GeekBrains

Архитектура ПО

Методичка к уроку N10

Паттерны проектирования и структура приложений с пользовательским интерфейсом, сервисами обработки и базой данных

Оглавление

1. Проектирование ПО и ИС

1.1. Подходы к проектированию ПО

1.1.1. Use Case Driven Approach

1.1.2. Domain Driven Design (DDD)

Единый язык предметной области

Ограниченный контекст

Предметная область, предметная подобласть, смысловое ядро

Слои

Строительные блоки DDD

1.1.3. “Programmer Driven Design”

1.2. Подходы к разработке ПО

1.2.1. FDD - Feature Driven Development

1.2.2. BDD - Behaviour Driven Development

1.2.3. TDD - Test Driven Development

1.2.4. PDD - Panic Driven Development

2. Проектирование прикладных приложений

2.1. Чистая архитектура

2.2. Виды архитектур прикладных приложений с точки зрения клиент-серверного взаимодействия

2.3. Архитектура приложения при использовании нескольких API

2.4. Способы организации UX/UI прикладных приложений

3. Паттерны проектирования прикладных приложений

Глоссарий

Дополнительные материалы

Используемые источники

Введение

Прикладное программное обеспечение предназначено для выполнения определённых задач и рассчитано на непосредственное взаимодействие с пользователем.

В большинстве ОС прикладные программы не могут обращаться к ресурсам вычислительной системы напрямую, а взаимодействуют с оборудованием и другими программами посредством сервисов и служб ОС.

К прикладному программному обеспечению относится программное обеспечение, разработанное для пользователей или самими пользователями для задания компьютеру конкретной работы. Программы обработки заказов, управления складом, управления банковским счётом или создания мультимедийного контента — примеры прикладного программного обеспечения.

Информационная система — система, предназначенная для хранения, поиска и обработки информации, и соответствующие организационные ресурсы (человеческие, технические, финансовые и т. д.), которые обеспечивают и распространяют информацию. ИС предназначена для своевременного обеспечения людей подходящей информацией, то есть для удовлетворения конкретных информационных потребностей в рамках определённой предметной области, при этом результатом функционирования информационных систем является информационная продукция — документы, информационные массивы, базы данных и информационные услуги.

В определённом смысле ИС относятся к категории прикладного программного обеспечения, так как реализуют набор прикладных функций в соответствии с некоторой прикладной областью.

Отличительной чертой всех прикладных приложений являются:

- Организация процессов получения данных (от пользователей, внешних систем, из реального мира посредством датчиков и др.)
- Организация процессов обработки данных (бизнес-логика, алгоритмы пред- и пост- обработки информации)
- Организация хранения данных (чтения, модификации, добавления и удаления данных)
- Организация процессов отправки данных (во внешние системы, подключаемые устройства, пользователям и т. д.)

Организация взаимодействия с пользователем (т. е. организация пользовательского интерфейса: текстового, графического, звукового, тактильного).

Вне зависимости от прикладных задач, решаемых проектируемым ПО, обычно необходимо анализировать и проектировать рассмотренные выше процессы. Естественно, что для большинства из них сложились определённые принятые наборы подходов и шаблонов.

На этом уроке

- Узнаем подходы к проектированию и разработке приложений
- Изучим варианты организации архитектуры клиент-серверных приложений
- Изучим Паттерны проектирования прикладных приложений
    - Паттерны связывания компонент и уровней приложения
    - Паттерны управления данными
    - Паттерны управления состоянием приложения
    - Паттерны структурирования приложений
- Узнаем, что такое Разрушающие изменения API — Breaking Changes API

1. Проектирование ПО и ИС

Если обратиться за ответом к словарю терминов Systems and software engineering (ISO/IEC/IEEE 24765:2017), то можно получить следующее определение:

Проектирование - процесс определения архитектуры, системных элементов, интерфейсов и других характеристик системы или её части.

Конечном продуктом процесса проектирования, является *проект*, который описывает физическую и логическую структуру системы, её поведение и отношение между элементами. Проектная документация должна быть достаточно полной для начала реализации системы.

Отправной точкой для начала проектирования архитектуры являются различные требования к системе. Требования можно разделить на две группы:

- *Функциональные требования* - это требования, которые определяют то, что должна делать система.
- *Нефункциональные требования* - это требования, которые определяют, как система должна выполнять свои задачи и какими качествами обладать.

Этапы проектирования типового приложения

Этап 1. Сбор требований и формализации их в виде ТЗ.

Источником требований являются могут быть **заказчики системы**, **будущие пользователи (клиенты) системы**, **обслуживающий персонал**, а также **требования стандартов** и **нормативно-правовых актов**.

В зависимости от подхода к проектированию, требований регуляторов и заказчика для фиксации требований может использоваться разный набор документов. Рассмотрим некоторые из них.

Техническое задание — это документ, в котором фиксируются требования к проекту информационной системы и исключающий двусмысленное толкование различными исполнителями.

В нем должно описываться все, что необходимо для разработки и внедрения системы.

Приведем примерную структуру ТЗ:

1. Общие сведения
2. Назначение и цели создания (развития) системы
3. Характеристики информационной системы
4. Требования к информационной системе
5. Состав и содержание работ по созданию (развитию) информационной системы
6. Порядок контроля и приемки информационной системы
7. Виды, состав, объем и методы испытаний информационной системы и ее составных частей
8. Общие требования к приемке работ по стадиям
9. Требования к составу и содержанию работ по подготовке информационной системы к вводу в эксплуатацию
10. Требования к документации

При написании ТЗ можно ориентироваться на следующие документы:

- IEEE 830-1998. Методика составления спецификаций требований к программному обеспечению.
- ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств.
- ISO/IEC/IEEE 29148-2011. Systems and software engineering — Life cycle processes — Requirements engineering.
- ГОСТ Р 54869-2011. Проектный менеджмент. Требования к управлению проектом.
- ГОСТы серии 34.

Следующим этапом после сбора требований и формализации их в виде ТЗ, или в любом другом согласованном с заказчиком виде, является непосредственно само проектирование.

Этап 2. Продолжение проектирования архитектуры.

На выходе этапа проектирования создается проектная документация, которая может быть представлена набором различных диаграмм и пояснительной записки. Однако стоит отметить, что в разных методологиях стадии и этапы проектирования могут отличаться.

Например для гибкой методологии, все этапы проектирования и разработки проходят за одну итерацию и корректировка требований к системе возможна в начале каждого цикла. В таком случае создание детальной проектной документации становится трудоёмкой задачей.

При таком подходе становится актуальным применение системы контроля версии для ведения проектной документации, а также различных электронных баз знаний.

Шаги проектирования

Так как большинство прикладных приложений по типам автоматизируемых процессов похожи друг на друга, устоялся набор часто используемых практик и подходов.

Базовым решением является применение идей DDD совместно с принципами чистой архитектуры и ряда устоявшихся паттернов проектирования отдельных элементов прикладного приложения (семейство паттернов MVC*, паттерн Repository, паттерн DTO и др.).

Примерный план проектирования для типового прикладного приложения может включать следующие моменты:

1. Анализ требований.
2. Анализ предметной области (домена), выработка единого языка, ограниченного контекста и формализация домена ядра.
5. Формирование сущностей домена (сущностей ядра).
6. Формализация сценариев (посредством пользовательских сценариев - Use Case и описаний бизнес процессов - BPMN).
7. Анализ сценариев, выработка уточненных требований к приложению и его частям (требования к хранению данных, требования по взаимодействию с пользователем).
8. Примерный выбор проектных и технологических решений по хранению и представлению данных.

9. Выбор архитектурных паттернов для связывания и реализации слоев и их частей
10. Проектирование слоя сценариев (бизнес-логики).
11. Проектирование слоя контроллеров и адаптеров.
12. Выбор фреймворков и библиотек для внешнего слоя, фиксация способов хранения и представления данных (работа с деталями).

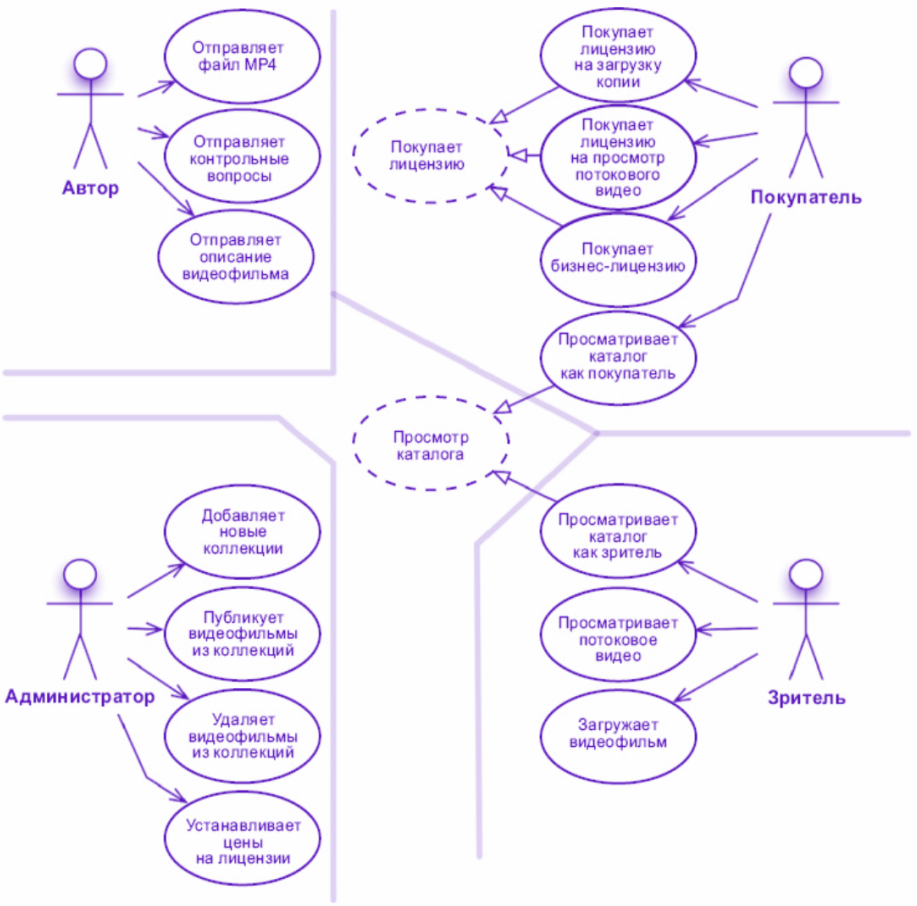
1.1. Подходы к проектированию ПО

Use Case Driven Approach

В данной методологии сценарии использования (Use Cases), определенные для системы, являются основой для всего процесса разработки. Из традиционной объектно-ориентированной модели системы трудно понять, что же система делает. Эта трудность возникает из-за отсутствия связующей нити через всю систему, когда та выполняет определенные задачи. В Rational Unified Process такой нитью являются сценарии использования, поскольку они определяют поведение, выполняемое системой .

Сценарии использования играют определенную роль в каждом из четырех основных рабочих процессов: требования, анализ и проектирование, реализация и тестирование.

Модель вариантов использования является результатом процесса разработки требований к информационной системе. В этом раннем процессе сценарии использования нужны для моделирования того, что система должна делать с точки зрения пользователя. Таким образом, сценарии использования представляют собой важную фундаментальную концепцию, которая должна быть приемлема как для заказчика, так и для разработчиков системы.



В анализе и проектировании сценарии использования реализуются в модели проектирования. Вы создаете реализации вариантов использования, которые описывают, как выполняется вариант использования в терминах взаимодействующих объектов в модели проектирования. Эта модель описывает в терминах объектов проектирования различные части внедряемой системы и то, как эти части должны взаимодействовать для выполнения сценариев использования.

Во время реализации роль спецификации для выполнения играет проектная модель. Поскольку сценарии использования являются основой для модели проектирования, они реализуются в терминах классов проектирования.

Во время тестирования примеры использования являются основой для определения тестовых примеров и процедур тестирования. То есть, система проверяется путем выполнения каждого варианта использования. У вариантов использования есть и другие роли:

- используются как основа для планирования итеративной разработки.
- формируют основу для того, что описывается в руководствах пользователя.

- могут использоваться в качестве единиц заказа. Например, заказчик может получить систему, сконфигурированную с определенным набором вариантов использования.

Domain Driven Design (DDD)

Domain-driven design или предметно-ориентированное проектирование - это концепция, согласно которой структура и язык программного кода (имена классов, методы классов, переменные классов) должны соответствовать бизнес-области. Данный термин был впервые введен Э. Эвансом в его книге с таким же названием «Domain-Driven Design»

Единый язык предметной области

Постоянно возникающая проблема при разработке программного обеспечения связана с пониманием кода, что он собой представляет, что он делает, как он это делает, почему он это делает. Еще сложнее понять код, если в нем используется терминология, отличная от терминологии экспертов домена, например, если эксперты домена говорят о старших пользователях, а код говорит о супервайзерах, это может внести много путаницы при обсуждении приложения. Однако большую часть этой двусмысленности можно решить с помощью правильного именования классов и методов, позволяя им выразить, чем является объект и что делает метод в контексте домена. Основная идея использования единого языка заключается в том, чтобы привести приложение в соответствие с бизнесом. Это достигается путем принятия общего языка между бизнесом и технологией в коде.

Источником *единого языка* является бизнес сторона компании, у них есть концепции, которые необходимо реализовать, но терминология затем согласовывается с технологической стороной компании (это означает, что бизнес сторона не всегда выбирает лучшее название) с целью создания общей терминологии, которая может использоваться бизнесом, технологией и в самом коде без каких-либо двусмысленностей.

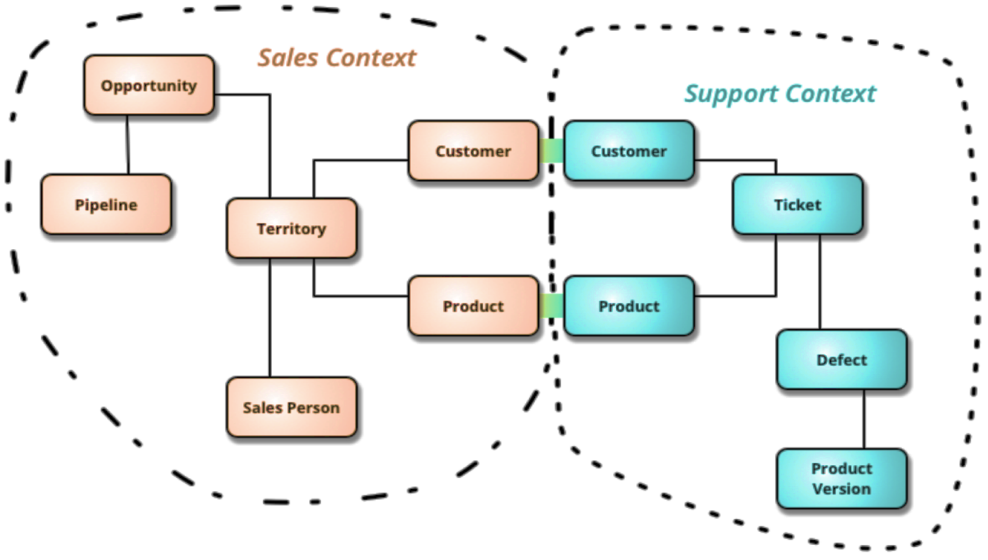
Код, классы, методы, свойства и наименование модулей должны соответствовать единому языку. При необходимости стоит провести рефакторинг кода!

Очень важно понимать, что в рамках предметной области смысл определенного термина или фразы может сильно отличаться. Существует некая граница, в пределах которой понятия единого языка имеют вполне конкретное контекстное значение.

Ограниченный контекст

Эта концептуальная граница называется ограниченный контекст (Bounded context). Это второе по значимости свойство DDD после единого языка. Оба эти понятия взаимосвязаны и не могут существовать друг без друга.

- В каждом ограниченном контексте существует только один единый язык.
- Ограниченные контексты являются относительно небольшими, меньше чем может показаться на первый взгляд. ограниченный контекст достаточно велик только для единого языка изолированной предметной области, но не больше.
- Единый значит «вездесущий» или «повсеместный», т. е. язык, на котором говорят члены команды и на котором выражается отдельная модель предметной области, которую разрабатывает команда.
- Язык является единым только в рамках команды, работающей над проектом в едином ограниченном контексте.
- Попытка применить *единый язык* в рамках всего предприятия или что хуже, среди нескольких предприятий, закончится провалом.
- При этом, **ограниченный контекст - это способ организации классов/объектов, которыми вы манипулируете в коде.** Между ними может быть взаимно-однозначная корреляция, или не может быть. Разумеется, нет никакой необходимой связи между понятиями.



Пример разделения контекстов

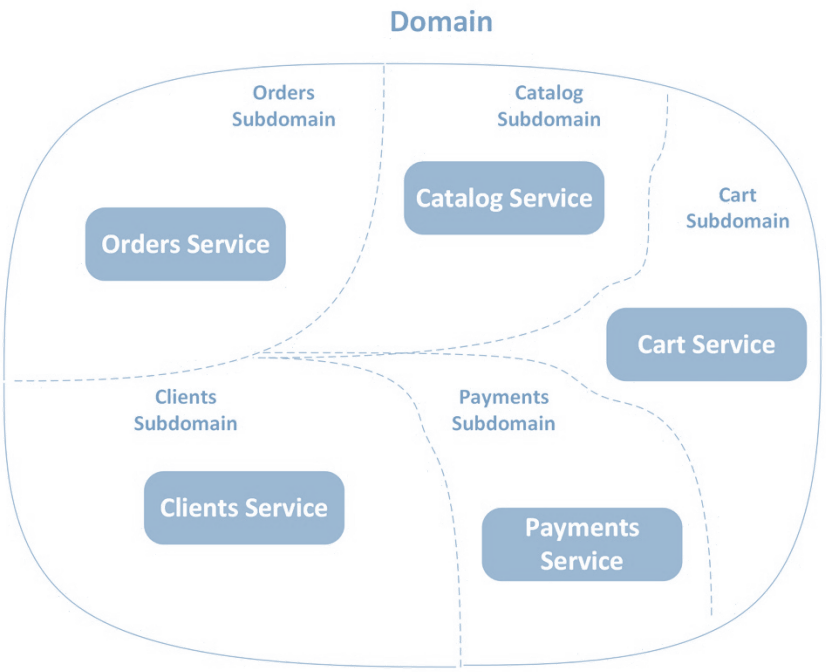
Предметная область, предметная подобласть, смысловое ядро

Предметная область (Domain) – это то, что делает организация, и среда, в которой она это делает. Разработчик программного обеспечения для организации обязательно работает в ее предметной области.

Область знаний/деятельности, для которой разрабатывается приложение. Например, что для юристов есть свои термины и важные понятия, которые отличаются от слов, используемых в нефтегазовой промышленности. Причем это не просто слова, они отражают важные для заказчика процессы, связи. Именно поэтому важно уделять внимание терминам, которые используются в данной сфере. Это обеспечивается постоянным общением двух сторон — разработчиков приложения и клиентов. Такой подход отражает главный принцип DDD — разработка не должна быть в отрыве от бизнес-задач. DDD не является инструкцией или методологией, а составляет набор правил и ориентиров.

Следует понимать, что при разработке модели предметной области необходимо сосредоточиться в определенной подобласти, так как практически невозможно создать единственную, всеобъемлющую модель бизнеса даже умеренно сложной организации.

Очень важно разделять модели на логические разделенные *предметные подобласти* (Subdomain) всей предметной области, согласно их фактической функциональности. Подобласти позволяют быстрее определить разные части предметной области, необходимые для решения конкретной задачи.



Декомпозиция по поддоменам

Также необходимо уметь определять *смысловое ядро* (Core domain). Это очень важный аспект подхода DDD. Смысловое ядро – это подобласть, имеющая первостепенное значение для организации. Со стратегической точки зрения бизнес должен выделяться своим смысловым ядром. Большинство DDD проектов сосредоточены именно на смысловом ядре. Лучшие разработчики и эксперты должны быть задействованы именно в этой подобласти. Большинство инвестиций должны быть направлены именно сюда для достижения преимущества для бизнеса и получения наибольшей прибыли.

Если моделируется определенный аспект бизнеса, который важен, но не является смысловым ядром, то он относится к *служебной подобласти* (Supporting subdomain). Бизнес создает служебную подобласть, потому что она имеет специализацию. Если она не имеет специального предназначения для бизнеса, а требуется для всего бизнеса в целом, то ее называют *неспециализированной подобластью* (Generic subdomain). Эти виды подобластей важны для успеха бизнеса, но не имеют первоочередного значения. Именно смысловое ядро должно быть реализовано идеально, поскольку оно обеспечивает преимущество для бизнеса.

Модель

Система абстракций, которая описывает отдельные характеристики домена. Как и физическая модель, упрощающая понимание и изучение объекта, помогает решить проблемы, связанные с данным доменом.

Слои

Рассмотрим слои, определенные в DDD:

1) **Пользовательский интерфейс**

Отвечает за рисование экранов, с помощью которых пользователи взаимодействуют с приложением, и преобразование пользовательского ввода в команды приложения. Важно отметить, что "пользователи" могут быть людьми, но могут быть и другими приложениями, подключающимися к нашему API.

2) **Прикладной уровень**

Оркестрирует объекты домена для выполнения задач, требуемых пользователями: Use Cases. Он не содержит бизнес-логики. На этом уровне располагаются службы приложений, поскольку они являются контейнерами, в которых происходит оркестровка сценариев использования, используя репозитории, службы домена, сущности, объекты значений или любые другие объекты домена.

3) **Слой домена**

Это слой, который содержит всю бизнес-логику, доменные службы, сущности, события и любые другие типы объектов, содержащие бизнес-логику. Это сердце системы. Службы домена содержат логику домена, которая не совсем подходит для сущности, обычно они организуют работу нескольких сущностей для выполнения некоторых действий в домене.

4) **Инфраструктура**

Слой инфраструктуры предоставляет технические возможности, которые поддерживают вышеуказанные слои, т.е. хранение данных, обмен сообщениями и т.п.

В DDD существуют артефакты для выражения, создания и получения моделей домена:

Строительные блоки DDD

Сущность (Entity)

Объект, который не определяется своими атрибутами, а скорее нитью преемственности и своей идентичностью.

Пример: Большинство авиакомпаний выделяют каждое место уникальным образом на каждом рейсе.

В этом контексте каждое место является сущностью. Однако Southwest Airlines, EasyJet и Ryanair не различают каждое место; все места одинаковы. В этом контексте место - это фактически объект значения.

Объект-Значение (Value Object)

Объект-значение - это объект, который содержит атрибуты, но не имеет концептуальной идентичности. Такие объекты рассматриваться как неизменяемые.

Пример: Когда люди обмениваются визитными карточками, они обычно не делают различий между каждой уникальной карточкой; их интересует только информация, напечатанная на карточке. В этом контексте визитные карточки являются объектами-значениями.

Агрегат (Aggregate)

Агрегат - это шаблон в доменно-ориентированном дизайне. Агрегат DDD - это кластер объектов домена, которые можно рассматривать как единое целое. Примером может быть заказ и его позиции, это будут отдельные объекты, но полезно рассматривать заказ (вместе с его позициями) как единый агрегат.

Один из объектов-компонентов агрегата будет корнем агрегата. Любые ссылки извне агрегата должны идти только в корень агрегата. Таким образом, корень может гарантировать целостность агрегата в целом.

Агрегаты являются основным элементом передачи хранилища данных - вы запрашиваете загрузку или сохранение агрегатов целиком. Транзакции не должны пересекать границы агрегатов.

Группа объектов, связанных вместе корневой сущностью: корнем агрегата. Объектам вне агрегата разрешается иметь ссылки на корень, но не на любой другой объект агрегата. Корень агрегата отвечает за проверку согласованности изменений в агрегате.

Пример: Когда вы управляете автомобилем, вам не нужно беспокоиться о том, чтобы крутить колеса, заставлять двигатель работать и т.д.; вы просто управляете автомобилем. В данном контексте автомобиль является агрегатом нескольких других объектов и служит корнем агрегата для всех остальных систем.

Агрегаты DDD иногда путают с классами коллекций (списками, картами и т. д.). Агрегаты DDD представляют собой концепции предметной области (заказ, посещение клиники, список воспроизведения), а коллекции являются общими. Агрегат часто содержит несколько коллекций вместе с простыми полями. Термин

«агрегат» является общепринятым и используется в различных контекстах (например, UML), и в этом случае он не относится к той же концепции, что и агрегат DDD.

DDD уменьшает связанность за счет использования Агрегатов. Но агрегаты, как и любые объекты должны взаимодействовать друг с другом. В DDD это взаимодействие осуществляется за счет публикации событий. В ходе жизненного цикла и изменение своего состояния агрегат может генерировать различные события, которые могут быть приняты и обработаны в другой части модели. Событийно-ориентированный подход также помогает снизить связность системы. Использование событий также можно рассматривать, как способ приведения распределенной системы к конечному согласованному состоянию.

Событие домена (Domain Event)

Объект домена, определяющий событие (то, что происходит). Событие домена - это событие, о котором заботятся эксперты домена.

Сервис (Service)

Это функции, которые не привязаны к сущностям или ценностным объектам. Грубо говоря, действие в себе.

Когда операция концептуально не принадлежит ни одному объекту. Следуя естественным контурам проблемы, вы можете реализовать эти операции в сервисах.

Репозиторий (Repository)

Это такие сервисы, которые используют глобальный интерфейс, чтобы обеспечить доступ ко всем сущностям и ценностным объектам, находящимся в конкретной группе агрегатов.

Методы извлечения объектов домена должны делегироваться специализированному объекту Repository, чтобы можно было легко заменять альтернативные реализации хранилища.

Фабрика (Factory)

Методы создания доменных объектов должны быть делегированы специализированному объекту Factory, чтобы можно было легко менять альтернативные реализации.

Этот блок шаблонов предлагает решения для проектирования и для декомпозиции, то есть разделения приложений на микросервисы.

“Programmer Driven Design”

Этот подход не подразумевает четкого процесса, когда разработчик реализует систему так, как понимает и видит он сам. Такой подход приводит к решениям, которые невозможно развивать и поддерживать. Также частью этого подхода является отсутствие какой-либо проектной документации, т.е. любая идея реализуется сразу в коде.

1.2. Подходы к разработке ПО

FDD - Feature Driven Development

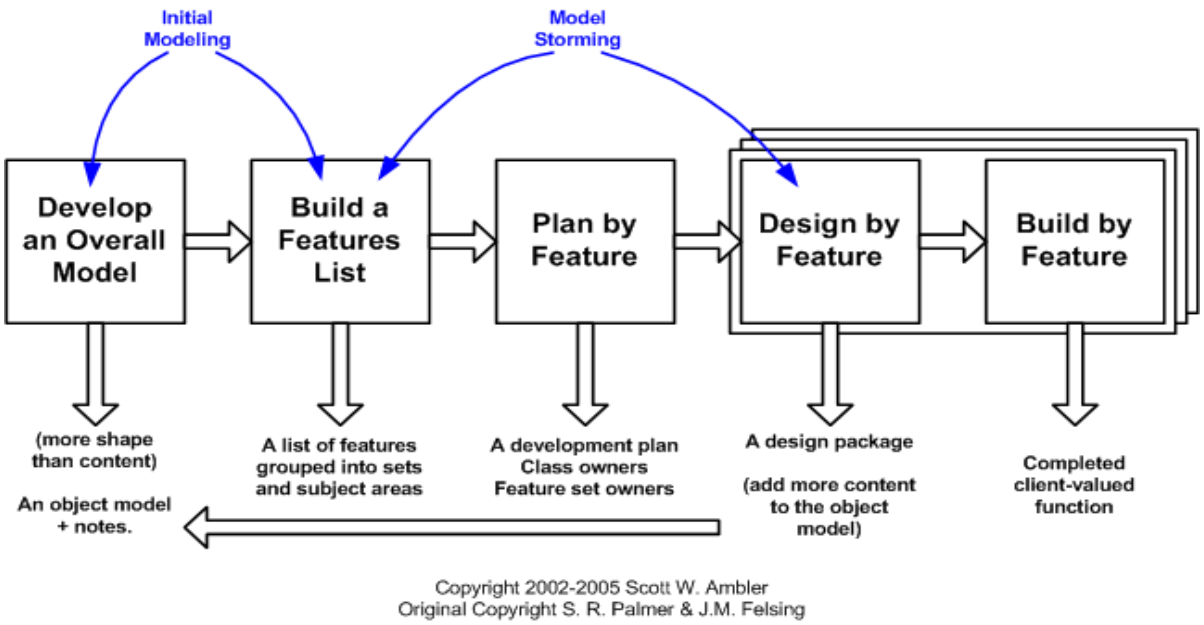
Feature-driven development (FDD) - это итеративный и инкрементный процесс разработки программного обеспечения. Основная цель - это своевременное предоставление осязаемого, рабочего программного обеспечения в соответствии с принципами, лежащими в основе Манифеста Agile.

Разработка на основе функциональности была изначально предложена Джеффом Де Люкой для проекта по разработке программного обеспечения для одного крупного сингапурского банка в 1997 году. Де Люка выделил набор из пяти процессов, охватывающий как разработку общей модели, так и ведение списка, планирование, проектирование и реализацию элементов функциональности.

Первое описание FDD появилось в 1999 году в главе 6 книги Java Modeling in Color with UML. В книге A Practical Guide to Feature-Driven Development описание FDD было обобщено, и в частности избавлено от привязок к конкретному языку программирования.

FDD включает в себя пять базовых видов деятельности:

1. Разработка общей модели информационной системы;
2. Составление списка необходимых функций системы;
3. Планирование работы над каждой функцией;
4. Проектирование функции;
5. Реализация функции.



Разработка общей модели.

Разработка начинается с высокоуровневого сквозного анализа широты решаемого круга задач и контекста системы. Далее для каждой моделируемой области делается более детальный сквозной анализ. Сквозные описания составляются в небольших группах и выносятся на дальнейшее обсуждение и экспертную оценку. Одна из предлагаемых моделей или их объединение становится моделью для конкретной области. Модели каждой области задач объединяются в общую итоговую модель, которая изменяется в ходе работы.

Создание списка функций.

Когда команда разработала общую модель, она определяет полезные для клиента функции. Осуществляется разбиением областей (доменов) на подобласти (предметные области) с точки зрения функциональности. Каждая отдельная подобласть соответствует какому-либо бизнес-процессу, шаги которого становятся списком функций (свойств). В данном случае функции — это маленькие части понимаемых пользователем функций, представленных в виде «<действие> <результат> <объект>».

Планирование работы над каждой функцией

Владение классами распределяется среди ведущих программистов путём упорядочивания и организации свойств (или наборов свойств) в классы. Необходимо иметь ресурсный план на всю команду.

Проектирование и дизайн функций

На основе **общей модели владелец продукта** выбирает группу функций, которые команда должна реализовать за определённый срок - создается проектировочный пакет.

Реализация функции

После успешного рассмотрения дизайна данная видимая клиенту функциональность реализуется до состояния готовности. Для каждого класса пишется программный код. После модульного тестирования каждого блока и проверки кода завершенная функция включается в основной проект

Плюсы	**Минусы**
- документация по свойствам системы; - тщательное проектирование; - проще оценивать небольшие задачи; - тесты ориентированы на бизнес-задачи; - проработанный процесс создания продукта; - короткие итеративные циклы разработки позволяют быстрее наращивать функциональность и уменьшить количество ошибок.	- FDD больше подходит для больших проектов. Небольшие команды разработки не смогут прочувствовать все преимущества данного подхода; - значительные затраты на внедрение и обучение.

BDD - Behaviour Driven Development

Поведенчески-ориентированная разработка (BDD) была впервые предложена и представлена Дэном Нортом.

Основной идеей данной методологии является совмещение в процессе разработки чисто технических интересов и интересов бизнеса, позволяя тем самым управляющему персоналу и программистам говорить на одном языке. Для общения между этими группами персонала используется предметно-ориентированный язык, основу которого представляют конструкции из естественного языка, понятные неспециалисту, обычно выражающие поведение программного продукта и ожидаемые результаты.

Полуформальный формат спецификации поведения требует использования ограниченного набора предложений, о которых управляющий персонал и разработчики должны предварительно договориться. Исходя из этого, фреймворки для поддержки BDD строятся по следующим принципам:

1. Некоторые части предложения могут являться входными параметрами, которые можно захватить, а остальные части могут никак не использоваться и служить только для понимания действия человеком. Обычно для такого захвата используется процессор регулярных выражений. Захваченные параметры могут быть переконвертированы и отправлены на вход конкретной исполняющей функции.
2. Каждое предложение может выражать один шаг теста.
3. Парсер может разбить спецификацию по её формальным частям, например по ключевым словам языка Gherkin. На выходе мы получаем набор предложений, каждое из которых начинается с ключевого слова.

На этом принципе строятся такие фреймворки как JBehave и RBehave, которые основаны на языке Gherkin. Некоторые фреймворки строятся по аналогии, например CBehave и Cucumber.

Gherkin - человеко-читаемый язык для описания поведения системы, который использует отступы для задания структуры документа, (пробелы или символы табуляции).

Каждая строчка начинается с одного из ключевых слов и описывает один из шагов.

Пример:

Функция: Короткое, но исчерпывающее описание требуемого функционала

Для того, чтобы достичь определенных целей

В качестве определенного участника взаимодействия с системой

Я хочу получить определенную пользу

Сценарий: Какая-то определенная бизнес-ситуация

Дано какое-то условие

И ещё одно условие

Когда предпринимается какое-то действие участником

И им делается ещё что-то

И вдобавок он совершил что-то ещё

То получается какой-то проверяемый результат

И что-то ещё случается, что мы можем проверить

Плюсы	**Минусы**
- Близость к языку бизнеса - Тесты могут быть написаны не только разработчиками, но и product owner'ми, тестерам и аналитиками - Легко читать тесты, так как они написаны на языке приближенном к естественному	- Требует большего времени на разработку тестов - Требует привлечения тестировщиков на раннем этапе разработки - Дороже чем остальные методологии разработки

TDD - Test Driven Development

Разработка через тестирование - это методология разработки, построенная на принципе сначала тест, а потом код. Изобретателем подхода считается Кент Бек

Цикл разработки по методологии TDD состоит из следующих шагов:

1. Разработка теста удовлетворяющего спецификации. Здесь основной фокус делается на требованиях.
2. Запуск теста, чтобы убедиться, что он не проходит, так как функционал удовлетворяющий требованиям ещё не написан.
3. Написание минимального кода необходимого для прохождения теста
4. Запуск теста, чтобы убедиться, что теперь тест проходит и реализация корректна
5. Рефакторинг по необходимости и переход к следующему требованию.

Плюсы	**Минусы**

- обеспечивает хорошее покрытие тестами, так тесты пишутся вместе с кодом - упрощает рефакторинг - переносит фокус внимания разработчика с кода на функционал - простой способ проверки корректности - способствует созданию чистой архитектуры	- есть задачи, которые невозможно адекватно протестировать модульными тестами (например взаимодействие между процессами) - модульные тесты обычно пишутся теми же, кто и пишет основной код, что может привести к тому, что неправильно понятая спецификация приведёт к неправильно написанному тесту. - тесты требуют поддержки, что может увеличивать время написания нового функционала и исправления дефектов

PDD - Panic Driven Development

Разработка на основе паники - это “методология” разработки, которая основана на нескольких принципах:

1. Любая новая задача появляющаяся в середине спринта имеет больший приоритет над любой запланированной работой.
2. Сначала надо решить проблему, а потом написать тесты. Лучше всего писать тесты в конце, когда основной код написан.
3. Писать нужно столько кода, чтобы решить проблему. Ошибки могут быть исправлены только кодом, нет смысла обсуждать дизайн.
4. Процесс гибок и его можно изменить, чтобы решить задачу в срок.
5. Бизнес превыше всего и поэтому не стоит тратить время на рефакторинг и технические улучшения.

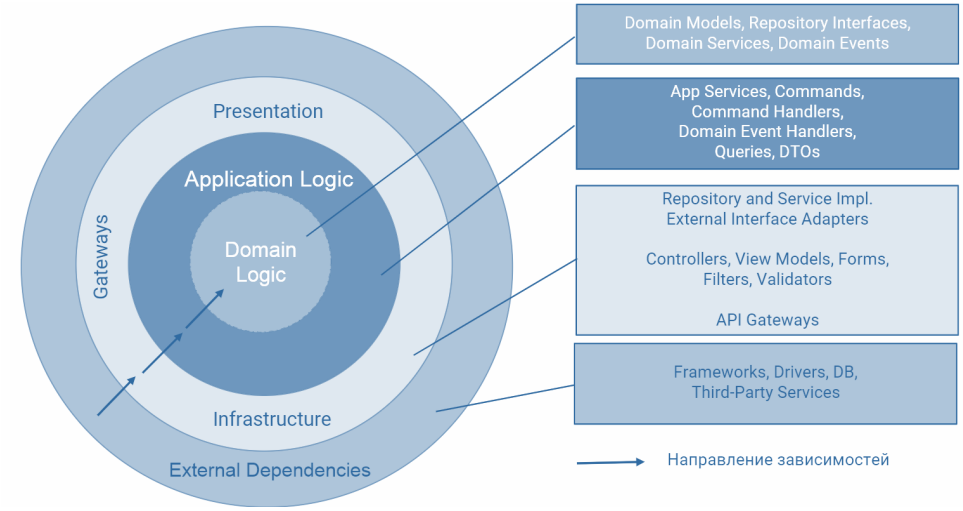
Данная “методология” скорее способ выживания команды разработчиков в сложившихся условиях.

Плюсы	**Минусы**
- высокая скорость разработки - заказчик всегда доволен “толковыми разработчиками” - дешевизна разработки	- все плюсы в итоге будут перечеркнуты техническим долгом и возросшей сложностью проекта.

2. Проектирование прикладных приложений

2.1. Чистая архитектура

Одна из проблем монолита - сильная зависимость бизнес-логики от деталей реализации. Для ее устранения мы воспользовались принципами «чистой архитектуры» (Роберт Мартин «Чистая архитектура. Искусство разработки программного обеспечения»), выделив внутри каждого модуля слои: domain, application, infrustructure.



Принципы чистой архитектуры.

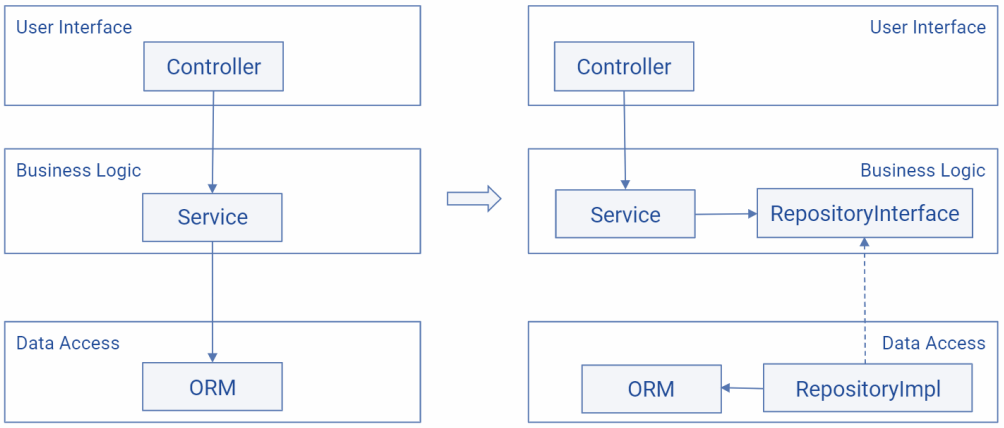
Согласно этой архитектуре, внутренний круг реализует логику предметной области, или домен. Этот слой не должен зависеть ни от чего, выходящего за его пределы. Вокруг доменной логики идет Application Logic — это классы, которые реализуют варианты использования моделей предметной области, то есть сценарии использования. Например, «добавить товар в корзину», «разместить заказ». Этот слой должен быть тонким, так как всего лишь манипулирует моделями доменного слоя и преобразует данные между доменом и внешним миром. Никакой сложной логики он содержать не должен и может зависеть только от домена. В следующем слою располагаются модели представления (view models), контроллеры, инфраструктурный код, реализации интерфейсов доменного слоя. Это более нестабильный код, поэтому внутренняя часть нашего приложения - ядро - не должна от него зависеть. На схеме стрелками показаны направления зависимостей слоев друг от друга. Это даёт нам то, что мы можем проектировать и сразу же тестировать бизнес-логику без реализации каких-то конкретных технологий.

Итак, основные принципы чистой архитектуры:

- приложение строится вокруг независимой от других слоев объектной модели;
- внутренние слои определяют интерфейсы, внешние слои содержат реализации интерфейсов;
- направление зависимостей — от внешних слоев к внутренним;
- при пересечении границ слоя данные должны преобразовываться.

Правило зависимостей (Dependency Rule) — ключевое правило. Для достижения такого направления зависимостей нам на помощь придет принцип инверсии зависимостей (**dependency inversion**). И если в традиционной трехслойной архитектуре бизнес-логика непосредственно зависела от слоя доступа к данным, то в чистой архитектуре она зависит только от интерфейса, который определен в этом же слое, а его реализация находится в слое инфраструктуры.

Таким образом бизнес-сервис не зависит от инфраструктуры. В рантайме, конечно, вместо интерфейса будет подставлена конкретная реализация, и этого можно добиться за счет механизма dependency injection, который предоставляют, наверно, все современные фреймворки.



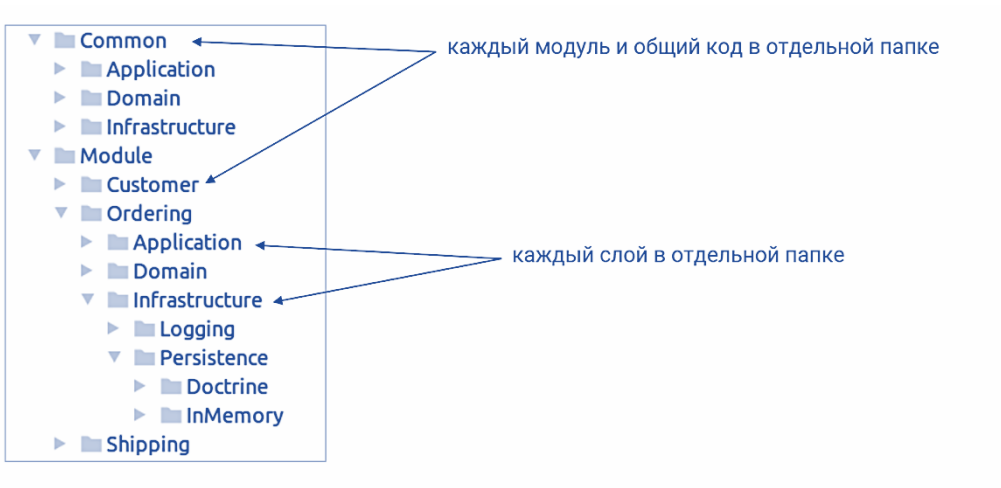
Механизм Dependency inversion.

Чистая архитектура дает независимость бизнес-слоя от:

- UI;
- Фреймворков;
- БД;
- Сторонних сервисов;

Соответственно, разработка модульных тестов сильно упрощается. Дополнительный бонус — ускорение выполнения модульных тестов, так как отсутствуют обращения к физической БД и шаги по ее инициализации тестовыми данными. Также мы получаем относительную простоту замены каких-либо реализаций.

При такой архитектуре структура проекта выглядит следующим образом:



Структура проекта в Чистой архитектуре

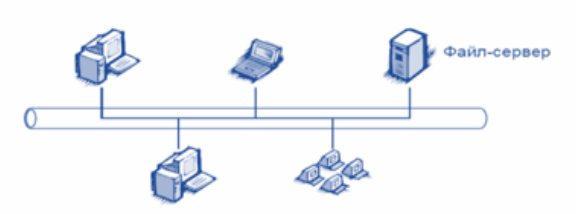
Каждый модуль выделен в отдельную папку, также как и каждый слой внутри модуля выделен в отдельную папку. Есть папка Common, там находится библиотечный код, не специфичный для какого-либо домена.

2.2. Виды архитектур прикладных приложений с точки зрения клиент-серверного взаимодействия

Файл-серверная архитектура

Все общедоступные файлы хранятся на выделенном компьютере – файл-сервере (пример архитектуры представлен на рисунке).

Файл-серверные приложения – приложения, использующие сетевой ресурс для хранения программы и данных.



Файл-серверная архитектура

Функции сервера: хранение данных и кода программы.

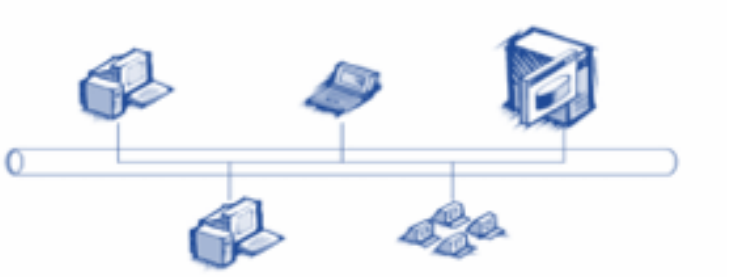
Функции клиента: обработка данных. Количество клиентов ограничено десятками.

Достоинства	**Недостатки**
• многопользовательский режим работы с данными; • удобство централизованного управления доступом; • низкая стоимость разработки.	• низкая производительность; • низкая надежность; • слабые возможности расширения.

Недостатки архитектуры с файловым сервером связаны с тем, что данные хранятся в одном месте, а обрабатываются в другом. Их нужно передавать по сети, что приводит к очень высоким нагрузкам на сеть и резкому снижению производительности приложения при увеличении числа одновременно работающих клиентов. Вторым важным недостатком такой архитектуры является децентрализованное решение проблем целостности и согласованности данных и одновременного доступа к данным. *Является морально устаревшим решением.*

Клиент-серверная архитектура

Ключевое отличие от архитектуры файл-сервер – это абстрагирование от физической схемы данных и манипулирование данными клиентскими программами на уровне логической схемы (рисунок).



Клиент-серверная архитектура

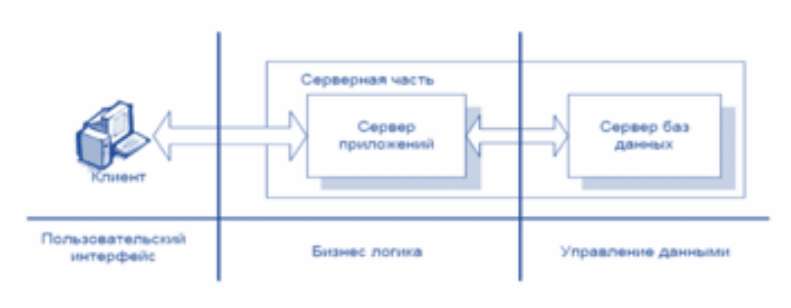
Это позволило создавать надежные многопользовательские ИС с централизованной базой данных (БД), независимые от аппаратной (часто и программной) части сервера БД и поддерживающие графический интерфейс пользователя на клиентских станциях, связанных локальной сетью.

Достоинства	Недостатки
• клиентская программа работает с данными через запросы к серверному ПО; • базовые функции приложения разделены между клиентом и сервером. • полная поддержка многопользовательской работы; • гарантия целостности данных.	• при любом изменении алгоритмов необходимо обновлять пользовательское ПО на каждом клиенте; • высокие требования к пропускной способности коммуникационных каналов с сервером; • слабая защита данных от взлома, в особенности от недобросовестных пользователей системы; • высокая сложность администрирования и настройки рабочих мест. • пользователей системы; • необходимость использовать мощные ПК на клиентских местах; • высокая сложность разработки системы из-за необходимости выполнять бизнес-логику и обеспечивать пользовательский интерфейс в одной программе.

Очевидным шагом дальнейшей эволюции архитектур ИС явилась идея "тонкого клиента": алгоритмы обработки данных разбивались на части, связанные с выполнением бизнес-функций и отображением информации в удобном для человека представлении, часть, связанная с первичной проверкой и отображением информации оставалась на клиентской машине, а вся реальная функциональность системы переносилась на серверную часть.

Трехуровневая клиент-серверная архитектура

Физическое разделение программ, отвечающих за хранение данных (СУБД) и их обработку. Такое разделение программных компонент позволяет оптимизировать нагрузки как на сетевое, так и на вычислительное оборудование комплекса (*пример представлен на рисунке 4*).

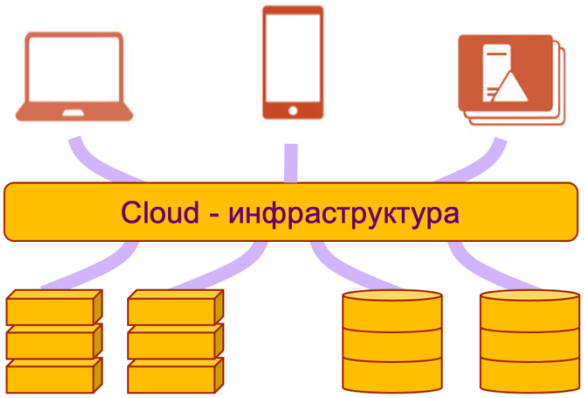


Трехуровневый клиент-сервер

Достоинства	**Недостатки**
• тонкий клиент; • между клиентской программой и сервером приложения передается лишь минимально необходимый поток данных – аргументы вызываемых функций и возвращаемые от них значения. Это теоретический предел эффективности использования линий связи; • сервер приложения ИС может быть запущен в одном или нескольких экземплярах на одном или нескольких компьютерах, что позволяет использовать вычислительные мощности организации столь эффективно и безопасно как этого пожелает администратор ИС; • дешевый трафик между сервером приложений и СУБД. Трафик между сервером приложений и СУБД может быть большим, однако это всегда трафик локальной сети, а их пропускная способность достаточно велика и дешева. В крайнем случае, всегда можно запустить СП и СУБД на одной машине, что автоматически сведет сетевой трафик к нулю; • снижение нагрузки на сервер данных по сравнению с 2.5-слойной схемой, а значит и повышение скорости работы системы в целом.	• повышенные расходы на администрирование и обслуживание серверной части.

Cloud-native архитектура - Распределённая архитектура

Cloud-native архитектура - это подход к созданию и запуску приложений, использующий преимущества облачных систем и которые были созданы специально для существования в облаке, а не в более традиционной локальной инфраструктуре.



Cloud архитектура

Облачное приложение - это набор небольших, независимых и слабо связанных между собой дискретных, многократно используемых компонентов, известных как микросервисы, которые предназначены для интеграции в любую облачную среду.

Принципы cloud-native:

- Микросервисы
- Контейнеризация
- DevOps
- Continious delivery

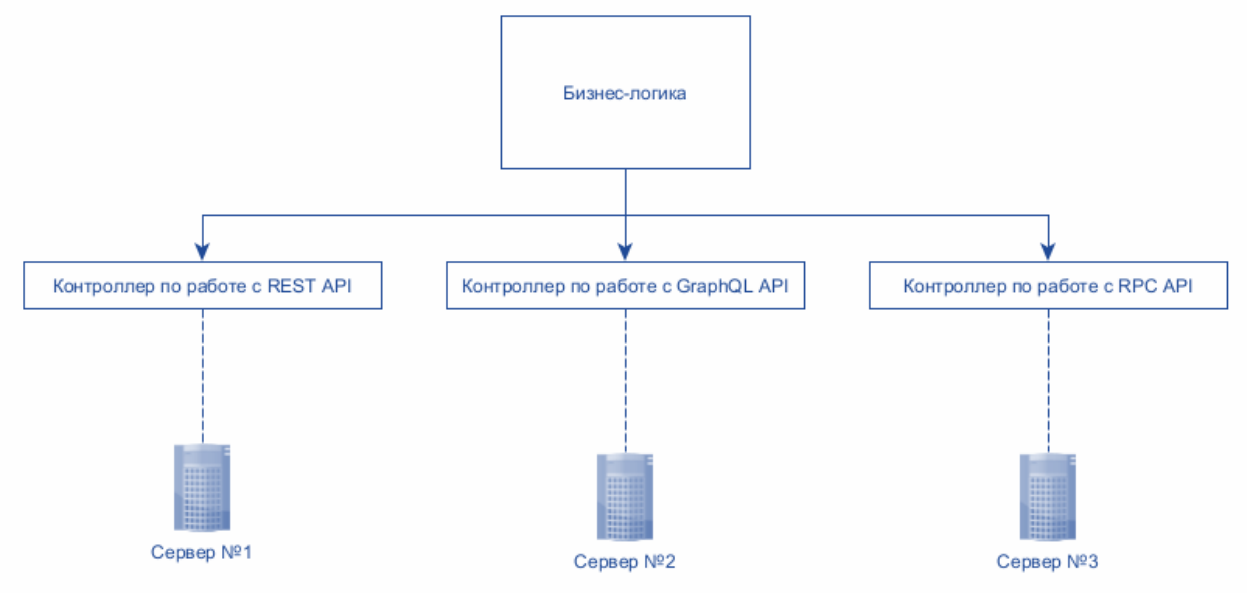
Достоинства	**Недостатки**

• Скорость работы и быстрое развертывание • Масштабирование. • Совместимость с мобильными устройствами. • Экономия бюджета • Фокус на основном бизнесе • Резервное копирование • Удобства для сотрудников • Экономия офисного пространства и единая рабочая среда	• Потеря контроля • Повреждение облачных сервисов • Закрытие Vendor Cloud • Потенциальная угроза безопасности • Сложность отладки приложений для некоторых сложно воспроизводимых состояний • Для работы с облаком требуется постоянное подключение к интернету

2.3. Архитектура приложения при использовании нескольких API

Представим себе приложение, включающее в себя базу данных, репозиторий, слой бизнес-логики и работу с API. Возможны ситуации, когда потребуется перейти на другой архитектурный стиль API или когда приложение интегрируется со сторонними API разной архитектуры. В таких случаях целесообразно отделять работу с api в разные контроллеры, причем бизнес-логика отделена от этих контроллеров. В бизнес-логику должны быть вынесены общие части при работе с API, а в контроллеры выносится логика,

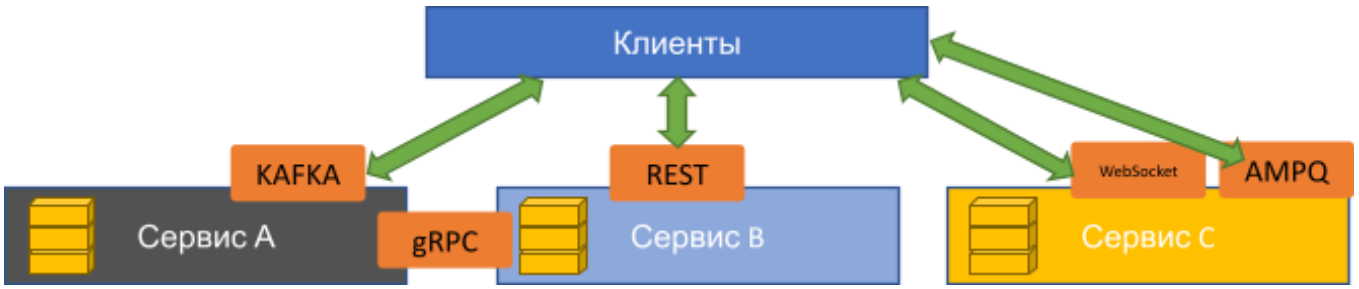
необходимая для выполнения запросов к конкретной архитектуре API.



Пример архитектуры приложения, использующее несколько API

Протокол взаимодействия или протокол передачи данных — набор определённых правил или соглашений интерфейса логического уровня, который определяет обмен данными между сторонами (например, между двумя сервисами). Эти правила задают единообразный способ передачи сообщений и обработки ошибок.

Стоит понимать, что в рамках архитектуры решения может применяться одновременно несколько подходов и протоколов по интеграции (рисунок), в зависимости от решаемых задач.



Пример интеграции на разных протоколах.

2.4. Способы организации UX/UI прикладных приложений

Обычно, пользовательские приложения предоставляют пользователю некоторый способ взаимодействия, т.е. обеспечивают пользовательский интерфейс (UX/UI).

По способу организации UX/UI можно выделить:

<table>
<tr><th>Десктопные приложения с GUI</th><th>Web-приложения</th><th>Мобильные приложения с GUI</th></tr>
<tr><td>Windows native
Mac native
Linux native
Multiplatform
Web-stack</td><td>MPA (Multi Page Application) / Тонкий клиент
MPA + Ajax (легаси-подход)
SPA (Single Page Application) / RIA / Толстый клиент
Изоморфные приложения (PWA)</td><td>Android native
IOS native
Multiplatform
Web-stack</td></tr>
<tr><td>Приложения для IoT, носимой техники, умных устройств, Smart TV</td><td>Бот-приложения</td><td>Public API</td></tr>
<tr><td rowspan="2">Linux native
Android native
IOS native
Multiplatform
Web-stack</td><td rowspan="2">Мессенджеры
Бот-платформы
SuperApp</td><td>CLI (Command line interface)-интерфейс</td></tr>
<tr><td>Голосовой помощник</td></tr>
</table>

Несмотря на кажущееся многообразие, архитектурно практически все эти варианты имеют много общего, что и будет рассмотрено в дальнейшем.

3. Паттерны проектирования прикладных приложений

3.1 Паттерны связывания компонент и уровней приложения

API Gateway

В маленьких проектах можно реализовать прямое обращение от клиента к сервису. Однако в приложениях корпоративного масштаба с большим числом микросервисов рекомендуется использовать шаблон API Gateway.

Этот паттерн основан на применении шлюза, который находится между клиентским приложением и микросервисами.

API-шлюз выступает в роли единой точки входа. Он отвечает за прием запросов от клиентов, вызов необходимых микросервисов и отправку результатов клиенту. Благодаря API-шлюзу клиент посылает только один запрос. Эта модель предлагает несколько преимуществ, перечисленных ниже:

- внутренняя сложность приложения скрыта от клиента, что упрощает реализацию клиентского кода;
- увеличение гибкости изменения, объединения, разделения или удаления микросервисов;
- уменьшение трафика между клиентом и приложением и увеличение за счет этого эффективности.

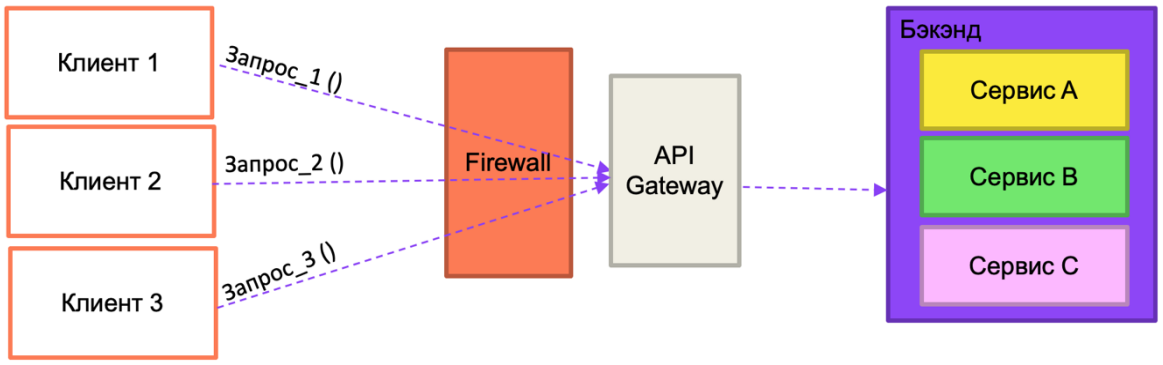
Кроме того, API-шлюз может служить точкой входа для балансировки нагрузки, аутентификации, мониторинга и управления. Он может предоставлять разные API разным версиям клиента, а также распознавать приоритеты запросов.

Самый большой недостаток этой модели состоит в том, что API-шлюз может стать единой точкой отказа и узким местом, причем как с точки зрения производительности, так и с точки зрения разработки. API-шлюз должен проектироваться, настраиваться и обслуживаться несколькими группами разработчиков, а значит, этот процесс должен быть простым и эффективным. Например, он должен обновляться одновременно с изменением, добавлением или удалением микро- сервисов. С точки зрения эксплуатации, эластичная подсистема балансировки нагрузки должна гарантировать соответствие характеристик производительности и доступности.

В зависимости от конкретной цели использования паттерна иногда выделяют следующие его разновидности:

- **Gateway Routing.** Шлюз используется как обратный Proxy, перенаправляющий запросы клиента на соответствующий сервис.
- **Gateway Aggregation.** Шлюз используется для разветвления клиентского запроса на несколько микросервисов и возвращения агрегированных ответов клиенту.
- **Gateway Offloading.** Шлюз решает сквозные задачи, которые являются общими для сервисов: аутентификация, авторизация, SSL, ведение журналов и так далее.

Применение паттерна сокращает число вызовов, обеспечивает независимость клиента от протоколов, используемых в сервисах: REST, AMQP, gRPC и так далее, обеспечивает централизованное управление сквозной функциональностью. Однако шлюз может стать единой точкой отказа, требует тщательного мониторинга и при отсутствии масштабирования бывает узким местом системы.



Паттерн API Gateway

Реализация Edge функций

Основными обязанностями API-gateway являются маршрутизация и объединение API, но он способен взять на себя также реализацию граничных функций. Граничная функция, как понятно из названия, — это операция обработки запросов на границе приложения.

Можно привести следующие примеры:

- аутентификация — проверка подлинности клиента, который делает запрос;
- авторизация — проверка того, что клиенту позволено выполнять определенную операцию;

- ограничение частоты запросов — контроль над тем, сколько запросов в секунду могут выполнять определенный клиент и/или все клиенты вместе;
- кэширование — кэширование ответов для снижения количества запросов к сервисам;
- сбор показателей — сбор показателей использования API для анализа, связанного с биллингом;
- ведение журнала (логов) запросов — запись запросов в журнал.

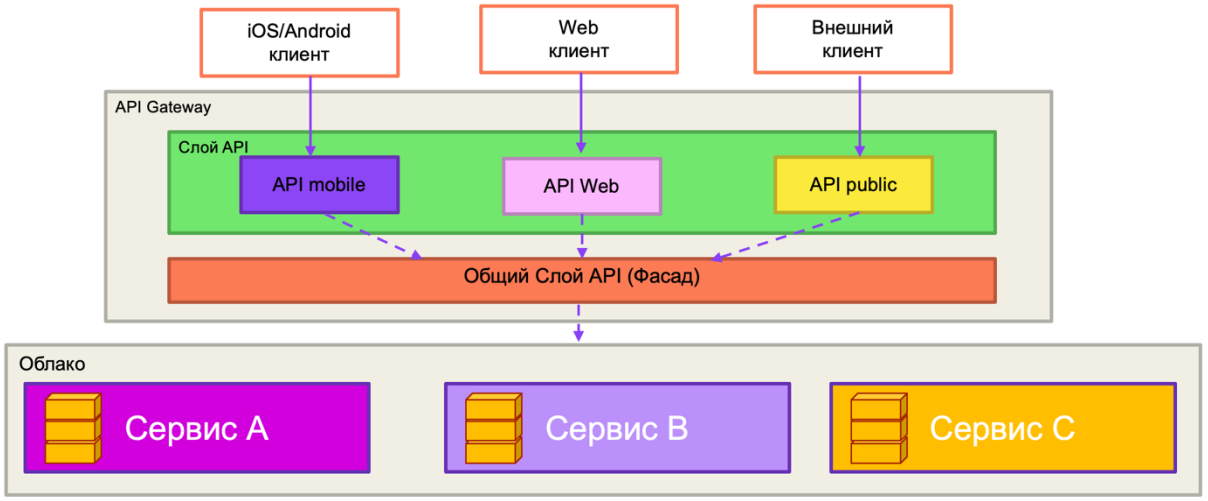
Общая архитектура API Gateway

API Gateway имеет многоуровневую модульную архитектуру. Его архитектура, показанная на рисунке 14.3, состоит из двух слоев: слоя API и общего слоя. Уровень API состоит из одного или нескольких независимых модулей API. Каждый модуль API реализует API для конкретного клиента.

Общий уровень реализует общую функциональность, включает Edge функции, такие как аутентификация.

В данном примере шлюз API имеет три модуля API:

- API для мобильных устройств - реализует API для мобильного клиента;
- API web (браузера) - реализует API для приложения JavaScript, запущенного в браузере;
- Публичный API - API для сторонних разработчиков.



Слои API Gateway архитектуры

Реализация API – gateway с помощью схем на GraphQL

Спецификация GraphQL API построена вокруг концепции *схемы* (schema) — набора типов, которые определяют структуру модели серверных данных и операции, доступные для клиента, например запросы.

Преимущества	**Недостатки**
• Внутренняя сложность приложения скрыта от клиента, что упрощает реализацию клиентского кода; • Увеличение гибкости изменения, объединения, разделения или удаления микросервисов; • Уменьшение трафика между клиентом и приложением и увеличение за счет этого эффективности; • Api-шлюз может служить точкой входа для балансировки нагрузки, аутентификации, мониторинга и управления; • Может предоставлять разные API разным версиям клиента; • Может распознавать приоритеты запросов.	• API-шлюз может стать единой точкой отказа и узким местом, причем как с точки зрения производительности, так и с точки зрения разработки; • API-шлюз должен проектироваться, настраиваться и обслуживаться несколькими группами разработчиков, а значит, этот процесс должен быть простым и эффективным. Например, он должен обновляться одновременно с изменением, добавлением или удалением микро- сервисов; • С точки зрения эксплуатации, эластичная подсистема балансировки нагрузки должна гарантировать соответствие характеристик производительности и доступности.

«Бэкенды для фронтендов» (Backends for Frontends, BFF)

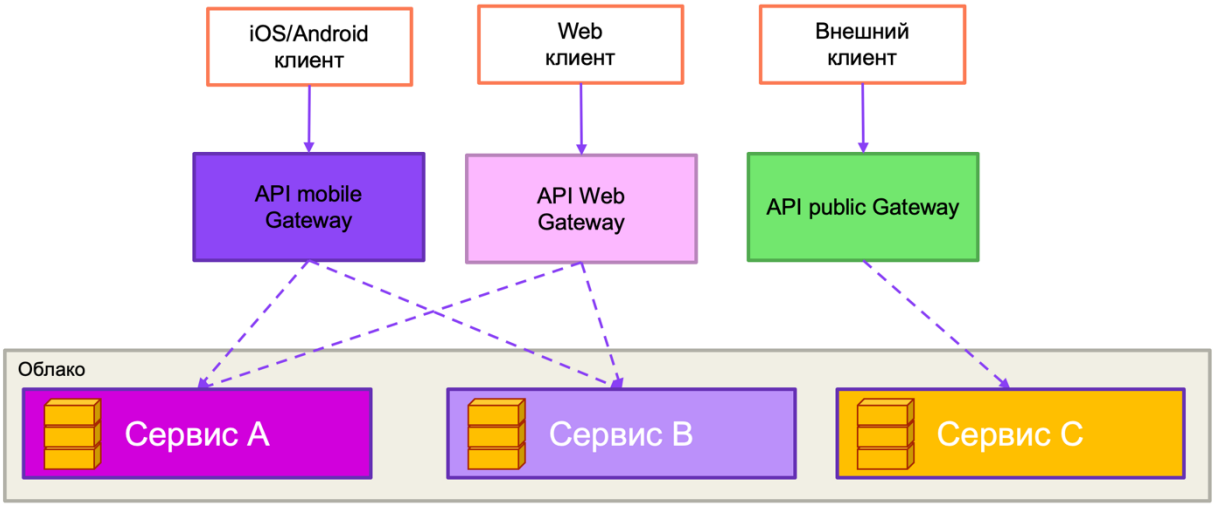
Этот паттерн является вариантом реализации шаблона API Gateway. Он также обеспечивает дополнительный уровень между микросервисами и клиентами, но вместо одной точки входа вводит несколько шлюзов для каждого типа клиента: Web, Mobile, Desktop и так далее.

В типичном цикле разработки продукта бэкенд-инженеры отвечают за создание сервисов, взаимодействующих с хранилищами данных, а фронтенд-инженеры разрабатывают пользовательские интерфейсы. Современные приложения должны быть построены с учетом как мобильных устройств, так и компьютеров.

Несмотря на то, что разрыв между мобильными и стационарными устройствами с точки зрения аппаратного обеспечения становится все меньше, доступ и использование по-прежнему представляют собой сложные задачи для мобильных устройств.

В таких случаях может оказаться полезным шаблон Backend-for-Frontend. Здесь ожидается, что вы создаете/настраиваете сервисы бэкенда под конкретный внешний интерфейс.

С помощью паттерна можно добавить API, адаптированные к потребностям каждого клиента, избавившись от хранения большого количества ненужных настроек в одном месте. Но шаблон не стоит применять в тех случаях, когда разница в требованиях к API у разных типов клиентов незначительна либо приложение само по себе небольшое: это приведет лишь к дублированию кода и увеличению числа компонентов.



Паттерн Backends for Frontends

Для оптимизации работы мобильных клиентов можно создать отдельную серверную службу, которая предоставляет легкие и разбитые на страницы ответы.

Можно воспользоваться данным шаблоном для агрегации различных сервисов, чтобы уменьшить лишний обмен сообщениями.

Примечание. Если вы пользуетесь API-шлюзом, то сегодня шаблон BFF легко поддается реализации непосредственно в самом шлюзе, так что вам не придется поддерживать отдельные службы.

Когда использовать этот шаблон:

- Когда вы хотите предоставлять продукт/услугу для разных клиентов (мобильных и стационарных устройств);
- Когда вы хотите оптимизировать ответ для конкретного типа клиента;
- Когда вы хотите уменьшить обмен сообщениями между мобильными клиентами и различными сервисами.

Когда не следует использовать этот шаблон:

- Когда ожидается, что пользовательский интерфейс будет единым для всех;
- Когда ожидается, что мобильные и компьютерные приложения будут демонстрировать аналогичную информацию и предоставлять аналогичную функциональность.

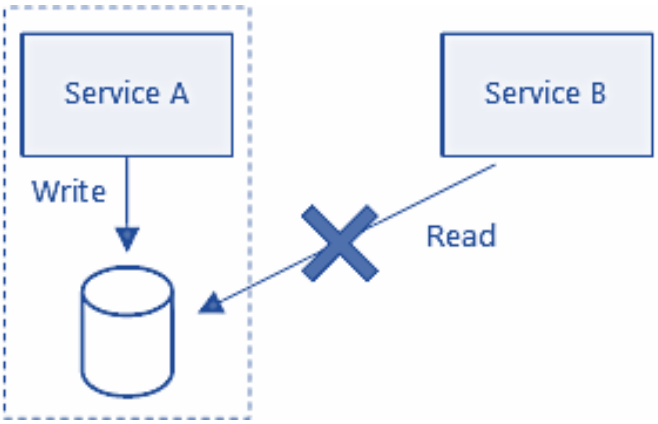
3.2 Паттерны управления данными

Основной принцип работы приложения с множеством сервисов (модулей) заключается в том, что каждый сервис управляет собственными данными.

Два сервиса не могут использовать одно и то же хранилище данных (рисунок). Наоборот, каждый сервис отвечает за собственное хранилище данных, которое напрямую недоступно для других сервисов.

Это позволяет устранить непреднамеренную взаимозависимость между сервисами, которая может возникнуть, когда сервисы совместно используют одни и те же базовые схемы данных.

Если в схему данных вносятся изменения, их нужно согласовать в каждом сервисе, которая зависит от этой базы данных. Изолируя хранилище данных каждого сервиса, мы можем ограничить область изменений и сохранить гибкость полностью независимых развертываний. Кроме того, у каждого микросервиса могут быть собственные модели данных, запросы или шаблоны операций чтения и записи. При использовании общего хранилища данных для каждой команды ограничивается возможность оптимизировать хранилище данных для какого-либо сервиса.



Взаимодействие сервисов при обращении к данным

Этот подход естественным образом приводит к Polyglot Persistence — использованию нескольких технологий хранения данных в одном приложении. Для работы одной службы могут требоваться возможности схемы при чтении, свойственные базе данных документов. Для другой — целостность данных, предоставляемая реляционной СУБД. Каждая команда может выбрать подходящий вариант для своей службы.

Эти паттерны описывают возможные варианты взаимодействия микросервисов с базами данных.

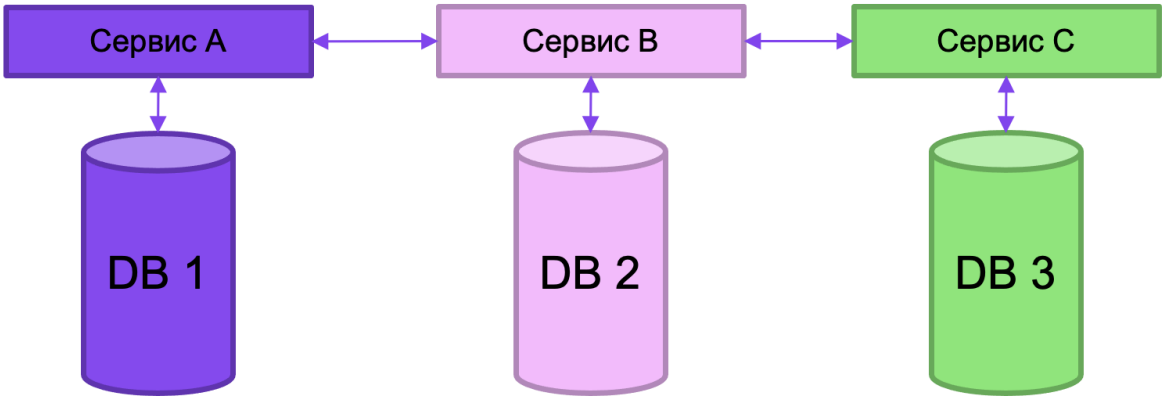
«База данных на сервис» (Database Per Service)

Основная рекомендация при переходе на микросервисы — предоставить каждому сервису собственное хранилище данных, чтобы не было сильных зависимостей на уровне данных. При этом имеется в виду именно логическое разделение данных, то есть микросервисы могут совместно использовать одну и ту же физическую базу данных, но в ней они должны взаимодействовать с отдельной схемой, коллекцией или таблицей.

Основанный на этих принципах паттерн Database Per Service повышает автономность микросервисов и уменьшает связь между командами, разрабатывающими отдельные сервисы.

У паттерна есть и недостатки: усложняется обмен данными между сервисами и предоставление транзакционных гарантий ACID. Паттерн не стоит применять в небольших приложениях — он предназначен для крупномасштабных проектов с большим числом микросервисов, где каждой команде требуется полное владение ресурсами для повышения скорости разработки и лучшего

масштабирования.



Паттерн Database Per Service

Паттерну Database Per Service часто противопоставляют другой шаблон — **Shared Database («Разделяемая база данных»)**. По сути, он представляет собой антипаттерн и подразумевает использование одного хранилища данных несколькими микросервисами. Его допускается использовать на начальных стадиях миграции на микросервисную архитектуру или в очень небольших приложениях, разрабатываемых одной командой (2–3 микросервиса).

Кэширование

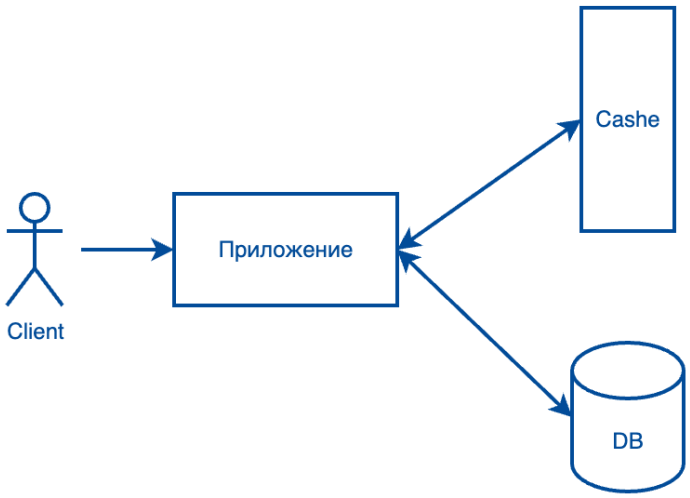
В сфере вычислительной обработки данных кэш – это высокоскоростной уровень хранения, на котором требуемый набор данных, как правило, временного характера. Доступ к данным на этом уровне осуществляется значительно быстрее, чем к основному месту их хранения. С помощью кэширования становится возможным эффективное повторное использование ранее полученных или вычисленных данных.

Кеширование — это способ оптимизации работы приложения, при котором повторно запрашиваемый контент сохраняется и используется для обслуживания последующих запросов. Механизм, с помощью которого можно повысить скорость работы приложения за счёт переноса часто используемых данных в очень быстрое хранилище.

Кэширование позволяет увеличивать производительность приложений за счёт использования сохранённых ранее данных, вроде ответов на сетевые запросы или результатов вычислений. В любом приложении встречаются медленные операции (SQL запросы или запросы к внешним API), результаты которых можно сохранить на некоторое время. Это позволит выполнять меньше таких операций, а большинству пользователей показывать заранее сохраненные

данные. Кэширование является частью стратегии доставки основного контента, реализованной в протоколе HTTP. Компоненты на всем пути доставки могут кэшировать элементы для ускорения обработки последующих запросов с учетом объявленных политик.

Существует множество различных типов кэширования, каждый из которых имеет свои плюсы и минусы. Кэш приложения и кэш памяти могут ускорять ответы на определенные запросы.



Данные в кэше обычно хранятся на устройстве с быстрым доступом, таком как ОЗУ (оперативное запоминающее устройство), и могут использоваться совместно с программными компонентами. Основная функция кэша – ускорение процесса извлечения данных. Он избавляет от необходимости обращаться к менее скоростному базовому уровню хранения.

Небольшой объем памяти кэша компенсируется высокой скоростью доступа. В кэше обычно хранится только требуемый набор данных, причем временно, в отличие от баз данных, где данные обычно хранятся полностью и постоянно.

О быстродействии жёстких дисков и оперативной памяти

Разница между временным хранением данных в оперативной памяти и постоянным хранением на жёстком диске проявляется в скорости работы с информацией, в стоимости носителей и в близости их к процессору.

Время отклика оперативной памяти составляет десятки наносекунд, в то время как жёсткому диску нужны десятки миллисекунд. Разница в быстродействии дисков и памяти составляет шесть порядков!

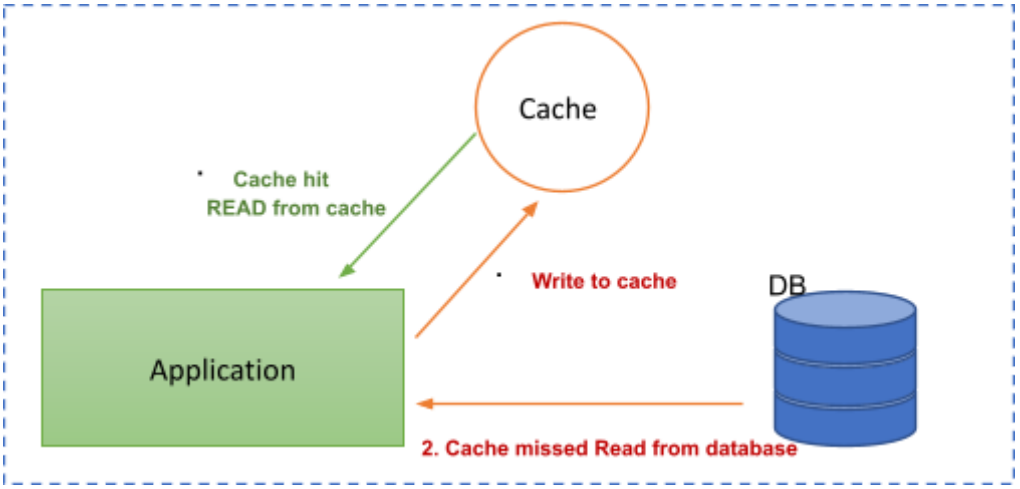
$$1\ Millisecond = 1000000\ Nanoseconds$$

Одна миллисекунда равна миллиону наносекунд

Паттерн кэширования Cache Aside

Это самый простой способ кэширования, зачастую множество фреймворков уже имеют встроенную реализацию.

В такой системе данные лениво загружаются в кэш. Пользователь делает запрос к нашей системе, после этого приложение сначала идет в кэш, если данные в нем есть, то возвращает их клиенту иначе идет в базу данных, обновляет кэш и отдает пользователю.



Cache-Aside предназначен для непредсказуемого спроса на данные, когда неизвестно, сколько раз запросят одни и те же данные.

1. Определяет, есть ли данные в кэше.
2. Если элемент отсутствует в кэше, читаем запись из хранилища данных.
3. Помещаем копию записи в кэш.

Плюсы	**Минусы**
• Отлично подходит для тяжелых операций чтения. • Обычно система устойчива к отказу кэша. Если	• Из-за того что запись идет напрямую в базу данных, данные в кэше могут стать не консистентными. Для этого нужно использовать TTL (время жизни данных в кэше) или инвалидировать кэш, когда

он упадет всегда можно пойти напрямую в базу данных. ● Возможно использовать разные структуры данных для кэша и базы данных. Особенно полезно когда нам нужно закешировать результат какой-то сложной выборки.	нужно гарантированно отдавать актуальные данные.

Области применения.

Кэш используется на разных технологических уровнях, включая операционные системы, сетевые уровни, в том числе сети доставки контента (CDN) и DNS, интернет-приложения и базы данных. С помощью кэширования можно значительно сократить задержки и повысить производительность операций ввода-вывода в секунду для многих рабочих нагрузок приложений с большой нагрузкой на чтение, например порталов для вопросов и ответов, игровых ресурсов, порталов для распространения мультимедиа и социальных сетей.

Кэшировать можно результаты запросов к базам данных, вычислений, которые требовательны к ресурсам, запросы к API и ответы на них, а также веб-артефакты, например файлы HTML, JavaScript и изображений. Рабочие нагрузки, требующие больших вычислительных мощностей для обработки наборов данных, например сервисы рекомендаций и высокопроизводительное вычислительное моделирование, тоже могут эффективно использовать уровень данных в памяти в качестве кэша.

В этих приложениях можно обращаться к очень большим наборам данных в режиме реального времени через кластеры машин, которые охватывают сотни узлов. Управление этими данными в дисковом хранилище является узким местом таких приложений из-за низкой скорости работы базового оборудования.

Кэширование очень активно используют во множестве систем. Например:

- Внутренний кэш баз данных.
- DNS кэш внутри нашего компьютера или браузера.
- Кэш статического контента в браузере.

- CDN также является своего рода кэшем.

В любом приложении есть операции, которые долго выполнялись, но результат которых можно сохранить на какое-то время. Это позволит меньше выполнять таких операций и отдавать заранее сохранённые данные.

Что можно кэшировать?

- Запросы в базу данных.
- Пользовательские сессии.
- Медленные операции внутри приложения (расчеты, какие-то итоговые данные и т. д.)
- Запросы к внешним системам.

Виды кэшируемых данных

Кэшировать нужно данные, которые медленно генерируются и часто запрашиваются:

- Результаты запросов к внешним сервисам (RSS, SOAP, REST и т.п.);
- Результаты медленных выборок из базы данных;
- Сгенерированные html блоки либо целые страницы;

Пример Кэширование в веб-приложениях

Необходимо понимать, что работу с данными можно производить как на стороне клиента, так и на сервере. Притом, серверная обработка данных централизована и имеет ряд несомненных преимуществ (особенно для службы поддержки).

Разумное кэширование контента – один из наиболее эффективных способов улучшить пользовательский опыт сайта или ресурса.

Веб-кэширование является основной конструктивной особенностью протокола HTTP, предназначенного для минимизации сетевого трафика и улучшения отзывчивости системы в целом. Контент кэшируется на каждом уровне от исходного сервера до браузера.

Веб-кеширование работает путем кэширования HTTP-ответов для запросов в соответствии с определенными правилами. Последующие запросы кэшированного контента затем можно извлечь из ближайшего кэша, а не отправлять запрос обратно на веб-сервер.

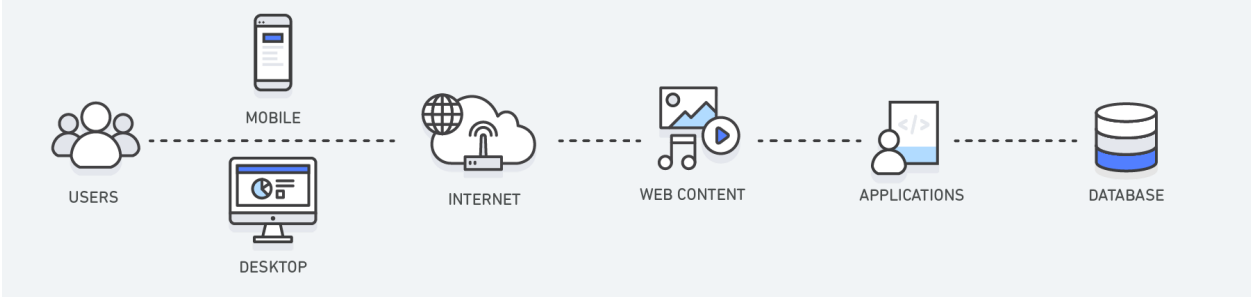
В самом распространенном типе кэширования, кэшировании веб-браузера, веб-браузер хранит копии статических данных на локальном жестком диске. Используя кеширование, веб-браузер может избежать многократных циклов приема-передачи на сервер и вместо этого получать доступ к тем же данным локально, что экономит время и ресурсы. Кэширование хорошо подходит для локального управления небольшим объемом статических данных, таких как статические изображения, файлы CSS и JavaScript.

Веб-кэш — это компонент, способный временно хранить HTTP-ответы, которые могут быть использованы для последующих HTTP-запросов при соблюдении определенных условий.

Веб-кэширование — это ключевая архитектурная особенность HTTP-протокола. Оно предназначено для снижения сетевого трафика во время увеличения предполагаемой ответной реакции в целом. Кэши находятся на каждом этапе перемещения контента — от исходного сервера к браузеру.

Если говорить простым языком, веб-кэширование позволяет повторно использовать HTTP-ответы, которые были сохранены в кэше, с HTTP-запросами аналогичного характера. Давайте рассмотрим простой пример, когда пользователь запрашивает с сервера определенный тип продукта (книги). Можно предположить, что весь этот процесс займет примерно 670 миллисекунд. Если чуть позже в тот же день пользователь выполнит этот же запрос вместо того, чтобы снова повторить те же вычисления и потратить 670 миллисекунд, HTTP –ответ может вернуться к пользователю. Это значительно сократит время отклика. В реальности это может составить менее 50 миллисекунд.

Примеры распределения кэширования в вэб-приложении



Уровень	Клиентские	DNS	Интернет	Приложение	База данных

Пример использования	Ускорение получения веб-контента от веб-сайтов (браузеры или устройства)	Определение IP-адреса для домена	Ускорение получения веб-контента от серверов веб-приложений Управление веб-сеансами (на стороне сервера)	Повышение производительности приложений и ускорение доступа к данным	Сокращение задержек, связанных с запросами к базе данных
Технологии	Управление кэшированием с помощью HTTP-заголовков (браузеры)	Серверы DNS	Управление кэшированием с помощью HTTP-заголовков, CDN, обратные прокси-серверы, веб-ускорители, хранилища пар «ключ – значение»	Хранилища пар «ключ – значение», локальные кэши	Буферы баз данных, хранилища пар «ключ – значение»
Примеры Решений	Для браузеров	Amazon Route 53	Amazon CloudFront, ElastiCache для Redis, ElastiCache для Memcached	Инфраструктуры приложений, ElastiCache для Redis, ElastiCache для Memcached	ElastiCache для Redis, ElastiCache для Memcached

Преимущества кэширования

Кэширование повышает скорость извлечения данных и сокращает расходы при работе в больших масштабах. Что достигается использованием ОЗУ и работающих в памяти сервисов, которые обеспечивают высокие показатели скорости обработки запросов, или IOPS (количество операций ввода-вывода в секунду).

Чтобы обеспечить аналогичный масштаб работы с помощью традиционных баз данных и оборудования на базе жестких дисков, требуются дополнительные ресурсы. Использование этих ресурсов приводит к повышению расходов, но все равно не позволяет достигнуть такой низкой задержки, какую обеспечивает кэш в памяти.

Благодаря кэшу, при очередном обращении клиента за одними и теми же данными, сервер может обслуживать запросы быстрее. Кэширование — эффективный архитектурный паттерн, так как большинство программ часто

обращаются к одним и тем же данным и инструкциям. Эта технология присутствует на всех уровнях вычислительных систем. Кэши есть у процессоров, жёстких дисков, серверов, браузеров.

Повышение производительности приложений

Поскольку память работает в разы быстрее диска (магнитного или SSD), чтение данных из кэша в памяти производится крайне быстро (за доли миллисекунды). Это значительно ускоряет доступ к данным и повышает общую производительность приложения. В том числе повышенная производительность на том же оборудовании.

Сокращение затрат на базы данных

Один инстанс кэша может обрабатывать тысячи операций ввода-вывода в секунду, потенциально заменяя несколько инстансов базы данных, что в результате дает снижение общих затрат. Это особенно важно, если плата взимается за пропускную способность базы данных. В таких случаях можно снизить затраты на десятки процентов.

Снижение нагрузки на серверную часть

Благодаря освобождению серверной базы данных от значительной части нагрузки на чтение, которая направляется на уровень памяти, кэширование может сократить нагрузку на базу данных и защитить ее от снижения производительности под нагрузкой и даже от сбоев при пиковых нагрузках.

Прогнозируемая производительность

Общей проблемой современных приложений является обработка пиков в использовании приложений. Примерами могут служить социальные сети во время Суперкубка или в день выборов, веб-сайты электронной коммерции в Черную пятницу и т. д. Повышенная нагрузка на базу данных приводит к повышению задержек при получении данных, и общая производительность приложения становится непредсказуемой. Эту проблему можно решить благодаря использованию кэша в памяти с высокой пропускной способностью.

Устранение проблемных мест в базах данных

Во многих приложениях небольшое подмножество данных, например профиль знаменитости или популярный продукт, может оказаться намного более востребованным, чем остальные данные. Это приводит к появлению проблемных мест в базе данных и требует избыточного выделения ее ресурсов, чтобы удовлетворить спрос на пропускную способность, которой достаточно для получения наиболее часто используемых данных. За счет хранения общих

ключей в кэше в памяти можно избавиться от необходимости избыточного выделения ресурсов и обеспечить быструю и предсказуемую работу системы при обращении к самым востребованным данным.

Повышение пропускной способности операций чтения (количество операций ввода-вывода в секунду)

Помимо сокращения задержек, системы в памяти обеспечивают намного более высокую скорость выполнения запросов (количество операций ввода-вывода в секунду) по сравнению с базами данных на диске. Один инстанс, который используется как распределенный дополнительный кэш, может обслуживать сотни тысяч запросов в секунду.

Эффективное кэширование помогает как пользователям, так и контент-провайдерам:

- Снижение сетевых затрат. Контент можно кэшировать в разных точках сетевого пути между потребителем и источником контента. Когда контент кэшируется ближе к потребителю, запросы не будут требовать значительной сетевой активности за пределами кэша.
- Повышение отзывчивости - Улучшенный отклик. Кэширование позволяет получать контент быстрее, потому что нет необходимости снова проделывать путь по всей сети. Кэши, поддерживаемые рядом с пользователем, например кэш браузера, могут сделать обслуживание запроса почти мгновенным.
- Повышенная производительность. Контент-провайдер может использовать мощные серверы в пути доставки, чтобы взять на себя основную нагрузку на обслуживание контента.
- Доступность контента во время сбоев сети. При использовании определенных политик кэширование может обслуживать контент пользователям даже в течение коротких сбоев.

Доступность контента даже при сетевых сбоях

При использовании определенных стратегий кэширования в случае сбоя сервера контент будет передан конечным пользователям из кэша за небольшой период времени . Так они смогут выполнить основные задачи без сбоя.

Недостатки кэширования

Кэш удаляется во время перезапуска сервера

Каждый раз, когда сервер перезапускается, кэш-данные также удаляются. Это происходит потому, что кэш нестабилен и может исчезнуть при потере источника питания. Однако вы можете придерживаться стратегий, в которых вы регулярно записываете кэш на свой диск, чтобы сохранить кэшированные данные даже во время перезагрузки сервера.

Обслуживание устаревших данных

Одна из основных проблем кэширования — это обслуживание устаревших данных. Устаревшие данные — это необновленные сведения, которые содержат предыдущую версию данных. Если вы кэшировали запрос продуктов, но в то же время, менеджер удалил четыре продукта, пользователи получат списки продуктов, которые не существуют. Такое положение сложно выявить и исправить.

Нарушение конфиденциальности данных

Необходимо использовать дополнительные методы защиты при сборе кэша в оперативной памяти.

Паттерн Repository (Репозиторий)

Система со сложной доменной моделью часто выигрывает от наличия слоя, подобного тому, что предоставляет Data Mapper, который изолирует объекты домена от деталей кода доступа к базе данных. В таких системах может быть целесообразно построить еще один слой абстракции над слоем отображения, где сосредоточен код построения запросов. Это становится более важным при наличии большого количества доменных классов или при интенсивном выполнении запросов. В таких случаях добавление этого слоя помогает минимизировать дублирование логики запросов.

Репозиторий является посредником между доменом и слоями отображения данных, действуя как коллекция объектов домена в памяти. Клиентские объекты создают спецификации запросов в декларативном виде и передают их в хранилище для удовлетворения. Объекты могут быть добавлены в хранилище и удалены из него, как и из простой коллекции объектов, а код отображения, инкапсулированный в хранилище, будет выполнять соответствующие операции за сценой. Концептуально, Repository инкапсулирует набор объектов, хранящихся в хранилище данных, и операции, выполняемые над ними, обеспечивая более объектно-ориентированное представление уровня персистентности. Repository также поддерживает цель достижения чистого разделения и односторонней зависимости между слоями домена и отображения данных.

По сути, он обеспечивает абстракцию данных, так что ваше приложение может работать с простой абстракцией, интерфейс которой приближен к интерфейсу коллекции. Добавление, удаление, обновление и выбор элементов из этой коллекции осуществляется с помощью ряда простых методов, без необходимости иметь дело с такими проблемами базы данных, как соединения, команды, курсоры или считыватели.

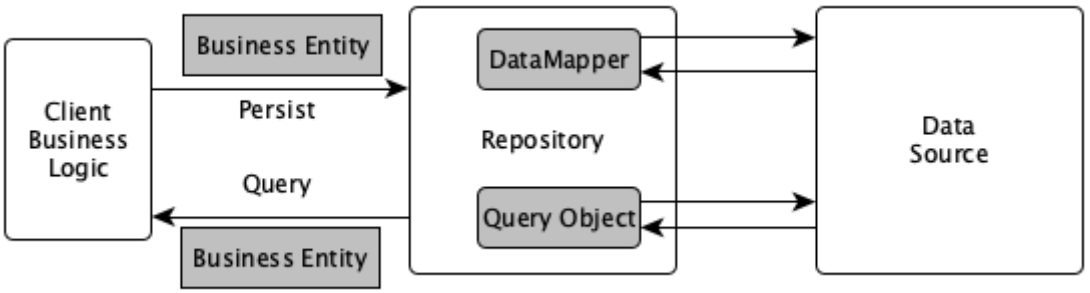
Использование этого паттерна может помочь достичь свободной связи и сохранить персистентность доменных объектов. Хотя этот паттерн очень популярен (а может быть, именно поэтому), его также часто неправильно понимают и используют. Существует множество различных способов реализации паттерна Repository. Рассмотрим несколько из них, их достоинства и недостатки.

Шаблон репозитория - это шаблон проектирования, который служит посредником при передаче данных от и к уровням домена и доступа к данным.

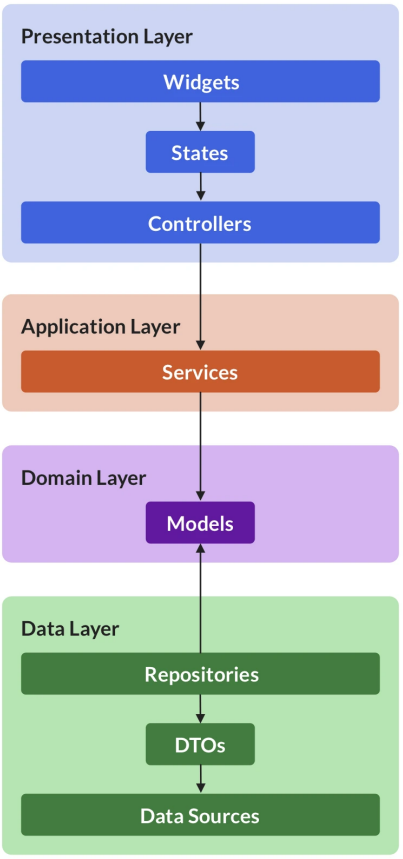
Репозитории - это классы, которые скрывают логику, необходимую для хранения или получения данных. Таким образом, наше приложение не будет заботиться о том, какой ORM мы используем, поскольку все, что связано с ORM, обрабатывается в слое репозитория. Это позволяет иметь более чистое разделение задач.

Одним из наиболее часто используемых паттернов при работе с данными является паттерн Репозиторий. Репозиторий позволяет абстрагироваться от конкретных подключений к источникам данных, с которыми работает программа, и является промежуточным звеном между классами, непосредственно взаимодействующими с данными, и остальной программой.

Репозиторий – это фасад для доступа к базе данных. На рисунке изображена схема паттерна Репозиторий.



Пример схемы паттерна репозиторий



Вариант паттерна Репозиторий

Разделение ответственности запросов и команд на обработку данных (CQRS)

CQRS (**Command Query Responsibility Segregation)** — очень полезный шаблон для современных приложений, взаимодействующих с хранилищами данных. Он основан на принципе разделения операций чтения (запрос) и записи/обновления (команда) в хранилище данных.

Шаблон проектирования CQRS нацелен на разделение операций чтения и записи распределенной системы для повышения масштабируемости и безопасности. Эта модель применяет команды для записи данных в постоянное хранилище и запросы для обнаружения и извлечения данных.

Их обработкой занимается центр управления, получающий запросы от пользователей. После этого он извлекает данные, вносит необходимые изменения, сохраняет их и уведомляет службу чтения, которая затем обновляет модель чтения для демонстрации изменения пользователю.

Предположим, вы создаете приложение, которое требует хранить данные в такой базе, как MySQL/PostgreSQL и т.п. Как известно, при записи данных в хранилище операция должна выполняться в несколько шагов, например

проверка, моделирование и сохранение. Следовательно типичные операции записи/обновления занимают больше времени, чем простые операции чтения.

Когда вы используете единичное хранилище данных одновременно для операций чтения и записи, к тому же в масштабе, у вас могут начаться проблемы с производительностью.

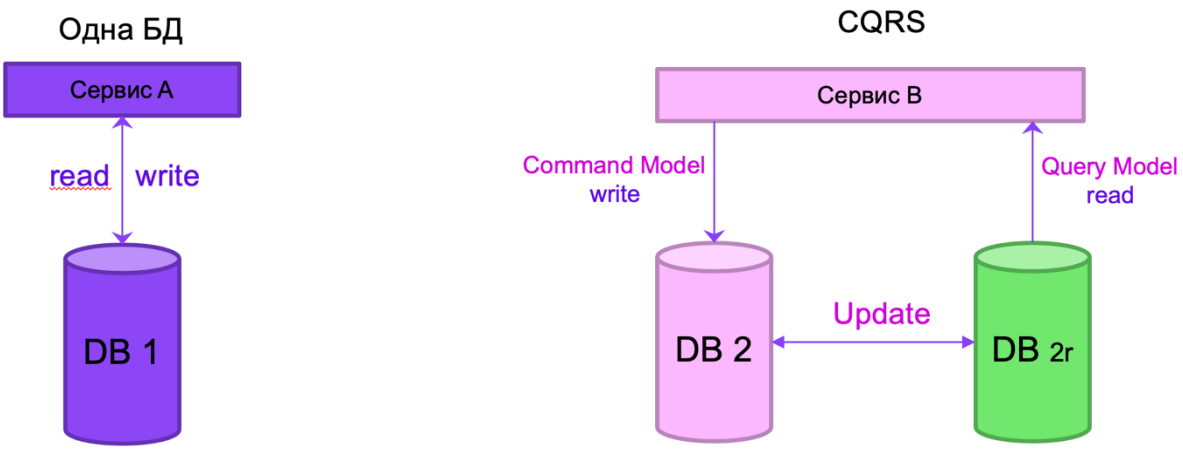
CQRS предполагает использование различных моделей данных для операций чтения и записи. Некоторые варианты также предполагают использование отдельных хранилищ данных для этих моделей.

CQRS - это просто создание двух объектов там, где раньше был только один. Разделение происходит на основе того, являются ли методы командой или запросом.

Команда - это любой метод, который изменяет состояние, а **запрос** - любой метод, который возвращает значение.

Как правило, это означает, что каждый из объектов будет использовать различное представление данных, чтобы соответствовать цели. На обычном языке здесь принято говорить о "модели записи" и "модели чтения". Обычно изменения сначала вносятся в модель записи, а затем асинхронно распространяются на модель чтения.

Таким образом, в модели CQRS мы обычно используем журнал в качестве модели сохранения, а затем из этого журнала создаем подходящие для запросов представления объекта и кэшируем эти представления так, чтобы запросы могли поддерживаться быстро (принимая компромисс, что представление, используемое в запросе, не всегда будет отражать абсолютно последнюю доступную информацию).



Паттерн CQRS

Преимущества	**Недостатки**
• Уменьшает сложность системы посредством делегирования задач; • Обеспечивает четкое разделение между бизнес-логикой и валидацией; • Помогает классифицировать процессы по выполняемой работе; • Уменьшает число непредвиденных изменений общих данных; • Уменьшает число сущностей, имеющих доступ к данным с правом изменения.	• Требует поддержания постоянного двустороннего взаимодействия между моделями команд и чтения; • Может привести к увеличению времени ожидания при отправке запросов с высокой пропускной способностью; • Отсутствуют средства для взаимодействия между сервисными процессами.

Когда использовать этот шаблон

CQRS лучше всего подходит для информационно ёмких приложений, таких как системы управления БД SQL или NoSQL, а также для микросервисных

архитектур с большими объемами данных. CQRS оптимален для приложений, сохраняющих состояния, поскольку разделение записывающих и считывающих компонентов помогает в работе с неизменяемыми состояниями.

- При масштабировании приложения, когда ожидается огромное количество операций чтения и записи;
- Если вы хотите отдельно настроить производительность для операций чтения и записи;
- Когда операции чтения происходят практически в реальном времени или же последовательно.

Когда не следует использовать этот шаблон

- Когда вы создаете обычное приложение CRUD, в котором не ожидается огромного количества одновременных операций чтения и записи.

Примечание

В настоящее время большинство баз данных PaaS предоставляют возможность создавать *реплики для чтения* (Google Cloud SQL, Azure SQL DB, Amazon RDS и т.д.) для хранилищ данных, которые значительно упрощают репликацию данных.

Многие корпоративные базы данных также предоставляют эту возможность, если вы имеете дело с локальными БД.

Сейчас также есть тенденция реализовывать реплики чтения как быстрые и производительные базы данных NoSQL, к примеру MongoDB и Elasticsearch.

3.3 Паттерны управления состоянием приложения

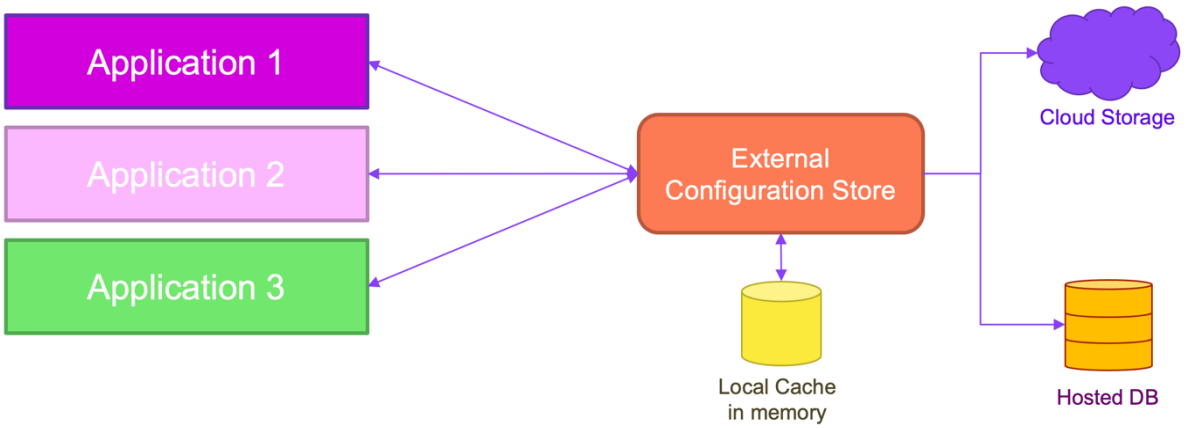
Внешняя конфигурация (External Configuration)

Практически все приложения во время работы используют разнообразные конфигурационные параметры: адреса служб, строки подключения к базам данных, учетные данные, пути к сертификатам и так далее. При этом параметры будут отличаться в зависимости от среды выполнения: Dev, Stage, Prod и так далее.

Хранить конфигурации локально — в файлах, развертываемых вместе с приложением, — считается очень плохой практикой, особенно при переходе на микросервисы. Это приводит к серьезным рискам безопасности и требует повторного развертывания при каждом изменении конфигурационных параметров.

Поэтому в приложениях корпоративного уровня рекомендуется использовать шаблон External Configuration, предлагающий хранить все конфигурации во внешнем хранилище. В качестве такого хранилища может выступать облачная служба хранения, база данных или другая система.

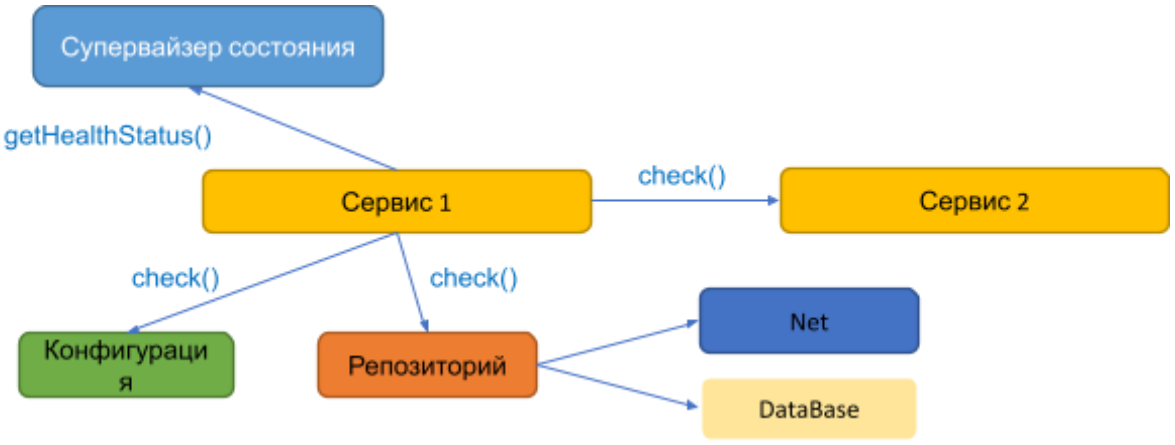
В результате применения шаблона процесс сборки будет отделен от среды выполнения, а риски безопасности будут сведены к минимуму, так как конфигурации для производственной среды перестанут являться частью кодовой базы.



Паттерн External Configuration

Самодиагностика (Self-checking)

Самопроверка сервисом своего состояния и готовности окружения. Проверяется полнота конфигурации, подключения к источникам данных и окружению. После подтверждения выдаётся статус готовности и сервис переходит в рабочий режим. Проводится обычно в момент запуска сервиса.



«Аккуратное отключение» (Graceful shutdown)

Мы можем говорить об изящном (аккуратном) завершении работы нашего приложения, когда все ресурсы, которые оно использовало, и весь трафик и/или обработка данных, которые оно обрабатывало, закрыты и освобождены должным образом.

Graceful Shutdown аккуратно выключает сервер, не прерывая активных соединений. GS работает, сначала закрывая все открытые слушатели, затем закрывая все неработающие соединения, а затем ожидая неопределенное время, пока соединения не вернутся в режим ожидания, после чего выключается. Если предоставленный контекст истекает до завершения выключения, GS возвращает ошибку контекста, в противном случае возвращается любая ошибка, полученная при закрытии основного слушателя (слушателей) сервера.

Это означает, что ни одно соединение с базой данных не остается открытым и ни один текущий запрос не срабатывает, потому что мы останавливаем наше приложение.

Возможные сценарии для аккуратного завершения работы веб-сервера:

- Приложение получает уведомление об остановке;
- Приложение сообщает балансировщику нагрузки, что оно не готово к новым запросам;
- Приложение обслужило все текущие запросы;
- Приложение корректно освобождает все ресурсы: БД, очередь и т.д.;
- Приложение завершается с кодом состояния "success" (process.exit()).

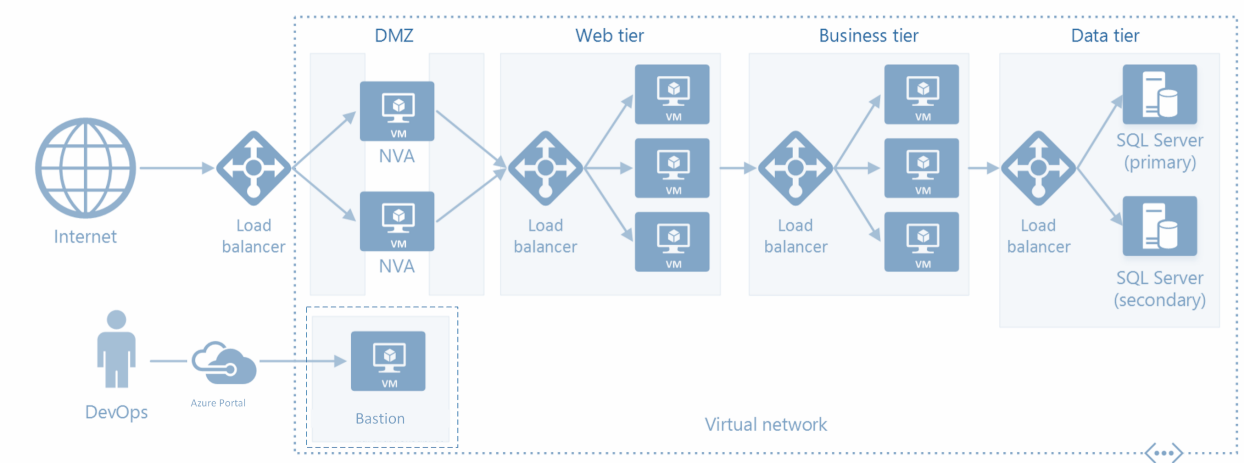
Почему это важно?

Если мы не останавливаем наше приложение правильно, мы тратим впустую ресурсы, такие как соединения с БД, а также можем нарушить текущие запросы. HTTP-запрос не восстанавливается автоматически - если мы не смогли его обслужить, значит, мы его просто пропустили.

3.4. Паттерны структурирования приложений

N-уровневый стиль архитектуры

В n-уровневой архитектуре приложение разделяется на логические слои и физические уровни.



Паттерн N-уровневая архитектура

Слои — это способ распределения ответственности и управления зависимостями. Каждый слой несет определенную ответственность. В более высоком слое могут использоваться службы из более низкого слоя, но не наоборот.

Уровни разделяются физически путем запуска на разных компьютерах. С одного уровня можно отправлять вызовы непосредственно на другой уровень или использовать асинхронный обмен сообщениями (очередь сообщений). Каждый слой можно разместить на отдельном уровне, но это не обязательно. Вы можете разместить несколько слоев на одном уровне. Физическое разделение уровней улучшает масштабируемость и устойчивость, но также приводит к увеличению задержки из-за дополнительных операций сетевого взаимодействия.

Традиционное трехуровневое приложение содержит уровень представления, средний уровень и уровень базы данных. Средний уровень является необязательным. В более сложных приложениях может быть больше трех уровней. На схеме выше показано приложение с двумя средними уровнями, которые заключают в себе разные области функций.

N-уровневое приложение может иметь **архитектуру с закрытыми слоями** или **архитектуру с открытыми слоями**:

- В архитектуре с закрытыми слоями из слоя могут отправляться вызовы только к слою, расположенному непосредственно под ним.
- В архитектуре с открытыми слоями из слоя могут отправляться вызовы к любому из нижних слоев.

- В архитектуре с закрытыми слоями зависимости между слоями ограничены. Но когда из одного слоя просто передаются запросы к следующему слою, может создаваться лишний сетевой трафик.

Когда следует использовать эту архитектуру

Как правило, n-уровневые архитектуры реализуются в виде приложений IaaS (инфраструктура как услуга), в которых каждый уровень выполняется в отдельном наборе виртуальных машин. Тем не менее n-уровневое приложение не обязательно должно представлять собой "чистую" модель IaaS. Часто целесообразно использовать управляемые службы для реализации некоторых элементов архитектуры, в частности кэширования, обмена сообщениями и хранения данных.

Используйте n-уровневую архитектуру для следующих сценариев:

- простые веб-приложения;
- перенос локального приложения в Azure с минимальным рефакторингом;
- унифицированная разработка локальных и облачных приложений.

N-уровневые архитектуры очень распространены в традиционных локальных приложениях, поэтому они естественным образом подходят для переноса существующих рабочих нагрузок в облако.

Особенности

- N-уровневые архитектуры не ограничиваются тремя уровнями. Как правило, для более сложных приложений используется больше уровней. В этом случае используйте маршрутизацию по протоколу седьмого уровня, чтобы направлять запросы к определенному уровню;
- Уровни соответствуют границам масштабируемости, надежности и безопасности. По возможности используйте отдельные уровни для служб с разными требованиями в этих областях;
- Используйте масштабируемые наборы виртуальных машин для автомасштабирования;
- Найдите места в архитектуре, где можно использовать управляемые сервисы без значительного рефакторинга. В частности, обратите внимание на кэширование, обмен сообщениями, хранилище и базы данных;
- Для более надежной защиты разместите сеть периметра перед приложением. Промежуточная подсеть содержит виртуальные сетевые

модули (network virtual appliances - NVA), в которых реализованы функции безопасности, например брандмауэры и проверка пакетов;

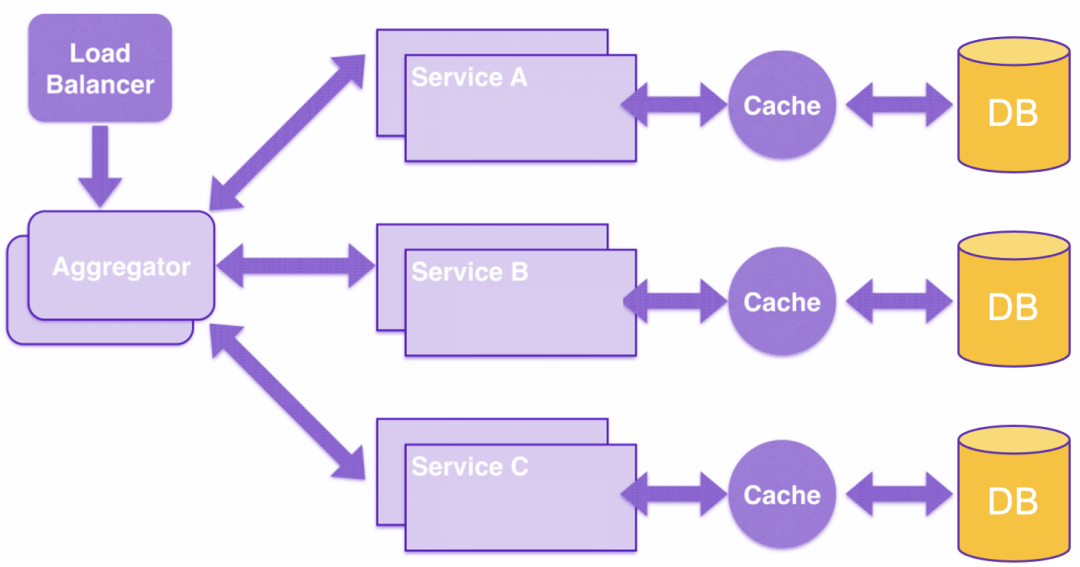
- Чтобы обеспечить высокий уровень доступности, разместите два или несколько модулей NVA в группе доступности с внешней подсистемой балансировки нагрузки для распределения запросов из Интернета по экземплярам;
- Запретите прямой доступ по протоколу RDP или SSH к виртуальным машинам, на которых выполняется код приложения. Вместо этого операторы должны будут входить на переходной узел, который также называется узлом-бастионом. Это виртуальная машина в сети, которую администраторы используют для подключения к другим виртуальным машинам. Она имеет группу безопасности сети, которая разрешает RDP или SSH только от утвержденных общедоступных IP-адресов;
- Вы можете расширить связь между виртуальной сетью облака и локальной сетью, используя виртуальную частную сеть "сеть — сеть";
- Если для управления удостоверениями ваша организация использует Active Directory, вы можете расширить среду Active Directory в виртуальную сеть облака;
- Если требуется более высокий уровень доступности, чем обеспечивает соглашение об уровне обслуживания облака для виртуальных машин, реплицируйте приложение в два региона и используйте диспетчер трафика для отработки отказа.

Преимущества	**Недостатки**
• Возможность переноса между облаком и локальной средой, а также между облачными платформами. • Быстрый процесс обучения для большинства разработчиков. • Естественный процесс перехода от	• Можно легко остановиться на среднем уровне, на котором просто выполняются операции CRUD в базе данных, что добавляет дополнительную задержку без выполнения требуемых задач. • Монолитные конструкции препятствуют независимому

традиционной модели приложений. ● Открытость для разнородных сред (Windows или Linux).	развертыванию компонентов. ● На управление приложением IaaS требуется больше усилий, чем на управление приложением, в котором используются только управляемые службы. ● Управление сетевой безопасностью в больших системах может оказаться сложной задачей.

Агрегатор (Aggregator)

В простейшей форме агрегатор представляет собой обычную веб-страницу, вызывающую множество сервисов для реализации функционала, требуемого в приложении. Поскольку все сервисы (Service A, Service B и Service C) предоставляются при помощи легковесного REST-механизма, веб-страница может извлечь данные и обработать/отобразить их как нужно. Если требуется какая-либо обработка, например, применить бизнес-логику к данным, полученным от отдельных сервисов, то для этого у вас может быть CDI-компонент, преобразующий данные таким образом, чтобы их можно было вывести на веб-странице.



Паттерн Агрегатор

Агрегатор может использоваться и в тех случаях, когда не требуется ничего отображать, а нужен лишь более высокоуровневый составной микросервис, который могут потреблять другие сервисы. В данном случае агрегатор просто соберет данные от всех отдельных микросервисов, применит к ним бизнес-логику, а далее опубликует микросервис как конечную точку REST. В таком случае, при необходимости, его смогут потреблять другие нуждающиеся в нем сервисы.

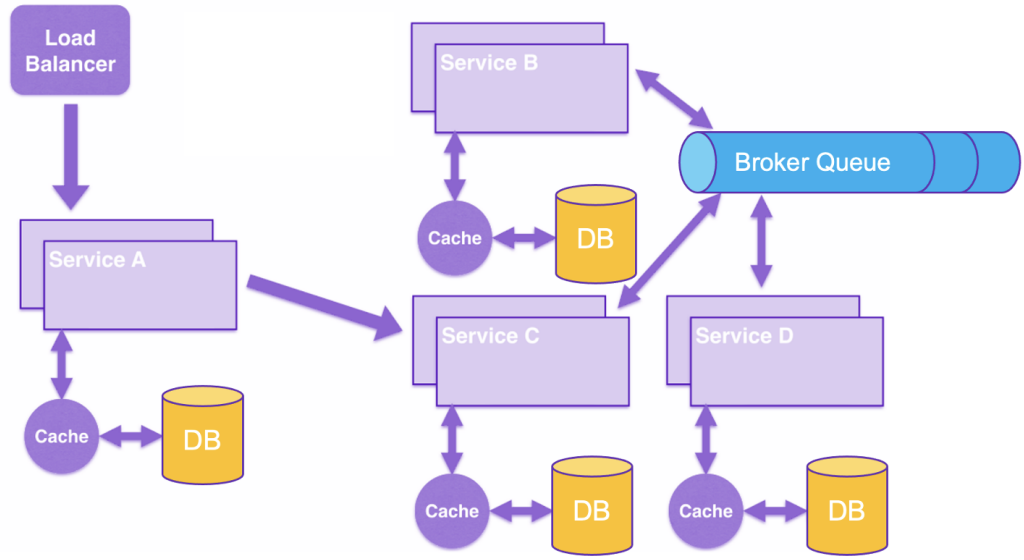
Этот паттерн следует принципу DRY. Если существует множество сервисов, которые должны обращаться к сервисам A, B и C, то рекомендуется абстрагировать эту логику в составной микросервис и агрегировать ее в виде отдельного сервиса. Преимущество абстрагирования на этом уровне заключается в том, что отдельные сервисы, скажем, A, B и C, могут развиваться независимо, а бизнес-логику будет по-прежнему выполнять составной микросервис.

Обратите внимание: каждый отдельный микросервис (опционально) имеет собственные уровни кэширования и базы данных. Если агрегатор – это составной микросервис, то и у него могут быть такие уровни.

Агрегатор также может независимо масштабироваться как по горизонтали, так и по вертикали. То есть, если речь идет о веб-странице, то к ней можно прикрутить дополнительные веб-серверы.

Асинхронные сообщения (Asynchronous Messaging)

При всей распространенности и понятности паттерна REST, у него есть важное ограничение, а именно: он синхронный и, следовательно, блокирующий. Обеспечить асинхронность можно, но это делается по-своему в каждом приложении. Поэтому в некоторых микросервисных архитектурах могут использоваться очереди сообщений, а не модель REST запрос/отклик.



Паттерн Асинхронные сообщения

В этом паттерне сервис A может синхронно вызывать сервис C, который затем будет асинхронно связываться с сервисами B и B при помощи разделяемой очереди сообщений. Коммуникация Service A -> Service C может быть асинхронной, скажем, с использованием веб-сокетов; так достигается желаемая масштабируемость.

Комбинация модели REST запрос/отклик и обмена сообщениями публикатор/подписчик также могут использоваться для достижения поставленных целей.

3.5. Разрушающие изменения API - Breaking Changes API

Разрушающее (ломающие) изменение API - это любое изменение, которое может нарушить работу клиентского приложения. Обычно разрывные изменения связаны с модификацией или удалением существующих частей API.

В последнем случае, при удалении, неизбежны сбои в работе приложений. Если клиент потребляет удаленный ресурс, поле или структуру, часть его приложения перестанет функционировать. Степень, в которой это действительно "ломает" приложение, может сильно варьироваться: от незначительного

косметического эффекта до полной непригодности приложения; независимо от этого, удаление все равно считается ломающим изменением.

Модификация с меньшей вероятностью приведет к поломке приложения. Даже если клиент использует ресурс, поле или конечную точку, которые были изменены, есть шанс, что его приложение продолжит работать как обычно, в зависимости от реализации. Например, если API мультимедиа переходит от возврата JPEG к PNG, приложения, сохраняющие файлы этих типов, скорее всего, продолжат работать.

Конечно, модификация все еще несет в себе реальный риск поломки клиентских приложений. Поскольку ответственный владелец API не оставляет такие вещи на волю случая, модификации также следует рассматривать как ломающие изменения.

Общие примеры ломающих изменений:

- Удаление ресурса или метода
- Удаление поля ответа
- Изменение URI ресурса или метода
- Изменение имени поля
- Изменение требуемых параметров запроса
- Изменение авторизации
- Изменение ограничения скорости

Что не является ломающим изменением?

В большинстве случаев изменение, которое добавляет API, не является разрушающим изменением.

Этот тип изменений, называемый аддитивным, может включать новые ресурсы или методы, новые поля ответа и даже новые параметры запроса. Поскольку эти изменения не влияют на существующие потоки, они не должны ломать приложения - но об этом подробнее чуть позже.

Общие примеры не ломающих изменений:

- Добавление ресурса или метода
- Добавление поля ответа
- Добавление необязательных параметров запроса

Список ломающих изменений:

1. Удаление end-point
2. Удаление поля из ответа ("поле" в данном случае - это пара ключ/значение JSON)
3. Удаление опции из поля в запросе - если поле запроса ранее могло принимать два значения String, а мы удаляем возможность принимать одно из них, это будет разарушающим изменением.
4. Добавление нового обязательного поля, аргумента, заголовка или параметра в запрос
   переименование поля
5. Изменение порядка аргумента
6. Удаление фильтра, функции, переменной или тега
7. Сокращение количества вызовов функции на странице (активити)
8. Изменение типа данных поля
9. Изменение длины поля сверх стандартных промышленных ограничений (int, long и т.д.)
10. Изменение кода или категории ответа на ошибку
11. Добавление зависимости к модулю, сборке
12. Изменение типов авторизации
13. Ограничение области видимости
14. Уменьшение лимитов вызова API
15. Повышение строгости проверки
16. Изменение порядка сортировки по умолчанию
17. Значительное увеличение времени ответа (10x)с
18. Изменения конфигурации

Данный список является обобщённым, так как он зависит от особенностей решения в которое привносятся изменения.

Глоссарий

Data Transfer Object (DTO) — один из шаблонов проектирования, используется для передачи данных между подсистемами приложения. Data Transfer Object, в отличие от Business Object или Data Access Object не должен содержать какого-либо поведения.

Object-Relational Mapping (ORM) – технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных». Существуют как проприетарные, так и свободные реализации этой технологии.

Архитектура программного обеспечения — совокупность важнейших решений об организации программной системы. Понятия «архитектура» и «дизайн» программного обеспечения зачастую тождественны.

Распределенные вычислительные системы — это физические компьютерные, а также программные системы, реализующие тем или иным способом параллельную обработку данных на многих вычислительных узлах.

Сервис-ориентированная архитектура (*COA*, англ. *service-oriented architecture- SOA*) — модульный подход к разработке программного обеспечения, базирующийся на обеспечении удаленного по стандартизированным протоколам использования распределенных, слабо связанных легко заменяемых компонентов (сервисов) со стандартизированными интерфейсами.

Персентиль (или перцентиль или процентиль) — методика измерения в статистике, которая показывает процент значений измеряемой метрики, который находится ниже значения персентиля. Например, если говорить о времени ответа системы, 99й персентиль на отметке 100 миллисекунд говорит о том, что 99% измеряемых запросов выполнились за 100 миллисекунд и менее

Согласованность в конечном счёте (англ. *eventual consistency*) — одна из моделей согласованности, используемая в распределённых системах для достижения высокой доступности, в рамках которой гарантируется, что в отсутствии изменений данных, через какой-то промежуток времени после последнего обновления («в конечном счёте») все запросы будут возвращать последнее обновлённое значение.

Конечный автомат — математическая абстракция, модель дискретного устройства, имеющего один вход, один выход и в каждый момент времени находящегося в одном состоянии из множества возможных.

Алгоритм — это последовательность инструкций для решения проблемы. При разработке алгоритма мы должны учитывать архитектуру компьютера, на котором будет выполняться алгоритм.

Параллельные вычислительные системы — это физические компьютерные, а также программные системы, реализующие тем или иным способом параллельную обработку данных на многих вычислительных узлах.

Последовательный алгоритм — алгоритм, в котором некоторые последовательные шаги инструкций выполняются в хронологическом порядке для решения проблемы.

Параллельный алгоритм. Задача делится на подзадачи и выполняется параллельно для получения отдельных выходных данных. Позже, эти отдельные выходы объединяются, чтобы получить конечный желаемый результат.

DRY - Don’t repeat yourself (**DRY**; с англ. — «не повторяйся») — это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоёв абстрагирования. Принцип DRY формулируется как: «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы». Он был сформулирован Энди Хантом и Дэйвом Томасом в их книге «Программист-прагматик».

Дополнительные материалы

1. Статья "Service Oriented Architecture (SOA)"
2. Статья "Architectural Patterns and Styles"
3. Статья "Understanding Service-Oriented Architecture"
4. Перечень интеграционных паттернов https://www.enterpriseintegrationpatterns.com/patterns/messaging/
5. Полный технический обзор приложения NET StockTrader 6.1 Sample Application -
6. CAP теорема - https://www.bigdataschool.ru/wiki/cap
7. Приложение .NET StockTrader 6.1 Sample Application - https://docs.microsoft.com/en-us/previous-versions/msdn10/bb499684(v=msdn.10)
8. Статья "Service Oriented Architecture (SOA)"
9. Статья "Architectural Patterns and Styles"
10. Статья "Understanding Service-Oriented Architecture"
11. Thomas Erl “Service-Oriented Architecture. Analysis and Design for Services and Microservices”
12. Mark Richards, Neal Ford "Fundamentals of Software Architecture. An Engineering Approach"
13. Chris Richardson “Microservices Patterns”

Используемые источники

Книги для чтения

1. Мартин Р. Чистый код: создание, анализ и рефакторинг. Москва : Юпитер, 2018. – 464 с.
2. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения. Москва : Юпитер, 2018. – 352 с.
3. Фаулер М. Архитектура корпоративных приложений. Издательский дом «Вильямс», 2006. – 544 с.
4. Thomas Erl “Service-Oriented Architecture. Analysis and Design for Services and Microservices”. 2nd edition (December 12, 2016), - 416 pages.
5. Mark Richards, Neal Ford "Fundamentals of Software Architecture. An Engineering Approach". 1st Edition (March 3, 2020), - 419 pages.
6. Chris Richardson “Microservices Patterns”. 1st edition (November 19, 2018), - 520 pages.
7. Иван Портянкин «Программирование Cloud Native. Микросервисы, Docker и Kubernetes» Архитектура микросервисов. Leanpub book 2022, - 172 с.

Онлайн-источники

8. BPMN BPMN — Википедия
9. The Clean Architecture: clean architecture of uncle Bob
10. Создание предметно ориентированных сервисов - https://medium.com/nuances-of-programming/создание-предметно-ориентированных-микросервисов-7283af13b827
11. Взаимодействие между микросервисами - Взаимодействие в архитектуре микрослужб | Microsoft Learn
12. REST REST — Википедия
13. Асинхронное взаимодействие на базе сообщений - Асинхронное взаимодействие на основе сообщений | Microsoft Learn

14. Сравнение шаблона шлюза API с прямым взаимодействием клиента и микрослужбы - Сравнение шаблона шлюза API с прямым взаимодействием клиента и микрослужбы | Microsoft Learn

15. Слои, Луковицы, Гексогоны, Порты и Адаптеры — всё это об одном - Слои, Луковицы, Гексогоны, Порты и Адаптеры — всё это об одном / Хабр

16. Уровни зрелости REST API - Модель зрелости Ричардсона

17. Bounded Context - BoundedContext

18. DDD - DomainDrivenDesign

19. DDD Aggregate - DDD_Aggregate