Building Internationalized Web Sites with Zope

This article introduces various approaches to build internationalized Web Sites with Zope, the open source / free application server developped in Python. The problem we discuss in this article goes beyond translating one site to another language. Our goal is to be able to build multilingual sites with Zope, which is both a requirement for community sites in Europe and for corporate information systems in companies where multiple languages are used. The approaches we discuss are based on our experience with the EuroLinux web sites, which have been successfuly localized in more than 12 languages, using various tools such as MLHT, Translator and ZBabel. Current weaknesses of the Zope environnement are highlited and improvements are suggested.

This article was written initially in december 2000. It has been update in december 2001. The examples used to describe existing approaches may have become innacurate.

Jean-Paul Smets-Solanes

Version 0.9.2

TODO: fix XXX, improve specifications, references, acknowledgements, clean-up

0 Introduction

Building a multilingual Web site is seldom taken into account as a default option by standard Web development tools. As a result, building a multilingual Web site can be very tedious, leading to expensive development costs and content management consistency issues. The internationalisation of Web sites has been successfuly achieved for static HTML sites with tools such as MLHT. Tools such as MLHT allow to keep consistency accross the various localised versions of a Web through a single multi-lingual repository and text processing tools. However, the MLHT approach needs to be adapted in the case of dynamic environments such as Zope where localisation has to be done at run time in order to be compatible with the "everything is dynamic" object publication paradigm.

Building an internationalised Web site involves multiple issues. A first issue involves text encoding. A second issue is language selection: the user should be able to select the language which should be used for display. A third issue is user interface: user interface widgets, such as buttons or field names in a form, should be localisable. A fourth issue is translation: the content itself of the Web site should be translated in various languages. A fifth issue is dates and numbers: dates and numbers should be displayed according to national conventions (ex. dates are counted in Japan from the emperor's birthday). Finaly, as the sixth and last issue, the content of certain pages should be adaptable to local constraints. For example, the European Union privacy Law is much stronger that its US counterpart. The European version of a Web site requires various privacy legal statements which are not always required for the US version of the same site. Differences between international

versions may also lead to layout modifications (ex. the order of items in a business letter changes from one country to another).

We are going to review each of these issues in this article, after discussing first the current general approaches to internationalisation of Web sites. We'll then review in detail the implementation of these approaches through an example: the EuroLinux petition for a software patent free Europe.

1 Current Approaches

Internationalising a web site involves answering to basic questions regarding URLs:

1. Should I use a single hierarchy of URLs for all local version or should I use multiple hierarchies of URLs ?
2. Should I put all translated content in a single file or should I span content accross multiple files ?

We'll review these various approaches hereafter.

Multiple files, multiple URLs

The most common approach consists in providing one HTML file per language. All files are stored in the same folder. Suffices are used to determine the language corresponding to each file. This is the approach of the Debian web site (http://www.debian.org), the FFII site (http://www.ffii.org) and the EuroLinux main site (http://www.eurolinux.org). For example, the URL

```
http://www.eurolinux.org/index.fr.html
```

corresponds to the French version of the EuroLinux home page while

```
http://www.eurolinux.org/index.de.html
```

corresponds to the German version. In the rest of the article, we shall call this kind of URL "**national URL**"

Selecting the proper file can be done automatically by Apache which includes a module (XXX to take into account the language preferences set by the user in his or her browser. Fo example, accessing the URL

```
http://www.eurolinux.org/index.html
```

should return the French version (index.frhtml) if the user has set French as his prefered language in his browser and if automatic language selection has been configured in Apache In the rest of the article, we shall call this kind of URL **"generic URL**".

The advantage of this approach is that it is straightforward to implement within Apache However, there are various inconveniences which are related to a decision conflict betwee ease of access and ease of language selection:

1. If the priority is given to ease of language selection, then each page should contain an area pointing to other national URLs. Also, each hyperlink in a page should be rewritten in order to point to national URLs rather than generic URLs. This leads to many URLs spread in the nature for a single content, which is not good for public relations (each local press release has to use a different URL) or weakens the impact of a document in a search engine (less cross-references due to URL multiplication).
2. If priority is given to ease of access, then the default policy should be never to use national URLs. This is a good for search engines (more cross references) and public relations (one message, one URL), but forces the user to change browser preferences in order to view a document in another language and may eventually break certain caches which do not hangle language selection (XXX).

Another issue with this approach is that it may be difficult to keep consistency if content is spread accross multiple files. This is true in particular in the case of HTML files which include a lot of programmatic elements (ex. forms, scripts, etc.) and mix contents and logic (ex. DTML, PHP).

Single repository, multiple URLs

In the case of the EuroLinux main site, the MLHT (http://mlht.ffii.org/) software has been used. This software has been developed originaly by profesionnal translators (A2E) and tested on the FFII web sites in Germany.

With MLHT, a web page is described as an Emacs LISP expression, representing an abstract structured representation of a document (paragraph, title, hyperlink, etc.). Each textual item of this expression is given a unique ID. For example, in the page

`http://www.eurolinux.org/indexfr.html`

the paragraph starting with "Plus généralement dit, nous voulons promouvoir" was givenID 15. Text files (in UTF-8) are generated automaticaly from this single expression and sent to translators in the original language. An example of such a text file can be viewed at the URL

`http://www.eurolinux.org/indexfr.txt`

Translators edit the file, provide a translation for each sentence, and send it back. The MLHT software is then used to generate all HTML pages, include menus, presentations templates URLs are rewriten in order to preserve language consistency accross pages. A language selection area with flags is included in each page.

If we compare this approach to the previous one, it solves the content consistency issue by separating content translation from HTML idiosyncrasis. It has all the advantages, and inconveniencies, of the "ease of language selection" scenario of the previous approach.

Multiple folders, multiple URLs (and eventually multiple servers)

Very few corporate web sites have adressed internationalisation through a unfied approach. I one looks for example at the Mandrake web, URLs for english contents start with the URL

`http://www.linux-mandrake.com/en/`

while French contents starts with the URL

```
http://www.linux-mandrake.com/fr/
```

French and English content are not necessarily consistent, except for the home page Someone accessing the French version of the site is not going to find the same content a someone accessing the English version.

Same stands for SuSE web site, with one additional feature: there are two english web sites one for the US market, one for the rest of the world. Compare for example

```
http://www.suse.de/en/
```

and

```
http://www.suse.com/
```

The Mandrake or SuSE approach represents the mainstream approach for multinational companies. It consists in developing multiple web sites, for each local market, rather tha developing a single site with mutiple languages. Consistency accross those multiple sites i provides through a common visual appearance and a language selection menu on the firs page.

The main advantage of this approach is that it allows to develop web sites which are perfectl suited to each local market. This is very important in particular for the US market or th japanese market, where custommers want to do business with an "American company" or "Japanese company" rather than with a "European company translated in English or i Japanese". Same stands for US or Japanese companies doing business in Europe, to a smalle degree.

The main drawbacks of this approach are: its cost, a lack of content consistency, and UR dissemination. It is simply not suited for small companies or small organisations.

Single repository, Single URL

The approach we are going to develop in section 2 tries to take the best from the approache described above:

1. ease of access: a single URL is used to access a page
2. ease of language selection: changing the language displayed should not require a complex operation from the user in the browser preferences
3. single repository: there should be no content duplication
4. translation repository: all translations should be stored in a central repository to minimise localisation effort
5. local content: it should be possible to develop local content and local layout

2 Internationalising Zope sites

Issue 0: text encoding

For ASCII-speaking people (ex. americans, australians, canadians, englishmen, etc.), text encoding has never been an issue. However, it is still a major issue for the rest of the world (actually more than 90% of human beings). In theory, there is a universal solution for text encoding: UTF-8.

UTF-8 allows to represent with standard character strings contents which requires encoding each letter with 1 byte, 2 bytes or eventually 4 bytes. Ideally, everything human content should be written, stored, searched, transmitted and displayed using UTF-8 since it really brings a universal way of manipulating any text from any language, alone or in combination (ex. a mixture of French accents such as 'éàèî' and of japanese letters such as "XXX"). Modern applications, such as the KOffice suite or the latest versions of Microsoft Office use UTF-8 as the default way of storing text contents. XML, XML-RPC, HTML recommend the use of UTF-8.

However, it is impossible nowadays to use UTF-8 everywhere:

- write: many text editors can not handle UTF-8 (ex. XXX)
- store: many databases can not handle UTF-8 (ex. MySQL, although this will change soon)
- search: many search engines can not handle UTF-8 (ex. Zope ZCatalog)
- transmit: many email programs can not handle UTF-8 (ex. XXX)
- display: many systems do not include nice UTF-8 fonts (ex. Mandrake Linux)

One should also notice that UTF-8 will take a long time before it is accepted in China and Japan. There is for example a FUD (fear, uncertainty doubt) rumour saying that Unicode is not enough to represent all Chinese or Japanese letters. Although this used to be true with the first versions of Unicode, it no longer is. But this rumour and the fact that national standardisation organisation have implemented their own encodings (ex. JIS) creates barriers to the wide adoption of UTF-8.

Considering the current state of technology, a reasonable approach for Zope internationalisation would likely consist in:

- **achieve all data processing within Zope using UTF-8** (there is'nt really any other clean way to do - combining in a single page Chinese ideograms, Japanese ideograms and German accents is for example a requirement in any patent documentation system or in a web site providing translation facilities)
- **store everything in the ZODB or in external databases using UTF-8** and, for compatibility issue with legacy content or legacy tools, provide an optional encoding attribute for legacy stored content
- **convert text encoding to/from UTF-8 whenever it is necessary** (ex. users requesting HTML page to be sent using ISO-8859-15 or JIS): this means including UTF-8 conversion at the ZPublisher level as well as at the level of any external data method (ex. SQL method, SMTP host, etc.) or any search component (ex. ZCatalog)

As we can see in this approach, the text encoding of an HTML page should be if possible a dynamic attribute. For example, in the http://www.osslaw.org web site, some pages are in English, orthers in French and some in Japanese. We use two encoding (ISO-8859-1 and SJIS) because most English or French speakers in the world use ISO-8859-1 fonts and most japanese speakers use JIS fonts. We would have loved to use UTF-8 encoding for rendered HTML

pages (easier and more universal) but it was simply not acceptable for most of our users. Therefore, the encoding attribute is generated by a <dtml-var> in the standard_header. We did the same for the EuroLinux petition site. Here is an excerpt of our standard_html_header:

```
<head>
 <title><dtml-var title></title>
 <dtml-if "REQUEST.has_key('LANG')">
  <dtml-if "LANG == 'cz'">
 <META http-equiv="Content-Type" content="text/html; charset=windows-1250">
  <dtml-elif "LANG == 'eo'">
 <META http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <dtml-else>
 <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  </dtml-if>
 </dtml-if>
 <meta  name="description" content="Freepatents">
 <meta  name="keywords" content="patents, patent, logiciel libre, free
software, patents, brevets">
</head>
```

We think this approach is of course dirty and should not be taken as an example. Ideally, we think the text encoding of a rendered HTML page should be a user option (some people who need to view multilingual content may eventually have good UTF-8 fonts) which default values should depend on the user selected language and which should lead to both a dynamic attribute in the DTML head section and to a dynamic encoding conversion of the page content. However, if high performance is required, default conversion values for each language could be provided in order not to break the http caching model.

Besides the general goal presented above, one should not forget that:

- for full text indexing, words in a text are not split the same way in French (spaces), German (spaces and radicals) or japanese (no spaces).
- for full text indexing, vocabularies may also built in a different way in English (plural forms), French (plural forms, gender), German (plural forms, gender, declination) and japanese (politeness forms, declination suffices).
- many browsers provide very bad encoding information. During a post, it is better to double check which encoding was used for the posted data and do not rely too much on browser provided information.
- both dtml and TAL should probably provide text encoding conversion attributes or expressions in order to deal with exceptions to the "everything UTF-8 rule"

Also, in order to be able to index multilingual pages (a page containing content using multiple languages), tools such as the ZCatalog should be able to discover language either automatically or through some kind of attribute. Word splitting and vocabulary construction heuristics should also be combinable within a single Zope site in that case.

Issue 1: selecting the language

Selecting the language to be displayed is far from trivial.

The current use cases for language selection user interface include the following

1. use the browser to select the language
2. provide buttons (ex. flags) or a pop-up menu to select the language
3. provide a different URL for each language

The approach we discuss here (single repository, single URL) relates to cas 1. and 2. The following technologies can be used for language selection:

1. Browser preferences
2. Use of cookies
3. Use of variables within the REQUEST namespace
4. Use of variables in the URL (ex. ?LANG=en at the end of the URL, also this somehow breaks the single repository / single URL principle)

We have tried the four approaches for the EuroLinux petition site for a software patent free Europe (http://petition.eurolinux.org). Selection through browser preferences and cookies was managed by a patched version of Translator (the ancestor of Localizer). We found through the 90.000+ signatures of the petition that many people in Europe simply refuse cookies and that there are many more users of browsers such as lynx than we would have thought. It should be noticed that recent European Laws recognise refusal of cookies as a right and that web sites should always provide both an alternative to cookies and an explanation of the use of cookies.

We also considered that language selection through browser preferences is not sufficient since most users do not even know this feature.

So, we patched Translator and forced the initialisation of cookies or of a variable through what is probably one of the most cryptic DTML script ever written:

```
<dtml-if "REQUEST.has_key('NO_COOKIE')">
 <dtml-if "not REQUEST.cookies.has_key('LANG')">
  <p> </p>
 <dtml-else>
  <dtml-call "RESPONSE.redirect(URL)">
 </dtml-if>
<dtml-elif "not REQUEST.cookies.has_key('LANG')">
 <dtml-if "not REQUEST.has_key('LANG')">
  <dtml-call "RESPONSE.setCookie('LANG','en',path='/')">
  <dtml-if "REQUEST.has_key('key')">
   <dtml-call "RESPONSE.redirect(URL + '?LANG=en&key=' + REQUEST['key'])">
  <dtml-elif "REQUEST.has_key('oid')">
   <dtml-call "RESPONSE.redirect(URL + '?LANG=en&oid=' + REQUEST['oid'])">
  <dtml-else>
   <dtml-call "RESPONSE.redirect(URL + '?LANG=en')">
  </dtml-if>
 <dtml-else>
  <dtml-call "RESPONSE.setCookie('LANG',LANG,path='/')">
  <dtml-call "RESPONSE.redirect(URL + '?NO_COOKIE=true')">
 </dtml-if>
<dtml-elif "not REQUEST.has_key('LANG')">
 <dtml-if "REQUEST.has_key('key')">
  <dtml-call "RESPONSE.redirect(URL + '?LANG=en&key=' + REQUEST['key'])">
 <dtml-elif "REQUEST.has_key('oid')">
  <dtml-call "RESPONSE.redirect(URL + '?LANG=en&oid=' + REQUEST['oid'])">
 <dtml-else>
  <dtml-call "RESPONSE.redirect(URL + '?LANG=en')">
 </dtml-if>
<dtml-elif "REQUEST.cookies['LANG'] != LANG">
 <dtml-call "RESPONSE.setCookie('LANG',LANG,path='/')">
 <dtml-call "RESPONSE.redirect(URL)">
<dtml-else>
<dtml-let myObjectIds="PARENTS[0].objectIds()">
```

```
<table><tr>
<dtml-if "'en' in myObjectIds">
<td>
<dtml-if "LANG != 'en'">
 <A HREF="<dtml-var URL>?LANG=en"><img src="/flags/en.jpg" border="1"></A>
<dtml-else>
 <img src="/flags/en.jpg" border="0">
</dtml-if>
</td>
</dtml-if>
<dtml-if "'fr' in myObjectIds">
<td>
<dtml-if "LANG != 'fr'">
 <A HREF="<dtml-var URL>?LANG=fr"><img src="/flags/fr.jpg" border="1"></A>
<dtml-else>
 <img src="/flags/fr.jpg" border="0">
</dtml-if>
</td>
</dtml-if>
<dtml-if "'de' in myObjectIds">
<td>
<dtml-if "LANG != 'de'">
 <A HREF="<dtml-var URL>?LANG=de"><img src="/flags/de.jpg" border="1"></A>
<dtml-else>
 <img src="/flags/de.jpg" border="0">
</dtml-if>
</td>
</dtml-if>
<dtml-if "'dk' in myObjectIds">
<td>
<dtml-if "LANG != 'dk'">
 <A HREF="<dtml-var URL>?LANG=dk"><img src="/flags/dk.jpg" border="1"></A>
<dtml-else>
 <img src="flags/dk.jpg" border="0">
</dtml-if>
</td>
</dtml-if>
<dtml-if "'pt' in myObjectIds">
<td>
<dtml-if "LANG != 'pt'">
 <A HREF="<dtml-var URL>?LANG=pt"><img src="/flags/pt.gif" border="1"></A>
<dtml-else>
 <img src="/flags/pt.gif" border="0">
</dtml-if>
</td>
</dtml-if>

<dtml-if "'it' in myObjectIds">
<td>
<dtml-if "LANG != 'it'">
 <A HREF="<dtml-var URL>?LANG=it"><img src="/flags/it.jpg" border="1"></A>
<dtml-else>
 <img src="/flags/it.jpg" border="0">
</dtml-if>
</td>
</dtml-if>
```

```
<dtml-if "'cz' in myObjectIds">
<td>
<dtml-if "LANG != 'cz'">
 <A HREF="<dtml-var URL>?LANG=cz"><img src="/flags/cz.gif" border="1"></A>
<dtml-else>
 <img src="/flags/cz.gif" border="0">
</dtml-if>
</td>
</dtml-if>

<dtml-if "'eo' in myObjectIds">
<td>
<dtml-if "LANG != 'eo'">
 <A HREF="<dtml-var URL>?LANG=eo"><img src="/flags/eo.png" border="1"></A>
<dtml-else>
 <img src="/flags/eo.png" border="0">
</dtml-if>
</td>
</dtml-if>

<dtml-if "'es' in myObjectIds">
<td>
<dtml-if "LANG != 'es'">
 <A HREF="<dtml-var URL>?LANG=es"><img src="/flags/es.png" border="1"></A>
<dtml-else>
 <img src="/flags/es.png" border="0">
</dtml-if>
</td>
</dtml-if>

<dtml-if "'fi' in myObjectIds">
<td>
<dtml-if "LANG != 'fi'">
 <A HREF="<dtml-var URL>?LANG=fi"><img src="/flags/fi.png" border="1"></A>
<dtml-else>
 <img src="/flags/fi.png" border="0">
</dtml-if>
</td>
</dtml-if>

<dtml-if "'no' in myObjectIds">
<td>
<dtml-if "LANG != 'no'">
 <A HREF="<dtml-var URL>?LANG=no"><img src="/flags/no.png" border="1"></A>
<dtml-else>
 <img src="/flags/no.png" border="0">
</dtml-if>
</td>
</dtml-if>

</tr></table>
</dtml-let>
</dtml-if>
```

What we have learned through this script (which is also used to generated a box of flags) is

that:

- language selection is far from simple in the real world (this script evolved during 3 months before we stopped receiving too many user complains)
- flags are nice but are not the best solution to select a language whenever the number of languages grows (pop-menu *à la Suse* is probably better ; language initials as on http://www.europa.eu.int/index_es.htm are also not bad)
- there is a need to have a request language variable available all over Zope in order to know what to do for each user request in terms of localisation
- cookies alone are not a reliable solution
- our DTML hack is too dirty to be acceptable

We think that a reasonable approach for language selection in Zope could consist in:

- having a default language value (likely set to the default value of major search engines)
- use the browser preferences to override the default language value
- try to store this variable in a cookie
- provide a way to the user to overwrite browser preferences through a pop-up menu (or flags or a list of language initials) which updates cookies or brings the user to a set of pages which explains how to change browser preferences whenever the user reject cookies. The latter page should also explain why and how cookies are being used.

We also think that it is reasonable in order to be able provide a simple way to register a given page to a given search engine in a given language to provide an optional multiple URLs approach in order to change the default language value and rewrite URLs (just like for virtual hosting in Zope). For example,

```
http://petition.eurolinx.org/fr/index.html
```

should provide the same content as

```
http://petition.eurolinx.org/index.html
```

where the default language is set to French. It remains to be discussed wether such "localise virtual hosts" should be used to completely override the "single URL" default language selection approach or just to provide a default value for situations where HTTP clients to d provide a language preferences. It also remains to be discussed whether language selectio items (pop-up, flags) should bring the user to another "localised virtual host" (ex. http://petition.eurolinx.org/de/index.html), to the "single URL" page (ex. http://petition.eurolinux.org/index.html) or to the language selection explanation. The latter is probably the approach which is the most consistent with the "single URL" paradigm developed in this article. However, we have no certain conclusion now since we do not have enough experience of search engines (although one can check that http://www.google.com/search?q=eurolinux+petition&hl=fr&lr= brings to the petition page which is what we intended). The use of metatags should also be researched in its relation with search engines.

Issue 2: localising the user interface

Localising the user interface of a web page consists in:

- translating the labels in a form (ex. http://petition.eurolinux.org)
- translating the buttons in a form (ex. http://petition.eurolinux.org) which may eventually require to take care of the button name during a test within the processing of the POST
- providing and selecting images (ex. logos, buttons) for each language
- formatting forms for each language (numbers, dates, etc.): this will be discussed in section 4

In order to achieve this, we used the Translator product on the EuroLinux petition site which allowed us to define so-called "localised variables" which are just entries in a dictionnary which values depend on the current selected language. This approach allows to develop user interfaces in a fashion very similar to what happens with resources on MacOS (one of the first OS providing a true localisation infrastructure). This requires to manually replace in an HTML or DTML page each reference to a string or to an image with a <dtml-lvar id> tag which does the lookup of object 'id' (a string or any other Zope object) either in the localisation dictionnary or in subfolders of the current folder named by their language (de, en, fr, ja, etc.).

Translator provides through acquisition a way of sharing localised strings common to a whole site, or common to a single directory.

The EuroLinux petition site is quite populated with Translator products. Translator was easy to install, to start with and set up as multilingual petition. However, we have found after using it for a while that is has the following shortcomings:

- populating a site with Translators everywhere, although quite nice from a namespace acquisition point of view, leads to unmanageable translation situations
- there is not that much difference between the process of translating content (ex. a sentence) and to translate a user interface string (ex. a label)
- it is very inconvenient to import / export data (for example to gettext based tools which are numerous and well designed) and the web interface is not efficient on large sets of localised data

We think that a reasonable redesign of Translator should

- use only one object for a whole site
- provide a way to manage translations independently from the team designing the web site
- use po files for import / export of translations rather than XML in order to benefit from the numerous tools that exists to edit a localisation dictionnaries
- implement namespace inheritance in a different way for strings
- use gettext technology since it is the default python technology for string translation
- implement the selection of localised images or files in a different way (subfolder is easy to understand but not very clean from a design point of view)
- integrate within a single Product the content translation issue

Part of the directions mentionned above are implemented in Localizer, the successor of Tranlator.

Issue 3: content translation

Translating content consist for exampe in:

- translating sentences in a complex HTML page
- providing multiple versions of a document (ex. a PDF file) to cover each language

For the EuroLinux petition site, we tried various approaches:

- use of ZBabel to translate documents sentence by sentence
- use of DTML dirty tricks to create localised files (ex. <dtml-call "RESPONSE.redirect('http://petition.eurolinux.org/pr/'+'en'+'/pr4.pdf')">)
- use of Translator to create localised files (ex. <drml-lvar document.html>)
- test of other solutions (ZI18N) to create localised objects

Regarding ZBabel, we found that

- it was hard to install (no longer now)
- it required MySQL (no longer now)
- translators hate to use a Web interface to translate long texts

We really liked many things in the ZBabel model and we even patched it and combined with Translator. The resulting (unreleased, experimental, dirty and now useless) version had the following features:

- use of MD5 to identify sentences
- storage of translated sentences in ZODB folders
- possibilty to click on a given page to translate it, which brings the user to a single form presenting a list of sentences to translate
- possibility to mix languages within a single DTML page

We also experienced many difficulties with the translation of simple HTML pages because users just took a page off the site, translated it and sent itback to us (we then had to clean it, remove the decoration tags and add the required dtml).

What we found about content translation is that:

- translators like to work by email and hate to work on line
- on line translation of sentences may only work if each page (or sentence) contains a button "translate this sentence" bringing directly to an online form which allows either to update a piece of translation of send a file by email to the translator.
- if the translation workflow is not obvious and easy, translators will take a piece of HTML, import it into whatever (incl. Microsoft Word), destroy the layout and send the result back, leaving all the reformating to the webmaster...
- in many cases, providing multiple versions of a single file is enough.

Fron this experience, we think that a content localisation product should be designed in two steps.

In a first step, provide a clean way to make a given object available in multiple languages (ex. a CMF Document, a CMF File, a ZPT object, a standard File). We can currently think of two very different ways:

1. Objects with multiple values
2. Caching of translations

The first approach can is implemented by ZBabel which contains one object for each language, contains default attributes and allows to represent a collection of contents as if it was single content.

The second approach is implemented through a caching mechanism which allows to make the

translation task independent from the task of building a web site. We favour this second approach. Il will be detailed later on.

In a second step, provide a clean way to translate sentences, which involves identifying sentences within a document and translating documents dynamically (with caching of pre-translated objects if necessary). Various approaches can be though of:

- manual segmentation and identifaction of sentences through MD5 (ZBabel) or user-defined identiers (*à la* MLHT)
- automatic segmentation and identification of sentences (would be quite easy on structured text)

The automatic segmentation and identification approach consists in determining automatically in a document which are the basic translatable elements. User-defined identifiers provide a way to identify a sentence even after it has been modified. MD5 identifier provide a simple way of identifying a sentence. Automatic segmentation of a document should be in any case provided as some kind of service before manual resegmentation and identification.

Great care should be put on providing a way to send translation taks to translators. A simple idea could consist in sending an HTML file with a list of one-line three columns tables (identifier, original text on the left, translated text on the right). The example bellow shows an example of translation from French to Ennglish:

BeauLille	Il fait beau à Lille	Weather is great in Lille

This would be fine even for users of Microsoft Word. An alternate approach consists in sending files following the MLHT convention.

In any case, this second step needs much research still. To let this research go on, it is a safe bet for DTML to standardize on a combination of Localize / ZBabel like syntax which parameters are:

- a source language (with default equal to an acquisition default)
- a destination language (with default equal to user language preferences)
- an MD5 automatic identification
- a user-definable identification

and wait for various implementations of the segmentation semantics. It is also a safe bet to extend this kind of process to ZPT and to structured text.

Issue 4: localised dates and numbers

An approach similar to text encoding should be implemented for dates and numbers. This means standardising for all data processing on a common date and number format (possibily ISO) and providing translations at input and output. As a reminder, dates in Japan are counted from the emperor's birthday, which makes conversion far from easy. Counting in Japan leads to grouping numbers 4 by 4. Counting in Japan uses different representations (one with Arab numbers and the other with Chinese numbers). In Europe, counting uses either Arab numbers (the default) of Roman numbers (used a lot in book publication).

Users should be able to choose their favourite representation of dates and numbers (just like on MacOS for example). Each REQUEST environnement should contain a date and number

formatting value, stored either in a cookie or obtained as a default for a given language. DTML should provide a way to format a value according to the user date formatting or user number formatting preferences. TAL should also be extended to provide similar conversions. Regarding conversion of posted data, conversion of strings to any value (defined in an SQL method or attribute type) should use user defined formatting preferences.

If Zope provides through DTML, ZPT and Python with a way to implement any date and / or number formatting issue, there is no need to implement any additional formatting service.

Issue 5: local content

Local content involves two different issues:

1. objects which are different in each language (ex. the layout of a letter)
2. content which only exists for certain countries (ex. software only available outside the US because of software patent issues in the US, privacy issues only compulsory in the EU, etc.)

The first case was adressed at section 4 (content translation). The second case can be solved in part using different files providing a different set of URLs (which makes navigation to non localised pages unavailable). Adressing the specific case of geographical content is probably beyong the scope of this document. The only approach we may suggest here is the use of extended language values such as fr_CH (French in switzerland).

3 Other issues

Localised products

As explained by Paul Everitt in Paris in november 2001, the coding style of DTML in Zope products and the coding style of DTML within Zope should evolve in order to become consistent if not identical. Localising DTML / ZPT code should therefore use the same approach in products or in the ZODB.

Products need to be localised beyond DTML. Current Zope products and the current implementation of Zope include the use of strings about everywhere. Such strings should be localised too. We tend to favour the use of the standard gettext python approach in order to localise Python code. However, this approach is not fully compatible with a multilingual model (different users work in different languages with the ZMI at the same time) because it uses environment variables to determine the target translation language.

If there is a way to determine at any time which REQUEST a Thread or a calculation is associated to, then this would solve this compatibility issue.

If not, the standard gettext approach should be extended in order to pass the REQUEST parameter to the gettext translation service. This would lead to strings like:

```
print _(REQUEST, "Authorisation failed.\n");
```

This approach requires the REQUEST variable to be available each time a string is generated.

Localising Zope

Some strings in Zope are not related to a REQUEST. This includes logging facility for example. Such strings should be localized using the standard gettext approach. Localisation should be at least optionnal. Some users for example do not like localising logs while others require it.

User Interface

The user interface is a very important issue in a localisation system. Adopting the gettext file format in a way or another is a very clever way of leveraging the numerous localisation tools. Our favourite localisation tools are:

- KBabel (a graphic editor of po files)
- KDict (a web database of standard translations)

Other tools are also rapidly emerging (ex. Qt Linguist, GTranslator).

Independence of translation, content and code

The success of the localisation of a project such as KDE should be studdied carefully in order to localise Zope.

Zope shares many similarities with the KDE project:

- core technology (Zope vs. Qt) developped by a rather small company (Zope corp. vs. Troll) acting as dictator on core technology
- applications based on the core technology (Products vs. Applications) developped by a rich community
- mutiple web sites dedicated to specific issues (ex. documentation, application database, localisation)

What probably differs however between the Zope project and the KDE project is the fact that the community site (zope.org vs. kde.org) belongs in this case of Zope to a company while it belongs for KDE to a community.

One of the most interesting part in KDE is the way localisation has been organised (http://i18n.kde.org). What we learn from this organisation is that there should be a complete independence between:

1. people who write code
2. people who write content
3. people who translate content

Code and content independence within Zope are underway. Python Products, skins (and eventually skin-based ZClasses ?) are good examples of code / content independence.

Translation efforts should be even more indepent than code and content. For example, whenever one installs KDE on Debian (and on most distributions)

- applications are provided on one package (ex. kdebase)
- full documentation for one application is provided in another package (ex. kdebase-doc)
- translation for all applications is provided in another package with one package per language (ex. kde-i18n-ja)

Any localisation tool for Zope should also in our opinion follow this strict separation

- Zope Corps. develops / adopts core technology with whatever is needed to provide localisability
- Product developers develop products and are required to use certain core technologies to make their products localisable
- Translators contribute to translations of any product they wish to contribute to. All translations for Zope products are stored in a big file (zope-i18n-ja) while translations for a Zope site are accessed trough user definable backends.

Multilingual cataloguing

Full text indexing of a multilingual Zope site for which translations are provided through external data sources is far from trivial. Possible strategies can be based on:

- having multiple catalogs (one per language) which are updated with newer translation values at the same time as the translation cache.
- having a single catalog in a single language (ex. english) and provide search forms which use vocabulary translation (ex. french words translated into english workds) before searching the single language catalog.
- having a single catalog in a multiple language with multiple vocabularies (one per language) and a language attribute (eventually multi-valued) for each catalogued object. Translations are added to the Catalog through their "virtual localised host" URL in relation with the translation cache.

4 Product chart

Many internationalisation products are available for Zope. Here is a short list (a full list is available here XXX).

As we can see, ZBabel and Localizer (with CMF Localizer) provide a very complete set of features. It is interesting to notice that both ZBabel are migrating slowly to an open backend architecture. ZBabel is has number formatting features which Localizer does not have. Localizer has a very good language selection approach and implements both TAL and DTML tags. Localizer even implements features very close to the "virtual localised host" described above.

5 Proposed Spec

The above chart allows to compare existing products and determine abstract services which could be used in order to refactor the current Zope architecture.

UTF-8 everywhere

Zope should use UTF-8 everywhere internaly (data processing, storage). Using ASCII or combining legacy encodings (ex. JIS, ISO-8859-1) is simply not an option because it does not allow to manipulate multilingual content. Migrating to UTF-8, though a hard step, is a future-proof rewarding step.

Action to be taken: stop using non UTF-8 strings, convert all strings to UTF-8

Dynamic encoding

UTF-8 conversions may be used in two situations:

1. conversion to render an object (as HTML or whatever format). In particular, encoding format should become a parameter of all page templates and of the Zope HTML

rendering machinery
2. conversion to access external data sources (ex. SQL Method)

Vocabulary extraction service / language recognition service

Extracting words from a UTF-8 string should be implemented as a service. Plug-in components should be selected by evaluating the source language (either from an attribute of the document, of the sentence or through some kind of automatic language recognition service).

Single Multilingual Catalog

Catalogs should use the word segmentation service for full text indexing. Objects in Catalogs should be tagged with a multi-valued language tag (a document may contain strings in multiple languages). Catalogs should try to index translated objects each time an object is updated. A default list of translation languages should be provided. Translated objects should be referenced by Catalogs through their "virtual localised host" URL (ex. http://petition.eurolinux.org/ja/index.html). Updating translations should update translated objects in the Catalog.

Localisation pseudo-variables

A common set of localisation variables should be defined:

- the user requested language (REQUEST pseudo-variable calculated through language selections service)
- the content source language (object attribute)
- the user requested date format (REQUEST pseudo-variable through language selections service)
- the user requested number format (REQUEST pseudo-variable through language selections service)
- the user requested page format (REQUEST pseudo-variable through language selections service)

Action to be taken: define the names of such pseudo variables

Localisation preferences service

The localisation preferences service provides a way of calculating localisation pseudo-variables and storing preferences for a given user or for a give request. Such a service should normally built on top of a session service used to store generic user preferences in a cookie, in an SQL database, etc.

Localisation selection service

The localisation selection service provides a way for users to change localisation preferences through user interface pages and widgets.

Localisation Backend service (translation lookup)

The localisation backend service allows to look up a translated content (string,

Backends may use:

- a repository of .mo files in the filesystem (great for packaging Product translations into .rpm or .deb packages - great also to share a common translation vocabulary for user interface)
- a ZODB repository (great for collaborative translation and ease of installation)
- an SQL repository (great for collaborative translation and interoperability with non Zope online tools)

The localisation backend service may eventually implement "intelligent lookup". For example, a file based repository may look-up for translation into multiple folders and implement an acquisition-like approach. Backends should of course implement caching in order to be efficient.

The API of a backend component should look up a translation from:

- Absolute URL (of the object)
- Id name (of an attribute, of a sentence, of a method)
- UTF-8 string value (of a content piece)
- MD5 (of a string, of a file)
- Source language
- Destination language

This API should allow to localize:

- attributes (ex. URL + id of attribute)
- text content stored in an attribute (ex. URL + id of attribute or method)
- binary content (ex. URL alone)
- sentences (ex. URL + id of sentence)
- etc.

It is the responsability of programmers to include the necessary tags and python calls to implement look-up.

Translation UI Service

Provides a user interface to edit translation data. User interface may include:

- import / export
- on-line edition
- email interface

Localisation Cache service

Localisation caches are in charge of

1. Cache pre-translated objects (ex. DTML Document, CMF Document, Files)
2. Lookup translation from time to time and update pre-translated objects
3. Update multilingual catalogs whenever pre-translated objects are updated

Localisation cache allow to provide the same performance as a system like zzLocale while preserving the "everything is dyunamic" model.