CAKEDC PLUGIN STANDARD

The CakeDC Plugin Standard is a blueprint design and guideline for creating coherent and maintainable plugins for the CakePHP framework.

The standard itself covers the following areas:

- Development: Contribution to plugins should be normalized, both at the process level as well as for integration.
- Versioning: Versions need to specify their version state, as well as their target framework code base.
- Contributing: Clarifying contribution guidelines is key to allowing developers to self organize.
- Documentation: Documentation for plugins should be consistent in it's structure and content, so the necessary information is available and the format is common for easy consumption.
- Roadmap: Laying out the direction is important for other developers to align themselves with the proposed objectives.
- Changelogs: Keeping a clear track of changes made helps everyone know what was fixed, added or removed, and when.
- Licensing: Although not a confusing topic, it's important you make the right choices about how people use your code before they actually do.

While this is defined as a standard, it's only proposed as a guideline for those who wish to adopt this design. The intention is to set a base from which everyone can build upon.

DEVELOPMENT

As plugins are used in live applications it's important to focus the development of the code base towards quality and accountability. Only then will the process of developing and maintaining the plugin promote confidence and stability.

Branches

The **master** branch of the plugin repository holds the latest STABLE version of the code, for the latest supported version of the CakePHP framework, and is used for applying bug fixes.

The **develop** branch then serves as integration to the master branch. This branch is considered UNSTABLE, and is used primarily to test and review patches and new features for the latest version of the plugin before release.

Previous releases of the framework which are supported by the plugin also have integration branches, in the form of **version** branches. For example, if the most recent supported version of the framework is 2.4, there may also be 2.3 and 2.2 branches.

```
$ git branch -a
    * master
        remotes/origin/HEAD -> origin/master
        remotes/origin/develop
        remotes/origin/2.3
        remotes/origin/2.2
```

This allows the plugin to support features sets which may vary from later versions of the framework. Only the major and minor versions of the framework are contemplated as version jumps.

When a new version of the framework is released, the current develop branch becomes a version branch, and the develop and master branches now target the latest version of the framework. IMPORTANT: When using the plugin in your application, always clone a tag for a specific release, and not the master or develop branches directly.

Features

All new features or modifications to base functionality occur as **feature** branches from **develop** on a forked repository. To create a **feature** branch, name your new branch with the format "feature/", followed by a dash ("-") separated description of your change, for example:

```
$ git checkout -b feature/changing-something develop
```

These branches are created and then merged as pull requests to the develop branch of the main plugin repository. No feature branches should be created on the main plugin repository (if you have direct access), favoring instead to always fork the repository first, and then create features there.

Issues

Bug fixes or any resolution of broken functionality occur as issue branches from master on a forked repository. To create an issue branch, name your new branch with the format "issue/", followed by a dash ("-") separated description of your patch, for example:

```
$ git checkout -b issue/fixing-something master
```

Also, if you're basing your changes on an issue which is currently open, reference the issue number at the beginning of your description, for example:

```
$ git checkout -b issue/1234-fixing-that-thing master
```

These branches are created and then merged as pull requests to the master branch of the main plugin repository. No issue branches should be created on the main plugin repository (if you have direct access), favoring instead to always fork the repository first, and then create issues there.

VERSIONING

Every time a new release of the plugin occurs a tag is created with it's corresponding version. The format of these versions is the major and minor version of the framework, followed by the semantic versioning of the plugin release.

For example, a tag for version 1.2.0 of the plugin, aimed at version 2.4 of the framework, would be named 2.4/1.2.0.

First, make sure to merge the develop branch into master, to keep the latest version of the plugin clearly visible as the most relevant version of the code to use.

```
$ git checkout master
$ git merge --no-ff develop
$ git push origin master
```

Then, to create a tag for a release, first make sure you're tagging from the master branch for the latest release, or the version branch for a legacy release, for example:

```
$ git checkout master $ git tag -a 2.4/1.2.0 $ git checkout 2.3 $ git tag -
a 2.3/1.2.0
```

Then, push the new tags to the repository.

```
$ git push origin --tags
```

New features are generally not considered for previous versions of the framework, and are only merged to previous versions if they are fully backwards compatible. However, all versions should be updated with security patches.

CONTRIBUTING

In order to contribute new features, enhancements or bug fixes to a plugin simply follow these steps:

- Create a GitHub account, if you don't own one already.
- Then, fork the plugin repository to your account.
- Create a new branch from the develop branch in your forked repository.
- Modify the existing code, or add new code to your branch, making sure you follow the CakePHP Conventions and Coding Standards.
- Modify or add unit tests which confirm the correct functionality of your code (requires PHPUnit 3.5+).
- Consider using the CakePHP Code Sniffer to check the quality of your code.
- When ready, make a pull request to the develop branch of the plugin repository.

There may be some discussion regarding your contribution to the repository before any code is merged in, so be prepared to provide feedback on your contribution if required.

DOCUMENTATION

Code is awesome, but without a firm understand of how to implement a plugin developers can sometimes waste valuable time trying to work out the intended usage. The following define some ways to help people help themselves

Readme

The README is possibly the most important part of the documentation, as it's usually the first point of reference. Therefore, it's important that from this location developers can find everything they need related to the plugin.

The README is typically separated into various sections, in a common order:

- Description: A description of the plugin and an overview of it's features.
- Requirements: Any requirements for the plugin, including PHP and CakePHP versions, as well as any third party dependencies.
- Documentation: The documentation for the plugin, or instructions on where to find support material, roadmaps, API docs, etc.
- Support: How people should request for support, and the preferred channels provisioned by the maintainers.
- Contributing: Specific details on how to contribute to the code, and any strategy used for integration.
- License: The license information for your plugin, as well as any specific copyright information.

Docs

If you have a large amount of documentation it can help to organize it as separate files. To do so, create a directory named "Docs" in the root of your plugin repository, then add a file named "Home.md". If you don't want to use markdown for you syntax then simply change the extension to your preferred format, for example:

```
/Docs/Home.md
```

Now, in your "Home" file, add an index to the sections which form part of your documentation. Here are a few sections you could consider including:

- Overview: A detailed description of the plugin and the functionality it provides.
- Installation: Step by step instructions for installing the plugin and any dependencies it may require (check out Composer to improve this process).
- Configuration: A list of the configuration options available and their usage.
- Getting Started: An introduction to using the plugin's features and integrating it into your application's code base.
- Examples: A snippet of code is worth a thousand words, so try and provide as many use cases as possible.
- Tutorials: Any topics which may require additional explanation, as well as key topics, such as an initial Hello World.

If needed, you may want to create a number of files under a common section, for example, "Tutorials". When doing so, create the section with a file of the same name, which serves as an index for the linked files.

```
/Docs/Tutorials.md
```

Then, simply create the files under a directory with the section's name.

```
/Docs/Tutorials/Hello-World.md
```

The files themselves can then be linked using a relative path, for example:

```
[Hello World](Tutorials/Hello-World.md)
```

File names should match the title of the topic, using dashes ("-") between words, so directory listing allows for easy discovery.

API

Although the technical documentation is accessible from the doc blocks in the code itself it can be useful to host the documentation for the API, using a tool such as http://www.apigen.org for example, similar to the API for CakePHP itself at http://api.cakephp.org.

ROADMAP

For both community involvement and clarity as to the direction of development it's important to maintain a roadmap, which outlines the pending features and desired improvements. If your roadmap is organized into milestones you can easily define your objectives for each iteration, which then allows people to get involved inline with your schedule.

For the items themselves, aim to clearly define what's pending, what's complete, and what's been discarded. See the following markdown syntax as an example:

```
* [ ] Make the thing better by improving that other thing
* [ ] Do that change that everybody wants
* [X] Make the bad thing work correctly (dropped because of X)
```

If the scope of your plugin is extensive, you may want to consider creating an RFC process for others to propose additions to the roadmap. This will help avoid the roadmap suffering from feature creep if it's open to community contribution.

CHANGELOGS

Although **git** provides extensive control over filtering the log for a repository, it can help to have a history covering the changes made and their respective release versions.

To easily generate a change log for a release you can use the following, or a similar command, to extract the CHANGELOG for the latest changes:

```
$ git log 1.1.6..HEAD --no-merges --oneline
```

Depending on the size of your project, you could store the revision history in a single CHANGELOG file, or create an individual file for each release.

LICENSING

Choosing the right license to release your code under is an important decision that shouldn't be overlooked. The CakePHP framework, for example, is released under the MIT license. However, there are many licenses, each with their own specific approach.

Before releasing your plugin, if you're not sure how other people should use your code, take some time to review the licenses available. If you find another license which isn't listed, but are still interested in your code being Open Source, then be sure to check for it's compliance with the OSR.

There's also the option of dual or multi-licensing. This allows for proprietary derivative works to

possibly pay a license fee to use the code, while the plugin remains free for Open Source projects. This can be especially useful if you require funding for the further development of your code, or to cover costs related to the development.