*Урок 9. Микросервисы*

Оглавление

[Введение 2](#_Toc91681895)

[Недостатки монолитной архитектуры 2](#_Toc91681896)

[Достоинства микросервисов 3](#_Toc91681897)

[Недостатки микросервисов 4](#_Toc91681898)

[Небольшие итоги 5](#_Toc91681899)

[Пример проекта с микросервисами 5](#_Toc91681900)

# Введение

Сегодня у нас необычная и достаточно важная тема. Это нужно для понимания особенностей взаимодействия программ друг с другом. Разберем плюсы и минусы микросервисной архитектуры, рассмотрим два основных варианта взаимодействия сервисов – синхронный и асинхронный, разберем, как научиться переходить от монолитной программы к микросервисной.

# Недостатки монолитной архитектуры

Монолитная архитектура не так плоха, если мы пользуемся паттернами, но не лишена недостатков:

* *Запуск тестов занимает длительное время (например, несколько дней).*

Монолит – это единая система, компоненты не независимы, прогон тестов занимает долгое время.

* *Код проекта становится таким большим, что он долго загружается в ide и в нём трудно ориентироваться.*

Такое бывает особенно у новичков и, если такой код мы видим в Пичарме, это очень непрезентабельно. В таком коде тяжело что-то найти, зависимости долго грузятся.

* *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 становится более жёсткой).*

Повышается хрупкость системы, изменения в одних компонентах порождают изменения в других и т.д.

* *Обычно организационная структура похожа на структуру системы (закон Конвея). Поэтому команда разработки становится огромной и так же жёстко связанной.*

Программная структура повторяет структуру тех людей, которые ее писали. Т.е. программа под какую-то фирму повторяет структуру этой фирмы. Если писал один фрилансер, у него будет один такой монолит. Если, например, два, то будут два компонента – фронтенд, бэкенд. Но если проект у нас растет, то и команда будет расти вместе с проектом. Тысячей человек управлять тяжело, нужно назначать какого-то главного менеджера и т.д.

* *В бизнес-логике системы появляется всё больше Божественных объектов, что ещё крепче связывает части системы.*

Система становится все более связанной, в нее все сложнее вносить изменения.

ВЫХОД ОДИН – перейти на микросервисную архитектуру. Но нет смысла делать это сразу. Желательно тогда, когда система стала громоздкой.

**Монолитная система выглядит так:**

Есть некоторая большая система, в ней есть модули, в которых есть функции, классы, мы можем их импортировать и настроить взаимодействие друг с другом.

**Микросервисная система выглядит так:**

В микросервисах все почти также, только каждый модуль – это отдельная программа. Эти подпрограммы могут быть написаны на других языках. Они могут взаимодействовать посредством сокетов, через REST-API-подход, через очередь сообщений. Каждая программа независима от другой.

# Достоинства микросервисов

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

Когда мы делаем такую независимую программу, то не можем ее изменить извне. Каждый сервис разрабатывается отдельной командой. Мы не можем вмешиваться в код другой команды. Каждая команда нам выставляет какой-то API, с которым мы и работаем. Получается, что программы независимые и мы не сможем их сломать.

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

Мы можем выпустить тестовую версию сервиса и передать ее пользователю, которую тот может сразу протестировать.

ПРОБЛЕМА монолитной программы еще в том, что у нас долго выпускаются версии. Когда она состоит их сервисов, т.е. мы работает с небольшими программками, этот процесс происходит динамичнее.

* *Отсутствие Божественных объектов.*

Нет компонентов, отвечающих буквально за все.

# Недостатки микросервисов

*Трудность развёртывания и интеграционного тестирования.*

Нам нужен DevOps-инженер, т.е. специалист, который занимается автоматизированным развертыванием всей системы.

*Риск превращении системы в распределённый монолит.*

Очень важно, как раз хорошо знать паттерны, разбираться в архитектуре и работать с человеком, который может все это интегрировать и поддерживать. Если этого нет, то переход к микросервисам может принести результат, что компоненты будут не независимы, а жестко связаны между собой. Придется при необходимости изменений звонить по командам отдельных сервисов и просить, чтобы они внесли изменения. Например, в проекте решили делать микросервисы, набрали разработчиков, которые ничего не понимают и в итоге получили монолит.

*Высокая нагрузка на сеть.*

Ранее были импорты и вызовы кода напрямую, теперь же вызов происходит по сети. Один сервис вызывает другой. Таким образом, получается серьезная нагрузка на сеть. Поэтому сервисы часто делают асинхронными. При асинхронности можно обработать много запросов сразу.

# Небольшие итоги

Признаки микросервисной архитектуры:

1. Программа состоит из небольших программ – сервисов, взаимодействующих между собой.
2. Как понять, что программа является сервисом? Есть принцип двух пицц. Программа должна быть такой, чтобы ее смогли писать люди, которым необходимо две пиццы на команду.
3. Сервисы могут быть написаны на разных языках, с использованием разных технологий.
4. Вызовы происходят только по сети – т.е. общение через API.
5. Отсутствие общей базы.

# Пример проекта с микросервисами

У нас будет следующая система: бизнес-система по продаже и ремонту смартфонов.

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

Как бы мы реализовали эту систему, опираясь на те средства, с которыми уже работали, например, Django:

1. Создать проект на Django
2. Создать модели «Телефон», «Заказ», «Ремонт».
3. Создать страницу с прайс-листом и необходимыми элементами управления.
4. Можно сделать ЛК пользователей, ремонтников.

**Пример 1. step1\_simple**

Начнем с какого-то простого функционала, например, выставления прайса.

Файл **models.py** – имеет две модели «Brand» и «Phone».

Сделаем шаблон для главной страницы, прайс-лист готов.

**Пример 2. step2\_orders**

Добавим возможность не только заказа ремонта товара, но и его покупки.

Файл **models.py** – две модели «Order» и «OrderItem».

И модифицируем модель «Phone».

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**)

**Пример 3. step3\_repair**

Добавим в систему функционал для мастеров.

Модифицируем модель «Phone».

repaid\_len = models.PositiveIntegerField(default=1,  
 verbose\_name=**'длительность '  
 'ремонта в днях'**)

Добавляем модель «Repair»

**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'**)

Из всех действующих лиц не каждому важно каждое из полей модели Phone:

**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'**)

Модель «Phone» уже начала превращаться в божественный объект и содержать информацию о разных областях системы.

И когда мы работаем с одной из областей, нам другая информация не нужна. Хотя в настоящий момент у нас все хранится вместе.

Почему так получается? Потому что в нашей системе мы можем делать что хотим и как хотим.

Божественный объект все испортит и не позволит нам далее развивать систему.

МОЖНО сказать, что пока в этой системе Монолит. Мы все делали на Джанго! Но эта система поможет нам спроектировать микросервисы.

Сейчас у нас «Phone» по сути является божественным сервисом. И чтобы что-то изменить в других компонентах, придется менять логику «Phone».

Здесь нам на помощь приходит DDD.

Что же такое домен и как на основании него проектировать?

Домен – некоторая область в нашей системе.

В текущем примере у нас три домена:

1. Ремонт. Техник берет телефон и его ремонтирует.
2. Продажи. Заказы и как мы их обрабатываем.
3. Прайс-лист.

Для каждого домена мы проектируем независимые модели. И модель телефона в этом случае будет для каждого домена.

Тогда модели будут независимыми и минус здесь только один – наличие дублей. Мы нарушим принцип DRY.

Но в микросервисах это не критично.

**Пример 4. step4\_ddd**

Приложение repairapp. В нем есть модель «Phone». Но включаем в нее только то, что нужно для ремонта.

**class** Phone(models.Model):  
 name = models.CharField(max\_length=32)  
 repaid\_len = models.PositiveIntegerField(default=1, verbose\_name=**'длительность ремонта в днях'**)

Теперь домен repairapp может «жить» независимо.

Также в приложении orderapp.

Мы теперь можем изолированно вносить изменения.

Но риски возникновения божественного объекта все равно остаются, хотя бы потому что у нас общая БД.

КАКОЕ решение?

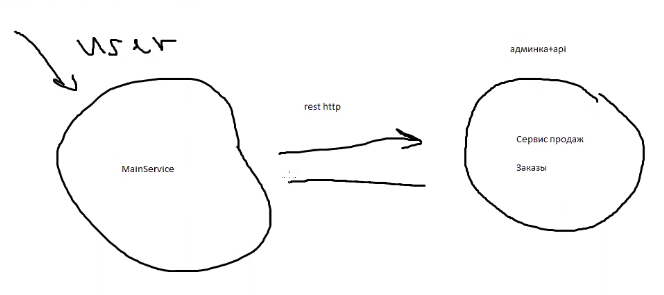
Нужно чтобы каждый домен перестал влиять на другие домены. Для этого у каждого должна быть своя база. Это раз. Мы не должны делать прямые импорты одного кода в другой. Это два.

Будем делать три разные программы с вызовом по HTTP.

Домены у нас и будут являться микросервисами.

1. У нас будет приложение прайс-листа (MainService). К нему будут стучаться пользователи.
2. Второй сервис – будет админка (сервис продаж). Туда будут приходить заказы.

Вызовы от 1) к 2) будут осуществляться по REST-протоколу, т.е., по сути, по HTTP.

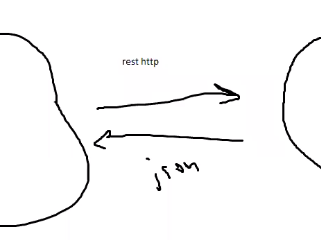


Здесь будет выполняться обмен данными в формате JSON.

Этот сервис уже будет взаимодействовать через очередь с третьим сервисом. В сервисе продаж добавляется опция ремонта и если мы инициируем ремонт, то отправляется сообщение в очередь. Третий сервис получает сообщение из очереди. И начинает ремонт. Как только ремонт закончен, отправляется сообщение в очередь и сервис продаж получает свой ответ.



Вот эта часть:

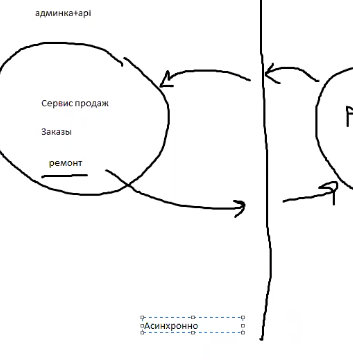


Это обычно синхронное взаимодействие и это REST API.

Синхронно это значит – есть запрос, есть ответ.

Так работает стандартный веб-сервер. Мы ему запрос – он нам ответ.

Вот эта часть:



Уже идет асинхронно. Обычно реализуется через сообщения (брокеры сообщений).

Мы отправляем запрос, но, когда получим ответ, мы не знаем.

Получается, что все, кто подключается к очереди, может читать из нее сообщения. Похоже на шину данных.

Рассмотрим упрощенную реализацию проекта.

**Пример 5. smartshop\_services**

Обратите внимание. Теперь у нас три программы.

Например, первая написана на Django. Это каталог товаров, которые посещает пользователь и откуда выбирает телефоны для покупки или ремонта.

Вторая написана на Flask. Работает без ORM, на SQL-запросах.

Третья программа – это будет просто Python-скрипт.

У первой и второй программ – свои БД.

**Проект на Джанго. Реализуем в нем доменную модель по прайс-листу. (step1\_sales\_django)**

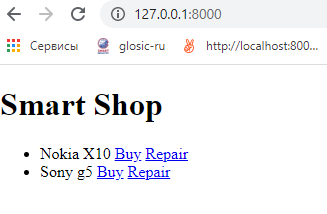
Контроллер отображения прайс-листа:

**class** PhoneListView(ListView):  
 paginate\_by = 10  
 model = Phone

Все три сервиса независимые. Их можно запускать отдельно друг от друга.

Запустим:

**python manage.py runserver**



Разберем код:

**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(**'/'**)

Мы берем из базы объект записи (модель телефона):

phone = get\_object\_or\_404(Phone, pk=pk)

Отправим запрос на сервер Фласк. Это как раз тот синхронный вариант, реализованный через API.

Формируем объект data:

data = {  
 **'phone'**: **f'{**phone.brand.name**} {**phone.name**}'**,  
 **'price'**: phone.sale\_cost  
}

Отправляем пост-запрос туда, где висит Фласк

requests.post(**'http://127.0.0.1:5000/sale/'**, data=data)

Это просто вызов другого API в другом сервисе с передачей некоторых данных.

То же самое с передачей в ремонт.

**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(**'/'**)

**Проект на Фласк (step2\_orders\_flask)**

Запустим:

**python main.py**

Это тоже сайт, у которого есть три вьюшки.

@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

Извлекает все из таблицы и возвращает ответ в формате JSON. И отдает на страницу.

Пока это пустые скобки:

print(result) -> []

Есть два метода – sale и repair:

@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

Начинается самое сложное! Как подружить два сервиса (Фласк и Джанго)?

Мы из Джанго отправляем на адрес **sale** некоторые данные.

requests.post(**'http://127.0.0.1:5000/sale/'**, data=data)

Соответственно, на Фласк данные придут нам POST-запросом. Там будут: телефон и цена.

data = request.form  
phone = data[**'phone'**]  
price = data[**'price'**]

Указываем, что телефон продан:

is\_sale = **True**status = **'DONE'**

И добавляем данные в таблицу в БД.

Формируем ответ и отправляем его на сервер Джанго.

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

Пробуем:

Купим Нокиа X 10

В этот момент ушел запрос на сервер Фласка.

Получим в браузере в таком виде список того, что есть в БД:

[[1, "Nokia X10", 1000, 0, "IN PROCESS"]]

Это был синхронный REST-API.

ТЕПЕРЬ САМОЕ СЛОЖНОЕ – взаимодействие с очередью через брокер сообщений:

Вызов из Джанго на ремонт осуществляется привычно – синхронно.

**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(**'/'**)

Во Фласке также все понятно. Наш сервис «размещает» в БД заказ на ремонт.

**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))

Но статус у нас «В процессе».

Теперь нам нужно отправить сообщение, что у нас есть заказ в третий сервис (который по ремонту).

Причем это первое сообщение в очередь может быть и синхронным, и асинхронным.

Мы как бы говорим, что ремонт начался.

НО в нашем примере мы все же отправляем асинхронно.

Во втором сервисе с помощью сторонней библиотеки **pika** мы подключаемся к нашей очереди!

Создаем канал для общения.

Создаем очередь в канале для ремонта: queue=**'repair'**.

После этого публикуем там сообщение.

И закрываем соединение.

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()

В сервис на Джанго говорим, что все в порядке.

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

**Сервис по ремонту (третий сервис)**

Это просто скрипт.

Он также подключается к каналу и очереди:

connection = pika.BlockingConnection(pika.ConnectionParameters(  
 **'localhost'**))  
channel = connection.channel()  
channel.queue\_declare(queue=**'repair'**)

И слушает, что там происходит.

Мы тем временем пишем функцию **callback()**.

Она будет отвечать на сообщения, которые пришли в канал 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'**})

Здесь мы через sleep имитируем процедуру ремонта.

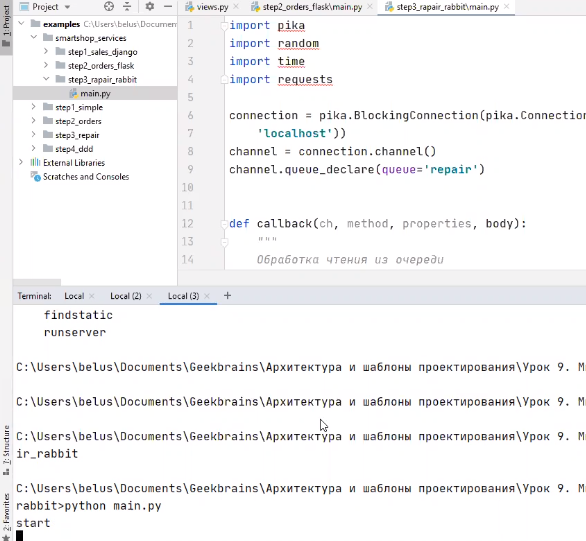
Отправляем ответ на Фласк-приложение, что все готово:

requests.post(**'http://127.0.0.1:5000/change/'**, data={**'phone'**: phone, **'status'**: **'DONE'**})

Ответ здесь, кстати, происходит синхронно.

Запускаем третий сервис:

**python main.py**



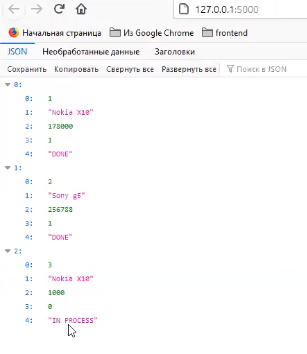
Скрипт будет висеть и ждать сообщений в очереди.

Теперь работают все три компонента.

Отправим телефон в ремонт.



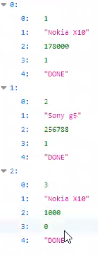
Фласк у нас получает запрос. Будет статус In Progress:



И сейчас он отправляет через очередь в третий сервис сообщение на ремонт.

При завершении ремонта – когда завершится sleep на третьем сервисе, Фласк получит ответ и выведет сообщение, что ремонт завершен:

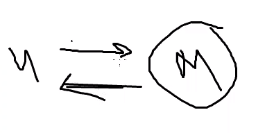
@app.route(**'/change/'**, methods=[**'POST'**])  
**def** change\_status():  
 data = request.form  
 phone = data[**'phone'**]  
 status = data[**'status'**]  
 print(status)  
 statement = **f"UPDATE {**TABLE\_NAME**} set status = ? where phone = ?"** cursor.execute(statement, (status, phone))  
  
 response = app.response\_class(  
 response=json.dumps({**'STATUS'**: **'OK'**}),  
 status=200,  
 mimetype=**'application/json'** )  
 **return** response



# Итоги проекта на микросервисах

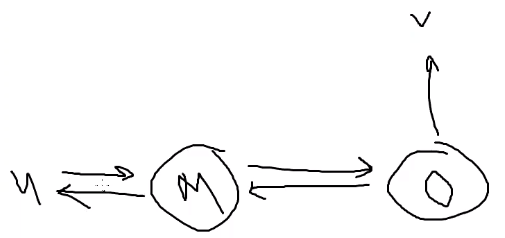
ИТАК, подведем итоги всей логики работы приложения.

1. Пользователь (U) делает запрос на получение списка товаров в приложении main (M) и получает ответ.

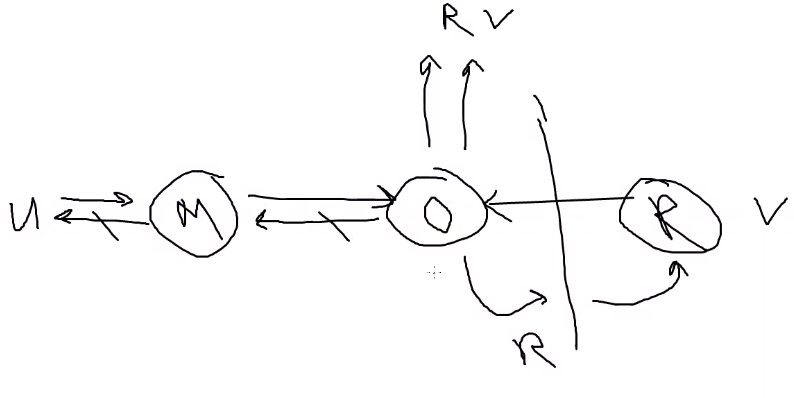


1. Пользователь (U) делает запрос на покупку телефона. Отправляет запрос в main (M).

Далее передаем запрос на Orders, где сохраняем параметры заказа в базу. И отправляем ответ пользователю, что все в порядке.



1. Самый сложный вариант – с ремонтом. Пользователь отправляет запрос на ремонт. Передаем в Order, сохраняем в базу данные. И в очередь через Orders отправляется запрос, что нужно отремонтировать. Третий сервис (R) получает запрос из очереди и начинает ремонтировать. После завершения ремонта третий сервис может послать ответ прямо в очередь – в ее новый канал. Но в нашем случае ответ идет напрямую в Orders через REST-API. Меняется статус в БД, что все выполнено (DONE). Возвращается ответ пользователю, что все готово.



В чем плюсы такого подхода? Повышается гибкость системы. А в чем минусы? Если сломается брокер сообщений Rabbit, то система «встанет».

Эта связка реализуется через Docker – то, чем занимается DevOps.