GeekBrains

Архитектура и шаблоны проектирования на Python

Микросервисная архитектура

На этом уроке

Плюсы и минусы. Sync или async. REST или сообщения. Переход от монолита к микросервисам.

Оглавление

На этом уроке

Минусы монолитной архитектуры

Микросервисная архитектура

- Плюсы микросервисной архитектуры
  - Независимость частей системы и, как следствие, гибкость и отсутствие хрупкости
  - Возможность очень быстро вносить изменения в систему и получать обратную связь от пользователей
  - Отсутствие Божественных объектов
- Минусы микросервисной архитектуры
  - Высокий порог вхождения разработчиков и необходимость в системном архитекторе
  - Трудность развёртывания и интеграционного тестирования
  - Риск превращении системы в распределённый монолит
  - Высокая нагрузка на сеть
- Признаки микросервисной архитектуры

Возникновение Божественных объектов и DDD

- Пример системы
- Божественные объекты и связывание системы
  - Шаг 1
  - Шаг 2
  - Шаг 3
- Domain Driven Design (DDD)
- Переход к микросервисам

Sync или async. REST и сообщения

- Синхронное
- Асинхронное
- Примеры реализации
- Шаг 1
- Шаг 2
- Шаг 3
- Итоги

Практическое задание

Дополнительные материалы

Используемая литература

Минусы монолитной архитектуры

До этого момента мы рассматривали монолитные архитектуры (классический пример многослойная архитектура), а также классические паттерны проектирования, которые позволяют избежать эрозии монолитной архитектуры и сделать систему рабочей и хорошо поддерживаемой долгое время.

Однако, как показала практика, при увеличении сложности и размера системы, поддерживать её и развивать становится всё сложнее, затраты времени и денег растут независимо от того, насколько разработчики и архитекторы следят за модульностью, тестируемостью и т. д. При этом проявляется ряд характерных признаков, которые были незаметны, пока система была менее сложной и не такой большой:

1) Запуск тестов занимает длительное время (например, несколько дней).
2) Код проекта становится таким большим, что он долго загружается в IDE и в нём трудно ориентироваться.
3) Из-за множества зависимостей система становится хрупкой и при каждом новом изменении есть риск сломать другие части (частично этого можно и нужно избегать, применяя принципы и паттерны, рассмотренные нами на предыдущих занятиях, но, как показывает практика, с течением времени система всё равно станет более жёсткой).
4) Обычно организационная структура похожа на структуру системы (закон Конвея). Поэтому команда разработки становится огромной и так же жёстко связанной.
5) В бизнес-логике системы появляется всё больше *Божественных объектов*, что ещё крепче связывает части системы.

Всё это приводит к тому, что теперь изменения (новые разработки или исправления ошибок) накапливаются и вносятся в продакшен очень редко. Если раньше можно было выпускать новую версию системы каждый день или даже чаще, то теперь она выходит раз в полгода или реже и содержит множество ошибок, на исправление которых нужно потратить еще полгода.

Микросервисная архитектура

Альтернатива монолитной архитектуре — микросервисная архитектура, которая возникала как решение проблем, описанных выше. Правильно спроектированная микросервисная архитектура позволяет решить проблемы, возникающие в больших монолитных системах, но, как и любой подход, у неё есть свои плюсы и минусы. Рассмотрим их подробнее. Начнем с плюсов.

Плюсы микросервисной архитектуры

Независимость частей системы и, как следствие, гибкость и отсутствие хрупкости

Микросервисная архитектура представляет собой набор независимых небольших программ (сервисов), которые взаимодействуют между собой не на уровне кода (например, импорта и вызова

функции), а на уровне сетевых вызовов. Таким образом, один сервис может меняться, не влияя при этом на работоспособность других.

Возможность очень быстро вносить изменения в систему и получать обратную связь от пользователей

Из-за небольшого размера одного сервиса в системе внести в него изменения и выпустить новую версию становится очень просто. Быстро проходят тесты, код программы небольшой, над ней работает небольшая независимая команда. Таким образом, можно выпускать новые версии системы максимально часто и получать обратную связь от пользователей быстро, исправлять ошибки, снова быстро вносить исправления, и так далее.

Отсутствие *Божественных объектов*

Т. к. один сервис — это независимая программа, у разработчиков есть жёсткие ограничения доступа к её внутреннему содержанию. Т. е., к сервису можно сделать обращение (например по API), но извне нельзя повлиять на бизнес-логику (как, например, можно было добавить свойства классу в монолитной системе). Это ограничение доступа препятствует возникновению *Божественных объектов* и жёсткой связи компонентов системы.

Минусы микросервисной архитектуры

На занятии по архитектуре Python-приложений мы рассматривали свойства хорошей бизнес-системы. Главными из них были гибкость и скорость внесения изменений. Микросервисная архитектура рассчитана на максимальную гибкость и хорошо подходит для больших бизнес-систем. Однако в ходе реализации возникает ряд проблем (у каждого подхода есть плюсы и минусы).

Высокий порог вхождения разработчиков и необходимость в системном архитекторе

В случае с микросервисами от разработчиков требуется хотя бы базовое понимание архитектуры, DevOps-практик и специфических инструментов (REST API, брокеры сообщений, …). Также, если раньше роль архитектора мог исполнять главный разработчик или технический лидер, то в случае с микросервисами желателен человек, который будет следить за системой в целом и способами взаимодействия её частей.

Трудность развёртывания и интеграционного тестирования

Монолитная система обычно представляет собой один запускаемый файл или одну точку входа. В случае микросервисов это отдельные программы, каждую необходимо развернуть в продакшен отдельно и после этого настроить взаимодействие между ними.

Риск превращении системы в распределённый монолит

Если в монолитной системе ошибки при проектировании влияли на функционирование системы, то в микросервисной системе это влияние усиливается. При неправильном проектировании архитектуры и взаимодействия сервисов система превратится в так называемый распределённый монолит. Это жёсткая структура, но теперь она находится не в одном месте, а распределена, и изменения в одном сервисе несут за собой изменения в других. Поддерживать такую систему в разы сложнее, чем аналогичную монолитную.

Высокая нагрузка на сеть

Признаки микросервисной архитектуры

1. Система состоит из набора небольших программ (сервисов), взаимодействующих между собой.
2. Каждый сервис — это небольшая программа. Нет чёткого понятия, насколько мал должен быть сервис, обычно говорится о «Команде двух пицц» (т. е., команда настолько мала, что им хватает две пиццы). Размер сервиса всегда зависит от конкретной задачи.
3. Сервисы могут быть написаны на разных языках программирования и использовать разные технологии. Вызовы происходят по сети.

Возникновение *Божественных объектов* и DDD

Рассмотрим на примере конкретной бизнес-системы сначала:

1) её связывание из-за возникновения *Божественных объектов*;
2) использование DDD для улучшения системы;
3) переход к микросервисам и создание более гибкой системы;
4) sync- и async-подходы к микросервисам.

Пример системы

Бизнес-система по ремонту и продаже смартфонов. Есть сайт с прайс-листом по продаже и ремонту смартфонов, на котором пользователь может сделать заказ на покупку или ремонт смартфона. После передачи смартфона в ремонт техник заказывает запчасти и запускает процесс. Также можно купить новый смартфон и оплатить различными способами (например. в личном кабинете).

Рассмотрим пример (один из возможных вариантов), как может происходить проектирование и разработка системы.

Божественные объекты и связывание системы

Шаг 1

Допустим, в самом начале владелец бизнеса хочет как можно скорее выпустить систему в продакшен и сделать рекламу и страницу с прайс-листом по ремонту смартфонов. Рассмотрим пример модели Phone на Django.

```
from django.db import models


class Brand(models.Model):
```

```
    name = models.CharField(max_length=16, unique=True, verbose_name='название')


class Phone(models.Model):
     name = models.CharField(max_length=16, unique=True, verbose_name='название
модели')
             brand     =     models.ForeignKey(Brand,     on_delete=models.CASCADE,
verbose_name='brand')
         repair_cost    =    models.DecimalField(max_digits=11,    decimal_places=2,
verbose_name='цена ремонта')
```

Пока проект небольшой и пользователи просто заходят на сайт, смотрят цену ремонта для своего телефона и заказывают обратный звонок.

Шаг 2

Мы решили добавить покупку телефонов. А также возможность создавать заказы в панели администратора и принимать оплату (может появиться личный кабинет, регистрация, авторизация, …).

У нас уже есть модель телефона и. следуя принципу DRY. разработчики вносят в неё расширение, добавляя новые свойства и методы:

```
class Phone(models.Model):
    ...
    # Возможность иметь пустое поле
           repair_cost     =     models.DecimalField(max_digits=11,     decimal_places=2,
verbose_name='цена ремонта', blank=True,
                                      null=True)
    # Новое поле: цена продажи
            sale_cost     =     models.DecimalField(max_digits=11,     decimal_places=2,
verbose_name='цена продажи', blank=True, null=True)
```

Так же добавляются новое приложение (python manage.py startapp orderapp) и модели для создания заказа и оплаты:

```
from django.db import models
from mainapp.models import Phone
from django.contrib.auth.models import User


class Order(models.Model):
             user     =     models.ForeignKey(User,     verbose_name='покупатель',
on_delete=models.PROTECT)


class OrderItem(models.Model):
            phone     =     models.ForeignKey(Phone,     on_delete=models.PROTECT,
verbose_name='смартфон')
    count = models.PositiveSmallIntegerField(verbose_name='количество')
            order     =     models.ForeignKey(Order,     on_delete=models.PROTECT,
```

```
verbose_name='заказ')
```

Для простоты не рассматриваются модель и способ оплаты. Модель OrderItem можно сделать более гибко, используя абстрактный класс Good для создания любых товаров, а не только телефонов.

Уже можно обратить внимание на свойства «Цена продажи» и «Цена ремонта». Если один телефон продаётся, но ремонт для него недоступен, цена ремонта будет 0 или None (нарушение принципа разделения интерфейса (Dependency Inversion Principle)).

Шаг 3

Теперь добавим в систему приложение по ремонту смартфонов (python manage.py startapp repairapp). И снова изменяем модель Phone.

```
class Phone(models.Model):
   ...
   # Новое поле: длительность ремонта
                  repaid_len      =      models.PositiveIntegerField(default=1,
verbose_name='длительность ремонта в днях')
```

Добавим модель для описания процесса ремонта:

```
from django.db import models
from mainapp.models import Phone


class Repair(models.Model):
             phone    =    models.ForeignKey(Phone,    on_delete=models.CASCADE,
verbose_name='Смартфон')
   STATUSES = (
       ('N', 'NO'),
       ('P', 'PROCESS'),
       ('D', 'DONE')
   )
              status    =    models.CharField(max_length=1,    choices=STATUSES,
verbose_name='статус')
    create = models.DateTimeField(auto_now_add=True, verbose_name='дата и время
создания')
```

После всех изменений модель Phone превратилась в классический *Божественный объект*. Её итоговый вид:

```
class Phone(models.Model):
    name = models.CharField(max_length=16, unique=True, verbose_name='название
модели')
             brand    =    models.ForeignKey(Brand,    on_delete=models.CASCADE,
verbose_name='brand')
         repair_cost   =   models.DecimalField(max_digits=11,   decimal_places=2,
verbose_name='цена ремонта', blank=True,
                                    null=True)
```

```
          sale_cost    =    models.DecimalField(max_digits=11,    decimal_places=2,
verbose_name='цена продажи', blank=True, null=True)
                    repaid_len       =       models.PositiveIntegerField(default=1,
verbose_name='длительность ремонта в днях')
```

Проблема в том, что для того, чтобы сделать заказ на покупку, не нужна цена ремонта и его время, для ремонта телефона важна не цена, а модель и бренд (чтобы понять, какие запчасти покупать, …). Все поля нужны только для прайс-листа, чтобы вывести информацию пользователю (но при дальнейшем развитии системы в этом направлении, скорее всего, количество ненужных полей будет увеличиваться).

В этой модели есть свойства и методы, которые относятся ко всем частям бизнес-системы, другие части более не независимы и не могут быть переиспользованы. В этом небольшом примере уже видна тенденция связывания монолитной системы.

Domain Driven Design (DDD)

Попробуем перепроектировать систему (пока ещё монолитную), пользуясь DDD. Идея в том, чтобы разбить систему на ряд доменов (domains). В нашем случае это может быть подсистема заказов, подсистема ремонта, подсистема продаж. И далее вместо одной модели Phone для каждого домена мы создаём свою модель Phone, только с теми свойствами и методами, которые нужны в конкретной подсистеме (домене).

Приложение «Продажи» (модель не изменилась):

```
class Phone(models.Model):
    name  =  models.CharField(max_length=16,  unique=True,  verbose_name='название
модели')
             brand     =     models.ForeignKey(Brand,     on_delete=models.CASCADE,
verbose_name='brand')
          repair_cost    =    models.DecimalField(max_digits=11,    decimal_places=2,
verbose_name='цена ремонта', blank=True,
                                       null=True)
          sale_cost    =    models.DecimalField(max_digits=11,    decimal_places=2,
verbose_name='цена продажи', blank=True, null=True)
                    repaid_len       =       models.PositiveIntegerField(default=1,
verbose_name='длительность ремонта в днях')
```

Приложение «Ремонт»:

```
from django.db import models


class Phone(models.Model):
   name = models.CharField(max_length=32)
                    repaid_len       =       models.PositiveIntegerField(default=1,
verbose_name='длительность ремонта в днях')


class Repair(models.Model):
             phone     =     models.ForeignKey(Phone,     on_delete=models.CASCADE,
```

```
verbose_name='Смартфон')
   STATUSES = (
        ('N', 'NO'),
        ('P', 'PROCESS'),
        ('D', 'DONE')
   )
             status    =    models.CharField(max_length=1,    choices=STATUSES,
verbose_name='статус')
    create = models.DateTimeField(auto_now_add=True, verbose_name='дата и время
создания')
```

В нём мы создали ещё одну модель Phone с полями, которые нужны только для ремонта (а лишние отбросили).

Приложение «Заказы»:

```
from django.db import models
from django.contrib.auth.models import User


class Phone(models.Model):
   name = models.CharField(max_length=32, unique=True, verbose_name='название')
            cost    =    models.DecimalField(max_digits=11,    decimal_places=2,
verbose_name='цена')


class Order(models.Model):
             user    =    models.ForeignKey(User,    verbose_name='покупатель',
on_delete=models.PROTECT)


class OrderItem(models.Model):
            phone    =    models.ForeignKey(Phone,    on_delete=models.PROTECT,
verbose_name='смартфон')
   count = models.PositiveSmallIntegerField(verbose_name='количество')
            order    =    models.ForeignKey(Order,    on_delete=models.PROTECT,
verbose_name='заказ')
```

Своя модель Phone с полями «Название» и «Цена».

Мы нарушаем принцип DRY (Обратите внимание, что данные по названию телефона, его стоимости и периоду ремонта будут дублироваться!), но тем самым делаем систему более гибкой.

Такой подход позволяет нам перейти от монолита к микросервисам, либо изначально правильно спроектировать гибкую микросервисную систему. Если не использовать DDD, то вместо *Божественного объекта* у нас появится *Божественный микросервис*, при модификации которого придётся менять все другие сервисы.

Переход к микросервисам

Domain Driven Design в данном случае позволил нам создать гибкий монолит, но ничто не ограждает неопытных разработчиков (а иногда и опытных) снова создать жесткую систему и *Божественные объекты* в будущем. Этому способствует хранение кода в одном месте, возможность вызова кода одного приложения в другом и общая база данных. Попробуем перейти от монолитной системы к микросервисам.

Sync или async. REST и сообщения

После применения DDD становится понятно, как можно разбить систему на микросервисы (по сути, каждый домен можно вынести в один независимый микросервис). Остаётся решить проблему их взаимодействия между собой.

Обычно выделяют два стандартных подхода: синхронное и асинхронное взаимодействие.

Синхронное

При синхронном взаимодействии у нас есть запрос к сервису и ответ на него (так работает стандартный http-протокол). В этом случае всегда должен быть инициатор и первый запрос, на который отвечает сервис. При использовании синхронного взаимодействия обычно выбирается http-протокол и REST API, хотя есть и другие альтернативные варианты, например, RPC (но рассматривать его на занятии мы не будем).

Асинхронное

При асинхронном взаимодействии сервису не требуется отправлять запрос. Он может работать в асинхронном режиме, выполняя некоторую задачу, а после её завершения уведомить об этом другие сервисы, которые, в свою очередь, среагируют на это уведомление. Здесь удобно использовать брокеры сообщений, например, RabbitMQ (есть и другие аналоги). Сообщение помещается в очередь с идентификатором, которую могут просматривать другие сервисы.

Примеры реализации

В системе будут такие сервисы:

1. Продажи (Данные о продуктах и прием заявки на ремонт и покупку).
2. Ремонт (Заказ запчастей, статус, …).
3. Заказы (Хранение информации о пользователях и заказах, оплата, ...).

Взаимодействовать с сервисом ремонта мы будем через очередь сообщений для демонстрации асинхронного варианта. Остальные взаимодействия будем делать через REST API.

Алгоритм взаимодействия с сервисом будет такой:

1) Пользователь заходит на сайт продаж и смотрит стоимость ремонта или продажи.
2) После отправления заявки на заказ мы делаем вызов по созданию заказов в сервис заказов.
3) Если это продажа, то сразу формируется заказ.
4) Если это ремонт, публикуем в очередь сообщение о заказе на ремонт.
5) Сервис по ремонту мониторит сообщения о заказах на ремонт и после получения сообщения начинает процедуру ремонта.

6) После завершения ремонта отправляем запрос сервису заказов об изменении статуса заказа на выполненный.

Шаг 1

Создадим микросервис для продаж на Django.

Код моделей:

```
from django.db import models


class Brand(models.Model):
   name = models.CharField(max_length=16, unique=True, verbose_name='название')


class Phone(models.Model):
    name = models.CharField(max_length=16, unique=True, verbose_name='название
модели')
            brand    =    models.ForeignKey(Brand,    on_delete=models.CASCADE,
verbose_name='brand')
        repair_cost   =   models.DecimalField(max_digits=11,   decimal_places=2,
verbose_name='цена ремонта', blank=True,
                                      null=True)
         sale_cost   =   models.DecimalField(max_digits=11,   decimal_places=2,
verbose_name='цена продажи', blank=True, null=True)
                  repaid_len      =      models.PositiveIntegerField(default=1,
verbose_name='длительность ремонта в днях')
```

Код views:

```
from django.shortcuts import get_object_or_404, HttpResponseRedirect
from django.views.generic import ListView
from .models import Phone
import requests


class PhoneListView(ListView):
   paginate_by = 10
   model = Phone


def sale_view(request, pk):
   phone = get_object_or_404(Phone, pk=pk)
   data = {
       'phone': f'{phone.brand.name} {phone.name}',
       'price': phone.sale_cost
   }
   requests.post('http://127.0.0.1:5000/sale/', data=data)
   return HttpResponseRedirect('/')
```

```
def repair_view(request, pk):
    phone = get_object_or_404(Phone, pk=pk)
    data = {
        'phone': f'{phone.brand.name} {phone.name}',
        'price': phone.repair_cost
    }
    requests.post('http://127.0.0.1:5000/repair/', data=data)
    return HttpResponseRedirect('/')
```

В ListView мы выводим список телефонов. В sale_view и repair_view отправляем запрос по http протоколу нашему второму микросервису по работе с заказами.

Шаг 2

Создадим микросервис для заказов на Flask. Для простоты будем пользоваться прямыми запросами в SQLite3.

Создание базы данных:

```
PRAGMA foreign_keys = off;
BEGIN TRANSACTION;

DROP TABLE IF EXISTS orders;
CREATE TABLE orders (id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL UNIQUE,
phone VARCHAR (32), price INT, is_sale BOOLEAN, status VARCHAR(32));

COMMIT TRANSACTION;
PRAGMA foreign_keys = on;
```

Код приложения на Flask:

```
from flask import Flask, json, request
import sqlite3
import pika
from settings import TABLE_NAME, DB_NAME

app = Flask(__name__)
connection = sqlite3.connect(DB_NAME, check_same_thread=False)
cursor = connection.cursor()


@app.route('/')
def index():
    statement = f"SELECT * FROM {TABLE_NAME}"
    cursor.execute(statement)
    result = cursor.fetchall()
    print(result)
    response = app.response_class(
        response=json.dumps(result),
        status=200,
```

```
        mimetype='application/json'
    )
    return response


@app.route('/sale/', methods=['POST'])
def sale():
    data = request.form
    phone = data['phone']
    price = data['price']
    is_sale = True
    status = 'DONE'
      statement = f"INSERT INTO {TABLE_NAME}(phone,price,is_sale,status) values
(?,?,?,?)"
    cursor.execute(statement, (phone, price, is_sale, status))
    response = app.response_class(
        response=json.dumps({'STATUS': 'OK'}),
        status=200,
        mimetype='application/json'
    )
    return response


@app.route('/repair/', methods=['POST'])
def repair():
    data = request.form
    phone = data['phone']
    price = data['price']
    is_sale = False
    status = 'IN PROCESS'
      statement = f"INSERT INTO {TABLE_NAME}(phone,price,is_sale,status) values
(?,?,?,?)"
    cursor.execute(statement, (phone, price, is_sale, status))

    # Публикуем сообщение
    connection = pika.BlockingConnection(pika.ConnectionParameters(
        'localhost'))
    channel = connection.channel()
    channel.queue_declare(queue='repair')
    channel.basic_publish(exchange='',
                          routing_key='repair',
                          body=phone)
    connection.close()

    response = app.response_class(
        response=json.dumps({'STATUS': 'OK'}),
        status=200,
        mimetype='application/json'
    )
    return response


@app.route('/change/', methods=['POST'])
```

```
def change_status():
    data = request.form
    phone = data['phone']
    status = data['status']
    print(status)
    statement = f"UPDATE {TABLE_NAME} set status = ? where phone = ?"
    cursor.execute(statement, (status, phone))

    response = app.response_class(
        response=json.dumps({'STATUS': 'OK'}),
        status=200,
        mimetype='application/json'
    )
    return response


if __name__ == '__main__':
    app.run(debug=True)
```

Все view работают как API, к которому можно обратиться из других сервисов. Во view заказа мы публикуем сообщение о ремонте телефона в очередь RabbitMQ. Все подписчики (другие сервисы) смогут читать сообщения из очереди и реагировать на них независимо.

Шаг 3

Создадим микросервис ремонта. Это будет просто модуль на Python с чтением сообщений из очереди и реакцией на них. Пример кода:

```
import pika
import random
import time
import requests

connection = pika.BlockingConnection(pika.ConnectionParameters(
    'localhost'))
channel = connection.channel()
channel.queue_declare(queue='repair')


def callback(ch, method, properties, body):
    """
    Обработка чтения из очереди
    """
    phone = body
    # Начинаем процедуру ремонта
    repair_time = random.randint(3, 40)
    time.sleep(repair_time)
    # После окончания отправляем запрос на обновление статуса заказа
          requests.post('http://127.0.0.1:5000/change/',   data={'phone':   phone,
'status': 'DONE'})
```

```
channel.basic_consume(queue='repair', on_message_callback=callback)

print('start')
channel.start_consuming()
```

Функция callback будет срабатывать после того, как как программа прочитает сообщение из очереди RabbitMQ. Здесь мы ждём сообщение о заказе на ремонт. После этого случайным образом имитируем процедуру ремонта и по окончании делаем вызов по API к микросервису заказов для изменения статуса заказа.

Итоги

Из этого небольшого примера видно, насколько гибко можно расширять микросервисную систему. Например, на событие может подписаться несколько сервисов и обрабатывать его независимо друг от друга, API можно расширять и изменять, при этом сохраняя старые версии API для бесперебойной работы других сервисов.

Также уже начинают проявляться распространённые проблемы. Для развёртывания такой простой системы требуется развернуть отдельно приложение на Django, приложение на Flask, очередь RabbitMQ и скрипт для сервиса по ремонту. Если какой-то сервис откажет, необходимо продумать, что делать в этом случае.

Практическое задание

В этой самостоятельной работе тренируем умения:

1. Применять на практике знания, полученные на курсе.

Зачем:

Для использования всего пройденного в будущих проектах.

1. Доделать проект в рамках практических заданий ко всем предыдущим урокам. Можно добавить любой новый полезный функционал в WSGI-фреймворк, бизнес-логику или взаимодействие с базой данных.
2. * Продумать архитектуру проекта для командной разработки (следующий курс).

Дополнительные материалы

1. По стопам лучших: микросервисная архитектура в разрезе.
2. Почему микросервисы подойдут не всем.
3. Первоисточник: «закон Конвея» / Блог компании проект «Энгельбарт» / Хабр.
4. Messaging that just works — RabbitMQ.
5. pika/pika: Pure Python RabbitMQ/AMQP 0-9-1 client library.

Используемая литература

1. Ричардсон Крис. Микросервисы. Паттерны разработки и рефакторинга. Питер 2019
2. Ньюмен Сэм. Создание микросервисов. Питер 2019