ACERCA DEL AUTOR

El profesor Juan Antonio Recio García es el director del Departamento de Ingeniería del Software e Inteligencia Artificial de la Universidad Complutense de Madrid (UCM). Su pronto interés por las nuevas tecnologías web le ha permitido conocer de primera mano los avances en esta área. Su experiencia docente comienza como profesor de los Cursos de Formación en Informática (CFI), orientados a todos los estudiantes de la UCM, donde se encargó de elaborar y tutorizar los cursos relacionados con Internet y la Web. Más adelante organizó distintos cursos de formación sobre HTML5, CSS3 y JavaScript/JQuery que también estaban dirigidos a estudiantes no necesariamente informáticos. Complementariamente, es profesor de creación de videojuegos con tecnologías web en el Máster de Desarrollo de Videojuegos de la UCM. Actualmente dirige distintos proyectos donde estas tecnologías son parte fundamental. A partir de esta experiencia se ha hecho cargo de la docencia de la asignatura Sistemas Web, impartida en los grados oficiales de la Facultad de Informática de la UCM. Este libro es, en cierta medida, una adaptación de sus apuntes y clases presenciales en la universidad.

PRÓLOGO

La obra que tienes en tus manos nos sumerge en el océano de las tecnologías web, unas tecnologías que se van infiltrando en nuestras vidas cotidianas, que están presentes en nuestros dispositivos electrónicos, desde el ordenador hasta el teléfono móvil, pasando por el frigorífico. Son las tecnologías que se utilizan en Internet y las que debemos conocer para crear espacios de comunicación en ese fascinante mundo.

El libro es el fruto de la experiencia y el trabajo de profesores de un departamento de la Universidad Complutense de Madrid que han ido elaborando y actualizando el material docente durante sucesivos cursos académicos. Se trata del Departamento de Ingeniería del Software e Inteligencia Artificial, el cual yo he tenido el honor de dirigir durante muchos años. El departamento ha sido y es responsable de la docencia de varias asignaturas sobre programación para la Web, desde sus mismos orígenes. Aunque las tecnologías cambian a velocidades vertiginosas, la experiencia docente acumulada durante todo este tiempo por los distintos profesores del departamento es el principal valor de la obra. Toma como base los elaborados materiales de reconocidos catedráticos, como los profesores Baltasar Fernández Manjón, Pedro González Calero o Juan Pavón Mestras. Incluso yo mismo he añadido mi humilde aportación. Esos materiales han sido continuamente refinados y actualizados por una nueva generación de profesores que ha tomado a su cargo la tarea de impartir estas asignaturas. Hoy son varios los profesores de este departamento que imparten asignaturas de programación web en los distintos grados oficiales de la Facultad de Informática. La docencia de forma coordinada en los distintos grupos de esas asignaturas ayuda a mejorar su calidad, al disponer de los mejores materiales, fruto de la participación de los profesores de todos los grupos, y gracias a contar con la experiencia de todos ellos en el diseño de las prácticas y las pruebas de evaluación. Todos los materiales están en continua revisión.

Así, esta obra no debe ser atribuida solo al autor, sino al conjunto de profesores que han colaborado en el material a lo largo del tiempo. Entre ellos es imprescindible destacar al profesor Guillermo Jiménez Díaz, quien elaboró las versiones iniciales de gran parte de este texto. También deben ser incluidos en esta mención los profesores que imparten las asignaturas de desarrollo web de forma coordinada junto con el autor de este libro: Pablo Moreno Ger, vicedecano de Innovación de la Facultad de Informática, e Iván Martínez Ortiz, asesor para Innovación del Vicerrectorado de Tecnologías de la Información de la Universidad Complutense.

Luis Hernández Yáñez
Vicerrector de Tecnologías de la Información
Universidad Complutense de Madrid

1

LAS TECNOLOGÍAS DE LA WEB

El desarrollo de una aplicación web implica toda una amalgama de tecnologías que abarcan tanto la transferencia de la información por Internet como el almacenamiento de información en los servidores y su presentación de forma interactiva en el navegador del usuario. Al principio uno puede agobiarse con tanta sigla extraña (HTML, CSS, HTTP, AJAX, PHP…), pero en realidad todas estas tecnologías son bastante fáciles de entender. Por tanto, antes de entrar en el detalle de cómo desarrollar sitios web vamos a dar un pequeño paseo por todas estas tecnologías una a una. Para ello partiremos de un breve repaso desde el punto de vista histórico que nos permita entender cómo funciona *la Web*.

1.1 UN POCO DE HISTORIA

Existen innumerables libros que narran la historia de la Web con todo lujo de detalles. Pero siguiendo el espíritu pragmático de este libro te propongo un resumen que solo te llevará 30 segundos de tu tiempo. Se trata de un vídeo realizado por Google donde resumen brillantemente la evolución de la Web (además de darse cierto autobombo). Encontrarás el vídeo en el material que acompaña el libro. Primero visualízalo y luego explicamos qué es lo que aparece.

https://www.youtube.com/embed/Jzxc_rR6S-U

Al principio del vídeo aparece una terminal negra donde se escribían las órdenes de los primeros sistemas operativos. Estos sistemas operativos eran aquellos Unix que luego evolucionaron en los Linux actuales como Ubuntu, Fedora, etc. y también son la base de los MacOS. En estos Unix primitivos ya se incluía un revolucionario protocolo denominado *TCP/IP* (¿te suena, verdad?) que permitía

transmitir datos entre máquinas y que había permitido la creación una red mundial de ordenadores interconectados llamada *Internet*.

El comando que vemos ejecutar en el vídeo (*telnet*) permitía conectar a máquinas remotamente. Pero lo más importante es la máquina a la que conecta. Si nos fijamos, veremos que es una máquina del famoso centro de investigación CERN en Suiza (*info.cern.ch*). Es en este centro de investigación donde podemos decir que nació la Web a principios de los 90. Fue creada por un científico británico llamado Tim Berners-Lee, al que se le reconoce el título de padre de la Web, que le ha reportado fama mundial e incluso el título de *sir* de manos de la reina de Inglaterra. Berners-Lee proponía la utilización del *hipertexto* como "un medio para vincular y acceder a información de diversos tipos como una red de nodos en los que el usuario pueda navegar a voluntad". Siendo rigurosos, hay que decir que la idea original se remontaba a los años cuarenta, cuando Vannevar Bush propuso un sistema similar, por lo que no podemos atribuir la "paternidad" de la Web solo a Berners-Lee, sino a muchos otros científicos que sentaron las bases de sus propuestas.

De esta forma, la World Wide Web, WWW o Web (como habitualmente la conocemos) nace como una colección de documentos interconectados entre sí mediante hiperenlaces (más comúnmente llamados enlaces o *links*) y accesibles a través de Internet. Para hacerla realidad se necesitan dos cosas: primero, un estándar o formato para representar estos documentos interconectados, y, segundo, un protocolo para enviarlos desde la máquina que los guarda –el servidor– a la máquina del usuario que quiere visualizarlos –el cliente–. Ambas cosas forman hoy parte de nuestras vidas y las conocemos de sobra. El lenguaje de representación de los documentos que creó Berners-Lee es el "lenguaje de marcado de hipertexto" o, en inglés, *hipertext markup language*, que se resume en el acrónimo **HTML**. El protocolo que permite la transferencia de archivos HTML es el protocolo de transferencia de hipertexto o *hipertext transfer protocol*, cuyo acrónimo es **HTTP**.

Todo esto aparece en el vídeo en menos de un segundo. Podemos ver que se realiza una petición de documento web mediante el protocolo HTTP y para ello se utiliza la orden *GET* de este protocolo. En realidad, y como veremos más adelante, el protocolo HTTP tampoco es mucho más complicado que este GET que recibe la dirección (URL, para ser exactos) del documento que se desea obtener. De inmediato, el vídeo muestra el código HTML del documento obtenido a través de esta petición HTTP.

A partir de aquí iré un poco más rápido, que solo llevamos seis segundos de vídeo ¿Qué se necesitaba a continuación? Pues simplemente visualizar en la pantalla el contenido de estos documentos HTML y permitir "navegar" entre ellos a través de sus enlaces. Es decir: aparecen los navegadores. A partir de ese momento (los años noventa), comienza la carrera en la evolución de los navegadores, que

cada vez incluyen mayores funcionalidades. Toda esta evolución se encauzó a través del *W3C*, el organismo encargado de generar y publicar los nuevos estándares de la web. El vídeo se centra en el estándar de hojas de estilo, en inglés *cascading style sheets* (**CSS**), que permitía mejorar el aspecto de las webs; aunque también aparecieron muchas otras, como los *applets* de Java (programas que se ejecutaban directamente en el navegador y que actualmente están en desuso) o el estándar XML para la transmisión de datos de distinta naturaleza entre el navegador y el servidor.

Además surgieron tecnologías que permitían modificar las páginas web directamente en el navegador y responder así a las acciones del usuario. La que ha prevalecido es *JavaScript*. Este lenguaje de programación, similar a Java, permite realizar tareas "simples" en el navegador sin necesidad de hacer peticiones al servidor.

A principios de la década de 2000 se produce el gran boom de la Web y nace lo que se dio a conocer como *la burbuja de las puntocom*, donde las empresas vinculadas a Internet comenzaron a tener un enorme valor económico, aunque muchas de ellas quebraron o dejaron de operar en poco tiempo. Al margen de esto, aparecen nuevas tecnologías, como XML, los controles y componentes ActiveX de Microsoft, animaciones Flash…, pero, sobre todo, aparece *AJAX*, la tecnología que revolucionó la Web.

Podemos decir que hasta la aparición de AJAX, la Web era algo estático. Visualizabas una página HTML y si querías cambiar algo de su contenido debías solicitar al servidor otra página HTML que incluyese la nueva información y que luego se recargaba completamente en el navegador. Esto no permitía hacer aplicaciones web como las actuales, donde se actualiza su contenido a partir de información obtenida del servidor pero sin tener que recargar toda la página. Por poner un ejemplo, no era posible tener un sistema como Gmail, donde tú pulsas en la bandeja de entrada y parte de la página cambia para mostrarte los mensajes correspondientes que se han descargado de forma dinámica o asíncrona del servidor. El estándar de JavaScript asíncrono con XML (*asynchronous JavaScript and XML* [AJAX]) sí que permitía crear este tipo de páginas web, tal y como podemos ver en el vídeo.

El vídeo continúa hasta 2008, cuando nace HTML5 como la gran evolución del estándar HTML, donde se eliminan muchos elementos anticuados y se abre la puerta a la revolución que han supuesto los dispositivos móviles. HTML5 ofrece la posibilidad de reproducir vídeos o audios directamente en el navegador, crear videoconferencias y, sobre todo, generar gráficos o videojuegos que se ejecutan en cualquier dispositivo. El vídeo nos ofrece varias muestras del potencial de este nuevo estándar. HTML5 sigue en continuo desarrollo, regulado por el W3C, y está consiguiendo superar las diferencias entre distintos navegadores, ya que,

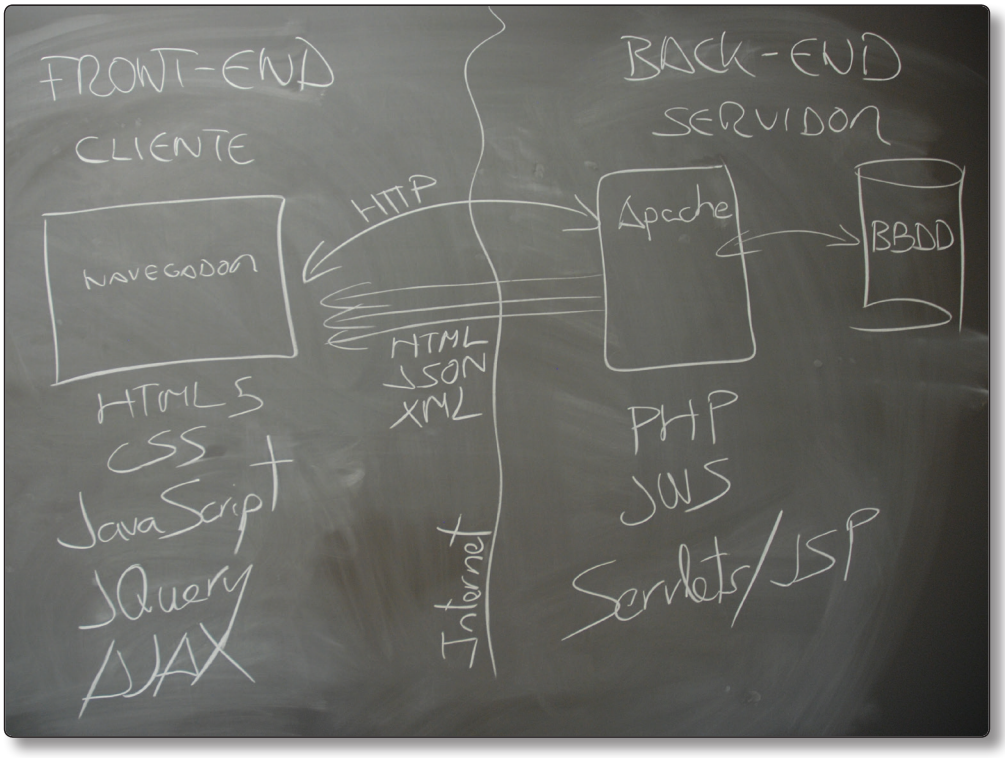
históricamente, cada uno ofrecía ciertas características propias. Es el presente y también el futuro de la Web.

Una vez terminado nuestro "repaso histórico en 30 segundos" quiero explicar una serie de conceptos y tecnologías que son la base de la Web actual.

1.2 ARQUITECTURA CLIENTE-SERVIDOR

Mis alumnos de las clases sobre desarrollo web acaban siempre aburridos de que repita una y otra vez el mismo esquema en la pizarra. Este esquema se muestra en la Figura 1.1 e intenta plasmar las dos partes en que se divide una aplicación web:

- Un cliente –el navegador– que solicita una información y la visualiza al recibirla.
- Un servidor que aloja la información solicitada, normalmente en una base de datos.



1.1. Arquitectura cliente-servidor

Existen distintas tecnologías para implementar cada lado (cliente o servidor) de un sistema web. Tecnologías que aparecen en la Figura 1.1 y que explicaremos en breve.

La parte del cliente encargada de obtener los datos y visualizarlos se denomina **front-end**. Utiliza numerosas tecnologías de las que ya hemos hablado: HTML, CSS, JavaScript, JQuery, AJAX… Por otro lado, el servidor –denominado **back-end**– utiliza otras tecnologías más orientadas al almacenamiento y procesado de datos. En este libro nos centraremos en las tecnologías del *front-end* y solo haremos una breve introducción a PHP y Java. También existen otras muchas tecnologías para crear el *back-end*, como puede ser *NodeJS,* que están muy de moda. Esta opción puede ser una alternativa si no tienes demasiados conocimientos de programación.

1.3 TECNOLOGÍAS DEL FRONT-END

Las tecnologías del *front-end* son aquellas que se utilizan en el lado del cliente, es decir, el navegador, para visualizar e interpretar los documentos que se obtienen del servidor (el *back-end*). El navegador es en realidad una avanzada amalgama de tecnologías que permiten visualizar documentos HTML y ejecutar código que los modifica y permite interactuar con el usuario.

En el siguiente enlace puedes encontrar una vistosa infografía que explica la vertiginosa evolución de los navegadores y la progresiva integración de las tecnologías que constituyen la Web actual.

http://evolutionofweb.appspot.com/#explore

A continuación explicaremos con ejemplos prácticos cada una de estas tecnologías. Puede que al principio no entiendas algunos detalles del código, pero no hay que preocuparse porque entraremos en detalle en los próximos capítulos.

1.3.1 HTML

Como ya he comentado, los documentos de la web visualizados por los navegadores se escriben en el lenguaje HTML (*hypertext markup language*). Este es un formato de texto, por lo que podremos utilizar cualquier editor para crearlos. Como bien indica su nombre, se basa en marcas que indican qué es cada parte

del documento. Siempre hay una marca de apertura para indicar el principio del contenido (`<marca>`) y otra de cierre (`</marca>`) para especificar dónde acaba. Por ejemplo, para indicar el título del documento utilizamos la marca `<title>Este es el título</title>`. Las marcas suelen anidarse. Así, tenemos una marca para indicar todo el contenido del documento `<html>...</html>`, que contiene otras dos para indicar la cabecera `<head>...</head>` y el contenido principal `<body>...</body>`. También hay marcas para establecer los títulos y su importancia `<h1>, <h2>, <h3>,...` añadir imágenes `<img>`, indicar párrafos `<p>`, etc.

Conocer HTML significa saber utilizar estas marcas para establecer la estructura y contenido del documento. En el Capítulo 2 explicaremos en detalle cada una de ellas, pero por ahora simplemente veamos un ejemplo para entender cómo funciona.

Si escribimos el siguiente contenido en un archivo de texto, lo renombramos con extensión *.html* y lo abrimos con nuestro navegador, obtenemos el resultado que se muestra en la Figura 1.2.

```
<html>
  <head>
    <title>Mi primera web</title>
  </head>
  <body>
    <h1>¡Hola Mundo!</h1>
    <p>Esto es un párrafo escrito en HTML.
    También podemos poner <em>parte del texto</em> con énfa-
sis.</p>
    <h2>Se pueden hacer titulos de menor tamaño</h2>
    <h3>Y más pequeños.</h3>
    <p>También se pueden poner imágenes (a menudo, de ga-
tos):</p>
    <img src="gato.png"/>
    <p>También podemos poner enlaces: <a href="masEjemplos.
html">Ver más ejemplos</a> </p>
    </body>
</html>
```

1.2. Visualización del archivo cap01_ejemplo01.html

Para que el ejemplo anterior funcione correctamente, solo necesitamos un editor de texto, como el Bloc de Notas de Windows o similar. Sin embargo, es mucho más práctico utilizar editores especializados, como los que se proponen en la Sección 2.1 del Capítulo 2. Además, también necesitamos una imagen llamada *gato.png* en el mismo directorio donde se encuentre el archivo HTML.

1.3.2 CSS

Las CSS (*cascade style sheets*) u *hojas de estilo* nos sirven para separar el contenido de su presentación. Veamos un ejemplo para entenderlo.

Al principio, HTML tenía etiquetas para indicar el formato de presentación: colores, tamaños de fuente, alineación del texto En el siguiente código establecemos que el color de fondo de todo el documento sea negro (`<body bgcolor="black">`) y luego en cada elemento establecemos que el color de texto sea rojo (`color="red"`).

```
<html>
  <head>
    <title>Mi primera web</title>
  </head>
  <body bgcolor="black">
    <h1 justify="center" color="red">¡Hola Mundo!</h1>
    <p align="justify" color="red">Esto es un párrafo con
    alineación justificada y en rojo.</p>
    <p align="justify" color="red">Esto es otro párrafo con
    alineación justificada y en rojo.</p>
  </body>
</html>
```

Esta forma de mezclar el contenido del documento con la forma en que se presenta no es una buena idea. Imaginemos que hiciésemos toda una web de esta manera y que después de escribir miles de líneas HTML con el color en rojo ahora necesitamos que el texto se muestre en azul. Tendríamos que recorrer todos los documentos HTML sustituyendo la palabra `red` por `blue` y eso sería un proceso muy tedioso y propenso a errores.

Para solucionarlo, las hojas de estilo CSS sirven para indicar el formato de cualquier elemento de nuestro código HTML. Simplemente obviamos toda la información sobre el formato en el código HTML e indicamos el fichero CSS donde la hemos centralizado. El código HTML del documento anterior quedaría así de sencillo y claro:

```
<html>
  <head>
    <link href="estilos03.css" rel="stylesheet" type="text/
css" />
    <title>Mi primera web</title>
  </head>
  <body>
    <center>
      <h1>¡Hola Mundo!</h1>
    </center>
    <p>Esto es un párrafo escrito en HTML con alineación jus-
tificada y en rojo.</p>
    <p>Esto es otro párrafo escrito en HTML con alineación
justificada y en rojo.</p>
  </body>
</html>
```

Hemos quitado toda la información sobre los colores y formato e indicado en la cabecera que todo eso se encuentra ahora el fichero `estilos03.css`. Ese fichero, también de texto, tendría el siguiente contenido:

```
body {
  background-color: black;
}

p {
  color:red;
  text-align: justify;
}

h1 {
  color:red;
  text-align: center;
}
```

Puedes ver el resultado de visualizar este ejemplo en la Figura 1.3. La sintaxis de los archivos CSS no puede ser más sencilla. Primero indicamos el elemento HTML sobre el que queremos establecer formato; luego, entre llaves, indicamos los valores. Ahora, si quisiéramos poner todos los párrafos en azul solo tendríamos que cambiar una línea de este archivo.

1.3. Ejemplo HTML con estilos (cap01_ejemplo03.html)

Al igual que con HTML, para dominar CSS deberemos conocer qué valores podemos establecer, o, al menos, los más importantes, ya que es muy difícil acordarse de todos (para eso ya tenemos a Google).

1.3.3 JavaScript

Recapitulando, por ahora tenemos lo siguiente: HTML, el lenguaje de marcas para estructurar el contenido; y CSS, el lenguaje de estilos para mostrarlo de una forma u otra.

Para hacer un sitio web funcional nos faltaría poder ejecutar código en el cliente (navegador), código que pueda modificar el contenido y formato dinámicamente según el usuario interactúa con la página. Es aquí donde entra en juego *JavaScript*.

JavaScript es un lenguaje ejecutado por el navegador. Su sintaxis es similar a la del archiconocido lenguaje de programación Java –de ahí su nombre–. La idea básica de JavaScript es permitir definir acciones cuando ocurren ciertos eventos en el navegador: la página se ha cargado completamente, el usuario hace un clic, etc.

Para entenderlo veamos un ejemplo sencillo donde contaremos cuántos clics hace un usuario sobre un botón.

Partiremos de una página HTML sencilla donde simplemente tenemos un botón, creado con la etiqueta HTML `<button>`, y un párrafo que hemos identificado como `contador` para poder manipularlo luego con el código JavaScript.

```
<html>
  <head>
    <title>Mi primera web con JavaScript</title>
  </head>
  <body>
    <h1>Ejecución de código JavaScript en el navegador</h1>

    <button type="button">Incrementar</button>

    <p id="contador">0</p>
  </body>
</html>
```

1.4. Visualización de cap01_ejemplo04.html

A continuación creamos el *script JavaScript* que podemos incluir directamente en la cabecera del documento HTML:

```
<html>
  <head>
    <title>Mi primera web con JavaScript</title>
    <script>
      var clicks=0;
      function incrementar(){
        clicks = clicks+1;
        document.getElementById('contador').innerHTML=
clicks;
      }
    </script>
  </head>
  <body>
    <h1>Ejecución de código JavaScript en el navegador</h1>

    <button type="button" onclick="incrementar()">
      Incrementar
    </button>

    <p id="contador">0</p>
  </body>
</html>
```

Entender el *script* es bastante sencillo. Creamos un contador `clicks` que inicializamos a 0 y a continuación una función `incrementar()` que se llamará cuando se pulse el botón. Esta función aumenta en uno el contador de clics (`clicks=clicks+1;`), busca el elemento del documento HTML identificado como `contador` y le cambia el texto, que inicialmente es 0 en el HTML, por el valor del contador (`document.getElementById('contador').innerHTML= clicks;`).

Ahora solo nos queda indicar cuándo se llamará a la función `incrementar()`. Si nos fijamos en el código HTML del botón veremos que hemos añadido que cuando se haga clic se llame a la función incrementar: `onclick="incrementar()"`.

Aunque JavaScript puede llegar a ser un lenguaje complejo, para la gran mayoría de tareas solo necesitaremos conocer un poco de su sintaxis básica que nos permita modificar los elementos de un documento HTML y "enlazarnos" a las distintas acciones que realice el usuario (`onclick` en el ejemplo anterior). Para ello, en el Capítulo 4 haré una introducción muy práctica de los elementos de JavaScript que más nos pueden interesar en la gran mayoría de los casos.

Siempre es necesario conocer algo de *JavaScript*, pero podemos aliviar un poco su dificultad utilizando una librería de este lenguaje que permite hacer las mismas cosas de una forma más sencilla. Esta librería es *JQuery*.

1.3.4 JQuery

JQuery es una librería de JavaScript muy popular entre los programadores web que permite simplificar significativamente el código. Por ejemplo, el código JavaScript del anterior quedaría así:

```
<html>
  <head>
    <title>Mi primera web con JQuery</title>
    <script src="jquery.js"></script>
    <script>
      var clicks=0;
      $(function(){
        $("button").click(function(){
            clicks = clicks+1;
            $("#contador").html(clicks);
        });
      });
    </script>
  </head>
  <body>
    <h1>Ejecución de código JavaScript en el navegador</h1>

    <button type="button">Incrementar</button>

    <p id="contador">0</p>
  </body>
</html>
```

Al principio de la cabecera hemos importado la librería JQuery que se encontrará en el fichero `jquery.js`.

La principal diferencia de este código es que no hemos modificado el código HTML del botón `Incrementar` para indicar qué función ejecutar. Lo hacemos todo desde el código JQuery. Para ello decimos:

1. Búscame un elemento dentro del documento HTML de tipo `button`: `$("button").`

2. Cuando se haga clic en ese elemento, ejecútame la siguiente función: `$("button").click(function(){...}`.

3. Dentro de esa función aumentamos el contador de clics `clicks = clicks+1;` y

4. buscamos un elemento identificado como `contador` y le cambiamos el contenido por el valor del contador de clics: `$("#contador").html(clicks);`

Mediante JavaScript o JQuery vamos a poder interactuar con el usuario; pero, como ya expliqué en la introducción, la verdadera revolución en el desarrollo de aplicaciones web ocurrió cuando se permitió ejecutar código JavaScript en el navegador que obtenía nuevos datos del servidor y los incluía en la página web sin necesidad de recargarla completamente. Esta tecnología es *AJAX*.

1.3.5 AJAX

AJAX (*asynchronous JavaScript and XML*) permite cargar contenido desde el navegador e incluirlo directamente en el documento HTML que se está visualizando. Casi cualquier aplicación web actual utiliza esta tecnología: Gmail, Google Maps, Youtube, Facebook, etc. La forma más fácil de utilizar AJAX es a través de la librería JQuery. Veamos un ejemplo.

Supongamos que estamos haciendo un sitio web para gestionar el correo. No tendría sentido que al cargar la página principal se descargasen todos los correos del usuario por si quiere visualizarlos. En su lugar, la página principal solo mostrará el asunto del mensaje con un botón y haremos que cada correo se descargue –de forma asíncrona– cuando el usuario pulse sobre el botón correspondiente. Simplificando al máximo y limitando a tres mensajes, la página principal sería algo así:

```
<body>
  <h1>Mi gestor de correo</h1>
  <button id="mensaje1">Título del primer mensaje</button>
  <button id="mensaje2">Título del segundo mensaje</button>
  <button id="mensaje3">Título del tercer mensaje</button>
  <p id="contenido"></p>
<body>
```

Supongamos que el contenido de cada mensaje se encuentra en el servidor en sendos archivos de texto: `email1.txt`, `email2.txt` y `email3.txt`. Queremos obtener el archivo de texto correspondiente cuando se pulse en cada uno de los

botones y cargar su contenido en el párrafo identificado como `contenido`. Para ello utilizaríamos el siguiente código JQuery:

```
<script>
$(function(){

    $("#mensaje1").click(function(){
        $("#contenido").load("email1.txt");
    });

    $("#mensaje2").click(function(){
        $("#contenido").load("email2.txt");
    });

    $("#mensaje3").click(function(){
        $("#contenido").load("email3.txt");
    });

});
</script>
```

Toda la "magia" la hace la función `load`. Prácticamente, AJAX es esta función con alguna variante. Primero identificamos el elemento cuyo contenido vamos a cambiar con los datos traídos del servidor. En nuestro caso es el párrafo `contenido`. Por ello utilizamos `$("#contenido")` y a continuación indicamos qué fichero hay que traer del servidor, fichero cuyo contenido se incluirá dentro: `.load(emailX.txt)`.

Juntándolo todo, el ejemplo completo quedaría así:

```
<html>
  <head>
    <title>Mi primera web con AJAX</title>
    <script src="jquery.js"/>
    <script>
      $(function(){

          $("#mensaje1").click(function(){
              $("#contenido").load("email1.txt");
          });

          $("#mensaje2").click(function(){
              $("#contenido").load("email2.txt");
          });

          $("#mensaje3").click(function(){
              $("#contenido").load("email3.txt");
          });
```

```
    });
</script>
</head>
<body>
  <h1>Mi gestor de correo</h1>
  <button id="mensaje1">Título del primer mensaje</button>
  <button id="mensaje2">Título del segundo mensaje</button>
  <button id="mensaje3">Título del tercer mensaje</button>
  <p id="contenido"></p>
<body>
</html>
```

Esto que parece –y es– tan sencillo fue toda una revolución para la Web y nos permite crear cualquier tipo de aplicación interactiva. Pero, obviamente, los datos que traemos del servidor con AJAX no suelen estar en ficheros de texto, sino que se generan de forma dinámica con distintos lenguajes de programación, y normalmente a partir de distintas bases de datos. Esto lo posibilitan las tecnologías del lado del servidor o del *back-end*.

1.4 TECNOLOGÍAS DEL BACK-END

Las tecnologías del lado del servidor permiten generar documentos HTML de forma dinámica. Normalmente se utilizan en conjunto con una base de datos donde almacenamos toda la información de nuestra aplicación web. Veamos las más importantes.

1.4.1 PHP

PHP es un lenguaje de programación que nos permite crear páginas de forma dinámica muy fácilmente. El código PHP ejecutado en el lado del servidor deberá ser interpretado por una aplicación cuando llegue una petición por HTTP. A esta aplicación la llamaremos *servidor web* y en breve aprenderás a instalarla y configurarla. Por ahora imaginemos que tenemos un servidor `"www.miserver.es"` donde hemos añadido código PHP en un fichero llamado `repite.php`. De este modo, para acceder al documento HTML generado con PHP deberemos utilizar la URL `http://www.miserver.es/repite.php`. En este fichero PHP vamos a añadir el código para repetir un mensaje de bienvenida tantas veces como indique el usuario. Y para ello, vamos a utilizar un parámetro en la propia URL llamado `veces`. Por ejemplo, si el usuario quiere que el mensaje se repita tres veces, tendrá que escribir la siguiente URL en el navegador: `http://www.miserver.es/repite.php?veces=3`.

El código que escribiremos en el fichero `repite.php` sería:

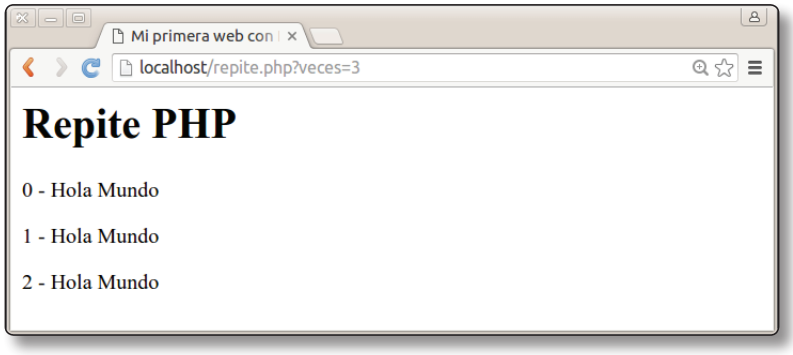
```
<html>
    <head>
        <title>Mi primera web con PHP</title>
    </head>
    <body>
        <h1>Repite PHP</h1>

        <?php
          $repeticiones = $_GET["veces"];

          $contador = 0;
          while ($contador < $repeticiones) {
            echo '<p>' . $contador . ' - Hola Mundo</p>';
            $contador = $contador+1;
          }
        ?>

    </body>
</html>
```

Las marcas `<?php ... ?>` indican el principio y fin del código PHP que se ejecutará cada vez que el fichero sea solicitado por el usuario a través del navegador (es decir, llegue una petición HTTP). En la primera línea obtenemos el valor del parámetro `veces` de la URL: `$repeticiones = $GET["veces"];`. En la siguiente línea creamos un contador inicializado a 0. A continuación, repetiremos mientras el valor del contador sea menor que el valor de repeticiones (`while ($contador < $repeticiones)`) el código que está llaves. En ese código entre llaves generamos los elementos HTML que queramos mediante la palabra clave `echo` y en la línea siguiente aumentamos el contador. En la línea `echo '<p>¡Hola Mundo!</p>';` estamos generando un párrafo que contiene el texto "Hola Mundo". El resultado final sería el que aparece en la Figura 1.5.



1.5. Visualización de repite.php?veces=3

1.4.2 AJAX y PHP

Como ya comenté anteriormente, lo más común es utilizar AJAX en combinación con un lenguaje del servidor tipo PHP. Veamos un ejemplo donde el usuario introduce el número de saludos que quiere recibir en un campo de texto del documento HTML y a continuación se carga por AJAX el contenido HTML con los saludos correspondientes. Este documento HTML con los saludos será generado por el código PHP anterior.

Este sería el código AJAX que se ejecutará en el navegador:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script src="jquery.js"></script>
    <script>
    $(document).ready(function(){
        $("#boton").click(function(){
          repeticiones = $("#veces").val();
          url = "repite.php?veces=" + repeticiones ;
          $("#contenido").load(url);
        });
    });
    </script>
  </head>
  <body>

    <h1>Mi primera web con AJAX y PHP</h1>

    <button id="boton">Saludame</button>

    <input id="veces" type="number" value="2"/>

    <p id="contenido"></p>

  </body>
</html>
```

Empecemos a explicar el código por el cuerpo del documento HTML. Tenemos un botón que pulsará el usuario, identificado como `boton`, y un cuadro de texto donde indicar el número de repeticiones, identificado como `veces`. Dentro del párrafo identificado como `contenido` cargaremos los saludos generados por PHP. Si ahora pasamos a la cabecera, veremos el código JQuery para hacer la llamada AJAX:

1. Decimos que queremos ejecutar algo cuando se pulse el botón: `$("#boton").click(function(){...})`.

2. Obtenemos el valor del cuadro de texto `veces` y lo guardamos en `repeticiones`: `repeticiones = $("#veces").val();`.

3. Creamos la URL correspondiente: `url = "repite.php?veces=" + repeticiones;`. Por ejemplo, si repeticiones vale 3 (porque lo ha introducido el usuario en el campo de texto), la URL sería: `repite.php?veces=3`.

4. Finalmente, cargamos esa URL por AJAX y el contenido obtenido (que se generará con PHP en el lado del servidor) lo insertamos dentro del párrafo `contenido`: `$("contenido").load(url);`.

Este ejemplo muestra el esquema general de infinidad de aplicaciones web. Como vemos, estamos combinando infinidad de tecnologías: HTML, JavaScript, JQuery, AJAX, PHP, y –por supuesto– HTTP. Solo nos faltaría utilizar CSS para ponerlo bonito y ya tendríamos nuestra web interactiva.

Aunque PHP es un lenguaje de *back-end* muy popular, las aplicaciones web más "serias" se hacen con el lenguaje de programación Java. Veámoslo rápidamente sin entrar mucho en detalle, ya que se necesitan conocimientos avanzados de programación para entenderlo completamente.

1.4.3 Java

Java es un lenguaje de programación muy completo y ofrece distintas alternativas para la creación del *back-end* de una aplicación web. Las principales son los *servlets*/JSP y los servicios web tipo REST. Hagamos un rápido repaso.

1.4.3.1 SERVLETS Y JAVA SERVER PAGES (JSP)

Los *servlets* y JSP son similares en funcionamiento al ejemplo anterior en PHP. Ambos sirven para generar el código HTML que se devuelve al usuario.

La versión Java de nuestro `repite.php` sería así en JSP:

```
<html>
    <head>
        <title>Mi primera web con JSP</title>
```

```
    </head>
    <body>
        <h1>Repite JSP</h1>

        <%
          int repeticiones = Integer.parseInt(request.
getParameter("veces"));
          int contador = 0;
          while (contador < repeticiones) {
            out.println("<p>" + contador + " - Hola Mundo</
p>");
            contador = contador+1;
          }
        %>

    </body>
</html>
```

En este caso, el código Java se indica con `<% ... %>` y podemos ver cómo es algo más complejo que en PHP. Por ejemplo, en Java tenemos tipos para indicar qué contiene cada variable. Un número es distinto de una cadena de texto y por eso tenemos que hacer la transformación explícita de cadena a entero con `Integer.parseInt()`. Para dominar Java se necesitan ciertos conocimientos de programación, así que no entraré más en detalle ya que daría para otro libro. Lo que sí es interesante, y comentaré a continuación, son los servicios web en Java.

1.4.3.2 SERVICIOS WEB EN JAVA

Los servicios web consisten en ejecutar código remoto que devuelva ciertos datos, tanto en formato HTML como en cualquier otro como XML o JSON.

Actualmente se utilizan unos servicios web conocidos como **REST**. Estos servicios contestan directamente a las peticiones HTTP y están adquiriendo gran popularidad.

Veamos cómo sería nuestro ejemplo de los saludos implementado como un servicio web:

```
@Path("repite")
public class MyResource {
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String repite(@QueryParam("veces") String nveces)
{
```

```
            StringBuffer sb = new StringBuffer();

            int repeticiones = Integer.parseInt(nveces);
            int contador = 0;
            while (contador < repeticiones) {
               sb.append("<p>" + contador + " - Hola Mundo</p>");
               contador = contador+1;
            }

            return sb.toString();

        }
    }
```

Dejando de lado algunos detalles del lenguaje Java, el código es bastante intuitivo:

- La línea `@Path("repite")` indica que este código se ejecutará cuando se solicite el archivo `repite` por HTTP. Por ejemplo: `http://www.miserver.es/repite` (vemos que, al contrario que antes, en `repite.php` no se utiliza extensión).

- La línea `@Produces(MediaType.TEXT_HTML)` establece que vamos a generar código HTML.

- A continuación creamos un método llamado `repite` que devuelve una cadena de texto `String` que contendrá el código HTML. Este método recibe un parámetro `nveces` que obtendremos de la URL mediante el parámetro `veces` de la misma: `@QueryParam("veces")` (recordemos que la URL sería `http://www.miserver.es/repite?veces=3`).

- El resto del código ya nos es bastante familiar. La única diferencia es el `StringBuffer` que utilizamos para ir almacenando el código HTML generado y cuyo contenido devolvemos al final como una cadena de texto.

Los servicios web REST en Java facilitan mucho la creación de aplicaciones avanzadas al permitir generar cualquier tipo de documento, tanto en HTML como en cualquier otro formato para el intercambio de datos. Veamos los dos formatos más importantes actualmente.

1.5 TECNOLOGÍAS DE INTERCAMBIO DE INFORMACIÓN

Ahora que ya hemos visto las tecnologías de ambos lados de una aplicación web, hay que hablar de los distintos formatos de archivo para intercambiar datos entre el *front-end* y el *back-end*. En la gran mayoría de los casos, el formato utilizado es HTML y las tecnologías del *back-end* generan directamente el código que se va a visualizar por el navegador.

Sin embargo, existen situaciones donde eso no es recomendable y es preferible utilizar algún formato "más neutro" y que no esté ligado a su visualización por un navegador. Por ejemplo, imaginemos que queremos desarrollar la aplicación web de un periódico digital y creamos para el *back-end* un servicio web en Java o PHP que nos devuelve el contenido de la noticia como HTML. En el lado del cliente (*front-end*), el navegador puede cargar por AJAX la noticia y mostrársela al usuario directamente. Perfecto. Pero ¿y si queremos reaprovechar toda esta infraestructura para crear una *app* para dispositivo móvil que muestre las mismas noticias? Esta *app* podría conectarse muy fácilmente a nuestro servicio web para obtener las noticias, pero como no es una página web le sobraría toda la información del formato HTML, ya que todo el contenido se va a visualizar de una forma completamente distinta. Como HTML es un formato orientado a la "visualización" de documentos web, existen casos, como el anterior, donde sería mejor utilizar algún formato "más neutro" u orientado solo a los datos, donde se omita todo tipo de detalle sobre el formato. Dos ejemplos hoy muy populares de estos formatos son XML y JSON.

De esta forma, sea cual sea la tecnología con que visualicemos los datos (una web o una *app*), los datos serán independientes de ella y, por tanto, nuestro *back-end*, mucho más reutilizable. Luego cada tecnología del *front-end* se encargará de formatear los datos de una forma u otra. Por ejemplo, nuestra aplicación web podría pedir los datos con AJAX, recibiendo un fichero JSON que luego transformaría en código HTML.

1.5.1 XML

XML (*extensible markup language*) es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) que se utiliza para almacenar datos en forma legible.

XML no nació solo para aplicarse en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas.

En realidad, el formato HTML es un subconjunto de XML. Eso quiere decir que XML es un lenguaje de marcas igual que HTML, pero donde no se limitan las marcas que se pueden utilizar (como sí se hace con HTML, donde solo podemos usar las marcas indicadas en el estándar: `<body>`, `<h1>`, `<p>`, etc.). Veamos un ejemplo de documento XML que podría generar nuestro *back-end* de noticias:

```
<Noticias>
  <Fecha>3 septiembre 2016</Fecha>
  <Noticia>
    <Titulo>Título de la noticia 1</Titulo>
    <Resumen>Resumen de la noticia 1 ...</Resumen>
    <Contenido>Contenido completo de la noticia 1 ...</Conte-
nido>
  </Noticia>
  <Noticia>
    <Titulo>Título de la noticia 2</Titulo>
    <Resumen>Resumen de la noticia 2 ...</Resumen>
    <Contenido>Contenido completo de la noticia 2 ...</Conte-
nido>
  </Noticia>
  <Noticia>
    <Titulo>Título de la noticia 3</Titulo>
    <Resumen>Resumen de la noticia 3 ...</Resumen>
    <Contenido>Contenido completo de la noticia 3 ...</Conte-
nido>
  </Noticia>
<Noticias>
```

1.5.2 JSON

JSON (*JavaScript object notation*) es un formato ligero para el intercambio de datos. El "problema" de XML es que las marcas de inicio y cierre son, en cierta manera, redundantes y suponen aumentar significativamente la cantidad de datos que hay que transmitir. Con JSON se transmite menos información, gracias a lo cual aumenta la velocidad de recepción y se consume un menor ancho de banda.

Otra característica importante de JSON es su fácil manipulación con JavaScript (de ahí su nombre), lo que lo hace idóneo para combinarlo con AJAX.

Veamos cómo quedaría el ejemplo anterior en JSON:

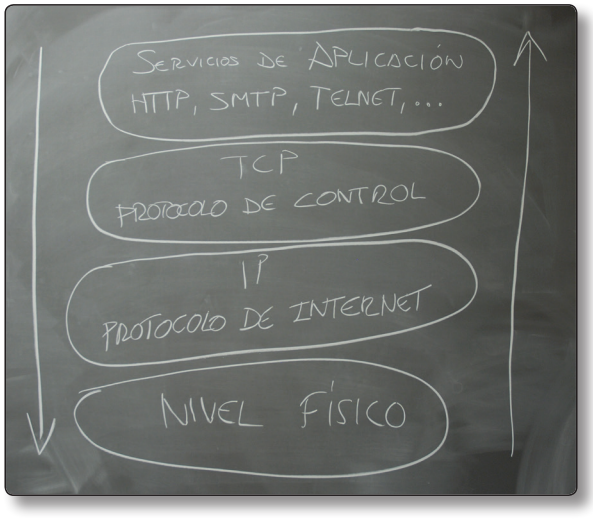
```
"Noticias": {
  "Fecha": "3 septiembre 2016",
  "Noticia": {
    "Título": "Título de la noticia 1",
    "Resumen": "Resumen de la noticia 1 ...",
```

```
      "Contenido": "Contenido completo de la noticia 1 ..."
    }
    "Noticia": {
      "Título": "Título de la noticia 2",
      "Resumen": "Resumen de la noticia 2 ...",
      "Contenido": "Contenido completo de la noticia 2 ..."
    }
    "Noticia": {
      "Título": "Título de la noticia 3",
      "Resumen": "Resumen de la noticia 3 ...",
      "Contenido": "Contenido completo de la noticia 3 ..."
    }
}
```

Una vez que hemos conocido tanto las tecnologías del *front-end* como las del *back-end*, lo siguiente es entender cómo se transmiten los datos de un lado al otro. Aunque entremos en un campo más técnico, es necesario comprender bien cómo funciona HTTP para poder utilizar formularios web, *cookies*, etc.

1.6 TECNOLOGÍAS DE TRANSFERENCIA DE DATOS WEB

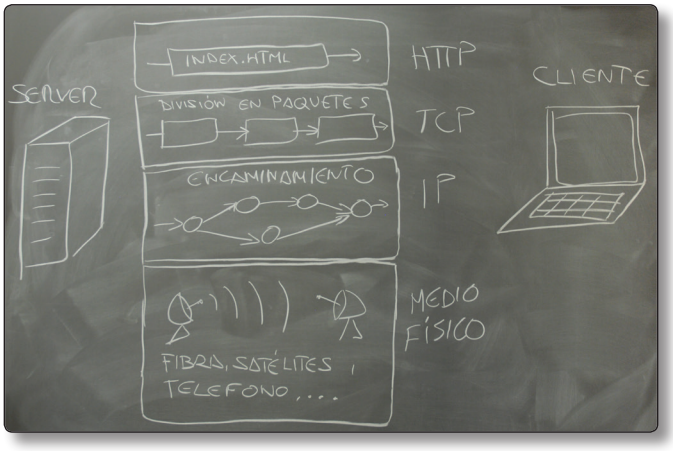
La transferencia de información entre el cliente y el servidor se basa en los distintos protocolos de red que construyen Internet. Como transferir información entre ordenadores es una tarea compleja, se dividió el problema en distintas capas, cada una de las cuales se encarga de una característica concreta. En la Figura 1.6 se muestra cómo se hizo esta división.



1.6. Modelo de capas de Internet

Dejando aparte la capa física encargada de transmitir la información (ethernet, fibra óptica, wifi, etc.), empezaremos por el *protocolo IP* (*Internet protocol*). Se encarga del encaminamiento de paquetes de datos entre las distintas máquinas y utiliza las *direcciones IP* para identificar a cada una de ellas. Las direcciones IP son cuatro números separados por un punto para identificar unívocamente a cada máquina. Por ejemplo, la IP 147.96.1.15 es la dirección en Internet del servidor web de la Universidad Complutense. Prueba a escribirlo en la barra de direcciones de tu navegador, verás que accedes a esa página web. En realidad, nuestras máquinas solo necesitan saber la dirección IP para poder comunicarse entre ellas, pero los creadores de Internet pronto se dieron cuenta de que los humanos somos realmente torpes recordando todas estas direcciones IP; por eso decidieron asignarles un "alias" fácil de recordar. Así surgieron los nombres de Internet, cada uno de los cuales está asociado a una IP. En nuestro caso, el alias de 147.96.1.15 es **www.ucm.es**. Existen IP y alias especiales; por ejemplo, **127.0.0.1**, cuyo nombre es **localhost** y que representa la propia máquina en la que estamos. Esto lo utilizaremos más adelante para visualizar las páginas que estemos desarrollando.

El protocolo IP se complementa con otro que garantiza la llamada de los paquetes y que se denomina TCP (protocolo de control del transporte o *transport control protocol*). Juntos forman los famosos protocolos TCP/IP que todo sistema operativo sabe "hablar" para comunicar datos con otras máquinas. A partir de ellos cada aplicación concreta creará sus propias normas para transferir la información, ya que no es lo mismo obtener una página web del servidor que enviar un correo electrónico. Estos protocolos son los llamados protocolos de aplicación. Existen innumerables protocolos para cada aplicación concreta; por ejemplo el protocolo DNS para traducir de nombre a IP. Sin embargo, HTTP es el protocolo más popular, ya que define las reglas para obtener archivos web. La Figura 1.7 intenta ilustrar todo este proceso de transmisión de información.



1.7. Ejemplo de transmisión de un fichero en Internet

Existiendo distintos protocolos para acceder a la información, innumerables máquinas que la alojan e incontables recursos en cada una de ellas, ¿cómo indica el cliente a qué servidor quiere hacer la petición y a qué recurso del servidor quiere acceder? Pues mediante un identificador único conocido como **URL** y del que hablaré a continuación.

1.6.1 Anatomía de una URL

Una **URL** (*uniform resource locator*) es la forma de identificar de manera única un recurso en Internet. Tiene el siguiente aspecto:

```
protocolo://dominio:puerto/ruta#fragmento?param1=val1&param2=
val2...
```

- `protocolo` es el lenguaje en el que se van a comunicar cliente y servidor. Como ya dijimos, el más común es el protocolo `http`, aunque existen otros.
- `dominio` es el nombre del servidor.
- `puerto` es el canal de comunicación con el servidor. Cada protocolo de aplicación tiene asignado un puerto distinto, ya que nos podemos comunicar con una misma máquina mediante distintos protocolos. Por defecto, el protocolo HTTP utiliza el puerto 80 (por eso no lo tenemos que escribir nunca).
- `ruta` indica el nombre del recurso al que queremos acceder en el servidor. Las rutas vacías (/) suelen indicar una ruta a un archivo por defecto que se configura en el servidor. Sin embargo, como sabemos por experiencia, las rutas suelen ser más largas.
- `#fragmento` se refiere a una parte concreta de la página y no hace ninguna petición al servidor (se usará para hacer algunas cosas en JavaScript).
- `?param1=valor1` son los parámetros de consulta. Es una forma de pasar información extra al servidor que ya hemos visto en los ejemplos de la sección anterior cuando indicábamos el número de repeticiones del saludo (`http://www.miserver.es/repite.php?veces=3`). Si pasamos más de un parámetro, entonces los separaremos con el símbolo `&`.

Una vez que conocemos cómo se identifican recursos en la web, podemos explicar el protocolo fundamental de toda aplicación web: el protocolo HTTP.

1.6.2 El protocolo HTTP

El lenguaje que los clientes y servidores web utilizan para comunicarse entre sí se conoce como **HTTP** (protocolo de transferencia de hipertexto). Todos los clientes y servidores web deben ser capaces de "hablar" HTTP para enviar y recibir documentos hipermedia.

Para que una máquina conectada a Internet pueda ofrecer archivos mediante HTTP, necesitamos ejecutar en ella un programa servidor de HTTP. Tanto a este programa como a la máquina que lo ejecuta se les denomina servidor web. En breve aprenderás los detalles del protocolo y cómo arrancar un servidor web en cualquier ordenador.

1.6.2.1 FUNCIONAMIENTO DE HTTP

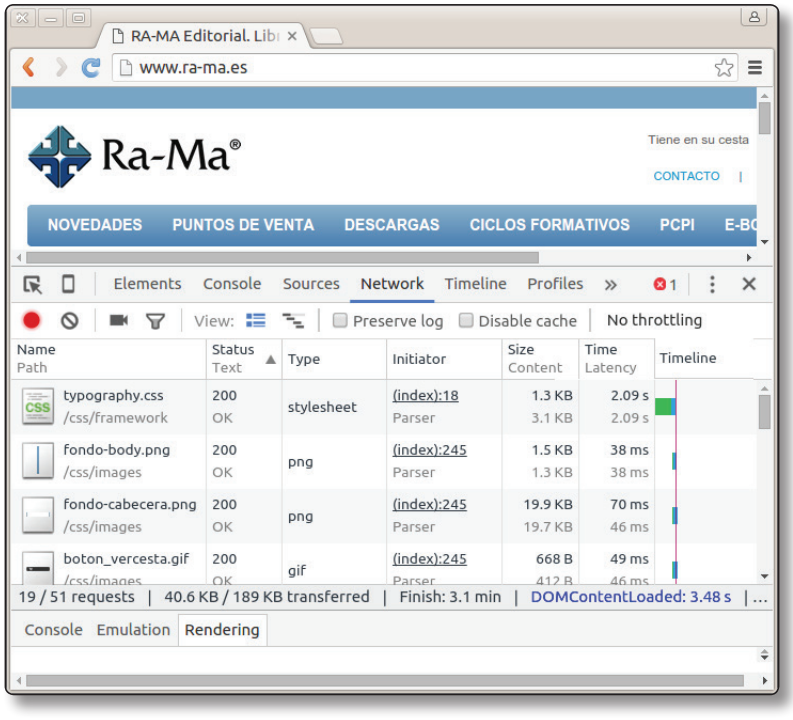
HTTP es un protocolo de comunicación de tipo petición-respuesta sin estado cuya operación básica es la siguiente:

1. Una aplicación ejecutada por un cliente (habitualmente un navegador web) se conecta al servidor web.
2. A través de la conexión, el cliente envía la petición codificada como un fragmento de texto.
3. El servidor web analiza la petición y localiza el recurso especificado.
4. El servidor envía una copia del recurso al cliente a través de la conexión.
5. El servidor cierra la conexión.
6. El navegador interpreta los datos recibidos del servidor y muestra al cliente el documento solicitado.
7. A veces el documento pide descargar datos adicionales (por ejemplo, imágenes). Para estos archivos, se abren nuevas conexiones HTTP independientes.

De esta forma, en el funcionamiento de HTTP intervienen dos actores distintos: el servidor HTTP y el cliente HTTP. Los clientes de HTTP son simplemente los navegadores web. La elección de un navegador web no es una decisión trivial a la hora de desarrollar una aplicación web; de hecho, deberíamos probar nuestro código web en cada uno de ellos para asegurarnos su compatibilidad. Siendo prácticos, lo mejor es utilizar el navegador más extendido para hacer el desarrollo y luego probar

con el resto. Por tanto, y teniendo una cuota de usuarios estimada en el 60 %, lo más aconsejable es utilizar Google Chrome. Esta elección no se basa solo su popularidad, sino en las herramientas de desarrollo que incluye.

Si eliges la opción **Herramientas para desarrolladores** de tu Chrome, aparecerán una serie de ventanas como las que se muestran en la Figura 1.8. Una de ellas nos permite, por ejemplo, visualizar todas las peticiones HTTP que se realizan entre el propio Chrome y el servidor web cuando introducimos una URL. Selecciona la pestaña de **Red** (Network) e introduce una URL cualquiera, por ejemplo: **www.ra-ma.es**. A continuación aparecerán las distintas peticiones HTTP enviadas y recibidas desde el servidor. Si echas un vistazo verás que se descargan los distintos elementos de un sitio web: archivos HTML, imágenes, hojas de estilo CSS, código en JavaScript…



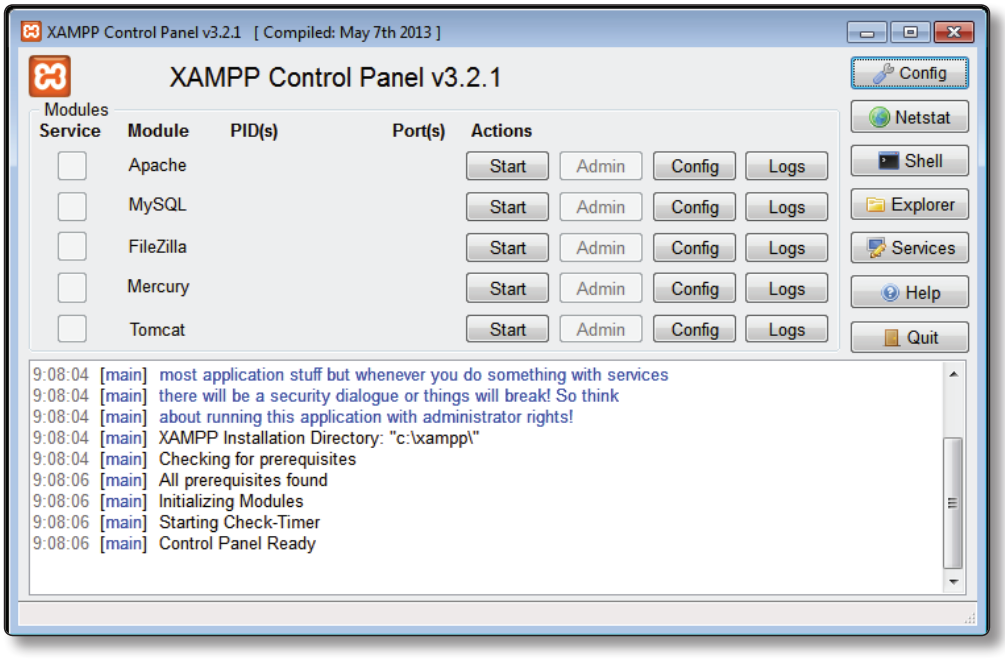
1.8. Información sobre tráfico HTTP en Chrome

Instalar y usar Chrome es muy fácil para cualquier tipo de usuario; sin embargo, para poder desarrollar sitios web necesitamos instalar y utilizar el software del "otro lado" de la conexión: el servidor HTTP. Esto no es tan obvio, así que veamos cómo hacerlo.

1.6.2.2 SERVIDORES HTTP

Hoy tampoco resulta demasiado complicado ejecutar un servidor HTTP en nuestro PC. El software más popular, que además es gratuito, se denomina Apache. Existen innumerables manuales en la web para aprender a instalarlo y utilizarlo. Tanto Linux como Mac vienen con un servidor **Apache** fácil de instalar y ejecutar que podemos utilizar como servidor web. Sin embargo, para Windows es algo más complejo, por lo que en la Facultad de Informática de la Universidad Complutense utilizamos un software que facilita su arranque, denominado XAMPP. Aquí quiero hacer un inciso sobre el software que recomendaré en este libro. Como casi todo, es una cuestión de gustos y existen numerosas alternativas, tanto libres como de pago. Es decisión de cada uno elegir el software que más se adapte a sus necesidades. Yo, personalmente, prefiero dar varias opciones y dejar a mis alumnos elegir el que más les guste, sin imponer ninguna opción concreta. Lo que haré a lo largo del libro es indicar qué programas son los más utilizados por mis compañeros y alumnos, dando por sentado que las opciones más utilizadas por ellos son las más recomendables.

Como indicaba, XAMPP es muy buena opción para arrancar un servidor web de forma fácil. Una vez descargado e instalado XAMPP, iniciar servidor es muy simple: abrimos el **Panel de control** y damos al botón **Start** que hay junto al servicio llamado Apache (véase la Figura 1.9).



1.9. Panel de control de XAMPP

La prueba de que está funcionando es que al escribir la URL `localhost` (alias de 127.0.0.1), en nuestro navegador aparecerá una pantalla de inicio como la de la Figura 1.10.

1.10. XAMPP funcionando

Dentro de la opción **config** podremos seleccionar el directorio de nuestro servidor donde se encuentran los documentos que haya que servir mediante HTTP. Es en este directorio donde tendremos que copiar los archivos HTML que queramos "servir" por HTTP. Por ejemplo, si elegimos la carpeta `c:\web` para servir los archivos y copiamos allí el archivo `principal.html`, la URL que tendremos que escribir en el navegador sería `http://localhost/principal.html`. Por supuesto, también podemos utilizar subdirectorios. En este caso, si el archivo estuviere en `c:\web\sitio1`, la URL sería `http://localhost/sitio1/principal.html`.

Como ejercicio, emplea el código HTML del ejemplo que vimos en la sección anterior para crear **principal.html** y publicarlo a través de tu propio servidor web en **localhost**.

Aprovecho este punto para indicar un error típico que aparece al desarrollar páginas web en nuestro PC. Normalmente tendemos a abrir los archivos HTML pulsando sobre ellos con el ratón. El navegador los abre y los visualiza, pero en este caso no estamos utilizando el protocolo HTTP para cargarlos, sino que se leen directamente del disco. Por eso la URL no es `http://localhost/principal.html` sino `file://c:\web\principal.html`. Aunque al principio nos parezca que da igual, cuando añadamos funcionalidad extra, como AJAX, no funcionará a menos que visualicemos el documento HTML a través de HTTP. Por tanto, **siempre** deberemos

abrir las páginas HTML que desarrollemos escribiendo la URL correspondiente mediante HTTP.

El software Apache es el que normalmente se emplea en servidores web de Internet. Así que podríamos utilizar nuestra propia máquina como servidor web para el público. Sin embargo, siempre son necesarias muchas tareas de mantenimiento y seguridad, por lo que se suele contratar un servicio profesional de *hosting*.

1.6.2.3 PETICIONES HTTP

Cuando un navegador utiliza el protocolo HTTP para comunicarse con el servidor, formula una serie de *peticiones* que constan de:

- Un método de petición (GET, POST, etc.)
- Una URL (http://…)
- Datos adicionales sobre la petición (IP, Referer, Timestamp, UserAgent, etc.).

Existen dos tipos de petición principales: GET y POST. La petición GET solicita el recurso identificado por la URL y es el mecanismo más habitual. De hecho, si vuelves a fijarte en la ventana de tráfico de red de Chrome comprobarás que en el método (Method) siempre aparece GET. Cuando hacemos una petición GET y tenemos que enviar información al servidor, esta se incluye en la URL como parámetros. Ya vimos en la introducción el ejemplo que repite el saludo tantas `veces` como indiquemos en la URL. Otro ejemplo fácil es el buscador de Google. Simplemente tenemos que añadir el parámetro `q=` al final de la URL: `https://www.google.es/search?q=ra-ma` nos devuelve todos los resultados que contengan el término "ra-ma". Otro claro ejemplo es el cuadro de búsqueda en la página web `www.ra-ma.es`; fíjate en la URL que aparece cuando realizas una búsqueda.

Muchas veces resulta poco recomendable que los datos enviados al servidor aparezcan en la URL. Es el caso de los formularios o cuando enviamos datos demasiado largos, como, por ejemplo, archivos. En ese caso es mejor utilizar el método POST, que incrusta los datos en la petición (paquete de datos) enviada al servidor. Aquí es importante tener claro que, aunque los datos enviados no aparezcan en la URL, siguen siendo fácilmente legibles por cualquiera que esté "escuchando" nuestra red TCP/IP, ya que HTTP se basa en peticiones de texto plano. Si queremos que los datos no puedan ser leídos por nadie (contraseñas, datos de transacciones económicas, etc.) deberemos encriptar la comunicación mediante HTTP seguro (HTTPS).

El protocolo HTTP es bastante sencillo y podemos utilizarlo sin necesidad de un navegador. Simplemente podemos abrir una conexión con el servidor web y escribir nosotros "a mano" los mensajes HTTP que queramos transmitir. Esto nos permite entender mejor cómo funciona este protocolo, así que a continuación haremos un pequeño ejercicio que consiste en "hablar" HTTP con nuestro servidor web.

Para conectar con el servidor podemos utilizar la aplicación **telnet**. Esta aplicación es un "dinosaurio" de la informática que ha sobrevivido hasta nuestros días, ya que nació a la vez que Internet y hoy apenas se utiliza. De hecho, ya hablamos de ella al explicar el vídeo que sirvió para repasar la historia de la Web. Esta herramienta simplemente abre una conexión TCP con el servidor y puerto indicado permitiéndonos enviar y recibir texto. En los sistemas operativos Linux y MacOS suele venir preinstalada para utilizarla a través de la consola de comandos. Para conectar con nuestro propio servidor web solo tendremos que ejecutar **telnet localhost 80**. En Windows también existe una aplicación nativa **telnet**, pero es mejor utilizar un programa específico como puede ser PuTTy (*http://www.putty.org/*). De nuevo, configura el *host* como *localhost* y el puerto *80* para conectar. Dependiendo del servidor utilizado tendrás que activar la casilla de modo *raw*.

Una vez que estemos conectados al servidor podremos enviarle mensajes HTTP. Un mensaje HTTP es un fragmento de texto que consta de los siguientes elementos:

- Línea inicial.
- Líneas de cabecera.
- Línea en blanco (CRLF).
- Cuerpo de mensaje opcional (un fichero, solicitud de datos, datos resultado de una solicitud).

La línea inicial indica qué tipo de mensaje estamos enviando. Ya conocemos los principales (**GET** y **POST**), aunque también existen otros menos utilizados:

- **HEAD**: como GET pero pide al servidor que solo envíe la cabecera de la respuesta. Es decir, no envía el contenido de la respuesta.
- **PUT**: sube archivos en el cuerpo de la solicitud
- **DELETE**: borra el archivo especificado en el campo URL (si el servidor le deja, claro).
- **Otros**: OPTIONS, TRACE y CONNECT…

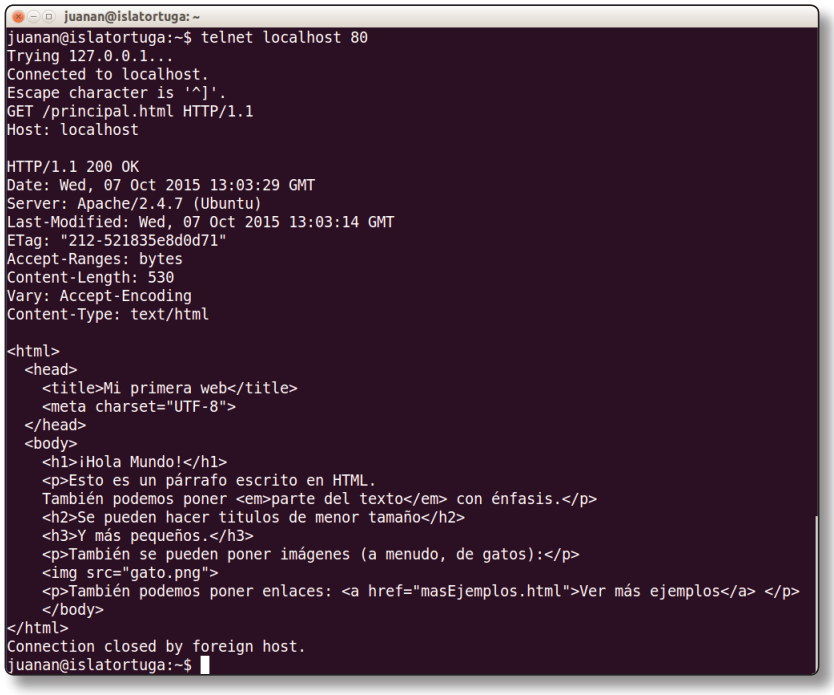
A continuación de la línea inicial vienen las líneas de cabecera, que siempre tienen la estructura **Nombre-cabecera: valor**. Existen hasta 46 cabeceras distintas para identificar el navegador, sistema operativo, *cookies*, etc. Pero solo una es obligatoria: la que identifica al *host* al que estamos conectando. Por ejemplo: **Host: localhost** para contactar con nuestro propio servidor web.

Veamos un ejemplo completo. Supongamos que queremos obtener el fichero que hemos creado anteriormente y que está en la URL `http://localhost/principal.html`. Una vez conectados con telnet tendremos que escribir el siguiente texto:

```
GET /principal.html HTTP/1.1
Host: localhost
```

Para enviar la petición tendremos que pulsar **Intro** dos veces. Vemos que a continuación del tipo de la petición estamos indicando la versión del protocolo HTTP que vamos a utilizar (la versión 1.1 es la común actualmente).

En la Figura 1.11 podemos ver un ejemplo de cómo simular una petición HTTP con telnet utilizando la consola de Linux.



1.11. Ejemplo de petición GET con telnet

1.6.2.4 RESPUESTAS HTTP

Cuando el servidor web recibe una petición, debe devolver una respuesta HTTP que contiene los siguientes elementos.

- Código de respuesta.
- Datos de cabecera.
- Cuerpo.

Los códigos de respuesta más habituales son las siguientes:

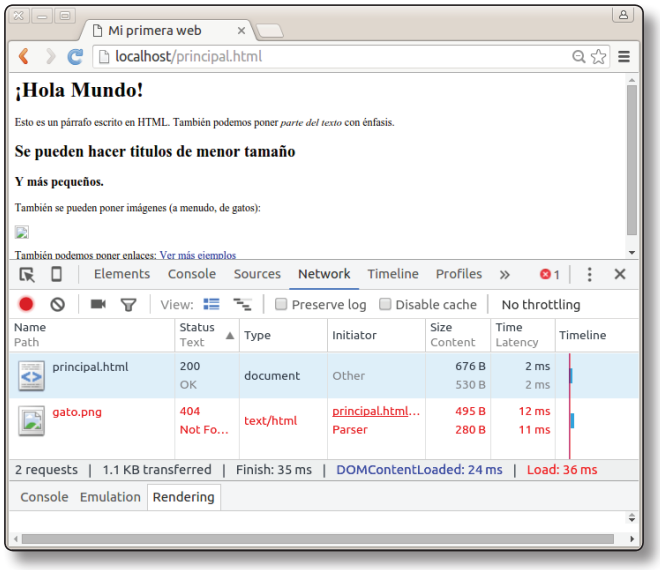
- **200**: éxito. En el cuerpo del mensaje se incluye el recurso solicitado.
- **404**: recurso no encontrado. Este error nos lo encontramos habitualmente cuando navegamos a una URL que no existe.
- **5xx**: indica un error en el servidor que impide dar respuesta a la petición. Es mucho menos común.

A continuación del código de respuesta se suelen incluir distintas líneas de cabecera con información sobre el servidor y los datos que nos devuelve:

- **Date**: fecha y hora actual en el servidor.
- **Server**: identifica el software del servidor "Program-name/x.xx".
- **Last-Modified**: última modificación del recurso (para gestionar cachés).
- **Content-Length**: número de bytes que va a contener la respuesta.
- **Content-Type**: tipo MIME del contenido.

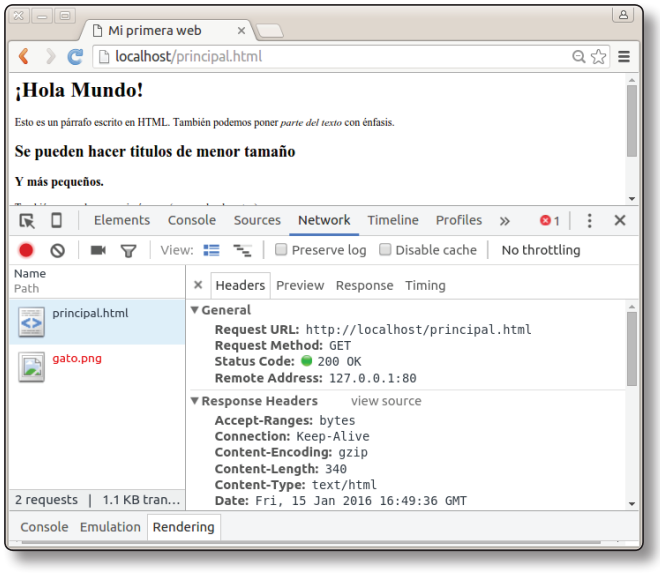
Comprueba en la Figura 1.11 la respuesta indicada por el servidor al realizar la petición **GET** que obtenía el archivo **principal.html** mediante telnet.

Aunque podemos "hablar" HTTP mediante telnet y resulta muy interesante hacerlo alguna vez, es más normal utilizar el navegador para inspeccionar el funcionamiento de este protocolo. Podemos hacerlo –de nuevo– mediante la pestaña de **Red** (Network) en las herramientas de Chrome. La Figura 1.12 muestra la salida de esta herramienta cuando realizamos la petición `http://localhost/principal.html`. Vemos que recibimos el código 200 para la página HTML, pero no así con la imagen que enlazamos desde ese archivo (*gato.png*), porque, en este caso, la hemos borrado deliberadamente del directorio desde donde sirve las páginas nuestro servidor web. Como no se encuentra esa imagen, el servidor devuelve un código 404.



1.12. Inspección de una petición GET con las herramientas de Chrome

Si pulsamos sobre la primera petición GET, podemos ver los detalles de la misma (Figura 1.13). En este caso, la herramienta nos muestra todos los detalles de la petición HTTP enviada al servidor, así como la respuesta recibida.



1.13. Detalle de una petición GET con las herramientas de Chrome

También puedes intentar conectar con cualquier servidor web y comprobar las peticiones y respuestas HTTP intercambiadas. Prueba a hacerlo también con telnet como curiosidad.

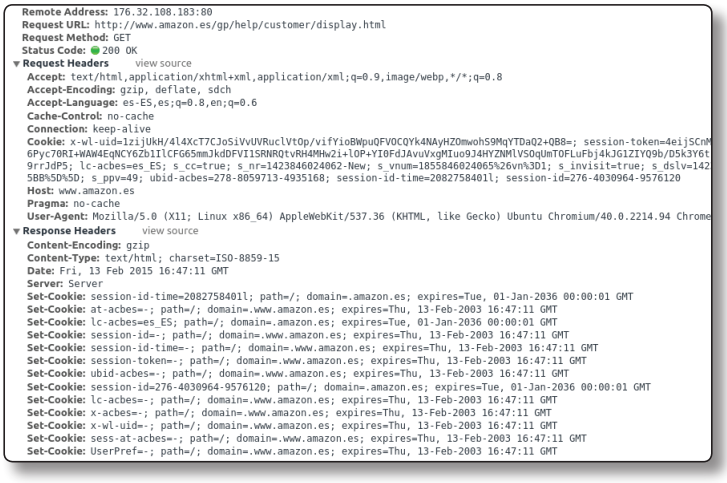
Ahora que ya conocemos todos los "secretos" de HTTP, veamos cómo podemos guardar información sobre el usuario de una llamada a otra. Para ello utilizaremos las **cookies**.

1.6.3 Cookies

Como hemos visto en los ejemplos anteriores, HTTP es un protocolo **sin estado**, lo cual quiere decir que no podemos guardar información sobre el usuario de una petición a otra, ya que son totalmente independientes. Esto es una limitación que tiene la ventaja de simplificar enormemente el protocolo, pero que debemos solventar para poder añadir funcionalidad a nuestra web. Para ello utilizaremos las **cookies**.

Una *cookie* es una cadena de texto que se pasa en una cabecera HTTP y que el navegador puede guardar en un pequeño fichero de texto. El contenido de la *cookie* siempre se genera en el servidor con información sobre el cliente. La *cookie* se reenvía luego al servidor HTTP con cada petición del cliente a ese servidor. De esta forma, podemos almacenar la información que queramos sobre el cliente y recibirla en cada petición de este.

Veamos un ejemplo. Comprueba mediante la herramienta de red de Chrome el mensaje `GET` de `http://www.amazon.es/gp/help/customer/display.html`. El resultado se muestra en la Figura 1.14.



1.14. Petición GET a www.amazon.es

En este ejemplo podemos intuir que al hacer la petición se envía una *cookie* y que al recibir la respuesta la *cookie* se actualiza. Aunque las *cookies* sean simplemente fragmentos de texto, las aplicaciones web suelen encriptarlas o codificarlas para que no puedan ser leídas por terceros, normalmente por seguridad. De hecho, es bastante común que lo único que se guarde en la *cookie* sea una especie de clave que identifique al usuario en la base de datos del servidor para así poder almacenar mucha información sobre este. Por eso, aunque una *cookie* pueda ocupar hasta 4 Kbytes, normalmente solo utilizan unos 100 bytes.

Si miramos el contenido de una *cookie*, veremos que está formado por un conjunto de pares `<nombre, valor>`. Pueden tener algún tipo de comentario que explique al usuario para qué sirve la *cookie*. Además, en la *cookie* se especifican las páginas y dominios a los que el navegador debe enviarla, así como su fecha de expiración. Esto nos permite mantener las sesiones de navegación de los usuarios cuando se han logueado en una aplicación web. Otros usos comunes de las *cookies* son guardar preferencias del usuario, reconocer antiguos usuarios y, sobre todo, recoger datos usados por aplicaciones de compra electrónica.

Gracias a las *cookies* podemos mantener información sobre el usuario de una sesión a otra, pero el protocolo HTTP todavía tiene otra gran carencia que solventar: la seguridad. Para ello disponemos de la versión encriptada de HTTP.

1.6.4 HTTPS

El HTTP seguro o HTTPS permite que la información sensible (datos de usuario, *passwords*, pagos, etc.) no pueda ser interceptada durante la transferencia de datos. Para ello se utiliza un tipo de criptografía denominado SSL/TLS.

Este tipo de criptografía requiere que nuestro servidor web cuente con un certificado que garantice su identidad. La veracidad de este certificado se comprueba a su vez mediante una serie de certificados que ya vienen instalados en el navegador y que se generan por las llamadas *autoridades de certificación*. Por tanto, si queremos utilizar HTTPS en nuestro servidor deberemos comprar un certificado a esas autoridades de certificación. Existen numerosas empresas que se encargan de ello y es fácil encontrarlas en la Web.

Una vez instalado nuestro certificado, los usuarios podrán conectarse de forma cifrada utilizando la URL `https://...`. En este caso se utilizará el puerto 443 en vez del puerto 80 estándar de HTTP.