Extracted from:

Rails Recipes

Adding Support for Localization

Credit

Long-time Ruby programmer and designer Bruce Williams (who, in a past life, worked as an Arabic translator) wrote this recipe.

Problem

Your application is (or will be) used all over the world. You'd like it to support multiple languages and format information such as times and currency specifically for each user's locale.

Ingredients

Josh Harvey and Jeremy Voorhis's Globalize plugin, installable from the root of your Rails application with the following:

```
bruce> ruby script/plugin install \
   http://svn.globalize-rails.org/svn/globalize/globalize/trunk
```

Solution

For this recipe, we're going to model a small online store that specializes in Middle Eastern and Asian food imports. The store's customer base is made up primarily of non-English-speaking Middle Eastern and Asian people, so localization is a must.

Assuming you have the Globalize plugin installed, the first thing you'll need to do is to set up its required tables and data:

```
bruce> rake globalize:setup
```

Next, you'll need to set your base language and default locale in config/environment.rb. You can add this anywhere at the end of the file. For an English speaker in the United States, it would be:

```
include Globalize
Locale.set_base_language 'en-US'
Locale.set 'en-US'
```

International Characters

By default, Rails isn't set up to handle non-English characters. Here's what you'll need to do to make it work:

1. Add the following to your config/environment.rb file:

   ```
   $KCODE = 'u'
   require 'jcode'
   ```

   This sets Ruby's character encoding to UTF-8.

2. Next you need to set your database connection to transfer using UTF-8. For MySQL and PostgreSQL, you can do this by adding a line to your database's configuration in your config/database.yml file. For MySQL it would be this:

   ```
   encoding: utf8
   ```

   And for PostgreSQL:

   ```
   encoding: unicode
   ```

   For SQLite, simply compiling UTF-8 support in is all you need to do. For other database systems, consult the system's documentation on how to set character encoding for connections.

3. Set the character encoding and collation for the database and/or tables you'll be accessing. *Collation* refers to the method that will be used for sorting. If you change your character set to Unicode but leave your database's collation untouched, you may end up with some unexpected results coming from ORDER BY clauses. For details on how to set character set and collation for your database, check your database software's manual.

4. Set encoding information in the content type your application returns for each request. The easiest way to do this is to put an after_filter() in your ApplicationController. Here's an example filter that will work in most cases (including RJS templates):

   ```
   after_filter :set_charset
   def set_charset
     unless headers["Content-Type"] =~ /charset/i
       headers["Content-Type"] ||= ""
       headers["Content-Type"] += "; charset=utf-8"
     end
   end
   ```

International Characters (continued)

5. Add encoding information to your templates. Even if you are transferring documents using Unicode, if a user saves them to his or her local hard disk, there needs to be some way of identifying the encoding. Prepend the following inside your layout's <*head*> section:

```
<![[CDATA
<meta http-equiv="Content-Type"
   content="text/html; charset=utf-8">
]]>
```

As long and drawn out as this procedure may seem, it's still not a total solution. There are major fixes in the works for Ruby 2.0, but for now internationalization is a difficult area for Ruby. There are several efforts underway to work around this. At the time of this writing they are all still experimental. Watch the Rails mailing list for announcements.

Congratulations, you're ready to start translating!

مبروك! أنت مستعد!

Now, digging into our little grocery store application, we'll turn our attention to setting the user's language/locale. We'll allow the user to do this in two ways:

- Set it when the user logs in (for users who log in before browsing our products)
- Allow users who are not logged in to manually set it (for users who want to browse first, creating or logging into an account right before checking out)

We'll use these two techniques to set a session variable and have a before_filter() that will call Locale.set() for each request. This will involve adding code to AccountsController (our controller that handles authentication) to set the session variable and adding a before_filter in ApplicationController to use Locale.set() to set the locale for each request.

Here's AccountsController; all we do here is set the session variable in login and the manual change_locale actions:

Download Globalize/app/controllers/accounts_controller.rb

```
class AccountsController < ApplicationController

  def login
    case request.method
    when :post
      begin
        user = User.authenticate(@params[:email], @params[:password])
        session[:user] = user.id
        session[:locale] = user.locale
        go_to = session[:destination]
        session[:destination] = nil
        redirect_to (go_to || home_url) unless performed?
      rescue User::InvalidLoginException => e
        flash[:notice] = e.message
        redirect_to login_url unless performed?
      end
    when :get
    end
  end

  def logout
    @session[:user] = nil
    redirect_to home_url
  end

  def change_locale
    session[:locale] = params[:locale] unless params[:locale].blank?
    redirect_to :back
  end

end
```

Nothing special appears there. In login we just use the stored locale value for the user, and in change_locale we use a CGI parameter. Now let's look at our before_filter in ApplicationController that will handle actually setting the locale during each request:

```
before_filter :set_locale

def set_locale
  Locale.set session[:locale] unless session[:locale].blank?
  true
end
```

So, we have a working system that can handle the selection of a locale. This is where things start to get fun.

Probably the easiest way to preview the usefulness of localization is in views; we'll get into models a bit later.

Globalize gives you a few easy-to-use string methods to handle view translations; translate() and t() (which is just an alias) are for simple translations, and /() is for printf-looking functionality. Here are a few examples from our application:

```
<% unless params[:search].blank? %>
  <p><%= "Found %d products." / @products.size %></p>
<% end %>
```

For quick, easy, little bits of translation, use the t() method. We use that on the page where customers can manage their order:

```
<%= link_to "Remove".t, :action => 'remove', :id => item.product_id %>
```

It turns out String#/() is really just syntactic sugar for String#translate() with a few arguments preset. Refer to Globalize's core_ext.rb for details.

Offering locale-friendly versions of dates and currency is also simply done, courtesy of the loc() (localize()) method:

```
<%= Time.now.loc "%H:%M %Z" %>
```

So, it turns out translating views is really easy. Since the translations themselves are stored in globalize_translations, it's just a matter of throwing up some scaffolding to edit them.

Now, in our little grocery app, the majority of what we're going to be displaying will be model data: our products. For this to really work as a truly international app, we'll have to be able to translate attributes on the model as well. It won't do to have "Place Order" in 25 languages if the only way to figure out what you're buying is by looking at pictures.

Good news—this is where Globalize really shines. Let's take a quick look at our Product model for an example:

```
class Product < ActiveRecord::Base
  translates :name, :description
end
```

The translates() method call lets Globalize know it will be handling translation of the name() and description() attributes. Now let's look at how you save a model with multiple translations by adding a new product:

```
Locale.set 'en-US'
prod = Product.create(:name => "Yemenese Coffee",
                      :description => "Coffee from the South of Yemen")
Locale.set 'ar-LB'
prod.reload
prod.name = "قهوة يمنية"
prod.description = "قهوة من جنوب اليمن"

prod.save
```

As you can see, Locale.set is very important. When a model's attributes are handled by Globalize, Globalize interprets any assignments to those attributes as their translation in the current locale. Globalize makes the process easy by handling the details behind the scenes using the globalize_translations table and some creative overriding of some of ActiveRecord::Base's internals (such as find()); this is a detail that you'll need to keep in mind if you're using find_by_sql(), which it doesn't override.

The locale name given to Locale.set consists of a language code (from globalize_languages) and a country code (from globalize_countries). This is nice, but as translations are stored by language, not locale, if we wanted a specific translation for Canadian English, for instance, a new language row would need to be added to the globalize_languages table.

Localization is fun stuff—it can seem a little complex, but with Globalize, it's easily manageable and simple to get running.

Discussion

Not all languages are read from left to right! Be kind to languages such as Arabic and Hebrew and support right-handed page layouts (hint: load another style sheet by checking Locale.active.language.direction() to change text alignment, and maybe even place labels for form fields on the left or right hand side depending on direction).

We certainly haven't touched on everything relating to Globalize; it has features such as support for pluralization, routing to locale-specific templates, and the Currency class that we haven't even looked at here. Globalize is just chock-full of goodies, so check it out—this was just an appetizer!

Also See

The Globalize website[29] has more background on the plugin, including a FAQ, examples, and information on more complex topics.

[29] http://www.globalize-rails.org

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help programmers stay on top of their game.

Visit Us Online

Rails Recipes Home Page
`pragmaticprogrammer.com/titles/fr_rr`
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
`pragmaticprogrammer.com/updates`
Be notified when updates and new books become available.

Join the Community
`pragmaticprogrammer.com/community`
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
`pragmaticprogrammer.com/news`
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragmaticprogrammer.com/titles/fr_rr.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	support@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com