To my family, old and new.
Acknowledgements

Ian Cox - for the inspired design of the books front and back cover.

Ant development team - Ant is a wonderful tool and I thank the cabal of developers who have spearheaded its development over the years.

Ant emailing list - I would not have learned Ant without the helpful answers embedded in this useful email list.

Tony Davis - Was the original editor for this book when it was in a much different form and was considering it for publication through a more normal route; quite a long time ago. I would like to thank him for his inspiration and advice.

Michael Caito - Helped pound some of my more unreadable text into something useful and inspiring me with his ‘wordsmithery’ over the years.
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>5</td>
</tr>
<tr>
<td>Text Recipes</td>
<td>9</td>
</tr>
<tr>
<td>Find and Replace</td>
<td>9</td>
</tr>
<tr>
<td>Managing Encoding and Whitespace</td>
<td>19</td>
</tr>
<tr>
<td>Handling Unicode</td>
<td>23</td>
</tr>
<tr>
<td>Facilitating Localisation</td>
<td>27</td>
</tr>
<tr>
<td>Advanced Text Processing</td>
<td>32</td>
</tr>
<tr>
<td>Conclusion</td>
<td>34</td>
</tr>
<tr>
<td>Database Recipes</td>
<td>35</td>
</tr>
<tr>
<td>Mysql import macrudef</td>
<td>35</td>
</tr>
<tr>
<td>Accessing SQL during an Ant build</td>
<td>36</td>
</tr>
<tr>
<td>The SQL task</td>
<td>37</td>
</tr>
<tr>
<td>A note about XML Database</td>
<td>38</td>
</tr>
<tr>
<td>Conclusion</td>
<td>40</td>
</tr>
<tr>
<td>XML Recipes</td>
<td>41</td>
</tr>
<tr>
<td>The Basics</td>
<td>41</td>
</tr>
<tr>
<td>Working with XSLT</td>
<td>53</td>
</tr>
<tr>
<td>Validating XML</td>
<td>58</td>
</tr>
<tr>
<td>Open Office XML Documents</td>
<td>74</td>
</tr>
<tr>
<td>Source Code Bonus: Advanced XML Databases</td>
<td>81</td>
</tr>
<tr>
<td>Conclusion</td>
<td>81</td>
</tr>
<tr>
<td>Web Recipes</td>
<td>83</td>
</tr>
<tr>
<td>Working with HTTP</td>
<td>83</td>
</tr>
<tr>
<td>Consistent and Clean HTML</td>
<td>91</td>
</tr>
<tr>
<td>Optimising the Web</td>
<td>103</td>
</tr>
<tr>
<td>Other Web Techniques</td>
<td>110</td>
</tr>
<tr>
<td>Source Code Bonus: Web Services</td>
<td>118</td>
</tr>
<tr>
<td>Conclusion</td>
<td>118</td>
</tr>
<tr>
<td>Case Study: Working with Multilingual...</td>
<td>121</td>
</tr>
<tr>
<td>Critical Concepts</td>
<td>122</td>
</tr>
<tr>
<td>Overview of Multilingual Site</td>
<td>127</td>
</tr>
<tr>
<td>Requirements</td>
<td>128</td>
</tr>
<tr>
<td>Running the Code</td>
<td>132</td>
</tr>
<tr>
<td>Looking at the Code</td>
<td>134</td>
</tr>
<tr>
<td>Working with Source XML / XSLT Distribution</td>
<td>135</td>
</tr>
<tr>
<td>Generate distribution with Ant Build</td>
<td>155</td>
</tr>
<tr>
<td>Conclusion</td>
<td>165</td>
</tr>
<tr>
<td>Appendix</td>
<td>167</td>
</tr>
</tbody>
</table>
Preface

This book is a collection of Apache Ant recipes that I have gathered up whilst doing Web development. The book is not intended to be an introduction to using Apache Ant, nor does it pretend to be a complete treatment on using Ant with the various technologies that I have presented. You will still need to refer to Ant’s rather good manual or any of the other proper Ant books to learn ‘how to use ant’.

This experimental book was a side project to investigate how to ‘self publish’ a small book. Doing a book by oneself is a powerful reminder of how difficult it is to create something of quality when you do not have a team of talented people behind you doing the copy editing, reviewing, graphics and all the inevitable checking that ends up making one look professional. Hence, the most difficult decision with self publishing was acknowledging that I had to be satisfied with much less then I am normally comfortable with.

I hope that readers will look past my foibles as a ‘newbie’ book publisher and get some value out of the Ant Recipes contained within.

— James Fuller

Who is this book for?

Web Developers using technologies such as javascript, HTML, RDBMS, XML, XSLT and perhaps some periphery Java, are the primary intended audience for this book.

The book is intended as a source of inspiration for solving problems rather than something one would read ‘cover to cover’.

Ant Recipe’s focuses on using Ant basics in conjunction with key technologies… this approach is geared for the reader to comprehend things without having to be an Ant expert, but be warned I do assume that the reader knows the basics of using Apache Ant, and if not that they can refer to Ant’s own excellent documentation for reference.

Why Apache Ant?

Ant is the Apache Software Foundation’s (ASF) tool for building, testing and deploying Java software. Over the last several years, Ant has become the standard utility for building java software superseding build utilities such as the venerable make along with, autoconf, gnumake, nmake, jam, etc …

The reasons I give to use Ant to solve problems are:
• **Cross Platform:** Solutions developed in Ant, to a certain extent, will run wherever Ant will run

• **Easy to understand:** The Ant XML description of a process is arguably easier for developers to comprehend.

• **Integration:** Ant’s extensible suite of Tasks is one of the biggest benefits for considering its usage outside of the build domain. Ant integrates with many important Java technologies; providing a uniform façade over the multiplicity of Java API’s in existance today.

• **Large User Base:** Ant has been around for a while and is used in lots of popular Open Source projects. The knock on effect of this wide spread adoption is support of Ant in most development environments. Furthermore, I find that there is a lot of casual ‘know how’ amongst web developers of how to use Ant in its most basic incarnation.

**Isn’t Apache Ant for Java?**

Ant Recipes for Web Programmers is about using Apache Ant beyond it’s role as a Java build tool, applied to solving problems specifically for Web Developers.

By dint of Ant’s Java nature, it can be useful if the reader also has cursory knowledge of the Java programming language but is not required. On the other hand, if you are a Java programmer you will find the book almost devoid of Java as I focus on using Apache Ant and its various extensions with non-Java web technologies.

**Overview by Chapter**

All chapters present Recipes — self-contained, real world examples you can use to solve problems.

**Text Recipes:** Users will learn many techniques to finding & replacing text and how to manipulate textual data using Ant; including managing pernickety Control Characters and white-spaces as well as dealing with language localisation issues.

**Database Recipes:** This chapter deals with using Ant with Relational Databases; learn how to control the loading, deployment, and backing up of SQL databases. In addition, I show techniques of integrating SQL queries directly during the Ant build process.

**XML Recipes:** A hodgepodge of XML related Recipes…from an introduction to using XML with ant to how to use Ant to control XSLT processing and manage Open Source XML documents. In addition, I show how to solve your XML schema validation issues using Ant’s own built in Tasks as well as sophisticated approaches including the Jing processor and Namespace Routing Language (NRL).
Web Recipes: Everything you can to do with Ant and the web … from spell-checking and validating websites to working directly with HTTP.

Case Study - Multilingual Websites: Learn how to use Ant to control the publication and generation of a sophisticated multilingual website. This last chapter is a bit different in that it gives a fuller treatment to the specific topic of generating multilingual websites and its designed to show what Ant is capable of.

What is needed to run the examples in this book?

This is the first book where I provide source code examples only for Linux, Unix or Mac OSX type operating systems. I do believe that, with a bit of tweaking, most code examples will work wherever Ant runs.

Each chapter has a matching source code example directory (contained in the download for the book). Each download directory will contain a README and directories relating directly to the sequence of Recipes as they are presented in the book.

A lot of the source code is dependent on 3rd party Open Source libraries which you will have to download. To help you with downloading with these dependencies you can go to http://www.antrecipes.com and get the code and associated libraries there.

The minimum requirement for running the included source code is as follows:

• Java JRE v1.3 or above as required minimum to run Apache Ant itself
• Java SDK/JDK v1.3 or above as required for all of Ant’s Tasks to operate
• Latest stable release (as of this writing v1.7) of Apache Ant: http://ant.apache.org

Website

The website for the book is located at

   http://www.antrecipes.com

You may download all code samples from here.

   http://www.antrecipes.com/ant_recipes_src.zip
This chapter's Recipes presents Ant's prodigious array of functionality in the management of textual data.

We will be covering the following topics:

- **Find and Replace**: Learn how to replace text and tokens contained in text files
- **Managing Encodings and Whitespace**: Manage character encoding data, including control characters and whitespace.
- **Handling Unicode**: Manage and convert text to UTF8
- **Facilitating Localisation**: Learn how to localize text against multiple languages
- **Advanced Processing**: Use powerful third party Tasks to perform sophisticated text manipulation

**Find and Replace**

One of the most common tasks that occurs during development is the need to find and replace snippets of textual data contained in files. As those readers familiar with using regular expressions will already know (via sed, grep, etc.), mastering textual manipulation can save a lot of time and effort. The finding and replacement of tokens is also the basis for enabling a wide range of 'templating' to assist in the creation and maintenance of documents.

Developers should endeavor to automate performing mundane 'find and replace' tasks. Doing this type of thing with Ant will mean that you will have a cross platform solution, easily embedded in your development and deployment environment, as well as readable build file.

Using its built-in Tasks and Filters, Ant can dramatically extend your range of techniques for finding and replacing text.

**<replace/> and <replaceregexp/> Tasks**

*Recipe: “Learn how to find and replace text”*
If you are anything like me (e.g. lazy), you are always looking for an existing solution to an existing problem ... Ant's built in Tasks represent just this kind of thinking. For example, Ant has a dedicated Task for text replacement operations, which is the <replace/> Task.

I show its basic form in the proceeding Ant snippet.

```xml
<replace dir="${src.dir}" value="Gabriela">
  <include name="**/*.txt"/>
  <replacetoken><![CDATA[firstname]]></replacetoken>
</replace>
```

I use the CDATA section in the <replacetoken/> to indicate to preserve whitespace which is handy when text spans multiple lines. When executed, this example will replace all occurrences of the string 'firstname' with 'Gabriela'. In all the examples for this recipe, remember that changes are applied to the actual source files, which is why I have kept a copy of original source in the copy_of_data directory.

A slightly more intricate example is presented below showing how to define multiple tokens for replacement using the <replacetoken/> element, in addition to providing an external property file containing token definitions.

```xml
<replace file="data/test.txt"
  propertyFile="data/some.properties"
  value="some default value">
  <replacefilter token="@token1@"/>
  <replacefilter token="@token2@" value="value2"/>
  <replacefilter token="@token3@" property="value3"/>
</replace>
```

The <replacetoken/> element is needed to replace items that cross multiple lines. The first token, @token1@, is replaced with the <replace/> Task default value. The other <replacetoken/> are substituted with values supplied from properties in data/some.properties. Run example1b Target to see how this operates.
If one desires more control over what tokens (strings) to replace, then there is the `<replaceregexp/>` Optional Task. It lets developers define token matches using regular expressions. The example2 Target will replace any references to Clinton, Bush, or Reagan with the text ‘former presidents’.

```xml
<replaceregexp file="data/regex-test.txt"
    match="(Clinton|Bush|Reagan)"
    replace=" former presidents"
    flags="g"
    byline="true"/>
```

The following is a more interesting example of using `<regexp/>` and `<substitution/>` tokens, example3 Target finds default XML namespace declarations and replaces them with xmlns:xsl, for all XSLT documents contained under the data directory.

```xml
<replaceregexp byline="true">
    <regexp pattern="xmlns=(.*)"/>
    <substitution expression="xmlns:xsl=$1"/>
    <fileset dir="data" includes="**/*.xslt"/>
</replaceregexp>
```

Now I will introduce the concept of Ant Filters which is the next evolutionary step in managing such operations in Ant.

*Note: remember there is no ‘undo’ type facility with these Tasks, so it is recommended to make backups of critical files before irreversible processing takes place.*

**<filter/> Task**

*Recipe: “substitute tokens found in text using Ant native filter functionality”*

A less prescriptive method of finding and replacing is to use the Ant `<filter/>` Task which will perform token replacement in all files that are manipulated by Ant. As with the `<replace/>` Task, I can use tokens (that look like this `@token@`) which get replaced.

Ant filters work in conjunction with any of Ant Tasks that perform file copy operation, such as `<move/>` or `<copy/>` allowing one to define filters (tokens) for replacement.
The `<filter/>` element may be declared at the project level or within the Target itself and is enabled by setting the filtering attribute of the relevant Ant task as demonstrated below.

```xml
<filter token="name" value="Jason"/>
<filter token="version" value="1.6.5"/>
<copy file="data/data.txt" tofile="result/data.out.txt" filtering="true"
    overwrite="true"/>
```

Running the filter Target copies a new file, data.out.txt, to the result directory. During the file copy operation, Ant will replace all occurrences of the tokens `@name@` and `@version@`, within data.txt, with their respective defined values.

If I take a look inside the data.txt file, in the example section for this chapter, I can confirm that tokens are being used to create a template for generating a personalized greeting.

Dear `@name@`,

We would like to understand your opinions on using Ant `@version@` the open source Java build tool.

regards, Apache

Similar to Ant properties, once a filter is set it cannot be ‘reset’ and there is no notion of scope with filters set at the top-level `<project>` or within `<target>` element’s. Also like properties, you can elect to define an external set of filters in a separate file, using the filtersfile attribute to load the filters in e.g;

```xml
<filter filtersfile="filter.txt"/>
```

This external file should define filters exactly how external properties are defined, that is using name=value pair format.

Ant Filters basically do the equivalent of a `<replace/>` task. There are some performance benefits associated with ‘piggybacking’ replacement operations whenever one uses the `<move/>` or `<copy/>`
Tasks, though I will show a better way of achieving the same thing using Ant FilterChains, which can be considered to supersede Ant Filters.

Though because Ant Filters are still widely used I will show the reader how to compose them into reusable sets before leaving the concept for `<filterchains/>`.

*Note: It should go without saying that filters are meant for text files, don’t expect any kind of reliable replacement behavior when using `<filter/>` or `<replace/>` on a set of binary files.*

**Working with Filtersets**

*Recipe: “create reusable sets of filters that perform token replacement”*

Ant allows one to define reusable sets of filters, via a `<filterset/>` element. This element is nested directly into an Ant Task that supports filtering.

```xml
<copy toDir="result/filterset">
<fileset dir="data">
  <include name="data.txt"/>
</fileset>
<filterset begintoken="@" endtoken="@">
  <filter token="name" value="Jim"/>
  <filter token="version" value="1.7"/>
</filterset>
</copy>
```

If I must use Ant Filters then I prefer the approach above as opposed to previous examples mainly because filters with the same token names can be defined within different filtersets. Another valuable feature is the ability to set begin and end token delimiters, useful if @ doesn’t suit your particular tastes. Lastly one doesn’t need to set a filtering attribute to *true* on the Task itself as it is implied whenever there is a `<filterset/>` present.

A `<filterset/>` may also be defined outside of a task and referred to by an id attribute, so that the filterset Target example can be rewritten as follows.

```xml
<filterset id="example-filterset"
  begintoken="@"
  endtoken="@">
```
After Ant version 1.5 a new functionality was introduced, FilterChains, which incorporates all of the power of <filters/> set within the concept of chaining together filters, expanding upon Ant’s capability to process text.

Understanding FilterChains

*Recipe: “create sophisticated find and replace pipelines”*

A pipeline, set in the context of UNIX commandline, is a sequence of one or more shell commands separated by the pipe character e.g. | . The following example shows how such a pipe maybe constructed from the command line:

> tail -n 100 /var/log/system.log | grep root

This command sequence will take the last 100 lines of a log file and print out just lines containing the term ‘root’. Pipes makes its possible to implement sophisticated workflows of text processing; it is one of the reasons why many developers like myself spend most of their time within a command line shell environment. Ant’s <filterchains/> are not exactly like unix pipes, but close enough for the purpose of an analogy.

Before v1.5, Apache Ant had no established facility for the piping of the output of an operation to be supplied as the the input of another operation. This shortcoming limited Ant’s native textual processing capabilities as I couldn't perfectly emulate command line ‘recipes’ that had been honed and tweaked over years.

Not having a standard mechanism for passing data around also meant confusion with respect to Ant’s own development, that is new functionality would commonly show up as a new attribute on a
Task … leading to an inconsistent set of attributes being progressively added to all Tasks by their authors.

‘Wouldn’t it be easier to leave Task attributes alone and implement some common mechanism with which to implement piping?’

Enter Ant <filterchains/> which the reader should imagine as an ordered chain of filterreaders (similar to <filters/>) each of which perform a process with an input and hand over the output to the next filter’s input.

Ant defines several native filters that work with <filterchain/>, which are in my opinion inconveniently called FilterReaders (probably mirroring their intention in Java terms).

In Apache Ant, one can create their own FilterReader or choose from the built-in FilterReaders, which are listed below.

- **FilterReader**: generic filterreader
- **ClassConstants**: outputs any found Java constants and outputs them into name=value format
- **EscapeUnicode**: convert input into Unicode format
- **ExpandProperties**: expand any property declarations into their values
- **HeadFilter**: output a number of lines at the beginning of a file
- **TailFilter**: output a number of lines at the end of a file
- **LineContains**: outputs a line which contains a certain string
- **LineContainsRegExp**: outputs a line which matches regular expression
- **PrefixLines**: prefixes each line with a defined string
- **ReplaceTokens**: similar to <filter/>
- **StripJavaComments**: remove Java comments
- **StripLineBreaks**: remove line breaks
- **StripLineComments**: remove line comments
- **TabsToSpaces**: convert tabs to spaces
- **DeleteCharacters**: delete specific characters
- **ConcatFilter**: prepend or append the contents of a file with another file

A `<filterchain/>` element will contain an ordered list of these FilterReaders. The only drawback with FilterChains, is its inability to define a reusable FilterChain (e.g. using a reference id attribute).

In the filterchain Target, I emulate previous `<filter/>` examples, by using the `<replacetokens/>` FilterReader.

```xml
<copy toDir="result/filterchain">
  <fileset dir="data">
    <include name="data.txt"/>
  </fileset>
  <filterchain>
    <replacetokens>
      <token key="name" value="John"/>
      <token key="version" value="1.7"/>
    </replacetokens>
  </filterchain>
</copy>
```

The embedded `<filterchain/>` contains a single FilterReader in a `<filterchain/>`, which doesn't quite illustrate the power of this approach. In the next example presents an example of FilterReaders processing text and handing off their output to the next FilterReader in series.

```xml
<copy toDir="result/multiple-filterchain">
  <fileset dir="data">
    <include name="data.txt"/>
  </fileset>
  <filterchain>
    <replacetokens>
      <token key="name" value="John"/>
      <token key="version" value="1.7"/>
    </replacetokens>
    <striplinebreaks/>
  </filterchain>
</copy>
```
The `<filterchain/>` performs textual substitution then strips line breaks with the `<striplinebreaks/>` filter.

A useful variant of `<filterreader/>` is the TokenFilter, which tokenizes an input stream into discrete line, string, or file tokens. Each token, in turn, can then be passed to other filters. I am specifically interested in the string tokenizers as one can process data on a string by string token basis using the following string filters.

Here is a list of string filters native to Ant.

- **ReplaceString**: replace a substring in a the string token
- **ContainsString**: output when string token contains a substring
- **ReplaceRegex**: output string token with regular expression substitution
- **ContainsRegex**: output string token which matches regular expression
- **Trim**: trim whitespace in string token
- **IgnoreBlank**: strips empty string token
- **DeleteCharacters**: delete characters from a string token

For ease of use, most stringreaders can be used directly in a filterchain, without having to explicitly declare a tokenfilter element.

**Tip**: You can use a `<scriptfilter/>` which will execute a script in an Apache BSF supported language, just like the `<script/>` Task.

**More Find and Replace tricks**

**Recipe**: “find and replace text using property expansion”

There is another `<filterchain/>` orientated approach to token replacement, that of using Ant properties as tokens for replacement operations. I hope that the following snippet is a familiar use of properties in Ant, as a method of text inclusion through property expansion.

```xml
<property name="lastname" value="Fuller"/>
<echo message="Dear Mr. ${lastname}"/>
```
Since properties are the data entity that developers get first accustomed to using in Ant, it’s only natural to want to avail oneself of these properties in text replacement operations within files.

Arguably, the easiest technique for doing this is to employ the `<expandproperties/>` filterreader in our handy `filterchain` element. The use of `<loadfile/>` task below illustrates how to use the `<expandproperties/>` filterreader, it is similar to previous examples, though I am now using Ant properties as tokens.

```xml
<property name="name" value="Danny"/>
<property name="version" value="1.6.5"/>
<loadfile property="mytext" srcFile="data/src.txt">
  <filterchain>
    <expandproperties/>
  </filterchain>
</loadfile>
<echo message="${mytext}"/>
```

The `<loadfile/>` Task ‘slurps’ up an entire file and places its contents into an Ant property. I have arbitrarily chosen this Task to illustrate using something other then `<move/>` or `<copy/>` Tasks, don’t get confused that the Task itself also sets a property.

Checking the contents of the `src.txt` file, contained in the example for this chapter, I find it containing tokens defined using property declaration syntax (e.g. `${name}`) instead of @ delimited tokens (e.g. `@name@`).

Dear ${name},

We would like to understand your opinions on using Ant ${version} the open source Java build tool.

regards, Apache

Running the `expandproperties` Target will print out the `mytext` property with the `${name}` or `${version}` properties correctly expanded.

*Buildfile: build.xml*
expandproperties:
  [echo] Dear Danny,
  [echo]
  [echo] We would like to understand your opinions on using Ant 1.6.5 the open
  [echo] source Java build tool.
  [echo]
  [echo] regards, Apache

If you have opted to parameterise your Ant scripts using <properties/> then this is a well-suited approach.

Tip: I recommend investigating the Ant <propertyset/> facility which groups a set of properties to be used by reference, as the <expandproperties/> FilterReader technique can be a bit tricky in situations, such as generating Ant build files themselves.

Managing Encoding and Whitespace

Managing character encoding, whitespace and control characters in text files can be a complicated issue.

For example, most modern programming languages do not consider white space characters, such as spaces, tabs and newlines, significant, usually just ignoring them during compilation or execution. Alternately in documents that contain XML and HTML, white space starts to become significant in the formatting of textual data, which is where many of the aforementioned headaches can start showing up.

The following outlines the meaning of ‘whitespace’.

XML: XML defines white space as the Unicode characters

- space (0x20)
- carriage return (0x0D)
- line feed (0x0A)
- tab (0x09)

Alternately, byte order mark and the non-breaking space (0xA0), are treated the same as visible characters.
**HTML**: HTML defines whitespace in http://www.w3.org/TR/REC-html40/struct/text.html#h-9.1 and http://www.w3.org/TR/REC-html40/struct/text.html#line-breaks. Whitespace is the individual characters

- #x20
- #x9
- #xC
- #x200B
- #x9
- #xD
- #x9 #xD pair.

Uniquely, whitespace in HTML, with the exception of `<pre/>` elements, is not contrastive, in other words whitespace characters are always considered to be equivalent to a single whitespace.

**<fixCRLF/> Task**

*Recipe: "manage translation of end of line, carriage returns, tabs, and end of file characters between operating systems”*

Unfortunately most operating systems have differences between how they interpret and generate control characters, which is somewhat related to what character encoding systems are in use. These incompatibilities tend to revolve around the following control characters;

- End of line (EOL)
- Carriage returns (CR)
- End of file (EOF)
- Tabs

Using Ant’s `<fixcrlf/>` Task, one can transform these characters into the correct representation for the current operating system or specifically define which control characters to use by supplying attributes to the Task. In its default settings, this Task corrects these control characters for the current operating system, with the notable exception for tab characters. The example below corrects
EOL, CR, and EOF characters. Additionally I have defined a tab attribute that instructs the Task to completely remove tabs.

<fixcrlf srcdir="data" includes="**/*.txt" tab="remove"/>

Here is another example where I start prescribing how to specifically handle EOL and EOF control characters.

<fixcrlf srcdir="data/ex2" tab="asis" eof="add" eol="unix"/>

This example would ignore any TAB characters, add an EOF character to the end of the file, and correct all end of line characters to conform to the UNIX platform convention. The <fixcrlf/> Task encapsulates FileSets and supports all attributes of <fileset>, such as shown with the <includes/> element. As of Ant version 1.7 the <fixcrlf/> may now operate as a FilterReader in a FilterChain, as illustrated below; all of the Task’s attributes are available in this form.

<copy todir="result">
  <fileset dir="data"/>
  <filterchain>
    <fixcrlf/>
  </filterchain>
</copy>

When FixCRLF is supplied with no attributes it will adjust text files to local conventions/platform.

<ignoreblank> FilterReader

Recipe: “fix line ending encodings”

An alternate method to using the <fixcrlf/> to fix line endings, is to use the <ignoreblank/> string fil-
terreader in a <filterchain/>.

<copy file="data/crlf.txt " tofile="result/crlf.txt">
  <filterchain>
    <tokenfilter>
      <ignoreblank/>
    </tokenfilter>
  </filterchain>
</copy>
Take care though when using this filter as the main purpose of `<ignoreblank/>` is to remove blank lines in a file versus solely fixing line endings.

**striplinebreaks**

*Recipe: “Compress a set of SQL statements”*

Linebreaks and spaces in SQL dump files can seriously slow down the performance of SQL insert and update operations when parsed. The following example SQL file would insert two users into a user table.

```sql
INSERT INTO users(id, firstname, lastname, email) VALUES
  (1, ‘Jacob’, ‘Smelt’, ‘jacob@example.org’),
  (2, ‘Jesse’, ‘Bleck’, ‘jesse@example.org’);
```

It should start becoming second nature to apply FilterChains to solve problems.

```xml
<copy file="data/user.sql" tofile="result/data.sql">
  <filterchain>
    <striplinebreaks/>
  </filterchain>
</copy>
```

**Working with character encoding**

How well can you answer the following questions concerning character encoding of your source files?

- Which character encoding is default on my operating system?
- Which character encoding is default in my editor?
- Which character encoding is defined for JVM on my operating system?
- Which character encoding should I explicitly encode my XML files?
- Which encodings should I define my Ant build files?

The answers to these questions can affect how your data gets parsed and understood by your programs. In this section, I present techniques on how to employ Ant as a first line of defense against such problems.
Text file Encoding

Recipe: "Output a specific encoding"

One somewhat superficial technique is to define an output encoding in conjunction with the <echo/> Task.

```xml
<echo message="message for output" file="output.txt"
       encoding="UTF-8" />
```

Of course this is limited to whenever one wants to print information to a file, standard console output or a log. If the reader requires to perform the opposite to this action, then consider the <load-file/> or <loadproperties/> Task.

File Operation Encoding

Recipe: “Preprocess source text files so that they are in the correct encoding and format for the current operating system”

In Ant, it’s easy to preprocess a set of files using the <copy/> task to ensure its properly encoding. This is achieved by defining an outputencoding attribute on the <copy/> Task itself.

```xml
<copy todir="result" outputencoding="utf8">
    <fileset dir="data">
        <include name=".*"/>
    </fileset>
</copy>
```

You may use a <presetdef/> or <macrodef/> to define default values for these types of attributes.

*Tip: the <move/> Task also has an outputencoding attribute.*

Handling Unicode

Recipe: “Process a set of text source files so all characters are encoded using Unicode”

Characters in most languages have historically been represented by single-byte values, restricting character sets to defining a maximum of 256 characters.
256 characters is not a lot of space to define any languages complete set of alphabet, digits, symbols, and punctuation symbols:

- New encodings to accommodate new languages are usually based on mutated versions of old encodings
- No standard approach to character encoding resulting in vendor and disparate standards activities to create their own proprietary encodings
- No single encoding, that is each language gets its own encoding

A complete solution to all of these problems is where Unicode has stepped in.

Unicode provides an unambiguous encoding of the content of text, covering most languages. Unicode is now in its fourth major version and is considered a mature standard. The standard is notable in that it’s an easy read.

Character and string classes in Windows 2000 and XP are now Unicode-based, with an associated set of APIs for working with Unicode. Java developers know that their character and string classes are Unicode-based, with the Java Class Library providing an extensive set of APIs for working with Unicode.

There exist many high quality third-party packages, including the open-source International Components for Unicode, for working with Unicode.

Unicode covers all of the modern European and America’s languages, in addition to Eastern and Asian character sets in use today. Additionally Unicode, or UTF-8 as it is referred to programmatically, provides good coverage to a wide variety of symbols, punctuation, and notations.

Here is a list of what Unicode defines;

- **Punctuation and symbols**: Numerals, math symbols, scientific symbols, arrows, blocks, geometric shapes, Braille, musical notation, etc.
- **European**: Latin, Greek, Cyrillic, Armenian, Georgian, IPA
- **Bidirectional** (Middle Eastern): Hebrew, Arabic, Syriac, Thaana
- **Indic** (Indian and Southeast Asian): Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Sinhala, Thai, Lao, Khmer, Myanmar, Tibetan, Philippine
- **East Asian**: Chinese (Han) characters, Japanese (Hiragana and Katakana), Korean (Hangul), Yi
- **Modern**: Mongolian, Ethiopic, Cherokee, Canadian Aboriginal
Historical: Runic, Ogham, Old Italic, Gothic, Deseret

Now I move on to presenting Recipes related to managing your source files character sets.

Note- The Unicode standard is available at http://www.unicode.org. The author can recommend ‘Unicode Demystified’ by Richard Gillam and ‘Unicode: A Primer’ by Tony Graham as examples of authoritative works on the subject.

Escape Unicode FilterReader and Native2Ascii tasks

Recipe: “task orientated unicode conversion”

For a Task orientated approach, Ant has a built in Unicode conversion facilities in the form of the <native2ascii/> task and <escapeunicode/> filter, which are directly based on Sun’s ‘native2ascii’ java utility.

Both of these Tasks will consume text, from either a file or string, and convert any characters not defined in the US ASCII encoding format into what is known as escaped Unicode.

The <native2ascii/> Task modifies a file in place, replacing any character not represented within the current encoding format for that file.

```
<target name="native-2-ascii">
  <native2ascii encoding="ASCII" src="src" dest="dest"
    includes="**/*.txt"/>
</target>
```

After Task execution, the reader should notice that any character not represented in the current encoding format for that file, will now be in escaped Unicode format. Here is an example of escaped Unicode, converting the English currency symbol (£) into its Unicode representation:

£ = \u00a3

Whenever such a conversion is performed its important to understand that it is done in the context of the current character encoding. Lets now investigate what Ant does with 3 differently encoded files ( ASCII, UTF8, and UNICODE). The build for this example will process 3 separate text documents using the <escapeunicode/> filter.

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Manage Unicode" basedir="." default="build">
  <description>Manage Unicode</description>
</project>
```
Running this example will convert 3 differently encoded files, escaping any characters that can’t be represented in the character encoding.

*Note*-You maybe surprised by the results, generally if you are working with Western European languages and want to play safe on most platforms I recommend always saving all text data in UTF-8 format.

This form of escaped Unicode is naturally parsed in Java, though do not assume that escaped Unicode will be automatically understood using other parsers and processors.

*So how can I get this type of escaped Unicode into an XML and HTML friendly format?*

One approach is that Unicode can be given as a numeric representation, shown below:
This type of representation will render properly in browsers and be understood in most XML and HTML parsers. I can take escaped Unicode encoded files and use regular expressions to put Unicode into a numeric representation, as shown below using a `<replaceregex/>` FilterReader.

```xml
<filterchain>
  <tokenfilter>
    <filetokenizer/>
    <replaceregex pattern="\u. (.*)
    flags="g"
    replace="&\1; ">
      
    </replaceregex>
  </tokenfilter>
</filterchain>
```

The example build contains this in a macrodef, that combines the process of first escaping Unicode and then converting the escaped Unicode to numeric entities.

*Note: In both Chapter Web Recipes and Chapter XML Recipes I make use of the Jtidy and TagSoup Java libraries to scrub HTML and XML into clean, valid, and schema valid formats.*

**Facilitating Localisation**

**Recipe: “Transform a set of source text files into a range of language variants”**

The process of developing software into multiple language variants is known as localization. I like to segregate localization into ‘build time localisation’ and ‘run time localisation’.

Build time localization generates a specific language version whenever the software is compiled, built, tested, and packaged … run time is the dynamic generation of language variant text.

**Replace Task**

Localization using `<replace/>` is a straightforward process; one can choose at build time which language variant by setting an Ant property and then using the property to reference an external filter file.

Here’s how the target would look like for a replace task approach to localisation;
Replace has a number of attributes with which to control its behavior, e.g. I have set the summary attribute to yes because I would like the operation to print out useful information about the number of strings found and replaced.

Localization with the replace task can be applied in those situations where it’s clearly the last step before deployment; though admittedly this is somewhat limited in that it cannot replace text as well as rename or copy the file all in one step.

The <copy/> task could use a <filterchain/> or <filter/> elements, shown below.

```xml
<copy file="index.html" tofile="index-processed.html">
  <filterset>
    <filter token="@datepublished@" value="${start.TSTAMP}"/>
    <filter filtersfile="replace.properties"/>
  </filterset>
</copy>
```

**Tip:** The Replace task does not have an output encoding, true you can instruct the task to ‘assume’ a certain encoding, but there is no guarantee that this will result in a properly encoded file after the replace operation has taken place, which is another reason why using a filterset or filterchain with <copy/> maybe a better option.

**Translate Task**

Using the <replace/> task is still not full featured enough for proper language localization; for example there is no fallback concept if a certain language variant doesn’t exist.
Instead I want a flexible mechanism that separates the code or text one is processing from the localization definition itself. Ant has an optional task that ships with it called the `<translate/>` task, and it provides the common Java approach to localization reusing the concept of external property files.

I might as well just dive into a complex example using the `<translate/>` task, where I have a few languages that will replace source text files in a `/src` directory, and copy to its own directory under a `/dist` directory.

```xml
<translate toDir="result/de"
    starttoken="@"
    endtoken="@"
    bundle="resource/local"
    bundlelanguage="de"
    forceoverwrite="yes"
    srcencoding="ISO8859_1"
    destencoding="ISO8859_1"
    bundleencoding="ISO8859_1">
    <fileset dir="data">
        <include name="**/*.xml"/>
    </fileset>
</translate>
```

As with our filter type tasks, the above example searches for tokens delimited with the `@` character. The task will then look up the deutsch version resource files, as specified by the `bundlelanguage` attribute, and then search and replace using the name-value pairs. All this should feel familiar, as this is how things pretty much well work with the `<replace/>` example though with a few added benefits like being able to explicitly determine encoding or handle fallback situations where a local language variant.

Let's review all of the `translate` attributes:

- **`toDir`**: defines destination directory
- **`begintoken` and `endtoken`**: defines `@` to be used as both the beginning and end character marking a string as a token to be replaced
- **`bundle`**: contains location of external language property files and defines the naming prefix resource files themselves
- **`bundlelanguage`**: defines the current language to be processed, in our case it is Deutsch
• **forceoverwrite**: if set to yes, will force the overwriting of files contained in the toDir defined directory

• **srcencoding, destencoding, and bundleencoding**: defines the character encoding of all the files, I have set all of them to ISO_8859_1.

Taking a look at the example in the download for this chapter, will reveal that there is indeed a resource directory that contains a set of property files, with name-value pairs. Since I do not speak German, I have resorted to tagging each defined property in English though stating it as Deutsch if appropriate; so when selected I will know which language is intended.

The mechanism which makes this all work, is the file naming convention applied to the resource files. The bundle attribute defined both the directory, e.g, /resource, and naming prefix ('local') of the resource files. When the translate task executes it searches the resource directory for a resource file with the appropriate language. The resource file format can accommodate differences in the language due to country; for example the french spoken and written in Paris, France is much different then that spoken in Quebec province of Canada. Additionally, there can be further variations of a language, either by region or local dialect.

The resource file format is thusly composed of the bundle prefix, plus the primary language, and variations for country or local dialect, all separated by underscores and appended with .properties.

For example, if I defined the translate task to render a Swiss German version, I would have to provide a file named; local_de_ch.properties. If this resource file did not exist, then the translate task would search for the next best match; local_de.properties. Finally, if there are no matches, the default property file, e.g. local.properties, will be used.

To summarise, the naming convention for resource files is as follows;

```
prefix + "_" + language1 + "_" + country1 + "_" + variant1
```

These files are exactly like external property files, and contain name-value pairs.

Obviously to match tokens equivalently the names must all be the same, in every resource files, which equal the language variant value.

I will need to call the translate task for each language, so I usually create a macrodef task which should make using the translate task easier. I have found that macrodef’s make your Ant build files easier to read, as well as making them more configurable.

```xml
<macrodef name="localise">
    <attribute name="lang" default="en"/>
</macrodef>
```
I have no need to explicitly define any different character encodings though I have defined a top-level property, which I use in the macrodef.

Additionally, I have made a few other assumptions in creating this macrodef:

• generate a /dist directory
• place resource files in a /resource directory
• always prefix resource files with ‘local’

With the macrodef defined, the call within our target is as follows;

Let’s wrap up everything into a single target to illustrate the complete example;
The macrodef has abstracted most of the details for the &lt;translate/&gt; task resulting in something altogether simple; e.g. if I need to tweak our translate task its just a matter of amending the macrodef once.

Advanced Text Processing

Recipe: “Advanced text processing with Jacson Ant Task”

If Ant’s built in capabilities are not full featured enough for your textual manipulation requirements you can always resort to something stronger. The 3rd party Jacson Ant Task might just be the solution.

http://jacson.sourceforge.net

Jacson is described at sourceforge.net thusly;

‘Jacson provides a pluggable and configurable alternative to UNIX tools like grep, awk, tail, head, cut, sort, uniq or even some perl based scripts in a Java environment. A bit like what Apache Ant means to good old make.’

Jacson is a sophisticated utility that can be applied to the analysis and manipulation of text files. If you need to work with large complex text files, like web logs, Jacson provides a complete tool-set.

Jacson can be executed from within your Ant builds as either a task or can be called upon as a filter. Either method must refer to an external configuration file which contains a series of processing steps one would want to apply to a set of source files. The Jacson example illustrates how it the task form is invoked.

<target name="jacson">
   <property name="count" value="10"/>
   <jacson config="head.conf" out="jacson-out.txt" in="access.log" />
   <echo>count="${count}"</echo>
</target>
This target prints out the first 10 lines of the access.log file to the jacson-out.txt file. The <property/> defines how many lines to print out. The build file defines a property that is used by Jacson; though it is up to us to know the property name.

head.conf

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<application>
  <se:param se:name="count" se:value="2"/>
  <se:param se:name="debug" se:value="false"/>
  <jacson>
    <filter se:this="HeadFilter" count="${count}"/>
    <eval se:this="LoggingEvaluator"/>
    <report se:this="ValuePrintingReport">
      <section>lines/log/chunk</section>
      <debug>${debug}</debug>
    </report>
  </jacson>
</application>
```

The configuration file defines parameters, filter operations, and how text is printed or reported on. The Jacson download comes with many example applications so you can learn how to create your own.

To use as a filter then you would refer to its classname and supply the desired configuration file via a <param/> element.

```xml
<filterreader classname="de.spieleck.app.jacson.ant.JacsonAntFilter">
  <param name="config" value="head.conf"/>
</filterreader>
```

If you are now wondering why one would use an Ant task that requires you to define filters in a completely different XML vocabulary, then you are either comfortable with extending Ant or to be frank your needs do not require such a drastic solution.

There is overlap between what Jacson and Ant does, though if you need something akin to scripting level control of text manipulation operations, then this task may be for you.
Conclusion

There are a lot of mature software packages that provide excellent text processing capabilities. Ant can be a useful adjunct to these packages and really comes into its own when you want to explicitly manage certain aspects like character coding or dealing with issues when porting data from one system to another.

Ant can also provide scripting solutions with a bit more structure, over plain shell scripting, which is handy when you are doing something that could evolve over time and be worked on by many developers.
Database Recipes

Apache Ant can help you manage, sync and organize data within relational databases and help automate tasks such as importing and exporting data.

- **Mysql import macrodef**: Learn how to manage SQL-based data stored within relational database tables or as SQL dump files, as well as learn how to import data into Mysql.

- **Accessing SQL during an Ant build**: query and use SQL interactively in your scripts

- **The SQL task**: Full blown integration with relational databases

All the recipes in this section will require you to install and run some sort of Relational Database (ex. mysql, postgresql) and its likely that you will have to amend examples with your database connection details.

**Mysql import macrodef**

*Recipe: "Manage import of csv delimited data into MySQL"*

MySQL ships with an import utility called mysqlimport. It can be invoked from the command-line and will parse delimited files of various formats then dump the data into an existing MySQL table. The import tool is roughly equivalent to the ‘LOAD DATA INFILE’ SQL statement, and relies on the fact that the text file must have the same name as the table into which the data should be imported.

In our example, I wish to import tab-delimited product data into an existing database named example. Now is the time to ensure that MySQL is installed and operating properly.

The next step is to create a database named ‘example’. Easy enough, via the mysqladmin command: mysqladmin create example.

```xml
<macrodef name="import_text">
  <attribute name="dbname"/>
  <attribute name="txtfile"/>
  <sequential>
    <echo message="importing @{txtfile} into @{dbname}"/>
    <exec dir="." executable="${exe.mysqlimport}">
      <arg line="@{db.name} @{txtfile}"/>
    </exec>
  </sequential>
</macrodef>
```
The mysqlimport file has been wrapped in a macrodef, which should make the task run faster as well as making it easier to reuse.

The catalog-import Target shows the macrodef in action, importing data from product.txt file.

<target name="catalog-import">
  <import_text dbname="${db.name}" txtfile="data/product.txt"/>
</target>

Note: Most relational databases have an equivalent utility so the technique presented can be easily ported.

Accessing SQL during an Ant build

Recipe: "Use the result of a SQL query as properties within your Ant build"

Sometimes you need to query a database and use the results directly during your Ant script execution. I present a simple way of converting the results of a SQL query into Ant properties, based upon the ability of most popular databases to now output directly to an XML format.

For example, the MySQL dump facility, mysqldump, can do just this and I illustrate its invocation below within an Ant <exec/> Task.

<exec executable="mysqldump" spawn="true" output="result.xml">
  <arg line="-xml"/>
</exec>

The next step is to extract the XML back into Ant by using the <xmlproperty/> Task.

<xmlproperty file="result.xml" semanticAttributes="true"/>

With the entire database available as properties, it is now facile to access data. If the XML structure is not known before execution, one can make good use of the 3rd party Ant library, Ant-Contrib’s <for/> or <foreach/> Task to iterate through each property.
The SQL task

Recipe: "Execute SQL statements in Ant"

Ant comes with a standard SQL task from which you may execute SQL statements embedded within your build scripts or provide source files that contain SQL statements to be processed.

You must have a JDBC connector installed and available for `<sql/>` to work properly. I will be using com.mysql.jdbc.driver from MySQL official website for use with a MySQL database for our examples shown here.

The proceeding example shows how to invoke the task with database specific details such as a username and password to access database, driver class name, if I would like to save the output to a file, etc...

```
<sql
    driver="com.mysql.jdbc.Driver"
    url="jdbc:mysql://127.0.0.1/mysql"
    userid="root"
    print="true"
    password=""
    output="result/test.out">
    <transaction>
        select * from user;
    </transaction>
    <classpath>
        <pathelement location="/wherever/your/jdbc/connector/jar/is/"/>
    </classpath>
</sql>
```

You will need to supply the example script with your own database details. The above invocation of the `<sql/>` Task should access a local mysql user table, which should be installed with your default MySQL installation.

The SQL statement I intend to run against this table is contained in a nested `<transaction/>` element.
The SQL Task has many options and attributes, so please refer to the Ant manual. One other item of note, is the ability to instruct the <sql/> task to look for a Java jar or set of class files to provide the JDBC driver for your ‘flavor’ of database.

Our next example, builds on the previous Ant script, showing how I can have a series of transactions, supplied from external files containing SQL statements, as well as mixing in SQL statements embedded in the <sql/> task itself, as before.

The use of the <transaction/> element can be used to specify embedded SQL statements or load in SQL statements from an external file.

```
<sql
    driver="com.microsoft.jdbc.sqlserver.SQLServerDriver"
    url="jdbc:microsoft:sqlserver://localhost\myinstance:1433;Databasename
userid="root"
    print="true"
    password=""
    output="test.out">
    <transaction src="data/somfile.sql"/>
    <transaction>
        select * from myTable;
    </transaction>
    <transaction src="data/someotherfile.sql"/>
</sql>
```

This example tries to connect to a Microsoft SQL server and then attempts to execute SQL statements from a source file named ‘somfile.sql’, then it performs the embedded SELECT statement. Finally, another file’s SQL statements would be processed.

The <sql/> statement does support the concept of Ant <fileset/>, though it is recommended to use the <transaction/> element, which guarantees the order of processing with each set of SQL statements.

A note about XML Database

Relational Databases are not the only databases in use today. Object databases, hierachical databases and key-value stores are all experiencing a resurgence in usage.
For example, the popular XML Database eXist is an Open Source database which lets you save XML documents in their native format and perform XML processing (xpath, XQuery, xslt, etc). More interestingly, eXist provides a complete Ant library for interacting with it.

To use these Tasks on a running instance of the eXist XML Database you will first need to use a TaskDef.

```xml
<typedef resource="org/exist/ant/antlib.xml"
uri="http://exist-db.org/ant">
  <classpath refid="classpath.core"/>
</typedef>
```

Where the `<classpath/>` has to been defined as:

```xml
<path id="classpath.core">
  <fileset dir="${server.dir}/lib/core">
    <include name="*.jar"/>
  </fileset>
  <pathelement path="${server.dir}/exist.jar"/>
  <pathelement path="${server.dir}/exist-optional.jar"/>
</path>
```

With these eXist Ant tasks I can:

- Store XML Documents
- List Resources/Collections
- Copy and move Resource/Collection
- Process an XPath or XQuery Expression
- Extract Resource/Collection
- Backup & Restore
- List Groups & Users
- Add and Remove User

Please refer to eXists own documentation (http://exist.sourceforge.net), but I will show one example of how to store an XML Document.
I need to make sure that the xmlns:xdb namespace is declared somewhere (I have done this on the Task itself but it could be on the <project/> element).

**Conclusion**

The broad set of topics for this chapter will mean that for some readers I will not have discussed about their particular data scenario, or presented examples using their ‘brand’ database. If this is the case, Ant’s cross platform nature should help you out making it easier to port the examples to your particular platform.

I also thought it was useful to show in this chapter that Ant works with other types of databases. I chose the eXist XML Database as I am an active developer, but I have seen plenty of alternative databases provide an Ant task for integration so make sure you check first if your database already has one developed.

I continue working with Data in the next chapter, XML Recipes, where techniques are shown how to work with XML.
XML Recipes

XML has been in existence for over 10 years and continues to experience adoption across the ambit of computing, augmenting and sometimes even eclipsing solutions built with relational databases.

I never advocate throwing any tool out of the ‘toolbox’ (like RDBMS), though XML is becoming the right choice, especially when you need to manage semi structured data.

This chapter presents Recipes on how to process and validate XML, furthermore I will teach readers advanced techniques on how to manage Open Source XML Documents (Office XML, Docbook, etc).

• **The Basics**: Manipulate, generate and integrate XML processing technologies

• **Working with XSLT**: Use Ant to manage XSLT processing

• **Validating XML**: I show how to use Ant’s built in schema Tasks, `<xmlvalidate/>` and `<schemavalidate/>` for validating and XML document. I then introduce the use of the Jing processor and how, with Ant, it can manage multiple schema technologies using the Namespace Routing Language(NRL)

• **Open Office XML Documents**: A few recipes working with Docbook or any of the various XML Office type formats (Sun, Microsoft, OS)

---

The Basics

Ant has a lot of built in and 3rd party Tasks that permit a range of XML processing. I will start with showing you how to get XML into and out of Ant, then I can move onto more sophisticated topics.

Creating XML

*Recipe:* “Generate XML directly from an Ant script”

Ant makes it easy to generate XML directly from within Ant using the `<echoxml/>` Task. The `<echoxml/>` Task will output nested XML to the console or file as shown below.

```xml
<echoxml file="example-output.xml">
```
When the Task is defined using an append attribute it will write output results to the end of an existing file.

Unfortunately there is a significant limitation with the <echoxml/> Task, in that it does not understand XML Namespaces. One way around this is to fallback to the standard <echo/> Task to create XML files, though you will need to wrap content in CDATA sections which is admittedly not very nice.

Note: If you are familiar with XML … you may find some strange things with respect to Ant’s own xml scripting format. There have been plenty of debate mulling over the merits of Ant sticking with an XML syntax to describe the build. Perhaps the best way to view these XML idiosyncrasies is that Ant was an early adopter of XML, though for XML power users you may notice that Ant is naive to common XML idioms (e.g. XML Namespaces, etc…).

### Loading XML into Ant

Recipe: “Load external XML into Ant Properties”

Getting XML data into Ant can be done using the <xmlProperty/> Task. This native Task parses a well formed XML file initializing Ant Properties with property names equivalent to their path within the hierarchical XML element structure.

The <xmlProperty/> Task is defined with a file attribute, as shown below, in the example1 Target.

```xml
<xmlProperty file="example.xml"/>
```

When the Ant Task is executed, it loads up the XML and parses names and values; emulating setting properties as you would with a <property/> Task.

```xml
<property name="p.ip" value="127.0.0.1"/>
```

You may run the example1 Target, with the commandline -debug flag so you can view how Properties are set by Ant via the console output.

```bash
ant -debug example1
```
Check the generated result/set.properties file which will contain all the properties; this output was created using the <echoproperties/> Task.

<echoproperties destfile="result/set.example1.properties"/>

Using <xmlproperty/>Task, the XML attributes also get set as Ant Properties, using a slightly different naming convention. The data/property1.xml file has a few examples of this.

<p>
<dir base="/usr/local">
<doc type="html"></doc>
<etc>etc</etc>
<htdocs>htdocs</htdocs>
<publish>${someotherdir}</publish>
</dir>
</p>

Now run the example build script with Target example2.

Buildfile: ../1_basics/2_xmlproperty/build.xml

example2:

[echo]
[echo] p.dir(base): /usr/local
[echo] p.dir.publish: ${someotherdir}
[echo]
BUILD SUCCESSFUL
Total time: 0 seconds

The text in bold, within the code listing above, highlights how attributes get named

p.dir(base)

Unlike loading plain Property files, Ant will not expand already set Property references embedded within the XML, this is shown with the value of p.dir.publish Property (e.g. ${someotherdir}).

You can vary how XML attributes get named by setting the collapseAttributes to true which forces <xmlproperty/> to apply the same rules used for element naming to attribute naming. For example, the following attribute would normally be named;

p.dir(base)
When collapseAttributes is enabled, the attribute’s property name changes to

\[ p.dir.base \]

**SemanticAttributes**

What if I want to have a Property defined with any of the other valid Property attributes, such as an id, refid or location?

\[
<property name="p.dir.base" location="c:\apache\htdocs"/>
\]

The `<xmlProperty/>` Task accomplishes this by providing a semanticAttributes attribute, which when set to `true` will match named attribute in the loaded XML document.

\[
<xmlProperty file="data/property2.xml" semanticAttributes="true"/>
\]

The example3 Target, in the build script, shows how this works using the property2.xml XML document, which as you can see defines attributes matching desired Property attributes.

\[
<p>
	<dir>
		<xslt location="xslt"/>
		<xml id="testid">xml</xml>
		<bin location="bin"/>
	</dir>
\</p>
\]

Check out the property names in the `result/set.examples3.properties` to understand how this alternate naming convention works.

**Other `<xmlProperty/>` Attributes**

There are a few other helper attributes to the `<xmlProperty/>` Task listed below:

- **prefix**: prepend each Property with a prefix
- **keepRoot**: stop the root element from being used in the naming of Properties.
- **validate**: if this is set to `true`, then any DTD or XML schema that the XML file references will be used to validate the XML document before instantiating Properties.
• **includeSemanticAttributes**: if semanticAttributes is enabled, then this flag sets the semantic attribute name (e.g. value, location, pathid, refid) to be appended to the Property name.

• **rootDirectory**: when using semanticAttributes, this flag tells Ant to set a base directory from which location and path like structures will be relative to.

**Handling duplicate elements**

When the XML document contains duplicate elements with different values, `<xmlProperty/>` will create a Property containing comma separated values.

The build example4 Target illustrates this behavior with the handling of duplicate elements using the property3.xml file which contains several log elements.

```xml
<p>
  <log id="1" file="test1.html"/>
  <log id="2" file="test2.html"/>
  <log id="3" file="test3.html"/>
  <log id="4" file="test4.html"/>
  <log id="5" file="test5.html"/>
  <log id="6" file="test6.html"/>
</p>
```

As you can discern from the last reference in the Ant output for the p.log(file) Property, each `<log/>` element value is now contained in that Property.

```
[echoproperties] p.log(id)=1,2,3,4,5,6
```

**Recipe: “For loop processing in Ant”**

I use `<xmlProperty/>` for lots of Property loading. I frequently find myself needing to parse a Property containing a string of comma delimited values and do some work based on the value of each item.

This can be done using the 3rd party AntContrib `<for/>` Task which will loop through each value. The build Target example5 shows how to do this, using the previously defined `${p.log(file)}` Property which contained multiple log elements.

```xml
<xmlproperty file="data/property3.xml"/>
  <for list="${p.log(file)}" param="srcfile">
  ```
The Target loads Properties from the data/properties3.xml document, then using the <for/> Task I generate a new file, under the /result directory, for each comma delimited value.

Note: Both the <for/> and <xmlproperty/> Tasks can be processor intensive, best to apply when you have smaller data sets.

**Editing and updating XML**

*Recipe:* "Perform update operations on existing XML documents"

The <xmltask/> Task allows one to perform all manner of operations on XML documents e.g. insert, append, remove and update operations on element and attribute nodes.

This Task is a good starting place for new Ant users looking to perform any kind of XML document manipulation. You can download the latest version (v1.13) of the Task from:

http://www.oopsconsultancy.com/software/xmltask/index.html

**Transclusion**

*Recipe:* "How to include an XML document into another using Xinclude"

When working with XML documents it is a common requirement to include a portion or the entire contents of one XML document into another XML document. This process is generically known as transclusion.

Ant's own <import/> element is a good example of achieving transclusion, though specifically on Ant build files.

Other XML based languages employ their own proprietary instruction for processing transclusions, For example, XSLT defines an xsl:include and xsl:import instructions and lets not forget to mention that most popular schema languages (ex. RelaxNG, Schematron and XML Schema) have a similar convention.
Transclusion is a good way to build up modular reusable blocks of code and data. Xinclude is a specification that is meant to simplify things by defining a single method of performing transclusion across any kind of XML document, instead of having to resort to a proprietary approach.

The following XML document shows how Xinclude instructions are used.

```xml
<?xml version="1.0"?>
<sequential>
<echo test="instructions from example.xml document"/>
  <xi:include
    href="test.xml"
    xmlns:xi="http://www.w3.org/2003/XInclude"/>
</sequential>
```

In the above case, the Xinclude statement is saying to include the entire test.xml document. If test.xml consisted of a single `<root/>` Element, I would have the resulting XML document after processing with some Xinclude processor.

```xml
<?xml version="1.0"?>
<sequential>
<echo test="instructions from example.xml document"/>
  <root/>
</sequential>
```

**Using LibXML**

One way to carry out Xinclude processing is to use the popular LibXML and LibXSLT library which exists on most UNIX based operating systems, including Mac OSX.

If you are on a Windows based system you can download executables at

http://libxslt.org

LibXSLT xsltproc (used for XSLT processing) and xmllint utility can process Xinclude statements. Since I am interested in just Xinclude processing I will use the xmllint command.

Below I have created a short macro, wrapping up the LibXSLT commandline xmllint command and enabling Xinclude processing using the -xinclude flag.
In the example1 Target, I have defined the MacroDef to run against a fictional web distribution, specifically the index.html file, which is contained under /web/en directory.

/web
    metatags.xml
    /en
        index.html
        banner.xml

The index.html shows two different examples of how XInclude can prescribe either the entire XML document, or just a part of an XML document. Taking the simpler of these two statements, I see how to include an entire file’s contents, in this case the banner.xml file; this is an example of when you want to include a common bit of navigation into all of your web pages, making it easier to maintain.

<xi:include href="banner.xml"/>

Straightforward...not so with the other XInclude statement which is defined just to include <meta/> elements from the metatags.xml file contained in the directory above index.html.

<xi:include
href="../metatags.xml
    xmlns(x=http://www.w3.org/1999/xhtml)xpointer(/x:html/x:head/x:meta)/"/>
XInclude uses XPointer syntax to define what gets selected from XML document defined by the href attribute. And much added verbosity is required, if there are any XML namespaces in affect in this document, as you will need to define them as well so the XPointer selects them properly.

To see the XInclude processing, in action, now run the example1 Target.
>ant example1

Buildfile: build.xml
example1:
  [exec] <?xml version="1.0"?>
  [exec] <!DOCTYPE html PUBLIC "+//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  xml:lang="en" lang="en">
  [exec]   <head>
  [exec]     <title>Example Website</title>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="published" content="@some-timestamp@"
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="version" content="@some-version@"
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="ROBOTS" content="ALL"
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="copyright" content="Copyright 2006 James Fuller, All Rights Reserved"
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="Expires" content="Tue, 01 Jun 1999 19:58:02 GMT"
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="ROBOTS" content="NOARCHIVE"
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="DESCRIPTION" content=""
  xml:base="web/metatags.xml"/>
  [exec]     <meta xmlns="http://www.w3.org/1999/xhtml" name="KEYWORDS" content=""
  xml:base="web/metatags.xml"/>
  [exec]   </head>
  [exec]   <body>
  [exec]     <div id="wrapper">
  [exec]       <div id="banner" xml:base="web/en/banner.xml">
  [exec]         <div id="Logo">
  [exec]           ExampleLogo
  [exec]         </div>
  [exec]         <div id="bannerTitle">Example Website</div>
  [exec]       </div>
  [exec]       <a href="#" target="_blank">XML Recipes</a>
  [exec]     </div>
  [exec]   </body>
  [exec] </html>
This MacroDef, using xml lint, outputs the result of processing direct to the console, which is the index.html file with it's two XInclude statements processed, as evidenced by the existence of meta-tags and top banner div element.

You could amend the MacroDef to save these results to a file as well.

The only other thing I need to take care of is how to process multiple files...which I do using Macrodef <file-elements/>, and replace the <exec/> Task with the <apply/> Task.

And here is example usage of the task.

<xinclude-better>
I have improved on the original MacroDef by adding the ability to process multiple files; `<xinclude-better/>` can now use a nested `<fileset/>` to prescribe files (and use good stuff like FilterChains and IO Redirectors).

Using this better Macrodef, I now prescriptively apply XInclude processing to XML documents of our choice.

Running the example2 Target will process on index.html, as before, as well as the example.xml.

```
>ant example2
```

The example.xml file is a demonstration of composing Ant builds files, when you do not want to use `<import/>`. Though be aware that you get none of `<import/>` special handling, that is, Target overrides, selectively accessing Properties or files being referenced from anywhere but the main build file.

**Using Xerces XML Parser**

Xerces, the Apache Software Foundation’s XML parser that comes bundled with Ant, has an implementation of XInclude processing built right in. You can instruct the Xerces XML Parser to process XInclude by default whenever the XML Parser is used.

Be careful with this approach as you are changing the default global behavior of the XML Parser, not just within Ant but across any application that uses the XML parser.

To turn on XInclude processing, you need to create a `xerces.Properties` file which contains the following `java` system Property, defined as below;

```
org.apache.xerces.xni.parser.XMLElementConfiguration=
   org.apache.xerces.parsers.XIncludeParserConfiguration
```

Put the file, containing the single property, in your Java JRE `/lib` directory, which should be somewhere on your hard drive.
When you do XInclude processing this way, you will not need to invoke any special Task within Ant.

You can try this out, by enabling the example3 Target, or on your own build scripts that use anything that invokes the XML Parser (ex. `<echoxml file=""/>` or `<xmlproperty/>`). This is also applicable to XSLT processing that Ant can control...which is the next XML topic where I discuss how to use the Ant’s native `<xslt/>` Task.

Note: Implementations of XInclude can adhere to earlier versions of the XInclude specifications then the current draft, causing the odd problem here and there; these are mostly to do with how namespace bindings are defined and the use of fragment identifiers. Use with caution.

**<parsersupports/> Condition**

Recipe: ‘Use conditions to check what the XML Parser supports’

To know if your underlying XML parser supports any specific features (like xinclude processing) you could take advantage of Ant’s `<parsersupports/>` Condition.

The following Condition tests for features most SAX2 parsers should have.

```xml
<condition property="supportnamespaces">
  <or>
    <parsersupports feature="http://xml.org/sax/features/namespaces"/>
  </or>
</condition>
<echo>
  Your parser supports XML Namespaces: ${supportnamespaces}
</echo>
```

and will echo out the property value, ${supportnamespaces}. Similarly, to check for XML Schema support run example2 Target.

```xml
<condition property="supportschema">
  <or>
    <parsersupports feature="http://apache.org/xml/features/validation/schema"/>
    <parsersupports feature="http://java.sun.com/xml/jaxp/properties/schemaSource"/>
  </or>
</condition>
<echo>
```

52
Which uses the <parsersupports/> condition to check schema features.

This condition can also check for parser specific features, for example Xerces can set a default location for the no namespace schema.

<condition property="supportxerces">
<or>
<parsersupports

property="http://apache.org/xml/properties/schema/

external-noNamespaceSchemaLocation"

value="document.xsd"/>

</or>

</condition>

Your parser supports external-noNamespaceSchemaLocation: ${supportxerces}

Working with XSLT

XSLT is one of the best ways of processing XML. Using Ant to control XSLT processing is a good match and plays to each technologies strengths, resulting in easier to maintain XSLT code whilst leveraging all the power of Ant’s tasks, in a nice cross platform package.

<xslt/> Task

Recipe: “Simple XSLT Processing with Ant”

There exists an <xslt/> Ant Task which is used to control XSLT processing from within Ant. As Apache Ant comes bundled with the Apache Software Foundation’s XSLT processor, Xalan, you can use the <xslt/> Task with a standard install of Ant.

The example1 Target does just this, taking data/test.xml and transforming using the test.xsl stylesheet and generating a file named result/test.xalan.out.

<xslt in="data/test.xml" out="result/test.xalan.out" style="test.xslt">
The output from running this Target will be a rudimentary html document listing out the processor that transformed it (using XSLT system-property() function).

<pre>
<html>
<body>
<p>
  Version:1.0<br>
  Vendor:Apache Software Foundation (Xalan XSLTC)<br>
  Vendor URL:http://xml.apache.org/xalan-j
</p>
</body>
</html>
</pre>

If you wanted to use another java XSLT processor you will need to explicitly tell where the XSLT processor lives. I recommend using Michael Kay's XSLT Processor Saxon as it is a superior implementation containing both an XSLT v1.0 and V1.0 implementation (it also has its own Ant Task if you care to use it).

The <xslt/> Task accepts a classpathref attribute which I supply with an Ant Path datatype which in turn points to the lib directory containing saxon XSLT processor jars.

<pre>
<path id="saxon.path">
  <fileset dir="../../lib/saxon">
    <include name="**/*.jar"/>
  </fileset>
</path>
</pre>

This path is referred to in the example1b Target <xslt/> invocation.

<pre>
<xslt in="data/test.xml" out="result/test.saxon.out"
  style="test.xslt" classpathref="saxon.path">
  <outputproperty name="encoding" value="utf8"/>
  <outputproperty name="indent" value="yes"/>
</xslt>
</pre>

I could use a <presetdef/> to define a Task that always uses SAXON, or alternatively a Macrodef as shown in example2 Target. The <macrodef/> is defined in a separate file called xslt-antlib.xml.
The example2 Target just needs to invoke the new Task `<saxon/>`.

```xml
<saxon xslt="xslt/test.xsl"
      xml="xml/test.xml"
      dest="out"
      ext=".out">
</saxon>
```

**Recipe: "More sophisticated XSLT processing"**

I find using the `<xslt/>` Task can be problematic if you intend to do more than basic XSLT processing. One way around this is to wrap up an XSLT processor call using the java task.

```xml
<macrodef name="saxon2">
  <attribute name="in"/>
  <attribute name="out"/>
  <attribute name="style"/>
  <sequential>
    <java classname="net.sf.saxon.Transform" fork="yes" failonerror="true">
      <classpath refid="saxon.path"/>
      <arg line="-l"/>
    </java>
  </sequential>
</macrodef>
```
Then I am able to set any set of desired arguments as part of the definition of a new task. The example3 target illustrates how this is invoked.

```
<saxon2 in="data/test.xml" style="test.xslt" out="result/output.xml"/>
```

Many developers (including myself) used such techniques to manage multiple pass transformations, but with Saxon being a robust and stable v2.0 XSLT processor it is now possible to achieve multistage pass within XSLT.

A good rule of thumb to follow is to keep things simple and avoid building complicated XSLT processing frameworks with Ant. Trust me, many of us have done this in the past and created things that were pretty ugly … there are now emerging technologies (such as XProc - http://www.xproc.org) which are much more appropriate.

**Documenting XSLT**

*Recipe:* "Generate documentation for your XSLT using XSLTDoc"

There are other things that Ant can do with XSLT apart from managing XSLT processing such as helping you document your XSLT. A framework called XSLTDoc can be integrated with Ant to automatically generate documentation from you XSLT.

`http://www.pnp-software.com/XSLTDoc/`

XSLTDoc works by applying a stylesheet to a configuration file which contains a list of the xslt that you want documented. Here is the example configuration I use, contained in the xsl/config.xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<XSLTDocConfig>
  <Title>The title used on the main page</Title>
  <Introduction>
    This section is copied to the main documentation page. It can include any HTML tags.
  </Introduction>
  <TargetDirectory path="../result"/>
</XSLTDocConfig>
```
In addition, you may add xd:* namespaced elements inside of your xslt documents to add specific
documentation. The xsl/example.xslt file contains how these elements can be used in an example
transformation.

<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xd="http://www.pnp-software.com/XSLTdoc" version="2.0">
  <xd:doc type="stylesheet">
    Example documentation
    <xd:author>jfuller</xd:author>
    <xd:copyright>2010, Webcomposite s.r.o.</xd:copyright>
  </xd:doc>
  <xsl:template match="/">
    test
  </xsl:template>
</xsl:stylesheet>

The xd:doc, xd:author, and xd:copyright elements are used by XSLTDoc to enhance the generated
documentation. Run the example1 Target which will generate a nice set of HTML documentation in
the result directory.

<target name="example1">
  <copy todir="result">
    <fileset dir="css">
      <filename name="*.css" />
    </fileset>
  </copy>
  <java fork="true" jar="lib/saxon8.jar">
    <arg value="xsl/config.xml" />
    <arg value="xsl/xsltdoc.xsl" />
  </java>
</target>
I have used the java task to directly call the embedded SAXON library that comes with XSLTDoc.

There are some drawbacks with XSLTDoc by the apparent need to place your projects assets (xslt and configuration) directly into the XSLTDoc source tree to generate the documentation.

*Tip- There are other frameworks for documenting XSLT out there that can be integrated with Ant, for example there exists a documentation function in commandline form that ships with Syncrosoft’s oXygenXML http://www.oxygenxml.com/*.

## Validating XML

Most programmers know that for XML to be ‘real XML’ it needs to be well formed e.g. syntactically correct markup. All this means is that an XML document must follow the rules of the XML 1.0 specification (balanced tags, quoted attributes, XML Namespaces, etc).

On the other hand, I find programmers less familiar with the concept of a valid XML document and the underlying technologies that enable schema validation.

At its simplest, validation is a process that checks a well formed XML document that it conforms to some desired structure, containing the correct elements and attributes, as well as defining the format of values. The set of rules defining validation is known as a schema.

The simplest approach to validation of XML (and HTML for that matter) is to use a DTD. A Document type definition (DTD) is a set(s) of ELEMENT, ENTITY, ATTLIST, and NOTATION declarations providing the definition of a class of valid documents. A DTD historically is used to constrain markup such as HTML or SGML (which is the grandfather of both HTML and XML).

A DTD can also be used to constrain XML markup, either being embedded directly in the XML document or referenced as an external file.

With the rise of XML, there has been several efforts of defining grammar based approaches to validation, some of which are listed below;

- **XML Schema**: W3C behemoth of a schema technology, has great vendor tool support and is touted as being a complete if not overly complex solution. The latest XML Schema v1.1 specification goes some way to addressing some of the perceived complexities, but it is unfortunately not widely deployed
• **RELAXNG**: Lightweight approach to XML schema validation which has both an XML and text based compact syntax.

• **XSLT / Schematron**: Using XPATH and XSLT one can create a surprisingly concise validation process

• **Programmatic**: There is nothing stopping someone using a SAX filter to check for the existence of elements or attributes with a certain value. Programming such one-off validation processes can sometimes be the right approach, as well as fast and easy to maintain.

W3C Schema is good in situations where there is deep tool support and complicated validation requirements

RelaxNG is the lightweight ‘darling’ of Schema technologies and is good for general validation but lack of integrated tool support makes things difficult. Schematron (and related examplotron) is useful in situations where you need to check co-constraints e.g. elements and attribute values are dependent upon each other

For the rest of this section, our discussions on XML validation I dwell on the usage of DTD, Relax NG or W3C XML Schema with Ant.

None of the above schema technologies are a perfect fit for validating and constraining all XML documents.

**Playing with AntStructure**

*Recipe:* ”Generate a DTD for Ant markup”

I know that Ant’s build files are XML, so what is stopping us from creating a DTD with which to validate with?

Nothing, other then the drudgery of manually producing a DTD and keeping it up to date with the current Ant installation. If one uses 3rd Party Tasks this becomes more difficult. This dilemma is partially solved by using Ant’s own built in DTD generator `<antstructure/>` Task.

```xml
<antstructure output="result/test.dtd"/>
```

Running the example1 Target will generate a test.dtd file into the result directory. Nevertheless this is DTD is also incomplete as the aforementioned 3rd party tasks will not be defined, but its a better starting point then nothing, for validating build scripts.

*Note: There is referenced in the Ant’s manual, a technique for generating a complete dtd definition at http://www.sdv.fr/pages/casa/html/ant-dtd.en.html though untested by the author.*
Ant’s native validation Tasks

Here are two recipes involving Ant’s native validation Tasks, the first being `<xmlvalidate/>`.

XmlValidate

Recipe: “Validate using xmlvalidate Task”

Ant includes support for validating XML documents through use of the `<xmlvalidate/>` Task, which validates using Document Type Definitions (DTD). The latest version of Ant enhances this Task to also apply XML Schema processing via the use of Xerces’s XML Schema implementation.

The following basic incarnation of `<xmlvalidate/>` Task determines whether the valid-xml-example.xml is well-formed;

```
<xmlvalidate failonerror="no" lenient="yes" warn="yes">
  <fileset dir="data" includes="*.xml"/>
</xmlvalidate>
```

The lenient attribute, highlighted in bold above, controls whether or not XML parsing checks for well formedness. You may also specify which SAX parser `<xmlvalidate/>` uses for validation by defining a classname attribute with a value of the SAX entry class. Note that choosing other then a SAX2 parser will automatically disable validating for well formedness.

Whilst checking if an XML document is well formed is useful, our goal is to determine if an XML document is schema valid. Validity is defined in the sense that an XML document’s structure and contents obeys constraints as defined in the Schema or DTD.

For example consider the following XML document which represents address data in markup.

```
<?xml version="1.0"?>
<address>
  <first>Gabriela</first>
  <last>Kratinova</last>
  <phone>555 555 555</phone>
</address>
```

The XML is well formed and this is can be confirmed by running the validate-wellformed target in the example script.

```
ant validate-wellformed
```
Buildfile: build.xml
validate-wellformed:
[xmlvalidate] 2 file(s) have been successfully validated.

Both XML documents, valid.xml and example.xml, contained in the xml directory were reported as well formed. Our next question would be is the XML document valid? This question can only be asked and answered once I define a schema or DTD for the XML document.

A schema creates rules for testing validity; for example, I could say that the above document would be invalid if the <phone/> element contained values other then numeric characters or if the <first/> and <last/> elements contained only numeric characters.

Now I demonstrate DTD to validate XML. The <xmlvalidate/> Task checks a set of XML documents against a DTD through the definition of a nested <dtd/> element, as shown in the following example.

```xml
<xmlvalidate warn="false" lenient="no">
  <fileset dir="data" includes="valid.xml"/>
  <dtd publicId="-//" location="etc/example.dtd"/>
</xmlvalidate>
```

If the XML document being checked also has a DTD declared in the body of the XML document or itself refers to an external DTD, then this will get processed as well.

Running the validate-dtd target will check the xml/valid.xml document against the example.dtd DTD.

```
ant validate-dtd
Buildfile: build.xml
validate-dtd:
[xmlvalidate] 1 file(s) have been successfully validated.
```

If you don’t believe Ant’s console listing take a peek inside of the example.dtd itself to see if the XML document conforms. You will need to know how DTD’s are defined, of which there exists many good tutorials on the Internet.

To validate XML documents against W3C XML Schema's I inherit functionality from the included Xerces XML parser.

```xml
<target name="validate-xmlschema">
  <xmlvalidate warn="false" lenient="no">
    <fileset dir="xml" includes=" address-valid-xmlschema-sample.xml"/>
  </xmlvalidate>
```
I have added a few <attribute/> elements which is necessary for instructing the parser Ant is using to turn on XML Schema processing. This is similar to our technique of enabling XInclude processing, shown in earlier section of the chapter. The XML document I are checking now refers to a DTD and an XML Schema;

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE date SYSTEM "address.dtd">
<address
   xmlns="http://www.apress.com"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.apress.com ../address-xmlschema.xsd">
   <first>Gabriela</first>
   <last>Kratinova</last>
   <phone>5555555555</phone>
</address>
```

**SchemaValidate**

**Recipe:** "Validate using schemavalidate Task"

With Ant v1.7 there now exists the <schemavalidate/> Task, which I think is more usable then <xmlvalidate/> for validating XML documents.

The nested <schema/> element, in the example below, informs the validating parser of what schema to use for what XML Namespace.

```xml
<schemavalidate>
   <schema namespace="http://www.example.com" file="data/example.xsd"/>
   <fileset dir="data" includes="good-example.xml"/>
</schemavalidate>
```

Example1 Target will pass validation on data/good-example.xml file.

Associating a schema or DTD to an XML Namespace is an easy and logical approach.
This Ant Task can be considered an extension of `<xmlvalidate/>`, supporting similar nested elements (as shown in `<xmlvalidate/>` example) like catalog, xmlcatalog and `<attribute/>` for setting parser attributes.

Instead of defining attributes, as I did with previously with the `<xmlvalidate/>` Task:

```
<attribute name="http://xml.org/sax/features/validation" value="true"/>
<attribute name="http://apache.org/xml/features/validation/schema" value="true"/>
```

the `<schemavalidate/>` Task comes ‘preset’ for specifically doing W3C XML Schema processing (listed below).

- XML parser is namespace aware
- Validation is turned on by default.
- Schema validation is turned on by default.

When I used `<xmlvalidate/>` for XML Schema processing, there was no easy method of specifying which XML schema to use other then embedding the reference to a schema directly within an XML document ala xsi:schemaLocation (which is just a hint to the processor).

The `<schemavalidate/>` example addresses this shortcoming, establishing how multiple `<schema/>` elements can designate which schema gets applied to a specific ‘namespaced’ element.

```
<schemavalidate file="example.xml">
    <schema namespace="http://www.example.com" file="example.xsd"/>
    <schema namespace="http://www.other-example.com" file="example1.xsd"/>
    <schema namespace="http://www.example2.com" file="example2.xsd"/>
    <schema namespace="http://www.ruminate.co.uk/test" file="ruminate.xsd"/>
    <fileset dir="xml" includes="*.xml"/>
</schemavalidate>
```

I find this a flexible way to maintain schema processing, as integrated into either my software’s build process or as part of a workflow when generating XML.

The example2 Target will run against an xml file which is invalid on purpose resulting in the build to fail.

```
Buildfile: /4_validate_xml/3_schemavalidate/build.xml
example2:
```
A note on alternate validation techniques

I often find that certain validating processors provide less information than others when it comes to certain classes of problems. That’s why I keep on hand a few other command-line processors to help out in complicated scenarios.

**Recipe:** "get additional error information on XML errors using xmllint"

The popular LibXML and LibXSLT processor for Linux comes with a useful commandline utility called xmllint. The following snippet shows how to use it.

```xml
<exec executable="xmllint">
  <arg value="somefiletovalidate.xml"/>
</exec>
```

I already used xmllint previously in this chapter, so you may already have it installed. Many linux based machines may already have xmllint installed and its easy enough to get from http://www.xmlsoft.org/.

**Recipe:** "get additional error information on XML errors using rxp"

rxp is another validator of worth, a bit older than most and very fast. As with xmllint its usage is via a straightforward `<exec/>` task.

```xml
<exec executable="rxp">
  <arg value="somefiletovalidate.xml"/>
</exec>
```

The RXP validator can be found at http://www.cogsci.ed.ac.uk/~richard/rxp.html

**Resolving DTD with <xmlcatalog/>**

**Recipe:** "Speed up resolving of DTD by using XML catalogs"
If you are finding validation processing slow, it might be due to the parser resolving references to remote schema definitions. This is usually addressed by defining an XML Catalog.

An XML Catalog allows an XML Parser, XSLT processor, or other process working with XML to use a local representation of a DTD or entity definition instead of resolving some remote network (e.g. Internet) resource.

Ant provides an interface to using XML Catalog’s through the definition of an `<xmlcatalog/>` resource containing `<dtd/>` elements.

```xml
<xmlcatalog id="common">
  <dtd
    publicId="-//Webcomposite s.r.o.//DTD Address 1.0//EN"
    location="example.dtd"/>
  <dtd
    publicId="-//OASIS//DTD DocBook XML V4.1.2//EN"
    location="docbook/docbookx.dtd"/>
</xmlcatalog>
```

The `<xmlcatalog/>` resource will perform Entity and URI resolution on an XML document when the document is parsed by some XML Ant Task. The resource itself is normally nested in the XML Ant Task (that supports `<xmlcatalog/>`).

When using either `<xmlvalidate/>` or `<schemavalidate/>` the above XML Catalog can then be referenced by nesting a `<xmlcatalog refid="common"/>` element, also making sure that the correct DTD exists at the location specified.

When an Ant Task references an `<xmlcatalog/>` then the following logic on the location attribute is applied in stepwise fashion, in the determination of where to retrieve the resource representation;

1. filesystem lookup: convert location to a file url
2. classpath lookup: will search classpath, including jar files for the referred DTD
3. JAXP resolver lookup: will use xml commons library for resolving resource
4. url lookup: finally if none of the above were successful then will convert the location attribute to a url

If you have set things up properly with `<xmlcatalog/>` then Ant should use the filesystem version of a resource first making things much quicker.
Tip: For more information, go to OASIS Open Catalog standard at http://oasis-open.org/committees/entity/spec-2001-08-06.html

Using `<attribute/>` to speed Schema Validation

Recipe: "Speed up Xerces Schema validation through grammar caching"

Akin to our use of XML Catalogs, schema validation speeds up when specifying `<schema/>` elements that reference local resources (when using `<schemavalidate/>`). You might investigate further on ways of relating schemas and resolving their locations by going to http://apache.org/xml/Properties/schema/external-schemaLocation

A more impressive optimisation technique is to activate a parser's ability to cache schema grammar. Xerces has just such a feature and is switched on by embedding an `<attribute/>` in either of the schema validation Tasks I showed earlier.

```xml
<attribute
name="org.apache.xerces.xni.parser.XMLParserConfiguration"
value="org.apache.xerces.parsers.XMLGrammarCachingConfiguration"/>
```

This approach is parser specific (above only works with Xerces) so you will need to check with your specific XML parser if it has any grammar caching capabilities.

Schematron Ant Task

Recipe: "validate using Schematron"

Schematron is another popular schema technologies, details can be found here.

http://www.schematron.com/

There exists a 3rd party schematron Ant task, which will invoke schematron processor against your xml. The example1 Target does this as follows.

```xml
<schematron schema="sample.sch" failonerror="false">
  <fileset dir="data" includes="*.xml"/>
</schematron>
```

Just a matter of supplying the schema with the schema attribute and using `<fileset/>` to select xml documents to validate. Running the example1 Target will output this from the console.

Buildfile: /4_validate_xml/schematron/build.xml
example1:
[schematron] Compiler warnings:
   [schematron] 3 file(s) have been successfully validated.
   BUILD SUCCESSFUL
   Total time: 1 second

A successful build will mean that all xml successfully validated, the task also generates a report in result.xml.

Tip: There are a few 3rd party tasks for validation in existence, one of which (untested by the Author) can be found here http://iso-relax.sourceforge.net/JARV/antTask.html

Multiple and Parallel Schema Validation

Recipe: ”Use Jing and Namespace Routing Language to orchestrate parallel and multiple schema validation”

Validation will save time for you and others during the development and maintenance of software systems that use XML. With the use of schema technologies, errors are caught earlier and data conforms to a prescribed format and content.

Since there does not exist any single schema technology that everyone likes or uses, it is becoming increasingly common for developers to have to manage validation across several technologies.

The ultimate schema processing technique would allow a developer to use all major schema technologies consistently from a single interface. I will show how to do this by using the Jing processor and Namespace Routing Language (NRL). This approach elegantly manages the validation of XML documents with multiple schemas across a range of schema technologies.

The following two schema processor permits you to validate against most of the major schema languages e.g. RelaxNG, XML Schema, DTD, and even lesser known technologies like the popular Schematron (http://www.schematron.com/).

Sun Multi-Schema XML Validator: MSV is Sun’s java tool which validates an XML document against many XML schemata. MSV currently supports RelaxNG, XML Schema, DTD’s and a various other schema technologies. If your application uses JAXP for all its XML document processing
you can ‘swap’ in MSV as a JAXP proxy (JAXP Masquerading). This means that all XML documents that go through JAXP will also be validated. MSV detects the schema language of the current XML document, irregardless of its file extension, with the exception of DTD which need to have a .dtd extension.

**Jing**: Created by the author of the RelaxNG specification, Jing provides support for both RelaxNG and XML Schema. XML Schema support is achieved through integration with Xerces. Jing comes with a rather limited Ant Task, as well as an implementation of the Namespace Routing Language (NRL) which I use later on in this section.

James Clark’s Jing processor has the features I need for creating a unified approach to schema processing. The source code for this chapter download includes the jing processor but here are the steps for getting and running it for yourself.

I will now show you how to use Jing and Apache Ant.

**Manually Getting and Running Jing**

For your convenience I have included a copy of Jing in the download for this chapter, but you may also want to download and get running manually.

To do this, first download the Jing processor from James Clark’s website

http://www.thaiopensource.com

and expand the compressed file. Place the /bin directory onto your Java’s classpath or copy these files to Ant’s own /lib directory. Jing can always be run from the command-line using the java –jar option:

```
java -jar path-to-dist/jing.jar address-xml.rng xml/address-valid-sample.xml
```

The /bin directory contains some third-party jar files, which are used for XML parsing and for validating with schema languages other than RELAX NG:

- **saxon.jar**: Michael Kay’s SAXON XSLT Processor from http://www.saxonica.com. xerces-simpl.jar and xml-apis.jar: Both of these come from come from the Xerces-J 2.4.0 distribution.

- **isorelax.jar**: Provides integration to schema validators that employ the JARV interface.

To test if your Jing is installed properly please amend the build script, for this section, `${jar.jing}` Property to contain your Jing jar location.

**Jing example**
The example1 Target uses the Jing Ant Task, validating address-valid-sample.xml against a RELAX NG schema. I first need to define the jing Task using `<taskdef/>

```xml
<Taskdef name="jing" classname="com.thaiopensource.relaxng.util.JingTask">
    <classpath>
        <pathelement path="${jar.jing}"/>
    </classpath>
</Taskdef>
```

The example1 Target shows how `<jing/>` accepts an RNG file and selects files using `<fileset/>` element.

```xml
<target name="example1" description="validate relaxng">
    <jing rngfile="schema/address-xml.rng">
        <fileset dir="data">
            <include name="address-valid-sample.xml"/>
        </fileset>
    </jing>
</target>
```

Executing this Target will prompt Jing to validate using address-xml.rng RelaxNG schema on XML document contained in the /data directory. If the build is a success then you know that Jing is working properly.

If I were to use Relax NG's compact syntax, the compactsyntax attribute must be included and given a value of 'yes'. Inspect and run the example2 Target to see an illustration of this.

**Jing Commandline**

*Recipe: "Use Jing to perform all major schema validation”*

OK, Jing can perform RelaxNG validation but what about being able to handle other schema processors? Jing supports the following validation technologies:

- RELAX NG 1.0 Specification,
- RELAX NG Compact Syntax, and
- parts of RELAX NG DTD Compatibility, specifically checking of ID/IDREF/IDREFS.

Jing has experimental support for the following schema languages
• W3C XML Schema (based on Xerces2-J);
• Schematron 1.5 (in a future version of Jing, the support for Schematron will be updated to support the ISO-standardized version of Schematron);
• Namespace Routing Language (in a future version of Jing, the support for NRL will be updated to support NVDL, which is an ISO-standardized schema language based on NRL).

Support of W3C XML Schema validation is done merely by the fact that Jing is inheriting the Xerces XML Parser which itself performs XML Schema validation.

One of the limitations of the <jing/> Ant Task is that it doesn’t provide any way of specifying XML Schema validation, which I can with the command line tool. So I need to forgo using the Ant task and wrap up the command-line Jing invocation using a <macrodef/>.

```xml
<macrodef name="jing2">
  <attribute name="schema"/>
  <attribute name="xml"/>
  <sequential>
    <java jar="${jar.jing}"
       fork="true"
       failonerror="true">
      <arg value="@{schema}"/>
      <arg value="@{xml}"/>
    </java>
  </sequential>
</macrodef>
```

The macrodef invokes the Jing jar as if I were using it from the command-line using <java/> Ant Task. The <java/> Task is supplied with a value for the location of the schema and xml documents.

Target example3 will use the newly defined <jing2/> Task. As with Sun’s Multiple Schema Validator, Jing automatically detects which kind of schema is being used through inspection of a schema document’s namespace URI.

```xml
<target name="example3" description="validate with macrodef">
  <jing2 schema="schema/address-xml.rng"
         xml="data/address-valid-sample.xml"/>
  <jing2 schema="schema/address-xmlschema.xsd"
         xml="data/address-valid-xmlschema-sample.xml"/>
</target>
```
Running the target gets across that I can handle RelaxNG and XML Schema using a single approach. In fact, Jing lets us validate in a few other schema languages, such as DTD or the previously mentioned Schematron.

Jing has a few cryptic command-line switches which you can amend your own <jing2/> Macroddef to use.

- **-c**: Jing will expect schema to be RELAX NG Compact Syntax.

- **-e enc**: Uses the encoding enc to read the schema.

- **-f**: Checks that the document is feasibly valid. A document is feasibly valid if it could be transformed into a valid document by inserting any number of attributes and child elements anywhere in the tree. This is equivalent to transforming the schema by wrapping every data, list, element and attribute element in an optional element and then validating against the transformed schema. This option may be useful while a document is still under construction. This option also disables checking that for every IDREF there is a corresponding ID.

- **-i**: Disables checking of ID/IDREF/IDREFS. By default, Jing enforces the constraints imposed by RELAX NG DTD Compatibility with respect to ID/IDREF/IDREFS.

- **-t**: Prints the time used by Jing for loading the schema and for validation.

**Working with Namespace Routing Language (NRL)**

*Recipe: "Use Namespace Routing Language to control binding of schema validation to XML Namespace”*

Jing contains an implementation of the Namespace Routing Language, this is a meta markup language that Jing will use to look-up what schema technology (and schema) to use on what XML Namespace.

NRL is the result of the efforts of a few individuals, two of whom I will mention;

- **James Clark** synthesised earlier efforts into the Namespace Routing Language, especially the early Modular Namespaces specification.

- **Eric Jellife** (Schematron creator) Namespace Switchboard efforts http://www.topologi.com/resources/NamespaceSwitchboard.html

Namespace Routing Language is now adopted by OASIS and is succinctly explained. The expressed goal of NRL was to have an XML vocabulary that would assist in the validation of XML documents using multiple schema technologies. This goal also happens to be the ambition for the next Ant solution as well!
The Namespace Routing Language (NRL) addresses the near future of XML; that of complex XML documents containing multiple namespaces all of which may have multiple schema validation definitions.

The next solution, using NRL can replace our previous efforts of achieving validation using Ant’s native schema related Tasks.

To repeat myself, the Namespace Resource Language is an XML language which defines a set of rules which define how schema processing should occur. The following NRL example associates two schemas with two namespaces, specifically a RelaxNG schema four RelaxNG itself and an RelaxNG schema for XSLT.

```xml
<rules xmlns="http://www.thaiopensource.com/validate/nrl">
  <namespace ns="http://relaxng.org/ns/structure/1.0">
    <validate schema="RELAXNG.RNG"/>
  </namespace>
  <namespace ns="http://www.w3.org/1999/XSL/Transform">
    <validate schema="XSLT.RNG"/>
  </namespace>
</rules>
```

Each NRL document must declare a <rules/> root element with the NRL namespace. The <rules/> element can contain one or more <namespace/> elements. Namespace elements link a specific XML namespace with a concrete schema definition. This is accomplished through the use of a ns attribute, defining a namespace, and an encapsulated <validate/> element’s schema attribute defining the physical schema.

Put simply, whenever our XML documents namespaced elements/attributes matches with any of the NRL namespace definitions, the defined schema is used to validate that element or attribute. Following the rules of XML namespaces this would validate encapsulated elements or attributes contained in the XML namespace. The first <namespace/> element would be used in validating RELAXNG schemas themselves, the second <namespace/> element can be used to check the validity of XSLT documents, which of course are XML documents as well.

For the purposes of using, the NRL document would take the place of a schema in the <jing2/> Task or command-line version.

Tip- NRL represents a somewhat older approach and I would recommend investigating NVDL for something more modern (http://en.wikipedia.org/wiki/Namespace-based_Validation_Dispachting_Language).

Concurrent Schema Validation
Recipe: “Use Namespace Routing Language to achieve concurrent schema validation”

NRL has one more trick up its sleeve, the ability to specify multiple schemas for a single namespace, enabling concurrent schema validation.

I will create a definition that relates to our earlier validation efforts with address XML documents. Now I have to define how NRL uses the two schemas, one in RelaxNG and one in XML Schema, which checks XML documents for valid <address/> elements. These two schemas are added via <namespace/> element to the existing NRL document which results in;

```
<rules xmlns="http://www.thaiopensource.com/validate/nrl">
  <namespace ns="http://www.example.org/address">
    <validate schema="address-xml.rng"/>
    <validate schema="address-xmlschema.xsd"/>
  </namespace>
</rules>
```

The reader should notice that all I’ve done is nest another <namespace/> within the <namespace/> element. Now there is a Relax NG schema and W3C XML Schema associated to the namespace http://www.example.org/address. Wherever these namespaces are encountered during Jing processing of XML document(s), both schemas will be used to validate.

Running the example, via the example4 Target, will result in the address element first being checked against a RelaxNG Schema and then checked against a W3C XML Schema.

The example4 Target will fail due to unknown <address/> element.

```
example4:
  [java] /4_validate_xml/5_multiple_schema_validation/data/address-nrl-sample.xml:2:49: error: cvc-elt.1: Cannot find the declaration of element ‘address’.
```

The only ‘gotcha’ when working with NRL is to take care of elements which have no namespaces, the default NRL Jing processing is to throw an error when it encounters an element that does not exist in any namespace.

To effectively ‘deactivate’ this behavior I need to add the following snippet, under the <rules/> element.

```
<anyNamespace>
```
This instruction tells the NRL processor to pass over all elements which have no namespace or a namespace with no corresponding `<namespace/>` instruction in NRL.

**Open Office XML Documents**

Office productivity documents are a vital part of everyday business activities, with today's office documents mainly being created by people for other people to read with information being structured visually e.g. with font styles, tables, paragraphs and bullet lists.

In recent years, I have seen XML's influence changing the ‘game’ in this area enabling new levels of integration previously not possible with proprietary document formats.

By using an XML based document to encapsulate business information, I avoid the creation of proprietary ‘silos’ of information each with their own peculiar processing characteristics.

**Working with OpenDocument**

*Recipe:* "applying an XSLT transformation to an OpenDocument"

The OpenDocument Format (ODF) is an open XML based file format for office productivity applications. The specification was originally created by the OASIS OpenDocument Technical Committee. The OASIS membership has ratified revision 1.1 of the specification and revision 1.0 is also available as an ISO/IEC standard.

- Standardized, free to use
- Easy to access programmatically
- Based on existing standards e.g. zip, xml, html, css, svg, mathml, etc..
- Download the spec at [http://www.oasis-open.org](http://www.oasis-open.org)

OpenDocument Format files are composed of a collection of XML files and other resources stored in a directory structure saved in a zip-file. The individual XML files in the collection are known as streams. An OpenDocument Format file may include any number of streams and resources, as well as a set of standard reserved files:
• **mimetype**, must be stored uncompressed as the first entry in the zip file structure. It contains the mime-type of the document. This allows for convenient file type detection, without actually having to parse the entire document package.

• **META-INF/manifest.xml**, contains information about all the streams and resources that make up the document. Declares type of embedded objects. Information about signatures and encryption.

• **content.xml**, contains the actual document content and automatic styles

• **styles.xml**, style information. Styles are referenced from the content.

• **meta.xml**, metadata about the document, such as title, author and keywords.

The main document types represented by the OpenDocument Format are

• text documents

• spreadsheet documents

• drawing documents

• presentation documents.

There can also be documents which solely represent formula/equations or charts. These documents are typically embedded inside other documents but are themselves standalone.

This recipe will show you how to apply a transformation to these OpenDocuments. I will create a macrodef that unzips the document, then applies XSLT to the content.xml (which contains document data), then repackages into a separate result.

So what kind of processing would you like to do with your document? Perhaps you would like to add information or amend styles, my example will strip out information; that is wherever an email address (@webcomposite.com) or a phone number (denoted with a prefix of ‘Phone’) occurs.

The following XSLT achieves this by matching any text:p elements then examining the text for any occurrences of what I want to remove.

```xml
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:text="urn:oasis:names:tc:opendocument:xmlns:text:1.0"
  xmlns:office="urn:oasis:names:tc:opendocument:xmlns:office:1.0">
```
Whenever a line contains the text '@webcomposite.com' or starts with the text 'Phone' the XSLT transformation outputs a comment, everything else is passed through.

I want to abstract the process into a single task, where I supply the document source directory and its name, along with where I want to place the result and the XSLT I want to apply.

```
<transform-opendoc document.src = "data"
    document.name = "test.odt"
    result.dir = "result"
    xslt ="conversion.xslt"/>
```

Here is the macrodef definition:

```
<macrodef name="transform-opendoc"/>
    <attribute name="document.name"/>
```
The steps this macrodef goes through are

1. **unzip task**: extract opendocument into its document hierarchy in a temporary directory (under the result directory)

2. **xslt task**: apply transformation onto content.xml

3. **copy task**: copy the original unmodified opendocument

4. **delete task**: delete old content.xml

5. **copy task**: copy newcontent.xml to content.xml

6. **delete task**: remove newcontent.xml

7. **touch**: set timestamp on content.xml

8. **zip task**: update original unmodified opendocument (copied across in step 3)
9. **delete task**: remove extracted document hierarchy

So when the example1 target, containing the transform-opendoc macrodef is run, on the following test.odt document contained in the data directory.

```
Here is an example open document
Phone: +420 7282 77 227
email: jim.fuller@webcomposite.com
Here is an example open document
```

It will create a new document in the result directory.

```
Here is an example open document
Here is an example open document
```

As expected, the new document has the lines containing email and phone information stripped.

There does exist a 3rd party task, ODFXSLTRunnerTask, which can do basically the same thing. Information on this task and other ODF related tools can be found at the following address.

```
http://wiki.services.openoffice.org/wiki/Xml/Tools
```

*Note: Microsoft has introduced in its 2007 Office System, Office XML Formats which are similar to Open Document in that its a zip file containing a specific markup vocabulary and directory layout, so the same technique applies.*

**Working with Docbook**

*Recipe: “transforming Docbook to HTML or WordML”*

Docbook is a popular xml format for creating documents. It’s vocabulary is comprised of elements that semantically describe the sections of a paper, book, or document and you then can use the various frameworks in existence to transform into a wide array of styled documents.

The primary framework that most people use for working with docbook and generating their final format, is a set of XSLT stylesheets which can transform docbook into:

- html
- wordml
- xsl:fo / pdf
- epub
• man / help pages

As this is a straight single stage transformation I will show you how to work with the docbook xslt stylesheets. The example1 target, I convert the test.xml docbook contained in the data directory into an html document.

```xml
<target name="docbook2html">
  <java classname="${xslt.processor.class}" fork="yes"
    dir="${in.dir}" failonerror="true">
    <classpath refid="xslt.processor.classpath"/>
    <arg line="-o ../${html.outfile}"/>
    <arg line="-l"/>
    <arg line="${main.infile} ../${html.stylesheet}"/>
  </java>
</target>
```

In this target you see I have used the java task to invoke the processor in the xslt.processor.classpath, which is a version of SAXON that ships with docbook stylesheets. The other thing to note is the need to select the correct stylesheet.

```xml
<property name="html.stylesheet" value="${docbook-xsl.dir}/html/docbook.xsl"/>
```

Which is set with the html.stylesheet property. This is a single pass transformation and really no need to use a macrodef, though feel free too if you plan to do more with your docbook.

In the second example target, I will generate a Microsoft WordML document.

```xml
<target name="docbook2wordml">
  <java classname="${xslt.processor.class}"
    fork="yes" dir="${in.dir}" failonerror="true">
    <classpath refid="xslt.processor.classpath"/>
    <arg line="-o ../${wordml.outfile}"/>
    <arg line="-l"/>
    <arg line="${main.infile} ../${wordml.stylesheet}"/>
    <arg line="wordml.template=template.xml"/>
  </java>
</target>
```

Supplying the correct stylesheet will convert docbook to wordml.
Recipe: "create sophisticated documentation using Docbook XML"

If you need to do more advanced docbook processing you could consider using the full featured Docbook framework, which is a part of the Apache Velocity project.

http://velocity.apache.org/docbook/

The example1 target included actually runs the Ant build that comes with the Docbook Framework, contained in the lib directory for this chapter.

```xml
<target name="example1">
  <ant antfile="${dbf.basedir}/build-docbook.xml" target="all">
    <property name="docbook.dir" value="dbf"/>
    <property name="docbook.file" value="example"/>
  </ant>
</target>
```

Running this target outputs a set of files under the target/dbf directory, generating the following variants:

- html as a single page
- htm as multiple pages
- zipped up html
- pdf

Docbook itself can be complicated enough, adding the Docbook framework adds even more complexity, I advise careful consideration before using.

Tip: David Weiss has also developed a docbook style Ant Task, which is untested by the author. More information can be found here http://www.cs.put.poznan.pl/dweiss/xml/projects/ant-docbook-styler/index.xml?lang=en
Source Code Bonus: Advanced XML Databases

As shown at the end of the Database chapter, the popular eXist XML Database comes with a set of useful Ant tasks to load and get XML data from the database.

A while back, I had an interesting requirement where I needed to mirror data from one instance of XML Database to another, so I reached for Apache Ant to try and solve the problem. The resultant code is included now in eXist project (where I am a source code contributor) and I have included it as a source code bonus.

You will need to edit the properties to reflect the location and credentials of the two XML Database instances involved.

Conclusion

This chapter introduced how to work with XML with Ant, going through the basic built in Tasks which can be used for all sorts of processing.

I then showed how to use Ant to control XSLT, as well as help develop XSLT with techniques such as generating documentation.

Another interesting area is the emerging XML formats which are starting to take over where proprietary formats used to hold significant sway. Open Source XML document formats, such as OpenDocument and Docbook, allow for a new level of integration and management and I showed how Ant can help automate working with these document packages.

I then showed a series of methods for validating xml; from using Ants built in xmlvalidate and schemavalidate Tasks to more advanced approaches with multi-validation and schematron.

The next chapter is all about using Apache Ant within the world of web development.
Web Recipes

Apache Ant has an assortment of methods for working with common web technologies which this chapter will go through.

- **Working with HTTP**: Learn how to craft HTTP GET/POST/PUT requests using Ant
- **Consistent and Clean HTML**: Let Ant help you make more compliant code for your websites
- **Optimising the Web**: Optimizations that can be applied to your HTML, CSS, and JS files
- **Other Web Techniques**: Learn how to create uptime dashboards, start and stop Apache web server and how to create Firefox XUL extensions.

**Working with HTTP**

*Recipe: “check that a website is live and accessible”*

Ant provides a few Conditions for testing TCP/IP connectivity to a specific host.

- `<http/>`: set true when URL is available
- `<socket/>`: set Property to true when TCP/IP listener responds to a specific port
- `<isreachable/>`: uses Java own network functionality to determine if a host is accessible. This can use a number of protocols e.g. ping, etc ...

In the example for this section, I have used the `<http/>` condition to test for a response from a web-server (e.g. www.google.com) and to also test for the existence of a specific page on the web-server.

```xml
<condition Property="google.exists1">
  <http url="http://www.google.com"/>
</condition>

<condition Property="google.exists2">
  <http url="http://www.google.com/somenonexistentpage"/>
</condition>
<echo message="Google home page running: ${google.exists1}"/>
<echo message="Google non-existent page running: ${google.exists2}"/>
```
To run the example1 Target you will need connectivity to the internet and here is the result that should output to the console.

```
example1:
  Google home page running: true
  Google non-existent page running: ${google.exists2}
  BUILD SUCCESSFUL (total time: 1 second)
```

An Ant Property is only set if a Condition is satisfied, so the first condition evaluates to true and the second condition will never evaluate, as this page does not exist which is why the use of the Property reference e.g. ${google.exists2}.

**<get/> Task**

**Recipe:**“perform a HTTP Get with Ant”

An HTTP GET operation is performed through the use of the Ant Core <get/> Task. There is not much to say about this Task other then it will resolve a URL and store the results in a local file. The example3 Target download a copy of Google home page to a local file named google-home-page.html.

```
<get src="http://www.google.com" dest="google-home-page.html"/>
```

Running example3 will output the following

```
Buildfile: build.xml
example1:
  [get] To: result/get.html
  [get] ....
  BUILD SUCCESSFUL
  Total time: 2 seconds
```

**<post/> Ant-Contrib Task**

**Recipe:**“perform a HTTP Post”

With HTTP Get I can perform a range of tasks, but the preponderance of interesting web services and form methods use HTTP Post to get ‘work’ done.

Ant has no native Tasks to manage doing an HTTP Post; there are some vestigial Tasks in Ant development ‘sandbox’ which in my opinion are abandoned and unsupported (if anyone finds this not to be the case, please inform me).
To make Ant do an HTTP POST operation I need to resort to using the Ant-Contrib \texttt{<POST/>} Task. The example will POST data to the W3C validator service (http://validator.w3.org/) which validates a web page against html/xhtml DTD.

\begin{verbatim}
<post to="http://validator.w3.org/check"
    verbose="true"
    logfile="result/example4a-result.html">
    <prop name="uri" value="http://www.google.com"/>
</post>
\end{verbatim}

When run, this Task will place results into a logfile (result/example4a-result.html), which is just a web page containing all the validity errors found on the Google home page.

Ant-contrib furnishes a few more HTTP related tasks, but their lack of documentation makes it difficult to recommend their usage, but I list them below so you are aware that there maybe other options out there.

- **HttpClient**: creates a reusable HTTP client
- **PostMethod**: performs a HTTP POST operation
- **GetMethod**: performs a HTTP GET operation
- **HeadMethod**: performs a HTTP HEAD operation

\textit{Tip: Testing reveals that there is no support for HTTPS protocol in the current Ant-contrib HTTP related Tasks. You could use some alternative like curl or Perl to achieve this, here is one example:}


\texttt{<put/>} Macrodef

\textit{Recipe: "perform a HTTP Put"}

Our toolbox is not complete without an implementation of HTTP PUT, which allows one to upload/store a resource onto a web server.

With no ready made options in existence for performing HTTP PUT from within Ant, I will show you a way to manufacture your own using javascript in a \texttt{<script/>} Task.

I need to ensure your Ant installation is set up for scripting you need to install the following into your Ant lib directory (and make available on the classpath)
Getting this to work may seem to the reader as a lesson in Ant dependency management. The following is the list of Jars needed by Ant, so that `<script/>` works as well as being able to access HTTP related functionality.

- bsf.jar
- rhino jar (js.jar)
- commons-httpclient.jar
- common-codec.jar
- common-logging.jar

I have created a reusable `<macrodef/> Task.`

```xml
<macrodef name="http-put">
    <attribute name="dir"/>
    <attribute name="url"/>
    <attribute name="user"/>
    <attribute name="pass"/>
    <sequential>
        <Property name="data.dir" location="@{dir}"/>
        <Property name="url" value="@{url}"/>
        <Property name="user" value="@{user}"/>
        <Property name="pass" value="@{pass}"/>
        <script language="javascript">
            <![CDATA[
                importClass(java.io.File);
                importClass(java.io.FileInputStream);
                importClass(Packages.org.apache.commons.httpclient.HttpClient);
                importClass(Packages.org.apache.commons.httpclient.UsernamePasswordCredentials);
                importClass(Packages.org.apache.commons.httpclient.methods.PutMethod);
                var srcDir = new File(project.getProperty("data.dir"));
                var files = srcDir.listFiles();
                var user = project.getProperty("user");
                var password = project.getProperty("pass");
                var url = project.getProperty("url");
                var client = new HttpClient();
                client.getState().setCredentials(null, null,
                    new UsernamePasswordCredentials(user, password));
                for (var i = 0; i < files.length; i++) {
                    if (files[i].isFile()) {
```
var put = new PutMethod(url + files[i].getName());
put.setRequestBody( new FileInputStream(files[i]) );
client.executeMethod(put);
    put = new PutMethod(url + files[i].getName());
put.setRequestBody( new FileInputStream(files[i]) );
client.executeMethod(put);
}
]
]]>
</script>
</sequential>
</macrodef>

This Task performs an HTTP PUT operations using javascript. The example5 Target uses a test webdav server (http://test.webdav.org) but you may have to supply your own webdav server to properly test.

<http-put dir="data"
    url="http://test.webdav.org/dav/"
    user="user1" pass="user1"/>

This example will attempt to upload a document to an eXist XML database which supports the HTTP Put command for uploading documents. Having an HTTP Put Task enables us to upload files to servers (if a server is configured to do so).

To create the <http-put/> Task I could have opted to use a scripting language and the <scriptdev/> Task to create a Macrodef, though since I mixed both Ant script with javascript it was more appropriate to use <script/> within a <macrodef/>.

Ant has native tasks for managing the other common net protocols, such as FTP, Telnet, and SMTP (Email).

As each of these Tasks may require you to install additional dependencies, its best for you to have Ant’s own documentation at hand (esp. for ‘cut n paste’), for the identical reason I have chosen not to provide any code examples.

Tip: Don’t forget that webdav uses HTTP Put and you can use mod_dav with Apache webserver to setup your own testing server for using the <http-put/> Task. More Internet Protocols. Alternatively
you could opt to use some other server like the eXist XML Database (http://exist.sourceforge.net) which also supports webdav protocol.

<ftp> Task
Recipe: “FTP with Ant”

This following Ant script shows how to upload all .gif images from local resource/images directory to a remote server (ftp.example.org) htdocs directory, using the FTP protocol.

```
<ftp server="ftp.example.org"
     remotedir="c:\htdocs"
     userid="user"
     password="pass"
     separator="">
 <fileset dir="web">
   <include name="resource/images/*.gif"/>
 </fileset>
</ftp>
```

</ftp>

<telnet> Task
Recipe: “Telnet with Ant”

This snippet logs onto a remote server, mine.example.org, via telnet and executes the ls command from the /home/mine directory.

```
<telnet userid="user"
        password="pass"
        server="mine.example.org"> 
 <read>/home/mine</read>
 <write>ls</write>
 <read string="/home/mine"/>
</telnet>
```

</telnet>

<email> Task
Recipe: “Email with Ant”

```
```

</email>
In the software build context, it is common to use the <email/> Task to send out build status and reporting information.

```xml
<mail mailhost="smtp.example.org"
       mailport="1025"
       subject="Functional Test Reports">
    <from address="build@example.org"/>
    <replyto address="root@example.org"/>
    <to address="build@example.org"/>
    <message>The ${buildname} functional tests have completed</message>
    <attachments>
        <fileset dir="reports">
            <include name="**/*.txt"/>
        </fileset>
    </attachments>
</mail>
```

**<scp/> Task**

*Recipe:* "Securely copy files between remote servers"

When you need to securely copy files from one machine to another, I recommend using the <scp/> Task. It copies file(s) to or from a (remote) machine running an SSH daemon.

```xml
<scp todir="someuser:somepassword@test.example.org:/home/jim/">
    <fileset dir="/usr/local/var/mysoftware"/>
</scp>
```

The <fileset/> element only works when you are copying files from the local machine to a remote machine.

**<ssh/> Task**

*Recipe:* "Securely log onto a remote machine and execute a command"

Ant’s optional <sshexec/> Task will log onto a remote machine via SSH and run a command, as if you typed it in a terminal yourself.
The following snippet shows how `<sshexec/>` could be used to restart an Apache webserver. It should go without saying, that you may have to provide the exact location of the binary and ensure that the user has the appropriate rights to run the application.

```xml
<sshexec host="test.example.org"
   username="user"
   password="pass"
   command="apachectl restart"/>
```

Tip: Some people find the 3rd party Maverick Ant Task does a better job with SSH related tasks, there is a free non-commercial version of this at 3SP.com website, here http://3sp.com/en/maverick-ant/

**A note about Proxies and Ant**

*Recipe: "Making Ant work with proxies "*

If your network environment is using a proxy you may need to specially configure Ant for it to work properly with any of the above protocols.

Ant tends to inherit the underlying JVM (Java Virtual Machine) proxy settings with the result usually being correct operation and ‘things just working’ with no intervention required on the part of the developer.

If you are having problems with using Ant with a network proxy, then I would advise that you set Ant’s own proxy properties. This is typically done by setting the environment `ANT_OPTS` environment variable, as shown below;

```bash
export ANT_OPTS="-Dhttp.proxyHost=proxy -Dhttp.proxyPort=8080"
```

Alternatively, you can always set Ant’s proxy properties using the `<setproxy/>` Task, with the syntax below.

```xml
<setproxy proxyhost="firewall.example.org" proxyport="8081"/>
```
Tip: If you are using Java 1.5 or above you may also invoke Ant with the -autoproxy flag. This will force Ant to use your computer's own JVM proxy settings. Working with (X)HTML

**Consistent and Clean HTML**

It wasn't so long ago that most software developers would opt for some binary data representation over everything else. With HTML (and XML) permeating throughout the software development tool-chain, things have changed and there now exists an acceptance amongst developers that working with markup is unavoidable.

Many software developer's used to hold negative views on markup, which may have come about because HTML (and XML) was created by a group of 'docheads' (people interested in documents versus data) rather then pure programmers.

There is a natural tension between 'docheads' and software developers, as developers tend to be more 'dataheads' e.g. 'dochead' data is semi-structured with mixed content, whilst 'datahead' data looks like the stuff put into spreadsheets and relational databases (rows and columns of data).

'Docheads' are more interested in solving problems managing, publishing and handling documents, and their associated semantics. Hence, during the specification of HTML/XML the needs of software programming algorithms, performance and pure data were given a lower priority, even though it can be argued that some of these 'docheads' are world class programmers in their own right.

If the characterization of document orientated data and data is confusing just imagine the difference between looking at a schedule for your local train or bus versus looking at an application form for working as a bus or train driver.

The tabular data format of a bus schedule is less variable and only allows certain data types (time, location) arranged in rows and columns. Alternately, the application form data is less 'regular' in structure and less constrained with some fields accepting whatever you decide to write down. The semi-structured data found in the application form is much more open and can be fiendishly hard to capture using data orientated systems. Perhaps this is one of the reasons why HTML of the web was so successful in that it catered for this kind of data.

So you can't blame older software engineers, who had been working with 'train schedule' type data for the past couple of decades (using relational databases e.g. RDBMS) for feeling that the HTML and XML were 'interloper' technologies interfering with a long established way of developing software. But as technologies change and underlying hardware get faster and cheaper, assumptions also change.
The many forms of X/HTML

As shown in Chapter XML Recipes, Apache Ant can make working with markup less of a chore. For reference, here are the various different flavours of X/HTML, along with a brief description and their related doctypes.

**HTML v3.2**: added widely deployed features (in 1996) such as applets, tables and text flow around images, while providing backwards compatibility with existing standard HTML 2.0

```xml
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
```

**HTML v4**: Offered 3 flavours of HTML, strict, transitional and frameset

- **strict**
  ```xml
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/strict.dtd">
  ```

- **transitional**
  ```xml
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/transition.dtd">
  ```

- **frameset**
  ```xml
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN" "http://www.w3.org/TR/html4/frameset.dtd">
  ```

**HTML v4.01**: Contained the original HTML v4.0 plus errata

- **transitional**
  ```xml
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
  ```

- **frameset**
  ```xml
  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Frameset//EN" "http://www.w3.org/TR/html4/frameset.dtd">
  ```

**HTML v.5**: A defunct draft that serves as the basis for existing work and can be considered equivalent to XHTML 1.0 with respect to supporting deprecated functionality

```xml
<!DOCTYPE html>
```

**XHTML BASIC**

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML Basic 1.0//EN" "http://www.w3.org/TR/xhtml-basic/xhtml-basic10.dtd">
```

**XHTML 1.0**

can be considered the refactored version of the original three HTML 4 document types as applications of XML 1.0
XHTML 1.1: A compatible version of with XHTML 1.0 and HTML 4 implemented in the context of XHTML modules (insert link). It also does not handle the deprecated functionality supported by XHTML 1.0 (HTML5) or HTML 4.

XHTML Basic 1.1:

XHTML 2.0:

All versions before and including HTML v4.01 are unaware of XML. This is due to the fact that XML came about much later on the web (circa 1998), so standards organisations felt compelled to rework existing HTML to ease adoption and bring some sanity to a complex situation.

The result of all this refactoring is what is called XHTML which itself has a few different versions; this variability addresses maintaining backwards compatible markup.

XHTML, being an XML vocabulary in its own right, adheres to all the rules of XML. Furthermore, the main difference between XHTML 1.0 (insert REF) and XHTML 1.1 document type is that XHTML v1.1 has been defined in terms of XHTML modules, which means that v1.1 does not handle any of the deprecated functionality supported by XHTML 1.0 or HTML 4.

So which is one to use today? The rules of thumb I follow are:

• try to create valid XHTML, generating old HTML just means more work for the browser and harder to maintain

• if you must work with HTML try to convert it to XHTML first

• if you want to extend the XHTML vocabulary consider using XHTML v1.1
Don’t feel too guilty if you are still confused about all this, after almost 15 years of working with HTML I am still unsure but if you follow the above rules you will be fine.

<xmlvalidate/> Task

Recipe: “validate xhtml, html, etc…”

Now that I have clarified the different flavours of X/HTML I would like to present to you how to validate markup using the <xmlvalidate/> Task. In the next listing I have used the Apache Ant <xmlvalidate/> Task which will validate markup against its DTD.

```xml
<xmlvalidate failonerror="yes">
    <xmlcatalog>
        <dtd publicId="-//W3C//DTD XHTML Basic 1.0//EN"
             location="etc/xhtml-basic10.dtd"/>
    </xmlcatalog>
    <fileset dir="data" includes="**/*.html"/>
</xmlvalidate>
```

I presented the same sort of thing in previous chapters, so let’s see if I can squeeze any more useful functionality out of the native Task.

For example, with all these versions of X/HTML it would be useful to have a single approach to validating against whatever version. This can be achieved using XML Catalogs, which were designed just for this purpose. In Ant, this requires us to first define an <xmlcatalog/> element, shown in the next listing, that refers to a separate file containing the XML Catalog.

```xml
<xmlcatalog id="xhtmlDTDs">
    <catalogpath>
        <pathelement path="etc/xmlcatalog.xml"/>
    </catalogpath>
</xmlcatalog>
```

Our XML Catalog defines three dtd’s that will be used whenever systemid match in the markup document.

```xml
<?xml version='1.0'?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xmns:xml:catalog">
    <system systemId="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
            uri="file:xhtml-1.0/DTD/xhtml1-transitional.dtd"/>
</catalog>
```
This XML catalog only deals with XHTML 1.0 variants, but you could amend to include all of them. The one thing to be aware of is making sure that you either have a local copy of the dtd’s or refer to their canonical versions online.

Using them in the `<xmlvalidate/>` task is just a matter of referring to the `<xmlcatalog/>` element.

```xml
<target name="example2" description="validate xhtml 1.1">
  <xmlvalidate failonerror="no" warn="yes">
    <xmlcatalog refid="xhtmlDTDs"/>
    <fileset dir="data" includes="**/*.html"/>
    <attribute name="http://xml.org/sax/features/validation" value="true"/>
  </xmlvalidate>
</target>
```

The Task has been explicitly configured to validate using an `<attribute/>` element. Ant ships with its own XML parser as well as being instructed on any other valid XML parser.

**Validate**

```xml
<attribute name = "http://xml.org/sax/features/validation"
    value = "true"/>
```

**Use XML Schema Validation**

```xml
<attribute name = "http://apache.org/xml/features/validation/schema"
    value = "true"/>
```

**Process XML Namespaces**

```xml
<attribute name="http://xml.org/sax/features/namespaces" value="true"/>
```

**XInclude again**

*Recipe:* “create HTML pages from components using xinclude”
Breaking up web pages into reusable components is a smart thing to do and makes maintenance and development of websites easier.

The ways of compositing a web page are multifarious, I recommend using XInclude. In the following listing I show two xinclude elements which will have the effect of bringing in header and footer xml documents.

```html
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
<title>Lorum Ipsum Example XHTML page</title>
</head>
<body id="top" class="index">
<div id="container">
  <xi:include href="header.xml" xmlns:xi="http://www.w3.org/2003/XInclude"/>
  <div id="siteinfo">
    <p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.</p>
  </div>
  <xi:include href="footer.xml" xmlns:xi="http://www.w3.org/2003/XInclude"/>
</div>
</body>
</html>
```

The code sections marked in bold, contain the `<xi:xinclude/>` elements that I would like to include into the web page. The act of running an XInclude processor over this file would insert the header and footer xml files, fully replacing the xi:include element with the contents of those files.

As explained in earlier chapters, XInclude processing is a matter of configuring the underlying XML parser to perform, but this a somewhat indirect method.

A more imperative way of XInclude processing could use a 3rd party Task, called xcluder (http://sourceforge.net/projects/xcluder), which sets the XML Parser setting for you. The next listing illustrates the usual way of importing a Task for use within Ant.
Xinclude processing, using the <xcluder/> Task, is demonstrated in the example1 Target in the next listing.

```xml
<target name="example1" description="perform xinclude on xhtml">
  <xcluder in="data/test.html"
    out="result/test.html.out"
    indent="yes"
  />
</target>
```

A more ‘unix’ orientated solution (that is without using Java) could take advantage of the widely deployed and popular LibXML xmllint utility. As this is a commandline utility that uses the LibXML parser we use <macrodef/> and <apply/>.

```xml
<macrodef name="xinclude">
  <attribute name="dest"/>
  <element name="file-elements" implicit="yes"/>
  <sequential>
    <apply executable="xmllint" failonerror="false">
      <arg line="-xinclude"/>
      <file-elements/>
    </apply>
  </sequential>
</macrodef>
```

Using this <xinclude/> Task, the example2 Target resolves any xinclude statements contained in the files contained in the /data directory, copying the result into the /result directory.

```xml
<target name="example2" description="perform xinclude on xhtml">
  <xinclude dest="result">
    <fileset dir="data"/>
  </xinclude>
</target>
```
After many years of 'cracking' apart websites using webserver specific technologies the usage of XInclude is a refreshing alternative, as you typically can find an obliging xinclude aware XML parser in just about any environment, allowing you to debug & maintain without the need for a webserver.

Spellcheck

Recipe: "use ispell to spellcheck your html"

We all make errors in spelling, probably more so now that we know something else will check it for us. Here is a handy little <macrodef/> I created to do just this which uses the common unix ispell utility.

```
<macrodef name="checkspell">
  <attribute name="dir"/>
  <attribute name="log"/>
  <sequential>
    <echo message="spellchecking @{dir}"/>
    <apply executable="ispell">
      <arg value="-l"/>
      <arg value="-H"/>
      <fileset dir="@{dir}">
        <include name="**/*.txt"/>
      </fileset>
      <redirector append="true">
        <outputmapper type="merge" to="@{log}"/>
      </redirector>
    </apply>
  </sequential>
</macrodef>
```

This will check both text and HTML files and I find it useful to use as a step when deploying websites.

Keeping it consistent

Recipe: "check that a distribution of files conforms to a desired directory and file structure"

Probably like a lot of developers I constantly reuse a certain set of directory layouts as a template whenever I start new projects. Using a common distribution layout means I have to think less. The
one problem with this approach is that it can be difficult to maintain because sometimes I can be sloppy, sometimes placing files and directories in the wrong place.

Take the following layout of directories and nested directories.

- conf
  htdocs
    - resource
    - image
    - js
    - css
  - en
    - resource
      - image
    - js
    - css
  - es
    - resource
  - log

Once we have copied this directory layout template and start using it I want a process that will scan the directories and warn me whenever (I inevitably) start deviating from the original layout.

This is achieved using the `<available/>` Task which can test for the existence of a file or directory, as shown with the example1 Target in following listing.

```xml
<available
    property="direxists"
    file="data"
    type="dir"/>
```

The above `<available/>` Task will set a direxists property to true if `/data` directory exists.

A more complete approach could use Ant `<condition/>` along with `<available/>` filter to test sets of directories and files, as shown using example2 Target.

```xml
<condition Property="layout.ok">
    <and>
        <available file="build.xml"/>
        <available file="data" type="dir"/>
    </and>
</condition>
```
Extending the concept of constraining a distribution further, we could check if a file contains specific data, using the <contains/> condition.

```xml
<condition Property="layout.ok">
  <isfileselected file="data/test.html">
    <contains text="Test Distro" casesensitive="no"/>
  </isfileselected>
</condition>
```

There are all sorts of variations with the <condition/> Task and I recommend reviewing the <condition/> Task documentation to see all of the filters you can bring to bear when trying to adhere a regular structure to your distributions.

**Cleansing XHTML**

An overwhelming weight of research conducted over the years indicates that a small percentage of the web’s HTML successfully validates against a dtd or schema. What I find even more remarkable about these statistics is that an even smaller percentage of the web’s HTML is not well formed.

The internet is forgiving in what it accepts, which gives rise to a lax attitude towards ensuring that tools and technologies generate well formed HTML.

Web browsers are incredibly clever at parsing error ridden HTML and has made us all lazy, but as developers, we really should endeavour to create syntactically correct HTML as it saves time, effort, and energy.

Using popular external libraries it is possible to fix HTML markup using Ant to drive the process.

**Recipe:** "clean up X/HTML with HTML Tidy"

One of the best tools for fighting invalid and non well formed HTML is applying David Ragget’s HTML Tidy (http://www.w3.org/People/Raggett/tidy/) command line utility.

Using <macrodef/> and <apply/> Task we can emulate the command line version of this utility as described in following code listing.

```xml
<macrodef name="tidy1">
```
The tidy command has attributes set via commandline switches or by reading an external configuration file. I find it easier to use an external config file, such as the following.

```
wrap: 0
hide-comments: yes
indent: no	
tab-size: 1
tidy-mark: no
char-encoding: utf8
input-encoding: utf8
output-encoding: utf8
output-xhtml: yes
write-back: no
quiet: yes
```

The principle configuration option which will instruct Tidy to generate well formed XHTML is output-xhtml. I also always use the write-back option to ensure I always retain the original source.

- `output-xhtml : yes` - set tidy to output well formed XHTML
- `write-back = no` - set tidy to avoid overwriting existing file with cleaned up markup

More information can be found on the multitude of HTML Tidy's options here:


Run the example1 Target to see `<tidy1/>` process an XML file.

```
<target name="example1">
  <tidy1 in="data/test.xml" out="result/test.out.xml"/>
</target>
```

Now you can use Apache Ant to ‘tidy’ up your HTML and output the fixed markup to a separate file.
<jtidy/> Task

Recipe: “clean up X/HTML with JTidy”

There is a more integrated approach possible, which is done by using the Java version of Tidy, known as JTidy (http://jtidy.sourceforge.net/snapshots/jtidy/jtidy/). JTidy also has the benefit of including its own Ant Task, which we import as per the usual way using a <taskdef/> element.

```xml
<taskdef name="jtidy" classname="org.w3c.tidy.ant.JTidyTask">
  <classpath>
    <pathelement location="../../lib/jtidy/jtidy.jar"/>
  </classpath>
</taskdef>
```

The <jtidy/> Task allows the definition of HTML Tidy configuration options as <parameter/> elements, which is shown in next code listing. I like this method of configuration better, but it’s a matter of preference and you may find referring to a separate file more appropriate especially when using Tidy outside of Ant.

```xml
<target name="example2">
  <jtidy destdir="result" properties="${tidy.config}"
    <fileset dir="data"/>
    <parameter name="drop-font-tags" value="true"/>
  </jtidy>
</target>
```

The <jtidy/> Task also has the benefit of working with <fileset/>, allowing us to process many files. Run the example2 Target to see files contained in the /data directory get corrected and generated into the /result directory.

<tagsoup/> Task

Recipe: “clean up X/HTML with TagSoup”

Lets not forgo another famous ‘fix html’ utility, e.g. John Cowan’s Tagsoup (http://home.ccil.org/~cowan/XML/tagsoup). Using TagSoup can sometimes yield better results in certain situations over Tidy.

Tagsoup is written as parser, rather than a processor of HTML, but fixes HTML just as good as HTML Tidy. And we typically use a <macrodef/> to create a new Task called <tagsoup/>.
Run the example1 Target to view Tagsoup working on an XML file or example2 Target to see how it handles an HTML file.

Tip: A good introduction to using Tag Soup can be viewed here http://www.ibm.com/developer-works/xml/library/x-tipstagsoup.html

Optimising the Web

Like most people, I like my software to be fast. Good performance characteristics are usually one of the hallmarks of a successful website. On larger ecommerce sites speed can directly relate to income, which means spending time to improve performance can be a worthwhile pursuit.

The following recipes attempts to help you make your websites more performant through a series of well known techniques.

Recipe: ”Compress a set of HTML files”

The following short regular expression is popular with the Perl language crowd for compressing text and html files.
This simple expression `normalises` all occurrences of whitespace found in text, by matching all contiguous strings of whitespace and then replacing each match with a single whitespace character.

To perform this operation in Ant, I use the native `<replaceregexp/>` Task which was designed find and replace using regular expression matching to find.

In the next listing, I would like to demonstrate how I can optimize a set of HTML files using a regular expression to strip out extraneous whitespace. The example1 Target will apply this regular expression against the text contained in each file defined by a `<fileset/>`.

If the example1 Target is executed then it will process the actual files in `/data` directory. As this is an irreversible action I would propose using a `<filterchain/>` and `<copy/>` Task which means I retain the originals and generate a new set of compressed files. For now I have commented out the `<replaceregexp/>` Task in the code example so you don’t permanently compress the file examples under `/data` directory.

What I really would like to do is apply our handy regular expression to a copy of the `/data` directory, which is what the following provides through the use of a `<filterchain/>` element within a `<copy/>` Task.

```xml
<target name="example2">
  <copy>
    <fileset dir="website"/>
    <filterchain>
      <tokenfilter>
        <filetokenizer/>
        <replaceregex pattern="\s+" flags="g" replace=" ">
        </replaceregex>
      </tokenfilter>
    </filterchain>
  </copy>
</target>
```

The reader will spot the use of a `<filetokenizer/>` element which instructs the filterchain to pass the contents of an entire file as a single token, as opposed to creating tokens for each string. Without the `<filetokenizer/>` element Ant normally treats each string as a token; the effect of applying the
regular expression to each separate token can dramatically slow things down. As long as your files are not gigantic, then you should see better performance using `<filetokenizer/>`.

Running the example2 Target, will copy the contents of the /data directory and compact file contents, placing the output into the /results directory.

```html
<html>
  <body>
    <h1>Test</h1>
    <p>”Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?”
    </p>
  </body>
</html>
```

Once you run the target, you should see the html files now are stripped of whitespace.

*Recipe:* "optimising css with yui.compressor"

Yahoo's YUI.compressor (http://developer.yahoo.com/yui/compressor/) can be used to compress any text based file. The example2 Target shows how to with CSS files.

```xml
<target name="example2">
  <java jar="${jar.yuicompressor}" fork="true" output="result/style.css">
    <arg value="data/style.css"/>
  </java>
</target>
```

*Recipe:* "compacting JS with yui.compressor"
Though the main culprits for web page ‘bloat’ these days is the inclusion of popular large Javascript libraries, making Javascript an ideal candidate for compression. As YUI compressor is written in Java its complementary to using with Ant. The following shows how I directly invoke the YUI compressor using <java/> Task, as I did before with compressing CSS.

```
<target name="example1">
  <java jar="${jar.yuicompressor}" fork="true" output="result/test.js">
    <arg value="data/test.js"/>
  </java>
</target>
```

I can improve its usability by creating a <macrodef/>, for the sake of variety, I have decided to show a technique of invoking Java using the <apply/> Task, this is accommodate for how the processor accepts multiple files.

```
<macrodef name="yuicompressor">
  <attribute name="destdir"/>
  <element name="cc-elements" implicit="yes"/>
  <sequential>
    <apply executable="java" parallel="false">
      <cc-elements/>
      <arg line="-jar"/>
      <arg path="${jar.yuicompressor}"/>
      <srcfile/>
      <arg line="-o"/>
      <mapper type="glob" from="*.js" to="*-min.js"/>
      <targetfile/>
    </apply>
  </sequential>
</macrodef>
```

I now run the example3 Target, which applies the YUI processor to the /data directory, placing the output into the /result directory.

```
<target name="example3">
  <yuicompressor destdir="result">
    <fileset dir="data"/>
  </yuicompressor>
</target>
```
As per the `<mapper/>` element in the `<yuicompressor/>` Macrodef, this will only compress and output javascript files, changing the .js extension to .min.js.

Recipe: “Reduce the number of HTTP requests by concatenating JS files together”

Each external resource used by a web page (external js files, images, stylesheets, etc) creates a separate HTTP request, so if you can reduce the overall number of HTTP requests you cut down on the time and effort the server and client needs to manage these additional connections.

I can teach how to use this procedure as applied to javascript by connecting all javascript files found under the `/data` directory. The Ant `<concat/>` Task will do exactly this, concatenating all the files selected by the `<fileset/>` and placing the output into a single file.

```
<target name="example4">
  <concat destfile="result/concat-all.js">
    <fileset dir="data" includes="**/*.js"/>
  </concat>
</target>
```

HTML can now reference a single Javascript file reducing the number of HTTP Requests made by the page which should result in an overall faster load time.

There is nothing specific to Javascript about this technique and you may also do the same with multiple CSS files.

Recipe: “Measuring website performance using Apache Benchmark”

Doing both qualitative and quantitative measurements on the performance of your website is the basis of optimisation and you should try very hard indeed to ‘benchmark’ your site and understand where the site ‘feels’ slow.

There are loads of third party products that can test the performance of your website. I have chosen the venerable Apache Benchmark utility that typically comes with the Apache webserver.

Running ab, without any arguments, from the commandline reveals all its options.

Usage: ab [options] [http[s]://]hostname[:port]/path
Options are:
- `-n requests` Number of requests to perform
- `-c concurrency` Number of multiple requests to make
- `-t timelimit` Seconds to max. wait for responses
- `-b windowsize` Size of TCP send/receive buffer, in bytes
The ab utility benchmarks the performance of a webserver, by emulating one or many HTTP clients accessing a supplied URL. The response time is measured which in turn is used to generate a report indicating important network performance metrics, such as ‘how many requests per second’ a webserver can respond too.

To invoke ab I will use a <macrodef/>the calls ab with the <apply/> Task, providing the following attributes to define the test.

- **url**: URL to test
- **requests**: total number of requests from each client
- **concurrent**: number of concurrent HTTP clients making requests

-p postfile   File containing data to POST. Remember also to set -T
-T content-type Content-type header for POSTing, eg.
   ‘application/x-www-form-urlencoded’
   Default is ‘text/plain’
-v verbosity   How much troubleshooting info to print
-w Print out results in HTML tables
-i Use HEAD instead of GET
-x attributes String to insert as table attributes
-y attributes String to insert as tr attributes
-z attributes String to insert as td or th attributes
-C attribute   Add cookie, eg. ‘Apache=1234. (repeatable)
-H attribute   Add Arbitrary header line, eg. ‘Accept-Encoding: gzip’
   Inserted after all normal header lines. (repeatable)
-A attribute   Add Basic WWW Authentication, the attributes
   are a colon separated username and password.
-P attribute   Add Basic Proxy Authentication, the attributes
   are a colon separated username and password.
-X proxy:port   Proxyserver and port number to use
-V Print version number and exit
-k Use HTTP KeepAlive feature
-d Do not show percentiles served table.
-S Do not show confidence estimators and warnings.
-g filename     Output collected data to gnuplot format file.
-e filename     Output CSV file with percentages served
-r Don’t exit on socket receive errors.
-h Display usage information (this message)
-Z ciphersuite  Specify SSL/TLS cipher suite (See openssl ciphers)
-f protocol     Specify SSL/TLS protocol (SSL2, SSL3, TLS1, or ALL)
The example1 Target code listing runs with four clients each making 10 HTTP Requests.

```xml
<target name="example1">
  <ab url="http://www.antrecipes.com"
      requests="10"
      concurrent="4"
      result="result/flame-test.txt"/>
</target>
```

The results of running the `<ab/>` Task are placed in a file under the /result directory which will look like the following.

```
This is ApacheBench, Version 2.3 <$Revision: 655654 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking www.flamedigital.com (be patient)…..done
Server Software: Apache/2.2.3
Server Hostname: www.flamedigital.com
Server Port: 80
Document Path: /index.html
Document Length: 8495 bytes
Concurrency Level: 4
Time taken for tests: 4.483 seconds
Complete requests: 10
Failed requests: 0
```
Write errors: 0
Non-2xx responses: 10
Total transferred: 89740 bytes
HTML transferred: 84950 bytes
Requests per second: 2.23 [#/sec] (mean)
Time per request: 1793.338 [ms] (mean)
Time per request: 448.335 [ms] (mean, across all concurrent requests)
Transfer rate: 19.55 [Kbytes/sec] received
Connection Times (ms)

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>mean[+/-sd]</th>
<th>median</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>151</td>
<td>250 179.5</td>
<td>193</td>
<td>729</td>
</tr>
<tr>
<td>Processing</td>
<td>721</td>
<td>1361 647.0</td>
<td>1163</td>
<td>2305</td>
</tr>
<tr>
<td>Waiting</td>
<td>456</td>
<td>616 121.2</td>
<td>611</td>
<td>860</td>
</tr>
<tr>
<td>Total</td>
<td>960</td>
<td>1611 581.5</td>
<td>1450</td>
<td>2474</td>
</tr>
</tbody>
</table>

Percentage of the requests served within a certain time (ms)

50%  1450
66%  1610
75%  2297
80%  2457
90%  2474
95%  2474
98%  2474
99%  2474
100% 2474 (longest request)

I have highlighted values in bold to show the key indicators of how well a webserver performs. You will have to interpret these results in the context of the web page or application being tested. Also, tests are dependent upon the environment it is run from e.g. if you have a slow machine or slow network connection then results may show poorer performance. Its good to run ab several times and if possible from different locations which should average out any 'local' effects.

Other Web Techniques

Recipe: ‘check if a server process is running”

Some server daemons may not have smart enough command line scripts that check the process list if a server is already in existence. With Ant I can do this by checking the process list using the <exec/> Task. The following UNIX command line would check for the existence of httpd (Apache Webserver) daemon.
> ps -ef | grep -v grep | grep -c httpd -lt 1

It's also common for a server to create a file with the process ID as its name, in this case it's easy to use an Ant `<condition/>` to test for the existence of the PID file.

The Apache web server comes with a smart script; the `apachectl` batch script checks first to see if there is an instance of Apache Web server already running before it attempts to start up a new instance. So I do not have to try and check this condition in Ant.

**Tip:** Using Microsoft IIS, you could achieve the same sort of thing using the `<exec>` task to invoke the `cscript.exe` with arguments that point to VBS scripts that the IIS web server includes in its default installation. For specific information, please refer to Microsoft documentation on these VBS scripts and put your Macrodef wrapper over that.

**Recipe: ‘starting and stopping apache web server’**

Once you have deployed code you usually need to restart the underlying web server to either recognise changes or reset cache. Most popular web servers, such as Apache, come with command-line scripts to start, stop and restart the server.

As I have shown time and again, I can use the basic ‘wrapper’ approach, by using Ant `<exec/>` Task to control the starting and stopping of an Apache Web server.

```xml
<macrodef name="apachectl">
  <attribute name="operation" default="status"/>
  <sequential>
    <exec command="apachectl">
      <arg line="@{operation}"/>
    </exec>
  </sequential>
</macrodef>
```

Here is how the Macrodef would be used in an Ant Target that starts then stops the Webserver.

```xml
<target name="start-and-stop-webserver">
  <apachectl operation="start"/>
  <apachectl operation="stop"/>
</target>
```

Here is a list of some of `apachectl` operations;
• **start** - start httpd

• **stop** - stop httpd

• **restart** - restart httpd if running by sending a SIGHUP or start if not running

• **graceful** - do a graceful restart by sending a SIGUSR1 or start if not running

• **configtest** - do a configuration syntax test

• **status / fullstatus** - The fullstatus and status operations require further configuration (e.g. enable mod_status in Apache) for them to return status information properly.

**Service Status**

*Recipe:* "check status of webserver, email and ftp server"

‘Are my servers running?’ is a question I ask myself, often ... if not a few times on a daily (and more importantly nightly!) basis.

Using Ant’s `<conditions/>` I can test if a service is up and running or check if a web page is available. The following Target does just this by using a `<condition/>` in conjunction to a `<fail/>` Task which will throw an error if the `${server.up}` Property does not exist.

```xml
<target name="example1" description="check if server is up">
  <condition Property="server.up">
    <and>
    </and>
  </condition>
  <fail unless="server.up" message="webserver is down."/>
  <echo message="status:${server.up}"/>
</target>
```

I have used the `<fail/>` Task but you may want to do something else; like notify whenever something is down, e.g. send out an email (or an SMS).

Running the example1 Target should hopefully show a status of true

*Buildfile: 3_more_web/2_server_up/build.xml*

example1:

  [echo] status:true
BUILD SUCCESSFUL
Total time: 16 seconds
The time it takes to perform this Target may seem long depending on your system and network configuration (timeout, network speed, etc).

For our next example, I could choose to ‘inject’ Ant properties, themselves set by <conditions/>, into an XSLT transformation to generate an HTML ‘dashboard’ showing the status of the service.

I will extend the example1 Target to check an email and ftp server, in addition to the web server.

```xml
<target name="example2"
    description="generate a web page detailing server health">
    <condition Property="webserver.up">
        <and>
        </and>
    </condition>
    <condition Property="emailserver.up">
        <and>
            <socket server="mail.example.org" port="25"/>
        </and>
    </condition>
    <condition Property="ftpserver.up">
        <and>
            <socket server="ftp.example.org" port="21"/>
        </and>
    </condition>
    <xslt in="data/site.xml" style="etc/server-status.xsl"
        out="result/site-status.html">
        <param name="webserver.up" expression="${webserverup.up}"/>
        <param name="emailserver.up" expression="${emailserverup.up}"/>
        <param name="ftpserver.up" expression="${ftpserverup.up}"/>
    </xslt>
</target>
```

After checking the services status, I then apply an XSLT transformation which will show red or green based on the value of the parameter.

*Tip: You could replace the etc/server-status.xsl with a xslt that generates an ATOM / RSS feed.*

Enabling XUL development
**Recipe:** “develop and deploy a Firefox toolbar”

XUL, is Mozilla XML User Interface Language used in the development of cross platform applications of which the Firefox Browser is an example. XUL employs a range of common technologies (Javascript, DOM, CSS, etc … ) which can build full blown applications as well as extend existing applications. This recipe will build an XUL toolbar with Ant helping us along the way.

You can get more information on XUL at Mozilla official site

   http://www.mozilla.org/projects/xul/

In the creation of a XUL Toolbar, one would follow these steps.

1. define a specific directory structure
2. use css for styling the toolbar
3. define XUL markup which determines toolbar elements
4. define javascript to define toolbar element behavior
5. package and deploy the toolbar as a jarred and zipped .xpi file

Steps 1 - 4 relate to building the toolbar itself, but to test in Firefox, I will need to generate step 5. This becomes boring quick which is where Ant comes in.

The final .xpi file is just a zip file containing a jar (which itself is a zip file) and a bit of meta description (install.rdf). to create a .xpi you need to first <jar/> up the data/chrome directory and then <zip/> up this plus install.rdf.

```xml
<target name="jar">
   <zip destfile="data/chrome/toolbar.jar">
      <zipfileset dir="data/chrome"/>
   </zip>
</target>
<target name="dist">
   <zip destfile="result/toolbar.xpi">
      <zipfileset dir="data/chrome" includes="toolbar.jar" prefix="chrome"/>
      <zipfileset dir="data" includes="install.rdf"/>
   </zip>
</target>
```

Running the example1 target will do just this
Buildfile: 3_firefox_dev/build.xml
/
clean:
  [delete] Deleting: 3_firefox_dev/data/chrome/toolbar.jar
  [delete] Deleting directory 3_firefox_dev/result
  [mkdir] Created dir: 3_firefox_dev/result
jar:
  [zip] Building zip: 3_firefox_dev/data/chrome/toolbar.jar
dist:
  [zip] Building zip: 3_firefox_dev/result/toolbar.xpi
run:
BUILD SUCCESSFUL
Total time: 1 second

You should now be able to use the generated .xpi file within your Firefox browser.

Ant is firstly a build tool so using it to help you build a deployment package is a natural application of it. You can find Ant being used in many of the Javascript framework projects as well as 3rd party Ant tasks that help you build web technologies, like

http://stefan.samaflost.de/blog/en/Java/GWT

which makes it easy to work with Google Web Toolkit.

Web site maintenance

Recipe: “Creating a web backup”

Even if you are using source control, you will probably always want a copy of the live site which may also use a database. For small to medium sized websites backing up could just be what you normally do from the command-line, for example zipping up a directory and exporting from the database.

The following script shows one approach to backing up a MySQL database and web directory.

```xml
<?xml version="1.0"?>
<project name="BackupWebApplication" default="backup">
  <description>Backup web server distribution and associated database</description>
  <tstamp>
    <format property="datetime" pattern="mm-dd-yyyy-hhmmss" />
```
The backup Target calls backup-web and backup-mysql-data Targets.

- The backup-web target: tars then gzips data
- backup-mysql-data: invokes mysql's own command for dumping database (mysqldump)

A better way would be to encapsulate these targets as `<macrodef/>`'s.

The macrodef version of backing up SQL would look like this:

```xml
<macrodef name="backup-sql">
    <attribute name="datasource" description="mysql datasource"/>  
    <attribute name="target" description="target directory to write tar.gz file to"/>  
    <attribute name="prefix" description="prefix to add to tar.gz file"/>  
    <attribute name="compression" default="gzip" description="we want gzip compression by default"/>  
    <sequential>
        <tstamp>
            <format property="datetime" pattern="mm-dd-yyyy-hhmmaa"/>  
        </tstamp>
        <exec command="mysqldump datasource" output="backup\datasource.sql"/>
        <gzip zipfile="backup\datasource${datetime}.gz" src="backup\datasource.sql"/>
        <delete file="backup\datasource.sql"/>
    </sequential>
</macrodef>
```
<exec command="mysqldump @{datasource}" output="@{target}\@{prefix}-@{datetime}.sql"/>
<tar compression="@{compression}"
tarfile="@{target}/@{prefix}${datetime}.tar.gz"
basedir="@{target}\@{prefix}-@{datetime}.sql" longfile="omit"/>
<delete file="@{target}\@{prefix}-@{datetime}.sql"/>
</sequential>
</macrodef>

Which uses the mysqldump command again in combination with <tar/> ability to perform gzip and tar operation all in one.

And to back up the web directory, the <macrodef/> looks like:

<macrodef name="backup-dir">
  <element name="filter"
    description="element is nested in tarfileset"/>
  <attribute name="source"
    description="target directory to write tar.gz file to"/>
  <attribute name="target"
    description="target directory to write tar.gz file to"/>
  <attribute name="prefix"
    description="prefix to add to tar.gz file"/>
  <attribute name="compression" default="gzip"
    description="we want gzip compression by default"/>
  <sequential>
    <tstamp>
      <format property="datetime" pattern="mm-dd-yyyy-hhmmaa"/>
    </tstamp>
    <tar compression="@{compression}"
      tarfile="@{target}/@{prefix}${datetime}.tar.gz"
      longfile="omit">
      <tarfileset dir="@{source}" mode="755" username="ant"
        group="ant">
        <filter/>
      </tarfileset>
    </tar>
  </sequential>
</macrodef>
This macrodef means we can use filters for selecting the files we want for backup.<

```xml
<backup-dir source="@{websrc}" target="${app.dir.backup}"
    prefix="${app.prefix.web}"/>
    <filter>
        <include name="**/.*css"/>
        <include name="**/.*gif"/>
        <include name="**/.*html"/>
        <exclude name="CVS"/>
    </filter>
</backup-dir>
```

**Source Code Bonus: Web Services**

With the rise of the Internet as a kind of ‘operating system’, there is an ever increasing level of website functionality being exposed as Web Services and feeds.

I have added bonus source code which shows how one would interact with Google web services; this is a much more advanced application of Apache Ant, in most situations you can use the basic HTTP techniques to work with feeds and web services but I thought it may interest some readers a more advanced example. You will have to amend properties with your own Google credentials.

*Tip: Check out how Apache Ant can help you with your Google App Engine projects here [http://code.google.com/appengine/docs/java/tools/ant.html](http://code.google.com/appengine/docs/java/tools/ant.html)*

**Conclusion**

In the panoply of software engineering, web development has now become a mainstream prerequisite skill which is why I started off the chapter showing how to work with raw HTTP requests using Ant.

I then illustrated how to can increase the quality of websites, by letting Ant take care of chores like validating web pages or spellchecking. Using Ant can also help you quantify performance as well as apply optimizations (such as compression) resulting in faster websites.

I especially enjoy using Ant to automate as much as possible in the web development tool chain, which is why I showed a series of recipes from backing up web servers to working with Firefox XUL.
When used in this capacity, you also get the benefit that Ant is a description of the process instead of an arcane invocation, useful for communicating intentions to other developers who may have to use or deploy your code.
Case Study: Working with Multilingual Websites

Many websites provide content in several different languages catering for a worldwide audience. In this chapter, I will demonstrate how to do this and create and deploy a multilingual website using Ant, XML, XSLT and HTML Content Negotiation.

The chapter’s core focuses on a discussion of the sample website’s underlying code, and how it is implemented. In particular, I see how to:

• **Work with source XML/XSLT**: I review construction of the source XML and XSLT documents which comprise the multilingual website and with which editors and developers work.

• **Generate the Distribution files**: I look at the creation of an Ant build file that generates the basic multilingual website. This in turn creates HTML which can be processed by HTTP/1.1 content negotiation rules.

• **Implement Continuous Publication**: Presentation of techniques for automating the publishing workflow using a variety of approaches, including cron jobs and Martin Fowler’s CruiseControl tool.

I thought it would be useful to include a chapter that presented a more complete solution using Ant, so I have focused on the synthesis of a few techniques instead of picking out specific recipes.

Background

A few years ago, in Switzerland, I found myself fumbling with a Swiss German keyboard. I wasn’t surprised to discover that when accessing www.google.com I was served the German version. Changing my preferred language setting to French, I refreshed the browser and, as expected, received the French version of the popular search engine.

Since I speak neither of these languages I configured English as the browsers preferred language. All major browsers allow the user to define an ordered set of preferred languages.

In Internet Explorer, you can do this from the Tools menu option, followed by the Languages… button on the main Explorer configuration tab page. The dialog box permits the user to choose multiple languages. Note that some major languages (for example, de) also include some variants (such as de-ch).

However, setting your browser’s language preference is no guarantee that the visited website will — or are able to — provide your desired language version. In the case of my borrowed Swiss German computer, the preferred language was Swiss German (de-ch). This meant that I should
receive, at a minimum, a German (de) version of a website’s resource, had one existed, and if the local swiss variant exists to deliver this to the browser.

On certain websites I discovered that I received a French version. Obviously, my browser’s settings did not explain this behavior, the server was making decisions about which fall-back language a Swiss German user may want.

Knowing that HTTP/1.1 Content Negotiation was the magic in the system, I was intrigued by the possibilities of using this in combination with XML, XSLT and Ant to assist in creating multilingual websites.

**Critical Concepts**

In this section, I examine issues related to the following:

- Content Negotiation
- XML and XSLT
- Ant Tasks and Targets
- Continuous Publication

**Content Negotiation**

Content Negotiation works by adding special HTTP Headers (Accept and AcceptXXX). When a browser POST or GET an HTTP Request, to a web server, which includes these Accept headers, the server tries to return resources and content in the correct language.

A wide assortment of content-related rules can be constructed using Accept headers, with each of these headers defining instructions to the receiving web server regarding which types of resources may be accepted by the browser. Content Negotiation is effectively about how a server responds to a browser’s Accept settings.

Sample Accept type headers are shown below.

```
Accept: text/xml, application/xml, application/xhtml+xml, text/html, text/plain, video/x-mng,image/png, image/jpeg,image/gif;q=0.2,text/css,/*;q=0.1
Accept: */*
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Accept-Language: en
```
Accept headers are not purely defined in terms of different language versions, but can be used to generally define which content is desired by the consuming browser.

Accept headers can define which resources and forms of resource that it will accept from a server. You will see in the above examples, that there are many different variations of the Accept header, with the q numbers providing a system of weighting which determines precedence of any particular resource. Alternately, the Accept: */* example informs the server that the browser will accept any form of any particular resource.

Accept headers give the browser client a way to inform servers of their preferences and capabilities: what they want, what they can literally accept, and what they don’t want. Servers then use this extra information to make more intelligent decisions about what to send. Accept headers benefit both sides of an HTTP connection. Clients get the form of resource that they want, and servers don’t waste time processing or sending some resource to a client which it can’t use, saving on bandwidth.

*Tip: Use Firefox extension LiveHTTPHeaders to inspect headers being sent from the server to your browser.*

**How Does this Work on the Server?**

On the server side, resources are provided as normal (that is plain html files) but are further identified with an additional file name extension. By resources, I mean html pages, images, basically any resource (with associated mime type) that a web server can serve up.

A browser notifies the server of its language preference via the addition of the following HTTP Accept header to the browsers HTTP Request:

```
Accept-Language: fr
```

In this case, the browser would be served up the content.gif.fr file. This convention applies wherever a URI exists within an HTML page, be it an img src attribute or a CSS stylesheet’s href attribute. In other words, if a web page is requested by this browser, all the derivative requests will also conform to what the browser wants as well…of course this is just because the browser is requesting these assets as it is parsing the HTML and compositing the page.

A more complex request (as found in Apache’s documentation), involving multiple Accept headers, is shown below:

```
Accept-Language: fr; q=1.0, en; q=0.5
Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6, image/jpeg; q=0.6, image/*; q=0.5, /*; q=0.1
```

Here, the browser tells the server that while it would always prefer French, if that doesn’t exist then English will do, additionally defining ‘q’ attributes which assist the server in weighting its selection.
This example also specifically defines media type preferences—in this case, HTML over plain text, and GIF over JPEG for image media types.

This technique works perfectly with the Apache server’s default httpd.conf configuration file and I will not be asking you to adjust this default configuration for our solution to work properly. Its through the customisation of this configuration that I can provide fine grained control over how the server responds to Accept-Language requests.

Note: The Apache Content Negotiation section of the Apache manual contains detail of configuration options. You should find that Content Negotiation is normally enabled in a plain vanilla installation, though if this is not the case you will need to refer to the Apache manual to get things working.

XML and XSLT

The solution presented in this chapter requires us to utilise XSLT processing. Processing consistently across XSLT processor implementations is no mean feat. The best practice involves use of a simple template that employs the most common XSLT elements. The following list summarises XSLT processors that achieve reliable XSLT processing:

- **Latest Mozilla / Firefox**: The latest browser version contains support for EXSLT and the Mozilla XSLT engine is reliable & robust.

- **Latest MSXML / Internet Explorer**: Wide deployment with built in xslt processor makes this an obvious client-side platform to target. Recent 3rd party advances e.g, EXSLT for MSXML4 by Dimitre Novatchev (http://fxsl.sourceforge.net/) have incorporated limited EXSLT support, which is crucial in order to reduce dependency on proprietary functionality.

- **Latest Saxon XSLT Processor**: Michael Kay’s Saxon is currently the standard against which all other XSLT implementations are measured. Some techniques presented in this book use XSLT 2.0, which is (as of this writing) only supported by the latest versions of Saxon. I use SAXON throughout the book as it supports the largest subset of XSLT 2.0 and XSLT 1.0 specifications.

There are other quality XSLT processors in existence but I only tested the solution with the above XSLT processors.

Server-side vs. client-side XSLT processing

For our multilingual website I use pairs of XML and XSLT files. For final display in the end user’s browser, files will be transformed into an XHTML format using our Ant XSLT transformation techniques. I therefore need an XSLT processor to render our data into XHTML.

I could create a batch process or set up server-side XSLT processing to render the source files into XHTML web pages whenever they are requested by a browser. A simpler method, involves use of a
browser with built-in (client-side) XSLT processing capabilities. This is ideal in many commonplace instances, such as a company intranet accessed only by employees using IE 6.0. In this case content editors and web developers would only need a local copy of the source file and could view changes immediately without having to transform to view such changes.

Having the ability to use both a client or server side XSLT processing can be advantageous as it allows our solution’s XML/XSLT to be compatible in any situation.

**Schemas and Namespaces**

As I talked about in XML Recipes chapter, there are many methods for formalising, validating and constraining your home-brewed XML vocabularies. These can include:

- DTD
- W3C XML Schema
- RelaxNG
- Schematron, XSLT, etc...

Working without a schema is a serious design decision with far-ranging ramifications. Schemas can constrain with the structure and content of your XML data. For those SQL veterans, its a bit disconcerting to define data without any form or constraint, as when a field is defined in a table...its very type is defined at the beginning of its lifetime; changing its type could cause potential problems such as mucking about with referential integrity. XML allows for schema validation to be applied as a separate and potentially plural set of processes.

As is the case when using schemas, diminishing gains are apparent when slavishly placing all XML under its own namespace. While in general it is good practice to place your own XML into its own namespace, this will make your XSLT significantly more difficult to read.

The web is a highly distributed system of documents, which allows people to use or not use schemas, even though I have a DTD for the varieties of HTML...their usage was never required. Even considering XML as the lingua franca of data exchange, it is unlikely that there will ever be universally accepted XML vocabularies for every type of data that may be encountered. This statement leads us to the conclusion that neither will there be a universally accepted schema technology. Supplying a schema would therefore aid interoperability and integration, but it is not critical.

The urge to apply rigorous taxonomy — such as namespaces and schemas – can be resisted until the developer is more confident that source XML documents are not going to change significantly. So this XSLT antipattern could be described as ‘Early Taxonomization’. Don’t be afraid to make ad hoc XML, though as you continue to build up such XML distributions, you should recognize a junc-
ture at which it’s time to formally describe your data. This may occur when the need to use external editors to maintain content arises, or when you want to integrate in-browser editing.

All these events signify the moment when your publication is turning into an application. It will at that point require the formalism which application development requires.

In many cases you may find that your application is operating on the edges of the public Internet, and possibly integrating with behind-the-firewall applications, you may then be presented with little choice but to provide schemas to assist in integration.

**Ant Tasks and Targets**

Since Apache Ant is the controller of our publication process, it is best to have the build process mirror the familiar Project/Target/Task structure within Ant. At present I focus only on requirements. I look at the code a little later in the chapter.

**Entry Target**

I want to ‘push a button’ to enable the build to progress through a series of processes. Ant’s default target (or ‘entry’) point should therefore sequence the execution of dependencies which perform everything required for a successful build.

Running the entry target should tell us whether the build was successful. I may also enable some notification (such as email) or report, which details how the build progressed.

**Targets for Directory Management**

For this website I’ll use two directories: one for source files which contains all the XML, XSLT and resource files (such as images), and another for final distribution files. So I need some general Ant targets for managing the copying, deleting, moving and renaming of files and directories. This type of directory management is common when using Ant.

**XSLT Processing Targets**

I use XSLT and Saxon to transform the source XML into various representations of the data. I therefore require a set of tasks that may be called easily via Ant’s `<java/>` task to perform these transformations. If I have multiple languages, then I probably need to perform repetitive processing. Some of this may be more easily carried out by Ant rather than using XSLT.

**Language Processing Targets**

Ant controls the generation of multiple language versions of a website, it is only natural that I need to perform some language dependent processing. The distribution directory will contain the results of XSLT transformations, as well as all associated binary assets (such as images, JavaScript and Flash) which may require the application of such language-dependent processing.
Continuous ‘Publication’

The concept of Continuous integration espouses quick and reproducible software build process that automate the testing, building, integration and deployment of software through time.

‘An important part of any software development process is getting reliable builds of the software. Despite its importance, I am often surprised when this isn’t done. Here I discuss the process that Matt has put into place on a major project at ThoughtWorks, a process that is increasingly used throughout the company. It stresses a fully automated and reproducible build, including testing, that runs many times a day. This allows each developer to integrate daily thus reducing integration problems.”—Martin Fowler http://www.martinfowler.com/articles/continuousIntegration.html

The programming concept of continuous integration, that is quickly becoming a defacto addition to most development methodologies, can also be applied to the publication of electronic documents, instead of software, which I have called Continuous Publication.

Overview of Multilingual Site

This chapter’s code sample creates a small site which utilises frames, CSS, images, href tags, Flash and is designed to support multiple languages (fr, en, it, de).

**Home Page:** The home page consists of a typical 4-frame index page which contains header, footer, main and navigation frames, shown below:

**Navigation:** The navigation frame in the home page links to the other example pages: About, Contact, Products, Services and News. Here is a screenshot of the example website.

To view the website open up the following html file in your browser (from the source included for this chapter), which is provided as a reference of how things should look after processing.

expected_result/website/navigation/index.html.en

The site has been kept simple to let me focus on the Content Negotiation techniques being described, with effort on the websites design itself. Textual content is clearly marked with the appropriate language.

Developing a complex, multilingual distribution of HTML can be just as demanding as developing software. This is especially true given the breadth of contemporary, functionally-rich sites with their use of a menagerie of xml based vocabularies.
In our example, Apache Ant controls the creation and deployment of a language-dependent, static website. I can configure Ant to reprocess whenever a change is made to the source XML or XSLT documents. This provides the basis for our ‘continuous publication’ process, in which the publication workflow is automated.

This means that with every subsequent change in content or layout, website generation remains straightforward and easy. I directly apply principles of continuous integration—as presented in the previous section—to publishing and maintaining a multilingual website.

In such a publishing process it maybe necessary for developers and content editors to work on XML, XSLT and resources at the same time. The production version of our website would consist solely of XML and XSLT, along with images, links, and other web resources.

I will expand on how this achieved, but here are some of the highlights;

• By employing such a standalone XML version it should be easy to generate and maintain site content using third party editors or customized ‘in browser’ solutions.

• Though the use of the xml:lang attribute throughout the source XML distribution, the content editor can supply language-dependent content.

• An inheritance hierarchy of XSLT templates must be created. These will give granular control over the rendering of language-dependent content such as text, CSS, HTML, images, meta tags etc. These XSLT templates should be designed and implemented so that the site build is modular, making it easier for web developers to layout and publish web content while allowing content editors to work on separate XML source documents.

• Language-dependent content — used in combination with the Apache server’s implementation of Content Negotiation — means end users can view correct language content based on their browser’s default language setting.

Requirements

You need the following components in order to run the code example presented in this chapter (brief setup instructions follow):

• Java™ 2 Standard Edition Runtime Environment: http://java.sun.com/j2se/1.4.2/runtime.html

• recent version of Firefox, http://www.mozilla.org, or Internet Explorer (5.0 with MSXML or later), http://windowsupdate.microsoft.com
• Apache Ant: http://ant.apache.org

• recent version of the SAXON XSLT Processor: http://saxon.sourceforge.net (a copy of this library has been included)

• Apache Web Server, with Content Negotiation enabled: http://www.apache.org

• Zip archive containing simple and advanced Ant build files, source XML/XSLT, and sample distribution and report files: http://www.ruminate.co.uk/multilingual.zip

The above software runs on most Windows platforms as well as UNIX and Mac OS X.

The target production client should use a browser with built in (client-side) XSLT processing capabilities such as those in Internet Explorer (MSXML) or the latest Mozilla. This will allow a web developer or content editor to fully view the standalone XML/XSLT version, as the web pages are dynamically being transformed in the client browser as opposed to being parsed and published to final HTML format.

Apache Ant

Being the last chapter of the book I hope that you have Ant installed by now, but here is a quick review just in case:

• Include ant/bin directory in your PATH environment variable

• Include ant/lib/ JAR files to your Java CLASSPATH, though this is not mandatory it will ensure that you can use Ant programmatically

• Add an ANT_HOME environment variable with /path/to/your/ant/ directory

• Please reference Ant’s documentation for installation or configuration problems.

• should have Saxon XSLT Processor installed

Ant has built-in XSLT processing capabilities through use of the excellent Apache Xerces XSLT implementation. The Ant task responsible for transforming XML is called <xslt/> and uses Xerces by default.

We’ll use the latest version of the Saxon processor that has XSLT 2.0 support. This gives our Ant transformations the ability to leverage XSLT 2.0 features.

Within the build file I will need to tell Apache Ant where the Saxon processor is by defining the jar.saxon property:
In theory, if you had included the Saxon.jar onto your Java CLASSPATH then you should not have to adjust the above property. You may also place a jar into into Ant’s own /lib directory though doing this, or adding it to your classpath, just binds your build.xml with the underlying environment. It is better practice to explicitly inform Ant which JAR to use from your build file instead of depending on some configuration of the environment.

*Tip: The example build download for this chapter has all the libraries you need, just run the default build.*

**ASF Apache Web Server**

Version v1.3 / v2 Apache web server is required for content negotiation to work. The author suggests using the latest stable v2 version of Apache.

Apache enables HTTP /1.1 Content Negotiation rule processing via its native mod_negotiation module, which selects the best representation of a resource based on client preferences for languages, media type and character encoding.

The logic for choosing the appropriate representation of a resource can become quite complex. A set of Apache directives define the heuristics, instructing Apache how to handle almost every possible scenario. Default httpd.conf settings are appropriate, though you can refer to Apache’s Content Negotiation documentation for further information.

To enable Apache Content Negotiation, the native mod_negotiation module needs to be activated. First, locate the Apache conf file — httpd.conf is in the /conf directory under the Apache distribution. On UNIX machines it is usually found under /usr/local/apache/conf or /etc/httpd/conf/httpd. On Windows, httpd.conf (or apache2.conf) is typically in C:\Program Files\apache group\apache\conf.

If the Apache build dynamically loads such modules, then you can use the following Apache directive to enable:

```
LoadModule negotiation_module modules/mod_negotiation.so
```

If this statement is missing, add or uncomment it then restart Apache. An error will occur if the module is not present or if Apache hadn’t been configured to work with dynamic modules. If you lack dynamic module loading capability, you need to explicitly recompile Apache with mod_negotiation activated. Please refer to Apache’s documentation for platform-specific instructions.
The next configuration involves enabling Multi_views for the web directory into which is placed source and distribution. Depending on how you have Apache set up (virtual directories or simple combinations of Alias and <directory/> directives), this may already be activated. If you see an Options keyword but no Multi_views keyword, add the keyword Multi_views after any other keywords which follow the Options keyword.

In cases where you have defined virtual directories (using <virtualhost/> ) or have added new web roots (using <directory/> or <location/>), add the following Options directive in the body of the directive:

```
Alias /multilingual/ "/path/multilingual/zip/archive"
<Directory "/path/multilingual/zip/archive">
    Options Multi_views
</Directory>
```

This sample Apache directive shows a path to a directory into which you should have decompressed the multilingual zip archive. Next, save the httpd.conf file and restart Apache. Refer to the build/example_httpd.conf file included in the code download for this chapter if you have problems. The example_httpd.conf file illustrates the correct configuration for a ‘plain vanilla’ Apache 2.0 server.

To confirm that Apache Content Negotiation is working, navigate to http://localhost/manual which is the Apache manual that comes with the webserver.

The browser should now show the Apache manual in your currently preferred language. A more thorough test involves changing your browser’s language defaults. Try French (fr), and then navigate back to the manual. Then, try a third language. Refreshing the browser accelerates this test.

**Structure of the Ant Project**

As with any Ant project, it is useful to map out the starting directory structure:

- **data**: contains the source XML/XSLT files for building; Ant creates it and can empty it in the ‘clean’ project.
- **result**: Distribution outputs go here; the directory is created in Ant and ‘clean’ empties it out
- **lib**: imported Java libraries go here
- **expected_result**: I have included a directory containing what should be generated to the /result directory
I will need a source directory (data) for the XML and XSLT files and web resources. Since I generate the website from this source directory, I will need to enable our build to generate a distribution directory (result), which in turn is deployed onto a properly configured web server.

The src directory contains all XML and XSLT files, along with resources. The source documents are what designers, developers and content editors work on. They should be able to render locally in XSLT-enabled web browsers, allowing changes to be viewed quickly and independently of any server-side mechanism.

**The Result**

The dist directory contains the output of the build process, and will ultimately be deployed onto the target server. This directory is the multilingual website, configured for use with Apache mod_negotiation. It consists of flat HTML files for each language version and all related web resources. These flat files are the result of performing an XML/XSLT transformation using Saxon via Ant.

**Running the Code**

In order to run the build on your machine, it’s important that you adjust the build’s properties, defined in build.xml described below.

**jar.saxon**

This property should contain the absolute path to your Saxon XSLT processor JAR. This book uses Saxon (v.7 which supports XSLT 2.0) which uses the net.sf.saxon namespace. For the build to work, you need this property to contain the correct value.

**dir.src**

Defines name of source directory relative to the basedir attribute of the `<project/>` element.

**dir.dist**

Defines name of publication directory relative to the basedir attribute of the `<project/>` element.

The build should work as long as the jar.saxon property is configured properly.

To run the build, just execute Ant in the build directory or cd to wherever you unzipped the example for this chapter:

```
> ant -buildfile build.xml
```
Ant can also run with a logger which outputs a log of how the build progressed. The XmlLogger generates valid XML which can be processed by a stylesheet to present your log data. The run-withlog.bat file contains the command:

> ant -logger org.apache.tools.ant.XmlLogger -verbose -logfile etc/build_log.xml

The resulting log file is written to the build directory and can be viewed in a browser which has a built in XSLT processor.

When running you will see Ant complain of some errors which are related to applying the XSLT transformation everywhere … wherever an XML file with an XML PI is found, SAXON will generate and save the results in a file. Since I store both XML and XSLT files in the same directories (under /data), SAXON throws an error when it encounters an XSLT file. This is just a warning and processing should not stop.

Buildfile: chp4/build.xml

create:

  [mkdir] Created dir: chp4/result

copy_resources:

  [copy] Copying 19 files to chp4/result
  [copy] Copied 11 empty directories to 4 empty directories under chp4/result

generate_dist:
generate_language_versions:
transform_src:
  .... lots of stuff ....

build:

  [echo] success: language versions (/dist) built from source (/src)

BUILD SUCCESSFUL
Total time: 6 seconds

If this is the first time you have run the build, the reader will notice that a /result directory has been constructed which contains the generated multilingual website.

The generated output is pure HTML, so to view either; directly open /navigation/index.html with any browser or place the /result under a webserver root directory (ex. htdocs) and view through browser though using a proper URL e.g. http://127.0.0.1/result/navigation/index.html

Hopefully you are now clicking through the sample website.

Note on Caching and Content Negotiation
A browser may cache a web page, saving it locally, allowing for the local page to be served up next time it is requested which is much faster than downloading. Caching may also occur on a server, this is especially effective if the web page in question is composited from a variety of dynamic database calls. Your ISP may employ caching either through a proxy or some other server side technology, which serves up a cached copy of a web page from their own local store.

**Web browser cache**: Your web browser can store recently visited HTML or even synchronize certain pages to be available offline. When testing Content Negotiation its important to turn off caching by either setting the amount to cache to zero or explicitly deleting the cache.

**Proxy cache**: If access to the internet is via a proxy, this can mean that multiple requests of the same resource get cached at the proxy. This technique reduces bandwidth usage as the proxy will check to see whether it has a fresh copy of a resource (e.g. HTML page, image, css file) and if it does it will serve that instead of going out to the Internet site to retrieve it. Embedding PRAGMA metadata tags in your /data documents can assist in forcing the proxy to always conduct lookups from the Internet instead of caching. So caching, in the web context, works by storing a representation of an HTML page associated with a particular request URL. The next time that URL is requested by the browser, the cache representation is used rather than requesting for the resources again.

If a resource is content negotiable at the server, you need to be aware of the problems that caching can cause. For example, if an HTTP Request is performed requesting an English variant of a web page, where an incorrectly configured cache is involved, may end up responding to subsequent HTTP Requests with the same cached version irregardless of language variant requested.

**Looking at the Code**

As I have reviewed the technologies and techniques that make up the solution I now turn towards the actual code that comprises the website and the build that embodies the publishing process.

I have broken down the code review into three categories:

- Construction of the XML and XSLT documents which serve as the basis for the multilingual website.

- Creation of the Ant build file which generates the basic multilingual website (a set of HTML files) able to be processed by HTTP/1.1 Content Negotiation rules.

- Automation of the publishing workflow using “Continuous Publication” techniques.
Working with Source XML / XSLT Distribution

I have seen many approaches to the creation of XML/XSLT distributions—from the older Docbook website DTDs to more contemporary approaches such as Apache’s Forrest documentation tool and Apache Cocoon’s web publishing framework (as employed using the ASF Lenya add-on).

It may thus seem a bit strange that to date, there has yet to emerge any single open source schema or markup language (apart from X/HTML of course) for creating websites and representing the content in them (apart from XML!)—not to mention one even attempting multilingual websites.

Our source XML represents an attempt to define a plausible format to use with multilingual websites. There is no schema involved. Instead, I demonstrate an approach that uses an amalgamation of XML and XSLT to create a website. The general technique has a low dependency on the actual format we’ve chosen so I am able to easily adapt this XML, or to use it as a basis for your own approach.

When creating a source XML distribution from scratch, the following challenges must be considered:

• Building an XSLT hierarchy to handle multiple languages: one that works within the context of both server and client XSLT processing

• Structuring the XSLT in a way that promotes easy modularization of the website, utilizing notions of global variables, templates, the separation of layout from content, etc

• Structuring XSLT so it is easy to layout generic web assets such as images, CSS, HTML, JavaScript, etc using modular presentation of XSLT components

• Ensuring that each language version will have the proper textual information, including correct DOCTYPE tags, HTML Lang attributes, meta tags, and so on

• Structuring the /data XML so it is easy for content editors to supply language dependent content

• Ensuring that the /data distribution can be processed by Ant without creating dependencies on 3rd party extensions

As I walk through the creation of the /data distribution of XML/XSLT, you’ll see the solution to these challenges.

There is a clear workflow in website development, whereby designers create XHTML which can be generated from a source distribution of XML and XSLT. This source distribution can be maintained by content editors and developers, and then published into final HTML format.
Defining the Structure of /data

The listing below shows only the ‘page’ documents of the website. Such pages are undistinguished insofar as one can find such a structure on most websites:

**Home**: 4 frame index page containing appropriate frameset code
- index.xml
- index.xsl

**Header / Footer**: defines the header and footer that appears on every page
- header.xml
- header.xsl
- footer.xml
- footer.xsl

**Navigation**: Left hand frame contains navigation links to all pages
- navigation.xml
- navigation.xsl

**Main**: The main frame will contain the following pages, when a link is clicked.
- main.xml
- main.xsl

**About**: Illustrates the use of images in /data distribution, as well as the content negotiation aspects when desiring to serve up language dependent versions of images. If that wasn’t enough I threw in some RDF in the form of dublin core.
- about.xml
- about.xsl

**Contact**: Contains EXSLT example
- contact.xml
- contact.xsl

**Products**: Contains JavaScript example
- products.xml
- products.xsl

**Services**: Contains Flash example
- services.xml
- services.xsl

**News**: Contains an example of generating multilingual news articles.
- news.xml
news.xsl

**Global:** Contains globally accessible xml and xslt
  - global.xml
  - global.xsl

**Asset:** Contains xml and xslt associated with web resources; css, images, etc…
  - asset.xml
  - asset.xsl

**Structure of the XML/XSLT Pairs**

In all of these pages, the XML and XSLT code adheres to a consistent convention. The XML contains a page’s metadata and the XSLT defines its format. I use XSLT in such a way that it is relatively easy to create modularity in our web pages, by using the apply-templates method. Each pair follows the same conventions, so by exploring one XML/XSLT pair it’s easy to see how all are constructed.

I will review only the About XML / XSLT pair. The about.xml page contains a root `<resource/>` element which itself contains `<RDF/>` and `<title/>` child elements. The RDF has its own namespace (encapsulating the use of the DublinCore namespace), whereas the rest of the XML lives in the default namespace. Our decision to use default namespaces is defined later on in the chapter.

The multilingual website contains Italian, English, German and French versions of the text. To show variety I’ve chosen an ISO-8859-1 character encoding, rather then UTF-8 (though you should really strive to use UTF-8):

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
```

Every document in the `/data` uses a standard XML PI, which is relatively addressed to an XSLT stylesheet of the same name.

```xml
<?xml-stylesheet href="about.xsl" type="text/xsl"?>
```

Next comes the root element of the XML document:

```xml
<resource name="about" type="document">
... 
</resource>
```
The `<resource/>` element is reused throughout the `/data`, which explains the presence of a name and type attribute.

Each document contains standard metadata information which may either be generated manually or with an editor. I have chosen the RDF and DublinCore vocabularies as a commonly accepted configuration for capturing information about content editors;

```xml
  <RDF:Description RDF:HREF="http://purl.org/metadata/dublin_core_elements"
    DC:Title="About Page"
    DC:Creator="Jim Fuller"
    DC:Subject="About Page"
    DC:Description="Contains general company details"
    DC:Publisher=""
    DC:Format="text/html"
    DC:Type="Technical Report"
    DC:Language="en"
    DC:Date="05/08/2003"/>
</RDF:RDF>
```

The only textual data within about.xml is contained in the `<title/>` element, which identifies each text with a `<txt/>` block. It is accompanied by an xml:lang attribute:

```xml
<title>
  <txt xml:lang="en">About Page(english)</txt>
  <txt xml:lang="fr">About Page(french)</txt>
  <txt xml:lang="de">About Page(deutsch)</txt>
  <txt xml:lang="it">About Page(italian)</txt>
</title>
```

Each `<txt/>` block defines a different language. Text intended to appear in a specific language will always have that language encapsulated in parenthesis.

The accompanying about.xsl XSLT stylesheet is more complex. It too reflects certain conventions that exist throughout every XSLT page. Every XSLT document inherits the master.xsl stylesheet, which in turn imports the global, and resource data and stylesheets.

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```
All the page documents inherit master.xsl. If I don’t need access to global or resource data, however, it is not necessary. As I discuss later, the master.xsl stylesheet imports other stylesheets and data, which may then be used in the about.xsl document.

The root matching template always initially matches the <resource/> element of the XML page.

```
<xsl:template match="resource">
  <html lang="{$lang}">
    <xsl:comment>
      <xsl:value-of select="$globalexample"/>
      <xsl:value-of select="$tstamp"/>
    </xsl:comment>
    <head>
      <xsl:apply-templates select="$html/asset[@name='metatags']"/>
      <xsl:apply-templates select="$html/asset[@name='pagetitle']"/>
      <xsl:apply-templates select="$css/asset[@name='site_style']"/>
    </head>
  </html>
```

The $lang variable is defined in the master stylesheet, and is used here for the HTML lang attribute. Generally, the $lang variable identifies the current language to the transform. This has an impact on which assets are rendered as well as instructing the transform to output all specific language textual content.

The above usage of xsl:comment illustrates the use of master and global XSL variables within a page stylesheet. In this case, I am outputting the $globalexample and $tstamp variables.

In this way, it’s relatively easy to create modularity in your web pages, using XSLT, by employing the xsl:apply-templates statement matching the resource’s name attribute.

```
<head>
  <xsl:apply-templates select="$html/asset[@name='metatags']"/>
  <xsl:apply-templates select="$html/asset[@name='pagetitle']"/>
  <xsl:apply-templates select="$css/asset[@name='site_style']"/>
</head>
```

This apply-templates method allows content editors to change or amend layout styles without needing to know specific presentation details of an asset.

Admittedly, this technique has its drawbacks from a designer’s perspective, insofar as many of their commonly-used editors will not work directly with XML/XSLT formats. Such compromises are toler-
ated, as I have already conceded that the initial /data distribution will be designed for all users of the system.

As you can see from the above code, the apply-templates method is used many times throughout the HTML <body/> element. Each page has a header and footer, which is another example of using XSLT to componentize HTML.

Notice that each category of asset has its own variable name. With the image asset, the select statement begins with a $img global variable which is defined in the global.xsl stylesheet. The $img variable is just a bit of syntactic sugar; it is mere shorthand for addressing the proper category in the asset.xml document.

I will not review each page of the /data distribution, its more important to gain an understanding of the inherited stylesheets. These contain a good portion of useful logic, so don't be concerned if you remain uncertain as to how the about.xslt and about.xml documents work; the same principles are in place throughout the entire /data distribution.

Building the inheritance hierarchy

The /data distribution uses inherited stylesheets to assist in creating modular presentation code, resources, and in assigning global templates and variables. Recall that each XSLT document imports the following stylesheets.

**master**: Contains current user specific information and imports global and resource stylesheets.

- master.xml
- master.xsl

**global**: Contains global metadata and global variables and templates. global.xsl imports the library XSL templates, which may in turn import EXSLT or FXSL, as well as your own standalone XSLT templates.
asset: Assets are arranged in categories e.g. images, Flash, HTML, CSS, etc. Resource.xml contains asset meta data, and resource.xsl contains presentation templates for each asset category.

This technique could introduce performance problems in addition to forcing an uncomfortable abstraction for developers to learn. Which reminds me of a quote I repeat often to myself:

You can solve any programming problem with an extra level of indirection, except the problem of too many levels of indirection.

— Tom Christiansen, just another Perl hacker and contributor to perl/unix

As the quote above alludes too, building too many layers to an inheritance structure can sometimes be the wrong thing to do.

Master.xsl

The entry point, master.xsl, is a logical place for us to start dissecting how XSLT is used to create the multilingual website.

I can control the output of all transforms by adjusting the xsl:output instruction contained in the master.xsl stylesheet. The xsl:output instruction is set to generate HTML with a character encoding of ISO-8859-1, which ensures that all major European language characters are supported:

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    <xsl:output method="html" encoding="iso-8859-1" indent="yes"/>
The global stylesheet (global.xsl) is brought into a transformation via the use of xsl:include. The global.xml is brought in and referenced via combined use of document() with the xsl:variable. To access global.xml data it would then be necessary to reference the $global variable.
    <xsl:include href="global.xsl"/>
    <xsl:variable name="global" select="document(‘global.xml’)"/>

The asset stylesheet and XML document are brought into the stylesheet using the same methods:
The next few instructions deal with setting xsl parameters. By convention, current variables can be overridden by the current developer, content editor or even end user consuming the /data instead of HTML. These variables control the version rendered in the browser, specifying the language and the client, and defining a timestamp (tstamp) variable used in marking each HTML version with the time published.

```xml
<xsl:param name="lang">en</xsl:param>
<xsl:param name="client">web</xsl:param>
<xsl:param name="tstamp"></xsl:param>
</xsl:stylesheet>
```

The use of xsl:param means that values can be supplied via injecting the xsl:param from a command line, or via script controlled XSLT processing.

By introducing the notion of current global parameters, this allows each user to have their own settings. For example a content editor who only writes in German may opt for his/her language setting to a value of de. This aspect of our isn’t illustrated but if the reader was extending they should always place user specific data within master.xsl and application specific data within the global stylesheets.

Let’s now turn our attentions to the imported global and resource stylesheets and metadata documents.

**Global XML/XSLT Pair**

The global.xml and global.xslt documents contain global variables and templates such as variables that contain the IP address of the server, or include standard templates for processing. One could imagine many publications sharing the same global elements, which is why I provide such an abstraction. There is a clear distinction between a resource, or asset, and global metadata or stylesheet. Assets revolve around typical HTML web elements.

I could use the global.xml document to contain metadata ranging from sitemap information to publisher information, though I leave resources e.g. script blocks, images for asset.xml. In our current version there is no data:

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
```
<?xml-stylesheet href="global.xsl" type="text/xsl"?>
<resource name="global" type="global">
</resource>

I do not use global.xml for our continuous publication process created in this chapter. The file is included as a placeholder for future use, making it more adaptable for the reader’s usage if they want to modify the towards their own ends.

I do however use the global.xsl file. This contains global variables, including the shorthand paths for addressing asset resources and for importing external stylesheets.

<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

The following xsl:variable statements can be accessed by every page in the /data. The $serverIP and $globalexample example variables are just examples, and are not required.

    <xsl:variable name="serverIP">127.0.0.1</xsl:variable>
    <xsl:variable name="globalexample">time published:</xsl:variable>

These shorthand xsl:variable statements make it easier to select assets to be used in a page (as demonstrated in the earlier about.xml document).

<!-- shorthand variables which make it easy to reference assets in templates //-->
<xsl:variable name="css" select="$asset/resource/category[@name='css']"/>
<xsl:variable name="html" select="$asset/resource/category[@name='html']"/>
<xsl:variable name="img" select="$asset/resource/category[@name='img']"/>
<xsl:variable name="jscript"
    select="$asset/resource/category[@name='javascript']"/>
<xsl:variable name="flash" select="$asset/resource/category[@name='flash']"/>

The global stylesheet contains just one global stylesheet which matches <txt/> elements.

<xsl:template match="txt">
    <xsl:choose>
        <xsl:when test="@xml:lang=$lang">
            <xsl:copy-of select="."/>
        </xsl:when>
    </xsl:choose>
</xsl:template>
This global template is critical in ensuring that all language dependent content renders appropriately with respect to the currently defined $lang variable.

All textual content, whether contained in a page of XML or elsewhere, is encapsulated in <txt/> element. I see this by returning to the preceding about.xml example. When the current language is set to de, then only the German <txt/> element is copied to the result document.

The use of xsl:copy within this template means that both elements and textual nodes are copied to the result. This is useful when content editors wish to include tactical markup within a page of XML.

It was easier to mark up text in this manner and have them interoperate with 3rd party tools such as:

- Altova XMLSPY (http://www.xmlspy.com)
- oXygenXML (http://www.oxygenxml.com)
- Xopus (http://www.xopus.com)

Instead of overriding the default text-processing template built into XSLT.

The <txt/> element is not used for simply marking up text nodes, but also provides branching logic on resource elements. This allows us to provide language-dependent markup. For example, the Italian version of the publication can contain <metatags/> that are specifically defined for the Italian language.

The xsl:include instruction is used to import all external third party stylesheets. These could be EXSLT or FXSL libraries, or part of your own home-cooked library.

I have illustrated this, in contact.xsl page template, by using the the EXSLT str:token function from the EXLST string module.

The most complex stylesheet, asset.xsl, revolves around the handling of the website’s assets. This is usually where one would like to build up a component library. Each component can have its own
presentational aspect, and multiple forms of that asset can be rendered when a specific client requests the resource in a specific language.

**Asset.xml**
As with the global xml file, asset.xml is not meant to be processed by XSLT. I have referred to global.xsl in an XML PI, because I may eventually want to create an html list of assets used in the website, a kind of asset register at some point.

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<?xml-stylesheet href="global.xsl" type="text/xsl"?>
```

As is the case with most other XML documents contained in the /data, the root element is a <resource/> element, with associated name and type attributes.

```xml
<resource name="asset" type="asset"/>
```

The metadata is broken down by <category/> elements, containing <asset/> child elements. The first category defines CSS assets.

```xml
<category name="css" resourcepath="css">
  <asset name="site_style" src="../resource/css/basic.css"/>
</category>
```

The resourcepath attribute of the <category/> element contains the name of the directory (in this case, css) underneath the resource directory where the associated files can be found. The encapsulated <asset/> child element defines where the CSS file can be found, relative to the asset.xml document, with the @src attribute.

The next category is HTML assets:

```xml
<category name="html" resourcepath="html">
  <asset name="pagetitle">
    <txt xml:lang="en">
      <title>Multilingual Website (english)</title>
    </txt>
    <txt xml:lang="fr">
      <title>Multilingual Website (french)</title>
    </txt>
  </asset>
</category>
```
This is a clean way to modularize code, as I’ve already shown from the preceding about.xsl example. Each asset takes advantage of its ability to define language variants using the <txt/> element with xml:lang attribute.

<asset name="header">
  <txt xml:lang="en">
  </txt>
  <txt xml:lang="fr">
  </txt>
  <txt xml:lang="de">
  </txt>
  <txt xml:lang="it">
  </txt>
</asset>
<asset name="footer">
  <txt xml:lang="en">
  </txt>
  <txt xml:lang="fr">
  </txt>
  <txt xml:lang="de">
  </txt>
  <txt xml:lang="it">
  </txt>
</asset>
Language dependent metatags inform the XSLT processor how to handle the content. It is possible to further modularize markup, enabling it to handle multiple clients, by using a technique similar to the one I use for languages.

A developer could publish text, browser specific content, or information in the format of Docbook, VoiceXML, and so on.

<asset name="copyright"/>
The final HTML asset is a snippet containing the copyright statement. The copyright asset has no xml:lang attribute which is the convention I use to indicate when I want the asset to be used for all language variants.

```
<category name="javascript" resourcepath="javascript">
  <asset name="js_sniffer" src="../resource/javascript/sniffer.js">
  </asset>
  <asset name="js_printversion">
    document.writeln('&lt;h3&gt;Example Javascript usage&lt;/h3&gt;');
    if (is_win) document.write('Operating System: Windows');
    else if (is_mac) document.write('Operating System: Mac');
    document.writeln('&lt;br/&gt;');
    if (is_ie) document.write('Browser:IE');
    else if (is_nav) document.write('Browser: Navigator');
  </asset>
</category>
```

I can also include JavaScript as an asset; in the above js_sniffer asset is a well-known script that gets processed on the client side, informing the webserver of exactly what browser client is being used.

The second JavaScript asset, js_printversion, illustrates how to include JavaScript scripting directly within the <asset/>element. Note that the src attribute has been omitted in order to denote the difference between external JavaScript file and embedded JavaScript.
Mixing any scripting language inside of XSLT poses its own problems, and you may find yourself needing to perform one of the following to make sure that the JavaScript works properly:

- Replace special XML characters (&, <, >) with their entity representation
- Use disable-output-escaping (d-o-e) XSLT feature
- Wrap the entire JavaScript code block within a CDATA section.

There is an image category element, which (unsurprisingly) contains image asset and related metadata.

```xml
<category name="image" resourcepath="image">
  <asset name="testimage" src="../resource/image/content.gif">
    <width>400</width>
    <height>400</height>
    <alt>content architecture</alt>
    <border>1</border>
  </asset>
</category>
```

The last category contains a Flash category, with one flash asset.

```xml
<category name="flash" resourcepath="flash">
  <asset name="test_flash" src="../resource/flash/test.swf">
    <width>550</width>
    <height>550</height>
  </asset>
</category>
```

Summarizing the assets contained in asset.xml, I list the following components available to place within our pages, using the xsl:apply-templates instruction:

- **HTML**: pagetitle, header, footer, metatags, copyright
- **JAVASCRIPT**: js_sniffer, js_printversion
- **IMG**: testimage
- **FLASH**: public_flash
How are these assets rendered when included in a page?

When, for example, a CSS asset is used, how does XSLT know how to render it properly in the page? The solution involves supplying a matching template for each category type. This is contained within the Asset.xsl document.

```xml
<?xml version="1.0" encoding="iso-8859-1"?><xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"><xsl:template match="category[@name='css']/asset">
  <xsl:param name="node" select="/.."><!--"-->
  <link rel="stylesheet" type="text/css">
    <xsl:attribute name="href"><xsl:value-of select="@src"/></xsl:attribute>
  </link>
</xsl:template>
</xsl:stylesheet>
```

The above xsl:template handles the presentation of CSS via the matching statement category[@name='css']/asset.

So if I wish to use CSS in a page, I can use the xsl:apply-templates instruction with the shorthand, noting that I want to select from the css category, so that $css supplies one part of the XPath.

A specific asset is then chosen by filtering on the asset.xml <asset/> name attribute;<xsl:apply-templates select="$css/asset[@name='site_style']"/>. The matching template subsequently takes care of generating the correct HTML <link/> element.

```xml
<xsl:template match="category[@name='img']/asset">
  <img src="{@src}" height="{height}" width="{width}" alt="{alt}"></img>
</xsl:template>
```

Attribute Value Templates (AVT) are used, instead of xsl:attribute, to insert the images relevant metadata. This reduces the number of keystrokes, still confused of how to use AVT…read up in the XSLT 1.0 spec at the W3c; http://www.w3c.org/TR/xslt#attribute-value-templates.

```xml
<xsl:template match="category[@name='html']/asset">
  <xsl:choose>
    <xsl:when test="txt">
```

150
The standard HTML template first checks for the existence of a <txt/> element. If one is present, further template processing is applied. This will allow our global language handling template, imported via the global stylesheet, to apply language dependent processing. If no <txt/> element is present, the XML is just copied into the result.

The JavaScript template checks for the existence of the @src attribute. If the @src exists, then it outputs a <script/> element which supplies the value of the @src as the external JavaScript file. Lack of a @src attribute tells the template to assume that the asset is embedded JavaScript, and just outputs the value of the asset.

Note that I have chosen to use the xsl:value-of instruction instead of xsl:copy-of. Any markup contained within the JavaScript should probably be escaped to make it safe to parse through the XSLT processor.
The more complex Flash template outputs metadata values to the correct elements and attributes. Sometimes, like here, I use xsl:attribute rather then AVT shorthand to make things readable.

You should now have a grasp of how elements are rendered via templates. Essentially there are three levels of templates, summarized in the following:

**Page**: Page templates are useful for outputting tables, laying out global or asset resources and calculating in page logic.

**Global**: Global templates are where general templates used by all your pages are placed. It also includes any third party XSLT libraries or templates.

**Assets**: Assets are arranged in categories such as images, Flash, HTML, CSS and so on. The asset.xml file contains asset metadata, while asset.xsl contains presentation templates for each asset category. Asset templates match to each asset category, ensuring that page XSLT style-sheets are modular and easy to maintain.

Each of the pages that comprise the site illustrates how to use assets or some feature of the /data distribution, as summarized here:

**About**: Illustrates the use of images in the source distribution, as well as the Content Negotiation aspects of serving up language-dependent versions of images
Contact: Illustrates how third party templates are used within a page.
- contact.xml
- contact.xsl

Products: Illustrates how a JavaScript asset can be used in a page.
- products.xml
- products.xsl

Services: Illustrates how a Flash asset can be used in a page.
- services.xml
- services.xsl

News: Shows how to include external data into the source distribution. This has been achieved by using the document() command.
- news.xml
- news.xsl

By editing these files, you define the website and control the layout.

When I reviewed how Content Negotiation works, I was under the impression that if I changed the current language I should see content specific to that language. I would expect a German user to view a German picture in the about.xml example, though this is only true if I provide a properly tagged German version of the picture.

Let’s break down the way in which image asset are rendered into the About page. It might be useful for you to open up this page in your browser so you can see the results of processing, whilst going through this code.

```xml
<asset name="testimage" src="../resource/image/content.gif">
  <width>400</width>
  <height>400</height>
  <alt>content architecture</alt>
  <border>1</border>
</asset>
```

The metadata contains the height and width of the image, alt tag value, and border value. I am not obliged to use any of the metadata. Since this is an image, I want to use the src attribute. Whenever meta data is changed, simply republish the website, a classic example of separating content from presentation.
The tricky part is how to decide how a designer may call an asset into the page stylesheet. Since I have decided on a simple abstraction through applying templates whenever I want an asset to be composited so I invoke an xsl:apply-templates command. Using this technique, the image asset is placed in the about.xsl page layout through use of the xsl:apply-templates instruction;

```xml
<xsl:apply-templates select="$img/asset[@name='testimage']"/>
```

The select attribute is the path to the testimage asset metadata contained in asset.xml. The $img variable is just shorthand for the unwieldy;

```xml
category[@name='img']
```

which is defined in the master.xsl stylesheet. The template that renders the image tag is contained in asset.xsl. This image template is accessible because the asset.xsl was imported via the master.xsl stylesheet.

```xml
<xsl:template match="category[@name='img']/asset">
  <img src="{@src}" height="{height}" width="{width}" alt="{alt}"/>
</xsl:template>
```

Thus the template would essentially match and output the correct `<img/>` tag.

Notice that there is no branching logic for choosing any specific language version of the image. XSLT is responsible for generating the correct language handling textual and markup content whereas Content Negotiation is left to delivering the correct language file version of a resource.

By allowing XSLT to generate all language versions into their static HTML format, I may then apply Content Negotiation to take care of delivering the correct resource to the browser.

In its default setting Apache Content Negotiation requires that the image also be appended with a language code. Looking into the /resource/image/ directory, I see five files:

- content.gif
- content.gif.en
- content.gif.de
- content.gif.fr
- content.gif.it

If the requesting browser wants the French version of the GIF, then a web server would serve up the content.gif.fr file. Let's remind ourselves that the same mechanism applies to all file based resources, images, Flash, CSS etc. If there is a default form of the resource, content.gif, this image
would be served up in lieu of any missing language versions or where the client browser requests it. If a particular language variant for a particular resource does not exist, then Content Negotiation imparts graceful handling of all assets by providing a default image.

This is an important concept in the design of the /data distribution. XML takes care of text and markup, and I just append different language version files. I still refer to the files by their generic name, as the web server will take care of serving up the correct asset. It would quickly become tiresome to need to build in xsl:choose or xsl:if every time I wanted to test for language version and deliver the appropriate asset. Though I do this for handling text / markup content, it is elegantly handled with a single template which matches text(). That way any text meta data can leverage the xml attribute xml:lang to provide language variant text data.

This convention does make developing more complex if you are viewing the /data distribution locally. For a developer or content editor to change the language being used to view the website, the following steps should be taken:

1. View /data from an Apache webserver with Content Negotiation enabled
2. Adjust the lang xsl:param in master.xsl
3. Adjust the language setting in the browser

In practice, Steps 2 and 3 may be automated as part of a larger approach to content management. Instead of using the browser’s XSLT processor, I could use a dynamic server-side processor. It maybe a mistake to build some dependency on external processing conditions into the XSLT templates, so if you desire this behavior I leave it to the reader. There are plenty of other approaches, not least that Michael Kay’s SAXON XSLT Processor can be used with a servlet capable of server side transformation.

Deciding on client or server side transformations will be informed by what type of application you are writing and who will be using it. For this example, I will keep the focus on making our XML / XSLT source compatible in both client- and server-side XSLT processing scenarios.

*Generate distribution with Ant Build*

If the end user has a browser with built in XSLT processor I could just use the /data version of the site which would require end users browsers to have a built in XSLT processor.

Not every browser has the ability to process XSLT, so instead I create a ‘flat’ format for clients who wish to render the website in any event. Even if the day arrives when XSLT processing capability is
built into every browser or client, there are clear advantages to publishing to a flat format such as caching.

There are other good reasons for generating flat HTML, for instance, many search engine scripts do a good job of indexing and searching through large collections of HTML files. Search engines do not however deal well with categorising websites which dynamically generate their web pages....this can sometimes be related to URL construction or a search engines spider prioritizing indexing static content over dynamic data.

As the Internet today is still primarily HTML, it would be best to continue to reuse all the wonderful tools that exist for such websites. In the past, I might have called them static websites...though what's the difference between a website that publishes every 5 minutes...or whenever there is a change to content, to a website that is dynamically generated on the fly? Since I now have a continuous publication process in place, the multilingual website is updated whenever a change occurs in the /data, ensuring that the deployed website is always (or nearly always) up-to-date.

Since I got ahead of ourselves in the previous section, I have already generated a website by running Ant on the build.xml file supplied in the download for this chapter. If all went well I should now see that a /result directory has been created which contains the flat multilingual website.

Even though this depends on many other xml technologies working in tandem, its now time to see how Ant achieves the generation of the multilingual website.

Build Decomposition

This section deconstructs the supplied build.xml file, in order to better understand how the final publication is generated. The first layer is the entry point into the Ant build, as defined by the default attribute of the <project/> element.

```xml
<?xml version="1.0"?>
<project name="Multilingual Website" default="build" basedir="..">
<description>Build target generates a multilingual website from source xml/xsl.</description>
    <property name="jar.saxon" location="/usr/java/saxon7.jar"/>
    <property name="dir.src" location="daa"/>
    <property name="dir.dist" location="result"/>
    ...
<target name="build" depends="clean, create, copy_resources, generate_dist">
    <echo message="success: language versions (/result) built from source (/data)"/>
```
The default build target actually does nothing but output a message; all the work is achieved via the targets specified in the depends attribute, which will be executed in sequence with the build target executing last:

```
clean
create
copy_resources
generate_dist
```

The reader should be familiar with these types of targets.

**clean Target**

The clean target deletes the existing /result directory, which ensures that I am not overlaying the result of XSLT transformations over an existing /result.

```
<target name="clean">
    <delete dir="${dir.dist}"/>
</target>
```

**create Target**

The create target is the clean target’s companion and physically makes the /result top level directory. So at this moment in the build, I have deleted the previous distribution using the clean target and then created an empty /result directory.

```
<target name="create">
    <mkdir dir="${dir.dist}"/>
</target>
```

**copy_resources Target**

The next build step recreates the directory structure of /data under /result and copies across all associated files, e.g., images, JavaScript files and CSS files.

```
<target name="copy_resources">
    <copy todir="${dir.dist}"/>
</target>
```
I exclude copying over the XML and XSLT documents since I want the final /result to just contain flat file content. When the copy target executes, the /result directory now contains a directory structure that emulates /data with /result/website/resource containing all associated files.

In the future, I may want to allow our multilingual website to contain XML files. For now we’ll exclude these files using the <fileset/> task.

**generate_dist Target**

The generate_dist target takes control of XSLT processing by explicitly calling other Ant tasks.

```
<target name="generate_dist">
    <antcall target="generate_language_versions">
        <param name="dir.src_dir_to_process" value="website/navigation"/>
    </antcall>
    <antcall target="generate_language_versions">
        <param name="dir.src_dir_to_process" value="website/company"/>
    </antcall>
</target>
```

Using the <antcall/> task I call another Ant target.

I showed early on in the book how to start refactoring your Ant scripts …and the above is a good candidate for such refactoring.

Perhaps the reader could create a more efficient approach using an Ant macrodef which encapsulates the generate_language_version target’s processing?

The company and navigation directories are the only directories which contain custom page documents. I decided that generate_dist should use the <antcall/> task to process those directories under the /data. This technique permits fine grained control over the process; I may find situations where certain directories just contain HTML, or other XML and XSLT documents that are processed via server-side processing.

The other directories are part of our /data framework:
The generate_language_versions target then processes the supplied directories.

```xml
<target name="generate_language_versions">
  <antcall target="transform_src">
    <param name="lang.current" value="de"/>
    <param name="client.current" value="web"/>
    <param name="dir.current" value="${dir.src_dir_to_process}"/>
  </antcall>
  <antcall target="transform_src">
    <param name="lang.current" value="en"/>
    <param name="client.current" value="web"/>
    <param name="dir.current" value="${dir.src_dir_to_process}"/>
  </antcall>
  <antcall target="transform_src">
    <param name="lang.current" value="fr"/>
    <param name="client.current" value="web"/>
    <param name="dir.current" value="${dir.src_dir_to_process}"/>
  </antcall>
  <antcall target="transform_src">
    <param name="lang.current" value="it"/>
    <param name="client.current" value="web"/>
    <param name="dir.current" value="${dir.src_dir_to_process}"/>
  </antcall>
</target>
```

This target repeatedly calls the transform_src target that performs the XSLT processing.

*Note*—As with the the generate_dist target, there is a good opportunity for simplifying and optimising this code using the macrodef task. Complex Ant usage is aided by reuse…and you will be pleasantly surprised by the performance of macrodef over that of the antcall task.

Each language is iteratively processed, which allows us to add or delete languages later, if desired.

**transform_src Target**
The final layer in the build is the workhorse — the <transform_src/> target. This target invokes Saxon to process directories supplied as parameters, which additionally are passed through for XSLT processing via xsl:param injection.

<target name="transform_src">
   <tstamp prefix="ts"/>

I provide a timestamp which is then used within XSLT transformations.

<java classname="net.sf.saxon.Transform"
       fork="true"
       failonerror="false">
   <arg value="-o"/>
   <arg value="${dir.dist}/${dir.current}"/>
   <arg value="-a"/>
   <arg value="${dir.src}/${dir.current}"/>
   <arg value="lang=${lang.current}"/>
   <arg value="client=${client.current}"/>
   <arg value="tstamp=${ts.TODAY}|${ts.TSTAMP}"/>
   <classpath>
      <pathelement location="${jar.saxon}"/>
      <pathelement path="${java.class.path}"/>
   </classpath>
</java>

The <java/> task invokes the commandline SAXON processor which scans for xml files which have an XSLT PI instruction to perform a transformation to, Saxon is invoked with the –a option, which instructs it to process any XML file it finds with an XML PI in it. The –o option instructs Saxon to send the output to the /result directory, albeit with a .xml file extension.

The XSLT transformation is supplied with three parameters via xsl:params:

- **lang**: This parameter is used to select textual and element markup within XSLT

- **client**: Just as I select certain resources, text and markup based upon language, the same technique could be used for rendering specific versions for specific clients.

- **tstamp**: Every resultant document is stamped with a publishing time.

Using xsl:params is the standard method of parameterizing your stylesheets.
<movetodir="${dir.dist}/${dir.current}"
  <filesedir="${dir.dist}/${dir.current}"/>
  <mapper type="glob" from="*.xml" to="*.html.${lang.current}"/>
</move>

Each time the <transform_src/> target is called, it is processing only one directory’s specific language version. Instead of controlling output from XSLT or from Saxon, I have a utility <move/> task to rename each .xml file contained in the /result to its correct .html extension, including appending the current language code (.en, .de, .fr, or .it). This renaming satisfies our Apache web server default configuration setup.

<replace dir="${dir.dist}/${dir.current}" value=".html">
  <include name="*.html.${lang.current}"/>
  <replacetoken>.xml</replacetoken>
</replace>

Another utility type task is needed to replace embedded URL references to XML files. This occurred wherever there was an href reference used in the website’s frameset or the navigation page’s <a/> href’s.

I could have chosen to find and replace links using XSLT processing techniques perhaps via a third party library (take a look at the included EXSLT str:replace function in the data/website/library directory). I instead choose to take a shortcut and use the Ant <replace/> task to replace all .xml extensions with .html.

This is a benefit of a hybrid approach of generating the distribution files using Ant as the controller and XSLT as the transformer. Instead of extending XSLT to perform everything, Ant slots in nicely as a fallback solution to just about any problem you need to solve.

As I am using Apache’s mod_negotiation module, all the developer or content manager needs to do is to ensure that every link has a .html extension, instead of explicitly generating the .html.en extension. This way, the /data documents can retain it’s .xml extensions with the knowledge that the final /result files will be correctly named so that Apache Content Negotiation works properly.

**A note on Passing parameters in XSLT**

I tend to restrict the usage of passing parameters to just the passing of strings instead of nodesets. If a nodeset is required, I would rather deliver it into the XSLT process as a fully formed XML document via the document() function rather then cram it into a parameter. By the way, there is nothing fundamentally wrong with stuffing XML into xsl:params, its just a convention I personally follow.
The design decision to control XSLT processing by using such parameters is paying off. If I need more parameters I can just add them in either the global.xsl or master.xsl stylesheets, as their default values can be overridden at the point they are transformed.

Be aware that each XSLT processor has its own syntax and method on how to inject a variable and sometimes it may make more sense to embed context and state in the XML instead of passing as XSLT parameter.

Viewing the Result

Having decomposed the build, lets take a final look into the /result/website/navigation directory. I see that all of the /data pages have been transformed into their language-specific HTML pages.

The /result directory, if it hasn’t been so already, should either be defined as a virtual web server directory or should exist under the web root. It’s easy enough to add a new Ant <target/> to take care of deployment to the target server.

One last tip if you are using Apache <directory/> configuration, make certain that the appropriate Apache httpd.conf section for the virtual directory or web root has its indexes option set to use the Multiviews.

As I did with the build, it won’t hurt to go through a few of the website’s individual pages, contrasting and comment as I go.

Main Page

An end user should now be able to view the multilingual website by supplying the appropriate URL (http://127.0.0.1/result if you are running this as a test on a local machine). The website language version that is delivered should reflect the user’s preferred language settings. The navigation frame contains links to all the areas, which just illustrate how certain resources are rendered.

<!DOCTYPE html
   SYSTEM "-//W3C//DTD HTML 4.0 Transitional//EN">
<html lang="en">
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
    <meta name="description" content="(en)">
    <meta name="keyword" content="(en)">
    <meta http-equiv="Pragma" content="no-cache">
    <meta http-equiv="Content-Language" content="en">
</html>
Case Study: Working with Multilingual...

The source for the index page reveals that everything is rendered as intended, correct meta tags are embedded, the timestamp is present as a comment and the <frame/> attributes from the source have been changed to refer to .html files instead of .xml files.

Product Page

The product page demonstrates how JavaScript resources, in the form of both an external file and an embedded script, are rendered in the HTML page. The HTML source (en version) is shown below.

```html
<!DOCTYPE html
SYSTEM "-//W3C//DTD HTML 4.0 Transitional//EN">
<html lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <meta name="description" content="(en)">
  <meta name="keyword" content="(en)">
</head>
```

163
Product Page (english)

```javascript
var is_win, is_mac, is_ie;
document.writeln('<h3>Example Javascript usage</h3>');
if (is_win) document.write('Operating System: Windows');
else if (is_mac) document.write('Operating System: Mac');

document.writeln('<br/>
if (is_ie) document.write('Browser: IE');
else if (is_nav) document.write('Browser: Navigator');
</script>
</script>
</body>
</html>

Sample Company Ltd. [ Copyright 2008, All Rights Reserved ]

<a href="../contact/index.html" target="main">(+44)55555555555 </a>

<a href="mailto:jim.fuller@ruminate.co.uk" target="_new">jim.fuller@ruminate.co.uk </a>
```
Conclusion

The final chapter’s solution showed one way how Ant can be used to control the publishing of multi-lingual websites interacting with XML/XSLT, using the range of recipes provided in earlier chapters.

Please do not take this approach as an attempt at some kind of ‘framework’. It was a ‘one off’ quick solution that did several years ago and whilst I never emulated the approach in any other solution I do know that this specific bit of software has been running for the past four years with no problems.
Appendix

Corrections and Errata
Please send corrections to text, source code or incorrect/missing license attribution to antrecipes@webcomposite.com
Errata will be maintained at http://www.antrecipes.com

Copyrights, License and Disclaimers

James Fuller, jim.fuller@webcomposite.com © 2009, 2010 Copyright for Source Code, Book Text and Layout

Ian Cox, ijcspam@gmail.com © 2009, 2010 Copyright for Design of front and back cover

The source code examples for this book is released under the Creative Commons Attribution 3.0 Unported License. A copy of which can be obtained here.

  http://creativecommons.org/licenses/by/3.0/

Up to date license and disclaimer information can always be obtained from the Ant Recipes for Web Developers website.

  http://www.antrecipes.com

SOFTWARE CONTAINED IN ‘ANT APACHE RECIPE FOR WEB DEVELOPERS’ BOOK OR INCLUDED SOURCE CODE EXAMPLES IS PROVIDED BY THE COPYRIGHT HOLDER AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
3rd party Licenses

For convenience I have included copies of software in the chapter downloads but urge readers to download libraries directly from their creators own websites.

Here are the attribution and websites for each third party library. If I have missed any accreditations or licenses please email jim.fuller@webcomposite.com and I will strive to correct immediately.

ANT-Contrib
http://ant-contrib.sourceforge.net/
Apache License (http://ant-contrib.sourceforge.net/LICENSE.txt)

Schematron / Ant-Schematron
http://www.schematron.com/index.html
Copyright (c) 2000,2001 Rick Jelliffe and Academia Sinica Computing Center, Taiwan
This software is provided ‘as-is’, without any express or implied warranty.
In no event will the authors be held liable for any damages arising from the use of this software.
Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:
1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Docbook
http://docbook.org
http://docbook.sourceforge.net/release/xsl/current/doc/copyright.html

Docbook Framework 1.0
http://velocity.apache.org/docbook/
Copyright © 2006-2007 The Apache Software Foundation.
Licensed under the Apache License, Version 2.0 (the “License”) you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0
Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

**jaxup-1.01.jar**
http://sourceforge.net/projects/jaxup
BSD license

**jing**
http://www.thaiopensource.com/relaxng/jing.html
http://www.thaiopensource.com/relaxng/copying.html
Jing Copying Conditions
Copyright (c) 2001-2003 Thai Open Source Software Center Ltd
All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
* Neither the name of the Thai Open Source Software Center Ltd nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

**Michael Kay XSLT and XQuery processor Saxon**
http://www.saxonica.com
is released under the MOZILLA PUBLIC LICENSE Version 1.0
xmltask-v1.12.jar
http://sourceforge.net/projects/xmltask/
Apache Software License (http://sourceforge.net/softwaremap/trove_list.php?form_cat=296)

bsf
http://www.beanshell.org/
Apache Software License v2.0 (http://www.apache.org/licenses/LICENSE-2.0.txt)

commons-httpclient.jar
Apache License v2.0
http://www.apache.org/licenses/LICENSE-2.0.txt

jtidy
http://jtidy.sourceforge.net/
License can be obtained here
http://jtidy.svn.sourceforge.net/viewvc/jtidy/trunk/jtidy/LICENSE.txt

rhino
http://www.mozilla.org/rhino/
MPL v1.1 http://www.mozilla.org/MPL/

tagsoup
http://home.ccil.org/~cowan/XML/tagsoup/
Apache License v2.0
http://opensource.org/licenses/apache2.0.php

xcluder
http://sourceforge.net/projects/xcluder
is jointly released under the GNU General Public License (GPL), GNU Library or Lesser General Public License (LGPL)

yuicompressor-2.3.5
http://developer.yahoo.com/yui/compressor/
All code specific to YUI Compressor is issued under a BSD license. YUI Compressor extends and implements code from Mozilla's Rhino project. Rhino is issued under the Mozilla Public License (MPL), and MPL applies to the Rhino source and binaries that are distributed with YUI Compressor, including Rhino modifications made by YUI Compressor. YUI Compressor also makes use of and distributes a binary of JArgs; the JArgs BSD license applies to this binary.
Installing Ant

The most complete and up to date set of installation instructions is found within Apache Ant’s own manual

http://ant.apache.org/manual/install.html

General Installation (Mac OSX, Windows, etc)

Download Apache Ant from http://ant.apache.org

Expand the download and set an environment variable ‘ANT_HOME’ to the location where you expanded the download. You will need to also add ‘ANT_HOME/bin’ directory to your path so you can invoke Ant from the terminal (Command console).

Linux (debian, ubuntu, etc)

Typically you can just use “apt-get install ant” to install Apache Ant.

Ant Commandline Options

Ant has a lot of command line options, many of which you can completely ignore when running most of the scripts contained within this book, though here are a few of the more important switches you may find useful.

-`help (-h)`: generates Ant command-line usage by listing all command-line switches

-`projecthelp`: generates description of build file, listing out targets with their descriptions.

-`buildfile ( -f )`: defines a specific Ant build file to execute, other then the build.xml default

Another set of switches concern themselves with how to get properties into Ant.

-`D`: Defines a value for a property to be used in the build.
--propertyfile: defines a specific name-value pair type propertyfile to use in a build

You can also control the amount of information Ant generates during a build (which can be a lot!) using the following.

-quiet: Defines a value for a property to be used in the build.

-verbose: Defines a value for a property to be used in the build.

-debug: Defines a value for a property to be used in the build.

To view Ant's own diagnostic report, use the --diagnostic switch. The following shows the output from my current version. To this I've added comments to further illustrate which section's report contains which text.

The output begins with general version info:

ant -diagnostics

———- Ant diagnostics report ———-
Apache Ant version 1.6beta3 compiled on December 5 2003

Implementation Version (JDK1.2+ only)

core tasks      : 1.6
optional tasks  : 1.6

Next we see all jar files used by Ant. Lots of open source software packages will install Ant on your machine and adjust your classpath without you being aware of it..at last count the author has 8 versions of Ant installed in one form or another on his Mac OSX due to various software packages.

This listing therefore is convenient when you need to determine which version of Ant is being invoked, as well as what jar files it is using.

ANT_HOME/lib jar listing

ant.home: E:\java\apache-ant-1.6beta3\bin..
ant-antlr.jar (5654 bytes)
ant-apache-bsf.jar (12333 bytes)
ant-apache-resolver.jar (4104 bytes)
ant-commons-logging.jar (3857 bytes)
...
ant-weblogic.jar (14449 bytes)
ant-xalan1.jar (3939 bytes)
ant-xalan2.jar (2491 bytes)
ant-xslp.jar (2238 bytes)
ant.jar (952936 bytes)
xercesImpl.jar (949628 bytes)
xml-apis.jar (124724 bytes)

Optional, unsupported tasks are listed next, along with required external classes. This is useful to know when trying to discern the reason why specific optional tasks aren’t working.

Tasks availability

sshexec : Missing dependency com.jcraft.jsch.UserInfo
scp : Missing dependency com.jcraft.jsch.UserInfo
...
stcheckin : Missing dependency com.starbase.starteam.Item
stcheckout : Missing dependency com.starbase.starteam.Item

org.apache.env.Which diagnostics

Not available.
Download it at http://xml.apache.org/commons/

XML Parser information

XML Parser : org.apache.xerces.jaxp.SAXParserImpl
XML Parser Location: E:\java\apache-ant-1.6beta3\lib\xercesImpl.jar

Lastly, a longer list of system properties is printed out to the console.

System properties

java.runtime.name : Java(TM) 2 Runtime Environment, Standard Edition
sun.boot.library.path : E:\j2sdk1.4.2\jre\bin
java.vm.version : 1.4.2-b28
java.vm.vendor : Sun Microsystems Inc.
java.vendor.url : http://java.sun.com/

... ... ...
Don’t forget that Ant defines many environment variables to control how the Java VM processes. For example, to increase the amount of memory used by Ant, you may use the following command:

```
set ANT_OPTS=-Xms500M
set ANT_OPTS=-Xmx500M;
export ANT_OPTS
```

The -Xms switch defines the JVM Ant runs under minimum heap size, with -Xmx defining maximum heap size (Remember: The -X defines these switches as extensions...introduced with Java 1.3).

You can set fixed command-line switches, so you won’t need to add them to the actual command line each time. For example, if you will always want –verbose output…

```
set ANT_ARGS=-verbose
export ANT_ARGS=-verbose
```

In addition, you may explicitly set the JAVACMD if you want to use a different JVM to execute Ant
Recipes Index

Text Recipes
Page 9 - “Learn how to find and replace text”
Page 11 - “Substitute tokens found in text using Ant native filter functionality”
Page 13 - “Create reusable sets of filters that perform token replacement”
Page 14 - “Create sophisticated find and replace pipelines”
Page 17 - “Find and replace text using property expansion”
Page 20 - ”Manage translation of end of line, carriage returns, tabs, and end of file characters between operating systems”
Page 21 - ”fix line ending encodings”
Page 22 - ”compress a set of SQL statements”
Page 22 - ”Output a specific encoding”
Page 23 - “Preprocess source text files so that they are in the correct encoding and format for the current operating system”
Page 23 - “Process a set of text source files so all characters are encoded using Unicode”
Page 25 - “Task orientated unicode conversion”
Page 27 - “Transform a set of source text files into a range of language variants”
Page 32 - “Advanced text processing with Jacson Ant Task”

Database Recipes
Page 35 - ”Manage import of csv delimited data into MySQL”
Page 36 - ”Use the result of a SQL query as properties within your Ant build”
Page 37 - ”Execute SQL statements in Ant”

XML Recipes
Page 41 - "Generate XML directly from your Ant scripts"

Page 42 - "Load external XML into Ant Properties"

Page 45 - "For loop processing in Ant"

Page 46 - "Perform update operations on existing XML documents"

Page 46 - "How to include an XML document into another using Xinclude"

Page 52 - "Use conditions to check what the XML Parser supports"

Page 53 - "Simple XSLT Processing with Ant"

Page 55 - "More sophisticated XSLT processing"

Page 56 - "Generate documentation for your XSLT using XSLTDoc"

Page 59 - "Generate a DTD for Ant markup"

Page 60 - "Validate XML using Ant’s built in facilities"

Page 62 - "Validate using schemavalidate Task"

Page 64 - "Get additional error information on XML errors using xmllint"

Page 64 - "Get additional error information on XML errors using rxp"

Page 64 - "Speed up the resolving of DTD by using catalogs"

Page 66 - "Speed up Xerces Schema validation through grammar caching"

Page 66 - "Validate using Schematron"

Page 67 - "Use Jing and Namespace Routing Language to orchestrate parallel and multiple schema validation"

Page 69 - "Use Jing to perform all major schema validation"

Page 71 - "Use Namespace Routing Language to control binding of schema validation to XML Namespace"

Page 72 - "Use Namespace Routing Language to achieve concurrent schema validation"

Page 74 - "applying an XSLT transformation to an OpenDocument"
Recipes Index

Page 78 - "transforming Docbook to HTML or WordML"
Page 80 - "Create sophisticated documentation using Docbook XML"

Web Recipes

Page 83 - "Check that a site is live and accessible"
Page 84 - "Perform a HTTP Get with Ant"
Page 84 - "Perform a HTTP Post"
Page 85 - "Perform a HTTP Put"
Page 88 - "FTP with Ant"
Page 88 - "Telnet with Ant"
Page 88 - "Email with Ant"
Page 89 - "Securely copy files between remote servers"
Page 89 - "Securely log onto a remote machine and execute a command"
Page 90 - "Making Ant work with proxies"
Page 94 - "Validate xhtml, html, etc…"
Page 95 - "Create HTML pages from components using xinclude"
Page 98 - "Use ispell to spellcheck your html"
Page 98 - "Check that a distribution of files conforms to a desired directory and filestructure"
Page 100 - "Clean up X/HTML with HTML Tidy"
Page 102 - "Clean up X/HTML with JTidy"
Page 102 - "Clean up X/HTML with TagSoup"
Page 103 - "Compress a set of HTML files"
Page 105 - "Optimising css with yui.compressor"
Page 105 - "Compacting JS with yui.compressor"
Page 107 - "Reduce the number of HTTP requests by concatenating JS files"

Page 107 - "Measuring website performance with Apache Benchmark"

Page 110 - ‘Starting and stoping apache webserver’

Page 112 - "Check status of webserver, email and ftp server"

Page 114 - "Develop and deploy a Firefox toolbar"

Page 115 - “Creating a web backup”