Feature Toggle Documentation

Release 0.2.0

Joshua Estes

February 16, 2015

Contents

This library allows you to easily add and modify various features to your code while in development. Please read the information below on instructions on how to use this library as well as how to customize and add to it for your own needs.

CHAPTER 1

Contents

1.1 Installation

Installation should be done with composer by adding this library to your `composer.json` file

```
"require": {
    "joshuaestes/feature-toggle": "~0.2"
}
```

You can also add this to your `composer.json` file using composer's require command like so

```
php composer.phar require "joshuaestes/feature-toggle:~0.2"
```

If you want to get the latest unstable release, use *@dev* when defining the requirement in your composer.json file.

```
"require": {
    "joshuaestes/feature-toggle": "~0.3@dev"
}
```

1.2 Usage

Using this library is fairly simple.

```
<?php

use JoshuaEstes\Component\FeatureToggle\FeatureBuilder;

$feature = FeatureBuilder::create('enable_a_cool_new_feature')
    ->getFeature();

if ($feature->isEnabled()) {
    // code for when the feature is enabled
} else {
    // code for when the feature is disabled
}
```

By default the feature is disabled. You will need to enabled the feature. You are able to do this two different ways.

```
<?php

use JoshuaEstes\Component\FeatureToggle\Toggle\FeatureToggleGeneric;
```

```
$feature = FeatureBuilder::create('enable_a_cool_new_feature')
    ->setFeatureToggle(
        new FeatureToggleGeneric(
            array(
                'enabled' => true
            )
        )
    )
    ->getFeature();
```

This will now enabled the feature, when you call *isEnabled()* it will return *true*. The other way to enable a feature is like so:

```
<?php

$feature = FeatureBuilder::create('enable_a_cool_new_feature')
    ->getFeature();

$feature->setFeatureToggle(
    new FeatureToggleGeneric(
        array(
            'enabled' => true
        )
    )
);
```

1.3 Features

A Feature is either enabled or disabled. It is enabled/disabled by the use of a toggle.

1.3.1 Feature Container

The feature container is used to put all your features into one place where you can easily loop through them.

```
<?php
use JoshuaEstes\Component\FeatureToggle\FeatureContainer;
use JoshuaEstes\Component\FeatureToggle\Feature;

$container = new FeatureContainer();
$feature   = FeatureBuilder::create('enable_a_cool_new_feature')
    ->getFeature();

$container->addFeature($feature);

$coolNewFeature = $container->getFeature('enable_a_cool_new_feature');

$thisIsNull = $container->getFeature('does_not_compute');

$thisIsFalse = $container->hasFeature('enable_that_sweet_new_feature');
$thisIsTrue  = $container->hasFeature('enable_a_cool_new_feature');

$numberOfFeatures = count($container);

foreach ($container as $f) {
```

```
    // @var FeatureInterface $f
    var_dump($f->isEnabled());
}
```

You can read the source code for more methods that you can call, such as *removeFeature* and *clearFeatures*.

1.3.2 Creating Custom Features

All features must implement the FeatureInterface.

In most situations you will only need to use the default Feature, however in some situations you might want to create your own.

1.4 Toggles

Toggles are reusable code snippets that enable/disable a feature. A toggle can be created to enable a feature by an IP address or something as simple as a configuration value.

A toggle is also passed an array that is used to configure the toggle.

1.4.1 Creating Custom Toggles

All toggles must implement the FeatureToggleInterface.

By creating a custom toggle, you can change the logic for figuring out if a feature is enable or not. Some ideas for custom toggles include:

- IP Based, can enable a feature if the user is on an internal network.
- Username, or something similar.
- Collection, a collection of toggles where it checks for any or all to be enable.
- Gradual, where you can release a feature to x% of a user base.

Creating a custom toggle based on username

You can create a custom feature toggle with ease.

```
<?php
use Symfony\Component\Security\Core\User\UserInterface;
use JoshuaEstes\Component\FeatureToggle\Toggle\FeatureToggle;
use JoshuaEstes\Component\FeatureToggle\FeatureInterface;

class FeatureToggleUsername extends FeatureToggle
{

    /**
     * @var UserInterface
     */
    protected $user;

    /**
     * Dependency injection
     *
```

```
     * @param UserInterface $user
     */
    public function setUser(UserInterface $user)
    {
        $this->user = $user;
    }

    /**
     * Used to set the options that are allowed to be used with this toggle
     *
     * @param OptionsResolverInterface $resolver
     */
    protected function setDefaultOptions(OptionsResolverInterface $resolver)
    {
        $resolver->setRequired(
            array(
                'username'
            )
        );
    }

    /**
     * Check some settings and return true if the feature should be enabled
     *
     * @param FeatureInterface $feature
     */
    public function isEnabled(FeatureInterface $feature)
    {
        return $this->options['username'] == $this->user->getUsername();
    }
}
```

Now that we have the toggle, we just need to create the toggle and assign it to a feature object.

```
<?php
use JoshuaEstes\Component\FeatureToggle\FeatureBuilder;

$toggle = new FeatureToggleUsername(
    array(
        'username' => 'joshua',
    )
);
$toggle->setUser($user);

$feature = FeatureBuilder::create('enable_for_joshua')
    ->setFeatureToggle($toggle)
    ->getFeature();
```

That's all there is to it! Note that the *$user* variable needs to be defined and must have a method *getUsername*. This feature will return true only for the user with the username *joshua* and will return false for all other users.

1.5 Development

This project uses semantic versioning.

1.6 Testing

This code is tested using Travis CI. You can run the build yourself which will lint check all code and run PHPUnit.

```
php bin/phing -f build/build.xml
```

1.7 License