

Information in Motion: An XML Technologies Primer

Abstract

Organizations need to deliver more relevant and timely information to their customers and employees, while lowering the cost of application development and management. A family of new and enhanced XML standards is providing the foundation for a radical new approach to application development that challenges many of the conventional tools and technologies used by IT.

July 2011

Copyright © 2011 EMC Corporation. All Rights Reserved.

EMC believes the information in this publication is accurate as of its publication date. The information is subject to change without notice.

The information in this publication is provided “as is.” EMC Corporation makes no representations or warranties of any kind with respect to the information in this publication, and specifically disclaims implied warranties of merchantability or fitness for a particular purpose.

Use, copying, and distribution of any EMC software described in this publication requires an applicable software license.

For the most up-to-date listing of EMC product names, see EMC Corporation Trademarks on EMC.com.

All other trademarks used herein are the property of their respective owners.

Part Number h4706

Table of Contents

Executive summary	4
Audience.....	5
XML is information in motion	5
Querying data.....	7
Transforming data.....	11
Validating data	12
Application building with XForms	13
Pipelining XML.....	15
Dynamic Delivery Services.....	17
Conclusion.....	19
References	19

Executive summary

Organizations need to provide more relevant and timely information to their customers and employees, while lowering the cost of application development and management. A family of new and enhanced XML standards is providing the foundation for a radical new approach to application development that challenges many of the conventional tools and technologies used by IT. As infrastructure, platforms, and applications shift to the cloud, pressure will grow to adopt these innovative technologies and practices.

Starting from XML's ability to model any type of data and its role as both the definer and the payload of information services, the new approach to application building is independent of proprietary data stores that lock up your information. Instead of complex programming against application-specific data silos, XML becomes *information in motion* between services. By moving from applications to services, you lose a lot of the complexity required when you have different computer languages and programming environments, which in turn makes building "mash-ups" from different data streams possible. These same characteristics that allow XML to speed application development, also facilitate rapid application retirement. XML allows data from legacy systems to be archived efficiently, drastically reducing the cost of ownership associated with the maintenance of legacy systems, while maintaining the integrity of the data for compliance reasons in a future-proof format.

The key XML standards that enable this shift in IT thinking include:

- XQuery and XPath: for querying collections of XML content and constructing new information from the query results
- XSLT: for transforming data from one XML structure to another, or to an output format like HTML or PDF
- XML Schema and Schematron: for providing declarative and rules-based validation of XML data
- XForms: for building dynamic user interfaces for interacting with XML data
- XProc: for constructing pipelines, or sequences of steps that control the processing flow for applications. Although it's the newest of the aforementioned standards, XProc will have a significant impact as the "umbrella" that can be used to tie all the other standards together into a cohesive application development framework.

EMC® Documentum® Dynamic Delivery Services (DDS) is an application platform that puts the power of these XML standards in reach of any developer. DDS is the perfect vehicle for applying these standards to large, complex and specialized XML data sets, as well as for processing this content for both embedded Java and Rich Internet applications.

Audience

This white paper will be of interest to application developers, architects, and IT managers seeking to use standards-based technologies to reduce development costs while improving application functionality and performance.

XML is information in motion

The modern relational database has become ubiquitous everywhere from being the memory of large enterprise operations to supplying session state information for websites. Yet most such databases suffer from a fundamental flaw that is inherent in the relational model — they don't have any consistent way to serialize that data.

Serialization is a way to convert the data structures within a program to something that other programs (or people) can understand. For instance, if you had a spreadsheet (at least until fairly recently), the information in that spreadsheet was effectively locked inside of it, unless you exported it in some format (such as comma separated values, or CSV) that another program could then load. This export process was known as serialization, because it converted a complex structure into a serial, or sequential, stream of information.

The flip side of serialization is called parsing. When you parse content, you take a sequential stream of information and use it to build up a data structure. Every time you load a program from a hard drive, you are in effect parsing a stream of content that had been previously serialized in order to set up the data structures for the program to work properly.

When the Structured Query Language (SQL) was first introduced in the early 1980s (based upon ground-breaking work from Dr. Ted Codd on relational data theory) the assumption was made that the language, used to search, create and modify databases, applied only to what went on within the database itself. It was the database vendor's responsibility to provide tools to make the results of queries made by programmers against the database accessible to those programmers.

The result of this was that most vendors created fairly complex data interface layers that would let programmers talk to the database through their own custom objects, with each vendor doing so in a different way. While some standards did arise (ODBC, or its Java equivalent JDBC), the reality was that if you wanted to get information from a database, not only did you have to know how to write SQL, but you had to kidnap a programmer to turn those SQL queries into something useful, like a graph or table for an annual report. Furthermore, as applications aged and alternative processing models become more desirable to solve new and evolving business problems, the database acted as an anchor — effectively “locking” the organization into a proprietary data format that is expensive to maintain. The lock-in problem is even of greater concern as the persistence and management of information moves to the cloud — amplifying the need for standards-based, future-proof, and application-independent technologies such XML.

Even today, this same principle is pervasive. Languages such as Ruby have exploded in popularity in great part because they make getting to this data (somewhat) easier, but with relational databases in particular, getting useful content from a database still requires the intervention of expensive and often overworked programming professionals.

This is where XML comes into the picture. Imagine that you could capture a snapshot of the information that you needed in a format that everyone understood, then could send that snapshot to someone else, who could in turn use that information anyway he or she needed. They wouldn't need to have the same program that the information came from (or even a program made by the same vendor). They could take the information and extract out the relevant parts without having to hire someone who knew what the database itself looked like, and what's more they could potentially generate new content that could be sent back to the database without needing to have direct access to that database.

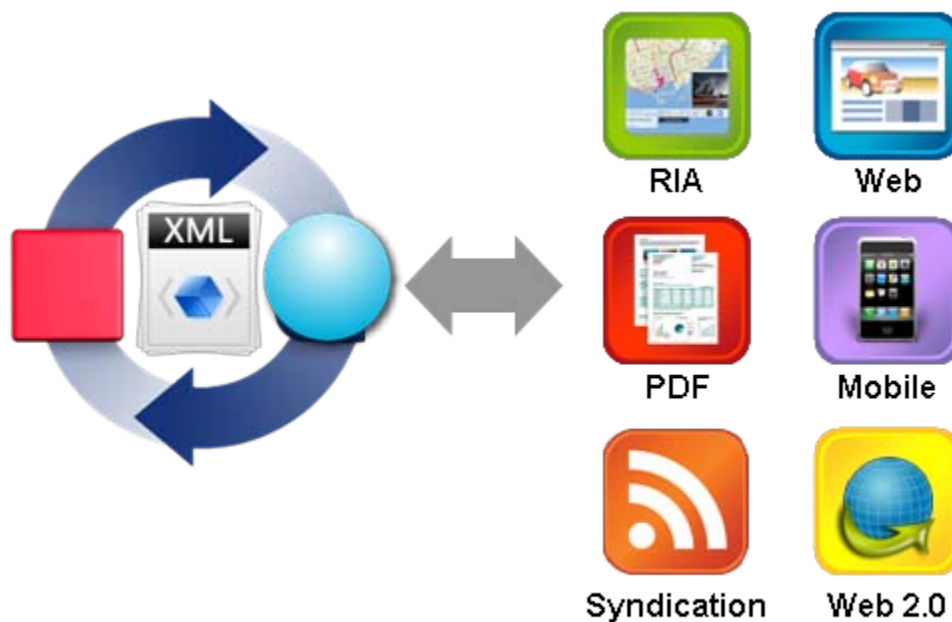


Figure 1. XML is "Information in Motion."

One interesting consequence of this is that the same effect holds true if the "database" in question is an application of a different sort — such as the aforementioned spreadsheet program. By presenting the output of the spreadsheet in an XML format, a consumer of that information needs only to know about the structure of the data, not specifically have access to that application. The application that created the data does not even need to be in production!

Put another way, one of the roles of XML is to abstract the data sources and data sinks, to make them less important in the overall scheme of things. This becomes especially true in the face of a second major innovation — the ability to refer to applications (and databases) as addresses. The essence of the Internet is that everything becomes addressable — you can assign a URL to an application, post content to that URL address, and retrieve content from that address. By taking this

approach, databases and data providers become far more abstract, as you no longer need to know specifically what technology sits behind the URL — only that the URL can provide you XML and can accept XML for updates and new content creation.

In this brave new world, the effect of such abstraction is that every address becomes a service, and XML becomes information in motion between such services. It's hard to overstate the importance of this shift in thinking — by moving from applications to services, you lost a lot of the complexity required when you have different computer languages and programming environments, which in turn makes building "mash-ups" from different data streams possible without having to worry about how those streams are produced or how they consume your stream of data. This is the real power of XML.

Querying data

Once you recognize what services give you, the next stage in that process comes in refining the information that you can get from a service. One way to think of a service, regardless of the technology behind it, is that it is the interface for a database. "A database of what?" is irrelevant here — it is more important just to see the service as being attached to that database in some manner. Normally, when you access the database, what you are doing is performing a query, and if you don't pass in some form of query information the query in question will use a set of default values for its parameters.

In either case, the result of this query is to pull out a subset of the overall database in some serialized form. That form doesn't necessarily have to be XML, by the way. Consider, for instance, using Google. When you perform a Google search, what you are in fact searching is a massive collection of web page abstracts (more or less — there's a huge amount of indexing that goes on there as well, but for sake of discussion consider it a giant database of abstracts). The query string in this case is the term or phrase that you're searching on, although it is also parsed to extract certain directives such as `site:`, which indicates that the URL fragment immediately following it should be used to filter the results to search only on the indicated website.

The query parameters that get passed to the search usually end up getting rewritten in some kind of internal query packet that is then passed to a program that uses that packet to retrieve links to those abstracts that fit the parameter, which are sorted according to a certain set criteria. These results in turn are partitioned into pages, both to keep the search from taking too much processing time on the server and to keep the results from taking too long to render on the browser.

Once partitioned, the page summaries are "rendered" as HTML entries with appropriate links, along with another "decoration" to round out the pages and add invisible metadata to the pages to make subsequent requests easier and faster.

Significantly, the same process also powers news or syndication feeds. The search engine performs a query upon the space, sorts the resulting subset, and partitions

the results, but instead of then "painting" the resulting set as a web page, it converts it into XML output (typically using either RSS 2.0 or Atom XML formats). This output can then be interpreted by simple news readers.

If the initial data that you have is in XML format (or can be readily converted into XML format) then the process becomes more interesting. For a small site, it's possible to actually have most of the relevant documents contained as files, in which cases the pipeline usually involves a process of performing loads and lookups on a sequence of files. However, beyond a certain size (and it doesn't take long to reach that stage) it is generally better to either have a relational database that can use table JOINS to create hierarchical documents, or, better to use an XML database itself. Such databases usually are optimized to both store and search XML content in a more efficient manner than pulling the information from relational tables.

However, until comparatively recently, there was no equivalent to the SQL used for relational databases in XML. This meant that each database had its own proprietary query language, most of which incorporated some variant of XPath in order to retrieve or query on individual documents. XPath is a language that describes the location of one or more blocks of information within an XML document. It's not by itself a full "language" for doing anything with that content; it only tells other programs where the information is and lets those other languages do the processing.

Despite it being "incomplete," XPath is actually remarkably useful. The ability to select "nodes" — locations in XML documents that contain tagged content — is critical for performing search on XML documents, but it is also useful for identifying what needs to be processed, and can also be extended to do some of that processing. Almost every other critical core XML standard makes use of XPath (the one exception being schema validation with the XML Schema Definition Language — XSD or XSDL 1.0 — and even that's changing in XSDL 1.1).

However, even given these capabilities, XPath 1.0 was principally designed to work on single documents. It would take another seven years and the realization that XPath (and associated technologies) needed to work beyond the document to the collection level before a new version, XPath 2.0, emerged in February 2007 from the W3C.

While XPath 2.0 looks much like 1.0 (and in general you can run 99.5 percent of all XPath 1.0 statements in 2.0), XPath 2.0 differs from its predecessor in two fundamental ways. First, it can work on sequences of "things" where those things could be strings (text), numbers, dates, or XML tags (elements) among other things. This change opened up XPath 2.0 to be able to work with multiple documents (that is, collections of documents) and not coincidentally it made possible the ability to create and convert XML structures on the fly and then process these — what's called intermediate processing, something which dramatically expanded what could be done with the language.

The second change seems minor, but is likely to have huge implications. The original version of XPath included a core set of a couple dozen functions. XPath 2.0 upped this number to 140, covering string and pattern matching, sequence functions, an

extensive date library, and more. Yet more significantly, XPath 1.0 didn't define a formal extension mechanism — each vendor was permitted to create their own extensions conventions, which meant that different vendors may end up developing dramatically different ways of extending the language. In XPath 2.0, on the other hand, a formal extension mechanism was defined, and it was one that could be implemented from any language (or multiple languages).

What this meant was that you could use Java to extend XPath to implement functions for determining if a latitude and longitude were within a geographic region, C# to implement functions for adding financial functions, or PHP to include functions for accessing or updating web server information. What's more, you could also use XSLT (the XML transformation language) or XQuery (the XML Query language) to create such modules. Because both languages are optimized for XML use, it is possible to create extensive XPath libraries that nonetheless can be made portable between different implementations.

The XML Query Language (XQuery or XQL) is a formal query language that can be used to perform queries against whole collections of XML documents and create new content from these queries. You can think of XQuery as being XPath with a few additional commands added in that make it possible to create new content from those selected queries.

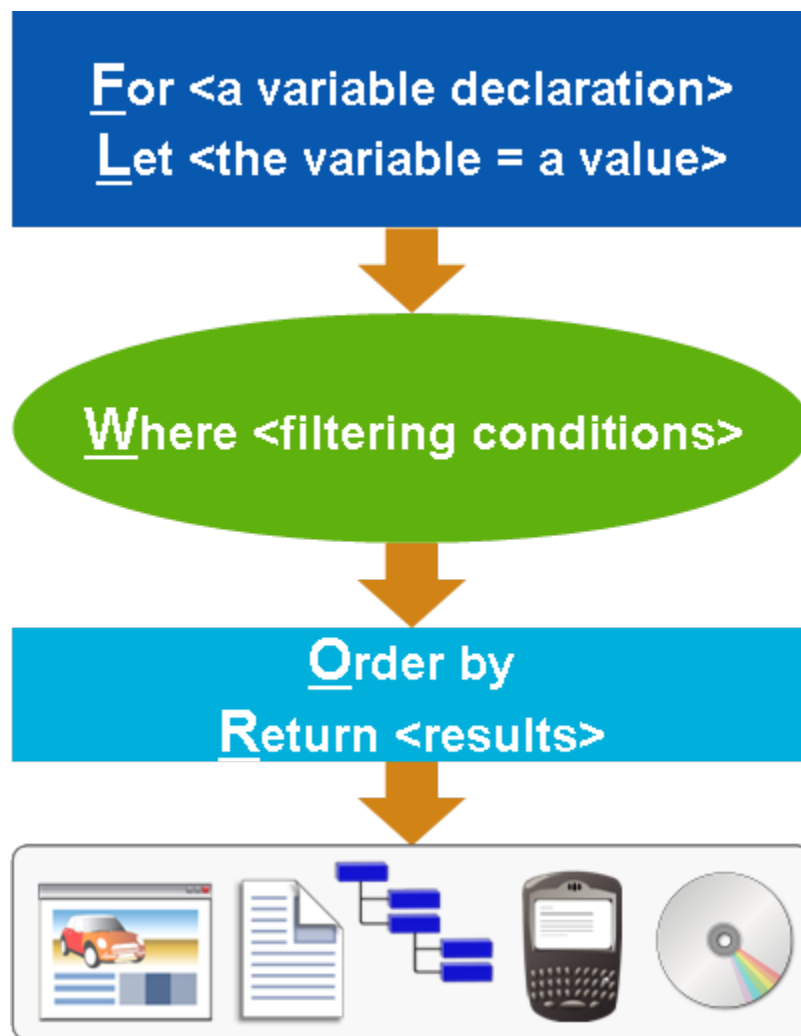


Figure 2. XQuery is a formal query language that combines XPath expressions with “SQL-like” operators for searching collections of XML documents.

For instance, you can perform the parse->query->sort->partition->render actions described with the Google example directly within an XQuery script. What's more, it's possible to wrap these in an XQuery function as part of a module, then later import the module and call a single function within the script to do this action.

The initial XQuery specification (W3C recommendation 1.0) covered the query side of the operation, and as such looks a lot like the SQL query statements such as SELECT combined with the power of stored procedures. XQuery itself does not provide a mechanism for updating content within a file or XML database, but a key extension to this specification — the XQuery Update Facility 1.0 -- is the standard currently recommendation by the W3C, and makes it possible to create new XML resources, and update parts of existing resources or delete resources. As such it provides a critical (standardized) piece to XQuery.

XQuery provides a way to refine information coming from a service without necessarily knowing what is being used to store that information. XML databases

obviously benefit directly from XQuery, as they combine XML searchable content with the notion of collections, but with the right extension support, it's certainly possible to create an XML representation of an e-mail box (and to be able to post mail by posting an XML "mail message" to a URL). Similarly, a SQL relational database query may be accessible from a URL as an XML structure, and updating the table could involve simply posting the right kind of XML representation to the server.

The key point here is that in both cases, the user need not know that the actions going on in the back end are anything other than services (or even virtual collections) being queried or added to. This powerful principle, known as RESTful Services, can be accomplished in other ways, but even reasonably complex RESTful services applications can usually be put together in a matter of a couple of hours, and can be modified or added to in functionality without forcing the whole system to be brought down.

Transforming data

XQuery is capable of creating new XML content from existing content (as well as being able to create "raw" output content without the need for some form of input). However, for fairly complex data, the process of doing such conversions can become quite extensive, especially if the output format is also complex. An alternative language, the XML Stylesheet Language for Transformations (XSLT), is often far better for handling these kinds of transformations than XQuery is.

XSLT is what's called a template language. What this means in practical terms is that a typical XSLT stylesheet consists of a collection of "templates," where each template has an XPath expression that it has to satisfy. When a document is passed to a stylesheet, the first node of the document is tested against the templates to find the best matching template, and this in turn either generates other XML content or contains additional instructions for which nodes should be tested next.

When XPath was updated to XPath 2.0, XSLT was similarly updated to XSLT 2.0. This means that just as XQuery can be used to define extension functions in separate modules, so too can XSLT be used to define extension functions in external stylesheet modules (and in cases where XSLT 2 and XQuery 1.0 are both functional, you can create stylesheets that can be invoked like functions from within XQuery expressions and vice-versa).

A consequence of this is that much of the role of XSLT and XQuery can be seen to overlap, as XQuery could be used to transform XML content while XSLT could be used to query content. In practice, however, the overlap is not as pronounced as it may seem in theory.

Typically transformations written in XQuery can only reach a certain level of complexity before they become unwieldy, while XSLT is a remarkably effective language for dealing with these more complex cases. On the flip side, XSLT's sometimes verbose syntax can definitely be overkill when trying to filter or sort

content, and there are few databases currently that let XSLT use the same efficiencies in retrieving content that XQuery can achieve.

Because of this, you should look upon XQuery and XSLT as processes that can feed into each other. For instance, XQuery may be used to query a database, sort the resulting content and partition it, while XSLT may be used to transform the result into the desired output. Similarly, an incoming XML document (such as a blog post) may be transformed by XSLT to add critical system data then stored in the database using XQuery.

Validating data

One of the key advantages of XML is the fact that a given XML document may very well have one or more schemas that can determine whether the XML is valid. Now, validity is something of a slippery concept, but in the simplest terms you can think of a valid XML document as being one that conforms to a given set of rules.

For instance, consider something like a simple purchase order rendered as XML. One schema form, the XML Schema Definition Language (XSDL, or occasionally XSD), indicates that an invoice consists of the name and address of the person sending the purchase order, the address of the company fulfilling the order, a set of line items, and a summary field for pre-tax cost, taxes, and total costs. Beyond indicating the structure of each of these pieces, the schema also indicates the "data-type" of individual fields in the XML (such as a percentage type for taxes, currency type for monetary items, strings (text) for titles and descriptions, and so forth).

A given XML document may also have more than one type of schema associated with it. XSDL doesn't handle relational constraints — while it can tell you that the `total_before_taxes` field has to be a number, there's nothing in XSDL that will specifically indicate that this field needs to be a sum of the totals of each line item. However, there's a second schema language called Schematron (an odd name but useful technology) that can in fact indicate that `total_before_taxes` has to be the same value as the sum of `line_item_total` items from each line.

Significantly, Schematron also uses XPath (1.0 or 2.0), and in at least some implementations can also use XSLT or XQuery to perform the actual validation. This has led to its use in a number of more complex XML "super data" models such as those used by XBRL for financial business reporting, and the combination of XSDL to ensure validity of structure combined with Schematron for validity of constraints is now being pushed by the W3C as being a "full spectrum" validation.

Why is validation important? In a typical data-centric application, one of the most critical tasks is ensuring that the data coming into the application is "safe" and "consistent." You want to make sure that a date entry field, when asked for the numeric value 1, doesn't receive the word "one" or "0[zero]ne" or a lower-case L or nothing at all. Similarly, if asked for the number of items purchased in a line item, the program should be able to signal errors if they entered "-1" or "1.5" or even "1 billion"

or "1,000,000,000" (which may be valid in terms of type, but would most likely well exceed the capacity of the factory to produce in a period under a couple of centuries).

The benefit of XML-based validation is that it can be done collectively on all of the potential errors in a given document, rather than just the first or last error encountered, making it much easier to report all of the errors at once, rather than one at a time. Moreover, because the XML doesn't normally have to instantiate the document in question (turn it into a binary object), XML validation can be used to "safely" determine if a given XML document has the potential to insert viruses or other malware code in the data stream, usually at a point far short of where such attacks could be dangerous.

Application building with XForms

The first 10 years of the web focused primarily on the problem of displaying individual pages, first directly via files, then increasingly indirectly via parameter-driven URLs and server scripts. The next 10 years have begun to move increasingly towards the editable web — being able to communicate with the web through increasingly sophisticated tools within the browser, being able to create and update web content on the fly.

One technology that is key to building XML-based applications is XForms. XForms is in some respect one of the most powerful and simultaneously worst named standards in the W3C canon. While it can be used to fill out forms (such as an income tax form) it should actually be seen as a more general purpose language for creating XML-friendly user applications.

The XForms application is usually embedded within some other XML, most typically XHTML (the XML version of HTML), though XForms can also be embedded in Open Office documents, Scalable Vector Graphics (SVG), HTML (in some implementations), and elsewhere. This can be done because XForms controls and constraints are "abstract" — it is up to the specific browser, library, or plug-in to implement them. (For instance, the XForms implementation in EMC Documentum Dynamic Delivery Services is implemented using the Google Web Toolkit so that it can run in any web browser.)

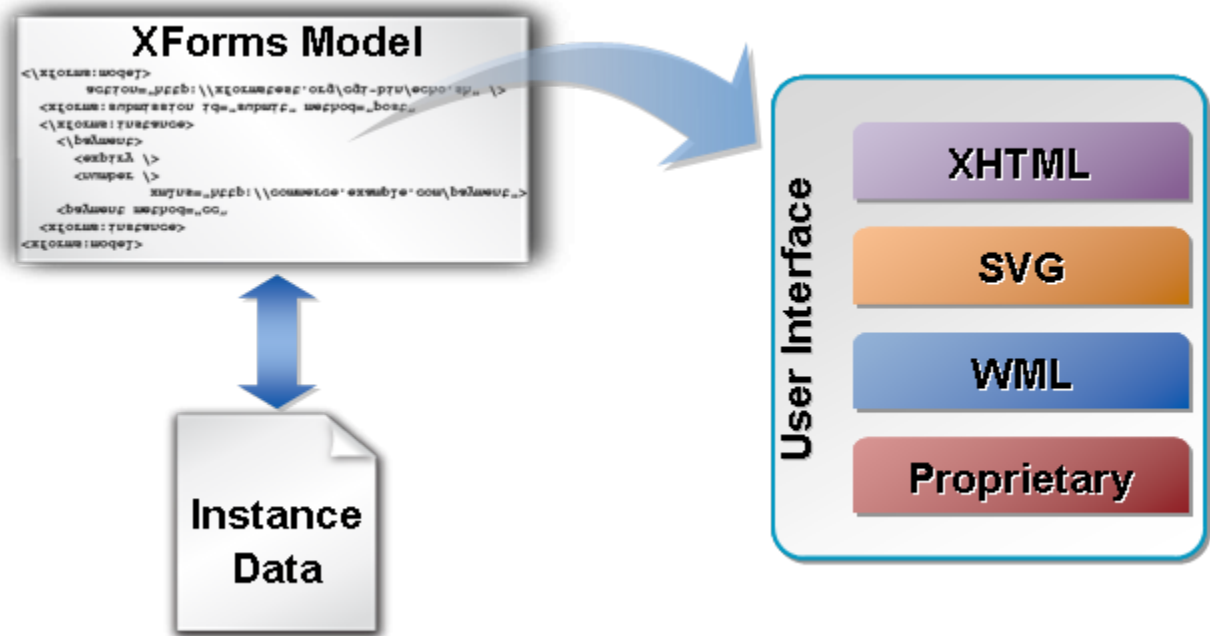


Figure 3. XForms uses a “Model-View-Controller” architecture that separates the presentation of a form from its logic, allowing a single form definition to be repurposed for different types of user interface.

XForms works by defining (or downloading) an XML model — a document of some sort. The XForms designer can then specify that a given control on the page is directly tied to an element or attribute within the XML document model. When the user changes the value in the field, the same change gets reflected back in the model automatically.

One of the most powerful aspects of XForms is that an XForms designer can set up constraints so that not only does one element value change when a user changes the text in a control, but so do any other values in the model that have a dependency or constraint on that particular element.

For instance, in a purchase order, increasing the number of pencils being ordered will automatically recalculate the total line item price, which will in turn change the total pre-tax price and post-tax price. These in turn will then refresh their respective controls. This kind of programming (what’s called declarative programming) may be familiar to you from the way that spreadsheets work — changing a value in a given cell will change the values of all cells that are dependent upon that cell in some way.

The benefit of this approach comes from the fact that at all times you are working with a complete XML document, one that can be constrained to ensure that no invalid content is entered (by using XML schemas) and that, when completed, can then be sent in its entirety back to the server. This means that the server has to work considerably less, since it doesn’t need to spend a lot of resources in validating that incoming XML, making the server layer overall much simpler.

Indeed, one way about thinking about XForms is that it provides an editor for a certain type of XML document. For example, a purchase order XML document could utilize an XHTML/XForms page to let users add address information, line items, and so forth to a baseline XML "purchase order" document, then send the resulting purchase order XML to the server to be processed.

XForms is only just beginning to be widely deployed, largely because it has taken awhile for specific implementations to reach a sufficient level of maturity and stability in various browsers. XForms works especially well in conjunction with XQuery and RESTful services, and as these technologies have become more clearly defined, so too has the role of XForms — with the combination of XQuery, RESTful services, and XForms becoming known as XRX, a term first used by XML guru Dan McCreary.

Pipelining XML

One of the most compelling visual metaphors for working with XML is the pipeline. XML documents can be passed into a pipeline composed of individual “pipes,” each of which is designed to take certain input documents and generate certain output documents to handle a variety of tasks, from running queries to transforming content to generating e-mail or persisting intermediate records.

The XML Pipeline Language, also known as XProc, is specifically established by the W3C as an XML-based language for describing these pipelines and individual pipes, otherwise known as steps. Such steps typically perform a single abstract action, though out of the box that action could be to invoke an XQuery script, transform a document using a specific stylesheet, validate a given document with a schema document, generate a PDF file, and so forth.

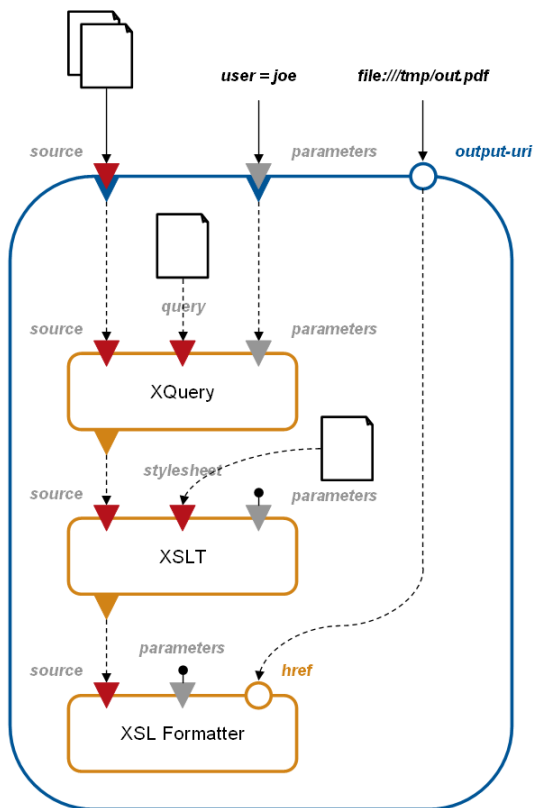


Figure 4. An XProc pipeline ties together multiple operations for processing content.

Additionally, XProc makes it possible to define new pipelines consisting of existing steps or previously defined pipelines. This encourages a highly modular approach to development, which can in turn be used to create libraries of such pipelines that can rapidly speed up building new applications.

For instance, consider two distinct operations: The first, which took a set of purchase orders and rendered them as pages of HTML records which showed only the title of the invoice, the date the purchase order was submitted, the fulfillment status, and the total amount of the invoice; and the second, which just rendered the content as XML data documents.

The fundamental actions in both cases are the same — query an initial collection, sort the results according to a given key, partition that result into pages and pick the appropriate page, then render the output. What differs in this case are parameters — what the initial collection is, what the query is, how the results are sorted, what page and page size is used, and what transformation is used to generate the final output. A declared pipeline step could be made to establish four distinct pipes for the query, sort, partition, and render actions respectively, then let the users pass in the parameters to perform this sequence.

Alternatively, each render step could be defined independently, so that you have a "renderpo-as-xml" step and "render-po-as-table" step, while the other three

operations are bundled into a single pipeline ("query-sort-page") that generates a paged set of records in the initial XML for output to be passed to the render steps.

What's most important here is that, once defined, such steps can be stored in a library and reused later. If you wanted to output the sorted results using an XForms document, you could just use the "query-sort-page" pipeline and add a "render-po-as-xforms" renderer step. This has special value for package vendors and third party value-add partners, because it means that you could create an entire package of steps for handling specific big data interactions such as HL7 records, book publishing, geographic information systems, XBRL business objects, and so on.

Note that such steps are abstractions: They represent operations, rather than perform them. This means that the steps could be defined to do a number of system functions, such as format and e-mail a given document, access an external database or LDAP store, and perform complex mathematical or financial calculations and so forth using XQuery, XSLT, or the host environment language (for example Java, in the case of the Documentum pipeline). This again is important from a packaging and marketing standpoint, as different libraries of extension "pipes" could be targeted to different markets.

An additional asset to XProc is the fact that its declarative structure makes it very easy to render graphically, either through XML tools such as SVG or applications such as Microsoft Visio. In this case, system administrators could represent each step as, well, a pipe, with the appropriate input and output ports, that could be dragged and dropped in conjunction with other pipes to create complex pipelines graphically, getting very close to the concept of true drag-and-drop programming.

Dynamic Delivery Services

To date, there is only one XML development platform that has the potential to incorporate all of these features – EMC Documentum Dynamic Delivery Services (DDS). When used in conjunction with the EMC Documentum xDB database, DDS is in fact the only fully W3C compliant platform on the market for building applications on an XML database.

At the core of DDS is its implementation of XProc, the first commercial implementation of the spec. XProc should be thought of as an orchestrator – it coordinates the action of a number of other XML processes, from queries to transformations to validators to external processes such as writing e-mail messages (or accessing e-mail) to reading external SQL databases and incorporating the results.

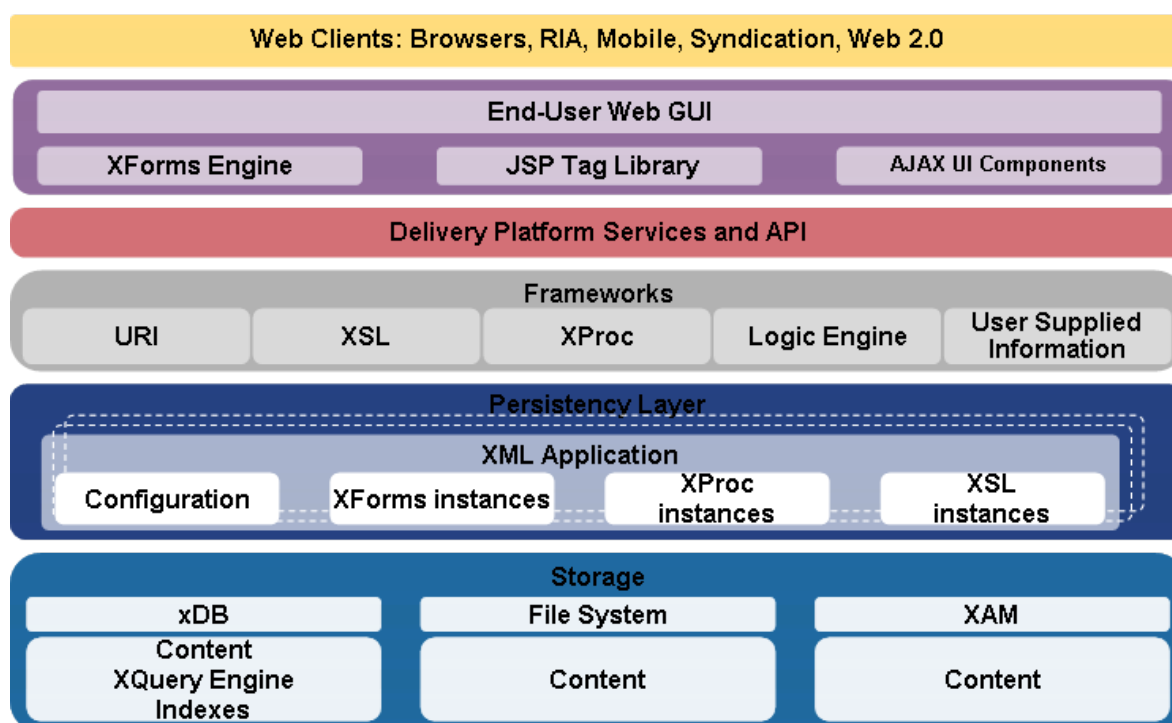


Figure 5. EMC Documentum Dynamic Delivery Services is a standards-based platform for dynamic Web applications.

Additionally, DDS provides a binding layer between the XML operations of xDB and other Java applications, such as Java servlets running in Tomcat or Jetty. These bindings make it possible to associate URLs with specific pipelines, which in turn makes it possible to use DDS as a way of creating web applications that are built around XQuery or XSLT transformations.

What this means in practice is that you can build Rich Internet applications just by setting up the appropriate pipes, which for the most part can be done using a drag-and-drop approach (EMC released a graphical Xproc Designer in July 2009 on its XML Technologies Developer Community page at <http://developer.EMC.com/XMLtech>). Additionally, in early 2011 EMC released a beta version of xProc for Documentum on the same XML Community page. This implementation includes a step library of over 100 ECM-related activities for acting on XML content in a Documentum repository.

Beyond XQuery and XSLT processing, XProc supports XInclude, which lets a given document include other documents (even dynamically generated ones) within a "container" document, as well as conditional processing that makes it possible to search through a sequence of XML documents and only process those that satisfy a given condition.

DDS and the xDB database should be looked upon as the perfect vehicles for working with large, complex and specialized XML documents, as well as for processing this content via XProc and other associated open standards technologies for both embedded Java and Rich Internet applications. No other commercial XML database platform can make this claim.

Conclusion

With the maturation of key XML standards like XQuery, XForms, XProc, and others described in this paper, developers have a powerful array of tools and technologies with which to easily build highly dynamic, interoperable, and compelling applications for information management and delivery. Using XML-oriented standards and technologies to create application services that take XML in and push XML out minimizes the cost, complexity, and effort in application development. Applications are easier to integrate and the use of standards minimizes the learning curve for developers and reduces maintenance costs. Adopting standards-based tools also lowers the risk of depending on proprietary technologies for your IT infrastructure – a key concern as information management moves to the cloud.

Finally, the use of declarative programming models based on XML standards reduces the amount of code you have to write and allows for more dynamic and personalizable behavior in your applications. Your users – whether they are citizens, customers, employees, or partners – will benefit from more responsive applications that adapt to their individual needs and preferences.

References

For more information on the XML standards and technologies described in this paper, please visit the EMC XML Technologies Developer Community at Developer.EMC.com/XMLtech.