Building Workflow Applications with XML and XQuery

By: Dr. Michael Kay

In my previous articles, I introduced some of the mechanics of coding with XQuery and XSLT 2.0 — in a rapid progression from novice to advanced level I started with a brief ten-minute introduction to XQuery, went on from there to explore FLWOR expressions in some depth, and then exposed you to the wonderful benefits of writing schema-aware XSLT and XQuery code. In this article I'm going to stick with the assumption that you're a fast learner, and assume that you've now advanced beyond the coding level and are designing entire applications.

(Actually, I don't really think of this as a hierarchy. Coders are the people who make the biggest difference to an IT system, which is why I decided five years ago after years of doing architecture and design that it was time to get back to coding. But application architecture is important too.)

In this article I want to explore the role of XML in workflow applications. I'm using the term workflow in a very general kind of way, to describe applications that one can think of in terms of documents moving around a community of people who each perform particular tasks. Processing an insurance claim, or a bug report, or a capital requisition are examples that come to mind. Many processes in publishing can be viewed as workflow applications, and one also can use the idea for a spectrum of business processes from the very everyday (like processing an invoice or scheduling a meeting) to the very complex and soft (like acquiring another company).

XML fits very well with workflow applications, because it's natural to think of them in terms of documents. In fact, I think it's such a good fit that I think one should often design an application as an XML-based workflow where we might have adopted a completely different approach in the past.

But I'm not just going to talk about workflow in the abstract: the aim is to translate this into concrete ideas about how specific XML technologies such as XML Schema, XSLT, XQuery, XForms and XML databases fit into the picture. And of course, I won't resist the occasional mention of how some of your favorite products like Stylus Studio and DataDirect XQuery (an XQuery and XQJ implementation) can help you turn these architectural ideas into working applications.

©2006 DataDirect Technologies, Inc. All Rights Reserved

Modeling Workflow Applications

Generally when we do the initial modeling for an application we split it into two parts: the data model and the process model. Sometimes we focus more on one, sometimes more on the other. One of the difficulties is often in seeing how the two models relate to each other.

A database-centric approach tends to start (naturally enough) with data modeling. You know the theory: identify the business objects, analyze their properties and relationships, translate this through normalization into a relational schema. Very often this takes no input at all from any process modeling. This is sometimes even stated as an advantage: databases should be designed without any consideration of how the data will be used, because you don't want to limit how it will be used, and the requirements will change over time.

When people use this approach to design XML documents rather than relational databases it often doesn't work too well. The reason for this is that XML generally plays a rather different role in the system. XML is used for moving information around the system — it doesn't just sit around on a mainframe waiting for people to do queries. And as a result, the design of XML data models (messages, or documents, if you prefer) depends on the process model as much as on the data model.

Let's try and make this more concrete. Let's suppose you're designing an HR (human resources) application to keep track of which employees have attended which training courses. If you take a database-centric approach, you'll probably start by identifying that there are a number of entities (people, training courses) and some relationships between them. Then you'll design a table structure that represents the fact that a particular employee went on a particular course on a given date. But if you take a more process-driven approach, you'll be more interested in how training needs are identified, how the decision is made and approved to send a particular employee on a particular course, how the schedule of courses is matched to the demand, and so on. Out of this will come some kind of flow diagram (I'm not concerned with what modeling notations you use, boxes and arrows on a whiteboard work perfectly well for me) that shows who is involved in these processes and what information they need to pass to each other. The information model we end up with is a set of messages, not a set of objects, relationships, and attributes. And very likely, you'll then translate these message or document designs into specific XML structures. Quite often, they will relate very closely to the paper documents that people are already using.

This approach doesn't do away with the need for a database. We still need to do queries, to find out who has been on which courses. But the kind of database we end up with might be

©2006 DataDirect Technologies, Inc. All Rights Reserved

very different, because its primary role is not to support ad-hoc queries, but to support the workflow.

The documents used in a workflow lend themselves very well to modeling as XML. Very often the people involved in the process think of the information in document terms —many processes have been using paper documents for years, and the whole terminology of the organization can revolve around the names of these documents (*have you seen the PS12 for this guy? No, it's held up in security vetting*). The old paper process worked with a physical document, or perhaps a sheaf of documents, which moved around the system, being added to as it made its rounds, eventually reaching some kind of completion and being filed. There are a number of benefits in modeling the IT system around these documents:

- A direct relationship between what's in the computer and the way that users think of the world is always a good thing if it can be achieved
- The design is inherently flexible. The design of documents can evolve much more easily than the design of a database, and there are fewer constraints on the information you can hold
- You don't lose information. It's much easier to keep the full history of what happened on a case. With databases, the tendency is to record only the current state, not the entire history. Sometimes we model history as well, for example the history of salary changes for an employee, but we rarely tie this to the detailed sequence of events that led to the salary change being approved. With a document-based design we get a much better audit trail.

And having established that the system design should be based on documents, it hardly needs saying nowadays that the documents should be XML. Well, it might need saying to some people: you might have a senior manager, for example, who doesn't understand why you can't handle it all using Word documents or Excel spreadsheets. So here's a checklist of the arguments you can use:

- XML can handle the full spectrum from free text to highly-structured tables. No more awkward choices whether to use Word or Excel, XML does both equally well and a lot more besides.
- XML doesn't require any particular software to be installed: it can be processed on any platform using a wide variety of tools, both server-side and client-side, and even hand-held. No more worrying about whether the application will still work when you roll out Word version 73 across your 50,000 desktops, or when you switch to a new mobile phone supplier.
- XML structure can be controlled using schemas through validation. Unlike office documents, you don't have to rely on management edicts to ensure that people follow

©2006 DataDirect Technologies, Inc. All Rights Reserved

the rules: if the schema says that a field is mandatory, then it is. But at the same time, the schema can be as flexible as you like: if you want people to be able to attach a copy of their birth certificate when they apply for a parking permit, you can allow it.

One could present this argument by saying that the XML approach to information management is a half-way house between the rigid discipline of the traditional database approach, and the uncontrolled chaos of the email-and-spreadsheet culture. But that doesn't mean it's a messy compromise: on the contrary, I would argue that it gives you the best of both worlds.

Centralized or Decentralized?

Having discussed workflow in terms of messages or documents passing around a community of people you might think that I'm advocating a very decentralized implementation. In fact, there are two ways you can build this kind of system: a design in which the documents really are sent around physically, and an approach in which they really live in some central database, and all you send around are URLs that point to them.

These days, given the good connectivity that the internet gives us, I would usually choose the second approach. It's easy to make such an application accessible to anyone with a web browser and an email client, whether they work for you or for your clients or suppliers. The great advantage is that you can give users the illusion that they are passing documents around the system to each other, but there is never any risk of the documents getting lost or delayed. Whoever manages the process can get a good overview of the state of the system, and can monitor the progress of individual cases.

There are some situations where a decentralized approach might be more appropriate, for example where the workflow operates across different organizations with no obvious central management. An example might be managing inter-library loans between different universities. But I'd say these are the exception.

XML is a key part of the solution either way. In the centralized architecture, you get the advantage that you can keep all the information in a database, which gives you the ability to find out what's happening at any point in time right across the system. And now I'll push a point that I've been leading up to: I would argue that this ought to be an XML database, not a relational database.

Why? Here are some reasons:

- The system is based on documents. Documents are what users understand, documents are what they want to ask about. *What's happened to Joe's PS12? How long is it taking to get a PF93 approved? How many capital requisitions over $1m were*

©2006 DataDirect Technologies, Inc. All Rights Reserved

approved last month, and how many were rejected? If the database holds the documents in the form users understand them, it's going to be much easier for them to ask meaningful queries.

- You don't have to design a separate schema for the database. You already have schemas for the documents, and these already contain all the structural information you need. More importantly, when the document design changes, you don't have to change the database schema to match (how do you handle all the old information that doesn't fit the new schema, anyway?). And if your documents have been designed to be very flexible — the birth certificate attached to the parking permit application — then you don't have the hassle of designing an equally flexible schema for your database (or the temptation to drop the data that doesn't fit your neat rectangular structure).
- You don't have to write code that converts documents into database updates, or that builds documents from rows and columns in the database. As the well-known analogy puts it, you don't have to take your car to bits before parking it in the garage: *you just park it*.

The traditional argument for a normalized relational database design is that it prevents update anomalies. You're only storing each fact in one place, so you remove the potential for inconsistencies. But let's turn that on its head: what are the facts? If two documents contain conflicting information, for example if one says that planning approval has been given for a new parking lot and another says that it hasn't, then a database that can only hold one version of the information is not doing us a service. Are we supposed to put nothing in the database until we've sorted out the truth of the matter? Or to assume that whichever document arrived most recently has the correct information? The fact is (excuse the pun) that the world is full of inconsistencies, and a database that captures them is giving you better service than one that pretends they don't exist. With a database designed to support workflow, it's not quite true to say that the database records what actually happened: but it captures the actual documents that were exchanged, which is the closest that you can get.

Of course, relational databases have their uses. It's a good idea to have one place where you record the names and address of your customers, or the prices of your products. Relational databases do this job rather well. And designing new applications around a workflow concept doesn't cut you off from all this important data. All the important relational databases are promising connectivity to the XML world, and DataDirect is delivering it today.

©2006 DataDirect Technologies, Inc. All Rights Reserved

The Life-Cycle of a Document

Typically in a document-based workflow, a document goes through a number of iterations as different people add to its content. This raises an interesting question about schema design. Do you try to have one schema that describes the document at every possible stage of its lifecycle (including, perhaps the stage where the originator has only half-completed it)? Or do you apply different schemas with different sets of rules at each stage of the life-cycle, using validity against a particular schema as the criterion to allow the document to progress to the next stage?

There's no easy answer to this one. The XML Schema spec, in my view, is a bit schizophrenic in this area.

- On the one hand, it's been carefully designed so that when you validate a document, you can choose which schema you want to use: part of the thinking behind that was that different rules could be applied to the same document at different stages, or by different people.
- But at the same time, XML Schema has a rather firm notion that there's a link between namespaces and schemas: if you know the namespace you're looking for, the system can magically find "the" schema for that namespace. You can get around this, by exploiting the fact that real products aren't blessed with such magical powers, so that at some point you have to tell them where to look for a real schema. However, products that cache schemas for efficiency typically won't be very happy if you try and put several schemas for the same namespace in the cache, and the same goes for XML databases, which will often allow only one schema for all the documents in a particular namespace.

In practice, you may be better off using a very permissive schema that's capable of describing the document at any stage in its life-cycle, and then doing validation for specific stages with a different technology, for example using Schematron or a custom XSLT stylesheet. However, that reduces the advantages you can get from schema-aware XSLT and XQuery processing — see my previous article.

Directories

One way of implementing your system is with a workflow package. I'm not going to go into depth here on the pros and cons of doing that, partly because I'm not up to speed on what the workflow vendors have been doing in the last couple of years. All I'll say is that I've seen several workflow-based applications implemented very successfully without such software (which is usually not cheap), and that's the approach I'm going to describe.

©2006 DataDirect Technologies, Inc. All Rights Reserved

One thing you almost certainly do need is a database of people, and the roles they play in the process. Maintaining such a database is a lot of work, and you need a strong commitment to make this happen, because everyone thinks that someone else ought to be doing it. In many organizations there's probably an LDAP directory of some kind used to run the email service, but it might not be kept up to date very reliably when people move from one part of the company to another (or leave), and it might not tell you precisely who the Health and Safety Manager in Corporate Purchasing is this week. You might find that you need to supplement the raw data in the LDAP directory with information maintained by your own application about the organization structure and the occupants of particular roles, so that the system can automatically send messages to the right people (or at least, prompt users to do so) when particular events occur.

With manual processes, we're all used to the fact that this is the weak point of any process handbook. If an accident occurs at work, we're supposed to fill in an accident report and have it countersigned by the Health and Safety Manager of the department we work in. Who is that? We make some phone calls: Oh that's Mary, but she's on maternity leave. I expect John will sign it, and John does. Computer systems, unfortunately, are less flexible.

LDAP and XML are conceptually a very good fit, because both use a hierarchic extensible data model. In fact, that's another very good reason for using XML in workflow systems. By contrast, LDAP and SQL are a very poor fit. Unfortunately the synergy between LDAP and XML hasn't been fully exploited in products yet. I'm waiting for the day when someone offers the ability to run XQuery directly against an LDAP directory, but as far as I know that's not possible yet. However, there's a workaround: it's probably quite feasible to take a daily dump of your directory as an XML document in a format called Directory Services Markup Language (DSML), and your application can run off that. It probably won't be more than ten or twenty megabytes, so it's quite feasible to hold it as an in-memory database on your server, refreshing it once a day from the latest copy. All the access to data in the directory can then be made using XQuery, XPath, or XSLT.

You'll probably have to supplement the existing LDAP directory with extra information about roles, so that your application can route documents to the right people. You can either extend the LDAP directory to hold that information, or you can maintain it somewhere else. For some applications, a simple XML document will do. You can then use XQuery to join this information with the LDAP data when you want, for example, to send an email to the HR manager of the packaging division. Since you've already got an XML database for the workflow documents, you can keep this data in XML form in the same database.

©2006 DataDirect Technologies, Inc. All Rights Reserved

Writing the Application

Given that you've chosen a workflow-based approach to application architecture, what is the actual code going to look like?

Generally, when a user needs to do something, they will be sent an email that contains a URL. When they click on that URL, it will fire off your application, which will do something like this:

- Fetch the relevant document(s) from the database and format the information for display to the user, along with a form indicating what information is required from the user
- Receive the completed form; add some data to the database, get some more data from the database, format the results for display to the user
- Generate some email to send to the next person in the process, containing a URL for them to visit in their turn
- Repeat ad nauseam

What's the best way to write such an application?

In my view, the best way to do it is entirely using high-level XML-based tools. XSLT for the HTML formatting, XQuery for the database access, XForms for capturing the user's input. There really isn't any need to go into procedural languages like Java or ASP.NET, and I would avoid them entirely: they just get you sucked into another round of data conversions. Much simpler is to keep the data in XML throughout. Of course it goes without saying that the more you use XML-based languages to develop the application, the more benefit you will gain from Stylus Studio's wide range of XML development tools.

There's another technology here that's important, and that's the one that binds the others together. I refer to it as the pipeline processor. There are various tools you could use in this role, but most of them are completely general-purpose. I would recommend instead that you consider an approach that specialized for handling XML. An example that I have used very successfully is the Orbeon Presentation Server (www.orbeon.com). There are also a number of similar products around, or you could use something like Apache's Cocoon. However, there are no standards yet — hopefully that's something which W3C will address.

XForms has an important role to play here. XSLT is good at getting data out from the system to the user, but you also need something to handle what the user sends back, and processing the stuff that comes from HTML forms is not much fun. With XForms, you can design the message sent back by the user independently of the way the data actually appears on the screen, and put this message straight into an XSLT stylesheet (or XQuery, if you prefer) for processing.

©2006 DataDirect Technologies, Inc. All Rights Reserved

It has taken a while for XForms to take off. Initially everyone hoped for native implementations in the browsers. But as we all know, Microsoft grew weary of the browser wars, and there's been little development there in the last few years. Even if that weren't the case, the problem of handling old browsers wouldn't go away. But what's happened instead is that there are a couple of excellent server-side XForms processors (one of them is part of the Orbeon package — and no, they aren't paying me commission!). What these do is to translate your XML forms definition into an HTML form, with all the necessary Javascript to give that important interactive experience; and when the contents of the form come back, the resulting mess is converted into nicely-formatted XML for your application to process. This works even better with the advent of Ajax, which allows for many lightweight messages to be exchanged between the browser and the server, for eample to do field validation, without the user ever noticing.

So for each interactive transaction, the components you need in your application are likely to include the following:

- A form that is displayed on the browser screen, containing XHTML and XForms, quite possibly generated by an XSLT stylesheet.
- A component that processes the XML "instance" returned by the forms engine, representing the data entered by the user: this component is likely to be written in XSLT or XQuery. In some applications I have seen, this component actually generates an XQuery which is then sent to an XML database to fetch the working document; in other cases the application might simply supply parameters to a hard-coded XQuery. In many cases the database will need to be updated as well, which will probably involve proprietary interfaces, since XQuery update is not yet standardized.
- A component that processes the data obtained from the database and transforms it to generate the next screenful of information for the user: perhaps a report, perhaps another form to be filled in (thus completing the cycle). Again, this could be written in XSLT or XQuery.
- Schemas that validate the data passing between the processing components. As I explained in my article on schema-aware processing, defining these schemas is extra work, but it pays of quickly by helping you find the inevitable bugs much more quickly.

All these components - queries, XSLT transformations, schemas - can of course be developed with the help of Stylus Studio.

©2006 DataDirect Technologies, Inc. All Rights Reserved

Summary: XML Workflow Applications in a Nutshell

XML encourages you to change the way you think about application design. Rather than starting in the traditional way by designing a central database for storing the data, it encourages you to think in terms of the data as it moves around the system: a process-oriented rather than data-centric approach. This works particularly well when you design the application as a document-based workflow, perhaps mimicking the design of an existing business process based on paper documents. The paper document becomes an XML document, and the application retains all the flexibility that existed in the paper-based world to vary and adapt the detailed structure of the information that flows around the system. The document is typically routed around the people who need to process it with the aid of data extracted from an LDAP directory.

I hope I've encouraged you in this article to recognize in particular that XML databases are not just a direct replacement for relational databases. They fulfill a different role in the architecture, and they work well when supporting an application designed around the notion of *data on the move* rather than *data in the warehouse*.

I hope I've also persuaded you that the bulk of your application logic can today be written in high-level XML processing languages, notably XSLT and XQuery, with individual components linked together in a pipeline processing framework. By writing the logic in these high-level languages (rather than say Java or C#), the biggest benefit you gain is flexibility and adaptability - the ability to change the application in response to changing business needs. XML gives you this flexibility in terms of data design; don't lose it by writing applications that freeze the data structure into Java or C# classes.

And if this means that you spend less time working in your Java IDE and more time working in Stylus Studio, who's going to complain?

©2006 DataDirect Technologies, Inc. All Rights Reserved