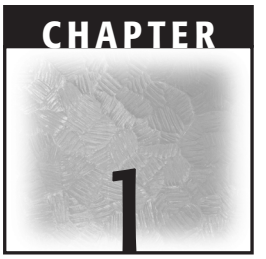

Introducing Oracle XSQL

Welcome to the exciting world of eXtended Structured Query Language (XSQL) development! What's so exciting? Efficiency and ease of use. XSQL isn't some razzle-dazzle technology to wow your users. It also isn't the latest X standard du jour that no one can stop talking about—until you ask, "But what does it do for me today?" The problem with all of the great stuff out there is that no one technology does it all. A Web server doesn't store all of the company's inventory data. A database, by itself, cannot present its data to its users in an attractive and usable manner. This is where XSQL comes in. XSQL allows you to easily leverage the most robust, mature, and usable technologies in the industry: Standard Query Language (SQL), HyperText Markup Language (HTML), HyperText Transfer Protocol (HTTP), eXtensible Markup Language (XML), Java, and the Oracle relational database management system (RDBMS).

Each of these technologies is arguably the best-of-breed in its space. When it comes to querying relational data, SQL has no competitors. HTML and HTTP are the wonder twins of the Internet. They have their faults, but they also have the ability to evolve. Java has had unparalleled success in the world of enterprise applications and will continue to do so. XML is the standard for structuring data in a platform and application-independent manner. Last but not least, the Oracle RDBMS is the technology, as well as the market, leader in its space.

In the next few hundred pages, XSQL allows you to bring these powerful pieces together. You'll learn how you can use XSQL to instantly present your database data on the Web. You'll develop a complete application with just XSQL and eXtensible

Stylesheet Language Transformation (XSLT). You'll also see how to use XSQL to create graphics on the fly and to render Portable Document Format (PDF) documents based on dynamic data. All the while, the easiest cases require no Java coding at all. You only have to use Java coding for the more complex interactions with the database.

This chapter serves as a general overview to XSQL, as well as the foundation technologies. The first topic covered is an examination of what XSQL solves. This includes some short code examples that illustrate how powerful XSQL can be. The next discussion explores how XSQL relates to other Oracle technologies. You don't have to use XSQL exclusively with Oracle, but XSQL makes a great combination. The chapter ends with some in-depth exploration of XML. XSQL and XSLT are derived from XML, so you need to have a good understanding of XML before diving in.

What XSQL Solves

Before trying to learn any new technology, it is worthwhile to thoroughly understand the problems that the technology solves. In the previous paragraphs, you were promised that XSQL brings technologies together. In this section, you'll see, at a high level, how XSQL delivers on that promise. More important, however, you'll see why the integration of the various technologies is a problem in the first place.

The first step is to understand what the key problems are with Web application development. The marriage of the Web with enterprise applications has its problem spots, just like any marriage. As you'll see, XSQL greatly simplifies a key component of the relationship: database access. This leads to the next discussion: How does the database fit into modern Web application development? As you'll see throughout the book, the Oracle database is great for storing not only relational information, but XML information, also. XML, in turn, offers a lot of solutions to the fundamental problems of Web application development. Because XSQL is fundamentally XML based, it's important to understand how XML provides these solutions. Always nearby XSQL is XSLT. XSLT allows you to transform XML into . . . well, almost anything. In a world of so much technology, the problem of transforming from one data format to another comes up regularly. XSLT solves this problem, and will start the examination in this section. The section ends with an examination of how XSQL bridges the gap between all of these technologies.

The Problems of Web Application Development

This chapter is all about perspective, and nothing gives a better perspective than history. So to begin our discussion of the current state of Web application development, it's worthwhile to first consider the history of the Web itself. In the beginning—way back in 1990-—there were just HTML, Uniform Resource Locators (URLs), and HTTP. The real beauty of the system was hyperlinks. Tim Berners-Lee was proposing that you could link documents together—get this—across a network! His paper, "Information Management, A Proposal" from 1990 says it best:

> *Imagine, then, the references in this document, all being associated with the network address of the thing to which they referred, so that while reading this document you could skip to them with a click of the mouse.*

From this concept, the hyperlink as we know it today was born. Now, Tim Berners-Lee wasn't the first to conceive of a hyperlink, but he implemented his system correctly and kept it simple enough so that it could propagate. He was also helped by a couple of other factors. First, the timing was right. The Internet's underlying protocol, Transmission Control Protocol/Internet Protocol (TCP/IP), was well formed and widely used by this time. Second, he invented the system in an academic setting. This is a commonality of many of the great Internet standards. It's easier to freely share a set of protocols through academia than in a commercial landscape. In the early 1990s, it was unheard of for software companies to give away complex software and designs for free.

However, the key reason why the Web grew is that it began as a very simple system. HTTP began as an extremely simple protocol and remains largely so to this day. You send a request and receive a response. It's the server's duty to figure out how to respond, and it's the client's duty to act upon the response. There were only three possible methods for asking: GET, POST, and the rarely used HEAD. This made it easy to develop Web servers. The client only had to understand a handful of possible responses. Likewise, HTML was designed with simplicity in mind. Instead of using the much more powerful but much more complex SGML, Berners-Lee opted for only a subset. Developing servers and clients was so easy, in fact, that much of the early development of the Web was completed by developers and students working in their spare time, or by professionals working on it as a special project. It's telling to note that the two institutions that did so much to give birth to the Web—the Conseil Européen pour la Recherche Nucléaire (CERN) and the National Center for Supercomputing Applications at the University of Illinois at Urbana-Champaign—had no research focus on network computing at the time the Web was invented in their labs!

Because of its simplicity, the Web spread like wildfire. It spread so far that now more than half of the American people have Web access. Most everyone who has a computer on their desk at work has Web access. This pervasiveness makes it an ideal platform for application development. You don't have to worry about installing a software application on everybody's desktop or requiring customers to install software at home. On top of that, you don't have to worry about different platforms. From a logistical standpoint alone, the Web is the obvious platform choice for many applications.

There's only one little problem: The Web was originally intended for simple document sharing! This causes some issues, the most obvious of which derives from the stateless nature of the Web. HTTP originally didn't support cookies, which allow you to bind different HTTP transactions together into user sessions. When you are just sharing static documents, you don't need to tie different HTTP transactions together. The documents always remain the same, so it doesn't matter what documents the user requested previously. However, when your data is dynamic, it often does matter what requests preceded the current one. A shopping cart application is a good example of this. When the user is ready to purchase the items, the Web application has to have tracked what items were selected across several HTTP transactions.

There are several techniques to address this problem, not the least of which is cookies. XSQL fully supports both cookies and servlet sessions. You'll learn about these mechanisms as the book progresses. More to the point, though: the mechanisms for supporting sessions were added to the original HTTP after the initial design, as were JavaScript and the concept of connecting databases to the Web. Perhaps most important, however, is that HTML documents are inadequate for conveying information.

HTML only gives instructions to a Web browser regarding how to display the information. This is a very important function, but it means that you can't interpret the HTML semantically. This is where XML comes in.

It's easy to say that the Web could have been designed better, but hindsight is always 20/20. In truth, the Web is great because it's so easy to extend. Though it was originally intended for static documents, it was easy to add support for images and dynamic data. A Web server doesn't care what kind of information it sends or where it came from. HTTP merely describes the mechanics of transferring the information in a simple and lightweight manner. If the HTML document contains JavaScript, that's fine—it's up to the browser to understand how to use that information.

Likewise, creating database-driven Web pages is just a burden on the server. Web browsers don't know the first thing about interacting with a database. Strictly speaking, an HTTP server process doesn't, either. It just knows to hand off certain URLs to servlets and other server-side modules that interact with the database and produce dynamic results.

This evolution continues today with Web services. HTTP is so simple that you can easily embed simple Web clients in your code. Then, you can grab data from remote machines and use their data in your programs. Because HTTP doesn't care what is sent, you can use XML to structure the data. HTTP is also very loose in what it receives, so you can send data back to the server. Thus, a protocol originally intended to make it easy for physicists to share documents can be used as the backbone for powerful distributed applications.

The process of developing Web applications is maturing. While early Web application developers had to concoct solutions as they encountered a wide variety of problems, a lot of the pioneering is done. The best solutions are being recognized as such and adopted widely. Java and Java Database Connectivity (JDBC) are good examples of this, as are XML and XSLT.

The XSQL framework is yet another evolution in Web development. With XSQL, producing dynamic Web pages that interact with the database is nearly as simple as writing an HTML page itself. In the not-so-distant past, you had to write a module in a language like Java, Perl, or C++ that managed the database connection, executed SQL against the database, and then processed the results. That just got you started. From there, you had to figure out what to do with the results. Because the number and type of results could vary widely for the same module, you had to deal with issues like how many results to put on a page and how to format different types. This model, often called the three-layered model, is illustrated in Figure 1.1.

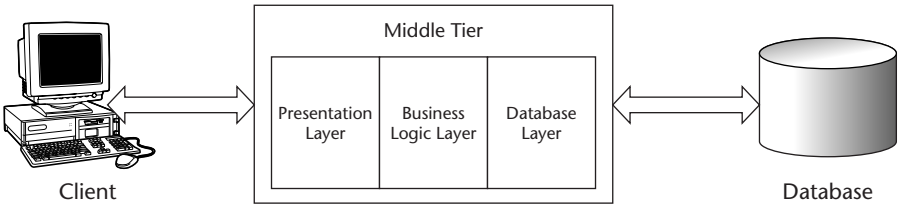

Figure 1.1 The three-layered model.

As already discussed, the user interface (UI) tier only knows how to present the data to the user. The database stores the data, usually for a variety of purposes beyond any one particular application. This leaves a lot of work to be done by that middle layer. A lot of architects like to refer to the middle layer as containing business logic. This euphemism seems to imply that the middle layer is a pristine set of simple, easy rules like "fill the warehouses to 80 percent of capacity" and "offer 10 percent discounts across the board." The client takes care of all the messy UI stuff, while the database does the hard work of managing the data.

When you peel back and examine that middle layer, it usually doesn't look like the drawings on the whiteboard. Instead, you find a lot of hard-coded SQL and UI code deep in the middle layer. Though many application development teams do their best to separate a presentation layer and a database layer, it's hard to define the lines sharply. Even if you use a scripting language like Java Servlet Pages (JSP), the snippets of code usually have dependencies deep in the system. What if the UI designer decides they want a slightly different set of data returned, or they want it ordered in a different way? Then you have to find the SQL statement and change it. That might have repercussions elsewhere in the system. You might have to create a new class. Thus, to make a relatively simple UI change, you are forced to make changes at the database layer. When the system inevitably has to be extended, then you'll probably find very little of your UI code to be truly reusable.

Now, let's assume, for a moment, that a particular application has achieved a good separation between the various layers. Everyone read all of the design pattern books and spent a lot of time at the whiteboard before coding started. They all had enough time to do the separation correctly, and the management or customer understood how important it is. There is still another problem: The output of the system is HTML. What if you want to make the data available to Web services or to a new platform such as wireless? Then you have to port the UI layer to an entirely new type of user interface. Because the code was written with only HTML in mind, you'll probably have to rewrite all of the interface widgets. If the system is perfectly designed, this is a lot of work. Now, imagine if the system isn't perfectly designed.

The Web is the greatest accidental application development platform ever. It started as an internal project at an academic institute in Switzerland and has grown into one of the great technological forces of our time. It has its complexities and intricacies, but it is infinitely adaptable. The key to the success of the Web is to understand it as an evolving technology. The art of developing Web applications is also evolving, and a successful Web application developer is always on the lookout for the next evolution. Now, it's time to see how XSQL greatly simplifies the process of developing for this platform.

XSQL as a Keystone Technology

XSQL is a keystone, rather than a cornerstone, technology. The Oracle RDBMS is a great example of a cornerstone technology. There are many companies whose entire businesses are built around their Oracle databases. Java and HTTP are also cornerstone technologies. However, XSQL is more like the keystone of an arch—the piece that holds the rest of the technologies together in a simple and elegant manner.

To see why, examine what you actually need when developing a database-driven Web application. Clear your mind of all of the three-tier and *n*-tier talk that you've heard. Many database-driven Web pages out there are really just beautified database queries. For these Web pages, the requirements are simple:

- SQL for extracting the data
- A way to produce HTML for presentation

Creating the SQL is simple. If you don't already know it, you'll learn all about it in Chapter 8. The problem is that you have to go from the SQL result set to the HTML markup. In many cases, this is what the entire middle tier in the three-tier model is doing. You start with a result set of some sort, like this:

```
>SELECT ename, job, sal FROM emp
    WHERE deptno=20
    ORDER BY sal;


ENAME   JOB              SAL
------- --------- ----------
SMITH   CLERK            800
ADAMS   CLERK           1100
JONES   MANAGER         2975
SCOTT   ANALYST         3000
FORD    ANALYST         3000
```

This isn't very palatable for the users, so you want it to look prettier. The results are shown in Figure 1.2.

Figure 1.2 Pretty SQL results.

In a case like this, you aren't doing a lot of work on the data. The data is the same, and the results are in the same order. You just want to present the results in a better way. All you need is a way to transform the results that come back from the database into what you want. The ideal situation is shown in Figure 1.3.

This is where XSQL and XSLT come in. XSLT will take an XML document and transform it into whatever you want. It is an open-standards solution to the problem of merging dynamic data with static HTML. This usually means transforming it into HTML, but you aren't limited to just that. For instance, in Chapter 15, you'll see how you can use XSLT to write scripts based on database data. If you know cascading style sheets (CSS), then you have a head start in understanding XSLT. If you are familiar with Server Side Includes, you can consider XSLT as Server Side Includes on steroids. XSLT is almost always used in combination with XSQL in some way. The core architecture is shown in Figure 1.4.

To get a better idea of how SQL, XSQL, and XSLT work together, here is some sample code. These two files are all you need to produce the Web page that was shown previously in Figure 1.2. The first file is the XSQL page. The `<xsql:query>` element, which is called an action, defines the SQL query that you need:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="emp-intro.xsl"?>
<page xmlns:xsql="urn:oracle-xsql" connection="demo">
  <xsql:query>
    SELECT ename, job, sal FROM emp
     WHERE deptno=20
     ORDER BY sal
  </xsql:query>
</page>
```

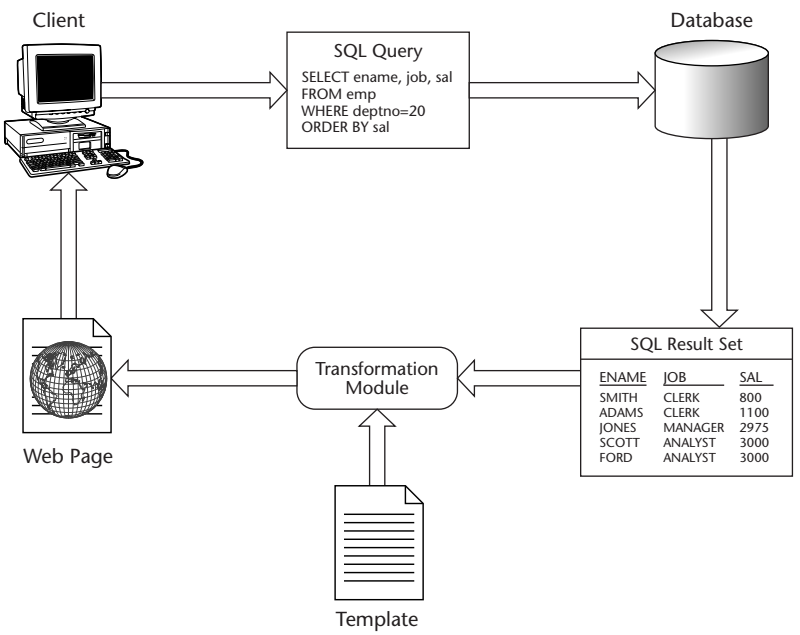

Figure 1.3 Transforming SQL results.

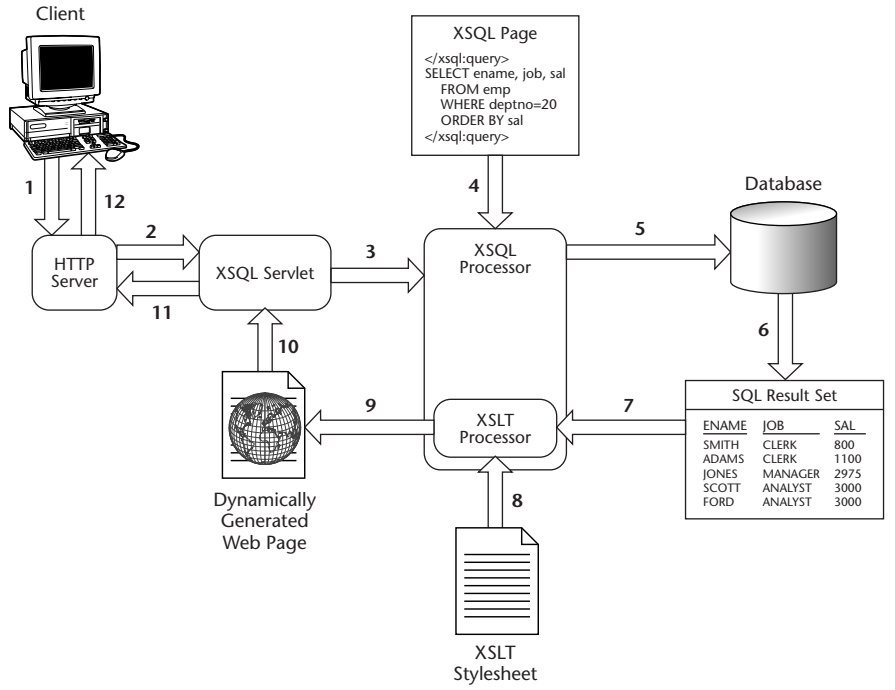

Figure 1.4 Core XSQL architecture.

The XSQL page processor connects to the database, gets the results back, and returns the following XML:

```
<page>
 <ROWSET>
  <ROW num="1">
   <ENAME>SMITH</ENAME>
   <JOB>CLERK</JOB>
   <SAL>800</SAL>
  </ROW>
  <ROW num="2">
   <ENAME>ADAMS</ENAME>
   <JOB>CLERK</JOB>
   <SAL>1100</SAL>
  </ROW>
  <ROW num="3">
   <ENAME>JONES</ENAME>
   <JOB>MANAGER</JOB>
   <SAL>2975</SAL>
  </ROW>
  <ROW num="4">
   <ENAME>SCOTT</ENAME>
   <JOB>ANALYST</JOB>
```

```
    <SAL>3000</SAL>
   </ROW>
   <ROW num="5">
    <ENAME>FORD</ENAME>
    <JOB>ANALYST</JOB>
    <SAL>3000</SAL>
   </ROW>
  </ROWSET>
 </page>
```

The second line of the XSQL file links to an XSLT stylesheet. The XSQL page processor tells the XSLT processor to take the XML and transform it according to the stylesheet. Here is the stylesheet that produces the output that you see in Figure 1.2:

```
<?xml version="1.0"?>

<xsl:stylesheet
 version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

 <xsl:template match="page">
  <html>
   <head><title>Simple Stylesheet</title></head>
   <body>
    <h1>Employees</h1>

    <table border="1">
     <tr bgcolor="#DDDDDD">
      <td><b>Name</b></td><td><b>Job</b></td><td><b>Salary</b></td>
     </tr>
     <xsl:apply-templates select="ROWSET/ROW"/>
    </table>
    <hr />
   </body>
  </html>
 </xsl:template>

 <xsl:template match="ROWSET/ROW">
  <tr>
   <td>
    <xsl:value-of select="ENAME"/>
   </td>
   <td>
    <xsl:value-of select="JOB"/>
   </td>
   <td>
    <xsl:value-of select="SAL"/>
   </td>
  </tr>
  </xsl:template>

 </xsl:stylesheet>
```

You'll learn all about stylesheets in Chapter 13, but you probably want a brief description of what is going on here. The root element of the XML document is called page, so the page template in the stylesheet is processed first. That template is everything from `<xsl:template match="page">` to the next `</xsl:template>` tag. All of the static HTML in the template is written out more or less verbatim. The second template, `ROWSET/ROW`, is called inside the table. It defines how the values for each row in the result set should be displayed. As you can see from the screenshot, this template is called for each row that was returned in the result set. If you change the queries so more rows are returned, they will all be displayed.

If you look at the stylesheet, the majority of the code is HTML. It also falls out very logically—the dynamic data appears amid the static HTML, precisely where you want it to. The principle part of the XSQL page is a SQL query. At the beginning of this section, it was stated that you only really needed a SQL statement and a way to turn the results to HTML. The XSQL solution is very close to this. You will need to learn about XSQL and XSLT, but notice that there has been no mention of Java, JSP, JDBC, business logic, or anything else having to do with the middle tier. XSQL handles all of those details for you.

This takes care of getting data from the database, but what about putting data into it? You can also do that with XSQL. You use a different action, called `<xsql:dml>`. Instead of specifying a select statement, you issue a statement that will modify the data. As you'll see in Chapter 14, you can use it in conjunction with forms to create editors. XSQL also provides you with built-in ways to call stored procedures.

You may be looking at this and thinking, "Simple! . . . but maybe a little too simple . . . ." Of course, the simple architecture isn't going to be good enough for all problems. But that doesn't mean that you can't use XSQL to solve harder problems that involve multiple queries or complex validation before putting data into the database. Luckily, you can easily extend XSQL. Remember the `<xsql:query>` that you saw in the same page? You can write your own special actions that you can use just like that one. The action handler code that processes the action is written in Java. Your action handler code can do whatever it likes, and you can pass to it any type of data you like from the XSQL page. As with the `<xsql:dml>` action, you can also make use of parameters passed on by the user. The only expectation is that the action handler generate XML to be added to the datagram. Then, a stylesheet specified in the XSQL page can be used to convert the datagram for presentation. Figure 1.5 diagrams the XSQL architecture with action handlers.

There is one final trick up XSQL's sleeve. You aren't limited strictly to text data! As you'll see in Chapter 19, you can use serializers to produce images and PDF documents. Once again, XSQL can be easily extended to conquer tough problems. The XSQL architecture with serializers is specified in Figure 1.6. The serializer can be used to control what is written as the final output.

XSQL makes it very easy to create simple database-driven Web pages. For the more complex cases, you can use custom action handlers and serializers to extend the architecture. Your custom code works seamlessly with the rest of XSQL, so you don't give up XSQL's elegance and simplicity—you just augment it. XSQL becomes even more powerful when you use it in conjunction with other Oracle technologies, such as Oracle Text and Oracle XML DB. You'll read more about that in the next section.

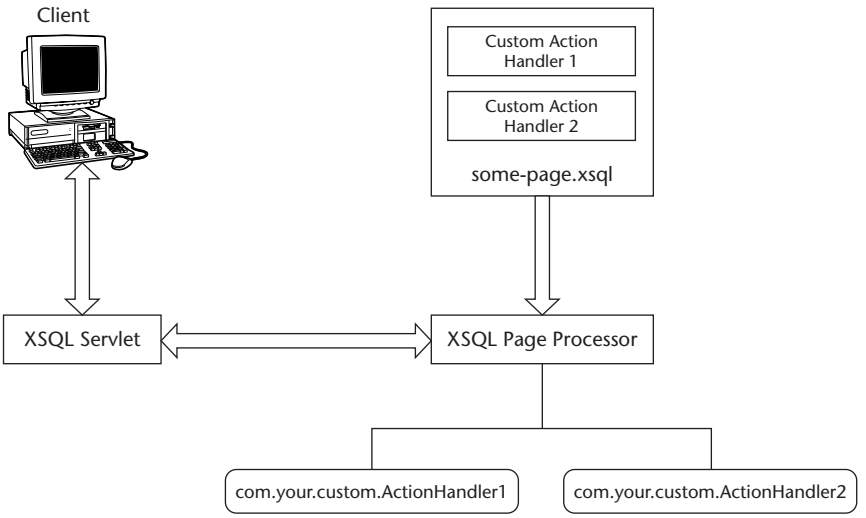

Figure 1.5 XSQL with action handlers.

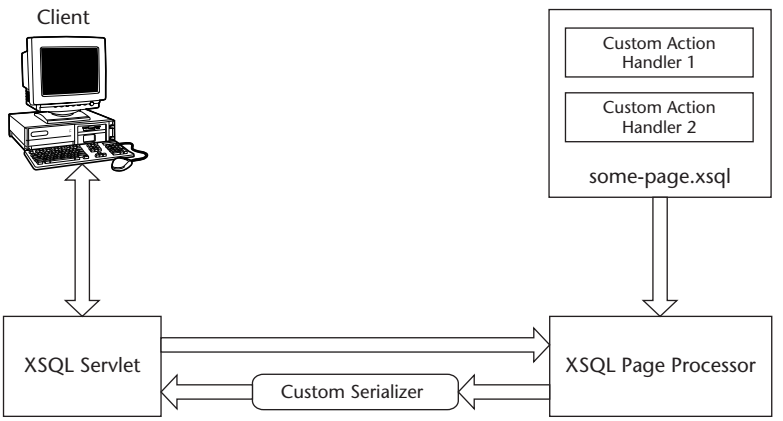

Figure 1.6 XSQL with serializers.

XSQL and Other Oracle Technologies

XSQL can be used with any database that supports JDBC. Being an Oracle product, though, it is optimized for use with the Oracle Server and its family of database

technologies. Oracle provides a rich Java application program interface (API) for database access and XML. You'll use these when extending XSQL with action handlers and serializers, and also when using XSQL from inside programs. You can also use Oracle JDeveloper to help you develop your XSQL pages. This section looks at the core technologies and how they relate to XSQL.

Oracle Text

SQL is great when the data is structured, like accounts receivable or inventory. A lot of data, however, is unstructured text. Searching text is different than searching records in a database. When searching text, you want to know if keywords occur in the text and how they occur. SQL, on the other hand, is used mainly to see if a record matches to a particular term. Oracle Text bridges the gap by allowing you to perform complex unstructured text searches inside SQL statements.

Because Oracle Text is tightly integrated with Oracle SQL, you can use it from any other SQL statement in your XSQL pages.

XML Support

XML is a fantastic way to store and transport data. However, it doesn't exist in isolation. If you are writing a program that consumes XML, you need to be able to parse, create, and update XML documents. If you want to store XML, you'll quickly run into problems if you strictly try to store XML as files. Multithreaded applications don't interact well with flat files. What if two users are trying to write to the file at the same time?

Oracle provides a variety of tools to help you in the brave new world of XML. Oracle provides an XML parser and XSLT processor. These are used by the XSQL servlet in the simple model, but you can use them directly in your applications, too. These tools are part of the XML Developer's Kit (XDK), which includes a robust set of classes that allow you to interface with XML in your code. Both the Document Object Model (DOM) and Simple API for XML (SAX) APIs are fully supported. In addition, you get a lot of additional methods to make your life easier. These APIs are covered in depth in Chapter 17.

This takes care of handling XML programmatically, but what about storing documents? New to Oracle 9i, you can store XML documents directly in the database using the XmlType. This is an object type and takes advantage of the object-relational capabilities of Oracle. Once in the database, you can use XPath, the XML search language, to search XML and extract data from inside the documents. The searching of XML is integrated with Oracle Text, so you can do complex text searches on an entire XML document or just certain nodes inside the XML document. All of these capabilities are integrated with Oracle SQL and are accessible from XSQL pages. You'll learn more about this capability in Chapter 11.

Oracle is already the best relational database on the market. Now, it is in step with the latest advances in XML. Not only can you store and search XML inside the database, you can also handle XML programmatically using Oracle-supported APIs. Throughout the book, you'll see how well Oracle is integrated with XML in all of its forms.

Oracle JDeveloper

You can use any development tool you wish to develop XSQL pages, but Oracle JDeveloper offers some advantages. It is a development environment for building all types of Java and Java 2 Enterprise Edition (J2EE) applications with great support for XML. It highlights your text for you, provides easy lookup of Java classes, and checks the syntax of your XML and XSLT documents. In addition, you get the project management capabilities found in most development environments.

Because XSQL pages are also XML documents, the XML syntax checking will keep your documents well formed. JDeveloper also provides specific support for XSQL page development. It will perform XSQL-specific syntax checking and will interactively execute your code. When you get into the development of action handlers and serializers later in the book, you can use it to interactively debug your Java code.

The approach of this book is to be a development tool agnostic. All of the various types of files—Java, XSLT, and XSQL—are all text files that can be developed with any text editor. This said, JDeveloper is tuned for XSQL page development. If you haven't already found a development environment that you like, JDeveloper is an obvious and promising candidate.

Introduction to XML

Now that you have an overview of the XSQL architecture, it is time to explore the nitty-gritty of XSQL. As with any language, you also need to know a bit about the syntax. Because all XSQL documents are also XML documents, the syntax is actually defined by XML. XSLT is also an XML-based language, so the same is true for it. This means that you need to learn a bit of the basics of XML.

XML is a very simple language. Its power lies partly in this simplicity. With XML, you have an easy, industry-standard way to package data. Because XML parsers are prevalent, anyone can parse your XML document and get at your data. Most important, you can design exactly how you want your data structured. Here, you are going to learn the basics of XML. Our aim is to educate you on the structure of XML documents.

The Role of XML

XML is a metalanguage that allows you to define your own markup languages. It is a project of the World Wide Web Consortium (W3C) and is an open standard. An XML document looks a lot like an HTML page, but they serve different purposes. Whereas HTML tells a Web browser how to present data, XML is about structuring data.

XML is best understood as a child of Standard Generalized Markup Language (SGML). SGML is also the parent of HTML, so XML and HTML can be considered cousins, and SGML can be considered the mother tongue. SGML has been in use since the mid-1980s as a way to describe and organize complex documents. Among other

things, SGML was used in the documentation of the stealth bomber. A typical SGML project results in the creation of a complex HTML-like language. In fact, HTML is an application of SGML.

Why didn't the Web creators just use SGML? SGML is much too heavyweight for popular use over the Internet. Though the Web would be a richer place if everyone used SGML to create their Web sites, no browser vendors were willing to implement SGML. SGML would also dramatically increase the network burden. Perhaps most important, SGML is complex to learn, whereas HTML is very simple. Without a simple markup language like HTML, the Web probably would have never reached critical mass.

So then, what's wrong with HTML? While SGML is too complex, HTML is a bit too simple for the demands of contemporary Web applications. HTML is intrinsically tied to how documents should look to their users. Though HTML documents are highly structured, they are structured around presentation. Consider the H1 tag. The text contained inside of an H1 tag is usually a headline of some sort, but you don't really know much more about it than that. About all you know is that the Web page author wants that text to stand out.

Along the same lines, consider Web sites that tell you when the Web site was last modified. You can look at a table of contents Web page and it will tell you instantly when the content was last changed. However, there is no way to look at the HTML code and algorithmically determine when it was last updated. Even though HTML documents are highly structured, they aren't semantically structured. There is no way to look at HTML and interpret it as much more than text data.

What would work is the ability to define your own tags somehow; however, HTML doesn't let you do that. The tag set is created by a standards body. Because all of the browser vendors are expected to implement the latest features of HTML, any additional functionality needs to be universally applicable. This is understandable, of course. HTML represents the presentation layer and will never be applicable to the structure of your data. What is needed, then, is a way to structure your data separate from HTML.

This is the value of XML. Instead of confining yourself to HTML, you are able to create your own language for your application. Oracle's canonical schema that you'll learn about in a few pages is an example of this. Oracle developed a way to represent SQL result sets in XML. Once established, XSQL developers can use it to present and manipulate the data. You also have the power to define your own markup language and can do it very quickly. You don't have to write a parser—XML parsers are widely available. Your burden is simply to think about what you need, create an XML schema, and document it. Once created, other people inside and outside of your organization can use it for data interchange.

NOTE **Defining your own tags is great, but an XML document isn't instantly useful by itself. While an HTML document is immediately usable by millions of Web browsers worldwide, an XML document isn't. You can create a `<LastUpdated>` tag, but what application will understand it? Thus, XML is primarily used, in conjunction with XSLT, to create HTML and other standardized document types, or by a Web services client that knows the XML schema you are using.**

As an XSQL developer, you will be creating your own implicit schemas along the way. However, it is a bit grandiose to think of these as brand new markup languages. In most cases, they are going to be the Oracle canonical representation plus some XML that you add around the edges. Still, this does give you an XML layer to your application that is separate from the actual presentation. In terms of XSQL development, this is the key advantage of XML. You are able to separate your application data from the actual presentation. XSLT yields the actual presentation while you are able to easily repurpose the XML layer.

Well-Formed versus Valid Documents

You may have heard of a document being referred to as valid or well formed. The definition and distinction is important to the discussion here. A well-formed document is simply an XML document that is syntactically correct. A valid document is a well-formed document that also follows the rules of a schema.

Anytime you write XML, it has to be well formed. The need for validity is determined by whether there is a schema associated with your document. On the one hand, if an application is processing the document, it is required to be valid. This makes the job of the application much easier. If the document is unacceptable, the parser will generate an error before the document gets to the application. This greatly reduces the number of error cases that the application must be prepared to handle. Because XSQL documents are fed to an application—the XSQL page processor—they must be valid. They must agree with the schema for XSQL pages.

On the other hand, the XML documents that XSQL generates only need to be valid. This does not mean that it is all free love and long hair. First, you may have a Web service consuming your XML that expects a schema. In most cases, your document is going to be transformed. If your generated XML doesn't come out right, your transformation will be ugly. Fortunately, the XML that XSQL generates is pretty tight. Bad transformations are really only a problem if you get fancy with custom action handlers or include foreign XML.

Document Structure

An XML document is structured like a tree. Trees are some of the most pervasive structures in all of computer science. Probably the most common usage is a file system. Trees are a great elegant structure because they can be used to establish relationships between any set of data. Even a table of data can be easily represented with a tree, as you'll see later in this chapter.

Trees can be generally described with the following rules:

- There can be only one root element.
- The root element is the only element that has no parent.
- Every nonroot element has exactly one parent element.
- Any element can have one or more children elements, or no children at all.

In XML, the parent-child relationships are represented by start and end tags. The following example shows a simple document that contains two child elements.

```
<?xml version="1.0"?>
<Root>
  <ChildElement1 />
  <ChildElement2 />
</Root>
```

This should look very familiar to anyone who has coded HTML. The <Root> element looks like a <UL> tag, and child elements look like <LI> tags. There is a key difference, though. If you look at the child elements, you'll notice that the tag ends with />. This is required in XML so that the parser knows that this element has no children.

The preceding example represents the three tag types in XML: (1) the start tag, (2) the end tag, and (3) the empty element tag. Table 1.1 documents the syntax of these tags. In the "Form" column, the unique syntax requirements for each type of tag are documented. There are also common syntactical requirements that apply to all tags.

As you would expect, start and end tags must be properly nested. A child element must be entirely contained within its parent's start and end tags. The following example demonstrates bad and illegal nesting:

```
<?xml version="1.0" ?>
<Root>
     <Troublemaker>
</Root>
</Troublemaker>
```

Our last structural consideration involves our children elements. In our previous example, there were only two children elements, and each had a different name. This isn't a requirement. You can have many children with the same name. For instance, the XML returned from the database by Oracle will have as many row child elements as there are rows in the result set. Our final example is of a valid document with multiple elements of the same name. Just for fun, this example is also more deeply nested.

```
<?xml version="1.0" ?>
<Root>
    <FirstLevelChild>
          <SecondLevelChild/>
    </FirstLevelChild>
    <RedHeadedStepChild />
    <FirstLevelChild />
    <FirstLevelChild>
        <GrandchildOfRoot>
           <KidsTheseDays />
        </GrandchildOfRoot>
    </FirstLevelChild>
</Root>
```

Table 1.1 XML Tags

TYPE	FORM	EXAMPLE	CONSTRAINTS
Start tag	Must start with <, end with >, not start or end with </ or />.	`<Root>`	Must have a matching end tag of the same name.
End tag	Must start with </, end with >, and not end with />.	`</Root>`	Must have a matching start tag of the same name.
Empty element tag	Must start with <, end with />, and not start with </.	`<Child Element1 />`	

Processing Instructions

You may have noticed that all of our sample documents begin with the same line. This is a processing instruction. Processing instructions give input to the application that is handling the document. The `<?xml version="1.0" ?>` processing instruction is required. It can also have an additional attribute to describe the character encoding of the document. Here is an example that reinforces the default encoding:

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

Your documents can have more than one processing instruction. For XSQL pages, your documents will usually have a processing instruction that references an XSLT stylesheet. This will be covered in depth later this chapter.

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

Attributes

Attributes are name-value pairs that are associated with an XML element. Most HTML elements also have attributes. The bgcolor attribute of the body element is one of many examples. You can have as many attributes in your element as you wish. However, each attribute name can appear only once for a particular element. Values always have to be quoted with either single or double quotes. Attribute names and values are restricted in that they can only contain some characters. These restrictions are the same as the restrictions on the names of XML elements and are covered in the next section.

Here is an example of an XSQL page where three attributes are used: <connection>, <xmlns:xsql>, and <tag-case>.

```
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
  <xsql:query tag-case="lower">
     select * from emp
  </xsql:query>
</page>
```

Syntax Nitty-Gritty

You know almost everything that you need to know to create well-formed XML documents. There are a couple of rules that you haven't seen yet, though. This section acts as a review of the rest of the syntax rules. The most common restrictions that you will encounter involve the names of elements and attributes. This is covered first. XML, like most languages, has reserved characters. Sometimes, you'll want to use these, so you'll learn how. One option you'll learn about is the CDATA entity, which allows you to define special sections of character data. The final section here covers XML comments.

There is one instruction that isn't discussed here: <!DOCTYPE>. It is used to specify a Document Type Definition (DTD), a type of schema. Before covering the rest of the syntax rules of XML, let's review the syntax rules that have already been covered:

- You must have an XML processing instruction at the top of the document.
- There can be only one root element.
- Start tags must have matching end tags, and vice versa.
- Tags must be nested correctly.
- A particular attribute can appear only once per element.
- Attribute values must be enclosed in single or double quotes.

Element Name and Attribute Restrictions

Element names and attributes share the same restrictions. Both must be composed entirely of the same set of characters. This set consists of all alphanumeric characters, the underscore, and the period. The colon is also valid, but has special meaning—you'll learn more about that when XML namespaces are discussed. A name or attribute can only start with an underscore or a letter. The last restriction is that no name or attribute may begin with the string xml. Case-insensitive Table 1.2 gives examples of legal and illegal strings.

Table 1.2 Example Name/Attribute Strings

STRING	LEGAL?	COMMENT
`howdy`	Yes	All valid characters; doesn't start with a number or xml.
`_howdy123-`	Yes	All valid characters; doesn't start with a number or "xml".
`_123-howdy`	Yes	All valid characters; doesn't start with a number or "xml".
`123howdy`	No	Starts with a number.
`howdy everybody`	No	Contains a space.
`howdy!`	No	! isn't a valid character.
`xml-howdy`	No	Starts with "xml".
`howdy:everyone`	Yes, but . . .	Use of a colon is forbidden, except in conjunction with XML namespaces. This is legal only if you have a namespace called `howdy`.

Special Characters

There are several characters that have special meaning in XML. From time to time, this will pose an inconvenience because you want to use these characters. The full set of special characters is described in Table 1.3.

Table 1.3 Special Characters

CHARACTER	RESTRICTIONS	WORKAROUND
'	Not allowed in attribute values quoted by "	Use &apos.
"	Not allowed in attribute values quoted by '	Use &quote.
&	Must always be escaped	Use &amp.
<	Must always be escaped	Use &lt.

The most common problem encountered with XSQL is with the < symbol. This symbol is also an operator in SQL, so its special status in XML causes problems. The following XSQL page will produce an error:

```
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
   <xsql:query>
    select * from emp where sal < 50000
   </xsql:query>
</page>
```

When the XML parser encounters the <, it thinks that it has encountered a new tag. When it encounters the next character, a space, it gives up. There are two workarounds: (1) Use the escape sequence as demonstrated in the following code, or (2) use CDATA, which is covered in the next section.

```
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
<xsql:query>
    select * from emp where sal &lt; 50000
</xsql:query>
</page>
```

CDATA

The CDATA entity allows you to declare a section of character data off-limits to the XML parser. When the parser encounters the CDATA declaration, it skips all characters inside of it. Here is an example that resolves the earlier problem with the < operator:

```
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
<xsql:query>
    <![CDATA[
    select * from emp where sal < 50000
    ]]>
</xsql:query>
</page>
```

CDATA entities are useful any time that you have sections that shouldn't be processed because they also take a load of the processor.

Comments

No programming language is complete without comments. The syntax for comments in XML is as follows. It is identical to HTML comments.

```
<!-- An XML Comment -->
```

Namespaces

As discussed earlier, XML allows you to create your own languages. However, what if someone else has developed a schema that you want to use? There is the chance that element names from this other schema will conflict with yours. XML has a solution for this: namespaces. Namespaces allow you to make your elements globally unique. It does this by attaching your element names to a Universal Resource Identifier (URI). A Uniform Resource Locator (URL) is an example of a URI, as is a Uniform Resource Name (URN). Even if you use an extremely common element name in your document, such as name, you can make it globally unique by specifying a URI that you control.

XSQL uses a URN—`<oracle-xsql>`—for its namespace. You may have noticed it in the examples:

```
<page connection="demo" xmlns:xsql="urn:oracle-xsql">
```

The `<xmlns:xsql>` attribute signifies that some children elements of the page element will belong to the XSQL namespace. Namespaces can overlap—in fact, that's the whole point. The following XSQL example shows this. This page uses both the XSQL namespace and the Mike namespace.

```
<?xml version="1.0"?>
<page connection="demo" xmlns:xsql="urn:oracle-xsql"
xmlns:mike="http://www.ibiblio.org/mdthomas">
  <xsql:query>
   select * from emp where ename='SMITH'
  </xsql:query>
  <mike:mikeNode>
     Value
  </mike:mikeNode>
</page>
```

The output of this XSQL is shown above. Notice that the `<xmlns:mike>` attribute is preserved but the `<xmlns:xsql>` namespace isn't. In the output, no member of the XSQL namespace is present. The `<xsql:query>` element was replaced with the results of the query. Because there are no members of the namespace in the document, the XSQL page processor removes the attribute.

Schemas

We spoke earlier about how XML allows you to create your own languages. These languages are formally known as schemas. A schema is a definition of how elements in your XML should relate to one another, what attributes are appropriate, and what values are appropriate for nonempty elements. You can program very successfully in XSQL without ever creating your own schema. At the least, it's important to be conversant about schemas.

The XML documents that you have seen so far are very simple, but XML can be very complex. The purpose of a schema is to rein in that complexity. This makes it easier for applications to be able to consume the XML—they know what they are expecting.

Since we've talked about creating your own languages with XML, perhaps we can extend that analogy. If the elements are the words of our language, the schema is the grammar. It tells the world how the words have to be arranged so that they are meaningful to our applications. You read earlier about valid XML documents. A schema is used to determine that an XML document is valid for that schema's particular set of rules.

As with natural languages, there is a lot that can go into determining that a document is valid. Think about it in terms of plain old English documents, such as this book. On one level, this book is valid if the individual sentences are grammatically correct. The editors have certain requirements about section headings before they will call it valid. The publisher wants the book to be a certain length and of a correct tone and quality before it is considered valid to ship it to the stores. Ultimately, the reader makes the call as to how valid the book is as a resource based on a number of factors.

The validity tests for XML documents can be as multifaceted as this. At the lowest level, a schema can be used to determine if particular nodes have values of the right type. The next step is to determine that the structure is right. Do the children elements belong with their parent? Does the parent have all of the children nodes for it to be valid? Then, you can look at how the different elements relate to each other to make determinations of integrity. From there, the sky is the limit. If you desire, you can pile complex business rules into your schema.

With an idea as to what schemas are about, it's time to focus on how to implement one. This can be as confusing as the complicated schemas we are talking about! There are several different ways to define schemas. These are called schema languages. (Good thing the languages we develop with XML are called schemas, or else they would have to be called language languages!) The original schema language is DTD. In fact, it has its own instruction built in to XML: `<!DOCTYPE>`. Though still widely used, DTDs are becoming unpopular for a number of reasons, including cumbersome syntax and the inability to define data types. The other popular schema languages are XML based. W3C XML Schema is the heir apparent. There are other players, including RELAX NG, Schematron, and Exampletron.

Moving On

In the previous pages, XSQL was covered at a high level. You learned the problems that XSQL addresses and how it integrates the technologies together. The XML basics covered in the previous section will come up again and again throughout the book. Now, it's time to dive into XSQL development. The next couple of chapters cover the installation and setup of XSQL. After you have XSQL installed, you'll be ready to move forward.