Guidelines

24. März 2014

Inhaltsverzeichnis

1 Coding Styleguide

1.1 Einleitung

Die Python-Community legte schon von Beginn an viel Wert auf Lesbarkeit und Konsistenz von Source Code. Dazu gehört auch ein einheitlicher Code-Stil. Guido van Rossum, der Autor von Python, schrieb deshalb seine Vorstellungen von sauberem Code in einem *Style Guide for Python Code* nieder. Dieser Style Guide wurde im Jahr 2001 als Python Enhancement Proposal 8 – kurz PEP8 – veröffentlicht[1].

[1] `https://python.org/dev/peps/pep-0008/`

Der PEP8 Style Guide hat seit dann beinahe universelle Verbreitung gefunden. Einer der zentralsten Punkte daraus – die Verwendung von 4 Spaces anstelle von Tabs – wird gemäss einem Analysetool[2] in 95% der Python Projekte auf Github so umgesetzt. Im Rahmen dieser Bachelorarbeit werden wir daher auch gemäss diesen Richtlinien arbeiten, mit einigen kleinen Anpassungen.

1.2 Coding Guidelines

PEP8 ist in unserer Software verbindlich, mit folgenden Ausnahmen:

- Maximale Zeilenlänge ist 109 Zeichen, nicht 79. Heutige Bildschirme sind viel grösser, es ergibt keinen Sinn Code umzubrechen um innerhalb der 80-Zeichen-Grenze zu bleiben wenn dadurch der Code weniger gut lesbar wird.
- Folgende Einrückungsregeln in den Code-Checking-Tools können in gewissen Fällen zu schlechter lesbarem Code führen und können deshalb ignoriert werden: *E126*, *E127*, *E128*.

1.3 Tools

1.3.1 Flake8

Flake8 (`https://flake8.readthedocs.org/en/2.0/`) verbindet das Style Checking Tool pep8[3] mit dem Static Code Analysis Tool pyflakes[4]. Für unser Projekt kann folgende Konfiguration (`/.config/flake8`) verwendet werden:

```
[flake8]
ignore = E126,E127,E128
max-line-length = 109
```

1.3.2 Pytest

Zum von uns verwendeten Testing-Framework Pytest[5] gibt es ein PEP8 Plugin. Wird dieses aktiviert, werden Style Guide Violations als Fehlerhafte Tests gewertet. Folgende Konfiguration wird dafür in `pytest.ini` verwendet:

```
[pytest]
addopts = --pep8
pep8ignore =
    *.py E126 E127 E128
    setup.py ALL
    settings.py ALL
    urls.py ALL
```

[2] `http://sideeffect.kr/popularconvention#python`
[3] `https://pypi.python.org/pypi/pep8`
[4] `https://pypi.python.org/pypi/pyflakes`
[5] `http://pytest.org/`

```
    */migrations/* ALL
    */tests/* ALL
pep8maxlinelength = 109
```

2 Definition of Done

Ein Task gilt als abgeschlossen, wenn folgende Punkte erfüllt sind:

- Alle Arbeiten gemäss Task-Beschreibung wurden ausgeführt.
- Der Code wurde auf Github in einen Feature Branch committed.
- Der Code wurde von einem Teammitglied reviewed.
- Der Feature Branch wurde in den Master Branch gemerged.
- Der Code ist sinnvoll kommentiert, Docstrings für Funktionen und Klassen sind vorhanden.
- Flake8 zeigt keine Fehler oder Warnungen.
- Tests existieren wo sinnvoll. Die Testsuite läuft erfolgreich durch.
- Dokumentation wurde nachgeführt.
- Die benötigte Arbeitszeit wurde erfasst.

3 Git Guidelines

Nachfolgend ein paar Regeln zum Umgang mit Git, mit dem Ziel eine möglichst saubere History zu haben.

3.1 Commit-Messages

- ...beginnen mit einem Grossbuchstaben
- ...sind in Englisch verfasst
- ...enthalten keine Typos
- ...enthalten keine Smileys
- ...sind kurz und prägnant
- ...enthalten keine Ticket-Referenzen wie `refs #1` oder `fixes #2`
- ...sind in past tense geschrieben (*fixed bug* statt *fix bug*)

Generell sollte die erste Zeile einer Commit Message in höchstens 72 Zeichen[6] die Änderungen zusammenfassen. Weitere Erläuterungen sollten durch eine Leerzeile getrennt werden.

Beispiel:

```
Capitalized, short (72 chars or less) summary

More detailed explanatory text, if necessary.  Wrap it to about 72
characters or so.  In some contexts, the first line is treated as the
subject of an email and the rest of the text as the body. The blank
line separating the summary from the body is critical (unless you omit
the body entirely); tools like rebase can get confused if you run the
two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, followed by a
  single space, with blank lines in between, but conventions vary here

- Use a hanging indent
```

3.2 History

3.2.1 Rewriting / Reordering / Squashing

Die Git History sollte sauber gehalten werden. Während der Entwicklung ist es kein Problem wenn man viele und häufige Commits macht, aber vor einem Merge/Rebase in den Hauptcode sollten die Commits sinnvoll reduziert (squashed) werden.

Alles zum verändern der Git History findet sich hier: `http://git-scm.com/book/en/Git-Tools-Rewriting-History` Pflichtlektüre!

3.2.2 Pulling

Beim `git pull` ist es sinnvoll, immer den `--rebase` Parameter zu verwenden. Bei einem Konflikt durch neue Remote Commits wird dann nämlich lokal rebased statt merged. Da der lokale Code noch nicht publiziert wurde, ist dies unbedenklich. Ein Rebase-Konflikt kann bei Problemen jederzeit mit `git rebase --abort` abgebrochen werden.

3.2.3 Force Pushing

Während der Entwicklungsphase in einem Branch ist es kein Problem wenn man geänderte History mit `git push --force` pushed, im `master` Branch sollte das

[6] `http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html`

jedoch nur in äussersten Ausnahmesituationen geschehen.

3.2.4 Merge / Rebase

Ist ein Pull Request abgeschlossen, sollte er vor einem Merge gegen den `master`-Branch rebased werden. Dies verhindert Konflikte und ermöglicht ein *fast-forward merge*, wodurch kein Merge Commit entsteht:

```
git checkout master
git pull --rebase
git checkout <featurebranch>
git rebase master
# fix potential conflicts
git push --force origin <featurebranch>
```

Danach sollte der Pull Request sofort gemerged werden:

```
git checkout master
git merge --ff-only <featurebranch>
git push
```

4 Workflow

4.1 Ablauf

Nachfolgend der Workflow einer Code-Änderung:

1. Ticket wird im Github Issue Tracker erstellt und jemandem zugewiesen.
2. Der zuständige Entwickler erstellt einen Feature Branch und entwickelt darin den benötigten Code.
3. Wenn der entwickelte Code sich im strongTNC Repository befindet, kann der Issue in ein Pull Request umgewandelt werden. Ansonsten einen separaten Pull Request erstellen und darin auf den Issue verweisen (`refs user/repo#issue`).
4. Code wird von jemandem reviewed. Korrekturen werden in den Branch pushed.
5. Commits werden wenn sinnvoll reduziert.
6. Rebase des Branches gegen `master`.
7. Merge in `master` via Kommandozeile.
8. Ticket mit Referenz auf relevante Commits schliessen.

4.2 Umwandeln von Issues in Pull Requests

Mit dem Kommandozeilen-Tool hub[7] kann ein Issue in ein Pull Request umgewandelt werden:

```
git checkout <featurebranch>
git push origin <featurebranch>
hub pull-request -b master -i <issue-number> -h <featurebranch>
```

[7] https://github.com/github/hub