Introducing DOM

When it comes to processing XML documents most developers look first to the two most popular techniques, SAX and DOM. I discussed SAX, the Simple API for XML, in a previous article at DevChannel, and in this article I'll be introducing the main alternative - the Document Object Model, known as DOM. The DOM approach is to load the whole XML document into memory, resulting in a single data structure that contains the XML's elements, attributes, text and everything else. It is this data structure with which the developer works, the DOM API containing all the facilities required to explore and manipulate the data.

The DOM is suited to applications where random access to parts of the XML document is required, or where the data has to have pieces added on, moved or deleted. The trade off for this level of flexibility is memory requirement. A significant amount of memory is required for a machine to hold a large XML document.

The Structure of the DOM Specification

The Document Object Model specification, which was recently updated to Level 3, comes in several parts. There is a core module, known, appropriately, as the DOM Core, which defines an interface that provides the facilities necessary for accessing and manipulating both HTML (version 4.01 onwards) and XML (version 1.0 onwards) documents. Built on top of this core are several other modules that add various specialised features. For this article, and for the majority of XML and HTML processing tasks, the DOM Core contains all the required features.

As with many complex specifications, an implementor needs to decide how much of it to get working, and which aspects to optimise. While it would be in everyone's interest to have a single, complete implementation, in practise, producing such a thing for the DOM is a huge amount of work. There is also a balance to be struck in terms of speed, correctness, time to market, and various other factors. Because of this there are many implementations of the DOM, and a developer often has a choice of libraries even for one programming language. Packages range from very fast and light partial implementations, to slower, more resource demanding implementations that claim to correctly handle most aspects of the specification. Anyone faced with a demanding DOM based task should investigate which combination of programming language and DOM implementation is best for their needs. This article looks at some of the options available to the Python programmer.

DOM options from Python

Since Python version 2.0, the library supplied with the language has contained a DOM implementation known as minidom. The name is a little inappropriate these days, since although minidom started off as a small implementation, it has got steadily larger and more feature complete. minidom is stable and feature rich, and since it comes with the language itself, it is an excellent first choice for a DOM implementation.

There is, however, an alternative. The PyXML project actually maintains two DOM implementations. The first of these is called, totally confusingly, minidom! This is no coincidence - the Python library version of minidom is actually a snapshot of the PyXML version of minidom. This means that the PyXML version is both more up to date than the Python library version, and a drop in replacement for it. Because all the names in the interfaces between the packages are identical, if you install the PyXML package (and most Linux distributions at least supply it if not install it by default) your Python script will automatically and transparently use the PyXML version of

minidom. This is normally a good thing – the PyXML version is better code – but bare it in mind if you run your script on various machines, some of which have PyXML installed and some of which don't, because you might see different behaviour.
PyXML also carries a completely separate DOM implementation called 4DOM, which has been developed, maintained and donated to PyXML by FourThought Inc. As of this writing 4DOM implements the DOM Core Level 2 specification very closely, and is one of the most complete DOM packages available. I chose it as the basis for the examples in this article because for the requirement – education – it seemed the most appropriate, despite its trade offs of being slow and heavy on resources.

Parsing XML into a DOM

Enough of the theory, let's look at some code. Building a Document Object Model structure from an XML document is very straightforward. Here is an example of some code that uses PyXML's xmlproc validating parser:

```
#!/usr/bin/env python

import sys
from xml.dom.ext.reader import Sax2

document = Sax2.FromXmlFile( sys.argv[1], validate=1 )
```

I'm using the static function FromXmlFile() in the 4DOM Sax2 module to read an XML document into a DOM structure. Yes, that's right – this method of building a DOM actually uses SAX to read the XML document and build its tree! Internally, the FromXmlFile() function creates an XML parser object and an empty 4DOM based DOM, then parses the XML document into the DOM structure. In this example I switch validation on, which isn't essential, but makes life easier later, as will be explained in a moment.
FromXmlFile() returns a Document object, which is defined in the DOM specification. It is worth becoming familiar with the DOM specification because many DOM implementations, including 4DOM, reference it directly in place of their own documentation. As explained on page 41 of the DOM Level 3 specification, the Document object is the one that is the 'root' of a Document Object Model tree. It is the single entry point to the data tree itself.

Simple Navigation

Now that we have a DOM tree structure in memory, and a Document object that references the top of it, we can look closely at the data structure. A Document Object Model is made up of nodes, of which there are several types. The Document object, which we've already seen, is actually of type DOCUMENT_NODE. Each element in the XML document is represented by an ELEMENT_NODE, each piece of text by a TEXT_NODE and so on. There are node types for comments, processing instructions, CDATA sections and every other type of component that might appear in an XML document. The DOM specification contains details of all the node types, and the exact methods and attributes which they each contain.
As with most tree-style data structures, each node has exactly one parent and zero or more children. Consider this simple XML document which describes three disk devices attached to a server (one of which is currently not mounted):

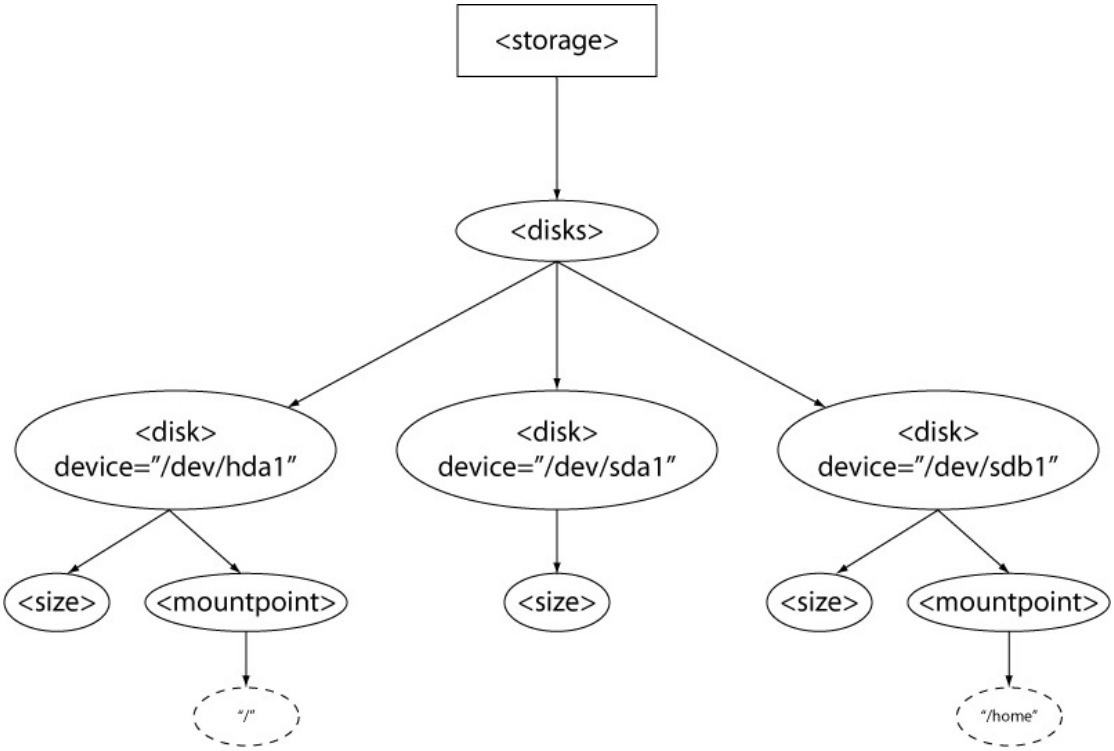
```
<?xml version="1.0" encoding=”UTF-8” standalone="yes"?>
<!DOCTYPE storage [
<!ELEMENT     storage    (disks)>
<!ELEMENT     disks      (disk*)>
<!ELEMENT     disk       (size, mountpoint?)>
<!ELEMENT     size       EMPTY>
<!ELEMENT     mountpoint (#PCDATA)>

<!ATTLIST     disk
  device      CDATA      #REQUIRED>
<!ATTLIST     size
  unit        CDATA      #REQUIRED
  capacity    CDATA      #REQUIRED>
]>
<storage>
  <disks>
    <disk device="/dev/hda1">
      <size unit="GB" capacity="80" />
      <mountpoint>
        /
      </mountpoint>
    </disk>

    <disk device="/dev/sda1">
      <size unit="GB" capacity="120" />
    </disk>

    <disk device="/dev/sdb1">
      <size unit="GB" capacity="120" />
      <mountpoint>
        /home
      </mountpoint>
    </disk>
  </disks>
</storage>
```

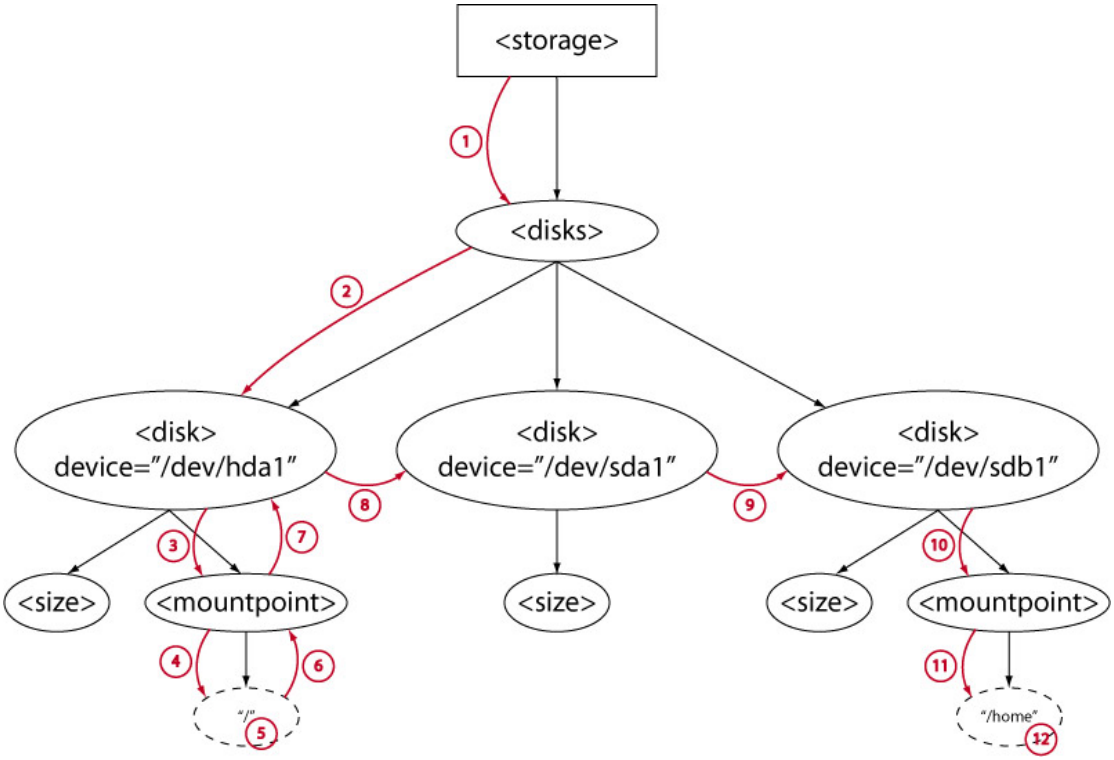
This XML has a DTD, which is always advisable when working with the DOM. When the XML can be validated the tree structure produced is a lot more predictable, and the parser can identify ignorable whitespace such as the indenting. A tree built without irrelevant whitespace is a lot easier to work with as long as you don’t need to recreate the exact XML document (in which case you must keep the whitespace and work around it). Using the above example code to parse this XML document into a DOM tree gives a data structure that looks something like this:



The top node, drawn as a rectangle, is the Document node returned by the parsing process. This points to the first XML element, the disks element. (I've drawn element nodes as ellipses and I've left out most of the attributes for clarity.) The disks element has three child nodes, which are the element nodes representing the disk elements. Each of those has child nodes representing their child elements, and at the bottom there are the text nodes (drawn as dashed ellipses) holding the text strings for the mountpoint elements.

Navigating around a document object model is much like changing up and down the directory structure of a disk filesystem. The analogy of the current working directory is the current node – the point we are logically "at". Just like a disk structure, you can start at the root node, then change the current node by moving up or down the tree one step at a time. Alternatively, if you know where you're going, you can jump straight to a certain point.

As an example, let's go into our DOM and collect the two mountpoint text strings. Referring to figure 2:



Pseudo code to navigate to these text nodes might look like this:

```
1) Fetch the first element from the DocumentNode
2) Jump to that node's first child
3) Jump to that node's last child
4) Jump to that node's first child
5) Read that node's value
6) Jump to the current node's parent
7) Jump to the current node's parent
8) Jump to the current node's next sibling
9) Jump to the current node's next sibling
10)Jump to the current node's last child
11)Jump to the current node's first child
12)Read that node's value
```

And the actual Python code, which uses the DOM API methods and attributes:

```
currentNode = document.documentElement
currentNode = currentNode.firstChild
currentNode = currentNode.firstChild
currentNode = currentNode.lastChild
currentNode = currentNode.firstChild

print str(currentNode.nodeValue).strip()

currentNode = currentNode.parentNode
currentNode = currentNode.parentNode
currentNode = currentNode.nextSibling
currentNode = currentNode.nextSibling
currentNode = currentNode.lastChild
currentNode = currentNode.firstChild

print str(currentNode.nodeValue).strip()
```

I coded this sequence of jumps to demonstrate the concept of navigation, but given that I know in advance exactly where the mountpoint strings are, this is a simpler approach:

```
1) Fetch the first element from the DocmentNode
2) Note this node ("disks")
3) Read the value from the first child of the last child of the
   first child of the "disks" node
4) Read the value from the first child of the last child of the
   last child of the "disks" node
```

Which in Python can be coded like this:

```
currentNode = document.documentElement
disksNode = currentNode.firstChild

print disksNode.firstChild.lastChild.firstChild.nodeValue.strip()
print disksNode.lastChild.lastChild.firstChild.nodeValue.strip()
```

This style of navigation works best when, as with this example, you know the exact format of the XML document you're navigating over. Some developers like to pre-process their XML into small, highly predictable XML document fragments using something like XSLT to make navigation simpler.
When you're not so sure of the tree structure, you have to examine the nodes to find your way around and home in on the data you want. The DOM API has many features which can help this process. These include methods like nodeType() which tells you the tag of the element node you're at, hasChildNodes() which allows you to test to see if a node has children, and the childNodes attribute which gives you a list of child nodes allowing you to loop over them. As usual, the details are all in the DOM specification.
One of the most useful methods is the Document object's getElementsByTagName() which returns a list of element nodes, one for each element in the document with the given tag name. For example, we could find our "mountpoint" strings with this code:

```
for mpNode in document.getElementsByTagName( "mountpoint" ):
    print mpNode.firstChild.nodeValue.strip()
```

This is a more typical use of the DOM API. All the node to node navigation is replaced by the getElementsByTagName() call which finds the required nodes quickly and easily. A single jump down to each mountpoint node's text node is all that is then required to get to the string value.
Other optimisations are sometimes appropriate, such as pushing nodes onto a stack as you navigate into the tree, then popping them off again to find your way back. There are lots of software engineering techniques that work well with tree structures, so it is often easier to consider the DOM API as a part of the solution, rather than all of it.

Working with Different DOM Implementations

We've already seen that Python with the PyXML package has at least two DOM implementations – minidom and 4DOM. For any particular project either implementation might do the job, but for others a specific implementation might be required. In the example above the question of choosing an implementation didn't arise because I used the Sax2.FromXmlFile() function directly. I happen to know that this code automatically uses the 4DOM DOM implementation, so I've bypassed the

issue until now. Sometimes, however, you need to specify exactly which implementation you want.
The PyXML package supplies a factory function called getDOMImplementation(), which returns an instance of a DOMImplementation object, and you can use it to name the implementation you want to use:

```
from xml.dom import getDOMImplementation

try:
    implementation = getDOMImplementation( "4DOM" )
except:
    print "Unable to find 4DOM - check your PyXML package."
    sys.exit( -1 )
```

The DOMImplementation interface is the entry point to the DOM API, and although its usage is normally hidden behind convenience functions like Sax2.FromXmlFile(), you can use it to check feature support and to create new DOM trees from scratch where necessary.

Modifying XML Documents Using DOM

I'll finish this article with two more examples of tasks the Document Object Model is typically used for. We've seen that freely moving around and collecting information from an XML document is one strength of the DOM. Another is the ability to modify a document. The DOM API contains everything required to add, move or delete sections from a document.
As an example, let's remove the middle disk element and its children from our example storage document. In order to do this we need to navigate to the node to remove, then call the DOM's Node.removeChild() method for its parent node:

```
targetNode = document.getElementsByTagName( "disk" )[1]
parentNode = targetNode.parentNode
parentNode.removeChild( targetNode )

from xml.dom.ext import PrettyPrint
PrettyPrint( document )
```

The call to getElementsByTagName() returns a Python list of all the nodes in the document with a tag of "disk". I know there are 3, and it's the middle one (with index 1) which we want to remove. I find its parent via its parentNode attribute, then call the removeChild() method for that parent node. Converting the DOM tree back to XML text shows the middle disk node has now gone:

```
<?xml version='1.0' encoding='UTF-8'?>
<storage>
  <disks>
    <disk device='/dev/hda1'>
      <size unit='GB' capacity='80'/>
      <mountpoint>
        /
      </mountpoint>
    </disk>
    <disk device='/dev/sdb1'>
      <size unit='GB' capacity='120'/>
      <mountpoint>
        /home
      </mountpoint>
```

```
    </disk>
  </disks>
</storage>
```

Finally, let's look at some code that builds a DOM tree from scratch. The methods to start this procedure are in the DOMImplementation object, so I need to start by getting one of those. Then the process involves creating a document type, then creating an empty document of that type. The required elements and attributes can then be created and placed into the tree in the right places.:

```
#!/usr/bin/env python

import sys

from xml.dom.ext import PrettyPrint
from xml.dom import getDOMImplementation

try:
    implementation = getDOMImplementation( "4DOM" )
except:
    print "Unable to find 4DOM - check your PyXML package."
    sys.exit( -1 )

### The xxx works around a (reported) bug in the 4DOM serialiser
#
docType =  implementation.createDocumentType( "storage", "xxx",
None )
document = implementation.createDocument( None, "storage", docType )

docNode = document.documentElement

disksElement = document.createElement( "disks" )

diskElement = document.createElement( "disk" )
diskElement.setAttribute( "device", "/dev/cdrom" )

sizeElement = document.createElement( "size" )
sizeElement.setAttribute( "unit", "MB" )
sizeElement.setAttribute( "capacity", "700" )

mountElement = document.createElement( "mountpoint" )
mountText    = document.createTextNode( "/media/cdrom" )
mountElement.appendChild( mountText )

docNode.appendChild( disksElement )
disksElement.appendChild( diskElement )
diskElement.appendChild( sizeElement )
diskElement.appendChild( mountElement )

PrettyPrint( document )
```

This prints the XML document:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE storage>
<storage>
  <disks>
    <disk device='/dev/cdrom'>
      <size capacity='700' unit='MB'/>
```

```
      <mountpoint>/media/cdrom</mountpoint>
    </disk>
  </disks>
</storage>
```

Conclusion

The Document Object Model is a very large specification, and a discussion of how to really exercise its power could fill a book. While this article has given an overview of what the DOM is, an awful lot remains for the reader to explore, both in the way DOM trees are structured, and how the DOM API can manipulate them. A scripting language like Python makes building and examining DOM trees quick and easy.
It is almost inevitable that anyone interested in low level HTML or XML document processing will need to manipulate a DOM tree at some point. Even when working with higher level facilities like XPATH, having an understanding of what can be done through the DOM interfaces is very useful.