Language Fallback Blog Series

Monday, January 20, 2014 1:01 PM

POST 1, Introduction to Language Fallback

For the past few years most of the websites I've built have been integrated with Sitecore. It didn't take very long to run into clients that requested multilingual solutions.

As anyone familiar with Sitecore knows, it can handle this requirement out-of-the-box. Items in Sitecore are created for a default 'en' language and those administrators with the appropriate permission levels can add many more languages. Once these languages have been added to Sitecore, then every item in the content tree can have a version created in that language. It is important to note that although we say 'language' here, it is truly more like a culture, where the language item can be based on a language/region combination, for example, 'en-GB' is English Great Britain. This becomes important if the content differs not just based on language translation, but has a completely different message in different regions of the world.

Viewing the front-end of the site can be done through the context of any of these languages, with the setting of the specific language executed in several ways (in order of priority).

- Set language with sc_lang query string parameter
  - www.yoursite.com/about-us?sc_lang=en-us
  - www.yoursite.com/about-us?sc_lang=de-de
- Embed the language in the url, where Sitecore will add the language between the hostname of the site and the path to the item. For example:
  - www.yoursite.com/en-us/about-us
  - www.yoursite.com/de-de/about-us
  - This is configured with the LinkManager languageEmbedding attribute (setting it to 'always') and Sitecore will automatically create urls with the language embedded and parse urls correctly that have the language in it.
- Programmatically set the context language by setting the special language cookie that Sitecore's code uses.
  - Note that I typically wouldn't do this unless I had a very specific exception for grabbing a value of an item for a particular language and outputting to the page. The reason for this is that if the URL doesn't specify the language that the site is being viewed through, then bookmarking the page would potentially not return the user to the content they were hoping to get back to. This would not be a great user experience. Additionally, this would have a negative impact on SEO.
- Map a url to a specific site node and for that site node specify the language that the site should load in. For example:
  - www.yoursite.com is mapped to en-us (America English)
  - www.yoursite.de is mapped to de-de (Germany German)
- The DefaultLanguage setting in the web.config

Whether to use language embedding in the URL with the same hostname or to have different hostnames per language accomplishes the same thing and ultimately is the preference of the client. Obviously having lots of different hostnames could create a lot of overhead in procuring those URLs. There may be some SEO repercussions that I won't speak to here, but that has to be a part of your strategy.

In addition to setting the context language in the above ways, there is a way to customize how to determine which context language should be used. John West has a great blog post here:
http://www.sitecore.net/community/technical-blogs/john-west-sitecore-blog/posts/2010/11/overriding-sitecores-logic-to-determine-the-context-language.aspx
You could hook in to check browser settings, GeoIP, cookies, settings saved to an authenticated user account, etc, to determine what the preference might be for the current user and then redirect to the appropriate site URL or set the Context Language.

One of my first multilingual Sitecore clients had an old solution in place from a previous development company. They had more than 60 language versions for which they allowed front-end users to change their 'country' by selecting one of these languages from a dropdown. What was super impressive is that they had all of their main content pages versioned into all of these languages. Even if they didn't have translations, they had still created the language version and copied the text from the 'en' version into every language version for each field. As you can imagine, that was an ongoing tedious process for their content authors.

When additional clients began requesting multilingual sites, I recognized that pain point had to be avoided and went searching for solutions that might be out there. Alex Shyba's Partial Language Fallback Module was relatively new, but seemed like the best option.
http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx

In the most basic of explanations, you set whether a language should fall back to another language (at a global level). Then when you create the language versions of an item, the fields are all null initially, and therefore pull their values from the language it falls back to. The biggest benefit, which you can see when you think about the example I gave above, is that the content author simply needs to create these language versions and does not have to worry about copying and pasting values from other language versions.

And so began my adventure with multilingual sites and language fallback. All of this stems from Alex's module. But I've found several ways to enhance it, fix some issues, and make it easier to use. In this series of blog posts I will discuss the following topics:

i. How to Install and Configure Alex Shyba's Partial Language Fallback Module
ii. Enforcing Language Version Presence, Out-Of-The-Box
iii. Enforcing Language Version Presence, Customizations
iv. Using Fallback with the Dictionary
v. Using Fallback with the Advanced Database Crawler
vi. Defaulting Language Versions for Fallback
vii. Language Migration Tool
viii. Language Fallback Report Tool
ix. Additional Miscellaneous Topics for Multilingual and Language Fallback Implementation
    1) Unsharing layout, versioning media, conditional rendering / personalization by language, translator tools, other cultural considerations: dates, currency, registering custom languages, etc

My goal is to give those who are working with multilingual Sitecore websites a single comprehensive place to get all of the necessary information.

Where applicable, I will provide sitecore packages so you can install elements individually and provide a full Visual Studio solution to see how everything comes together.

https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 2, How to Install and Configure Alex Shyba's Partial Language Fallback Module

As I mentioned in my first post in this blog series, the basis of this is Alex Shyba's Partial Language Fallback Module. You can find it in the Sitecore Marketplace:
http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx
Note that there are two different approaches in this location, you want to download Alex's Partial Language Fallback module.

Like other shared source modules, it's smart to keep the source code somewhere so you can switch the references to specific versions of Sitecore.Client, Sitecore.Kernel, etc. That way you can build it for your specific site's Sitecore version and avoid conflicts.

Not only is the code provided in the zip, but you can find the package that needs to be installed in /data/packages/
The package contains the following files:

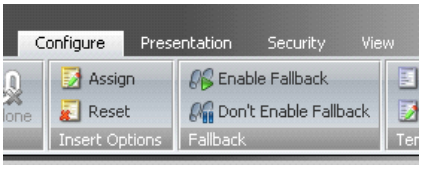
- Sitecore.SharedSource.PartialLanguageFallback.config, which will go into App_Config/Include and contains patches, references and configurations needed for fallback to work. T**he config file in the package is out-of-date. You should copy it from the source code within Sitecore.SharedSource.PartialLanguageFallback\App_Config\Include folder and overwrite the one in your project that the package installed.**
- Sitecore.SharedSource.PartialLanguageFallback.dll, which will go into the bin. Obviously this will be what you replace if you need to recompile the source code for a specific version of Sitecore. **You should initially compile the Sitecore.SharedSource.PartialLanguageFallback project to get the dll to use, since the dll in the package appears to be out-of-date as well (fallback didn't work until I did this).**
    - You can take this opportunity to add a 6.6 folder to the references folder, copy your 6.6 Kernel and Client dlls into there, and then update the references of the partial language fallback project.

The package will install the following sitecore items:

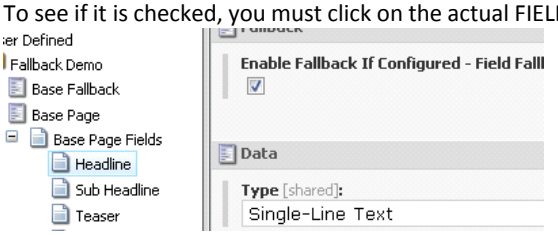
- In core: Do Not Enable Fallback and Enable Fallback ribbon commands for the content editor
- In master: Fallback Language droplink field for the System Language template
- In master: Enable Fallback If Configured checkbox field for the basic Field template

This module is called PARTIAL for a reason. It enables the developer who is setting up the templates to specify on a field by field basis whether it should fallback. Because it could become rather tedious to have to go to every field and select the checkbox for fallback, Alex has provided the Enable and Do Not Enable ribbon buttons.

- These buttons can be found in the Configure tab of the ribbon, in the Fallback section, and you have to select an Item (folder or template) first in order for that section to show up:



- By clicking one of these buttons while on a particular item, the logic will recursively loop down through the children of the item and if is a field, it will check or uncheck the Enable Fallback if Configured checkbox.
- To see if it is checked, you must click on the actual FIELD item



- Don't forget to publish your templates after you enable the fallback checkbox!

Note that in my experience, I haven't found a reason to NOT check this checkbox for all of my templates' fields. So I make sure to periodically click on an item and click the Enable button so that it cascades down through all the templates and fields. That is one of the few drawbacks, not knowing if you may have missed a checkbox. It can lead to some frustration when you don't know why a value isn't falling back, only to find out the field wasn't checked.
WARNING: do not get clever and go to the standard values of the field template and check the checkbox so that all fields fall back. I tried this and had some seriously negative results as apparently lots of system fields were falling back that probably shouldn't have been.

Once you have installed the package into your Sitecore solution, make sure to get the config file from the source code, compile a fresh dll, and copy both into your project overriding what came over with the package, as those are out-of-date.

Open up the Sitecore.SharedSource.PartialLanguageFallback.config file. I will explain here, to the best of my abilities, what the different configurations and settings mean.

```
<databases>
         <!-- Custom GetItemCommand in order to support the "Enforcing of Version Presence"
functionality -->
         <database id="master" singleInstance="true" type="Sitecore.Data.Database, Sitecore.Kernel">
            <Engines.DataEngine.Commands.GetItemPrototype>
               <obj type="Sitecore.SharedSource.PartialLanguageFallback.DataEngine.GetItemCommand,
Sitecore.SharedSource.PartialLanguageFallback" />
            </Engines.DataEngine.Commands.GetItemPrototype>
         </database>
         <database id="web" singleInstance="true" type="Sitecore.Data.Database, Sitecore.Kernel">
            <Engines.DataEngine.Commands.GetItemPrototype>
               <obj type="Sitecore.SharedSource.PartialLanguageFallback.DataEngine.GetItemCommand,
Sitecore.SharedSource.PartialLanguageFallback" />
            </Engines.DataEngine.Commands.GetItemPrototype>
         </database>
      </databases>
```

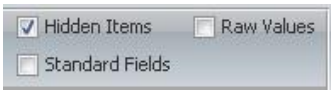
There are two similar entries here, one for each database (master and web) that adds a DataEngine Command. This is executed when the GetItem command is run. The code will make sure items that don't have a language version for the current context language won't be returned. This will only be applied if the enforceVersionPresence attribute of the site is set to true AND the item's template (or base template) is in the Fallback.EnforceVersionPresenceTemplates setting. I will get into this more in my next blog series entry.

WARNING: We may have found some issues with TDS and/or programmatically inserting into Sitecore when these database configurations are included. You might need to considering commenting these out if you see issues.

```
 <pipelines>
         <renderContentEditor>
            <processor
type="Sitecore.Shell.Applications.ContentEditor.Pipelines.RenderContentEditor.RenderSkinedContentEditor,
Sitecore.Client">
               <patch:attribute name="type">
Sitecore.SharedSource.PartialLanguageFallback.Pipelines.RenderContentEditor.CustomRenderSkinedContentEditor
,Sitecore.SharedSource.PartialLanguageFallback</patch:attribute>
            </processor>
         </renderContentEditor>
      </pipelines>
```

This is to customize the look of the content editor so it is more clear that item values are falling back. I've peeked around in this code and it should theoretically display 'fallback' next to a field, similarly to how it displays 'standard value', but my experiences have not shown this to sometimes not work.

I've found that if there are competing modules that want to override the content editor skin or functionality in the content editor, it will cause problems.
NOTE: Selecting the Standard Fields checkbox in the View tab of the ribbon will also mess with this because it is a different content editor renderer.



It will cause the fallback field note to not display and instead give you a generic 'standard value' message.

But at least you can take that to mean it is falling back instead of being overridden.

```
<standardValues>
         <providers>
            <add name="sitecore">
               <patch:attribute name="type">
Sitecore.SharedSource.PartialLanguageFallback.Providers.FallbackLanguageProvider,Sitecore.SharedSource.Part
ialLanguageFallback</patch:attribute>
               <!-- Pipe separated list of the databases that support the fallback
               example: master|web
          -->
               <SupportedDatabases>master|web</SupportedDatabases>
            </add>
         </providers>
      </standardValues>
```

This the key to it all. This is where the Sitecore standard values provider is overridden by the FallbackLanguageProvider, which will take the necessary steps to check for values in the fallback languages. You want to make sure all of your databases are included in here. If you leave out master, it won't show the values falling back when you are in the content editor viewing the 'master' database. And the same goes for when you are in web mode. I don't see a reason to not include a database. Although if you have a CM/CD production environment, you will have to have a slightly different version of the config file that doesn't reference master.

```
<sites>
         <site name="shell">
            <patch:attribute name="enableFallback">true</patch:attribute>
         </site>
         <site name="website">
            <patch:attribute name="enableFallback">true</patch:attribute>
            <!--<patch:attribute name="enforceVersionPresence">true</patch:attribute>-->
         </site>

         <!-- When setting up "enforceVersionPresence" feature for the website above
           make sure to uncomment the following for "publisher" site as well
      -->
         <!--<site name="publisher">
        <patch:attribute name="enforceVersionPresence">true</patch:attribute>
      </site>-->
      </sites>
```

You want to add all of your site nodes in here. You will need to set enableFallback to True if you want to enable language fallback on your site. If you want to enforce version presence, set enforceVersionPresence to true as well. If you don't set enforceVersionPresence to true, then if an item does NOT have a specific language version, it will load the item anyway when the page is viewed in the context of that language (as long as its set to fall back to another with content). There are pros and cons to this which I will discuss in my next blog post.

You MUST leave the 'shell' site set to enableFallback = true. Otherwise fallback won't work correctly in the content editor.

And apparently if using enforceVersionPresence as how Alex has implemented it here, you should also uncomment the publisher site so the enforceVersionPresence = true. I had problems when this was set to true (performance issues when publishing) and left this commented out. I haven't had any negative repercussions by not including it.

```
<!--
            If enabled, the 'enable fallback' checkbox on the field definition item will be looked up
            in the SAME language version as the context language.
            If set to 'true', then when 'en-CA' tries to fallback to 'en', the checkbox field will be
retrieved
            from the field definition item in 'en-CA', thus such version should exist.
            This feature enables you to have different fallback settings for the same field in different
content languages
            If 'false', then the setting will be looked up from Fallback.MasterLanguage ('en' by default)
            Default value: false
         -->
         <setting name="Fallback.VaryFallbackSettingsPerLanguage" value="false" />
```

As the comments explain, you COULD allow whether a field is enabled to fall back to vary on a language by language basis. If you set this to true, it means that you will have to create language versions of all your templates as well. That way, when viewing the site through a particular language context, it will check the same language version of the item's template to see if the field should fallback before retrieving the value from the fallback language.
Personally, I've found no need for this granular level of configuration. It creates a lot more overhead, especially if you have a lot of languages. I prefer to only have a single version of my templates: en, and that's it.

```
 <!--
            Pipe separated list of feild IDs that will be ignored during the fallback process
            Example: {GUID}|{GUID}
            Default value: empty
         -->
         <setting name="Fallback.IgnoredFields" value="{C8F93AFE-BFD4-4E8F-9C61-152559854661}|{4C346442-
E859-4EFD-89B2-44AEDF467D21}|{7EAD6FD6-6CF1-4ACA-AC6B-B200E7BAFE88}|{86FE4F77-4D9A-4EC3-9ED9-263D03BD1965}"
/>
```

As the comments explain, you can add fields that should be ignored during fallback. I'm not entirely sure why this is necessary if you have to enable fields to fallback with the checkbox. I've found no reason to modify this setting and have left it as is.

```
<!--
            Whether to process language fallback for the system fields that start with "__"
            Default value: false
         -->
         <setting name="Fallback.ProcessSystemFields" value="false" />
```

As the comments explain, it will process system fields if set to true. I'm not sure if this means you wouldn't have to check the enable fallback checkbox for those fields or if this is another setting that would be ignoring that checkbox. Either way, I've left this set to false, I haven't found reason to fallback values from these fields.

```
<!--
            Pipe separated list of template GUIDs that support "Enforcing of Version Presence"
functionality
            Default value: empty
         -->
         <setting name="Fallback.EnforceVersionPresenceTemplates" value="" />
```

If you have set that a site should enforceVersionPresence = true, then you should include all templates' guids that should do so here, delimited by a pipe. You can include a Base Template guid as well. So you could create an Enforce Language Version template and set it as a base template for all templates that should enforce a language version and just include that one guid here.
Something you'll need to consider is what templates do you truly want to enforce versioning on. It could be that on all pages and callouts you do, but on all media, you don't. In that case, you wouldn't include any media template guids here.
In my experience, I've found that if I need to enforce versioning, then I might want it to be on all http requests. In a later post I discuss an enhancement that will run a check for version presence during the httpRequest pipeline and where you could tweak that code to remove the check against the EnforceVersionPresenceTemplates setting.

```
<setting name="Fallback.CacheSize" value="10MB" />
```

Anyone familiar with sitecore caches will know that you want it to be able to cache values from the database. Make sure the CacheSize value is big enough to handle your content. Alex clears this cache upon certain actions. I haven't found a reason yet to change the default value.

```
<!--
            Registered Sitecore language to be used for enforcing language fallback, i.e. 'master' language
            It is VERY important NOT to have this language to fallback to any other language on the item
definition

            The value could be different depending on a locale. For US, that would be 'en' or 'en-US' for
example
            If 'enforcing from master language' feature is enabled, this language will be used as a source
language.

            Default value: en
         -->
         <setting name="Fallback.MasterLanguage" value="en" />
```

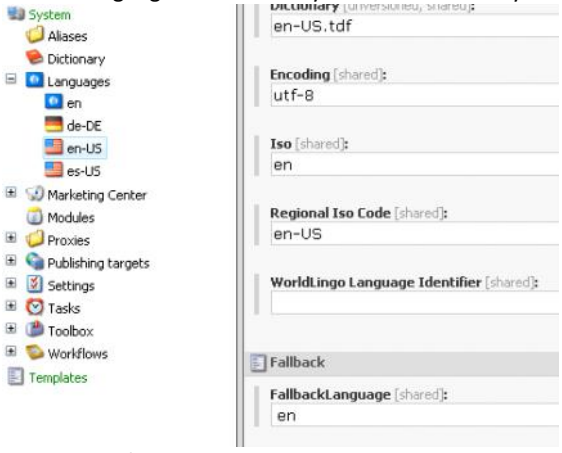
You want this MasterLanguage setting to stay 'en' or whatever the main language that sitecore is viewed through. I think I tried changing this once for a client who wanted to use en-GB as their default language. Ultimately I convinced them to go back to 'en' since we saw some really weird results in the content editor. I can't speak for those in countries where English is not the main language, but I would recommend always having 'en' be your content editor language, the master language here, and the global language that everything falls back to, even if you won't have 'en' be a language that is viewed explicitly from the front-end of the site.

```
<commands>
         <command name="flp:setupfallback"
type="Sitecore.SharedSource.PartialLanguageFallback.Commands.SetupFallbackCommand,Sitecore.SharedSource.Par
tialLanguageFallback" />
      </commands>
```

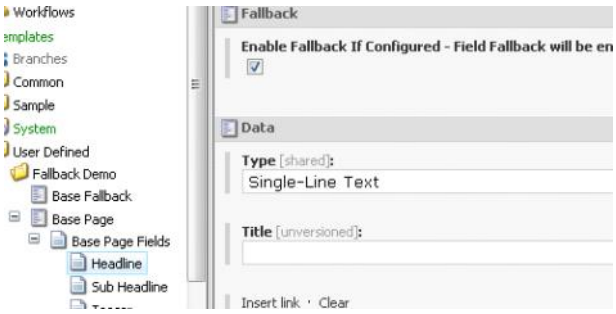
This final configuration is the command that sets up the Enable Fallback and Do Not Enable Fallback ribbon buttons that check and uncheck the field item checkbox.

I have extended the Sitecore.SharedSource.PartialLanguageFallback.config file to add enhancements and fixes and will address each of those in the rest of the blog series.

Once you have this configured, you will want to test that it is working properly.

- Make sure that the site node you are using has enableFallback set to true in the config file
- Add a second language to sitecore if you have not already done so. Set this language to fall back to 'en'
  - 

  - WARNING: If you decide to change which language a particular language falls back to, you may not immediately see a different when you click on a content item that would be falling back. There is caching involved. You might have to go to the /sitecore/admin/cache.aspx and clear your Sitecore cache, and then go back to the content item. It should be show fallback correctly at that point.
- Click on one of your templates and then click the Enable Fallback button in the ribbon
  - Double check the checkbox has been checked by expanding to the field level in the content tree and then click on the field.

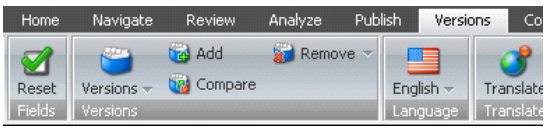


- Validate fallback in content editor
  - Go up to the content tree and add a content item instance of that template, make sure you are doing so in the 'en' language
  - Add some content to the fields
  - Click on the language button in the upper right portion of the content editor and select the new language
  - You will see a prompt that no version exists for the current language

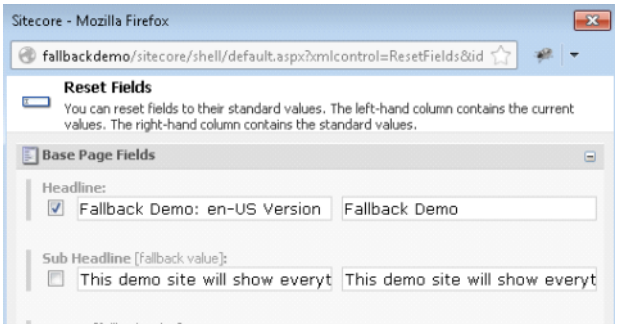
  - Click the link to create the language version
  - It will then display the full content window and the fields should be pre-filled with the values from the original fallback language
  - It should say either 'fallback' or 'standard values' next to the field names

  - Switch back to the original language ('en').
  - Change the value of a field
  - Switch back to the new language, see how the value is automatically updated
  - Override one of the fields on the new language, see how the 'fallback'/'standard value' note is no longer displayed
    - TIP: If you are also viewing 'Standard Fields', it will show 'standard value', if you have 'Standard Field' in the view tab unchecked, it will show 'fallback value'

  - Switch back to the original language and modify that same field
  - Switch back to the new language and see how the field is still the value that was overridden
  - While in the new language, select the Versions tab in the ribbon and then click the 'Reset fields' button



  - Select the checkbox for the field that was overridden and click OK



- See how the field is now falling back again to the original

- NOTE, you MUST reset in order to have the field falling back again. If you simply delete the value out of the field and save, all you are doing is overriding the original with a blank value, which is a valid scenario. You must set it back to null with the Reset button if you want it to fall back again.

- Validate fallback in front-end
  - Publish your templates so that it pushes all fields that have their Enable Fallback checkbox checked
  - Publish the item, but only in the 'en' language, not the new language
  - Go the page in the front-end, might be easiest to test by going to the page using the parameter method for setting language:
    - fallbackdemo/?sc_lang=en
  - And verify your content is there
  - Go the page for a language that is set to fallback to 'en'
    - fallbackdemo/?sc_lang=en-us
    - You'll see that the page loads AND it is falling back (even though you haven't yet published that version to web)
      - This is assuming that enforceLanguagePresence is false, otherwise it would redirect to a 404 page
  - Go the page for a language that is NOT set to fallback to anything and you haven't yet added a version in that language
    - fallbackdemo/?sc_lang=de-de
    - You'll see that the page loads, but with no content
      - This is assuming that enforceLanguagePresence is false, otherwise it would redirect to a 404 page
    - Set the language (de-de) to fallback to 'en' and publish the language
    - Refresh the front end page for the new language, see that the content is now there
  - Change the value in the ORIGINAL language of one of the fields that are falling back (value not set yet in new version)
  - Publish the original version only ('en')
  - Check the front-end page in both the original and new languages (eg: 'en' and 'en-us')
    - You'll see that the update is in both. No need to publish again in the new language, just in the original, as long as you aren't changing values directly in the new language.
  - Override the value in the new language
  - Publish in the new language, it will show up there
    - You only need to publish in the new language if you make specific changes to it, whether it be to reset a field or override a field.
  - Change enforceVersionPresence to 'true' in the Sitecore.SharedSource.PartialLanguageFallback.config site node
  - Remove the version of the item in en-US and publish the 'en-us' version
  - Try going to the page again in the frontend, it will throw an error

At this point, your site is ready to roll with Language Fallback. There are additional steps that must be taken to make sure the Dictionary and Advanced Database Crawler work correctly. There are also some things to consider when deciding if the presence of a language version should be enforced. Additionally, having to add language versions for content just so it can fall back, especially when you start to add more languages to your site, could become tedious. There are some customizations you can add to make this more automated. And what about knowing what items are missing language versions? A simple tool could really help with that. All of this and more will be addressed in the rest of this blog series.

https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 3, Enforcing Language Version Presence, Out-Of-The-Box

To enforce language version or not to enforce language version? That is the question!

As a developer, you may have stumbled upon the issue when first implementing language versions and thought you were doing something wrong. You added an item in your default language, say 'en'. Then you added another language to the site, say 'de-de'. But you have not yet added a version of your homepage in the new language. You then go to your homepage and decide to see what happens when the context language is the new language: www.mysite.com/?sc_lang=de-de. And to your surprise, it loads… with blank content.

What's going on? Well out-of-the-box, Sitecore items do not fallback and Sitecore does not consider missing language versions to mean that the item does not exist. The item does exist, it's just the language version does not. So it loads it up, but can't pull any field values, because the language version isn't there and therefore there are no values.

With Alex Shyba's Partial Language Fallback module, you have a choice.
http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx
In the Sitecore.SharedSource.PartialLanguageFallback.config file, on a site- by-site basis, you can set an enforceVersionPresence site attribute and then specify the templates that should enforce version presence in the Fallback.EnforceVersionPresenceTemplates setting.

Envision the scenario as mentioned above: Your homepage item has a version in 'en' but not yet had the 'de-de' added to it. Assume that the homepage template guid (or one if its base template guids) is added to the EnforceVersionPresenceTemplates setting. You are viewing the site through the context of the 'de-de' language. Your site has enableFallback set to true, and the 'de-de' language is set to fallback to 'en'.

If you set enforceVersionPresence to true, your homepage will be treated as not existing, and will load a 404 page.

If enforceVersionPresence is set to false, then 'en' values of the item will display, even though the 'de-de' version was not added yet to the item (fallback will work without the version presence). Of course if you did add a 'de-de' version and overrode any of the fields, it would show a combination of 'de-de' and 'en', depending on what has been overridden.

So which is preferable? Well that really depends on the strategy of your client and their multilingual needs. Perhaps your client never wants to have a page not exist in the other languages, it should always just fallback unless overridden. They will add language versions when they need to override something, but otherwise, don't want to worry about it. In this case, enforceVersionPresence should be set to false.

But on the other side of the coin, it is quite possible your client won't want certain pages to show up in certain languages. Perhaps there is a section all about America that is not pertinent to those who have chosen to view the site for Germany German. The client needs a way to specify that. Some might say to have another site node for Germany. But what if there are 8 sections full of content on this site, and the America section is just one? Do you really want to duplicate all of those pages across sites and not reuse the content on both sites? And if there is that much content, cloning could become a real headache. In this case, perhaps the client would find it easier to have the language version in the 'en' language only, not create it in the 'de-de' language, and then set enforceVersionPresence to true. Now, whether the language version exists or not is the way the content author can dictate if the page should load in that language.

One area that requires careful consideration is Media. A benefit of being able to specify which templates enforce version presence via the Fallback.EnforceVersionPresenceTemplates setting is that you can enforce version presence on some things and not others. Media may be one location where you don't enforce version presence. If an image or file is being included on a page that is valid for a language, you might not want pieces of that page missing. You also may not want to have to add a language version for every media item, just so it could fall back (that could exponentially increase the size of your media library in the index). Your answer could be to set enableFallback=true, enforceVersionPresence=true, Media.UploadAsVersionableByDefault = true, and then not include your media template guids in EnforceVersionPresenceTemplates **setting**. As stated above, the media items WILL fall back as long as the languages are set to fall back, and then on one-off situations you could add a language version if you need to override the media item for a particular language.

There is however a small problem with Alex's code if you did not want to enforce version presence and you were planning to have chained fall back (es-us -> en-us -> en). The ideal scenario in this case is that you don't have to add a language version for each language, but his method ReadFallbackValue in the FallbackLanguageManager would stop at the first language because it checks Versions.Count. This needs to be changed.

Once I began implementing a site with fall back and enforce language version presence, there was clearly more to consider. I found that some customizations would be helpful and I discuss them and the update to ReadFallbackValue in my next blog post.

https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 4, Enforcing Language Version Presence, Customizations

In my previous post, I discussed the out-of-the-box functionality for enforcing language version presence in Alex Shyba's Partial Language Fallback Module.
http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx

By enabling the enforceVersionPresence setting, content authors are able to dictate whether a page exists on a site in a particular language by adding or removing the language version for that item.

There are a few helpful customizations in this vein that I'd like to discuss in this post.

First, if planning to NOT enforce language version presence on specific types of items, like Media, and you are planning to use chained languages (es-US falls back to en-US which falls back to en), then an update needs to be made within the Sitecore.SharedSource.PartialLanguageFallback project and compiled dll. Within the FallbackLanguageManager.cs there is a method called ReadFallbackValue: it gets the fallback language version for a particular item language and returns the field value from that version. The problem with that is if you don't create that middle language version in a chained scenario, then the check it makes against fallbackItem.Versions.Count, will prevent it from getting the fallback value from the final language in the chain, and will return a null. It is really important that for this scenario it won't be concerned with the language version existing. Therefore, you replace:

```
return fallbackItem != null && fallbackItem.Versions.Count > 0 ? fallbackItem[field.ID] : null;
```

With

```
return fallbackItem != null ? fallbackItem[field.ID] : null;
```

Next, when enforceVersionPresence is not turned on, if a language version does not exist for an item and you load it on the front-end in that language, it will load, but it could be blank, if that language does not fall back or falls back to a language that also does not exist for that item. I felt that this isn't an ideal user experience. Why would you ever want a page to load up blank? And so although enforceVersionPresence may not be on, if fallback is enabled, then I would still want some sort of enforcement to prevent this blank content.

I also felt that the need to enter template guids to check against via the EnforceVersionPresenceTemplates configuration setting could be limiting. I understand why it was necessary in Alex's code, because of where he is running that code. It is run when any kind of item is retrieved and therefore is hit in ways beyond just serving up a page. You don't want it running for many items that help make Sitecore work. But if you hook into the process a little later down the line, the httpRequestBegin pipeline, then you are only dealing with content being served with an HTTP request.

You can use this instead of Alex's Enforce Version logic (by commenting out the GetItemCommand configurations in the Sitecore.SharedSource.PartialLanguageFallback.config), or in addition to it (demo uses both).

We had noticed that the GetItemCommand custom updates that come with Partial Language Fallback potentially caused some issues with TDS and programmatic inserts to Sitecore, so you may need to comment it out anyway based on whether you run into similar issues.

I added a custom processor to the Sitecore.SharedSource.PartialLanguageFallback.config which is patched into the httpRequestBegin pipeline, after the ItemResolver:

```
<processor patch:after="*[@type='Sitecore.Pipelines.HttpRequest.ItemResolver, Sitecore.Kernel']"
type="Verndale.SharedSource.SitecoreProcessors.ItemLanguageValidation, Verndale.SharedSource" />
```

Within the ItemLanguageValidation Process method, if the Sitecore Context item is not null, and it is not the current site's Page Not Found page, and if my custom method LanguageHelper.HasContextLanguage comes back false, then it will redirect the user to the 404 page setup for the current site. The guid of this page is set within the site node in a pageNotFoundGUID setting.

IMPORTANT: Your 404 page should be setup for every language of your site. That language version NEEDS to exist so that it can redirect to it in that language. Otherwise you'll get a 404 error for your 404 page.

The HasContextLanguage method has the guts of the logic for determining whether enforceVersionPresence is explicitly set, or should still be enforced because fallback is enabled and it won't fallback to anything. The logic within it does the following: (note, by returning true, it is telling the ItemLanguageValidation process that the item exists and should be served up.

- If Sitecore.Context.Site is null or the enableFallback property doesn't exist, none of the rest should matter, return true
- If the item has a version in the current Context language, return true
- If the item does not have a version in the current Context Language and enforceVersionPresence is turned on and the item's template or base template guid is in the Fallback.EnforceVersionPresenceTemplates, return false (404)
  - At this step, you could ignore that setting completely and just enforce for everything, if you did want to worry about adding specific template guids.
- If enableFallback is not enabled for the site and there isn't a version in the current Context Language, then it doesn't matter if

enforceVersionPresence is on or not, we don't want a blank version, return false (404)
- If enableFallback is turned on and there is no Context Language version:
  - If the current Context Language doesn't fallback, return false (404)
  - If the current Context Language DOES fallback, then recursively check the fallback language for a language version
    - If it never finds a language version, return false (404), otherwise return true.

Not only will I reference this HasContextLanguage method from the httpRequestBegin pipeline, but I also wrap calls to Children and GetChildren with a method, RemoveItemsWithoutLanguageVersion, which will call the HasContextLanguage method to remove anything that shouldn't be displayed.

Recently I found another scenario that falls outside of just enforcing versioning on language.

My client had multiple regional sites, with 90% of the content and site structure exactly the same. And there was A LOT of content. This made me rule out different site nodes and cloning. Most of the time language versioning with enforceVersionPresence turned on would do the job.

But users could select one of several dozen countries to view the site through, and then within each country at least one or two different language options. Assuming that in most of those cases it truly was a multilingual translation difference (no differences in content or structure by region), there would be no need to add an English, Spanish, and German language version for EACH country, that would be totally unnecessary overhead.

Instead, we could have English, Spanish and German for each major region, and create Country items in sitecore that map to the same exact languages. When the user selects a Country and Language, we save the Country to session and then the language is the same Language version regardless of which country they selected in that region.

BUT there is always an exception. What if there is a page that should only be visible in one of those countries? Now what? Do we have to add a language for that country, and add that language version to EVERY item in the content tree and media library? (if we decided yes, there is a tool to do that! I will discuss that in a later blog post). This is a one-off scenario. My solution was to add a treelist field to a base page template named 'Limit To Countries' which lists all Country items in the system. If this field has no value, the logic ignores it completely. If one or more Country items are selected, then the code uses the value in Session to determine if the requested page is a match on country and if not, will return a 404 page.

I could not hook into this via the httpRequestBegin pipeline, since the Session was not available and that is where the current user's Country is saved. Instead, I added it to the web.config, in two locations:

In <system.webServer><modules>
```
<add type="Verndale.SharedSource.SitecoreProcessors.RegionValidationModule, Verndale.SharedSource"
name="RegionValidationModule" />
```

And in <system.web><httpModules>
```
<add type="Verndale.SharedSource.SitecoreProcessors.RegionValidationModule, Verndale.SharedSource"
name="RegionValidationModule" />
```

This references a class called RegionValidationModule. It implements IHttpModule and the RequestHandler starts after acquiring state. This is almost the same exact logic as the ItemLanguageValidation processor: if the Sitecore Context item is not null, and it is not the current site's Page Not Found page, and if my custom method LanguageHelper.IsValidForCountry comes back false, then it will redirect the user to the 404 page for the current site.

IsValidForCountry will first make sure it can get at the Session variable for Country (making sure session isn't null, etc). It then checks the item's 'Limit to Countries' field. If no countries are selected, then it will return true. We only want to restrict if at least one country is selected. If there are any countries selected, it checks if the current Session country is one of them and if it is NOT, then it will return false, which will result in a 404 page. The RemoveItemsWithoutLanguageVersion that I mentioned above will also use this method to make sure whatever we wrap with it won't include those items not valid for the country.

So to sum up, there is a lot to consider when enforcing language version presence via the Partial Language Fallback Module. The above customizations could help with more complex scenarios. Ultimately, it all comes down to the strategy that is needed for the site you are implementing.

https://github.com/Verndale-Corp/Sitecore-Fallback-Enforce-Version-Customizations
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 5, Using Fallback with the Dictionary

When creating a multilingual site in Sitecore, a common practice is to leverage the out-of-the-box functionality of the Sitecore Dictionary. Every piece of text on the site should be dynamic so that when users switch languages, everything can be translated.

The Sitecore Dictionary can be found in any Sitecore installation at/sitecore/system/dictionary. You can create one or more Sitecore Dictionary Folder items in this location which enables any preferred method of organization. We typically create folders for each letter of the alphabet within the Dictionary Folder: A, B, C, etc.

Each Dictionary Entry item has two fields: Key and Phrase. Key is a Shared Field which means that the field is shared across all language and number versions and is not meant to be different across versions. The Phrase field is a multiline text field and is an Unversioned Field, meaning it can differ across language versions (but not number versions). The idea is that you can create several different language versions of the same item and then change the Phrase value for each language.

In the code, wherever you would have 'static' text (text not otherwise being managed through content items), instead of setting the text directly in the html or code-behind, you would add a Dictionary Entry item and then reference the Key via the Globalization Translate method.

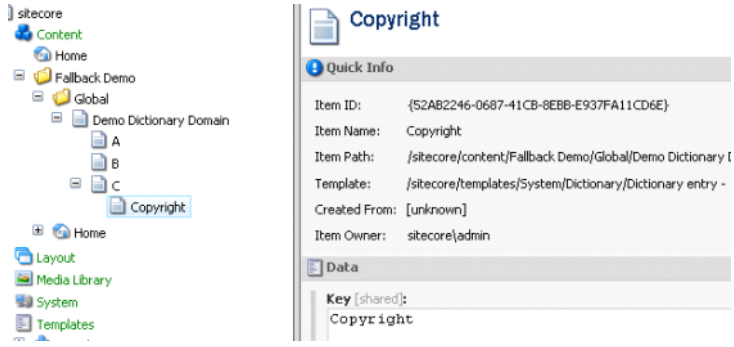
```
Sitecore.Globalization.Translate.Text("Keyword")
```

The naming convention for keywords should be something that makes the most sense for your content authors, since ultimately they will be the ones maintaining this text and translating it. In versions of sitecore before 6.6, all dictionary entries had to be within the /sitecore/system/dictionary folder and ALL sites in your sitecore instance used the same pool of entries. This limitation meant that you would need to prefix all of your keywords for each site. So this would have to be part of your naming strategy. You could set the Key values to be something having to do with where the key is being used, but that would limit reuse of the same phrase across the site. It would have to be in the context of the front-end of the site location, not code location, since the latter obviously would be meaningless to the content author. We typically will use the main words in the phrase being output, eg: "View_All" or "Read_More".

Since Sitecore 6.6, you can now set different Dictionary locations for each site in your Sitecore instance. I HIGHLY recommend doing this. This removes the need for having Site prefixes added to your Dictionary Keys. You would add this to your site node in your Sitecore.SharedSource.PartialLanguageFallback.config file or whatever config file where you prefer adding your site attributes:

```
<patch:attribute name="dictionaryDomain">{GUID}</patch:attribute>
```

You would set {GUID} to the guid of the Dictionary Domain item under which the Dictionary Folders and Entries for the current site live. In Sitecore you might have the following content tree:



Once this is all setup, when developing the site, you simply need to add the dictionary entry for every piece of static text that comes up and then reference the Key with the Translate method. On publish of the dictionary item, it prompts Sitecore to update a Dictionary.dat file that is stored in the temp folder of all of the server specific sites. The front-end will use this dictionary.dat file to pull the values for display.

Note that versions of Sitecore before 6.6 had trouble with CD environments and notifying them that their dictionary.dat file needed to be refreshed. Sitecore should be able to provide a patch that fixes that for earlier versions of Sitecore.

There is, however, a customization that needs to be added to make the Dictionary work with Alex Shyba's Partial Language Fallback Module. http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx

We discovered this need towards the end of development of a site that was implementing both Dictionary and the fallback module. We had updated the Dictionary template fields to have 'Enabled Fallback If Configured' checked true (which is a necessary step for any template that needs to fall back). The language was set to fall back. We did indeed have blank versions of each language created for the Dictionary items. The dictionary item was successfully falling back in the content editor. Yet on the front-end, it was displaying the Key instead of the Phrase when viewing the site through the context of a language that falls back. Sitecore displays the Key when the Key/Phrase is not found in the dictionary.

WARNING: This specific example is compatible with Sitecore 6.6 and above. If you are looking to fix the dictionary for a version before 6.6, you will need to take a look at some of the getTranslation pipeline processors using reflection and use the concepts from what I do below, but make it specific for your version. This is how I came up with this logic, specifically looking at the guts of the TryGetFromCoreDatabase method.

After spending some time looking at the getTranslation pipeline, I found that it was not taking fallback into consideration (which makes sense since fallback is only added in with an extra module). To resolve the issue, I added in an additional processor to the getTranslation pipeline, TryGetFromFallbackLanguage, patching it in after the TryGetFromCoreDatabase processor within the Sitecore.SharedSource.PartialLanguageFallback.config file.

```
<getTranslation>
        <!-- Custom pipeline processor that will get the fallback language of the current language
        and attempt to get the translation of that, as a final step in getTranslation -->
        <processor patch:after="*[@type='Sitecore.Pipelines.GetTranslation.TryGetFromCoreDatabase,
Sitecore.Kernel']" type="Verndale.SharedSource.SitecoreProcessors.TryGetFromFallbackLanguage,
Verndale.SharedSource" />
      </getTranslation>
```

Using other getTranslation steps as examples, TryGetFromFallbackLanguage implements TryGetFromFallbackDomains. It gets the correct dictionary domain and then calls a method called TryTranslateTextByFallbackLanguage and does the following:

- check if the language passed in with the args has fallback assigned.
- if so, then get that fallback language.
- must try to get the translation based on that language.
- Set the following: this.Args.Language = fallbackLanguage.
    - This will allow us to use the TryGetTranslation that was added to the TryGetFromCoreDatabase processor.
        - This is important because the methods it uses to get the actual translation is internal and our code can't use it directly.
    - If TryGetTranslation for the fallback language is successful, great, it will return that result.
    - If it is NOT successful, it will recursively call itself passing the fallbackLanguage to TryTranslateTextByFallbackLanguage.
        - This way, it will keep searching back through fallback to any potential fallback value, no matter how far back in chained languages it has to go.

By adding this pipeline processor to your solution, you will now be ready to use the Sitecore Dictionary and Alex's Partial Language Fallback Module together. Don't forget to update the Dictionary entry template so that it's fields fallback (by checking the Enable fallback checkbox for each field or clicking the Enable Fallback ribbon button while on the Dictionary entry template).

https://github.com/Verndale-Corp/Sitecore-Fallback-Dictionary-Updates
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 6, Using Fallback with the Advanced Database Crawler

In this blog series about Language Fallback, I have been discussing what language fallback is all about, how to install and configure the fallback module, helpful customizations that can be added to it, and required customizations for using it with the Dictionary and other modules.

Two modules that could easily make their way into a Sitecore solution together are Alex Shyba's Partial Language Fallback Module and his Advanced Database Crawler (ADC).

- http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx
- http://marketplace.sitecore.net/en/Modules/Search_Contrib.aspx
    - which essentially points you at some simple documentation: http://sitecorian.github.io/SitecoreSearchContrib/
    - And the actual code: https://github.com/sitecorian/SitecoreSearchContrib

Please note that ADC is unnecessary for versions of sitecore 7 and above and in fact won't work correctly.

I've found both to be integral in the solutions that I have implemented in recent years. As you may know, the ADC leverages indexing, where upon publish, a crawler will grab the values from the fields of the items and add them to the index.

This index can be used to query the data in a faster, more efficient manner compared to constantly hitting the Sitecore database. In areas of the site where you are searching content, or listing content of a specific template, etc, it would make sense to use ADC's GetItems methods, instead of navigating the sitecore content tree. One parameter of the GetItems method is for language and out-of-the-box, ADC will retrieve the correct values of the fields of an item for the language passed in.

When language fallback comes into play though, ADC needs an update. When you add a new version of an item in a different language and leave all of the fields as is, they are null in the database. This is what tells the fallback code to look at the language it falls back to and display those values instead. If the ADC crawler is not updated to check fallback as well, then when the item is indexed by the crawler, it would only

save the values that are overridden and explicitly set for that language version. If you have an 'About Us' page in 'en', create a German 'de-DE' language version, and allow the fields to fallback, the index would not save any values for that page in German. When a search is run on the front end for 'About' while viewing the site through the context for 'de-DE', the About Us page will NOT be returned in the results.

The crawler needs to be updated in ADC so that as it indexes each field, it will check if it falls back, gets the fallback value, and saves that value instead to the index for that language.

Assuming you have downloaded the source code of the Advanced Database Crawler, you will need to modify and add some code.

First, in the AdvancedDatabaseCrawler.cs file, go to the FilteredFields method:

```
protected virtual List<SCField> FilteredFields(Item item)
```

The following line should be commented out:

```
filteredFields.AddRange(item.Fields);
```

And instead it should be replaced with the following:

```
    foreach (SCField field in item.Fields)
                {
                    var currentItem = item;
                    currentItem = CrawlerHelper.GetSitecoreFallbackItem(item, field);
                    if (currentItem.Fields[field.ID] != null)
                        filteredFields.Add(currentItem.Fields[field.ID]);
                }
```

Within a new CrawlerHelper class, we have the method, GetSitecoreFallbackItem. It essentially checks if the field in the language version of the item has a value. If it does, great, return it. Otherwise, use item.GetFallbackItem extension method (extension added via the Fallback module code) to get the fallback language version of the item and pass it into a recursive call to GetSitecoreFallbackItem. This will eventually return a value, if one exists, and that is what will get saved to the index.

Once this code is added to the ADC solution, you can compile and use that dll in your Sitecore solution instead of the original compiled dll that came with the ADC package.

I'd like to acknowledge my amazing co-worker Summit Phadke here. He and I discovered and talked through this issue together and he ultimately came up with and tested the solution for one of our clients.

https://github.com/Verndale-Corp/Sitecore-Fallback-ADC-Updates
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 7, Defaulting Language Versions for Fallback

Once you have decided that using Enforce Language Version Presence is the way to go for your website, you commit your content authors to certain actions for content to show up on the website in the various languages. If the any of the language versions of an item are missing, then within the context of that language, the item will be considered missing.

Alex Shyba's Partial Language Fallback Module (http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx) has cut one tedious aspect out of creating additional language versions. Content authors no longer need to copy and paste content from original versions to new languages if they want the content to be the same across languages. As long as the fields in the templates have their "Enable Fallback If Configured" checkboxes checked, the fallback language value will be displayed in both the content editor and on the front-end of the site.

There is still the potential tedious task of creating these language versions, though. If you have a site with more than two or three languages, it could become time consuming to switch the language icon and click the link to create the language version.

It is with this in mind that I added the next two features.

The first is a new item command. Click on any item and navigate to the Versions tab in the ribbon. In the Language section, there is a new button: "Add All Lang Versions". Clicking this button will run a new method: CreateVersionInAllLanguages. I have to give 'boro' credit for showing me the way in this blog post: http://blog.boro2g.co.uk/create-a-version-of-an-item-in-all-languages-in-the-sitecore-client/

I created a class that implements Command. The Execute method kicks off the Run method which gets the current item that the user is on. It will first prompt the user with a confirm box to make sure they want to create a version for each language. If the answer is 'yes', it will run a method in the LanguageHelper class named CreateVersionInEachLanguage which takes a parameter of Item.

This method gets all of the languages in Sitecore that are not the item's current language and loops through them. It gets the item in each language and checks if it exists by checking the version count. If there are no versions of the item in that language, it will create it.

```
public static void CreateVersionInEachLanguage(Item item)
        {
      IEnumerable<Language> languages = LanguageManager.GetLanguages(item.Database).Where(a => a != 
item.Language);

            foreach (Language language in languages)
            {
                Item localizedItem = item.Database.GetItem(item.ID, language);

                //if Versions.Count == 0 then no entries exist in the given language
                if (localizedItem.Versions.Count == 0)
                {
                    localizedItem.Editing.BeginEdit();
                    localizedItem.Versions.AddVersion();
                    localizedItem.Editing.EndEdit();
                }
            }
        }
```

With this method in place, you need to hook up a command that the user can click within the content editor. First add an entry to the Commands.config file in App_Config

```
  <command name="item:addversiontoalllanguages" type="sharedsource_verndale.
_Classes.SitecoreCommands.CreateVersionInAllLanguages,Verndale.SharedSource" />
```

Then in Sitecore, open the core database and navigate to the following location:
/sitecore/content/Applications/Content Editor/Ribbons/Chunks/Language

Add a Large Button (/sitecore/templates/System/Ribbon/Large Button), named Add All Language Versions, and set the following values:

- Header: Add All Lang Versions
- Icon: Network/32x32/download.png
- Click: item:addversiontoalllanguages
  - (this is referencing the value set in the commands.config)
- Tooltip: Add all language versions to this item.

That's it. When you click on an item and click this button in the content editor, you will find that it adds blank versions of all languages that have not yet been added.

Some clients want to take it to the next level though and ALWAYS have all language versions added when the item is created. They would rather assume every item should have every language version and then have content authors remove versions if necessary.

We already have the logic for this, we can reuse the CreateVersionInEachLanguage in the LanguageHelper class. We just need to add something to the item created event, to make the call. ALSO, we don't want this to run for every item that gets created, only items in certain locations (or maybe certain templates).

This solutions assumes item locations is enough, but you could extend it to check for template and make a decision based on that.

In my Sitecore.SharedSource.PartialLanguageFallback.config file, I added a setting named Fallback.PathsToCheckForLanguageVersions. Multiple paths can be entered, separated by pipe.

```
<setting name="Fallback.PathsToCheckForLanguageVersions" value="/sitecore/content/fallback
demo|/sitecore/media library/files|/sitecore/media library/images" />
```

Then I added a class to be called from the item created event, CreateVersionInAllLanguagesOnCreate.
It checks to see if the Fallback.PathsToCheckForLanguageVersions setting is set. If not, it returns without doing anything. It splits on the '|' character and then loops through the paths in the list. It checks if the current item that has just been created is within that path. If it finds that it was, it will call the CreateVersionInEachLanguage method in the LanguageHelper class.

Finally, add the configuration in the Sitecore.SharedSource.PartialLanguageFallback.config to hook into that event:

```
<events timingLevel="custom">
      <event name="item:created">
        <handler type="Verndale.SharedSource.SitecoreProcessors.CreateVersionInAllLanguagesOnCreate,
Verndale.SharedSource" method="OnItemCreated"/>
      </event>
    </events>
```

Now, whenever an item is created within any of those locations, it will automatically create all of the language versions for you.

These little additions are a nice-to-have for content authors when implementing fallback with enforcement of language version presence. In my next post, I will provide information about a tool that will help when needing to do mass updates to language versions: useful when adding a brand new language to a site, or removing a language from a site, or copying content from one language to another, or just verifying all content is created in a particular language version.

https://github.com/Verndale-Corp/Sitecore-Fallback-Default-Language-Creation
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 8, Language Migration Tool

As I mentioned in my previous post in this Language Fallback blog series, once you have decided that using Enforce Language Version Presence is the way to go for your website, you commit your content authors to certain actions that must be taken for content to show up on the website in the various languages. If any of the language versions of an item are missing, then within the context of that language, the item will be considered missing.

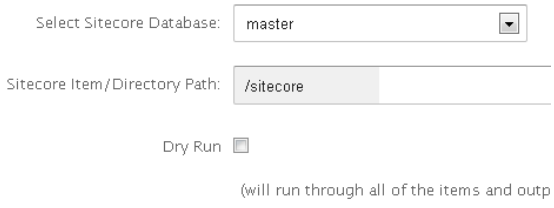
I have already discussed how to add a few customizations so that as new items are added to your sitecore content tree, language versions in all available languages can be automatically added, or added via a ribbon button click.

But there is a scenario where another tool would come in handy. Say the client decides to add a new language to the site. All of the content items already exist and having to click on each one to add the language versions would be tedious at best.

The Language Migration Tool is a great way to quickly add all of the versions of the items in a specific location in the content tree. I got an initial version of this tool from Alex Shyba a couple of years ago. Since then I have added additional functionality to it and my UX team has spruced up the look (it was very engineerish before :)) I'd also like to thank Husain Abbasi for helping me out from a UX perspective!

First, before running this tool in 'non dry run mode', make sure to make a backup of your database, always better to be safe.

Now, in the tool, the first three options are the following:

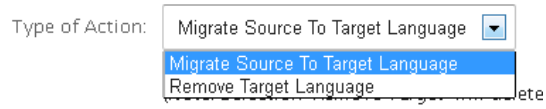


'Select Sitecore Database': this is the database on which the actions should be executed. Master seems like the only logical database, but you do have the options to choose whichever other ones are setup in your system.

'Sitecore Item/Directory Path': this is the path to the root item you want to work with. Depending on additional settings, it will be the single item you are migrating, or the starting point for it and it's children. It defaults to /sitecore for you. You will have to put in the rest of the path.

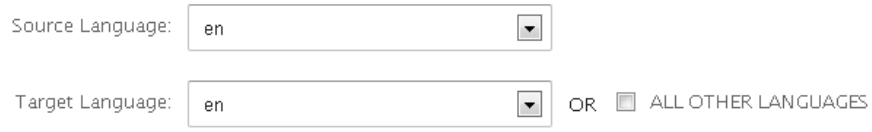
'Dry Run': if this is checked, no actions will executed. It will loop through and output to the log file what it would be doing IF dry run was not turned on.

The next field allows you to specify the action you are taking with this tool, as there are two potential actions.



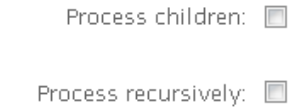
The first action is the typical action you will almost always be using, Migrate Source To Target Language.
The second action, Remove Target Language, gives you a way of removing a language entirely from the specified location.

The next two fields are Source Language and Target Language. These are defaulted to 'en', but the idea is that you would change the Target Language to the one you want the system to be adding to each item.



An additional checkbox has been added for 'All Other Languages'**.** When selecting this checkbox, it will disable the Target Language dropdown, as it is no longer relevant. All available languages will have a version added to the item, if they do not already exist. If it comes to a point where you aren't sure if you might have missed some languages in a section and would like some peace of mind that they are all there, this checkbox is useful.

Next are two checkboxes which will allow you to set if the children of the item specified in the Path textbox should also be processed and if it should be recursively, meaning the children's children, etc.



Obviously this is handy when you are adding a new language to a site. There are a lot of content versions to add in this case and it makes the most sense to add it all at once.

The next checkbox:

is one that I added because the original version of this tool would add a version in the target language regardless of whether it was there or not. So it would add another version, incrementing from 1 to 2, for example. I felt that the main point of this tool is to create missing language versions, there is no need to increment if it is already there.

The following checkbox is critical for solutions using fallback:

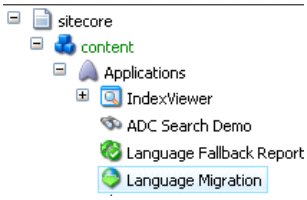
When fallback is enabled, we don't want the values to be copied over into the new language version. We want those values to stay null, so they fall back to the original. This checkbox allows you to specify that you don't want the values to be copied. If you do want the values copied over, you would uncheck this box.

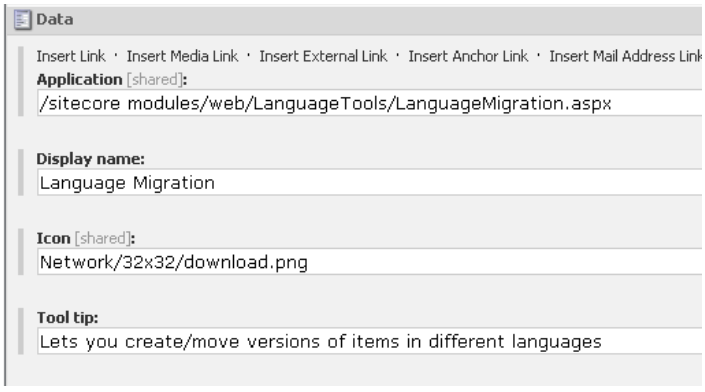
In the case where you are copying values, you may want to set that only the content fields should copy (and not system fields). In that case, you would check the following box:

The final checkbox in the tool allows you to specify if the source version should be deleted.

This would only be the case if there was a problem and all the content was entered into the wrong language by mistake. For example, say you entered all of the Spanish content for a section in the French language versions. You would select French as your Source, Spanish as your target and then check this checkbox. As it adds the Spanish version of each item, it will remove the French version afterwards. Obviously in that case you would have specified that it should transfer the field values as well.

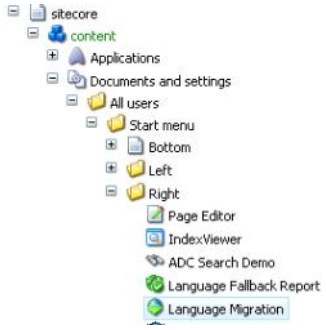
To make this tool easily accessible, put the code into your code base, in 'sitecore modules'. Then in the core database, add a new Application item to the Applications node, named Language Migration:

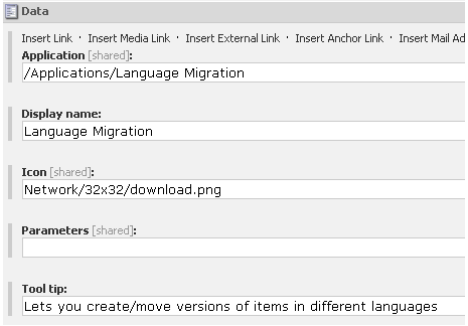




- Application: /sitecore modules/web/Languages/LanguageMigration.aspx
- Display Name: Language Migration
- Icon: Network/32x32/download.png
- Tool tip: Lets you create/move versions of items in different languages
- Appearance Section, Icon: /sitecore/shell/themes/standard/Network/32x32/download.png

And then add it to your Sitecore Start menu as an Application Shortcut, named Language Migration:





- Application: /Applications/Language Migration
- Display Name: Language Migration
- Icon: Network/32x32/download.png
- Tool tip: Lets you create/move versions of items in different languages
- Appearance Section, Icon: /sitecore/shell/themes/standard/Network/32x32/download.png

Running this tool is a big deal and not something that I would make available to content authors or those that are not very familiar with engineering in Sitecore. When our clients need to add another language, we typically have them create a support job and we spend 20 to 30 hours running this tool, QAing, and doing any other additional configurations.

Stay tuned for my next blog post, in which I'm going to discuss another tool, the Language Fallback Report Tool.

http://marketplace.sitecore.net/en/Modules/Language_Migration_Tool.aspx
https://github.com/Verndale-Corp/Sitecore-Language-Migration-Tool
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 9, Verndale's Language Fallback Reporting Tool

For several posts now, I've been talking about multilingual site strategy with Alex Shyba's Partial Language Fallback module and various enhancements and custom integrations to leverage the concept of Fallback and Enforcing Language Version Presence to its fullest capabilities.
http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx

Here I would like to talk about a tool that we implemented at Verndale to help our clients understand where their Sitecore instance stands in integrating languages that have been added to the instance. Manoj Balaraman and Husain Abbasi were a big help in getting the tool cleaned up and looking good from a UX perspective. Summit Phadke and Ankit Joshi also helped with the logic.

When a strategy involves multiple site nodes, very quickly a content author can tell what content is going to appear in which site by simply looking at the content tree. In a solution that has a single site node and multiple versions of a page in different languages, it gets more tricky.

When you change the language through which you are viewing your content items, regardless of which language you choose, you are going to see all of your content items in the tree. It is difficult to see what is missing from a language (by design or not), what language has its own unique content, and what language has content that is falling back to the content of another language.

Some of our clients consider the different language versions of their site as different 'sites', from a business perspective. Their users will pick Germany, or Spain, or France, and to the client, that is a different site, even though it is the same site node and potentially some of the same content falling back to English.

Our clients wanted to be able to see what their site content tree looked like in different languages. And thus the necessity for the Language Fallback Report Tool.

This tool has a similar look and feel and even some similar filter options as the Language Migration Tool.

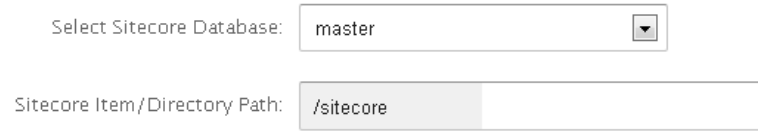
At the top, it outputs the list of fields that are used to check whether a language is falling back. This setting is configured in the Sitecore.SharedSource.PartialLanguageFallback.config:

```
<setting name="Fallback.FieldToCheckForReporting" value="Headline,Sub Headline,Main Content,Teaser" />
```

And you should include the fields that are common and important to most items, to determine if they are falling back or not.

Remember, Alex's module is called 'Partial' for a reason, fallback optionally occurs on ANY of the fields individually, depending on whether any of those fields have values explicitly set into them on any language. You could change the Headline on an item and then let all of the rest of the fields fall back. So in that case, would you consider the item as a whole falling back, or unique? In our opinion, it made more sense to consider that unique, if the point of this reporting tool is to figure out what items might still be pending translation, etc. You could update this setting to have more fields in it, but in this example, we are only picking the most important.

The first few filter options allow the user to specify the database and the path in sitecore to begin the report:



Obviously the less specific you are in the path, the longer the report will take to run. It is recursive and will go down through all of the descendents of the starting path.

The next field allows you to pick the language for which you want to run the report and the way you want to filter the results.

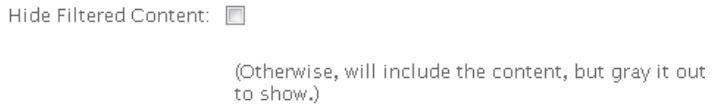
This will will allow you to choose from all languages added to the system and defaults to 'en'.
The select box contains 4 options:

i. Items With Version In This Language
    1) In order to qualify in the results, the item must have the selected language version

ii. Items WITHOUT Version In This Language
    1) In order to qualify in the results, the item must NOT have the selected language version
iii. Items With Content In This Language (explicit)
    1) In order to qualify in the results, the item must have content added to the selected language version in one of the important fields configured and listed at the top of the tool (eg Headline, etc)
iv. Items WITHOUT Content In This Language (fallback only)
    1) In order to qualify in the results, the item must NOT have content added to the selected language version in ANY of the important fields configured and listed at the top of the tool (eg Headline, etc), all should be null in that language version.

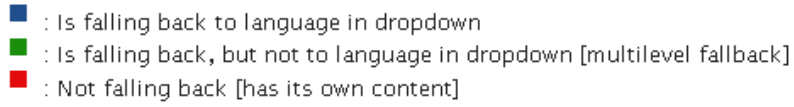
If it does not qualify, by default, all of the items will still be output, but the font color of the item name will be gray instead of black.

HOWEVER, if you selected the Hide Filtered Content checkbox:

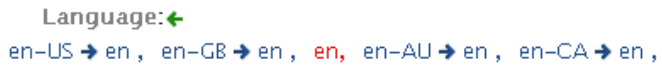


Then the items that do not qualify will not be output at all, with the exception being if a parent item that doesn't qualify has children that DO qualify, then the parent will be output anyway, in gray.

Once the report is run, the results will show below in an expandable tree. A legend will explain the color-coding of the language output to the right of each item.



By default, the languages are hidden and can be toggled to view by clicking the icon to the right of the item name:



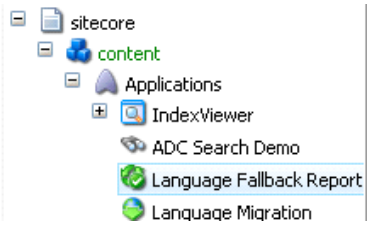
Each language version is displayed and will show with an arrow if it is falling back to another language. NOTE, if there is multi-chained fallback, eg: es-US -> en-US -> en, it won't show the full chain, but only the language it is ultimately falling back to.
So if en-US had one of those fields with explicit content set, then es-US would look like this: es-US -> en-US.
BUT if en-US didn't have values set, es-US would look like this: es-US -> en

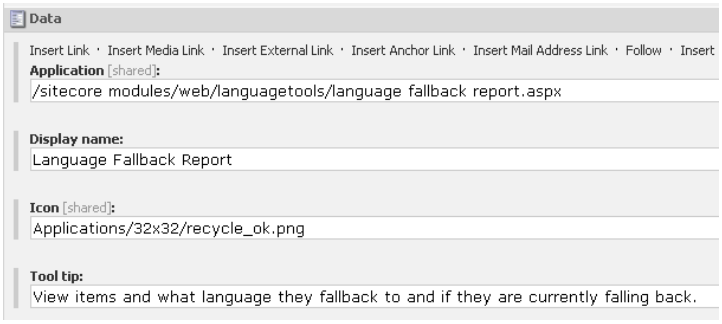
Clicking the Expand All button will open up all of the children items AND all of the language listings for all items.

An example of a good use of this tool is if a client isn't sure what items remain to be translated into a particular language. They can run this report, set the path to the home node, select the language, and then select 'Items WITHOUT Content In this Language' and check 'Hide Filtered Content'. The resulting list of black colored items will be all the items that still need to be translated.

The addition of this tool should help alleviate any concerns of your client when recommending a solution that uses only one site node for a multilingual site. They can rest assured that they will still be able to quickly see what content exists or does NOT exist in specific languages.

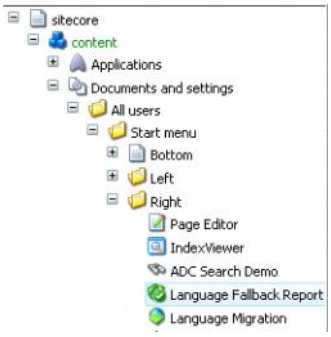
Similarly to the Language Migration Tool, you can make this tool easily accessible by putting the code into your code base, in 'sitecore modules'. Then in the core database, add a new Application item to the Applications node, named Language Fallback Report:

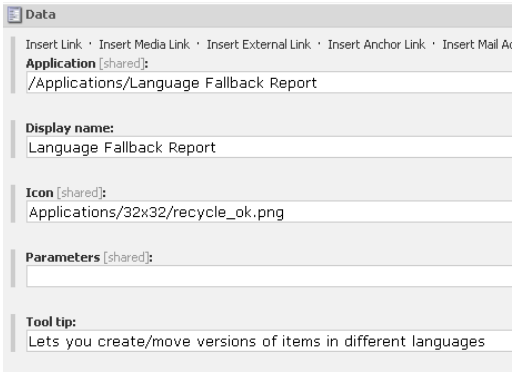




- Application: /sitecore modules/web/languagetools/language fallback report.aspx
- Display name: Language Fallback Report
- Icon: Applications/32x32/recycle_ok.png
- Tool tip: View items and what language they fallback to and if they are currently falling back.
- Appearance Section, Icon: /sitecore/shell/themes/standard/Applications/32x32/recycle_ok.png

And then add it to your Sitecore Start menu as an Application Shortcut, named Language Fallback Report:





- Application: /Applications/Language Fallback Report
- Display name: Language Fallback Report
- Icon: Applications/32x32/recycle_ok.png
- Tool tip: View items and what language they fallback to and if they are currently falling back.
- Appearance Section, Icon: /sitecore/shell/themes/standard/Applications/32x32/recycle_ok.png

In my next and final post on the matter of multilingual Sitecore sites and Language Fallback, I will discuss additional configurations and settings, layouts and media, conditional rendering, personalization, translator tools, and other cultural considerations.

http://marketplace.sitecore.net/en/Modules/Language_Fallback_Report_Tool.aspx
https://github.com/Verndale-Corp/Sitecore-Language-Fallback-Report-Tool
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 10, Additional Miscellaneous Topics for Multilingual and Language Fallback Implementation

We've come to the final post in my blog series on Language Fallback and using Alex Shyba's Partial Language Fallback Module. http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx

In my previous posts, I've gone over strategy for multilingual sites with language fallback, instructions on how to install and configure the Partial Language Fallback Module, using the module to Enforce Language Version Presence and when to do so, customizations that can be made involving that Enforcement, using fallback with the Sitecore Dictionary, using fallback with Advanced Database Crawler, defaulting language version creation for fallback, the Language Migration Tool, and the Language Fallback Report Tool.

Now I'd like to discuss a few other things to consider during implementation, including references to other blogs and tools that will definitely aid you in finishing your multilingual and multi-cultural solution.

i. LinkManager attributes for languageEmbedding and useDisplayName:

```
<add name="sitecore" type="Sitecore.Links.LinkProvider, Sitecore.Kernel" addAspxExtension="false"
alwaysIncludeServerUrl="false" encodeNames="true" languageEmbedding="always"
languageLocation="filePath" lowercaseUrls="true" shortenUrls="true" useDisplayName="true" />
```

- This LinkManager setting is in the web.config, but may be patched and overridden in other config files, you'll want to make sure you update these attributes in the config that is ultimately used by Sitecore.
- The languageEmbedding attribute, when set to 'always', will embed the current language into the url when outputting any links to items in the site, between the hostname and the path, eg: www.yoursite.com/en-us/about-us
  - A simple strategy is that if your site is multilingual and you do not have custom hostnames for each language (eg: www.yoursite.de), then set the languageEmbedding attribute to 'always'. Otherwise set it to 'never'. 'asNeeded' is also an option, but can has unexpected results.
- NOTE, a limitation to this setting is if you set it for one site in your instance, it is set for all. That can be a problem if you have a second site that is not multilingual and you don't want the language showing up in the url. A custom workaround to this is to create a LinkProvider that will check a languageEmbedding property in the Site node. There is an example of this in my demo.

```
<linkManager>
      <patch:attribute name="defaultProvider">custom</patch:attribute>
      <providers>
        <clear />
        <add name="custom"
             type="Verndale.SharedSource.SitecoreProviders.CustomLinkProvider,
Verndale.SharedSource"
             addAspxExtension="false" alwaysIncludeServerUrl="false" encodeNames="true"
languageEmbedding="always"
             languageLocation="filePath" lowercaseUrls="true" shortenUrls="true"
useDisplayName="true" />
      </providers>
    </linkManager>
```

  - In the CustomLinkProvider class, it checks to see if a languageEmbedding site attribute is set using the method CheckOverrideLanguageEmbedding in LanguageHelper.cs, before using the global default one in the LinkManager configuration. I have to give Aaron Paul credit for this idea.
- The useDisplayName attribute tells Sitecore to use the Display Name field when rendering the link to the item, instead of the Name field. You can change this Display Name within different language versions. This way you can translate the page name into different languages so that the url isn't outputting English words if you are viewing the site through Spanish, for example. If you don't set Display Name at all, then LinkManager will use Name.
  - The strategy here is based on client preference, is it important to their multilingual users to have the urls translated as well?

ii. Sitecore Setting Media.UploadAsVersionableByDefault

```
<!--  MEDIA - UPLOAD AS VERSIONABLE AS DEFAULT
        This setting controls if uploaded media is versionable by default or not.
      -->
 <setting name="Media.UploadAsVersionableByDefault" value="false" />
```

- By default, Media items are not versionable and the above setting is set to false in the web.config. If you upload an image in one language, it will persist across all language versions.
- If you change this to true, then versioning will apply and you would have to set the media item into all language versions, or enable fallback, but if enforce version presence is turned on, you'll have to make sure all language versions at least exist.
  - NOTE: To enable fallback on versioned media items, please remember to go to the /sitecore/templates/system/media/versioned folder and for each template that has fields, click the Enable Fallback ribbon command so that the fields will fallback.
- I would recommend setting this to true, if you plan on using fallback on your site.
- The obvious benefit to doing this is you can create a different media item version for every language version of a logo (for example). You can have the single logo, and just attach different images to each language version. It could keep the media library a little cleaner, depending on how your content authors are hoping to organize the media library.
- WARNING 1: caching of images and other media that is done by browsers could prevent different versions in languages from being

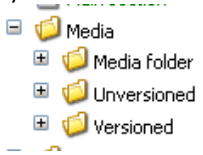
shown if a user is switching back and forth between languages.

- This is because, by default, language is not embedded into media urls and so switching between languages would point at the same media url, which many browsers would cache.
- You could override and extend the GetMediaUrl of the MediaProvider so that it takes the languageEmbedding attribute into consideration. This method is not only accessible programmatically, but the sc:Image and rich text fields use it to output the media urls. The demo project implements this and sets the CustomMediaProvider in the Fallback.config file.

```
<mediaLibrary>
      <mediaProvider>
        <patch:attribute name="type">
Verndale.SharedSource.SitecoreProviders.CustomMediaProvider,
Verndale.SharedSource</patch:attribute>
      </mediaProvider>
    </mediaLibrary>
```

- A different way to handle this need would be to have different media items entirely for different languages and then just override the language version of the item that is using the media to reference different media items.

- WARNING 2: if you are making this update for a site that already has media items added to it, then those media items were probably added as 'Unversioned' media templates



- When this attribute is changed to true, going forward, media items will be added as 'Versioned' media templates.
- So now you need to determine if and how you can change the media that is already in the system from 'unversioned' to 'versioned' and what repercussions that may have.

iii. Determining Sitecore Context Language

- A few years ago, John West wrote a blog post about overriding Sitecore's logic for figuring out the context language through which the user is viewing the site. http://www.sitecore.net/Community/Technical-Blogs/John-West-Sitecore-Blog/Posts/2010/11/Overriding-Sitecores-Logic-to-Determine-the-Context-Language.aspx
- The gist is that out-of-the-box, Sitecore considers the following logic for determining context, in priority of this order:
  1) The sc_lang query string parameter. (results in session cookie)
  2) The language prefix in the path in the requested URL. (results in session cookie)
  3) The language cookie associated with the context site.
  4) The default language associated with the context logical site.
  5) The DefaultLanguage setting specified in web.config.
- John's solution describes how to add a custom processor after the ItemResolver. The Fallback parts are probably unnecessary now considering the Partial Language Fallback Module, but SetCulture and PersistLanguage could be very helpful.
- You may want to add additional checks like:
  - Consider the browser language preference
  - Default another language if missing (not so necessary with Fallback enabled)
  - Load from a persistent cookie
  - Check authenticated user's preference saved to profile
- And you may want to add in custom code to set CurrentCulture (for dates, currency and such) based on language of the site, rather than operating system settings. There is an example of this in my demo.
  - Add another patch in the Fallback.config file's httpRequestBegin pipeline:

```
<!-- Custom pipeline processor that will set the .net culture based on the current
sitecore context language -->
        <processor
patch:after="*[@type='Verndale.SharedSource.SitecoreProcessors.ItemLanguageValidation,
Verndale.SharedSource']" type="Verndale.SharedSource.SitecoreProcessors.CultureResolver,
Verndale.SharedSource" />
```

  - And then in the CultureResolver code, set the .NET culture based on the Context Language:

```
System.Threading.Thread.CurrentThread.CurrentUICulture =
                          new
System.Globalization.CultureInfo(Sitecore.Context.Language.Name);
System.Threading.Thread.CurrentThread.CurrentCulture =
System.Globalization.CultureInfo.CreateSpecificCulture(Sitecore.Context.Language.Name);
```

iv. Other Cultural Considerations

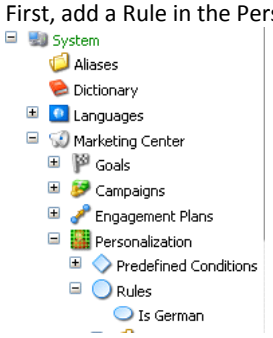
- As mentioned above, the current culture through which .NET is displaying things such as date is important to consider. You could set it in a processor during httpBeginRequest pipeline. Or you could set it at the time the user selects the language or when language is loaded from a cookie. Regardless, you would set it with the above code.
- If you don't set it at this global level, you would have to pass in a CultureInfo object for the specific language whenever you are

working with something that could differ based on culture (string methods, date methods, currency methods, etc).

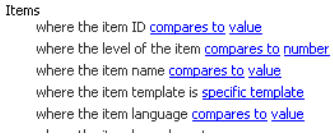
```
System.Globalization.CultureInfo ci = new
System.Globalization.CultureInfo(Sitecore.Context.Language.Name));
DateTime dt = DateTime.Parse(date, ci);
And
System.Globalization.CultureInfo CultureInfo ci = new
CultureInfo(Sitecore.Context.Language.Name));
amount.ToString("C", ci);
```

v. Unsharing the Layout fields
   - By default, the Layout fields, such as Renderings, are 'Shared' fields. This means that whatever it is set to in any language or version number is shared across all languages, and presentation cannot be changed based on language version. You can use conditional rendering and personalization to change out datasources of sublayouts on the page and change visibility of the sublayouts, but otherwise, you are pretty much committed to having the same presentation across languages.
   - Although I personally have not done this before, you can un-share the Layout fields. Jan Hebnes discusses doing so for a project here and it is worth reading when considering this strategy:
   - http://www.sitecore.net/Community/Best-Practice-Blogs/Jan-Hebnes/Posts/2012/09/Unsharing-the-Layout-field-in-Sitecore.aspx
   - There are pros and cons to doing this and extra customization work that you must put in place. I have also read that some unexpected results can occur when using fallback with these fields, so be forewarned!

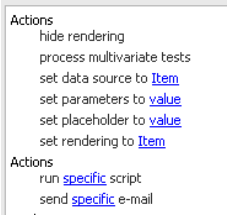
vi. Conditional Rendering and Personalization by Language
   - A simpler, less intrusive alternative to unsharing the Layout fields is to setup some rules that can be used in conditional rendering.
   - First, add a Rule in the Personalization section of the Marketing Center.



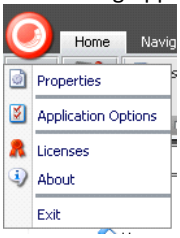
   - There several filters you can choose from, including 'where the item language compares to value'.



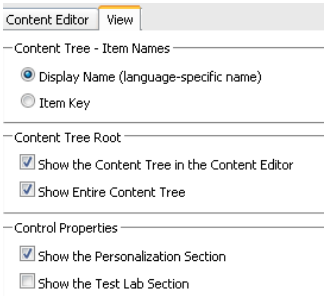
   - And many result actions:



   - Once the rule is setup, you should make sure that the personalization section is turned on within Controls by going to the sitecore icon and choosing Application Options.



     - Switch to the View tab and select the 'Show the Personalization Section' in Control properties

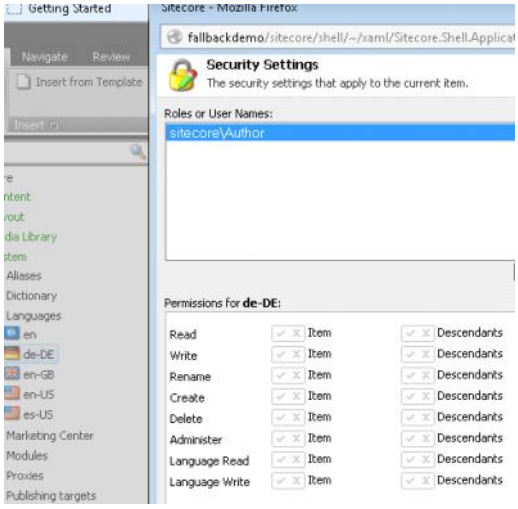


- Then when you are setting the control properties of a sublayout, you will be able select the rule:

- Sitecore will apply the rule upon rendering.

vii. Security For Languages

- There is most likely going to be a need for different content authors to have different access rights to different languages. It could be that an author in Germany will need to be able to go in and view/translate only the German version of all the pages. For this reason, Sitecore has two additional permissions that can be set when you are on a language item (in /sitecore/system/languages): Language Read and Language Write. You could create a German Editor Role that only has access to the German language. Give that role to anyone who will be editing German content only.



viii. Translating Content and Tools to help

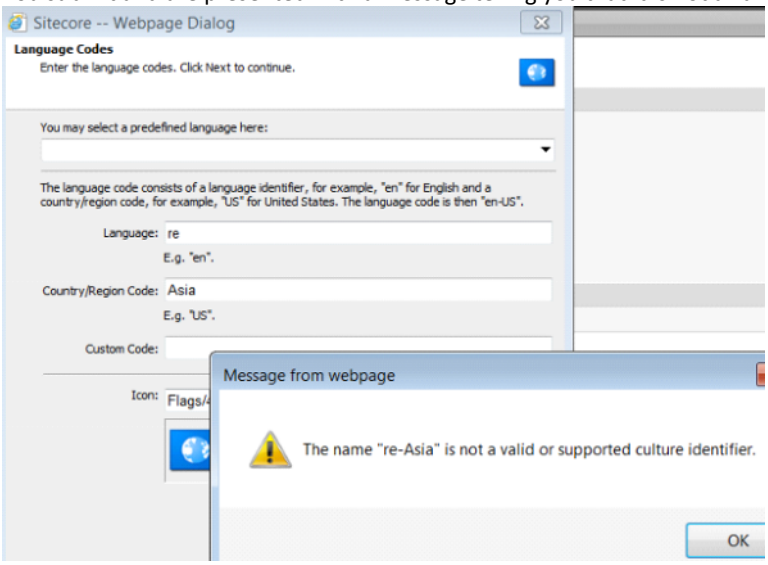
- The whole point of a multilingual site is to have content in other languages. Content authors could supply their English content to a professional translating service, receive back the content and then manually enter it. They could have translators on staff who log into sitecore and update the content. But there are also some tools that integrate seamlessly into Sitecore to aid with this.
  - Clay Tablet
    - http://www.sitecore.net/Partners/Technology-Partners/Clay-Tablet.aspx
    - Per the above link: "You can select one item or all your content and send it out to any translation provider or technology (like Google Translate) instantly. More importantly, translated content is automatically returned into the correct locations in Sitecore, saving more time."
    - This is the Sitecore recommended solution and was co-developed with Sitecore. It is not free, however.
    - We recently integrated a client with this and it went well.
    - My colleague Julie Moynihan had this to say about it: Clay tablet is a Sitecore integrated module that provides the ability to send items to an external system/ vendor for translation. Translation can be batched and multiple types of translation (machine, human) or multiple vendors. The items are placed into a workflow and sent through the Clay

tablet system to the translation vendor. Clay tablet works with the translation vendor regarding integration with their software. Items are then returned through Sitecore and placed back into workflow.

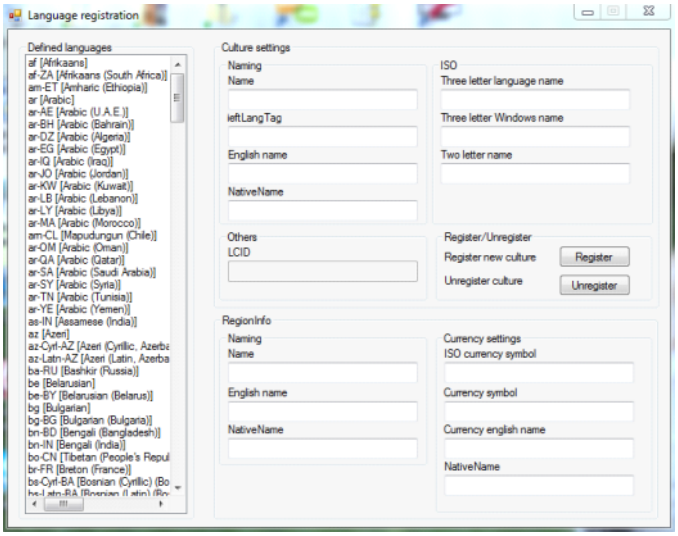
- Item Translator: Sitecore Marketplace Module
  - http://marketplace.sitecore.net/en/Modules/Item_Translator.aspx
  - I haven't used this myself, however it seems pretty cool. It leverages the Google translate functionality. The content author will click a button and it will translate the content. Granted, Google translate is not perfect, but it could be a good starting point (and it's free).

ix. Custom Language Registration

- My coworker Summit Phadke recommended that I include the following information since he had to implement this for one of our clients as well.
- When you add a new language to Sitecore, in /sitcore/system/language, it allows you to either choose from a pre-defined language culture, or you can enter your own. If you decide to enter your own, you can specify the code for Language and optionally a code for Culture
- At this point you decide you want a language specifically for German in the United State, so you enter 'de' and 'US' in the language and Culture code fields.
  - Or maybe you want to make up a culture that represents a region, eg: 're' for the language, meaning region, and 'Asia' for the culture.
- You submit and are presented with a message telling you that it is not a valid option



- The problem is that Sitecore is limited to using what the .NET framework already has registered on the server/computer you are running the site on.
- Therefore, you must register the new custom language on the server/computer first.
- WARNING: This will need to be done on ALL servers that this application will ever live on, and must be considered when deciding if you indeed want to use a custom language/culture. Will you want to do that or even be able to do that?
  - ADDITIONALLY, if you use custom cultures, you will need to tweak any code that you may have written to set the culture within the application (see Other Cultural Considerations above). .NET won't know how to format text, currency, dates, etc if the culture is custom, so you'll need to figure out a way to default it to something else.
- If you decide this is something you want to do, then there is a Sitecore Marketplace tool for this:
  - http://marketplace.sitecore.net/en/Modules/Custom_Language_Registration
  - It contains an exe file (and the source code) for a windows app
  - You run the exe file and it will open the app:
  - You can pick from pre-defined languages or set everything custom, including Names, ISO, Region Info, and Currency Info
  - Finally click Register
  - You also have the option to Unregister



This brings me to the end of my blog series on Partial Language Fallback in conjunction with multilingual strategy in Sitecore websites. I hope that after reading through all the posts, you have a comprehensive understanding of how Alex Shyba's Partial Language Fallback Module should be installed and configured, what to consider when deciding to enforce language version presence, and additional updates to ensure fallback works with the dictionary and Advanced Database Crawler. And now at your disposal are several tools to make it easier to work with multilingual sites using fallback, including the Language Migration Tool and Language Fallback Report Tool.

https://github.com/Verndale-Corp/Sitecore-Misc-Multilingual
https://github.com/Verndale-Corp/Sitecore-Fallback-FullDemo

POST 11, Sitecore 7 and Language Fallback

Earlier this year, I blogged in a series on Language Fallback and using Alex Shyba's Partial Language Fallback Module.
http://marketplace.sitecore.net/en/Modules/Language_Fallback.aspx

The various posts discussed all aspects of fallback, theory and things to consider, useful tools, and enhancements. Here are some quick links to those posts:

1. Introduction to Language Fallback
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-1.aspx)
2. How to Install and Configure Alex Shyba's Partial Language Fallback Module
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-2.aspx)
3. Enforcing Language Version Presence, Out-Of-The-Box
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-3.aspx)
4. Enforcing Language Version Presence, Customizations
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-4.aspx)
5. Using Fallback With The Dictionary
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-5.aspx)
6. Using Fallback With The Advanced Database Crawler
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-6.aspx)
7. Defaulting Language Versions For Fallback
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/03/Fallback-Series-Post-7.aspx)
8. Language Migration Tool
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/04/Fallback-Series-Post-8.aspx)
9. Verndale's Language Fallback Reporting Tool
   (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/04/Fallback-Series-Post-9.aspx)
10. Additional Miscellaneous Topics for Multilingual and Language Fallback Implementation
    (link to http://www.sitecore.net/Learn/Blogs/Technical-Blogs/Elizabeth-Spranzani/Posts/2014/05/Fallback-Series-Post-10.aspx)

Much of my experience with all of the above were with Sitecore versions 6.5 and 6.6, though. With Sitecore 7 being the norm now, and the impending Sitecore 8 on the horizon, I figured it was about time to run some tests and make sure all of this still worked on the latest publicly

released version: Sitecore 7.2 Update 2 (revision 140526).

I am happy to announce that with the exception of the Advanced Database Crawler post, everything here is still relevant and works like a charm.

As per the "How to Install and Configure" post, you will need to use the source code to compile the project against the version of the Sitecore Kernel and Client dlls that you will be using. Once you do that, drop your fallback dll into your main project and you are all set.

As mentioned above, the glaring exception here is the Advanced Database Crawler situation. As most people reading this will know, the release of Sitecore 7 was the death knell of ADC. RIP ADC…

One of the most important overhauls in Sitecore 7 is the Sitecore.Search enhancements. It is a huge change and there is a ton of information you need to absorb. With the exponential increase in flexibility comes a responsibility to really get to know the options and I recommend spending quality time reading not only the official Sitecore documentation like the Search and Indexing guide and Search Scaling guide for 7, but also the plethora of blog posts that various members of the community have been churning out.

I'm not going to go into any more detail about Search here, but will include in the github files my SearchHelper that replaces some of the ADC GetItems methods with Linq to Sitecore functionality.

What I will talk about is the replacement for enabling fallback with Lucene indexes. If you read my post about fallback with ADC, you will have seen that I addressed fallback at the time of index. I felt this was the simplest way to deal with it.

To refresh the scenario: Say you have an About Us page and all of the text is in the en-US language. You also have an en-ES language version, but have no overridden any content on it, it falls back to en-US. When viewing the site in en-ES, the About Us page displays the en-US content. The user searches for the word 'About' in the global search and gets no results. This is because the Search crawler, by default, knows nothing about fallback and to it, the About Us item simply has no content, and therefore no content is entered into the index for it.

I had added to the crawler logic in ADC to look for the fallback language version of each item field that was null and check if that fallback language version had a value. If so, then I added that value to the index for the language falling back instead. It worked well.

Now I had to figure out how to replicate the concept with Sitecore 7, without the benefit of having the source code at my fingertips as I had with ADC. The sheer amount of exposed logic and features available with search in Sitecore 7 added to the effort but I finally found the setting I was looking for in the Sitecore.ContentSearch.Lucene.DefaultIndexConfiguration.config file. In here is the following:

```
<!-- DOCUMENT BUILDER
        Allows you to override the document builder. The document builder class processes all the fields in the Sitecore items and prepares
        the data for storage in the index.
        You can override the document builder to modify how the data is prepared, and to apply any additional logic that you may require.
    -->
    <documentBuilderType>Sitecore.ContentSearch.LuceneProvider.LuceneDocumentBuilder,
Sitecore.ContentSearch.LuceneProvider</documentBuilderType>
```

I reflected this class and after spending time reviewing the various methods, I identified that the AddField(IIndexableDataField field) method was the one that I needed to override. I copied the logic and added in my own steps that are better than the implementation that I originally did with ADC, as I realized I could take advantage of the Partial Language Fallback's method: FallbackLanguageManager.ReadFallbackValue. This method will take care of the recursions automatically (in the case of chained fallback), because the Partial Language Fallback module additionally overrides the StandardValues provider.

So essentially what you need to do for this to work right is to:

1. Add a class that implements LuceneDocumentBuilder and overrides the AddField method, leveraging the FallbackLanguageManager.ReadFallbackValue method
2. Update the documentBuilderType attribute in the config (or better yet patch it in your Fallback.config file)
3. Use Lucene as your index

I don't yet have a solution for SOLR but I'm currently working a project that uses SOLR so I'm sure I'll have that over the next few months.

I will paste the source code here for the AddField method, but I have also added it to the git repository and you can download not only the fallback config with the patch and CustomLuceneDocumentBuilder.cs file, but also the example of the SearchHelper method that I have so far (still a lot more that I want to add to that helper class though!)

```
using Sitecore.ContentSearch.LuceneProvider;
using Sitecore.Data.Items;
using Sitecore.Data.Fields;
```

```
using Sitecore.SharedSource.PartialLanguageFallback.Extensions;
using Sitecore.SharedSource.PartialLanguageFallback.Managers;

public class CustomLuceneDocumentBuilder : LuceneDocumentBuilder
  {
    private readonly LuceneSearchFieldConfiguration defaultTextField = new LuceneSearchFieldConfiguration("NO", "TOKENIZED", "NO", 1f);

    public CustomLuceneDocumentBuilder(IIndexable indexable, IProviderUpdateContext context)
     : base(indexable, context)
    {
    }

    public override void AddField(IIndexableDataField field)
    {
      object fieldValue = this.Index.Configuration.FieldReaders.GetFieldValue(field);

      //UPDATED, Added By Verndale for Fallback
      //<!--ADDED FOR FALLBACK DEMO-->
      if (fieldValue == null || fieldValue == "")
      {
        // Get the Sitecore field for the Indexable Data Field (which is more generic) that was passed in
        // If the field is valid for fallback, then use the ReadFallbackValue method to try and get a value
        Sitecore.Data.Fields.Field thisField = (Sitecore.Data.Fields.Field)(field as SitecoreItemDataField);
        if (thisField.ValidForFallback())
        {
          // ReadFallbackValue will get the fallback item for the current item
          // and will try to get the field value for it using fallbackItem[field.ID]
          // Merely calling fallbackItem[field.ID] triggers the GetStandardValue method
          // which has been overridden in the standard values provider override FallbackLanguageProvider
          // which will in turn call ReadFallbackValue recursively until it finds a value or reaches a language that doesn't fallback
          fieldValue = FallbackLanguageManager.ReadFallbackValue(thisField, thisField.Item);

          // BELOW ARE EXAMPLES OF CASTING BACK AND FORTH FOR SITECORE FIELDS AND ITEMS TO INDEXABLE AND BACK
          // -- Get Indexable from Sitecore Item
          // SitecoreIndexableItem indexableFallbackItem = (SitecoreIndexableItem)fallbackItem;

          // -- Get Sitecore Item from Indexable variable
          // Item thisItem = (Item)(Indexable as SitecoreIndexableItem);

          // -- Get Indexable field from indexable item's field by field id
          // IIndexableDataField fallbackField = indexableFallbackItem.GetFieldById(field.ID);
        }
      }

      string name = field.Name;
      LuceneSearchFieldConfiguration fieldSettings = this.Index.Configuration.FieldMap.GetFieldConfiguration(field) as
LuceneSearchFieldConfiguration;
      if (fieldSettings == null || fieldValue == null)
        return;
      float boost = BoostingManager.ResolveFieldBoosting(field);
      if (IndexOperationsHelper.IsTextField(field))
      {
        LuceneSearchFieldConfiguration fieldConfiguration = this.Index.Configuration.FieldMap.GetFieldConfiguration("_content") as
LuceneSearchFieldConfiguration;
        this.AddField("_content", fieldValue, fieldConfiguration ?? this.defaultTextField, 0.0f);
      }
      this.AddField(name, fieldValue, fieldSettings, boost);
    }
  }
```

https://github.com/Verndale-Corp/Sitecore-7-Fallback-Lucene-Indexing