開発アーキテクチャー

目次

機能の設計と実装

ここで登場する用語はほぼ ロバストネス分析 のもの。このプロジェクト用に簡略化して再定義する。

システム:	このプロジェクトで開発する範囲
アクター:	システムと接点をもつ外部の何か。ユーザーや、別システム等、システムに入力を能動的に行う何か。DB等は入出力をもつがアクターとはしない。
ユースケース:	アクターの入力によって、システムに何が起きて、結果としてアクターになにが出力されるのか、をまとめたストーリー、手順
ステレオタイプ:	ロバストネス分析等で、同じ種類のアイコンなんだけどあるアイコンの部分には特別な意味を持たせたい、という場合に、アイコンにステレオタイプというラベルを付けて見分ける

プログラミング

- レビュー観点 に沿って実装する
- 外部パッケージの制限は素直に受け入れる
  - むやみに Forkしたり、パッチを当てたり、ラッパー層を作ったりしない（修正を [PR] しよう）
  - ドキュメント化されていない内部関数を直接使わない（変更に弱くなる）
  - 公開関数・公開設定値でも、明示的にドキュメント化されていないなら、それらを無条件では信用しない
    - 例: django の `SESSION_COOKIE_AGE` がセッションのバックエンドに使われているキャッシュのタイムアウトに適用されるなど
    - 適宜相談の上、利用方針を決定する
- urlとパッケージ名をそろえる(django.conf.urls.include をつかう)
  - 例: アプリ名dashboardなら、 /dashboard/... というURLにする
- 抽象度の高い機能はコード書く前にpipで公開されてる良い先行実装がないか確認する、人に聞く
  - 独自バリデータをがんばって実装せず、dajngo.forms か colander を使う
- リリースブランチ・masterブランチへのマージ前に、マージ先のブランチでrebaseする -> Github
- クラス
  - 概念をまとめるためだけにクラスを導入するのは止めよう
    - NG: モジュール内に存在する関数のグルーピングをクラスの @staticmethod で行う
    - NG: メソッド内でselfを使っていない
- バッチ
  - 再実行可能（冪等性がある）
  - リカバリ可能とする
  - リカバリ手順を明確にする
  - 検証可能, エラー検知可能とする
  - バッチの最後に、検証クエリを実行してバッチが期待通りに終了したか確認し、異常時はエラー通知する

Django project structure

Djangoプロジェクトの構成。Djangoのstartpr

```
<project-root>/
  +-- <django-project>/
  |     +-- <django-project>/  #
  |     |    +-- urls.py
  |     |    +-- wsgi.py
  |     |    +-- <utility-module
  |     +-- <django-app>/
  |     +-- <django-app>/
  |     |    +-- admin.py
  |     |    +-- api.py  # appmo
  |     |    +-- apps.py
  |     |    +-- appmodels.py  #
  |     |    +-- forms.py  # Dja
  |     |    +-- management/
  |     |         +-- commanc
  |     |                   +--
  |     |                   +--
  |     |    +-- migrations/
  |     |    +-- models.py  # Dj
  |     |    +-- models_sqla.py
  |     |    +-- repositories.py
  |     |    +-- tasks.py  # bou
  |     |    +-- urls.py  # Djar
  |     |    +-- views.py
  |     +-- settings.py
  |     +-- manage.py
  +-- ChangeLog.rst  # リリース時にどんな機能が公開されるかを記載、PR時に概要が分かるようにしておく
  +-- README.rst  # このプロジェクトの始め方等を書く
  +-- circle.yml  # CIによる自動テストの設定
  +-- constraints.txt  # Pythonパッケージのバージョン指定を書く
  +-- mypy.ini  # 型チェックの設定(toxのtypecheckセクションから
  +-- requirements.txt  # 本番環境で使用するPythonパッケージを
  +-- requirements_dev.txt  # 開発環境で使用するPythonパック
require.txtに書く)
  +-- run-requires.txt  # 本番環境で使用するPythonパッケージを
  +-- scripts/  # スクリプトファイルがあればここに置く
  +-- site_media/  # メディアファイルがあればここに置く
  +-- static/  # Django標準では各appディレクトリに置くが、本プロジ
  +-- templates/  # Django標準では各appディレクトリに置くが、本プ
  +-- tests/  # Django標準では各appディレクトリに置くが、本プロジ
  |     +-- conftest.py  # pytest設定
  |     +-- <django-app>/
  |     |     +-- test_<test-purpose>.py
  |     |     +-- test_<test-purpose>.py
  |     +-- <django-app>/
  +-- tests-require.txt  # テストで使用するPythonパッケージを
  +-- tox.ini  # テスト実行、型チェック、pyflakes実行を行う
  +-- wheelhouse/  # 依存パッケージは全てwhl形式でここに保存しリ
```

- <django-project>/
  - manage.pyを含むDjangoのプロジェクトルートディレクトリ
  - 同名のサブディレクトリ(アプリ)にはurls.py, wsgi.pyがある（Djangoのルール）
- <django-app>/
  - 機能ごとに<django-app>を作成する
- 汎用モジュール
  - 汎用的な名前を持つモジュールの総称。汎用モジュール
  - api, views, forms, models, models_sqla, repositories, utils, tasks
  - 汎用モジュールをパッケージ/モジュールに分割しない(`views/item_view.py` とか `utils/console` とか)
- <utility-module>.py
  - プロジェクト全体で共有する汎用ツールを適切な名称でモジュール化し django-project 以下に置く
  - cache.py, console.py, stftp.py, s3.py 等
- api.py
  - アプリケーションモデルのインスタンスを組み立てて、呼び出し元へ返す -> 実例
  - 同一Djangoアプリの boundary だけがapi.pyの公開関数を利用できる
  - boundary はapi.py経由でアプリケーションモデルを読み込み、そのメソッドを使う（アプリケーションモデルを直接インポートしない）。
  - api.py のテストは不要（テストが必要な量のコードを実装しない）
  - viewのテストでは、api.pyで提供する公開関数をmockする
- views.py

構成管理

ChangeLog.rst

書くこと

- `#<チケット番号> <追加・変更・削除した機能が分かる短い説明>`
- dev、本番、開発者の個人環境へのデプロイ時にその環境で行うコマンドライン操作等
  - `pip install`
  - `tox -r`
  - `python manage.py migrate`
  - ファイル・ディレクトリ作成
  - rundeck 設定変更

記入例例, 手順 -> Github

パッケージバージョン管理

以下のファイル・ディレクトリで管理する。

- constraints.txt
- requirements.txt
- requirements_dev.txt
- run-requires.txt
- tests-require.txt
- wheelhouse/

使い方

- 直接依存しているパッケージは、 `run-requires.txt` に記載する
- パッケージのバージョンは、直接・間接依存に依らず `constraints.txt` て

詳しい書き方、読み方、利用方法、更新方法については、 依存パッケージの

Pythonコーディング

Type Hints

- PEP 484 に従う

docstring

- docstring は、 reStructuredText 記法で書く
- 関数・メソッドのdocstring:
  - 10行を超えるならばdocstringを書くべき、20行を超えるなら
  - 引数と戻り値の記述は、Sphinxで扱える形式の"詳細情報フィー
- クラスのdocstring:
  - クラスの目的を必ず書く
- モジュールのdocstring:
  - できるだけ書く
  - 後で「この関数どこに置こうかな」という場合に適切なモジュー
  - このモジュールはどんな目的なのだろう？というのがすぐに分か

実例等の詳細は ドキュメンテーション を参照

UnitTest

- UnitTestを置く場所
  - `/tests` に置く（Djangoプロジェクトとは分ける）
  - `/tests/<django-app>/test_*.py` のようにアプリ毎にテストディレクトリを分ける
- Testクラスは接頭辞にTestをつける: `class TestHoge`
- テストランナーにはpytestを使う
- テストの実行にはpytest を使う
  - `django.test.TestCase` より `unittest.TestCase` を使う
  - DjangoのDB処理を伴うテストクラス/メソッドには `@pytest.mark.django_db` を付ける
  - `pytest.mark.parametrize` を使う場合は関数形式のテスト(`test_foobar()`)で記述する
- データフィクスチャの作成にpytest.fixtureを使う
  - pytest.fixture で `FooModel.objects.create()` でテスト毎に作る
  - `django.test.TestCase.fixtures` を使わない
  - factory_boyを使わない
- testfixturesを積極的に使う
  - `testfixtures.replace` デコレータは使わず `testfixtures.Replacer` を使う(pytestとの兼ね合いがあり
  - 一時ディレクトリとして /tmp ではなく `testfixtures.tempdir` を使う
  - `datetime.now()` や `date.today()` をmockする場合 `testfixtures.test_datetime` を使う
  - log出力のテストには `testfixtures.LogCapture` を使う
- settingsの書き換え
  - setUp/tearDown で行わない
  - `django.test.override_settings` をデコレータで使う
  - pytest-djangoのsettings fixture を使う
- viewの単体テスト
  - `test_views.py` に書く
  - django の client やdjango_webtetを使わず、view関数を直接呼び出してテストする
  - clientを使うテストは 結合テスト として書く
- 1つの機能（関数やビュー）をテストするために1つのテストクラスを用意する
  - そのテストクラスに `target` fixtureを用意、テスト対象を返す
  - パラメータ化できるテストは pytest.mark.parametrize を使う
- テストメソッドは 3A スタイル (Arrange, Act, Assert) [4] で記述する -> 実例

[4] http://wiki.c2.com/?ArrangeActAssert

- テストClient を使うテストは、 test_it (integration test) メソッドに記述する
- テストClientには `django_webtest` を使う (元々 `django.test.TestCase.client` はtemplateとcontextをテ
のとdocstringに書いてある)

CI

- 継続的結合 (continuous integration) としてCircleCIを使う
- circle.yml で自動テストを設定する
  - toxのターゲット flake8, typecheck, cov を実行する
  - cov (coverage) の結果をCircleCIで閲覧できるように配置する

Review-guideline 参照

レビュー

- レビューしやすい小さい単位でレビューを依頼する
- 新しいブランチを作成したら
  - 最初にChangeLogを更新する
  - WIP PRを作成する
    - WIP=Work In Progress=作業中. タイトルに `[WIP]` と付けておくとよい（完成じゃないよと伝える）
    - PR=Pull Request. githubでPRする
- レビュー前にやること
  - pep8, flake8, typecheck など、機械チェックは事前に行っておく（テストに組み込む）
  - 仕様面、技術面などについて不安点などを早めに確認する（実装中に聞いておく）
  - 不具合修正の場合は経緯、修正方針などをまとめておく
- レビュー依頼時にすること
  - PR本文に、レビューしてほしいことを明記する
  - PR本文に、実装の根拠となった仕様、または仕様へのリンクを記載する
  - PRコード差分に、読んだだけで意図が分かりづらい部分があればコメントしておく
    - コード説明の場合、コード内にコメントを書いた方が良い
    - ファイル説明等の場合、docstringを書いた方が良い
  - PRの”Reviewer”に依頼したい相手をアサインする
- レビュアーのアクション（GithubのReviewerにアサインされたとき）
  - 確認してほしいことをチェックする
    - 不明確ならその時点で「ここが分からない」と書いて差し戻す
  - 差分にコメントする
  - レビュー結果を書く
    - 再レビュー不要
      - コメントしたけど軽微なので、修正してもしなくてもよい場合
      - レビューコメントに"LGTM with nits"と書く
      - ステータスをApproveにする
    - 再レビュー必要
      - コメントした箇所に重要な修正が必要そうな場合
      - 重要な箇所に "[必須]" or "[重要]" と書く
      - ステータスをRequest Changesにする