**1) Зачем нужен Spring?**

Это платформа которая ускоряет и упрощает разработку приложения. Суть заключается в следующем - мы управление, создание между зависимостями объектов передаем фреймворку. Нам не нужно следить за жизненным циклом объектов их созданием, удалением, внедрением.

**2) Откуда Spring берет зависимости?**

Берет из контейнера зависимостей. Он работает на основе метаданных, которые мы отдаем спрингу в виде xml или javaconfig или повешенных аннотаций, либо смесь аннотаций и xml, так же можно отдать groovy скрипт

**3) Что произойдет если спринг не найдёт нужные зависимости?**

Упадет приложение с ошибкой- данная зависимость не найдена

Чтобы приложение не упала если бин не был найден – можно исп @Autowired аннотацию, в которую передать параметр required = false, И если спринг не найдёт этот бин, то он передаст туда просто null. Пометив таким образом что бин не обзательный, и его можно не инжектить

*Способ не особо*- повесить аннотацию @Lazy, но если мы обратимся к бину, его там не окажется и все равно упадет приложение. Либо исп Conditional установив условия использования

**4) А случай кода нашлось несколько бинов?**

Будет ошибка определения какой бин внедрять (несколько кандидатов. Это решается аннотацией **@Primary** говоря о том что какой то бин первичный. Либо повесить **@Qualifire** где указать имя бина

**5) Какие типы внедрение зависимостей**

- Конструктор

- Сеттер

- в Поля которые приватные непосредственно(использует рефлексию)

**6) Что использовать если пишем приложение на спринге с использованием БД**

- Можно исп JDBC – либа для работ с бд

- Hibernate (преобразуем объекты бд(строки) в классы джава)

**7) Проблемы виде когда мы объект мапим на реляционную модель БД (проблемы при представлении данных в объектном)**

- В объектах есть наследование, а в реляционных таблицах этого нет и мы не можем спроецировать наследников(у которых добавляться поля) от базового класса, мы не сможем сложить нормально в одну таблицу в бд. Нам придаться сделать свалку данных и 1го объекта и его наследника.

Решение- делать таблицу для каждого объекта, но будет дублирование данных, потому что родитель имеет поле, а его наследники дополняют данные. И тогда не будет общей таблицы с полями.

- Нельзя проецировать коллекции на реляционную модель. В бд нет понятия коллекции, мы оперируем таблицами там

Решение – делать доп таблицу и вспомогательные ключи и отображать связи

**8) Проблемы Hibernate и как решаються**

N+1 – избыточные запросы

Проблема N + 1 возникает, когда фреймворк доступа к данным выполняет N дополнительных SQL-запросов для получения тех же данных, которые можно получить при выполнении одного SQL-запроса.

Она встречается, когда мы достаем список сущностей со связями (которые нам необходимо подтянуть) И вместо того чтобы сразу сделать джоин двух таблиц, хибер делает селект из 1 таблицы и потом для каждой найденной строки будет делать дополнительный селект чтобы во второй таблице найти связанную строку. Проблема не только у хибера наблюдаеться

Решение: делать джоин и не делаем n+1 – сообщаем хиберу путем указания нужного джоина

Либо же используем аннотацию @EntityGraph для указания связанной сущности

можно использовать JOIN FETCH, чтобы избежать проблемы с N + 1:

List<PostComment> comments = entityManager**.createQuery**("""  
 select pc  
 from PostComment pc  
 join fetch pc.post p  
 """, PostComment**.class**)  
**.getResultList**();

@Lazy – загрузка сущностей (связей) только в момент обращения. Даже если вы явно перейдете на использование FetchType.LAZY для всех ассоциаций, то вы все равно можете столкнуться с проблемой N + 1.

@Eager -загрузка жадная- сразу же

https://habr.com/ru/companies/otus/articles/529692/

**9) Какие оптимизации запросов в БД существуют?**

Вместо кучи insert делать один большой на все строки.

Использование индексов при поиске по таблицам.

Индекс – сущность в бд, работает как указатель (вешаем на колонку). Индекс ускоряет процесс запроса, предоставляя быстрый доступ к строкам данных в таблице, аналогично тому, как указатель в книге помогает вам быстро найти необходимую информацию.

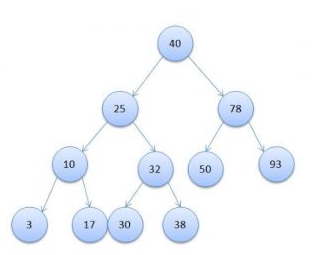
<https://www.youtube.com/watch?v=LpEwssOYRKA>

В общем виде, индекс в базах данных — это файл с последовательностью пар ключей и указателей.

Какие типы индексов бывают

Bitmap-индексы - построение двухмерного массив

Дерево (древовидный индекс) — это специального вида структура, у которой есть корневая вершина и у каждого узла может быть несколько дочерних узлов. При этом каждый узел встречается только один раз и может иметь всего один родительский узел. Выглядит это так:



Хеш индекс – считает хеш для значений столбца и сопоставляет ему колонку

Джин индекс – который исп для индексации сложны структру и полнотекстового поиска

Составные индексы — строятся по нескольким полям, при этом расположение полей является важным.

Обычные индексы — состоят из одного поля. Здесь, вероятно, все понятно. Обычный каталог страничек.

**10) Отличие Dependency inversion от Dependency Injection**

https://techrocks.ru/2021/02/15/dependencies-inversion-and-injection/

**Микросервисы**

1) Плюсы и минусы

Удобство поддержки для разработчика

Простота разработки

Минусы-нужна развитая инструктора и поддержка со стороны девопсов

- подключение балансировщика если сервис откажет

Частично решает проблему докер - слоями ПО

Проблема распределенного состояния и отсутствие консистентности в микро-сервисной архитектуре

Когда приходит запрос – по системе он продвигается постепенно и какие то сохранили результаты а какие то нет.

Проблема обнаружения ошибок- нужно постоянно анализировать и отлавливать логи какой микросервис выкидывает ошибку. Проблема децентрализации логов сервиса. Здесь приходит на помощь кибана и эластик серч

**Монолиты**

Традиционный подход к созданию приложений путем объединения всех функций в единую сущность, объединяющую интерфейс и серверную часть, по-прежнему актуален. Дело в том, что не всегда есть смысл в усложнении архитектуры. Если требования к сервису (например, к веб-сайту) включают только скорость, доступность и возможность обслуживания статического контента, который находится в свободном доступе, монолитная архитектура для таких целей — вполне естественный выбор. А всё остальное было бы чрезмерным усложнением.

**Плюсы монолитов**

* Скорость, простота и низкая стоимость создания. Приложения, основанные на монолитной архитектуре, как правило, быстрее, проще и дешевле создавать, поскольку они не требуют синхронизации и объединения сложных частей. Поэтому, если монолитное приложение будет делать то, что от него требуется, без ущерба для продвижения, то можно использовать монолитную архитектуру.
* Удобное тестирование. Протестировать монолитное приложение гораздо проще, чем микросервисы или бессерверное. Можно запустить и протестировать приложение на сервере разработчика или в промежуточной среде, а также применить стандартный процесс развертывания для проверки изменений перед запуском приложения в продакшн.
* Простая инфраструктура. Монолитные приложения используют один сервер для внешнего интерфейса, серверной части и базы данных, что упрощает требования к инфраструктуре. А ​​для повышения скорости, масштабируемости, доступности и безопасности монолитных веб-приложений можно добавить сеть доставки контента (например, Cloudflare).
* Снижение затрат на поддержку. При монолите вам нужно поддерживать только один репозиторий. Вам также понадобится только один конвейер тестирования и развертывания, что может значительно снизить затраты, поскольку создание, настройка и обслуживание нескольких конвейеров выйдет дороже, ведь нужно будет обеспечить согласованность между ними. Все данные, используемые приложением, также могут храниться в единой базе данных.
* Мониторинг. Простая инфраструктура обеспечивает дополнительное преимущество в виде упрощения мониторинга.
* Легкое управление транзакциями и поддержка целостности данных. Тот факт, что монолитные приложения обычно используют одну базу данных, означает, что здесь проще управлять транзакциями и поддерживать целостность данных.

**Минусы монолитов**

* Большие массивы кода. Монолитная архитектура подразумевает единую кодовую базу. Если объемы кода растут, его становится труднее читать и понимать, что затрудняет интеграцию новых разработчиков в проект.
* Могут быть проблемы с обновлениями. Обновление монолитного приложения означает перестройку и повторное развертывание всего кода.
* Высокий уровень зависимостей. Единый код означает, что компоненты тесно связаны, и изменения в логике одного модуля или службы сопряжены с гораздо более высоким риском влияния на код и работу других модулей. Порой трудно предсказать последствия для приложения даже после небольших изменений в коде. Поэтому каждое обновление вынуждает повторно тестировать весь продукт.
* Отсутствие гибкости. Монолитная архитектура одновременно ограничивает диапазон технологий, которые вы можете использовать, и означает, что технологический стек должен быть строго согласован в глобальном плане. Поэтому вы не можете использовать другую, возможно, более подходящую технологию для каких-то отдельных частей приложения.
* Проблемы с масштабированием. Монолитные приложения плохо масштабируются. Поэтому, если вы создаете монолитное приложение, которое неожиданно становится чрезвычайно популярным и должно справляться с гораздо большими нагрузками, а также расширяться, вам придется перенести его на другой шаблон. Либо на микросервисы, либо на бессерверную архитектуру.

**Примеры монолитов**

Монолитная архитектура обычно лучше всего подходит для небольших и простых приложений. Мы уже приводили пример веб-сайта. Другим примером может быть быстрая и экономичная разработка MVP или программного обеспечения, предназначенного для выполнения какой-то простой задачи.

**Микросервисы**

С развитием цифровых технологий растет сложность приложений. Как мы уже говорили выше, монолитная архитектура может стать непреодолимой преградой для масштабирования. Кроме того, очень важно стало и распределение рисков путем разделения сервисов таким образом, чтобы сбой одного не приводил к остановке всего приложения. Для этого и был разработан микросервисный подход. Архитектура микросервисов разделяет приложение на отдельные службы или группы.

**Плюсы микросервисов**

* Итеративная разработка. Архитектура программного обеспечения на основе микросервисов хорошо подходит для организации гибкой разработки ПО, основанной на итеративном подходе. Разбиение уже работающего приложения на отдельные, слабо связанные между собой сервисы означает, что новые возможности и функции могут быть проверены и добавлены без риска остановки приложения.
* Гибкость. Микросервисы обеспечивают большую гибкость процесса разработки ПО сразу на нескольких уровнях. Это позволяет разным командам одновременно работать над разными сервисами и упрощает интеграцию в процесс разработки новых членов команды, ведь разделенный на отдельные функциональные части код легче читать и понимать.
* Свобода в выборе технологического стека. В монолитной архитектуре, где сервисы тесно связаны, важно поддерживать согласованность технологий, используемых в приложении. Они также должны быть хорошо совместимы друг с другом, что характерно не для всех языков программирования и средств разработки ПО. Слабо связанные микросервисы обеспечивают гораздо большую свободу в выборе стека технологий. Вплоть до того, что команда разработчиков может даже создавать отдельные сервисы с очень разным набором инструментов разработки.
* Доступность. Одним из самых больших преимуществ архитектуры микросервисов является то, что слабо связанные сервисы обеспечивают большую надежность приложения и более высокую доступность. Если одна служба выходит из строя, то это, как правило, не влияет на остальную часть приложения, которое продолжит функционировать для пользователей.
* Масштабируемость. Еще одним плюсом микросервисов является то, что этот подход хорошо подходит для масштабирования приложений. Модульность микросервисов позволяет легко добавлять новые функции прямо в работающее приложение. Хотя первоначальные затраты на архитектуру микросервисов выше, ее масштабирование может быть значительно дешевле, чем у монолитного приложения, благодаря сочетанию уже упомянутых сильных сторон.
* Производительность. Также за счет разделения служб и нагрузки между несколькими серверами возможно значительно повысить производительность.

**Минусы микросервисов**

* Сложность синхронизации. Распределенная система, такая как микросервисы, неизбежно создает дополнительную сложность, поскольку ее части необходимо синхронизировать таким образом, чтобы они могли работать как единая программная система. И если службы разделены между серверами, вам придется подготовить инфраструктуру для взаимодействия микросервисов.
* Сложность тестирования. С одной стороны, тестировать отдельные микросервисы проще, с другой, необходимость тестировать каждый сервис по отдельности может значительно усложнить приложение по мере его масштабирования. А необходимость тестировать и поддерживать связь между службами создает дополнительную сложность.
* Более высокие первоначальные затраты. Хотя попытки значительно масштабировать монолитное приложение могут оказаться сущим кошмаром, который приводит к резкому росту прямых затрат на разработку, архитектура микросервисов требует более высоких первоначальных затрат. Для каждого микросервиса требуется своя команда разработчиков (хотя одна команда может отвечать и за несколько). Кроме того, придется наладить и процесс автоматического тестирования и развертывания (CI/CD). Всё это приводит к более высоким первоначальным затратам.
* Потребность в DevOps. Распределенная система, такая как микросервисы, требует квалифицированной оркестровки, обычно с использованием Kubernetes и других инструментов и процессов DevOps. Это означает, что вам нанять хотя бы одного инженера DevOps, что также увеличит расходы.

Впрочем, в микросервисной архитектуре повышенные затраты и сложность компенсируются большей гибкостью и значительным улучшением производительности. Таким образом, создание приложения на основе микросервисов увеличивает первоначальные затраты, но взамен предлагает большую независимость и гибкость, что ускоряет циклы выпуска. И для многих современных компаний микросервисный подход стал скорее необходимостью, чем роскошью.

**Примеры микросервисов**

Микросервисная архитектура подходит для крупных и сложных приложений. В качестве примера можно привести приложение электронной коммерции, которое можно разбить на следующие микросервисы:

* фронтенд (пользовательский интерфейс);
* служба поиска, которая позволяет пользователям искать товары в базе данных с помощью поисковых запросов;
* служба связанных товаров, которая рекомендует альтернативные или дополнительные продукты, используя различные алгоритмы;
* сервис корзины покупок;
* сервис оплаты, который занимается обработкой платежей.

[**https://timeweb.cloud/blog/monolity-mikroservisy-besservernaya-arhitektura**](https://timeweb.cloud/blog/monolity-mikroservisy-besservernaya-arhitektura)

**3) Api GATEWAY**

Проблема когда фронтенду нужно 5 разных микросервисов опросить чтобы получить данные- например статус заказа, баланс на карте, данные карточки. На помощь приходит Api GATEWAY отдельный сервис точка входа для фронтенда. Серввис собирает нужный ответ – по опр урлу отправляется в нужные 5 сервисов. Согрегирует и вернет на фронт. Здесь соблюдается инкапсуляция

**4) Типы взаимодействия между микросервисов**

Рест, соап(xsd схема и xml), графКуЭль. JRPS. Можно исп очередь данных(тибко, ребит, кафка)

Очередь используем когда несколько экземпляров (асинхронное взаимодействие). Либо же когда нам не важно когда обработают сообщения, тоесть асинхрон. Кафка исп -когда нужен гарантии что сообщение будет доставлено.

Рест же – синхронное.