社内の勉強会でこのテーマで発表しました。

せっかくなのでmarkdownのまま以下に保存しておきます。最後らへんは時間がなくてめちゃくちゃ分かりにくいです。。。すいません

Django/DRFの考え方

どのようにDjango/DjangoRestFrameworkで新機能・新APIを作るかを書き出してみた

どのようにDjango/DjangoRestFrameworkで新機能・新APIを作るかを書き出してみました。

- ※僕の理想なので偏見や主張が強い書き方があるかもしれません、そういうのも併せてみんなでディスカッションしたいです！
- ※あくまで一論なので、実務では実際のユースケースに照らし合わせて考えることが必要です

理想の順番

1. そもそもその機能が必要か?
2. DBにおいてどう表現する？
3. Modelを作る
4. どのようなエンドポイントを生やすか？
5. Viewを作る
6. Serializerを作る
7. Modelを作る

そもそもその機能が必要か?

そもそもバックエンド側で修正が必要であるか

- 見た目関する機能であれば、無理にバックエンドに持たせる必要はないと考えてる。フロントエンド側で吸収するという選択も十分あり
    - 例えば、**初めてアプリを開いたユーザーにはバナー（Xを押すと消える）を表示する**という機能
- しかし、フロントエンド側で対応することの問題点もある
    - ユーザーの端末変更時にリセットされる
    - ユーザーのその機能の利用状況が分からない
        - 例えば、追跡してリマインドメールを送ることができない

DBにおいてどう表現する？

データベースにおいてテーブルはどのように表現するか

- DBがどのような形でデータを保存するべきかを考える
- この段階ではDjango/DRFの考慮は不要
    - データベースはアプリケーションサーバーより長生き、なのでDBとしてあるべき表現に集中する

具体的に何を注意する

- 将来的な拡張性を考える
  - 例：都道府県→ `enum型` や `int型`
    - 都道府県は拡張しないのでマスタ不要
- 柔軟性を考える
  - 例：googleカレンダーのような柔軟な繰り返し設定を作りたい→ `JSON型`
    - 仕様の変更頻度が高く、変更時の変更範囲をできるだけ減らすため
- 特定のデータの状態管理が複雑になりそうであれば状態ごとにテーブルを分割する（例：ActiveUserテーブル、LeavedUserテーブル）
  - 状態ごとに保持する属性が異なるデータを同じテーブルで表現してしまうとnullableなカラムができてしまう
    - →データを扱う時にnullableに注意しなければならない呪いにかかる
- 論理削除も同様に ~~避けた方が良い~~ 慎重に採用しなければならない
  - アクティブなデータを取得したいたびに `is_deleted=False` をクエリに付与する呪いにかかってしまう

Model

- 考えたテーブルをDjangoのModelを使って表現する
- Modelのためにテーブルを変える、みたいなことは極力避ける

どのようなエンドポイントが生えているべきか考える

取得系のAPIで考えた方が良いこと

- ドメイン全体/フロントエンドから見て、どのような粒度でデータが返却されるのが自然か考える
    - 例えば、出会い系のアプリにユーザーが所属していた部活を複数登録する機能があって、DBにはユーザーテーブルと部活ユーザーテーブルが存在するとする
        - この状況ではユーザーの取得APIとユーザーの部活取得APIを分ける意味は薄い（と考えれる）
        - それは、このコンテキストにおいてユーザーの部活はユーザーの属性の一部として見做せるから
        - 仮に、学校の教師が利用する学生管理システムだった場合は、**ユーザーの部活は、ユーザーと独立した概念**であると言いやすい
            - なので、ユーザーの取得APIとユーザーの部活取得APIを分ける意味はありそう

どのようなエンドポイントが生えているべきか考える

登録/更新系のAPIで考えた方が良いこと

- 積極的にRESTfulに沿って作成する。POST/PUT/PATCH/DELETEなど
    - ただし、更新処理は乱立しがち
        - 潔く動詞で表現したり、利用するユーザーの種類によってエンドポイントを分けることが大事
- アトミックな更新処理であれば1つにまとめる

どのようなエンドポイントが生えているべきか考える

共通

- **全てのテーブルに関してCRUDのエンドポイント存在することは避ける。**管理するコストが上がる/フロントエンドが考慮することが増えるから
- エラーハンドリング
  - エラーには大きく二分できると考えている
    - 1.内容に関するエラー（string型だと思ったら数値型だったみたいな、シンタックスのエラー）
    - 2.ドメインロジックに関するエラー（「このユーザーは現在利用できません」みたいな、セマンティクスなエラー）
  - 特に、ドメインロジックに関するエラーに関しては、実装漏れが単体テストで判明しにくいため、実装時に洗い出しておくことが重要

View

必要なエンドポイントから逆算する

- DRFの恩恵を受けれるよう積極的に利用していく
  - CRUDの一部でもDRFの仕組みを使えるなら積極的に使う
  - 例）listとretrieveはReadOnlyModelViewSet で作成し、 createは自前で作成し、deleteはDestroyModelMixinで作成する

Serializer

1つ1つ別のSerializerを作らない
→管理コストが増える・フロントエンドが考慮する必要がある型が増えてしまう

権限とパフォーマンスがトレードオフ。うまくバランスを見て考える

何を持ってSerializerを分割するかはアプリケーションの規模次第。APIごとか、リソースごとか、リソースx呼び出し元ユーザー種類ごとか、リソースxRead/Writeごとか

Serializer

おすすめは、

何に使われているかでなく、何を表しているかで命名する

→前者だと利用される場面が限定されるため、

→しかし「何を表しているか」というのも難しい、、

```
# X NewsPostSerializer
# O NewsCreateSerializer / NewsCreateByAdminSerializer（誰に使われるかSerialzer）
class Meta:
    model = News
    fields = ["id", "title"]

# X NewsSerializer
# O NewsPublicSerializer / NewsFullSerializer（どのような属性を持つかで表現した例）
class Meta:
    model = News
    fields = "__all__"
```

Public/Private/Admin/

Serializer

serializerに処理が肥大化することが多いため、ModelのQuerySetを積極的に使っていく
例えば、お知らせテーブルがあったとして、各ユーザが見れるのは
`opened_at<現在日時` かつ `現在日時<closed_atかつis_active=True` なレコードのみ

この条件は「ユーザーへの検索時のフィルタリング」でも「運営管理者が現在このお知らせが配信されているかチェック」時など複数回利用される。

お知らせのQuerySetに書くと、一箇所で管理できて綺麗になる

Model

ドメインが持つ処理を書く
更新処理などが存在する

インタフェースを狭くしたい

ドメインを持たせたい

積極的にQuerySetを持たせる