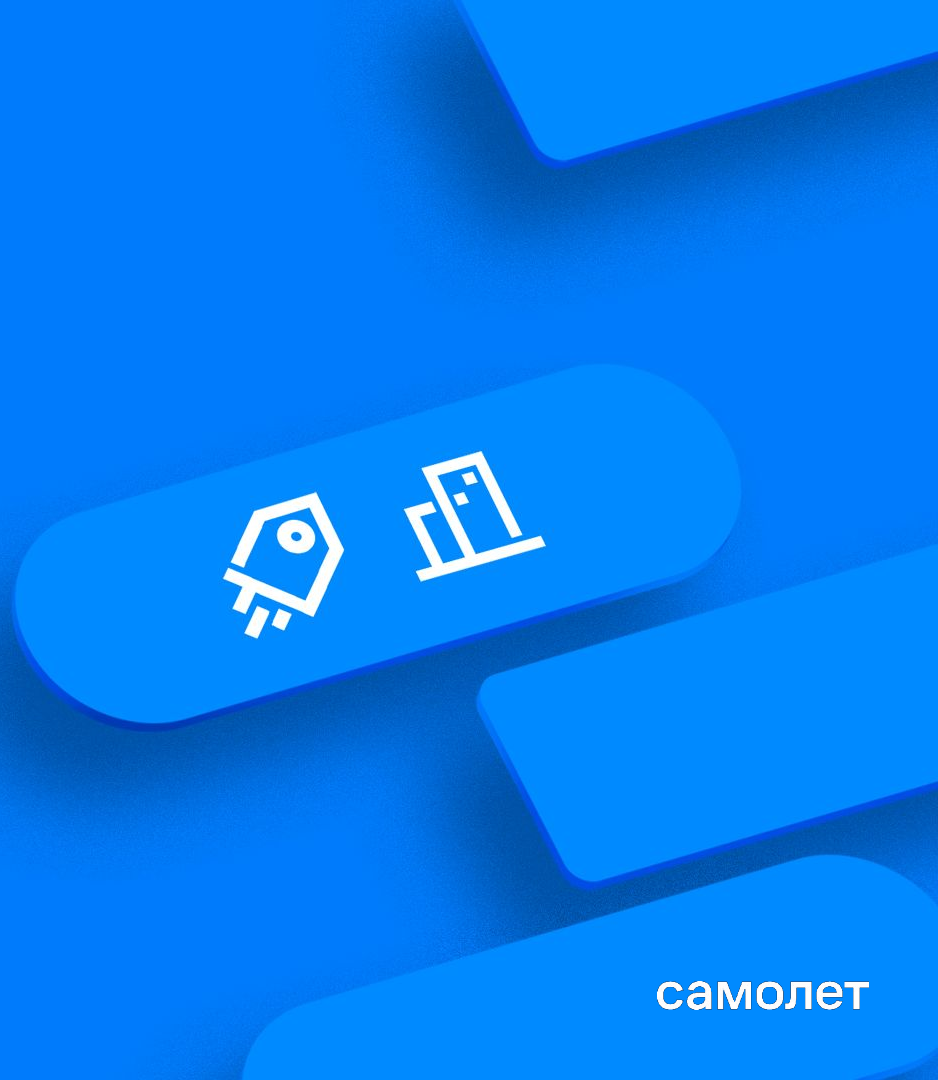
Fake Injection



самолет

Занимаюсь разработкой с 2017

Начинал как frontend/mobile разработчик в game-dev

С 2019 пишу backend используя преимущественно Django

Работал как в продуктовых командах, так и в заказной разработке

Крупный девелопер и PropTech компания

Мы нацелены на автоматизацию и цифровизацию процессов на стройке

50+

внутренних проектов с использованием Django

Много проектов со сложными бизнес процессами

Один из проектов компании

Мы помогаем подрядчикам получать актуальную рабочую документацию

Относительно небольшой проект и молодой проект

Успели столкнуться с классическими проблемами

Расскажем про проблемы которые часто встречаются в Django приложениях

1

Поговорим про подход который мы выработали для написания бизнес логики

2

Как этот подход помогает тестировать код

3

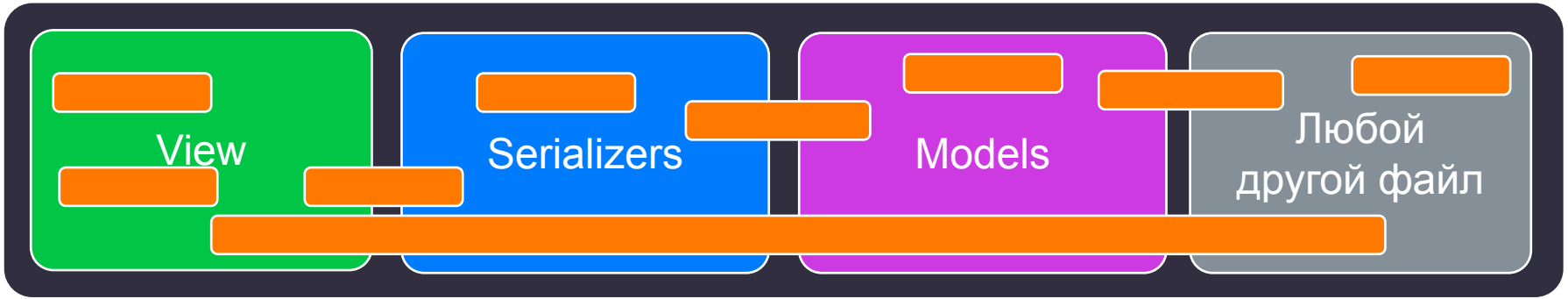
Что получилось в итоге

4

Если вы не любите Django, то все равно будет интересно

5

самолет



Можно сказать что есть проблема размытия, если логика для отдельного бизнес процесса разделена между несколькими слоями/файлами

Частый случай

view вызывает serializer, в методе которого вызывается метод из Model, где в свою очередь вызов проксируется в services.py

Толстая модель, это модель которая содержит бизнес-логику в методах

Иногда это просто вспомогательные свойства/методы которые добавляют выразительности коду, а иногда ловушка с сайдэффектом

Не страшный пример, просто свойство для сокращения рутинных действий

```python
class Profile(models.Model):
    first_name = models.CharField(...)
    last_name = models.CharField(...)

    def get_full_name(self) -> str:
        return f"{self.first_name} {self.last_name}"
```

Потенциально опасный пример который легко приведет к N + 1

```python
class Profile(models.Model):
    user = models.OneToOne(User, …)

    def get_full_name(self) -> str:
        return (
          f"{self.user.first_name} {self.user.last_name}"
        )
```

Толстые сериалайзеры аналогичны толстым моделям, это сериалайзеры которые содержат внутри:

- сложную логику проверок
- дополнительные запросы, вызовы сервисов внутри SerializerMethodField

Как итог, сильно увеличивается связность в коде

Кандидат на рефакторинг

```python
class CommentRegistryDetailSerializer(serializers.Serializer):

    @extend_schema_field(ProjectShortSerializer)
    def get_project(self, obj: Comment):
        project = obj.version.documentation_kit.project
        return ProjectShortSerializer(project).data
```

Почти тоже самое, но тут выборка гораздо сложнее

```python
class CommentRegistryListSerializer(serializers.ModelSerializer):


 @extend_schema_field(ResponsibleProjectTeamShortSerializer(many=True))
    def get_responsible_engineers(self, obj: Comment):
        responsible_queryset = get_responsible_for_agreement_engineers(
            obj, include_responsible=True,
        )
        responsible_queryset = responsible_queryset.annotate(
            full_name=Concat("user__first_name",
                             Value(" "), "user__last_name"),
        )

        return ResponsibleProjectTeamShortSerializer(
            responsible_queryset, many=True
        ).data
```

Проблема: utils.py / helpers.py без четкой зоны ответственности

Сюда попадает все что не подходит для привычных views.py / models.py / serializers.py

1

Часто нет структуры, новые элементы добавляются хаотично

2

Часто нарушается принцип единственной ответственности

3

Описанные проблемы применимы не только к Django

1

Сам по себе монолит не проблема, если отвечает за одну доменную область

2

Легко придумать свои пункты за и против

3

Если чистые функции, это функции которые не имеют побочных эффектов, то одним из свойств чистого unit теста, будет независимость от любых третьих сервисов

Из коробки Django предлагает широкий набор инструментов для тестирования вместе с базой, такой подход скорее относится к интеграционному тестированию

```
with (
        mock.patch(S3_SERVICE_PATH) as mocked_common_s3_module,
        mock.patch(UPDATE_SERVICE_PATH, new=mocked_service),
    ):
```

Любые изменения в путях требуют ручного поиска путей в моках

Любые новые запросы к сервисам требуют анализа существующих тестов и новых моков

Следствие первого пункта — нельзя после рефакторинга просто запустить тесты быть уверенным что все ок

Проблемы которые стали ключевыми:

Периодические сбои тестов из за недостаточной изоляции отдельных сценарием, большая сложность в поиске конкретного места утечки

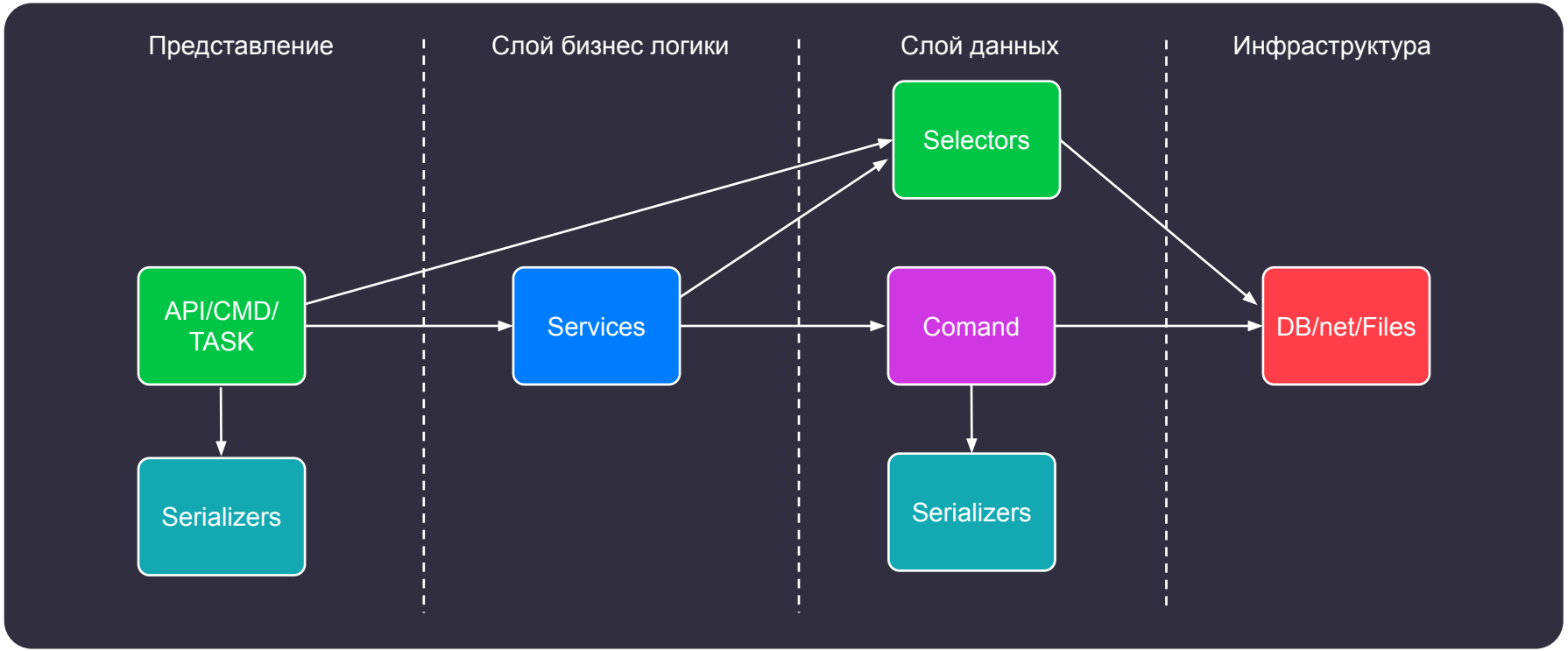
1

Неприлично высокое время запуска тестов которое многократно увеличивалось из периодических сбоев в flaky тестах

2

Flaky тесты не применимы в CI/CD

3


Представление
Слой бизнес логики
Слой данных
Инфраструктура
Selectors
API/CMD/
TASK
Services
Comand
DB/net/Files
Serializers
Serializers

Разделение на слои привносит естественное разделение в коде

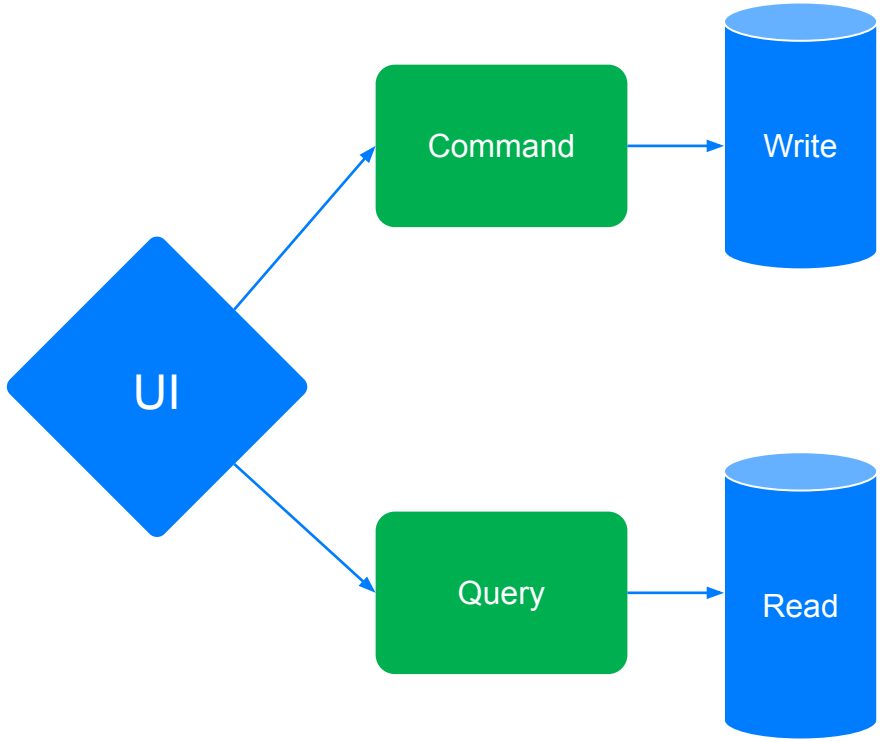
Views.py — минимум логики, теперь роль view, это оркестрация работы со специальными сервисами и работа с http

Сервисы отвечают исключительно за бизнес логику, делегируя получение данных и обновление селекторам и командам

Команды и селекторы передаются сервису как зависимости

Разделяем логику обновления и логику получения данных

Используем сервисные классы для оркестрирования команд и запросов в рамках одного бизнес процесса



Используем DI для объединения команд и запросов и других сервисов

1

Отказались от сторонних библиотек для упрощения подхода

2

Используем dataclass для удобной инициализации сервиса

3

Используем метод __call__ как единую точку вызова сервиса

4

```
some_service_name      # python пакет, который может быть импортирован
    __init__.py
    commands.py        # обновление данных
    selectors.py       # запросы на чтение
    services.py        # описание сервисного класса
    utils.py
```

```
api.awesome_feature.retrieve # python пакет
    __init__.py
    selectors.py          # запросы на чтение
    services.py           # описание сервисного класса
    views.py              # view аналогично обычному views.py
    serializers.py
    utils.py
```

```python
@final
@dataclass(frozen=True, slots=True, kw_only=True)
class SomeAwesomeService:
    _load_some_items_from_database: Callable[[PK], Iterable[Item]]

    def __call__(self, item_id: PK) -> dict:
        item = self._load_some_items_from_database(item_id)

        return {
            “id”: item.id,
        }

awesome_service: Final = SomeAwesomeService(
    _load_some_items_from_database=load_some_items_from_database,
)
```

Плюсы такого подхода

Сервис класс можно протестировать отдельно, в каждом тесте создавая новый экземпляр класса с тестовыми зависимостями +

Легко писать unit-тесты, так как зависимости, которые взаимодействуют с базой можно подменить на имитирующую функцию +

Такой подход не требует следить за путями при подмене, так как подмены не происходит, а происходит создание экземпляра класса с другим набором зависимостей +

Унифицируется подход к написанию бизнес-логики +

Можно строить сложные системы используя композицию и используя одни сервисы как зависимости для других +

```python
def force_approve_version(version: DocumentationVersion):
    logger.info(f"force version approve by chief engineer, version id: {version.id}")

    version.approved = True
    version.approved_at = timezone.now()
    version.save()

    version.documents.update(sproject_task_approved=True)

    # Создаем трекеры принятия РД для подрядчиков
    project_contractors =
ProjectContactor.objects.get_responsible_contractors_ids_for_kit(version.kit_catalog_id)
    StatusTracker.objects.create_trackers_for_project_contractors(version, project_contractors)

    # отправляем email уведомления
    # Собираем id пользователей которым нужно отправить уведомление.
    send_version_approved_email_notification.delay(version.kit_catalog_id)
```

```python
@final
@dataclass(frozen=True, slots=True, kw_only=True)
class ApproveVersionService:
    """Содержит логику по выпуску новой версии в производство работ в ручном режиме."""

    _get_latest_version_for_kit: Callable[[DocumentationKit], DocumentationVersion | None]
    _perform_save: Callable[[DocumentationVersion, list[Documentation]], None]
    _create_trackers_for_version: Callable[[DocumentationVersion], None]

    def __call__(self, documentation_kit: DocumentationKit, approved_by: User) -> None:
        version = self._get_latest_version_for_kit(documentation_kit)

        version.approved_by = approved_by
        version.approved_at = timezone.now()
        version.approved = True
        self._perform_save(version)

        # Создаем трекеры принятия РД для подрядчиков
        self._create_trackers_for_version(version)

        logger.info("Approve process completed.")
```

```python
@pytest.mark.parametrize("version", [None, DocumentationVersion(approved=False, version=4)])
def test_upload_architectural_supervision_file_without_any_versions(version):
    """Проверяем что в случае если комплект пустой или содержит новую версию,
    то к нему нельзя загрузить лист авторского надзора."""
    documentation_kit = KitCatalogFactory.build()

    attach_architectural_supervision_file = AttachArchitecturalSupervisionFileService(
        _get_latest_version_for_kit=lambda *args, **kwargs: version,
        _get_version_files_map=Mock(),
        _get_file_storage_for_kit=Mock(),
        _save_to_database=Mock(),
    )

    with pytest.raises(ServiceException):
        attach_architectural_supervision_file(
            documentation_kit=documentation_kit,
            uploaded_file=TemporaryDocumentationFile(documentation_kit=documentation_kit,
file_name="docs.pdf"),
            uploaded_by=UserFactory.build(),
        )

    # Проверяем что все команды после проверки версии не были вызваны
    attach_architectural_supervision_file._get_file_storage_for_kit.assert_not_called()
    attach_architectural_supervision_file._save_to_database.assert_not_called()
    attach_architectural_supervision_file._get_version_files_map.assert_not_called()
```

```python
@pytest.mark.parametrize("version", [None, DocumentationVersion(approved=False, version=4)])
def test_upload_architectural_supervision_file_without_any_versions(version):
    documentation_kit = KitCatalogFactory.build()
```

Тестовые данные создаем без вызова сохранения данных в базу

Запрещаем работу с базой на уровне pytest

```
attach_architectural_supervision_file = AttachArchitecturalSupervisionFileService(
    _get_latest_version_for_kit=lambda *args, **kwargs: version,
    _get_version_files_map=Mock(),
    _get_file_storage_for_kit=Mock(),
    _save_to_database=Mock(),
)
```

```
with pytest.raises(ServiceException):
        attach_architectural_supervision_file(
            documentation_kit=documentation_kit,
            uploaded_file=TemporaryDocumentationFile(
                documentation_kit=documentation_kit, file_name="docs.pdf",
            ),
            uploaded_by=UserFactory.build(),
        )

# Проверяем что все команды после проверки версии не были вызваны
attach_architectural_supervision_file._get_file_storage_for_kit.assert_not_called()
attach_architectural_supervision_file._save_to_database.assert_not_called()
attach_architectural_supervision_file._get_version_files_map.assert_not_called()
```

```python
def test_contractor_accept_accepted_change_request(fake_data: dict):
    change_request = fake_data["change_request"]
    change_request.status = ChangeRequestStatusChoices.IN_PROGRESS.value

    accept_change_request_service = AcceptChangeRequestService(
        _get_access_controller=get_access_controller,
        _start_transaction=fake_transaction_atomic_context_manager,
        _get_change_request=MagicMock(return_value=change_request),
        _save_object=MagicMock(),
        _send_change_request_status_changed_notification=MagicMock(),
    )

    with pytest.raises(ServiceException):
        accept_change_request_service(
            change_request_id=22,
            responsible=fake_data["contractor_user"],
            planned_date=date(year=2035, month=4, day=1),
        )

    accept_change_request_service._save_object.assert_not_called()
    accept_change_request_service._send_change_request_status_changed_notification.assert_not_called()
```

Тестовые зависимости могут быть сложнее


```
accept_change_request_service = AcceptChangeRequestService(
    _get_access_controller=get_access_controller,
    _start_transaction=fake_transaction_atomic_context_manager,
    _get_change_request=MagicMock(return_value=change_request),
    _save_object=MagicMock(),
    _send_change_request_status_changed_notification=MagicMock(),
)
```

```
@contextmanager
def fake_transaction_atomic_context_manager(*args, **kwargs):
    yield
```

85%

total code coverage

85%

integration tests coverage

Unit tests: единичные тесты, почти не влияющие на покрытие

85%

total code coverage

82%

integration tests coverage

44%

unit tests coverage

39 сек

общее время выполнения тестов

37 сек

время выполнения только интеграционных тестов

4,91 сек

время выполнения юнит тестов, большая часть — инициализация

0,09 сек

среднее время выполнения unit теста

310

всего тестов в проекте

176

интеграционных тестов

134

юнит тестов, почти все были добавлены

Integration тесты все еще значимая часть нашего подхода к тестированию

Добавление unit тестов не замедлило общее время запуска тестов

Unit тесты основа тестирования сервисов, с большим количеством бизнес правил, там где основная сложность не в запросах

Integration тесты дополняют unit тесты для сервисов, но часто только основные сценарии — happy path

Стало проще писать тесты для бизнес логики

У предложенного подхода ряд минусов, которые стоит принять во внимание

Большая когнитивная сложность для простых случаев

Как следствие, такой подход не является заменой для generic view/viewset в случае простых CRUD операций

Как следствие в некоторых местах будет оставаться разнородность в коде

Проверку входных данных с использованием RelatedField или get_object_or_404 можно назвать нарушением подхода

Спасибо за внимание

самолет