about me

3 yrs @ Markido
Co-Founder & CTO

11 yrs @ Fuel Industries
Co-Founder & CTO
100 employees, $8M+ in '07

2 yrs @ Lockheed Martin

Comp Sys B. Eng. @ Carleton U

contact

twitter.com/NesbittBrian

brian@nesbot.com

what i am doing

*Engage*

Carbon

what i play with

sparkjava.com

dropwizard.io

slimframework.com

github

# Multilingual site using Slim

*Jun 26, 2012*



A question entitled Multilingual site with Slim was posted to the help forum the other day. Here is my take on how I would start to create such an application with 3 pages: Home, About and Login.

## Overview

We will start by looking at how the current active language is parsed and extracted from the requested URI using a `slim.before` hook. Extracting the language from the URI is important as it will allow us to use the same set of routes for all of the languages. We will create a simple custom view template that will use a translator service to perform the lookups we will need to display the multilingual content. The translations will be stored in language resource files. We will also subclass the Slim application class to ensure `urlFor()` is still available and working for us. By the end of this post we will have a full multilingual application that will be able to responsd to the following URI's:

```
http://127.0.0.1/ http://127.0.0.1/en http://127.0.0.1/de
http://127.0.0.1/ru http://127.0.0.1/about
http://127.0.0.1/en/about http://127.0.0.1/de/about
http://127.0.0.1/ru/about http://127.0.0.1/login
http://127.0.0.1/en/login http://127.0.0.1/de/login
http://127.0.0.1/ru/login
```

## Spoiler Alert

If you want to skip my ramblings below you can run the following commands to have the application up and running in about 10 seconds, depending on typing speed. This assumes you are using PHP 5.4+ with the embedded webserver. If not it will take you longer.

```
git clone git://github.com/briannesbitt/Slim-Multilingual.git
cd Slim-Multilingual curl -s http://getcomposer.org/installer |
php php composer.phar install php -S 127.0.0.1:80
```

The project uses composer to install the latest `1.6.*` version of Slim and also to generate an autoload for all of the classes in the `app/lib/` directory via the composer classmap feature.

### Parsing the current language and common routes

As you can see from above some of the URIs implicitly specify the language while others do not. When the language isn't specified and a request for the index is made, we first use the `ACCEPT_LANGUAGE` header to try and guess the appropriate language from our sites available languages. If a suitable language is not matched we will fallback to the site default and go ahead and render the page. You could put a language chooser page in place if you wanted to rather than using the default language. Our application will simply render the requested page using english but also show a language switch option which will maintain the current page context when switching.

Lets start this by first showing the common route for the homepage. We want the following route to match against URI's #1, #2, #3 and #4 from above.

```
$app->get('/', function () use ($app) { $app->render('home.php'); });
```

The only way to do that (without using optional route parameters due to their limitations) is to parse and extract the language from the URI. This must happen before Slim performs its routing logic so the proper matches take place. For this we will use a `slim.before` hook. Slim will perform its route matching based on the URI in `$env['PATH_INFO']` where `$env = $app->environment()`. So as part of the `slim.before` hook we simply loop through the `$availableLangs` and see if the URI begins with `/lang/`. If so we can `substr()` the `$env['PATH_INFO']` to extract the language and write the shorter URI back to the `$env['PATH_INFO']` variable. If we are quiet enough then Slim won't know the difference and it will go ahead and match routes against the modified URI.

```
$pathInfo = $env['PATH_INFO'] . (substr($env['PATH_INFO'], -1) !== '/' ? '/' : ''); // extract lang from PATH_INFO
foreach($availableLangs as $availableLang) { $match = '/'.$availableLang; if (strpos($pathInfo, $match.'/') === 0) { $lang = $availableLang; $env['PATH_INFO'] = substr($env['PATH_INFO'], strlen($match)); if (strlen($env['PATH_INFO']) == 0) { $env['PATH_INFO'] = '/'; } }
}
```

The first line of code from above is necesssary to match against a URI like `/en`. Our attempted match string will be `/en/` so we need to append the trailing `/`. If we only match against `/en` then we could improperly intercept other routes

like `/entertain` which would be a request for the entertain page without a language specified. The full `slim.before` hook can be seen at https://github.com/briannesbitt/Slim-Multilingual/blob/master/app/hooks.php

Once we have the `$lang` determined we can set it in our view at the end of the hook as the following code shows. We also initialize some variables that will always be available to the view. The custom view will be examined in a bit.

```
$app->view()->setLang($lang); $app->view()->setAvailableLangs($availableLangs); $app->view()->setPathInfo($env['PATH_INFO']);
```

With the URI modifications completed Slim will go ahead and perform its matching as usual, not knowing anything about the language that was once there. This allows us to create our 3 page application with the following 4 routes.

```
<? $app->get('/', function () use ($app) { $app->render('home.php'); })->name('home'); $app->get('/about', function () use ($app) { $app->render('about.php'); })->name('about'); $app->get('/login', function () use ($app) { $app->render('login.php'); })->name('login'); $app->post('/login', function () use ($app) { $app->render('login.php', array('error' => $app->view()->tr('login-error-dne', array('email' => $app->request()->post('email'))))); })->name('loginPost');
```

You can see the login page requires 2 routes, one for displaying the form and a second to receive the form POSTing.

### Custom view with Master Template

The templating provided with Slim is pretty basic (there is a Slim-Extras repo that integrates other templating engines into Slim). To prevent the duplication of html the typical Slim template will look like this:

```
<? include 'header.php'; <p>This is my content.</p> include 'footer.php';
```

I would suggest using a Slim-Extras template but to keep this application simple and from having any external dependencies we will turn the tables. With no change to your routes and very little code you can add master template functionality to your custom template and thus preventing the duplication of including the header and footer on each page.

First we create a custom view class `MasterView` that extends `Slim_View`. We setup a constructor that accepts a master template parameter and stores it.

Now to complete the view code we just need to override the `render()` function. If the `masterTemplate` was set then

`render()` swaps the template with the master template and sets up a `$childView` variable so the master template can `require` it. Here is the full `MasterView` class in all of its 15 lines of glory.

```
<? class MasterView extends Slim_View { private
$masterTemplate; public function __construct($masterTemplate) {
parent::__construct(); $this->masterTemplate = $masterTemplate;
} public function render($template) { $this-
>setData('childView', $template); $template = $this-
>masterTemplate; return parent::render($template); } }
```

### The Translator and the MultilingualView

Is it just me or does that sound like a cheesy horror movie name? Anyway, of course our view needs to provide translation of content for us. This I did using a translator service that gets injected into the `MultilingualView` constructor. The view then provides a simple helper `tr()` to make it easier for our templates to access. With a simple `str_replace()` the translator provides a way to inject variables into the translated content. So you can setup error messages like `Sorry, there is no user with an email of "{{email}}".`

As we saw earlier, the `slim.before` hook is where the language is determined. At the end of that hook the langauge, available languages and path info is set. These setter functions simply use the parent `setData()` function to create variables that will exist in the context of the view. So in the view if you want to know the current language you can just do `$lang` or looping through the available lanagues can be done like `foreach($availableLangs as $availableLang)`. This also makes the helper methods to perform the translation easier to access as the current language is known and doesn't have to be passed in, `$this->tr('home-content')`.

The language resoure files follow the naming convention of `lang.en.php` where `en` is the language code. These are simple PHP files that use an associative array to setup the translations. Since it is just PHP you can require other files, load them from a datastore or use HEREDOC for longer paragraphs. If a translation is requested for a key that does not exist a blank string is returned and an error is written to the Slim application log. This could be changed to throw an exception to ensure its not missed during testing. I also auto require a `lang.common.php` file that has common terms. A good example for using this is the language chooser. You don't display `German` when on the english site. You always dipslay `Deutsch` for all languages so someone looking for the German

version will be able to find it.

Sometimes its easier to view all of the content in the context of the page and html rather than in the language resource file. Here you have a few choices. Say you have an about page that is broken into 4 <p> tags. You can split the p tags over a few language resource keys, use a HEREDOC and put the html in a single resource key or include a language specific sub template. Here are the implementation options for the `about.php` view file:

```
<?php echo $this->tr('about-p1')?>
<?php echo $this->tr('about-p2')?>
<?php echo $this->tr('about-p3')?>
<?php echo $this->tr('about-p4')?>
```

```
<?php echo $this->tr('about-content')?>
```

```
<?php require 'about '.$lang.'.php'?>
```

I think for this example I would choose the 3rd option. Anything more complicated would require mixins, more child templates etc. etc. and I would really start to lean to looking at the many templating engines out there. What I do like about this one is that its really simple, easy to understand and flexible since it is just PHP.

### Ensuring $app->urlFor() still works

This is actually much easier that it seems. You can subclass the Slim application and override the `urlFor()` function. It just needs to prepend `/lang` (lang being the current language) to the url returned by the parent `urlFor()`. The full 6 line class is shown here:

```
<? class MultilingualSlim extends Slim { public function urlFor( $name, $params = array() ) { return sprintf('/%s%s', $this->view()->getLang(), parent::urlFor($name, $params)); } }
```

Follow this up by changing your application creation from `$app = new Slim(array('templates.path' => './app/views/'));` to `$app = new MultilingualSlim(array('templates.path' => './app/views/'));` and your done.

I'll also mention that rather than setting the custom view when the application is created, I used the `$app->view()` function to pass in an instance of the `MultilingualView` class. This allowed me to pass some arguments when constructing the view class.

## Wrap up

Thats it! This is a first pass at creating a simple multilingual site using Slim. There isn't much code here so I think its safe to say someone could to take a look and be up and running in a few minutes. Oh and to try out the repo, use composer to install Slim and setup the application autoload you will be up and running in no time! Jump back to the Spoiler Alert at the beginning to see the commands necessary to try this out.

## Links

http://slimframework.com
http://github.com/briannesbitt/Slim-Multilingual
http://help.slimframework.com/discussions/questions/244-multilingual-site-with-slim


‹ Slim wildcard routes via route middleware
Home
Slim wildcard routes improved ›