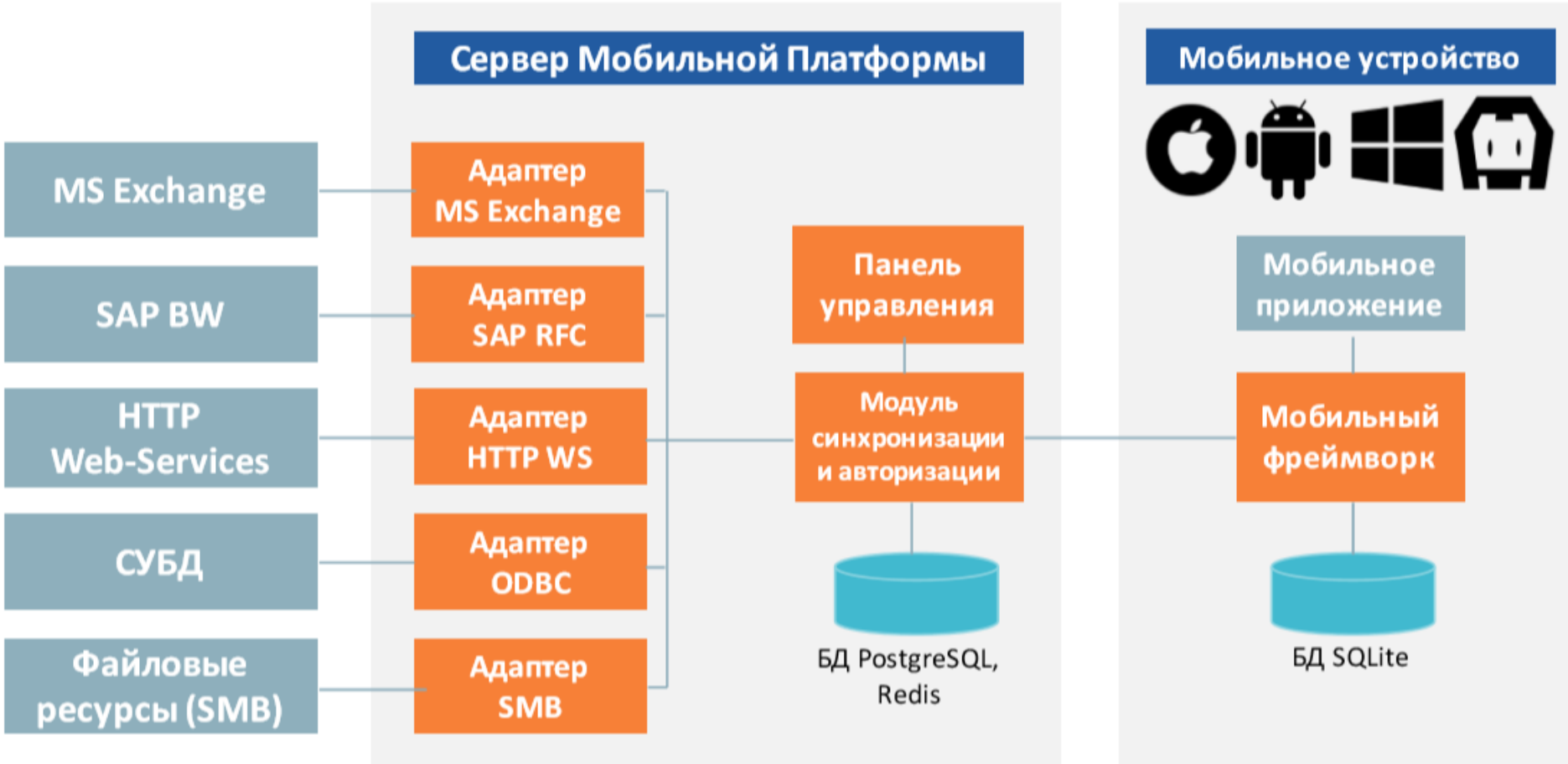
Динамическая структура базы данных с использованием Django ORM и без неё

Иван Ларин

Контекст


Сервер Мобильной Платформы
Мобильное устройство
MS Exchange
SAP BW
HTTP Web-Services
СУБД
Файловые ресурсы (SMB)
Адаптер MS Exchange
Адаптер SAP RFC
Адаптер HTTP WS
Адаптер ODBC
Адаптер SMB
Панель управления
Модуль синхронизации и авторизации
БД PostgreSQL, Redis
Мобильное приложение
Мобильный фреймворк
БД SQLite

Контекст

Проксирование и кэширование внешних таблиц для мобильного API

Контекст

Проксирование и кэширование внешних таблиц для мобильного API

Версионирование и отдача изменений (дельта) в API

Контекст

Проксирование и кэширование внешних таблиц для мобильного API

Версионирование и отдача изменений (дельта) в API

Схема источников данных неизвестна заранее

Контекст

Проксирование и кэширование внешних таблиц для мобильного API

Версионирование и отдача изменений (дельта) в API

Схема источников данных неизвестна заранее

Необходимо хранить структуру и типы данных внешних источников

Проблематика

Django ORM не поддерживает динамические модели "из коробки"

Типы данных у всех источников различаются, в том числе от базы данных используемой в качестве backend'a django

Производительность операций

Миграция схемы данных

Способы решения

Готовые батарейки
noSQL базы данных
EAV
Сырой SQL

Исходные данные

Data Access Object (DAO) как python класс с набором методов реализующих CRUD и преобразование данных от источника данных в собственные типы данных и SQL.

PostgreSQL в качестве backend’a django.

Необходимость использовать менеджер django, чтобы строить queryset’ы и применять инструменты django, например пагинацию и миграции.

Готовые решения

dynamic-models

 willhardy/dynamic-models

django-mutant

 charettes/django-mutant

django-dynamo

 schacki/django-dynamo

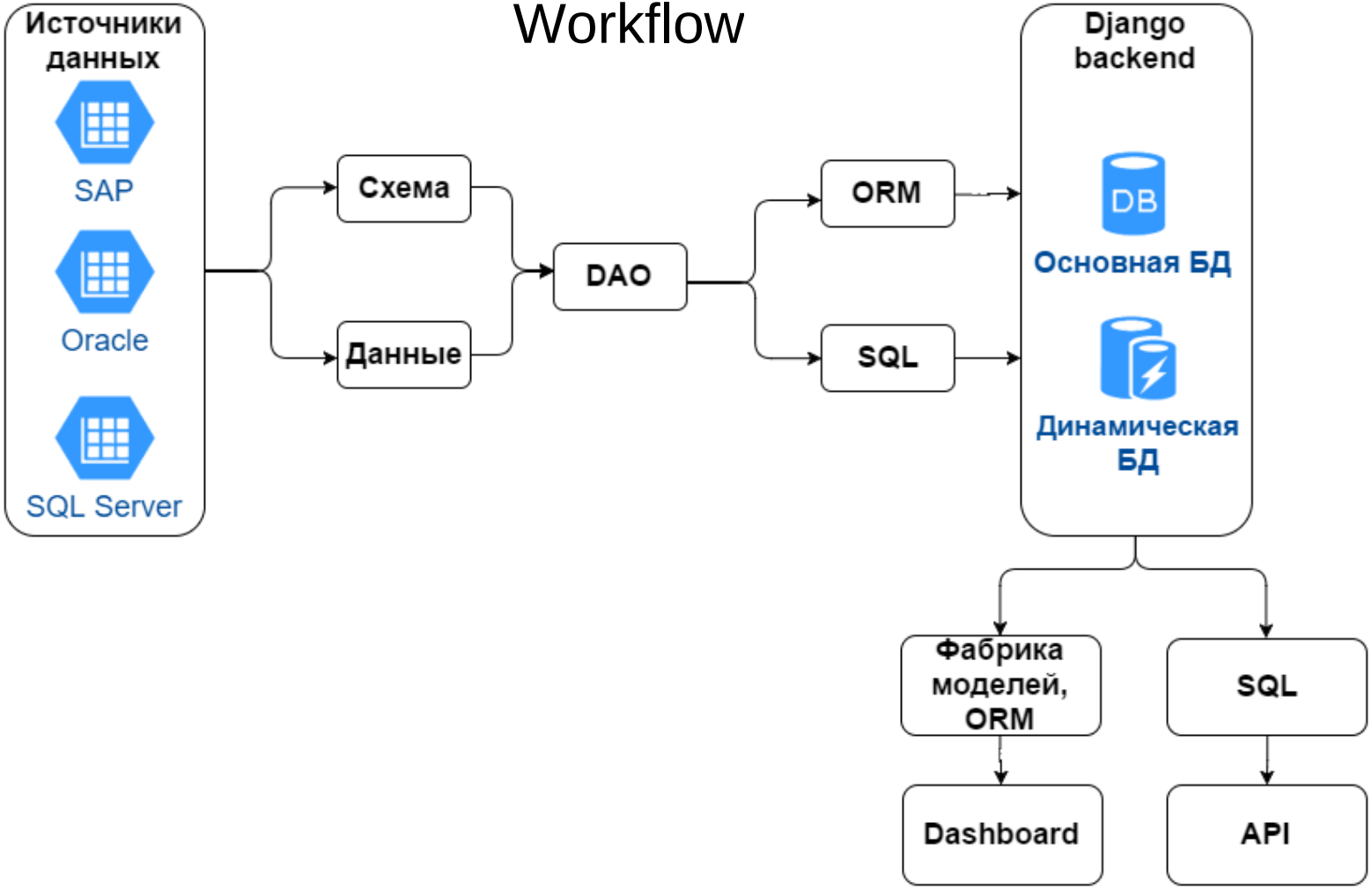
django-eav

 mvpdev/django-eav

eav-django

 neithere/eav-django

Workflow


Источники данных
SAP
Oracle
SQL Server
Схема
Данные
DAO
ORM
SQL
Django backend
DB
Основная БД
Динамическая БД
Фабрика моделей, ORM
SQL
Dashboard
API

Фабрика моделей

Объявление модели:

```
class DynamicManager(models.Manager):
    pass


class ModelName(models.Model):
    objects = DynamicManager()
    name = models.CharField(max_length=32)

    class Meta:
        db_table = 'table_name'
```

Фабрика моделей

Динамическое объявление модели:

```
meta = type('Meta', (object,), {'db_table': 'table_name'})
attrs = {
    'Meta': meta,
    'objects': DynamicManager(),
    'name': models.CharField(max_length=32),
    '__module__': 'module.models',
}
ModelName = type('ModelName', (models.Model,), attrs)
```

Фабрика моделей

По созданной таблице генерируется модель Django ORM.
`./manage.py inspectdb` — готовит текст модели по структуре таблиц в базе данных полученных через интроспекцию (самоанализ).

django.db.connections[DYNAMIC_DB].introspection
содержит методы для получения информации о структуре таблиц:

get_table_list
get_field_type
get_relations
get_key_columns

get_indexes
get_storage_engine
get_constraints

Фабрика моделей

Поля модели создаются в runtime, по ним генерируется SQL.

Django < 1.9

```
queries = connection.creation.sql_create_model(
    model, style, known_models)
queries = connection.creation.sql_destroy_model(
    model, all_models, references)


for sql_query in queries:
    cursor.execute(sql_query)
```

Фабрика моделей

Поля модели создаются в runtime, по ним генерируется SQL.

Django >= 1.9

```
with connection.schema_editor() as schema_editor:
    schema_editor.create_model(model)
    schema_editor.delete_model(model)
```

Менеджер и разделение пространства имён

Удобным решением оказалось хранение всех динамический моделей в отдельной БД.

```
DATABASES = {
    'default': {
        'NAME': 'db_name',
    },
    DYNAMIC_DB: {
        'NAME': 'dyn_db_name',
    }
}
```

```
class DynamicManager(models.Manager):
    def get_queryset(self):
        qs = super().get_queryset()
        return qs.using(settings.DYNAMIC_DB)
```

Наследование и миграции

Если модели зарегистрированы, в то ContentType и в django.apps.registry.App, то возможно применять стандартный механизм миграций.

```
django.apps.apps.all_models[app_name][model_name] = model
```

```
django.contrib.contenttypes.models.ContentType
```

Выводы

Все готовые решения работают одинаково и различаются лишь в своих ограничениях;

Миграция динамических моделей имеет свои особенности, но не сильно отличается от миграции обычных моделей;

Там где нужна производительность - без "сырого" SQL не обойтись;

Структуру динамических таблиц надо хранить в основой базе, а сами таблицы в отдельной;

Вопросы?

email: pentusha@gmail.com

github: https://github.com/Pentusha