KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект >2 урок > Application Service & Service Discovery

> Оглавление

Примеры из лекции: скачать

> Введение

Сегодня поговорим о самом приложении, которое будет сервить модель, а именно о веб-сервисе (или Application Service). Наверняка вы уже поднимали веб-сервис в предыдущих модулях или хотя бы запускали какой-нибудь простейший `hello world` сервис. То есть вы уже понимаете, что сервис в нашем случае — это обычный питоновский процесс, который слушает выбранный порт, куда приходят запросы от клиента. Также вы, наверное, уже понимаете, что если запрос был правильный и был прочитан нашим процессом, то процесс вызывает обработчики (или handler'ы, ручки, представления), обрабатывающие запрос и возвращающие ответ на тот же порт.

Обработчики, как вы помните, это и есть питоновские функции, которые мы пишем. Благодаря удобству современных фреймворков, вся разработка веб-сервиса фактически сводится к разработке этих ручек. Но есть ещё работа с ресурсами и конфигами, о которой нужно помнить. Также не стоит забывать и про graceful shutdown, т.е. плавное завершение, про корректный старт и проверки, которые позволяют нам понять, что приложение запустилось и готово принимать трафик, про хелфчеки и др.

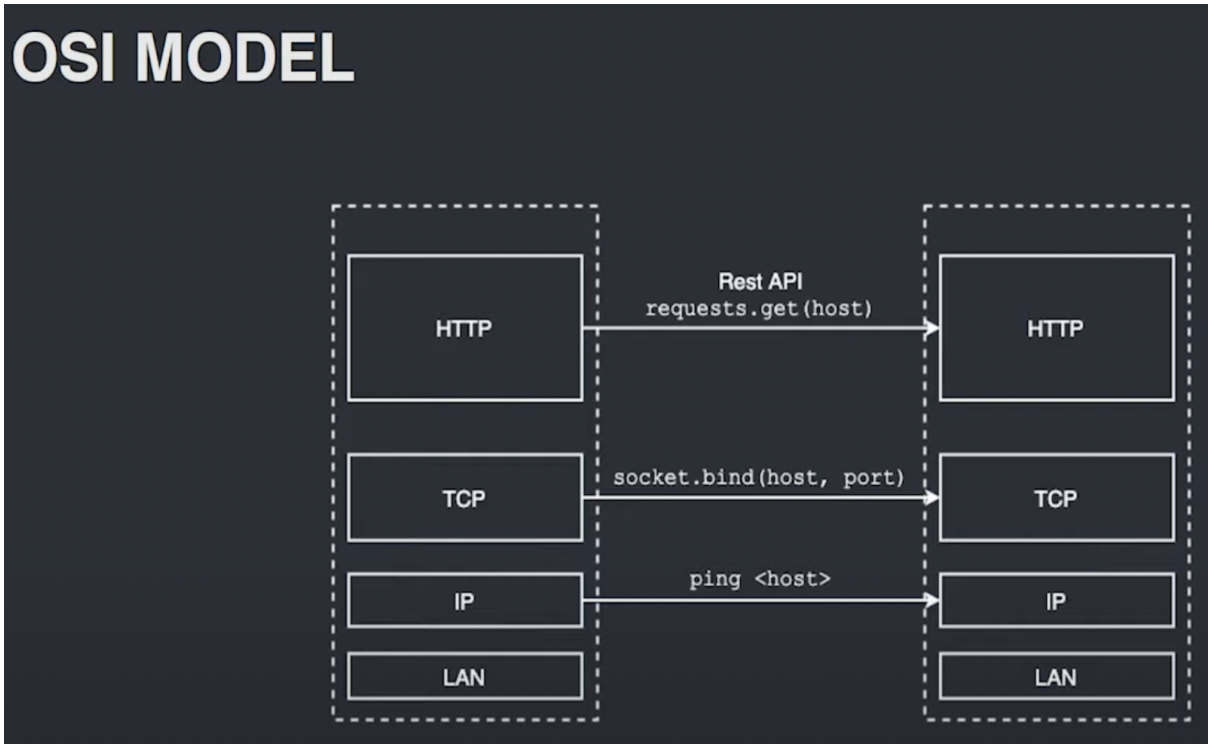
Сегодня наша цель — ознакомиться с некоторыми шаблонами проектирования веб-сервиса, с одним микросервисным паттерном, с одним шаблоном проектирования, а также выработать полезные рефлексы по разработке serving web service'ов, которые помогут нам быстро собирать легко поддерживаемые сервисы, готовые к масштабированию. В то же время необходимо помнить, что этот курс не про бэкенд-разработку, и то, что мы здесь рассмотрим, хотя и можно смело применять для создания больших и сложных бэкендов, но всё же является недостаточным.

Несмотря на то что занятие про риалтайм сервисы, многие вещи можно использовать в приложениях другого типа.

> RestAPI

> Определение REST API

Рассмотрим сетевую техническую базу Rest.



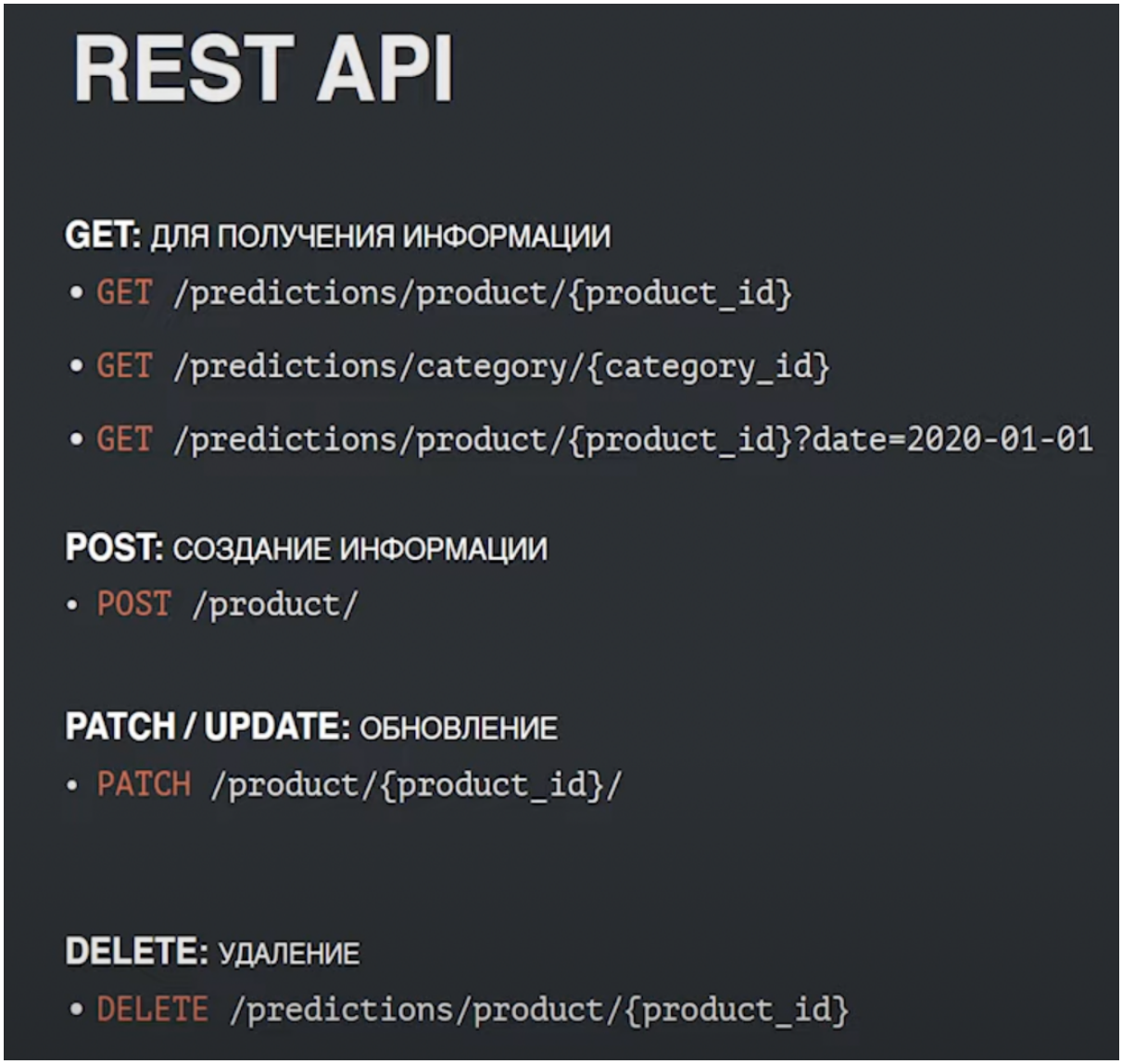
Модель OSI: есть несколько разных уровней, следующий строится на предыдущем

Команда `ping` использует IP уровень. Например, `ping yandex.ru` не проверяет доступность сайта — команда проверяет, доступен ли IP адрес, по которому располагается `yandex.ru`

TCP уровень — этот уровень используют сокеты.

HTTP уровень — на этом уровне происходит передача данных по HTTP, но это совсем не значит, что если HTTP, то и REST API. А вот если REST API, то точно HTTP.

REST API — это только архитектура, поэтому можно было бы использовать не HTTP, а другой протокол.



Эта архитектура определяет вид: "Типы запросов, формат url, формат ответов, статус-коды".

Архитектуру можно реализовать на других протоколах, но вживую вы её вряд ли увидите.

Очень упрощённый пример разницы REST и не REST

Все плюсы архитектуры:

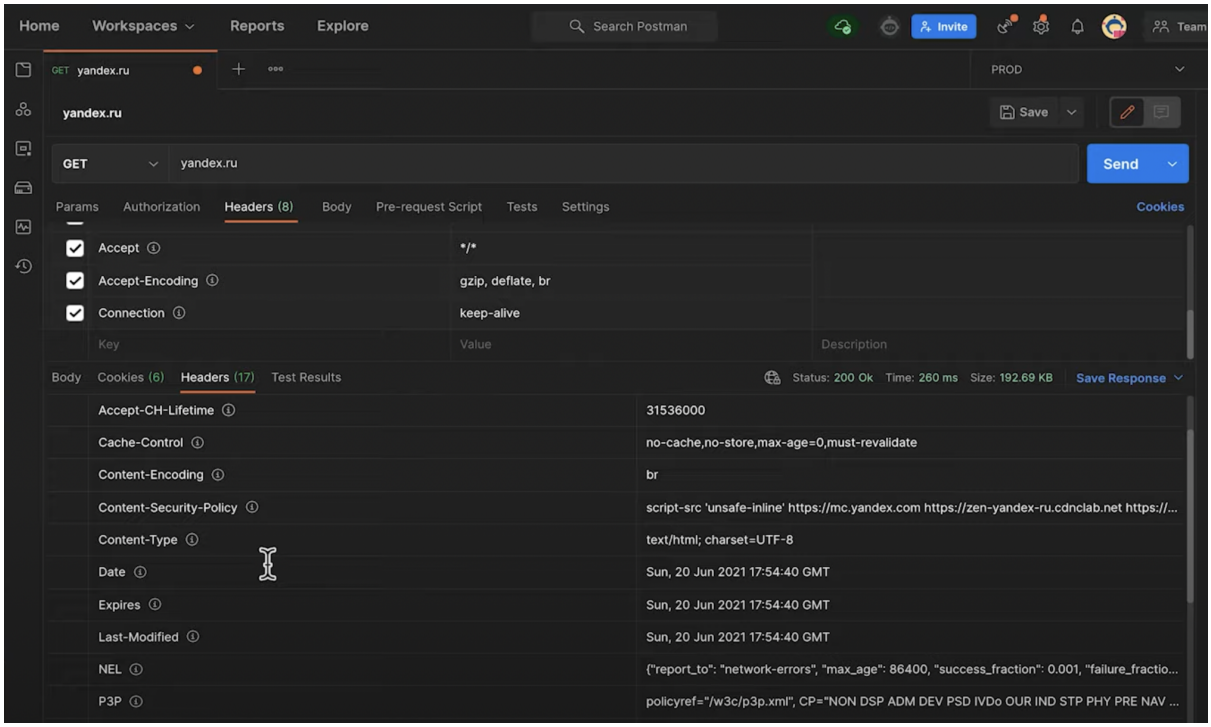
1. Она более понятная;
2. Более чистый код;
3. Ускорение разработки CRUD API во много раз.

> Немного об HTTP

Напомним про структуру HTTP-запросов/ответов с помощью программы `Postman` — советуем её установить, т.к. она сильно упрощает работу с эндпоинтами. Как видно, для составления запроса обязательными являются:

- Метод запроса
- URL

После их заполнения мы уже можем отправить запрос и получить ответ.



Результат запроса get

Нам вернулся какой-то ответ, но в запросе также использовались данные, которые мы не заполняли, потому что они по умолчанию проставлены `Postman` — это заголовки. Ну и конечно, тело запроса — в нашем случае оно пустое, и мы можем отправлять его в разном формате. В итоге в `http` сообщении это будет набор байтов с указанием формата, а сервис уже сам распарсит сообщение согласно формату. Например, мы можем выбрать формат JSON, чтобы вводить тело в удобном формате, которое так же, как и остальные данные, закодируется `Postman` и отправится в виде массива байтов.

Теперь обратимся к полученному ответу.

У нас имеется:

- Тело ответа, в котором нам вернулась главная страница;
- Заголовки и куки, которые тоже фактически являются заголовками;
- Статус-код ответа.

> Типы запросов

С ними вы наверняка уже знакомы. К основным типам запросов относятся:

- `GET` — для чтения;
- `POST` — для создания;

- `PATCH` вместе с `UPDATE` — для обновления;
- `DELETE` — для удаления.

У `GET` и у `DELETE` нет тела, т.е. вы можете записать в тело запроса в HTTP запросе, но это бесполезно, так как сервисы и серверы его не читают.

> URL

URL должен максимально возможным образом идентифицировать ресурс, который запрашивается или изменяется. То есть если у нас уже витрина предрассчитанных цен на товары, то возможные URL могут выглядеть следующим образом:

- `GET /predicitions/category/{category_id}`, который позволит получить предсказания по всем товарам в категории;
- `GET /predictions/product/{product_id}`, который отдаст предсказание по товару;
- `PATCH /product/{product_id}/`, который обновит информацию по товару, в результате чего вы отправите товар на пересчёт.

Все эти три URL ходили за конкретными ресурсами, и все эти ресурсы у вас существовали.

Для создания ресурса делается почти всё то же самое:

`POST /product/` — в конце URL ставится слэш, и мы не передаём ID, просто потому что его пока нет.

ID создастся эндпоинтом и вернётся вам. Необходимо отметить, что здесь `product_id` и `category_id` — это именно часть пути до ресурса, а не query параметр, как допустим, 3-й URL GET-запроса на изображении выше.

При проектировании URL стоит понимать, что у вас ресурс, и проектировать URL нужно соответствующим образом. Чтобы добраться до конкретного ресурса, возможно, придётся пройти через несколько айдишников и для каждой потребности создать свой эндпоинт со своим URL. Некоторые в таких случаях начинают использовать тело запроса и query parameters.

Наш совет: держаться как можно дольше, потому что когда у вас будет настроен сбор логов, то анализировать их будет проще. Также в query parameters и тело может передаваться какая-то дополнительная информация, и тогда идентификатор ресурсов смешается с ней, что усложнит работу с кодами запроса.

> Коды

`Результат запроса = текстовый статус + статус код`

- С кодами всё просто — существует их список;

- Для максимальной информативности и прозрачности некоторых статус-кодов нужно делать правильный `url` (по REST с ресурсами) `404` (или `400`), ведь `400/404` — это исчерпывающее описание;
- Коды запроса;
- Дженерики в фреймворках, которые делают REST за вас;
- SWAGGER.

Резюмируем: REST — это архитектурный подход с простыми и логичными рекомендациями по проектированию URL, выбору методов запроса и статус-кодов. Он широко известен, и по нему есть много наработок, благодаря чему использовать его просто, и разработка становится быстрой.

Плюсы**:**

- Используется везде, узнаваем и понятен;
- Много удобных фреймворков;
- Легко работать при большой нагрузке;
- Гибкие форматы;
- Легко мониторить;
- Легко кешировать.

Минусы (наследуются от http):

- Нет состояния (поэтому нельзя стримить);
- Низкая безопасность.

> Flask

> Flask как веб-фреймворк

Плюсы:

- Минималистичен;
- Не отяжелён замедляющими механизмами;
- Имеет много расширений.

Минусы:

- Не проводит автоматическую валидацию данных из коробки;
- Не предоставляет нативные миграции, ORM.

Перейдём непосредственно к приложению, которое использует Rest. `Flask` самый популярный после `Django`, но почему именно Flask?

Для начала давайте убедимся, что мы не путаем веб-сервис и веб-сервер.

Веб-сервер — это:

- Управление ресурсами;
- Принятие запросов;
- Кеширование;
- И другое, о чём мы поговорим в следующей лекции.

От фреймворка требуется только удобство и быстрота работы его механизмов. Таких, например, как сериализация данных или вызов нужного обработчика с правильными параметрами.

Поэтому ориентируемся на быстроту работы механизмов и их удобство. Flask:

- Не нагружен (сессионные механизмы, сериализации, авторизация);
- Удобен в использовании.

Но несмотря на это существует множество плагинов.

Фреймворки, безусловно, отличаются по скорости из-за того, что они по-разному сериализуют данные и работают с подключениями, но эта разница для ML-сервисов обычно несущественна.

По ссылке можно ознакомиться с неплохим сравнением веб-фрейморков: они сравниваются по-честному с использованием одного и того же веб-сервера одной конфигурации (т.е. сравнивается именно перфоманс приложений, а не приложений + веб-сервера) и это делается для трёх разных частых кейсов. Кстати, в этом сравнении выгодно выделяются асинхронные веб-фреймворки, но как о них, так и о веб-серверах поговорим чуть позже.

> Архитектура приложения

> Выбираем архитектуру приложения

Правильная архитектура обладает рядом преимуществ и позволяет избегать спагетти-кода и добиваться переиспользования кода и докер-контейнеров.

Для веб-сервисов существует несколько популярных паттернов, которые можно использовать в качестве базы для архитектуры. Их суть заключается в делении приложения на определённые независимые друг от друга слои. Разберём один из них — MVC паттерн, от которого мы будем отталкиваться.

Model-View-Controller (*MVC*) паттерн (`models != ml-модели!`), который, кстати, удачно реализован в Django, разделяет приложение схематично на:

- Данные приложения;
- Пользовательский интерфейс;
- Управляющую логику.

Это бэкендовый паттерн — в нём нет ничего про ML, но мы сможем его адаптировать для нашей модели, но при этом все обучения и прочее мы не будем держать в сервинге (но об этом чуть позже).

- Данные приложения

Когда мы всю логику с данными выносим в модели, это называется паттерн фасад.

- Пользовательский интерфейсПредставления, которые в нашем случае сервят модели. Представления зависят от `db_models`. Есть паттерны, которые позволяют отделять логику представлений от логики данных (паттерн фасад), но в ML не так много бизнес-логики, чтобы эти абстракции были оправданны.
- Контроллеры или роутеры — это объекты фреймворка, которые знают соответствие представлений и урлов запросов. Они естественным образом зависят от представлений, поскольку занимаются их регистрацией в приложении. Во Flask эти объекты называются блюпринтами.

И конечно, у нас будет один инстанс самого приложения, в котором мы будем регистрировать блюпринты и который запускает процесс веб-сервиса. Код архитектурно организуем в пакеты и субпакеты, используя файлики `init.py`.

> Layout

Выбрав MVC паттерн, мы сейчас, скорее всего, имеем следующую структуру приложения:

MODEL.
слой данных.

Описание данных, с которыми работает приложение.
В этом слое можно описывать базовые операции
над этими данными.

- ORM Модели sqlalchemy
- ORM Модели redis
- Data classes
- data managers

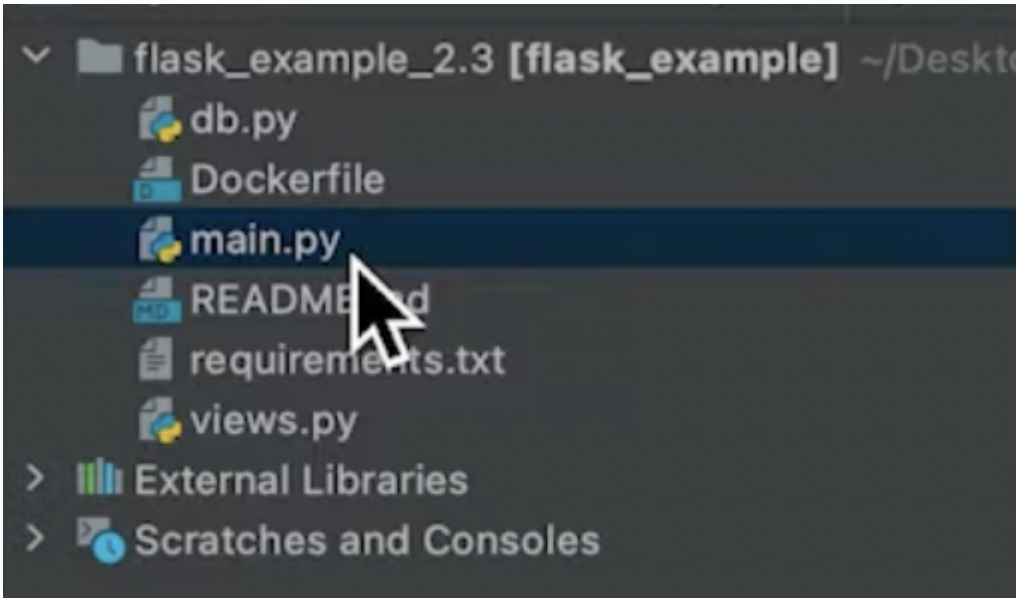
Этот слой не зависит ни от чего и проектируется первым

VIEW+CONTROLLER.
слой представления(view) и слой, создающий представление(controller)

В вебе в этом случае происходит рендеринг страницы и обработка форм/кнопок. В нашем случае мы помещаем в слой логику инференса

Слой зависит от слоя данных.
Паттерн «фасад» отвязывает зависимость слоя представления от слоя данных, но полезен только для больших приложений

В проекте пакет с приложением находится здесь:



Тут могут лежать конфиги, файлы репозиториев, энтрипоинт вашего приложения `app.py`.

В принципе, конфиги можно хранить для каждого приложения по отдельности, но лучше держать общий в корне проекта, потому что вы будете работать с ним в энтрипоинте.

Из стандартных тем мы не рассмотрели только инстансы/коннекторы к хранилищам. Django, например, внутри неявно инициализирует коннекты к базе и сам ассоциирует их с моделями — таким образом, пользователь вообще не взаимодействуют с коннекторами. Это снимает с вас ответственность за коммуникацию с базой, но такое решение очень негибкое. В остальных

фреймворках, в частности во Flask, гибкость остаётся, но всю логику нужно писать самим.

Коннекты к базе — это чаще всего локальные пулы коннектов.

Внутри каждого приложения нам нужно будет хранить драйверы к базам в отдельных модулях, например `db.py`. У каждого приложения в теории могут быть свои стораджи, поэтому проще разделять драйверы по приложениям. Смысл отделения модулей драйверов от модуля моделей проясним в следующей части.

Тут же на одном уровне с нашим приложением можно создать ещё одно с точно таким же содержимым. К примеру, это может быть приложение для healthcheck'ов в том случае, когда вам, помимо одного эндпоинта проверки активности самого приложения, хочется иметь много разных проверок, например держать статистики по последним N запросов и хелфчекать их.

> Импорты, пакеты, модульный запуск

`app.py` находится в корне, чтобы `PYTHONPATH` был правильным при запуске и импортировались модули.

Сейчас мы схематично собрали проект, он структурирован в виде отдельных приложений и общих файлов в корне репозитория.

Его структура простая и узнаваемая. Но есть и другие шаблоны — их можно поискать на гитхабе по ключевым словами "`flask-cookiecutter`" или "`ml-flask-cookiecutter`".

> Миграции

Также нам нужно где-то хранить миграции — мы храним их в папках приложений рядом с драйверами к хранилищам, потому что миграции приложений могут использовать разные базы.

Суть миграций в том, чтобы сделать работу со схемами в базах версионированной и автоматизированной. Представьте, что вы собрали релиз, изменили базу, накатили в прод и... вдруг обнаружилась критическая бага. Приложение вы смогли оперативно откатить до предыдущей версии, а вот с базой так быстро не получилось. Если релиз содержал только добавления колонок, то базу можно было даже не откатывать, но что если в релизе вы меняли ключи, переименовывали или перемещали колонки между таблицами (например, денормализовывали базу)? Такое откатить будет трудно. Также

всегда нужно учитывать стандартные проблемы, связанные с человеческим фактором, временем и т.д.

Решение

На каждое изменение в базе нужно создать специальный скрипт, в котором будет код, как накатывающий эти изменения, так и откатывающий эти изменения. Мы сможем выбирать, что именно делать: запускать накатку изменений или их откатку.

Добавим в скрипты указатели друг на друга, чтобы скрипты изменений превратились в упорядоченную последовательность. То есть скрипт №2 ссылается на скрипт с первыми изменениями, поэтому если у вас будет чистая база, то вы сможете без труда накатить на неё все изменения, запустив последовательное выполнение этих скриптов.

Если несколько человек работают с базой, то возможно повторение тех же проблем, что и с гитом: кто-нибудь забыл подтянуть актуальную версию миграций и написал новую миграцию на основе неактуальной и устаревшей, и из-за этого применение новой миграции потенциально может сломать схему. Чтобы такого не происходило, будем вести в базе учёт всех применённых миграций, каждый раз перед применением сравнивать свою историю с историей в базе и отказывать в проведении миграции, если у разработчика замечена неактуальная история. Ну и самое приятное: сделаем скрипты миграций автогенерируемыми. В скринкасте продемонстрирован полный цикл миграций с помощью инструмента `alembic`.

Инструменты миграции не идеальны, поэтому, если вы делаете не элементарные действия, то лучше перепроверять, что они для вас сгенерировали. А сложные вещи, вроде переноса колонки из одной таблицы в другую, придётся писать вручную :)

Внутри своего приложения рядом с моделями и драйверами мы хотим ещё иметь папку для миграций. Собственные миграции для каждого приложения делаются на основе той же логики, что и для драйверов.

> Проектирование элементов

Теперь от структуры и наполнения перейдём к тому, как лучше проектировать элементы архитектуры.

> Проектирование моделей, представлений и подключений к БД

Для нас интерфейс пула выглядит как одно подключение, именно поэтому объект пула должен быть один. Когда один объект должен использоваться во всей программе, то чаще всего подходит паттерн Singleton. В питоне нет реализации Singleton, но его реализацию просто написать самому — в интернете есть много примеров.

```
class Singleton(object):
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(
                Singleton, cls).__new__(cls)
        return cls.instance
```

Если вы убеждены, что синглтон — это антипаттерн или просто не хотите делать синглтон из объекта подключения, можете воспользоваться тем, что модули в питоне сами по себе синглтоны. Это значит, что можно создать один объект пула подключений на уровне модуля и импортировать его всегда из этого модуля. То есть в модулях `db` и `model`, которые мы создали в архитектуре приложения, будут жить наши как бы синглтоны, которые мы будем импортировать в представления. По такому же принципу можно разделять данные между запросами: вы так же создаёте условный словарь, который импортируете в модуль представлений как `global` переменную. Или же вы можете использовать переменные, которые предоставляет `framework` — например, во Flask это сессии `flask.sessions`, которые работают приблизительно по этому же принципу.

Кстати, Flask — это синхронный фреймворк, и поэтому все ваши действия будут выполняться по очереди, блокируя выполнение друг друга. Если в вашем представлении кроме инференса модели и запроса в основное хранилище вызываются сетевые команды, то каждая из них может подвиснуть на `N` секунд. Например, если вы дёргаете API другого микросервиса, который отвечает долго, то ваше представление зависнет в ожидании ответа, и ваш клиент, пришедший с запросом на получение предикта, будет долго ждать. Поэтому лучше явно ставить таймауты на ожидание запросов и делать эти таймауты покороче, потому что если нагрузка начнёт расти, то в

микросервисной архитектуре висящие запросы будут накапливаться лавинообразным образом — и в результате можно вообще положить всю систему. Поэтому нам лучше поставить короткие таймауты и зафейлить запрос целиком, чем подвергать риску всю систему. По тем же причинам не рекомендуется делать retry обращений к сервисам, если нет специальных показаний, потому что это опять же может создавать двойную нагрузку, а также скрывать от вас проблему довольно долгое время.

Сейчас мы обратили внимание на типичные проблемы, возникающие у вашего сервиса при необходимости обращаться во множество других ресурсов в сети. Проблемы с отказом ресурсов решаются `fault-tolerance` логикой и кешированием (как с помощью key-value storage, так и с помощью кеширования внутри приложения). В качестве примера можете рассмотреть `functools.lru_cache()`. Проблемы с большим response time из-за множества обращений решаются асинхронными фреймворками, в которых возникают уже другие заботы, о которых мы тоже расскажем, чтобы у вас был более полный базис для принятия решений.

Теперь перейдём к жизненной фабуле приложения. Тут всё довольно просто.

Во-первых, любой инженер хочет, чтобы приложение как можно быстрее сообщало не только о проблемах, которые мешают его запуску, но и о потенциальных проблемах, которые могут произойти в рантайме. Большая часть проблем связана с конфиг-файлами и переменными окружения. Чтобы их избежать, инженеру в энтрипоинте перед запуском Flask приложения следует делать парсинг всех конфигов, проверку наличия всех необходимых переменных окружения, чтобы их отсутствие не выяснилось в рантайме. Если же инфраструктура не очень зрелая (т.е. вы не в Кубере, и у вас нет service discovery), не лишним будет сделать проверочные запросы на те сервисы, с которыми приложение будет контактировать.

Во-вторых, после того как мы выполнили все возможные проверки, мы загружаем синглтон с моделью.

После первых двух пунктов приложение может инициировать подключение наших пулов или просто подключений, используя стартап-хендлеры. То есть создание объекта класса подключения происходит не в модуле `db.py` — в данном случае в энтрипоинте у этого объекта дёргается метод подключения. Если драйвер сразу производит подключение при создании объекта, то в модуле `db.py` можно, к примеру, создать глобальную переменную, присвоить ей `None`, а в энтрипоинте уже выполнить создание объекта подключения и присвоить этот объект глобальной переменной.

Почему лучше делать всё именно в такой очередности? Дело в том, что если сервис оркестрируется с высоким фактором репликации, то при наличии проблем, из-за которых приложение пойдёт на перезапуск (т.е. все реплики начнут перезапускаться), или же просто при канареечном обновлении возникнет ситуация, в которой у вас будет в 2 раза больше активных инстансов, открывших подключение, что вполне может заспамить хранилище, превысив лимит подключений к нему. Если лимит будет превышен, то часть новых реплик приложений не смогут к нему подключиться и опять уйдут в перезапуск. Что будет дальше, наверное, объяснять не нужно.

В четвёртом пункте мы минимизируем вероятность возникновения этой проблемы: мы делаем gracefull shutdown. Как вы наверняка знаете, межпроцессорное взаимодействие во многом происходит с помощью сигналов: например, когда в консоли вы нажимаете `ctrl+c` для остановки работающего питоновского скрипта, то процесс управляющего терминала, т.е. вашей консоли, отправляет питоновскому процессу сигнал `SIGINT` (signal interrupt), который просит питоновсий процесс завершиться. Но этот сигнал присылается из консоли, и если один процесс хочет прервать другой процесс, то он пошлёт сигнал `SIGTERM`.

По умолчанию `SIGTERM` и `SIGINT` завершают процесс, принимающий сигнал, но Linux предоставляет возможность процессам переопределить поведение при получении этих сигналов, что используется для обеспечения gracefull shutdown. Есть сигналы, моментально завершающие процесс без возможности блокировки или переопределения, например `SIGKILL`, а также множество других сигналов, но мы переопределим только `SIGTERM` и `SIGINT`, чтобы локально приложение тоже завершалось корректно.

Отлично, теперь осталось позаботиться о хелфчеках. Как мы уже говорили, существуют системы, которые занимаются сбором метрик.
Есть PUSH и PULL подходы.

При PULL подходе ваше приложение экспозит, т.е. предоставляет эндпоинты, которые дёргаются системой. На основе того, что вернул этой системе ваш запрос, она принимает решение, нужно ли создавать алерт или нет. Пример такой PULL системы — `Prometheus`.

> SD & client balancing

> Что такое Client-side LB

Познакомимся с ещё одним микросервисным паттерном — `Service Registry`, а также с `CLient-side Load Balancing`.

Особенности паттерна Service Discovery:

- Связывает систему, избавляя человека от необходимости конфигурации;
- Включает слежение за здоровьем сервисов;
- Гибкость в части балансировки;
- K8S и docker swarm предоставляют его из коробки;
- Его просто реализовать самостоятельно.

Балансировка на стороне клиента встречается реже, чем традиционная Server-side балансировка, по нескольким причинам:

- Для её реализации требуется изменить код на клиенте;
- Требуется настраивать больше вещей, чем обычно;
- Гегемония K8S и Docker, которые упрощают server-side балансировку и сами дают своего рода service discovery;
- Опасна, если совсем-совсем ходить на таргет (но можно использовать API-gateway);
- Дополнительная сетевая нагрузка.

Однако у балансировки на стороне клиента есть свои преимущества:

- Может быть более быстрой (прямое общение между сервисами без LB);
- Может быть более отказоустойчивой (избавляет от единой точки входа);
- Кастомная балансировка (фактически балансировщик переносится в самого клиента).

> Кейсы использования Client-side LB

Основной кейс: когда нельзя или нет смысла делать более простую Server-side LB. Например, когда есть 2 дата-центра/облака.

В 1-й лекции у нас была схема с 2-мя балансировщиками. Client-side LB — один из способов балансировать по балансировщикам.

Очевидно, когда хочется реализовать свои нетривиальные политики балансировки, для нас это означает, что мы можем выносить AB логику в

потребителей либо в отдельные API AB gateway и не городить эту логику в самих деплойментах, оставив их простыми.

Service Discovery подходит почти для всех кейсов.

> Сделай сам

Чтобы реализовать Service Discovery, вам помогут такие инструменты, как `Zookeeper`, `Consul` и т.д.

Но в своей простейшей форме такие системы могут быть просто KV стораджом.

К KV нужны HTTP обёртка и healthchecks. Поэтому давайте спланируем, как быстро сделать свой SD и CSLB с помощью KV. Для таких нагрузок можно поднять свою KV в докере, и она будет прекрасно жить, если нет общего кластера.

😀 *Если вы сейчас подумали, что ничего хорошего из таких костылей не выйдет, то следует заметить, что в некоторых известных сервисах с большим числом пользователей service discovery — это просто табличка в постгресе без всяких healthchecks и прочего, что, конечно, не поощряется и не ставится в пример.*

> Что нужно, когда по нам хотят балансироваться

Итак, что нам необходимо, если мы хотим зарегистрировать сервисы с моделями, по которым кто-то будет балансироваться? Сначала рассмотрим сценарий, когда перед репликами не стоит балансировщик.

Можно сделать следующее:

- Завести в редисе структуру `RLIST`, которая будет отвечать за список реплик, которые зарегистрировались. Затем на каждый элемент `RLIST` создать ключ, у которого будет свой `TTL`, причём это нужно делать каждый раз при обновлении. `RLIST` нужен для того, чтобы отдавать все возможные реплики для сервиса, а сами ключи нужны для того, чтобы выяснять, живы ли сервисы. Каждой из реплик придётся раз в `N` секунд ходить в KV и сообщать о том, что она жива. Если она это не сделает, то ключ исчезнет по политике TTL и потребитель больше не будет ходить на реплику. Этот сценарий подходит для тех кейсов, когда балансировка потребителя будет происходить непосредственно по репликам.

- Если группа реплик за балансировщиком, то достаточно обойтись одним ключом, который будут пушить все реплики за балансировщиком. Если хоть одна из них не отстучится, что она жива, то ключ пропадёт по политике TTL, и для потребителя группа реплик полностью исчезнет. Этот кейс неприятен тем, что репликам приходится регистрировать адрес балансировщика, что связывает руки.
- Разумеется, стараться разрегистрироваться из реджистри, удалив себя из списка при `shutdown`

> Что необходимо, когда мы хотим балансироваться

Нам необходимо следующее:

- Алгоритм балансировки;
- Внутри нашей логики ретраиться на другую реплику (собственно то, что придаёт смысл всей балансировке);
- Мейнтейнить список таргетов для балансировки, синхронизируя её с service discovery;
- Хелфчеки.

Поскольку балансировка переносится, то и сам алгоритм можно тоже перенести. Например, таким образом можно реализовать `round-robin` алгоритм. Это не единственный алгоритм балансировки — например, ещё есть `hash ring`.

Если в кейсах с сервер-сайд балансировкой внутри здоровой сети ретраи не только не нужны, но и вредны, то с клиентской — они почти необходимы. Спросите, почему? Потому что это client-side балансировка (реплики отваливаются).

Список таргетов мейнтейним как глобальную переменную благодаря питоновскому GIL.

С хелфчеками всё просто — SD системы их собирают сами, а мы с Redis придумали pull healthcheck'и.

> Полезные ссылки

- Web API performance: profiling Django REST framework

- Быстро и понятно про импорты
- Документация по импортам
- Моки моделей для тестов
- Тестирование глубинного обучения
- Про 4 паттерна (в том числе и client-side LB)
- Consul + swarm
- Service discovery with swarm
- Хорошая презентация с разными примерами
- Имплементация балансировщика на питоне
- Про регистрацию сервисов (полезная схема)
- Питоновские примеры с консулом