CHAPTER 24

SHARING DATA WITH XML

In this chapter

About XML and Web Services 662
FileMaker and XML 663
Transforming XML 667
XML Import: Understanding Web Services 673
Working with Web Services 680
Troubleshooting 685
FileMaker Extra: Write Your Own Web Services 686
ABOUT XML AND WEB SERVICES

XML (which stands for Extensible Markup Language) is a language that describes data in a semantic way. HTML, the standard language of the Web, mixes data and formatting together. XHTML, the successor to HTML, normally contains only data; formatting instructions are in companion stylesheets.

XML provides a way to structure data in a hierarchical manner. It can be displayed using external stylesheets of one sort or another; it also can be read by XML-savvy applications without the need for any intermediate presentation. FileMaker can export and import XML data to and from such other applications.

XML also serves as a key component in web services, a phrase that is used to describe the interaction between a client computer and a web services information provider.

NOTE

In Chapter 21, “Connecting to External SQL Data Sources,” you saw how to integrate external SQL data sources into FileMaker so that it appeared that they were in fact FileMaker databases. You also saw how to use ODBC to accomplish the reverse: to make FileMaker databases appear to ODBC-savvy applications as their own native databases. Chapters 22, “Importing Data into FileMaker Pro,” and 23, “Exporting Data from FileMaker,” demonstrated the other extreme: batch imports and exports of data using standard formats that have been around for decades.

This chapter begins a series of four that show you how to use the Web and contemporary data formats (primarily XML) for importing and exporting data (this chapter), as well as for publishing FileMaker data on the Web and providing interaction to users running web browsers.

As an example, let’s say there’s a computer out there somewhere on the Internet that knows the current temperature at various points all over the world. If you send that computer a latitude and longitude, in the correct format, the remote computer sends back the nearest current temperature it can find. Figure 24.1 shows such a transaction.
You’ll notice a couple of things about this picture. The machine making the request, which we’ve called the Web Services Client, has sent its request in the form of a URL (a *uniform resource locator*, the standard way of making a request for content over the Web). And the responding computer has sent the requested information back in a tagged message format that you might recognize as XML.

The important thing about this transaction is that it doesn’t require any specialized communication protocols to exist between the two machines. The request and response both use standard HTTP, the well-established protocol that powers the entire Web. And the data returned by the server is presented as XML—a standardized and widely accepted way to present data.

This model applies to the XML technology described in this chapter, as well as the two Custom Web Publishing technologies described in Chapters 26, “Custom Web Publishing with XML/XSLT,” and 27, “Custom Web Publishing with PHP.” Those technologies are XML/XSLT (XSL Transformations, where XSL stands for Extensible Stylesheet Language), which builds on the basic XML described in this chapter, and PHP (PHP: Hypertext Preprocessor). This chapter provides an introduction to XML.

**FileMaker and XML**

FileMaker can both *import* and *export* data as XML. XML also serves as a key component of Custom Web Publishing. Thus, it plays a role both in the importing and exporting of data as well as in the dynamic real-time updating of FileMaker data over the Web. Before we delve deeper, a brief overview of XML might be useful.

**The Basics of XML**

XML is a text-based means of representing data, which is at the same time *rich* and *portable*. By *rich*, we mean that the data is more than mere text: An XML document is capable of describing its own structure so that in looking at an XML document you can tell a chapter heading from a bullet point, or a personnel ID from a health-insurance deductible. (Note that structure is not the same as format; the bullet point or personnel ID structural element can be displayed in any format.) By *portable*, we mean that XML documents are stored as plain text and can be read by a wide variety of programs on a wide variety of computers and operating systems.

As an example, consider the XML document that appears in Listing 24.1. This is a short document containing information about motors. You’ll notice the document is full of tags (called *markup*) that might look superficially familiar to you if you’ve seen some HTML before. You’ll notice that the tags always occur in pairs, with some content between them, and you’ll notice that the tags seem to describe the data they contain. These tag pairs are known in XML jargon as *elements*. 
Listing 24.1  A Small XML File Containing Motor Data

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<motors>
  <motor>
    <model>Rotary 17</model>
    <weight>1200</weight>
    <part_number>M3110A-3</part_number>
    <volume>312</volume>
  </motor>
</motors>
```

This XML document is rich in the sense that it contains two kinds of information: It contains raw data, but it also contains tags telling a reader what the data means. In this document, M3110A-3 is not just a string of numbers and letters; it’s specifically a part number.

The document is also portable in the sense that it’s stored as plain text, meaning you don’t need a special “motor processing” application to read it. Any tool or program that can read plain text can work with this data.

XML documents have to follow some simple rules. Each must begin with an XML declaration, like the first line of Listing 24.1. Each must have a single outermost, or document, element—in Listing 24.1, the document element is called motors. Each tag must be properly closed—if you have a <model> tag, you’d better have its closing counterpart, called </model>. And, although tags may be nested (for example, in Listing 24.1, the weight element is completely enclosed within the motor element), it is not permissible for tags to overlap. Therefore, something like

```xml
  <model>Rotary<weight>500</model></weight>
```

would not be allowed because the weight tag, rather than being completely enclosed in the model tag, instead overlaps it.

XML documents that follow these few simple rules, as well as some rules about allowable characters, are said to be well formed.

**NOTE**

XML is a rigorous standard, with plenty of technical documents that describe it in exact detail. In this book we opt for clarity over rigor, so we encourage you to get hold of additional resources to explore the full details of XML concepts such as well-formedness. The description we’ve given is fairly complete, but the last word can be found at http://www.w3.org/TR/2004/REC-xml-20040204/#sec-well-formed.

**FileMaker’s XML Grammars**

XML syntax rules, as you might have noticed, don’t say anything about how to mark up your data. XML doesn’t force you to use a motor element when talking about motors, nor would it specify what other elements a motor element should contain. If you’re designing an XML document, the exact structure of the document, as far as what data it contains and how that data is marked up, remains up to you, the document designer.
FileMaker is capable of presenting its data as XML, and when it does so, it uses its own, FileMaker-specific set of elements to describe its data. FileMaker can actually present its data in either of two XML structures, called grammars; you, as the user or developer, get to choose which one suits your current situation best.

NOTE

FileMaker can actually present its data in as many as four XML grammars, but two of these are only meaningful in the context of Custom Web Publishing, which is the subject of Chapters 26 and 27. Furthermore, of the two grammars used for basic data export, FMPDSSRESUTE is now deprecated, and you are advised not to use it.

Suppose that you have some product data in a FileMaker table, which looks like Figure 24.2.

![Figure 24.2](Image)

Some sample widget data in a FileMaker table.

To export these records as XML, choose File, Export Records, and then choose a file type of XML in the following dialog. When you do this, before seeing the familiar Export dialog, you see an XML options dialog, as shown in Figure 24.3.

![Figure 24.3](Image)

FileMaker’s XML/XSL export options dialog.

Here you can choose which of FileMaker’s XML grammars to apply. You can also apply an XSL stylesheet to the output, which is an important topic we’ll deal with in its own section later in this chapter. If you were to export the widget data as XML with FMPXMLRESULT as the grammar, you’d see something like the document in Listing 24.2.
<?xml version="1.0" encoding="UTF-8" ?>
<FMPXMLRESULT xmlns="http://www.filemaker.com/fmpxmlresult">
  <ERRORCODE>0</ERRORCODE>
  <PRODUCT BUILD="11-08-2005" NAME="FileMaker Pro" VERSION="8.0v2"/>
  <DATABASE DATEFORMAT="M/d/yyyy" LAYOUT="" NAME="Widget.fp7" RECORDS="3" TIMEFORMAT="h:mm:ss a"/>
  <METADATA>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="WidgetID" TYPE="NUMBER"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Description" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Color" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Weight" TYPE="NUMBER"/>
  </METADATA>
  <RESULTSET FOUND="3">
    <ROW MODID="1" RECORDID="1">
      <COL>
        <DATA>W1</DATA>
      </COL>
      <COL>
        <DATA>Medium Frobisher</DATA>
      </COL>
      <COL>
        <DATA>Mauve</DATA>
      </COL>
      <COL>
        <DATA>12.2</DATA>
      </COL>
    </ROW>
    <ROW MODID="1" RECORDID="2">
      <COL>
        <DATA>W2</DATA>
      </COL>
      <COL>
        <DATA>Gosset Socketeer</DATA>
      </COL>
      <COL>
        <DATA>Red</DATA>
      </COL>
      <COL>
        <DATA>4</DATA>
      </COL>
    </ROW>
    <ROW MODID="0" RECORDID="3">
      <COL>
        <DATA>W3</DATA>
      </COL>
      <COL>
        <DATA>Triple Hex Ping Nut</DATA>
      </COL>
      <COL>
        <DATA>Steel</DATA>
      </COL>
      <COL>
        <DATA>2.3</DATA>
      </COL>
    </ROW>
  </RESULTSET>
</FMPXMLRESULT>
FMPXMLRESULT wraps all the database data in generic-looking `<COL>` and `<DATA>` elements. There’s also a `<METADATA>` element, which you can see contains subelements for each field that give a lot of information about the field, such as its data type, whether it’s a repeating field, and whether it’s allowed to be empty. The data within the `<ROW>` elements then matches up to the field descriptions in the `<METADATA>` section based on position: The value `Mauve` in the first row is the third data element, and when you consult the `<METADATA>` section, you can see that this means it corresponds to the Color field.

**Transforming XML**

By itself, the capability to turn FileMaker data into XML is not terribly useful. The reason is that FileMaker emits the XML in one of its specialized grammars. Even though other applications can read the file containing the exported data, they might not be able to make much sense of the FMPXMLRESULT grammar. In fact, this is a general issue with XML-aware applications: They all work with different formats and structures of XML. But, because of the strict XML structure, it is easily parsed by computer programs, and the conversion from one grammar to another is often not difficult. Let’s say there exists a tool (call it WidgetPro) that can read information about widgets from an XML file, as long as the XML file looks like what’s shown in Listing 24.3.

**Listing 24.3  THE WidgetPro XML FORMAT**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<widgetset>
  <widgetrecord index="1">
    <id>789</id>
    <widget_description>Large Flanger</widget_description>
    <widget_color>blue</widget_color>
    <widget_weight>45</widget_weight>
  </widgetrecord>
  <widgetrecord index="2">
    <id>790</id>
    <widget_description>Granite Auger</widget_description>
    <widget_color>Slate</widget_color>
    <widget_weight>715</widget_weight>
  </widgetrecord>
</widgetset>
```

We can get widget data from FileMaker, and we can get it as XML, but the two XML documents have different structures—the same data but expressed with different tag names and tag structures.

This is an important point to understand about XML: XML is not in itself a language or a file format. Using XML, the same data can be described (marked up, as it’s often said) in many different ways. For applications to share data via XML, it’s not enough for each application to support reading and writing data in its own, specific XML format. There has to be some means to translate between different forms of XML as well. In the widgets example,
this means that we need to take the “FileMaker widget XML” and make it look like “WidgetPro widget XML.” This leads to the concept of XML transformations.

Figure 24.4 illustrates the idea of an XML transformation. From the FileMaker Pro database of widget information, we first have to export the widget data in an FMPXML structure. Next, we need to transform that XML so that it looks like WidgetPro’s XML structure instead. Finally, we bring the transformed XML into WidgetPro. Figure 24.4 sketches what this process would look like.

**Figure 24.4**
An XML transformation pipeline.

FMPXMLRESULT (emitted by FileMaker) is transformed into WidgetPro XML (accepted by WidgetPro).

**Introducing XSL Stylesheets**

It turns out that XML already has a transformation technology available for us. That technology is XSL. The stylesheet turns out to be the transformer. In much the same way that a word processing or page layout stylesheet can take ordinary text and transform it into formatted text, an XSL stylesheet can take one form of XML and transform it into another (or into any other text-based format, actually), as shown in Figure 24.5.

**Figure 24.5**
An XML transformation pipeline that uses an XSL stylesheet to accomplish the transformation.

**Note**

You might often see the terms XSL and XSLT used interchangeably. Technically they’re distinct; XSLT is in fact a subset of XSL. But when people speak of XSL they’re generally referring to XSL transformations, so we won’t make a major point of distinguishing between the two terms.

So, what is an XSL stylesheet? It’s a series of commands that describe how to transform XML input into some new form. The new form can also be XML (and often is), but it’s possible to use a stylesheet to transform your XML into other text-based formats as well: tab-separated text, HTML, or more complex formats such as PDF and RTF. Interestingly,
the XSL transformation language is itself a variety of XML, so XSL stylesheets are also valid XML documents in their own right. Here’s an example of an XSL stylesheet that would transform the “FileMaker widget XML” into “WidgetPro XML” (see Listing 24.4).

**Listing 24.4  An XSL Stylesheet**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
➥ xmlns:fmp="http://www.filemaker.com/fmpxmlresult"
➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
➥ exclude-result-prefixes="fmp">
 <xsl:output indent="yes" method="xml"/>
 <xsl:template match="fmp:FMPXMLRESULT">
  <widgetset>
   <xsl:for-each select="fmp:RESULTSET/fmp:ROW">
    <widgetrecord index="{position()}">
     <xsl:for-each select="fmp:COL">
      <xsl:choose>
       <xsl:when test="position()='1'">
        <id>
         <xsl:value-of select="fmp:DATA"/>
        </id>
       </xsl:when>
       <xsl:when test="position()='2'">
        <widget_description>
         <xsl:value-of select="fmp:DATA"/>
        </widget_description>
       </xsl:when>
       <xsl:when test="position()='3'">
        <widget_color>
         <xsl:value-of select="fmp:DATA"/>
        </widget_color>
       </xsl:when>
       <xsl:when test="position()='4'">
        <widget_weight>
         <xsl:value-of select="fmp:DATA"/>
        </widget_weight>
       </xsl:when>
      </xsl:choose>
     </xsl:for-each>
    </widgetrecord>
   </xsl:for-each>
  </widgetset>
 </xsl:template>
</xsl:stylesheet>
```

Our goal, remember, is to take the original XML output from FileMaker (Listing 24.2) and translate that output into a new form of XML that contains much the same information, but in a different structure (Listing 24.3). The stylesheet in Listing 24.4 contains two kinds of statements: XSL commands (which you can tell by their `xsl:` prefix) and the actual XML tags that the stylesheet will output. The stylesheet’s job is to pick through the original XML document and decide which pieces of it to output, and in what order.
Chapter 24  SHARING DATA WITH XML

ANALYZING A STYLESHEET

If you’ve never read through an XSL stylesheet before, this section might be useful. We’ll go through the stylesheet in Listing 24.4 line by line to illustrate its inner workings.

THE XML DECLARATION

```
<?xml version="1.0" encoding="UTF-8" ?>
```

Every XML document begins with an XML declaration—and XSL stylesheets are XML documents. Simple enough.

THE STYLESHEET STATEMENT

```
<xsl:stylesheet version="1.0"
➥ xmlns:fmp="http://www.filemaker.com/fmpxmlresult"
➥ xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

The `xsl:stylesheet` statement announces the document as an XSL stylesheet. The stylesheet statement also declares two XML *namespaces* (that’s what the `xmlns` stands for). Namespaces are an important XML concept, but like most of the finer points of XML, namespaces are a bit too complex a topic for us to spend much time on in this book. Suffice it to say that both the namespaces declared here are necessary. The second namespace, abbreviated `xsl`, is common to all XSL stylesheets, and distinguishes all the XSL stylesheet commands from other forms of XML. These commands, again, begin with the same `xsl:` prefix specified by the namespace. And the `fmp` namespace declaration is crucial as well because it matches the namespace declaration that appears at the start of any XML document output by FileMaker. We’ll have a bit more to say about the FileMaker namespace farther on.

**NOTE**

Notice also that the stylesheet declaration includes an `exclude-result-namespaces` statement. This rather important command prevents namespaces declared in the source document from being carried through to the output document. In general, we recommend you use this command in the `<xsl:stylesheet>` element of your stylesheets to strip all FileMaker-specific namespaces from your output. The `exclude-result-namespaces` attribute uses a space-delimited list of namespace prefixes to decide what to strip out. In Listing 24.5, just one namespace is being stripped, so it says `exclude-result-prefixes="fmp"`. If there were multiple namespaces to strip (as there often are in FileMaker’s Custom Web Publishing), you would say something like `exclude-result-prefixes="fmp fml fmr fmrs"`. This would exclude all four namespaces from the stylesheet’s output.

FileMaker’s Custom Web Publishing with XML/XSLT is covered in depth in Chapter 26, p. 715.
**Specifying the Output Type**

```xml
<xsloputput indent="yes" method="xml"/>
```

The `xsloputput` statement tells the XSL processor what type of document is being output. If you’re trying to produce XML output, you have to include a statement like this one so that the XSL processor adds the appropriate XML declaration to the final document. The output statement also includes an attribute called `indent`—when this is set to `yes`, the XSL processor tries to format the XML output in a pleasing and readable way.

**Using a Template to Find the Result Set**

```xml
<xs1:template match="fmp:fmpxmlresult">
  <widgetset>
    [...] code omitted ...]
  </widgetset>
</xs1:template>
```

The concept of a template is crucial to XSL. Templates are a way for the stylesheet writer to specify which parts of the source document she’s interested in. In this case, we’re telling the processor we want to find the element called `<FMPXMLRESULT>` in the source document and do something with it. You’ll notice that this template takes up all the rest of the stylesheet—so the rest of the stylesheet tells what to do with the `<FMPXMLRESULT>` after we’ve found it.

Just inside the `xs1:template` statement is some actual XML in the form of a `<widgetset>` tag. This tag is matched by the `</widgetset>` tag at the very end of the template instruction, almost at the end of the document. These two tags, unlike the `xs1: commands, aren’t instructions at all—they represent XML that will be output. So, the `xs1:template` here is saying “When you find a `<RESULTSET>` tag, output a `<widgetset>` ... `<widgetset>` tag pair, and then go on to do some other things inside it.” Of course, inside the `<widgetset>` tag, we want the stylesheet to emit XML that describes the individual widget records, which is what the next part of the stylesheet does.

**Using `xs1:for-each` to Loop Over a Result Set**

```xml
<xsl:for-each select="fmp:RESULTSET/fmp:ROW">
  <widgetrecord index="{position()}">
```

The `<xs1:for-each>` tag is a looping construct. Right now, we’re inside the XSL template that matches an `<FMPXMLRESULT>` tag, so the command tells the XSL processor to find all `<ROW>` elements that are children of `<RESULTSET>` elements inside the `<FMPXMLRESULT>`. The additional commands inside the `<xs1:for-each>`...`<xs1:for-each>` tag pair furnishes instructions on what to do with each `<ROW>` element that we find.

For each `<ROW>` in the original FileMaker XML, we want to output a `<widgetrecord>` element, and that’s what the next line does. Additionally, the `<widgetrecord>` element needs to have an attribute called `index`, which shows the numerical position of the widget, in sequence. The `position()` function used in that line gives the position of the current element. We wrap the function call in curly braces so that the XSL process knows it’s a command, and not literal text to be output with the XML.
**Using xsl:choose to Determine Output**

```xml
<xsl:for-each select="fmp:COL">
    <xsl:choose>
        <xsl:when test="position()='1'">
            <id>
                <xsl:value-of select="fmp:DATA"/>
            </id>
        </xsl:when>
        [other columns omitted for brevity]
    </xsl:choose>
</xsl:for-each>
```

At this point, we’re inside the `<ROW>` element in the original FileMaker XML, and from an output standpoint, we’re inside the `<widgetrecord>` element in the output XML. (Read that sentence a few times if it doesn’t sink in right away!) Given what we know the output is supposed to look like, all that’s left is to find the four data elements from this `<ROW>` in the FileMaker XML, and output each one wrapped in a tag that correctly names its data field.

This is a little trickier because in the FileMaker XML, a `<ROW>` contains only `<COL>` elements, with no mention of the actual field name in question. You might recall that in the `FMPXMLRESULT` output grammar, field names appear near the top of the document in the `<METADATA>` section. We have to loop through all the `<COL>` elements inside the row, and for each one, we decide how to output it based on its position in the group.

Again we use `<xsl:for-each>` to loop over a set of elements—in this case, all the `<COL>` elements inside the current `<ROW>`. For each `<COL>` element we process, we need to make a choice as to how to output it. If it’s in the first position, we output it as an `<id>` element; if it’s in the second position, we output it as a `<widget_description>` element, and so forth.

If we were trying to program this type of multiple choice in a FileMaker calculation, we’d use a `Case()` statement or perhaps a `Choose()`. Here we use the XSL equivalent, which is `<xsl:choose>`. Like FileMaker’s `Case` statement, `xsl:choose` lets you choose from several options, each one corresponding to a logical test of some kind. The `<xsl:choose>` element contains one or more `<xsl:when>` statements. Each one corresponds to a particular choice, and each choice is associated with a logical test. In the code we showed in the preceding section, the first test inside the `<xsl:choose>` element is the test for columns whose position equals 1. In this case, we go on to output the `<id>`...`</id>` tag pair with the data value inside it. To do this, we use the `<xsl:value-of>` element, which can pull out a piece of the source XML document to output. In this case, each `<COL>` element in the FileMaker source XML has a `<DATA>` element inside it, and it’s that element we want to grab and add to the output.

The rest of the `<xsl:when>` statement contains the remaining tests, for the columns in positions 2, 3, and 4 of the output. At this point, we’re done! The rest of the code consists of emitting closing XML tags that match the opening tags we’ve already sent, and of closing our XSL constructs, like `<xsl:for-each>`.
Applying an Export Transformation to FileMaker XML

FileMaker lets you use XSL stylesheets to transform your data when moving data into or out of FileMaker with XML. Let’s consider the export example first. If you have a table of FileMaker data, such as the widget data we’ve been using, and you choose to export the data as XML, you’ll see the dialog box shown in Figure 24.6.

Figure 24.6
When exporting XML from FileMaker, you can also choose to apply a stylesheet to transform the outbound XML.

Here you can choose your XML export grammar, as we’ve already seen, but you can also choose whether to apply an XSL stylesheet to transform the XML as it’s being output. If you want to apply a stylesheet, you can pick a local file; in other words, a file resident on your local hard drive or on a mounted server volume. You can also pick a stylesheet file that’s available over HTTP; namely, on a web server somewhere.

Working with Remote Stylesheets

The HTTP feature was a very smart choice on the part of FileMaker’s development team. It’s very important in a multiuser FileMaker deployment, where many users use the same solution files hosted by a single server. If the only option for stylesheet work was to use a file from the user’s locally accessible hard drive or drives, you’d either have to distribute the stylesheet to all system users and make sure that they put it in the right location on their hard drives or have a common server volume that all users would have to mount. With the capability to pull a stylesheet from a web server, you can create one single stylesheet and place it on a central web server, and let it be accessed by all users.

One important qualification to this capability is that FileMaker currently supports remote access to stylesheets via only the HTTP protocol. HTTPS (secure HTTP) is not currently supported.

Using XSL stylesheets in the export process in this way, you can transform FileMaker data into a wide variety of output formats: other variants of XML, or HTML, or XML suitable for import into applications such as Excel or Quark XPress, or even a complex text format such as PDF.

XML Import: Understanding Web Services

In addition to its to export data via XML, either with or without an XSL stylesheet, FileMaker has the capability to work with remote XML data sources, often referred to under the umbrella term web services. This capability was added in FileMaker version 6 and
is a significant addition to its XML strengths. Using this capability, you can bring data from a variety of remote data sources directly into FileMaker, as we’ll discuss in the sections that follow.

**FileMaker’s XML Import Capability**

The concept of XML exporting ought to seem straightforward: Take some FileMaker data, pick an XML grammar for the export, and optionally apply an XSL stylesheet to transform the XML data as it heads out. But what about the concept of importing XML? What does this mean, and what is it good for?

Stated simply, FileMaker can import any XML data that conforms to the FMPXML grammar. FileMaker reads the `<METADATA>` section of the document to determine the field structure, and reads the individual row and column data to figure out the actual data values that should be imported.

To demonstrate this for yourself, find some suitable FileMaker data and export it as XML, using the FMPXMLRESULT grammar, without applying any XSL stylesheet to it. Starting from the same file and table, go back and reimport the file you just exported, treating it as an XML data source—you’ll see that FileMaker correctly reads the field structure and data from the XML document. Or, to test it in a different way, drag the new XML file onto the FileMaker application icon to open it. FileMaker should, without intervention from you (except for choosing a filename for the new file), open the XML file, read its structure, and create a new FileMaker file with a new table containing the correct fields, field types, and data values.

**NOTE**

One of the things developers have often wanted from FileMaker is a way to save a file’s field structure as a text document, and then use that text document to move the field structure somewhere else and re-create it. The capability to open an XML document and have it create a new FileMaker file might seem to make that possible, but there are caveats. The XML export doesn’t preserve important information about your field structure, such as the definitions of calculation fields and summary fields. In the XML output, these fields are treated simply as their underlying data types, so a calculation that produces a number is treated in the XML metadata as a simple number field, without preserving the calculation’s definition.

So, FileMaker can import any XML data file that conforms to the FMPXMLRESULT grammar. Additionally, as you might have seen, this XML data stream can come from a local file, or it can come from a file available over HTTP—in other words, a file from somewhere on the Web. This is where things get interesting, so let’s delve further into the concept of a web service.
If you try to import data that doesn’t conform to the FMPXMLRESULT grammar, FileMaker gives you an error. For more information, see “Wrong XML Format” in the “Troubleshooting” section at the end of this chapter.

NOTE

As was the case with using remote stylesheets for XML export, FileMaker is also unable to work with data from an HTTPS data source when importing XML. If the data source from which you want to import is available only over secure HTTP, FileMaker isn’t able to import it.

WEB SERVICES REVIEWED

We started this chapter by saying that Web services was a term referring to the sharing of data between computers via the Web’s HTTP protocol and that the data was often exchanged in XML format. Imagine you have two computer systems that need to exchange data. One is a large student information system that resides on a mainframe computer. The other is a system that generates complex forms for each student, to conform to governmental guidelines. Periodically, the forms application has to consult the mainframe application to see whether any new students have been added so that those students are accounted for in the forms system.

There are many ways to make this kind of sharing happen. The mainframe programmer could export a file of new students every night, in some plain-text format, and the forms programmers could write routines to grab the file and process it in some way. Or the mainframe could be made accessible via a technology such as ODBC, and the forms application could be configured to make ODBC requests to the mainframe.

➔ For more information on ODBC, see Chapter 21, p. 603.

Another option, though, is to make it possible to send queries to the mainframe via HTTP, and get XML back in response. This is simpler than either of the previous scenarios: It doesn’t require any complicated processes such as writing and then fetching an actual file, nor does it involve the client-side complexities of ODBC transactions. It uses the widely (almost universally) available HTTP protocol, and requires only that one side be able to generate a form of XML and the other side be able to read it. Don’t get us wrong—web services transactions can still be plenty complicated, but standards such as XML and HTTP make them less complex than they might be otherwise. Refer to Figure 24.1 for a sketch of a possible web services transaction.

Let’s say that, in our example, the forms application was written in FileMaker. And let’s assume the mainframe student information system was accessible as a web service, meaning that you could send a request via HTTP and get back a listing of new students in some XML format (that would likely not conform to the FMPXMLRESULT grammar). To import that data into FileMaker, you could perform an XML import, use the URL of the student information system as the data source, and apply an XSL stylesheet that would transform the
new student XML into `FMPXMLRESULT`. The concept is sketched out in Figure 24.7. To retrieve student data from this mainframe, make a request to a URL that’s able to produce an XML representation of the student, and then bring that XML back through a stylesheet into FileMaker.

![Figure 24.7](http://students.mydistrict.org/fetchStudent?id=0908123)

This is a graphical representation of the process of retrieving data via XML.

**A Stylesheet for XML Import**

For the sake of argument, assume that we have an XML *data stream* representing new students. (We use the term *data stream* rather than *file* as a reminder that the data need not come from a file, but can also come from a networked data source over HTTP.) Listing 24.5 shows what that data might look like.

---

**Listing 24.5 Sample XML File Containing Data**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<newStudentSet count="4" date="11/1/2003">
  <student id="414">
    <nameFirst>Jonathan</nameFirst>
    <nameLast>Middlesex</nameLast>
    <nameMiddle>A</nameMiddle>
    <address>123 Oak Way</address>
    <city>Bensenville</city>
    <state>AK</state>
    <zip>09080-1001</zip>
    <county>Hightower</county>
    <district>Sparta</district>
    <school>Bensenville Junior High</school>
    <grade>4</grade>
    <parents>
      <parent>
        <nameFirst>Sharon</nameFirst>
        <nameLast>Middlesex</nameLast>
        <relationship>Parent</relationship>
      </parent>
      <parent>
        <nameFirst>Martin</nameFirst>
        <nameLast>Middlesex</nameLast>
        <relationship>Parent</relationship>
      </parent>
    </parents>
  </student>
</newStudentSet>
```
It's a simple enough structure, consisting of a `<newStudentSet>` filled with one or more `<student>` elements, where each `<student>` has a number of fields associated with it. The only wrinkle has to do with parent information: Clearly a student can have more than one parent, so each student contains a `<parents>` element with one or more `<parent>` elements inside it. We'll have to think about what to do with that.

That's the XML file that the hypothetical data source can put out. But remember, for FileMaker to import this XML data, it has to be structured in the `FMPXMLRESULT` format. Such a structure would look like Listing 24.6.

**Listing 24.6 Data in the Importable FMPXMLRESULT Format**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<FMPXMLRESULT xmlns="http://www.filemaker.com/fmpxmlresult">
  <ERRORCODE>0</ERRORCODE>
  <PRODUCT BUILD="8-11-2005" NAME="FileMaker Pro" VERSION="8.0v2"/>
  <DATABASE DATEFORMAT="M/d/yyyy" LAYOUT="" NAME="Student.fp7" RECORDS="1"
      TIMEFORMAT="h:mm:ss a"/>
  <METADATA>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="NameFirst" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="NameLast" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="NameMiddle" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Address" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="City" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="State" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Zip" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="County" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="District" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="School" TYPE="TEXT"/>
    <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Parents" TYPE="TEXT"/>
  </METADATA>
  <RESULTSET FOUND="1">
    <ROW MODID="0" RECORDID="1">
      <COL><DATA>Jonathan</DATA></COL>
      <COL><DATA>Middlesex</DATA></COL>
      <COL><DATA>A</DATA></COL>
      <COL><DATA>123 Oak Way</DATA></COL>
      <COL><DATA>Bensenville</DATA></COL>
      <COL><DATA>AK</DATA></COL>
      <COL><DATA>09080-1001</DATA></COL>
      <COL><DATA>Hightower</DATA></COL>
      <COL><DATA>Sparta</DATA></COL>
      <COL><DATA>Bensenville Junior High</DATA></COL>
      <COL><DATA>Sharon Middlesex (Parent),
           Martin Middlesex (Parent)</DATA></COL>
    </ROW>
  </RESULTSET>
</FMPXMLRESULT>
```

Web services scattered through the ether are unlikely to emit XML that conforms to the `FMPXMLRESULT` grammar. So, before bringing that data into FileMaker, we need to transform it into `FMPXMLRESULT`. And the tool for doing that is, of course, an XSL stylesheet. This is
exactly the reason FileMaker lets you apply a stylesheet to inbound XML (in other words, on import). Odds are that the XML data source does not produce the FMPXMLRESULT grammar directly, so it's our job to translate the source XML into the form that FileMaker can read.

We need an XSL stylesheet to make that transformation. The stylesheet needs to make sure to output all the initial information found in an FMPXMLRESULT file, such as database name, and all the metadata describing the field structure. Then, in the context of a <RESULTSET>, we need to output the actual student data. Listing 24.7 shows what the stylesheet for transforming student data prior to importing it into FileMaker should look like.

**Listing 24.7 An XSL Stylesheet**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output indent="yes" method="xml"/>
  <xsl:template match="newStudentSet">
    <FMPXMLRESULT xmlns="http://www.filemaker.com/fmpxmlresult">
      <ERRORCODE>0</ERRORCODE>
      <PRODUCT BUILD="8-11-2005"  NAME="FileMaker Pro" VERSION="8.0v2"/>
      <DATABASE DATEFORMAT="M/d/yyyy" LAYOUT="" NAME="Student.fp7" RECORDS="{@count}" TIMEFORMAT="h:mm:ss a"/>
      <METADATA>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="NameFirst" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="NameLast" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="NameMiddle" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Address" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="City" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="State" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Zip" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="County" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="District" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="School" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Grade" TYPE="TEXT"/>
        <FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Parents" TYPE="TEXT"/>
      </METADATA>
      <RESULTSET FOUND="{@count}"/>
      <xsl:for-each select="student">
        <ROW MODID="0" RECORDID="{position()}">
          <COL><DATA><xsl:value-of select="nameFirst"/></DATA></COL>
          <COL><DATA><xsl:value-of select="nameLast"/></DATA></COL>
          <COL><DATA><xsl:value-of select="nameMiddle"/></DATA></COL>
          <COL><DATA><xsl:value-of select="address"/></DATA></COL>
          <COL><DATA><xsl:value-of select="city"/></DATA></COL>
          <COL><DATA><xsl:value-of select="state"/></DATA></COL>
          <COL><DATA><xsl:value-of select="zip"/></DATA></COL>
          <COL><DATA><xsl:value-of select="county"/></DATA></COL>
          <COL><DATA><xsl:value-of select="district"/></DATA></COL>
          <COL><DATA><xsl:value-of select="school"/></DATA></COL>
          <COL><DATA><xsl:value-of select="grade"/></DATA></COL>
        </ROW>
      </xsl:for-each>
    </FMPXMLRESULT>
  </xsl:template>
</xsl:stylesheet>
```
We won’t spend as much time dissecting this stylesheet as we did with the last one. The mechanics should be easy to discern. After the usual declarations, we declare a template that matches to the source document’s `<newStudentSet>` element (it’s the root element, so there will be only one). That’s the occasion to output all the header-type information particular to the `FMPXMLRESULT` grammar, including the field structure metadata. We then go on to output a `<RESULTSET>`...</RESULTSET> tag pair, and do some more work inside that.

Within the `<RESULTSET>` tags, we use an `<xsl:for-each>` to loop over all the `<student>` elements inside the `<newStudentSet>`. For each one, we output the corresponding `<ROW>` element. Each `<ROW>`, in turn, is a collection of `<COL><DATA>...</COL></DATA>` tag pairs. We output one of these for each inbound field, and insert the correct data into it, using `<xsl:value-of>`.

The only thing at all noteworthy is the treatment of the parent information. The inbound student information is not completely “flat.” The nested `<parents>` element almost implies a new table, in relational database terms. We could choose to handle it that way, and bring the parent information into a separate table, but we chose instead to flatten the parent data into a single field. This was more to illustrate a particular technique than because it’s actually a good idea to do that. Whether it really is a good idea depends on the application.

In any case, the technique here is to loop over the individual `<parent>` elements by using `<xsl:for-each>`. For each parent, we output the first name, last name, and the family relationship in parentheses. You might notice that we use the `<xsl:text>` command liberally, to output the spaces between words and the parentheses around the relationship. The reason for this is that XML treats certain characters, such as whitespace, specially. Whitespace, in particular, XML ignores. Wrapping it in an `<xsl:text>` tag ensures that the processor treats it as real whitespace and outputs it as such.

The last wrinkle here is that we want the parent list to be comma separated. So, we write a little piece of logic that requests that a comma and its following whitespace be output, but only if the current `<parent>` element is not the last one in the group. The check is performed with `<xsl:if>`.

As you can see, the stylesheet isn’t too complicated. The hardest part is getting all the `FMPXMLRESULT`-specific elements and attributes correctly included.
Chapter 24 Sharing Data with XML

After you’ve written the stylesheet, you would apply it in the course of the import. If everything goes smoothly, the stylesheet is successfully applied, it emits pure FMPXML, and this is cleanly imported into FileMaker.

If your stylesheet contains a programming error, FileMaker presents an error dialog and tries to alert you as to where in the stylesheet the problem occurred. For more information, see “Errors in Stylesheets” in the “Troubleshooting” section at the end of this chapter.

You might need to do a bit of work to make sure that the fields line up correctly on import. The easiest way to ensure this is to write your XSL stylesheet in such a way that the field names in the resulting <METADATA> section of the XML are exact matches for your FileMaker field names. If that’s the case, you need to specify only that fields should import based on matching names. If for any reason there’s a discrepancy between the field names used in the resultant FMPXMLRESULT and the field names in the target table, you have to specify the import matching by hand.

➔ For more details about specifying import field mappings, see “The Import Field Mapping Dialog,” p. 629.

Of course, the import might not go smoothly. See “Correct Stylesheet, Failed Import” in the “Troubleshooting” section at the end of this chapter for some tips on how to handle stylesheets that don’t perform as expected.

Working with Web Services

The previous section, on importing XML via a stylesheet, tells you more or less all you need to know to work effectively with web services in FileMaker. The only other thing you need is a real web service to work with.

It can actually be a bit difficult to find interesting web services to play with. Many of the really meaty web services, because they’re providing useful information, charge an access or subscription fee of some kind. These might include web services that provide current weather information from satellites, or financial information, for example. Many of the free web services, by contrast, are either of limited scope, or else represent hobby work, student programming projects, and the like.

Happily, there are a few exceptions. We’re going to take a look at Amazon.com’s web service offerings. Amazon, of course, has a user interface, presented via HTML, that you can use to
conduct Amazon searches by pointing and clicking with your mouse in a web browser. But Amazon has also been a pioneer in offering XML-based web services that let you do the same thing, allowing you to integrate Amazon data into other applications.

Suppose that you have a FileMaker database containing information about books. For each book you’d like to be able to check whether it’s available from Amazon, and if so, at what price. With FileMaker’s XML Import capability, you can do this easily.

**Accessing the Amazon Web Services**

Working with Amazon’s web services is straightforward, but you need to do a couple of things first. You should visit http://www.amazon.com/webservices; there, you’ll be able to download the web services developer’s kit, which provides useful documentation, and you’ll also be able to apply for a developer’s token, which is a special personal key you’ll need to send along with your web service requests for validation. There’s no charge for either the developer’s kit or the token.

**NOTE**

URLs in this section use yourKey as a placeholder for the key that you obtain from Amazon.

**Types of Web Services**

As you browse the Internet looking for web services, and as you look at the Amazon material, you might see a lot of references to different types of web services, using terms such as SOAP and XML-RPC. Here’s what you need to know about this: Although all the web services we’re interested in work by sending and receiving XML over an HTTP protocol, there are several different ways to do this. HTTP requests are divided into broad categories, called GET and POST. A GET request passes information in the URL: If you’ve ever seen a long, ugly-looking URL in your browser, like http://my.ecommerce-site.com/cartApp.astj?userID=ED45jUJJ&sessKey=6a45Rffe4, you’re looking at information being sent via a GET request. On the other hand, if you’ve ever filled out an HTML form and clicked the Submit button, odds are the data was being sent behind the scenes, in what’s called a POST request.

Web services can work with either GET or POST requests. FileMaker’s XML Import feature, however, can make only a GET request. But a number of the most common web services techniques work exclusively via POST. SOAP is an example of such a web services protocol. SOAP transactions send a complex XML request to the server via a POST operation. Many web services are available only via SOAP. A good example is the Google web service, which lets you interface directly to the Google search engine. Because Google is SOAP-based, there is no direct way for FileMaker to interact with the Google web service now. In looking for FileMaker-ready web services, you’re looking for web services that support simple XML over HTTP via GET requests.

The developer’s kit comes with documentation that shows how to formulate various types of HTTP requests for data. Here’s a sample URL:

```
http://xml.amazon.com/onca/xml3?t=xxx&dev-t=yourKey
PowerSearch=title:Genet&mode=books&type=lite&page=1&f=xml
```

This searches Amazon for books with the word *Genet* in the title. Try entering the preceding URL in Firefox or Internet Explorer 5 or greater (which render the resulting XML
nicely), and you’ll see what the returned results look like. This returns data in Amazon’s lite format, which has less information than the corresponding heavy format.

Amazon’s XML format, whether lite or heavy, is clearly not FileMaker’s FMPXMLRESULT. So, if you want to bring this book data back into FileMaker via an XML import, you need a stylesheet to transform it appropriately on the way in.

**Writing a Stylesheet to Import Amazon Data**

Let’s say we have a book database with the field structure shown in Figure 24.8.

![Figure 24.8](image)

Field structure for a database of book information.

We can bring Amazon data into this FileMaker structure by performing an import from an XML data source and applying a stylesheet to the inbound data. The stylesheet looks and works a lot like the one in Listing 25.7. We show it here, for completeness, as Listing 24.8.

**Listing 24.8 Stylesheet for Transforming Amazon XML into FMPXMLRESULT XML**

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<xsl:stylesheet version='1.0' xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:output indent='yes' method='xml'/>
  <xsl:template match='ProductInfo'>
    <FMPXMLRESULT xmlns='http://www.filemaker.com/fmpxmlresult'>
      <ERRORCODE>0</ERRORCODE>
      <PRODUCT BUILD='11-08-2005' NAME='FileMaker Pro' VERSION='8.0v2'/>
      <DATABASE DATEFORMAT='M/d/yyyy' LAYOUT='' NAME='Student.fp7' RECORDS='10' TIMEFORMAT='h:mm:ss a'/>
      <METADATA>
        <FIELD EMPTYOK='YES' MAXREPEAT='1' NAME='Title' TYPE='TEXT'/>
        <FIELD EMPTYOK='YES' MAXREPEAT='1' NAME='Publisher' TYPE='TEXT'/>
        <FIELD EMPTYOK='YES' MAXREPEAT='1' NAME='ISBN' TYPE='TEXT'/>
        <FIELD EMPTYOK='YES' MAXREPEAT='1' NAME='AmazonURL' TYPE='TEXT'/>
        <FIELD EMPTYOK='YES' MAXREPEAT='1' NAME='ListPrice' TYPE='NUMBER'/>
      </METADATA>
    </FMPXMLRESULT>
  </xsl:template>
</xsl:stylesheet>
```
<FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="OurPrice" TYPE="NUMBER"/>
<FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="UsedPrice" TYPE="NUMBER"/>
<FIELD EMPTYOK="YES" MAXREPEAT="1" NAME="Authors" TYPE="TEXT"/>
</METADATA>

<Resultset FOUND="10">
  <xsl:for-each select="Details">
    <Row MODID="0" RECORDID="{position()}">
      <Col>
        <Data>
          <xsl:value-of select="ProductName"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:value-of select="Manufacturer"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:value-of select="Asin"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:value-of select="url"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:value-of select="ListPrice"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:value-of select="OurPrice"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:value-of select="UsedPrice"/>
        </Data>
      </Col>
      <Col>
        <Data>
          <xsl:for-each select="Authors/Author">
            <xsl:value-of select="."/>
            <xsl:if test="position() != last()">
              <xsl:text>, </xsl:text>
            </xsl:if>
          </xsl:for-each>
        </Data>
      </Col>
    </Row>
  </xsl:for-each>
</Resultset>
</FMPXMLRESULT>
</xsl:template>
</xsl:stylesheet>
This is extremely similar to the earlier stylesheet, even down to the treatment of book authors, which occur in nested groups: Here, we loop over authors in the same way we looped over parent records, flattening them into a single text field.

**BUILDING A MORE FLEXIBLE INTERFACE TO A WEB SERVICE**

The previous section concentrated on the stylesheet that you would use to import data from Amazon. But so far, we’ve just assumed that FileMaker is issuing some hard-coded URL to perform an Amazon search. In fact, we probably want our users to be able to compose their own queries and submit them to Amazon.

There’s no great mystery to this. The Amazon developer’s kit documents the different types of search strings that the Amazon web service can accept. If we’re just searching for books, a lot of the more interesting options can be found as part of the overall “power search” option.

So far, we’ve imported XML only from data sources we specified via a hard-coded URL. It’s also possible, though, when importing XML into FileMaker via a script, to draw the XML from a data source specified by a calculation. Figure 25.9 shows the relevant dialog choice.

![Figure 24.9](image)

*Figure 24.9* When importing XML into FileMaker via a script, you can use a calculation to create the source URL on the fly.

This makes it possible to compose the Amazon URL on the fly based on user input. For example, if we wanted to search for books by Naguib Mahfouz, published by (say) Anchor, the Amazon URL would look like this:

```
http://xml.amazon.com/onca/xml3?t=xxx&dev-t=yourKey&PowerSearch=
➥author:Mahfouz and publisher:Anchor&mode=books
➥&type=lite&page=1&f=xml
```

(In the real URL, you would use your Seller ID, if you had one, instead of the nonsense string `xxx`.) To compose this URL dynamically, you’d have to offer the user a couple of global fields in which to type. Assume that the user called `gAuthorSearch` and
gPublisherSearch. You could then define a calculation field that would look something like this:

http://xml.amazon.com/onca/xml3?t=xxx&dev-t=yourKey
➥&PowerSearch=author:" & gAuthorSearch & " and publisher:"
➥& gPublisherSearch & "&mode=books&type=lite&page=1&f=xml"

And, as shown in Figure 24.9, you can instruct FileMaker to derive the URL from a calculation, which could point directly to this dynamic field. This snippet is useful only as an example of how you might go about this conceptually. In reality, you’d need to do some work to build a nice interface to Amazon. You’d want to add fields for all the types of Amazon searches; there are about seven. You’d also want to provide for the fact that the user might choose to search on some but not all criteria, making it a good idea to omit the unused search types from the URL. You’d have to account for the fact that it’s possible for searches to have multiple words, in which case they have to be enclosed in quotes. And you’d want to account for the different search types Amazon allows, such as searching by exact match or initial match.

**Troubleshooting**

**Wrong XML Format**

*I’m trying to import an XML file someone gave me, but I can’t even get to the Import Field Mappings dialog. FileMaker says there’s an unknown element in the document.*

FileMaker can import only XML that’s in the FMPXMLRESULT grammar. If you got the XML document from some source other than FileMaker, it’s very unlikely to conform to FMPXMLRESULT. You’ll need to apply a stylesheet to the XML as you import it to transform it into valid FMPXMLRESULT XML.

**Errors in Stylesheets**

*FileMaker says there’s a parse error in my XSL stylesheet.*

There’s a lot of programming in an XSL stylesheet—and XSL and XML are fairly unforgiving languages. A single bracket out of place in your stylesheet, and the XML parser rejects it as being ill-formed. You need to be able to track down the syntax error and fix it. A good XML development environment, such as Oxygen (Mac/Windows) or XMLSpy (Windows only), can be a big help in tracking down such problems.

**Correct Stylesheet, Failed Import**

*I’m using my stylesheet and in theory it should work, but when I use it FileMaker keeps rejecting the XML.*

It’s perfectly possible to write a stylesheet that’s correct in itself, but does not produce correct output. When you’re importing into FileMaker, the inbound data has to be in correct FMPXMLRESULT format. Any deviation from that format and FileMaker rejects the data. You might have written a stylesheet that is correct and runs perfectly without an error, but that
nonetheless doesn’t produce correct FMPXMLRESULT output as you intended. Here again, you need to figure out what went wrong and how to fix it.

There are other possible errors as well. For example, if you are fetching either your XML data or an XSL stylesheet from an HTTP server, you get an error if that server isn’t available to you when you try to perform the import.

Unfortunately, FileMaker isn’t much of an XML debugger. If you run into either of the errors we just discussed, FileMaker gives you a terse error message, which could possibly lead you to the line of the file that produced the problem. If the problem is that you produced bad XML from your stylesheet, you might not even get that much information.

This is no fault of FileMaker’s. XML development is a big area and it’s not in the scope of FileMaker’s capabilities to be a full-fledged development environment for generating and debugging XML files. But if you’re at all serious about using FileMaker and XML together, you’ll want to invest in such a tool.

An XML development tool generally consists of an XML editor that provides a lot of assistance in writing XML and XSL files. It might include features such as tag balancing (automatically closing tags when it seems right to do so), command completion (for example, being able to finish your XSL commands for you after you type a few letters), automatic indentation, and, of course, document validation and debugging.

To use such a tool to develop an export stylesheet for FileMaker, for example, you could first do a sample XML export from your FileMaker database into a test file. You could bring this FileMaker XML file into your XML development tool. Then you could write up your XSL stylesheet, have the tool check its syntax to make sure that it’s technically correct, and then have the tool apply the stylesheet to the FileMaker XML. You could then inspect the result for correctness.

We strongly recommend you look into such a tool if you plan on doing much XML work with FileMaker. The Oxygen XML editor, for Mac or PC, is full featured (http://www.oxygenxml.com). On the PC, Altova’s XMLSpy is highly regarded (http://www.altova.com).

**FileMaker Extra: Write Your Own Web Services**

We generally think of web services as being something that someone else has and that we want access to. But web services have many other uses as well. They can provide a powerful way to extend the capabilities of your FileMaker application.

For example, suppose that you needed to compute a Fourier transform, based on some measured signal data. FileMaker has no built-in facility for such analysis—computing the transform requires complex mathematics. (Well, with enough diligence, you might be able to write a FileMaker script to perform a discrete Fourier transform. But its cousin, the fast Fourier transform, requires mathematical operations that FileMaker can’t perform.)
FileMaker already provides a number of extension mechanisms to developers. Many problems can be solved with a custom function. Those that can’t might be addressed by a plug-in already in existence.

For more details about installing and using FileMaker plug-ins, see “Plug-ins,” p. 778.

Web services provide another way to extend FileMaker’s capabilities. They are, in our view, easier to write than plug-ins, which require knowledge of a low-level programming language such as C++, and knowledge of how to program in each specific client environment supported by FileMaker 7, namely Windows and the Mac OS. Web services, by contrast, can be written in the lighter-weight scripting languages, which we feel are easier to learn, and because they execute in a server environment, they don’t require that you have any knowledge of how to program specifically for the Mac or Windows.

Of course, this points up one of the hurdles involved in writing your own web service: You still have to know how. Web services can be written in a wide variety of programming languages, such as PHP, Perl, JSP, ASP, Visual Basic, Tango, Lasso, or any of many other web scripting languages, not to mention hardcore languages such as Java, C, and C++. There’s literally no limit to the kinds of work you can perform with web services written in these languages. The only catch is, again, you have to know how.

Of the languages discussed, we feel the web scripting languages are probably the most approachable. PHP is a superb general-purpose web scripting language. JSP has a Java base, whereas ASP and Visual Basic are particular to a Windows server environment. Tango and Lasso were once exclusively FileMaker-aware web tools but have since grown into more general-purpose languages. All these languages presume some familiarity with the fundamentals of computer programming and familiarity with the specific language in question.

Let’s return to the hypothetical example. Let’s say you’re importing signal data from an electronic instrument of some kind. You have the raw data and you want to compute a discrete Fourier transform of the samples. Our strategy for doing this is twofold: First we write a web service capable of doing the math, and then we call that web service from FileMaker, hand it our data, and get our results back in return.

In a book of this kind, we can’t explain in detail how to write the kind of web service that would do this. Conceptually, if you know a language like PHP, you can write a PHP program, designed for access over the Web, which expects to receive a vector of numbers in the request. You would call the web service via a URL that might look something like this:

```
http://webservices.my-company.net/DFT.php?samples=
Formatting para
"1.0, .45, 3.2, -.23, 1.76, 1.55, 2.01, 1.23, .34, -.78, -.64, -.09"
```
The samples represent the actual data you are sending to the web service for processing. You would compose the URL dynamically in FileMaker, much as we demonstrated for the Amazon example earlier in this chapter. The URL accesses the web server at webservices.my-company.com, requests access to the discrete Fourier transform program, and passes the DFT program a series of sample values. The DFT program processes the information and returns some information. For the purposes of getting the resulting data back into FileMaker, we want to work through the FileMaker XML Import feature, so the DFT program should output XML of some sort—either straight FMPXML that we can import back into FileMaker, or some other XML flavor that we can transform with a stylesheet.

So, what’s DFT.php? Well, it would be a program, written in the PHP language, which knows how to compute a discrete Fourier transform from a vector of numbers and output the results in XML. You might choose to write the program in straight PHP. For more advanced math, though, such as the more complex fast Fourier transform, you might choose to use PHP to call a code library on the web server computer, which would perform the complex math in a very fast language such as C.

The one downside to using web services in this way (besides the need to learn one or more additional languages) is that the web service functionality doesn’t really live “in FileMaker.” It lives on a server someplace, so if you are creating standalone FileMaker applications meant to work in a single-user, nonhosted environment, or possibly an environment with no network or Internet connection, home-brew web services are probably not the way to go.

Few limits exist to the kinds of programming tasks you can accomplish in FileMaker just by hooking it up to an appropriately written web service. Of course, those services are not trivial to produce in practice. But if you have the knowledge to do so, or access to someone with such knowledge, the potential uses are almost limitless.

To recap, web services can provide a way to extend the capabilities of FileMaker. The range of possible functionality is much wider than that afforded by custom functions, and programming web services is, in general, easier than programming FileMaker plug-ins (which requires writing platform-specific compiled code).

With the Web Viewer, you can retrieve the HTML source of a page. Using a custom function such as ExtractData (available from http://www.briandunning.com/cf/1), you can then parse the data without worrying about XSLT.