Pelican-CMS Documentation

Ezequiel Leonardo Castaño

Apr 10, 2019

Contents

This Section is UNDER CONSTRUCTION

CHAPTER 1

Installation

This Section is UNDER CONSTRUCTION

CHAPTER 2

Getting Started

This Section is UNDER CONSTRUCTION

CHAPTER 3

Settings

This Section is UNDER CONSTRUCTION

CHAPTER 4

License

This Section is UNDER CONSTRUCTION

CHAPTER 5

Contributing

5.1 Setup

5.1.1 Fork on GitHub

Before you do anything else, login/signup on GitHub and fork Pelican-CMS from the GitHub project.

5.1.2 Clone your fork locally

If you have git-scm installed, you now clone your git repo using the following command-line argument where <my-github-name> is your account name on GitHub:

```
git clone git@github.com:<my-github-name>/pelican-cms.git
```

5.1.3 Installing Pelican CMS

Follow our detailed *Installation* instructions. Please record any difficulties you have and share them with the Pelican CMS community via our issue tracker.

5.2 Issues!

Pick an unassigned issue that you think you can accomplish, add a comment that you are attempting to do it, and shortly your own personal label matching your GitHub ID will be assigned to that issue.

Feel free to propose issues that aren't described!

5.2.1 Tips

#. **starter** labeled issues are deemed to be good low-hanging fruit for newcomers to the project or even Python. #. **doc** labeled issues must only touch content in the docs folder.

5.3 Setting up topic branches and generating pull requests

While it's handy to provide useful code snippets in an issue, it is better for you as a developer to submit pull requests. By submitting pull request your contribution to Pelican CMS will be recorded by Github.

In git it is best to isolate each topic or feature into a "topic branch". While individual commits allow you control over how small individual changes are made to the code, branches are a great way to group a set of commits all related to one feature together, or to isolate different efforts when you might be working on multiple topics at the same time.

While it takes some experience to get the right feel about how to break up commits, a topic branch should be limited in scope to a single `issue` as submitted to an issue tracker.

Also since GitHub pegs and syncs a pull request to a specific branch, it is the **ONLY** way that you can submit more than one fix at a time. If you submit a pull from your develop branch, you can't make any more commits to your develop without those getting added to the pull.

To create a topic branch, its easiest to use the convenient `-b` argument to `git checkout`:

```
git checkout -b fix-broken-thing
Switched to a new branch 'fix-broken-thing'
```

You should use a verbose enough name for your branch so it is clear what it is about. Now you can commit your changes and regularly merge in the upstream develop as described below.

When you are ready to generate a pull request, either for preliminary review, or for consideration of merging into the project you must first push your local topic branch back up to GitHub:

```
git push origin fix-broken-thing
```

Now when you go to your fork on GitHub, you will see this branch listed under the "Source" tab where it says "Switch Branches". Go ahead and select your topic branch from this list, and then click the "Pull request" button.

Here you can add a comment about your branch. If this in response to a submitted issue, it is good to put a link to that issue in this initial comment. The repo managers will be notified of your pull request and it will be reviewed (see below for best practices). Note that you can continue to add commits to your topic branch (and push them up to GitHub) either if you see something that needs changing, or in response to a reviewer's comments. If a reviewer asks for changes, you do not need to close the pull and reissue it after making changes. Just make the changes locally, push them to GitHub, then add a comment to the discussion section of the pull request.

5.4 Pull upstream changes into your fork regularly

Pelican CMS is still in development and therefore, it is critical that you pull upstream changes from develop into your fork on a regular basis. Nothing is worse than putting in a days of hard work into a pull request only to have it rejected because it has diverged too far from develop.

To pull in upstream changes:

```
git remote add upstream https://github.com/ELC/pelican-cms.git
git fetch upstream develop
```

Check the log to be sure that you actually want the changes, before merging:

```
git log upstream/develop
```

Then merge the changes that you fetched:

```
git merge upstream/develop
```

For more info, see http://help.github.com/fork-a-repo/

5.5 How to get your pull request accepted

We want your submission. But we also want to provide a stable experience for our users and the community. Follow these rules and you should succeed without a problem!

5.5.1 Run the tests!

Before you submit a pull request, please run the entire Pelican CMS test suite via:

```
pytest tests/
```

The first thing the core committers will do is run this command. Any pull request that fails this test suite will be **rejected**.

5.5.2 If you add code/views you need to add tests!

We've learned the hard way that code without tests is undependable. If your pull request reduces our test coverage because it lacks tests then it will be **rejected**.

For now, we use the Pytest Test framework.

Also, keep your tests as simple as possible. Complex tests end up requiring their own tests. We would rather see duplicated assertions across test methods then cunning utility methods that magically determine which assertions are needed at a particular stage. Remember: *Explicit is better than implicit*.

5.5.3 Don't mix code changes with whitespace cleanup

If you change two lines of code and correct 200 lines of whitespace issues in a file the diff on that pull request is functionally unreadable and will be **rejected**. Whitespace cleanups need to be in their own pull request.

5.5.4 Keep your pull requests limited to a single issue

Pelican CMS pull requests should be as small/atomic as possible. Large, wide-sweeping changes in a pull request will be **rejected**, with comments to isolate the specific code in your pull request. Some examples:

#. If you are making spelling corrections in the docs, don't modify the settings.py file. #. If you are adding a new view don't '*cleanup*' unrelated views. That cleanup belongs in another pull request. #. Changing permissions on a file should be in its own pull request with explicit reasons why.

5.5.5 Follow PEP-8 and keep your code simple!

Memorize the Zen of Python:

```
>>> python -c 'import this'
```

Please keep your code as clean and straightforward as possible. When we see more than one or two functions/methods starting with *_my_special_function* or things like *__builtins__.object = str* we start to get worried. Rather than try and figure out your brilliant work we'll just **reject** it and send along a request for simplification.

Furthermore, the pixel shortage is over. We want to see:

- *package* instead of *pkg*
- *grid* instead of *g*
- *my_function_that_does_things* instead of *mftdt*

5.5.6 Test any css/layout changes in multiple browsers

Any css/layout changes need to be tested in Chrome, Firefox, Safari and Edge across Mac, Linux, and Windows. If it fails on any of those browsers your pull request will be **rejected** with a note explaining which browsers are not working.

5.6 How pull requests are checked, tested, and done

This section is under construction

CHAPTER 6

FAQ

This Section is UNDER CONSTRUCTION

CHAPTER 7

Changelog

This Section is UNDER CONSTRUCTION

- #73: (via #77) Integrate Invoke to automatically build and serve the documentation built by Sphinx.

```
def f(a, b):
    print("hola", a, b)
```