Abilian Developer Guide Documentation

Release 2023.01

Stefane Fermigier

2023-01-02 10:44:54.076515

Contents

1 Introduction, vision and principles **3**
- 1.1 About this document . . . 3
- 1.2 Vision . . . 4
- 1.3 Principles . . . 5

2 Getting started **7**
- 2.1 Developer laptop set-up . . . 7

3 Process **9**
- 3.1 Prioritizing and scheduling tasks . . . 9
- 3.2 Git . . . 10
- 3.3 Versionning and releasing . . . 11

4 Software Design and Architecture **13**
- 4.1 “Clean code” . . . 13
- 4.2 The “4 principles of simple design” . . . 13
- 4.3 SOLID & GRASP . . . 13
- 4.4 Domain Driven Design . . . 14
- 4.5 Hexagonal Architecture . . . 15
- 4.6 Test Driven Development . . . 16
- 4.7 API design . . . 17
- 4.8 Other topics . . . 17
- 4.9 Books . . . 17

5 Modeling **19**
- 5.1 Data modeling . . . 19
- 5.2 Object modeling . . . 19
- 5.3 Business process modeling . . . 19

6 Back-end **21**
- 6.1 Intro . . . 21
- 6.2 Links . . . 21

7 Python specific recommendations **23**
- 7.1 Style guide & tools . . . 23
- 7.2 Python 2 vs. Python 3 . . . 23
- 7.3 General Python recommandations . . . 24

7.4 Python tooling . . . 24
7.5 Standard structure for a Python project . . . 25
7.6 Additional Python links . . . 26

8 Libraries and Frameworks (Python) 27
8.1 TL;DR . . . 27
8.2 Intro and guidelines . . . 27
8.3 Flask . . . 27
8.4 SQLAchemy . . . 28
8.5 Jinja2 . . . 29
8.6 Werkzeug . . . 29
8.7 WTForms . . . 30
8.8 Babel . . . 30
8.9 Other libraries . . . 30

9 Front-end 31
9.1 Update 2018 . . . 31
9.2 References . . . 31
9.3 Past, present, future . . . 32
9.4 UX . . . 32
9.5 JavaScript . . . 33
9.6 CSS . . . 33
9.7 Patterns libraries aka style guides . . . 34
9.8 Build tools . . . 34
9.9 Quality assurance . . . 34

10 Testing 35
10.1 Unit testing . . . 35
10.2 Integration testing . . . 35
10.3 Front-end testing . . . 35
10.4 Full stack testing . . . 36
10.5 Links . . . 36

11 Documentation 37
11.1 Principles . . . 37
11.2 Comments . . . 37
11.3 Tools . . . 38
11.4 References & tips . . . 38

12 Development (and devops) tools and processes we are using 39
12.1 Build tools . . . 39
12.2 Source code management . . . 40
12.3 Continuous integration . . . 40
12.4 Tasks and issues management . . . 40
12.5 Devops tools . . . 40
12.6 Desktop tools . . . 40
12.7 Documentation . . . 41

13 Devops 43
13.1 Principles . . . 43
13.2 Tools . . . 43
13.3 Platforms . . . 43
13.4 Monitoring, etc. . . . 44

14 Open Source and Community 45

14.1 Selecting third party components . . . 45
14.2 Checklists from Toughtbot's "Maintaining Open Source Projects" . . . 45

15 Miscelanous subjects **47**
15.1 Dataviz . . . 47
15.2 Pandas . . . 47

16 Important checklists **49**
16.1 Committing . . . 49
16.2 Releasing software packages . . . 49
16.3 Scrum checklist . . . 50
16.4 Definition of Done . . . 50
16.5 Other / misc checklists . . . 51

Contents:

CHAPTER 1

Introduction, vision and principles

1.1 About this document

We are writing an *Employee Handbook* <http://en.wikipedia.org/wiki/Employee_handbook> for our current and future developers.

Here are some other similar document that have served as inspiration as we wrote this documents:

- <https://djaodjin.com/blog/guidelines.book.html>
- <https://github.com/thisissoon/Handbook>
- <http://playbook.thoughtbot.com/>

This doc is readable on <https://abilian-developer-guide.readthedocs.io/en/latest/>

1.1.1 Status

This is a work in progress. Actually, what you are reading now is a very early draft.

Even if it was "finished" at some point, it should still be updated at least twice a year to reflect the evolutions of our business and of our technology choices.

1.1.2 Why is this document public?

Why not? There are no trade secrets in this document.

Here are a few reasons:

- Hopefully, it will be useful to other people / companies.
- By publishing it, we hope to attract like-minded developers to strengthen our team.

1.2 Vision

1.2.1 About Abilian

Abilian develops a software platform (also called Abilian), horizontal products (Abilian SBE = an Enterprise Social Networking platform, Abilian CRM = a CRM platform, etc.) and vertical products (ex: our software for competitiveness clusters, etc.).

The goals for the technical team are:

- To create a useful (for us and for external users / contributors) software platform. By useful, we mean that we, and others, can achieve greater productivity when developing products and projects.
- To create successful products and customer projects based on our platform. By successful, we mean that they are useful to our customers, and they sell well.

1.2.2 Our open source projects

We are the main developers of the following open source projects:

- Abilian Core
- Abilian SBE
- OlaPy

1.2.3 Open source vs. customers projects

We are working both on public, open source, projects, and private projects for our customers (usually some customizations or extensions to our public projects).

When working on open source projects, we need to take a great care of not just making them successful technically, but also everything that will help them get other open source developers interested in the project.

1.2.4 Languages

We expect our developers to have knowledge of the following languages:

- Python 3
- JavaScript (ES6 and later)
- HTML
- CSS
- SQL
- RestucturedText and Markdown
- English

1.3 Principles

Here is a short list of principles (or list of lists of principles) that should guide you when working at Abilian:

- Write code for others (including "future you"), not just for the computer.
- Minimize all feedback loops. This includes for instance:
  1. The time it takes to run the unit test suite (ideally, no more than a few seconds).
  2. The time it takes between a change in the code and a change in the web Browser.
  3. The time between a customer request and the moment it can be used.
- Practice Test-Driven Development (TDD).

  Why? For two reasons:

  1. TDD (practiced correctly) helps come with better software design.
  2. It give us confidence that our product work properly, and that we can refactor them without fear of breaking things beyond repair.
- Practice "Documentation Driven Development", for public projects.

  Write documentation alongside the code. The simple act of trying to explain to others what we are trying to achieve with our software, and how, helps us clarify our thinking. If it's too hard to explain, then it's probably badly architected, designed or implemented.

  References:

  - README Driven Development
  - Documentation-driven Development
- Practice Domain Driven Design (DDD), for complex applications

  See the Architecture & Design chapter in this document.
- To ease developments on modern hosting platforms (including our own servers), we've adopted the "Twelve Factor" principles, see: <http://12factor.net/>.

CHAPTER 2

Getting started

2.1 Developer laptop set-up

2.1.1 Vagrant

Check out the Abilian Devbox project on GitHub (NB: doesn't exist yet.)

2.1.2 Linux box

Prefer a Debian or Ubuntu based distribution if you want to use the advice below.

Use the `install.sh` script from Abilian Devbox (NB: doesn't exist yet).

2.1.3 Mac OS X (Native)

Install Homebrew.

Install some dependencies:

```
brew install cairo cloc curl fontconfig freetype gdbm git \
   harfbuzz icu4c libffi libmagic libpng node openssl pango \
   pcre pixman pkg-config postgresql python python3 redis wget
```

(TODO: this is actually overkill, since some of these packages are actually dependencies of others.)

2.1.4 Generic (Python tools)

Install virtualenvwrapper:

```
pip install virtualenvwrapper
```

Then create one virtualenv for each of your projects, activate it when you start working on it, and you're done.

You may need to start PostgreSQL and Redis manually (or use launchrocket if you prefer).

CHAPTER 3

Process

3.1 Prioritizing and scheduling tasks

3.1.1 Scrum & Kanban

TODO.

References:

- http://leansoftwareengineering.com/ksse/scrum-ban/
- http://www.infoq.com/minibooks/kanban-scrum-minibook

3.1.2 Bug tracking

We use bug trackers to tracks bugs.

Of course the workflow is different when we work on customer projets and on open source projects.

One important thing to keep in mind is how to prioritize issues. For this, we're roughly following the *IEEE Standard Classification for Software Anomalies*:

- **Blocking**: Testing is inhibited or suspended pending correction or identification of suitable workaround.
- **Critical**: Essential operations are unavoidably disrupted, safety is jeopardized, and security is compromised.
- **Major**: Essential operations are affected but can proceed.
- **Minor**: Nonessential operations are disrupted.
- **Inconsequential**: No significant impact on operations.

3.1.3 Tools

For both our public projects and our internal communication, we're using GitHub's issue tracker and GitHub "Projects" (kanban-style vue on issues and pull requests).

When we need to communicate (privately) with our customers, we're using (currently) either Redmine (private instance) or Trello.

3.2 Git

3.2.1 Branch model

There are basically two main branching models for git-based development:

- git-flow or its close cousin GitHub flow.
- Trunk based developmen.

Wether we use the former or the latter depends on several factors, some of which are contextual.

Here are a few principles:

- Anything in the trunk (also called "main") branch is deployable.
- Ensure that the code you work on is thouroughly tested (~100% coverage)
- **Either**: commit improvement directly to trunk, and count on a posteriori code reviews to ensure ("better ask forgiveness than permission")
- **Or:**
  - To work on something new, create a descriptively named branch off of master (ie: new-oauth2-scopes)
  - Commit to that branch locally and regularly push your work to the same named branch on the server
  - When you need feedback or help, or you think the branch is ready for merging, open a pull request
  - After someone else has reviewed and signed off on the feature, you can merge it into master
- When making a release, we tag our tree with version numbers and make a branch from a release when we need to backport a patch.

3.2.2 Commit messages

Prefix each commit with one of the following:

- feat: A new feature
- fix: A bug fix
- docs: Documentation only changes
- style: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
- refactor: A code change that neither fixes a bug nor adds a feature
- perf: A code change that improves performance
- test: Adding missing or correcting existing tests
- chore: Changes to the build process or auxiliary tools and libraries such as documentation generation

See:

- Use Conventional Style to format your commit messages.
- Use Auto-Changelog to generate Changelogs from commit messages.
- Some additional remarks on Commit messages:.
- See also: https://github.com/angular/angular.js/blob/master/DEVELOPERS.md#commits

3.2.3 Code reviews

See: <https://blog.scottnonnenberg.com/top-ten-pull-request-review-mistakes/>.

3.3 Versionning and releasing

3.3.1 (Semantic) Versionning

See:

- <http://semver.org/>
- <https://www.python.org/dev/peps/pep-0440/>
- <https://github.com/relekang/python-semantic-release>

3.3.2 Releasing

We're (currently) using the setuptools-scm package to define version numbers for packages from git tags.

To make a release using this system:

- Update the following files: `CHANGES.rst` and `CONTRIBUTORS.rst`.
- Test locally
- Commit and push, wait for CI server to report status.
- Tag: `git tag vX.Y.Z`, then `git push --tags`
- *make clean*
- Check source build: `python setup.py sdist`
- Upload: `python setup.py sdist upload`

When releasing a new package on PyPI, add the `python setup.py register` phase.

This may change in the future.

CHAPTER 4

Software Design and Architecture

4.1 “Clean code”

Follow these principles:

- See <http://mike.zwobble.org/on-software-development/clean-code/>

4.2 The “4 principles of simple design”

This short series of principles is easy to remember, but surprisingly deep.

A design is simple to the extent that it:

- Passes its tests
- Minimizes duplication
- Maximizes clarity (reveals its intent)
- Has fewer elements (classes/modules/packages…)

References:

- <http://www.jbrains.ca/permalink/the-four-elements-of-simple-design>
- <http://blog.thecodewhisperer.com/2013/12/07/putting-an-age-old-battle-to-rest/>

4.3 SOLID & GRASP

A solid grasp (or “SOLID GRASP”, see below) of object-oriented techniques is expected to work on the Abilian project.

SOLID means:

- Single responsibility principle: a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class).
- Open/closed principle: software entities ... should be open for extension, but closed for modification.
- Liskov substitution principle: "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
- Interface segregation principle: many client-specific interfaces are better than one general-purpose interface."
- Dependency inversion principle: one should "Depend on Abstractions. Do not depend on concretions."

Here are a few references:

- SOLID and GRASP (according to Wikipedia)
- See also: <http://nikic.github.com/2011/12/27/Dont-be-STUPID-GRASP-SOLID.html>

Counterpoint:

- <https://speakerdeck.com/tastapod/why-every-element-of-solid-is-wrong>

4.4 Domain Driven Design

Domain-driven design (DDD) is an approach to software development for complex needs by connecting the implementation to an evolving model.[1] The premise of domain-driven design is the following:

- Placing the project's primary focus on the core domain and domain logic.
- Basing complex designs on a model of the domain.
- Initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

(Source: Domain-driven design on Wikipedia).

These are very general (and important) principles, which we will develop in future versions of this guide.

There are also important and useful principles on how to architect and design your application. See below for some patterns.

4.4.1 Tactical patterns

Here is the list of the technical (or tactical) patterns that are relevant to DDD.

- Layered (or onion, or hexagonal) Architecture: "Isolate the expression of the domain model and the business logic, and eliminate any dependency on infrastructure, user interface, or even application logic that is not business logic."
- Entities: "When an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. Keep the class definition simple and focused on life cycle continuity and identity."
- Value Objects: "When you care only about the attributes and logic of an element of the model, classify it as a value object. Make it express the meaning of the attributes it conveys and give it related functionality. Treat the value object as immutable."
- Domain Events: "Model information about activity in the domain as a series of discrete events. Represent each event as a domain object. These are distinct from system events that reflect activity within the software itself."
- Domain Services: "When a significant process or transformation in the domain is not a natural responsibility of an entity or value object, add an operation to the model as a standalone interface declared as a service."

- Modules: "Choose modules that tell the story of the system and contain a cohesive set of concepts. Give the modules names that become part of the ubiquitous language."
- Aggregates: "Cluster the entities and value objects into aggregates and define boundaries around each. Choose one entity to be the root of each aggregate, and allow external objects to hold references to the root only (references to internal members passed out for use within a single operation only)."
- Repositories: "For each type of aggregate that needs global access, create a service that can provide the illusion of an in-memory collection of all objects of that aggregate's root type."
- Factories: "Shift the responsibility for creating instances of complex objects and aggregates to a separate object, which may itself have no responsibility in the domain model but is still part of the domain design."

Some of these patterns may look a bit scary for a Python developer at first, but they make sense. See Deliver Domain Driven Design Dynamically for a good discussion.

4.4.2 Other patterns

See Eric Evans' book and/or the Domain-Driven Design Reference below.

4.4.3 References

- Domain-Driven Design Reference.
- Deliver Domain Driven Design Dynamically (talk about DDD in Python).
- DDD with Ruby.
- DDD in Ruby.
- Code in the Language of the Domain (short, but profound)

4.4.4 Additional topics

- Aggregate design: <https://www.infoq.com/news/2014/12/aggregates-ddd>

4.5 Hexagonal Architecture

(Also known as "Ports & Adapters" or "Onion Architecture").

Here's a high-level view of how we should structure our framework:

- <http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html> (+ all the references in the text).
- <http://alistair.cockburn.us/Hexagonal+architecture>

This architectural principle is compatible with DDD (see: <http://www.infoq.com/news/2014/10/ddd-onion-architecture>).

Additional references:

- <http://fideloper.com/hexagonal-architecture> (for PHP)
- <http://victorsavkin.com/post/42542190528/hexagonal-architecture-for-rails-developers> (for Rails)

4.6 Test Driven Development

> The best thing about years of TDD practice is that I will *never* commit a test without seeing it fail, whether I write it first or not. – Gary Bernhardt <https://twitter.com/garybernhardt/status/572856330572075010>

4.6.1 Motivation and principles

After seeing Gary Bernhardt video "Slow test / fast test" (see also this report on the same talk), I'm convinced that it's important, and possible to achieve, to have unit tests that pass as fast as possible (< 1 sec!), and possibly slower tests that are not run as often.

Our approach should be to distinguish between different tests classes:

- Unit tests (in tests/unit), that test classes mostly in isolation, using mocks or stubs if needed. These are the most important tests from the software design point of view, and these are tests that should run really fast (a few seconds for a whole test suite).
- Integration tests (in tests/integration), that test integration of actual components (no mocks).
- Functional web tests, that test the web apps using the web interface, either using a browser (Selenium / WebDriver) or that leverage the framework to a similar effect.
- Functional web API tests, that thoroughly test a web API using either an external tool (ex: FunkLoad) or the testing framework provided by Flask.
- Load tests, using something like FunkLoad.
- System tests, that test the full system (in a VM), including upgrade scenarios.

An important source of confusion for Python developers that are not experienced with TDD is that just because you are importing the `unittest` module doesn't mean you are doing unit testing (same if you are using, as we do, the `py.test` framework). Unit testing means that you are testing units in isolation.

At this point, our functional tests are merged with integration tests, load tests are non-existent. Regarding system tests, the tests that we are running on the Travis CI platform could qualify as systems tests, since we're rebuilding a whole VM each time we're running the test suite on Travis. But we are are not testing upgrade at this point.

We should aim for at least 80% measurable line coverage.

4.6.2 Tools for Test Driven Development

We're using py.test as our primary test runner and test framework, as we believe it to me the most "pythonic" of all testing frameworks (much more so that the standard library's `unittest` module, which is clearly heavily influenced by Java and indirectly SmallTalk). This was not always the case, so we plan to migrate our tests progressively to fully leverage py.test as a testing framework (and not just a test runner).

TODO:

- Links to pytest docs & tutorials.
- Mocking
- Web testing

(Or move this section to other chapters.)

4.7 API design

As library / frameworks author, we must be extra careful wrt the quality of our API. A good project should have APIs that are stable (so if you make a mistake, you must live with it for a long time), easy to use and remember, etc.

- <http://qt-project.org/wiki/API_Design_Principles>
- <http://lcsd05.cs.tamu.edu/slides/keynote.pdf>
- <http://pyvideo.org/video/1705/api-design-for-library-authors>

This is both true for "regular" API (in whatever language we are working on) and for "Web" API.

For Web API, we're promoting the REST architectural style.

4.8 Other topics

4.8.1 Naming things

- Stefan Holek, "Choosing Good Names".
- <http://hilton.org.uk/presentations/naming>
- <http://journal.stuffwithstuff.com/2016/06/16/long-names-are-long/>
- <http://journal.stuffwithstuff.com/2009/06/05/naming-things-in-code/>

4.9 Books

A few books relevant to this subject:

- Patterns of Enterprise Application Architecture (Martin Fowler)
- Refactoring (Martin Fowler)
- Domain Driven Design (Eric Evans)
- Growing Object-Oriented Software, Guided by Tests (Steve Freeman et Nat Pryce)
- Object Design: Roles, Responsibilities, and Collaborations (Rebecca Wirfs-Brock; Alan McKean)

CHAPTER 5

Modeling

5.1 Data modeling

5.2 Object modeling

5.3 Business process modeling

We favor BPMN2.

CHAPTER 6

Back-end

6.1 Intro

Our back-end are written in Python (usually).

6.2 Links

- <https://github.com/futurice/backend-best-practices>

CHAPTER 7

Python specific recommendations

7.1 Style guide & tools

We're mostly following PEP8 and the Google Python Style Guide.

We've been using automated code checkers (mostly Flake8 nowadays) with appropriate settings to ensure that deviations from the coding standard, as well as some non-stylistic issues, are detected.

Furthermore, we're using YAPF with the `google` settings, to automatically format our Python code.

We're also using isort to sort imports according to semantic and cosmetic rules.

We plan to use Git commit hooks to ensure that every new commit in the future fully respects these standards.

More:

- <https://github.com/amontalenti/elements-of-python-style>
- <https://github.com/marrow/marrow.github.io/wiki/Zen>

7.2 Python 2 vs. Python 3

We're currently still using Python 2 in our production code.

From 2017 on, we plan to start new projects using only Python 3, unless there are strong technical reasons not to do so.

We've started work to support Python 3 (alongside Python 2) on our open source projects, using the "six" library. We plan to finish it in 2017.

We've used the Python Future project for this, but more recently switched to six because it's more standard.

`pylint --py3k` is quite useful to help identify issues in current Python 2 code preventing migration to Python 3.

More info:

- <https://docs.python.org/3/howto/pyporting.html>

- <http://python3porting.com/>
- <https://tech.yplanapp.com/2016/08/24/upgrading-to-python-3-with-zero-downtime/>
- <http://python-future.org/compatible_idioms.html>
- <http://lucumr.pocoo.org/2013/5/21/porting-to-python-3-redux/> (NB: we don't follow Armin's advice of not using *six*).

7.3 General Python recommandations

- <http://stevenloria.com/python-best-practice-patterns-by-vladimir-keleshev-notes/>

TODO: add more.

7.4 Python tooling

In this section, we list tools that directly support our Python development process.

- YAPF (see above)
- Flake8 (see above)
- isort (see above)
- `pip-tools` (see below)
- Mypy (static checking using annotations)

Not used yet:

- Pylint (static checker, more powerful than Flake8 but needs more tuning). Currently used only to check for py3k compatibility (see above).

7.4.1 Managing dependencies

We manage dependencies for our projects and build them using pip.

Mandatory read: <https://caremad.io/2013/07/setup-vs-requirement/>.

Use pip-tools (<https://github.com/nvie/pip-tools>).

7.4.2 Virtualenv

Virtualenv is a tool to create isolated Python environments. Why is this good? You can create a new Python environment to run a Python/Flask/whatever app and install all package dependencies into the virtualenv without affecting your system's site-packages. Need to upgrade a package to see how it affects your app? Create a new virtualenv, install/copy your app into it, run your tests, and delete it when you are done. Just like git makes branching inexpensive and easy, virtualenv makes creating and managing new Python environments inexpensive and easy.

The doc: <https://virtualenv.pypa.io/en/stable/>.

7.4.3 PyPI mirror

Note: as of early 2017, this is not true anymore.

We have set up an internal PyPI mirror, using the devpi project.

Its main use is for our CI server (Jenkins).

Of course, our public open source projects should be buildable without depending on it.

7.4.4 Packaging

We package our Python projects using the standard tools (distutils, setuptools, pip).

The packaging allows us to distribute our modules for all developers who would like to use them.

The packages are hosted on Pypi mirror and can be installed with pip or easyinstall.

Links:

- <https://blog.ionelmc.ro/2014/05/25/python-packaging/>
- <https://blog.ionelmc.ro/presentations/packaging/>

7.5 Standard structure for a Python project

A Python project MUST have the following files:

- `README.rst`: we prefer `.rst` over `.md` because that's the markup language to use if you want your project to look good on PyPI.
- `LICENSE.txt`
- `setup.py` and `setup.cfg`
- `requirements.txt`
- `tox.ini`
- `.travis.yml`
- `Makefile`: cf. supra.

It SHOULD also have the following files:

- `setup.cfg`: used to contain config for additional files (ex: Babel, pep8, pyflakes, Sphinx...)
- `etc/`: containing additional configuration files for tools that don't support `setup.cfg` (ex: `pylint.rc`).

It MAY also contain:

- `Vagrantfile`
- `Dockerfile`
- `deploy/`: for scripts related to deployment (bash, Ansible...)
- `fabfile.py`: same

7.6 Additional Python links

These documents are rich collections of tips and links to sources of knowledge:

- <http://docs.python-guide.org/>
- <http://www.fullstackpython.com/table-of-contents.html>
- <https://github.com/kirang89/pycrumbs/>

CHAPTER 8

Libraries and Frameworks (Python)

8.1 TL;DR

Here are the most important Python libraries we're using daily to develop the Abilian platform and projects:

- Flask: http://flask.pocoo.org/
- Bootstrap: http://twitter.github.com/bootstrap/
- SQLAlchemy: http://sqlalchemy.org/
- WTForms: http://wtforms.simplecodes.com/
- Jinja 2 (for template makers): http://jinja.pocoo.org/docs/templates/

We are also using Django for one project we are maintaining (COMT).

8.2 Intro and guidelines

TODO.

Need a library: check there first <https://awesome-python.com/>, then PyPI.

8.3 Flask

Flask is a micro web framework written in Python and based on the Werkzeug toolkit and Jinja2 template engine. It is BSD licensed.

The latest stable version of Flask is 0.11 as of June 2016. Applications that use the Flask framework include Pinterest, LinkedIn and the community web page for Flask itself

Flask is called a micro framework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common

functions. However, Flask supports extensions that can add application features as if they were implemented in Flask itself. Extensions exist for object-relational mappers, form validation, upload handling, various open authentication technologies and several common framework related tools. Extensions are updated far more regularly than the core Flask program.

8.3.1 Reference:

- http://flask.pocoo.org/
- Manuel PDF (250 pages)

8.3.2 Books and tutorials

- <https://exploreflask.com/>: Free Book, beginners level.
- <https://realpython.com/blog/python/flask-by-example-part-1-project-setup/>: blog series.
- <http://blog.miguelgrinberg.com/category/Flask>: huge blog series + <http://flaskbook.com/>: book by the same author.

8.3.3 Some insightful presentations:

- <http://mitsuhiko.pocoo.org/pyladies.pdf>: very basic intro by Armin himself.
- <http://mitsuhiko.pocoo.org/flaskfun.pdf>: high level design considerations, but also some very useful code snippets.
- <https://speakerdeck.com/kennethreitz/flasky-goodness>
- <https://speakerdeck.com/mitsuhiko/advanced-flask-patterns>: must read (several times) once you've mastered the basics
- http://ua.pycon.org/static/talks/davydenko.pdf
- https://speakerdeck.com/jperras/large-scale-applications-with-flask-doing-more-with-less
- <http://slides.skien.cc/flask-hacks-and-best-practices/>

8.3.4 Well-done projects:

- https://github.com/hasgeek/ (Coaster, Eventframe, hasjob. . .)

8.4 SQLAchemy

SQLAlchemy is an open source SQL toolkit and object-relational mapper (ORM) for the Python programming language released under the MIT License.

SQLAlchemy provides "a full suite of well known enterprise-level persistence patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language". SQLAlchemy's philosophy is that SQL databases behave less and less like object collections the more size and performance start to matter, while object collections behave less and less like tables and rows the more abstraction starts to matter. For this reason it has adopted the data mapper pattern (like Hibernate for Java) rather than the active record pattern used by a number of other object-relational mappers. However, optional plugins allow users to develop using declarative syntax.

SQLAlchemy was first released in February 2006 and has quickly become one of the most widely used object-relational mapping tools in the Python community, alongside Django's ORM.

8.4.1 Reference:

- http://sqlalchemy.org/
- PDF Manual (560 pages)

8.4.2 Insightful articles

- http://lucumr.pocoo.org/2011/7/19/sqlachemy-and-you/
- http://www.aosabook.org/en/sqlalchemy.html
- http://derrickgilland.com/posts/demystifying-flask-sqlalchemy/

8.4.3 Presentations:

- http://techspot.zzzeek.org/files/2011/sqla_arch_retro.key.pdf
- Several videos (search "Michael Bayer" on youtube or PyVideo.org).

8.4.4 More resources

- <https://github.com/dahlia/awesome-sqlalchemy>

8.5 Jinja2

Jinja is a template engine for the Python programming language and is licensed under a BSD License created by Armin Ronacher. It is similar to the Django template engine but provides Python-like expressions while ensuring that the templates are evaluated in a sandbox. It is a text-based template language and thus can be used to generate any markup as well as sourcecode.

The Jinja template engine allows customization of tags, filters, tests, and globals. Also, unlike the Django template engine, Jinja allows the template designer to call functions with arguments on objects. Jinja is Flask's default template engine.

Doc for template designers is here: http://jinja.pocoo.org/docs/templates/

8.6 Werkzeug

Werkzeug is the WSGI framework underlying Flask. It's normally not needed to learn too much about it, but in case the doc is here: http://werkzeug.pocoo.org/docs/

8.7 WTForms

WTForms is a flexible forms validation and rendering library for python web development.

- http://wtforms.simplecodes.com/

8.8 Babel

Babel is an integrated collection of utilities that assist in internationalizing and localizing Python applications, with an emphasis on web-based applications.

Doc : <http://babel.pocoo.org/en/latest/>

(NB: not to be confused with babeljs, the EE6->ES5 transpiler).

8.9 Other libraries

Celery.

TODO.

CHAPTER 9

Front-end

9.1 Update 2018

9.1.1 JavaScript

We use eslint to keep our code tidy.

We may switch to TypeScript in the future.

We use prettier to auto-format our code, with appropriate plugins to format JS code inside HTML or Vue components.

We use Jest for tests.

9.1.2 VueJS

We now use VueJS for our new front-end projects.

Some references:

- Official Style Guide: <https://vuejs.org/v2/style-guide/>
- Testing: use <https://github.com/vuejs/vue-test-utils>
- Additional resources on testing: - <https://medium.com/pixelmatters/unit-testing-with-vue-approach-tips-and-tricks-part-1-b7d3209384dc>

9.2 References

Since this section is a little short on content, here's a terrific list of recommendations:

- <https://github.com/bendc/frontend-guidelines>

Also, this coding guide for HTML and CSS:

- <http://codeguide.co/>

9.3 Past, present, future

9.3.1 Current (aka old) state

Our front-end code base (at least for Abilian Core and Abilian SBE) is currently composed of the following JavaScript libraries:

- JQuery
- Datatables: http://datatables.net/
- Select2: http://ivaynberg.github.com/select2/

We're also using the Boostrap CSS framework (including some of its JavaScript extensions).

9.3.2 Where we are heading

A lot have changed in the JavsScript world over the last three years, since we've started the Abilian project (and company).

While the "jQuery + plugins + spaghetti code" approach that we've been using so far is probably enough for the kind of interaction we're featuring currently, it's now time to be more ambitious with our front-end code.

We need to move our code base to a more modern, decoupled, archiecture.

Our framework of choice is now Vue.js.

9.3.3 Constraints

Like everyone who is doing front-end development, we have to deal with buggy browsers from the pasts.

9.4 UX

- <https://www.smashingmagazine.com/2010/02/designing-user-interfaces-for-business-web-applications/>
- <https://uxdesign.cc/ux-trends-2017-46a63399e3d2>
- <https://www.youtube.com/watch?v=wrlHi2jKw_Y>
- <http://ux.handson.co/>
- <http://www.designingsocialinterfaces.com/patterns/Main_Page> + associated book
- <https://medium.com/@kennycheny/the-best-user-experience-design-links-of-2016-46e472660a52>
- <https://en.99designs.fr/blog/tips/7-unbreakable-laws-of-user-interface-design/>
- <https://medium.com/@erikdkennedy/7-rules-for-creating-gorgeous-ui-part-1-559d4e805cda>
- <https://medium.com/@erikdkennedy/7-rules-for-creating-gorgeous-ui-part-2-430de537ba96>

9.5 JavaScript

9.5.1 Embrace the future, now

We've used CoffeeScript in the past, but have reversed the decision to base all our front-end developments on CoffeeScript before it was put in practice.

We're now using ES6 aka ES2015 using the Babel transpiler for our new projects (part of our existing code base is still based on ES5).

Here are some references regarding ES2015:

- <http://slides.com/drksephy/ecmascript-2015>
- <https://github.com/getify/You-Dont-Know-JS> (book series)
- <http://exploringjs.com/es6/>

We plan to try using TypeScript (a statically typed variant of JS) in the future.

9.5.2 Libraries / frameworks

We've used Angular on customers projects in the past.

Our current JavaScript framework of choice is Vue.js.

9.5.3 Links

- <https://medium.com/@housecor/12-rules-for-professional-javascript-in-2015-f158e7d3f0fc>

9.6 CSS

We're currently using Bootstrap (v3), because it's extremely popular and quite comprehensive.

We're using LESS (or SASS, depending on the phase of the moon).

We're also trying to find a way to write better (more manageable) CSS/LESS/SASS code to go along our pluggable architecture. The following posts (and all the references given therein) give a lot of interesting advice and point to several useful methodologies, that we should put to use:

- <https://github.com/jareware/css-architecture>
- <https://speakerdeck.com/mdo/at-mdo-ular-css>
- <https://medium.com/@fat/mediums-css-is-actually-pretty-fucking-good-b8e2a6c78b06>
- <https://mattstauffer.co/blog/organizing-css-oocss-smacss-and-bem>
- <http://benfrain.com/the-ten-commandments-of-sane-style-sheets/>
- <https://medium.com/@Heydon/things-to-avoid-when-writing-css-1a222c43c28f>
- <http://rscss.io/>
- <https://github.com/edx/ux-pattern-library/wiki/Styleguide:-Sass-&-CSS>
- <http://alistapart.com/article/css-audits-taking-stock-of-your-code>
- <http://ecss.io/>

- <https://benfrain.com/the-ten-commandments-of-sane-style-sheets/>
- <https://sass-guidelin.es/>
- <https://philipwalton.com/articles/decoupling-html-css-and-javascript/>

9.7 Patterns libraries aka style guides

Generalities:

- <http://bradfrost.com/blog/post/style-guides/>
- <http://www.designforfounders.com/style-tiles/>
- <https://www.smashingmagazine.com/taking-pattern-libraries-next-level/>
- <https://24ways.org/2016/designing-imaginative-style-guides/>
- <https://www.springload.co.nz/blog/introduction-pattern-libraries/>

Specifics:

- <https://medium.com/ge-design/ges-predix-design-system-8236d47b089>
- <https://lightningdesignsystem.com/>
- <https://experience.sap.com/fiori-design-web/>
- <https://design.atlassian.com/>
- <http://dropbox.github.io/scooter/>
- <https://buffer.com/style-guide>

More here: <https://github.com/gztchan/awesome-design#style-guidebranding-octocat>

9.8 Build tools

We're using NPM for package management (and also YARN), and WebPack for build.

WebPack provides live reloading (with the right extension) so that's cool.

We **don't** use gulp or grunt.

- <https://medium.com/@dabit3/introduction-to-using-npm-as-a-build-tool-b41076f488b0>
- <https://yarnpkg.com/>
- <https://webpack.js.org/>

9.9 Quality assurance

JavaScript: We've started using *eslint* on some projects.

CSS: <http://benfrain.com/floss-your-style-sheets-with-stylelint/>

Unit tests: TODO.

Functional tests: we should be using Selenium (via Webdriver) more.

CHAPTER 10

Testing

Testing is very important. If it's not tested, it's broken.

General concepts

- <https://blog.nelhage.com/2016/12/how-i-test/>
- <https://blog.nelhage.com/2016/03/design-for-testability/>

10.1 Unit testing

Unit tests should be fast (less than a few seconds to run a full test suite).

They should use py.test because it's our testing framework of choice.

- <http://devork.be/talks/advanced-fixtures/advfix.html>

10.2 Integration testing

TODO.

10.3 Front-end testing

When developing SPA (single page applications), use your framework of choice prefered testing tools to implement unit and integration tests.

- https://www.sitepoint.com/javascript-testing-unit-functional-integration/>

10.4 Full stack testing

To run full stack (back end + front end) integration testing, use Selenium or a variant of it (or phantomjs / casperjs / ...).

Use the "page object pattern": <https://nulogy.com/who-we-are/company-blog/articles/using-the-page-object-pattern-in-your-automated-tests/> to make your test cleaner an maintainable.

Example for Python: <https://github.com/alisaifee/holmium.core/>

Best practices can be explored at SeleniumConf (for instance: <http://2016.seleniumconf.co.uk/>).

10.5 Links

- <http://www.obeythetestinggoat.com/pages/book.html> "Test-Driven Web Development with Python": (focusses on Django)

CHAPTER 11

Documentation

Good developers don't just write code, they also write good documentation alongside the code.

11.1 Principles

Given the tools we are using, the following principles should be applied:

1. We should treat the documentation as a first-class deliverable of our software. When you write a public API for a module, always build the doc as you develop and check that the doc makes sense.
2. It's important to treat both the narrative part and the API part properly. If you document only your API, as some developers do, this won't be enough to get potential users on board with our efforts.
3. For the API part, one should document all the public parts (classes, methods, attributes, functions) of the API, and nothing else, with proper docstrings.

Examples of outstanding documentation (among many others) include: Flask, SQLAlchemy and scikit-learn.

11.2 Comments

> *A delicate matter, requiring taste and judgment. I tend to err on the side of eliminating comments, for several reasons. First, if the code is clear, and uses good type names and variable names, it should explain itself. Second, comments aren't checked by the compiler, so there is no guarantee they're right, especially after the code is modified. A misleading comment can be very confusing. Third, the issue of typography: comments clutter code.* – Rob Pike

- <http://ericholscher.com/blog/2017/jan/27/code-is-self-documenting/>
- <https://dev.to/raddikx/dont-document-your-code-code-your-documentation>

11.3 Tools

For Python software, we use Sphinx to generate documentation from both standalone texts, and text embedded in the code (using Docstrings).

See: <http://ericholscher.com/blog/2016/jul/1/sphinx-and-rtd-for-writers/>

This means that our documentation is written using the RestucturedText (ReST) language. This cheat sheet can be useful:

<http://github.com/ralsina/rst-cheatsheet/raw/master/rst-cheatsheet.pdf>

BTW, we're not using Markdown (or just parcimonuously) for reasons outlined there: <http://ericholscher.com/blog/2016/mar/15/dont-use-markdown-for-technical-docs/>.

We publish our documentation on ReadTheDoc.

The `README.rst` files at the root of our projects are also extremely important (as they are often the first things that people read when discovering our projects, either on GitHub or on PyPI).

11.3.1 Specific recommendations regarding Sphinx

For Sphinx to produce outstanding documentation, we need to pay attention to the way we write docstrings.

We should use the Napoleon Sphinx plugin.

See the following links (from the Napoleon doc) for additional recommendation:

- <https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt>
- <https://github.com/Khan/style-guides/blob/master/style/python.md#docstrings>

11.4 References & tips

- <http://docs.python-guide.org/en/latest/writing/documentation/>
- <http://ericholscher.com/blog/2014/feb/27/how-i-judge-documentation-quality/>
- <http://docs.writethedocs.org/>
- <http://docness.readthedocs.io/>
- <http://jacobian.org/writing/great-documentation/>
- <https://speakerdeck.com/ogrisel/documentation>
- <https://medium.com/@limedaring/five-tips-for-improving-your-technical-writing-and-documentation-47353723c8a7>

CHAPTER 12

Development (and devops) tools and processes we are using

Here's a list of guidelines that are currently used. They are not set in stone, of course, and will evolve as new techniques and methodologies are explored by the team.

12.1 Build tools

12.1.1 Make

We like having a Makefile at the root of our projects.

It should at least support the following targets:

- `tests` (default)
- `clean` and `tidy`
- `run`
- `lint`
- `format`

Some inspiration here: <https://github.com/aclark4life/project-makefile> + <http://slides.com/aclark/project-makefile#/>

12.1.2 Others

As already stated elsewhere in this document, we're using *setuptools* to build Python packages. See the fine blog posts by Hynek and Ionel (referenced elsewhere in this doc).

For front-end packages: use *npm* and *webpack* (not *gulp* or *grunt*, they are probably fine but not needed). Investigate *rollup*.

12.2 Source code management

We're using Git as our primary source code management system, and GitHub as our main repository.

For public projects, we have a public project account (https://github.com/abilian).

For private projects, I have a private personal account. We'll move to a private group account later (it's more expensive ;).

12.3 Continuous integration

We're using Travis for public projects, and Circle CI for both public and private projects.

- https://travis-ci.org/abilian/
- https://circleci.com/projects/gh/abilian

We used to use Jenkins, and might use it again in the future. Or gitlab CI. But not ATM.

12.4 Tasks and issues management

See the "Process" chapter. [TODO: link].

12.5 Devops tools

12.5.1 Fabric

Deployment to staging and production servers is currently managed by Fabric using the Fabtools add-ons.

Later, we'll look into using Ansible or Salt, though at this point (for single server deployment), I believe Fabric is the best choice.

Our Fabric scripts, however, are both a bit simplistic and convoluted. For instance, they don't deal with database migration, code rollback, etc.

12.5.2 Vagrant

We're using Vagrant to run full deployment tests on pristine virtual machines. It should become most valuable when we will have actual projects in production, and will need to address issues such as upgrades.

12.6 Desktop tools

12.6.1 Editors and IDEs

The best Python IDE is PyCharm, though its quality seems to be going down recently.

If you don't like IDEs, use a regular text editor, such as Emacs, Vim, Atom, Sublime... But make sure you have installed the plugins that will make you more productive, such as:

- Syntax highlighting

- Dynamic code linting
- Autocompletion

12.6.2 Chrome plugins

Here are a few Chrome plugins that we find useful:

- BuildReactor: monitors CI builds (supports both Jenkins and Travis) and sends Growl-style notifications when a build fails.
- Check My Links: link checker, useful when testing a web application.

12.6.3 Firefox plugins

- Opcast Desktop: a Firefox extension that allows you to launch more than 450 tests (UX, SEO, accessibility, performance...) on the webpages you're currently browsing or coding.

12.7 Documentation

You probably want a tool on your laptop that gives you quick access to the documentation of the languages and libraries you are using.

On the Mac we're using Dash.

We've also found that printing "cheat sheets" can be useful.

CHAPTER 13

Devops

(Very scketchy for now).

13.1 Principles

- <https://12factor.net/>
- <http://www.clearlytech.com/2014/01/04/12-factor-apps-plain-english/>

Note: we're not yet 12 factor compatible, actually, due to our reliance on local file storage. This will probably be fixed in 2017.

- <https://zachholman.com/posts/deploying-software>

13.2 Tools

- Fabric
- Invoke <https://github.com/pyinvoke/invoke>
- Docker
- Vagrant (for dev, not prod)
- to investogate:
- <http://platter.pocoo.org/dev/>
- <http://pythonhosted.org/fapistrano/>

13.3 Platforms

- Heroku (for playing purposes)

- Dokku (open source Heroku alternative,
- Kubernetes
- Or plain old "a la mano" deployment (using Fabric, Invoke. . .) using nginx as a front-end web server and either gunicorn (currently preferred) or uwsgi as application server, with supervisord for process management.

13.4 Monitoring, etc.

- Sentry / getsentry
- <http://uptimerobot.com/>

CHAPTER 14

Open Source and Community

14.1 Selecting third party components

Our open source projects are currently under the LGPL license.

14.2 Checklists from Toughtbot's "Maintaining Open Source Projects"

Cf. <https://gumroad.com/l/maintaining-open-source-projects/>

14.2.1 Contributions

Points to address:

- Adopt a style guide
- Use static analysis tools
- Request regression tests for every change
- Run tests on every commit
- Set priorities

14.2.2 Documentation

Points to address:

- Write a Readme document (Cf. <https://github.com/noffle/art-of-readme>).
- Write a History document

- Write a Contributing document
- Write a Guidelines document
- Write a Releasing document

14.2.3 Versionning & Releasing

Points to address:

- Semantic versioning
- Regular release cycles
- Deprecation cycles
- Security releases

14.2.4 Community

Points to address:

- Communication channels
- Answering questions
- Issue tracker gardening
- Enough communication already!
- On effective feedback
- Expectations and guilt

CHAPTER 15

Miscelanous subjects

15.1 Dataviz

- <http://chimera.labs.oreilly.com/books/1230000000345/index.html> Book: Interactive Data Visualization for the Web (O'Reilly)
- <http://jsdatav.is/intro.html> Book: Data Visualization with JavaScript
- <http://plottablejs.org/components/>
- <http://keen.github.io/dashboards/>

15.2 Pandas

CHAPTER 16

Important checklists

Here we will list important checklists.

They are not written yet.

16.1 Committing

Before a commit:

- TODO.

16.2 Releasing software packages

To release software (NB: TODO):

- Check that the following files are OK: README, INSTALL, CHANGELOG, CONTRIBUTORS
- Run tests locally
- Tag
- Build
- Push on PyPI

16.2.1 README checklist

A helpful checklist to gauge how your README is coming along:

- One-liner explaining the purpose of the module
- Necessary background context & links
- Potentially unfamiliar terms link to informative sources

- Clear, runnable example of usage
- Installation instructions
- Extensive API documentation
- Performs cognitive funneling
- Caveats and limitations mentioned up-front
- Doesn't rely on images to relay critical information
- License

(From https://github.com/noffle/art-of-readme).

16.3 Scrum checklist

<https://www.crisp.se/wp-content/uploads/2012/05/Scrum-checklist.pdf>

16.4 Definition of Done

Definition: <http://guide.agilealliance.org/guide/definition-of-done.html>

See also: <http://www.scrum-breakfast.com/2014/05/the-three-faces-of-done.html>

Expected benefits:

- the Definition of Done provides a checklist which usefully guides pre-implementation activities: discussion, estimation, design
- the Definition of Done limits the cost of rework once a feature has been accepted as "done"
- having an explicit contract limits the risk of misunderstanding and conflict between the development team and the customer or product owner

For our open source projects, here is our definition (draft):

- Code produced (all 'to do' items in code completed)
- Code commented according to our commenting guidelines
- Code checked in in VCS, feature branch merged into development branch
- Peer reviewed (or produced with pair programming) and meeting development standards
- Builds without errors
- Quality check tools (linters) pass
- Unit tests written and passing
- Deployed to system test environment and passed system tests
- Demonstrated to product owner
- Any build/deployment/configuration changes implemented/documented/communicated
- Relevant documentation/diagrams produced and/or updated
- Release notes updated for upcoming release
- Ticket move to "Done" and/or closed

16.5 Other / misc checklists

- <http://webdevchecklist.com/> Web Developer Checklist
- <https://github.com/futurice/backend-best-practices#release-checklist>