Справочник паттернов микросервисов из книги Криса Ричардсона Microservices Patterns: With Examples in Java (ДОПОЛНЕНИЕ, НО НЕ ЗАМЕНА ЧТЕНИЯ).

Шаблон проектирования (Design Patterns) — это многоразовое решение проблемы, возникающей в определенном контексте.

Это идея, которая возникает как часть реальной архитектуры и затем показывает себя с лучшей стороны при проектировании программного обеспечения.

Interprocess communication, IPC - межпроцессное взаимодействие.

Фреймворк для написания многоязычных клиентов и серверов **gRPC,** представляет собой двоичный протокол на основе сообщений.

Закон Мелвина Конвея гласит: "Организация, проектирующая системы, обречена воспроизводить архитектуру, иммитирующую структуру собственных коммуникаций" (стр. 60)

стр. книги	Паттерн	Группа	Описание	ссылка
71	**Monolithic Architecture** Монолитная архитектура	**Application architecture patterns** Архитектура приложения	**Структурирует приложение в виде единого/исполяемого развёртываемого компонента.** **Решение:** Создайте приложение с монолитной архитектурой. Например: - один файл Java WAR. - единая иерархия каталогов кода Rails или NodeJS	https://microservices.io/patterns/monolithic.html
72	**Microservice Architecture** Микросервисная архитектура	**Application architecture patterns** Архитектура приложения	**Пролема:** Какова архитектура развертывания приложения? **Решение:** Шаблон **структурирует приложение в виде набора слабо связанных сервисов,** которые развёртываются независимо друг от друга.	https://microservices.io/patterns/microservices.html
	Decompose by business capability Декомпозиция по бизнес-возможностям	**Decomposition** Декомпозиция	**Проблемы:** Как разложить приложение на сервисы? **Решение:** Определить услуги, соответствующие возможностям бизнеса. Бизнес-возможности — это концепция моделирования бизнес-архитектуры. Это то, что бизнес делает для создания ценности. Бизнес-возможность часто соответствует бизнес-объекту, например. Управление заказами отвечает за заказы Менеджер по работе с клиентами несет ответственность за клиентов Бизнес-возможности часто организованы в многоуровневую иерархию. Например, корпоративное приложение может иметь категории верхнего уровня, такие как «Разработка продукта/услуги», «Предоставление продукта/услуги», «Формирование спроса» и т. д.	https://microservices.io/patterns/decomposition/decompose-by-business-capability.html
86	**Decompose by subdomain** Декомпозиция по поддомену	**Decomposition** Декомпозиция	**Определяет сервисы в соответствии с поддоменами в DDD.** **Пролемы:** Как разложить приложение на сервисы? **Решение:** Определите сервисы, соответствующие поддоменам **Domain-Driven Design (DDD).** DDD относится к проблемному пространству приложения — бизнесу — как к домену. Домен состоит из нескольких поддоменов. Каждый субдомен соответствует отдельной части бизнеса. Поддомены можно классифицировать следующим образом: Ядро — ключевое отличие для бизнеса и самая ценная часть приложения. Поддержка - связана с тем, чем занимается бизнес, но не является отличительной чертой. Они могут быть реализованы внутри компании или переданы на аутсорсинг. Общие - не специфичные для бизнеса и идеально реализуются с использованием готового программного обеспечения.	https://microservices.io/patterns/decomposition/decompose-by-subdomain.html
	Self-contained service Автономный сервис	**Decomposition** Декомпозиция	**Проблема:** Как сервис должен взаимодействовать с другими сервисами при обработке синхронного запроса? **Решение:** Спроектируйте сервис так, чтобы он мог отвечать на синхронный запрос, не дожидаясь ответа от какого-либо другого сервиса.	https://microservices.io/patterns/decomposition/self-contained-service.html
	Service per team Сервис на команду	**Decomposition** Декомпозиция	**Проблема:** Какая связь между командами и сервисами? **Решение:** Каждый сервис принадлежит команде, которая несет единоличную ответственность за внесение изменений. В идеале у каждой команды есть только один сервис.	https://microservices.io/patterns/decomposition/service-per-team.html
499	**Strangler application** Душитель	**Refactoring to microservices** Переход на микросервисы	**Выполняет модернизацию постепенно возводя новое (удушающее) приложение вокруг старого кода.** **Проблема:** Как перенести устаревшее монолитное приложение на микросервисную архитектуру? **Решение:** Модернизируйте приложение, постепенно разрабатывая новое (удушающее) приложение на основе устаревшего приложения. В этом сценарии приложение-душитель имеет микросервисную архитектуру.	https://microservices.io/patterns/refactoring/strangler-application.html
517	**Anti-corruption layer** Предохранительный слой	**Refactoring to microservices** Переход на микросервисы	**Программный слой, выступающий посредником между двумя доменными моделями, не давая им засорить друг друга своими концепциями.** **Пролема:** Как не допустить, чтобы модель предметной области устаревшего монолита загрязняла модель предметной области нового сервиса. **Решение:** Определите уровень защиты от коррупции, который преобразуется между двумя моделями предметной области.	https://microservices.io/patterns/refactoring/anti-corruption-layer.html
46, 56, 201, 273, 278	**CQRS / command query responsibility segregation /** разделение ответственности командных запросов	**Data management** Управление данными	**Реализует запрос, которому нужны данные из нескольких сервисов. Для поддержания представления, реплицирующего данные из разных источников и доступного только для чтения, используются события.** **Проблема**: Как реализовать запрос, извлекающий данные из нескольких сервисов в микросервисной архитектуре? **Решение** Определите базу данных представления, которая является доступной только для чтения репликой, предназначенной для поддержки этого запроса. Приложение поддерживает реплику в актуальном состоянии, подписываясь на события домена, публикуемые сервисом, которому принадлежат данные.	https://microservices.io/patterns/data/cqrs.html
46, 150, 151, 185	**SAGA** **Повествование**	**Data management** Управление данными	**Обеспечивает согласованность данных между сервисами, используя последовательность локальных транзакций, которые координируются с помощью асинхронных сообщений.** **Проблема:** Как реализовать транзакции, охватывающие сервисы? **Решение:** Реализуйте каждую бизнес-транзакцию, охватывающую несколько сервисов, в виде саги. **Сага — это последовательность локальных транзакций.** Каждая локальная транзакция обновляет базу данных и публикует сообщение или событие, чтобы инициировать следующую локальную транзакцию в саге. Если локальная транзакция терпит неудачу из-за нарушения бизнес-правила, сага выполняет серию компенсирующих транзакций, которые отменяют изменения, внесенные предыдущими локальными транзакциями. Обеспечить согласованность сервисов.	https://microservices.io/patterns/data/saga.html
39	**Database per service** Одна БД на один сервис	**Data management** Управление данными	**Пролема:** Какова архитектура базы данных в приложении микросервисов? **Решение:** Держите постоянные данные каждого микросервиса закрытыми для этого сервиса и доступными только через его API. Транзакции сервиса включают только его базу данных. Таким образом обеспечивается слабая связанность микросервисов.	https://microservices.io/patterns/data/database-per-service.html
268	**API Composition** Оъединение API	**Data management** Управление данными	**Реализует запрос, извлекающий данные из разных сервисов, оращаясь к ним через их API и объединяя результаты.** **Прблема:** Как реализовать запросы в микросервисной архитектуре? **Решение:** Реализуйте запрос, определив API Композитор, который вызывает сервисы, владеющие данными, и выполняет объединение результатов в памяти.	https://microservices.io/patterns/data/api-composition.html
	Shared database	**Data management** Управление данными	**Проблема:** Какова архитектура базы данных в приложении микросервисов? **Решение:** Используйте (единственную) базу данных, совместно используемую несколькими сервисами. Каждый сервис имеет свободный доступ к данным, принадлежащим другим сервисам, с помощью локальных транзакций ACID.	https://microservices.io/patterns/data/shared-database.html

200	**Domain event** Доменное событие	**Data management** Управление данными	**Агрегат публикует доменное событие во время своего создания или в ходе какого-то другого существенного изменения.** В конечном счете доменные события доходят до брокера (например, Apache Kafka) в виде сообщений. **Проблема:** Как сервис публикует событие при обновлении своих данных? **Решение:** Организуйте бизнес-логику сервиса в виде набора агрегатов DDD, которые генерируют события предметной области при их создании или обновлении. Сервис публикует эти события предметной области, чтобы их могли использовать другие сервисы.	https://microservices.io/patterns/data/domain-event.html
224, 264	**Event sourcing** Порождение событий	**Data management** Управление данными	**Сохраняет агрегат в виде последовательности доменных событий, которые представляют изменения состояния.** **Проблема:** Как надежно/атомарно обновлять базу данных и отправлять сообщения/события? **Решение:** Хорошим решением этой проблемы является использование источников событий. Источник событий сохраняет состояние бизнес-объекта, такого как Заказ или Клиент, в виде последовательности событий, изменяющих состояние. При каждом изменении состояния бизнес-объекта к списку событий добавляется новое событие. Поскольку сохранение события — это одна операция, она по своей сути является атомарной. Приложение реконструирует текущее состояние объекта, воспроизводя события. Приложения сохраняют события в хранилище событий, которое является базой данных событий. В магазине есть API для добавления и получения событий сущности. Хранилище событий также ведет себя как брокер сообщений. Он предоставляет API, который позволяет сервисам подписываться на события. Когда сервис сохраняет событие в хранилище событий, оно доставляется всем заинтересованным подписчикам. Некоторые объекты, такие как клиент, могут иметь большое количество событий. Чтобы оптимизировать загрузку, приложение может периодически сохранять моментальный снимок текущего состояния объекта. Чтобы восстановить текущее состояние, приложение находит самый последний моментальный снимок и события, которые произошли после этого снимка. В результате меньше событий для воспроизведения.	https://microservices.io/patterns/data/event-sourcing.html
134, 146	**Transactional outbox** Публикация событий	**Transactional messaging** Транзакционный обмен сообщениями	**Публикует событие или сообщение в рамках транзакции БД, сохраняя его в таблицу OUTBOX.** **Проблема:** Как надежно/атомарно обновлять базу данных и отправлять сообщения/события? **Решение**: Сервис, использующий реляционную базу данных, вставляет сообщения/события в таблицу исходящих сообщений (например, MESSAGE) как часть локальной транзакции. Сервис, использующий базу данных NoSQL, добавляет сообщения/события к атрибуту обновляемой записи (например, документа или элемента). Отдельный процесс ретрансляции сообщений публикует события, вставленные в базу данных, в брокер сообщений.	https://microservices.io/patterns/data/transactional-outbox.html
136, 146	**Transaction log tailing** Отслеживание журнала транзакций	**Transactional messaging** Транзакционный обмен сообщениями	**Публикует изменения, внесённые в БД, отслеживая журнал транзакций.** **Проблема:** Как публиковать сообщения/события в папке «Исходящие» в базе данных брокеру сообщений? **Решение:** Следите за журналом транзакций базы данных и публикуйте каждое сообщение/событие, вставленное в папку «Исходящие», брокеру сообщений.	https://microservices.io/patterns/data/transaction-log-tailing.html
135, 146	**Polling publisher** Опрашивающий издатель	**Transactional messaging** Транзакционный обмен сообщениями	**Публикует сообщения, опрашивая таблицу OUTBOX в БД.** **Проблема:** Как публиковать сообщения/события в папке «Исходящие» в базе данных брокеру сообщений? **Решение:** Публикуйте сообщения, опрашивая таблицу исходящих сообщений базы данных.	https://microservices.io/patterns/data/polling-publisher.html
	Service Component Test	**Testing** Тестирование	**Проблема:** Как легко протестировать сервис? **Решения:** Набор тестов, который тестирует службу изолированно, используя тестовые двойники для любых служб, которые он вызывает.	https://microservices.io/patterns/testing/service-component-test.html
355	**Service Integration Contract Test** Тестирование контрактов с расчётом на потребителя	**Testing** Тестирование	**Проверяет, соответствует ли сервис ожиданиям своих потребителей.** **Проблема:** Как легко проверить, что сервис предоставляет API, который ожидают его клиенты? **Решение:** Набор тестов для сервиса, написанного разработчиками другого сервиса, который его использует. Набор тестов проверяет, соответствует ли сервис ожиданиям сервиса-потребителя.	https://microservices.io/patterns/testing/service-integration-contract-test.html
357	**Consumer-side contract test** Тестирование контрактов на стороне потребителя	**Testing** Тестирование	**Проверяет, может ли клиент взаимодействовать с сервисом.**	https://microservices.io/patterns/testing/consumer-side-contract-test.html
	Multiple service instances per host	**Deployment patterns** Развёртывание	**Пролема:** Как услуги упаковываются и развертываются? **Решение:** Запустите несколько экземпляров разных сервисов на хосте, сервере (физическом или виртуальном компьютере). Существуют различные способы развертывания экземпляра сервиса на общем хосте, в том числе: Разверните каждый экземпляр сервиса как процесс JVM. Например, экземпляры Tomcat или Jetty на экземпляр сервиса. Разверните несколько экземпляров сервиса на одной JVM. Например, в виде веб-приложений или пакетов OSGI.	https://microservices.io/patterns/deployment/multiple-services-per-host.html
51 гл 12	**Single Service Instance per Host** Один сервис — один сервер	**Deployment patterns** Развёртывание	**Проблема:** Как услуги упаковываются и развертываются? **Решение:** Развертывание каждого экземпляра микросервиса на отдельном сервере	https://microservices.io/patterns/deployment/single-service-per-host.html
455	**Service Instance per VM** Развёртывание сервиса в виде VM	**Deployment patterns** Развёртывание	**Развёртывает в промышленной среде сервисы, упакованные в виде образов для вирмуальной машины. Каждый экземпляр сервиса является отдельной ВМ.** **Пролема:** Как услуги упаковываются и развертываются? **Решение:** Упакуйте сервис как образ виртуальной машины и разверните каждый экземпляр сервиса как отдельную	https://microservices.io/patterns/deployment/service-per-vm.html
51, 459	**Service instance per container** Один сервис - один контейнер	**Deployment patterns** Развёртывание	**Развёртывает в среде выполнения сервис, упакованный в виде образа контейнера. Каждый экземпляр сервиса является отдельным контейнером.** **Проблема:** Как услуги упаковываются и развертываются? **Решение:** Упакуйте сервис как образ контейнера (Docker) и разверните каждый экземпляр сервиса как контейнер.	https://microservices.io/patterns/deployment/service-per-container.html
476	**Sidecar**	**Deployment patterns** Развёртывание	**Реализует общие функции в подключаемом процессе или контейнере, который находится рядом с экземпляром сервиса.**	
	Serverless deployment	**Deployment patterns** Развёртывание	**Проблема:** Как услуги упаковываются и развертываются? **Решение:** Используйте инфраструктуру развертывания, которая скрывает любую концепцию серверов (т. е. зарезервированных или предварительно выделенных ресурсов) — физических или виртуальных хостов или контейнеров. Инфраструктура берет код вашего сервиса и запускает его. Плата взимается за каждый запрос в зависимости от потребляемых ресурсов. Чтобы развернуть свой сервис с использованием этого подхода, вы упаковываете код (например, в виде ZIP-файла), загружаете его в инфраструктуру развертывания и описываете желаемые характеристики производительности. Инфраструктура развертывания — это утилита, управляемая поставщиком общедоступного облака. Обычно он использует либо контейнеры, либо виртуальные машины для изоляции сервисов. Однако эти подробности скрыты от вас. Ни вы, ни кто-либо еще в вашей организации не несет ответственности за управление любой низкоуровневой инфраструктурой, такой как операционные системы, виртуальные машины и т. д.	https://microservices.io/patterns/deployment/serverless-deployment.html

	Service deployment platform	**Deployment patterns** Развёртывание	**Проблема:** Как услуги упаковываются и развертываются? **Решение:** Используйте платформу развертывания, которая представляет собой автоматизированную инфраструктуру для развертывания приложений. Он предоставляет абстракцию сервиса, которая представляет собой именованный набор высокодоступных (например, с балансировкой нагрузки) экземпляров сервиса.	https://microservices.io/patterns/deployment/service-deployment-platform.html
441	**Microservice chassis** Шасси микросервисов	**Cross cutting concerns** Решение сквозных проблем	**Создание сервисов на основе фреймворков, реализующих такие общие функции, как отслеживание исключений, ведение журнала, проверка работоспособности, работа с внешней конфигурацией и распределённая трассировка.** **Проблема:** Как команда может быстро создать и настроить поддерживаемую кодовую базу для готовых к работе сервисов, чтобы они могли приступить к разработке их бизнес-логики? **Решение:** Создайте каркас шасси микросервисов, который может стать основой для разработки ваших микросервисов. Шасси - повторно используемая логика сборки, которая создает и тестирует сервисы. Сюда входят, например, плагины Gradle. Механизмы, которые решают сквозные проблемы. Шасси обычно собирает и настраивает набор фреймворков и библиотек, которые реализуют эту функциональность. Шаблон сервиса — это пример сервиса, в котором используется корпус микросервиса.	https://microservices.io/patterns/microservice-chassis.html
443	**Service mesh** Сеть сервисов	**Cross cutting concerns** Решение сквозных проблем	**Пропускает весь трафик между сервисами через сетевой слой, реализующий такие функции, как предохранители, распределённая трассировка, обнаружение сервисов, балансирование нарузки и маршрутизация трафика с учётом правил.**	
	Service Template	**Cross cutting concerns** Решение сквозных проблем	**Пролема:** Как команда может быстро создать и настроить поддерживаемую кодовую базу для готовых к работе сервисов, чтобы они могли приступить к разработке ее бизнес-логики? **Решение:** Создайте шаблон исходного кода, который разработчик может скопировать, чтобы быстро приступить к разработке нового сервиса. Шаблон — это простой работающий сервис, который реализует необходимую логику сборки и сквозные функции вместе с логикой примера приложения.	https://microservices.io/patterns/service-template.html
422	**Externalized configuration** Конфигурация, вынесенная вовне	**Cross cutting concerns** Решение сквозных проблем	**Предоставляет запущенному сервису значения конфигурационных свойств, такие как учётные данные для доступа к БД и сетевое размещение.** **Пролема:** Как включить сервис для работы в нескольких средах без изменений? **Решение:** Экстернализация всей конфигурации приложения, включая учетные данные базы данных и сетевое расположение. При запуске сервис считывает конфигурацию из внешнего источника, например. переменные среды ОС.	https://microservices.io/patterns/externalized-configuration.html
	Remote Procedure Invocation (RPI)	**Communication style** Стиль взаимодействия	**Проблема:** Как взаимодействуют сервисы в микросервисной архитектуре? **Решение:** Используйте RPI для межсервисного взаимодействия. Клиент использует протокол на основе запроса/ответа для выполнения запросов к сервису.	https://microservices.io/patterns/communication-style/rpi.html
106, 119	**Messaging** Удалённый вызов процедур.	**Communication style** Стиль взаимодействия	**Клиент обращается к сервису по синхронному протоколу на основе удалённого вызова процедур, такого как REST.** Вариант (стр. 119): **Клиент общается с сервисом посредством асинхронных сообщений.** **Проблема:** Как взаимодействуют службы в микросервисной архитектуре? **Решение:** Используйте асинхронный обмен сообщениями для взаимодействия между сервисами. Связь обеспечивается по каналам обмена сообщениями. Существует несколько различных стилей асинхронной связи: Запрос/ответ - сервис отправляет сообщение с запросом получателю и ожидает получить ответное сообщение в кратчайшие сроки. Уведомления - отправитель отправляет сообщение получателю, но не ожидает ответа. Ни один не отправлен. Запрос/асинхронный ответ — сервис отправляет сообщение запроса получателю и ожидает в конечном итоге получить ответное сообщение. Публикация/подписка — сервис публикует сообщение для нуля или более получателей. Публикация/асинхронный ответ — сервис публикует запрос одному или нескольким получателям, некоторые из которых возвращают ответ.	https://microservices.io/patterns/communication-style/messaging.html
190	**Domain-specific protocol** Доменная модель	**Communication style** Стиль взаимодействия	**Организует бизнес-логику в виде объектной модели, состоящей из классов с сотоянием и поведением.** **Контекст:** Вы применили шаблон архитектуры микросервиса. Сервисы должны обрабатывать запросы от клиентов приложения. Кроме того, сервисы иногда должны сотрудничать для обработки этих запросов. Они должны использовать протокол межпроцессного взаимодействия. **Решение:** Используйте специфичный для домена протокол для взаимодействия между сервисами.	https://microservices.io/patterns/communication-style/domain-specific.html
189	**Сценарий транзакции**	**Transactional messaging** Транзакционный обмен сообщениями	**Организует бизнес-логику в виде набора процедурных сценариев транзакций, по одному для каждого типа запросов.** **Проблема:** в некоторых ситуациях оъектно-ориентированных подход излишен — например, при разработке простой бизнес-логики. **Решение:** классы, реализующие поведение, отделены от классов, которые хранят состояние. При использовании этого шаблона сценарии обычно размещаются в классе сервис-слоя (например, OrderService). Класс сервиса имеет по одному методу для каждого запроса или системной операции. Метод реализует бизнес-логику для определенного запроса. Он обращается к БД с помощью объектов доступа к данным (data access object, DAO), таких как OrderDao.	
194	**Агрегат**	**Communication style** Стиль взаимодействия	**Агрегат — это кластер доменных объектов. Организует доменную модель в виде набора агрегатов - графов объектов, с которыми можно работать, как с единым целым.** Агрегат состоит из корневой сущности и иногда одной или нескольких сущностей и объектов значений. Многие бизнес-объекты моделируются в виде агрегатов. Агрегаты разбивают доменную модель на блоки, в которых легче разобраться по отдельности. **Агрегат часто загружается из базы данных целиком, что позволяет избежать любых проблем с ленивой загрузкой.** Правило 1. Ссылайтесь только на корень агрегата. Правило 2. Межагрегатные ссылки должны применять первичные ключи Правило 3. Одна транзакция создает или обновляет один агрегат	
	Idempotent Consumer	**Communication style** Стиль взаимодействия	**Пролема:** Как потребитель сообщений правильно обрабатывает повторяющиеся сообщения? **Решение:** Реализуйте идемпотентного потребителя, который является потребителем сообщений, который может правильно обрабатывать повторяющиеся сообщения. Некоторые потребители естественно идемпотентны. Другие должны отслеживать сообщения, которые они обработали, чтобы обнаруживать и отбрасывать дубликаты. Вы можете сделать потребителя идемпотентным, записав в базу данных идентификаторы сообщений, которые он успешно обработал. При обработке сообщения потребитель может обнаруживать и удалять дубликаты, запрашивая базу данных. Есть несколько разных мест для хранения идентификаторов сообщений. Один из вариантов заключается в том, чтобы потребитель использовал отдельную таблицу PROCESSED_MESSAGES. Другой вариант заключается в том, чтобы потребитель сохранял идентификаторы в бизнес-сущностях, которые он создает или обновляет.	https://microservices.io/patterns/communication-style/idempotent-consumer.html
308	**API Gateway / Backends for Frontends** API шлюз	**External API** Внешний API	**Реализует сервис, который служит точкой входа в микросервисное приложение для клиентов внешнего API.** **Проблема:** Как клиенты приложения на основе микросервисов получают доступ к отдельным сервисам? **Решение:** Внедрите шлюз API, который является единой точкой входа для всех клиентов. Шлюз API обрабатывает запросы одним из двух способов. Некоторые запросы просто проксируются/маршрутизируются на соответствующий сервис. Он обрабатывает другие запросы, разветвляясь на несколько сервисов.	https://microservices.io/patterns/apigateway.html
315	**Шаблон BFF**	**External API** Внешний API	**Реализует отдельный API-шлюз для каждого типа клиентов.**	

	API Gateway / Backends for Frontends API шлюз	**External API** Внешний API	**Проблема:** Как клиенты приложения на основе микросервисов получают доступ к отдельным сервисам? **Решение:** реализовать шлюз API, который является единой точкой входа для всех клиентов. Шлюз API обрабатывает запросы одним из двух способов. Некоторые запросы просто проксируются/маршрутизируются на соответствующий сервис. Он обрабатывает другие запросы, разветвляясь на несколько сервисов.	https://microservices.io/patterns/apigateway.html
117, 146	**Client-side service discovery** Обнаружение на клиентской стороне	**Service discovery** Обнаружение	**Пролема:** Как клиент сервиса — шлюз API или другой сервис — обнаруживают расположение экземпляра сервиса? **Решение:** Делая запрос к сервису, клиент получает местоположение экземпляра сервиса, запрашивая реестр, который знает расположение всех экземпляров сервиса.	https://microservices.io/patterns/client-side-discovery.html
117, 146	**Server-side service discovery** Обнаружение на стороне сервера	**Service discovery** Обнаружение	**Клиент извлекает из реестра список доступных экземпляров сервиса и выбирает один из них с учётом балансирования нагрузки. Клиент делает запрос к маршрутизатору, который отвечает за обнаружение сервисов.** **Проблема:** Как клиент сервиса — шлюз API или другой сервис — обнаруживает расположение экземпляра сервиса? **Решение:** Делая запрос к сервису, клиент делает запрос через маршрутизатор (также известный как балансировщик нагрузки), который работает в хорошо известном месте. Маршрутизатор запрашивает реестр сервисов, который может быть встроен в маршрутизатор, и перенаправляет запрос доступному экземпляру сервиса.	https://microservices.io/patterns/server-side-discovery.html
	Service registry	**Service discovery** Обнаружение	**Проблема:** Как клиенты сервиса (в случае обнаружения на стороне клиента) и/или маршрутизаторы (в случае обнаружения на стороне сервера) узнают о доступных экземплярах сервиса? **Решение**: Реализуйте реестр сервисов, который представляет собой базу данных сервисов, их экземпляров и местоположений. Экземпляры сервисов регистрируются в реестре при запуске и отменяются при завершении работы. Клиент сервиса и/или маршрутизаторы запрашивают реестр сервисов, чтобы найти доступные экземпляры сервисов. Реестр может вызывать API проверки работоспособности экземпляра сервиса, чтобы убедиться, что он может обрабатывать запросы.	https://microservices.io/patterns/service-registry.html
116, 146	**Self Registration** Саморегистрация	**Service discovery** Обнаружение	**Экземпляр сервиса регистрирует себя в реестре.** **Проблема:** Как экземпляры сервиса регистрируются и не регистрируются в реестре сервисов? **Решение:** Экземпляр сервиса отвечает за регистрацию в реестре сервисов. При запуске экземпляр сервиса регистрируется (хост и IP-адрес) в реестре и становится доступным для обнаружения. Обычно клиент должен периодически продлевать свою регистрацию, чтобы реестр знал, что он все еще жив. При завершении работы экземпляр сервиса отменяет регистрацию в реестре. Обычно это обрабатывается **платформой шасси сервисов.**	https://microservices.io/patterns/self-registration.html
119, 146	**3rd Party Registration** Сторонняя регистрация	**Service discovery** Обнаружение	**Экземпляры сервиса автоматически регистрируются в реестре сторонним компонентом.** **Проблема:** Как экземпляры сервиса регистрируются и не регистрируются в реестре сервиса? **Решение:** Делая запрос к сервису, клиент получает местоположение экземпляра сервиса, запрашивая реестр, который знает расположение всех экземпляров сервиса.	https://microservices.io/patterns/client-side-discovery.html
111, 112, 146	**Circuit Breaker** **Предохранитель**	**Reliability** Надёжность	**RPI-прокси, который в случае достижения определённого лимита последовательных отказов начинает отклонять все вызовы, пока не истечёт определённое время.** **Проблема:** Как предотвратить каскадирование отказа сети или сервиса на другие сервисы? **Решение:** Клиент сервиса должен вызывать удаленный сервис через прокси-сервер, который работает аналогично автомату защиты цепи (предохранителю). Когда количество последовательных сбоев превышает пороговое значение, срабатывает предохранитель (сеть размыкается), и в течение периода тайм-аута все попытки вызвать удаленный сервис немедленно прекращаются. По истечении тайм-аута предохранитель пропускает ограниченное количество тестовых запросов. Если эти запросы выполняются успешно, предохранитель возобновляет нормальную работу. В противном случае, если происходит сбой, период тайм-аута начинается снова.	https://microservices.io/patterns/reliability/circuit-breaker.html
413	**Access token** Токен доступа	**Security** Безопасность	**API-шлюз передаёт токен с информацией о пользователе, включая идентификатор и роли, сервисам, к которым тот обращается.** **Пролема:** Как сообщить личность запрашивающего сервисам, обрабатывающим запрос? **Решение:** Шлюз API аутентифицирует запрос и передает токен доступа (например, веб-токен JSON), который надежно идентифицирует запрашивающую сторону в каждом запросе к сервисам. Сервис может включать токен доступа в запросы, которые он отправляет другим сервисам.	https://microservices.io/patterns/security/access-token.html
430	**Log aggregation** Агрегация журналов	**Observability** Наблюдаемость	**Агрегирует журналы всех сервисов в центральной БД с поддержкой поиска и уведомлений.** **Проблема:** Как понять поведение приложения и устранить неполадки? **Решение:** Используйте централизованный сервис ведения журналов, который объединяет журналы из каждого экземпляра сервиса. Пользователи могут искать и анализировать журналы. Они могут настроить оповещения, которые срабатывают при появлении определенных сообщений в журналах.	https://microservices.io/patterns/observability/application-logging.html
435	**Application metrics** Показатели приложения	**Observability** Наблюдаемость	**Сервис шлёт отчёты с показателями центральному серверу, который предоставляет агрегацию, визуализацию и оповещения.** **Пролема:** Как понять поведение приложения и устранить неполадки? **Решение:** Инструментируйте сервис для сбора статистики об отдельных операциях. Агрегируйте метрики в централизованном сервисе метрик, который предоставляет отчеты и оповещения. Существует две модели агрегирования метрик: push — сервис отправляет метрики в сервис метрик pull — сервисы метрик извлекают метрики из сервиса	https://microservices.io/patterns/observability/application-metrics.html
440	**Audit logging** Ведение журнала аудита	**Observability** Наблюдаемость	**Записывает действия пользователя в БД, чобы помочь сервису поддержки, обеспечить соблюдение нормативно-правовых норм, и обнаружить подозрительную активность.** **Проблема:** Как понять поведение пользователей и приложения и устранить неполадки? **Решение:** Запись действий пользователя в базу данных.	https://microservices.io/patterns/observability/audit-logging.html
432	**Distributed tracing** Распределённая трассировка	**Observability** Наблюдаемость	**Назначает каждому внешнему запросу уникальный идентификатор и отправляет данные о его перемещениях по системе от одного сервиса к другому на центральный сервер, который предоставляет визуализацию и возможность анализа.** **Пролема:** Как понять поведение приложения и устранить неполадки? **Решение:** Инструментальные сервисы с кодом, который Присваивает каждому внешнему запросу уникальный идентификатор внешнего запроса. Передает идентификатор внешнего запроса всем сервисам, участвующим в обработке запроса. Включает идентификатор внешнего запроса во все сообщения журнала Записывает информацию (например, время начала, время окончания) о запросах и операциях, выполняемых при обработке внешнего запроса в централизованном сервисе. Это инструментирование может быть частью функциональных возможностей, предоставляемых платформой Microservice Chassis.	https://microservices.io/patterns/observability/distributed-tracing.html
438	**Exception tracking** Отслеживание исключений	**Observability** Наблюдаемость	**Сервис сообщает об исключениях центральному сервису, который их дедуплицирует, отслеживает их исправление и генерирует оповещения.** **Пролема:** Как понять поведение приложения и устранить неполадки?	https://microservices.io/patterns/observability/exception-tracking.html

427	**Health Check API** API проверки работоспособности	**Observability** Наблюдаемость	**Сервис открывает доступ к конечной точке (endpoint) наподобие GET /health , которая возвращает данные о работоспособности сервиса.** **Проблема:** Как определить, что запущенный экземпляр сервиса не может обрабатывать запросы? **Решение:** Сервис имеет конечную точку API проверки работоспособности (например, HTTP/health), которая возвращает работоспособность сервиса. Обработчик конечной точки API выполняет различные проверки, такие как 1) состояние подключений к сервисам инфраструктуры, используемым экземпляром сервиса 2) статус хоста, например. дисковое пространство 3) специфическая логика приложения Клиент проверки работоспособности — сервис мониторинга, реестр сервисов или балансировщик нагрузки — периодически вызывает конечную точку для проверки работоспособности экземпляра сервиса.	https://microservices.io/patterns/observability/health-check-api.html
	Log deployments and changes	**Observability** Наблюдаемость	**Проблема:** Как понять поведение приложения и устранить неполадки? **Решение:** Регистрируйте каждое развертывание и каждое изменение в (производственной) среде.	https://microservices.io/patterns/observability/log-deployments-and-changes.html
	Server-side page fragment composition	**UI patterns** Шаблоны пользовательского интерфейса	**Проблема:** Как реализовать экран или страницу пользовательского интерфейса, отображающую данные из нескольких сервисов? **Решение:** Каждая команда разрабатывает веб-приложение, которое создает фрагмент HTML, реализующий область страницы для их сервисов. Команда пользовательского интерфейса отвечает за разработку шаблонов страниц, которые создают страницы, выполняя агрегацию на стороне сервера (например, механизм включения стилей на стороне сервера) фрагментов HTML, специфичных для сервисов.	
	Client-side UI composition	**UI patterns** Шаблоны пользовательского интерфейса	**Проблема:** Как реализовать экран или страницу пользовательского интерфейса, отображающую данные из нескольких сервисов? **Решение:** Каждая команда разрабатывает компонент пользовательского интерфейса на стороне клиента, такой как директива AngularJS, которая реализует область страницы/экрана для их сервиса. Команда пользовательского интерфейса отвечает за реализацию скелетов страниц, из которых строятся страницы/экраны, путем составления нескольких компонентов пользовательского интерфейса для конкретных сервисов.	https://microservices.io/patterns/ui/client-side-ui-composition.html