



Руководство по стилю Django



ITcoder

Вступление от автора перевода.

В данном руководстве автор очень хорошо рассматривает все ключевые моменты в **Django** и **Django Rest Framework**.

Но данное руководство к сожалению не для новичков. Руководство схоже с книгой "**Django two scoops**". Но в ней рассматриваются практику данной компании **HackSoft**.

Оригинал на английском можно прочесть тут :

<https://github.com/HackSoftware/Django-Styleguide>

Перевела : К.Жумабекова

связь **Telegram**:

<https://t.me/kjumabekovva>

- Оглавление:
- Что это такое?
- Как его использовать?
- Обзор
- Почему нет?
- Cookie Cutter
- Модели
 - Базовая модель
 - Валидация - `clean` и `full_clean`
 - Валидация - ограничения
 - Свойства
 - Методы
 - Тестирование
- Сервисы
 - Пример - сервис на основе функций
 - Пример - сервис, основанный на классах
 - Соглашение об именовании
 - Модули
 - Селекторы
 - Тестирование
- API и сериализаторы
 - Соглашение об именовании
 - Основанные на классах и функциях
 - Список API
 - Обычный
 - Фильтры + пагинация
 - API детализации
 - API создания
 - API обновления
 - Получение объектов
 - Вложенные сериализаторы
 - Расширенная сериализация

- Ссылки **Url**
- **Settings**
 - Префикс переменных окружения с помощью **DJANGO_**
 - Интеграции
 - Чтение из **.env**
- Ошибки и обработка исключений
 - Как работает обработка исключений (в контексте **DRF**)
 - Ошибка проверки достоверности в **DRF**
 - **Django's ValidationError**
 - Опишите, как будут выглядеть ваши ошибки **API**.
 - Знайте, как изменить поведение обработки исключений по умолчанию.
 - Подход 1 - Использовать исключения по умолчанию **DRF**, с очень небольшими изменениями.
 - Подход 2 - предложенный **HackSoft** способ.
 - Больше идей
- Тестирование
 - Соглашения об именовании
- **Celery**
 - Основы
 - Обработка ошибок
 - Конфигурация
 - Структура
 - Периодические задачи
 - За пределами
- Кулинарная книга
 - Обработка обновлений с помощью службы
- **DX (опыт разработчиков)**
 - **mypy / аннотации типов**
- **Django Styleguide in the Wild**
- Дополнительные ресурсы
- Вдохновение

Что это такое?

Приветственная программа

Это руководство по стилю **Django**, созданное нами, сотрудниками **HackSoft**.

Несколько важных заметок о нем:

Он основан на многолетнем опыте и многих проектах **Django**, как больших, так и малых.

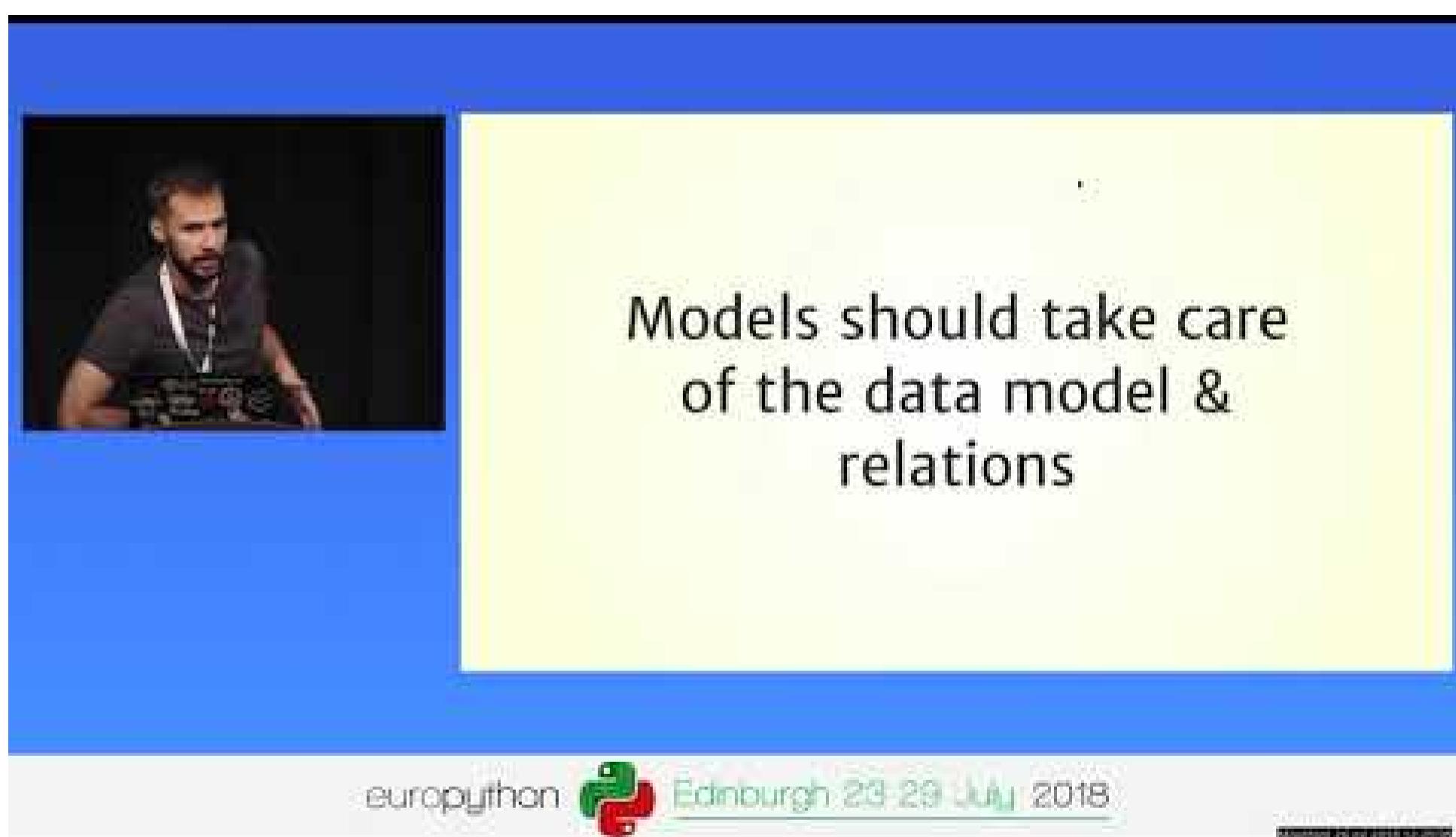
Он прагматичен. Все вещи, упомянутые здесь, проверены в производстве.

Это мнение. Именно так мы создаем приложения на **Django**.

Это не единственный способ. Существуют и другие способы построения и структурирования **Django**-проектов, которые могут сделать работу за вас.

У нас есть **Django-Styleguide-Example**, чтобы показать большую часть **styleguide** в реальном проекте.

Вы можете посмотреть статью Радослава Георгиева "[Структура Django для масштаба и долговечности](#)", чтобы понять философию, лежащую в основе **styleguide**:



Как его использовать?

Когда речь заходит о **Django Styleguide**, есть **3** основных способа его использования:

- Строго следовать всему, что здесь написано.
- Выбирайте то, что имеет смысл для вас, основываясь на вашем конкретном контексте.
- Не следовать ничему, что здесь написано.

Мы рекомендуем пункт номер **2**:

- Прочитайте руководство по стилю.
- Решите, что будет работать лучше для вас.
- Адаптируйте для вашего конкретного случая.

Обзор

Основу руководства по стилю **Django** можно кратко изложить следующим образом:

В Django бизнес-логика должна жить в:

- **Services** - функции, которые в основном занимаются записью в базу данных.
- **Selectors** - функции, которые в основном заботятся о получении данных из базы данных.
- Свойства модели (за некоторыми исключениями).
- **clean** метод модели для дополнительных проверок (за некоторыми исключениями).

В Django бизнес-логика не должна жить в:

- API и представлениях.
- СерIALIZаторах и формах.
- Теги форм.
- Метод сохранения модели.
- Пользовательские менеджеры или наборы запросов.
- Сигналы.

Свойства модели в сравнении с селекторами:

- Если свойство охватывает несколько отношений, лучше использовать селектор.
- Если свойство нетривиально и может легко вызвать проблему **N + 1** запросов при сериализации, лучше, чтобы это был селектор.

Общая идея состоит в том, чтобы "разделить проблемы", чтобы эти проблемы могли быть сопровождаемыми / тестируемыми.

Почему бы и нет?

размышления Почему бы не разместить бизнес-логику в **API** / представлениях / сериализаторах / формах?

Полагаясь на общие **API** / представления, с комбинацией сериализаторов и форм, вы делаете две основные вещи:

1. Фрагментирует бизнес-логику в нескольких местах, что делает очень трудным отслеживание потока данных.
2. Скрывает вещи от вас. Для того чтобы что-то изменить, вам нужно знать внутреннюю работу абстракции, которую вы используете.

Generic APIs & Views, в сочетании с сериализаторами и формами, действительно отлично подходит для прямого случая "**CRUD** для модели".

По нашему опыту, пока что этот простой случай случается редко. И как только вы покидаете счастливый путь **CRUD**, все начинает идти наперекосяк.

А когда все начинает запутываться, вам нужно больше "коробок", чтобы лучше организовать ваш код.

Это руководство по стилю направлено на то, чтобы:

- Дать вам эти "коробки".
- Помочь вам придумать свои собственные "коробки" для вашего конкретного контекста и потребностей.

Почему бы не поместить бизнес-логику в пользовательские менеджеры и/или наборы запросов(querysets**)?**

На самом деле это хорошая идея, и вы можете внедрить пользовательские менеджеры и наборы запросов, которые могут предоставлять лучший **API**, адаптированный к вашему домену.

Но пытаться поместить всю бизнес-логику в пользовательский менеджер - не самая лучшая идея, поскольку это связано со следующим:

- 1. Бизнес-логика имеет свой собственный домен, который не всегда напрямую сопоставлен с вашей моделью данных (моделями).**
- 2. Бизнес-логика чаще всего охватывает несколько моделей, поэтому очень сложно выбрать, где что разместить.**
 - Допустим, у вас есть пользовательская часть логики, которая затрагивает модели **A, B, C** и **D**. Куда вы ее поместите?
- 3. Там могут быть дополнительные обращения к сторонним системам. Не стоит размещать их в пользовательских методах менеджера.**

Идея заключается в том, чтобы позволить вашему домену жить отдельно от вашей модели данных и слоя **API**.

Если мы возьмем идею создания пользовательских наборов запросов/менеджеров и объединим ее с идеей позволить домену жить отдельно, мы получим то, что мы называем "слой сервисов".

Сервисы могут быть функциями, классами, модулями или чем угодно, что имеет смысл для вашего конкретного случая.

Учитывая все это, пользовательские менеджеры и наборы запросов являются очень мощными инструментами, и их следует использовать для создания лучших интерфейсов для ваших моделей.

размышления Почему бы не поместить бизнес-логику в сигналы?

Из всех доступных вариантов, пожалуй, этот быстро приведет вас в очень плохое место:

- Сигналы - это отличный инструмент для соединения вещей, которые не должны знать друг о друге, но вы хотите, чтобы они были связаны.
- Сигналы также являются отличным инструментом для обработки недействительности кэша вне уровня бизнес-логики.
- Если мы начнем использовать сигналы для вещей, которые сильно связаны между собой, мы просто сделаем связь более неявной и усложним отслеживание потока данных.

Вот почему мы рекомендуем использовать сигналы для очень специфических случаев использования, но в целом мы не рекомендуем использовать их для структурирования домена / бизнес-слоя.

Cookie Cutter

Мы рекомендуем начинать каждый новый проект с какого-либо кукинг-конструктора. Наличие правильной структуры с самого начала приносит свои плоды.

Несколько примеров:

- Вы можете использовать проект **Styleguide-Example** в качестве отправной точки.
- Вы также можете использовать **cookiecutter-django**, так как он содержит тонну полезного материала.
- Или вы можете создать что-то подходящее для вашего случая и превратить это в проект **cookiecutter**.

Модели

Модели должны заботиться о модели данных и не более того.

Базовая модель

Хорошой идеей является определение базовой модели (**BaseModel**), которую вы можете наследовать.

Обычно такие поля, как **created_at** и **updated_at**, являются идеальными кандидатами для включения в **BaseModel**.

Там же можно определить первичный ключ. Потенциальным кандидатом для этого является поле **UUIDField**.

Вот пример **BaseModel**:

```
from django.db import models
from django.utils import timezone

class BaseModel(models.Model):
    created_at = models.DateTimeField(db_index=True, default=timezone.now)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True
```

Затем, когда вам понадобится новая модель, просто наследуйте **BaseModel**:

```
class SomeModel(BaseModel):
    pass
```

Валидация - **clean** и **full_clean**

Давайте рассмотрим пример модели:

```
class Course(BaseModel):
    name = models.CharField(unique=True, max_length=255)

    start_date = models.DateField()
    end_date = models.DateField()

    def clean(self):
        if self.start_date >= self.end_date:
            raise ValidationError("End date cannot be before start date")
```

Мы определяем метод **clean** модели, потому что хотим убедиться, что в нашу базу данных попадут хорошие данные.

Теперь, чтобы метод **clean** был вызван, кто-то должен вызвать **full_clean** на экземпляре нашей модели перед сохранением.

Наша рекомендация - сделать это в сервисе, прямо перед вызовом **clean**:

```
def course_create(*, name: str, start_date: date, end_date: date) -> Course:
    obj = Course(name=name, start_date=start_date, end_date=end_date)
    obj.full_clean()
    obj.save()
    return obj
```

Это также хорошо работает с **Django admin**, потому что формы, используемые там, будут вызывать **full_clean** на экземпляре.

У нас есть несколько общих правил, когда следует добавлять валидацию в метод **clean** модели:

- Если мы проверяем на основе нескольких нереляционных полей модели.
- Если сама валидация достаточно проста.

Валидацию следует перенести на сервисный уровень, если:

- 1.Логика проверки является более сложной.
- 2.Требуется охват отношений и получение дополнительных данных.

Вполне нормально иметь валидацию и в чистом виде, и в сервисе, но мы склонны переносить все в сервис, если это так.

Валидация - ограничения

Как было предложено в этом выпуске, если вы можете сделать валидацию, используя ограничения **Django**, то вы должны стремиться к этому.

Меньше кода для написания, меньше кода для поддержки, база данных позаботится о данных, даже если они будут вставлены из другого места.

Давайте посмотрим на пример!

```
class Course(BaseModel):  
    name = models.CharField(unique=True, max_length=255)  
  
    start_date = models.DateField()  
    end_date = models.DateField()  
  
class Meta:  
    constraints = [  
        models.CheckConstraint(  
            name="start_date_before_end_date",  
            check=Q(start_date__lt=F("end_date"))  
    ]
```

Теперь, если мы попытаемся создать новый объект через **course.save()** или через **Course.objects.create(...)**, мы получим **IntegrityError**, а не **ValidationError**.

На самом деле это может быть недостатком подхода, потому что теперь нам придется иметь дело с **IntegrityError**, который не всегда имеет лучшее сообщение об ошибке.

Документация **Django** по ограничениям довольно скучная, поэтому вы можете ознакомиться со следующими статьями Адама Джонсона, где приведены примеры их использования:

1. Использование контрольных ограничений **Django** для обеспечения установки только одного поля
2. Выбор полей в **Django** не ограничивает ваши данные
3. Использование контрольных ограничений **Django** для предотвращения самозаполнения

Использование контрольных ограничений Django для обеспечения установки только одного поля

Ранее я рассказывал об использовании класса **CheckConstraint** в Django для проверки полей с вариантами выбора и процентных полей, которые составляют **100%**. Вот еще один случай использования.

В проекте, над которым я недавно работал, хранятся "баллы", которые могут иметь одно типовое значение: целое число, десятичная дробь или длительность. Поскольку у них разные типы, они хранятся в отдельных колонках в базе данных. Поэтому для одного балла может быть заполнен только один из этих столбцов (**не null**).

Вы можете решить эту проблему, используя наследование модели. Недостатком этого способа является то, что при использовании конкретного или абстрактного наследования вам потребуется несколько таблиц. Это также много работы для одного различающегося поля.

Вместо этого мы остановились на единой модели с несколькими столбцами значений, только один из которых должен быть установлен. Модель выглядит примерно так:

```
from django.db import models

class ScoreType(models.IntegerChoices):
    POINTS = 1, "Points"
    DURATION = 2, "Duration"

class Score(models.Model):
    type = models.IntegerField(choices=ScoreType.choices)
    value_points = models.IntegerField()
    value_duration = models.DurationField()
```

(**IntegerChoices** - это один из новых типов перечислений **Django 3.0**).

Если тип **ScoreType.POINTS**, следует установить столбец **value_points**. Аналогично, если тип **ScoreType.DURATION**, следует установить столбец **value_duration**.

Как бы вы ни были уверены в том, что наш код **Python** удовлетворяет этому ограничению, если вы не заставите базу данных соблюдать его, одна ошибка может разрушить это предположение. Например, вы можете случайно написать запрос вида:

```
Score.objects.update(value_points=1, value_duration=dt.timedelta(1))  
(Импорт datetime как dt.)
```

Такие плохие данные могут иметь самые разные непредвиденные последствия. Если вы обнаружите, что такие плохие данные были созданы в течение некоторого времени, на их распутывание может уйти много времени. Лучше всего предотвратить это на ранней стадии, если это возможно!

Для этого вы можете добавить **CheckConstraint**, чтобы убедиться, что заполненный столбец значений соответствует типу. Это будет выглядеть следующим образом:

```
from django.db import models  
  
class ScoreType(models.IntegerChoices):  
    POINTS = 1, "Points"  
    DURATION = 2, "Duration"  
  
class Score(models.Model):  
    type = models.IntegerField(choices=ScoreType.choices)  
    value_points = models.IntegerField(null=True)  
    value_duration = models.DurationField(null=True)  
  
    class Meta:  
        constraints = [  
            models.CheckConstraint(  
                name="%(app_label)s_%(class)s_value_matches_type",  
                check=  
                    (models.Q(  
                        type=ScoreType.POINTS,  
                        value_points_isnull=False,  
                        value_duration_isnull=True,) |  
                     models.Q(  
                        type=ScoreType.DURATION,  
                        value_points_isnull=True,  
                        value_duration_isnull=False,))),]
```

Ограничение задается с помощью объектов **Q**, которые принимают те же аргументы, что и **filter()**, для ограничения случаев. Здесь мы, по сути, перечисляем два допустимых случая и объединяем их с помощью оператора побитового **OR |** в **Python**. **Q** не может использовать для этого обычный оператор **or** из-за ограничений **Python**.

После создания миграций и применения этой миграции вы можете проверить это ограничение. Вы можете создавать легитимные экземпляры **Score** без проблем:

```
In [3]: Score.objects.create(type=ScoreType.POINTS,value_points=1337)
Out[3]: <Score: Score object (1)>
```

```
In [4]: Score.objects.create(type=ScoreType.DURATION,value_duration=dt.timedelta(seconds=1234))
Out[4]: <Score: Score object (2)>
```

Но если вы попытаетесь сохранить плохие данные, вы получите ошибку **IntegrityError**:

```
In [5]: Score.objects.create(type=ScoreType.POINTS, value_points=1337,
value_duration=dt.timedelta(seconds=1234))
...

```

```
IntegrityError: CHECK constraint failed: score_value_matches_type
```

Это отлично работает.

Вы также можете объединить это с прокси-моделями, чтобы разделить Оценки на основе типа. Я не играл с этим полностью, но это выглядит как жизнеспособная альтернатива многотабличному наследованию. Можно даже добавить помощника для автоматической генерации классов.

Вы можете сделать простую реализацию "Очков" без автоматической генерации, как здесь:

```
class PointsScoreManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(type=ScoreType.POINTS)

class PointsScore(Score):
    objects = PointsScoreManager()
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.type = ScoreType.POINTS

class Meta:
    proxy = True
```

Реализацию **DURATION** нужно просто немного скопировать-вставить-заменить.

Здесь используется пользовательский менеджер, который возвращает только экземпляры нужного типа. А также переопределяет `_init_` для принудительного определения типа. Я уверен, что есть еще несколько переопределений, которые нужно добавить, чтобы все прошло гладко, но вы поняли идею.

Финал

Исходный код примера проекта, использованного в этой заметке, включая бонусную автогенерацию ограничения проверки, можно найти здесь, на [GitHub](#).

Надеюсь, эта статья поможет вам в дальнейшем применении **CheckConstraint**,

-Адам

Выбор полей в Django не ограничивает ваши данные.

Этот пост - PSA о несколько неинтуитивном способе работы `Field.choices` в Django.

Возьмем определение этой модели Django:

```
from django.db import models

class Status(models.TextChoices):
    UNPUBLISHED = "UN", "Unpublished"
    PUBLISHED = "PB", "Published"

class Book(models.Model):
    status = models.CharField(max_length=2, choices=Status.choices, default=Status.UNPUBLISHED)
    def __str__(self):
        return f"{self.id} - {Status(self.status).label}"
```

Если мы откроем оболочку `manage.py` для работы с ними, мы сможем легко создать книгу с заданным выбором статуса:

In [1]: `from core.models import Status, Book`

In [2]: `Book.objects.create(status=Status.UNPUBLISHED)`
Out[2]: <Book: 1 - Unpublished>

Список `choices` ограничивает значение `status` во время проверки модели в Python:

```
In [3]: book = Book.objects.get(id=1)
In [4]: book.status = 'republished'
In [5]: book.full_clean()
-----
ValidationError                                     Traceback (most recent call last)
<ipython-input-7-e64237e0a92a> in <module>
----> 1 book.full_clean()

.../django/db/models/base.py in full_clean(self, exclude, validate_unique)
1220
1221     if errors:
-> 1222         raise ValidationError(errors)
1223
1224     def clean_fields(self, exclude=None):

ValidationError: {'status': ["Value 'republished' is not a valid choice."]}
```

Это отлично подходит для **ModelForms** и других случаев, использующих валидацию. Пользователи не могут выбирать недопустимые варианты и получать сообщения о том, что не так.

К сожалению, нам, разработчикам, по-прежнему легко записать эти недействительные данные в базу данных:

In[6]: book.save()

Упс!

Также можно обновить все наши экземпляры до недействительного состояния в одной строке:

In[8]: Book.objects.update(status="republished")

Out[8]: 1

Так в чем же дело? Почему **Django** позволяет нам объявить набор вариантов, которые мы хотим, чтобы поле принимало, но затем позволяет нам легко обойти это?

Ну, валидация модели в **Django** предназначена в основном для форм. Он полагается на то, что другие пути кода в вашем приложении "знают, что делают".

Если мы хотим предотвратить это, то наиболее общее решение - заставить саму базу данных отбрасывать плохие данные. Это не только сделает ваш код **Django** более надежным, но и любые другие приложения, использующие базу данных, тоже будут использовать ограничения.

Мы можем добавить такие ограничения с помощью класса **CheckConstraint**, добавленного в **Django 2.2**. Для нашей модели нам нужно определить и назвать один фильтр **CheckConstraint** в **Meta.constraints: 0508**

```
class Book(models.Model):
    status = models.CharField(
        max_length=2,
        choices=Status.choices,
        default=Status.UNPUBLISHED,
    )

    def __str__(self):
        return f"{self.id} - {Status(self.status).label}"

class Meta:
    constraints = [
        models.CheckConstraint(
            name=f"%({app_label})s_{(class)s_status_valid}",
            check=models.Q(status__in=Status.values),
        )
    ]
```

Объект **Q** представляет собой одно выражение, которое мы передаем в **Model.objects.filter()**. Ограничения могут иметь любое количество логических операций над полями в текущей модели. Сюда входят все виды поиска, сравнения между полями и функции базы данных.

Запустив **makemigrations**, мы получим миграцию, которая выглядит следующим образом:

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [
        ("core", "0001_initial"),
    ]

    operations = [
        migrations.AddConstraint(
            model_name="book",
            constraint=models.CheckConstraint(
                check=models.Q(status__in=["UN", "PB"]),
                name="% (app_label)s_% (class)s_status_valid",
            ),
        ),
    ]
]
```

Если мы попытаемся применить это, пока база данных содержит недопустимые данные, это приведет к неудаче:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0002_book_status_valid...Traceback (most recent call last):
...
  File "/.../django/db/backends/sqlite3/base.py", line 396, in execute
    return Database.Cursor.execute(self, query, params)
django.db.utils.IntegrityError: CHECK constraint failed: status_valid
```

Если мы очистим эти данные вручную и повторим попытку, она пройдет:

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0002_book_status_valid... OK
```

С этого момента база данных не позволит нам вставить недействительные строки или обновить действительные строки на недействительные:

```
In [4]: book.save()
...
/.../django/db/backends/sqlite3/base.py in execute(self, query, params)
    394         return Database.Cursor.execute(self, query)
    395     query = self.convert_query(query)
--> 396     return Database.Cursor.execute(self, query, params)
    397
    398 def executemany(self, query, param_list):
    IntegrityError: CHECK constraint failed: status_valid

In [5]: Book.objects.update(status='republished')
...
/.../django/db/backends/sqlite3/base.py in execute(self, query, params)
    394         return Database.Cursor.execute(self, query)
    395     query = self.convert_query(query)
--> 396     return Database.Cursor.execute(self, query, params)
    397
    398 def executemany(self, query, param_list):
    IntegrityError: CHECK constraint failed: status_valid
```

Отлично!

В настоящее время **Django** не имеет способа показать эти **IntegrityErrors** пользователям при валидации модели. Ничто не может поймать и превратить их в **ValidationErrors**, которые могут нести сообщения для пользователей. Согласно документации:

В общем случае ограничения не проверяются во время **full_clean()** и не вызывают **ValidationErrors**.

Существует открытый тикет **#30581** для улучшения этого.

В нашем случае, поскольку мы все еще используем **choice**, это нормально. Валидация уже не позволит пользователям выбирать недопустимые статусы.

Для более сложных ограничений мы, возможно, захотим продублировать логику в **Python** с помощью пользовательского валидатора.

Дальнейшее чтение

Пост Джеймса Кука о сохранении в сравнении с **full_clean**.

Документация **MariaDB** по ограничению **CHECK**.

Финал

Проверочные ограничения действительно хороши. Ограничение данных на самом низком уровне дает нам самые сильные гарантии их качества.

Надеюсь, этот пост поможет вам рассмотреть возможность их использования,

-Адам

Использование контрольных ограничений в Django для предотвращения самофокусировки

Еще один способ использования класса **CheckConstraint** от Django для обеспечения достоверности данных. Основано на моем ответе на вопрос на форуме **Django**.

Представьте, что у нас есть модель пользователя, в которую мы хотели бы внедрить модель "следования" в социальных сетях. Пользователи могут следить за другими пользователями, чтобы получать обновления на нашем сайте. Мы хотели бы убедиться, что пользователи не следят за собой, поскольку это потребует особой осторожности во всем нашем коде.

Мы можем заблокировать следование за собой на уровне пользовательского интерфейса, но всегда есть риск, что мы случайно включим его. Например, мы можем добавить функцию "импортировать контакты", которая разрешает самофолловинг.

Как и во всех других постах этой серии, лучший способ предотвратить появление плохих данных - это заблокировать их в базе данных.

Настройка связей

Чтобы добавить отношения последователей, мы будем использовать **ManyToManyField**. По умолчанию **ManyToManyField** создает скрытый класс модели для хранения отношений. Поскольку мы хотим настроить нашу модель, добавив дополнительное ограничение, нам нужно использовать аргумент **through**, чтобы определить наш собственный видимый класс модели.

Мы можем определить первую версию этого следующим образом:

```
from django.db import models

class User(models.Model):
    ...
    followers = models.ManyToManyField(
        to="self",
        through="Follow",
        related_name="following",
        symmetrical=False,
    )

class Follow(models.Model):
    from_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="+")
    to_user = models.ForeignKey(User, on_delete=models.CASCADE, related_name="+")

    class Meta:
        constraints = [
            models.UniqueConstraint(
                name="%(app_label)s_%(class)s_unique_relationships",
                fields=["from_user", "to_user"],
            ),
        ]

```

Примечание:

Мы используем **to="self"**, чтобы определить, что связь идет от пользователя к самому себе. **Django** называет это рекурсивным отношением.

Мы используем строковый формат **through**, потому что мы определяем **User** перед **Follow**. Мы могли бы сначала определить **Follow**, но тогда нам пришлось бы использовать строки для определения **User** в его определении.

Мы объявляем отношения асимметричными с симметрией=False. Если Алиса следует за Бобом, это не значит, что Боб следует за Алисой. Если бы наши отношения были взаимными в стиле "запрос друга", мы бы сделали отношения симметричными.

Класс **Follow** использует два внешних ключа для связи между связанными пользователями. **ManyToManyField** автоматически использует первый внешний ключ в качестве "источника" отношений, а второй - в качестве назначения.

Возможно, **Follow** может иметь третий внешний ключ **User**, например, для отслеживания другого пользователя, который предложил следовать за ним. В этом случае нам нужно будет использовать **ManyToManyField.through_fields**, чтобы указать, какие внешние ключи действительно формируют связь.

Мы уже добавили ограничение к модели - **UniqueConstraint**, чтобы гарантировать, что между пользователями существует только одна связь. Без этого между, например, Алисой и Бобом могло бы существовать несколько отношений, и было бы непонятно, что это значит. Это копирует стандартную скрытую сквозную модель **Django**.

Мы используем интерполяцию строк в имени нашего ограничения, чтобы присвоить его к нашей модели. Это предотвращает столкновения имен с ограничениями в других моделях. Базы данных имеют только одно пространство имен для ограничений во всех таблицах, поэтому нам нужно быть осторожными.

После добавления миграции и ее запуска мы можем опробовать наши модели в оболочке:

In [1]: `from example.core.models import User`

In [2]: `user1 = User.objects.create()`

In [3]: `user2 = User.objects.create()`

In [4]: `user1.following.add(user2)`

In [5]: `user1.following.add(user1)`

In [6]: `user1.following.all()`

Out[6]: `<QuerySet [<User: User object (2)>, <User: User object (1)>]>`

Обратите внимание, что мы все еще могли заставить пользователя `user1` следовать за самим собой. Давайте теперь это исправим.

Предотвращение самопоследований

Пришло время настроить наше ограничение **CheckConstraint**.

Сначала мы добавим ограничение в **Meta.constraints**:

```
class Follow(models.Model):
    ...
    class Meta:
        constraints = [
            ...,
            models.CheckConstraint(
                name=f'{app_label}s_{class.__name__}_prevent_self_follow',
                check=~models.Q(from_user=models.F('to_user')),
            ),
        ]
    ]
```

Наша проверка здесь объявляет, что поле `from_user` не равно полю `to_user`. Мы используем объект `F()` для представления поля `to_user`. Мы отрицаем условие с помощью легко заметного оператора тильда (`~`), который превращает `==` в `!=`. Таким образом, в синтаксисе **Python** условие звучит как `from_user != to_user`.

Во-вторых, мы запускаем **makemigrations** для создания новой миграции:

```
$ ./manage.py makemigrations core
Migrations for 'core':
  example/core/migrations/0003_auto_20210225_0339.py
    - Create constraint core_follow_prevent_self_follow on model follow
```

Мы проверяем правильность миграции:

```
from django.db import migrations, models

class Migration(migrations.Migration):

    dependencies = [
        ("core", "0002_auto_20210225_0320"),
    ]

    operations = [
        migrations.AddConstraint(
            model_name="follow",
            constraint=models.CheckConstraint(
                check=models.Q(_negated=True, from_user=models.F("to_user")),
                name="core_follow_prevent_self_follow",
            ),
        ),
    ]
]
```

Пока все хорошо!

В-третьих, мы можем добавить тест, который проверяет, что наше ограничение проверки работает так, как задумано:

```
from django.db import IntegrityError
from django.test import TestCase

from example.core.models import Follow, User

class FollowTests(TestCase):
    def test_no_self_follow(self):
        user = User.objects.create()
        constraint_name = "core_follow_prevent_self_follow"
        with self.assertRaisesMessage(IntegrityError, constraint_name):
            Follow.objects.create(from_user=user, to_user=user)
```

Тест пытается выполнить самостоятельное выполнение и гарантирует, что это вызовет ошибку базы данных, которая содержит имя нашего ограничения.

В-четвертых, мы должны рассмотреть, как наша миграция будет работать с существующими плохими данными. Если мы попытаемся выполнить миграцию, пока существуют отношения самопоследования, миграция завершится с ошибкой **IntegrityError**:

```
$ ./manage.py migrate
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0003_auto_20210225_0339...Traceback (most recent call last):
  File "/.../django/db/backends/utils.py", line 84, in _execute
    ...
  File "/.../django/db/backends/sqlite3/base.py", line 413, in execute
    return Database.Cursor.execute(self, query, params)
django.db.utils.IntegrityError: CHECK constraint failed: core_follow_prevent_self
_follow
```

Мы можем удалить все самоподдерживающиеся отношения в миграции с помощью операции **RunPython**. Это позволяет нам использовать **ORM** для изменения базы данных. Мы можем добавить функцию уровня модуля для **RunPython** и затем ссылаться на нее в операции:

```
from django.db import migrations, models

def forwards_func(apps, schema_editor):
    Follow = apps.get_model("core", "Follow")
    db_alias = schema_editor.connection.alias
    Follow.objects.using(db_alias).filter(from_user=models.F("to_user")).delete()

class Migration(migrations.Migration):

    dependencies = [
        ("core", "0002_auto_20210225_0320"),
    ]

    operations = [
        migrations.RunPython(
            code=forwards_func,
            reverse_code=migrations.RunPython.noop,
            elidable=True,
        ),
        migrations.AddConstraint(
            model_name="follow",
            constraint=models.CheckConstraint(
                check=models.Q(_negated=True, from_user=models.F("to_user")),
                name="core_follow_prevent_self_follow",
            ),
        ),
    ]
```

Примечание:

Мы используем тот же шаблон **forwards_func**, что и в документации **RunPython**.

Мы получаем версию модели **Follow** через **apps.get_model()**, а не импортируем последнюю версию. Использование последней версии может привести к ошибке, поскольку она может ссылаться на поля, которые еще не были добавлены в базу данных.

Мы также используем текущий псевдоним базы данных. Это лучше сделать, даже если наш проект использует только одну базу данных, на случай, если в будущем их станет несколько.

Мы объявляем **reverse_code** как **по-оп**, чтобы миграция была обратимой. Обратный переход не сможет восстановить удаленные отношения самопоследовательности, потому что мы нигде их не резервируем.

Мы объявляем операцию как устранимую. Это означает, что **Django** может отказаться от операции при уничтожении истории миграции. Это всегда стоит учитывать при написании операций **RunPython** или **RunSQL**, так как это поможет вам делать меньшие и более быстрые сквоши.

Теперь мы можем запустить миграцию без проблем:

```
$ ./manage.py migrate
Operations to perform:
  Apply all migrations: core
Running migrations:
  Applying core.0003_auto_20210225_0339... OK
```

Отлично!

В-пятых, мы должны пройтись по нашему пользовательскому интерфейсу и **API** и проверить, что возможность самостоятельного следования нигде не раскрыта. Если у нас есть **API**, мы хотим убедиться, что он возвращает разумное сообщение об ошибке при попытке самостоятельного следования, поскольку в противном случае он просто завершится с ошибкой **IntegrityError**.

Финал

Вы не можете следить за собой, но вы можете следить за мной в **Twitter**,

-Адам

Свойства

Свойства модели - это отличный способ быстрого доступа к производному значению экземпляра модели.

Например, давайте рассмотрим свойства **has_started** и **has_finished** нашей модели **Course**:

```
from django.utils import timezone
from django.core.exceptions import ValidationError

class Course(BaseModel):
    name = models.CharField(unique=True, max_length=255)

    start_date = models.DateField()
    end_date = models.DateField()

    def clean(self):
        if self.start_date >= self.end_date:
            raise ValidationError("End date cannot be before start date")

    @property
    def has_started(self) -> bool:
        now = timezone.now()

        return self.start_date <= now.date()

    @property
    def has_finished(self) -> bool:
        now = timezone.now()
        return self.end_date <= now.date()
```

Эти свойства удобны, потому что теперь мы можем ссылаться на них в сериализаторах или использовать их в шаблонах.

У нас есть несколько общих правил, когда следует добавлять свойства в модель:

- Если нам нужно простое производное значение, основанное на нереляционных полях модели, добавьте **@property** для этого.
- Если расчет производного значения достаточно прост.

Свойства должны быть чем-то другим (сервисом, селектором, утилитой) в следующих случаях:

1. Если нам нужно охватить несколько отношений или получить дополнительные данные.
2. Если вычисления более сложные.

Имейте в виду, что эти правила расплывчаты, потому что часто важен контекст. Используйте свои лучшие суждения!

Методы

Методы модели также являются очень мощным инструментом, который можно построить поверх свойств.

Рассмотрим пример с методом `is_within(self, x)`:

```
from django.core.exceptions import ValidationError
from django.utils import timezone
```

```
class Course(BaseModel):
    name = models.CharField(unique=True, max_length=255)

    start_date = models.DateField()
    end_date = models.DateField()

    def clean(self):
        if self.start_date >= self.end_date:
            raise ValidationError("End date cannot be before start date")

    @property
    def has_started(self) -> bool:
        now = timezone.now()

        return self.start_date <= now.date()

    @property
    def has_finished(self) -> bool:
        now = timezone.now()

        return self.end_date <= now.date()

    def is_within(self, x: date) -> bool:
        return self.start_date <= x <= self.end_date
```

is_within не может быть свойством, так как требует аргумента. Поэтому вместо него используется метод.

Еще один отличный способ использования методов в моделях - это использование их для установки атрибутов, когда установка одного атрибута всегда должна сопровождаться установкой другого атрибута с производным значением.

Например

```
from django.utils.crypto import get_random_string
from django.conf import settings
from django.utils import timezone
```

```
class Token(BaseModel):
    secret = models.CharField(max_length=255, unique=True)
    expiry = models.DateTimeField(blank=True, null=True)

    def set_new_secret(self):
        now = timezone.now()
        self.secret = get_random_string(255)
        self.expiry = now + settings.TOKEN_EXPIRY_DELTA

    return self
```

Теперь мы можем смело вызывать **set_new_secret**, который будет выдавать правильные значения для **secret** и **expiry**.

У нас есть несколько общих правил, когда следует добавлять методы в модель:

1. Если нам нужно простое производное значение, требующее аргументов, основанное на нереляционных полях модели, добавьте для этого метод.
2. Если расчет производного значения достаточно прост.
3. Если установка одного атрибута всегда требует установки значений других атрибутов, используйте для этого метод.

Модели должны быть чем-то другим (сервисом, селектором, утилитой) в следующих случаях:

1. Если нам нужно охватить несколько отношений или получить дополнительные данные.
2. Если вычисления более сложные.

Имейте в виду, что эти правила расплывчаты, потому что часто важен контекст. Используйте свои лучшие суждения!

Тестирование

Модели нужно тестировать только в том случае, если в них есть что-то дополнительное - например, валидация, свойства или методы.

Вот пример:

```
from datetime import timedelta

from django.test import TestCase
from django.core.exceptions import ValidationError
from django.utils import timezone

from project.some_app.models import Course

class CourseTests(TestCase):
    def test_course_end_date_cannot_be_before_start_date(self):
        start_date = timezone.now()
        end_date = timezone.now() - timedelta(days=1)

        course = Course(start_date=start_date, end_date=end_date)

        with self.assertRaises(ValidationError):
            course.full_clean()
```

Здесь необходимо отметить несколько моментов:

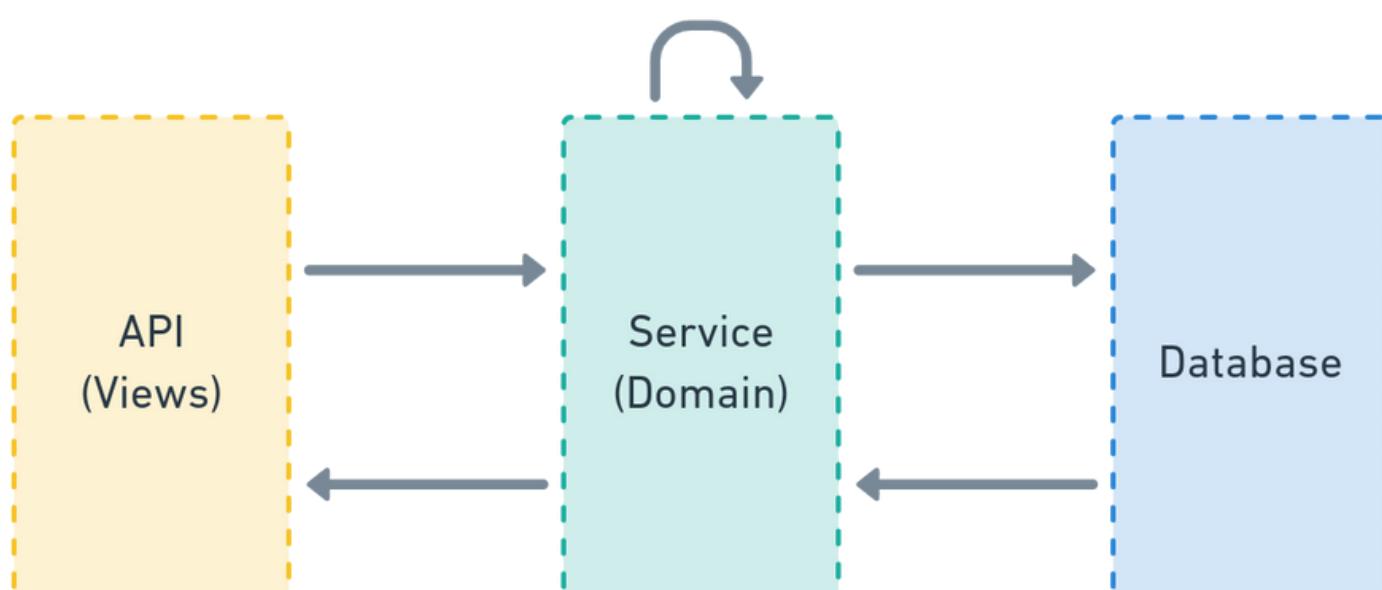
1. Мы утверждаем, что при вызове **full_clean** возникнет ошибка валидации.
2. Мы вообще не обращаемся к базе данных, поскольку в этом нет необходимости. Это может ускорить выполнение некоторых тестов.

Сервисы

Сервисы - это место, где живет бизнес-логика.

Сервисный слой говорит на специфическом языке домена программного обеспечения, может обращаться к базе данных и другим ресурсам, а также взаимодействовать с другими частями вашей системы.

Вот очень простая диаграмма, позиционирующая сервисный уровень в наших приложениях **Django**:



Сервис может быть:

- Простая функция.
- Классом.
- Целый модуль.
- Все, что имеет смысл в вашем случае.

В большинстве случаев сервис может быть простой функцией, которая:

- Находится в модуле <ваше_приложение>/services.py.
- Принимает аргументы, состоящие только из ключевых слов, если не требует ни одного или одного аргумента.
- Аннотирована по типу (даже если вы не используете **тур** в данный момент).
- Взаимодействует с базой данных, другими ресурсами и другими частями вашей системы.
- Выполняет бизнес-логику - от простого создания модели до сложных сквозных задач, вызова внешних сервисов и задач.

Пример - сервис на основе функций

Пример сервиса, который создает пользователя:

```
def user_create(  
    *,  
    email: str,  
    name: str  
) -> User:  
    user = User(email=email)  
    user.full_clean()  
    user.save()  
  
    profile_create(user=user, name=name)  
    confirmation_email_send(user=user)  
  
    return user
```

Как видно, эта служба вызывает **2** другие службы - **profile_create** и **confirmation_email_send**.

В этом примере все, что связано с созданием пользователя, находится в одном месте и может быть отслежено.

Пример - сервис на основе класса

Кроме того, мы можем иметь сервисы, основанные на классах, что является причудливым способом сказать - обернуть логику в класс.

Вот пример, взятый прямо из [Django Styleguide Example](#), связанный с загрузкой файлов:

```
class FileStandardUploadService:
```

```
    """
```

This also serves as an example of a service class,
which encapsulates 2 different behaviors (create & update) under a namespace.

Meaning, we use the class here for:

1. The namespace

2. The ability to reuse ` _infer_file_name_and_type` (which can also be an util)

```
"""
```

```
def __init__(self, user: BaseUser, file_obj):  
    self.user = user  
    self.file_obj = file_obj
```

```
def _infer_file_name_and_type(self, file_name: str = "", file_type: str = "") -> Tuple[str, str]:
```

```
    if not file_name:  
        file_name = self.file_obj.name
```

```
    if not file_type:  
        guessed_file_type, encoding = mimetypes.guess_type(file_name)
```

```
        if guessed_file_type is None:  
            file_type = ""  
        else:  
            file_type = guessed_file_type
```

```
    return file_name, file_type
```

```
@transaction.atomic
```

```
def create(self, file_name: str = "", file_type: str = "") -> File:
```

```
    _validate_file_size(self.file_obj)
```

```
    file_name, file_type = self._infer_file_name_and_type(file_name, file_type)
```

```
    obj = File(  
        file=self.file_obj,  
        original_file_name=file_name,  
        file_name=file_generate_name(file_name),  
        file_type=file_type,  
        uploaded_by=self.user,  
        upload_finished_at=timezone.now())
```

```
    obj.full_clean()
```

```
    obj.save()
```

```
    return obj
```

```
@transaction.atomic
```

```
def update(self, file: File, file_name: str = "", file_type: str = "") -> File:
```

```
    _validate_file_size(self.file_obj)
```

```
    file_name, file_type = self._infer_file_name_and_type(file_name, file_type)
```

```
    file.file = self.file_obj  
    file.original_file_name = file_name  
    file.file_name = file_generate_name(file_name)  
    file.file_type = file_type  
    file.uploaded_by = self.user  
    file.upload_finished_at = timezone.now()  
    file.full_clean()  
    file.save()  
    return file
```

Как указано в комментарии, мы используем этот подход по двум основным причинам:

- Пространство имен. У нас есть единое пространство имен для создания и обновления.
- Повторное использование логики `_infer_file_name_and_type`.

Вот как используется эта служба:

```
class FileDirectUploadApi(ApiAuthMixin, APIView):  
    def post(self, request):  
        service = FileDirectUploadService(  
            user=request.user,  
            file_obj=request.FILES["file"]  
        )  
        file = service.create()  
        return Response(data={"id": file.id}, status=status.HTTP_201_CREATED)
```

И

```
@admin.register(File)  
class FileAdmin(admin.ModelAdmin):  
    # ... other code here ...  
    # https://github.com/HackSoftware/Django-Styleguide-  
    # Example/blob/master/styleguide_example/files/admin.py  
  
    def save_model(self, request, obj, form, change):  
        try:  
            cleaned_data = form.cleaned_data  
  
            service = FileDirectUploadService(  
                file_obj=cleaned_data["file"],  
                user=cleaned_data["uploaded_by"]  
            )  
  
            if change:  
                service.update(file=obj)  
            else:  
                service.create()  
        except ValidationError as exc:  
            self.message_user(request, str(exc), messages.ERROR)
```

Кроме того, использование сервисов на основе классов - хорошая идея для "потоков" - вещей, которые проходят через несколько этапов.

Например, эта служба представляет собой "поток прямой загрузки файлов", с началом и завершением (и дополнительно):

```
# https://github.com/HackSoftware/Django-Styleguide-  
Example/blob/master/styleguide_example/files/services.py  
  
class FileDirectUploadService:  
    """  
    This also serves as an example of a service class,  
    which encapsulates a flow (start & finish) + one-off action (upload_local) into a namespace.  
  
    Meaning, we use the class here for:  
    1. The namespace  
    """  
  
    def __init__(self, user: BaseUser):  
        self.user = user  
  
    @transaction.atomic  
    def start(self, *, file_name: str, file_type: str) -> Dict[str, Any]:  
        file = File(  
            original_file_name=file_name,  
            file_name=file_generate_name(file_name),  
            file_type=file_type,  
            uploaded_by=self.user,  
            file=None  
        )  
        file.full_clean()  
        file.save()  
        upload_path = file_generate_upload_path(file, file.file_name)  
        """  
        We are doing this in order to have an associated file for the field.  
        """  
        file.file = file.file.field.attr_class(file, file.file.field, upload_path)  
        file.save()  
  
        presigned_data: Dict[str, Any] = {}  
  
        if settings.FILE_UPLOAD_STORAGE == FileUploadStorage.S3:  
            presigned_data = s3_generate_presigned_post(  
                file_path=upload_path, file_type=file.file_type  
            )  
  
        else:  
            presigned_data = {  
                "url": file_generate_local_upload_url(file_id=str(file.id)),  
            }  
  
        return {"id": file.id, **presigned_data}  
  
    @transaction.atomic  
    def finish(self, *, file: File) -> File:  
        # Potentially, check against user  
        file.upload_finished_at = timezone.now()  
        file.full_clean()  
        file.save()  
        return file
```

Соглашение об именовании

Принятие названий зависит от вашего вкуса. Полезно иметь что-то последовательное во всем проекте.

Если мы возьмем пример выше, наш сервис будет называться **user_create**. Шаблон такой - <сущность>_<действие>.

Это то, что мы предпочитаем в проектах **HackSoft**. Поначалу это кажется странным, но у него есть несколько приятных особенностей:

Пространство имен. Легко обнаружить все сервисы, начинающиеся с **user_**, и это хорошая идея - поместить их в модуль **users.py**.

Удобство использования. Другими словами, если вы хотите увидеть все действия для конкретной сущности, просто перейдите по запросу **user_**.

Модули

Если у вас достаточно простое приложение **Django** с кучей сервисов, все они могут счастливо жить в модуле **service.py**.

Но когда все становится большим, вы можете захотеть разделить **services.py** на папку с подмодулями, в зависимости от различных поддоменов, с которыми вы имеете дело в вашем приложении.

Например, допустим, у нас есть приложение для аутентификации, где в модуле **services.py** есть один подмодуль, который работает с **jwt**, и один подмодуль, который работает с **oauth**.

Структура может выглядеть следующим образом:

services

```
|__ __init__.py  
|__ jwt.py  
|__ oauth.py
```

Здесь есть множество вариантов:

- Вы можете сделать танец импорта-экспорта в **services/__init__.py**, так что вы можете импортировать из **project.authentication.services** везде.
- Вы можете создать папку-модуль **jwt/__init__.py** и поместить код туда.
- В принципе, структура зависит от вас. Если вы чувствуете, что пришло время перестроить структуру и провести рефакторинг - сделайте это.

Селекторы

В большинстве наших проектов мы различаем понятия "загрузка данных в базу данных" и "получение данных из базы данных":

- Сервисы заботятся о том, чтобы подталкивать данные.
- Селекторы занимаются извлечением.
- Селекторы можно рассматривать как "подслой" сервисов, который специализируется на получении данных.
- Если эта идея вам не очень нравится, вы можете просто иметь сервисы для обоих "видов" операций.

Селектор подчиняется тем же правилам, что и сервис.

Например, в модуле `<your_app>/selectors.py` мы можем использовать следующее:

```
def user_list(*, fetched_by: User) -> Iterable[User]:  
    user_ids = user_get_visible_for(user=fetched_by)  
    query = Q(id__in=user_ids)  
    return User.objects.filter(query)
```

Как вы видите, `user_get_visible_for` - это еще один селектор.

Вы можете возвращать `querysets`, или списки, или все, что имеет смысл в вашем конкретном случае.

Тестирование

Поскольку сервисы содержат нашу бизнес-логику, они являются идеальным кандидатом для тестов.

Если вы решите покрыть слой сервисов тестами, у нас есть несколько общих правил, которым следует придерживаться:

Тесты должны исчерпывающим образом охватывать бизнес-логику.

Тесты должны затрагивать базу данных - создание и чтение из нее.

Тесты должны имитировать вызовы асинхронных задач и все, что выходит за пределы проекта.

При создании необходимого состояния для конкретного теста можно использовать комбинацию из:

- Фейки (мы рекомендуем использовать `faker`)
- Другие сервисы для создания необходимых объектов.
- Специальные тестовые utilиты и вспомогательные методы.
- `Factory` (Мы рекомендуем использовать `factory_boy`)
- Обычные вызовы `Model.objects.create()`, если теории еще не внедрены в проект.
- В общем, все, что вам больше подходит.

Давайте посмотрим на наш сервис из примера:

```
from django.contrib.auth.models import User
from django.core.exceptions import ValidationError

from project.payments.selectors import items_get_for_user
from project.payments.models import Item, Payment
from project.payments.tasks import payment_charge


def item_buy(
    *,
    item: Item,
    user: User,
) -> Payment:
    if item in items_get_for_user(user=user):
        raise ValidationError(f'Item {item} already in {user} items.')

    payment = Payment.objects.create(
        item=item,
        user=user,
        successful=False
    )

    payment_charge.delay(payment_id=payment.id)

    return payment
```

Сервис:

- Вызывает селектор для валидации.
- Создает объект.
- Задерживает выполнение задачи.

Это наши тесты:

```
from unittest.mock import patch
from django.test import TestCase
from django.contrib.auth.models import User
from django.core.exceptions import ValidationError
from django_styleguide.payments.services import item_buy
from django_styleguide.payments.models import Payment, Item

class ItemBuyTests(TestCase):
    def setUp(self):
        self.user = User.objects.create_user(username='Test User')
        self.item = Item.objects.create(
            name='Test Item',
            description='Test Item description',
            price=10.15
        )

    @patch('project.payments.services.items_get_for_user')
    def test_buying_item_that_is_already_bought_fails(self, items_get_for_user_mock):
        """
        Since we already have tests for `items_get_for_user`,
        we can safely mock it here and give it a proper return value.
        """
        items_get_for_user_mock.return_value = [self.item]

        with self.assertRaises(ValidationError):
            item_buy(user=self.user, item=self.item)

    @patch('project.payments.services.payment_charge.delay')
    def test_buying_itemCreates_a_payment_and_calls_charge_task(
        self,
        payment_charge_mock
    ):
        self.assertEqual(0, Payment.objects.count())

        payment = item_buy(user=self.user, item=self.item)

        self.assertEqual(1, Payment.objects.count())
        self.assertEqual(payment, Payment.objects.first())

        self.assertFalse(payment.successful)

        payment_charge_mock.assert_called()
```

API и Serializers

При использовании сервисов и селекторов все ваши **API** должны выглядеть просто и одинаково.

Когда мы создаем новые **API**, мы следуем этим общим правилам:

- На каждую операцию должно приходиться по **1 API**. Это означает, что для **CRUD** на модели нужно иметь **4 API**.
- Наследовать от самого простого **APIView** или **GenericAPIView**.
- Избегайте более абстрактных классов, поскольку они склонны управлять вещами через сериализаторы, а мы хотим делать это через сервисы и селекторы.
- Не делайте бизнес-логику в вашем **API**.
- Вы можете выполнять выборку объектов / манипулирование данными в **API** (потенциально, вы можете извлечь это куда-то еще).
- Если вы вызываете какой-то сервис в вашем **API**, вы можете извлечь извлечение объектов / манипуляции с данными в **some_service_parse**.
- В общем, держите **API** как можно более простыми. Они являются интерфейсом к вашей основной бизнес-логике.

Когда мы говорим об **API**, нам нужен способ сериализации данных - как входящих, так и исходящих.

Вот наши правила для сериализации **API**:

- Должен быть выделенный входной сериализатор и выделенный выходной сериализатор.
- Входной сериализатор заботится о входящих данных.
- Выходной сериализатор заботится о выходных данных.
- Что касается сериализации, используйте любую абстракцию, которая вам подходит.

Если вы используете сериализаторы **DRF**, вот наши правила:

- Сериализатор должен быть вложен в **API** и называться либо **InputSerializer**, либо **OutputSerializer**.
- Мы предпочитаем, чтобы оба сериализатора наследовались от более простого **Serializer** и избегали использования **ModelSerializer**.
- Это вопрос предпочтения и выбора. Если **ModelSerializer** работает для вас хорошо, используйте его.
- Если вам нужен вложенный сериализатор, используйте **inline_serializer util**.
- Как можно меньше используйте сериализаторы повторно.
- Повторное использование сериализаторов может привести к неожиданному поведению, когда что-то изменится в базовых сериализаторах.

Соглашение об использовании имен

Для наших **API** мы используем следующее соглашение об именовании:
<Entity><Action>Api.

Вот несколько примеров: **UserCreateApi**, **UserSendResetPasswordApi**,
UserDeactivateApi и т.д.

Class-based vs. Function-based

Это в основном зависит от личных предпочтений, поскольку вы можете достичь одинаковых результатов с помощью обоих подходов.

Мы придерживаемся следующих предпочтений:

1. По умолчанию выбирайте **APIs / представления**, основанные на классах.
2. Если все остальные предпочитают функции и чувствуют себя комфортно с ними, используйте **API/представления** на основе функций.

Для нас дополнительные преимущества использования классов для **API / представлений** заключаются в следующем:

- Вы можете наследовать **BaseApi** или добавлять миксины.
- Если вы используете **API / представления** на основе функций, вам нужно сделать то же самое, но с декораторами.
- Класс создает пространство имен, где вы можете вложить что-то (атрибуты, методы и т.д.).
- Дополнительная настройка **API** может быть выполнена с помощью атрибутов класса.
- В случае **API/представлений**, основанных на функциях, вам нужно сложить декораторы.

Вот пример с классом, наследующим **BaseApi**:

```
class SomeApi(BaseApi):  
    def get(self, request):  
        data = something()  
        return Response(data)
```

Вот пример с функцией, использующей декоратор **base_api** (реализация зависит от ваших потребностей)

```
@base_api(["GET"])  
def some_api(request):  
    data = something()  
    return Response(data)
```

List APIs

Обычный

Смертельно простой **API** для работы со списками должен выглядеть следующим образом:

```
from rest_framework.views import APIView
from rest_framework import serializers
from rest_framework.response import Response

from styleguide_example.users.selectors import user_list
from styleguide_example.users.models import BaseUser
```

```
class UserListApi(APIView):
    class OutputSerializer(serializers.Serializer):
        id = serializers.CharField()
        email = serializers.CharField()

    def get(self, request):
        users = user_list()
        data = self.OutputSerializer(users, many=True).data
        return Response(data)
```

Помните, что по умолчанию этот **API** является общедоступным.
Аутентификация зависит от вас.

Фильтры + пагинация

На первый взгляд, это сложно, поскольку наши **API** наследуют обычный **APIView** от **DRF**, а фильтрация и пагинация встроены в общие(**generics**):

1. [DRF Filtering](#)
2. [DRF Pagination](#)

Поэтому мы используем следующий подход:

- Селекторы обеспечивают фактическую фильтрацию.
- **API** заботятся о сериализации параметров фильтра.
- Если вам нужны некоторые из общих пагинаций, предоставляемых **DRF**, **API** должен позаботиться об этом.
- Если вам нужна другая пагинация или вы реализуете ее самостоятельно, либо добавьте новый слой для обработки пагинации, либо позвольте селектору сделать это за вас.

Давайте рассмотрим пример, в котором мы полагаемся на пагинацию, предоставляемую **DRF**:

```
from rest_framework.views import APIView
from rest_framework import serializers

from styleguide_example.api.mixins import ApiErrorsMixin
from styleguide_example.api.pagination import get_paginated_response,
LimitOffsetPagination

from styleguide_example.users.selectors import user_list
from styleguide_example.users.models import BaseUser

class UserListApi(ApiErrorsMixin, APIView):
    class Pagination(LimitOffsetPagination):
        default_limit = 1

    class FilterSerializer(serializers.Serializer):
        id = serializers.IntegerField(required=False)
        # Important: If we use BooleanField, it will default to False
        is_admin = serializers.NullBooleanField(required=False)
        email = serializers.EmailField(required=False)

    class OutputSerializer(serializers.Serializer):
        id = serializers.CharField()
        email = serializers.CharField()
        is_admin = serializers.BooleanField()

    def get(self, request):
        # Make sure the filters are valid, if passed
        filters_serializer = self.FilterSerializer(data=request.query_params)
        filters_serializer.is_valid(raise_exception=True)

        users = user_list(filters=filters_serializer.validated_data)

        return get_paginated_response(
            pagination_class=self.Pagination,
            serializer_class=self.OutputSerializer,
            queryset=users,
            request=request,
            view=self
        )
```

Когда мы смотрим на **API**, мы можем определить несколько вещей:

1. Есть **FilterSerializer**, который позаботится о параметрах запроса. Если мы не сделаем это здесь, нам придется сделать это в другом месте, а сериализаторы **DRF** отлично справляются с этой задачей.
2. Мы передаем фильтры селектору **user_list**.
3. Мы используем утилиту **get_paginated_response**, чтобы вернуть ... постраничный ответ.

Теперь давайте посмотрим на селектор:

```
import django_filters

from styleguide_example.users.models import BaseUser

class BaseUserFilter(django_filters.FilterSet):
    class Meta:
        model = BaseUser
        fields = ('id', 'email', 'is_admin')

def user_list(*, filters=None):
    filters = filters or {}
    qs = BaseUser.objects.all()
    return BaseUserFilter(filters, qs).qs
```

Как вы видите, мы используем мощную библиотеку **django-filter**.

Ключевым моментом здесь является то, что за фильтрацию отвечает селектор. Вы всегда можете использовать что-то другое в качестве абстракции фильтрации. Для большинства случаев **django-filter** более чем достаточно.

Наконец, давайте посмотрим на **get_paginated_response**:

```
from rest_framework.response import Response

def get_paginated_response(*, pagination_class, serializer_class, queryset, request, view):
    paginator = pagination_class()

    page = paginator.paginate_queryset(queryset, request, view=view)

    if page is not None:
        serializer = serializer_class(page, many=True)
        return paginator.get_paginated_response(serializer.data)

    serializer = serializer_class(queryset, many=True)

    return Response(data=serializer.data)
```

Это, по сути, код, извлеченный из DRF.

То же самое относится к **LimitOffsetPagination**:

```

from collections import OrderedDict

from rest_framework.pagination import LimitOffsetPagination as _LimitOffsetPagination
from rest_framework.response import Response


class LimitOffsetPagination(_LimitOffsetPagination):
    default_limit = 10
    max_limit = 50

    def get_paginated_data(self, data):
        return OrderedDict([
            ('limit', self.limit),
            ('offset', self.offset),
            ('count', self.count),
            ('next', self.get_next_link()),
            ('previous', self.get_previous_link()),
            ('results', data)
        ])

    def get_paginated_response(self, data):
        """
        We redefine this method in order to return `limit` and `offset`.
        This is used by the frontend to construct the pagination itself.
        """

        return Response(OrderedDict([
            ('limit', self.limit),
            ('offset', self.offset),
            ('count', self.count),
            ('next', self.get_next_link()),
            ('previous', self.get_previous_link()),
            ('results', data)
        ]))

```

По сути, мы сделали обратную разработку общих **API**.

Опять же, если вам нужно что-то другое для пагинации, вы всегда можете реализовать это и использовать таким же образом. Бывают случаи, когда селектор должен позаботиться о пагинации. Мы подходим к этим случаям так же, как и к фильтрации.

Код примера **API** списка с фильтрами и пагинацией вы можете найти в проекте [Styleguide Example](#).

Detail API

Вот пример:

```
class CourseDetailApi(SomeAuthenticationMixin, APIView):
    class OutputSerializer(serializers.Serializer):
        id = serializers.CharField()
        name = serializers.CharField()
        start_date = serializers.DateField()
        end_date = serializers.DateField()

    def get(self, request, course_id):
        course = course_get(id=course_id)

        serializer = self.OutputSerializer(course)

        return Response(serializer.data)
```

Create API

Вот пример:

```
class CourseCreateApi(SomeAuthenticationMixin, APIView):
    class InputSerializer(serializers.Serializer):
        name = serializers.CharField()
        start_date = serializers.DateField()
        end_date = serializers.DateField()

    def post(self, request):
        serializer = self.InputSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        course_create(**serializer.validated_data)

        return Response(status=status.HTTP_201_CREATED)
```

Update API

Вот пример:

```
class CourseUpdateApi(SomeAuthenticationMixin, APIView):
    class InputSerializer(serializers.Serializer):
        name = serializers.CharField(required=False)
        start_date = serializers.DateField(required=False)
        end_date = serializers.DateField(required=False)

    def post(self, request, course_id):
        serializer = self.InputSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)

        course_update(course_id=course_id, **serializer.validated_data)

        return Response(status=status.HTTP_200_OK)
```

Получение объектов

Когда наши **API** получают идентификатор объекта, возникает вопрос: Где мы должны получить этот объект?

У нас есть несколько вариантов:

1. Мы можем передать этот объект в сериализатор, который имеет **PrimaryKeyRelatedField** (или **SlugRelatedField**).
2. Мы можем выполнить некую выборку объекта в **API** и передать объект сервису или селектору.
3. Мы можем передать **id** в сервис / селектор и выполнить выборку объекта там.

Какой подход мы используем - это вопрос контекста проекта и предпочтений.

Как правило, мы получаем объекты на уровне **API**, используя специальную утилиту **get_object**:

```
def get_object(model_or_queryset, **kwargs):
    """
    Reuse get_object_or_404 since the implementation supports both Model && queryset.
    Catch Http404 & return None
    """
    try:
        return get_object_or_404(model_or_queryset, **kwargs)
    except Http404:
        return None
```

Это очень простая утилита, которая обрабатывает исключение и возвращает вместо него **None**.

Что бы вы ни делали, убедитесь, что все работает последовательно.

Вложенные сериализаторы

Если вам необходимо использовать вложенный сериализатор, вы можете поступить следующим образом:

```
class Serializer(serializers.Serializer):
    weeks = inline_serializer(many=True, fields={
        'id': serializers.IntegerField(),
        'number': serializers.IntegerField(),
    })
```

Реализацию **inline_serializer** можно найти здесь, в репозитории [Styleguide-Example](#).

Продвинутая сериализация

Иногда конечный результат работы **API** может быть довольно сложным. Иногда мы хотим оптимизировать запросы, которые мы делаем, и сама оптимизация может быть довольно сложной.

Попытка использовать только **OutputSerializer** в этом случае может ограничить наши возможности.

В таких случаях мы можем реализовать сериализацию вывода в виде функции и получить необходимые нам оптимизации там, а не в селекторе.

Возьмем в качестве примера этот **API**:

```
class SomeGenericFeedApi(BaseApi):
    def get(self, request):
        feed = some_feed_get(
            user=request.user,
        )

        data = some_feed_serialize(feed)

    return Response(data)
```

В этом сценарии **some_feed_get** отвечает за возврат списка элементов **feed** (это могут быть объекты **ORM**, могут быть просто идентификаторы, может быть все, что вам подходит).

И мы хотим переложить сложность сериализации этого **feed** оптимальным образом на функцию сериализатора - **some_feed_serialize**.

Это означает, что нам не придется делать ненужные предварительные замеры и оптимизацию в **some_feed_get**.

Вот пример **some_feed_serialize**:

```
class FeedItemSerializer(serializers.Serializer):
    ... some fields here ...
    calculated_field = serializers.IntegerField(source="_calculated_field")

def some_feed_serialize(feed: List[FeedItem]):
    feed_ids = [feed_item.id for feed_item in feed]

    # Refetch items with more optimizations
    # Based on the relations that are going in
    objects = FeedItem.objects.select_related(
        # ... as complex as you want ...
    ).prefetch_related(
        # ... as complex as you want ...
    ).filter(
        id__in=feed_ids
    ).order_by(
        "-some_timestamp"
    )

    some_cache = get_some_cache(feed_ids)

    result = []

    for feed_item in objects:
        # An example, adding additional fields for the serializer
        # That are based on values outside of our current object
        # This may be some optimization to save queries
        feed_item._calculated_field = some_cache.get(feed_item.id)

        result.append(FeedItemSerializer(feed_item).data)

return result
```

Как вы можете видеть, это довольно общий пример, но идея проста:

1. Повторная выборка данных с необходимыми объединениями и предварительными выборками.
2. Получить или создать кэши в памяти, которые позволяют вам сэкономить на запросах для определенных вычисляемых значений.
3. Верните результат, готовый для ответа **API**.

Несмотря на то, что это обозначено как "продвинутая сериализация", паттерн действительно мощный и может быть использован для всех сериализаций.

Такие функции сериализатора обычно живут в модуле **serializers.py**, в соответствующем приложении **Django**.

Urls

Обычно мы организуем наши ссылки так же, как мы организуем наши API - 1 url на API, что означает 1 url на действие.

Общее правило - разделять ссылки с разных доменов в их собственном списке **domain_patterns** и включать в **urlpatterns**.

Вот пример с API, приведенный выше:

```
from django.urls import path, include
```

```
from project.education.apis import (
```

```
    CourseCreateApi,
```

```
    CourseUpdateApi,
```

```
    CourseListApi,
```

```
    CourseDetailApi,
```

```
    CourseSpecificActionApi,
```

```
)
```

```
course_patterns = [
```

```
    path('', CourseListApi.as_view(), name='list'),
```

```
    path('<int:course_id>', CourseDetailApi.as_view(), name='detail'),
```

```
    path('create/', CourseCreateApi.as_view(), name='create'),
```

```
    path('<int:course_id>/update/', CourseUpdateApi.as_view(), name='update'),
```

```
    path(
```

```
        '<int:course_id>/specific-action/',
```

```
        CourseSpecificActionApi.as_view(),
```

```
        name='specific-action'
```

```
),
```

```
]
```

```
urlpatterns = [
```

```
    path('courses/', include((course_patterns, 'courses'))),
```

```
]
```

Подобное разделение url может дать вам дополнительную гибкость при переносе отдельных доменных шаблонов в отдельные модули, особенно для действительно больших проектов, где часто возникают конфликты слияния в **urls.py**.

Теперь, если вам нравится видеть всю структуру дерева url, вы можете сделать именно это, не извлекая специфические переменные для url, которые вы включаете.

Вот пример из нашего **Django Styleguide Example**:

```
from django.urls import path, include

from styleguide_example.files.apis import (
    FileDirectUploadApi,
    FilePassThruUploadStartApi,
    FilePassThruUploadFinishApi,
    FilePassThruUploadLocalApi,
)

urlpatterns = [
    path(
        "upload/",
        include([
            path(
                "direct/",
                FileDirectUploadApi.as_view(),
                name="direct"
            ),
            path(
                "pass-thru/",
                include([
                    path(
                        "start/",
                        FilePassThruUploadStartApi.as_view(),
                        name="start"
                    ),
                    path(
                        "finish/",
                        FilePassThruUploadFinishApi.as_view(),
                        name="finish"
                    ),
                    path(
                        "local/<str:file_id>/",
                        FilePassThruUploadLocalApi.as_view(),
                        name="local"
                    )
                ]),
                "pass-thru"
            )),
            "upload"
        ])
]
```

Некоторые люди предпочитают первый способ, другие - видимую древовидную структуру. Это зависит от вас и вашей команды.

Settings

Когда дело доходит до настроек **Django**, мы обычно следуем структуре папок из **cookiecutter-django**, с небольшими изменениями:

- Мы отделяем настройки, специфичные для **Django**, от других настроек.
- Все должно быть включено в файл **base.py**.
- Не должно быть ничего, что включено только в **production.py**.
- То, что должно работать только в **production**, контролируется с помощью переменных окружения.

Вот структура папок нашего проекта **Styleguide-Example**:

config

```
├── __init__.py
└── django
    ├── __init__.py
    ├── base.py
    ├── local.py
    ├── production.py
    └── test.py
    └── settings
        ├── __init__.py
        ├── celery.py
        ├── cors.py
        ├── sentry.py
        └── sessions.py
    └── urls.py
    └── env.py
    └── wsgi.py
└── asgi.py
```

В **config/django** мы помещаем все, что связано с **Django**:

- **base.py** содержит большую часть настроек и импортирует все остальное из **config/settings**
- **production.py** импортирует из **base.py** и затем перезаписывает некоторые специфические настройки для **production**.
- **test.py** импортирует из **base.py** и затем перезаписывает некоторые специфические настройки для запуска тестов.
- Его следует использовать в качестве модуля настроек в **pytest.ini**.
- **local.py** импортирует из **base.py** и может перезаписывать некоторые специфические настройки для локальной разработки.
- Если вы хотите использовать его, укажите на **local** в файле **manage.py**. В противном случае используйте **base.py**

В `config/settings` мы помещаем все остальное:

- Конфигурация **Celery**.
- Конфигурации сторонних разработчиков.
- и т.д.

Это дает хорошее разделение модулей.

Кроме того, у нас обычно есть `config/env.py` со следующим кодом:

```
import environ
```

```
env = environ.Env()
```

И затем, когда нам нужно будет прочитать что-то из окружающей среды, мы импортируем таким образом:

```
from config.env import env
```

Обычно в конце модуля `base.py` мы импортируем все из `config/settings`:

```
from config.settings.cors import * # noqa
from config.settings.sessions import * # noqa
from config.settings.celery import * # noqa
from config.settings.sentry import * # noqa
```

Префикс переменных окружения с помощью **DJANGO_**

Во многих примерах вы увидите, что переменные окружения обычно имеют префикс **DJANGO_**. Это очень полезно, когда рядом с вашим приложением **Django** работают другие приложения и читают из того же окружения.

Мы склонны ставить префикс **DJANGO_** только для **DJANGO_SETTINGS_MODULE** и **DJANGO_DEBUG** и не ставим никаких других префиксов.

Это в основном зависит от личных предпочтений. Просто убедитесь, что вы последовательны в этом.

Интеграции

Поскольку все должно быть импортировано в `base.py`, но иногда мы не хотим настраивать определенную интеграцию для локальной разработки, мы использовали следующий подход:

- Настройки, специфичные для интеграции, помещаются в `config/settings/some_integration.py`
- Там всегда есть параметр под названием **USE_SOME_INTEGRATION**, который считывается из окружения и по умолчанию имеет значение **False**.
- Если значение равно **True**, то происходит считывание других настроек и отказ, если что-то не присутствует в окружении.

Например, давайте посмотрим на **config/settings/sentry.py**:

```
from config.env import env

SENTRY_DSN = env('SENTRY_DSN', default="")

if SENTRY_DSN:
    import sentry_sdk
    from sentry_sdk.integrations.django import DjangoIntegration
    from sentry_sdk.integrations.celery import CeleryIntegration

# ... we proceed with sentry settings here ...
# View the full file here - https://github.com/HackSoftware/Styleguide-Example/blob/master/config/settings/sentry.py
```

Чтение из .env

Наличие локального **.env** - это хороший способ предоставить значения для ваших настроек.

И хорошо то, что **django-environ** предоставляет вам способ сделать это:

That's in the beginning of base.py

```
import os
```

```
from config.env import env, environ
```

```
# Build paths inside the project like this: os.path.join(BASE_DIR, ...)
BASE_DIR = environ.Path(__file__) - 3
```

```
env.read_env(os.path.join(BASE_DIR, ".env"))
```

Теперь вы можете иметь файл **.env** (но это не обязательно) в корне вашего проекта и поместить туда значения для ваших настроек.

Здесь стоит упомянуть две вещи:

- Не размещайте **.env** в вашем контроле исходников, так как это приведет к утечке учетных данных.
- Лучше поместите **.env.example** с пустыми значениями для всего, чтобы новые разработчики могли понять, что используется.

Ошибки и обработка исключений

Если вам нужен код, перейдите к проекту **Styleguide-Example** -
https://github.com/HackSoftware/Styleguide-Example/blob/master/styleguide_example/api/exception_handlers.py.

Обработка ошибок и исключений - это большая тема, и довольно часто - детали специфичны для конкретного проекта.

Поэтому мы разделим тему на две части - общие рекомендации, а затем некоторые специфические подходы к обработке ошибок.

Наши общие рекомендации таковы:

- Знайте, как работает обработка исключений (мы приведем контекст для **Django Rest Framework**).
- Опишите, как будут выглядеть ваши ошибки **API**.
- Знать, как изменить поведение обработки исключений по умолчанию.

Далее следуют некоторые конкретные подходы:

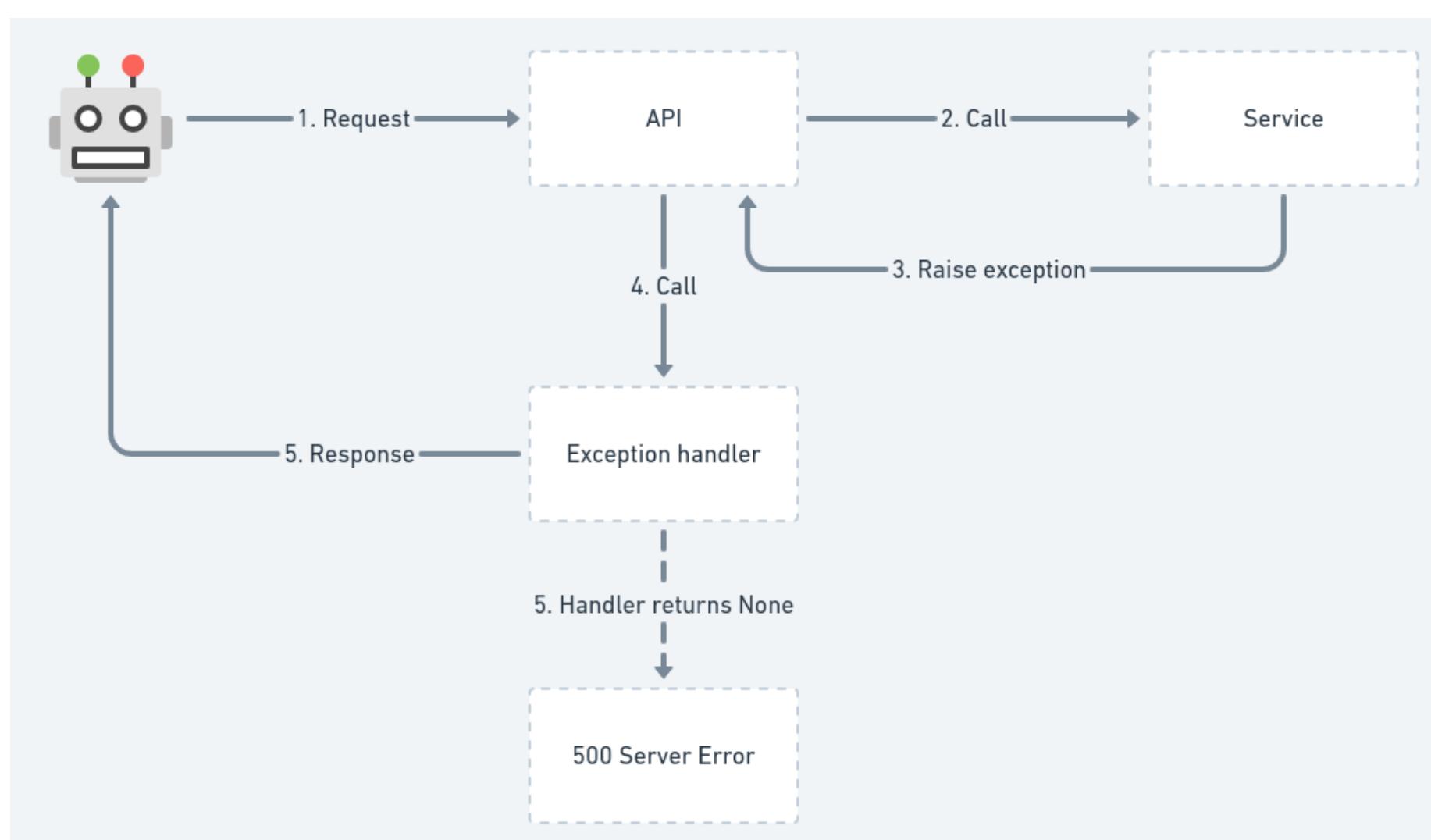
1. Использовать исключения DRF по умолчанию с очень небольшими изменениями.

2. Подход, предложенный HackSoft.

Как работает обработка исключений (в контексте DRF)

В DRF есть отличное руководство по обработке исключений, поэтому обязательно прочитайте его - <https://www.django-rest-framework.org/api-guide/exceptions/>.

Кроме того, вот аккуратная диаграмма с обзором процесса:



В принципе, если обработчик исключений не может обработать данное исключение и возвращает **None**, это приведет к необработанному исключению и ошибке **500 Server Error**. Это часто хорошо, потому что вы не будете закрывать ошибки, на которые нужно обратить внимание.

Есть некоторые особенности, на которые нам нужно обратить внимание.

ValidationError от DRF

Например, если мы просто поднимем ошибку

`rest_framework.exceptions.ValidationError` следующим образом:

```
from rest_framework import exceptions

def some_service():
    raise ValidationError("Error message here.")
```

Полезная нагрузка ответа будет выглядеть следующим образом:

```
[  
    "Some message"  
]
```

Это выглядит странно, потому что если мы сделаем это следующим образом:

```
from rest_framework import exceptions
```

```
def some_service():
    raise exceptions.ValidationError({"error": "Some message"})
```

Полезная нагрузка ответа будет выглядеть следующим образом:

```
{  
    "error": "Some message"  
}
```

В основном это то, что мы передали в качестве детали **ValidationError**. Но это структура данных, отличная от исходного массива.

Теперь, если мы решим поднять еще одно из встроенных исключений **DRF**:

```
from rest_framework import exceptions
```

```
def some_service():
    raise exceptions.NotFound()
```

Полезная нагрузка ответа будет выглядеть следующим образом:

```
{  
    "detail": "Not found."  
}
```

Это полностью отличается от того, что мы видели в поведении **ValidationError**, и это может вызвать проблемы.

Пока что поведение **DRF** по умолчанию может дать нам:

- Массив.
- Словарь.
- Конкретный {"деталь": "что-то"} результат.

Поэтому, если нам нужно использовать поведение **DRF** по умолчанию, мы должны позаботиться об этом несоответствии.

Django's ValidationError

Теперь, обработка исключений по умолчанию в DRF не очень хорошо сочетается с **ValidationError** в Django.

Вот этот фрагмент кода:

```
from django.core.exceptions import ValidationError as DjangoValidationError

def some_service():
    raise DjangoValidationError("Some error message")
```

Приведет к необработанному исключению, вызывающему **500 Server Error**.

Это также произойдет, если эта ошибка **ValidationError** возникнет, например, при валидации модели:

```
def some_service():
    user = BaseUser()
    user.full_clean() # Throws ValidationError
    user.save()
```

Это также приведет к **500 Server Error**.

Если мы хотим начать обрабатывать это, как если бы это была **rest_framework.exceptions.ValidationError**, нам нужно развернуть свой собственный обработчик исключений:

```
from django.core.exceptions import ValidationError as DjangoValidationError

from rest_framework.views import exception_handler
from rest_framework.serializers import as_serializer_error
from rest_framework import exceptions

def custom_exception_handler(exc, ctx):
    if isinstance(exc, DjangoValidationError):
        exc = exceptions.ValidationError(as_serializer_error(exc))

    response = exception_handler(exc, ctx)

    # If unexpected error occurs (server error, etc.)
    if response is None:
        return response

    return response
```

В основном это реализация по умолчанию, с добавлением этого фрагмента кода:

```
if isinstance(exc, DjangoValidationError):
    exc = exceptions.ValidationError(as_serializer_error(exc))
```

Так как нам нужно сопоставить `django.core.exceptions.ValidationError` и `rest_framework.exceptions.ValidationError`, мы используем для этого параметр `as_serializer_error` из `DRF`, который используется внутри сериализаторов.

Благодаря этому, мы можем теперь иметь `Django's ValidationError`, который хорошо работает с обработчиком исключений `DRF`.

Опишите, как будут выглядеть ваши ошибки API.

Это очень важно и должно быть сделано как можно раньше в любом проекте.

В основном это описание интерфейса ваших ошибок **API** - как ошибка будет выглядеть в виде ответа **API**?

Это очень специфично для конкретного проекта, вы можете использовать некоторые популярные **API** для вдохновения:

- **Stripe** - <https://stripe.com/docs/api/errors>

В качестве примера, мы можем решить, что наши ошибки будут выглядеть следующим образом:

- **4**** и **5**** коды статуса для различных типов ошибок.
- Каждая ошибка будет представлять собой словарь с одним ключом `message`, содержащим сообщение об ошибке.

```
{
  "message": "Some error message here"
}
```

Это достаточно просто:

- **400** будет использоваться для ошибок валидации.
- **401** - для ошибок авторизации.
- **403** - для ошибок разрешения.
- **404** для ошибок "не найдено".
- **429** для ошибок дросселирования.
- **500** для ошибок сервера (нужно быть осторожным, чтобы не заглушить исключение, вызывающее **500**, и всегда сообщать об этом в сервисах типа **Sentry**).

Опять же, это зависит от вас, и это специфично для проекта. Мы предложим что-то похожее для одного из конкретных подходов.

Знайте, как изменить поведение обработки исключений по умолчанию.

Это также важно, потому что когда вы решите, как будут выглядеть ваши ошибки, вам нужно будет реализовать это как пользовательскую обработку исключений.

Мы уже приводили пример этого в параграфе выше, рассказывая о **Django's ValidationError**.

Мы также приведем дополнительные примеры в следующих разделах.

Подход 1 - Использование исключений DRF по умолчанию, с очень небольшими изменениями.

Обработка ошибок в **DRF** хороша. Было бы здорово, если бы конечный результат всегда был последовательным. Именно эти небольшие модификации мы и собираемся сделать.

Мы хотим, чтобы в итоге ошибки всегда выглядели именно так:

```
{  
    "detail": "Some error"  
}
```

или:

```
{  
    "detail": ["Some error", "Another error"]  
}
```

или:

```
{  
    "detail": { "key": "... some arbitrary nested structure ..." }  
}
```

В общем, убедитесь, что у нас всегда есть словарь с подробным ключом.

Кроме того, мы хотим обрабатывать и ошибку проверки достоверности (**ValidationError**) **Django**.

Чтобы добиться этого, вот как будет выглядеть наш пользовательский обработчик исключений:

```
from django.core.exceptions import ValidationError as DjangoValidationError, PermissionDenied
from django.http import Http404

from rest_framework.views import exception_handler
from rest_framework import exceptions
from rest_framework.serializers import as_serializer_error


def drf_default_with_modifications_exception_handler(exc, ctx):
    if isinstance(exc, DjangoValidationError):
        exc = exceptions.ValidationError(as_serializer_error(exc))

    if isinstance(exc, Http404):
        exc = exceptions.NotFound()

    if isinstance(exc, PermissionDenied):
        exc = exceptions.PermissionDenied()

    response = exception_handler(exc, ctx)

    # If unexpected error occurs (server error, etc.)
    if response is None:
        return response

    if isinstance(exc.detail, (list, dict)):
        response.data = {
            "detail": response.data
        }

    return response
```

Мы как бы повторяем оригинальный обработчик исключений, так что после этого мы можем работать с **APIException** (ищем детали).

Теперь запустим набор тестов:

Code:

```
def some_service():
    raise DjangoValidationError("Some error message")
```

Response:

```
{
    "detail": {
        "non_field_errors": [
            "Some error message"
        ]
    }
}
```

Code:

```
from django.core.exceptions import PermissionDenied

def some_service():
    raise PermissionDenied()
```

Response:

```
{
    "detail": "You do not have permission to perform this action."
}
```

Code:

```
from django.http import Http404

def some_service():
    raise Http404()
```

Response:

```
{
    "detail": "Not found."
}
```

Code:

```
def some_service():
    raise RestValidationError("Some error message")
```

Response:

```
{
    "detail": [
        "Some error message"
    ]
}
```

Code:

```
def some_service():
    raise RestValidationError(detail={"error": "Some error message"})
```

Response:

```
{
    "detail": {
        "error": "Some error message"
    }
}
```

Code:

```
class NestedSerializer(serializers.Serializer):
    bar = serializers.CharField()

class PlainSerializer(serializers.Serializer):
    foo = serializers.CharField()
    email = serializers.EmailField(min_length=200)

nested = NestedSerializer()

def some_service():
    serializer = PlainSerializer(data={
        "email": "foo",
        "nested": {}
    })
    serializer.is_valid(raise_exception=True)
```

Response:

```
{
    "detail": {
        "foo": [
            "This field is required."
        ],
        "email": [
            "Ensure this field has at least 200 characters.",
            "Enter a valid email address."
        ],
        "nested": {
            "bar": [
                "This field is required."
            ]
        }
    }
}
```

Code:

```
from rest_framework import exceptions

def some_service():
    raise exceptions.Throttled()
```

Response:

```
{
    "detail": "Request was throttled."
}
```

Code:

```
def some_service():
    user = BaseUser()
    user.full_clean()
```

Response:

```
{
    "detail": {
        "password": [
            "This field cannot be blank."
        ],
        "email": [
            "This field cannot be blank."
        ]
    }
}
```

Подход 2 - предложенный компанией HackSoft способ

Мы собираемся предложить подход, который может быть легко расширен до чего-то, что хорошо работает для вас.

Вот основные идеи:

1. Ваше приложение будет иметь свою собственную иерархию исключений, которые будут выбрасываться бизнес-логикой.
2. Допустим, для простоты, что у нас будет только одна ошибка - **ApplicationError**.
3. Она будет определена в специальном приложении ядра, в модуле **exceptions**. По сути, у нас будет **project.core.exceptions.ApplicationError**.
4. Мы хотим позволить **DRF** обрабатывать все остальные ошибки по умолчанию.
5. **ValidationError** теперь специальный, и он будет обрабатываться по-другому.
6. **ValidationError** должна возникать только при сериализации или валидации модели.

Мы определим следующую структуру для наших ошибок:

```
{
    "message": "The error message here",
    "extra": {}
}
```

Дополнительный ключ может содержать произвольные данные для передачи информации фронтенду.

Например, когда у нас есть ошибка **ValidationError** (обычно исходящая от сериализатора или модели), мы собираемся представить ошибку следующим образом:

```
{  
    "message": "Validation error.",  
    "extra": {  
        "fields": {  
            "password": [  
                "This field cannot be blank."  
            ],  
            "email": [  
                "This field cannot be blank."  
            ]  
        }  
    },  
}
```

Это может быть передано во фронтенд, чтобы они могли искать **extra.fields**, чтобы представить эти специфические ошибки пользователю.

Чтобы достичь этого, пользовательский обработчик исключений будет выглядеть следующим образом:

```
from django.core.exceptions import ValidationError as DjangoValidationError, PermissionDenied
from django.http import Http404

from rest_framework.views import exception_handler
from rest_framework import exceptions
from rest_framework.serializers import as_serializer_error
from rest_framework.response import Response

from styleguide_example.core.exceptions import ApplicationError


def hacksoft_proposed_exception_handler(exc, ctx):
    """
    {
        "message": "Error message",
        "extra": {}
    }
    """

    if isinstance(exc, DjangoValidationError):
        exc = exceptions.ValidationError(as_serializer_error(exc))

    if isinstance(exc, Http404):
        exc = exceptions.NotFound()

    if isinstance(exc, PermissionDenied):
        exc = exceptions.PermissionDenied()

    response = exception_handler(exc, ctx)
    # If unexpected error occurs (server error, etc.)
    if response is None:
        if isinstance(exc, ApplicationError):
            data = {
                "message": exc.message,
                "extra": exc.extra
            }
            return Response(data, status=400)

    return response

    if isinstance(exc.detail, (list, dict)):
        response.data = {
            "detail": response.data
        }

    if isinstance(exc, exceptions.ValidationError):
        response.data["message"] = "Validation error"
        response.data["extra"] = {
            "fields": response.data["detail"]
        }
    else:
        response.data["message"] = response.data["detail"]
        response.data["extra"] = {}
        del response.data["detail"]

    return response
```

Посмотрите на этот код и попытайтесь понять, что происходит. Стратегия такова - повторно использовать как можно больше из DRF и затем корректировать.

Теперь мы получим следующее поведение:

Code:

```
from styleguide_example.core.exceptions import ApplicationError

def trigger_application_error():
    raise ApplicationError(message="Something is not correct", extra={"type": "RANDOM"})
```

Response:

```
{ 
    "message": "Something is not correct",
    "extra": {
        "type": "RANDOM"
    }
}
```

Code:

```
def some_service():
    raise DjangoValidationError("Some error message")
```

Response:

```
{ 
    "message": "Validation error",
    "extra": {
        "fields": {
            "non_field_errors": [
                "Some error message"
            ]
        }
    }
}
```

Code:

```
from django.core.exceptions import PermissionDenied
```

```
def some_service():
    raise PermissionDenied()
```

Response:

```
{ 
    "message": "You do not have permission to perform this action.",
    "extra": {}
}
```

Code:

```
from django.http import Http404

def some_service():
    raise Http404()
```

Response:

```
{  
    "message": "Not found.",  
    "extra": {}  
}
```

Code:

```
def some_service():
    raise RestValidationError("Some error message")
```

Response:

```
{  
    "message": "Validation error",  
    "extra": {  
        "fields": [  
            "Some error message"  
        ]  
    }  
}
```

Code:

```
def some_service():
    raise RestValidationError(detail={"error": "Some error message"})
```

Response:

```
{  
    "message": "Validation error",  
    "extra": {  
        "fields": {  
            "error": "Some error message"  
        }  
    }  
}
```

Code:

```
class NestedSerializer(serializers.Serializer):
    bar = serializers.CharField()

class PlainSerializer(serializers.Serializer):
    foo = serializers.CharField()
    email = serializers.EmailField(min_length=200)

nested = NestedSerializer()

def some_service():
    serializer = PlainSerializer(data={
        "email": "foo",
        "nested": {}
    })
    serializer.is_valid(raise_exception=True)
```

Response:

```
{
    "message": "Validation error",
    "extra": {
        "fields": {
            "foo": [
                "This field is required."
            ],
            "email": [
                "Ensure this field has at least 200 characters.",
                "Enter a valid email address."
            ],
            "nested": {
                "bar": [
                    "This field is required."
                ]
            }
        }
    }
}
```

Code:

```
from rest_framework import exceptions

def some_service():
    raise exceptions.Throttled()
```

Response:

```
{  
    "message": "Request was throttled.",  
    "extra": {}  
}
```

Code:

```
def some_service():
    user = BaseUser()
    user.full_clean()
```

Response:

```
{  
    "message": "Validation error",  
    "extra": {  
        "fields": {  
            "password": [  
                "This field cannot be blank."  
            ],  
            "email": [  
                "This field cannot be blank."  
            ]  
        }  
    }  
}
```

Теперь это можно расширить и сделать более подходящим для ваших нужд:

Вы можете иметь **ApplicationValidationError** и **ApplicationPermissionError**, как дополнительную иерархию.

Вы можете реализовать обработчик исключений по умолчанию **DRF**, вместо того, чтобы использовать его повторно (скопируйте-вставьте его и адаптируйте под свои нужды).

Общая идея такова: выясните, какой тип обработки ошибок вам нужен, а затем реализуйте его соответствующим образом.

Больше идей

Как видите, мы можем приспособить обработку исключений к нашим потребностям.

Вы можете начать обрабатывать больше вещей - например, перевести `django.core.exceptions.ObjectDoesNotExist` в `rest_framework.exceptions.NotFound`.

Вы даже можете обрабатывать все исключения, но тогда вы должны быть уверены, что эти исключения регистрируются должным образом, иначе вы можете упустить что-то важное.

Тестирование

В наших **Django** проектах мы разделяем наши тесты в зависимости от типа кода, который они представляют.

То есть, обычно у нас есть тесты для моделей, сервисов, селекторов и **API / представлений**.

Структура файлов обычно выглядит следующим образом:

```
project_name
├── app_name
│   ├── __init__.py
│   └── tests
│       ├── __init__.py
│       ├── models
│       │   └── __init__.py
│       │   └── test_some_model_name.py
│       ├── selectors
│       │   └── __init__.py
│       │   └── test_some_selector_name.py
│       └── services
│           └── __init__.py
│           └── test_some_service_name.py
└── __init__.py
```

Соглашения об именовании

Мы следуем двум общим соглашениям об именовании:

Имена файлов тестов должны быть

`test_the_name_of_the_thing_that_is_tested.py`

Тестовый пример должен быть класса

`TheNameOfTheThingThatIsTestedTests(TestCase):`

Например, если у нас есть:

```
def a_very_neat_service(*args, **kwargs):
    pass
```

В качестве имени файла мы возьмем следующее:

project_name/app_name/tests/services/test_a_very_neat_service.py

И следующее для тестового примера:

```
class AVeryNeatServiceTests(TestCase):
    pass
```

Для тестов служебных функций мы следуем аналогичной схеме.

Например, если у нас есть **project_name/common/utils.py**, то мы заведем **project_name/common/tests/test_utils.py** и поместим в этот файл различные тестовые случаи.

Если мы разделим модуль **utils.py** на подмодули, то то же самое произойдет и с тестами:

```
project_name/common/utils/files.py
project_name/common/tests/utils/test_files.py
```

Мы стараемся согласовать структуру наших модулей со структурой соответствующих тестов.

Celery

Мы используем **Celery** в следующих общих случаях:

- Общение со сторонними сервисами (отправка электронной почты, уведомлений и т.д.).
- Выгрузка более тяжелых вычислительных задач за пределы **HTTP**-цикла.
- Периодические задачи (с использованием **Celery beat**)

Основы

Мы стараемся относиться к **Celery** так, как будто это просто еще один интерфейс к нашей основной логике - то есть - не размещайте там бизнес-логику.

Давайте рассмотрим пример сервиса, который отправляет электронные письма (пример взят из **Django-Styleguide-Example**).

```
from django.db import transaction
from django.core.mail import EmailMultiAlternatives

from styleguide_example.core.exceptions import ApplicationError
from styleguide_example.common.services import model_update
from styleguide_example.emails.models import Email


@transaction.atomic
def email_send(email: Email) -> Email:
    if email.status != Email.Status.SENDING:
        raise ApplicationError(f"Cannot send non-ready emails. Current status is {email.status}")

    subject = email.subject
    from_email = "styleguide-example@hacksoft.io"
    to = email.to

    html = email.html
    plain_text = email.plain_text

    msg = EmailMultiAlternatives(subject, plain_text, from_email, [to])
    msg.attach_alternative(html, "text/html")

    msg.send()

    email, _ = model_update(
        instance=email,
        fields=["status", "sent_at"],
        data={
            "status": Email.Status.SENT,
            "sent_at": timezone.now()
        }
    )
    return email
```

Отправка электронной почты имеет вокруг себя бизнес-логику, но мы все равно хотим запускать эту конкретную службу из задачи.

Наша задача выглядит следующим образом:

```
from celery import shared_task
```

```
from styleguide_example.emails.models import Email
```

```
@shared_task
def email_send(email_id):
    email = Email.objects.get(id=email_id)

    from styleguide_example.emails.services import email_send
    email_send(email)
```

Как видите, мы рассматриваем задачу как **API**:

- Получить необходимые данные.
- Вызвать соответствующую службу.
- Теперь представьте, что у нас есть другой сервис, который запускает отправку электронной почты.

Это может выглядеть следующим образом:

```
from django.db import transaction
```

```
# ... more imports here ...
```

```
from styleguide_example.emails.tasks import email_send as email_send_task
```

```
@transaction.atomic
def user_complete_onboarding(user: User) -> User:
    # ... some code here

    email = email_get_onboarding_template(user=user)

    transaction.on_commit(lambda: email_send_task.delay(email.id))

return user
```

Здесь необходимо обратить внимание на два важных момента:

1. Мы импортируем **task** (который имеет то же имя, что и сервис), но даем ему суффикс **_task**.
2. И когда транзакция фиксируется, мы вызываем задачу.

Итак, в целом, способ использования **Celery** можно описать следующим образом:

1. **Task** вызывают сервисы.
2. Мы импортируем сервис в тело функции задачи.
3. Когда мы хотим вызвать задачу, мы импортируем задачу на уровне модуля, передавая суффикс **_task**.
4. Мы выполняем задачи в качестве побочного эффекта всякий раз, когда наша транзакция фиксируется.

Такой способ смешивания задач и сервисов также предотвращает циклический импорт, который может возникать достаточно часто при использовании **Celery**.

Обработка ошибок

Иногда наш сервис может дать сбой, и мы можем захотеть обработать ошибку на уровне задачи. Например, мы можем захотеть повторить выполнение задачи.

Этот код обработки ошибок должен находиться в задаче.

Давайте расширим пример задачи **email_send**, приведенный выше, добавив обработку ошибок:

```
from celery import shared_task
from celery.utils.log import get_task_logger

from styleguide_example.emails.models import Email

logger = get_task_logger(__name__)

def _email_send_failure(self, exc, task_id, args, kwargs, einfo):
    email_id = args[0]
    email = Email.objects.get(id=email_id)

    from styleguide_example.emails.services import email_failed

    email_failed(email)

@shared_task(bind=True, on_failure=_email_send_failure)
def email_send(self, email_id):
    email = Email.objects.get(id=email_id)

    from styleguide_example.emails.services import email_send

    try:
        email_send(email)
    except Exception as exc:
        # https://docs.celeryq.dev/en/stable/userguide/tasks.html#retrying
        logger.warning(f"Exception occurred while sending email: {exc}")
        self.retry(exc=exc, countdown=5)
```

Как вы видите, мы выполняем множество повторных попыток, и если все они оказываются неудачными, мы обрабатываем это в обратном вызове **on_failure**.

Обратный вызов следует шаблону именования **_имя_задачи_failure** и вызывает сервисный уровень, как и обычная задача.

Конфигурация

Мы практически полностью следуем официальному руководству по интеграции Celery с Django - <https://docs.celeryq.dev/en/stable/django/first-steps-with-django.html>.

Для полного примера вы можете посмотреть конфигурацию **Celery** в проекте **Django-Styleguide-Example**:

- https://github.com/HackSoftware/Django-Styleguide-Example/tree/master/styleguide_example/tasks
- https://github.com/HackSoftware/Django-Styleguide-Example/blob/master/styleguide_example/tasks/celery.py

Celery - сложная тема, поэтому стоит потратить время на чтение документации и понимание различных вариантов конфигурации.

Мы постоянно делаем это и находим новые вещи или лучшие подходы к решению наших проблем.

Структура

Задачи находятся в модулях **tasks.py** в различных приложениях.

Мы следуем тем же правилам, что и со всем остальным (**API**, сервисы, селекторы): если задачи для данного приложения становятся слишком большими, мы разделяем их по доменам.

То есть в итоге вы можете получить **tasks/domain_a.py** и **tasks/domain_b.py**. Все, что вам нужно сделать, это импортировать их в **tasks/__init__.py**, чтобы **Celery** автоматически обнаружил их.

Общее эмпирическое правило гласит: разделяйте задачи так, как вам удобно.

Периодические задачи

Управление периодическими задачами очень важно, особенно когда у вас их десятки или сотни.

Мы используем **Celery Beat** + **django_celery_beat.schedulers:DatabaseScheduler** + **django-celery-beat** для наших периодических задач.

Дополнительная вещь, которую мы делаем - это команда управления, называемая **setup_periodic_tasks**, которая содержит определение всех периодических задач в системе. Эта команда находится в приложении **tasks**, о котором говорилось выше.

Вот как выглядит **project.tasks.management.commands.setup_periodic_tasks.py**:

```

from django.core.management.base import BaseCommand
from django.db import transaction

from django_celery_beat.models import IntervalSchedule, CrontabSchedule, PeriodicTask

from project.app.tasks import some_periodic_task


class Command(BaseCommand):
    help = f"""
    Setup celery beat periodic tasks.

    Following tasks will be created:

    - {some_periodic_task.name}
    """

    @transaction.atomic
    def handle(self, *args, **kwargs):
        print('Deleting all periodic tasks and schedules...\n')

        IntervalSchedule.objects.all().delete()
        CrontabSchedule.objects.all().delete()
        PeriodicTask.objects.all().delete()

        periodic_tasks_data = [
            {
                'task': some_periodic_task,
                'name': 'Do some peridoic stuff',
                # https://crontab.guru/#15_*_*_*
                'cron': {
                    'minute': '15',
                    'hour': '*',
                    'day_of_week': '*',
                    'day_of_month': '*',
                    'month_of_year': '*',
                },
                'enabled': True
            },
        ]

        for periodic_task in periodic_tasks_data:
            print(f'Setting up {periodic_task["task"].name}')

            cron = CrontabSchedule.objects.create(
                **periodic_task['cron']
            )

            PeriodicTask.objects.create(
                name=periodic_task['name'],
                task=periodic_task['task'].name,
                crontab=cron,
                enabled=periodic_task['enabled']
            )

```

Несколько ключевых моментов:

- Мы используем эту задачу как часть процедуры развертывания.
- Мы всегда помещаем ссылку на **crontab.guru**, чтобы объяснить работу **cron**. Иначе он будет нечитабельным.
- Все находится в одном месте.
- предупреждение : Мы используем, почти исключительно, расписание **cron**. Если вы планируете использовать другие объекты расписания, предоставляемые **Celery**, пожалуйста, прочитайте их документацию и важные заметки - <https://django-celery-beat.readthedocs.io/en/latest/#example-creating-interval-based-periodic-task> - об указании на один и тот же объект расписания. **warning**

За пределами

Celery обладает мощными инструментами для реализации сложных рабочих процессов -

<https://docs.celeryq.dev/en/stable/userguide/canvas.html>.

Если вы решите использовать их, правила все равно остаются в силе.

Возможно, вам придется немного реорганизовать все, но пока у вас есть четко определенный интерфейс к ядру приложения, вы сможете смешивать и сопоставлять задачи и сервисы в более сложных сценариях.

Более сложные сценарии зависят от контекста. Убедитесь, что вы осознаете архитектуру и принимаемые вами решения.

Кулинарная книга

Здесь хранятся некоторые реализации общих многократно используемых частей кода.

Обработка обновлений с помощью сервиса

Что касается обновления, у нас есть общий сервис обновления, который мы используем внутри реальных сервисов обновления. Вот как выглядит пример сервиса **user_update**:

```
def user_update(*, user: User, data) -> User:  
    non_side_effect_fields = ['first_name', 'last_name']  
  
    user, has_updated = model_update(  
        instance=user,  
        fields=non_side_effect_fields,  
        data=data  
    )  
    # Side-effect fields update here (e.g. username is generated based on first & last name)  
    # ... some additional tasks with the user ...  
    return user
```

- Мы вызываем общий сервис **model_update** для полей, которые не имеют побочных эффектов, связанных с ними (то есть они просто устанавливаются в значение, которое мы предоставляем).
- Этот паттерн позволяет нам извлечь повторяющиеся настройки полей в общий сервис и выполнять только специфические задачи внутри сервиса обновления (побочные эффекты).

Общая реализация **model_update** выглядит следующим образом:

```
def model_update(
    *,
    instance: DjangoModelType,
    fields: List[str],
    data: Dict[str, Any]
) -> Tuple[DjangoModelType, bool]:
    has_updated = False

    for field in fields:
        if field not in data:
            continue

        if getattr(instance, field) != data[field]:
            has_updated = True
            setattr(instance, field, data[field])

    if has_updated:
        instance.full_clean()
        instance.save(update_fields=fields)

    return instance, has_updated
```

Полную реализацию этих сервисов можно найти в нашем примере проекта:

- **model_update**
- **user_update**

DX (Developer Experience)

Раздел с различными вещами, которые могут сделать вашу жизнь разработчика **Django** лучше.

типу / аннотации типов

Когда речь заходит об использовании аннотаций типов, наряду с **типу**, этот твит во многом перекликается с нашей философией.

- У нас есть проекты, где мы применяем **типу** и очень строго относимся к типам.
- У нас есть проекты, где типы более свободны и **типу** не используется вообще.

Контекст здесь - король.

В примере **Django-Styleguide-Example** мы настроили **трупу**, используя как <https://github.com/typeddjango/django-stubs>, так и <https://github.com/typeddjango/djangorestframework-stubs/>. Вы можете проверить это в качестве примера.

Кроме того, этот конкретный проект - <https://github.com/wemake-services/wemake-django-template> - также имеет конфигурацию **трупы**.

Определите, что будет работать лучше для вас.

Django Styleguide в дикой природе

Здесь собрана коллекция различных людей и компаний, которые нашли руководство по стилю полезным:

Майкл Валенсия, технический директор **Facturedo**

Исходный код нашего основного проекта в **Facturedo** начал становиться запутанным. Бизнес-логику можно было найти во многих, бессвязных местах. Нам нужно было решение для структурирования нашего **Django** проекта, и мы нашли его в **Django Styleguide**.

Мы рекомендуем его всем, кто хочет структурировать проект среднего или большого размера. Это хорошо проработанное руководство, которое постоянно развивается.

Дополнительные ресурсы

Дополнительные ресурсы, которые мы сочли полезными и которые могут добавить ценность к руководству по стилю.

- [Дэн Палмер - Масштабирование Django до 500 приложений \(DjangoCon US 2021\)](#)

Вдохновение

То, как мы делаем **Django**, вдохновлено следующими вещами:

- Общая идея разделения забот
- **Boundaries** by Gary Bernhardt
- Объекты сервисов **Rails**