Localization

Introduction

Laravel's localization features provide a convenient way to retrieve strings in various languages, allowing you to easily support multiple languages within your application. Language strings are stored in files within the `resources/lang` directory. Within this directory there should be a subdirectory for each language supported by the application:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files return an array of keyed strings. For example:

```
<?php

return [
    'welcome' => 'Welcome to our application'
];
```

Configuring The Locale

The default language for your application is stored in the `config/app.php` configuration file. You may modify this value to suit the needs of your application. You may also change the active language at runtime using the `setLocale` method on the `App` facade:

```
Route::get('welcome/{locale}', function ($locale) {
    App::setLocale($locale);

    //
});
```

You may configure a "fallback language", which will be used when the active language does not contain a given translation string. Like the default language, the fallback language is also configured in the `config/app.php` configuration file:

```
'fallback_locale' => 'en',
```

Determining The Current Locale

You may use the `getLocale` and `isLocale` methods on the `App` facade to determine the current locale or check if the locale is a given value:

```
$locale = App::getLocale();

if (App::isLocale('en')) {
    //
}
```

Defining Translation Strings

Using Short Keys

Typically, translation strings are stored in files within the `resources/lang` directory. Within this directory there should be a subdirectory for each language supported by

the application:

```
/resources
    /lang
        /en
            messages.php
        /es
            messages.php
```

All language files return an array of keyed strings. For example:

```
<?php

// resources/lang/en/messages.php

return [
    'welcome' => 'Welcome to our application'
];
```

Using Translation Strings As Keys

For applications with heavy translation requirements, defining every string with a "short key" can become quickly confusing when referencing them in your views. For this reason, Laravel also provides support for defining translation strings using the "default" translation of the string as the key.

Translation files that use translation strings as keys are stored as JSON files in the `resources/lang` directory. For example, if your application has a Spanish translation, you should create a `resources/lang/es.json` file:

```
{
    "I love programming.": "Me encanta programar."
}
```

Retrieving Translation Strings

You may retrieve lines from language files using the `__` helper function. The `__` method accepts the file and key of the translation string as its first argument. For example, let's retrieve the `welcome` translation string from the `resources/lang/messages.php` language file:

```
echo __('messages.welcome');

echo __('I love programming.');
```

If you are using the Blade templating engine, you may use the `{{ }}` syntax to echo the translation string or use the `@lang` directive:

```
{{ __('messages.welcome') }}

@lang('messages.welcome')
```

If the specified translation string does not exist, the `__` function will return the translation string key. So, using the example above, the `__` function would return `messages.welcome` if the translation string does not exist.

> The `@lang` directive does not escape any output. You are **fully responsible** for escaping your own output when using this directive.

Replacing Parameters In Translation Strings

If you wish, you may define placeholders in your translation strings. All placeholders are prefixed with a `:`. For example, you may define a welcome message with a placeholder name:

```
'welcome' => 'Welcome, :name',
```

To replace the placeholders when retrieving a translation string, pass an array of replacements as the second argument to the `__` function:

```
echo __('messages.welcome', ['name' => 'dayle']);
```

If your placeholder contains all capital letters, or only has its first letter capitalized, the translated value will be capitalized accordingly:

```
'welcome' => 'Welcome, :NAME', // Welcome, DAYLE
'goodbye' => 'Goodbye, :Name', // Goodbye, Dayle
```

Pluralization

Pluralization is a complex problem, as different languages have a variety of complex rules for pluralization. By using a "pipe" character, you may distinguish singular and plural forms of a string:

```
'apples' => 'There is one apple|There are many apples',
```

You may even create more complex pluralization rules which specify translation strings for multiple number ranges:

```
'apples' => '{0} There are none|[1,19] There are some|[20,*] There are many',
```

After defining a translation string that has pluralization options, you may use the `trans_choice` function to retrieve the line for a given "count". In this example, since the count is greater than one, the plural form of the translation string is returned:

```
echo trans_choice('messages.apples', 10);
```

You may also define placeholder attributes in pluralization strings. These placeholders may be replaced by passing an array as the third argument to the `trans_choice` function:

```
'minutes_ago' => '{1} :value minute ago|[2,*] :value minutes ago',

echo trans_choice('time.minutes_ago', 5, ['value' => 5]);
```

If you would like to display the integer value that was passed to the `trans_choice` function, you may use the `:count` placeholder:

```
'apples' => '{0} There are none|{1} There is one|[2,*] There are :count',
```

Overriding Package Language Files

Some packages may ship with their own language files. Instead of changing the package's core files to tweak these lines, you may override them by placing files in the `resources/lang/vendor/{package}/{locale}` directory.

So, for example, if you need to override the English translation strings in `messages.php` for a package named `skyrim/hearthfire`, you should place a language file at: `resources/lang/vendor/hearthfire/en/messages.php`. Within this file, you should only define the translation strings you wish to override. Any translation strings you don't override will still be loaded from the package's original language files.

Become a Laravel Partner

Laravel Partners are elite shops providing top-notch Laravel development and consulting. Each of our partners can help you craft a beautiful, well-architected project.

Our Partners

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.