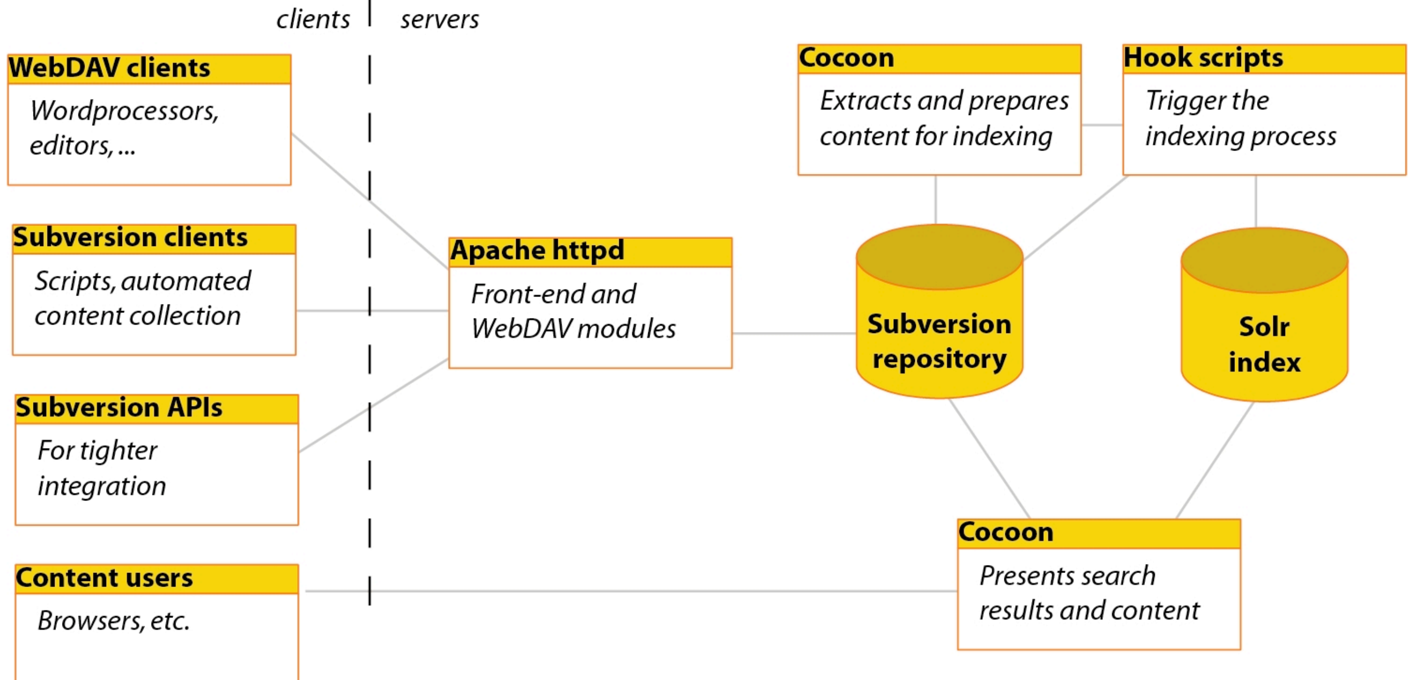
SUBVERSION AND SOLR - YOUR NEXT CONTENT REPOSITORY?

Bertrand Delacretaz, bdelacretaz@codeconsult.ch, www.codeconsult.ch
Presented at the Cocoon GetTogether 2006, www.cocoongt.org
$Id: subversion-solr-bdelacretaz.html,v 1.4 2006/10/02 08:53:53 bdelacretaz Exp $

Overview

This paper presents a content repository based on the Subversion version control system, coupled with Solr, a Lucene-based indexer, using Cocoon-based filtering to prepare content for indexing.



Why Subversion?

Most software developers cannot work without a version control system, even more so in open source projects. Subversion (http://subversion.tigris.org/) and its ancestor CVS are in daily use in the software world. Version control systems reliably store every version of a file or a group of files, and provide detailed revision histories, differences between revisions, "blame" information showing who wrote what, and automatic notifications of changes via email, RSS feeds or other channels.

Version control does not have to be for source code only: all documents of a project, (or of a business) benefit from being traceable, recoverable and reliably stored in a central location.

Besides the cases where a human user creates content, Subversion's many interfaces (command-line, WebDAV, APIs) make it easy to collect content from various sources, live feeds, measurement systems, etc. The inherently multi-user nature of Subversion enables "distributed content collection" scenarios, where many sources, human or automatic, collaborate to create content.

Such scenarios, where content arrives continuously from various sources over many interfaces, are impossible to manage reliably in a file-based system, and can be hard to implement using traditional database systems.

A version control systems based on open protocols handles the various content collection interfaces with ease, helps keep an eye on what's happening, and allows unwanted changes to be rolled back easily when needed.

Subversion, however, cannot currently index the content that it stores, and that's where Solr comes in.

Why Solr?

Solr (incubator.apache.org/solr) is a scalable search server based on the well-known Lucene indexing library.

Running on top of Lucene, Solr provides a simple HTTP/XML interface to manage and query its index.

In addition, Solr provides a powerful caching mechanism for search results, including cache warmup for often-used queries. Simple rsync-based replication features allow several Solr servers to work together to handle high loads.

To index a new document, the values of the fields to index are POSTed to Solr in the form of a simple XML document like this one:

```
<add>
  <doc>
    <field name="id">9885A004</field>
    <field name="name">Canon PowerShot SD500</field>
    <field name="category">camera</field>
    <field name="features">3x optical zoom</field>
    <field name="features">aluminum case</field>
    <field name="weight">6.4</field>
    <field name="price">329.95</field>
  </doc>
</add>
```

This simple interface allows us to use Solr as a loosely-coupled indexing component, by feeding it XML documents representing the indexable part of our content.

A Solr index is based on configurable *field types*, including the definition of *analysis chains* to tokenize, filter and normalize the indexed content using the appropriate language rules.

Solr's analysis features are based on Lucene analyzer components, chained together by configuration. Here's an example configuration where a field is split in words, accented characters are removed, words are lowercased, stop words removed and finally stemming is applied to convert plural words to their singular or "root" forms.

```
<fieldtype name="text_fr" class="Solr.TextField">
  <analyzer>
    <tokenizer class="Solr.StandardTokenizerFactory"/>
    <filter class="Solr.ISOLatin1AccentFilterFactory"/>
    <filter class="Solr.LowerCaseFilterFactory"/>
    <filter
      class="Solr.StopFilterFactory"
      words="french-stopwords.txt"
      ignoreCase="true"/>
    <filter
      class="Solr.SnowballPorterFilterFactory"
      language="French"/>
  </analyzer>
</fieldtype>
```

All this detailed filtering of the content occurs on a field-by-field basis in Solr when indexing, based on such configurations. The field content is extracted from the stored documents by Cocoon pipelines, described below.

For more information on Solr, see the tutorial at http://incubator.apache.org/solr/tutorial.html, or my introductory article on xml.com, http://tinyurl.com/hrkbx.

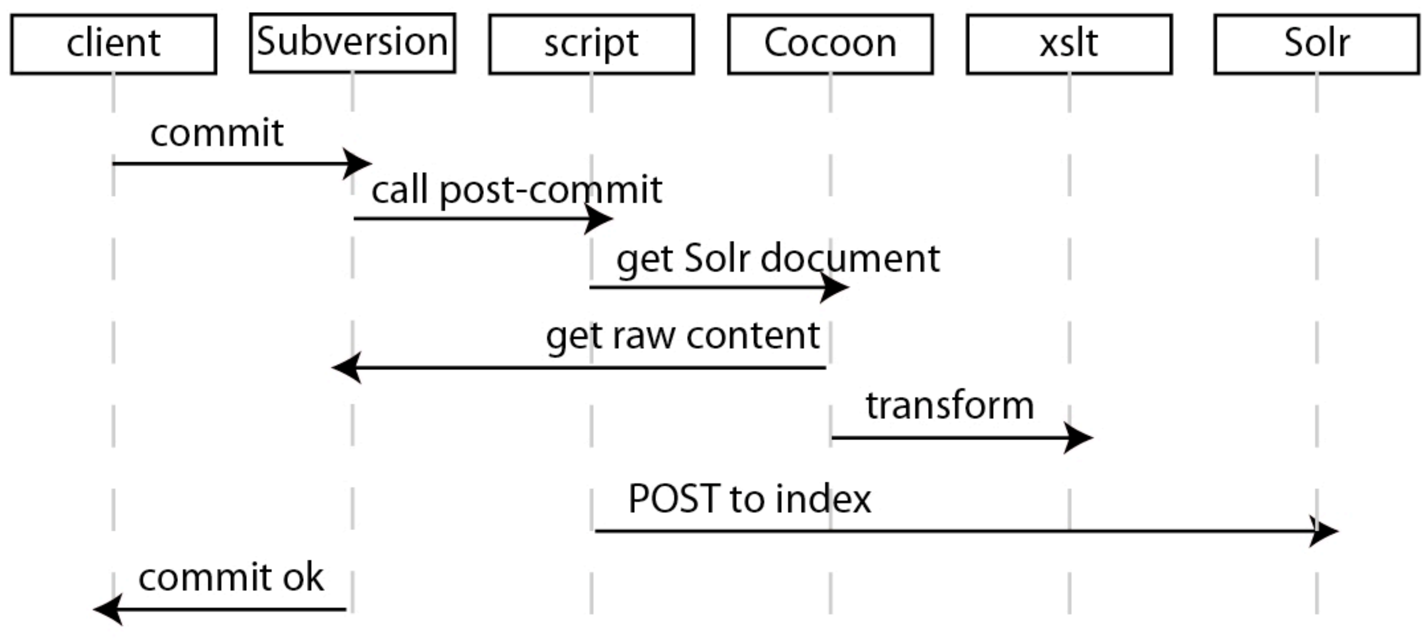
Why Cocoon?

Our content, which comes in various formats and various XML vocabularies, needs to be filtered to extract the *fields* indexed by Solr.

Cocoon provides many ways of extracting XML from various formats, and its powerful *XML processing pipelines* will help us in filtering and normalizing our content on the way to the indexing. If we don't find what we need in the Cocoon toolbox, for example to extract text from exotic binary formats, additional java components or external processes can be called fairly easily from Cocoon.

Although Cocoon includes direct interfaces to Lucene, we will not use them for this prototype, and only use Cocoon to prepare the data that is fed to Solr. The idea is to have a very modular system, and to be able to benefit from Solr's scalability features when needed.

The indexing process



Once a document is committed to the repository, Subversion calls the repository's *post-commit* script, passing it the new revision number. The script is activated simply by storing it in the right place in the (server-side) Subversion repository.

Using the *svnlook* utility, the script finds out which files are affected by the indicated revision. Action codes indicate if the file was added, updated or deleted.

Then, for each added or updated file, the script calls a Cocoon URL with the full filename. Cocoon gets the file from the repository, and based on its sitemap matching mechanisms, selects a filtering pipeline to prepare the content for indexing.

The XML document returned by Cocoon is ready to be fed to Solr; the script only needs to POST it to the Solr *update* URL, and send another POST message to commit the changes.

This simple process allows us to automatically index any document when it is commited, assuming we have created the necessary Cocoon pipelines to extract the indexable content.

Our prototype doesn't handle file deletion yet, but it would be simply a matter of recognizing the Subversion action code and sending a `<delete/>` command to Solr.

Content filtering

In our example, we extract content from OpenOffice documents: starting from an OpenOffice document stored in Subversion, Cocoon creates a "Solr document" ready to be POSTed to Solr for indexing.

Physically, the OpenDocument format used by OpenOffice is a ZIP archive, from which we extract the *content.xml* and *meta.xml* documents to obtain the document content and metadata.

This is done using the jar: protocol in the sitemap, which can extract individual files from a ZIP archive. Note the hack to avoid caching the jar - it is a prototype after all...

```
<!-- extract a single file from an OpenDocument zip archive -->
<map:match pattern="odt-part/**/*">
  <map:generate src="jar:{global:svn-base-url}/{1}?defeat-jar-cache={date:now}!/{2}"/>
  <map:serialize type="xml"/>
</map:match>
```

A simple XSLT transform extracts all fields from the *Dublin Core* and *OpenOffice metadata* categories of the input document, and adds one Solr field for each of those fields in the Solr document.

Metadata and Dublin Core fields, for example, look like this in the OpenDocument XML:

```
<dc:title>My SVN doc test</dc:title>
<dc:description>Testing data and metadata</dc:description>
<meta:initial-creator>Bertrand Delacretaz</meta:initial-creator>
<meta:creation-date>2006-10-01T14:21:17</meta:creation-date>
```

They are handled by a simple generic XSL template, to send all meta fields to Solr for indexing (a similar one is used for the dc: fields):

```
 <xsl:template match="meta:*">
   <xsl:variable name="content" select="normalize-space(.)"/>
   <xsl:if test="$content">
     <field name="meta.{local-name()}">
       <xsl:value-of select="$content"/>
     </field>
   </xsl:if>
 </xsl:template>
```

Resulting in the following XML for Solr:

```
<field name="dc.title">My SVN doc test</field>
<field name="dc.description">Testing data and metadata</field>
<field name="meta.initial-creator">Bertrand Delacretaz</field>
<field name="meta.creation-date">2006-10-01T14:21:17</field>
```

The same XSLT transform removes all markup from the OpenOffice document content, creating a large "content" field to be indexed. Stemming, stopword removal and other field analysis is done entirely by Solr, at this stage we only have to provide clean content.

The resulting document is POSTed to Solr's HTTP interface by the post-commit script, making the new or updated document immediately available in search results.

To index any meta.* and dc.* fields, we use Solr's *dynamic fields* configuration to define a family of fields with the same characteristics:

```
<dynamicField name="dc.*" type="string" indexed="true" stored="true" />
<dynamicField name="meta.*" type="string" indexed="true" stored="true" />
```

WebDAV interface

One of the many goodies that come with Subversion is the WebDAV interface, provided by two Apache http modules, *mod_dav* and *mod_dav_svn*.

With this standard interface, any application can be used to write to the repository or to edit its documents, with access control provided by the standard httpd mechanisms.

However, few or no wordprocessing applications are Subversion-aware today, so they usually access the repository by mounting it on a disk partition. The advantage is that users do not even have to know that there's a version control system in the background, but it also means users cannot decide when to commit their documents and when to save them to a local sandbox, and cannot set commit messages.

When mounted as a disk partition, Subversion's WebDAV interface has to be configured in *autocommit* mode, where a new revision is created every time the document is saved.

Conclusions

Although our prototype repository is in the early stages of development (after only a few hours of work!), the advantages of combining powerful tools instead of using a monolithic system are already apparent. Each component performs very well in its domain, the interfaces between them are simple, many tools are available for each of them and the resulting system is very flexible.

One possible issue is the non-transactional way in which indexing is performed: currently we use the Subversion post-commit script, which cannot cause a commit to fail if the document cannot be indexed. A safer two-phase indexing could be implemented, by using the pre-commit hook script to verify that the document can be converted to an indexable form, and doing the actual indexing in the post-commit hook script, when the commit is confirmed.

We expect to use such a system in a customer project soon, to collect and archives newsfeeds and editorial content asynchronously from many heterogenous sources. Considering our prototype, it looks like we'll just have to write shell scripts to collect the data and XSLT transforms to defined what is indexed. Looking forward to it!