Montage Developer Guide

Release 1.0-dev

Derek Payton

Aug 18, 2016

Contents

This guide outlines how we write code at Montage, covering topics such as coding style and git workflow.

Read it. Know it. Use it. Thanks!

CHAPTER 1

Contents

1.1 Coding style

> **Reability counts!**
>
> Programs must be written for people to read, and only incidentally for machines to execute.
>
> – Abelson & Sussman, *Structure and Interpretation of Computer Programs*

Please follow these guidelines when writing new code.

1.1.1 General guidelines

These are our general guidelines to be used throughout the codebase. Not all of these will apply to every language we use, but they cover a lot of the overlap.

Whitespace

- Indentation should use spaces, not tabs
- No extra spacing between arguments and expressions
- Delete all trailing whitespace
- End files with a trailing new line

Braces

- One True Brace Style
- Never omit braces for single-statement blocks

EditorConfig

EditorConfig helps define and maintain consistent coding styles. You would do very well to install the plugin for your editor of choice.

```
# http://editorconfig.org

root = true

[*]
indent_style = space
indent_size = 4
insert_final_newline = true
trim_trailing_whitespace = true
end_of_line = lf
charset = utf-8

# Docstrings and comments use max_line_length = 79
[*.py]
max_line_length = 119

# Use 2 spaces for the HTML files
[*.html]
indent_size = 2

# The JSON files contain newlines inconsistently
[*.json]
indent_size = 2
insert_final_newline = ignore

# Minified JavaScript files shouldn't be changed
[**.min.js]
indent_style = ignore
insert_final_newline = ignore

# Makefiles always use tabs for indentation
[Makefile]
indent_style = tab

# Batch files use tabs for indentation
[*.bat]
indent_style = tab
```

The Zen of Python

Also known as PEP 20, these are the guiding principals of Python's design and are generally good advice for any software developer.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
```

```
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.1.2 Python

PEP 8: Style guide for Python code

PEP 8 provides coding conventions for the the Python code in the standard library, and has been widely adopted by the community. All code should be checked for PEP 8 conformity.

http://www.python.org/dev/peps/pep-0008/

Checking code with pycodestyle

You can use pycodestyle to verify that your code conforms to PEP 8 standards:

```
$ pycodestyle montage/
```

Or if you just want an overview of PEP 8 violations:

```
$ pycodestyle montage/ -qq --statistics
```

Django's style guide

The Django project builds on top of PEP 8, and includes a number of useful guidelines.

https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/

The Hitchhiker's Guide to Python

Written by Kenneth Reitz, of requests fame, The Hitchhiker's Guide to Python is an opinionated guide to code style, best practices, and common idioms. It's a good guide on how to write code that is *Pythonic*.

http://docs.python-guide.org/en/latest/

1.1.3 Javascript

TBD, but build from this:

https://github.com/Seravo/js-winning-style

Checking code with ESLint

TBD.

http://eslint.org/

1.2 Git workflow

1.2.1 Commit messages

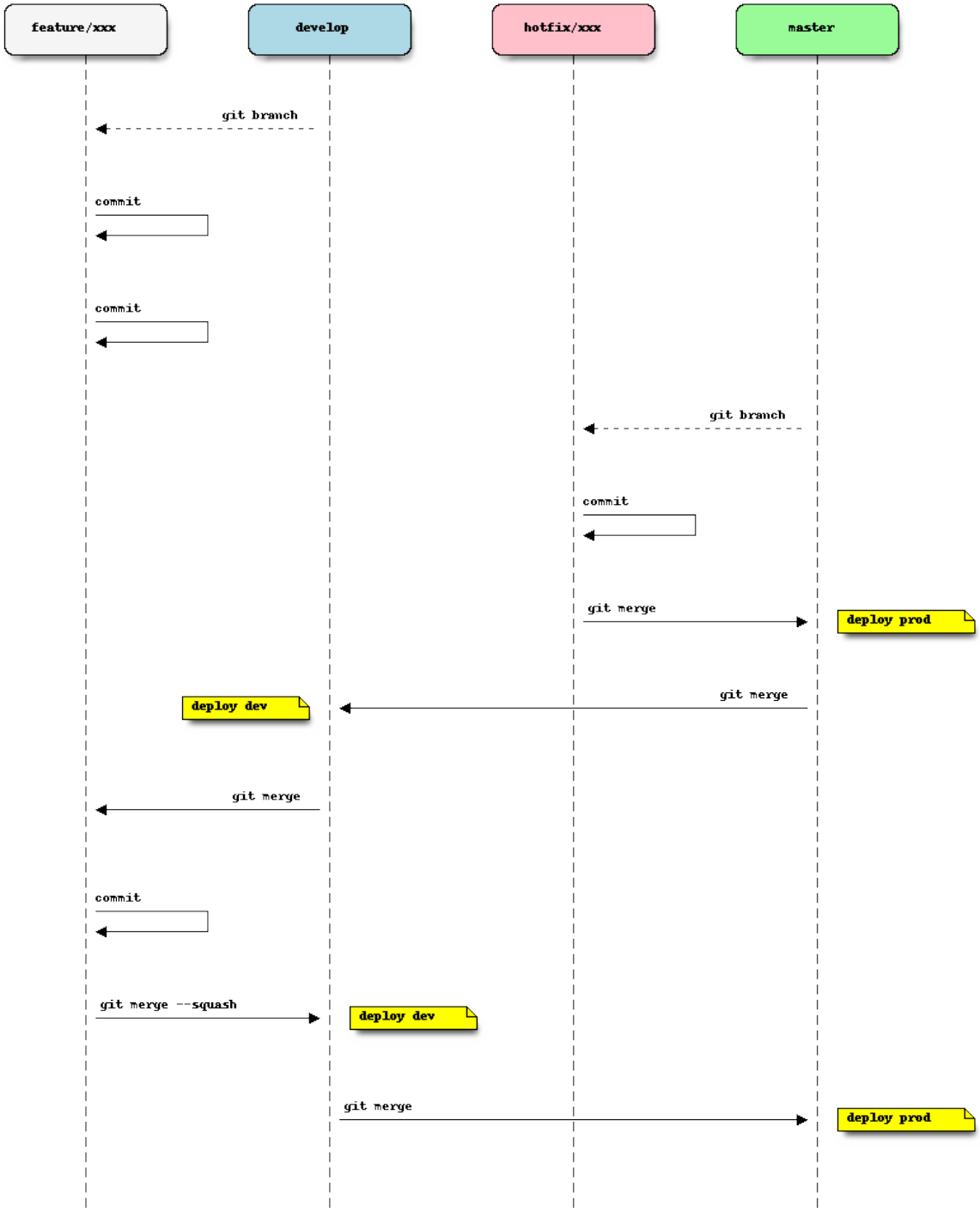
Karma commits.

http://karma-runner.github.io/1.0/dev/git-commit-msg.html

Karma says 70 characters for the first line, but Github wraps at 50. Let's go with 50.

1.2.2 Branching model

Our workflow is based on Vincent Driessen's "successful git branching model", with some minor adjustments. We still utilize `develop`, `master`, and various types of feature branches, however we do not tag releases and there are no release branches.



git-flow is a git plugin that helps facilitate this branching strategy. It's not required, but can help make things a bit easier to manage. There is also a good write up on using git-flow.

Feature branching

All development happens in a feature branch. Feature branches are named as `type/branch-slug`, where `type` can be one of:

- **feature** – New features, both major and minor, in development
- **bugfix** – Fixing a specific bug or regression in the code
- **refactor** – Re-writing or re-architecting parts of the system
- **hotfix** – Patches applied directly to the master branch and deployed to live

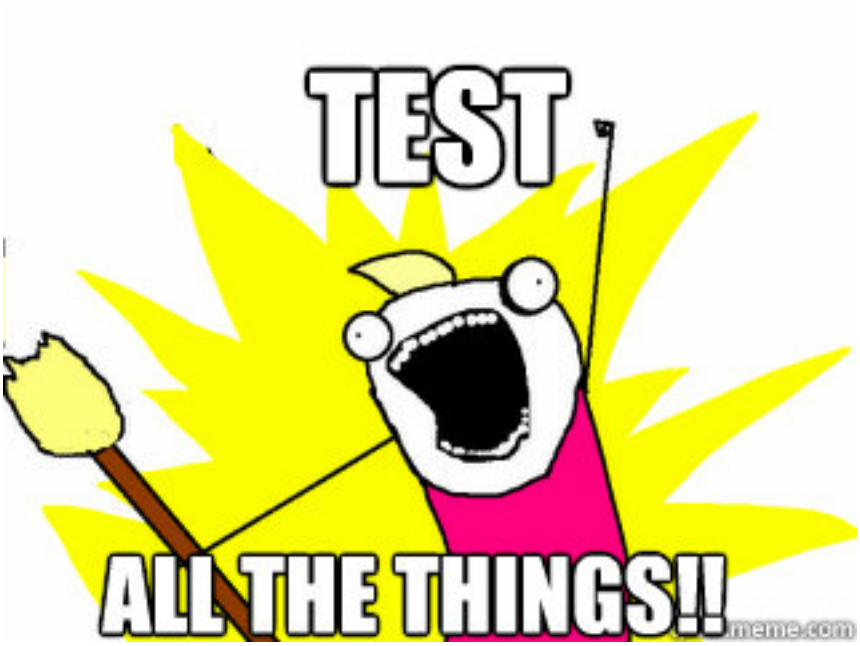
---

Important: Feature branches are not considered complete until they include all of the following:

- **Code** that conforms to our *coding style*.
- **Unit tests** that pass locally and in our CI environment.
- **Documentation** updates, as needed.

When a feature branch has been merged into develop, it is the responsiblity of that branches main developer to ensure it is deleted from Github.

1.3 Testing



1.3.1 Testing in Python

Defining test cases with unittest

Tests cases should be defined with the built-in unittest module. This allows for tests to be easily structured, and related tests can be grouped together.

Test assertions should be made with the `assert` statement, **not** the `assertFoo` methods provided by unittest; it's cleaner and easier to parse, and pytest understands it just fine.

```
import unittest

class YourModuleTests(unittest.TestCase):
    def test_add(self):
        assert 2 + 2 == 4
```

Running tests with pytest

Use pytest to run the tests. It integrates well with unittest, and there are a ton of useful plugins.

There are two plugins that every project should use:

- pytest-pep8 to check code style against PEP 8
- pytest-cov to measure test coverage

When all is said and done, the command to run your tests should look something like this:

```
py.test tests/ --cov your_module --cov-append --cov-report term-missing --pep8
```

Automating tests with tox

tox is a tool that aims to automate and standardize tests. It's especially useful when you need to test code in different environments (i.e., multiple versions of Python).

`tox` is a much simpler command to remember than a long `py.test` command, so it's good to use even if you're only targeting a single environment.

A simple `tox.ini` looks like this:

```
[tox]
envlist = py27,py35

[testenv]
commands=py.test tests/ --cov your_module --cov-append --cov-report term-missing --
 ↪pep8
deps =
    pytest
    pytest-cov
    pytest-pep8
    -rrequirements.txt
```

Now when you run `tox`, you should see something like this:

```
$ tox

... a whole lot of output, including test output and coverage summary ...
```

```
_____________ summary _____________
  py27: commands succeeded
  py35: commands succeeded
  congratulations :)
```

Continuous Integration with CircleCI

Getting tests to run with tox on CircleCI requires the use of tox-pyenv to make different Python versions available.

In this example `circle.yml`, we'll run our tests and upload the coverage results to Codecov:

```
dependencies:
  override:
    - pip install tox tox-pyenv codecov
    - pyenv local 2.7.10 3.5.0

test:
  override:
    - tox
    - codecov
```

1.3.2 Testing in Javascript

TBD.

1.4 Bugs and issues

Please follow these guidelines when creating new issues in Github.

1.4.1 Writing a useful bug report

Useful bug reports are ones that get bugs fixed. A useful bug report normally has two qualities:

1. **Reproducible.** If your bug is not reproducible it will never get fixed. You should clearly mention the steps to reproduce the bug. Do not assume or skip any reproducing step. Described the issue, step-by-step, so that it is easy to reproduce and fix.
2. **Specific.** Do not write a essay about the problem. Be Specific and to the point. Try to summarize the problem in minimum words yet in effective way. Do not combine multiple problems even they seem to be similar. Write different reports for each problem.

Bug report template

```
Environment
===========

* **OS**: Operating System and version (Windows 7, OS X 10.8, Ubuntu 13.04)
* **Browser**: Browser name and version (IE 10, Firefox 27, Google Chrome 30)
```

```
Steps to Reproduce
==================

1. Step-by-step instructions that detail how to reproduce the bug.
2. Don't leave steps out or make any assumptions.
3. If screenshots are called for, provide them here.
4. Make sure these steps reliably reproduce the issue.

Actual Result
=============

Describe what happens when the above steps were followed. If there is any
relevent information in the developer tools window, make note of it here.

Expected Result
===============

Explain what should have happened -- or what you expected to happen -- when
the above steps were followed.

Workaround
==========

If you have a way to work around the problem, describe it here.
```

1.4.2 Effectively organizing tickets

Properly labeling tickets is *essential* to keeping our tracker organized. Some labels stand by themselves, while others are organized into `group:value` pairs.

When creating a new ticker, it should, at the very minimum, have a label from each of these groups:

The `env` group specifies which environment (or site) the issue occurred on.

- `env:dev`
- `env:prod`

The `severity` group specifies how severe the issue is, ranging from trivial to catastrophic. Feature requests also fall under this group.

- `severity:minor` – Minor loss of function, trivial UI problem
- `severity:major` – Major loss of function
- `severity:critical` – Application crash, loss of data
- `severity:blocking` – The issue is preventing further testing or work from being done

1.4.3 Triaging tickets

When a ticket is first created, it is considered *unreviewed*. Unreviewed tickets will be evaluated by a member of the development team or anyone else qualified to make a judgement on the validity of the ticket.

If the ticket is deemed to contain a valid issue or viable feature, the triager should apply the `accepted` label and leave a comment describing the action to be taken or provide any other information necessary to get the issue resolved.

When closing a ticket, you should always leave a comment providing the reason.

D

G