САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

02.03.02 «Фундаментальная информатика и информационные технологии»

ООП: Программирование и информационные технологии

**ОТЧЕТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ**

**Тема задания: Разработка высоконагруженного сервиса для обработки событий и генерации реакций в приложении такси с микросервисной архитектурой**

**Выполнил: Абдуллин Д.Н. 433**

Фамилия И. О. номер группы

**Руководитель научно-  
исследовательской работы: Корхов В. В., кандидат физ.-мат. наук, доцент**

ФИО, должность, ученая степень

Санкт-Петербург

2019

# Содержание

[**Содержание**](#_c7ukhllvs371) **2**

[**Термины и сокращения**](#_vgb6jslfx059) **3**

[**Введение**](#_ssxrq675wnow) **4**

[**Глава 1. Формулировка проблемы**](#_es8hmr8hgrk8) **5**

[1.1 Данные для обработки](#_mifdm8b3r0v0) 5

[1.2 Описание структуры данных и правил](#_a56v5570zkna) 6

[1.2.1 Структура данных](#_zfb31qawdt2l) 6

[1.2.2 Структура правил](#_r7eh2vo1ku5p) 6

[**Глава 2. Архитектура сервиса**](#_9lc4ymhvlyxo) **9**

[2.1 Двухзвенная архитектура](#_rnjab533u3w) 9

[2.1.1 Сервер БД](#_s1wyvjywb7r9) 10

[2.1.2 Сервер приложений](#_rymctvgugt4t) 11

[2.2 Трехзвенная архитектура](#_qzjiqt6nrnra) 12

[**Заключение**](#_gjy3ucrqbue6) **14**

[**Список использованной литературы**](#_7l1yx5urito3) **15**

# 

# Термины и сокращения

* Task Queue (tq, тэку, очередь задач) - сервис предоставляющий возможность планирования задач, автоматического перепланирования задач при неуспешном завершении
* Поставщик событий - некоторый сервис, который с помощью тэку отправляет события на обработку
* Сервис обработки событий (сос) - разрабатываемый сервис, который позволяет применять некоторый набор действий на основе заданных правил
* Правила - слабо структурированные данные в формате Json, предоставляющие информацию о необходимости применить некоторые действия к объекту при выполнении некоторых условий. Легко конфигурируемые, схематизированные
* Процессинг (от англ. processing) - обработка
* Хранилище событий (event storage, es, еэс) - сервис, предоставляющий доступ к базе данных событий

# Введение

В связи с ростом темпов жизни среднестатистического человека и дороговизной содержания собственного автомобиля возрастает спрос к услугам такси. С развитием технологий пришли удобные мобильные приложения, агрегирующие множество компаний, предоставляющих услуги развозки пассажиров (таксопарков), и предоставляющие удобный интерфейс. В разработке и поддержке такого приложения возникает много проблем, которые требуют оперативного решения. Одна из таких проблем - мотивация водителей принимать заказы, которые, возможно, им не хочется принимать по некоторым причинам. Очевидно, что случаи, когда пассажир не может уехать из-за того, что все водители отказываются от поездки, плохо влияет на лояльность клиентов по отношению к приложению. Возникает задача мотивации таксистов принимать такие заказы. Было принято решение в введении рейтинга водителей (не путать с тарифами эконом, бизнес и т.п.), в зависимости от которого он получает больший приоритет при назначении на заказ и т.п.. Возникла нужда в разработке микросервиса, позволяющего начислять мотивировать, блокировать и разблокировать водителей, отправлять пуши и другие действия в соответствии с заданными правилами.

# Глава 1. Формулировка проблемы

## 1.1 Данные для обработки

Имеется некоторый процессинг любых событий, которые возникают на стороне приложения водителя и пассажира. Таковыми являются успешное завершение заказа (order complete), отказ водителя от заказа (order reject\_manual) или ситуация, когда водитель не успел принять заказ (order offer\_timeout). Событие состоит из имени и типа - в примерах выше order это имя события, а complete, reject\_manual, offer\_timeout - типы. Получается, что мы имеем поток событий в real time и можем применять какие-то действия на их основе. Далее необходимо продумать систему поощрения и наказания, согласно которой будут расставляться приоритеты назначения поездок. Введем числовую характеристику водителя - продуктивность

Продуктивность - натуральное число от 0 до 100. Гибкий показатель, который будет изменяться за счет более широкого круга событий и на основе которого мы сможем временно блокировать водителя.

Для ясности рассмотрим пример процесса обработки заказа в нашем сервисе. Клиент вызвал такси от точки А до точки Б. Алгоритмы назначения подбирают кандидатов, при всех равных условиях приоритет отдаем водителю с более высоким показателем продуктивности. Водителю приходит заказ, но ему не понравилось, что придется ехать так далеко, создается событие order типа reject\_manual. Сервис получает это событие, находит правило согласно которому из-за такого события нужно отнять у водителя 4 продуктивности. Обновляем информацию в профиле водителя. Также проверяем историю событий этого водителя, и сверяем ее с правилами блокировки. Допустим, что у этого водителя уже третье событие такого типа за 15 минут, а у нас имеет правило, согласно которому за такие три события необходимо его заблокировать. Обновляем изменения продуктивности,и состояние блокировки в базе данных, переходим к следующей обработке.

## 1.2 Описание структуры данных и правил

### 1.2.1 Структура данных

Заказ формируется на стороне поставщика событий, затем передается очереди задач на планировка в обработку сервиса. Приходим к первой необходимой части нашего сервиса - способа обработки событий, которые приходят при помощь тэку. Заказ представлен в формате json, атрибутами которого являются уникальный идентификатор события, его имя и тип, зона, в которой это событие произошло, timestamp, некоторый идентификатор водителя и другие. Дополнительные необходимые поля можно получить из большого key-value хранилища, где представлена вся актуальная информация о заказе. Выбор именно key-value базы данных объясняется особенностью структуры и необходимостью быстрого доступа по ключу к слабоструктурированной информации об объекте.

### 1.2.2 Структура правил

Для удобства визуализации и использования будем и здесь придерживаться Json формата. Первый уровень будет задаваться зоной, в которой событие произошло. Например moscow, spb, urengoi и т.п.. Также зададим некоторую зону под названием default, в которой мы можем описать общие случаи, применимые ко всем зонам, для экономии места. На следующем уровне расположится массив из правил, которые буду иметь следующую структуру:

{

actions: [

{

action:

{

name: string,

type: string,

… # additional properties specified for each action type

},

tags: Optional[tag\_predicat]

}

],

disabled: Optional[bool],

tags: Optional[tag\_predicat],

events: [

{

name: string,

type: string,

events\_to\_trigger\_count: Optional[int],

events\_to\_trigger\_period: Optional[int],

tags: Optional[tag\_predicat],

}

]

}

Напротив каждого поля указаны тип данных, который должен быть передан в качестве аргумента. Конструкция вида Optional[int] означает, что поле не является обязательным и будет задано значением по умолчанию. Поле actions описывает действия, которые будут предприняты в случае, если все условия будут выполнены. Они заданы массивом чтобы предоставить более гибкий инструмент. Каждое действие имеет поле tags, которое позволяет применять действия только при условии, что водитель, событие имеют определенный набор тэгов. Тэгом мы называем некоторый динамический признак сущности. Под динамическим подразумевается, что такой признак в реальном времени очень часто меняется, добавляется или удаляется. Хранить такие признаки как булевы поля слишком дорого и в плане занимаемого места, и в плане скорости обработки. В качестве примера можно привести тэг дальности поездки: long\_trip, medium\_trip, short\_trip. Применение становится очевидным - в случае выполнения заказа мы хотим по разному наградить водителя в зависимости от того, сколько времени он потратил на его выполнения. Преимущество такого подхода заключается в крайней гибкости:

* Нет необходимости добавлять изменения в коде, если со стороны бизнеса приходят новые требования
* Нет ограничения в природе тэгов
* Возможность составления сложных условий

Один из типов данных tag\_predicat, который представляет из себя json объект, представляющий из себя предикат. Пример:

{

operation: OR,

operands: [

{

another tag\_predicat

},

...

]

}

И последний поле - events. Оно отвечает за события, которые могут послужить поводом для действий. Можно указать количество необходимых событий, временной промежуток рассматриваемых событий и опять же тэги.

# Глава 2. Архитектура сервиса

Начать проектирование стоит с политикой взаимодействия с базой данных, в которой будут храниться обработанные и необработанные события.

## 2.1 Двухзвенная архитектура

Микросервисное приложение работает по принципу клиент-сервер посредством запросов (к примеру HTTP). Двухзвенная архитектура используется в клиент-серверных системах, где сервер отвечает на клиентские запросы напрямую и в полном объеме, при этом используя только собственные ресурсы, т.е. сервер не вызывает сторонние сетевые приложения и не обращается к сторонним ресурсам для выполнения какой-либо части запроса. Схематически архитектура представлена на рис. 1.

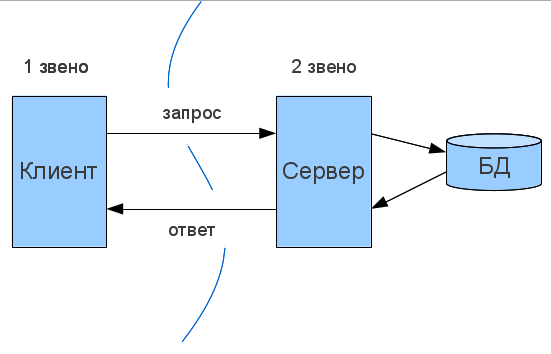


Рис. 1. Двухзвенная архитектура

Рассмотрим 2 варианта реализации двухзвенной архитектуры:

1. Сервер БД
2. Сервер приложений

### 2.1.1 Сервер БД

Рассмотрим требования, представляемые по отношению в хранилищу данных, которые сервер базы данных гарантирует:

* В любой момент времени данные должны быть непротиворечивы. Это требование заключается в соблюдении всех типов данных, ограничениях, наложенных на содержимое. Например, дата создания события представляется в виде неотрицательного целого числа (timestamp), но ограничений сверху не имеется, потому что мы можем запланировать создание события, если хотим, чтобы оно обработалось позже
* БД должна отражать некоторые правила предметной области, законы, по которым она функционирует (business rules). Количество событий вида заказ могут отражать “здоровье” бизнеса в целом, ведь отражают основную метрику бизнеса
* Необходим постоянный контроль за состоянием БД. При резком увеличении количества событий возможна необходимость подключение дополнительного узла в распределенной системе. При увеличении количества блокировок водителей, возможно стоит задуматься о смягчении правил блокировок и т.д.

Основу данной модели составляют:

* механизм хранимых процедур как средство программирования SQL-сервера
* механизм триггеров как механизм отслеживания текущего состояния информационного хранилища
* механизм ограничений на пользовательские типы данных, который иногда называется механизмом поддержки доменной структуры

Схематическое представление архитектуры представлено на рис. 2

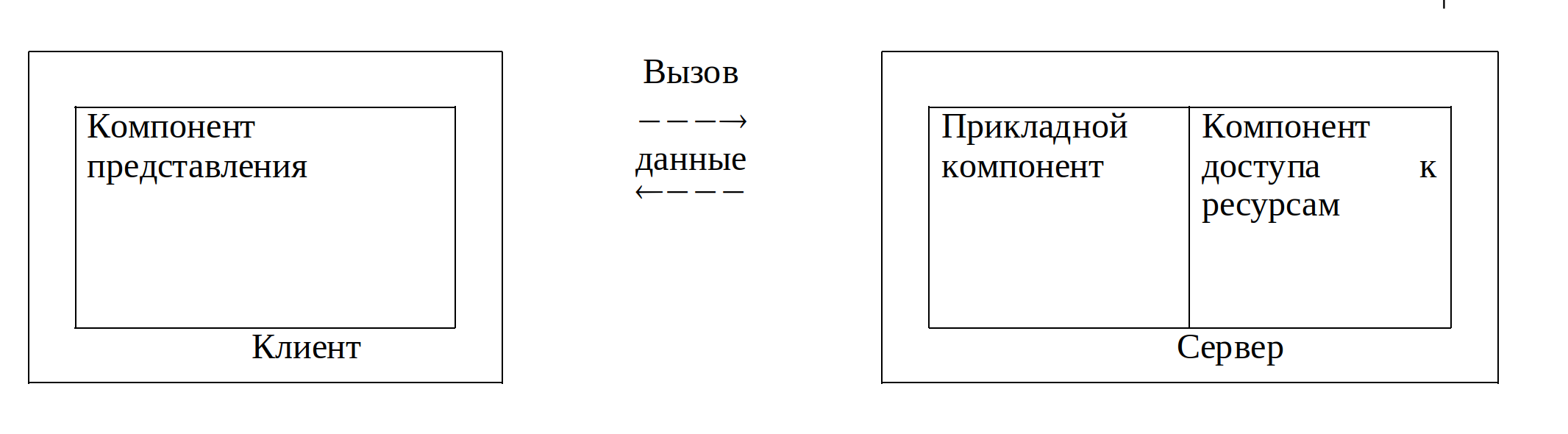


Рис.2 Архитектура сервера БД

Общение между сервером и клиентом происходит на некотором диалекте SQL. На стороне сервера непосредственная работа работа с БД происходит за счет хранимых процедур, что позволяет сэкономить время на построении плана выполнения запроса, его оптимизации и т.п.

Данную модель поддерживают большинство современных СУБД: Informix, Ingres, Sybase, Oracle, MS SQL Server. Такая архитектура позволяет избежать ошибок на стороне поставщика данных за счет системы триггеров. Но есть и существенные недостатки такие как:

* Ограниченность средств, используемых для написания хранимых процедур, которые представляют собой разнообразные процедурные расширения SQL, не выдерживающие сравнения по изобразительным средствам и функциональными возможностями с языками третьего поколения, такими как C++, Java, Python
* Сфера их использования ограничена конкретной СУБД, в большинстве СУБД отсутствует возможность отладки и тестирования хранимых процедур
* Очень большая загрузка сервера

### 2.1.2 Сервер приложений

Данное решение является логическим следствием из сервера БД. Мы хотим расширить функционал хранимых процедур, снизить требования к конфигурации клиента за счет повышения требований к производительности, безопасности и надежности сервера. Архитектура представлена на рис. 3.

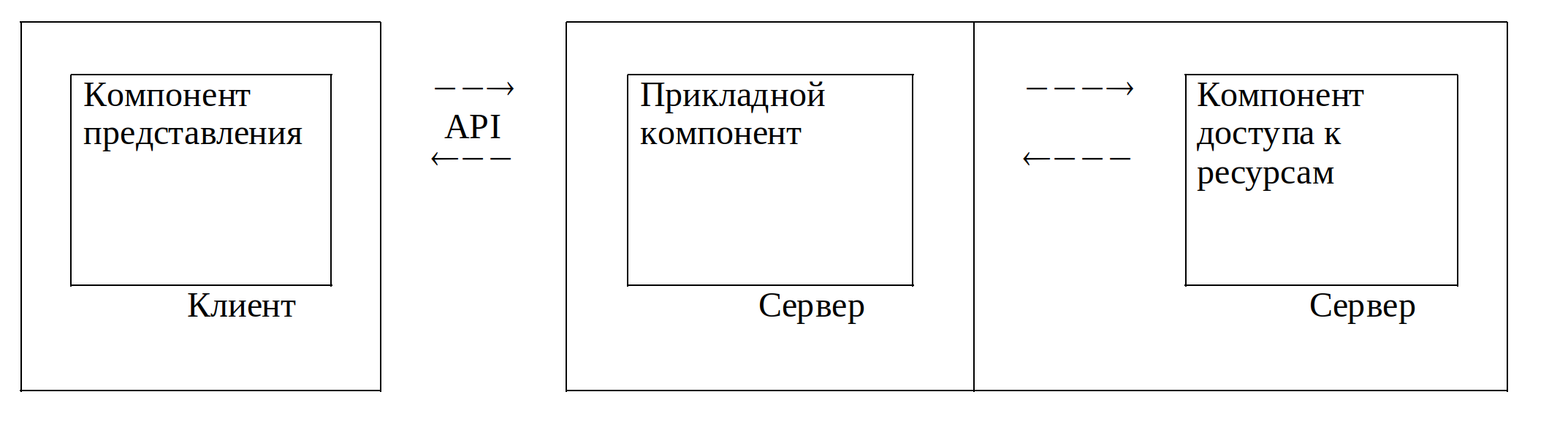


Рис. 3. Архитектура сервера приложений

Прикладной компонент выделен как важнейший изолированный элемент приложения. Для его определения используются универсальные механизмы многозадачной операционной системы, и стандартизованы интерфейсы с двумя другими компонентами. Таким образом прикладной компонент может предоставить те возможности поддержания непротиворечивости данных, которые не могут быть реализованы средствами механизма триггеров.

## 2.2 Трехзвенная архитектура

Последний вариант модели клиент-серверной парадигмы - трехзвенная архитектура. В силу высоконагруженности сервиса (порядка 150 событий в секунду) появляется необходимость в реализации распределенный вычислений. Такой подход может быть реализован по нескольким причинам:

* Несвязанность событий между собой, есть только одна необходимость - обработка событий одного водителя в одном процессе, чтобы избежать гонок данных
* Предоставлены технические ресурсы и поддержка
* СУБД, позволяющие реализовать работу с распределенными вычислениями. Реализации блокировок из-за транзакций и т.п.

Схематически архитектура представлена на рис. 4.

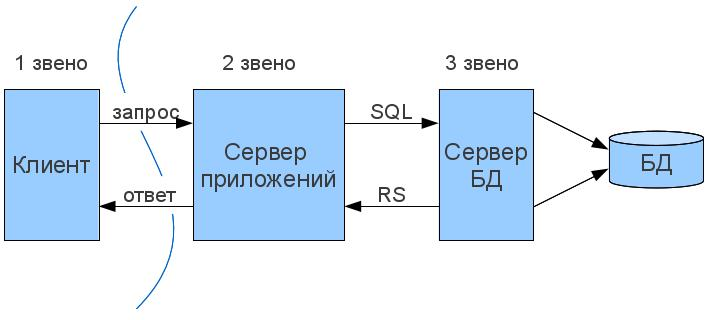


Рис. 4 Трехзвенная архитектура клиент-серверного приложения

Стоит обратить внимание, что сетевое приложение разделено на две и более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате.

Как правило, третьим звеном в трехзвенной архитектуре становится сервер приложений, т.е. компоненты распределяются следующим образом:

1. Представление данных — на стороне клиента
2. Прикладной компонент — на выделенном сервере приложений (как вариант, выполняющем функции промежуточного ПО)
3. Управление ресурсами — на сервере БД, который и представляет запрашиваемые данные

# Заключение

Двухзвенная архитектура проще, так как все запросы обслуживаются одним сервером, но именно из-за этого она менее надежна и предъявляет повышенные требования к производительности сервера.

Трехзвенная архитектура сложнее, но благодаря тому, что функции распределены между серверами второго и третьего уровня, эта архитектура представляет:

* Высокую степень гибкости и масштабируемости
* Высокую безопасность (т.к. защиту можно определить для каждого сервиса или уровня)
* Высокую производительность (т.к. задачи распределены между серверами)

На стороне СУБД будут реализованы триггеры, которые контролируют непротиворечивость данных, вакуумизация (на основе “возраста” событий), которая позволяет контролировать переполнение базы, индексация на основе наиболее часто используемых запросов, партиционирование, реплицирование на два других сервера (один мастер и 1-2 реплики) для максимальной отказоустойчивости, поддержаны механизмы, позволяющие сохранить согласованность данных между репликами, а также реплицирование в более медленное в плане скорости доступа хранилище для OLAP.

# Список использованной литературы

1 НОУ “ИНТУИТ”. Лекция - Различные архитектурные решения, используемые при реализации многопользовательских СУБД. Краткий обзор СУБД URL: https://www.intuit.ru/studies/courses/508/364/lecture/8643?page=2(дата обращения 10.12.2019).

2 StudFiles. - URL: https://studfile.net/preview/2566967/page:4/(дата обращения 15.12.2019).

3 Martin Kleppmann Designing Data-Intensive Applications - O’Reilly Media, Inc., 2017 - 613 с.