A Proposal for an OpenDocument Developers Kit

XML has proven itself to be one of the pivotal technologies of the past 10 years. But when you look at the ecosystem of tools, applications and systems that use XML, you find that there is very little code that directly manipulates XML. We all work at least one level higher, with the XML parsers and the API's they support, such as DOM or SAX. Otherwise we would all need to deal in the minutia of various XML character encodings, UTF-8 versus UTF-16, the exact lexical definitions provided in the XML specification, whitespace processing, etc. But for the vast majority of application developers, DOM and SAX are what XML means to them. Said differently: XML gave us the standard for interchanging formatted data in a language and platform neutral way, but DOM and SAX are what made application developers productive with it.

We have a similar situation with Open Document Format (ODF). The 700-page ODF specification defines what an application and platform neutral office document in XML format looks like. But we are lacking high-level developer tools that allow us to be productive with this format. An ODF toolkit will go a long way to accelerating the arrival of new ODF-supporting applications and solutions since it reduces these difficulties that currently confront the developer. Dealing with ODF as XML using an XML parser presumes mastery of the ODF specification, and this is time consuming and tedious. It is as unreasonable for the average application developer to read the ODF 1.0 specification as it is for them to read and understand all the nuances of the XML specification and write their own XML parser in order to work with XML.

For ODF to become more attractive to application developers, we need tools that are easy to master and that work with the tools the application developers already use. This paper outlines some ideas on how we could accomplish this via something I think of as the ODDK – the OpenDocument Developers Kit.

Use cases

Let us first consider the range of operations commonly performed on electronic documents:

1. interactive creation in an a heavy-weight client application
2. interactive creation in a light-weight web-based application
3. collaborative (multi-author) editing
4. automatic creation in response to a database query (report generation)
5. indexing/scanning of document for search
6. scanning by anti-virus
7. other types of scanning, perhaps for regulatory compliance, legal or forensic purposes
8. validation of document, to specifications, house style guidelines, accessibility best practices, etc.

9. read-only display of document on machine without the full editor (viewer)
10. conversion of document from one editable format to another
11. conversion of document into a presentation format, such as PDF, PS, print or fax
12. Rendering of document via other modes such as sound or video (speech synthesis)
13. Reduction/simplification of document to render on a sub-desktop device such as cell phone or PDA.
14. Import of data from an office document into a non-office application, i.e., import of spreadsheet data into statistical analysis software.
15. Export of data from a non-office application into an office format, such as an export of a spreadsheet from a personal finance application.
16. Application which takes an existing document and outputs a modified version of that presentation, e.g., fills out a template, translates the language, etc.
17. Software which adds or verifies digital signatures on a document in order to control access (DRM)
18. Software which uses documents in part of a workflow, but treats the document as a black box, or perhaps is aware of only basic metadata.
19. Software which treats documents as part of a workflow, but is able to introspect the document and make decisions based on the content.
20. Software which packs/unpacks a document into relational database form

Requirements

The above uses cases taken together cluster into several basic needs which can be articulated as:

1. A programmatic way to read, write and manipulate ODF documents at a high level of abstraction, i.e., the level needed by an application developer.
2. An embeddable ODF viewer component.
3. An embeddable ODF editor component.

Further, each of these requirements should be available across the range of client and server operating systems, including Windows and Linux. The programmatic interfaces should provide bindings in the most used application development languages, such as C/ C++, Java, Perl, Python, Ruby etc.

The ideal tool for the application developer would:

- be simple to use, requiring a minimal amount of code to solve simple tasks. The application developer should be able to write a script to generate a formatted spreadsheet with a loan amortization table in 30 lines of code.
- allow easy integration with scripting hosts, such as Perl and Python.
- provide a high-level model, closer to the problem domain of application developers, with objects which represent spreadsheets, cell and column formats.

The ODF API Toolkit

This toolkit would meet the need of high and low level developers needing to read, create and manipulate ODF documents. It would be comprised of the following pieces:

1. An ODF DOM API
2. An ODF Parser
3. And ODF Serializer

The ODF Document Object Model (DOM) would be akin to the the W3C's HTML DOM. So instead of an application developer working with Element and Attribute objects as with an XML DOM, they will deal with Sheet, Cell and Chart objects. The ultimate goal would be to have an API which covers the entire ODF 1.0 specification, such that every valid ODF document could be represented in an ODF DOM. Of course, initially, we would expect only a subset of most-commonly used functionality to be implemented. The Toolkit lends itself to incremental development and delivery.

The needs of higher-level application developers could be met in one of two ways:

- layer a high-level API on top of the low-level API via a facade.
- ensure that the lower level API has reasonable defaults for all attributes and styles, so the application developer would only need to deal with objects in their problem domain.

It would be highly desirable to standardize any such ODF DOM via OASIS or another standards body.

The ODF DOM API by itself does nothing but manipulate an in-memory model of a document. The ODF Parser/Serializer is what connects it to real ODF XML documents. The parser component takes in an ODF Document, from disk or streamed over the network, and returns the ODF DOM representation of it.

The Serializer component works in the opposite direction, taking an ODF DOM representation as input, and writing out a ODF document as output.

Some typical patterns of use for the above three components include:

1. Using the ODF DOM API, programmatically create in-memory model of the document, then call the Serializer to create the actual ODF document. An example of this would be a report generation application that would query a database, create an ODF DOM, and then call a Serializer to output an ODF document containing the

aggregated report.

2. Starting with an existing ODF document, use the ODF Parser to get the ODF DOM representation of it, and then act upon that in-memory image. Virus scanners, full text indexers, etc. would operate like this.
3. Starting with an ODF document, parse it, manipulate the ODF DOM, then serialize back out. This round-trip use would be typical of a templating application.

Ideally, a basic Parser/Serializer would be incubated as an open source project.

To ease adoption, the toolkit would need to be rounded-out with a documentation set and sample code to illustrate the basic patterns of use.

Beyond ODF

The basic Parser, DOM, Serializer mechanism described above generalizes to interactions with other formats.

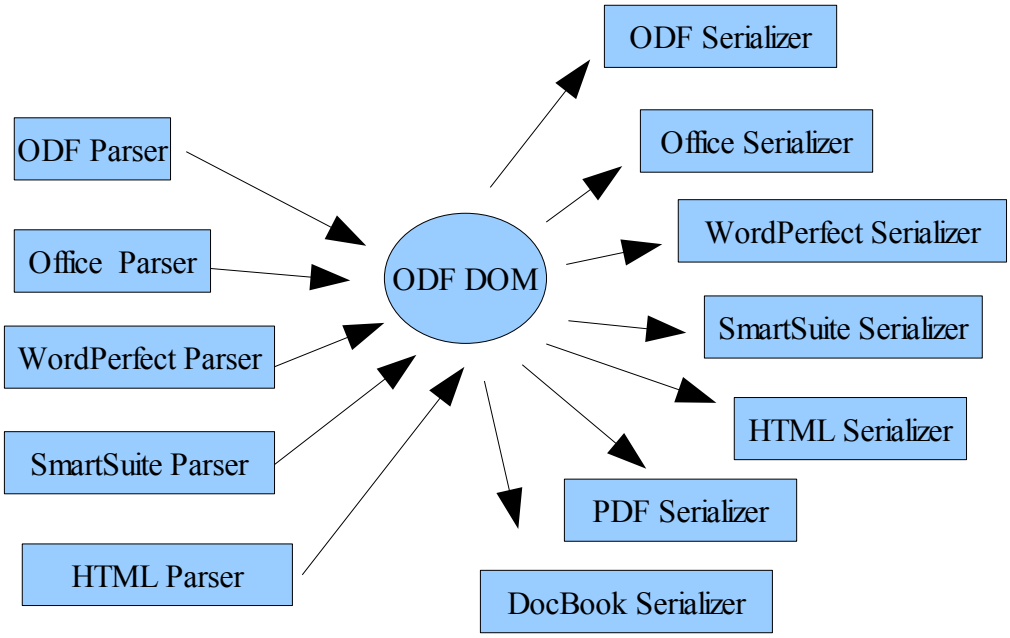
In general a Parser is a component that produces an ODF DOM object from any file or stream. The basic case is to parse an ODF document. But you could also create a Parser which would read a Microsoft Office document and create an ODF DOM object from it. Ditto for WordPerfect Office, SmartSuite, or even HTML.

The generalized Serializer takes an in-memory ODF DOM object and writes it out to another format. Beyond ODF format, it could write to Microsoft Office, HTML, PDF, DocBook, etc.

Combining different Parsers and Serializers allows new patterns of use when you wire them in different combinations:

- Parse an ODF document into ODF DOM and then serialize to HTML to get a quick server-based lightweight ODF viewer.
- Use an Office Parser to produce the ODF DOM, then immediately pass the DOM to an ODF Serializer to create a file conversion utility.

This gives us the familiar wheel-and-spokes pattern, where the ODF DOM is at the center, and Parsers and Serializers are the spokes:



There would be clear benefits if Parsers and Serializers could be treated polymorphically, i.e., be pluggable, so that the same underlying application logic would work regardless of what the incoming our outgoing format were.

From the application developer's perspective, the above model is simple, elegant and powerful. Consider this, an ODF DOM, as outlined above, could easily become the preferred way of creating or manipulating any office document, not just ODF documents, though naturally it would work best with ODF. So an application developer could use this toolkit to take an Office 2003 document as input and convert it to DocBook if that is what their customer desired. But the cost of adding incremental support for ODF is near zero, since the parsers and serializers would be pluggable. So, the long-term impact of the ODF DOM might be to make file formats irrelevant from the perspective of the application developer. They would write their application logic and plug in whatever Parser/Serializer they need. This is similar to what standards like ODBC/JDBC did for how application developers deal with databases. This tends to lower barriers to entry and allow implementations to compete on their merits, and this is good for ODF and vendors with ODF compatible products.

Viewers/Editors

Although a viewer or editor component could be built upon the above framework, we should acknowledge that the hard part of creating a spreadsheet editor component is the spreadsheet runtime code, not the file-reading code. So the fastest path to these

components would be to extract/refactor portions of an existing implementation, such as OpenOffice.org to make an embeddable viewer or editor component. By using #defines and build flags it may be possible to do this within the existing OpenOffice.org project, without forking the source code. Support for ODF DOM could then be retrofitted to these refactored components.

An intriguing possibility is to have the runtime automation API of the editor component be ODF DOM. Runtime manipulations of this DOM would then be reflected as real-time changes in the editor. This is similar to how browsers update the page in response to DOM changes, giving us Dynamic HTML. Supporting this for an ODF component could allow us to write a simple script which would automatically update a cell every five seconds to update a stock quote, for example. Add in a standardized event model, and you could build some sophisticated document-based applications. Can we even imagine an AJAX approach with ODF?

Rob Weir

robert_weir@us.ibm.com