Модуль 1, урок 1

На этом уроке обсудим концепции монолитной и микросервисной архитектуры, плюсы и минусы каждого подхода, разберемся, в каких случаях стоит применять то или иное решение.

Начнем с монолитной архитектуры, как наиболее понятной и простой. **Монолитное приложение, или монолит,** — это приложение, которое доставляется конечным пользователям в виде единого неделимого артефакта.

Пример. Предположим, вы разрабатываете интернет-магазин по продаже книг. Интернет-магазин — это приложение, которое состоит из разных модулей: *авторизация*, *просмотр заказов, email-рассылки, SMS-уведомления* и так далее. Все модули являются частью одного приложения или, проще говоря, единой программы, которая запускается целиком. В таких случаях монолит работает с одной базой данных и чаще это реляционная БД.

Как видно на схеме, независимо от того, насколько монолит сложный, он компилируется в единый артефакт. В случае платформы JVM это может быть jar-файл или же docker-контейнер. Но, так или иначе, это приложение, которое поднимает модули рассматриваемого монолита целиком.

Плюсы монолита

- Простой деплой
- Низкий time-to-market
- Согласованность данных
- Инкапсуляция зависимостей
- Мониторинг
- Единые правила контрибьютинга

- **Простой деплой**
  Это главный плюс монолита. Если приложение представлено в виде единого бинарного артефакта, то запустить его на сервере достаточно просто. С этим может справиться разработчик — к задаче не нужно призывать DevOps-специалиста или администратора.

- **Низкий time-to-market**
  Благодаря тому, что процесс развертывания приложения происходит просто, можно быстро и эффективно доставлять новые фичи конечным клиентам. Это важно на ранних этапах разработки, если вы, скажем, стартап и для вас критично показать инвесторам готовую функциональность, чтобы получить инвестиции. Поэтому стартапы чаще начинаются с монолита.

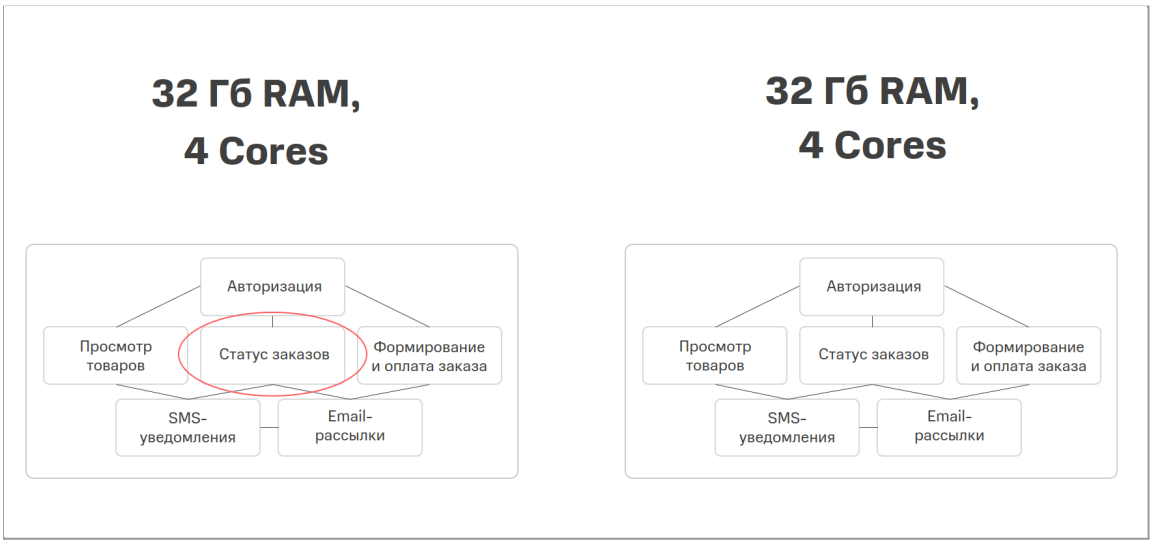
- **Согласованность данных**
  Взаимодействие с монолитом происходит следующим образом: к нам приходит внешний запрос. Мы открываем транзакцию, производим определенные действия, и, если запрос завершается успешно, транзакция коммитится. Если нет — откатывается. Независимо от того, какие ошибки происходят в рамках запроса, мы можем легко откатить изменения назад и гарантировать согласованность данных на всем отрезке существования системы. Это важное преимущество в контексте разработки, тем более больших и сложных приложений.

- **Инкапсуляция зависимостей**
  Монолит предоставляет некий API, по которому взаимодействует пользователь. Это может быть REST API, как пример. Но то, как в этом случае взаимодействуют модули внутри монолита, не сильно интересно, потому что клиенты знают о неком фасаде, к которому они обращаются. Это дает возможность добавлять новые модули, удалять старые, менять взаимодействие между ними при необходимости.

- **Простота настройки мониторинга**
  Поскольку монолит является единым приложением, для него требуется настроить мониторинг один раз. Этого достаточно, чтобы получить исчерпывающую информацию о работе программы.

- **Единые правила контрибьютинга**
  Монолит — это продукт, который хранится в одном репозитории. Так как разработчики работают с одним-единственным репозиторием, мы можем с помощью статического анализа, гайдлайнов, договоренности между командами определить качество конечного кода, который должен доставляться в мастер-ветку. Если контролировать эти изменения, то можно гарантировать планку, которая была задана изначально. Такой подход плодотворно влияет на дальнейшую разработку и поддерживаемость.

Минусы монолита



- **Горизонтальная масштабируемость**
  Несмотря на плюсы, у монолита есть недостатки. Главная проблема — горизонтальная масштабируемость. Предположим, что приложение по продаже книг через интернет-магазин потребляет 32 Гб оперативной памяти и работает на 4 ядрах процессора. В ходе мониторинга мы обнаружили, что модуль *статус заказов* не справляется с нагрузкой. Чтобы запросы обрабатывались более эффективно, мы решили развернуть второй инстанс — точную копию используемого монолита. Потребление ресурсов остается на прежнем уровне: 32 Гб RAM и 4 ядра процессора. В сущности модуль *статус заказов* может потреблять гораздо меньше ресурсов, но мы вынуждены развернуть его целиком, потому что монолит является приложением, которое нельзя раздробить на части.

- **Низкая отказоустойчивость**
  Если в какой-либо части монолита возникает непредвиденная ошибка, он может перестать работать. При бизнес-подходе даже двухминутный простой может стоить сотни тысяч долларов, что критично для большинства компаний.

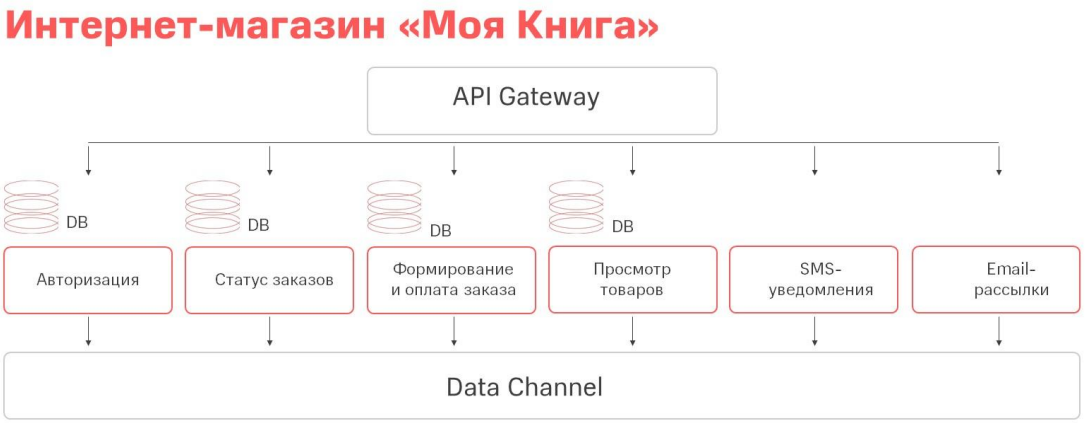
- **Спагетти-код**
  Это не яркая черта монолита, потому что его можно встретить абсолютно в любых программных продуктах. Но здесь работает банальная логика. Поддерживать маленькие проекты легче, чем большие. А если продукт существует более 10 лет, обеспечивать такой же уровень качества, как в начале проекта, довольно сложно. Поэтому часто возникает ситуация, когда по истечении долгого времени монолит превращается в трудно поддерживаемый код.

- **Сложность внедрения новых технологий**

  Каждый разработчик слышал о таком понятии, как legacy-код, когда программный продукт реализован на старом фреймворке. Зачастую возникает потребность перейти на более современный и оптимальный формат. Устаревание объясняется тем, что 10 лет назад используемые фреймворки были вполне современными и отвечали необходимым требованиям. Но прошло время, и они устарели. Возможно, вышли новые фреймворки, появились более оптимальные языки программирования. Перейти на новые рельсы бывает непросто, потому что в монолите могут использоваться функции, которые были объявлены как deprecated, и теперь от них сложно отказаться. Возможно, используемый фреймворк больше не поддерживается, а перейти на новый фреймворк или даже на новый язык может быть непосильной задачей.

Микросервисная архитектура

Озвученные проблемы стали драйвером нового подхода — микросервисной архитектуры. Это вариант сервис-ориентированной архитектуры ПО, направленный на взаимодействие небольших, слабо связанных и легко изменяемых модулей — микросервисов.



Рассмотрим уже знакомый пример с интернет-магазином по продаже книг. В контексте микросервисов он может выглядеть как на схеме. Здесь также присутствуют модули *авторизация, статус заказов, оплата* и так далее. Но теперь каждый из этих модулей является отдельным независимым приложением, которое разворачивается параллельно с остальными. Обратите внимание: у некоторых из приложений есть своя база данных. Между собой они взаимодействуют через Data Channel — это может быть синхронное взаимодействие, например через REST, или асинхронное —

через очередь сообщений. Также может использоваться комбинированный вариант.

API Gateway — сервис, который предоставляет единый API для клиентов. Он занимается проксированием запросов на необходимые микросервисы в зависимости от того, какого поведения вы хотите добиться. У вас может быть API Gateway для мобильных устройств, для веб-браузера, для каких-то других задач в зависимости от специфики вашего продукта.

Плюсы микросервисов

- **Хорошая горизонтальная масштабируемость** — главный плюс. Вернемся к примеру со статусом заказов. Если мы обнаружили, что модуль не справляется с нагрузкой, можно увеличить количество инстансов. Причем мы потратим ровно то количество ресурсов, которое необходимо.

- **Обеспечение отказоустойчивости** — если по какой-то причине отказал один из модулей, а остальные продолжают работать, то система также продолжает функционировать. Но здесь могут быть ограничения.

  Например, если ломается модуль по оплате заказа, можно ли приложение считать работоспособным? Едва ли, потому что оплата заказов — основной модуль. Именно он приносит деньги. И если пользователи могут делать все, но не могут покупать книги, польза от такого интернет-магазина будет сомнительной.

- **Независимая деградация.** Если один из микросервисов выходит из строя, он не влияет на работоспособность остальных.

- **Легче внедрять новые технологии.** Если какую-то часть приложения можно переписать на более современный язык и это даст рост производительности, такая задача будет вполне посильной. Можно полностью отказаться от предыдущего и написать все заново. В случае с монолитом такую задачу решить сложнее.

- **Большая команда быстрее разрабатывает.** Может показаться, что монолит — это приложение, над которым работает 20—30 человек одновременно. Но это не так. Большое количество разработчиков — скажем, 10 человек — не могут эффективно работать над одним проектом по ряду причин: задачи нужно разбить на части, они не должны пересекаться, работа должна выполняться параллельно. Но, даже если мы сможем это сделать для 10 человек в рамках одного сервиса, возникнет большое количество merge-конфликтов. Мы получим не прирост производительности, а снижение.

> Вспоминается старая добрая шутка: если один программист может решить задачу за один день, то два программиста решат ее за два дня.

С микросервисами ситуация другая. Если у вас есть 20 микросервисов, вы можете нанять 60 человек и закрепить по три разработчика за каждым микросервисом.

- **Независимые релизы.** Поскольку микросервисы разрабатываются отдельными командами, они могут релизиться с разной частотой. Например, у вас есть микросервис, который должен релизиться часто, он критичный, и вы хотите видеть изменения в продакшене каждый день. И есть другой микросервис — не настолько важный, его достаточно релизить раз в две недели. С микросервисами можно легко этого добиться, а с монолитом практически невозможно, потому что нужно прийти к соглашению: релизиться либо раз в день, либо раз в две недели. Других вариантов нет.

Минусы микросервисов

- **Версионирование API** — главная проблема. Для понимания рассмотрим пример, в котором используются два микросервиса: *авторизация* и *статус заказов*.

Статус заказов по REST API обращается к модулю *авторизации*, чтобы узнать, есть ли у пользователя права на совершение тех или иных действий. Предположим, что модуль *авторизации* релизится каждый день, а модуль *статуса заказов* — раз в неделю. Вышел новый релиз модуля *авторизация*, и коллеги решили переделать API, сделав его более удобным и функциональным.

В итоге выпустили новую версию, но модуль *статус заказов* про это ничего не знает — он продолжает обращаться по старому API и получает ошибку. Поэтому, когда вы разрабатываете микросервисы, вам приходится поддерживать несколько версий одновременно. Это может быть две, три, пять версий — все определяется контекстом вашей бизнес-области. Но

отойти от этого вы не можете. А поддерживать несколько версий одновременно гораздо сложнее, чем одну.

- **Сложный деплой.** Если в случае с монолитом требовалось развернуть одно приложение, то в случае с микросервисами потребуется развернуть 10, 20, 30 или даже 100 приложений, которые взаимодействуют между собой по сложным алгоритмам. При этом нужно обеспечить отказоустойчивость, где-то развернуть три, где-то пять инстансов и следить, чтобы все работало корректно. Здесь сил одних разработчиков не хватит. Потребуется отдельный DevOps-специалист, который будет заниматься решением таких задач.

- **Согласованность в конечном счете.** Один из плюсов монолита заключается в возможности обеспечить строгую согласованность. Если в случае с монолитом вы стартовали транзакцию, откатили ее, то с микросервисным подходом другая история: каждый микросервис использует свою базу данных.

> **Пример.** При поступлении запроса на совершение действия запрос отправляется на другой микросервис, чтобы внести какие-то изменения. Микросервис отвечает положительно: да, я все сделал. Мы пытаемся завершить какие-то действия, и у нас возникает ошибка. Получается так, что на одной стороне все отработало корректно, а на другой — мы не внесли данные. В результате возникли расхождения.

Как решаются такие проблемы? Есть вариант использовать двухфазный коммит. Но, как правило, прибегают к *согласованности в конечном счете*. Мы гарантируем, что система когда-то придет в согласованное состояние, но какое-то время она может быть несогласованной.

> **Пример.** Вы поставили лайк в соцсетях и неважно, увидит ли его человек сразу, через 10 секунд, через 15 или даже через минуту.

С другой стороны, возникают ситуации, когда согласованность важна.

> **Пример.** Если вы переводите деньги и человек узнает о них только через две недели — это уже проблема.

Обработка ошибок. Если в монолите происходит ошибка, можно заглянуть в логи. Они могут храниться на диске, складываться в систему типа Kibana, но, так или иначе, прослеживается логика выполнения запроса, в результате чего можно понять, что конкретно было не так.

Пример. Допустим, в цепочке из пяти микросервисов возникла ошибка и нужно понять, где именно она произошла. Ведь, если каждый из микросервисов сообщил Error, это не значит, что ошибка произошла на всех микросервисах. Она могла произойти на одном из них. Остальные — просто не смогли обработать запрос. При этом пять разных лог-файлов складываются в пять разных пространств, и понять, где конкретно возникла ошибка, довольно затруднительно.

Решение находится в уникальном идентификаторе запросов. На каждый запрос генерируется ключ, который проставляется в Header на протяжении выполнения всей бизнес-операции. Проанализировав логи, вы сможете понять, что и где пошло не так.

- **Логирование.** В случае с монолитом используется один лог, в случае с микросервисами — отдельно взятые журналы. Ориентироваться в таком количестве информации затруднительно. Можно использовать системы ELK, но они не решают проблему полностью.

- **Мониторинг.** В случае с монолитом вы следите за одним приложением, в случае с микросервисами — за 20 или 30.

- **Тестирование.** Когда возникает необходимость протестировать монолит, нужно локально развернуть одно приложение. Возможно, потребуется мощный компьютер, но задача достаточно тривиальная. В случае с микросервисами вам нужно «поднять» всю систему локально и проверить все сложные взаимодействия. Как правило, так никто не делает. Необходимо проверить частичное взаимодействие критичных составляющих. Если вы хотите проверить систему целиком, необходимо выделить так называемый Staging-сервер (окружение, которое является точной копией вашего продакшен-сервера) и задеплоить всю архитектуру там. Но тот факт, что разработчики не могут развернуть разом все микросервисы на своей машине, несколько затрудняет разработку, увеличивает требования к качеству кода и отказоустойчивости.

Выводы

Монолит	Микросервисы
Быстрый старт, сложности в перспективе	Сложности на старте, плюсы в перспективе

Монолит — отличное решение на начало проекта, позволяет быстро стартовать и быстро доставлять новые фичи. Но при росте могут возникнуть проблемы с отказоустойчивостью, потреблением ресурсов и деградацией.

Микросервисы — сложны в начале проекта, не дают преимуществ молниеносно. Но если у вас большая высоконагруженная система, то микросервисы помогут решить проблемы, связанные с отказоустойчивостью и деградацией.

Вывод: серебряной пули не существует. Выбирайте решение, которое подходит под ваш бизнес-кейс.