



2

Now Let's Start Digging

Chapter Roadmap

Now that we have presented a high-level concept of XML and Perl, it's time to see some of the tasks that we can accomplish when we combine these two tools. As we mentioned earlier, we will show you how XML and Perl complement each other and make tasks that initially seem to be difficult almost simple (well, at least a lot easier). We're going to discuss XML and Perl, and how they are used together to create portable, powerful, and easily extensible applications.

This chapter covers the concepts and basics of the most popular XML technologies. Each of these technologies is utilized in Perl applications throughout the book, and we provide a working example that clearly illustrates the proper way to use the respective Perl module. Here are the topics that we'll discuss in this chapter:

- What is XML Processing?
- XML Parser Paradigms
- Parsing an XML Document and Extracting Statistics
- Generating XML Documents
- Searching XML Documents

- Transforming XML Documents
- Our First Perl XML Programs
- The Problem: Generating and Parsing an XML Document
- What is XML Processing?

What Is XML Processing?

What is XML processing? That is an often-asked question today given the popularity of XML. In the context of this book, I'll define XML processing as any task that involves XML data (for example, reading, writing, generating, and transforming).

Examples of XML Processing

XML processing can mean different things to different people, depending on what is important to them. What are some XML-related tasks? A few of the more important XML-related topics that will be covered throughout the book are XML generation, XML validation, XML parsing, and XML transformation. Each of these topics will be introduced in the next few sections and then discussed in detail a little later in the book.

XML documents must be generated before they can be processed, so generating XML documents is an important topic. There are several Perl modules that are specifically designed to facilitate generating XML from a variety of input sources. The input data source will usually determine the best XML generating module for each situation. For example, if you are converting an input CSV file to XML, you could use either the `XML::Writer` or `XML::SAXDriver::CSV` Perl modules. If your input data is store in a Microsoft Excel file, you could use the `XML::SAXDriver::Excel` Perl module to help with the generation. Finally, if you need to convert the results of a database query to XML, you could use the `XML::Writer` or the `XML::Generator::DBI` modules.

As you can see, you have a number of options when it comes to support for generating XML. I'll discuss all of these modules in greater detail and present examples a little later in the book.

Validating XML

Once we've generated an XML document (or received one from another source), there are Perl modules that will help you validate the XML document. An XML document is considered valid if it complies with a Document Type Definition (DTD) or XML schema. There are several Perl modules that support XML validation (for example, Apache Xerces), and they will be discussed later in the book.

Parsing XML

Parsing is a concept that goes back a long time (in programmer's time, not historical time). *Parsing* data, in simple terms, is to break the data down into the fundamental components that mean something to the application or the end user. An XML parser is a program that does exactly that—it parses an XML document based on the XML rules and syntax defined in the XML standard, and it provides that information to the requesting application. Information usually comes in a data structure for each construct. If the construct is queried by the application, it will give you information about the construct. For example, for the open element construct (which is recognized whenever the parser encounters a combination of characters similar to `<element_name>`), the parser provides the element's name and other relevant information to the appropriate event handler.

Transforming XML

XML can be transformed to a number of different formats. For example, an XML document can easily be converted to another XML document, HTML, CSV, or any of a number of different formats. This feature of XML is very important and allows a great deal of flexibility when presenting XML data. There are a number of Perl modules that support XML transformation and they'll be discussed in detail throughout the book.

Note

Remember, there is a difference between data and information. The term data can be used to describe an entire XML document (data = markup + information). Information, on the other hand, is the actual information contained in the XML data.

Note

A short XML primer is included in Appendix A, "What Is XML?" to help refresh your memory or give you an abbreviated introduction to XML.

XML Parsers

Parsers come in many different forms. Some XML parsers don't require any programming, you just provide an input XML document, and the parser generates the output data. Other XML parsers provide Application Programming Interfaces (APIs) that allow the programmer to manipulate its functionality from within the application code. APIs are usually highly customizable and come with many different features. In Perl, a parser module provides an API to an XML parser engine. This API hides most of the details of the inner workings from the typical user, while providing a clean, easy-to-use, and well-defined interface.

Parsing an XML document involves reading the XML data and navigating through it while separating the contents of the XML document into meaningful components. Remember, XML data can come from a file, input stream (for example, a socket), a string, or just about any other possible data source. The most popular methods of parsing XML follow either the push model or the pull model. Each of these parsing methods is discussed in the following sections.

Push Model of an XML Parser

The push model, as shown in Figure 2.1, is the core of the event-driven parsers. In the push model, special subroutines called *event handlers* are defined within your application. These event handlers are registered or tied to specific events and are automatically called by the XML parser when a particular event occurs. The push model was given its name because whenever a particular event is encountered, it pushes the event (and some event-specific data) to a predefined handler in the application. For example, you can define subroutines that are called when a start tag is encountered, when an end tag is encountered, and so forth. Using a little logic that we'll demonstrate, it is fairly straightforward to parse an XML document and extract its contents.

As you can see, this is a passive approach to parsing because your application waits for data to arrive. Basically, three steps are involved in this approach.

- Write the event handler subroutines.
- Parse the XML data.
- Wait for the event handlers to be called.

Of course, there is a little more to it, but that is basically what happens. It's important to understand the concept at a high level. This way, when we present the examples and discuss the code that actually performs each of these steps, you will have a good understanding of the process. Event-driven parsers and their underlying concepts are discussed in detail in Chapter 3, “Event-Driven Parser Modules.”

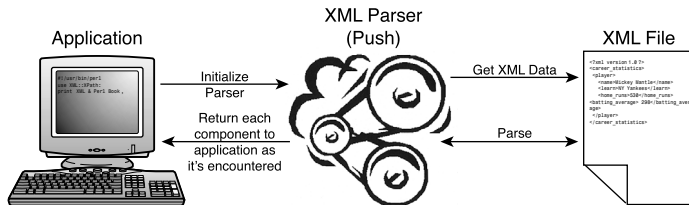


Figure 2.1 Applications built using a push-based XML Parser play a passive role and receive data that is pushed from the XML parser.

Pulling XML Parser Events to the Application

The pull model defines another parsing model. In this model, the application controls the parser and pulls the information from the parser as the parser iterates through the XML data. Figure 2.2 illustrates the pull XML parser model.

As opposed to the push model, the pull model requires the application to play a more active role in the parsing of the document. The pull model basically parses the XML data and holds it in a tree structure. In order to retrieve data, the application then must walk up and down through the tree and extract the desired sections of the XML data. The pull model is discussed in detail in Chapter 4, “Tree-Based Parser Modules.”

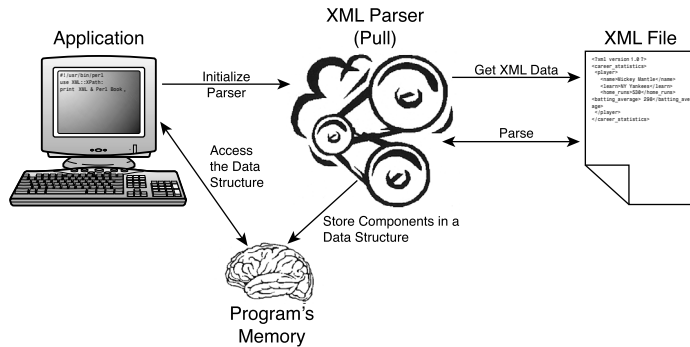


Figure 2.2 Applications built using a pull-based XML parser play an active role in the parsing by pulling data from the XML document.

Together, these two paradigms form the basis of most XML parsers. We'll discuss the advantages and disadvantages of the push and pull methods throughout the book.

XML Parser Paradigms

Now that we have discussed the low-level parser models, push and pull, let's look at the more commonly discussed models that are built on top of the push and pull concepts. The two major types of parsers are event-driven and tree-based parsers.

Event-Driven XML Parsers

An event-driven parser is a model that can be closely associated with the push model. Event-driven parsers enable an application to define events for each recognized construct, and after that construct (event) is encountered, the parser calls the correct function in the application associated with that event. For example, we can define an event that is called whenever the opening tag (for example, `<name>`) of an element is encountered. Simple API for XML (SAX) is the most popular event-driven parser specification. It defines a standard way to access and manipulate a SAX-compliant parser. This is a very powerful model due to its speed and efficiency. We'll discuss event-driven modules in Chapter 3, "Event-Driven Parser Modules."

Tree-Based XML Parsers

When we memorize some information (for example, a friend's telephone number), it's easier to remember it than it is to look it up in the telephone book. Unless we have amnesia, our memory serves as the fastest and most reliable source for retrieving data. If you look at the tree-based or pull parser model shown in Figure 2.2, you'll notice the parser has a brain of its own and, therefore, stores the parsed information in a data structure. After the information is stored in the memory of these data structures, it can be easily accessed from the application. One of the most popular tree-based XML parser models is called Document Object Model (DOM), which is a specification that defines the behaviors and data structures of DOM-compliant XML parsers. We'll talk more about DOM and other tree-based XML parsers in Chapter 4, "Tree-Based Parser Modules."

XML Parsers Versus XML Processors

Try not to confuse XML parsers with XML processors. XML parsers are different from processors in that an XML parser is only one of the components of an XML processor. An XML processor may include several different components required to accomplish its task.

A perfect example of an XML processor is the eXtensible Stylesheet Language (XSL) Transformations (XSLT) processor. The XSLT processor is composed of many components, one of which is the actual XML parser itself. It uses the parsing component to decompose an XML document into meaningful parts so that other XSLT components can operate on the individual components. An XSLT processor converts requires two inputs: an XML document and an XSLT stylesheet. The XSLT stylesheet is an XML document that contains rules (called templates) that describe the format of the output document. The XSLT processor then applies these rules to the input XML document and generates the desired output. For example, we can start with an XML document, use an XSLT processor, and the end product could be an HTML document (for display on a web page), or even another XML document that is a subset of the original XML document. XSLT and other related topics will be discussed in Chapter 8, "XML Transformation and Filtering."

Parsing an XML Document and Extracting Statistics

Now that you've got a good understanding of what it means to parse an XML document, let's take a look at a short example that uses one of the tools that were just discussed. For this example, we'll use the XML::Parser Perl module to parse a small XML document. Listing 2.1 shows a small XML document that contains statistics from two great baseball players. As you can see, the XML document is very simple. It has two <player> elements, and each <player> element has four child elements (<name>, <team>, <home_runs>, and <batting_average>). Granted, this is a small and simple XML document, but it is perfect for illustrating simple XML parsing. Our goal for this example is to parse this document and extract the statistics for each player.

Listing 2.1 Career statistics for two Hall of Fame baseball players stored in an XML document. (Filename: ch2_baseball_stats.xml)

```
<?xml version="1.0"?>
<career_statistics>
  <player>
    <name>Mickey Mantle</name>
    <team>NY Yankees</team>
    <home_runs>536</home_runs>
    <batting_average>.298</batting_average>
  </player>
  <player>
    <name>Joe DiMaggio</name>
    <team>NY Yankees</team>
    <home_runs>361</home_runs>
    <batting_average>.325</batting_average>
  </player>
</career_statistics>
```

We wrote a small application to parse this XML document. Don't worry about the source code yet, we'll start off slow and start showing Perl applications a little later in this chapter. For now, just focus on how an XML parser works, and don't be too concerned with all of the small details yet.

We used the XML::Parser Perl module to build an application to parse the XML document. The XML::Parser Perl module is an event based parser that will be discussed in Chapter 3, "Event-Driven Parser Modules." The output generated by the Perl XML parsing application is shown in the following:


```

career_statistics =

player =

name = Mickey Mantle
team = NY Yankees
home_runs = 536
batting_average = .298

player =

name = Joe DiMaggio
team = NY Yankees
home_runs = 361
batting_average = .325

```

Note that in the output, each element is printed out in the same order as the order that the opening tags appear in the XML document. As you look at the output listing, you're probably asking why the `<career_statistics>` and `<player>` elements are empty, right? Well, take a look at the original XML document in Listing 2.1 again. Notice that the original XML file really contains a `<career_statistics>` element that is made up of two child elements named `<player>`. Each `<player>` element is made up of four child elements: `<name>`, `<team>`, `<home_runs>`, and `<batting_average>`. So, when parsing the XML document, the `<career_statistics>` and `<player>` elements don't contain any character data themselves, but rather contain other elements—their child elements. After the `<name>`, `<team>`, `<home_runs>`, and `<batting average>` elements are encountered, the parser can extract the character data contained by each element. Of course, more complex scenarios exist in which elements contain attributes, character data, and child elements, but let's go one step at a time. You will have examples later that cover all these scenarios.

Now that we've given a quick introduction to parsing XML, the first question is, how do we do it? That's the easy part: use any number of Perl XML modules. It's only natural to try to use these two technologies together to solve complex problems. A large number of Perl modules are available that perform just about every conceivable task related to XML. As of this writing, there are nearly 500 Perl modules related to XML. We're going to explain the most popular modules, show you how to use them, and demonstrate them using real-world examples that can easily be extended to support more complex applications.

Generating XML Documents

Where do XML documents come from? It's kind of like the famous question: Which came first, the chicken or the egg? You can't parse an XML document until you build one. The source data for an XML document can come from just about any imaginable source. For example, an XML document can be generated from a CSV file, a tab-delimited text file, the results of a database query, a Common Gateway Interface (CGI) web-based user input form that collects data from a user, Web services, and many, many other sources. You can see that there is an almost infinite number of data sources for XML documents.

Let's take a look at two methods of generating XML documents. First, an XML document can be created in a standard text editor or an XML-specific What You See Is What You Get (WYSIWYG) editor. This is called *static generation* because the XML document is created by typing the document into a text editor. The data in the file remains the same unless someone or something (that is, an application) modifies it. There isn't too much to it—you basically just type the contents of the XML file as you would type any other document. This is an option for very small XML files (for example, configuration files); however, it isn't practical for larger files. The larger and more complicated an XML document is, the greater the chance of error if the file is edited by hand.

XML documents can also be generated dynamically by an application. Dynamically generating XML is more interesting and applicable than statically generated documents, so let's take a more detailed look at this topic in the following section.

Dynamically Generating XML Documents

Another method of generating an XML document is to dynamically generate the contents using an application (preferably Perl based). Because XML is just plain text, an entire XML document can be easily contained within a Perl scalar. It also can be printed to any filehandle with a simple print command. Of course, plenty of modules are available that generate XML, which takes away some of the complexities of doing everything yourself. We'll discuss these modules in Part III of this book, "Generating XML Documents Using Perl Modules." Some of the topics discussed include generating XML documents from text files, databases, and even other XML documents.

A good example to demonstrate dynamic XML file generation is a web form that is filled out by a user and then submitted to a Perl application running on the web server. This is certainly dynamic because we don't know what the user will enter as data. Assume that the web form shown in Figure 2.3 allows the user to fill out certain information and then submit the collected information to the server. The information provided by the user will be processed by a Perl script and used as input or source information for an XML document. An XML document will be generated containing the information provided by the user.

We won't get into too much detail here, but we will revisit this example in Chapter 5, "Generating XML Documents from Text Files." The purpose here is to provide a few high-level examples to illustrate the concepts instead of diving right into source code.

User Input Form	
First Name	John
Last Name	Smith
Address	1155 Perl Drive
City	Los Angeles
State	California
Zip Code	90102
Phone Number	818-555-1212

Figure 2.3 Web-based HTML submission form used to collect data for our XML document.

When the Perl application receives the information from the form, it comes in a certain format. The Perl application can then parse this format to separate each field individually. Let's say the user fills in the required information and then submits the form. Now, inside our Perl application, we have the following data:

```
First Name: John
Last Name: Smith
Address: 1155 Perl Drive
City: Los Angeles
State: California
Zip: 90102
Phone Number: 818-555-1212
```

The Perl script can now process and modify the data and then generate an XML document containing this information. Listing 2.2 shows the submitted data in an XML document. This file now can be processed as an XML document. For example, now that the information submitted by the user is in XML, you have a variety of options. The XML document can be transformed to another format (using XSLT), searched (using Xpath), or sent to another application as a document that is based on an agreed-to format.

Listing 2.2 **Sample XML file generated from the web-based form input.**
(Filename: ch2_webform_input.xml)

```
<?xml version="1.0"?>
<submission>
  <first_name>John</first_name>
  <last_name>Smith</last_name>
  <address>1155 Perl Drive.</address>
  <city>Los Angeles</city>
  <state>CA</state>
  <zip>90102</zip>
  <phone_number>818-555-1212</phone_number>
</submission>
```

This wasn't a very complicated example. As you can see, we've mapped the web form names (for example, first name, last name, and so forth) to element names inside the XML document for consistency. This is a good example of when you would dynamically generate XML.

Now that we have discussed XML file generation, we can proceed to more complex issues, such as searching XML data and transforming XML into other data formats.

Searching XML Documents

Information storage is only useful if the information can be easily retrieved. Usually, information is retrieved from a relational database by sending commands (for example, Structured Query Language [SQL]) to the server. An XML document can be considered a facility for data storage, and therefore, can also be considered a rudimentary database. Just as SQL is the standard language for relational databases, the XML community has developed their own set of standards for retrieving information from an XML document. XPath is a standard that defines the syntax for addressing and fetching parts of an XML document.

Note

XPath and many other XML-related standards can be found on the web at <http://www.w3.org>.

Other XML standards exist, such as XML Query and XML Query Language (XQL), for querying XML documents. Each of these standards for querying XML documents uses its own syntax (though it might be similar) on top of the query engine.

Being able to extract data from an XML document can be very convenient. You can easily retrieve information from a certain part of the document by writing a query using XPath, XML Query, or XQL. Let's take a look at Figure 2.4, which shows (at a high level) just how XPath's query engine works.

As you can see in Figure 2.4, the application starts the query engine, initializes the query, and submits it to the engine for processing. The query engine then determines what is requested and goes to work to retrieve the requested information. After the data is located, it is returned to the application, which can proceed to process it. As you can see, the concept of the process is very simple and straightforward.

One area of XML processing that uses XPath quite extensively is XSLT. XPath is used to find portions of the XML document that match the criteria defined in the eXtensible Stylesheet Language (XSL) files (that is, which elements in the XML document will be extracted to create the output file).

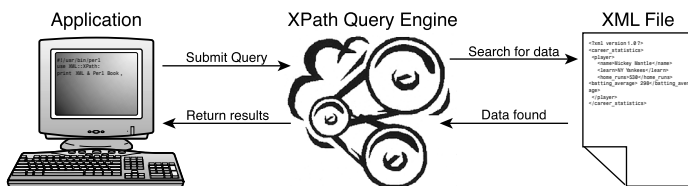


Figure 2.4 XML XPath Query Engine.

Transforming XML Documents

Now that we've covered parsing, generating, and searching an XML document, we get into another very important part of XML processing—XML transformation. You can transform, or in other words convert, an XML document to another format by applying various transformation techniques. You can transform the XML document to either another XML document, which might be a subset of the original document by filtering certain info, or you can transform to a whole different data format, like HTML, eXtensible HyperText Markup Language (XHTML), Wireless Markup Language (WML), CSV, and so forth.

Why would you want to convert our neatly formatted XML document to another format? We can read and understand the data as-is. Well the answer is, although as a technically savvy individual you took the time to learn XML and are now able to read it, most people would prefer their data represented in a user-friendly format.

One of the most popular methods of transforming XML to another format is to use the eXtensible Stylesheet Language for Transformations (XSLT) standard. An XSLT processor provides the interface needed to transform XML data. One of the processor's components is the XML parser itself, which is needed for the XSLT processor to understand and interact with XML. This makes sense if you think about it—the XSLT processor needs to parse the document to extract the information contained in the XML before it can format the contents. The processor requires an XSLT stylesheet to perform the transformation. An XSLT stylesheet is itself an XML document that contains embedded rules used to perform the transformation. An XSLT processor uses XPath and the rules contained in the XSLT stylesheet to transform the document. It searches through the XML data using XPath and tries to find the desired elements.

Note

We will come back to XML transformations and cover them in greater detail in Chapter 8, "XML Transformation and Filtering."

One of the most common (and easily demonstrated) transformations in use today is from XML to HTML. While the data is transferred and processed in a nicely formatted XML document, the XML document must be transformed into HTML to display in the browser with any type of formatting.

We don't want you to have to wait until Chapter 8 to see an example using an XSLT processor, so let's take a look at a simple example of transforming an XML document to HTML. To provide a common starting point, we'll reuse the XML document we discussed earlier, and it is presented here again as Listing 2.3 for your convenience. For this example, let's say that your task is to convert an XML report into HTML, so it can be displayed on a web page. Listing 2.3 shows the XML document we would like to convert and display in HTML.

Listing 2.3 Career statistics for two Hall of Fame baseball players stored in an XML document. (Filename: ch2_baseball_stats.xml)

```
<?xml version="1.0"?>
<career_statistics>
  <player>
    <name>Mickey Mantle</name>
    <team>NY Yankees</team>
    <home_runs>536</home_runs>
    <batting_average>.298</batting_average>
  </player>
  <player>
    <name>Joe DiMaggio</name>
    <team>NY Yankees</team>
    <home_runs>361</home_runs>
    <batting_average>.325</batting_average>
  </player>
</career_statistics>
```

If you recall, the previous XML file contains career statistics for two of the greatest baseball players ever to play the game. The data for each player includes the following elements: `<name>`, `<team>`, number of `<home_runs>`, and career `<batting_average>`.

As mentioned earlier, to perform the transformation, the XSLT processor requires two inputs: the XML document and an XSLT stylesheet. The XSLT stylesheet contains formatting information that determines both the content and format of the output HTML file. For example, if you wanted the text to be in a certain color or font, or if you want the information to be displayed in a table with particular column headings, this information would appear in the stylesheet. A sample XSLT stylesheet is shown in Listing 2.4.

Listing 2.4 XSLT stylesheet used to generate an output HTML file.
(Filename: ch2_baseball_stats.xslt)

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=http://www.w3.org/1999/XSL/Transform
version="1.0">
<xsl:output omit-xml-declaration="yes"/>
<xsl:template match="career_statistics">
<html>
<body>
<h2>Baseball Players</h2>
  <table border="1">
<xsl:for-each select="player">
<tr>
  <td><xsl:value-of select="name"/></td>
  <td><xsl:value-of select="team"/></td>
  <td><xsl:value-of select="home_runs"/></td>
  <td><xsl:value-of select="batting_average"/></td>
</tr>
</xsl:for-each>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

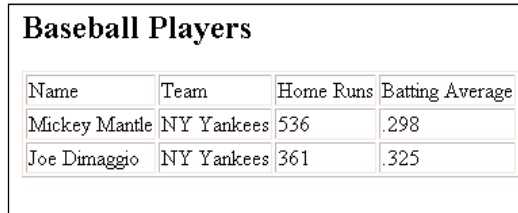
When we apply our custom XSLT stylesheet to the original XML document, an HTML file is generated that contains the data from the XML file, presented in a customized view. Remember, XML contains only data with no information about the formatting, so we're free to format it as we see fit. That's one of the best features of XML—you can use several XSLT stylesheets to transform the same XML source document into different output HTML files. For example, let's say that your company's annual report is stored in XML and that you are responsible for generating different HTML versions of it. One version of the report would be used for internal purposes, and the other version would appear on the public web site. The internal version may contain proprietary information that shouldn't be displayed on the public web site. In order to generate different HTML versions of the annual report from a single XML document, just define one stylesheet for each target report. The XSLT stylesheet that generates the public version can easily filter out any proprietary information.

After applying the stylesheet in Listing 2.4 to the XML file, the XSLT processor generated the HTML output shown in Listing 2.5. Don't worry about the underlying transformation technology at this point; we just wanted to illustrate the capability of transforming XML into other formats. This is covered in detail in Chapter 8, "XML Transformation and Filtering."

Listing 2.5 **HTML document generated by using Perl and XSLT.**
(Filename: ch2_baseball_stats.html)

```
<html>
<body>
<h2>Baseball Players</h2>
  <table border="1">
    <tr>
      <td>Name</td>
      <td>Teams</td>
      <td>Home Runs</td>
      <td>Batting Average</td>
    </tr>
    <tr>
      <td>Mickey Mantle</td>
      <td>NY Yankees</td>
      <td>536</td>
      <td>.298</td>
    </tr>
    <tr>
      <td>Joe DiMaggio</td>
      <td>NY Yankees</td>
      <td>361</td>
      <td>.325</td>
    </tr>
  </table>
</body>
</html>
```

Now that we have transformed the original file to HTML, it can easily be displayed in a web browser as shown in Figure 2.5. As you can see, the end result is a nicely formatted HTML table.



Name	Team	Home Runs	Batting Average
Mickey Mantle	NY Yankees	536	.298
Joe Dimaggio	NY Yankees	361	.325

Figure 2.5 HTML document generated by an XSLT transformation.

Our First Perl XML Programs

This section presents two Perl programs (including source code) using several Perl XML modules. We will demonstrate some of the topics previously discussed (XML generation, parsing, and transformation) in Chapter 1, “Basics of XML Processing in Perl,” and apply them to a simple, real-life application. We know that you’ve been patiently waiting for this moment. After all, you purchased this book to learn how to develop programs with Perl and XML, right? Well, since you asked for it, here it is.

Note

To run the programs discussed in this section, you need to install the following Perl modules:

- XML::Simple
- Data::Dumper

For Perl module installation instructions please see Appendix B, “Perl Essentials.”

The Problem: Generating and Parsing an XML Document

The two most common XML programming tasks are XML document generation and XML document parsing. In this section, we will present two applications that demonstrate how to perform these tasks. Both of these tasks are commonly performed during XML processing and form the basis for more advanced applications throughout the book.

The first program reads a CSV file that contains user information and transforms it into XML. The second program parses the XML file that was generated by the first program, retrieves the user information, and transforms it into HTML for display in a web browser.

Generating an XML Document

Let’s assume you are given the task of converting the information in an electronic mailing list from CSV to XML because a new email application supports only XML input. The fields in the CSV file will be formatted as follows, with each user record on a separate line:

```
First Name, Last Name, Email Address
```

Simple enough? That's why we decided to use the Perl XML::Simple module. It's designed especially for simple tasks, although it can be used for more complex tasks. The rest of the book focuses on other Perl XML modules that are better suited for more difficult tasks. After you become familiar with the other XML modules, you'll be able to make a sound judgment and pick the right tool for the task at hand.

Note

Additional information about the XML::Simple Perl module can be found by typing `perldoc XML::Simple`. Also, XML::Simple will be discussed in greater detail in Chapter 4, "Tree-Based Parser Modules."

Now that you have an understanding of the program's requirements, you can really get your hands dirty. The CSV is a simple file with only three fields: first name, last name, and email address. The input CSV file is shown in Listing 2.6.

Listing 2.6 **The Input CSV file containing source data for XML document.**
(Filename: `ch2_user_email.csv`)

```
Ilya,Sterin,isterin@cpan.org
Mark,Riehl,mark_riehl@hotmail.com
John,Smith,j.smith@xmlproj.com
```

Granted, this is a very small input file because it only has three records. However, the techniques used in this example can be applied to input data files with 3,000 or 30,000,000 records.

Now that we have the format of the input information, let's take a quick look at the required format of the output XML file. Let's assume that the output file is required to have the following format:

```
<users>
  <user>
    <first_name> </first_name>
    <last_name> </last_name>
    <email_address> </email_address>
  </user>
  ...
</users>
```

Note that we have a `<users>` element at the root of the document that is made up of multiple `<user>` elements. Each `<user>` element has one `<first_name>`, `<last_name>`, and `<email_address>` element.

The data for each user is stored in a predefined order: <first_name>, <last_name>, <email_address>. Let's take a minute and list the steps that we would need to perform if we were doing this conversion by hand.

1. Read each line of the input file.
2. Parse the line into columns based on location of the delimiters.
3. Print each column surrounded by the proper start and end tags.

Because this is such a simple example, this is exactly how our program will work. Listing 2.7 shows the Perl program used to parse the CSV file and generate the desired output XML document. This example doesn't use any XML modules, it just parses the input file and then creates an XML document using the Perl print function. The example is explained in detail in the following sections.

Listing 2.7 Perl application that converts the CSV input file to XML.
(Filename: ch2_csv_to_xml_app.pl)

```

1. use strict;
2.
3. # Open the ch2_xml_users.csv file for input
4. open(CSV_FILE, "ch2_xmlusers.csv") ||
5.     die "Can't open file: $!";
6.
7. # Open the ch2_xmlusers.xml file for output
8. open(XML_FILE, ">ch2_xmlusers.xml") ||
9.     die "Can't open file: $!";
10.
11. # Print the initial XML header and the root element
12. print XML_FILE "<?xml version='1.0'?>\n";
13. print XML_FILE "<users>\n";
14.
15. # The while loop to traverse through each line in users.csv
16. while(<CSV_FILE>) {
17.     chomp; # Delete the new line char for each line
18.
19.     # Split each field, on the comma delimiter, into an array
20.     my @fields = split(/,/ , $_);
21.
22.     print XML_FILE<<"EOF";
23.     <user>
24.         <first_name>${fields[0]}</first_name>
25.         <last_name>${fields[1]}</last_name>
26.         <email_address>${fields[2]}</email_address>
27.     </user>
28. EOF
29. }
30.

```

```

31. # Close the root element
32. print XML_FILE "</users>";
33.
34. # Close all open files
35. close CSV_FILE;
36. close XML_FILE;

```

1–9 The opening section of the program starts with the `use strict` pragma. This is a good practice in all Perl programs. The file `ch2_xmlusers_csv` is opened for input and `ch2_xmlusers.xml` is opened for output. After both files have been opened, we print the XML declaration and the opening tag for the `<users>` elements to the output file by using the `XML_FILE` filehandle.

```

1. use strict;
2.
3. # Open the ch2_xml_users.csv file for input
4. open(CSV_FILE, "ch2_xmlusers.csv") ||
5.     die "Can't open file: $!";
6.
7. # Open the ch2_xmlusers.xml file for output
8. open(XML_FILE, ">ch2_xmlusers.xml") ||
9.     die "Can't open file: $!";
10.
11. # Print the initial XML header and the root element
12. print XML_FILE "<?xml version='1.0'?\n";
13. print XML_FILE "<users>\n";

```

Note

In your program, `use strict` enforces error checking, and `use diagnostics` expands the diagnostics if the Perl compiler encounters an error or any questionable constructs.

15–29 This portion of the program performs a majority of the work in this example. The input file (`ch2_xmlusers.csv`) is read line by line within the `while` loop, and each line is broken up into columns by the delimiter, which in our case is a comma. The columns are then printed to `ch2_xmlusers.xml`, surrounded by the appropriate tags. To print this example, we're using a Perl “Here” document, which allows us to print a block of text without having to use a print statement for each line. In this case, we're printing everything between `print XML_FILE<<"EOF"` and `EOF`. Note that variables are being evaluated inside this block. For additional information on Perl “Here” documents, please see `perldoc perldata`.

Because we know the order of the input columns (`first_name`, `last_name`, `email_address`), we can print the input contents directly to the output XML file. This procedure is repeated for each line (user) in the `ch2_xmlusers.csv` file, and a `<user>` element is created each time through the `while` loop.

```

15. # The while loop to traverse through each line in users.csv
16. while(<CSV_FILE>) {
17.     chomp; # Delete the new line char for each line
18.
19.     # Split each field, on the comma delimiter, into an array
20.     my @fields = split(/,/, $_);
21.
22.     print XML_FILE<<"EOF";
23.     <user>
24.         <first_name>$fields[0]</first_name>
25.         <last_name>$fields[1]</last_name>
26.         <email_address>$fields[2]</email_address>
27.     </user>
28. EOF
29. }

```

31–36 The final section of our Perl application just prints the closing root `</users>` tag to the output file. Note that the opening and closing `<user>` element tags were outside the `while` loop. The code inside the `while` loop creates each `<user>` element.

```

31. # Close the root element
32. print XML_FILE "</users>";
33.
34. # Close all open files
35. close CSV_FILE;
36. close XML_FILE;

```

The Perl application will generate the output that is shown in Listing 2.8.

Listing 2.8 **Generated XML file containing the data from the input CSV file. (Filename: ch2_xmlusers.xml)**

```

<?xml version="1.0"?>
<users>
  <user>
    <first_name>Ilya</first_name>
    <last_name>Sterin</last_name>
    <email_address>isterin@cpan.org</email_address>
  </user>
  <user>
    <first_name>Mark</first_name>
    <last_name>Riehl</last_name>
    <email_address>mark_riehl@hotmail.com</email_address>
  </user>
  <user>
    <first_name>John</first_name>
    <last_name>Smith</last_name>
    <email_address>j.smith+xmlproj.com</email_address>
  </user>
</users>

```

As you can see in Listing 2.9, the XML file that was generated by our program matches the required output file format. Don't worry, there are better ways to generate an XML file. We won't always be using the Perl `print` function to generate XML files. The goal was to illustrate the concept of generating an XML document using another file (in this case, CSV) as the input data source.

Now that we've generated our first XML file, how do we parse it? That is discussed in the next section.

Parsing an XML Document

We've accomplished the first part of the application that showed the basics of generating a simple XML document. Now we can move on to the next task in which we transform the `ch2_xmlusers.xml` file in Listing 2.9 into an HTML file, so we can display the contents of the XML document in a browser.

Note

This chapter demonstrates the very basic concepts of XML generation, parsing, and transformation. Quite a few XML-related Perl modules are available to deal with each of these tasks. You'll see the benefits of the XML-related modules and when to use them (as well as when not to) as we move ahead and discuss more complicated examples.

Converting the input file from CSV to XML provides us with a document that is in a structured, clear, and easy-to-read format. Because XML is self-describing (because of the element names), anyone can look at the contents of an XML document and instantly see the layout of the data as well as the relationships between different elements. To display this information to someone not familiar with XML (or just for nicer formatting into tables), we need to convert it to another format. For this example, we have decided that we want this XML document to be displayed in a browser, so we must convert the XML document to HTML.

Note

A majority of the web browsers available today are able to display HTML in a consistent way. However, XML support among browsers isn't consistent and varies greatly among the most portable browsers (for example, some browsers will display XML whereas others require stylesheets). At the time of this writing, the most portable way to display an XML document with a browser is to convert the XML document to HTML.

To transform the XML document from XML to HTML, we need to parse it first. Using the `XML::Simple` module, this task is easily accomplished with its `XMLin` function, which reads XML from a file (string or an IO object), parses it, and returns a Perl data structure containing the information that was stored in XML. We can then traverse through the data structure, retrieve the required values, and print out the HTML file.

Additional XML::Simple Information

Note that there is also an `XMLout` function provided by `XML::Simple`. This function performs the reverse function of `XMLin` and returns XML based on the provided Perl data structure. Therefore, if `XMLout` is fed the same, unmodified data structure returned by `XMLin`, it will generate the original XML document. One potential issue is that the returned XML will be semantically alike, but not necessarily the same. This is due to the way Perl stores hashes, which is in no particular order. It's not necessarily a problem because many applications retrieve the data randomly and have no particular use for the element order, but it might pose a problem to applications that require a particular sequential order of elements (for example, sorted by a particular element). For additional information on `XML::Simple`, please see `perldoc XML::Simple`.

A number of attributes can be set to control `XML::Simple`'s behavior. For example, we will use the `forcearray` attribute, which forces nested elements to be output as arrays, even if there is only one instance present. Also, we'll use the `keeproot` attribute, which causes the root element to be retained by `XML::Simple` (it is usually discarded).

Now that we have a high-level understanding of the `XML::Simple` module, let's take a detailed look at the program shown in Listing 2.9 that converts the input XML file to HTML.

Listing 2.9 Program that converts an input XML file to HTML.
(Filename: `ch2_xml_to_html_app.pl`)

```

1.  use strict;
2.
3.  # Load the XML::Simple module
4.  use XML::Simple;
5.
6.  # Call XML::Simple's new constructor function to create an
7.  # XML::Simple object and get the reference.
8.  my $simple = XML::Simple->new();
9.
10. # Read in ch2_xmlusers.xml and return a data structure.
11. # Note that forcearray and keeproot are both turned on
12. my $struct = $simple->XMLin("./ch2_xmlusers.xml", forcearray => 1,
    └─keeproot => 1);
13.
14. # Open ch2_xmlusers.html file for output
```



```

15. # It will create it if it doesn't exist
16. open(HTML_FILE, ">ch2_xmlusers.html") ||
17.     die "Can't open file: $!\n";
18.
19. # Print the initial HTML tags
20. print HTML_FILE "<html>\n<body>\n";
21.
22. # The for loop traverses over each user.
23. # $_ points to the user
24. for (@{$struct->{users}->[0]->{user}}) {
25.     # Print the each field in the user structure
26.     print HTML_FILE <<"EOF";
27.     First Name: $_->{first_name}->[0]<br>
28.     Last Name: $_->{last_name}->[0]<br>
29.     Email Address: $_->{email_address}->[0]<br>
30.     .....<br>
31. EOF
32. }
33.
34. # Print the ending HTML tags
35. print HTML_FILE "</body><br></html>";
36.
37. # Close ch2_xmlusers.html
38. close (HTML_FILE);

```

1–12 This initial block of the program starts with the standard `use strict` pragma. Because we're using the `XML::Simple` module, we must also include the `use XML::Simple` statement to load the `XML::Simple` module. After these initialization calls, we create a new `XML::Simple` object and then parse the XML file by calling the `XMLin` function. The scalar `$struct` contains a reference to a data structure that contains the information from the XML file.

```

1. use strict;
2.
3. # Load the XML::Simple module
4. use XML::Simple;
5.
6. # Call XML::Simple's new constructor function to create an
7. # XML::Simple object and get the reference.
8. my $simple = XML::Simple->new();
9.
10. # Read in ch2_xmlusers.xml and return a data structure.
11. # Note that forcearray and keeproot are both turned on
12. my $struct = $simple->XMLin("./ch2_xmlusers.xml", forcearray => 1,
    ↪keeproot => 1);

```

19–38 The second half of the program actually performs all the work. First, we open the output HTML file and print the initial required HTML tags. Then, we loop through an array of references to data structures that contain the information from the XML file. The information from the XML file is then printed to the HTML file in the proper order.

We're taking advantage of the "Here" document construct again. This way, we don't need multiple print statements. Note that each time through the `for()` loop corresponds to one row in the CSV file and one `<user>` element in the XML document. After looping through the array, we print the required closing tags to the HTML file. The resulting HTML file is shown in Listing 2.10.

Listing 2.10 **Contents of the dynamically generated HTML file.**
(Filename: `ch2_xmlusers.html`)

```
<html>
<body>
First Name: Ilya<br>
Last Name: Sterin<br>
Email Address: isterin@cpan.org<br>
.....<br>
First Name: Mark<br>
Last Name: Riehl<br>
Email Address: mark_riehl@hotmail.com<br>
.....<br>
First Name: John<br>
Last Name: Smith<br>
Email Address: j.smith+xmlproj.com<br>
.....<br>
</body>
</html>
```

This HTML document is shown in a browser in Figure 2.6. As you can see, we didn't apply any formatting to this document, just the required headers and footers so that it displays properly. Examples later in the book will demonstrate how to dynamically generate HTML documents containing tables of data.

```

First Name: Ilya
Last Name: Sterin
Email Address: istryn@cpan.org
-----
First Name: Mark
Last Name: Riehl
Email Address: mark_riehl@hotmail.com
-----
First Name: John
Last Name: Smith
Email Address: j.smith@xmlproj.com
-----

```

Figure 2.6 The `ch2_xmlusers.html` file displayed in a browser.

Viewing the Contents of Data Structures

This section discusses a Perl module that allows us to visualize the data structure that is returned by the `XML::Simple` function `XMLin`. All we need is a simple script and the Perl `Data::Dumper` module. The Perl `Data::Dumper` module is very useful in situations like this, when we want to verify that we understand a particular data structure. It prints out the contents of a particular data structure in such a way that it enables us to see the hierarchy and relationships inside the data structure.

What we're going to do is follow the same steps of the last example: instantiate an `XML::Simple` object and then pass it the name of the input XML document (`ch2_xmlusers.xml`). The XML document will be parsed by the `XML::Simple` module and stored in a Perl data structure. At this point, we'll take a look at the contents of the Perl data structure by using the Perl `Data::Dumper` module. The Perl application that performs these steps is shown in Listing 2.11.

Note

For additional information the Perl `Data::Dumper` module, look at `perldoc Data::Dumper`.

Listing 2.11 Perl application that uses the `Data::Dumper` module to visual a complex data structure. (Filename: `ch2_data_dumper_app.pl`)

```

1. use strict;
2. use XML::Simple;
3. use Data::Dumper;
4.
5. my $simple = XML::Simple->new();
6. my $struct = $simple->XMLin("./ch2_xmlusers.xml", forcearray => 1, keeproot
   => 1);
7.
8. # Use Data::Dumper Dumper function to return a nicely
9. # formatted stringified structure
10. print Dumper($struct);

```

1–10 This program is basically the same as the last example with a few exceptions. Note that we need the `use Data::Dumper` pragma to load the `Data::Dumper` module. After creating a new `XML::Simple` object, we parse the XML document using the `XMLin` function. A Perl data structure that contains the parsed XML document stored in a Perl data structure and that is named `$struct` is returned.

All we need to do is pass the Perl data structure to the `Dumper` function that is provided by the Perl `Data::Dumper` module. That's all there is to it. The output is a nicely formatted report that can be used to study the data structure. The output from the `Data::Dumper` module is shown in Listing 2.12. Notice that the output from `Data::Dumper` is a mirror of the input XML file.

Listing 2.12 Output from the `Data::Dumper` module showing the hierarchy of the parsed XML file from `XML::Simple`. (Filename: `ch2_data_dumper_output.txt`)

```

$VAR1 = {
  'users' => [
    {
      'user' => [
        {
          'first_name' => [
            'Ilya'
          ],
          'email_address' => [
            'isterin@cpan.org'
          ],
          'last_name' => [
            'Sterin'
          ]
        }
      ],
      {
        'first_name' => [
          'Mark'

```

```

    ],
    'email_address' => [
        'mark_riehl@hotmail.com'
    ],
    'last_name' => [
        'Riehl'
    ]
},
{
    'first_name' => [
        'John'
    ],
    'email_address' => [
        'j.smith+xmlproj.com'
    ],
    'last_name' => [
        'Smith'
    ]
}
]
}
};

```

As you can see, the Perl data structure follows the same structure as original XML document (which we would expect). We recommend using the `Data::Dumper` module often. It is useful in situations where you need to access data that is stored in a complex Perl data structure. By looking at the output from the Perl `Data::Dumper` module, it will be easier to write the code to access different parts of the data structure and extract the desired information.

Summary

This chapter introduced XML processing and illustrated the many facets of XML processing—parsing, generation, and transformation. A brief introduction to each of these topics was presented in addition to several high-level examples. The examples were purposely made almost trivial to illustrate XML processing at a high level. Don't worry, the examples in the upcoming chapters become increasingly more difficult as we move on to more advanced topics.

For example, the next chapter, Chapter 3, “Event-Driven Parser Modules,” focuses solely on event-driven parsers. Detailed examples for several of the more popular event driven parsers are provided.

Exercise

1. Create a program using XML::Simple that will parse the following XML document shown below and convert it to a CSV formatted file. Using the same structure, change all first names to reflect only a first capitalized initial (for example, Ilya to I), then output this file to a new XML document.

```
<?xml version="1.0" ?>
- <users>
-   <user>
      <first_name>Ilya</first_name>
      <last_name>Sterin</last_name>
      <email_address>isterin@cpan.org</email_address>
    </user>
-   <user>
      <first_name>Mark</first_name>
      <last_name>Riehl</last_name>
      <email_address>mark_riehl@hotmail.com</email_address>
    </user>
-   <user>
      <first_name>John</first_name>
      <last_name>Smith</last_name>
      <email_address>j.smith+xmlproj.com</email_address>
    </user>
  </users>
```

For suggested solutions to this exercise, be sure to check out the web site:
<http://www.xmlproj.com/book/chapter02>.