# 2. Микросервисы

“Невозможно описать термин “микросервис”, потому что не существует даже словарного запаса для этой области. Вместо этого мы опишем список типичных черт микросервисов, но сделаем это со следующей оговоркой - большинству микросервисных систем присущи лишь некоторые из приведенных черт, но не всегда, и даже для совпадающих черт будут значительные различия от канона.” Мартин Фаулер (Martin Fowler), одно из первых выступлений, посвященных глубокому анализу микросервисов).

Как мы выяснили из первой главы, обзора концепции и технологий, “созданных для облака” (cloud native), практически неотъемлемой частью проектирования и разработки приложений для работы в облаке стали “микросервисы” (microservices), особенно бурно ворвавшиеся в тренд популярности на волне успеха стека технологий и способа разработки Netflix, Twitter, Uber, и до этого идей от Amazon.

Определить точно, что это за архитектура, и чем она формально отличается от очень известного до этого подхода SOA (service oriented architecture), то есть архитектуры ориентированной на сервисы, довольно сложно. Многие копья сломаны на конференциях и форумах, создано множество блогов, можно сделать определенные выводы. Прежде всего микросервисы отличаются от “монолитов” (monolith), приложений, созданных с помощью единой технологии или платформы, внутри которой находятся вся деловая логика системы, анализ данных, обслуживание и выдача данных пользовательским интерфейсам. Любое взаимодействие модулей, сервисов и компонентов внутри монолита как правило происходит в рамках одного или максимум несколько процессов.

Плюсы монолита очевидны - мгновенная скорость общения между сервисами и компонентами, зачастую в рамках одного процесса, общая база кода, меньшее количества ограничений на общений между компонентами и модулями, менее общие, более точные и выделенные интерфейсы между ними.

Однако с развитием облачных вычислений, и особенно легких контейнеров, изолирующих любые технологии, возрастанием скорости обмена данных по сети, и общей надежности и встроенной устойчивости к отказам, предоставляемых основными провайдерами облака, стало особенно удобно разбивать приложение на множество более мелких приложений. Они предоставляют друг другу сфокусированные, маленькие услуги и сервисы с помощью обмена информацией (как правило, это текстовый формат HTTP/JSON, или двоичный формат gRPC), независимые от использованных микросервисом технологий.

Подобное разбиение идеально ложится на разделение бизнес-функций в общем приложении, а что еще лучше, великолепно разделяет обязанности большой команды инженеров на независимые, маленькие команды, способные к экспериментам, быстрым изменениям, и использованиям любых технологий.

## Монолиты

Красивое слово монолит (monolith) описывает хорошо известный, наиболее часто используемый способ разработки программного продукта. Ваша команда определяется с набором требований к продукту и делает примерный выбор технологий и архитектуры. Далее вы создаете репозиторий для исходного кода (чаще всего GitHub), выделяете общую функциональность и библиотеки (пытаясь сократить количество повторного кода, DRY - don’t repeat yourself!), и вся команда добавляет новый код и функциональность в этот единственный репозиторий, как правило, через ветви кода (branch). Код компилируется единым блоком, собирается одной системой сборки, и все модульные тесты прогоняются также сразу, для всего кода целиком. Рефакторинг и масштабные изменения в таком коде сделать довольно просто.

Однако, если брать разработку в облаке, и зачастую мгновенно и кардинально меняющиеся требования современных Web и мобильных приложений, описанные удобства грозят некоторыми недостатками.

### Склонность к единой технологии

Единый репозиторий кода и одна система сборки естественным образом ведут к выбору основной технологии и языка, которые будут исполнять большую часть работы. Компиляция и сборка разнородных языков в одном репозитории неудобны и чрезмерно усложняют скрипты сборки и время этой сборки. Взаимодействие кода из разных языков и технологий не всегда легко организовать, проще использовать сетевые вызовы (HTTP/REST), что еще сильнее может запутать код, который находится рядом друг с другом, однако общается посредством абстрактных сетевых вызовов.

Тем не менее, для каждой задачи есть свой оптимальный инструмент, и языки программирования не исключение. Микросервисы, максимально разбивая и изолируя код частей системы, дают практически неограниченную свободу в выборе языка, платформы и реализации задачи, без взрывной сложности сборки проекта. Как мы вскоре увидим, контейнеры с блеском справляются с задачей легковесной виртуализации, и совершенно различные технологии способны с легкостью взаимодействовать друг с другом.

### Сложность понимания системы

Часто говорят, что большая, созданная единым монолитом система сложна для понимания для новых членов команды. В мире технологий нередко размер команды резко растет, требуется срочно создать новую функциональность, и ключевым фактором становится скорость начала работы с ней и ее кодом ранее незнакомых с ней программистов. Мне кажется это довольно неоднозначный момент, и качественно сделанная система с разбиением на модули, правильной инкапсуляцией и скрытием внутренних винтиков системы будет не сложнее для понимания, чем сеть из десятков микросервисов, взаимодействующих по сети. Все зависит от дисциплины и культуры команды.

Но в общем случае стоит признать, что созданная командой (с ее внутренней дисциплиной и культурой) система скорее будет более прозрачной и понятной в виде микросервисов и качественно разделенных друг от друга репозиториев, чем в виде огромного кода размером в сотни тысяч строк, особенно если новый программист начинает работу над четко определенной задачей в одном микросервисе.

### Трудность опробования инноваций

Монолитная, сильно связанная система, где код может легко получить доступ к другим модулям, и начать использовать их в тех же благих целях не делать ту же работу заново (общие модули и библиотеки), может в результате затруднить создание новых возможностей, не способных идеально вписаться в существующий дизайн.

Представим себе, что мы написали отличную биржу для криптовалют. Неожиданно появляется новая валюта, но правила работы с ней совершенно не похожи, и просто не будут работать с нашими системами обработки заказов. В случае монолитной системы нам надо вносить изменения в общий код, и продумать множество граничных случаев, чтобы обработка всех валют могла сосуществовать. В концепции микросервисов идеально было бы обработку новой валюты отвести совершенно новому микросервису. Да, придется заново писать много похожего кода, но в итоге эта работа будет идеально соответствовать требованиям. Более того, если эта новая криптовалюта окажется “пустышкой”, ненужный микросервис удаляется, нагрузка на систему и ее сложность немедленно снижается - сделать такой рефакторинг в сильно связанном коде монолита может быть непросто, да это будет и не самым первым приоритетом.

### Дорогое масштабирование

Монолитное приложение собирается в единое целое и, в большинстве случаев, начинает работать в одном процессе. При возрастании нагрузки на приложение возникает вопрос увеличения его производительности, с помощью или *вертикального масштабирования* (vertical scaling, усиления мощности серверов на которых работает система), или *горизонтального* (horizontal scaling, использования более дешевых серверов, но в большем количестве для запуска дополнительных экземпляров (replicas, или instances).

Монолитное приложение проще всего ускорить с помощью запуска на более мощном сервере, но, как хорошо известно, более мощные компьютеры стоят непропорционально дороже стандартных серверов, а возможности процессора и размер памяти на одной машине ограничены. С другой стороны, запустить еще несколько стандартных, недорогих серверов в облаке не составляет никаких проблем. Однако взаимодействие нескольких экземпляров монолитного приложения надо продумать заранее (особенно если используется единая база данных!), и ресурсов оно требует немало - представьте себе запуск 10 экземпляров серьезного корпоративного Java-приложения, каждому из них понадобится несколько гигабайт оперативной памяти. В коммерческом облаке все это приводит к резкому удорожанию.

Микросервисы решают этот вопрос именно с помощью своего размера. Запустить небольшой микросервис проще, ресурсов требуется намного меньше, а самое интересное, увеличить количество экземпляров можно теперь не всем компонентам системы и сервисам одновременно (как в случае с монолитом), а точечно, для тех микросервисов, которые испытывают максимальную нагрузку. Kubernetes делает эту задачу тривиальной.

## Архитектура на основе сервисов (SOA)

Более гибким решением является разработка на основе компонентов, отделенных друг от друга, прежде всего на уровне процессов, в которых они исполняются. Архитектуру подобных проектов называют ориентированной на сервисы (service oriented architecture, SOA).

Разработка приложения в виде компонентов, и стремление свести сложные приложения к набору простых, хорошо стыкующихся между собой компонентов известна видимо с тех самых времен как программы стали разрабатывать. По большому счету подобная техника применима во многих областях человеческой деятельности.

Часто говорят, что классическая архитектура на основе сервисов отличается от ставших популярными сейчас “микро”-сервисов тем, что многое отдает на откуп “посредникам” (middleware), например системам обмена, настройки и хранения сообщений между компонентами и сервисами, так называемым “интеграционным шинам” (ESB, enterprise service bus). Микросервисы же эпохи облака минимизируют зависимость от работы со сложными посредниками и их правилами и проводят свои операции напрямую друг с другом.

## Микросервисы по Мартину Фаулеру

Мартин Фаулер знаменит своими, как правило, очень успешными попытками найти структуру и систему в динамичном, хаотичном мире программирования, архитектуры, хранения данных, и особенно в мире сложных, высоконагруженных, распределенных приложений эпохи Интернета. Конечно же он попытался проанализировать и упорядочить волну популярности микросервисов.

Если попытаться вчитаться в цитату Мартина в начале этой главы, становится понятно, что микросервисы - явление вне рамок известных ранее архитектур, они не совсем укладываются в стандарты, и довольно разнообразно применяются на практике. Тем не менее, характерные черты микросервисов, используемые Мартином, очень хороши для первого, вводного анализа этой архитектуры, что нам для знакомства и нужно. Давайте посмотрим.

### Сервисы как компоненты системы

Компонент (component) - одна из основополагающих идей в разработке программ. Идея проста - компонент отвечает за определенную функцию, интерфейс API, или целый бизнес модуль. Главное - компонент можно убрать из системы, заменив на другой, со сходными характеристиками, без больших изменений для этой системы. Замена компонентов целиком встречается не так часто, но независимое обновление (upgrade) компонента без потери работоспособности системы в целом очень важно для легкости ее обновления и поддержки. Классический пример - качественная библиотека (к примеру, JAR), модуль, или пакет, в зависимости от языка, которую можно подключить и использовать в своем приложении.

Микросервисы - это использование компонентов (зависимостей, библиотек и модулей) своего приложения не внутри одного, единого процесса приложения, а запуск их в отдельных, собственных процессах, и доступ к ним не напрямую, а через сетевые вызовы RESTful API / GRPC.

### Логика в микросервисах, но не в сети и посредниках

Данная черта микросервисов противопоставляется предыдущим версиям распределенных систем (часто их называют SOA, service oriented architecture). В архитектуре SOA большая роль отводилась компонентам-“посредникам” (middleware), таким как сложные очереди сообщений (message queue), адаптерам данных, и общему набору логики между различными компонентами системы, называемому *интеграционной шиной ESB* (enterprise service bus). Зачастую львиная доля сложной логики всей системы находится не в самих компонентах, а именно в шине ESB.

В мире микросервисов это явно выраженный анти-шаблон (anti-pattern). Вся логика в обязательном порядке должна находиться внутри микросервисов (“разумные” точки доступа к сервису, smalt endpoint), даже если она повторяется в различных микросервисах. Сеть (и любой механизм передачи данных в целом), должна только передать данные, но не обладать никакой логикой (как еще говорят, “неразумные” линии передачи данных, dumb pipes).

### Децентрализованное хранение данных

В монолитной архитектуре данные зачастую хранятся в единой, обычно реляционной, базе данных, со строго определенной схемой и обширным использованием запросов SQL. Микросервисы снова требуют противоположного подхода - в идеале никакого разделения данных, особенно через единую базу данных. Весь доступ, любые изменения данных происходят только через программный интерфейс API, предоставляемый микросервисом.

Микросервисы, таким образом, свободны в своем выборе способа хранения данных. Это может быть любой тип базы данных, например, нереляционная база NoSQL, база на основе документов, графовая база, или что-либо еще. Конечно, возникает значительная избыточность данных, но хранение данных не так дорого, а автономность и независимость развития каждого микросервиса, в идеале, должна окупить избыточность данных.

### Культура автоматизации

Учитывая лавинообразный рост зависимостей между компонентами, и независимый набор технологий в каждом из микросервисов, надежный, автоматизированный выпуск системы и ее быстрые обновления - это жизненно необходимый элемент микросервисной архитектуры. Ручная сборка и управление микросервисами практически невозможны.

Непрерывная интеграция и тестирование (CI, continuous integration), непрерывное развертывание новых версий (CD, continuous delivery) - это обязательный атрибут команд, создающих микросервисы. Здесь огромную помощь оказывают контейнеры, со своей быстрой, легковесной виртуализацией, и “чистым”, изолированным пространством, в котором запускаются микросервисы, а развертывание значительно упрощает оркестратор Kubernetes, предоставляя мощный декларативный подход. Их мы тщательно рассмотрим в следующих главах, представить микросервисы без них уже практически невозможно.

### Расчет на постоянные сбои

Микросервисы приводят к распределенной, гетерогенной, динамичной системе. Ей просто предначертано испытывать постоянные сбои, из-за сетевых вызовов, нагрузок, несовместимости версий, и многого другого. Этого не избежать, и на это нужно просто рассчитывать изначально, планируя каждый вызов к остальным компонентам системы как изначально не способный гарантировать успешное завершение.

Планировать сбои можно не только на этапе дизайна кода и отдельных тестов, но и более активным способом. Так, компания Netflix использует “обезьяну с гранатой” (Chaos Monkey, перевод не дословный, но наша фраза на русском языке вполне подходит!), отдельный сервис, постоянно выводящий из строя случайный набор микросервисов. Остальные компоненты системы должны подстроиться к ситуации, и правильно реагировать на сбои системы, проводя разумную политику отката своих действий, или же используя вспомогательные компоненты и альтернативные способы исполнения логики.

### Небольшая команда, акцент на своей бизнес-области

Микросервисы как правило разрабатываются небольшой командой (известен практически анекдот от компании Amazon, что команда, работающая над микросервисом, всегда сможет насытиться двумя пиццами. Пиццы скорее всего большие, возможно, с разными наполнителями - так что в итоге это может быть и чуть больше 10 человек). Идея состоит в том, что небольшой команде проще координировать свои действия и быстро проводить новые идеи и изменения в жизнь.

Второй аспект - команда больше не состоит только из программистов серверной части (back end), пользовательских интерфейсов (front end, или UI), или же системных администраторов. Они работают вместе над одной, сконцентрированной бизнес-областью. Классический пример - онлайн магазин, и сервисы по доставке, оформлению заказов, и проведению платежей. В мире микросервисов все эти сервисы разрабатываются, выпускаются в эксплуатацию, и развертываются отдельными командами, в своем ритме.

Более того, у каждой команды могут быть отдельные требования, процесс, планирование, и даже заказчики - внутренние или внешние. Общаются эти команды независимо, и микросервисы работают независимо, предоставляя четко очерченные программные интерфейсы API.

Интересно, что подобное требование лежит за пределами чисто технической, программисткой зоны ответственности - такое распределение обязанностей должно лежать в основе всей организации целиком, с множеством автономных зон ответственности, без единого, “жесткого” центра управления. Как гласит известный закон Конвея (Conway’s law), структура организации обязательно проявит себя в планировании и производстве любых продуктов и сервисов этой организации.

### Владение продуктом, а не реализация проекта

Зачастую, в классической модели разработки, команда программистов и дизайнеров реализует проект, исполняя поставленные перед ней требования (requirements). Что происходит после того, как проект “сдан”? Команда начинает работать над чем-то новым, переходит в другие проекты, а в случае субподрядчиков и контрактных услуг, работа с ней может быть полностью закончена. Обслуживание системы, исправление ошибок, ее обновление нередко попадает совершенно в другие руки.

Парадигма микросервисов предпочитает, чтобы команда разработчиков “владела” (*own*) своим проектом в начале его дизайна, в процессе создания и настройки микросервисов, и обязательно после формальной сдачи системы, непрерывно продолжая работу над ее эволюцией. Постоянный контакт с пользователями системы, наблюдение за ней в условиях реальной эксплуатации позволяет максимально эффективно понять, как ее развить и улучшить. Как вариант подобной системы популярна концепция DevOps (developer + operations, совместная работа как над разработкой, так и над поддержкой и эксплуатацией системы). Оркестратор Kubernetes сам по себе подталкивает работать в концепции DevOps.

### Эволюционный дизайн

“Эволюционный дизайн (evolutionary design) - попытка добиться результата не с одной попытки, чудесным образом дающей идеальное попадание, а в результате многих постепенных изменений, приводящих к эволюции дизайна и продукта”, Джошуа Кириевски (Joshua Kerievsky), адепт Agile/XP-разработки.

Микросервисы дают большую свободу приверженцам эволюционного дизайна, противникам тщательного предварительного анализа системы, и попытки понять ее целиком еще до того, как написана первая строка кода, и сделан выбор первых технологий для ее реализации. Вместо классического планирования и дизайна всей необходимой функциональности системы, создается первое “примитивное” приближение (primitive whole). Система выглядит практически реально, но умеет очень мало. Пользователи получают возможность впервые посмотреть на систему, и конечно же, немедленно просят поменять или улучшить ее, не целиком, но значительно. В подобных итерациях и рождается истина.

Если создание монолитной системы немедленно приводит к сильными зависимостям от выбранной технологии, системы хранения данных и библиотек, то добавка новых микросервисов, и даже быстрое удаление “неудачных” или “временных” микросервисов становятся делом техники. У разработчиков развязаны руки - они могут свободно обращаться с выбором технологий, а также передвигать границы микросервисов - если изначально они были выбраны неверно, в первых итерациях их легко объединить или разбить.

### Краткие итоги

Первый обзор типичных черт микросервисов внушает оптимизм. Четкое разделение обязанностей, абсолютная свобода в выборе технологий и хранении данных, точно настроенное масштабирование, и помощь контейнеров и Kubernetes выглядят очень заманчиво. Однако не стоит забывать о сложности распределенной системы, присущей любой, даже самой простой такой системе. Сетевые вызовы часто приносят с собой неизвестные заранее задержки (latency), и для быстрой работы приходится выполнять работу асинхронно (asynchronous), заранее не зная, в какой момент приходит ответ от остальных компонентов системы.

Сложность асинхронной, распределенной, динамично развивающейся системы в разы превышает сложность единого, монолитного приложения. Разделить систему на микросервисы зачастую очень сложно, если в компании “все делают все”, и нет четко очерченных бизнес-областей. В таких случаях вместо микросервисов получается “распределенный монолит” (distributed monolith), неповоротливая, сложная система, вместо которой мог бы иметь место более эффективный монолит.

Не забывайте, что два остальных столпа концепции Cloud Native совершенно безразличны к битве монолитов, микросервисов и SOA! Вы можете совершенно спокойно развернуть классическое, монолитное приложение Enterprise Java в контейнере под управлением Kubernetes, получив многие преимущества без излишней сложности.

Один из давних соратников Мартина, Сэм Ньюмен (Sam Newman), написал отдельную, и кстати, не слишком “толстую”, книгу по микросервисам, которую можно рекомендовать для чтения, и существует ее перевод на русский язык.

## Разбиение системы на микросервисы

Если осмыслить основные качества системы, созданной на основе микросервисов, начинает казаться, что их использование - совершенно универсальное, великолепное решение, практически панацея, или как любят говорить в технологиях, “серебряная пуля”. Запустив систему в облаке под управлением Kubernetes, где многие неприятности и специфичные проблемы микросервисов решаются за нас, можно ли спокойно получить все их преимущества?

Проблема же заключается больше в попытке понять, что будет микросервисом в вашей системе, а что будет лишь частью или библиотекой, работающей в составе большого сервиса. Неверное определение границ микросервисов (boundary) приведет к запутанному, сложному коду, чрезмерно раздутым программным интерфейсам API, и может сорвать все сроки разработки, а то и к поспешной попытке “склеить” все обратно в монолит.

Качественный процесс дизайна и архитектуры приложения подразумевает разделение компонентов, сервисов и объектов, представляющих собой данные, согласно области бизнеса (domain), для которого приложение разрабатывается. Это основа DDD, дизайна архитектуры приложения на основе области его применения (domain driven design). Однако именно здесь для микросервисов кроется неприятные подводный камень. Дело в том, что разбить компоненты и сервисы заранее очень тяжело, требования, как водяные знаки, появляются лишь по мере проявления приложения, пользователи зачастую полностью меняют свое мнение как только видят первые варианты приложения.

Это меньшая проблема для монолитов - рефакторинг компонентов и их интерфейсов довольно просто сделать внутри одного процесса и одной базы технологий и языков. В случае микросервисов перенести часть функциональности в другой сервис, зачастую написанный на другом языке или платформе, полностью поломать и поменять крупные интерфейсы REST/gRPC между ними очень непросто.

Отсюда вытекает один из начальных этапов разработки, направленный на выявление границ микросервисов. Разработка идет в соответствии с базовыми принципами DDD, и приложение разбивается на модули согласно выявленным *ограниченным контекстам* (bounded context) - области работы, автономной самой по себе. В примерах с вездесущими электронными магазинами это может быть обслуживание склада и инвентаря магазина, отдельно от них существует система доставки, обработка счетов и так далее. Но, модули работают как часть монолита, первого приближения эволюционного дизайна, упомянутого нами выше.

После получения первых, очень грубых приближений системы в целом, с минимумом возможностей, начинают проявляться более четкие границы ее ограниченных контекстов. Этот момент прекрасно подходит для разбиения модулей на микросервисы, если нужно, смену технологий в них, и децентрализацию данных. Принятые решения еще нужно будет пересмотреть, однако они уже основаны на реальном коде, дизайне, и не являются просто плодом сухой теоретической подготовки.

## Обратная сторона медали

Подчеркнем еще раз - за блеском и преимуществами микросервисов, лежащих на поверхности, легко не заметить всех сложностей и совершенно другой парадигмы именно работы, эксплуатации всей системы в целом. Даже если удастся удачно разбить систему по ограниченным контекстам и минимизировать их зависимости, создать хорошо настроенные программные интерфейсы API, изучить Kubernetes и успешно развернуть и масштабировать свое приложение, то понять в итоге, что же происходит в работающей системе, будет в разы сложнее.

Можно забыть о прямой отладке в вашем редакторе IDE, если ваш код взаимодействует с несколькими микросервисами, и все они отвечают асинхронно. Интеграционные тесты будут очень сложны и поддержка их требует много ресурсов. Понять по журналам (logs) одного компонента, что происходит в системе в целом, невозможно. Производительность системы будет настолько распределена между отдельными микросервисами и сетевыми вызовами, что измерять нужно будет все сразу. Распределенные транзакции между различными хранилищами данных. Расчет на сбои всего и вся. Доверие образам контейнеров. Задача защиты трафика между микросервисами системы, обслуживание сертификатов SSL, авторизация, безопасность, роли… Продолжать можно еще долго.

Есть хорошие новости - экосистема созданных для облака приложений Cloud Native буквально наводнена инструментами и решениями для перечисленных нами вызовов. Зачастую они бесплатны и с открытым кодом, и каждый день появляются новые решения. Самая распространенная проблема - управление сетевыми вызовами между микросервисами, отслеживание задержек, шифрование трафика - неплохо решается так называемыми *микросервисными сетками* (service mesh) - такими как Istio и Linkerd. Мы еще вспомним про них в дальнейших главах. Сбор распределенных журналов также отлично решается, например стеком ELK (Elastic, Logstash, Kibana), или Fluentd. Стандарт OpenTracing, метрики Prometheus, и отчеты Graphana уже встроены во многие библиотеки для создания микросервисов, и просто используя их, вы получите мощнейший центр наблюдения на своей системой.

Тем не менее, все это богатство надо изучить, выбрать нужное и подходящее вам, и настроить - эти издержки надо обязательно добавить в общую стоимость разработки реальной системы из микросервисов.

## Резюме

Микросервисы выглядят заманчиво, а вместе с контейнерами и оркестрацией Kubernetes и вовсе как очевидный выбор. Тем не менее, существует высокая цена эксплуатации системы, созданной на их основе, и экосистема для работы с ними требует инвестиций времени и ресурсов. В этой главе нет примеров - каждая система уникальна, и зависит прежде всего от области своего применения (domain). Архитектурные решения высокого уровня трудно описывать без сложного конкретного примера, поэтому мы увидели все с высоты птичьего полета, но все концепции данной главы можно попробовать перенести на свой собственный проект.

Чтобы понять архитектуру и философию микросервисов чуть лучше, можно посоветовать следующие книги и ресурсы:

* Сэм Ньюмен (Sam Newman), “Создание микросервисов”.
* www.cncf.io - главный сайт фонда Cloud Native Foundation, посмотрите раздел проектов (projects), многие посвящены работе микросервисных систем, в том числе OpenTracing и Prometheus.
* Некоторые видео с конференций KubeCon удачно описывают микросервисы для людей с разной степенью подготовки, найдите их канал на YouTube.
* Эрик Эванс (Eric Evans), “Предметно-ориентированное проектирование (DDD).”
* www.martinfowler.com - статьи про микросервисы и связанные концепции. Кое-что доступно и на русском языке.