearán una base nueva y colocarán las filas en su interior. phpMyAdmin nos pide elegir la ubicación del archivo a importar, y con eso se completa el ciclo.

Siempre que generemos un *script* de exportación es recomendable verificar las consultas que ha creado el cliente dentro del archivo, para asegurarnos de contar con las consultas que crean la base, las tablas y las filas, de acuerdo a cuál sea el objetivo de nuestra exportación. No es raro encontrar *backups* que se generan cotidianamente pero al momento de precisarse se descubren incompletos. Es necesario hacer *backups* regularmente pero también asegurar que el proceso de reconstrucción sea correcto y completo,

aunque nuestra intención sea no tener que pasar nunca por ese proceso de recuperación de catástrofes.

7.13 CONSULTAS DESDE PHP

El intérprete del lenguaje PHP posee un módulo que nos permite conectar y ejecutar consultas en una base de datos MySQL. Esto quiere decir que, desde PHP, podemos acceder programáticamente a un **cliente MySQL**. De hecho, una de las grandes fortalezas del lenguaje PHP es la variedad de clientes de bases de datos disponibles.

Crearemos un archivo llamado `basedatos.php` dentro de la raíz `C:\wamp\www` para examinar el uso de estas funciones. Utilicemos la primera de ellas:

```
<?php
$conexion = mysqli_connect("localhost", "root", "", "empresa");
var_dump($conexion); //object
?>
```

Al abrir con un navegador la URL `http://localhost/basedatos.php` debemos observar que el tipo retornado es `object`. Si la conexión falla se retornará un `false`.

Los parámetros de la función `mysqli_connect` son, respectivamente: servidor a conectar, usuario, contraseña y base de datos a utilizar.

Una vez que hemos establecido una conexión podemos ejecutar consultas en el servidor. Realicemos un `SELECT` para recuperar los nombres de los empleados:

```
<?php
$conexion = mysqli_connect("localhost", "root", "", "empresa");
$consulta = "SELECT nombre FROM empleados";
$resultado = mysqli_query($conexion, $consulta);

var_dump($resultado); //object
?>
```

Podemos observar que la función `mysqli_query` recibe dos parámetros: el objeto que representa a la conexión y una string con la consulta, respectivamente.

El resultado de ejecutar la consulta es otro objeto. Para acceder efectivamente a los datos que han vuelto, como resultado de ejecutar la consulta en el servidor, debemos utilizar otra función que extrae una fila por vez:

```
<?php
$conexion = mysqli_connect("localhost", "root", "", "empresa");
$consulta = "SELECT nombre FROM empleados";
$resultado = mysqli_query($conexion, $consulta);

$fila = mysqli_fetch_assoc($resultado);
var_dump($fila); //array con 1 elemento ["nombre"]
?>
```

La función `mysqli_fetch_assoc` toma la próxima fila del resultado y nos devuelve un array. En las **claves** de este array coloca los nombres de los campos tal como figuran en

el resultado y en los **valores** coloca los datos de la fila convertidos a string. Por esta razón es que recomendamos utilizar alias en las consultas para los nombres de campos extensos o con caracteres extraños, ya que se convertirán directamente en las claves de estos arrays de resultados. De igual forma, es importante recordar que las claves de los arrays de PHP son sensibles a mayúsculas y minúsculas. Cuando los arrays de PHP contienen strings en lugar de enteros se los suele llamar "array asociativos", lo que en otros lenguajes es llamado también *mapa* o *diccionario*. De aquí proviene el nombre de esta función, ya que retorna un array con claves de tipo string.

Esta función está diseñada para utilizarse dentro de un bucle, ya que avanza automáticamente fila por fila cada vez que es llamada, con una especie de puntero interno. Cuando no hay más filas en el resultado retorna un valor **null** que se puede utilizar para cortar la ejecución de un **while**. Típicamente, el uso habitual de estas funciones es:

```
<!-- se omite el resto del documento HTML -->
<?php
$conexion = mysqli_connect("localhost", "root", "", "empresa");
$consulta = "SELECT nombre FROM empleados";
$resultado = mysqli_query($conexion, $consulta);

while($fila = mysqli_fetch_assoc($resultado)) { ?>

    <p><?=$fila[ 'nombre' ]?></p>

<?php
}
?>
```

Cuando **\$resultado** no tenga más filas por extraer, **mysqli_fetch_assoc** retornara un valor **null**, que al ser asignado a **\$fila** cortará el bucle. Recordemos que el *valor* de una expresión de asignación es igual al valor que está siendo asignado y que **null** se evalúa como el valor booleano **false**.

Aunque es muy poco habitual, existe la posibilidad de que precisemos recorrer varias veces el mismo resultado. Como el puntero avanza automáticamente cada vez que se extrae una fila, podemos volverlo hacia atrás con esta función:

```
mysqli_data_seek($resultado, 0);
```

De hecho, esta función puede utilizarse para mover el puntero de manera arbitraria a cualquier posición del conjunto resultante. Esto, insistimos, es muy poco común y tal vez sea indicación de un mal diseño o una consulta que puede mejorarse.

Una función que también resulta muy útil es la que retorna las cantidad de filas:

```
<?php
$conexion = mysqli_connect("localhost", "root", "", "empresa");
$consulta = "SELECT nombre FROM empleados";
$resultado = mysqli_query($conexion, $consulta);

$cant = mysqli_num_rows($resultado);
var_dump($cant); //int
?>
```

La función `mysqli_num_rows` nos retorna un entero, con la cantidad de filas que ha tenido una consulta como resultado. Recordemos que, si el sólo objetivo de la consulta es contar filas, lo más eficiente es utilizar la función de agregación `COUNT` de SQL para obtener una sola fila con el resultado del conteo ya ejecutado.

Por último, existe una función que puede resultar muy útil al desarrollar:

```
<?php  
$conexion = mysqli_connect("localhost", "root", "", "empresa");  
$consulta = "SELECT nombre FROM empleadossss";  
$resultado = mysqli_query($conexion, $consulta);  
  
$problema = mysqli_error($conexion);  
var_dump($problema); // (string) Table 'empresa.empleadossss' doesn't exist  
?>
```

La función `mysqli_error` recibe como parámetro la conexión y retorna una string con el posible mensaje de error de la *última* consulta ejecutada. Si la última consulta se ejecutó correctamente, es decir sin producir errores, `mysqli_error` retornará una string vacía. El mensaje devuelto proviene directamente del servidor MySQL.

7.14 ADVERTENCIA DE SEGURIDAD

Ensamblando los temas que presentamos en los sucesivos capítulos, y seguramente aplicando mucha paciencia, el lector está en condiciones de llevar adelante el desarrollo de un sitio web dinámico de pequeña y mediana escala.

No obstante, es muy importante destacar que es muy fácil crear aplicaciones inseguras por el mero desconocimiento de los posibles ataques que se pueden efectuar contra nuestro sistema. No disponemos en este libro del lugar suficiente para explayarnos sobre este tema, pero debemos dejar planteada la cuestión para alertar al lector y ponerlo sobre aviso.

Las consultas al servidor de base de datos suelen ser, en el contexto de una aplicación web, el lugar donde más **problemas de seguridad** existen. Estos problemas pueden llevar a que un tercero con malas intenciones destruya todos nuestros datos, sencillamente enviando paquetes manipulados de una forma que no habíamos previsto.

Cuando una aplicación realiza una consulta a la base de datos, y en esta consulta se han *concatenado* datos provenientes del *lado cliente*, existe el potencial riesgo de crear una **inyección SQL**. Estas inyecciones ocurren cuando desde el lado cliente se envían datos que contienen comandos SQL, y el programador los concatena dentro de una consulta y los ejecuta sin haberlos **validado** previamente. Las inyecciones SQL son noticias muy habituales, lamentablemente, en el mundo del software.

Por lo pronto, y sin lugar para explayarnos más sobre el problema ni la solución, diremos que es preciso validar *todo* dato que provenga del cliente. Los números, strings, fechas: absolutamente todo debe ser validado y **sanitizado** *antes* de concatenarse dentro de una consulta SQL.

Este programa es **vulnerable** a una inyección SQL porque se concatena directamente:

```
<?php  
$conexion = mysqli_connect("localhost", "root", "", "empresa");  
  
$nombre = $_GET['nombre'];  
$cargo = $_GET['cargo'];  
  
mysqli_query($conexion, "insert into empleados (nombre,cargo_id)  
values ('$nombre', $cargo)"); //MAL!!!  
?>
```

Este programa no es vulnerable porque los datos fueron sanitizados:

```
<?php  
$conexion = mysqli_connect("localhost", "root", "", "empresa");  
  
$nombre = substr($_GET['nombre'], 0, 50);  
$nombre = mysqli_escape_string($conexion, $nombre);  
  
$cargo = (int) $_GET['cargo'];  
  
mysqli_query($conexion, "insert into empleados (nombre,cargo_id)  
values ('$nombre', $cargo)"); //mejor  
?>
```

Confiamos en que el lector pondrá todo su empeño en interiorizarse sobre estos aspectos tan importantes de la programación profesional. Normalmente decimos que, antes que tener una aplicación con problemas de seguridad potencialmente dañinos legal y económicamente, es mejor no tener nada.

7.15 BIBLIOGRAFÍA

El manual online del servidor MySQL puede encontrarse en dev.mysql.com/doc/refman/5.0/es/. Este recurso es exhaustivamente completo y provee de toda la información sobre el funcionamiento de este motor, además de la sintaxis completa de todas las cláusulas SQL que soporta. Las funciones del módulo cliente de PHP se encuentran documentadas en el manual online: php.net/manual.

Sobre el modelo relacional, normalización y el diseño de bases de datos, la bibliografía clásica es C. J. Date, “*Introducción a los Sistemas de Bases de Datos*”, séptima edición, Pearson Educación, México: 2001.

Por último, aunque los conceptos del diseño relacional son simples de entender a primera vista, la verdadera comprensión y fluidez vendrá de la mano de la experiencia práctica con aplicaciones reales. Algunas veces, incluso, las necesidades de performance llevan a diseños intencionadamente no relacionales.

7.16 EJERCICIOS

Dada la base de datos "empresa", utilizada en el desarrollo del capítulo, resuelva las siguientes consultas. Agregue filas a las tablas en caso de ser necesario, para probar la correcta ejecución de todos los puntos.

1. Liste los 3 adelantos con monto mayor.
2. Obtenga el nombre del empleado que solicitó el adelanto con menor monto.
3. Liste los adelantos que fueron solicitados por los gerentes.
4. Liste los 3 adelantos más grandes que fueron pedidos por no-gerentes.
5. Obtenga cuál fue el monto de dinero total pedido en el año 2012, por todos los empleados en conjunto.
6. Obtenga el nombre del empleado que menos dinero total pidió, de los que han pedido adelantos alguna vez. Considere la cantidad de dinero y no la cantidad de adelantos.
7. Liste los nombres de los empleados que no son gerentes junto a la cantidad de adelantos que han pedido.
8. Liste los nombres de los empleados que pidieron más de 2 adelantos.
9. Liste los nombres de los empleados que no han pedido adelantos nunca.
10. Determine en qué mes se solicitan más adelantos.

A. JSON

JSON (*JavaScript Object Notation*), pronunciado [shéison], es un formato para el intercambio de información entre distintas plataformas. Originalmente surge en el lenguaje JavaScript, pero dado que existen formas de convertir hacia y desde JSON los datos de diversos lenguajes, paulatinamente se ha convertido en un formato de intercambio más general. En ese sentido compite con otros formatos como XML y YAML.

El formato JSON permite serializar un objeto. La **serialización** es la representación de un dato mediante caracteres que garanticen que pueda transportarse de diversas maneras, y poder ser reconvertido a su forma original. Los datos que pueden ser representados con el formato JSON son:

- Números: enteros o flotantes. En JavaScript son el mismo tipo.
- Booleanos
- Strings
- El valor *null*
- Vectores: secuencias ordenadas de valores
- Objetos: asociaciones de claves y valores, también llamados *diccionarios* o *mapas*. Similares a los arrays de PHP.

Realicemos algunas pruebas con JavaScript, utilizando la consola del navegador.

Mediante el método `stringify` del objeto global `JSON` podemos obtener el valor de una variable convertido a JSON, y retornado en una string. En el caso de valores escalares, esta representación no difiere del valor mismo.

Intentemos convertir un array de JavaScript a JSON:

```
var original = new Array();
original[0] = "hola";
original[1] = 666;
var convertido = JSON.stringify(original);
alert(convertido); // ["hola", 666]
```

Los vectores se representan en JSON mediante **corthetes** `([])`, y los valores se ubican separados por comas. Podemos realizar la misma prueba utilizando PHP, para examinar si el resultado es el mismo:

```
<?php
$original = array(0 => "hola", 1 => 666);
$convertido = json_encode($original);
echo $convertido; // ["hola", 666]
?>
```

La función `json_encode` de PHP realiza la misma conversión de un dato a su representación en JSON. Podemos ver que ambos resultados son idénticos, lo que significa que

son **interoperables**: podemos enviar un array desde PHP a JavaScript, y viceversa, utilizando el formato JSON como punto en común. En cada lenguaje podemos volver a obtener el valor original de este modo:

```
var convertido = '["hola", 666]';
var original = JSON.parse(convertido);
alert(original[1]); // 666
```

Claramente, el método `parse` de JavaScript realiza la conversión contraria a la realizada con `stringify`, es decir, la **deserialización**. A partir de una string vuelve a recomponer el dato original: un array.

De manera equivalente, se puede lograr en PHP con `json_decode`:

```
<?php
$convertido = '["hola", 666]';
$original = json_decode($convertido);
var_dump($original); //vemos el array con dos elementos
?>
```

Veamos ahora cómo se representan los mapas, llamados "objetos" en la terminología JSON. Crearemos una prueba de concepto partiendo de un objeto sencillo en PHP. Convertiremos este objeto a JSON y lo escribiremos dentro de un pequeño programa JavaScript que hará lo opuesto, y demostrará el intercambio de información con el mantenimiento de los tipos:

```
<?php
class Prueba {
    public $nombre;
    public $edad;
}

$original1 = new Prueba; //instanciación
$original1->nombre = "Pepe"; //cargamos algunos datos en las variables miembro
$original1->edad = 30;

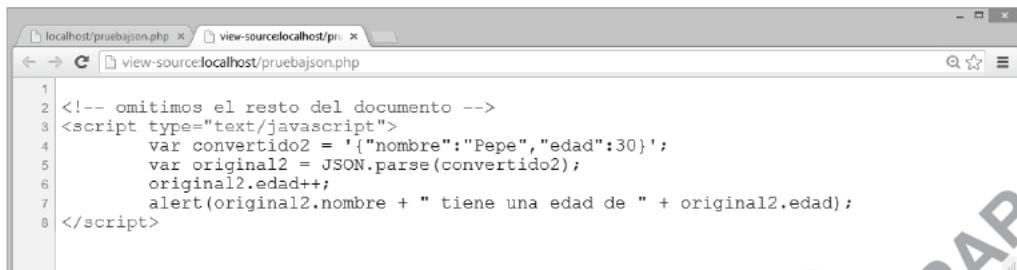
$convertido1 = json_encode($original1);

?>

<!-- omitimos el resto del documento --&gt;
&lt;script type="text/javascript"&gt;
    var convertido2 = '&lt;?= $convertido1 ?&gt;';
    var original2 = JSON.parse(convertido2);
    original2.edad++;
    alert(original2.nombre + " tiene una edad de " + original2.edad);
&lt;/script&gt;</pre>
```

Si navegamos hacia este programa veremos que el alert muestra correctamente el mensaje "Pepe tiene una edad de 31 años". Detallaremos el funcionamiento de esta interesante prueba de concepto.

En primer lugar, es muy importante notar que existen dos momentos de ejecución. La porción de código PHP se ejecutó en el servidor, lo que generó que la variable convertido2 llegue al navegador con una string producto de la serialización a JSON. Este es el código fuente que llega al navegador, es decir, el resultado de la ejecución del PHP:

A screenshot of a web browser window. The address bar shows 'localhost/pruebajson.php' and 'view-source:localhost/pruebajson.php'. The main content area displays the following PHP code:

```
1 2 <!-- omitimos el resto del documento -->
3 <script type="text/javascript">
4     var convertido2 = '{"nombre":"Pepe","edad":30}';
5     var original2 = JSON.parse(convertido2);
6     original2.edad++;
7     alert(original2.nombre + " tiene una edad de " + original2.edad);
8 </script>
```

The code is displayed in a monospaced font, with line numbers on the left.

Figura A.1: Resultado de la ejecución del programa PHP

Al llegar al navegador, este código fuente es interpretado y tiene como resultado la ejecución del programa JavaScript, es decir, se realiza un `alert`. Este es el segundo momento que distinguimos.

En segundo lugar, puede verse que los tipos se han conservado en el intercambio, no solamente el objeto original -lo que queda demostrado por los operadores miembro de cada lenguaje (`->` en PHP y `.` en JavaScript)- sino que la edad pasó correctamente de tipo `int` en PHP a `number` de Javascript. El incremento de la variable `original2` lo demuestra.

En tercer lugar, inspeccionando el código fuente se puede ver cómo es la sintaxis de los objetos en JSON:

```
var convertido2 = '{"nombre":"Pepe","edad":30}';
```

Las llaves (`{}`) demarcán los objetos JSON, teniendo cada uno de sus miembros un nombre y un valor que se separa con dos puntos (`:`). Cada miembro se separa del anterior por medio de una coma (`,`).

Por supuesto, los tipos se pueden anidar, de manera que un objeto puede incorporar a otros o mezclarse con vectores y escalares de la manera que se necesite.

Este ejemplo ha demostrado cómo podemos intercambiar un dato entre distintos lenguajes, manteniendo el tipo y garantizando la interoperabilidad. La simplicidad de JSON lo ha convertido en un formato muy utilizado en las aplicaciones web y en el diseño de interfaces de aplicación (API). Por ejemplo, accediendo a esta URL puede obtenerse un JSON con información sobre un usuario de Facebook: `graph.facebook.com/4`

La sintaxis de JSON se puede utilizar además directamente en JavaScript para construir objetos, ya que se trata originalmente de un formato de *notación* de este lenguaje. Estos dos ejemplos ilustran este hecho:

```

var a = [2,3,4];
var b = { c: "uno", d: "dos" };
alert(a.length); //3
alert(b.c); // uno

```

Por esta razón, cuando encontremos una llave (`{`) en JavaScript podemos estar ante una apertura de bloque (de un bucle, un `if`, una función, etc.) o ante el comienzo de un JSON.

De hecho, esta notación es tan compacta y conveniente que es muy utilizada para el pasaje de parámetros a las funciones, cuando la cantidad de parámetros es variable. La siguiente porción de código corresponde al *widget Dialog* de jQuery UI:

```

$("#mensaje").dialog({
    autoOpen: false,
    show: {
        effect: "blind",
        duration: 1000
    },
    hide: {
        effect: "explode",
        duration: 1000
    }
});

```

En este código, se está llamando a un método `dialog` sobre un objeto returned por la función principal de jQuery (`$`). Lo interesante es que como parámetro del método se está enviando un JSON representando un objeto. Este objeto tiene las claves `autoOpen`, `show` y `hide`. La clave `autoOpen` tiene un valor booleano, mientras que las otras dos claves tienen a su vez otros objetos con dos claves cada uno, con valores de tipo `string` y `number`. Podemos representar el objeto que se está enviando como parámetro de esta forma:

<code>autoOpen</code>	<code>false</code>	
<code>show</code>	<code>effect</code>	"blind"
	<code>duration</code>	1000
<code>hide</code>	<code>effect</code>	"explode"
	<code>duration</code>	1000

Figura A.2: Representación esquemática del objeto JSON

B. AJAX

AJAX (*Asynchronous JavaScript And XML*) es una técnica que utiliza un objeto nativo muy útil de los navegadores. Este objeto nos permite realizar peticiones HTTP a un servidor, de manera que podemos, desde JavaScript, dialogar con un servidor web en un programa y sin la intervención del usuario. Esto tiene como ventaja poder recuperar información del lado servidor sin tener que realizar una carga de página, como haríamos con un formulario HTML tradicional. La idea original es que la información proveniente del lado servidor se transmita como un documento XML, y de allí la sigla de esta técnica. Nosotros, sin embargo, utilizaremos JSON como formato de intercambio que es, además, una elección muy común en las aplicaciones reales actuales.

Si bien la sigla se ha popularizado muchísimo en los últimos años, es importante destacar que se trata de una técnica y no de un lenguaje de programación nuevo, ni mucho menos. En otras palabras, AJAX no es más que JavaScript.

Se puede realizar AJAX con el objeto nativo que hemos mencionado, llamado `XMLHTTPReuest` o mediante la librería jQuery, que posee métodos muy fáciles de usar y que son adecuados para la mayoría de los usos habituales. Los métodos abreviados de jQuery encapsulan el funcionamiento del objeto nativo además de prometernos portabilidad entre distintos navegadores y versiones.

Supongamos el siguiente `select`, en donde un usuario elegirá un país, y otro `select` asociado donde queremos colocar las ciudades del país elegido:

```
<!-- se omite el resto del documento HTML -->
<select id="pais">
    <option value="1">Argentina</option>
    <option value="2">Brasil</option>
    <option value="3">Chile</option>
</select>

<select id="ciudad">
</select>
```

El objetivo es, mediante JavaScript, enviar el país elegido hacia el servidor y esperar que éste nos devuelva las ciudades que le corresponden. Una vez recibidas las ciudades, crearemos los `option` dentro del `select` de ciudades para que el usuario elija. Realizaremos esta tarea con AJAX, para que el usuario no tenga que realizar esta selección en dos pasos, como lo haría con un formulario.

Veamos primero el programa que devolverá las ciudades dado un país, programado en PHP y alojado en el servidor:

```
<?php
//ciudades.php
if($_GET['p']==1) $ciu = array("Buenos Aires", "Rosario", "Córdoba");
if($_GET['p']==2) $ciu = array("Rio de Janeiro", "San Pablo", "Bahía");
if($_GET['p']==3) $ciu = array("Santiago", "Valparaíso", "Viña del Mar");
```

```

echo json_encode($ciu);
?>

```

Este sencillo programa envía a la salida (`echo`) un array convertido a JSON. Este array lo hemos creado mediante tres `if` para simplicidad del ejemplo. En una aplicación típica, el programa tomaría el parámetro enviado por GET y realizaría una consulta a la base de datos mediante `mysqli_query`. Entonces, recorriendo el resultado, crearía de igual modo un array con las ciudades correspondientes.

Podemos verificar que este programa del lado servidor funcione adecuadamente solicitando la URL `http://localhost/ciudades.php?p=1`



Figura B.1: Vector JSON enviado a la salida, generado por la función `json_encode` de PHP

Nótese cómo se codifica correctamente el carácter "ó" utilizando el estándar Unicode. Volveremos a nuestro documento HTML en donde ubicamos los `select` y añadiremos el código JavaScript necesario para realizar el AJAX:

```

<!-- CDN -->
<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>

<select id="pais">
    <option value="1">Argentina</option>
    <option value="2">Brasil</option>
    <option value="3">Chile</option>
</select>

<select id="ciudad">
</select>

<script type="text/javascript">

    $("#pais").change( function() {           // (1)
        var enviar = this.value;               // (2)
        $("#ciudad").html("");               // (3)

        $.get("ciudades.php", {p: enviar}, function(recibido) {   // (4)
            recibido = JSON.parse(recibido);           // (5)
            for(i in recibido) {
                var nuevo = document.createElement("option"); // (6)
                nuevo.innerHTML = recibido[i];
                document.getElementById("ciudad").appendChild(nuevo);
            }
        });
    });

</script>

```

En primer lugar, se incluye una etiqueta `script` para incluir la librería jQuery ya que realizaremos el AJAX utilizando sus métodos abreviados. Preferimos utilizar la CDN (*Content Delivery Network*) para evitar tener que descargar el archivo. Con esta forma, el navegador tomará la librería de la URL indicada en el `src`, por medio de una solicitud GET y no ocupará el ancho de banda de nuestro hosting.

En (1) relacionamos una función anónima con el evento `change` del primer `select`, en donde el usuario selecciona un país. Este evento ocurre cuando se *cambia* el ítem que está seleccionado, es decir, cuando se elige efectivamente una opción.

En (2) nos encontramos dentro de la función que maneja este evento, por lo que el momento de ejecución es *posterior* a que el usuario elige un país. La variable `enviar` se inicializa con el `value` del `option` elegido. Recordemos que `this` es, en este contexto, una referencia al originante del evento: el `select` de países. Por lo tanto, `enviar` tendrá el código de país elegido por el usuario (1, 2 ó 3).

En (3), mediante el método `html` de jQuery, eliminamos todas las opciones presentes en el `select` de ciudades. De otra forma, cada elección iría *agregando* en lugar de *reemplazando* las ciudades del `select`. El método `html` de jQuery es similar al miembro `innerHTML` del DOM nativo de JavaScript.

En (4) se realiza propiamente el AJAX. El método `$.get` de jQuery armará y enviará un paquete GET al servidor, con la URL indicada y los datos que le enviamos como segundo parámetro por medio de un JSON. `p` es el nombre del parámetro que el programa PHP espera por GET, y el valor de esta variable será el número de país que seleccionó el usuario. Es interesante notar que este objeto JSON se conforma con una clave "p" y su valor se toma de una variable definida más arriba (`enviar`).

Como tercer parámetro, `$.get` espera una función, que será ejecutada cuando la *respuesta* a la petición llegue desde el lado servidor. Aquí encontramos el **asincronismo**, porque entre el momento en que se envíe el paquete GET al servidor indicando un país y llegue la respuesta con las ciudades que le corresponden pasará un tiempo indefinido, en donde el navegador atenderá otros eventos en lugar de detenerse.

En (5) nos encontramos dentro del cuerpo de la función que debe ejecutarse al llegar la respuesta, por lo que este tiempo que hemos mencionado ya ha transcurrido. En esta línea se convierte el texto recibido en una variable que contiene efectivamente un array con nombres de ciudades (recordemos que en PHP hacemos `json_encode`).

Finalmente, se recorre este array y, (6) por medio de nuevos `option` creados por programa se carga el segundo `select` para selección de ciudades. Esto termina el ciclo.



Figura B.2: El select se carga correctamente con los datos recibidos por AJAX

La librería jQuery nos provee de más funciones para realizar diálogos AJAX más complejos. También se pueden realizar peticiones POST, enviar archivos adjuntos, monitorear el progreso del envío y trabajar sobre los errores de comunicación, entre otros.

Podemos verificar que la cantidad de líneas de programa necesarias para realizar la tarea no son tantas, aunque existe gran complejidad conceptual por detrás. Es importante entender los conceptos de HTTP tales como peticiones y respuestas, parámetros por GET, ejecución de lado cliente y de lado servidor, JSON, jQuery y algunos atributos HTML para comprender el funcionamiento de este ejemplo sencillo.

Como hemos dicho, la complejidad de las aplicaciones web no radica en cada tecnología en particular, sino en la inevitable situación de trabajar con todas ellas simultáneamente. Además, el asincronismo y el trabajo con eventos resulta, aun en este ejemplo tan sencillo, muy desafiante en comparación a la programación imperativa donde los pasos son consecutivos y seguir la ejecución es relativamente simple. La programación de interfaces en el lado cliente es inherentemente compleja.

La técnica AJAX es utilizada en muchísimas interfaces para distintas tareas. Una de las más comunes es la recuperación de información del lado servidor para modificar una parte del documento, como hemos visto con los `select` dependientes de país y ciudad.

En algunas interfaces, AJAX reemplaza por completo a los formularios tradicionales, como sucede en Gmail. Trabajar vía AJAX es más complejo pero desde el punto de vista de la usabilidad es muy conveniente porque es muy veloz. Para el navegador es mucho más fácil realizar estas transacciones HTTP simples que toda una carga de página completa. Esto se traduce en una velocidad de carga mucho mayor.

La técnica AJAX también se utiliza para crear aplicaciones que no serían viables de otro modo, por ejemplo, en sistemas de chat. Sería una pesadilla de usabilidad si una aplicación realizara un envío de formulario por cada mensaje con un interlocutor.

Otro uso posible, aunque exótico, puede ser como herramienta de análisis de conductas. Si nuestro interfaz de lado cliente informa al servidor, mediante envíos AJAX, la posición del mouse cada cierta cantidad de segundos, podremos almacenar en una base de datos el comportamiento del usuario, permitiendo un seguimiento histórico de sus movimientos. Este servicio es ofrecido por varias empresas dedicadas a consultoría de usabilidad de interfaces. Los sucesivos envíos de las posiciones son transparentes para el usuario, ya que son realizadas por AJAX que, por definición, son invisibles para él.

Esta cuestión de la transparencia nos trae hacia una última consideración. Establecer comunicación con el servidor por medio de AJAX es muy conveniente al ser veloz y totalmente manejable por programa. Pero por esta misma razón, también nos aleja de los elementos de interfaz que el usuario conoce en cuanto a estado de la aplicación, tiempos de espera y progreso de cada tarea. Por ejemplo, si el usuario de nuestro simple programa elige un país y, por la razón que fuera, no se logran cargar las ciudades en el segundo `select`, será necesario diseñar un mecanismo para notificar los errores o para indicar que todavía está en curso la petición. Al evitar los formularios tradicionales ganamos en flexibilidad pero también perdemos los elementos de interfaz que resuelven

estos problemas de **comunicación con el usuario**. Si utilizamos AJAX, la responsabilidad de notificar los problemas también corre por nuestra cuenta y los usuarios pueden ser muy intolerantes, principalmente cuando la interfaz *no les dice nada*.

PROHIBIDA LA REPRODUCCIÓN TOTAL O PARCIAL
COPIA PARA USO EDUCATIVO

C. Encodings

(Artículo publicado originalmente en *ForoAlfa.org* en agosto de 2008)

El *encoding* (codificación) es el proceso a través del cual se transforma información textual humana (caracteres alfabéticos y no alfabéticos) en un conjunto más reducido para ser almacenado o transmitido. Podemos nombrar el Código Morse como un sencillo ejemplo que clarificará el concepto de “encoding”: cada letra tiene su correspondencia en forma de sonidos, y todas las letras se codifican con combinaciones de dos signos: el punto y la raya. El conjunto de información se transforma, se reescribe, con un código de sólo dos signos, lo que hace posible una transmisión óptima en canales donde, por ejemplo, sería imposible la transmisión de la voz humana.

En el mundo de las computadoras el encoding asocia nuestros signos alfabéticos y no alfabéticos con ciertos números. Todos los signos que utilizamos al componer un texto en la computadora deben traducirse a estos números si queremos almacenarlos. “Almacenar” en una computadora es una operación fundamental, porque se almacena cuando algo debe mostrarse en pantalla, cuando queremos guardar en un archivo y también cuando queremos transmitir algo a través de una red. Siempre hay almacenamiento, aunque sea muy efímero, en alguna parte del proceso.

Por lo tanto, para la computadora todos nuestros signos serán números y nada más que números. Recordemos que la computadora no es más que una gran calculadora, que sólo “entiende” dos signos: el uno y el cero.

El dilema surge cuando una computadora tiene un conjunto de números que *sabe* representan a un texto y necesita mostrarlos. En ese momento acude a una **tabla de encoding** para reinterpretar de qué caracteres se trataba antes de convertirlos en esos números.

DOS ENCODINGS FAMOSOS

Veamos un primer ejemplo de tabla de encoding: el extendido **ISO-8859-1** (más conocido como *Latin1*, y prácticamente coincidente con *Windows-1252*). Este tipo de encoding utiliza números de 8 bits (un byte) para representar todos los signos. Es decir que todos los signos se transforman en un número entre el 0 y el 255 a partir de una especie de tabla predefinida. En este encoding nuestra letra eñe se transforma en el número 241 (que en lenguaje de computadora es 11110001; nosotros lo representamos en el decimal 241 para hacerlo más manejable).

Otro de los encodings más utilizados, fuertemente recomendado y que se convierte en un estándar, es el **UTF-8**. Este encoding es distinto del anterior ya que no tiene una cantidad fija de bits para representar los caracteres. Utiliza un sistema de largo variable para lograr mayor flexibilidad y ser eficiente. UTF-8 puede representar todos los caracteres de *Unicode*, un estándar creado a fines de los ochenta para codificar todos los caracteres de todas las lenguas escritas del mundo: un total de más de 100 mil signos. En

UTF-8 la eñe se representa con el número hexadecimal C3B1. El modo de cálculo sería muy extenso de explicar, y probablemente poco útil a nuestros objetivos.

LOS ENCODINGS EN LA WEB

Cuando un autor crea los contenidos de su blog está ingresando texto en algún formulario desde su computadora. Ese texto viaja hacia el servidor para ser almacenado en la base de datos. Luego, cuando alguien quiere acceder al artículo el texto se recupera de la base de datos, se coloca en la página y la página se envía de vuelta a otra computadora.

Este relato parece sencillo, pero debemos identificar el rol de los encodings en cada etapa:

- el navegador del autor trabaja con cierto encoding, por lo que al ingresar texto en un formulario, ese texto se convertirá en números de acuerdo a ese encoding.
- el lenguaje de programación (por ejemplo, PHP) que vive en el servidor y recibe el texto que el autor creó también trabaja con cierto encoding y trata a los textos según ese encoding.
- la base de datos que almacena y recupera el texto lo hace con cierto encoding.
- la página que se envía de vuelta al lector también tiene su propio encoding.

Los problemas ocurren cuando alguno de estos encodings *no coincide* con el resto, o cuando alguno de estos sistemas cree que está tratando con textos en cierto encoding cuando realmente es otro. Estos errores son los que llevan a los “caracteres extraños” en nuestras páginas web.

EL EJEMPLO DE LA EÑE

Utilizando nuestro editor favorito crearemos una simple página web que contenga un texto con eñe. El editor de texto también trabaja con un determinado encoding, por lo que elegiremos UTF-8.

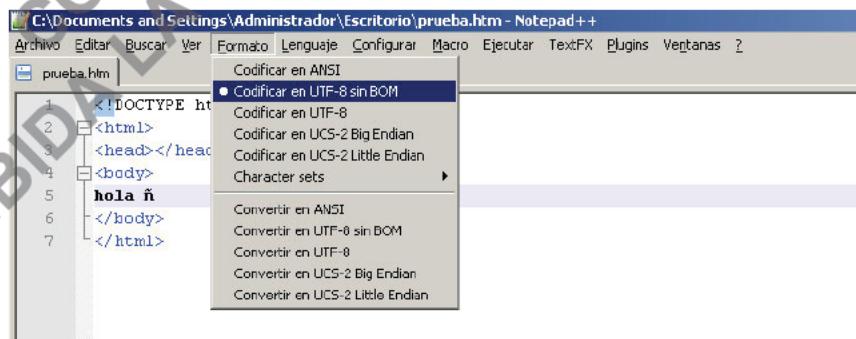


Figura C.1: El editor de texto Notepad++ y la selección de encoding. “ANSI” se corresponde con ISO.

Almacenaremos esta página web de prueba y la abriremos en un navegador. El navegador reconocerá que el archivo está en UTF-8 y mostrará correctamente la eñe.

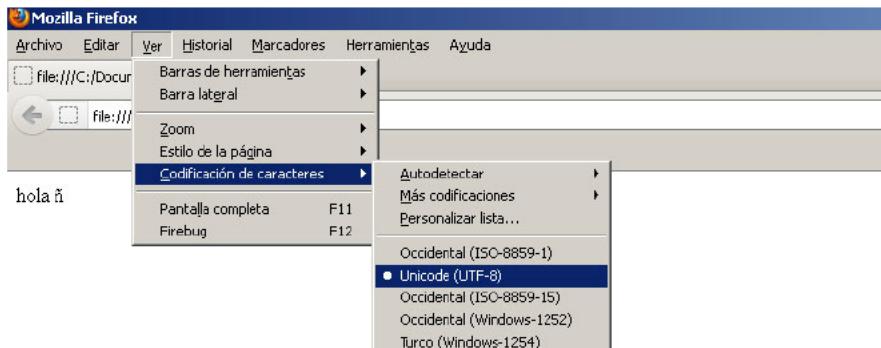


Figura C.2: Encoding UTF-8 correctamente identificado por Mozilla Firefox

Si forzamos al navegador para que interprete este archivo según otro encoding, entonces el resultado es el de nuestras pesadillas:

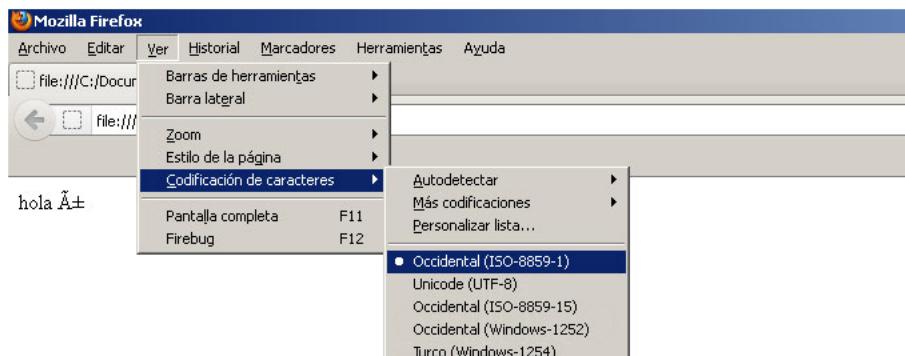


Figura C.3: Un texto UTF siendo interpretado con el encoding erróneo ISO

Si observamos detenidamente este caso, veremos que este comportamiento tiene absoluto sentido para la computadora. En ISO-8859-1, el byte “C3” se corresponde con el signo “Ã” y el byte “B1” con el signo “±”. Recordemos que “C3B1” era la representación de eñe en UTF-8. Lo que ocurre es que se están decodificando los números con una tabla distinta a la que se utilizó para codificarlos. Fácil, ¿verdad?

También se puede probar el caso inverso: crear una página con una eñe utilizando el editor en modo ISO-8859-1 y abrirla en un navegador web indicando, erróneamente, que se trata de un archivo codificado en UTF-8.



Figura C.4: El caso inverso: un archivo ISO decodificado con UTF.

Cuando el navegador utiliza el encoding UTF-8 y no puede interpretar una secuencia de números presente en el documento, nos indica el signo de “caracter desconocido”. Por lo tanto, siempre que veamos este rombo con un signo de interrogación significa que el encoding que está siendo utilizado para la decodificación es de la familia UTF.

Además podemos confirmar que el editor crea documentos en distintos encodings porque el archivo en UTF-8 ocupa 67 bytes en el disco, mientras que los mismos caracteres en ISO-8859-1 ocupan 66 bytes. En este caso de prueba, la diferencia la hace la eñe. Esto es así porque los restantes caracteres presentes en el documento se representan con un solo byte tanto en ISO como en UTF.

EL NAVEGADOR Y LOS ENCODINGS

Es importante notar que el navegador web no pregunta al usuario cuál encoding desea utilizar. Deberá deducir el encoding a partir de la información que la página provea y, en el peor de los casos, deberá adivinar de qué encoding se trata.

Las formas de indicar cuál es el encoding de un documento son las siguientes:

- utilizar una etiqueta `meta` con los atributos `http-equiv` o `charset` en la sección `head` del documento HTML.
- configurar nuestro servidor HTTP (por ejemplo, Apache) para que sirva los documentos con la cabecera `Content-Type` adecuada (ésta es una configuración del *hosting*, y puede no ser accesible para los desarrolladores).
- en documentos XHTML se puede utilizar el atributo `encoding` de la etiqueta `xml`

Si tenemos en cuenta estas posibilidades y los distintos sistemas que listamos más arriba, vemos claramente que son muchas cosas las que pueden salir mal.

La respuesta rápida ante un problema de visualización de una página web es que *alguna* etapa está tratando los textos con un encoding erróneo. Resolverlo es más difícil que enunciarlo, porque hay que investigar dónde está el problema.

Para resumir podemos decir que la mayor parte de las veces lo que ocurre es:

- El navegador interpreta erróneamente el encoding del documento. Lo más probable es que alguna indicación (etiqueta `meta`, cabeceras HTTP, etiqueta `xml`) sea incorrecta. Solución: corregir las indicaciones.
- El navegador no cuenta con la información de qué encoding se trata y adivina, haciéndolo incorrectamente. Solución: agregar las indicaciones.
- El contenido se está almacenando en una base de datos con un encoding que no coincide con el de la página. Cuando el documento llega al navegador es interpretado con un encoding que, en la parte donde ese contenido aparezca, no será el correcto. Solución: corregir el almacenamiento en la base de datos.

CONCLUSIÓN

Los encodings son una característica fundamental de las computadoras. Desde el inicio de la informática los ingenieros debieron representar con números nuestros signos de escritura y, para ello, crearon diferentes tablas de codificación.

Los desarrolladores y diseñadores web serán interpelados por estos conceptos, ya que el futuro de Internet es decididamente multilenguaje, multicultural y multiplataforma.

D. Rutas absolutas y relativas

Los distintos sistemas operativos utilizan los denominados **sistemas de archivo (filesystem)** para almacenar la información en el disco rígido. Estos sistemas de archivo especifican cómo y dónde deben guardarse los datos físicamente para luego ser recuperados. También determinan la forma en que esos archivos se ordenan, de manera que podemos agrupar en directorios o carpetas, unas dentro de otras, jerárquicamente.

Existen muchos sistemas de archivo en el mercado, y los distintos sistemas operativos soportan varios de ellos. Nos centraremos aquí en las generalidades de Microsoft Windows y los herederos de UNIX, entre ellos, Linux. No es nuestro objetivo entrar en detalles técnicos sobre el funcionamiento particular de cada sistema de archivos.

LAS RUTAS

Además de soportar distintos sistemas de archivo (que a su vez tienen sus propios objetivos de diseño), los sistemas operativos determinan cómo “ubicar” un archivo determinado. Esta forma de expresar la ubicación de un archivo dentro del sistema de archivos se denomina **ruta**, algo así como el *camino* para llegar al archivo.

Veamos una ruta típica de Windows:

C:\WINDOWS\Temp\a.txt

Aquí el carácter más importante es la contrabarra (\), que en este sistema expresa la relación entre las carpetas. De izquierda a derecha, cada contrabarra indica que ingresamos a una nueva carpeta, o sea, que lo que se encuentra a la derecha está “dentro” de lo que está a la izquierda. Por lo tanto, viendo la ruta podemos decir:

1. Existe un archivo llamado a.txt
2. Tal archivo se encuentra “dentro” de una carpeta Temp
3. La carpeta Temp, a su vez, está “dentro” de la carpeta WINDOWS
4. La carpeta WINDOWS está “dentro” del disco identificado con la letra C

También es posible que dentro de la carpeta Temp haya más archivos además de éste, y dentro de WINDOWS más carpetas además de Temp... Por lo tanto nos podemos imaginar al sistema de archivos como un árbol.

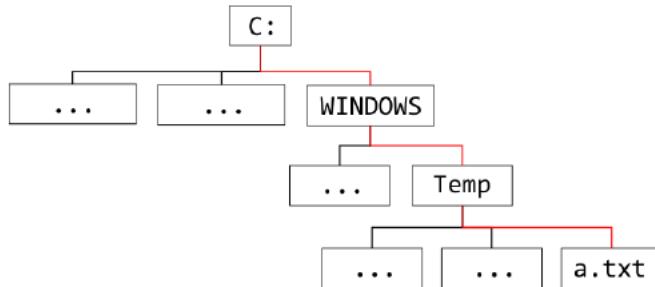


Figura D.1: El sistema de archivos representado como un árbol

Exploraremos ahora una ruta típica de Linux:

/var/www/index.html

Como primera diferencia respecto de Windows encontramos que la barra es diferente. En los sistemas operativos herederos de UNIX el carácter para expresar el “ingreso” a una carpeta “hija” es la barra (y no la contrabarra). Además notamos que no hay una letra seguida de dos puntos al principio, como el caso de Windows. Esto se debe a que estos sistemas no trabajan con **unidades** y “letras de unidad” como Windows, sino que todos los archivos se encuentran en un mismo árbol gigante.

De igual manera podemos decir:

1. Existe un archivo llamado `index.html`
2. Este archivo está dentro de una carpeta llamada `www`
3. La carpeta `www` está dentro de otra carpeta llamada `var`
4. La carpeta `var` está en el nivel superior y ya no hay nada a la izquierda

Los sistemas de archivo tienen un lugar llamado **raíz**, y es justamente donde el árbol de carpetas comienza. A medida que llegamos a los primeros niveles, es decir ascendiendo por el árbol, nos acercamos a la raíz. En los ejemplos anteriores podemos decir:

- La carpeta `WINDOWS` está en el raíz (de la unidad C)
- La carpeta `var` está en el raíz

Es indistinto decir *la raíz* o *el raíz*. Estas diferencias de género provienen del nombre que se suele utilizar en el mundo Unix/Linux: las carpetas se llaman **directorios** y los documentos se llaman **archivos** o **ficheros**. Son éstas diferencias históricas, culturales y de traducción desde el inglés.

Existen además dos archivos especiales en toda carpeta o directorio. Estas dos entidades son:

- El punto `(.)`
- El doble punto `(..)`

El punto es una referencia a la misma carpeta. El doble es también una referencia, pero a la carpeta superior (al *padre*). Estos dos conceptos (el punto y el doble punto) quedarán más claros con el próximo tópico, ya que son necesarios para construir ciertas rutas.

LAS RUTAS RELATIVAS

Dentro de las computadoras es muy común tener que, desde un archivo, referenciar a otro archivo. Ya sea porque es un documento que tiene dentro una imagen, porque es un programa que tiene que llamar a otro, o por muchísimos otros escenarios. Cuando estamos ante esta situación nos encontramos con el problema de “ubicar” el archivo al cual queremos hacer referencia.

Hemos visto ejemplos de **rutas absolutas**, utilizadas justamente para indicar la ubicación de un archivo puntual, es decir, caminos *completos* que parten desde la raíz del sistema de archivos.

¿Pero qué sucede si nuestros archivos hacen referencia a otros mediante rutas absolutas y en algún momento son transportados a otra computadora? ¿Qué sucede si nuestro disco deja de ser el “C” y pasa a ser el “E”? ¿Qué pasa con las rutas si copiamos los archivos a un pendrive? Las rutas absolutas indican el camino completo para localizar un archivo puntual dentro de un árbol con cierta estructura. Si cambia la estructura de las carpetas, esta ruta absoluta dejará de ser válida.

El problema se resuelve con otro tipo de rutas: las **rutas relativas**. Estas son rutas que indican cómo localizar un archivo desde el lugar en el que nos encontramos, sin hacer referencia a todo el árbol sino sólo a una parte: un camino que comienza desde la posición actual y no desde la raíz.

Supongamos que tenemos un archivo creado en un procesador de texto. Si trabajamos en Windows, una posible ruta para nuestro archivo de texto podría ser:

C:\textos\rutas.doc

Esta es una ruta absoluta hacia el archivo. Supongamos que insertamos dentro del texto una imagen que tiene la siguiente ruta absoluta:

C:\textos\imágenes\esquema.jpg

Si dentro del archivo se guardase la ruta absoluta de la imagen insertada, por ejemplo, nunca podríamos mover el archivo a un pendrive, ya que como hemos dicho esta ruta absoluta dejaría de ser una referencia válida para la imagen (porque cambiaría la “letra de unidad”). La imagen no podría ser encontrada.

Pero si en el archivo de texto la ruta se almacena de manera *relativa* sí es posible mover todo el conjunto. La ruta relativa en este caso sería:

.\imágenes\esquema.jpg

Recordemos que el punto representa a “esta misma carpeta”, por lo que esta ruta indica que la imagen a mostrar **esquema.jpg** se ubica ingresando en una carpeta **imágenes** dentro de la misma carpeta actual. La ruta es relativa porque indica cómo llegar a la imagen desde la posición del otro archivo. Ahora sí podemos mover todo el conjunto. La ruta relativa seguirá siendo válida mientras no se altere esta pequeña estructura de

carpetas, es decir, que la imagen se encuentre dentro de una carpeta llamada `imagenes` que está al mismo nivel que `rutas.doc`.

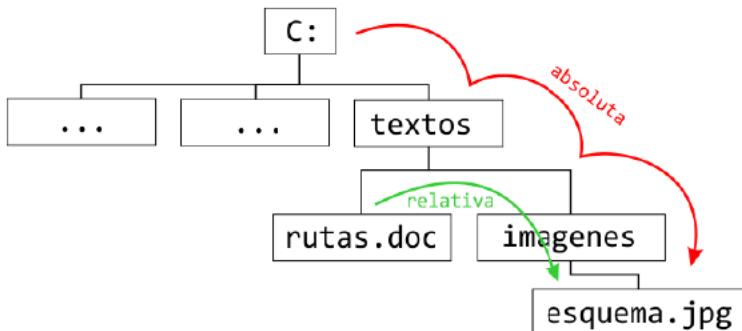


Figura D.2: Rutas absoluta y relativa entre dos archivos (`rutas.doc` y `esquema.jpg`)

Si movemos los archivos a las siguientes posiciones:

```
E:\trabajo\articulos\terminar.doc  
E:\trabajo\articulos\imagenes\esquema.jpg
```

La ruta relativa sigue siendo válida porque no cambió la relación entre los archivos, es decir, su posición *relativa* uno del otro.

RUTAS RELATIVAS Y WEB

Es muy útil el uso de URLs relativas en web, ya que permite desplazar los archivos a distintas carpetas, dominios y servidores, sin tener que preocuparse por romper vínculos entre los documentos. Las URLs son indicaciones sobre la ubicación de un recurso, al igual que las rutas en los sistemas de archivos.

Recordemos que para insertar una imagen en una página web colocamos una etiqueta:

```

```

El navegador buscará la imagen en la URL actual, ingresando a una carpeta `imagenes` y con el nombre de archivo `esquema.jpg`. La URL tiene una forma similar a las rutas que exploramos del mundo Linux, representando el *punto* y los *dos puntos* las mismas referencias, respectivamente, a la misma carpeta y a la carpeta superior.

Si la página web que estamos visualizando, la cual tiene esta imagen, se encuentra en:

www.prueba.com/articulo.html

El navegador buscará la imagen en:

www.prueba.com/imagenes/esquema.jpg

Ahora, supongamos que tenemos una estructura más compleja para la página:

www.prueba.com/articulos/mayo/2000/texto.html

Pero queremos conservar todas las imágenes dentro de la misma carpeta que teníamos antes. Es decir que la nueva carpeta **articulos** estará al mismo nivel (será hermana) de **imagenes**. ¿Cómo referenciar las imágenes en este esquema?

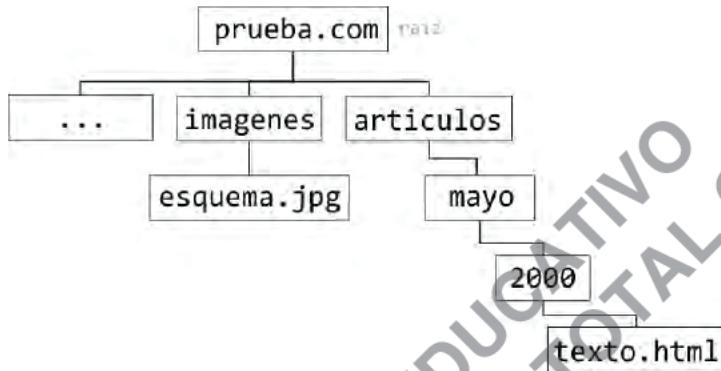


Figura D.3: Esquema de ejemplo

Un documento con esta ubicación:

prueba.com/articulos/mayo/2000/texto.html

necesitaría hacer referencia de esta manera:

`../../../../imagenes/esquema.jpg`

Los pasos son, leídos de izquierda a derecha:

1. antes que nada, reconozco que me encuentro en la carpeta **2000**
2. subo un nivel, llego a la carpeta **mayo**
3. subo otro nivel, llego a la carpeta **articulos**
4. subo otro nivel, llegó a la raíz **(/)**
5. ingreso a la carpeta **imagenes** que se encuentra en la raíz
6. ubico el archivo **esquema.jpg** dentro de esta carpeta

También se pueden contar la cantidad de saltos que deben hacerse, siendo en este caso tres hacia atrás-arriba (acercándose a la raíz) y uno hacia adelante-abajo (entrando en la carpeta **imagenes**).

Índice

Objetivos, herramientas y organización 4

1. HTTP 6

1.1	La Web como tráfico	6
1.2	Arquitectura cliente-servidor	7
1.3	Tipos de paquetes	8
1.4	Métodos HTTP	11
1.5	Códigos de estado	13
1.6	Inspección de tráfico	14
1.7	Productos más utilizados	16
1.8	Ubicación de recursos	16
1.9	Dominios y su traducción a IP	17
1.10	Pasos para publicar un sitio web	19
1.11	Conservación de estado (cookies)	20
1.12	El futuro	21
1.13	Bibliografía	22
1.14	Ejercicios	22

2. HTML 23

2.1	Lenguaje de marcas	23
2.2	Sintaxis de las marcas	24
2.3	Semántica vs. grafica	25
2.4	Estándares	26
2.5	Estructura base	27
2.6	Árbol del documento	29
2.7	Etiquetas para marcado de texto	30
2.8	Etiquetas de agrupamiento	34
2.9	Etiquetas para datos tabulares	35
2.10	Etiquetas de sección	38
2.11	Contenedores genéricos	39
2.12	Código fuente, renderización y espacio blanco	40
2.13	Entidades HTML	43
2.14	Formularios HTML	45
2.15	Campos de formulario	48
2.16	Resumen de etiquetas y atributos más utilizados	52
2.17	Bibliografía	52
2.18	Ejercicios	53

3. CSS 55

3.1	Estándares, versiones y niveles	55
3.2	Formas de inclusión en HTML	57
3.3	Sintaxis	58
3.4	Los selectores	60
3.5	Especificidad de los selectores	63
3.6	Pseudoclases	67

3.7	Selectores avanzados	68
3.8	Herencia de propiedades	69
3.9	Introducción a la maquetación	70
3.10	Propiedades más utilizadas	77
3.11	Bibliografía	78
3.12	Ejercicios	78
4.	JAVASCRIPT	80
4.1	Formas de inclusión en HTML	81
4.2	Cliente/servidor	82
4.3	Sintaxis básica	82
4.4	Utilización de la consola	84
4.5	Variables y tipos	85
4.6	Funciones	88
4.7	Acceso al DOM	92
4.8	Eventos	98
4.9	El objeto String	100
4.10	El objeto Date	102
4.11	Validación de formularios y HTML5	104
4.12	El futuro	105
4.13	Bibliografía	105
4.14	Ejercicios	106
5.	JQUERY	107
5.1	Uso de selectores	107
5.2	Manejo de eventos	109
5.3	<i>Traversing</i>	111
5.4	Otros usos de jQuery	114
5.5	Bibliografía	115
6.	PHP	116
6.1	Programación del lado servidor	116
6.2	Relación con HTML	118
6.3	Variables y tipos	124
6.4	Arrays	126
6.5	Datos de formulario	131
6.6	Conservación de variables	135
6.7	Algunas funciones útiles	137
6.8	División del código en partes	140
6.9	Bibliografía	141
6.10	Ejercicios	142
7.	MYSQL	143
7.1	Persistencia de información	143
7.2	Cliente/servidor	144
7.3	Usuarios y privilegios	145
7.4	Diseño relacional	146
7.5	Tipos de datos	149
7.6	Lenguaje SQL	150
7.7	Uso de phpMyAdmin	151

7.8	Consultas de selección	155
7.9	Consultas de modificación	162
7.10	Manejo de texto	163
7.11	Formato de fechas	164
7.12	Exportación e importación	164
7.13	Consultas desde PHP	166
7.14	Advertencia de seguridad	168
7.15	Bibliografía	169
7.16	Ejercicios	170

APÉNDICES

A.	JSON	171
B.	AJAX	175
C.	Encodings	180
D.	Rutas absolutas y relativas	185

PROGRAMACIÓN WEB

Este libro introduce los *conceptos Web fundamentales*, haciendo uso de las siguientes tecnologías y presentando la relación entre ellas: HTTP, HTML, CSS, JavaScript, jQuery, PHP y MySQL.

Como su nombre lo indica, es un libro pensado para quienes ya tienen *alguna* experiencia en el desarrollo de software. Ya sea para programadores que desean incursionar en Web o para alumnos en una carrera informática, este libro fue pensado para ellos.

Con ejercicios y bibliografía adicional en cada capítulo, el lector encontrará rápidamente las herramientas que necesita para comenzar a implementar aplicaciones de pequeña y mediana escala con un espíritu moderno. Se han incluido apéndices que presentan temas adicionales como JSON y AJAX.

Mauro Gullino es Maestrando en Ingeniería del Software (UNLP) y Licenciado en Comunicación Visual (UCES). Tiene una amplia trayectoria como desarrollador, consultor y docente en diversas universidades y empresas argentinas.

ISBN 978-987-33-1505-0



9 789873 345050