



---

19

# Desarrollo de servicios Web

---

---

Los clientes actuales que existen en la Web, ya sean usuarios físicos o aplicaciones de terceros, son consumidores de servicios. Conseguir que la actual red, compuesta principalmente de páginas HTML y archivos, se convierta en una red de servicios es uno de los objetivos de la plataforma .NET. Un servicio Web es, como su propio nombre indica, un servicio que una determinada empresa pone a disposición de los usuarios. Según las necesidades, puede tratarse de un servicio de uso interno o accesible para el público de Internet en general.

Para crear un servicio Web podemos usar cualquiera de los lenguajes disponibles en la plataforma .NET, mientras que los consumidores pueden potencialmente estar en cualquier lugar y ejecutándose sobre cualquier hardware y sistema operativo. Realmente, cualquiera puede ofrecer o consumir un servicio Web si utiliza el protocolo adecuado, no es necesario disponer de Windows, la plataforma .NET o el lenguaje Visual C# .NET.

Aunque hay implementaciones que permiten publicar servicios Web sobre otros sistemas, gracias a IBM es posible, por ejemplo, usar Apache, en este capítulo nos centraremos en el uso de IIS sobre Windows. Los consumidores pueden utilizar una simple solicitud HTTP, desde un cliente Web, o bien usar un protocolo RPC sobre XML conocido como SOAP (*Simple Object Access Protocol*).

## ¿Qué es un servicio Web?

---

Actualmente raro es el usuario habitual de informática que no sabe lo que es una página Web. Si le hablamos de un servicio Web, sin embargo, seguro que

la mayoría de ellos no lo tendrán muy claro. Es lógico al tratarse, como ya se ha dicho, de un concepto relativamente nuevo. ¿Qué es en realidad un servicio web? ¿Cuál es su aspecto? ¿Cómo podemos crear uno o usarlo? Éstas son muchas de las cuestiones que surgen en un principio.

Imagine que usted adquiere un ordenador nuevo y que en él tan sólo encuentra inicialmente un sistema operativo, nada más. Para efectuar su trabajo, sin embargo, seguramente necesitará una libreta de direcciones, una calculadora y una aplicación de cálculo de estructuras, por poner algunos ejemplos. Obviamente, pensará, lo que tiene que hacer es buscar y adquirir el software que necesita, instalándolo en su ordenador para usarlo cuantas veces necesite.

Este es el modo en que *funcionamos* actualmente con nuestros ordenadores. Periódicamente, además, solemos actualizar todo ese software que hemos adquirido para disponer de una serie de posibilidades que, en ocasiones, ni siquiera necesitamos.

## Servicios y aplicaciones

---

En cierta forma, una aplicación es un servicio o un conjunto de servicios instalados en nuestro sistema. Microsoft Excel, por hablar de un programa bien conocido, es una aplicación formada por varios componentes o servicios distintos que facilitan la edición de los libros, diseño de gráficos, etc.

Suponga que los distintos componentes que forman Microsoft Excel no están instalados en su ordenador sino en un servidor Web, por ejemplo en la sede de Microsoft, y que para usarlos lo que hace es enviar solicitudes mediante su navegador habitual. Los componentes serían servicios Web, la sede de Microsoft sería la que expondría dichos servicios y, finalmente, nosotros actuariamos como consumidores.

La aparición de los servicios Web va a influir en la forma de hacer negocio de muchas empresas. Con la expansión de las redes empresariales y su interconexión a través de Internet, serán muchos los que opten por alquilar u ofrecer sus productos en forma de servicios Web, en lugar de hacerlo como hasta ahora, vendiendo licencias de uso de esos productos. A las empresas consumidoras, asimismo, puede interesarles más pagar por el uso de esos servicios que adquirir las aplicaciones equivalentes, sabiendo que siempre van a estar usando las últimas versiones sin necesidad de actualizar sus programas.

Para que este mecanismo de oferta de servicios tenga éxito, no obstante, antes es necesario que las distintas empresas implicadas lleguen a acuerdos de estándares para la descripción de servicios, la comunicación con ellos, la búsqueda, etc.

## Universalidad de un servicio

---

Actualmente, al publicar un documento en un servidor Web no nos preocu-pamos demasiado sobre la plataforma hardware, el sistema operativo ni las aplicaciones que usará el cliente para acceder a él. Esto es posible porque HTML es un estándar universal, lo mismo que el protocolo HTTP usado para solicitar

y transmitir esos documentos. De esta forma se consigue que un cierto servicio, en este caso la Web, sea realmente universal, ya que está accesible para todos sin limitaciones.

Conseguir que los servicios Web alcancen ese mismo nivel de universalidad es uno de los objetivos de empresas como Microsoft e IBM que, conjuntamente con muchas otras, están definiendo protocolos y borradores de estándares que permitan describir un servicio, publicarlo para que los consumidores puedan encontrarlo, establecer una comunicación con él por parte de los clientes, etc.

Aunque sea algo reiterativo, hay que incidir en el hecho de que es posible crear un servicio Web utilizando cualquier lenguaje, sobre cualquier sistema operativo y arquitectura o plataforma hardware. Para conseguir que ese servicio sea accesible para los consumidores hay que describirlo usando un lenguaje estándar que es WSDL (*Web Service Description Language*). El cliente podrá estar creado también en cualquier lenguaje y ejecutarse sobre cualquier sistema operativo y hardware, obteniendo la descripción WSDL de un servicio de directorio conocido como UDDI (*Universal Description, Discovery and Integration*). Con esta información se comunicaría con el servicio usando protocolos como HTTP o SOAP.

## Protocolos y lenguajes

Los servicios Web tienen como objetivo hacer más realidad que nunca la meta perseguida por la computación distribuida, para la cual se han usado hasta ahora tecnologías como CORBA, DCOM o Java RMI. El problema es poder efectuar llamadas a procedimientos remotos con un protocolo RPC (*Remote Procedure Call*) que no dependa de un cierto sistema, como es el caso de DCOM, o lenguaje, como ocurre con RMI.

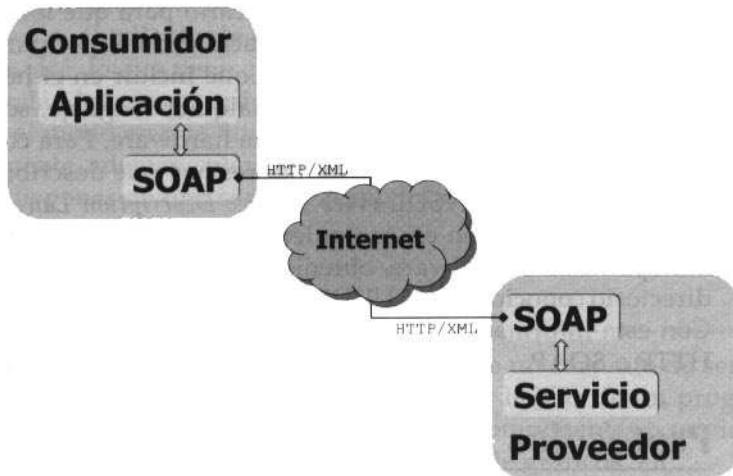
Hace un par de años tres empresas, Developmentor, UserLand y Microsoft, propusieron un protocolo que permitía efectuar llamadas RPC usando el lenguaje XML para describir la llamada y parámetros y el protocolo HTTP como transporte. El uso de HTTP para el transporte de las llamadas salva muchos obstáculos con los que se encontraban CORBA y DCOM, ya que la mayoría de las empresas utilizan cortafuegos que impiden la entrada en sus servidores por puertos no estándar. El puerto utilizado para HTTP, la base de la Web, es uno de los considerados estándar y, por tanto, ideal para ser utilizado por SOAP.

Dado que SOAP describe sus llamadas y respuestas utilizando XML, en la práctica es posible utilizarlo desde cualquier sistema operativo y lenguaje. Actualmente existen diversas implementaciones que permiten utilizar SOAP desde Windows, Unix y Linux.

Utilizando SOAP cualquier cliente puede consumir un servicio ofrecido por un proveedor, como puede verse en la figura 19.1. Es necesario, no obstante, que el cliente conozca de antemano la localización y naturaleza del servicio, sabiendo a qué métodos puede llamar, con qué parámetros y qué tipo de respuesta obtendrá.

La descripción de un servicio se efectúa, como se dijo antes, en WSDL (*Web Service Description Language*). Un módulo WSDL es un archivo XML en el que

se identifica el servicio y se facilita el esquema para poder utilizarlo, generalmente entregándose información sobre los distintos protocolos que es posible utilizar. Más adelante veremos cómo obtener el WSDL correspondiente a un servicio Web creado por nosotros, ya que ASP.NET es capaz de generar esa descripción automáticamente.



**Figura 19.1.** Un consumidor usa SOAP para, a través de Internet, acceder a un servicio ofrecido por un proveedor

Resueltos los problemas de conectividad entre consumidor y proveedor y descripción de los servicios, nos encontramos con un último obstáculo: cómo encontrará el consumidor a los proveedores que ofrecen los servicios que él necesita. Traslade la situación a otro entorno. Piense que necesita los servicios de un fontanero, ¿dónde buscaría lo que necesita? Efectivamente, en algún tipo de guía o páginas amarillas.

Tras llegar al acuerdo de creación del lenguaje WSDL, las tres empresas citadas antes anunciaron a principios de septiembre de 2000 que estaban trabajando en un servicio llamado UDDI. Se trata de un servicio de directorio a nivel mundial en el que los proveedores de servicios, de manera gratuita, podrán registrar sus productos. Los consumidores usarán UDDI para consultar ese registro a la búsqueda de lo que necesitan. En él encontrarán una descripción de los servicios, información sobre el proveedor y todo lo necesario para contactar con él.

Mientras que algunos protocolos y lenguajes están ya finalizados y propuestos como estándar, es el caso de SOAP, otros por el contrario se encuentran en plena fase de desarrollo y evolución, como WSDL y UDDI. La parte más importante de este capítulo está centrada en la creación y consumo de servicios usando Visual Studio .NET. En los puntos siguientes encontrará una introducción a los protocolos y lenguajes más importantes relacionados con los servicios Web. Si desea más información sobre ellos visite las sedes de Microsoft, IBM y el W3C.

## Introducción a XML

Como antes ocurriera con otras tecnologías surgidas de Internet, los desarrolladores están encontrando en XML múltiples aplicaciones. Si aún cree que XML es una variante de HTML o que tan sólo sirve para componer documentos para la Web, le interesa seguir leyendo.

No es que XML (*eXtensible Markup Language*, Lenguaje de marcas extensible) sea algo realmente nuevo, pero aún así sigue existiendo una cierta confusión en torno a él. Son muchos los que piensan que XML es una variante o derivado de HTML, y otros tantos creen que tan sólo es útil para insertar tablas de datos en las páginas HTML. No obstante, XML puede marcar, y de hecho está haciendo, el futuro en campos tan importantes como las bases de datos o el comercio electrónico, prueba de ello es la apuesta en este estándar por parte de centenares de empresas de todos los sectores.

Como programador, posiblemente tenga que vérselas con XML en un plazo relativamente corto, si es que no ha tenido que hacerlo ya. Mediante XML el intercambio de información, entre distintas aplicaciones, sistemas y plataformas, es mucho más fácil. El diseño de servicios Internet para acceso a bases de datos empresariales, una necesidad cada vez más presente, también resulta una tarea más sencilla usando XML, conjuntamente con hojas de estilo XSL (*eXtensible Stylesheet Language*).

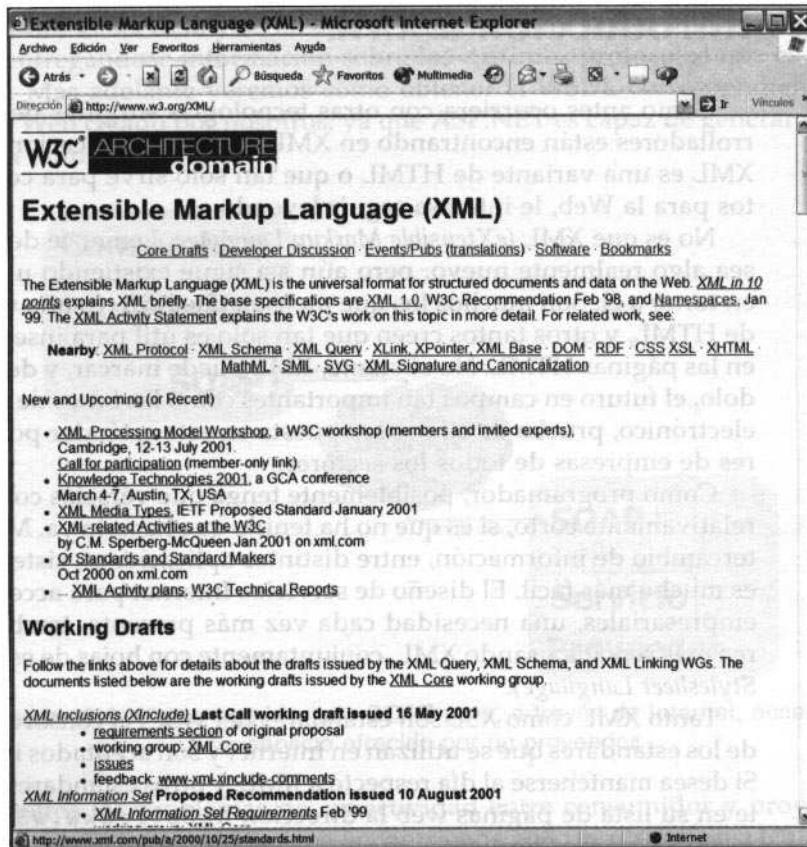
Tanto XML como XSL son estándares del W3C, el consorcio que rige muchos de los estándares que se utilizan en Internet y son aceptados internacionalmente. Si desea mantenerse al día respecto a XML, y otros estándares relacionados, anítele en su lista de páginas web la dirección <http://www.w3.org>.

En los puntos siguientes encontrará la información necesaria para saber qué es XML, qué es un *parser* XML, cómo crear documentos XML bien formados y cómo comprobar su validez. También conoceremos lo que son los documentos de definición y las hojas de estilo, con las que podremos generar HTML a partir de documentos XML.

### ¿Qué es XML?

Como su propio nombre indica, XML es un lenguaje de etiquetas extensible, es decir, en el que podemos crear nuestras propias marcas o etiquetas. Las etiquetas son la base de XML, al igual que ocurre con HTML. No es de extrañar, ya que ambos son lenguajes derivados de SGML (*Standard Generalized Markup Language*). El uso de etiquetas, no obstante, es uno de los pocos puntos comunes que hay entre XML y HTML.

La finalidad de un documento XML es definir la estructura de una información, contenida asimismo en el propio documento. Las etiquetas XML no indican, por el contrario, cuál es el formato visual de la información. HTML es un lenguaje de marcas, pero su finalidad consiste solamente en establecer la apariencia de unos datos contenidos en el documento, en ningún caso describir su estructura.



**Figura 19.2.** Sede web del *World Wide Web Consortium* dedicada a XML

Utilizando XML el usuario puede definir sus propias etiquetas según la información cuya estructura necesite describir. Esto significa que no hay un límite preestablecido, en la propia definición del lenguaje, que actúe como frontera en cuanto a los datos que es posible describir. HTML, por el contrario, cuenta con un conjunto bien conocido de etiquetas, con una finalidad preestablecida y no ampliable por el usuario final.

### Trabajo con etiquetas

Tanto un documento XML como uno HTML contienen datos y etiquetas. Las etiquetas HTML afectan a la apariencia de los datos, presumiblemente de una forma independiente del dispositivo final en que se muestren. Las etiquetas XML no indican cómo debe mostrarse un dato, sino qué es el dato.

Observe el listado siguiente, correspondiente a un documento HTML, y el que hay inmediatamente detrás, que tiene la misma información pero en XML. A no ser que se tengan conocimientos de HTML, será difícil saber cuál es el significado de etiquetas como `<p>` o `<li>`. En el documento XML, por el contrario, las etiquetas son autodescriptivas y, en consecuencia, es fácil apreciar que la información es una lista de productos clasificados en varios grupos.

```

<html>
<body>
LISTA DE PRODUCTOS
<p>Software
<ul>
<li>
Visual Basic .NET</li>
<li>
Visual C++ .NET</li>
<li>
VisualAge C# .NET</li>
<li>
Visual J# .NET</li>
</ul>

<p>Libros
<ul>
<li>
Programación en Windows 2000</li>
<li>
Cómo programar con Visual Basic para torpes</li>
<li>
Programación con Visual Basic .NET</li>
<li>
Programación con Visual Studio .NET</li>
</ul>
</body>
</html>

```

Como puede ver a continuación, el documento XML es autodescriptivo:

```

<?xml version="1.0"?>
<ListaProductos>
    <Software>
        <Producto>Visual Basic .NET</Producto>
        <Producto>Visual C++ .NET</Producto>
        <Producto>VisualAge C# .NET</Producto>
        <Producto>Visual J# .NET</Producto>
    </Software>
    <Libros>
        <Libro>Programación en Windows 2000</Libro>
        <Libro>Cómo programar con Visual Basic para torpes</Libro>
        <Libro>Programación con Visual Basic .NET</Libro>
        <Libro>Programación con Visual Studio .NET</Libro>
    </Libros>
</ListaProductos>

```

### Visualización de los documentos

Para crear un documento tan simple como el propuesto y ver su estructura, no es necesario usar ninguna herramienta específica. El Bloc de notas de Windows es una utilidad suficiente para este trabajo. No obstante, siempre puede utilizar el propio editor de Visual Studio .NET puesto que, como se aprecia en la figura 19.3, reconoce la sintaxis de HTML y XML, destacando sintácticamente los diferentes elementos.

## 19. Desarrollo de servicios Web



Figura 19.3. Edición de documentos HTML y XML en el editor de Visual Studio .NET

Cuando se trabaja con documentos HTML y XML, el editor de Visual Studio .NET muestra normalmente una serie de pestañas, situadas en la parte inferior, que nos permiten alternar entre el código propiamente dicho y una previsualización del documento resultante, en el caso de HTML, o la tabla de datos que contiene el documento XML. De hecho, puede utilizar la vista previa de HTML con el fin de crear el documento visualmente, sin necesidad de tener que introducir marcas y, de la misma forma, la pestaña **Datos** del editor XML hace

posible la introducción de nuevos datos en el documento de manera mucho más simple.

En la figura 19.4 vemos el documento HTML, correspondiente al primer listado, abierto en la página **Diseño** del editor de Visual Studio .NET. El aspecto es el de una lista de productos agrupados en categorías, como se esperaba. Su aspecto en otro cliente, por ejemplo Netscape Navigator, sería casi idéntico, ya que las etiquetas HTML especifican cuál debe ser la apariencia de la información.

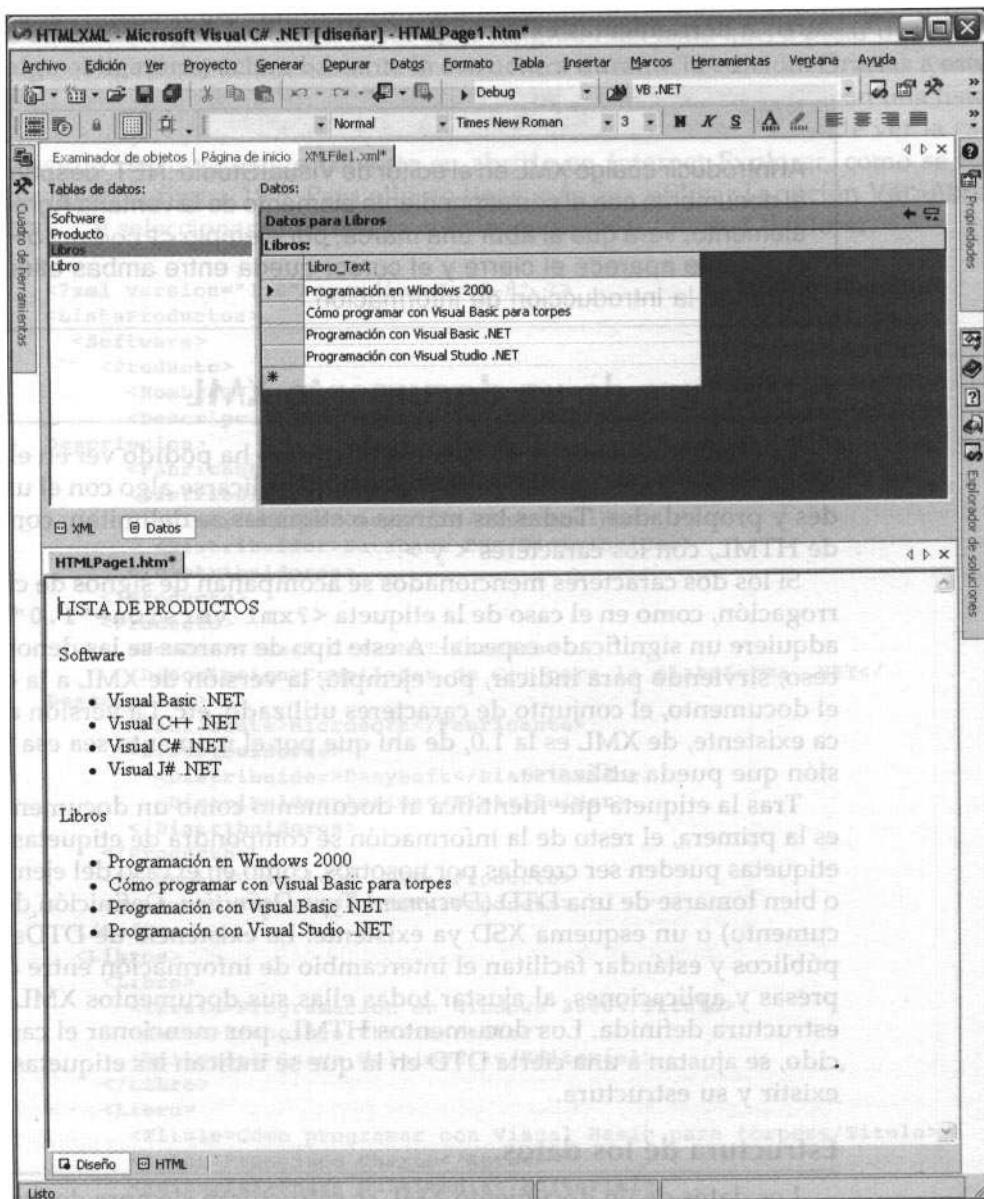


Figura 19.4. Edición visual de los documentos HTML y XML

En la parte inferior de la misma figura aparece el documento XML en la página **Datos** del editor XML de Visual Studio .NET. El resultado, como puede verse, es bien distinto. La plataforma .NET incorpora un analizador XML y otras herramientas que Visual Studio .NET utiliza para comprobar la estructura del documento y, en caso de que esté *bien formado*, recuperar su contenido y facilitar la edición. La lista de datos, además, es dinámica. Puede abrir y cerrar las diferentes ramas que existen.

Gracias a la capacidad de los editores de Visual Studio, no será necesario recurrir a herramientas externas para trabajar con documentos HTML o XML.

#### Nota

Al introducir código XML en el editor de Visual Studio .NET, después de crear el documento con el correspondiente elemento de la ventana Agregar nuevo elemento, verá que al abrir una marca, por ejemplo <Producto>, automáticamente aparece el cierre y el cursor queda entre ambas etiquetas, esperando la introducción de información.

## Estructura de un documento XML

La estructura de un documento XML, como ha podido ver en el ejemplo anterior, es bastante sencilla, si bien puede complicarse algo con el uso de entidades y propiedades. Todas las marcas o etiquetas se delimitan, como en el caso de HTML, con los caracteres < y >.

Si los dos caracteres mencionados se acompañan de signos de cierre de interrogación, como en el caso de la etiqueta <?xml version="1.0"?>, la marca adquiere un significado especial. A este tipo de marcas se las denomina de proceso, sirviendo para indicar, por ejemplo, la versión de XML a la que se ajusta el documento, el conjunto de caracteres utilizado, etc. La versión actual, y única existente, de XML es la 1.0, de ahí que por el momento sea esa la única versión que pueda utilizarse.

Tras la etiqueta que identifica al documento como un documento XML, que es la primera, el resto de la información se compondrá de etiquetas y datos. Las etiquetas pueden ser creadas por nosotros, como en el caso del ejemplo anterior, o bien tomarse de una DTD (*Document Type Definition*, Definición de tipo de documento) o un esquema XSD ya existente. La existencia de DTDs y esquemas públicos y estándar facilitan el intercambio de información entre distintas empresas y aplicaciones, al ajustar todas ellas sus documentos XML a una cierta estructura definida. Los documentos HTML, por mencionar el caso más conocido, se ajustan a una cierta DTD en la que se indican las etiquetas que pueden existir y su estructura.

### Estructura de los datos

Los datos de un documento XML se estructuran siempre de forma jerárquica. Esto significa que existe un elemento raíz, en el caso de nuestro documento

es <ListaProductos>, que contiene a los demás: <Software> y <Libros>. Éstos, a su vez, pueden contener otros elementos. Entre las etiquetas de apertura y cierre se introducirá la información que, en el ejemplo propuesto, es el nombre de los productos.

A pesar de que en el ejemplo anterior tan sólo existen dos niveles de datos, lo cierto es que pueden crearse tantos como se necesiten. Observe el documento XML siguiente, que es una ampliación del anterior. En él se detallan ahora ciertos datos de cada producto, como el nombre, descripción, fabricante, autor o editorial, según los casos. Fíjese en el sangrado del texto que, a pesar de no ser algo obligatorio, aclara bastante la estructura durante la edición. Gracias a este sangrado es fácil ver que la etiqueta <Distribuidores> cuenta con una lista de datos, por poner un ejemplo. No obstante, la forma más clara de ver la estructura del documento consiste en abrirlo en Internet Explorer, como se ha hecho en la figura 19.5. Para ello no tiene más que utilizar la opción Ver>Abrir con ... y seleccionar el elemento correspondiente a Internet Explorer.

```
<?xml version="1.0" encoding="utf-8" ?>
<ListaProductos>
  <Software>
    <Producto>
      <Nombre>Visual Basic .NET</Nombre>
      <Descripcion>Entorno de desarrollo RAD basado en BASIC</
      Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>Danysoft</Distribuidor>
        <Distribuidor>Database DM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual C++ .NET</Nombre>
      <Descripcion>Compilador de C++ para la plataforma .NET</
      Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>DanySoft</Distribuidor>
        <Distribuidor>Action</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>Visual C# .NET</Producto>
    <Producto>Visual J# .NET</Producto>
  </Software>
  <Libros>
    <Libro>
      <Titulo>Programación en Windows 2000</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Cómo programar con Visual Basic para torpes</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>Programación con Visual Basic .NET</Libro>
```

## 19. Desarrollo de servicios Web

```
<Libro>Programación con Visual Studio .NET</Libro>
</Libros>
</ListaProductos>
```

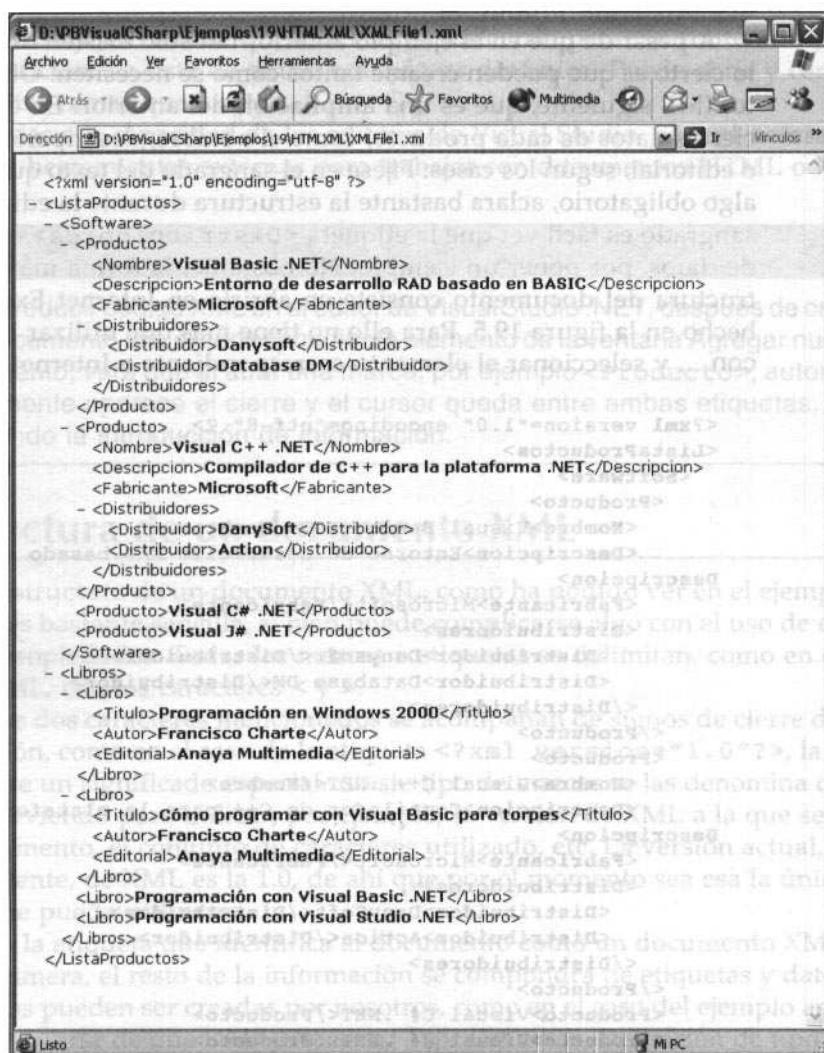


Figura 19.5. Aspecto del documento XML abierto en Internet Explorer

### Conjuntos de caracteres en XML

Los documentos XML pueden encontrarse codificados en distintos conjuntos de caracteres, siendo el asumido por defecto UTF-8, que es un Unicode de 8 bits. En caso de utilizarse cualquier otra codificación, será preciso indicarlo en la etiqueta de proceso que hay al inicio del documento.

Si observa los documentos XML correspondientes a los ejemplos anteriores, podrá ver que en el primero no existen caracteres acentuados. Éstos han sido

sustituidos por *entidades* como &#243; que representan al carácter que desea incluirse. Alternativamente, podría haberse utilizado &oacute;.

Lo anterior es necesario porque en la codificación por defecto no tienen cabida algunos caracteres europeos, como las vocales con acentos graves o agudos o la Ñ. Si deseásemos incluir estos caracteres directamente, tendríamos que especificar de manera explícita una codificación que los permitiese.

Aunque utilizar una codificación diferente es algo perfectamente válido, en ocasiones nos podemos encontrar con problemas si un cierto analizador no reconoce ese conjunto de caracteres. La solución pasa por usar siempre UTF-8, sustituyendo los mencionados caracteres por sus códigos.

### Documentos XML bien formados

A diferencia de lo que ocurre con los documentos HTML, en los que existe una cierta permisividad en cuanto a su estructura, los documentos XML deben estar *bien formados* para que su proceso sea posible. Para conseguir que un documento XML esté bien formado tan sólo hay que seguir unas reglas básicas, reglas que, además, podrían aplicarse también a HTML.

Un documento XML debe iniciarse siempre con la etiqueta de proceso que le identifica como XML y en la que, además, se indica la versión, el conjunto de caracteres utilizado, etc. A continuación tiene que existir una etiqueta que actúe como raíz, en la que estarán contenidas todas las demás. En cada documento XML tan sólo puede existir una raíz, en caso contrario no estará bien formado.

Todas las etiquetas XML deben tener su pareja, es decir, toda apertura debe contar con un cierre. Hay que tener en cuenta, además, que no pueden existir solapamientos entre las aperturas y cierres, sino que cada pareja de marcas debe contener totalmente a los datos que define. Por ello, los dos fragmentos de código XML no estarían bien formados.

```
<?xml version="1.0"?>
<ListaProductos>
  <Software>
    <Producto>Visual Basic .NET
  </Software>
</ListaProductos>

<?xml version="1.0"?>
<ListaProductos>
  <Software>
    <Producto>Visual Basic .NET
  </Software></Producto>
</ListaProductos>
```

En el primer caso la etiqueta `<Producto>` no cuenta con la correspondiente `</Producto>`, mientras que en el segundo el orden de las etiquetas no es el correcto, puesto que `<Software>` debería contener totalmente al grupo de datos `<Producto>`. Al intentar procesar este código, simplemente cambiando a la página **Datos** del editor, el analizador XML generará un error. En la figura 19.6 vemos cómo se indica que el documento no está bien formado, comunicándose

el punto donde está el error y su naturaleza. Algo parecido ocurrirá si intenta abrir el documento en Internet Explorer.

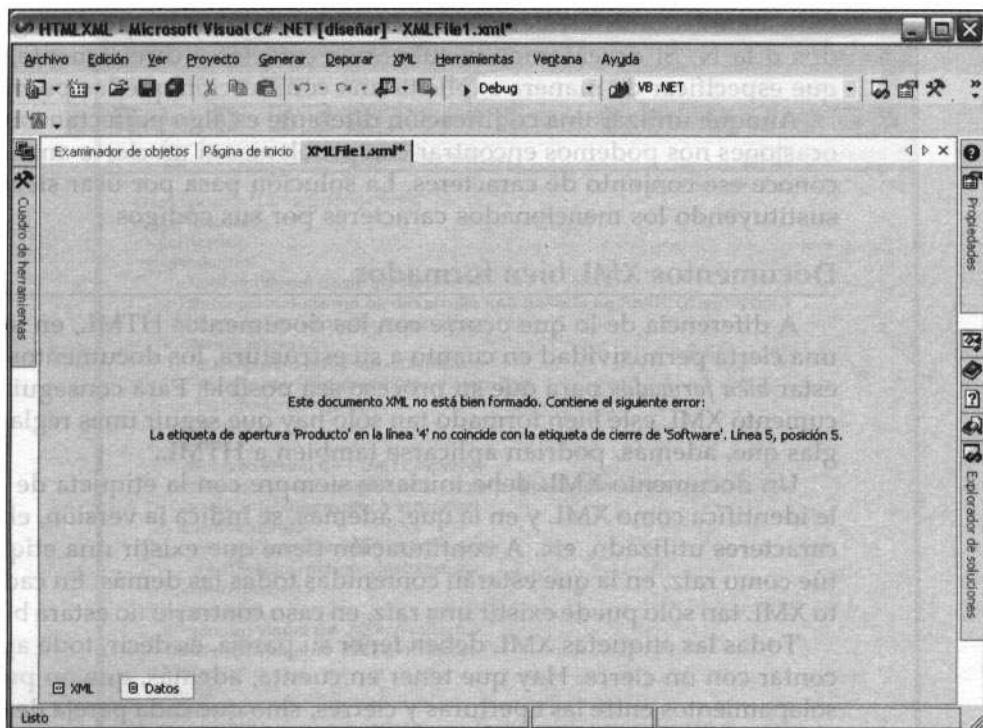


Figura 19.6. El analizador nos indica dónde ha encontrado el error

Existen otras normas adicionales para que un documento esté bien formado, como por ejemplo la obligatoriedad de que los valores de las propiedades estén entrecomillados. Conocerá estas normas a medida que vaya introduciéndose en el uso de XML.

Como puede ver en el ejemplo de HTML mostrado anteriormente, para que este sea válido no es preciso cumplir todas las normas. La marca `<p>`, por ejemplo, no cuenta con sus correspondientes cierres: `</p>`, y no por ello el cliente web emite error alguno.

### Documentos XML válidos

Que un documento XML esté bien formado significa que sigue las normas de codificación XML, pero ello no implica que sea un documento válido. El ejemplo que se muestra en la figura 19.5, por ejemplo, es un documento XML bien formado, pero a juzgar por su estructura posiblemente no sea válido. ¿Por qué? Observe que algunos productos cuentan con datos de detalle, mientras que otros no. Lo lógico sería que todos siguiesen la misma estructura.

Si no se tiene una definición previa de la estructura del documento XML, lo que se conoce como una DTD o un esquema, es posible incurrir en fallos que

harían que el documento no fuese válido, a pesar de que sí estuviese bien formado. Puede realizar la siguiente prueba: partiendo del ejemplo mostrado en la figura 19.5 añada a la lista de distribuidores del primer producto una pareja `<Editorial>xxx</Editorial>`. Al visualizar el documento podrá ver que no se genera error alguno, puesto que el documento está bien formado. No obstante, no es lógico que en el interior de una lista de distribuidores de un producto, que además no es un libro, haya una entrada correspondiente a una editorial.

La única forma de evitar este tipo de fallos consiste en utilizar una DTD o un esquema, que no es más que un documento XML en el que se indica cuál debe ser la estructura de los documentos que se ajusten a esa definición. Puede disponerse de una DTD o un esquema de dos formas: creándolo uno mismo o adquiriéndolo de terceros. Actualmente, y como se indicaba al inicio del capítulo, existen decenas de empresas que, de forma conjunta, están creando esquemas para los tipos de negocio más habituales. Esto contribuirá a que existan unos esquemas estándar para determinadas tareas como, por ejemplo, la gestión de pedidos entre empresas.

Como conclusión de lo explicado en éste y el punto anterior, resumir que un documento XML debe estar bien formado, lo cual no implica que sea un documento válido. Para validar un documento XML es necesario disponer de una DTD o un esquema, que podemos crear nosotros mismos si no existe una que se ajuste a nuestras necesidades. Un documento XML válido siempre será un documento bien formado.

## Analizadores de XML

Un documento XML es, como ya ha podido ver, un archivo de texto en el que se utiliza un cierto conjunto de caracteres. Para extraer la información contenida en dicho documento, adaptándola a la estructura que puede deducirse de él mismo, se utiliza un analizador o *parser*. Dada la relativa simplicidad de los documentos XML, escribir un analizador no constituye una tarea compleja en exceso. No obstante, no tiene por qué escribir su propio analizador puesto que existen múltiples disponibles para varias plataformas y escritos en distintos lenguajes.

Los analizadores XML se agrupan en dos categorías: validantes y no validantes. Los analizadores no validantes, que son los más sencillos, solamente comprueban que el documento XML esté bien formado. Los validantes, además, comprueban si el documento se ajusta a la estructura definida en la DTD o esquema correspondiente, comunicando si el documento es o no válido.

Una vez que el documento se ha analizado y comprobado, existen distintos medios para acceder a la información, ya sea obteniéndola o modificándola. Los dos métodos más conocidos son SAX (*Simple API for XML*) y DOM (*Document Object Model*). Mediante SAX es posible manipular los datos del documento XML de una forma bastante simple, como su propio nombre indica, utilizando manejadores y llamando a métodos. DOM, por el contrario, es un modelo jerárquico de objetos generado a partir de la estructura del documento XML, siendo mucho más flexible y potente.

Trabajando con Visual Studio .NET en un sistema Windows actual, como ya ha podido ver, basta con usar el editor de XML para escribir el código y la página **Datos** para acceder a los datos previa comprobación del documento. Usando el menú emergente asociado al documento XML, en la ventana Explorador de soluciones, puede ejecutar la opción **Generar y examinar** para validar el documento y verlo jerárquicamente, como en Internet Explorer, pero sin salir de Visual Studio .NET (véase figura 19.7).

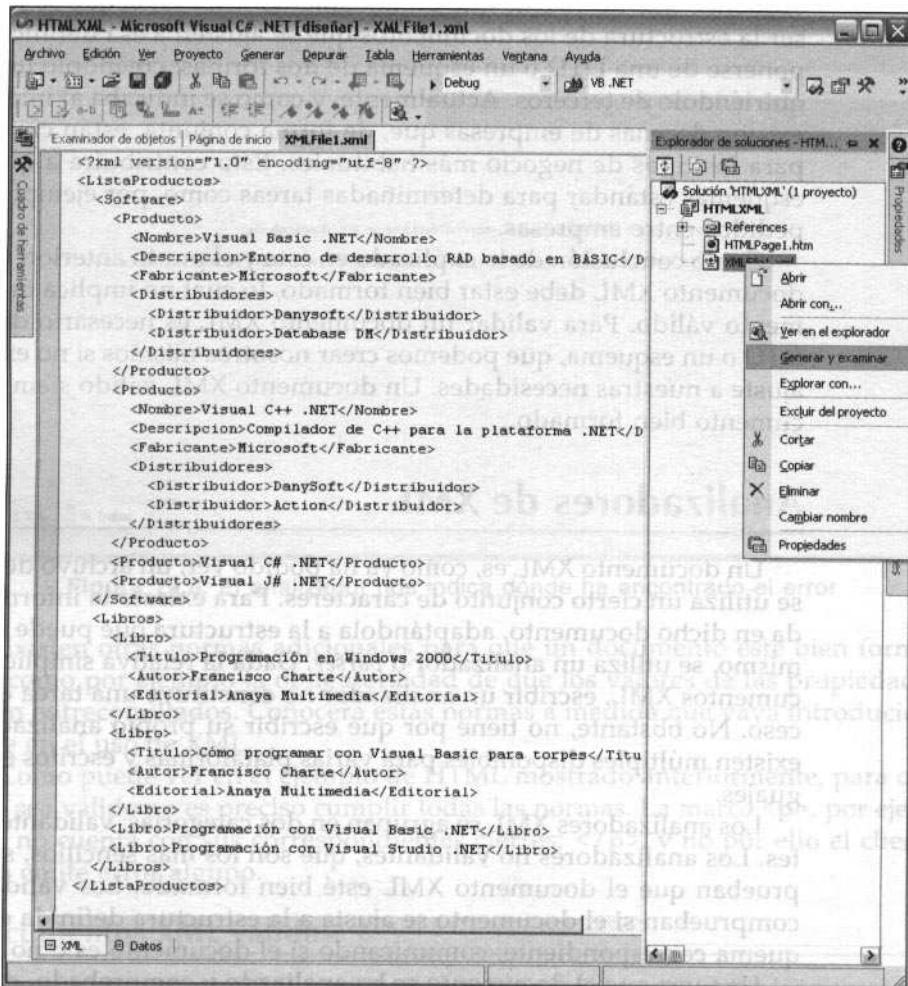


Figura 19.7. Desde el entorno de Visual Studio .NET podemos validar los documentos XML

## Validación de documentos

Un documento XML, según se ha dicho antes, puede estar bien formado y, sin embargo, ello no implica que sea válido. En los siguientes puntos explicamos

cómo usar una DTD para validar documentos, introduciendo también nuevos conceptos como los atributos.

### Nota

Desde el surgimiento de XML el estándar para definir su estructura ha sido el uso de las DTD, lo cual hace que existan muchas definiciones de este tipo. Actualmente, sin embargo, el estándar XSD hace mucho más fácil la creación de esquemas XML. Conocerá básicamente XSD más adelante en este mismo capítulo.

En los puntos previos se ha indicado qué era un documento XML bien formado y qué un documento válido, haciendo hincapié en las diferencias entre ambos términos. También se ha visto que existen analizadores XML validantes y no validantes. Los primeros comprueban que el documento esté bien formado y, además, que sea válido. Los segundos, por el contrario, tan sólo verifican si el documento está bien formado.

Aunque es posible verificar manualmente la corrección de un documento, no se trata desde luego de un método práctico, sobre todo cuando se trata de documentos extensos y, además, elaborados por distintas aplicaciones o usuarios. Lógicamente, es mucho más cómodo y eficiente utilizar un analizador validante. Éste, no obstante, necesita conocer las reglas que rigen el documento, de lo contrario no sabrá si es o no válido.

El método utilizado para facilitar las reglas al analizador consiste en escribir una definición de tipo de documento, conocida abreviadamente como DTD. No es necesario escribir una DTD nueva para cada documento XML. De hecho, uno de los mayores objetivos actuales de muchas empresas es crear DTDs estándar que puedan ser usadas para el intercambio de documentos en formato XML. Una DTD generalmente se almacena en un archivo independiente, aunque también puede facilitarse integrada en el propio documento XML.

### ¿Qué es una DTD?

Físicamente, una DTD es un documento con estructura similar a un documento XML. El contenido, sin embargo, no son datos propiamente dichos, como en el caso de XML, sino indicaciones acerca de la estructura de un determinado tipo de archivo.

A pesar de que una DTD puede parecer, sobre todo al principio, relativamente compleja, en su formato más básico no es más que una enumeración de los elementos que podrán existir en un documento XML, indicando el tipo de cada uno de ellos. Cada uno de los elementos se describe en el interior de una etiqueta `<!ELEMENT>`, en la cual se incluirá el nombre del elemento y, entre paréntesis, su tipo o la lista de elementos que podrá contener.

Como es obvio, antes de escribir una DTD es preciso un análisis y, posiblemente, una plasmación gráfica de la estructura del documento que quiere definirse. Partiendo de este trabajo, escribir la DTD no será una tarea muy compleja.

En lugar de describir teóricamente cuál sería la estructura de una DTD, definición que puede encontrar en cualquier momento en la especificación del estándar, vamos a ver en la práctica cómo crear una DTD. Ésta nos servirá para validar el documento XML usado anteriormente como ejemplo, documento que contenía una lista de productos.

### Análisis de la estructura del documento

El ejemplo estará basado en el documento XML utilizado anteriormente, conteniendo una lista de productos de un hipotético comercio. Como puede verse en la figura 19.5, existe un nodo raíz, <ListaProductos>, que contiene a otras dos: <Software> y <Libros>. Ambas sirven como contenedores de varias etiquetas, <Producto> en el primer caso y <Libro> en el segundo.

Unas etiquetas <Producto> contienen directamente un dato en su interior: el nombre del producto. Otras sirven como contenedor de las etiquetas <Nombre>, <Descripción>, <Fabricante> y <Distribuidores>, en el caso de los productos de software, o <Titulo>, <Autor> y <Editorial> si el producto es un libro. Validar un documento de este tipo resultaría relativamente complejo, puesto que las etiquetas <Producto> y <Libro> no siguen un patrón fijo de comportamiento.

Tras remodelar la jerarquía de elementos y pensar la relación existente entre ellos, una estructura mucho más lógica podría ser la mostrada en la figura 19.8. Sigue existiendo un elemento raíz contenedor de otros dos, pero los problemas que existían con los elementos <Producto> y <Libro> desaparecen por completo. Esta representación gráfica y parcial de la estructura nos servirá para escribir la DTD que, a su vez, se utilizará para validar el documento.



**Figura 19.8.** Estructura de nuestros documentos XML

No hace falta decir que el documento XML del ejemplo que estamos utilizando, según se ve en la figura 19.5, no cumple con la estructura de la figura 19.8. Esto significa que, tras escribir la DTD y aplicarla, el analizador nos comunicará que no se trata de un documento XML válido, a pesar de que esté bien formado.

### Elaboración de la DTD

Teniendo a la vista la figura 19.8, veamos cómo podemos elaborar la definición de tipo de documento para las listas de productos que gestionarán nuestras aplicaciones. Como se ha indicado antes, cada uno de los elementos que aparecen en la mencionada figura se traducirá en una etiqueta `<!ELEMENT>`, en la cual se indicará el contenido que puede tener el elemento.

El primer elemento, actuando como raíz de todos los demás, es `ListaProductos`. Éste contendrá en su interior dos subelementos: `Software` y `Libros`. Observe el término *contendrá*, casi imperativo. No se ha dicho *podrá contener*, en cuyo caso los subelementos anteriores podrían o no existir en la lista de productos. Fíjese también en la imposibilidad de que se repitan los subelementos. `ListaProductos` tan sólo contendrá en su interior dos elementos, los ya indicados, ni uno más ni uno menos.

La etiqueta DTD con la que se describiría el elemento `ListaProductos` sería `<!ELEMENT ListaProductos (Software, Libros)>`. Tras la marca de proceso `!ELEMENT` se indica el nombre de la etiqueta raíz y, entre paréntesis, el nombre de las etiquetas que contendrá.

A diferencia de `ListaProductos`, los elementos `Software` y `Libros` pueden contener un número indeterminado de subelementos aunque, eso sí, todos ellos con la misma estructura. Dentro de un elemento `Software` existirán uno o más `Producto`, mientras que en el interior de un `Libros` el elemento repetitivo sería `Libro`. La correspondiente entrada en la DTD, por ejemplo en el caso del elemento `Software`, sería `<!ELEMENT Software (Producto+)>`. Observe el símbolo `+` dispuesto detrás de `Producto`, él es el que especifica que el elemento se repetirá tantas veces como sea necesario.

La definición DTD de un `Producto` o un `Libro` es bastante sencilla. Basándonos en la definición anterior de `ListaProductos`, lo único que habría que hacer sería enumerar los elementos que habría en su interior, indicando la posible repetición.

En el último nivel se encuentran los elementos que contienen datos, como `Descripcion` o `Autor`. Éstas etiquetas no servirán como contenedoras de otras, sino que contendrán un texto. Este tipo de dato se identifica en una DTD como `#PCDATA`. De este modo, la definición del elemento `Titulo`, por ejemplo, quedaría como `<!ELEMENT Titulo (#PCDATA)>`.

Con todo, la DTD completa para nuestro documento sería la mostrada a continuación. El sangrado no es algo obligatorio, pero ayuda a identificar los distintos niveles de contención de unos elementos en otros.

```
<!ELEMENT ListaProductos (Software, Libros)>
<!ELEMENT Software (Producto+)>
```

```
<!ELEMENT Producto (Nombre, Descripcion, Fabricante,
Distribuidores)>
<!ELEMENT Nombre (#PCDATA)>
<!ELEMENT Descripcion (#PCDATA)>
<!ELEMENT Fabricante (#PCDATA)>
<!ELEMENT Distribuidores (Distribuidor+)>
<!ELEMENT Distribuidor (#PCDATA)>
<!ELEMENT Libros (Libro+)>
<!ELEMENT Libro (Titulo, Autor, Editorial)>
<!ELEMENT Titulo (#PCDATA)>
<!ELEMENT Autor (#PCDATA)>
<!ELEMENT Editorial (#PCDATA)>
```

Este texto lo almacenaríamos en un documento con extensión DTD que, generalmente, estará alojado en el mismo servidor desde el que se obtenga el documento XML.

### Cómo aplicar una DTD a un documento XML

---

Una definición de tipo de documento no es útil *per sé*, aunque puede utilizarse como referencia para preparar los documentos XML. La finalidad de una DTD, como ya se ha indicado previamente, no es otra que servir a un analizador validante como referencia, facilitando la revisión y validación de cualquier documento XML que se adapte a dicha DTD.

Los documentos XML basados en una cierta DTD tienen dos opciones a la hora de indicarlo al analizador: incluir la DTD completa en el propio documento XML o hacer referencia a una DTD externa. El primer caso es útil cuando el propio documento XML es una singularidad, que no va a repetirse y, consecuentemente, consta de una DTD no válida en otros casos. El segundo, mucho más común, es aplicable siempre que la DTD vaya a utilizarse en múltiples documentos. Manteniendo en éstos una referencia externa, se evitará la repetición de la DTD en cada documento.

La instrucción de proceso para especificar el tipo de documento, en el interior del documento XML, es `<!DOCTYPE>`, que irá seguida de un identificador. Tras éste pueden existir unos corchetes, en cuyo interior estaría contenida la DTD, o bien una referencia a la DTD externa. En el ejemplo siguiente puede ver un fragmento del documento XML, conteniendo la DTD completa.

```
<?xml version="1.0"?>
<!DOCTYPE ListaProductos [
<!ELEMENT ListaProductos (Software, Libros)>
<!ELEMENT Software (Producto+)>
<!ELEMENT Producto (Nombre, Descripcion, Fabricante,
Distribuidores)>
<!ELEMENT Nombre (#PCDATA)>
<!ELEMENT Descripcion (#PCDATA)>
<!ELEMENT Fabricante (#PCDATA)>
<!ELEMENT Distribuidores (Distribuidor+)>
<!ELEMENT Distribuidor (#PCDATA)>
<!ELEMENT Libros (Libro+)>
<!ELEMENT Libro (Titulo, Autor, Editorial)>
<!ELEMENT Titulo (#PCDATA)>
```

```
<!ELEMENT Autor (#PCDATA)>
<!ELEMENT Editorial (#PCDATA)>
]>
<ListaProductos>
<Software>
<Producto>
<Nombre>Visual Basic .NET</Nombre>
<Descripcion>Entorno de desarrollo RAD basado en BASIC</
Descripcion>
...

```

A continuación se muestra el mismo fragmento, pero en este caso con una referencia a la DTD externa. Ésta se ha almacenado en un archivo denominado **ListaProductos.dtd**, que contiene la definición mostrada anteriormente.

```
<?xml version="1.0"?>
<!DOCTYPE ListaProductos SYSTEM "ListaProductos.dtd">
<ListaProductos>
<Software>
<Producto>
<Nombre>Visual Basic .NET</Nombre>
<Descripcion>Entorno de desarrollo RAD basado en BASIC</
Descripcion>
...

```

En cualquiera de los dos casos, el documento XML ya cuenta con una DTD en la que se especifica la estructura que debería tener ese documento. Un analizador no validante no usaría dicha información para nada, por lo que el documento mostrado anteriormente en la figura 19.5 seguiría siendo correcto, al estar bien formado. Un analizador validante, por el contrario, generaría un error, por ejemplo al encontrar que una etiqueta **<Producto>** contiene directamente el nombre del producto, en lugar de contar con las etiquetas **<Nombre>**, **<Fabricante>**, etc.

### Validación del documento

Tras añadir la DTD, ya sea en el interior del documento XML o haciendo referencia a ella, habrá que proceder a la validación. Para ello puede utilizarse un analizador validante, de los cuales hay bastantes disponibles en Internet. También puede, simplemente, abrir el documento XML con algún editor o visor de este tipo de documentos.

A pesar de que no disponga de ningún analizador ni editor, si tiene instalado en su sistema la última versión de Internet Explorer ya cuenta con un analizador validante. Éste puede ser usado desde cualquier herramienta de desarrollo, así como desde un *script*. Aprovechando la existencia de WSH (*Windows Scripting Host*) en Windows, que permite la ejecución directa de guiones en VBScript, podríamos servirnos de un *script* como el mostrado a continuación.

```
' Creamos el objeto XMLDOM
Set DocXML = CreateObject("Microsoft.XMLDOM")
' Activamos la validación durante el análisis
```

```
DocXML.validateOnParse = True
    ' Si el documento no es válido
If Not DocXML.Load("ListaProductos.xml") Then
    ' indicarlo con un mensaje
    MsgBox "Hay errores en el documento " & DocXML.url
Else ' en caso contrario indicar que sí es correcto
    MsgBox "El documento " & DocXML.url & " es correcto"
End If
```

#### Nota

Puede añadir un archivo VBScript al proyecto que tenga abierto usando la opción adecuada de la ventana Agregar nuevo elemento. Usando la opción Generar y examinar del Explorador de soluciones, anteriormente indicada, y pulsando sobre el botón Abrir de la ventana que aparece ejecutaremos el guión y veremos el resultado.

El analizador XML incorporado en Internet Explorer es un componente ActiveX, que puede ser usado prácticamente desde cualquier aplicación Windows. En el citado guión, lo que hacemos es crear una copia de ese componente, dando valor a una propiedad con la que se indica que deseamos validar el documento, además de comprobar si está bien formado. A continuación, utilizamos el método `Load()` para abrir el documento, un documento que sabemos que existe en el disco y que no habrá problemas en recuperar. Si el valor devuelto por dicho método es `False`, algo que comprobamos con el operador `Not`, indicará que el documento no es válido. La validez o no del documento se comunica mediante un mensaje.

Después de guardar el guión en un archivo, por ejemplo con el nombre `validacion.vbs`, será suficiente con hacer doble clic sobre él desde el Explorador de Windows para ejecutarlo. El resultado será un mensaje como el de la figura 19.9. El documento XML no es válido, puesto que su estructura no cumple con la definición de la DTD.

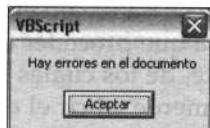


Figura 19.9. Nuestro script nos indica que hay errores en el documento XML

Tendríamos que realizar algunos cambios, dejando el documento tal y como se muestra en el listado siguiente. Al efectuar la validación, ejecutando de nuevo el guión VBScript, el mensaje obtenido ahora sería el que se muestra en la figura 19.10. El documento sí es correcto y válido, puesto que se ajusta a las directrices de la DTD.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE ListaProductos SYSTEM "ListaProductos.dtd">
```

```
<ListaProductos>
  <Software>
    <Producto>
      <Nombre>Visual Basic .NET</Nombre>
      <Descripcion>Entorno de desarrollo RAD basado en BASIC</
      Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>Danysoft</Distribuidor>
        <Distribuidor>Database DM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual C++ .NET</Nombre>
      <Descripcion>Compilador de C++ para la plataforma .NET</
      Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>DanySoft</Distribuidor>
        <Distribuidor>Action</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual C# .NET</Nombre>
      <Descripcion>Nuevo lenguaje orientado a componentes</
      Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>Danysoft</Distribuidor>
        <Distribuidor>Database DM</Distribuidor>
      </Distribuidores>
    </Producto>
    <Producto>
      <Nombre>Visual J# .NET</Nombre>
      <Descripcion>Lenguaje Java para la plataforma .NET</
      Descripcion>
      <Fabricante>Microsoft</Fabricante>
      <Distribuidores>
        <Distribuidor>En preparación</Distribuidor>
        <Distribuidor>En preparación</Distribuidor>
      </Distribuidores>
    </Producto>
  </Software>
  <Libros>
    <Libro>
      <Titulo>Programación en Windows 2000</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Cómo programar con Visual Basic para torpes</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
    </Libro>
    <Libro>
      <Titulo>Programación con Visual Basic .NET</Titulo>
      <Autor>Francisco Charte</Autor>
      <Editorial>Anaya Multimedia</Editorial>
```

```
</Libro>
<Libro>
    <Titulo>Programación con Visual Studio .NET</Titulo>
    <Autor>Jorge Serrano y Francisco Charte</Autor>
    <Editorial>Anaya Multimedia</Editorial>
</Libro>
</Libros>
</ListaProductos>
```

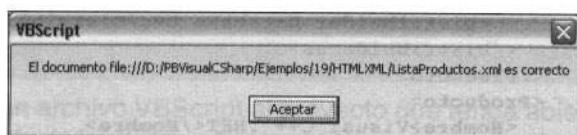


Figura 19.10. Tras los cambios nuestro documento XML sí es válido

## Atributos o propiedades

Además de otros elementos y datos, cada una de las etiquetas de un documento XML puede contar también con atributos. Éstos pueden ser considerados como propiedades del elemento, que definen alguna característica. En el ejemplo que estamos usando nosotros, una propiedad de un producto software podría ser el tipo de máquina al que va destinado: PC, Mac, etc. Para los libros, podría existir un atributo que nos indicase su categoría: ofimática, programación, etc.

En HTML existen marcas con atributos. La marca `<a>`, por ejemplo, cuenta con un atributo llamado `href` cuyo valor determina el URL del sitio web al que apunta un enlace. La marca `<font>`, por mencionar otro caso conocido, cuenta con atributos como `face` y `size` para establecer el tipo de letra y su tamaño. En todo caso, el formato de las marcas con atributos es siempre el mismo:

```
<marca atributo="valor" atributo="valor">
```

De manera análoga, en XML podrían introducirse etiquetas con atributos, como en el fragmento siguiente. El único inconveniente es que, en principio, a los atributos `Tipo` y `Categoría` podrían asignársele cualquier valor. Como sería de esperar, es posible especificar qué valores son los válidos mediante las entradas apropiadas en la DTD.

```
<ListaProductos>
<Software>
    <Producto Tipo="Mac">
        <Nombre>VisualAge C++ 4</Nombre>
        <Descripcion>Entorno de desarrollo Java</Descripcion>
        <Fabricante>IBM</Fabricante>
    <Distribuidores>
        <Distribuidor>IBM</Distribuidor>
    </Distribuidores>
    </Producto>
    <Producto Tipo="PC">
        <Nombre>Visual Caf&#233; Java 3</Nombre>
```

```
<Descripcion>Entorno de desarrollo Java</Descripcion>
<Fabricante>Symantec</Fabricante>
<Distribuidores>
    <Distribuidor>Symantec</Distribuidor>
</Distribuidores>
</Producto>
</Software>
<Libros>
    <Libro Categoria="Programacion">
        <Titulo>Programaci&on en Windows 2000</Titulo>
        <Autor>Francisco Charte Ojeda</Autor>
        <Editorial>Anaya Multimedia</Editorial>
    </Libro>
```

Si para definir el contenido de un elemento se utiliza la etiqueta `<!ELEMENT>`, para hacer lo propio con los atributos existe la etiqueta `<!ATTLIST>`. Ésta especificará el nombre del atributo, así como los posibles valores de éste en caso de que sea un conjunto limitado. En principio, la existencia de un atributo en la DTD no implica que en el documento XML dicho atributo se incluya siempre. Si deseamos que esto sea así, que el atributo sea obligatorio, tan sólo hay que finalizar la declaración con la palabra `#REQUIRED`.

En el listado siguiente puede ver una segunda versión de la DTD original, en este caso incluyendo la definición de los atributos posibles. Puede aplicar esta nueva DTD a una versión modificada del documento XML anterior, en el que se incluyesen los atributos necesarios. El mismo guión de validación le serviría también para comprobar el documento.

```
<!ELEMENT ListaProductos (Software, Libros)>
<!ELEMENT Software (Producto+)>
<!ELEMENT Producto (Nombre, Descripcion, Fabricante,
Distribuidores)>
    <!ATTLIST Producto Tipo (PC | Mac | Consola) #REQUIRED>
    <!ELEMENT Nombre (#PCDATA)>
    <!ELEMENT Descripcion (#PCDATA)>
    <!ELEMENT Fabricante (#PCDATA)>
    <!ELEMENT Distribuidores (Distribuidor+)>
        <!ELEMENT Distribuidor (#PCDATA)>
<!ELEMENT Libros (Libro+)>
    <!ELEMENT Libro (Titulo, Autor, Editorial)>
    <!ATTLIST Libro Categoria (Sistema | Ofimatica | Programacion)
#REQUIRED>
    <!ELEMENT Titulo (#PCDATA)>
    <!ELEMENT Autor (#PCDATA)>
    <!ELEMENT Editorial (#PCDATA)>
```

## Introducción a XSL

---

Ya conocemos la estructura de un documento XML, lo que es una DTD y cómo validar un documento. La visualización de los documentos, hasta ahora, siempre se ha efectuado directamente en el editor de Visual Studio .NET o en Internet Explorer en forma de árbol jerárquico. Internet Explorer incorpora no

sólo un analizador XML, sino también una hoja de estilo XSL por defecto, que muestra los documentos de este tipo en forma de árbol jerárquico.

La visualización directa del código XML no es lo habitual, ya que el formato puede no ser siempre el más apropiado. Mediante XSL es posible convertir un documento XML en un documento HTML, dando formato a los datos según interese en cada caso.

Los documentos XML pueden ser transformados en HTML o cualquier otro formato gracias a la existencia de XSL. En los siguientes puntos vamos a conocer las bases de XSL.

## Transformación de documentos

Los documentos XML son autodescriptivos pero, a pesar de ello, su formato no es el más apropiado para generar informes, ya sean éstos impresos o en forma de documentos web. Mediante XSL, tal y como veremos a continuación, es posible generar documentos HTML a partir de documentos XML, aplicando el formato que se deseé.

En un principio puede parecer que XSL es un lenguaje para aplicar hojas de estilo a los documentos XML, de forma similar a CSS (*Cascading Style Sheets*). La realidad, sin embargo, es que XSL es mucho más potente, facilitando la selección, filtrado y ordenación de los datos, aparte de aplicarles un determinado formato. Por ello, y a pesar de su nombre, XSL puede considerarse más un lenguaje de transformación de documentos, desde formato XML a otros formatos. De hecho, XSL ha sido la base de XSLT que, aunque muy básicamente, conocerá más adelante en este mismo capítulo.

Al igual que XML, XSL es un lenguaje que se utiliza para crear documentos. Éstos siguen las reglas de sintaxis de XML. Dicho de otra forma, un documento XSL es un documento XML, aunque en él se utilizan elementos predefinidos que permiten manipular la información. Todas las etiquetas en un documento XSL, al igual que en XML, deben contar con sus respectivos cierres.

Lo habitual, al igual que ocurre con las DTD que hemos conocido anteriormente, es que los documentos XSL se almacenen en archivos independientes, siendo referenciados desde el documento XML. De esta forma, en la cabecera de un documento XML se indicaría cuál es la DTD a la que debería ajustarse y el XSL necesario para transformarlo.

## XSL básico

Más que entrar en una definición más o menos formal de qué es XSL y cuál es su sintaxis, información que podrá encontrar en <http://www.w3.org/TR/WD-xsl>, vamos a ver con unos ejemplos sencillos qué podemos conseguir con XSL. Nuestro primer objetivo será generar un documento HTML a partir del documento XML usado como ejemplo en puntos previos, un documento conteniendo información relativa a productos de software y libros.

La primera línea de un documento XSL es la conocida marca `<?xml version="1.0" ?>`, con la que se inicia todo documento XML. Para diferenciar

al documento XSL de cualquier otro XML, y poder así usar elementos de manipulación de datos en lugar de introducir dichos datos, habrá que añadir la etiqueta **xsl:stylesheet**, que identifica al documento como una hoja de estilo. Dicha marca cuenta con un parámetro, llamado **xmlns** (*XML NameSpace*), que se utiliza para definir un espacio o ámbito de nombres. Tras dos puntos se facilitará el nombre de dicho espacio, mientras que como parámetro se entregará un URL, concretamente el URL en el que se define XSL.

El inicio de cualquier hoja de estilo XSL, por tanto, estará compuesto por las dos líneas siguientes:

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl= "http://www.w3.org/TR/WD-xsl" >
```

La etiqueta **xsl:stylesheet** no es de proceso, por lo que deberá contar con su respectiva marca de cierre. En este caso se ha asignado al espacio de nombres el nombre **xsl**, que es lo habitual. Podría, no obstante, asignarse cualquier otro nombre. En cualquier caso, hay que tener en cuenta que los elementos definidos en la [www.w3.org/TR/WD-xsl](http://www.w3.org/TR/WD-xsl), usados en el resto del documento, deberán ir precedidos del identificador del espacio de nombres. Para aplicar una plantilla de estilo, por ejemplo, se usa el elemento **template**. Si el documento se inicia con el fragmento anterior, dicho elemento aparecería como **xsl:template**.

Una hoja de estilo compuesta tan sólo de la cabecera anterior, junto con la etiqueta de cierre, no es que sea precisamente útil, pero ya podemos almacenarla en un archivo con extensión **XSL** y hacer referencia a ella desde un documento XML. Dicha referencia se efectúa mediante una marca de proceso como la siguiente:

```
<?xml-stylesheet type="text/xsl" href="EstiloSimple.xsl"?>
```

Como puede verse, la marca **xml-stylesheet** cuenta con dos parámetros: **type** y **href**. Con el primero se indica el tipo de hoja de estilo, que está codificada en XSL en este caso concreto, y con el segundo se facilita el URL en que se encuentra.

Si el documento XML es obtenido a través de la red desde un determinado servidor, normalmente su manipulación está restringida a hojas de estilo recuperadas del mismo servidor y no desde cualquier otro punto, ya sea local o remoto.

## Cómo aplicar plantillas

En el interior de la hoja de estilo se aplicarán plantillas a determinadas partes del documento XML. Estas plantillas establecen el formato de salida de la información, por ejemplo generando las etiquetas HTML asociadas a cada elemento del documento XML.

El elemento XSL a utilizar es **template**, pudiendo aplicarse a todo el documento o a una parte de él. Mediante el parámetro **match** se establecerá el conjunto

de datos al que se va a aplicar la plantilla. El argumento de dicho parámetro será `/`, para seleccionar todo el documento, o bien el nombre del elemento al que desee aplicarse la plantilla, por ejemplo `Libro`.

Entre la marca de apertura y la de cierre de la plantilla podremos introducir todos los elementos que deseemos aplicar, tanto de formato como de proceso. Los de formato serán simples etiquetas HTML, por ejemplo, mientras que los de proceso serán otros elementos, conoceremos algunos de ellos en un momento, que nos permitirán seleccionar datos del documento XML e introducirlos en la plantilla.

Observe el código siguiente, correspondiente a una hoja de estilo XSL almacenada en el archivo `PlantillaGeneral.xsl`. Tan sólo contiene una plantilla que se aplicará al documento completo, compuesta de varias etiquetas HTML. Observe que los elementos XSL están precedidos del prefijo `xsl` y dos puntos. Esto se debe a que en la marca `stylesheet` se ha llamado `xsl` al espacio de nombres. Si lo hubiésemos llamado de otra forma, habría que sustituir el prefijo `xsl` por el que correspondiese.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >

  <xsl:template match="/" >

    <H1>Lista de productos</H1>

    <H2>Software</H2>

    <H2>Libros</H2>

  </xsl:template>

</xsl:stylesheet>
```

La hoja de estilo por sí sola no es útil, lógicamente. Hay que aplicarla al documento XML, para lo cual habrá que incluir en éste la correspondiente marca de proceso `?xml-stylesheet`, tal y como puede apreciarse en el siguiente fragmento. Vamos a seguir usando en nuestros ejemplos el mismo documento de artículos utilizado con anterioridad.

```
<?xml version="1.0"?>
<!DOCTYPE ListaProductos SYSTEM "ListaProductos2.dtd">
<?xml-stylesheet type="text/xsl" href="PlantillaGeneral.xsl"?>
<ListaProductos>
  <Software>
    <Producto>
      <Nombre>Visual Basic .NET</Nombre>
      ...
    </Producto>
  </Software>
</ListaProductos>
```

Si usamos el editor XML de Visual Studio .NET seguiremos obteniendo la misma tabla de datos con los elementos del documento XML, pero abriendolo en Internet Explorer podremos ver que ya no aparece la lista jerárquica con todos los elementos y datos del documento. En su lugar, como puede verse en

la figura 19.11, sólo aparecen los títulos correspondientes a las etiquetas HTML introducidas en el interior de la plantilla XSL.

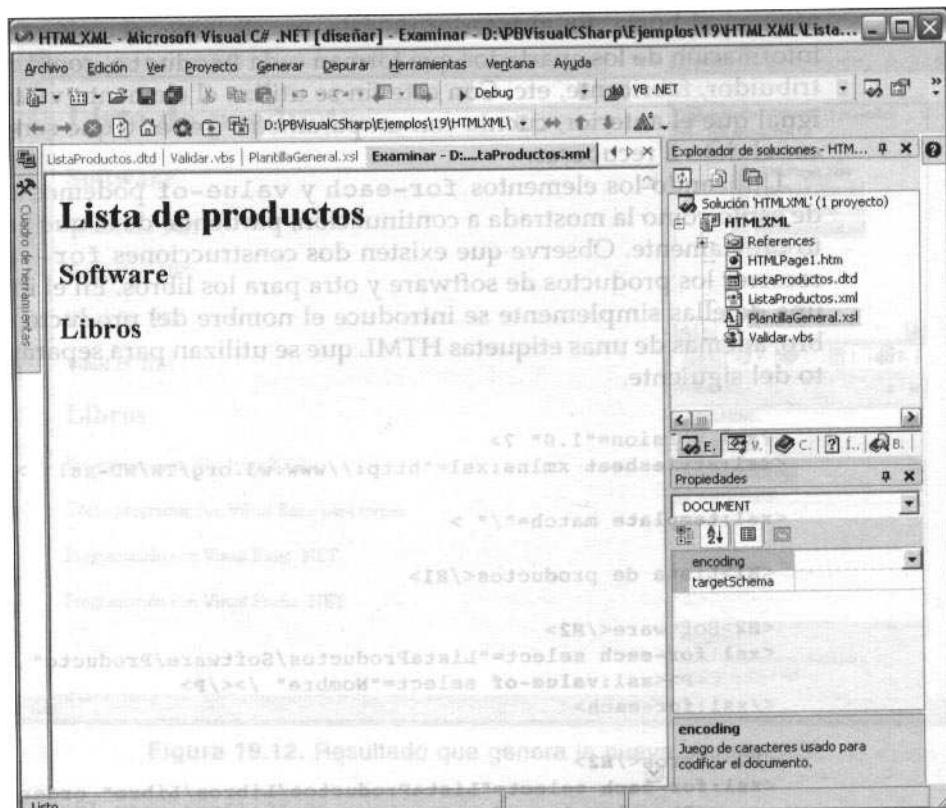


Figura 19.11. Resultado que genera la plantilla XSL simple

## Selección de datos

Una plantilla que genera un documento tan sólo con unos títulos, sin tomar la información almacenada en el documento XML, no es nada útil. Veamos, por tanto, cómo podemos seleccionar los datos de dicho documento, incluyéndolos en el interior de esa plantilla genérica para crear otra nueva, relativamente simple pero más útil.

Puesto que la estructura de todo documento XML suele estar compuesta de patrones repetitivos, en nuestro caso elementos `Producto` y `Libro` que se repiten en todo el documento, en XSL existe un elemento de enumeración llamado `for-each`. Éste dispone de un parámetro, llamado `select`, cuyo argumento determinará qué datos son los que se van a elegir. La etiqueta `<xsl:for-each select="ListaProductos/Software/Producto" >`, por ejemplo, elige todas las entradas `Producto` que se encuentren en la rama `Software` de la `ListaProductos`. Como puede deducirse, este elemento XSL lo que hace es

crear un bucle en el cual los elementos siguientes, hasta encontrar la etiqueta de cierre, se repiten una vez por cada entrada **Producto**.

Con el elemento **for-each**, por lo tanto, podemos determinar a qué datos afectará el bucle. En el interior de éste, no obstante, necesitaremos recuperar información de los apartados que forman cada **Producto**, como el nombre, distribuidor, fabricante, etc. Con este fin se utiliza el elemento **value-of** que, al igual que el anterior, cuenta con un parámetro **select** que servirá para indicar el dato a recuperar.

Utilizando los elementos **for-each** y **value-of** podemos crear una hoja de estilo como la mostrada a continuación, partiendo de la que habíamos escrito previamente. Observe que existen dos construcciones **for-each**, una para recorrer los productos de software y otra para los libros. En el interior de cada una de ellas simplemente se introduce el nombre del producto o título del libro, además de unas etiquetas HTML que se utilizan para separar cada elemento del siguiente.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >

<xsl:template match="/" >

    <H1>Lista de productos</H1>

    <H2>Software</H2>
    <xsl:for-each select="ListaProductos/Software/Producto" >
        <P><xsl:value-of select="Nombre" /></P>
    </xsl:for-each>

    <H2>Libros</H2>
    <xsl:for-each select="ListaProductos/Libros/Libro" order-by="Autor" >
        <P><xsl:value-of select="Titulo" /></P>
    </xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

Aplicando esta nueva hoja de estilo al documento XML, para lo cual habrá que sustituir la referencia que hay en éste, ahora obtendremos un resultado similar al de la figura 19.12. El documento es bastante más útil e, incluso, más claro que la visualización directa del árbol jerárquico XML, sobre todo para una persona que desconozca dicho lenguaje.

El resultado obtenido sigue siendo simple, pero realmente no necesitamos conocer ningún elemento XSL más para generar documentos mucho más elaborados. En el listado siguiente puede ver otra hoja XSL en la que tan sólo se usan los elementos ya conocidos, pero en la que se ha introducido más formato utilizando etiquetas HTML. Básicamente, se crea una tabla con unos títulos y en las celdillas centrales se muestran algunos datos del documento XML original. El resultado (véase la figura 19.13) es mucho mejor que el anterior, válido para una presentación al usuario.

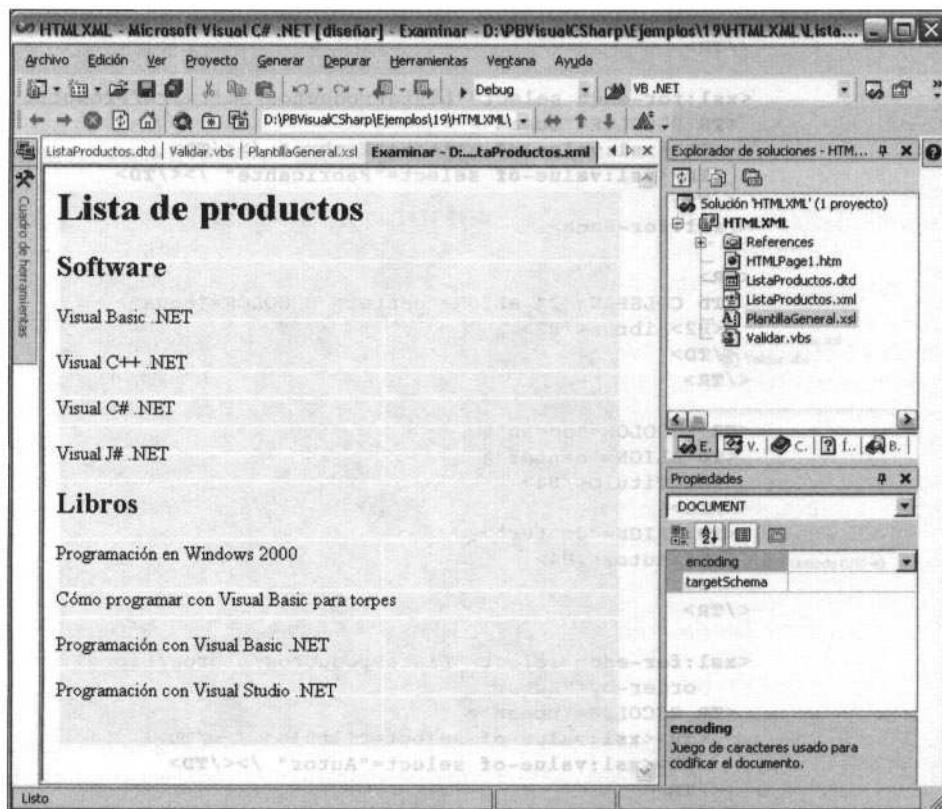


Figura 19.12. Resultado que genera la nueva plantilla XSL

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >

<xsl:template match="/" >

<TABLE style="border:1px solid">
<TR>
<TD COLSPAN="2" ALIGN="center" BGCOLOR="silver">
<H1>Lista de productos</H1>
</TD>
</TR>

<TR>
<TD COLSPAN="2" ALIGN="center" BGCOLOR="aqua">
<H2>Software</H2>
</TD>
</TR>

<TR BGCOLOR="ocean">
<TD ALIGN="center">
<H4>Nombre</H4>
</TD>
<TD ALIGN="center">
<H4>Fabricante</H4>
```

```

        </TD>
    </TR>

    <xsl:for-each select="ListaProductos/Software/Producto" >
        <TR BGCOLOR="ocean">
            <TD><xsl:value-of select="Nombre" /></TD>
            <TD><xsl:value-of select="Fabricante" /></TD>
        </TR>
    </xsl:for-each>

    <TR>
        <TD COLSPAN="2" ALIGN="center" BGCOLOR="aqua">
            <H2>Libros</H2>
        </TD>
    </TR>

    <TR BGCOLOR="ocean">
        <TD ALIGN="center">
            <H4>Titulo</H4>
        </TD>
        <TD ALIGN="center">
            <H4>Autor</H4>
        </TD>
    </TR>

    <xsl:for-each select="ListaProductos/Libros/Libro"
        order-by="Autor">
        <TR BGCOLOR="ocean">
            <TD><xsl:value-of select="Titulo" /></TD>
            <TD><xsl:value-of select="Autor" /></TD>
        </TR>
    </xsl:for-each>
    </TABLE>
</xsl:template>

</xsl:stylesheet>

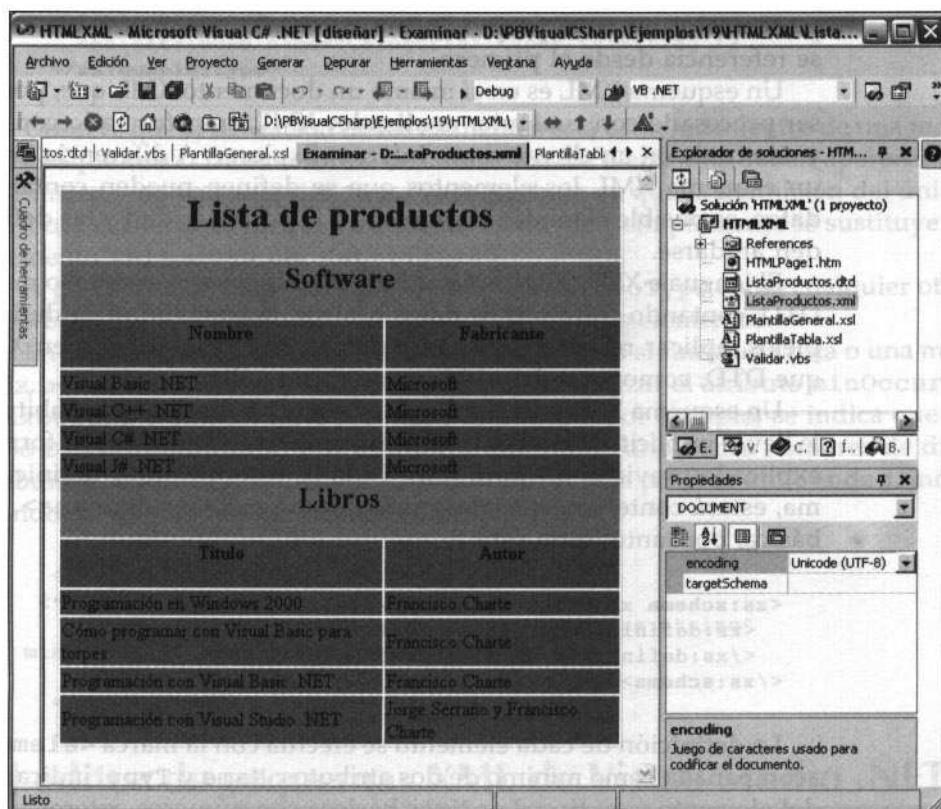
```

## Las posibilidades de XSL

---

Tan sólo con tres elementos XSL, los explicados `template`, `for-each` y `value-of`, es posible convertir cualquier documento XML en una página HTML tan compleja como se deseé. Las posibilidades de XSL, no obstante, no terminan aquí. Hay más elementos de manipulación de datos, así como un complejo y eficiente lenguaje de comparación de patrones que facilita la selección de información y toma de alternativas.

Al seleccionar los conjuntos de datos, con el elemento `for-each`, es posible ordenarlos según el criterio que nos interese. Podríamos, por ejemplo, ordenar los productos por fabricante o los libros por autores. Con elementos tales como `choose`, `when` y `otherwise` es posible plantear alternativas, o tomar decisiones en el argot de los programadores. Esto puede utilizarse, por ejemplo, para aplicar un formato condicional a los datos. Mediante el elemento `eval` podríamos evaluar expresiones incluyendo el resultado en el documento que va a ser generado.



**Figura 19.13.** La nueva plantilla XSL genera una tabla HTML, que resulta mucho más legible

Además de los elementos propiamente dichos, en XSL existe una serie de métodos predefinidos que facilitan, por ejemplo, la salida de fechas, horas y números, así como la obtención de los índices que identifican a cada uno de los elementos del documento XML.

## Introducción a XSD

Previamente, en este mismo capítulo, hemos visto cómo la estructura de un documento XML puede representarse mediante una DTD que, además, puede ser utilizada por el analizador XML para efectuar el proceso de validación. En principio las DTD eran el único mecanismo para documentar la estructura, pero durante meses un grupo de trabajo del W3C se ha centrado en la definición de un nuevo sistema basado en esquemas.

Los esquemas XML son más flexibles que las DTD y se trata de un mecanismo que Microsoft está potenciando en sus aplicaciones. El analizador XML de Microsoft es capaz de trabajar con esquemas que, como en el caso de las DTD,

pueden alojarse en el propio documento XML o en un archivo separado al que se referencia desde el primero.

Un esquema XML es en sí mismo un documento XML que, por tanto, puede ser procesado con cualquier analizador XML, transformado con una hoja XSL y mostrado en un cliente como Internet Explorer. A diferencia de una DTD, en un esquema XML los elementos que se definen pueden contar con tipos de datos, es posible extender el esquema con posterioridad y las definiciones pueden anidarse.

El lenguaje XSD (*XML Schema Definition Language*) es mucho más versátil que DTD, contando con tipos de datos básicos, la posibilidad de definir tipos complejos, aplicar restricciones, etc. Además, XSD está basado en XML, mientras que DTD, como ya se ha visto, no se ajusta a dicho estándar.

Un esquema XSD se inicia con la etiqueta `<schema>` que, habitualmente, suele llevar implícita la declaración de un ámbito con nombre, de forma similar a lo explicado previamente para XSL. Toda la descripción, la definición del esquema, estará contenida entre las marcas `<schema>` y `</schema>`. La estructura básica, por tanto, sería ésta:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:declaración de elementos>
    </xs:declaración de elementos>
</xs:schema>
```

La definición de cada elemento se efectúa con la marca `<element>`. Ésta irá acompañada como mínimo de dos atributos: `Name` y `Type`, indicando el nombre del elemento y su tipo. Los tipos básicos son números enteros y con parte decimal, cadenas, fechas, etc. Prácticamente se contempla el uso de cualquier tipo de dato existente en los lenguajes actuales.

```
<xs:element name="Titulo" type="xs:string"/>
```

Con la etiqueta anterior, por ejemplo, se define un elemento que tendrá por nombre `Titulo` que contendrá una cadena de caracteres. Observe que se utiliza el prefijo `xs` como nombre del espacio que asumimos se ha definido previamente.

En caso de que el tipo no sea tan simple como un número o una cadena, será necesaria una descripción indicando los elementos de que se compone. En este caso la etiqueta que nos interesa es `<ComplexType>` que, como puede ver en el fragmento siguiente, dispone de un atributo `name` que permite indicar el nombre del tipo. En su interior se definirán los elementos, según la sintaxis de la etiqueta `<element>` que acaba de explicarse, y, en caso necesario, también los atributos.

```
<xs:complexType name="LibroType">
    <xs:sequence>
        <xs:element name="Titulo" type="xs:string"/>
        <xs:element name="Autor" type="xs:string"/>
        <xs:element name="Editorial" type="xs:string"/>
```

```
</xs:sequence>
<xs:attribute name="Categoria" type="xs:string"/>
</xs:complexType>
```

Observe que la definición de los elementos está contenida dentro de una marca `<sequence>`. De esta forma se indica que los elementos deben aparecer en ese orden y no en cualquier otro. Fíjese también en que la definición del único atributo existente es idéntica a la de los elementos, simplemente se sustituye el elemento `<element>` por `<attribute>`.

A partir de este momento, podríamos usar `LibroType` como cualquier otro tipo básico, utilizándolo en la definición de nuevos elementos.

En caso de que un elemento deba repetirse, como si fuese una lista o una matriz, aparte del nombre y el tipo será necesario utilizar el atributo `minOccurs`, `maxOccurs` o ambos. En la siguiente definición, por ejemplo, se indica que el tipo `DistribuidoresType` estará compuesto de uno o más nombres de distribuidor. Observe que el valor de `maxOccurs` es la palabra `unbounded`, indicando así que no hay un máximo preestablecido.

```
<xs:complexType name="DistribuidoresType">
  <xs:sequence>
    <xs:element name="Distribuidor" type="xs:string"
      minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

## El editor de esquemas XML de Visual Studio .NET

Visual Studio .NET tiene un editor de esquemas que podríamos calificar de diseñador ya que, de forma casi visual, podemos establecer los tipos de datos y sus elementos, así como las relaciones que darán lugar a la jerarquía del documento. Para añadir un nuevo esquema XML nos serviremos, como es habitual, del cuadro de diálogo **Agregar nuevo elemento** mostrado en la figura 19.14. Automáticamente se abrirá el diseñador de esquemas en el que, utilizando las opciones del menú emergente, iremos añadiendo elementos, atributos y tipos.

Usando este diseñador sería fácil definir los elementos de cada tipo que contendrá un documento, en nuestro caso esos tipos serían `Software` y `Libros` que, a su vez, contendrán otros tipos. Si ya tenemos un documento XML creado, podemos generar automáticamente el esquema XSD utilizando la opción `XML>Crear esquema`. Generado el esquema, ése se asociará con el documento XML y podremos usar la opción `XML>Validar datos XML` para comprobar su validez. Al ir introduciendo datos, además, los posibles errores se indicarán como es habitual en el editor de código de Visual Studio, con un subrayado de color.

## Codificación manual del esquema

Aunque sólo se han descrito los conceptos mínimos de XSD, tenemos suficiente información para crear un esquema que describa la estructura de nuestros

documentos XML, concretamente el último usado como ejemplo compuesto de una lista de productos, sin necesidad de recurrir al diseñador que incorpora Visual Studio .NET. El esquema sería el siguiente:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="ListaProductos" type="ListaProductosType" />
  <xs:complexType name="ListaProductosType">
    <xs:sequence>
      <xs:element name="Software" type="SoftwareType" />
      <xs:element name="Libros" type="LibrosType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="SoftwareType">
    <xs:sequence>
      <xs:element name="Producto" type="xs:string"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="LibrosType">
    <xs:sequence>
      <xs:element name="Libro" type="LibroType"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="ProductoType">
    <xs:sequence>
      <xs:element name="Nombre" type="xs:string" />
      <xs:element name="Descripcion" type="xs:string" />
      <xs:element name="Fabricante" type="xs:string" />
      <xs:element name="Distribuidores" type="DistribuidoresType" />
    </xs:sequence>
    <xs:attribute name="Tipo" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="LibroType">
    <xs:sequence>
      <xs:element name="Titulo" type="xs:string" />
      <xs:element name="Autor" type="xs:string" />
      <xs:element name="Editorial" type="xs:string" />
    </xs:sequence>
    <xs:attribute name="Categoria" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="DistribuidoresType">
    <xs:sequence>
      <xs:element name="Distribuidor" type="xs:string"
        minOccurs="1" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

En este esquema se indica que nuestro documento tendrá un elemento denominado **ListaProductos** que será de tipo **ListaProductosType**. En dicho tipo se especifica que el elemento contará con dos subelementos: **Software** y **Libros**, que deberán aparecer en ese orden. Continuando con el análisis llegaremos hasta los tipos más sencillos, los elementos y atributos que contienen un dato simple como **string**.

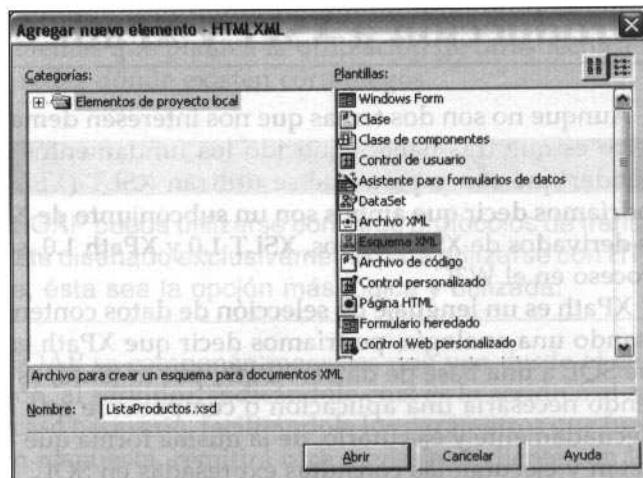


Figura 19.14. Añadimos un nuevo esquema XML

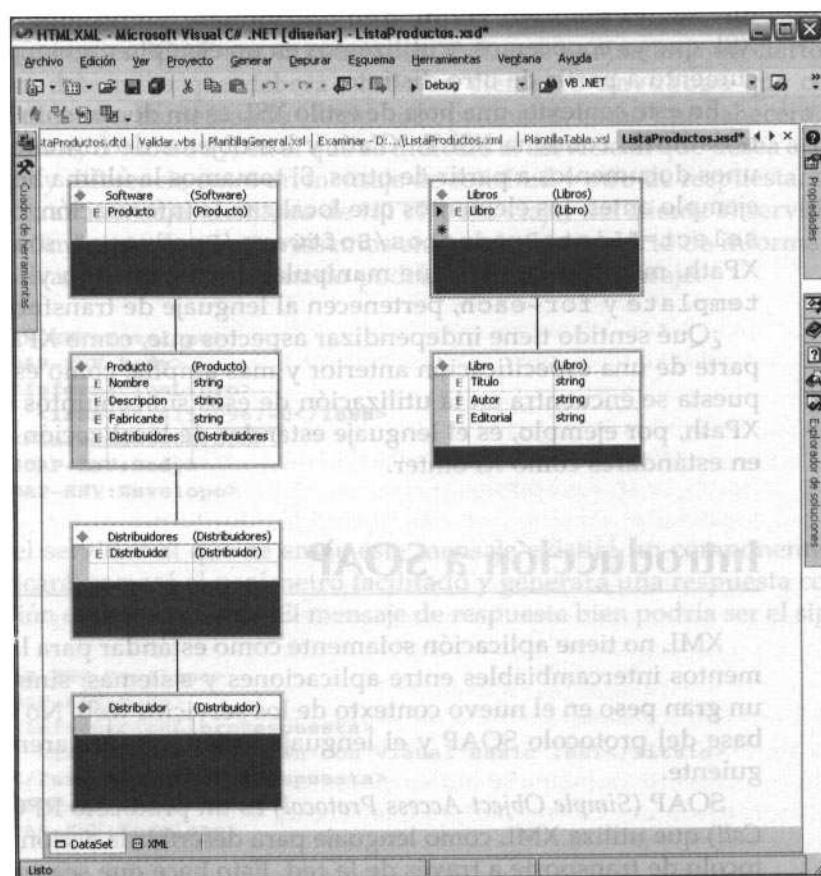


Figura 19.15. Aspecto del diseñador de esquemas durante la creación del esquema para nuestro documento

## Introducción a XSLT y XPath

Aunque no son dos temas que nos interesen demasiado en este momento, lo cierto es que tras haber conocido los fundamentos de XSL no es difícil comprender qué son y para qué se utilizan XSLT (*XSL Transformations*) y XPath. Podríamos decir que ambos son un subconjunto de XSL, o bien que son lenguajes derivados de XSL. Ambos, XSLT 1.0 y XPath 1.0, son actualmente trabajos en proceso en el W3C.

XPath es un lenguaje de selección de datos contenidos en documentos XML. Usando una analogía, podríamos decir que XPath es a un documento XML lo que SQL a una base de datos. XPath, por lo tanto, es simplemente un lenguaje, siendo necesaria una aplicación o componente que sea capaz de interpretarlo adecuadamente y ejecutarlo, de la misma forma que los sistemas RDBMS interpretan y ejecutan las consultas expresadas en SQL.

XSLT es un lenguaje de transformación de datos, utilizando XPath para seleccionar la información de un documento XML y, posteriormente, generar otro distinto, ya sea XML, HTML o en otro formato. En realidad, XSLT es la parte de XSL que ya conocemos, y utilizamos en un ejemplo previo, para generar un documento a partir de otro distinto.

En este contexto, una hoja de estilo XSL es un documento que, utilizando sintaxis XML, combina XSLT, XPath y los objetos de formato XSL para generar unos documentos a partir de otros. Si tomamos la última hoja XSL usada como ejemplo antes, los elementos que localizan la información, como `match="/" o select="ListaProductos/Software/Producto"`, son parte del lenguaje XPath, mientras que los que manipulan la información y la convierten, como `template` y `for-each`, pertenecen al lenguaje de transformación XSLT.

¿Qué sentido tiene independizar aspectos que, como XPath y XSLT, forman parte de una especificación anterior y más amplia, como es la de XSL? La respuesta se encuentra en la utilización de esos subconjuntos en otros contextos. XPath, por ejemplo, es el lenguaje estándar de localización de elementos XML en estándares como XPointer.

## Introducción a SOAP

XML no tiene aplicación solamente como estándar para la creación de documentos intercambiables entre aplicaciones y sistemas, sino que tiene también un gran peso en el nuevo contexto de los servicios web. No en vano, XML es la base del protocolo SOAP y el lenguaje WSDL que trataremos en el punto siguiente.

SOAP (*Simple Object Access Protocol*) es un protocolo RPC (*Remote Procedure Call*) que utiliza XML como lenguaje para describir la acción y HTTP como protocolo de transporte a través de la red. Esto hace que sea compatible con cualquier lenguaje y sistema operativo, ya que XML es un estándar ampliamente aceptado. Al mismo tiempo, usando HTTP como protocolo de transporte se

evitan los problemas que plantea la utilización de otras técnicas, como DCOM o CORBA, en redes donde existen cortafuegos.

#### Nota

Realmente SOAP puede utilizarse con otros protocolos de transporte, como SMTP, no está diseñado exclusivamente para utilizarse con HTTP aunque, actualmente, ésta sea la opción más lógica y utilizada.

Mediante SOAP se componen mensajes que van desde el cliente, que efectúa una petición, al servidor, indicándole cuál es la acción o el servicio que desea usar y, en caso necesario, facilitándole los parámetros que fuesen necesarios. El servidor, en respuesta, remitirá otro mensaje al cliente con los resultados.

## Estructura de un mensaje SOAP

Los mensajes SOAP son documentos XML que se ajustan a un cierto esquema. Éste determina que deberá existir un elemento **Envelope** que contenga todo el mensaje, conteniendo elementos opcionales, como una cabecera, y otros obligatorios, como **Body**, en el que se indicará el servicio al que desea accederse.

No hay diferencia entre un mensaje de solicitud y otro de respuesta, salvo en que utilizan mensajes distintos de HTTP para viajar del cliente al servidor o viceversa. Imaginando que tuviésemos en marcha un servicio de información bibliográfica, un cliente imaginario podría enviar este mensaje:

```
<SOAP-ENV:Envelope>
<SOAP-ENV:Body>
  <InformacionLibro>
    <ISBN>84-415-0967-0</ISBN>
  </InformacionLibro>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

En el servidor al que se envíe este mensaje existirá un componente que lo identificará, tomará el parámetro facilitado y generará una respuesta con la información correspondiente. El mensaje de respuesta bien podría ser el siguiente:

```
<SOAP-ENV:Envelope>
<SOAP-ENV:Body>
  <InformacionLibroRespuesta>
    <Titulo>Programación con Visual Basic .NET</Titulo>
  </InformacionLibroRespuesta>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Elementos como **InformacionLibro** e **InformacionLibroRespuesta** estarían definidos en algún esquema, mientras que los elementos **Envelope** y **Body** son parte de la especificación de SOAP.

## SOAP y Visual Studio .NET

---

Visual Studio .NET utiliza SOAP para permitir la comunicación entre servicios Windows y las aplicaciones que consumen dichos servicios. Todo el proceso de generación de los mensajes, transferencia, recepción e interpretación queda en manos de objetos ya predefinidos en la plataforma .NET, de modo que nosotros nunca tendremos que trabajar manualmente con este tipo de información.

## Introducción a WSDL y UDDI

---

Para poder invocar a un servicio web, componiendo adecuadamente el mensaje SOAP para efectuar la solicitud, es necesario contar con una descripción de ese servicio. En esa descripción se enumeran las funciones con que cuenta el servicio, los parámetros que necesitan, las respuestas que generan y los tipos de datos utilizados en todo el proceso. Las empresas ofrecerán esos servicios a través de Internet, y la descripción, lógicamente, estará almacenada en un servidor. Es necesario, por tanto, algún mecanismo que permita localizar la descripción. Aquí es donde entran en escena UDDI y WSDL.

UDDI (*Universal Discovery, Description and Integration Service*) es un servicio de directorio puesto en marcha por empresas como Microsoft e IBM. Su finalidad es servir, a modo de una guía telefónica, para localizar el tipo de servicio por el que está interesado el cliente. El servicio UDDI le facilitará las referencias necesarias para acceder a la descripción de esos servicios.

La descripción de un servicio web se efectúa mediante WSDL (*Web Service Description Language*), otro lenguaje basado en XML. En un archivo WSDL se combinan elementos XML específicos para la descripción del servicio, por ejemplo los conectores para acceder a él, y esquemas que definen los datos que se enviarán o devolverán las funciones del servicio.

El funcionamiento conjunto de UDDI, WSDL y SOAP queda reflejado en la figura 19.16. La aplicación recurre, en primera instancia, al registro UDDI para localizar el servicio que está buscando. De dicho registro obtiene uno o varios URL, que usa para acceder a los servidores que ofrecen esos servicios en busca de la descripción WSDL. Disponiendo de ésta, la aplicación ya puede componer adecuadamente los mensajes SOAP para consumir esos servicios.

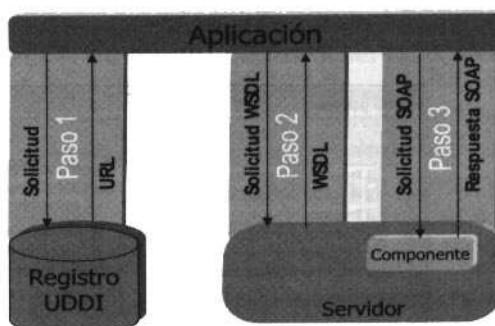
No tenemos que preocuparnos por la generación de la descripción WSDL, ya que ésta es una tarea que realiza automáticamente Visual Studio .NET como podremos experimentar en los puntos siguientes. También hay herramientas de terceros para efectuar ese trabajo.

## Creación de un servicio Web

---

Para poder hacer pruebas con un servicio Web, desde cualquier tipo de aplicación, primero debemos disponer del propio servicio. Por ello comenzaremos

creando un servicio simple, dispondrá tan sólo de un método cuya finalidad será devolver una cadena de caracteres con la fecha y hora actuales. Será suficiente, no obstante, para que conozcamos la estructura básica de un servicio Web y podamos, posteriormente, utilizarlo como base para crear otro de mayor complejidad.



**Figura 19.16.** Esquema de funcionamiento de una aplicación que consume un servicio web

Aunque vamos a usar Visual Studio .NET para crear nuestro servicio Web, realmente no necesitaríamos más herramientas que el Bloc de notas para escribir un módulo que sería compilado dinámicamente por parte de ASP.NET.

Los servicios Web son accesibles, generalmente, a través de un servidor Web. El punto de entrada es un archivo con extensión **asmx**. Éste puede contener el código del servicio o una referencia a un módulo que contiene dicho código.

## Anatomía de un servicio Web

La clase de la cual deben derivarse los servicios web es **WebService**, alojada en el ámbito con nombre **System.Web.Services**. En el interior de esta nueva clase, que heredará los miembros de **WebService**, podemos incluir todos los métodos que necesitemos, ya sean privados, protegidos o públicos, usando la sintaxis propia del lenguaje que estemos usando. Ninguno de esos métodos, sin embargo, sería accesible para un consumidor del servicio.

Aparte de listas de parámetros, tipos de retorno y especificadores, un método también puede contar con atributos, según vimos ya en un capítulo previo. Éstos se introducen entre los símbolos [ ] y ] en el caso de Visual C# .NET, y se colocan, generalmente, delante de la cabecera del método.

Si deseamos que una cierta clase pueda considerarse un servicio web, aparte de derivarla de la clase **WebService** también deberemos definir al menos un método con el atributo **WebMethod**. De esta forma indicaríamos qué métodos de la clase estarán accesibles para los consumidores, simplemente disponiendo [**WebMethod**] delante de la cabecera de los métodos que nos interesen.

Por último, el documento **asmx** deberá iniciarse con la directiva **<%@ WebService**, indicándose el lenguaje en el que está escrito el código o, si procede, facilitando el ámbito y nombre de la clase si ésta ya se encuentra precompilada.

## El servicio horario

Para crear nuestro servicio Web nos serviremos, una vez más, de los asistentes que hay en la ventana **Nuevo proyecto**, como puede verse en la figura 19.17. Este asistente generará el archivo **asmx** indicando en él que el código del servicio Web se encuentra en el módulo **asmx.cs** que, como puede suponer, sólo contiene código Visual C# .NET.

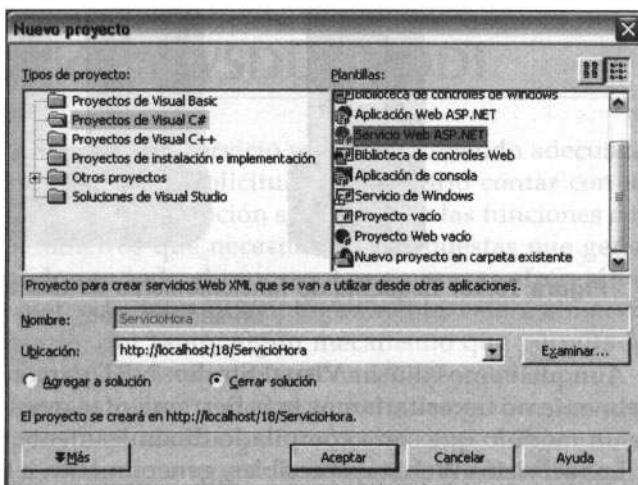


Figura 19.17. Iniciamos el desarrollo de nuestro servicio Web

En principio se encontrará con un diseñador muy similar al utilizado en un capítulo previo para la creación de formularios Web. Pulse sobre el enlace que le permite acceder al código y después observe la definición de la clase, derivada de **System.Web.Services.WebService**. En el constructor hay una llamada al método **InitializeComponent()**, como en otros tipos de aplicación, si bien en este caso dicho método está vacío.

Añada el código siguiente al final de la clase:

```
[WebMethod()]
public string Horas()
{
    return DateTime.Now.ToString();
}
```

Como puede ver, el método difícilmente podría ser más sencillo. Recuperamos la fecha y hora actuales, de la propiedad **Now** de la clase **DateTime**, convertimos el dato en una cadena y lo devolvemos. Lo único especial es el atributo **[WebMethod]** delante del método.

Sólo con esto ya tiene desarrollado su primer servicio Web, no necesita hacer nada más. Desde cualquier ordenador podría obtenerse la hora del servidor en que se ejecuta el servicio, sin necesidad de escribir ninguna aplicación y sin importar la plataforma.

## El módulo asmx

Hasta ahora hemos trabajado con el módulo que contiene el código del servicio, un módulo que será compilado desde Visual Studio .NET generando un ensamblado y que, por tanto, no habrá que compilar dinámicamente. Aunque en el Explorador de soluciones aparece el módulo **asmx**, si hace doble clic sobre él se encontrará con el módulo de código, el módulo **asmx.cs**. Esto es así porque el módulo **asmx**, que es el punto de entrada al servicio Web, es mantenido internamente por Visual Studio .NET.

Para abrir el módulo **asmx** tendrá que desplegar su menú emergente, pulsando sobre dicho módulo en el Explorador de soluciones con el botón secundario del ratón, y elegir la opción **Abrir con**. Aparecerá una ventana, similar a la de la figura 19.18, en la que podremos elegir el tipo de editor a utilizar. Tras pulsar el botón **Abrir** podrá, finalmente, ver el contenido del módulo **asmx**.

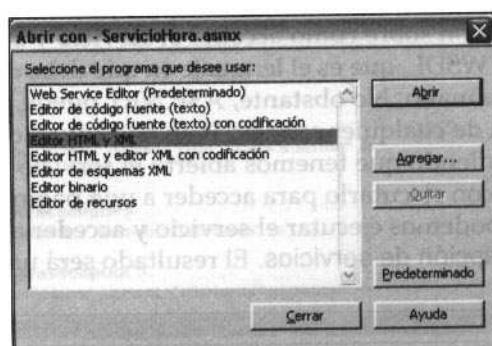


Figura 19.18. Seleccionamos el tipo de editor para abrir el módulo asmx

El contenido inicial correspondiente a este módulo es el que puede verse a continuación:

```
<%@ WebService Language="c#"
    Codebehind="Servicel.asmx.cs"
    Class="ServicioHorario.Servicel" %>
```

Simplemente se indica el lenguaje, módulo y clase donde se encuentran los métodos del servicio Web. En la práctica, podríamos sustituir ese código por el siguiente y olvidarnos del módulo **asmx.cs**, incluyendo toda la lógica en el módulo **asmx**. La compilación, además, sería dinámica.

```
<%@ WebService Language="c#" class="ServicioHorario.ServicioHorario"
%>

using System;
using System.Web.Services;

// Nuestra clase es derivada de WebService
public class ServicioHora : WebService
```

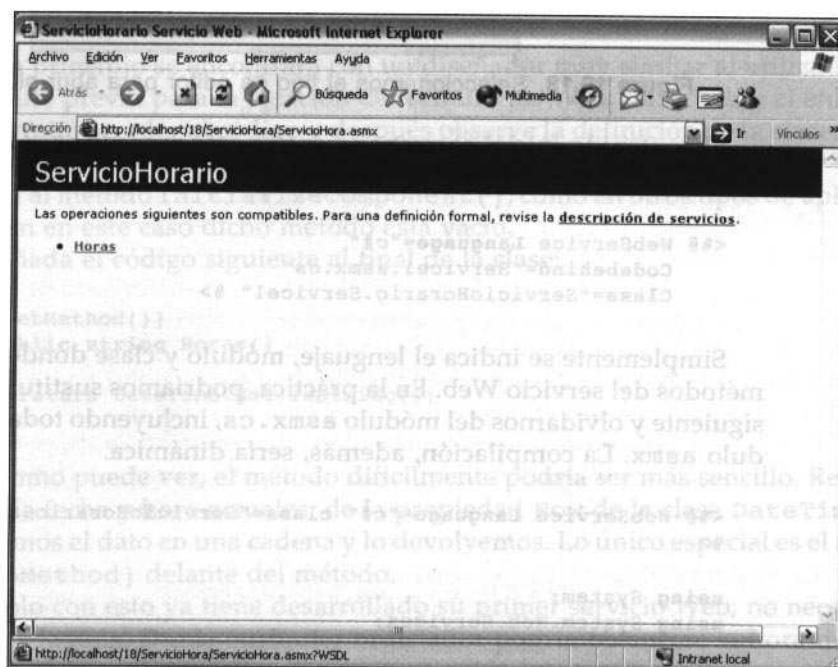
```
// Marcamos el único método como WebMethod
[WebMethod]
public string Horas()
{ // devolvemos la fecha y hora
return DateTime.Now.ToString();
}
```

Aunque nuestro proyecto tiene, aparte de los mencionados módulos `asmx` y `asmx.vb`, un archivo de configuración, `web.config`, y algunos elementos más, lo cierto es que un archivo con extensión `asmx` conteniendo el código anterior y colocado en la carpeta raíz del servidor bastaría para tener un servicio web.

## Módulo de descripción del servicio

Si queremos registrar nuestro servicio Web en un directorio, para que cualquiera pueda hacer uso de él, necesitamos un módulo de descripción. En él habría información sobre cómo acceder al servicio y a sus métodos. Conociendo la sintaxis de WSDL, que es el lenguaje en que debe darse esta descripción, podríamos crearla a mano. No obstante, ASP.NET puede generar automáticamente la descripción de cualquier servicio Web que hayamos creado con Visual Studio .NET.

Asumiendo que tenemos abierto el servicio horario que hemos desarrollado, bastará con ejecutarlo para acceder a una página como la de la figura 19.19. Desde ella podemos ejecutar el servicio y acceder a su descripción pulsando el enlace **descripción de servicios**. El resultado será un documento XML (figura 19.20).



**Figura 19.19.** Página obtenida al ejecutar el proyecto

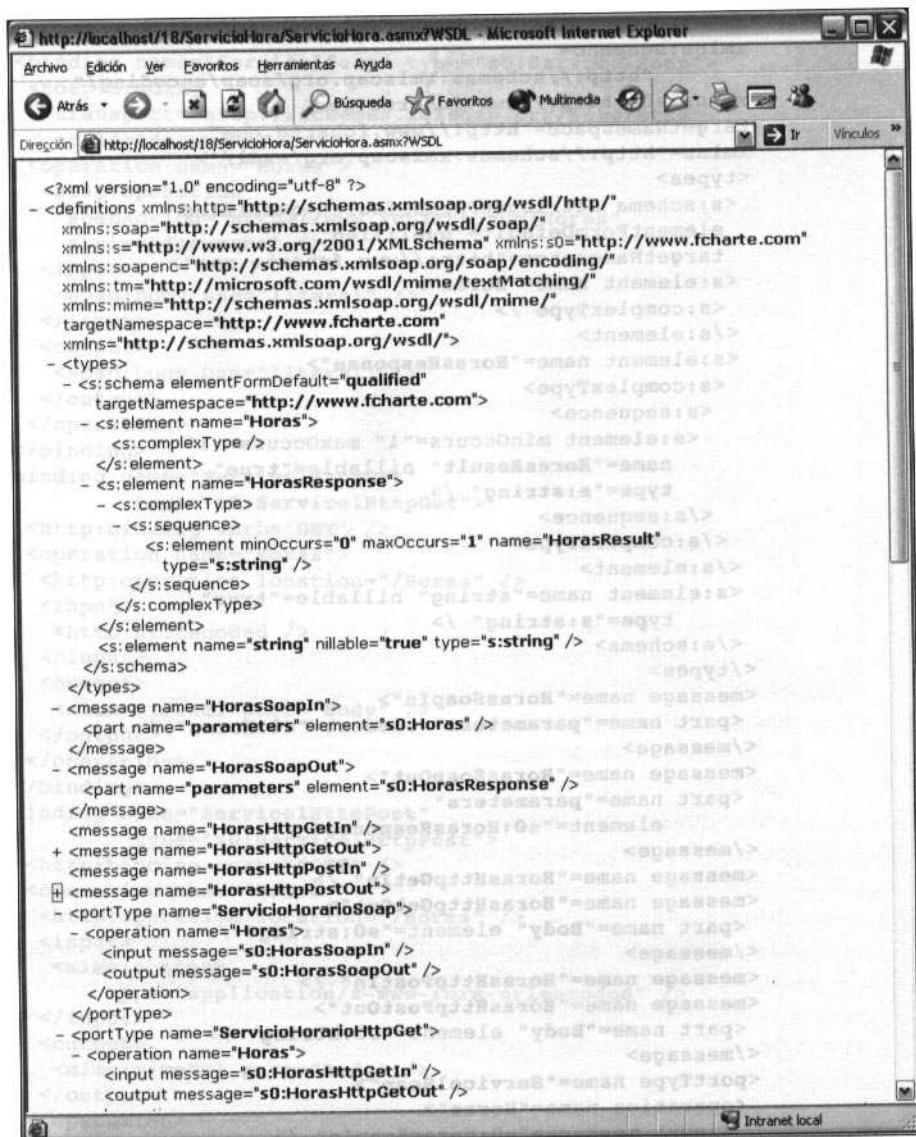


Figura 19.20. Descripción WSDL tal y como aparece en Internet Explorer

Tal y como se aprecia en el código siguiente, la descripción WSDL de nuestro servicio se compone, claramente, de varias secciones distintas.

```

<?xml version="1.0" encoding="utf-8" ?>
<definitions
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tm=
    "http://microsoft.com/wsdl/mime/textMatching/">
```

```
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc=
    "http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="http://www.fcharte.com"
targetNamespace="http://www.fcharte.com"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
    <s:schema attributeFormDefault="qualified"
        elementFormDefault="qualified"
        targetNamespace="http://www.fcharte.com">
        <s:element name="Horas">
            <s:complexType />
        </s:element>
        <s:element name="HorasResponse">
            <s:complexType>
                <s:sequence>
                    <s:element minOccurs="1" maxOccurs="1"
                        name="HorasResult" nillable="true"
                        type="s:string" />
                </s:sequence>
            </s:complexType>
        </s:element>
        <s:element name="string" nillable="true"
            type="s:string" />
    </s:schema>
</types>
<message name="HorasSoapIn">
    <part name="parameters" element="s0:Horas" />
</message>
<message name="HorasSoapOut">
    <part name="parameters"
        element="s0:HorasResponse" />
</message>
<message name="HorasHttpGetIn" />
<message name="HorasHttpGetOut">
    <part name="Body" element="s0:string" />
</message>
<message name="HorasHttpPostIn" />
<message name="HorasHttpPostOut">
    <part name="Body" element="s0:string" />
</message>
<portType name="Service1Soap">
    <operation name="Horas">
        <input message="s0:HorasSoapIn" />
        <output message="s0:HorasSoapOut" />
    </operation>
</portType>
<portType name="Service1HttpGet">
    <operation name="Horas">
        <input message="s0:HorasHttpGetIn" />
        <output message="s0:HorasHttpGetOut" />
    </operation>
</portType>
<portType name="Service1HttpPost">
    <operation name="Horas">
        <input message="s0:HorasHttpPostIn" />
        <output message="s0:HorasHttpPostOut" />
    </operation>
</portType>
```

```
</portType>
<binding name="Service1Soap" type="s0:Service1Soap">
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <operation name="Horas">
    <soap:operation
      soapAction="http://www.fcharte.com/Horas"
      style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
</binding>
<binding name="Service1HttpGet"
  type="s0:Service1HttpGet">
  <http:binding verb="GET" />
  <operation name="Horas">
    <http:operation location="/Horas" />
    <input>
      <http:urlEncoded />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<binding name="Service1HttpPost"
  type="s0:Service1HttpPost">
  <http:binding verb="POST" />
  <operation name="Horas">
    <http:operation location="/Horas" />
    <input>
      <mime:content
        type="application/x-www-form-urlencoded" />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<service name="Service1">
  <port name="Service1Soap"
    binding="s0:Service1Soap">
    <soap:address
      location="http://localhost/ ServicioHora.asmx" />
  </port>
  <port name="Service1HttpGet"
    binding="s0:Service1HttpGet">
    <http:address
      location="http://localhost/ ServicioHora.asmx" />
  </port>
  <port name="Service1HttpPost"
    binding="s0:Service1HttpPost">
    <http:address
```

```
    location="http://localhost/ServicioHora.asmx" />
  </port>
</service>
</definitions>
```

Todo el documento se encuentra contenido en un árbol cuyo elemento raíz es la etiqueta `<definitions>`, en cuyo interior podemos distinguir varias partes. En la primera se definen los tipos de datos utilizados en el servicio, en este caso `HorasResponse` que se define como una cadena de caracteres que puede ser nula. Después encontramos la enumeración de los mensajes que pueden enviarse al servicio y los puertos de comunicación. Éstos describen el enlace necesario para poder utilizar el servicio mediante SOAP, una solicitud GET de http o un POST de ese mismo protocolo. Finalmente, encontramos una descripción del servicio y las direcciones y puertos de acceso.

## Consumo de un servicio Web

---

Ya tenemos un servicio web creado y, tras compilarlo, instalado en nuestro sistema, pero no sabremos si realmente funciona o no hasta que lo consumamos. Como se ha visto en la descripción WSDL, podríamos utilizar con tal fin el protocolo HTTP, bien a través del método GET o con POST, o el protocolo SOAP.

Si pretendemos acceder al servidor desde un cliente Web, como es Internet Explorer, lo normal es que nos sirvamos de HTTP. Realmente no necesitamos diseñar una página ni nada parecido, puesto que es suficiente con introducir la solicitud indicada en la propia descripción WSDL. Inténtelo con el URL `http://localhost/ServicioHorario/ServicioHorario.asmx/Horas`, asumiendo que ha utilizado los mismos nombres del ejemplo desarrollado previamente. Debe aparecer una respuesta como la de la figura 19.21. Es un documento XML conteniendo el resultado de la ejecución del servicio.

Haciendo en el URL referencia directa al servicio Web, sin indicar nombre de método alguno, conseguiremos que ASP.NET obtenga de la WSDL toda la información necesaria para crear una página dinámica en la que se indiquen todos los métodos que existen, los parámetros que toman, sus valores de retorno, etc. En esa página, además, existirán los controles necesarios para que podamos facilitar parámetros y efectuar llamadas. Es una forma rápida de ver el funcionamiento de un servicio.

Introduzca el siguiente URL: `http://localhost/ServicioHorario/ServicioHorario.asmx` y observe lo que ocurre. En el documento devuelto haga clic sobre el enlace `Horas` con el fin de acceder a la página, similar a la de la figura 19.22, en la que puede ver toda la información existente sobre el método. Como se vio antes, también puede abrir la descripción WSDL usando el enlace apropiado.

Aunque es éste un método cómodo y rápido para comprobar el funcionamiento de los servicios Web, desde luego no será el preferido para que los clientes lo utilicen. Lo habitual será crear una aplicación que, sirviendo de interfaz para

el usuario final, sea la que consuma el servicio. En los puntos siguientes, por tanto, lo que haremos será actuar como consumidores de ese servicio desarrollando aplicaciones para el usuario.

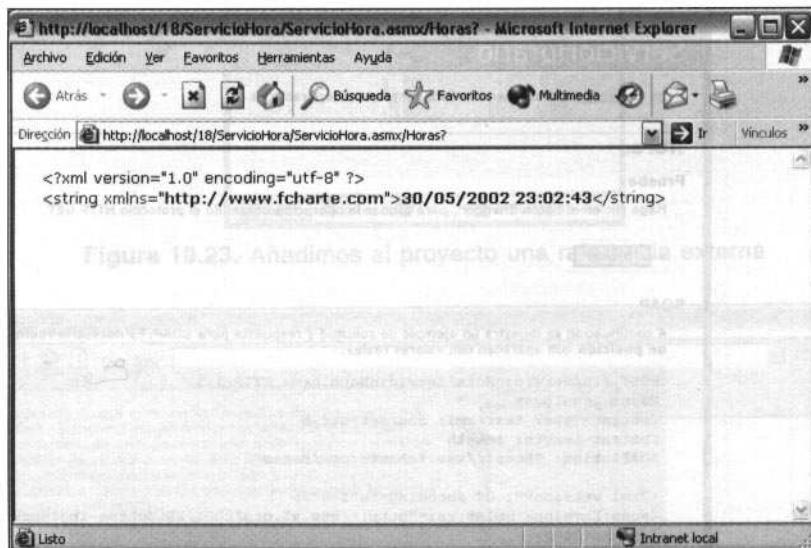


Figura 19.21. Consumimos el servicio introduciendo una solicitud HTTP

## Referencias externas

Un servicio Web puede ser consumido desde cualquier tipo de aplicación, ya se ejecute en Windows nativamente, sea una aplicación Web o use otro sistema operativo, sin importar el lenguaje que se haya usado. Para poder consumir el servicio que acabamos de escribir, asumiendo que la aplicación consumidora estará en cualquier punto y no precisamente en el ordenador donde se ejecuta el servicio, necesitaremos lo que se conoce habitualmente como un *proxy*.

El *proxy* no es más que un módulo de código que, utilizado en el cliente o consumidor, hace aparecer como local un servicio que se ejecuta remotamente, de tal forma que su consumo resulta igual de simple que si utilizásemos un componente cualquiera. Para generar este *proxy* tan sólo tenemos que usar una opción de Visual Studio .NET.

Asumiendo que hemos iniciado un nuevo proyecto, por ejemplo una aplicación Windows desde la que pretendemos consumir el servicio, utilizaremos el menú emergente de la carpeta **Referencias**, en el **Explorador de soluciones**, para añadir una nueva referencia Web, como se ha hecho en la figura 19.23. Esta opción también está disponible para cualquier otro tipo de proyecto, no es exclusiva de las aplicaciones basadas en formularios Windows.

De inmediato nos encontraremos con la ventana mostrada en la figura 19.24. En ella tenemos distintas opciones para localizar el servicio Web que deseamos consumir: usar uno de los registros UDDI de Microsoft, buscar servicios locales o, finalmente, introducir en la parte superior el URL del servicio.

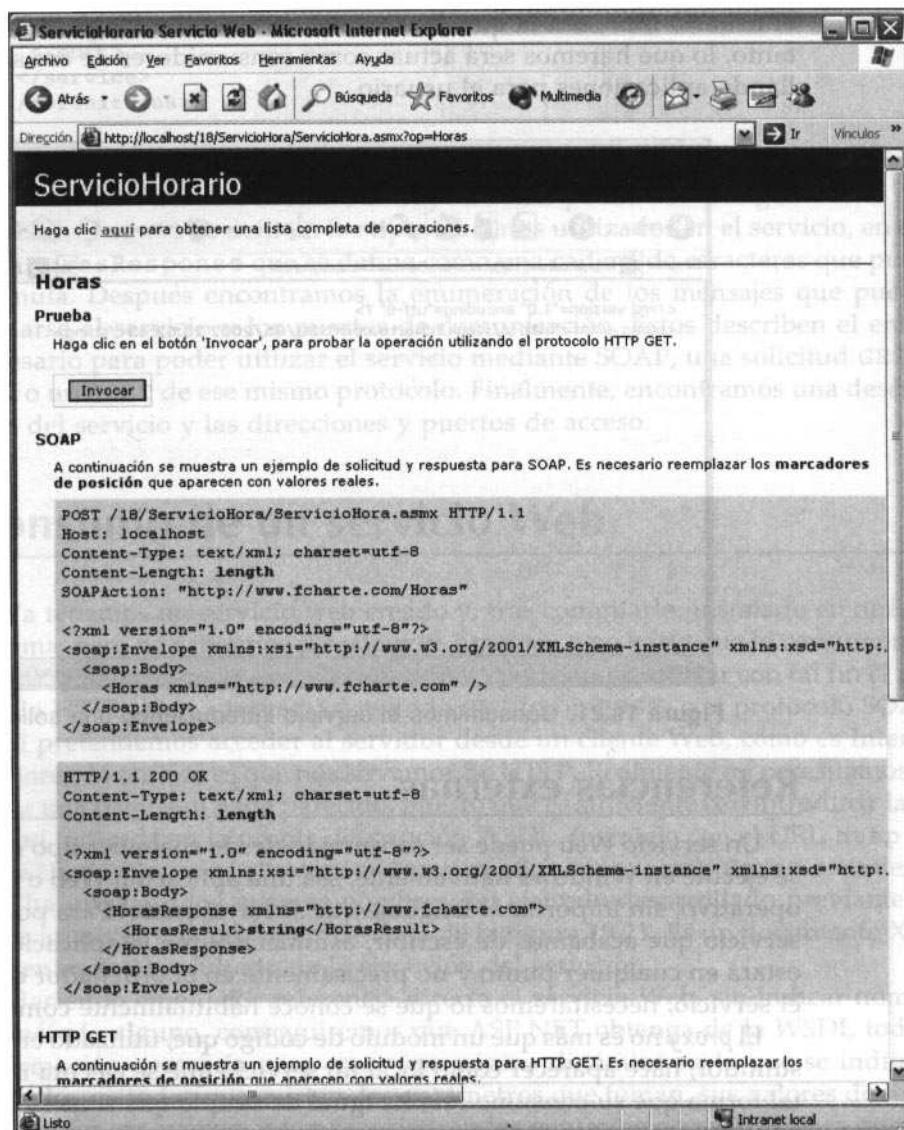


Figura 19.22. Página generada dinámicamente por ASP.NET para comprobar el funcionamiento de un servicio Web

En nuestro caso, suponiendo que estamos desarrollando la aplicación en el mismo ordenador donde se encuentra el servicio, pulsaríamos el último enlace de la ventana o, alternativamente, introduciríamos el URL en la barra Dirección. Veríamos aparecer la página en la que se describe el servicio, desde la cual podemos acceder también al módulo WSDL. Terminaríamos pulsando el botón **Agregar referencia** que hay en la parte inferior (véase figura 19.25) para importar la referencia en nuestro proyecto. En ese momento Visual Studio .NET se encargará de crear el *proxy* para poder consumir ese servicio.

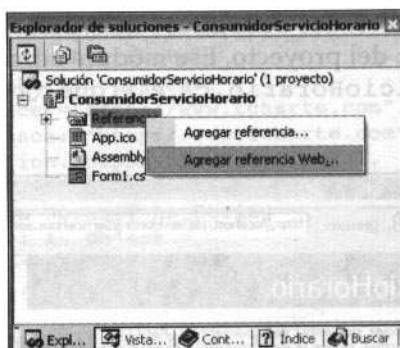


Figura 19.23. Añadimos al proyecto una referencia externa

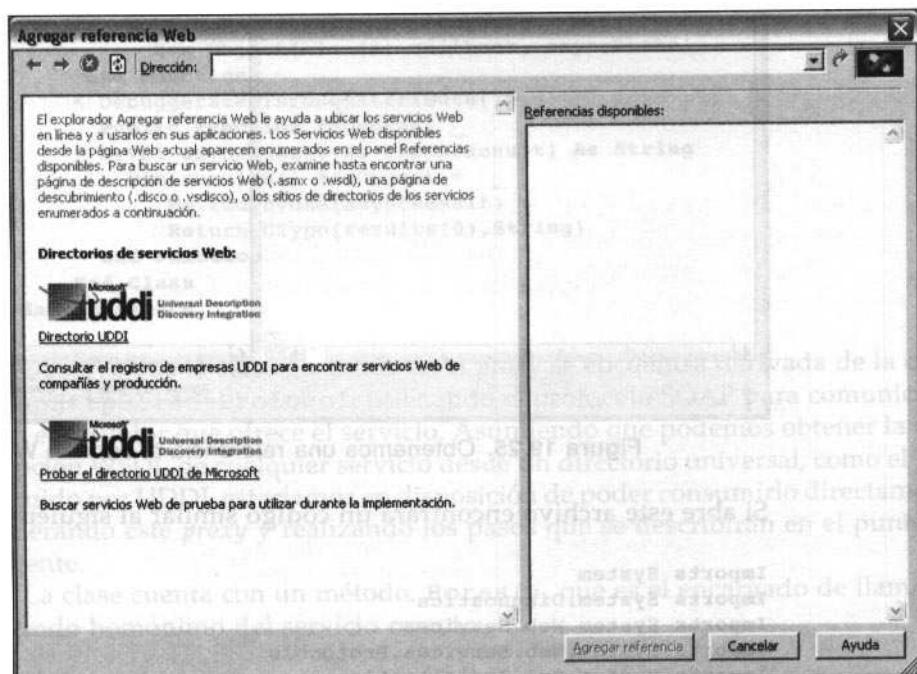


Figura 19.24. Tenemos diferentes opciones para localizar el servicio

Importada la referencia verá cómo se añade al Explorador de soluciones un elemento llamado localhost que contiene varios módulos, entre ellos la descripción WSDL del servicio Web.

## ¿Dónde está el proxy?

Además de los tres módulos que inicialmente vemos en la carpeta localhost, el proceso de importación de la referencia al servicio Web ha generado otro que no aparece en el Explorador de soluciones. Puede verlo accediendo, por ejemplo

## 19. Desarrollo de servicios Web

desde el Explorador de archivos de Windows, a la carpeta **Referencias web\localhost** del proyecto. Ese módulo adicional, denominado en este caso concreto **ServicioHorario.cs**, es el que contiene el código del *proxy* propiamente dicho.

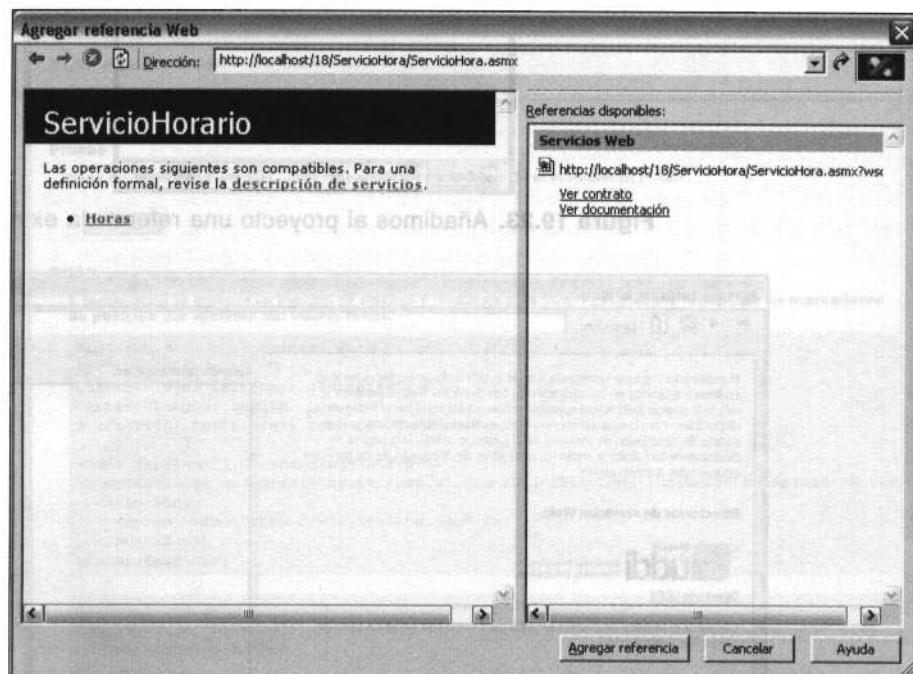


Figura 19.25. Obtenemos una referencia al servicio Web

Si abre este archivo encontrará un código similar al siguiente:

```
Imports System
Imports System.Diagnostics
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Xml.Serialization

Namespace localhost

    < WebServiceBindingAttribute( _
        Name:="ServicioHoraSoap", Namespace]:= _
        "http://www.fcharte.com")> _
    Public Class ServicioHorario
        Inherits SoapHttpClientProtocol

        < DebuggerStepThroughAttribute()> _
        Public Sub New()
            MyBase.New
            Me.Url =
                "http://localhost/ServicioHorario/ServicioHorario.asmx"
        End Sub
    End Class
End Namespace
```

```
< DebuggerStepThrough(), _  
SoapDocumentMethodAttribute( _  
"http://www.fcharte.com/Horas", _  
RequestNamespace:="http://www.fcharte.com", _  
ResponseNamespace:="http://www.fcharte.com", _  
Use:= Description.SoapBindingUse.Literal,  
ParameterStyle:= SoapParameterStyle.Wrapped)> _  
Public Function Horas() As String  
    Dim results() As Object =  
        Me.Invoke("Horas", New Object(-1) {})  
    Return CType(results(0),String)  
End Function  
< DebuggerStepThrough()> _  
Public Function BeginHoras( _  
 ByVal callback As System.AsyncCallback,  
 ByVal asyncState As Object) As IAsyncResult  
    Return Me.BeginInvoke("Horas",  
        New Object(-1) {}, callback, asyncState)  
End Function  
< DebuggerStepThrough()> _  
Public Function EndHoras( _  
 ByVal asyncResult As IAsyncResult) As String  
    Dim results() As Object =  
        Me.EndInvoke(asyncResult)  
    Return CType(results(0),String)  
End Function  
End Class  
End Namespace
```

Observe que la clase generada en el *proxy* se encuentra derivada de la clase `SoapHttpClientProtocol`, utilizando el protocolo SOAP para comunicarse con el servidor que ofrece el servicio. Asumiendo que podemos obtener la descripción WSDL de cualquier servicio desde un directorio universal, como el seguido por UDDI, estaríamos en disposición de poder consumirlo directamente generando este *proxy* y realizando los pasos que se describirán en el punto siguiente.

La clase cuenta con un método, `Horas()`, que es el encargado de llamar al método homónimo del servicio remoto.

## Uso del servicio Web desde el consumidor

Para poder acceder al servicio Web desde nuestro programa tendremos, por lo tanto, que crear un objeto de la clase generada en el *proxy*, en este caso concreto `ServicioHorario`. Dicha clase se encuentra en el ámbito con nombre `Consumidor.localhost`, mientras que el formulario de nuestra aplicación está en el ámbito `Consumidor`. Esto significa que tenemos dos opciones: importar el ámbito `Consumidor.localhost` o bien utilizar una referencia tipo `localhost.ServicioHora`.

Insertamos en el formulario Windows, que actualmente está vacío, un control `Label` y un botón. Hacemos doble clic sobre éste e introducimos el código siguiente:

```
localhost.ServicioHorario MiServicio = new  
localhost.ServicioHorario();  
Label1.Text = MiServicio.Horas();
```

Es todo lo que necesitamos. Cada vez que se pulse el botón se solicitará la hora al servicio Web y se mostrará en la etiqueta de texto. En este caso tenemos el servicio y el consumidor en el mismo ordenador pero, en la práctica, el servicio podría encontrarse en cualquier servidor del mundo. En la figura 19.26 puede ver el programa en funcionamiento.



Figura 19.26. El consumidor en funcionamiento

Podemos hacer el servicio Web mucho más complejo, añadiendo nuevos métodos, y el cliente en vez de una aplicación Windows puede ser una aplicación Web tipo ASP.NET, un componente u otro servicio. El esquema de desarrollo, no obstante, siempre será el descrito en los puntos previos.

## Puntos clave

---

- Un servicio Web es un componente que se ejecuta en un servidor y al que es posible acceder desde cualquier cliente, sin importar el sistema operativo, hardware o lenguaje utilizado para desarrollarlo.
- Partiendo del estándar XML, se han desarrollado diversos protocolos y lenguajes a propósito para facilitar el desarrollo y la comunicación entre servicios Web.
- Con el lenguaje WSDL se describe la estructura de un servicio Web: funciones que pone a disposición del cliente, canales para comunicarse con él.
- UDDI es un servicio de directorio que tiene por objetivo simplificar la publicación y localización de servicios Web.
- Para comunicar a los consumidores con un servicio se utiliza generalmente el protocolo SOAP, definido como un protocolo de llamadas remotas que usa HTTP como transporte.

- En torno a XML han surgido muchos otros estándares del W3C, como XSL, XSD, XSLT y XPath que tienen utilidad no sólo en el campo de los servicios Web, sino también en otros muy distintos como la gestión de bases de datos.
- Visual Studio .NET dispone de editores y diseñadores que reconocen la sintaxis de XML y XSD, haciendo más fácil la creación de esquemas y edición de datos contenidos en documentos XML.
- Gracias al asistente de creación de servicios Web de Visual Studio .NET, el único trabajo manual del que tenemos que hacernos cargo es la definición de las funciones del servicio, marcándolas como accesibles desde el exterior mediante el atributo `[WebMethod]`.
- El punto de entrada a un servicio Web es un módulo `.asmx` que será procesado por el servidor, de igual forma que el punto de entrada de una aplicación Web es un módulo `.aspx`.
- ASP.NET se encarga de generar de manera automática el módulo WSDL de descripción del servicio a partir del código Visual C# .NET que lo compone.
- Simplemente importando una referencia a un servicio web, usando la correspondiente opción del Explorador de soluciones, Visual Studio .NET genera automáticamente el *proxy* necesario para poder utilizarlo.

## Resumen

Como ha podido ver en este capítulo, la Web puede utilizarse para algo más que la publicación de documentos más o menos dinámicos. Gracias a recursos como SOAP, XML, HTTP y WSDL es posible el ofrecimiento de servicios universales, que pueden ser consumidos independientemente de plataformas hardware y software, de sistemas operativos y lenguajes de programación.

Hemos desarrollado un servicio Web simple y visto los pasos que hemos de dar para consumirlo desde una aplicación basada en formularios Windows, generando por el camino un *proxy* que, desde Visual Studio .NET, queda oculto. El motor que hay detrás de los servicios Web, en el caso de la plataforma .NET, es ASP.NET.