

# Глава 5. Системный дизайн в backend-разработке

Спикер: Сергей

## Часть 1. Системный дизайн на собеседованиях и в работе backend-разработчика

Привет, меня зовут Сергей Филичкин, я ментор и python backend разработчик с опытом работы от мелких компаний, где к системному дизайну относятся халатно, до компаний с десятками тысяч сотрудников, где без хорошо спроектированной системы компания может понести многомиллионные (если не миллиардные) убытки

И сразу же нужно понять, системный дизайн для бэкенд-разработчика — это не какая-то теория из статей и учебников, это крайне важный практический навык, от которого реально зависит масштабируемость, надёжность и безопасность продуктов. Если фронтенд отвечает за пользовательский интерфейс, а мобильная разработка — за работу на устройствах, то бэкенд формирует некий фундамент системы: обработку данных, бизнес-логику, интеграции и инфраструктуру.

Моя глава будет разделена на 4 части, где в первой я расскажу про системный дизайн на собеседованиях и в работе. Во второй про масштабирование и нагрузку. Третья будет посвящена балансировке нагрузки. И четвертая — хранению данных

Итак, в первой части мы разберём:

- какие зоны ответственности есть у бэкенд-разработчика в системном дизайне;
- что ожидают на собеседованиях;
- какие типовые задачи дают и что именно оценивают интервьюеры;
- и какие архитектурные паттерны и практики нужно обязательно держать в голове.

Итак, давайте рассмотрим, за что именно отвечает backend-разработчик при проектировании системы.

И коротко поговорим основные зоны ответственности:

### **Архитектура и управление данными**

Бэкенд-разработчики отвечают за то, где и как хранятся данные. Вы проектируете схемы Базы Данных, которые должны масштабироваться от тысяч до миллионов записей, обеспечиваете целостность данных в распределённых системах и

оптимизируете запросы. Сюда также входит выбор между SQL и NoSQL базами данных, проектирование индексов и внедрение стратегий миграции

Чтобы понять о чем эта зона — Можете задать себе вопрос: “что будет, если база внезапно упадет”? Это вам поможет правильной спроектировать систему

## **Проектирование и интеграция API**

Это про то, как сервисы общаются между собой и снаружи. Вы делаете REST или GraphQL API, следите, чтобы новые версии не ломали старые клиенты, и обрабатываете ошибки сторонних сервисов.

Можете проверять простую вещь: что произойдет, если сервис ответит с задержкой и как это скажется на пользователе

## **Реализация бизнес-логики**

Тут вы превращаете бизнес-требования в уже реально работающий код. Например, проектируете платёжный процесс с учетом транзакций или делаете рекомендательную систему для миллионов пользователей и гарантируете корректную валидацию данных на всех точках входа.

Важно продумать “крайние случаи” — это те самые редкие ситуации, когда часто все и ломается. Например клиент шлет вам один и тот же запрос несколько раз или один из микросервисов или базы данных получил запрос, но ответ не вернулся из-за timeout и вы теперь не знаете применился ли эффект

## **Инфраструктура и производительность**

Бэкенд-разработчики проектируют системы под нагрузку и обеспечивают надёжность. Вы внедряете кэширование для ускорения откликов, проектируете шардинг и партиционирование БД, добавляете мониторинг для раннего обнаружения проблем и создаёте CI/CD пайплайны для безопасных и частых релизов

По сути, это будет вашей некой страховкой от внезапных ночных “вызовов на работу” и срочных релизов, если вы все сделаете правильно

При проектировании нужно помнить, что ваши решения имеют долгосрочные **последствия**:

- Например неудачная схема БД может тормозить систему годами, всего из-за одной ошибки в проектировании;
- неэффективный API так же будет тормозить весь продукт и может стать узким местом для всех пользователей;
- ошибка в безопасности может привести к катастрофическим последствиям и убыткам в десятки миллионов рублей, а иногда и долларов

Именно поэтому на этот этап уделяется особое внимание, и об этом спрашивают на собеседованиях. Причем, чем на более высокий уровень разработчика вы претендуете, тем больше доля вопросов, направленных на системный дизайн.

Теперь давайте поговорим что от вас ожидают на собеседовании

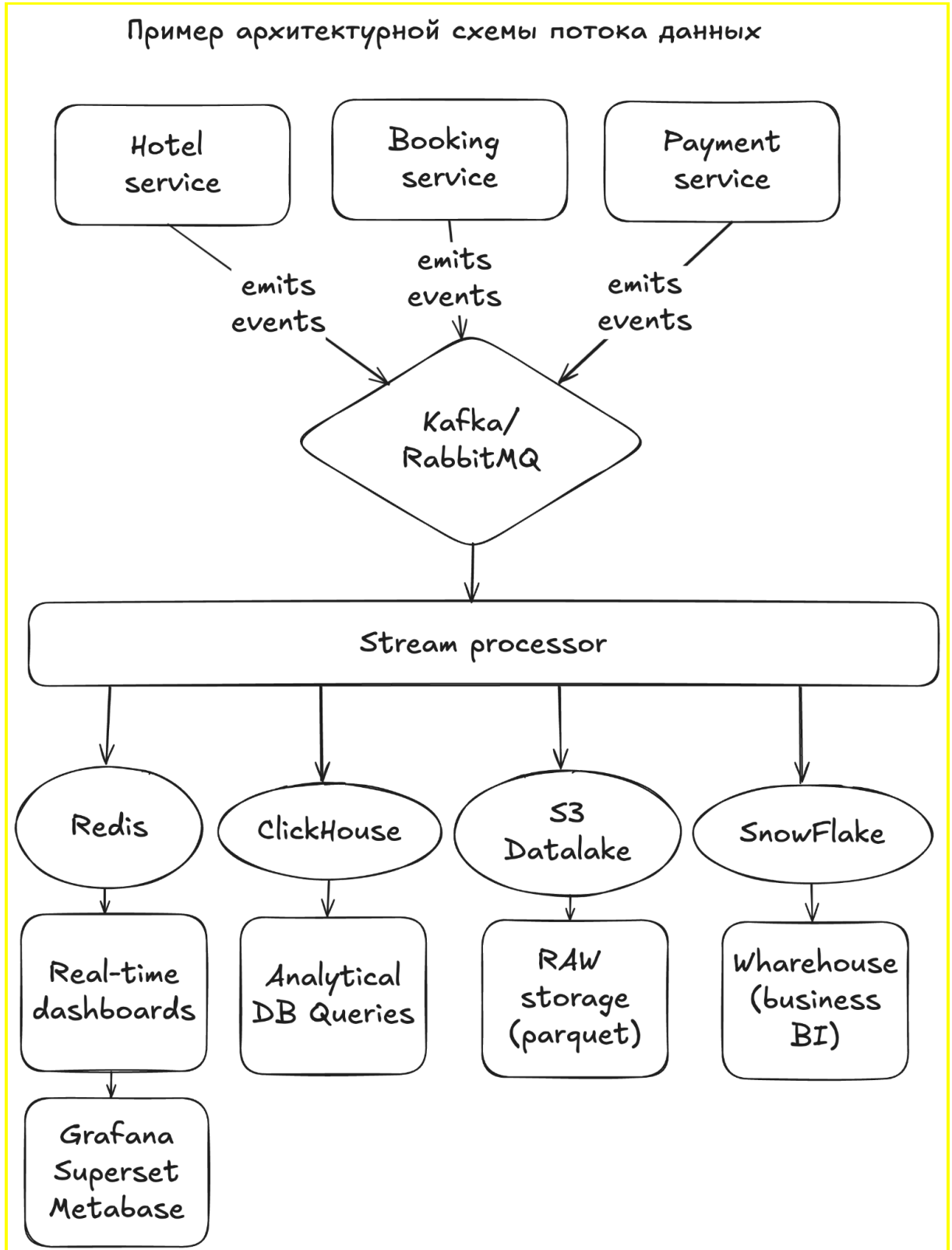
Обычно процесс собеседования состоит из нескольких этапов, но акцент делается на области, где бэкэнд-разработчик приносит наибольшую ценность: работа с данными, масштабируемость и надёжность. Обычно интервью проходит так:

1. **Вы получаете задачу на проектирование** — например, «Спроектируйте социальную сеть» или «Сервис такси». При получении такого задания, вам крайне **важно в первую очередь** задавать правильные уточняющие вопросы: какой объём данных, какие требования к консистентности, SLA и так далее. Это покажет, что вы смотрите на задачу комплексно и не бежите реализовывать сервис без полного понимания что и для кого вы делаете.
2. **Дальше проектируете архитектуру** — сначала рисуете общую схему, затем делаете акцент на бэкэнд: взаимодействие сервисов, хранение данных, отказоустойчивость, модель согласованности. Главное идти от общего к частному, сохраняя общую картину.
3. **После этого оптимизируете сделанное решение** — проработка Базы Данных, архитектуры сервисов, других оптимизаций. Могут попросить нарисовать схему таблиц, описать алгоритмы бизнес-логики или объяснить, как выдержать всплески нагрузки.
4. **И в конце могут быть уточнения** — про безопасность, эксплуатацию, альтернативные подходы. Например: почему выбрали SQL, а не NoSQL? Как работает аудит? Что с логированием?

Интервьюеру важно увидеть, что вы **не просто знаете инструменты**, а умеете рассуждать и объяснять свои решения.

При проектировании архитектуры обращайтесь внимание на:

1. **Потоки данных** — что, как и куда движется, какие преобразования происходят.



2. **Границы сервисов** — правильная декомпозиция на сервисы с балансом между монолитом и микросервисами.
3. **Необходимо Продумать структуру Базы Данных** — выбор технологии (SQL, NoSQL, графовые, time-series, поисковые), важно проектировать схему под запрос сервиса, который вы реализуете и дальнейший рост нагрузки.

4. **И, конечно, на производительность** — кэширование, оптимизация запросов, эффективное взаимодействие сервисов.

Также важно упомянуть архитектурные паттерны, безопасность и надежность, а именно:

- **Что выбрать: монолит или микросервисы** — ваша задача — найти баланс простоты и гибкости в конкретном случае
- **про событийную архитектуру** — это, кстати, хороший способ сделать систему менее зависимой между частями, более масштабируемой и устойчивой.
- **Согласованность данных:** где-то нужна строгая, где-то можно обойтись частичной — выбирайте под конкретную задачу.
- **Безопасность:** не забывайте про аутентификацию, авторизацию, шифрование, защиту от атак и нормальное логирование с аудитом.
- **и надёжность:** ретраи, circuit breaker, health-check — всё это нужно, чтобы система не падала при первом же сбое. А резервирование и disaster recovery помогут быстро восстановиться, если всё-таки что-то пойдёт не так.

На самом деле на собеседованиях часто дают типовые задачи, поэтому я бы рекомендовал их разобрать заранее. Вот некоторые из таких задач:

- **URL Shortener** — проектирование сервиса сокращения ссылок под миллиарды записей, с кэшированием, шардированием и аналитикой.
- **Чат** — реализация real-time коммуникации: доставку сообщений, хранение истории, оффлайн-очереди, масштабирование групповых чатов.
- **Лента соцсетей** — генерация персонализированных фидов под миллионы пользователей с балансом между актуальности данных и производительностью
- **Платёжная система** — здесь обязательно нужно рассказать про транзакции с ACID, борьба с мошенничеством (т.е. уделить внимание безопасности), интеграции с провайдерами, аудит.
- **Видео-платформа** — часто хотят поговорить про загрузку, CDN, подписки, рекомендации, аналитику.

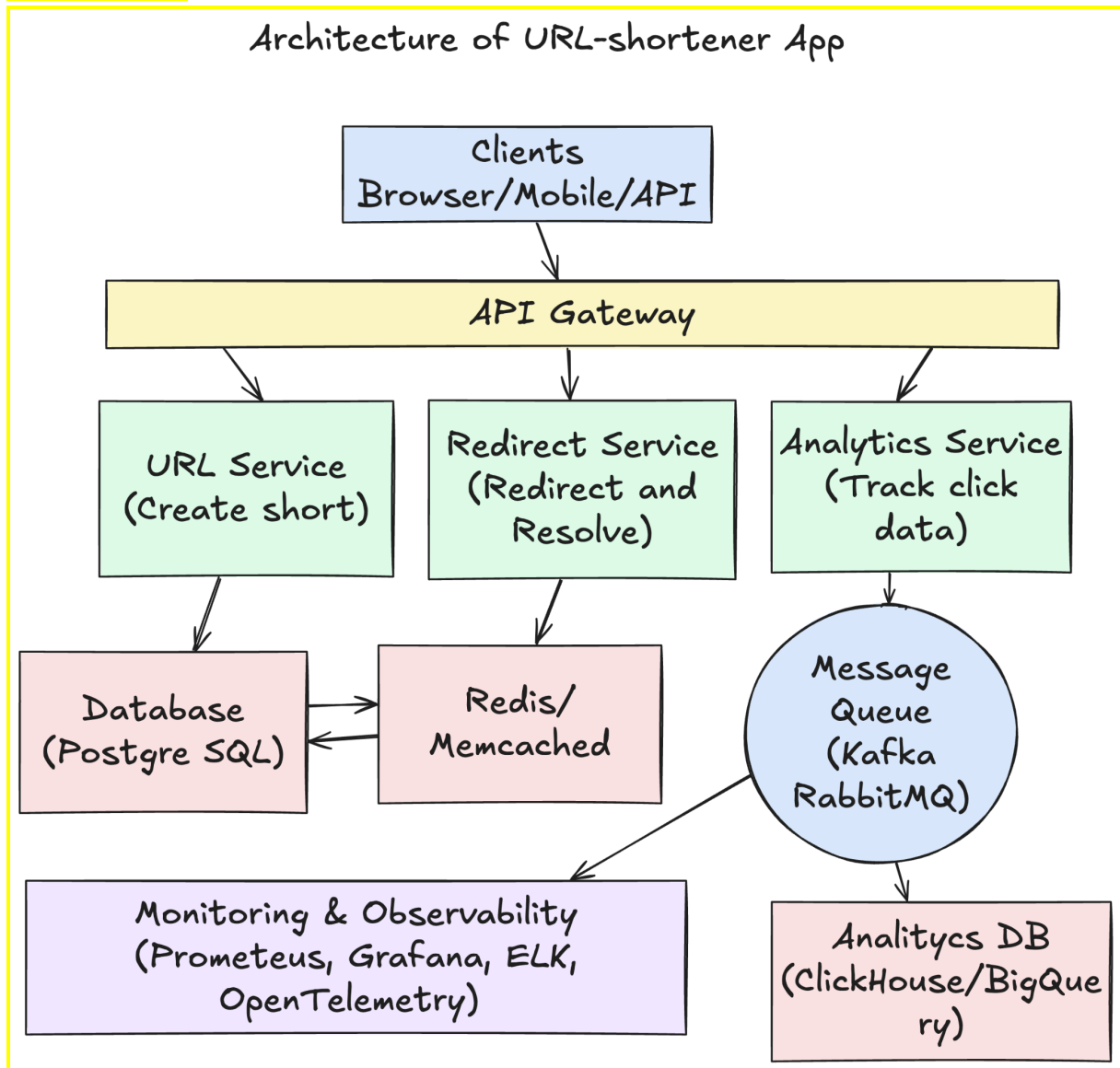
Готовясь к собеседованию, попробуйте спроектировать подобные системы и нарисовать их схему.

И обязательно отвечая на вопросы, учтите, что нужно делать это в диалоге, описывая ваши действия и задавая уточняющие вопросы, если они появятся.

Как пример можно рассмотреть супер простую схему сокращателя ссылок. Не всегда ваша система должна использовать 30 компонентов, 10 микросервисов, 5 баз данных и тд. Важно показать именно такую схему, которая применима для задачи после

уточнения требований. Если вам все-таки сказали, что система должна выдерживать миллионные нагрузки, то схема могла бы выглядеть как-нибудь так

Рисунок. Схема архитектуры одного из примеров: URL-shortener. Перерисовать в едином стиле.



Комментарий к этому рисунку(для :  
Здесь пользователь отправляет запрос, на

фронте, который может быть, например на (Next.js или React)

- Форма для ввода длинной ссылки.
- Отображение короткой ссылки и статистики (клики, переходы).

Дальше API Gateway / Load Balancer

- Балансирует нагрузку между бекенд-сервисами.

Далее уже наш Backend, который может быть написан на чем угодно FastAPI / Django / Node.js)

- **URL Service** — создаёт короткую ссылку, хранит соответствие.
- **Redirect Service** — по короткой ссылке делает редирект на оригинал.
- **Analytics Service** — может собирать статистику кликов.

Database (PostgreSQL / MySQL / Redis)

- Таблица `urls` → хранит mapping short → long URL, дату создания, owner.
- Таблица `clicks` → хранит события переходов (для аналитики).

Cache (Redis / Memcached)

- Кэширует короткие ссылки для быстрых редиректов.

Message Broker (Kafka / RabbitMQ)

- Асинхронная отправка событий кликов в Analytics Service.

Если вы объясните зачем здесь нужен каждый компонент, то интервьюеру уже может этого хватить

Как я уже говорил — на интервью не обязательно реализовывать очень сложную схему, чаще всего интервьюер обращает внимание на следующие пункты:

- **Системное мышление:** видите ли вы общую картину и взаимодействие компонентов? Думаете ли про свои и крайние случаи?
- **Моделирование данных:** умеете ли проектировать оптимальные схемы БД, выбирать подходящие типы баз, объяснять шардирование?
- **Масштабируемость:** знаете ли вы типичные узкие места, умеете ли проектировать горизонтально масштабируемые решения?
- **Практичность:** умеете ли обосновать выбор технологий, учитывать эксплуатационные риски.

Как лучше готовиться и вести себя на собеседовании:

- Обязательно изучите распределённые системы, базы данных, кэширование, безопасность и масштабируемость.
- На собеседовании уточняйте требования, покажите системное мышление, аргументируйте выбор технологий, рисуйте схемы, обсуждайте trade-off'ы.
- Не торопитесь сразу погружаться в детали реализации без архитектуры, не игнорируйте нефункциональные требования, не придумывайте то, что не нужно этой системы (это называется оверинжиниринг)

## Часть 2. Масштабирование и нагрузка: продвинутая часть

Теперь давай поговорим о масштабировании

Для backend-разработчика понятие масштабируемости выходит далеко за рамки “простого добавления серверов, когда пользователей станет больше”. На практике нужно проектировать системы так, чтобы они оставались стабильными, предсказуемыми и управляемыми под высокими нагрузками. Важно уметь отличать разные виды масштабируемости, понимать узкие места инфраструктуры и применять проверенные архитектурные решения.

Давайте разберём, какие вообще бывают виды масштабирования.

### 1. Вертикальное масштабирование (Vertical Scaling, Scale Up)

- Это когда мы усиливаем один сервер: добавляем CPU, память, более быстрые диски. Такой подход подходит для монолитов или баз данных, которые сложно разделить на части. **Но нужно понимать, что у него есть предел — “железо” нельзя добавлять бесконечно.**

### 2. Горизонтальное масштабирование (Horizontal Scaling, Scale Out)

- Добавляем новые узлы — серверы или контейнеры. Это уже ключевой подход для современных распределённых систем. Чаще всего используется вместе с балансировщиками, распределёнными БД и очередями сообщений.
- Например «ВКонтакте» изначально работал на одном PostgreSQL, но с ростом аудитории пришлось масштабироваться через десятки серверов и собственные системы хранения. Сегодня VK — это тысячи узлов, которые балансируют нагрузку между собой.

### 3. И можно дополнительно отметить — Эластичное масштабирование (Elasticity)

- Это когда система **сама** подстраивается под нагрузку: ресурсов становится больше, когда нужно, и меньше, когда трафик падает. Типичные примеры — AWS Auto Scaling, Kubernetes Horizontal Pod Autoscaler.
- **Главное помнить, что** у облаков есть так называемый “cold start” — увеличение ресурсов не мгновенное.
- Многие компании используют Kubernetes для автоматического масштабирования в пиковые дни — «Чёрная пятница», «11.11» и так далее.

Давайте теперь рассмотрим что реально применяется в продакшене

## 1. Шардинг (Sharding)

Это когда мы Разделяем данные по нескольким базам или кластерам.

Пример: часть пользователей, которых мы распределяем по определенным правилам, хранится в одном шарде, остальные — в другом.

Проблемами здесь могут быть – балансировка между шардами, миграция данных, перекрёстные запросы.

## 2. Репликация (Replication)

Здесь Данные просто копируются на несколько узлов.

Например Master-Slave (или Primary-Replica): один узел принимает записи, остальные только читают.

или Multi-Master: все узлы могут принимать записи, но сложнее обеспечить согласованность.

## 3. Event-driven архитектура и очереди сообщений

RabbitMQ, Kafka, и другие брокеры — позволяют не ждать синхронных ответов и помогают разгрузить систему. По сути Бэкенд ставит задачу в очередь, и дальше она обрабатывается асинхронно.

Благодаря этому повышается устойчивость и это нам позволяет масштабировать отдельные компоненты по мере необходимости

## 4. CQRS и Event Sourcing

- **CQRS (Command Query Responsibility Segregation)** разделяет команды (например запись) и запросы (например чтение).
- Такой подход позволяет оптимизировать разные части системы под конкретные сценарии.
- **Event Sourcing** же сохраняет все изменения в виде событий, что упрощает репликацию и восстановление состояния.
- По сути, вы храните “историю” системы, а не только её текущее состояние.

## 5. Кэширование на всех уровнях

- **Вы можете кэшировать на уровне приложения(application-level caching):** внутри backend (например, через библиотеки, в python это может быть `functools.lru_cache` или, например, Redis).

- Или использовать **распределенное кеширование** Redis Cluster, Memcached.
- Или же **CDN** для статики и API-ответов

И Для backend-разработчика критично понимать, что именно становится узким местом, чтобы понимать по какой стратегии идти

Давайте посмотрим по типам нагрузки и возможные узкие места

#### 1. CPU-bound задачи

- Это по сути вычислительные задачи (шифрование, машинное обучение, обработка изображений).
- Решением здесь может быть параллелизм, асинхронность, специализированные worker-узлы.

#### 2. Memory-bound

- Большие объёмы данных в оперативной памяти (in-memory кэш, большие коллекции).
- Решение – оптимизация структур данных, распределённые кэши (Redis Cluster, Memcached).

#### 3. I/O-bound задачи

- Это когда происходит чаще всего ожидание дисковых операций или сетевых вызовов.
- Решение: асинхронный Input/Output, batching, кеширование, использование очередей сообщений.

#### 4. И Network-bound

- Ограничение по пропускной способности сети или задержкам.
- Решение: сжатие данных, gRPC вместо REST, CDN для статических ресурсов.

Так, давайте теперь предположим, что мы уже все сделали, но система получилась непростая, постоянно возникают различные проблемы по мере увеличения нагрузки или под воздействием внешних факторов. Что мы можем сделать тут?

Есть несколько стратегий, но вы должны понимать, что это всегда компромисс. Не на все готов будет пойти бизнес, но вы должны знать об этих вариантах и как минимум предлагать их

#### 1. Graceful Degradation (Постепенное ухудшение)

- При перегрузке часть функционала можно постепенно отключать (например, отключаем рекомендации, но бронирования работают).

#### 2. Rate Limiting и Throttling

- Здесь мы ограничиваем число запросов от одного клиента.
- Чаще всего это можно сделать с помощью Nginx, Envoy, API Gateway.

#### 3. Backpressure (бэкпрэшэр)

- Механизм, когда система сообщает клиенту или producer'у, что не может принять больше данных.
  - Используется в Kafka или, например, в gRPC streaming.
4. **Circuit Breaker**
- Если сервис недоступен, временно «разрываем цепь», чтобы не заспамить его новыми запросами.
  - Реализуется через библиотеки (Hystrix, Resilience4j) или на уровне gateway.
5. **Bulkhead Pattern (Переборки)**
- Разделяем ресурсы, чтобы сбой одного компонента не смог уронить всю систему целиком
  - Например, мы можем выделить отдельные worker-пулы под разные типы задач.

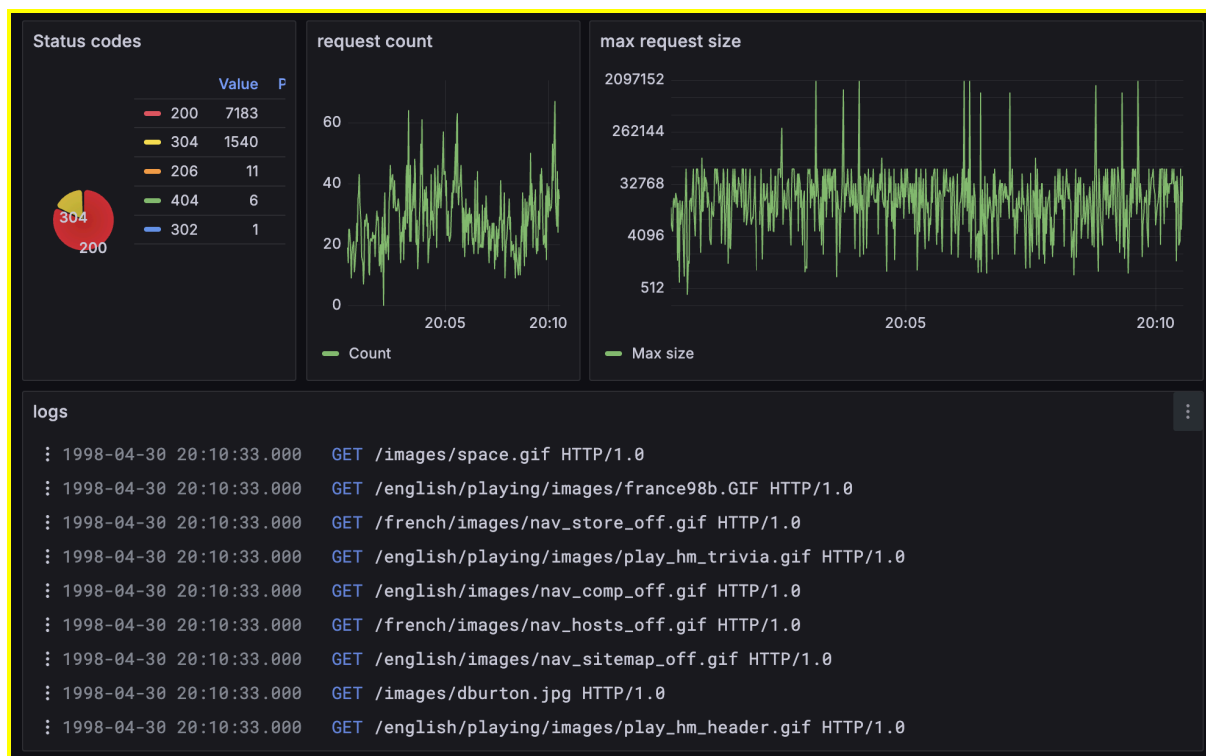
Некоторых этих вещей можно было бы избежать, если бы у вас было заложено **observability (наблюдаемость за системой)**, чтобы быть готовым локализовать источник проблемы и найти пути увеличения быстродействия системы,

За чем мы могли бы наблюдать, чтобы заранее обнаружить проблемы, вот некоторые из ключевых параметров:

- **Latency** (время ответа). Можно наблюдать как за отдельными api, так и за средним временем ответа
- **Throughput (RPS/QPS)** — количество запросов в секунду. Здесь мы могли бы своевременно увидеть повышающуюся нагрузку и предпринять необходимые меры
- **Error Rate** — процент ошибок. Может у вас стало что-то постоянно падать, а вы об этом не знали. Такая метрика поможет это заметить
- **Resource Utilization** — загрузка CPU, памяти, диска, сети.
- **Saturation** — насколько близки ресурсы к пределу (очереди, соединения).

Для этого используются специализированные инструменты, например: Prometheus + Grafana, OpenTelemetry, ELK/EFK стеки, Jaeger для трассировки. Подробно про них здесь мы не будем говорить, но в комплексных системах они всегда присутствуют.

Рисунок с скриншотом Grafana, где видно состояние системы:



Масштабируемость — это не про “добавим пару серверов и всё станет хорошо”.

Это про то, **как заранее построить систему, которая выдержит рост** — пользователей, данных и трафика — без падений и лишних тормозов.

Бэкенд-разработчику важно понимать, где могут быть узкие места, какие паттерны можно применять, и как следить за состоянием системы в реальном времени.

Это и есть настоящее искусство — предвидеть проблемы до того, как они случатся.

### Часть 3. Балансировка нагрузки

Когда мы говорим о масштабируемых системах, первым делом встаёт вопрос: *как распределить миллионы запросов так, чтобы система не сломалась и оставалась предсказуемой?* Именно здесь нам потребуется **балансировка нагрузки**.

Backend-разработчики часто сталкиваются с этим понятием на собеседованиях, но в реальности роль балансировщиков куда шире, чем просто “раскидывать запросы по серверам”. Балансировка — это способ сделать из множества машин **одну устойчивую систему**, которая может пережить всплески трафика и сбои.

Давайте попробуем понять зачем вообще нужна балансировка

На самом деле балансировка нагрузки выполняет несколько ключевых функций:

- **Распределяет запросы** между серверами, чтобы ни один не был перегружен.
- **Обеспечивает отказоустойчивость**: т.е. если один сервер упал — трафик просто уходит на другие.
- **Помогает с плавными деплоями** — можно направлять часть трафика на новую версию системы (это еще называют Канареечным развёртыванием (canary deployment))
- **Оптимизирует использование ресурсов**: здесь все просто — мощные машины получают больше нагрузки, слабые — меньше.
- **и делает архитектуру гибкой**: можно добавлять и убирать сервера “на лету”, без остановки системы.

**Можете представить картину, когда в интернет-магазине на “Чёрную пятницу”** трафик вырастает в десять раз. Один сервер бы просто лёг. А балансировщик равномерно раздаёт запросы по пулу серверов — и пользователи даже не замечают, что под капотом идёт настоящая борьба за выживание.

Еще что было бы полезно знать, так это то, что современные балансировщики работают на разных уровнях стека:

А нас интересуют уровни балансировки: L4 и L7

- **L4 (Transport Layer)** — решения принимаются на основе IP-адресов и портов. Это быстрый и лёгкий способ, но он **не знает** что находится внутри запроса. Такой вариант может хорошо подойти для TCP/UDP сервисов.
- **и L7 (Application Layer)** — анализируются данные прикладного уровня (HTTP-заголовки, пути URL, cookies). Это позволяет маршрутизировать запросы гибко: например, отправлять `/api/payments` на один сервис, а `/api/profile` — на другой.

## Балансировщик нагрузки

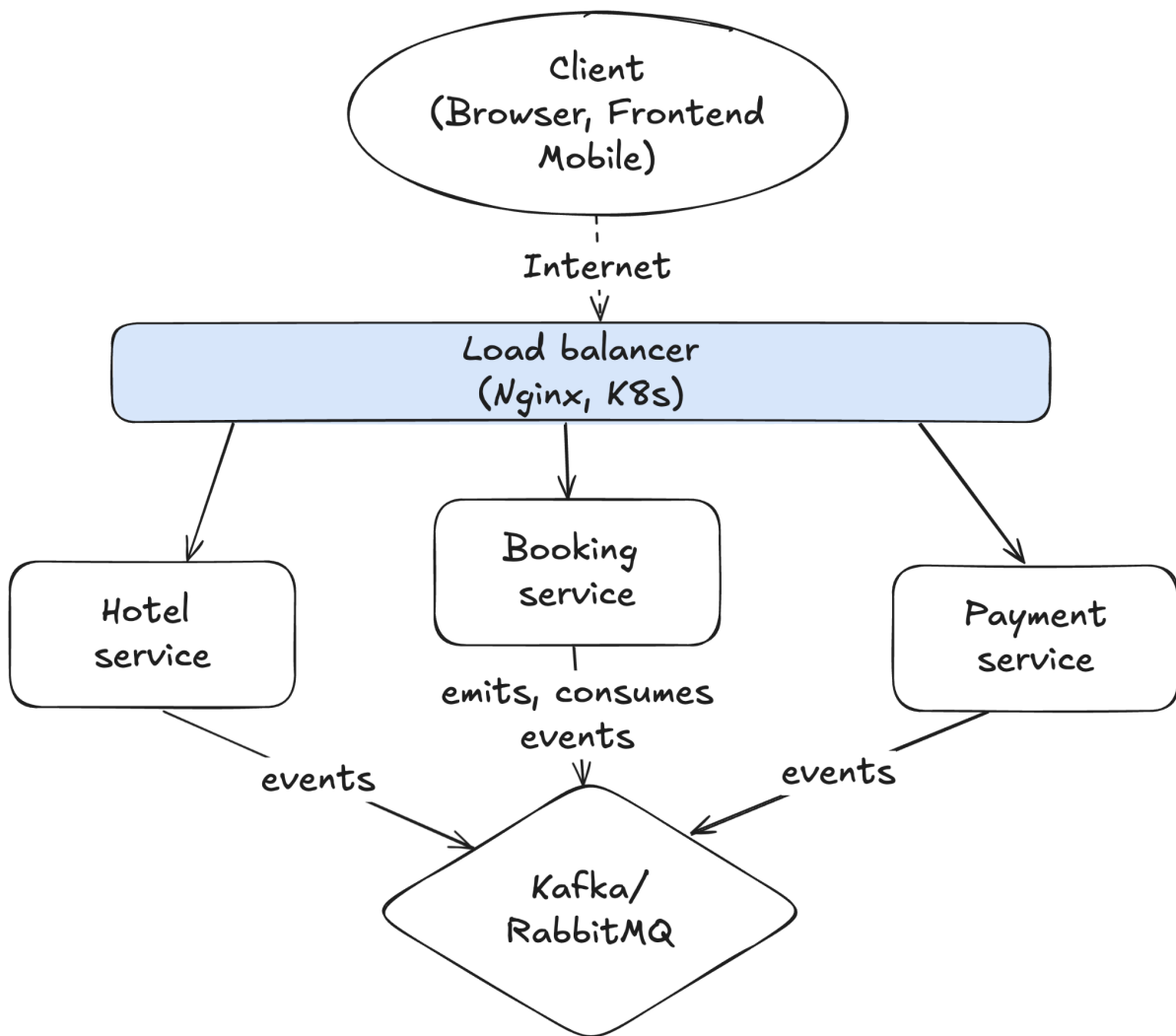


Схема: на картинке несколько серверов API, перед ними балансировщик, распределяющий запросы по разным маршрутам в зависимости от URL.

Что еще полезно понимать: Балансировщики бывают **внешние** и **внутренние**. Давайте рассмотрим оба варианта:

1. **Внешние балансировщики** — обрабатывают трафик от пользователей.
  - Принимают на себя удары DDoS.
  - Завершают SSL-сессии.
  - Часто интегрируются с CDN, чтобы вынести часть нагрузки на периферию.

**Например:** “Яндекс.Музыка” использует CDN и внешние балансировщики — поэтому миллионы людей слушают музыку без лагов, даже в пиковые часы.

2. **Внутренние балансировщики** — распределяют трафик уже *внутри* микросервисной архитектуры. Работают с минимальной задержкой, поддерживают gRPC и REST.
3. **и есть еще балансировщики баз данных** — отдельная категория.
  - Они решают задачу разделения запросов на чтение и запись.
  - Учитывают наличие реплик и мастера.
  - Обеспечивают согласованность данных.

Помимо всего этого, backend-разработчику также важно понимать, что выбор алгоритма балансировки напрямую влияет на поведение системы.

Одними из самых популярных алгоритмов являются следующие:

- **Round Robin** — циклическая раздача запросов. Хорошо для stateless-приложений. По сути запросы ходят по кругу между серверами и таким образом распределяются
- **Weighted Round Robin** — учитывает "вес" сервера. Более мощный сервер получает больше запросов.
- **Least Connections** — запрос отправляется серверу с наименьшим количеством активных соединений. Подходит для API с непредсказуемой нагрузкой.
- **Weighted Least Connections** — более гибкий вариант: серверы получают нагрузку в зависимости от мощности и текущей загруженности.
- **Response Time-based** — выбор сервера по минимальной задержке.
- **Consistent Hashing** — полезен для кэширования и шардинга: один и тот же пользователь всегда попадает на один сервер.

*Например **Consistent Hashing** может быть полезен для распределения кешей изображений. Это позволяет пользователю при повторном открытии объявления загружать картинку с того же сервера, снижая задержку.*

Здесь может возникнуть вопрос: а как быть с сессиями?

Для этого тоже есть некоторые решения

- **Session Affinity (липкие сессии)** — пользователь закрепляется за одним сервером. Простое решение, но плохо масштабируется.
- **Cookie-based affinity** — сервер помечает пользователя через cookie.
- **IP-based affinity** — думаю из названия понятно – здесь мы пользователя просто закрепим по IP, но это может быть ненадежно, т.к. у пользователя может быть не всегда 1 IP
- **Внешнее хранилище (Redis, DB)** — хороший вариант. т.к. любой сервер может обработать запрос, потому что сессия хранится централизованно.
- **Stateless-сессии (JWT)** — когда информация о пользователе хранится прямо в токене.

*Много где сессии пользователей могут храниться в Redis-кластере. Это позволяет при падении любого сервера продолжать работу без разлогинивания.*

И так, у нас все работает, но не можем же мы круглосуточно вручную следить за тем, что сервера у нас нормально функционируют? Поэтому, балансировщик должен уметь проверять состояние серверов:

Здесь стоит упомянуть про Health-check и failover.

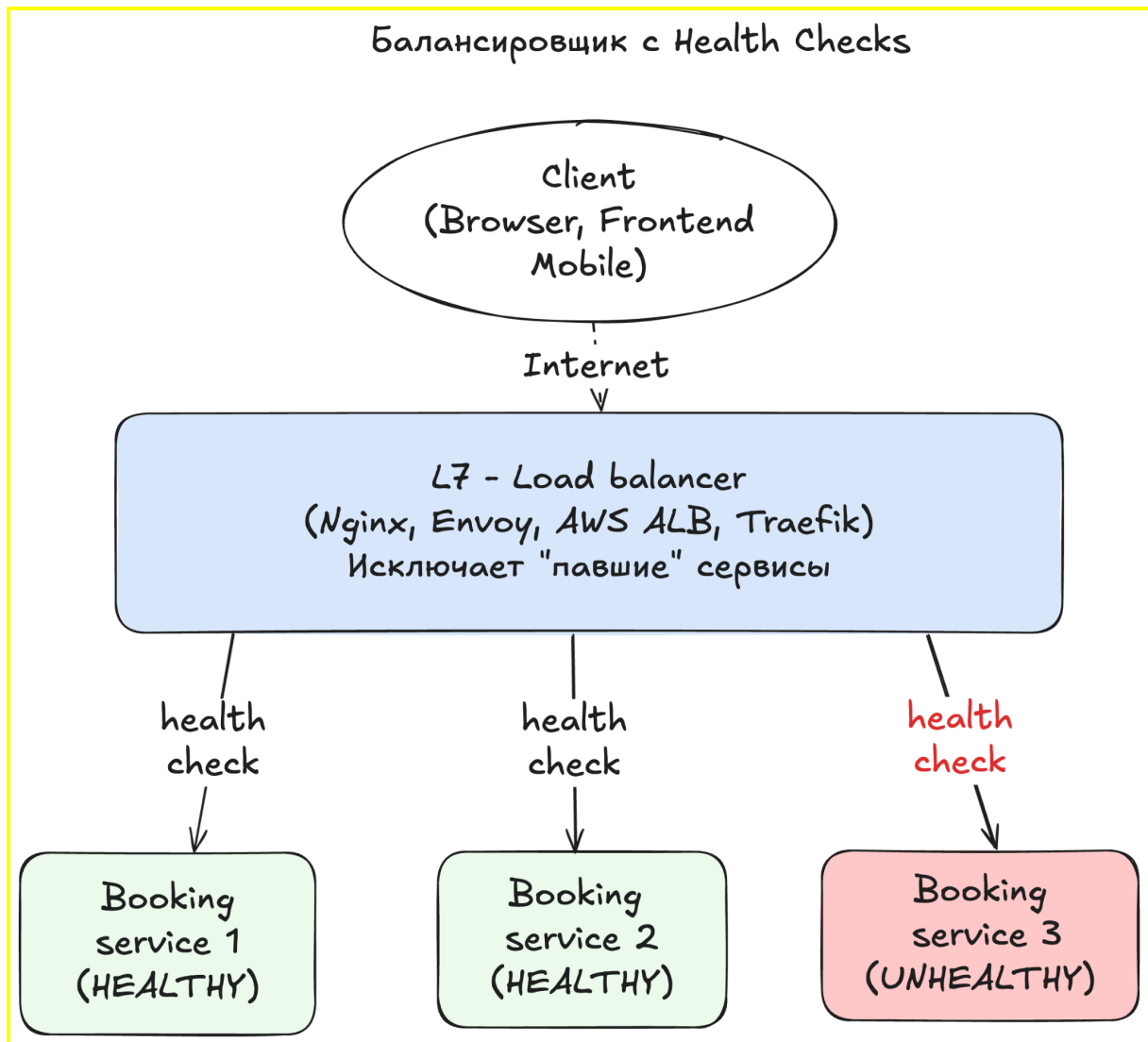
Health check может быть 2-х видов

- **Active checks** — сам отправляет тестовые запросы.
- **Passive checks** — анализирует реальные запросы от пользователей.

Если сервер не отвечает, срабатывает **failover**:

- сервер исключается из пула,
- нагрузка плавно перераспределяется,
- включается circuit breaker, чтобы не отправлять запросы на упавший сервис.

Схема: балансировщик регулярно пингует серверы, отключает те, что не отвечают, и перенаправляет трафик на здоровые.



В микросервисных системах всё ещё интереснее. Могут применяться следующие подходы:

- **Service Discovery** — это когда сервисы сами регистрируются и сами же снимаются с регистрации (Consul, etcd, Kubernetes).
- **Health-aware discovery** — маршрутизация только на сервисы, которые гарантированно работают.
- **Traffic Splitting / Canary Deployments** — при деплое новой версии на эту версию идет сначала идёт 1% трафика, затем 10%, потом 100%.
- **Feature Flags** — это когда мы часть функционала можем включать и выключать. Т.е. часть пользователей увидит новую функцию, часть — старую.

*В крупных приложениях часто используется именно canary deployment — сначала трафик идет на малую аудиторию, если все ок — выкатывают дальше.*

Нам, конечно, мало всего этого, и мы еще должны учесть, что если система работает в нескольких регионах, важна еще **географическая маршрутизация**:

**Географическая балансировка может быть следующей:**

- **Latency-based routing** — это когда пользователь идёт в ближайший дата-центр.
- **Geo-routing** — маршрутизация по геолокации (IP).
- **и Disaster Recovery** — это аварийное переключение трафика на другой регион.

И, конечно, классика — базы данных, где всё держится на балансировке подключений:

- **Connection Pooling** — управляет количеством открытых соединений.
- **Read/Write Split** — чтение на реплики, запись на мастер.
- **Failover** — если мастер падает, одна из реплик становится новым мастером.

Что в итоге. Балансировка нагрузки — это не просто "распределить запросы". Для backend-разработчика она означает:

1. **Выбор алгоритма** под конкретный сценарий.
2. **Выбрать правильную архитектуру приложений** (stateless, централизованные сессии).
3. **Построение грамотной работы с базой данных.**
4. **Про мониторинг и диагностику** для предотвращения сбоев.
5. **и Интеграцию с микросервисами и DevOps-практиками** (canary, feature flags).

## Часть 4. Хранение данных

Итак, у нас осталась заключительная глава по системному дизайну в бэкенд-разработке — это хранение данных

Хранение данных — это один из ключевых элементов системного дизайна. Независимо от того, разрабатываем ли мы социальную сеть, онлайн-магазин или аналитическую платформу, данные — это то, без чего чаще всего система не может существовать

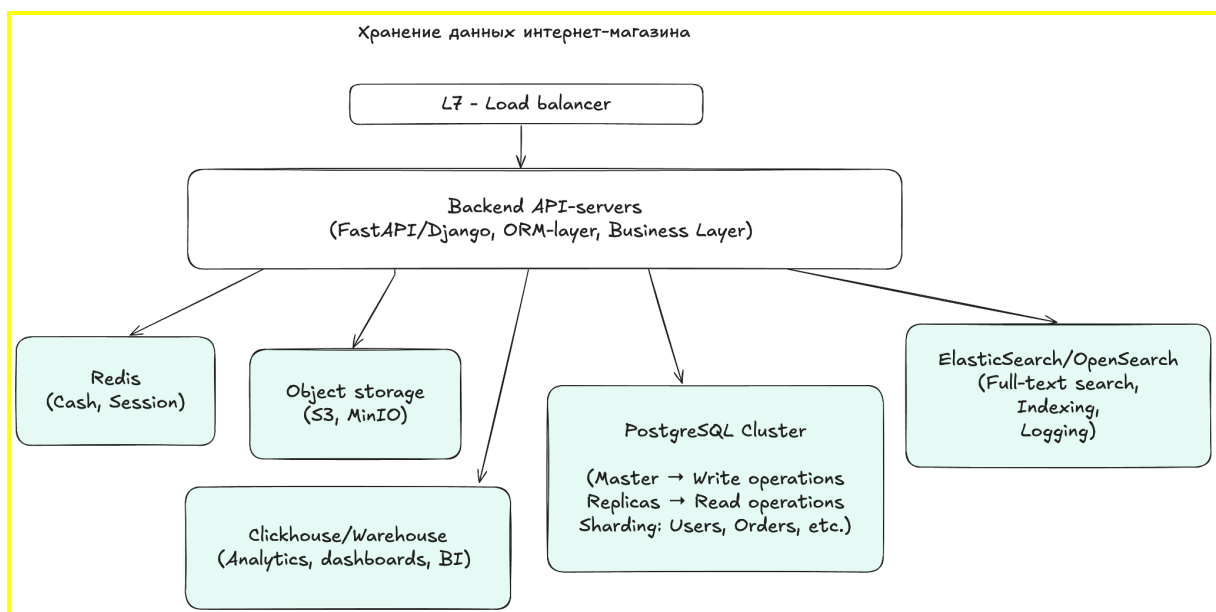
Часто можно услышать, что “главное — это выбрать СУБД, и все”.

**На самом деле этого мало.** Нужен комплексный подход и чёткое понимание, как именно данные к нам попадают, обрабатываются, обновляются и защищаются.

**В этой части разберём,** какие подходы и технологии позволяют хранить данные надёжно, эффективно и с возможностью масштабирования.

Вот пример схемы проекта системы хранения данных некоего интернет-магазина:

**Описание рисунка:** показан пример схемы проекта системы хранения данных интернет-магазина(стилизовать под единый дизайн):



Когда мы проектируем хранилище, важно понимать, какие характеристики для нас критичны. Обязательно нужно обратить внимание на следующее:

- **Надёжность (Reliability):** т.е. наши данные не должны теряться. Даже если сервер упал, информация должна восстановиться.
- **Доступность (Availability):** данные должны быть доступны пользователю в любой момент.
- **Масштабируемость (Scalability):** это когда система выдерживает рост количества пользователей и объёма данных.
- **Производительность (Performance):** — операции чтения и записи должны выполняться быстро.
- **и согласованность (Consistency):** пользователи должны получать актуальные данные (но уровень согласованности может варьироваться).

По сути проектирование систем хранения данных обычно сводится к обеспечению **оптимального** соотношения этих характеристик.

Давайте посмотрим, как этого добиться и в какой последовательности производится проектирование хранения данных.

### **Шаг 1. Нам нужно определить цели и сценарии использования данных**

Т.е. отвечаем на вопросы

- **Какие данные будут храниться?** (транзакции, логи, медиафайлы, аналитика).
- **Кто потребляет эти данные?** (пользователи, внутренние сервисы, аналитики).
- **Какие SLA (Service Level Agreement) нужны?** (99.9% доступности, согласованность, время отклика).

Здесь еще нужно помнить про CAP-теорему (Consistency, Availability, Partition Tolerance), подробно останавливаться не будем

### **Шаг 2. Необходимо классифицировать данные**

- **По структуре:** т.е. это структурированные таблицы, полуструктурированные (какой-нибудь JSON) или неструктурированные (видео, изображения).
- **По времени жизни:** горячие (те, которые активно используются), тёплые (нужны иногда) или холодные (архив).
- **и по критичности:** критичные данные (деньги, заказы), некритичные (логи, кэши).

### **Далее переходим к Шагу 3, где нам нужно выявить нефункциональные требования**

А именно:

- **Объём данных** (в GB/TB/петабайтах).
- **Скорость роста** (ежедневный прирост, пиковые нагрузки).
- **Согласованность vs доступность** (ориентируемся на CAP-теорему).
- **и Частоту доступа** (онлайн-запросы или офлайн-аналитика).

### **На 4 шаге выбираем модель хранения**

- **Это может быть реляционная БД (SQL):** где есть строгая схема и транзакции (PostgreSQL, MySQL, Oracle).
- **документоориентированная БД (NoSQL):** где бы получаем гибкую структуру (MongoDB, CouchDB).
- **База Ключ-значение** где у нас преимуществом будет быстрый доступ (как пример Redis, DynamoDB).

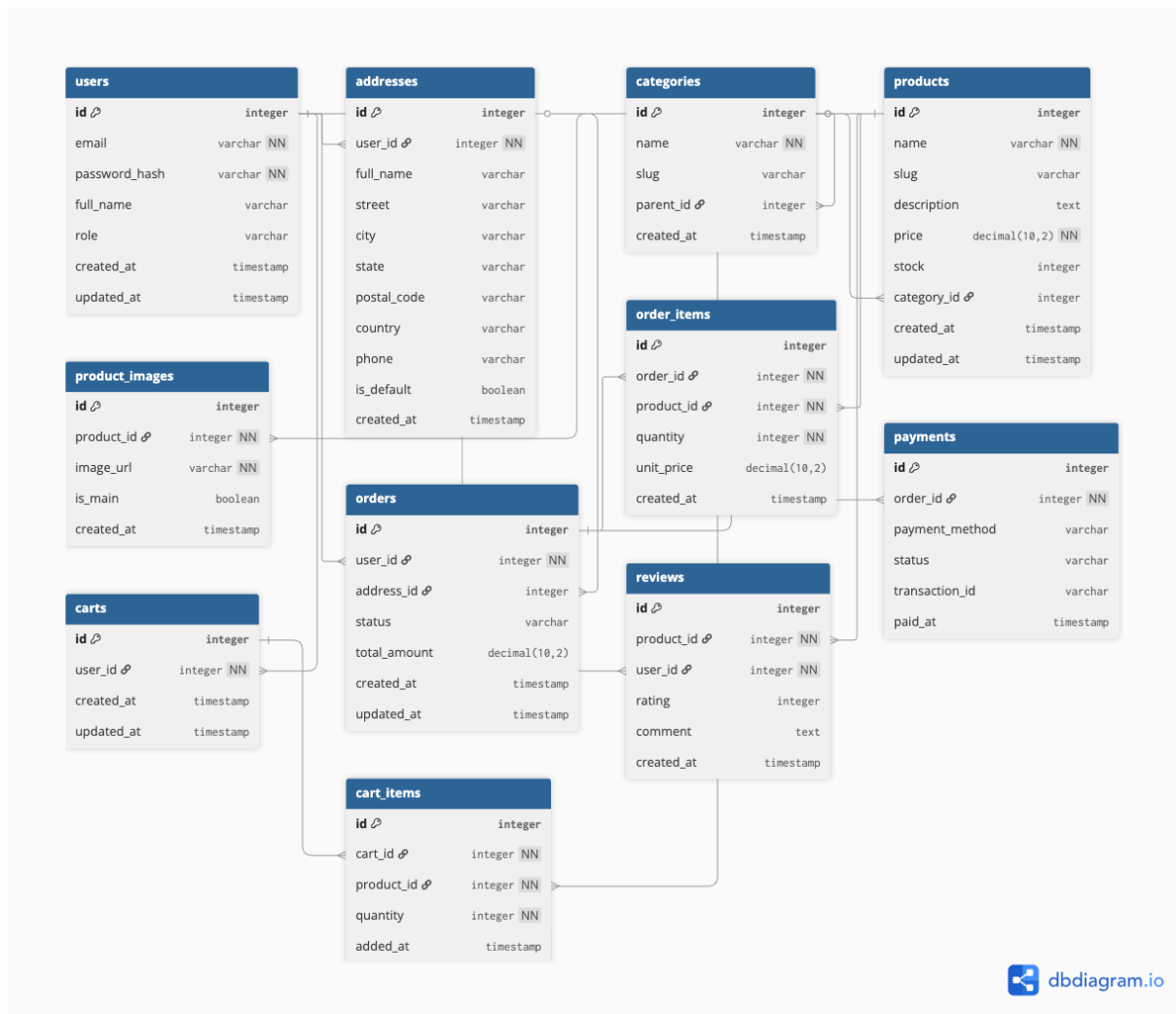
- **Колоночная**, которая чаще всего используется для аналитики (ClickHouse, Cassandra).
- **Графовая** для связей
- или мы выберем **объектное хранилище** для файлов и медиа (S3, MinIO).

И нужно понимать, что мы можем выбрать несколько баз под каждую задачу

**Дальше нам нужно Спроектировать логическую схему. Это уже 5 шаг**

- Здесь просто определяем сущности и связи (рисуем ER-диаграмму для SQL или JSON-структура для NoSQL).
- Определяем ключи (primary key, partition key).
- Планируем индексы.

Пример ER-диаграммы для интернет-магазина (если мелко, можно показать только фрагмент):



## **Дальше Шаг 6. Проектируем физическую схему**

- **Определяем шардинг, т.е.** как будем делить данные (по пользователям, регионам, диапазонам дат).
- **Думаем про репликацию:** master-slave, master-master, read-replicas.
- **и про Архивирование:** перенос холодных данных в отдельное хранилище.

## **Супер полезно будет Определить стратегии оптимизации. По сути это 7 шаг**

- **Еще раз думаем про индексы:** B-Tree, GIN, hash и другие
- **думаем про Кэширование:** Redis, Memcached.
- **учитываем Pre-aggregation:** заранее посчитанные данные (материализованные представления).
- **и партиционирование:** деление больших таблиц на части.

## **На 8 Шаге. Нужно обеспечить отказоустойчивость и подумать про бэкапы**

- Т.е. будут ли у нас регулярные бэкапы (full, incremental).
- что с Гео-репликация.
- и Тестирование восстановления системы после сбоев.

## **Далее Шаг 9. где нам нужно Настроить мониторинг и алерты**

- Это метрики:, где мы отражаем время отклика запросов, рост базы, нагрузка на реплики.
- Логи: медленные запросы, дэдлоки.
- и Алерты: падение реплики, нехватка места на диске.

Часто здесь используют Prometheus + Grafana, pg\_stat\_statements (PostgreSQL).

## **И завершающий шаг - это обязательно нужно Документировать и пересматривать дизайн по мере роста системы**

- Для этого желательно иметь документацию схем, политик, SLA.
- Проводить Регулярный аудит нагрузки.
- и Пересматривать стратегию при росте системы.

**Дизайн хранилища данных — это не просто “выбор базы”, а последовательный процесс:**

от анализа требований и SLA → к выбору модели хранения → проектированию схем, шардинга, репликации → до мониторинга и резервирования.

И еще несколько практических советов.

- Лучше всего начинать с одной базы (обычно PostgreSQL), затем добавлять NoSQL и кэш по мере роста нагрузки.
- Всегда проектируйте **резервное копирование и восстановление**. Этот совет вас убережет от кучи потраченных нервов
- Используйте **миграции** (Alembic, Django ORM migrations)
- и Следите за **метриками БД**: количество соединений, время запросов, размер индексов.

И еще раз – хранение данных – это баланс между **производительностью, согласованностью и масштабируемостью**. Backend-разработчику важно уметь выбирать правильное хранилище под конкретные задачи, комбинировать SQL и NoSQL, проектировать репликацию, шардинг и кэширование.

В этом и заключается искусство системного дизайна — хранить данные так, чтобы они всегда были доступны при этом хранились достаточно надёжно и их можно было получить максимально быстро.

Главное не пытайтесь понять все и сразу, постепенно возвращайся к полученной информации и применяй ее по мере необходимости. Надеюсь, ты обязательно справишься с этой нелегкой темой и спроектируешь такую систему, которая выдержит самые большие нагрузки!

## Глава 6. Системный дизайн во frontend-разработке

Спикер: Автушенко Андрей. (Черновой текст вышел на **50 минут** чтения)

Оранжевым выделил то, что нужно добавить на монтаже в виде плашек с текстом или картинок

### 6.0.0 О себе и о чем будет эта глава

Привет! Меня зовут **Автушенко Андрей**. Я занимаю должность **TechLead'a фронтенд-команды** в компании **PREMIER.ONE** и также являюсь **ментором сообщества Frontend Alliance**.

В этой главе я расскажу о том, **как системный дизайн применяется во фронтенде**, какие задачи он помогает решать и что важно знать о собеседованиях по SD для