Архитектура современных информационных систем

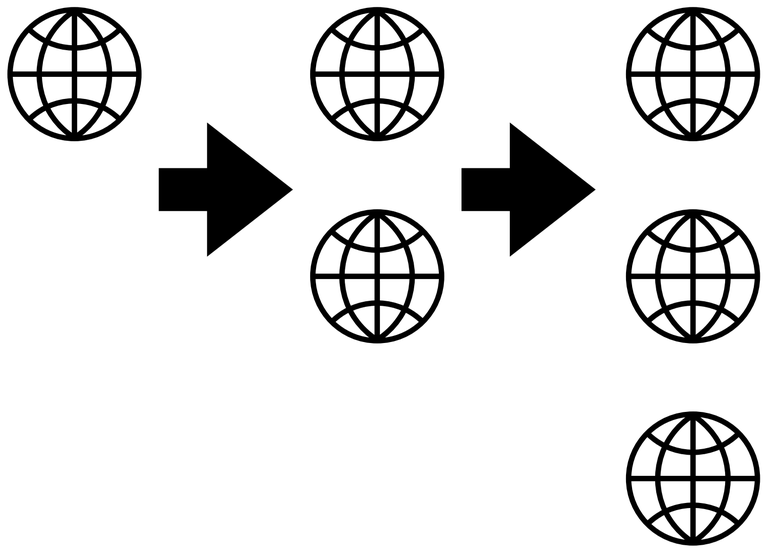
Введение

О чем думает мир

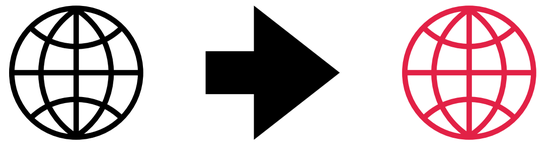
- Как обрабатывать все запросы?
- Где и как хранить данные?
- Сколько это будет стоить?

Как решить проблему высокой нагрузки?

Добавить мощности!



Придумать что-то новое!



Бизнес и разработка

Одно не может без другого

Бизнес

- Стремится минимизировать издержки
- Полагается на разработчиков в части выбора технологии
- Хочет все быстро, качественно и недорого

Разработка

- Стремится облегчить себе сам процесс
- Не любит изобретать «велосипеды»
- Сложно нормировать время
- Творческий процесс

«Чтобы создать систему, дизайн и архитектура которой способствуют уменьшению трудозатрат и увеличению продуктивности, нужно знать, какие элементы архитектуры ведут к этому»

Robert Martin, «Clean Architecture»

Что такое архитектура?

Что определяет архитектуру приложения?

- Бизнес-задача
- Потребители
- Доступный бюджет



В чем выражается архитектура?

Хорошая или плохая?

- Трудозатраты при разработке и доработке
- Красиво выглядящие схемы приложения (в хорошей архитектуре она даже схематично выглядит красиво)
- Скорость, отзывчивость, безопасность, отказоустойчивость
- Количество матов разработчиков при доработке (меньше - лучше)

Современный мир

Что есть для реализации?

- Разнообразный стек языков программирования под любые задачи (C#, Java, C++, Go, JS)
- Разнообразный стек технологий под эти языки программирования (ASP.Net, Spring, React и т.д.)
- Разнообразное ПО для ускорения доставки и развертывания (CI/CD) (Jenkins, Kubernetes)
- Разнообразное ПО для хранения и передачи данных (Apache Kafka, RabbitMQ, PostgreSQL, MongoDB)

Архитектура информационных систем

Архитектура, клиент-сервер

Виды архитектуры и виды ПО

Как может выглядеть ваше приложение?

Только клиент

Клиент-сервер

Для чего нужен клиент?

Когда вам хватить одного компьютера?

- Ваше приложение не должно коммуницировать с чем либо другим
- Ваше приложение если и содержит БД, то ему не нужно синхронизировать её с другими приложениями
- Можно пренебречь безопасностью

Для чего нужен сервер?

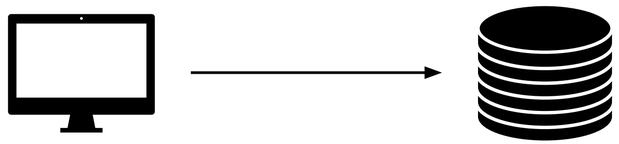
Когда нужно подключить сервер?

- Вам нужны какие-либо внешние данные, которые вы не можете получить без внешних систем
- Вам нужно синхронизировать данные с внешними системами
- Вам нужно контролировать безопасность

База данных

База данных может быть сервером, если

- Вы будете строго ограничивать роли в БД для каждого пользователя
- Вы сможете производить обработку в БД посредством процедур и функций
- Маленький проект с небольшим количеством клиентов, которые не могут конфликтовать друг с другом, одновременно меняя внешние данные
- Или нет, или есть всего 1-2 внешняя система



Сервис перед базой данных

Перед базой данных нужно ставить сервис, если

- У вас на сервере должны производится сложные расчеты или обновления данных
- У вас может быть бесконечное количество клиентов, которых нужно разводить между собой

Сервис позволяет

- Производить сложные операции обновления данных, сохраняя при этом разграничения пользователей
- Делает базу данных независимой от клиента

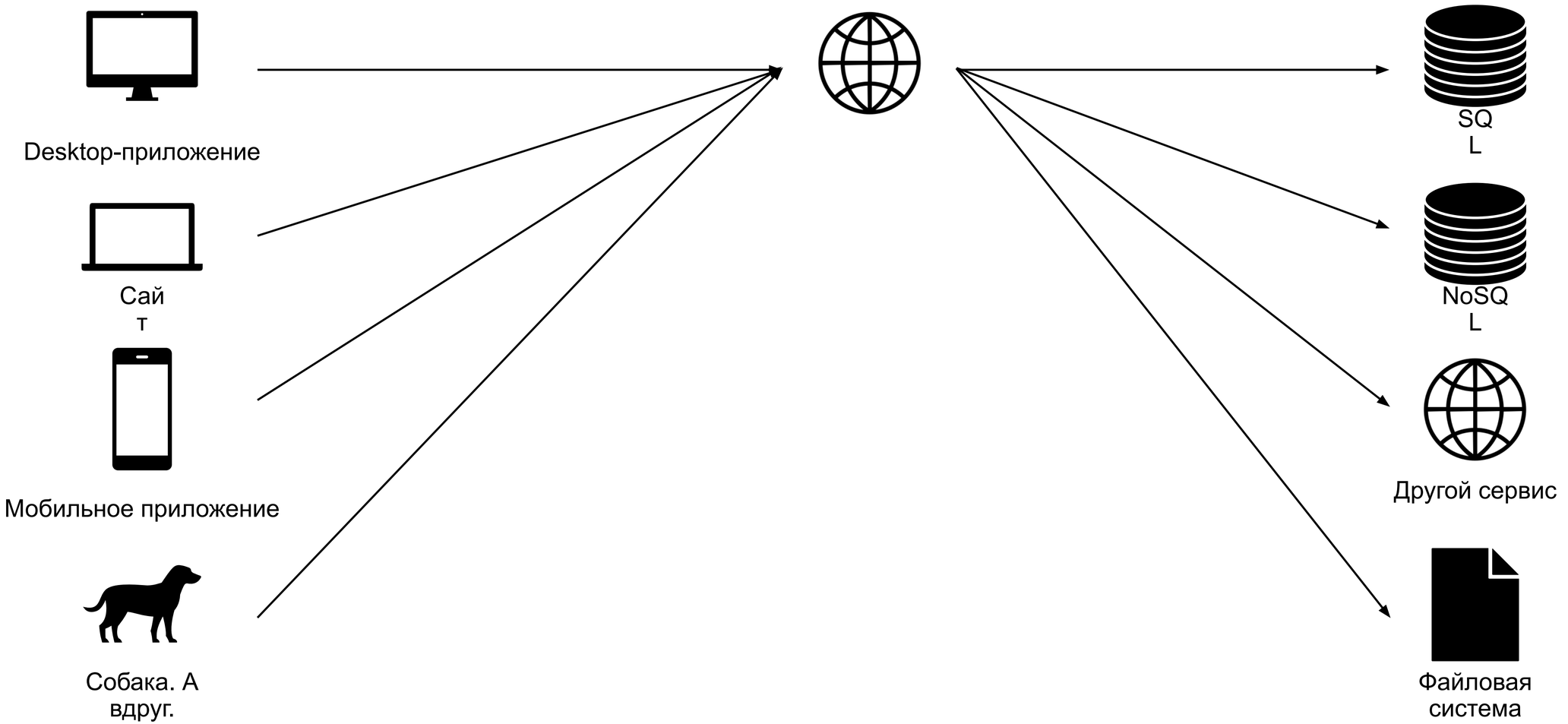
Сервис как абстракция

И причем тут SOLID

- S - Single Responsibility - каждая единица системы выполняет только одну и только свою функцию
- O - Open-Closed - единица системы открыта для расширения, но закрыта для изменения
- L - Liskov substitution - единицы системы должны быть заменяемыми другими с теми же контрактами без нарушения её работы
- I - Interface segregation - лучше много маленьких интерфейсов, чем один большой
- D - Dependency Inversion - единицы системы должны быть завязаны на абстракциях, а не на конкретной реализации

Любой клиент, любая база

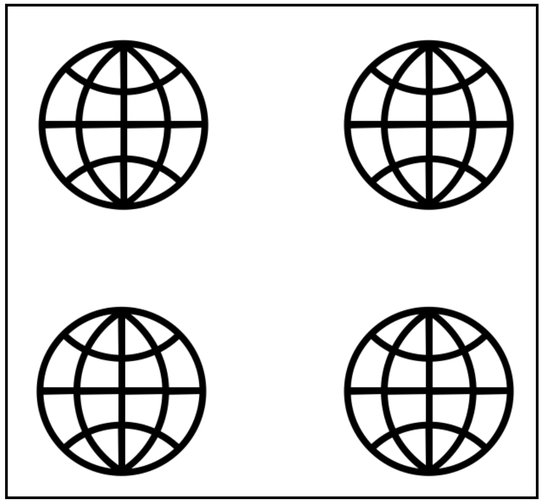
Главное - контракт



Сервис - узкое место?

А как расширить?

Для сервиса нужен особо тщательный подход к архитектуре.

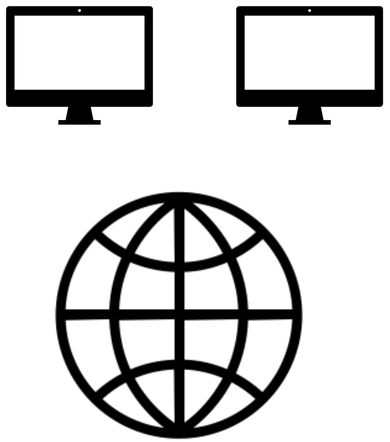


Монолиты и микросервисы

А что ж лучше?

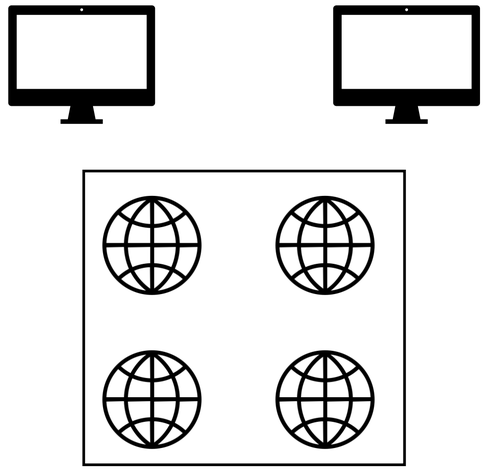
Монолит

- Выполняют множество функций
- Цельный кусок
- Легки в развертывании



Микросервисы

- Каждый сервис - одна функция
- Распределяют нагрузку
- Сложны в развертывании

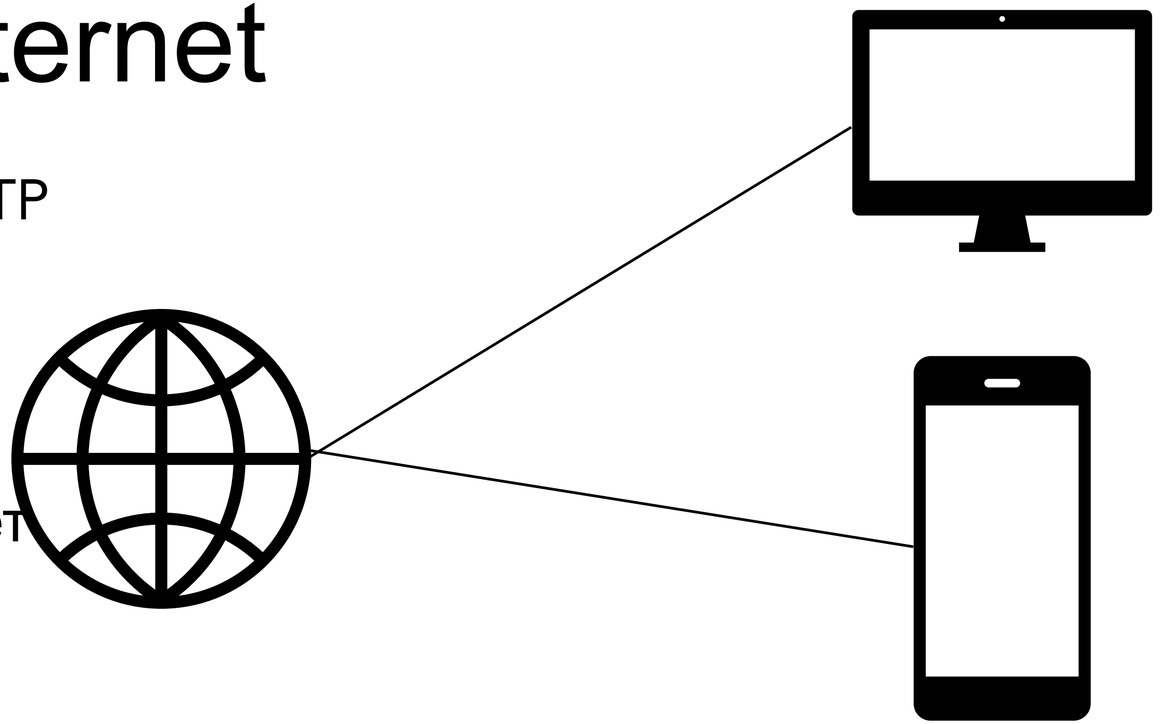


Взаимодействие

Клиент общается с сервером

Via Internet

HTTP



- Взаимодействие по протоколу HTTP
- Клиент делает запрос — сервер отвечает
- Составные части:
- Глагол: **GET** http://google.com/search?query=hello&p=20
- URL - GET **http://google.com/search**?query=hello&p=20
- Params: GET http://google.com/search?**query=hello&p=20**
- Headers — заголовки
- Body: пустое, файл, текст

REST-API

Общие положения

- Стандарт взаимодействия по протоколу HTTP
- Неофициально принят в мире для упрощения разработчикам — все сервисы по REST-API работают одинаково
- Основные глаголы: GET, POST, PUT, DELETE
- Путь=ресурс. http://localhost/students — каталог ресурсов, http://localhost/students/1 — ресурс
- Вся работа идет только с цельными ресурсами. Никаких http://localhost/addStudent.

REST-API

Глаголы

- GET — получение данных с ресурса. GET http://localhost/students — получение списка всех студентов, GET http://localhost/students/1 — получение студента с ID 1, GET http://localhost/students?q='Иванов' — поиск по списку студентов с параметрами. Идемпотентен.
- POST — создание нового ресурса. POST http://localhost/students — создание и добавление нового студента в каталог. Неидемпотентен.
- PUT — обновление существующего ресурса. POST http://localhost/students/1 — обновляет старого студента с ID 1 по новым данным. Идемпотентен.
- DELETE — удаление ресурса. DELETE http://localhost/students/1 — удаляет студента с ID 1. Идемпотентен.

REST-API

Статусы

- HTTP 200 — результат успешно получен
- HTTP 201 — создано. Может содержать ссылку на созданный ресурс и редирект.
- HTTP 301-302 — ресурс по какой-то причине перемещен навсегда/временно.

REST-API

Статусы «Ошибки клиента»

- HTTP 400 — некорректный запрос. Ошибка в параметрах.
- HTTP 401 — требуется авторизация
- HTTP 403 — авторизация прошла, но нет доступа.
- HTTP 404 — ресурс не найден
- HTTP 405 — нельзя использовать такой глагол
- HTTP 408 — время ожидания истекло
- HTTP 414 — URL слишком длинный
- HTTP 415 — неподдерживаемый тип данных
- HTTP 418 — «Я чайник»

REST-API

Статусы «Ошибки сервера»

- HTTP 500 — внутренняя ошибка сервера
- HTTP 501 — неверный метод
- HTTP 502 — Bad Gateway — ошибка шлюза/прокси
- HTTP 503 — Сервис недоступен
- HTTP 504 — Gateway Timeout — шлюз/прокси слишком долго ждал ответ

REST-API

Объекты

- Формат JSON — объект (Map, Dictionary) в читабельном человеку формате

```json
{
    "stringField": "Строка",
    "intField": 5,
    "doubleField": 5.0,
    "anotherObject": {
        "stringField": "Строка",
        "anotherIntField": 5,
        "array": [
            5,
            3,
            2
        ]
    },
    "objectArray": [
        {
            "stringField": "str"
        },
        {
            "stringField": "str1"
        }
    ]
}
```

SOAP

XML **как описание объекта**

- Тот же объект, но формате XML
- Строится по XSD-схеме
- Используется в we

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
     <getProductDetailsResponse xmlns="http://warehouse.example.com/ws">
       <getProductDetailsResult>
         <productID>12345</productID>
         <productName>Стакан граненый</productName>
         <description>Стакан граненый. 250 мл.</description>
         <price>9.95</price>
         <currency>
             <code>840</code>
             <alpha3>USD</alpha3>
             <sign>$</sign>
             <name>US dollar</name>
             <accuracy>2</accuracy>
         </currency>
         <inStock>true</inStock>
       </getProductDetailsResult>
     </getProductDetailsResponse>
   </soap:Body>
</soap:Envelope>
```

MediaType

- Form-data — Формы с веб страниц
- X-www-form-urlencoded — кодированные даты с формы
- Raw — Text, JSON, XML, HTML
- Binary — Файлы
- GraphQL

WebSocket

- Прямое соединение в Socket.
- Работает по принципу тесного взаимодействия и пересылки сообщений по сокету.
- Более тесное общение — сокет работает всегда, взаимодействие идет всегда.

Server-Sent Events

- Упрощенная версия WebSocket
- Слушатель слушает, вещатель отправляет сообщения
- Соединение постоянно

Базы данных

Типы, виды, зачем и когда нужны

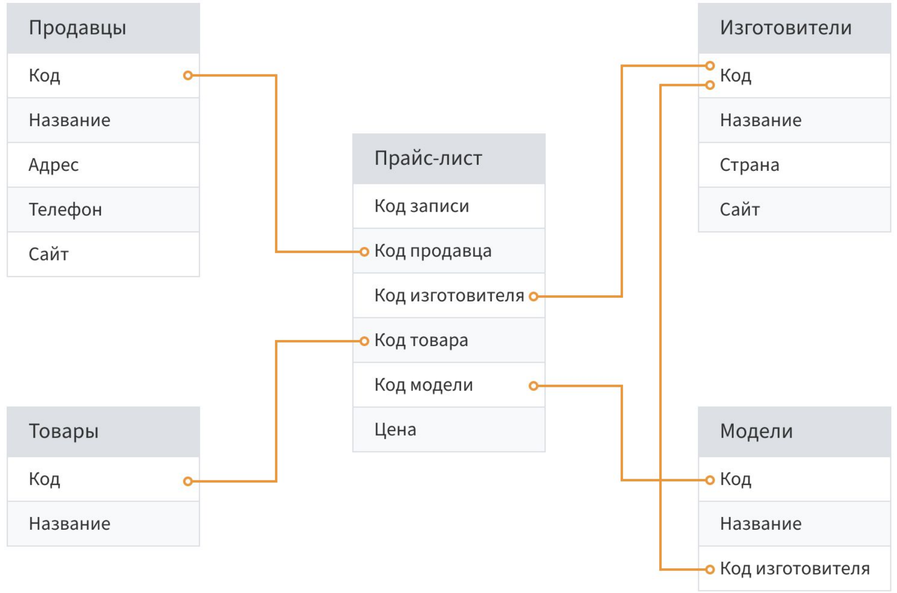
Базы данных

Общие положения

- База данных в данном случае — любое хранилище для любых данных.
- Базы данных:
- Память приложения
- Файлы
- СУБД

Базы данных

Реляционные (SQL)

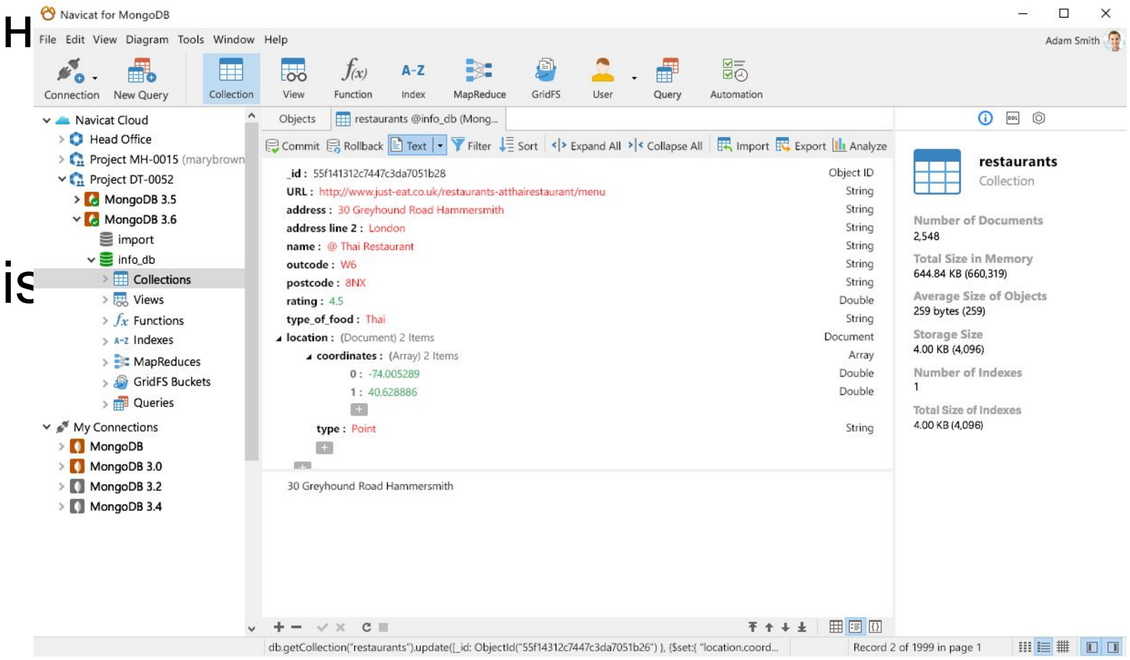


- Система взаимосвязанных таблиц
- Данные хранятся по столбцам
- Каждая таблица — отдельная сущность
- Взаимодействие через SQL-запросы
- Ключи, индексы
- Представители: MySQL, Microsoft SQL Server, PostgreSQL, OracleDB
- Транзакционность — гарантия того, что пока идет взаимодействие — данные не изменятся из-за других источников

Базы данных

Документные (NoSQL)

- Данные хранятся в виде полноценных обособленных документов/объектов
- Различные форматы хранения — JSON, файлы и т.п.
- У каждого документа свой ID (ключ-значен
- Свой язык запросов
- Представители: MongoDB, CouchDB, Redis



Базы данных

Функции

- Вызов функции БД для её обработки на стороне БД
- Если нужно единоразово собрать данные из множества таблиц/документов, при этом нет возможности сделать это в один запрос.
- Если одну и ту же сложную операцию с данными в БД нужно проводить в нескольких приложениях, работающих с этой БД

Базы данных

Кластеризация, Master **и** StandBy

- Кластеризация — несколько инстансов базы данных даже на разных серверах работают как одна.
- Master — главная БД, к которой доступны все операции, как чтения, так и записи
- StandBy — вспомогательные БД, которые доступны только для чтения и «зеркалят» данные с master-базы
- Позволяет добиться большей производительности — операции записи более «тяжелые».

Базы данных

Блокировки данных

- На случай, если одни и те же данные могут быть обработаны/использованы несколькими приложениями
- Механизм: делается запрос на изменение/обработку/взятие данных. Если до этого блокировка не была поставлена — она ставится и данные отдаются в обработку. Если блокировка уже стоит — база отвечает отказом.
- Имеет большое значение, когда инстансов приложений становится больше чем 1.

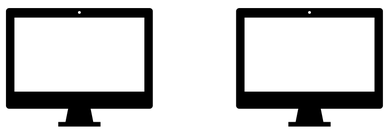
Монолиты

Как, когда и зачем

Монолиты

Самый простой способ сделать систему

- Система едина, поставляется и развертывается целиком
- Возможно отключение отдельных функций, при этом сама кодовая база останется той же
- Все компоненты системы взаимосвязаны
- Упадет одна часть системы — упадет вся система



Монолиты

Когда могут выиграть?

- Система небольшая
- Компоненты системы взаимодействуют слишком тесно и часто, чтобы дробить её
- Небольшие бюджеты
- Небольшие бюджеты на эксплуатацию, настройку и сопровождение

Монолиты

Когда могут проиграть?

- Большие системы
- Компоненты взаимосвязаны, хотя не должны
- Требуется надежность всей системы
- Требуется недопустить ситуации «Упала одна часть системы — упала вся система»

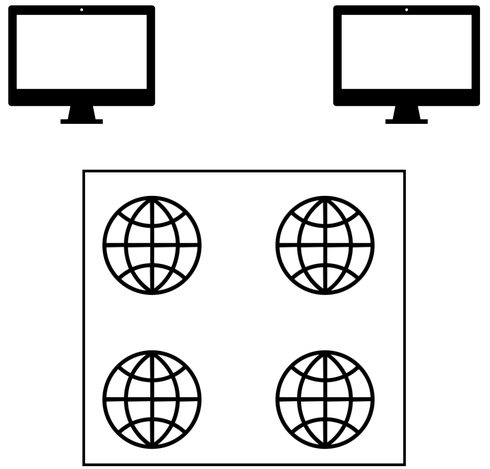
Микросервисы

Как, когда и зачем

Микросервисы

Самый сложный способ сделать систему

- Все компоненты системы разделены на мелкие приложения
- Компоненты системы общаются между собой через сеть
- Одно приложение никак не влияет на другие
- Отключение функций ведет к полному отключению отдельных приложений



Микросервисы

Когда могут выиграть?

- Система очень большая
- Компоненты системы могут работать независимо друг от друга или с минимальной зависимостью
- Есть средства и квалификация на архитектуру, развертывание и эксплуатацию

Микросервисы

Когда могут проиграть?

- Недостаточная квалификация
- Низкий бюджет на эксплуатцию и сопровождение
- Система слишком маленькая и её не требуется делить еще сильнее

Общие элементы архитектур

Балансировка нагрузки

Когда нельзя слишком сильно нагружать один компонент

- У системы есть несколько инстансов.
- Балансировка нагрузки — некий компонент сам распределяет запросы к инстансам системы и её работу в зависимости от текущей нагрузки инстансов.
- Позволяет распределить работу системы равномерно.

Gateway

Единый вход во все приложения

- У системы есть несколько инстансов.
- У системы есть несколько отдельных компонентов.
- Идет запрос к http://localhost/API/methodName. В зависимости от API/methodName Gateway сам отправит запрос нужному компоненту, получит ответ и вернет пользователю
- Позволяет сделать единую точку доступа ко всем компонентам системы и настраивать её отдельно от приложений

Аутентификациия и авторизцаия

Как отследить пользователя

- Авторизация — проверка подленности, например, по паролю.
- Аутетифкация — процесс проверки доступа пользователя к определенному ресурсу.

Аутентификация и авторизация

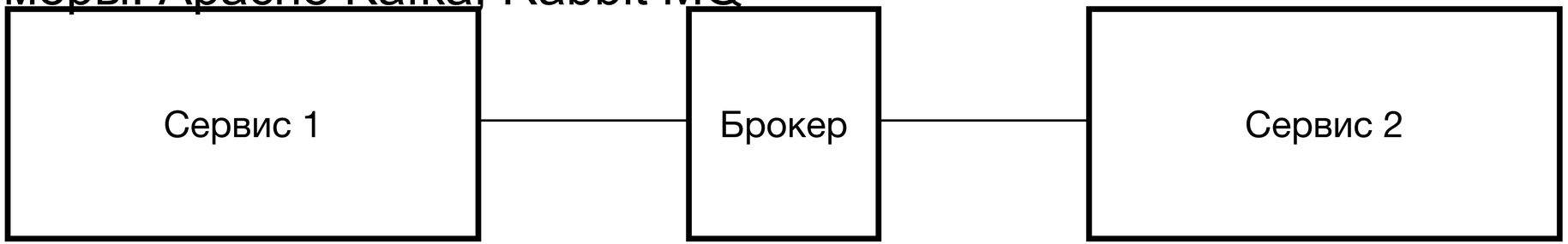
Сессии и токены

- Сессия — значение, позволяющие системе определить пользователя, что послал сообщение.
- Настраивается и управляется системой. Передается в Header-ах
- Токен — сессия, но с условием Token и Refresh Token.
- Token — токен, с которым идет обращение к системе. Имеет срок жизни.
- После того, как срок жизни Token пройдет — нужно получить новый с помощью Refresh Token, который тоже имеет срок жизни.
- После того, как срок жизни Refresh Token пройдет — нужно авторизоваться заново.

Брокеры сообщений

Что это и зачем нужно

- Работают по принципу «В компонент отправляется сообщение». И все.
- Асинхронная работа. Сообщение отправляется в брокер и компонент забывает о нем.
- Компонент не знает, кому и когда и для каких целей нужно это сообщение и будет ли оно доставлено ему и каким образом будет доставлено. Ему достаточен тот факт, что он его отправил
- Примеры: Apache Kafka, Rabbit MQ



Контейнеризация

- Контейнеризация - процесс упаковки системы в отдельную изолированную ОС.
- Ситуация: у вас есть система, которая запускается на Linux. Благодаря контейнеризации вы создаете виртуальную машину под неё на базе Linux, после чего получившийся контейнер можно будет запустить на любой ОС, которая поддерживает контейнеризацию.
- Контейнер содержит ТОЛЬКО необходимые для приложения компоненты, без прочих служб
- Все контейнеры друг от друга изолированы и независимы
- Самый распространенный механизм: Docker

Кластеризация и ноды

- Нода — одна единица системы, работающая независимо от других.
- Кластер — набор нод, работающих едино как одна система.
- Master-нода — нода, управляющая всеми остальными нодами.
- Slave-нода — нода, управляемая master-нодами.