
Java XML Application Categories

Author Name : Nazmul Idris
Created On : June 12, 1999
Modified On : June 13, 1999 12:58 pm

Table of contents

Introduction

Java Application Layer

3 Main categories

Client side - Graphical Java Applications

Client and Server side - Application Servers

Web-based Applications

API Coverage per category

Introduction

The applications that you create with Java and XML will rely on the services provided by your Java XML Parser (using DOM or SAX). The information itself might be stored in a variety of persistence engines (object databases, relational databases, file systems, dynamic websites, etc.). The information however that comes out of these persistence storage engines must be converted to XML (if they are not in XML already). Once this is done, you have to be concerned with the material covered in this document. This document outlines the most popular Java XML application categories that are possible in an environment where data is encoded with XML, where web access is ubiquitous and platform independence is a necessity.

Java Application Layer

All of the code that you write (in your Java classes) might be considered the Java application layer. Other layers are the XML Parser layer, the XML source (that supplies the XML data that is necessary), and the persistence engine (where the data is actually stored and retrieved by the source).

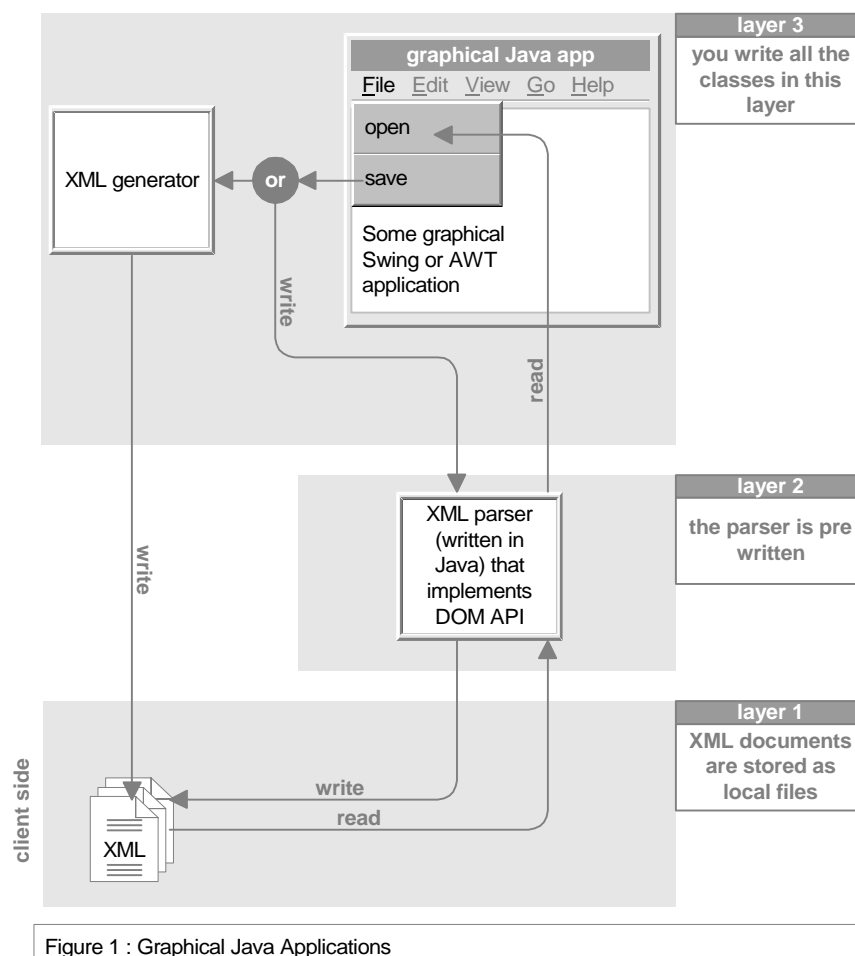
Your code (in the Java application layer) has to make use of the DOM or SAX API and the XML parser in order to access the information in XML documents (that come from your source). The source might be responsible for pulling data from different persistence engines (relational or object databases) and even the web (dynamically generated websites that supply only XML data).

In your application layer, you can create many interesting Java applications. The apps can run on the server side or client side or both. They may have graphical user interfaces or they may be web based. When I use the word application or app in this chapter, I don't exclude Java applets; I mean application (or app) in the broad sense of the word, i.e., I mean it to describe a software system written in Java that solves a real-world problem.

3 Main categories

There are many different types of software that you can write in Java to make use of XML. I have created 3 major categories to describe certain types of apps (that are currently popular) that are really well suited to the use of XML. This is by no means a comprehensive set of categories; you can create your own, and many more major categories will emerge as XML becomes more popular.

Client side - Graphical Java Applications



The simplest category of XML Java applications is the kind of Java application that stores information in XML documents (files). This is illustrated in Figure 1. By using XML to create your own markup languages (i.e. your own file formats for your information) in an open way, you don't have to use proprietary and binary file formats. Using XML over proprietary binary file formats, allows your applications to have immense inter operability across platforms, applications and even programming languages. Since any kind of markup language can be defined using XML (you can even formalize it by creating a DTD for it) applications can store their information using

their own markup languages. For example, address book information can be stored in an AddressBookML file. A few commercial programs currently available allow saving their application data to XML files, e.g., Framemaker can save its documents as XML files.

In order to create applications of this category, you might have to define a DTD for your information. Then you have to write classes to import and export information from your XML document(s) (validating using your application's DTD if you have one). You must also write the classes which create the user interface in your application. The user of your application can view and modify information using the GUI (graphical user interface), and they can save (and load) their information to (and from) an XML file (that might use your DTD); in other words, they can save (and load) their information to (and from) an *ApplicationML* file (where *Application* is the name of your application). Some examples are AddressBookML, MathML, SVGML, etc.

The classes that import and export information from your ApplicationML file must use the parser and SAX or DOM API in order to import the information. These classes can access this information by using one of the following strategies:

1. Use DOM to directly manipulate the information stored in the document (which DOM turns into a tree of nodes). This document object is created by the DOM XML parser after it reads in the XML document. This option leads to messy and hard-to-understand code. Also, this works better for document-type data rather than just computer generated data (like data structures and objects used in your code).
2. Create your own Java object model that imports information from the XML document by using either SAX or DOM. This kind of object model only uses SAX or DOM to initialize itself with the information contained in the XML document(s). Once the parsing and initialization of your object model is completed, DOM or SAX isn't used anymore. You can use your own object model to accessed or modify your information without using SAX or DOM anymore. So you manipulate your information using your own objects, and rely on the SAX or DOM APIs to import the information from your ApplicationML file into memory (as a bunch of Java objects). You can think of this object model as an in-memory instance of the information that came was "serialized" in your XML document(s). Changes made to this object model are made persistent automatically, you have to deal with persistence issues (ie, write code to save your object model to a persistence layer as XML).
3. Create your own Java object model (adapter) that uses DOM to manipulate the information in your document object tree (that is created by the parser). This is slightly different from the 2nd option, because you are still using the DOM API to manipulate the document information as a tree of nodes, but you are just wrapping an application specific API around the DOM objects, so its easier for you to write the code. So your object model is an adapter on top of DOM (ie, it uses the adapter pattern). This application specific API uses DOM and actually accesses or modifies information by going to the tree of nodes. Changes made to the object model still have to be made persistence (if you want to save any changes). You are in essence creating a thin layer on top of the tree of nodes that the parser creates, where the tree of nodes is accessed or modified eventually depending on what methods you invoke on your object model.

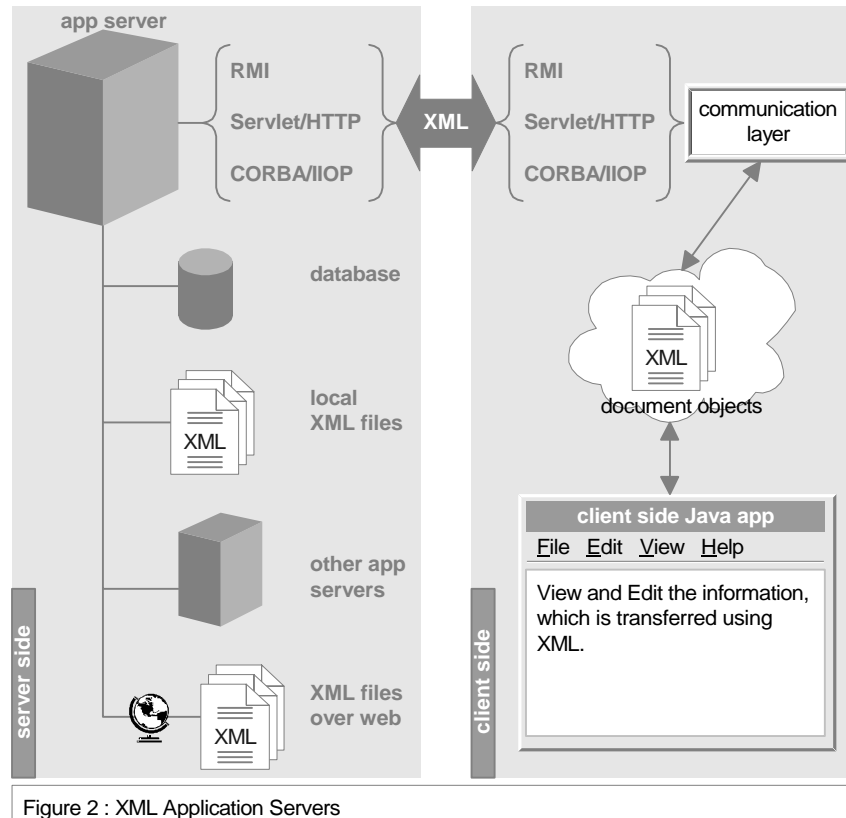
Depending on which of the three options you use to access information using your Java classes, this information must at some point be saved back to a file (probably to the one from which it was read). When the user of your application invokes a File->Save action, the information in the application must be written out to an *ApplicationML* file. Now this information is stored in memory, either as a (DOM) tree of nodes, or in your own proprietary object model. Also note that most

DOM XML parsers can generate XML code from DOM document objects (but its quite trivial to turn a tree of nodes into XML by writing the code to do it yourself). There are 2 basic ways to get this information back into an *ApplicationML* file:

- You can generate the XML yourself (from your object model). If you created an object model that simply imports information from your XML document (using SAX or DOM), you would have to write a class that would convert your object model into an XML file (or set of XML files). This class would have to create an *ApplicationML* file that contains the information in your Java object model (which is in memory). Since this object model is not an adapter on top of DOM, it is not possible to use the DOM parser to generate the XML for you.
- You can use the DOM parser to generate the XML for you if you created an object model that is an adapter on top of DOM. Since your object model uses the document object tree, all the information contained in it is actually stored in the tree. The XML parser can take this tree and convert it to XML for you, you can then save this generated XML to a file. So the DOM parser can generate the *ApplicationML* file for you.

There are advantages and disadvantages to using some of the strategies to import and export XML. The complexity of your application data and available system resources are factors that would determine what strategy should be used.

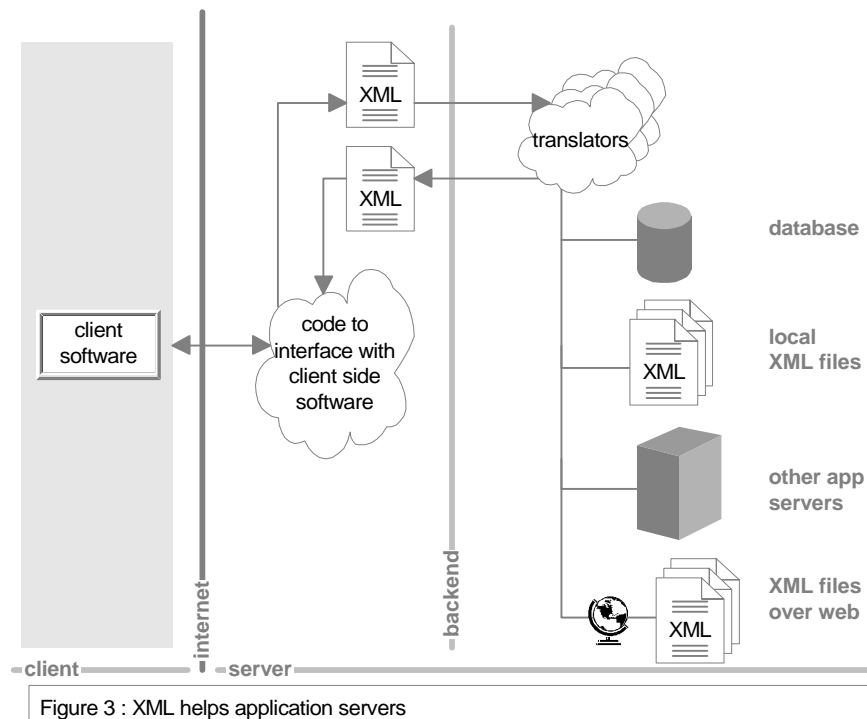
Client and Server side - Application Servers



The 2nd category of Java applications called Java Application Servers (or app servers) and they make good use of XML. Unlike client side graphical Java apps (from the previous section) which are very standalone in their operations, app servers tie many different networked software components together in order to provide information from multiple sources to a set of client side Java apps or web browsers (maybe even running on different devices). This is shown in Figure 2. An app server is actually a conglomeration of several distributed and client/server software systems. So when you write an app server, you are actually writing many different software systems which are all networked to work together, to process information that comes from various sources, and distribute this information to a set of client apps (that you also have to write) running on different devices and platforms.

How can XML help app servers do their work? As you can see in Figure 2, in order for the app server to harvest information from such a rich variety of sources, there must be some common ground between all of these sources (each of which might be running on a different hardware and software system). This common ground is the information which flows throughout the entire system, regardless of what source the information comes from. CORBA is an example of tying disparate systems together based on the interfaces that certain remote objects implement. XML does the same thing for data. It allows these disparate systems to share information in a medium that consists only of pure information (and the structural relationships that exist inside of that information). By taking the lowest common denominator approach by using plain text to encode data,

XML allows these systems to talk with each other without requiring any special binary information format converters or other service layers to translate between binary formats (for encoding data). Also, since HTTP already supports transmission of plain text, it is completely natural to move XML around using the Hyper Text Transfer Protocol through firewalls and disparate networks. This is shown in Figure 3. XML can be transmitted between systems using one of the most prevalent protocols in use today, Hypertext Transfer Protocol or HTTP 1.1 (which is the protocol of the web).



App server developers are not restricted to using HTTP, they can transmit and receive XML information using simple remote CORBA objects and RMI objects. The key is that by using XML, it makes these remote services or objects easier to build. And, by sticking with XML, any one of these technologies can be used in your design of your app server. You can use whatever technology is most appropriate to getting the job done, knowing that all the information flows as XML and can be processed by any part of the system. The reason Java object serialization did not achieve this is because it encodes object data to a binary format that is dependent on too many things (like the JVM version, and the existence of classes when things are deserialized, etc). XML is not limited by any of these restrictions (or problems), which makes it much easier to create systems that allow XML information to flow between different subsystems. Also by relying only on the data, large portions of the system can be replaced with better or different implementations for future-readiness.

App servers traditionally give their client apps access to information in remote databases, remote file systems, remote object repositories, remote web resources, and even other app servers. All these information sources don't even need to reside on the machine that hosts the app server. These remote resources may be on other machines on the Intranet or the Internet. Using Java and

XML, RMI, JDBC, CORBA, JNDI, Servlet and Swing, you can create app servers that can integrate all kinds of remote and local information resources, and client apps that allow you to remotely or locally access this information from the app server.

In the future, with publicly available DTDs that are standardized for each vertical industry, XML based app servers will become very popular. Also when XML schema repositories become available and widely used, app servers will be able to take on a new role and provide application services that are not offered now. Companies will need to share information with other companies in related fields, and each company might have a different software system in which all their data is housed. By agreeing upon a set of DTDs or schemas (encoded in XML), these companies can exchange information with each other regardless of what systems they are using to store this information. If their app servers can exchange XML documents (based on some shared DTD or schema), then these disparate app servers can understand each other and share information. One of the uses for XML foreseen by the W3C is just this, vertical industries (like insurance and health care) creating sets of DTDs and schemas that all companies in the industry agree upon. Then these companies' app servers can talk to each other using some popular protocol (like HTTP or CORBA/IIOP) to exchange information between each other. This has the potential to save a lot of time and money in the daily business operations of these companies.

Web-based Applications

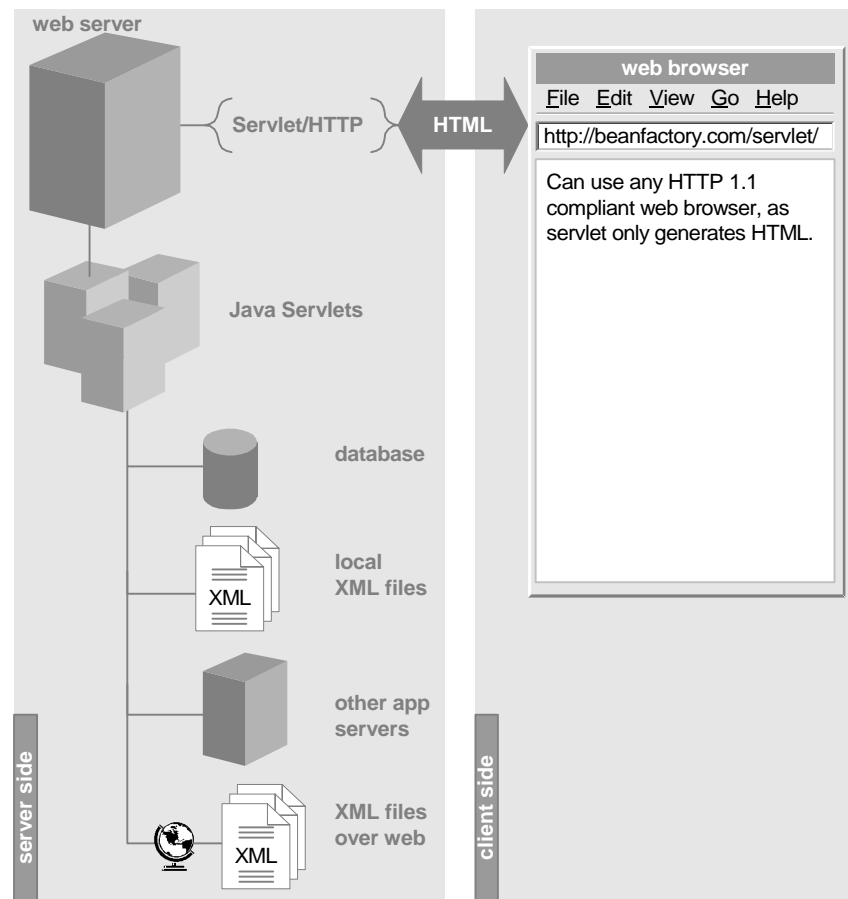


Figure 4 : Web-based XML applications

Web-based applications are similar to app servers, except for one thing: Web-based applications don't have client apps, instead they use web browsers on the client side. They generate their front ends using HTML, which is dynamically generated by the web-based app. In the Java world, Servlets are best suited for this job.

Web-based apps might themselves rely on another app server to gather information that is presented on the client web browser. Also, you can write Servlets that get information from remote or local databases, XML document repositories and even other Servlets. One good use for web-based apps is to be a wrapper around an app server, so that you can allow your customers to access at least part of the services offered by your app server via a simple web browser. So web-based apps allow you to integrate many components including app servers, and provide access to this information over the web via a simple web browser.

Web-based apps are very deployable, since they don't require special Java VMs to be installed on the client side, or any other special plug ins, if the creator of the web-based app relies solely on HTML. Unfortunately, this can restrict the level of service that can be offered by a web-based app when compared to the functionality offered by custom clients of an app server, but they are a good

compromise when it comes to providing web-based access to your information. In fact, in a real world scenario, both a web-based app and app server may be used together, in order to provide your customers access to their information. In an Intranet setting, you might deploy the clients that come with the app server, and in an Internet setting it would be better to deploy a web-based app that sits on top of this app server, and gives your customers (relatively) limited access to their data over the web (via a simple web browser).

Web-based apps and app servers integrate very well, and this is another reason why Java and XML make a powerful combination for developing systems that give your customers access to their information from anywhere, using any browser over the web. In the future, you can imagine various different web-based apps servicing different kinds of clients, e.g. web browsers on desktops, web browsers on PDAs, and web browsers on all kinds of different consumer electronics devices. By keeping your information structured in a pure way (by using XML), and by allowing access to this information through app servers, you can write many different web-based apps that render this information by customizing it uniquely for each different device that is allowed access to this information. This is more a more scalable solution than storing all this information in web pages, even if these web pages are dynamically generated. So you can have one app server that stores all the data in XML format. You can write a web-based app (which sits on top of this app-server) that allows PalmPilots to access this information over the web. You can write another web-based app (that also sits on top of the same app server) that allows conventional web browsers to access this information over the web. XML and Java have the potential to make this truly platform independent and device independent computing a reality.

API Coverage per category

All these application categories sound very exciting, but what APIs are involved in implementing such systems? I have compiled a list of the most frequently used APIs used for each of these application categories. This list is by no means exhaustive or comprehensive, this list should be useful for most of your application development needs. Table 1 contains a list of APIs most likely needed for each application category.

Table 1: APIs most likely used in the 3 application categories

Application Category	APIs needed
Client side - Graphical Java Applications	W3C DOM or SAX API Java2 Core API Java Swing API
Client and Server side - Application Servers	W3C DOM or SAX API Java2 Core API Java Servlet API Java Swing API Java JDBC API

Table 1: APIs most likely used in the 3 application categories

Application Category	APIs needed
Web-based Applications	W3C DOM or SAX API Java2 Core API Java Servlet API Java RMI API Java JDBC API

In your projects, you might not use some of these APIs and you might use other APIs not listed in Table 1. I have not listed version numbers for any of the APIs in the table, because they might have changed by the time you read this document.