**12. DDD/CQRS и Event Sourcing**

**DDD (Domain-Driven Design)**

**Определение**  
Domain-Driven Design — подход к разработке ПО, при котором основное внимание уделяется модели предметной области (domain). Цель — максимально точно отразить бизнес-логику в программном коде и поддерживать консистентность этой модели по мере развития продукта.

**Назначение**

1. Чёткое разделение бизнес-логики (предметной области) и инфраструктурных деталей.
2. Формирование «единого языка» (Ubiquitous Language), который используют как бизнес-аналитики, так и разработчики.
3. Разделение системы на контексты (Bounded Context), в каждом из которых модель имеет чёткие границы и является консистентной.

**Применение в микросервисной архитектуре**

* Позволяет разбивать большую систему на микросервисы по доменным контекстам (Bounded Context).
* Каждый контекст/микросервис реализует свою модель и защищает её инварианты, взаимодействуя с другими сервисами через договорённые API или события.

**CQRS (Command Query Responsibility Segregation)**

**Определение**  
Шаблон архитектуры, разделяющий операции чтения (Query) и записи (Command) в отдельные модели и слои.

* **Command-модель** (запись/изменение состояния) отвечает за валидацию команд и обновление состояния.
* **Query-модель** (чтение) оптимизирована для выборки и отображения данных.

**Назначение**

1. Уменьшить сложность модели за счёт разделения логики чтения и записи.
2. Повысить производительность чтения/записи (за счёт оптимизированных под задачу структур данных).
3. Облегчить масштабирование: можно масштабировать сервисы чтения и записи независимо.

**Применение в микросервисах**

* Часто используется совместно с Event Sourcing.
* Позволяет гибко дорабатывать как «поток команд», так и «проекции» данных для чтения.

**Event Sourcing**

**Определение**  
Подход, при котором система не хранит текущее «состояние» как таковое, а формирует его на основе журнала (лога) событий. Каждое событие описывает изменение состояния (например, «Счёт пополнен на 1000 руб»).

**Назначение**

1. Лёгкая реализация аудита: вся история изменений хранится в исходном виде.
2. Возможность «пересобрать» текущее состояние из всех событий заново (replay).
3. Естественная интеграция со встроенной event-driven моделью взаимодействия микросервисов.

**Применение**

* Широко используется при необходимости вести полную историю изменений и/или выстраивать масштабируемую архитектуру, ориентированную на события (event-driven).
* Часто сочетается с CQRS: команды порождают события, а проекции (read model) формируются на их основе.

**13. gRPC**

**Определение**

**gRPC** — это высокопроизводительный RPC-фреймворк с открытым исходным кодом, разработанный Google. Он позволяет приложениям вызывать методы на удалённых серверах так, словно это локальные объекты.

**Назначение**

1. Обеспечить высокую пропускную способность и малое время отклика в распределённых системах.
2. Упростить коммуникацию между микросервисами благодаря чётко описанным интерфейсам.

**Протокол**

* Основан на **HTTP/2**, что даёт преимущества: мультиплексирование, потоковое взаимодействие, компрессию заголовков.
* Поддерживает как уни-directional RPC (один запрос — один ответ), так и bi-directional streaming (двунаправленные потоки).

**Protobuf**

* **Protocol Buffers** (Protobuf) — формат сериализации данных, разработанный Google.
* gRPC использует Protobuf для описания структуры сообщений и автогенерации клиентских и серверных «заглушек» (stubs) на разных языках.
* Данные в Protobuf компактны, легко и быстро сериализуются/десериализуются.

**14. AWS (Amazon Web Services)**

**Определение**

**AWS** — крупнейшая платформа облачных сервисов от Amazon, предоставляющая инфраструктуру и программные сервисы «по требованию».

**Назначение**

1. Хостинг и запуск приложений в облачной среде.
2. Гибкое и масштабируемое управление вычислительными ресурсами, хранением данных, сетевыми настройками.
3. Платформа для DevOps и микросервисных решений.

**Основные сервисы**

1. **EC2 (Elastic Compute Cloud)**
   * Предоставляет виртуальные машины (инстансы).
   * Позволяет быстро поднимать и масштабировать серверы под нужные нагрузки.
2. **S3 (Simple Storage Service)**
   * Объектное хранилище для файлов любых типов.
   * Высокая надёжность (11 девяток), удобная интеграция с другими сервисами AWS.
3. **Lambda (AWS Lambda)**
   * Serverless-вычисления: запуск кода в ответ на события без управления серверами.
   * Автоматически масштабируется и тарифицируется только «за время выполнения» функции.
4. **API Gateway**
   * Управляемый сервис для создания и публикации API (REST, WebSocket).
   * Интегрируется с Lambda, EC2, другими сервисами.
   * Позволяет контролировать доступ, реализацию CORS, метрики.

**15. Kafka**

**Определение**

**Apache Kafka** — это распределённая платформа потоковой передачи данных (streaming platform), разработанная LinkedIn и переданная Apache Software Foundation.

**Назначение**

1. Обмен сообщениями (publish-subscribe) с высокой пропускной способностью и устойчивостью к сбоям.
2. Хранение потоковых данных (логирование, метрики, события) и обработка их в реальном времени.
3. Связующее звено между микросервисами: сервисы могут «издавать» (publish) события, а другие подписываться (subscribe).

**Основные компоненты**

1. **Broker** — узел кластера Kafka, хранящий и обрабатывающий сообщения.
2. **Topic** — тема (логический канал) для классификации сообщений.
3. **Partition** — раздел темы, обеспечивающий параллелизм и масштабирование.
4. **Producer** — отправитель (публикатор) сообщений.
5. **Consumer** — получатель (подписчик) сообщений.

**Принципы работы**

* Сообщения (events) записываются в Topic последовательно и хранятся в разделах (partitions).
* Подписчики (consumers) читают сообщения в своём темпе и могут объединяться в Consumer Group для балансировки нагрузки.
* Высокая производительность достигается за счёт линейной масштабируемости и записи «на диск» с использованием механизма commit log.

**16. OSGi**

**Определение**

**OSGi (Open Services Gateway initiative)** — это спецификация динамической модульной системы для Java-приложений. Позволяет разбивать приложение на набор «пакетов» (бандлов), которые могут загружаться, выгружаться и обновляться во время работы без перезапуска всей системы.

**Назначение**

1. Упрощённая модульность: чёткое разделение кода на независимые, легко меняемые модули (bundles).
2. Гибкое управление зависимостями между модулями.
3. Возможность «горячей» замены/обновления частей приложения.

**Основные компоненты**

1. **Bundle** — модуль, упакованный в формате JAR со специальными OSGi-манифестами.
2. **OSGi Framework** — среда выполнения, управляющая жизненным циклом бандлов (установка, запуск, остановка, обновление).
3. **Service Registry** — реестр, через который бандлы могут публиковать/искать сервисы.
4. **Module Layer** — механизм видимости пакетов и защиты классов.

**Принципы работы**

* Каждый бандл имеет собственный ClassLoader, что предотвращает конфликты версий библиотек.
* Зависимости между бандлами управляются через декларативные manifest-файлы.
* Бандлы могут динамически регистрировать и находить сервисы в реестре OSGi.

**17. Паттерн Saga**

**Определение**

**Saga** — это паттерн управления распределённой транзакцией в микросервисной архитектуре, при котором бизнес-операция разбивается на ряд локальных транзакций. Каждая транзакция публикует событие после выполнения, а следующая транзакция запускается при получении события.

**Назначение**

1. Обеспечить согласованность данных (consistency) в распределённой среде без использования классических двухфазных (XA) транзакций.
2. Уменьшить взаимные блокировки и повысить отказоустойчивость.
3. Определить «компенсирующие действия» (compensation) на случай неудачи одного из шагов.

**Применение**

* В микросервисах, когда нужно обеспечить консистентность сложной операции, затрагивающей несколько сервисов.
* Реализуется двумя способами:
  + **Оркестрация (Orchestration)** — выделенный «оркестратор» управляет шагами саги.
  + **Хореография (Choreography)** — сами сервисы реагируют на события друг друга.

**18. Паттерн Circuit Breaker**

**Определение**

**Circuit Breaker** — шаблон (паттерн) отказоустойчивости, который «обрывает» (breaks) запросы к сервису, если он становится недоступен или слишком медленно отвечает.

**Назначение**

1. Предотвратить лавинообразные сбои и перегрузки в распределённой системе.
2. Быстро выявлять неработающие сервисы и временно прекращать к ним запросы (или выдавать fallback-ответ).
3. Позволять «проверочные» (test) запросы, чтобы определить, восстановился ли сервис.

**Применение**

* Используется в микросервисах для защиты от цепочки отказов (cascade failure).
* Может быть реализован в коде (например, библиотека Hystrix, Resilience4j) или на уровне сервисной mesh (Istio).
* Переходит в состояния: **Closed** (проходят все запросы), **Open** (запросы блокируются) и **Half-Open** (тестовые запросы).

**19. Паттерн Log Aggregation**

**Определение**

**Log Aggregation** — паттерн сбора логов и метрик от множества микросервисов (или узлов), с целью их централизованного хранения, анализа и визуализации.

**Назначение**

1. Упростить диагностику и мониторинг распределённой системы.
2. Позволить находить коррелирующие события в логах разных сервисов.
3. Ускорить реакцию на инциденты и упростить аудит.

**Применение**

* Реализуется с помощью систем типа **ELK Stack** (Elasticsearch, Logstash, Kibana) или **Splunk**, **Graylog** и т.п.
* Сервисы публикуют логи в централизованное хранилище.
* Позволяет строить дашборды, искать по тексту логов, создавать алерты по ключевым словам/паттернам.

**20. Паттерн Health Check**

**Определение**

**Health Check** — паттерн, при котором каждый микросервис предоставляет простой способ проверить своё состояние (здоровье) через специальную «health check» точку (endpoint) или API.

**Назначение**

1. Позволить оркестраторам (Kubernetes, Docker Swarm) и балансировщикам трафика обнаруживать «живые» и «здоровые» инстансы.
2. Упростить автоматическое масштабирование и ротацию экземпляров при сбоях.
3. Дать системным администраторам и разработчикам средство оперативной диагностики.

**Применение**

* Реализуется как HTTP endpoint (например, /health), который возвращает статус UP/DOWN или подробности о сервисе.
* При интеграции с Kubernetes: livenessProbe, readinessProbe, startupProbe.
* Позволяет автоматически выводить из балансировщика «плохие» инстансы и перезапускать их.

**Итог**

* **DDD/CQRS и Event Sourcing** помогают правильно моделировать предметную область, отделять чтение от записи и хранить историю изменений в виде событий.
* **gRPC** с Protobuf — высокопроизводительный фреймворк RPC для микросервисных коммуникаций.
* **AWS** (EC2, S3, Lambda, API Gateway) — облачная платформа, предоставляющая широкий спектр сервисов для построения и развертывания микросервисов.
* **Kafka** — распределённая платформа для стриминга и обмена сообщениями.
* **OSGi** — спецификация модульности для Java-приложений, позволяющая динамически управлять компонентами.
* **Saga** — паттерн для координации распределённых транзакций в микросервисах.
* **Circuit Breaker** — защищает систему от каскадных отказов и перегрузок.
* **Log Aggregation** — централизованный сбор и анализ логов в распределённой системе.
* **Health Check** — даёт возможность быстро проверять состояние каждого сервиса и принимать решение о маршрутизации и перезапуске.

**Сервис** (в контексте IT) — это программная сущность (или набор компонентов), которая предоставляет определённый функционал через стандартизированный интерфейс и доступна по сети (обычно через Интернет). Клиенты, будь то другие программы или пользователи, обращаются к сервису, чтобы получить данные, выполнить операции или воспользоваться другим функционалом, не вдаваясь в технические детали его реализации.

**RPC (Remote Procedure Call)**

**Определение**

**RPC (Remote Procedure Call, удалённый вызов процедур)** – это парадигма или механизм, позволяющий одной программе (клиенту) вызывать функции (процедуры/методы), которые физически выполняются в другом процессе или на другом компьютере (сервере), так, как будто эти функции находятся локально в памяти клиента. При этом все сетевые операции по передаче запроса и получению ответа обычно скрыты от конечного разработчика.

**Протоколы и форматы передачи данных**

1. **JSON-RPC**
   * Использует формат JSON для кодирования запроса и ответа.
   * Лёгкий и популярный формат, легко читается и разбирается во многих языках программирования.
2. **XML-RPC**
   * Один из первых широко распространённых протоколов, использующих XML для сериализации данных.
   * Передача чаще всего идёт поверх HTTP.
   * В современных системах XML-RPC уступил место более гибким подходам (REST, gRPC) или более лёгким форматам (JSON-RPC).
3. **SOAP**
   * Хотя SOAP часто относят к web-сервисам, технически он тоже является реализацией RPC (XML-сообщения с чёткой структурой).
   * Использует WSDL (Web Services Description Language) для описания интерфейсов.
4. **gRPC**
   * Высокопроизводительный фреймворк удалённого вызова процедур, разработанный Google.
   * Основан на протоколе HTTP/2 и использует бинарный формат сериализации **Protocol Buffers** (Protobuf).
   * Позволяет организовывать uni-directional и bi-directional streaming.
5. **Proprietary/Custom RPC**
   * Некоторые системы могут использовать «собственный» (кастомный) формат или бинарные протоколы для RPC.
   * Например, Thrift (от Facebook), Avro (Apache) и т.д.

**Proxy/Stub-объекты**

В контексте RPC часто используются специальные объекты (proxy на клиенте и stub на сервере), которые скрывают детали сетевого взаимодействия:

* **Client Stub (Proxy)**
  + «Поддельный» объект на стороне клиента, который выглядит как обычная локальная функция или класс.
  + При вызове метода этот stub прозрачно формирует и отправляет сетевой запрос на сервер, а затем возвращает ответ обратно вызывающему коду.
* **Server Stub (Skeleton)**
  + Код на стороне сервера, который получает входящий запрос, десериализует данные и вызывает соответствующую «реальную» процедуру (бизнес-логику).
  + После выполнения операции формирует ответ, сериализует его и отправляет клиенту.

Использование stub и skeleton (или proxy и skeleton) позволяет «спрятать» низкоуровневые детали коммуникации (сетевые протоколы, форматы кодирования) и сделать удалённый вызов процедур похожим на локальный вызов функции с точки зрения разработчика.