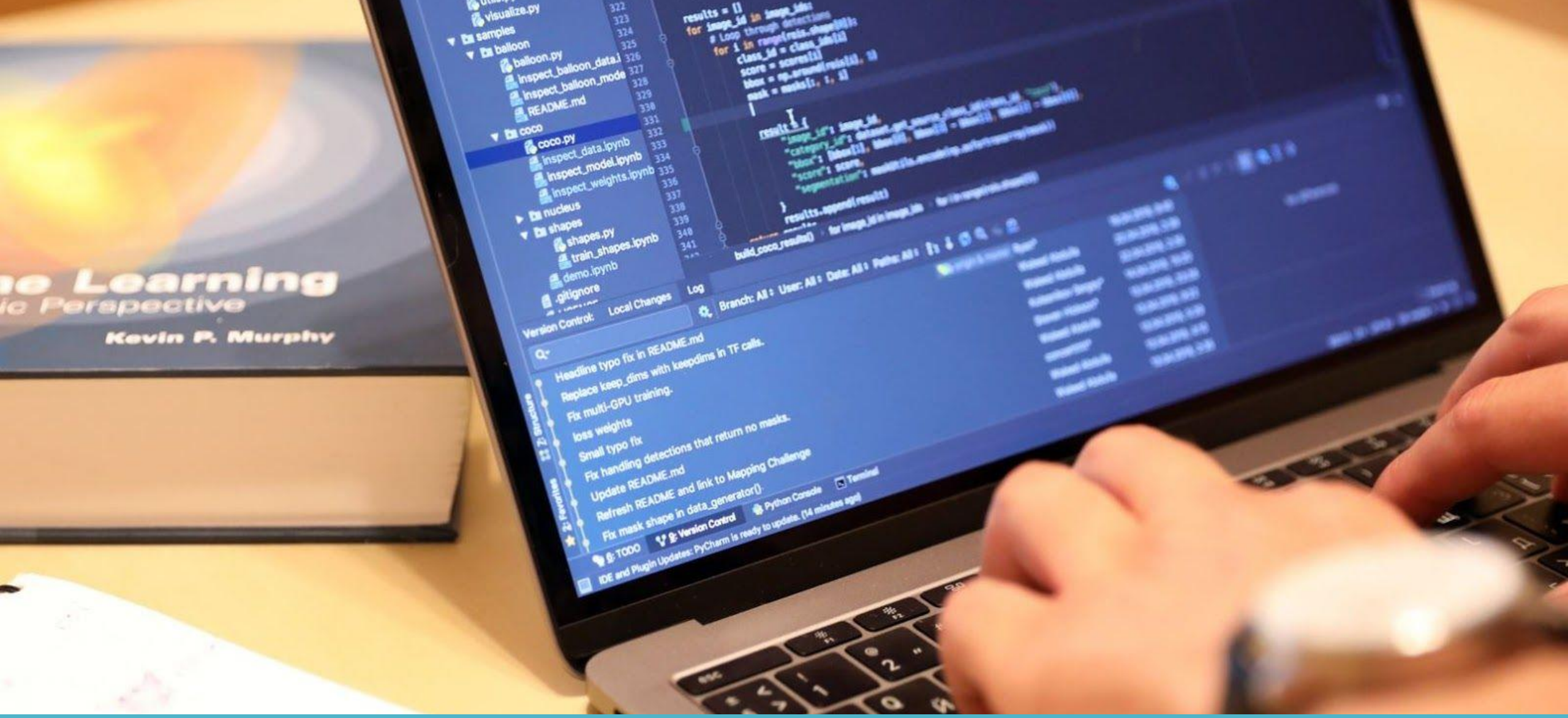
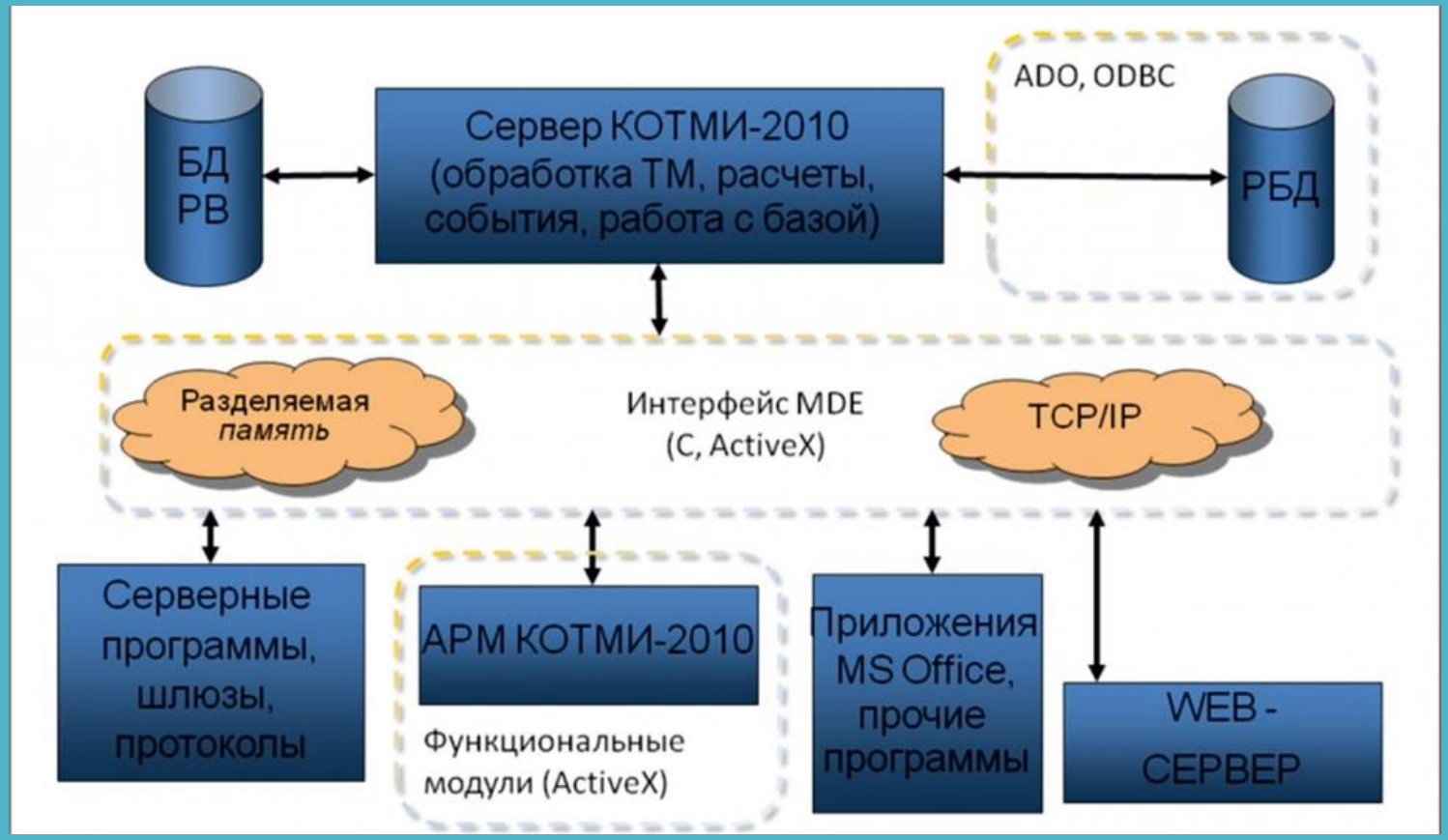


Тема 11: Архитектура программного продукта.

Цель занятия:

Ознакомиться с понятиями архитектура и структура ПП.



Основные понятия: архитектура, структура, компоненты.

Архитектура:

Архитектура программного продукта - это высокоуровневое описание его организации, компонентов, их взаимодействия и структуры. Она определяет общую концепцию, на которой строится программа, и служит основой для её проектирования и разработки. Архитектура включает в себя разделение системы на части, определение взаимосвязей между ними и выбор архитектурных стилей и паттернов.

Структура:

Структура программного продукта описывает его организацию внутри, включая разделение на модули, компоненты, классы и другие элементы. Структура определяет, как компоненты взаимодействуют друг с другом, какие зависимости между ними существуют, и какие модули выполняют какие функции. Определение четкой структуры помогает упростить разработку, тестирование и сопровождение программы.

Компоненты:

Компоненты представляют собой логически обособленные и переиспользуемые части программного продукта, имеющие четко определенные функции и интерфейсы. Они способствуют модульности программы, позволяя разделять сложные системы на более простые и независимые части. Компоненты могут быть классами, модулями, библиотеками и другими единицами разработки, обладающими определенной функциональностью.

Различия между архитектурой и структурой:

Структура программного продукта и архитектура программного продукта - это два важных аспекта проектирования, которые тесно связаны, но имеют различные уровни абстракции и задачи.

Структура программного продукта: Структура программного продукта отражает его организацию на уровне файлов и директорий, компонентов и модулей, их взаимосвязи и расположение в исходном коде. Это своего рода физическая организация кода, файлов и ресурсов в проекте. В структуре можно определить, как компоненты связаны друг с другом, как они организованы в папках и пакетах, какие ресурсы (изображения, шрифты, стили и т.д.) используются и где они располагаются.

Архитектура программного продукта: Архитектура программного продукта представляет собой высокоуровневый план, который описывает организацию компонентов и модулей на более абстрактном уровне. Это концептуальное представление системы, которое определяет её ключевые компоненты, их взаимодействие, структуру и порядок выполнения операций. Архитектура обычно описывает, как система разбивается на более мелкие компоненты, как они взаимодействуют друг с другом и как обеспечивается общая целостность и функциональность всего продукта.

Различие между архитектурой и дизайном.

Архитектура и дизайн - это два ключевых аспекта разработки программных систем, но они имеют разные фокусы и задачи. Вот основные различия между архитектурой и дизайном:

Архитектура:

Фокус: Архитектура обращает внимание на общую структуру, организацию и взаимодействие компонентов системы.

Задачи: Определение основных компонентов системы, их связей и взаимодействия, выбор архитектурных паттернов и стилей.

Цель: Обеспечить общую организацию системы, ее расширяемость, управляемость, производительность и другие высокоуровневые характеристики.

Пример: Разделение приложения на клиентскую и серверную части, выбор микросервисной или монолитной архитектуры.

Различия между архитектурой и дизайном.

Дизайн:

Фокус: Дизайн сосредотачивается на внешнем виде, интерфейсе пользователя и пользовательском опыте.

Задачи: Проектирование интерфейсов, выбор цветовой палитры, компоновка элементов на экране, создание графики и анимаций.

Цель: Создать удовлетворительный пользовательский опыт, обеспечивая интуитивную навигацию, эффективное взаимодействие и эстетичный внешний вид.

Пример: Дизайн интерфейса веб-приложения, включая расположение кнопок, шрифты, иконки и цветовую схему.

Проектирование архитектуры

Проектирование архитектуры программной системы требует внимательного анализа требований, понимания принципов дизайна и лучших практик в индустрии разработки программного обеспечения. Архитектура программной системы - это структура и организация системы, определяющая ее компоненты, взаимодействие между ними и принципы организации данных, входящих в систему.

Некоторые основы проектирования программной системы и ее архитектуры:

Разделение на компоненты: Программная система разбивается на отдельные компоненты, которые выполняют определенные функции или задачи. Это позволяет разделить сложную систему на более простые и понятные модули, что облегчает разработку, тестирование и поддержку системы.

Интерфейсы: Компоненты системы должны иметь четко определенные интерфейсы, через которые они взаимодействуют друг с другом. Интерфейсы определяют методы обмена данными и коммуникации между компонентами, обеспечивая слабую связность и гибкость системы.

Архитектурные стили: Существуют различные архитектурные стили, такие как клиент-серверная, трехуровневая, микросервисная и др. Выбор архитектурного стиля зависит от требований системы, ее целей и ограничений. Каждый стиль имеет свои преимущества и недостатки, и выбор должен быть обоснован и основан на конкретных потребностях проекта.

Управление данными: Архитектура определяет, как данные будут организованы, храниться и обрабатываться в системе. Это включает в себя выбор баз данных, структур данных, методов доступа к данным и принципов обработки данных.

Безопасность: Архитектура должна учитывать требования безопасности системы. Это может включать в себя механизмы аутентификации, авторизации, шифрования данных, контроля доступа и другие меры для защиты информации от несанкционированного доступа и злоумышленников.

Производительность и масштабируемость: Архитектура должна обеспечивать высокую производительность системы и возможность масштабирования при росте нагрузки. Это может включать в себя оптимизацию производительности, кэширование, параллельную обработку, горизонтальное и вертикальное масштабирование и другие техники.

Тестируемость и сопровождаемость: Архитектура должна облегчать тестирование и сопровождение системы. Хорошо спроектированная архитектура позволяет проводить модульное тестирование, интеграционное тестирование и обеспечивает простоту внесения изменений и обновлений в систему.

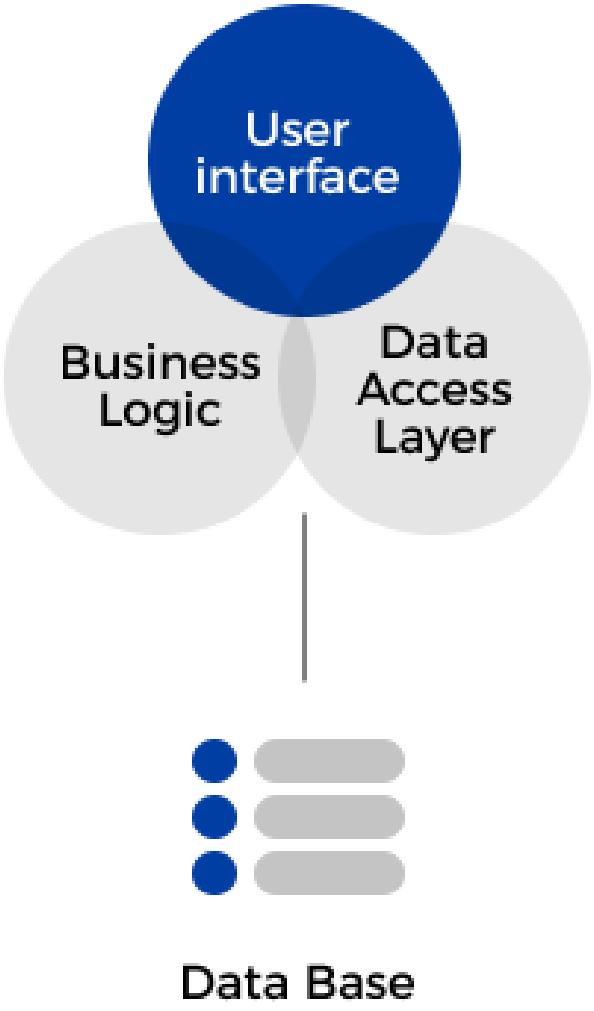
Проектирование архитектуры программной системы требует учета требований, целей и ограничений проекта, а также понимания принципов дизайна и лучших практик в индустрии.

Архитектурные стили

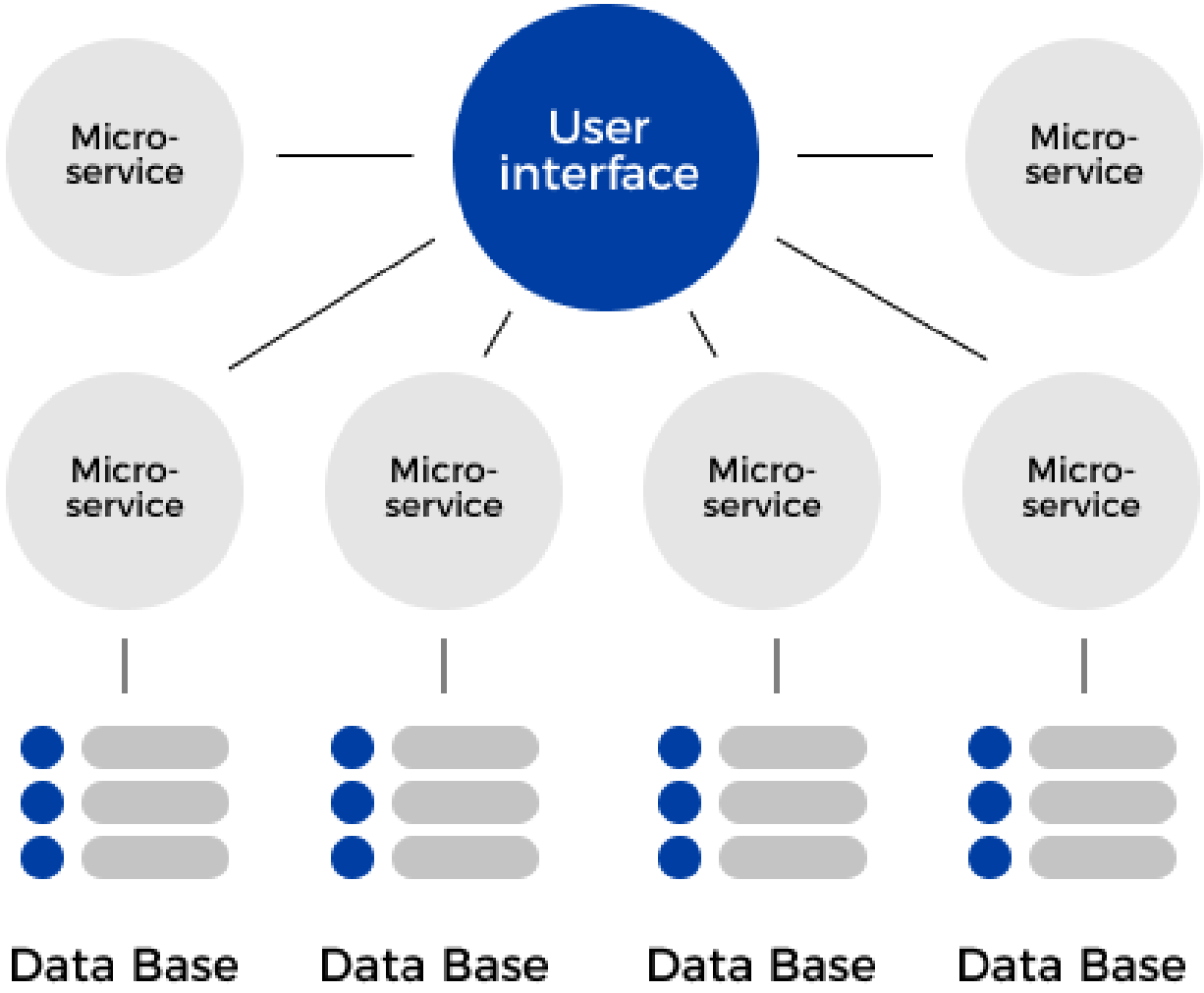
Монолитная архитектура (Monolithic Architecture):

В монолитной архитектуре все компоненты приложения объединены в единый исполняемый файл или пакет. Она характеризуется тесной связью между компонентами и отсутствием четкой границы между ними. Весь код приложения выполняется в одном процессе и имеет общее состояние. Этот образец архитектуры прост в разработке и развертывании, но может стать сложным для масштабирования и поддержки в случае больших и сложных приложений.

MONOLITHIC ARCHITECTURE



MICROSERVICE ARCHITECTURE



Особенности монолитной архитектуры:

Единое приложение: Вся функциональность системы упаковывается вместе и развертывается как единое приложение. Весь код, бизнес-логика и доступ к данным находятся внутри этого приложения.

Монолитный код: Код системы обычно организован в несколько модулей или компонентов, но все они взаимодействуют напрямую друг с другом. Обычно нет явного разделения на слои или сервисы.

Централизованное развертывание: Монолитные приложения развертываются на одном сервере или виртуальной машине. Это означает, что все компоненты и зависимости системы работают в одной среде.

Преимущества монолитной архитектуры:

Простота разработки: Разработка и тестирование монолитного приложения может быть проще, поскольку нет необходимости управлять сложной инфраструктурой для связи между компонентами.

Упрощенное развертывание: Монолитные приложения могут быть относительно легко развернуты на сервере и масштабированы вертикально (путем увеличения ресурсов сервера) для обеспечения производительности.

Простота отладки: Отладка монолитного приложения может быть проще, поскольку весь код и логика находятся в одном месте.

Ограничения монолитной архитектуры:

Масштабируемость: Монолитные приложения могут иметь ограниченные возможности масштабирования. При увеличении нагрузки требуется масштабирование всего приложения, даже если только некоторые его компоненты испытывают высокую нагрузку.

Гибкость и развитие: Изменение и дополнение функциональности монолитного приложения может быть сложным и рискованным, поскольку изменения могут затронуть всю систему и требовать ее полной пересборки и развертывания.

Зависимость компонентов: Компоненты монолитного приложения обычно сильно связаны друг с другом, что может повлечь сложности при поддержке и изменении системы.

Вот некоторые типы программных продуктов, в которых целесообразно применять монолитную архитектуру:

Небольшие и средние веб-приложения:

Монолитная архитектура хорошо подходит для небольших и средних веб-приложений, где требуется относительно небольшое количество компонентов и функциональностей. Она обеспечивает простоту разработки и развертывания, поскольку весь код находится в одном приложении.

Прототипы и концептуальные продукты:

Если у вас есть идея для нового продукта или концепции, монолитная архитектура может быть быстрым способом создания прототипа или демонстрационной версии. Она позволяет быстро развернуть минимально работающую систему без необходимости заниматься сложным разделением на компоненты.

Внутренние информационные системы:

Внутренние информационные системы, такие как системы управления ресурсами предприятия (ERP), системы управления клиентами (CRM) или системы управления складом (WMS), могут быть разработаны с использованием монолитной архитектуры. Это обусловлено тем, что такие системы обычно требуют небольшого числа пользователей и имеют ограниченные интеграционные потребности.

Относительно статичные системы:

Если ваша система имеет стабильные требования и небольшую склонность к изменениям, монолитная архитектура может быть достаточно эффективной. Например, информационные сайты, блоги или онлайн-магазины с небольшим количеством функциональных изменений могут быть построены как монолитные приложения.

Продукты с ограниченными ресурсами:

В случаях, когда у вас ограничены ресурсы для разработки, развертывания и поддержки, монолитная архитектура может быть предпочтительной. Поскольку все компоненты находятся внутри одного приложения, это может упростить процесс разработки и управления системой.

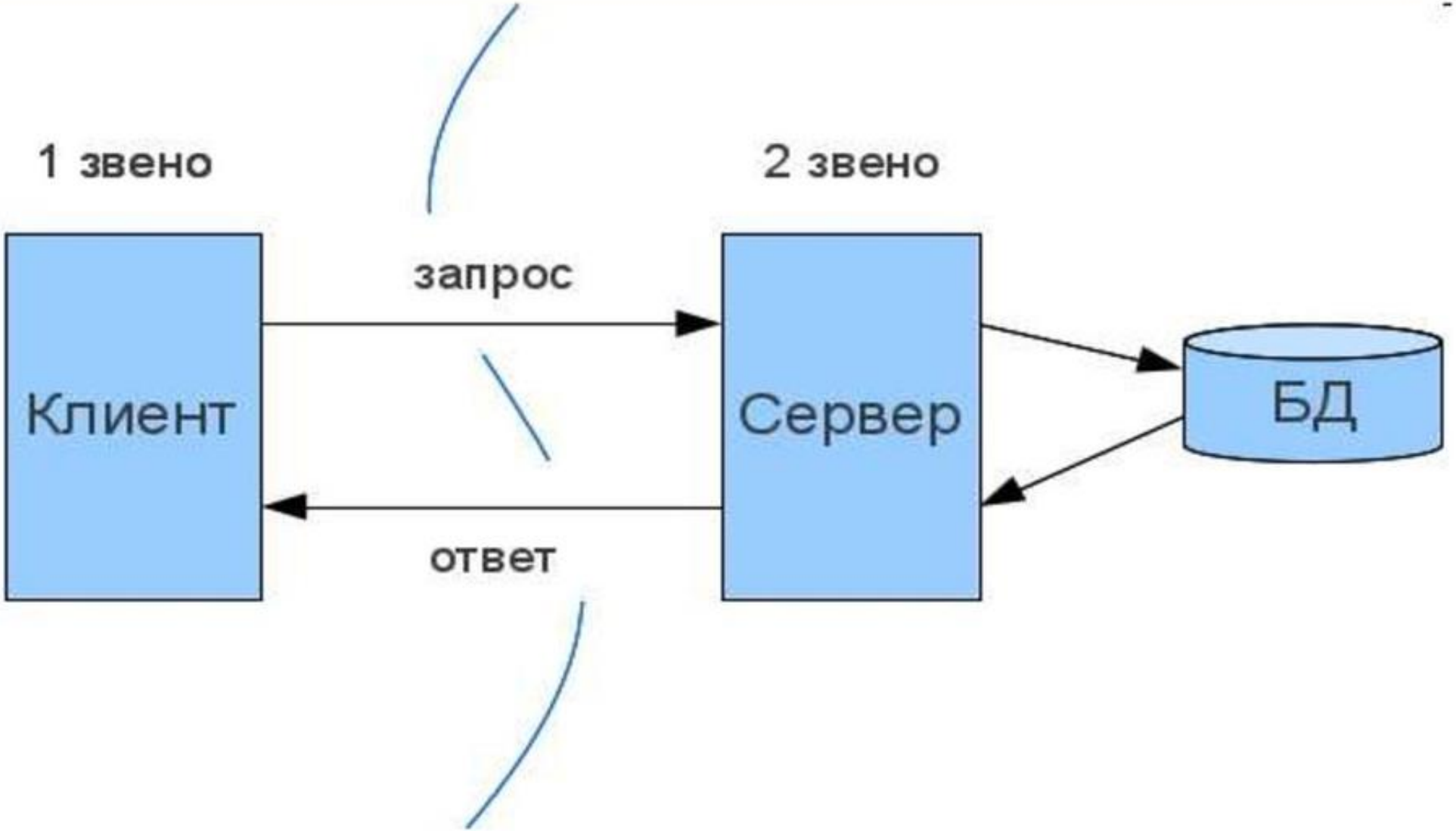
Клиент-серверная архитектура (Client-Server Architecture): В этой архитектуре система разделена на две основные компоненты: клиенты и серверы. Клиенты обращаются к серверам для получения ресурсов или выполнения операций. Серверы предоставляют запрашиваемые ресурсы, обрабатывают запросы клиентов и выполняют бизнес-логику. Примеры таких архитектур включают веб-приложения, где клиентом является веб-браузер, а сервером - веб-сервер.

В клиент-серверной архитектуре обычно выделяются две основные роли:

Клиент: Клиент - это пользовательское приложение или устройство, которое запрашивает определенные услуги или данные у сервера. Клиент отправляет запросы на сервер и получает ответы.

Сервер: Сервер - это выделенное устройство или программное обеспечение, которое предоставляет услуги или данные клиентам. Сервер принимает запросы от клиентов, обрабатывает их и отправляет обратно ответы.

Клиент-серверная архитектура



Особенности клиент-серверной архитектуры:

Разделение обязанностей: Функциональность приложения разделена между клиентской и серверной частями. Клиент отвечает за пользовательский интерфейс и взаимодействие с пользователем, а сервер обеспечивает обработку запросов, бизнес-логику и доступ к данным.

Коммуникация по сети: Клиент и сервер общаются друг с другом посредством сетевого соединения, обычно по протоколу TCP/IP. Клиент отправляет запросы на сервер, а сервер обрабатывает эти запросы и отправляет обратно ответы.

Масштабируемость: Клиент-серверная архитектура обеспечивает гибкость и масштабируемость. Клиенты и серверы могут быть развернуты на разных устройствах или серверах, что позволяет легко масштабировать систему в зависимости от требований.

Разделение обязанностей: Разделение функциональности между клиентом и сервером облегчает разработку и сопровождение приложения, так как изменения в одной части системы не требуют изменений в другой.

Централизованное управление данными: Сервер может обеспечить централизованное управление данными, что обеспечивает согласованность и надежность хранения данных.

Масштабируемость: Клиент-серверная архитектура позволяет масштабировать систему путем добавления или удаления серверов в зависимости от нагрузки.

Ограничения клиент-серверной архитектуры:

Единственная точка отказа: Если сервер недоступен, клиенты не смогут получить доступ к требуемым услугам или данным. Это делает сервер узким местом и может привести к проблемам доступности.

Зависимость от сети: Клиент-серверная архитектура требует наличия надежной сети связи между клиентами и серверами. Проблемы сети могут привести к задержкам или потере соединения между клиентом и сервером.

Безопасность: Клиент-серверная архитектура требует особых мер безопасности для защиты данных, передаваемых между клиентом и сервером, и предотвращения несанкционированного доступа к серверу.

Примеры программных продуктов, где целесообразно использовать клиент-серверную архитектуру:

Веб-приложения:

Множество веб-приложений основаны на клиент-серверной архитектуре. Браузер является клиентом, который отправляет HTTP-запросы на веб-сервер, а сервер обрабатывает запросы и возвращает HTML-страницы, данные или другие ресурсы в ответ.

Мобильные приложения:

Многие мобильные приложения также используют клиент-серверную архитектуру. Мобильное устройство выполняет роль клиента, взаимодействуя с сервером через сеть. Сервер может предоставлять данные, обновления, аутентификацию и другие сервисы для мобильного приложения.

Базы данных:

Базы данных также могут использовать клиент-серверную архитектуру. Клиентские приложения, такие как приложения для управления базами данных или веб-приложения, могут отправлять запросы на сервер базы данных для выполнения операций чтения, записи и обновления данных.

Почтовые серверы:

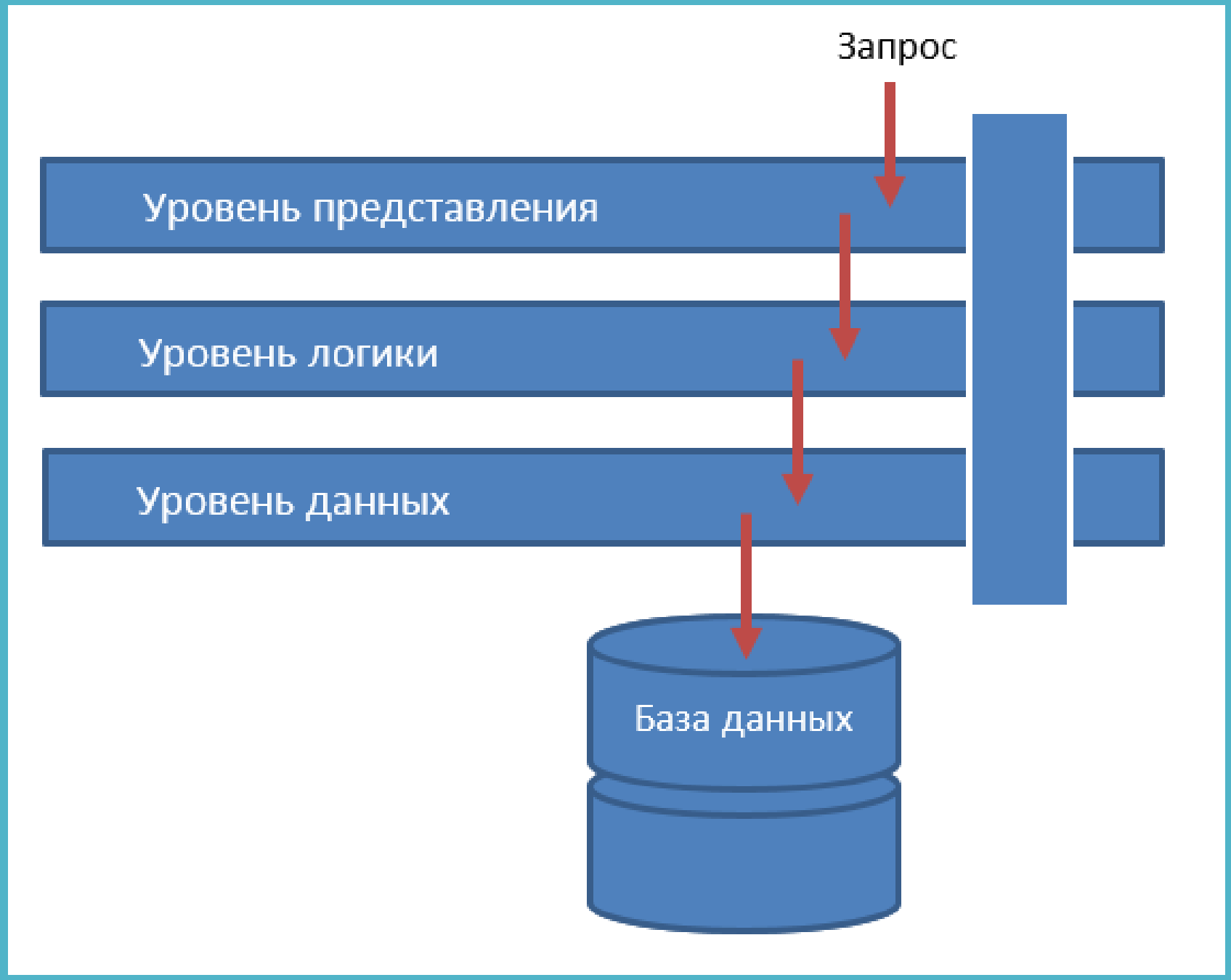
Серверы электронной почты являются примером клиент-серверной архитектуры. Почтовые клиенты, такие как почтовые программы или веб-интерфейсы, взаимодействуют с почтовым сервером для отправки и приема электронной почты.

Игровые серверы:

В онлайн-играх клиент-серверная архитектура позволяет множеству игровых клиентов взаимодействовать с игровым сервером. Клиенты отправляют команды и получают обновления от сервера, который координирует игровой процесс и обрабатывает действия игроков.

N-уровневая архитектура (N-Tier Architecture):

В N-уровневой архитектуре приложение разделено на несколько уровней, каждый из которых выполняет определенные функции. Обычно выделяют три основных уровня: представление (пользовательский интерфейс), бизнес-логика и уровень данных. Это позволяет достичь модульности, повторного использования кода и упрощения тестирования и поддержки приложения.


Запрос
Уровень представления
Уровень логики
Уровень данных
База данных

Преимущества модели, основанной на слоях:

Разделение ответственности: Разделение на слои позволяет явно определить и разделить функциональность приложения, упрощая разработку, тестирование и поддержку. Каждый слой выполняет свою специфическую задачу, что облегчает понимание и изменение системы.

Повторное использование: Разделение на слои способствует повторному использованию компонентов и модульности. Каждый слой может быть независимо разработан, тестирован и заменен без влияния на другие слои, если интерфейсы остаются неизменными.

Гибкость и масштабируемость: Модель, основанная на слоях, обеспечивает гибкость и масштабируемость, так как каждый слой может быть масштабирован или заменен при необходимости без влияния на другие слои.

Тестируемость: Разделение функциональности на слои упрощает тестирование системы. Каждый слой может быть протестирован отдельно с использованием модульных тестов, что позволяет обнаруживать и исправлять ошибки на ранних стадиях разработки.

Однако модель, основанная на слоях, также имеет свои ограничения. Сложность и перегрузка слоев могут привести к избыточности кода и усложнению архитектуры системы. Кроме того, неправильное разделение на слои может привести к проблемам с производительностью и распределением ответственности между слоями.

Важно тщательно спроектировать и организовать слои, чтобы достичь баланса между модульностью и простотой системы. Это может включать в себя правильное определение границ между слоями, управление зависимостями и создание четких интерфейсов между слоями.

В общем, модель, основанная на слоях, является распространенным подходом при проектировании приложений и систем. Она позволяет достичь модульности, повторного использования и гибкости, что способствует более эффективной разработке и поддержке программного обеспечения.

Некоторые типы программных продуктов, где целесообразно применять N-уровневую архитектуру:

Веб-приложения:

N-уровневая архитектура широко применяется в веб-приложениях. Обычно она состоит из клиентского уровня (отображение пользовательского интерфейса в браузере), серверного уровня (обработка запросов и бизнес-логика) и уровня базы данных (хранение и извлечение данных). Этот подход позволяет легко масштабировать и поддерживать веб-приложения.

Приложения с разделением на клиентскую и серверную части:

Когда программа имеет клиентскую и серверную части, N-уровневая архитектура может быть полезной. Например, клиентское приложение может быть мобильным или настольным приложением, а серверная часть обрабатывает запросы клиентов и предоставляет данные или сервисы. Это позволяет легко разделять ответственность между клиентом и сервером.

Корпоративные информационные системы:

N-уровневая архитектура широко применяется в корпоративных информационных системах, таких как системы управления ресурсами предприятия (ERP), системы управления клиентами (CRM) или системы управления складом (WMS). Разделение на уровни позволяет логически организовать систему, упростить ее разработку и поддержку.

Системы обработки данных:

В системах обработки данных, таких как системы аналитики данных или системы управления содержимым (CMS), N-уровневая архитектура может быть полезной. Разделение на уровни позволяет эффективно обрабатывать и хранить данные, применять специализированные компоненты для анализа и обработки данных на разных уровнях.

Распределенные приложения:

Если система должна функционировать в распределенной среде с несколькими узлами или серверами, N-уровневая архитектура может быть полезным подходом. Каждый узел может выполнять свою роль на определенном уровне, обмениваясь данными и коммуницируя с другими узлами.

Микросервисная архитектура (Microservices Architecture):

В этой архитектуре система разделяется на набор небольших, автономных и слабо связанных сервисов, которые могут быть развернуты и масштабированы независимо друг от друга. Каждый сервис выполняет конкретную функцию или обрабатывает определенную бизнес-задачу. Контейнеризация и оркестрация сервисов часто используются для управления и развертывания микросервисов.

Вот некоторые ключевые аспекты микросервисной архитектуры:

Разделение на сервисы: Большое приложение разбивается на отдельные микросервисы, каждый из которых выполняет определенную функцию или ответственность в пределах своей ограниченной области.

Автономность: Каждый сервис является автономным и может быть разработан, развернут и масштабирован независимо от других сервисов. Это позволяет командам разработчиков работать над различными сервисами параллельно и вносить изменения без влияния на другие сервисы.

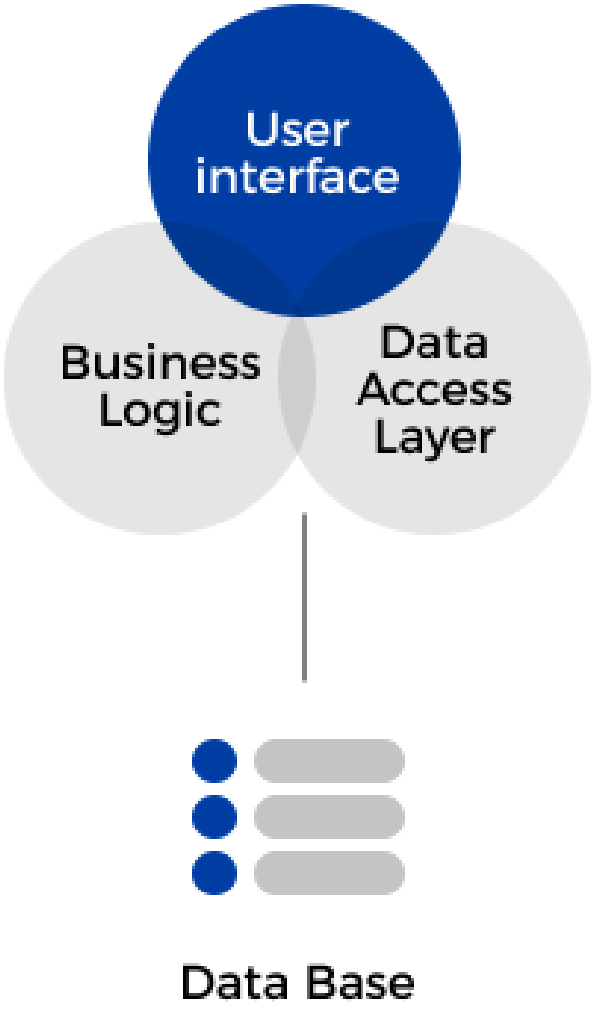
Локализованная база данных: Каждый сервис обычно имеет свою собственную базу данных, специфичную для его потребностей. Это помогает избежать сложностей, связанных с централизованной базой данных и облегчает изменение структуры данных внутри сервиса.

Коммуникация через сеть: Сервисы общаются между собой через сетевые протоколы, обычно с использованием API. Это позволяет сервисам работать на разных хостах или даже в разных центрах обработки данных.

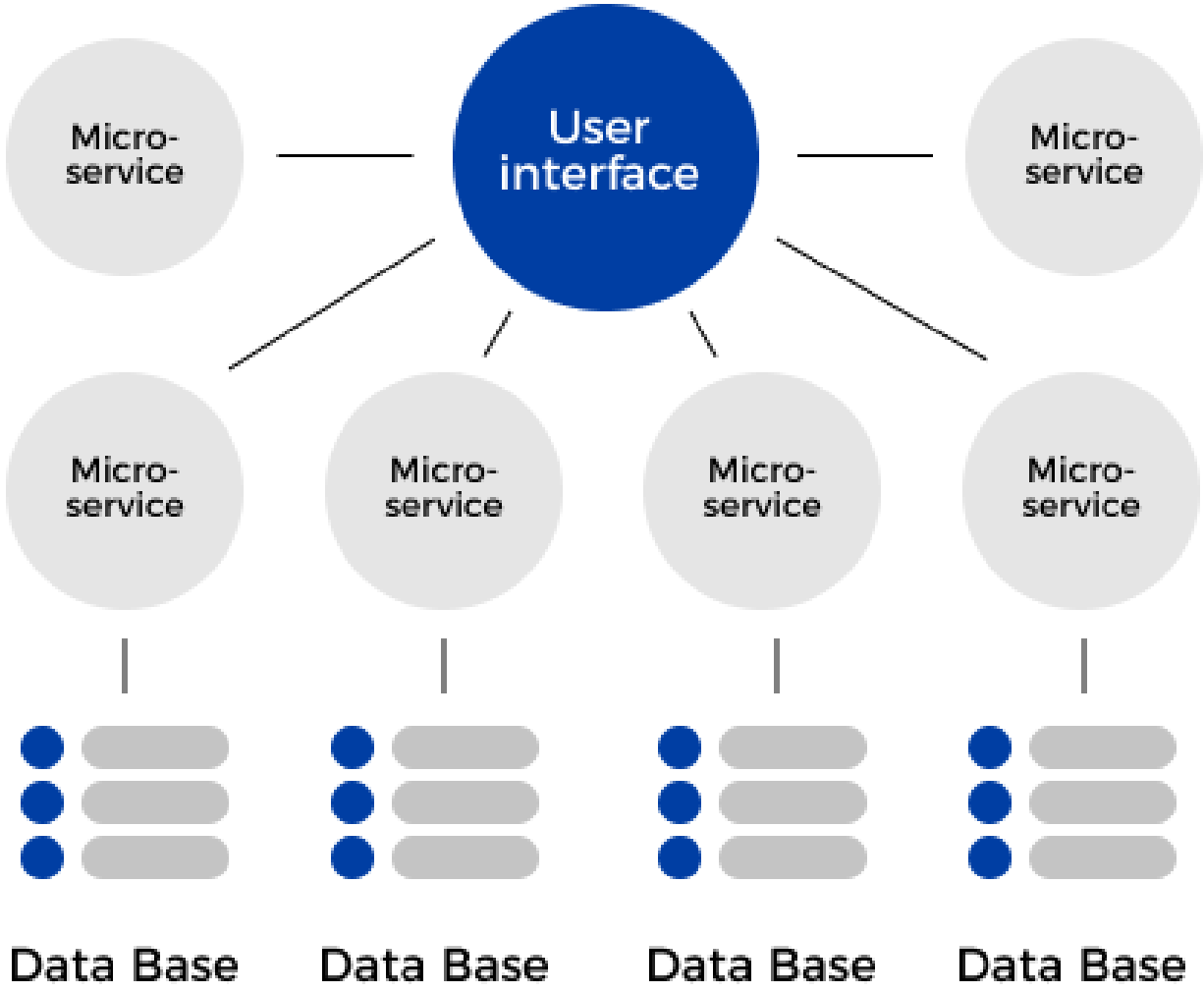
Независимое масштабирование: Каждый сервис может быть масштабирован отдельно в зависимости от его нагрузки. Это позволяет гибко управлять ресурсами и обеспечивать высокую доступность и производительность.

Управление ошибками: В микросервисной архитектуре неизбежны сбои и ошибки. Поэтому важно иметь стратегии обработки ошибок, резервное копирование и мониторинг каждого сервиса.

MONOLITHIC ARCHITECTURE



MICROSERVICE ARCHITECTURE



Преимущества микросервисной архитектуры включают:

Гибкость: Микросервисы могут быть разработаны с использованием разных технологий и могут быть развернуты и масштабированы независимо друг от друга. Это позволяет командам использовать наиболее подходящие инструменты и технологии для каждого сервиса.

Масштабируемость: Каждый сервис может быть масштабирован независимо, что обеспечивает гибкость в управлении нагрузкой и обеспечивает высокую производительность.

Легкость в сопровождении: Изоляция сервисов позволяет командам разработчиков изменять и обновлять сервисы без воздействия на другие части приложения.

Улучшение разработки: Микросервисная архитектура позволяет командам разработчиков работать над различными сервисами параллельно, сокращая время разработки и улучшая производительность.

Однако микросервисная архитектура также имеет некоторые вызовы и сложности, включая:

Сетевая сложность: Взаимодействие между сервисами через сеть может добавить сложности в управлении и отладке приложения.

Управление транзакциями: Обеспечение консистентности данных и управление транзакциями между различными сервисами может быть сложной задачей.

Управление конфигурацией: Необходимо эффективно управлять конфигурацией и развертыванием множества сервисов.

Мониторинг и отладка: С увеличением количества сервисов усложняется мониторинг и отладка всей системы.

Микросервисная архитектура - это мощный подход к разработке приложений, который позволяет создавать гибкие, масштабируемые и легко сопровождаемые системы. Однако перед принятием решения о переходе к микросервисной архитектуре необходимо учитывать специфические требования вашего проекта и оценить сложности, связанные с ее внедрением и поддержкой.

Примеры программных продуктов, где целесообразно применять микросервисную архитектуру:

Крупномасштабные веб-приложения:

Микросервисная архитектура идеально подходит для крупномасштабных веб-приложений, которые имеют сложную бизнес-логику и требуют высокой масштабируемости. Каждый сервис может быть отвечать за определенный функционал, такой как аутентификация, обработка платежей, управление контентом и другие. Это позволяет гибко масштабировать и разрабатывать отдельные части приложения независимо друг от друга.

Системы электронной коммерции:

В системах электронной коммерции микросервисная архитектура может быть полезной для разделения функциональности, такой как управление инвентарем, обработка заказов, управление доставкой и т.д. Каждый сервис может быть разработан и масштабирован независимо, что способствует гибкости и масштабируемости системы.

Системы управления контентом:

В системах управления контентом, таких как блоги, новостные порталы или системы управления обучением, микросервисная архитектура может быть полезной. Каждый сервис может отвечать за управление определенным типом контента или функциональностью, такой как создание и редактирование статей, управление пользователями, обработка комментариев и т.д.

Системы аналитики и обработки данных:

В системах аналитики и обработки данных, микросервисная архитектура может быть полезной для обработки и агрегирования больших объемов данных. Каждый сервис может выполнять специализированные задачи, такие как сбор данных, их обработка, анализ и предоставление результатов. Это позволяет гибко масштабировать систему в зависимости от объема данных и требуемых вычислительных ресурсов.

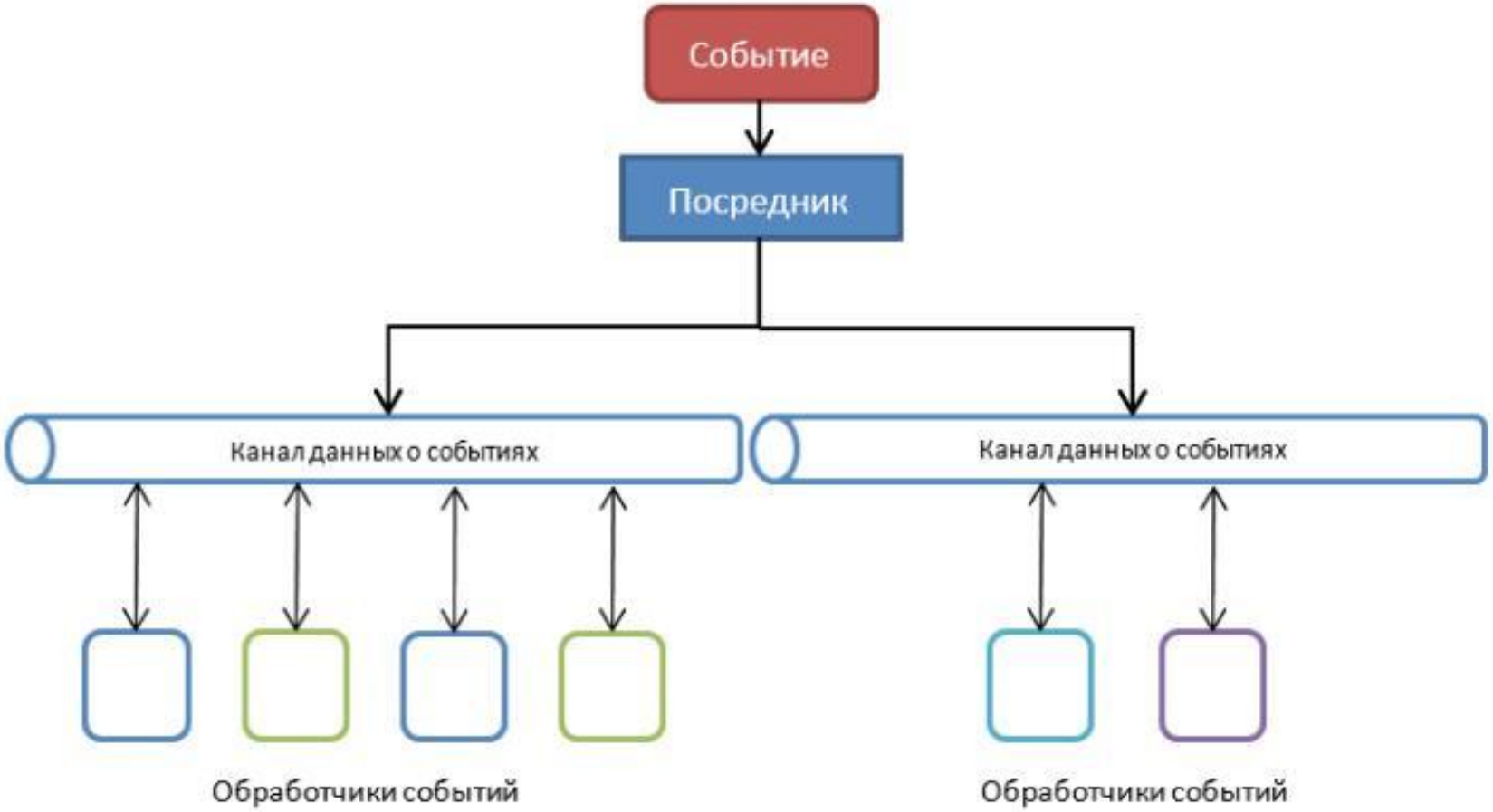
Событийно-ориентированная архитектура (Event-Driven Architecture):

Это шаблон проектирования с применением событий для запуска и передачи изменений между компонентами системы. Службы здесь взаимодействуют, обмениваясь событиями, то есть сообщениями о наступлении события или изменении состояния.

Компоненты могут генерировать и слушать события, и реагировать на них с помощью асинхронных обработчиков.

Это позволяет создавать более гибкие и масштабируемые системы, особенно при интеграции и обмене данными с внешними системами.

СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА



Примеры программных продуктов, где целесообразно применять событийно-ориентированную архитектуру:

Системы обработки и анализа данных в реальном времени:

EDA может быть полезна в системах, которые обрабатывают и анализируют поток данных в реальном времени, таких как системы мониторинга, системы обработки событий IoT или системы аналитики веб-трафика. События, такие как поступление данных от сенсоров или события веб-трафика, могут быть обработаны асинхронно и инициировать соответствующие реакции или действия в системе.

Микросервисные архитектуры:

Событийно-ориентированная архитектура хорошо сочетается с микросервисными архитектурами. Каждый микросервис может генерировать и слушать события, что позволяет им взаимодействовать и обмениваться информацией асинхронно. Это способствует более слабой связанности между сервисами и позволяет им быть независимыми в плане разработки, развертывания и масштабирования.

Системы с распределенной обработкой и интеграцией:

В системах, где требуется обработка и интеграция данных из различных источников, событийно-ориентированная архитектура может быть полезна. События могут использоваться для передачи и обмена информацией между различными компонентами системы, позволяя легко интегрировать новые источники данных или компоненты.

Системы событийного логирования и аудита:

EDA может быть применена в системах, где требуется регистрация и аудит событий. События могут быть записаны в событийный журнал (event log) для последующего анализа, отладки или аудита. Это позволяет системе сохранять историю событий и обеспечивает прозрачность происходящего в системе.

Принципы проектирования архитектуры ПО

Разделение ответственностей (Separation of Concerns): Архитектура должна быть разделена на отдельные компоненты или модули, каждый из которых отвечает за конкретные задачи или функциональные области. Это позволяет упростить разработку, поддержку и изменение системы.

Модульность (Modularity): Система должна быть разбита на независимые модули, которые могут быть разработаны, изменены и тестированы независимо друг от друга. Модули должны быть связаны четкими интерфейсами, чтобы обеспечить снижение зависимостей и повысить переиспользуемость.

Инкапсуляция (Encapsulation): Каждый модуль или компонент должен скрывать свою внутреннюю реализацию и предоставлять только необходимый интерфейс для взаимодействия с другими модулями. Это способствует уменьшению взаимозависимостей и облегчает изменения внутренней реализации без влияния на другие компоненты.

Расширяемость (Extensibility): Архитектура должна быть спроектирована с учетом возможности легкого добавления новых функций, модулей или компонентов в систему. Это позволяет системе адаптироваться к изменяющимся требованиям и расширяться без необходимости кардинальных изменений.

Повторное использование (Reusability): Компоненты или модули системы должны быть разработаны таким образом, чтобы их можно было повторно использовать в различных контекстах или проектах. Это позволяет сократить время разработки и повысить качество программного обеспечения.

Гибкость (Flexibility): Архитектура должна быть гибкой и адаптивной, чтобы легко поддерживать изменения и модификации в системе. Это включает возможность добавления новых функций, изменения существующих и удаление устаревших компонентов.

Производительность (Performance): Архитектура должна быть спроектирована таким образом, чтобы обеспечивать высокую производительность системы. Это включает оптимизацию алгоритмов, эффективное использование ресурсов и учет требований к производительности на различных уровнях системы.

Безопасность (Security): Архитектура должна предусматривать механизмы защиты данных и ресурсов системы от несанкционированного доступа или злоумышленников. Это включает обеспечение конфиденциальности, целостности и доступности информации.

Тестируемость (Testability): Архитектура должна обеспечивать легкость тестирования системы, включая модульное тестирование компонентов, интеграционное тестирование и проверку системы в целом. Это позволяет выявлять и исправлять ошибки на ранних этапах разработки и обеспечивает надежность и качество системы.

Простота (Simplicity): Простота является одним из ключевых принципов проектирования архитектуры ПО. Архитектура должна быть понятной, легкой для понимания и поддержки разработчиками. Избегайте излишней сложности и лишних абстракций, стремитесь к простоте и ясности.

Принципы проектирования архитектуры ПО

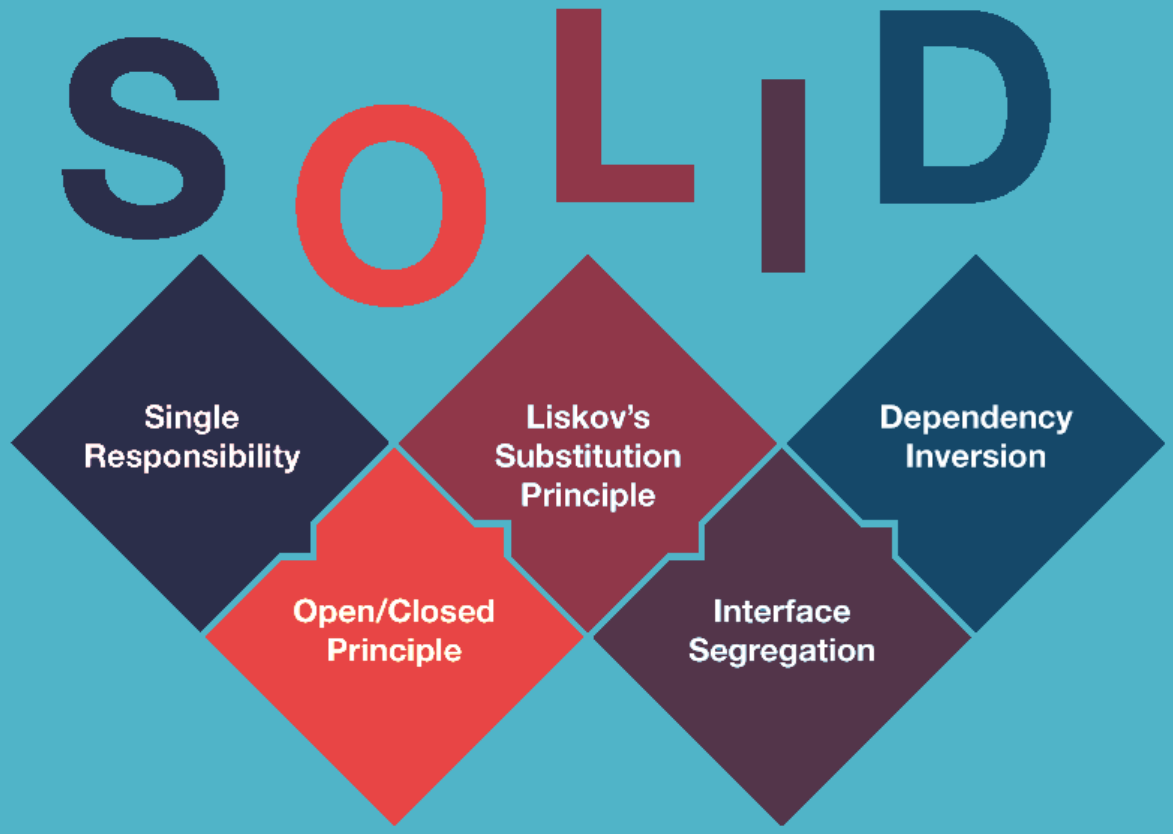
Принцип единой ответственности (Single Responsibility Principle):

Каждый модуль или компонент должен иметь только одну ответственность.

Это означает, что каждый элемент системы должен быть ответственен только за выполнение одной четко определенной функции или задачи.

Это упрощает понимание, тестирование и поддержку компонента, а также делает систему более гибкой и расширяемой.

Принципы SOLID и их роль в разработке ПО



SOLID - это аббревиатура, обозначающая первые пять принципов объектно-ориентированного программирования, сформулированные Робертом С. Мартином (также известным как дядя Боб). Эти принципы устанавливают практики, помогающие создавать программное обеспечение, которое можно обслуживать и расширять по мере развития проекта. Применение этих практик также поможет избавиться от плохого кода, оптимизировать код и создавать гибкое или адаптивное программное обеспечение

SOLID включает следующие принципы:

S - SRP (Single Responsibility Principle) принцип единственной ответственности

O - (Open/Closed Principle, OCP) принцип открытости/закрытости

L - (Liskov Substitution Principle, LSP) принцип подстановки Лисков

I - (Interface Segregation Principle, ISP)принцип разделения интерфейса

D - (Dependency Inversion Principle, DIP) принцип инверсии зависимостей

1.SRP (Single Responsibility Principle)

Принцип единственной ответственности является одним из пяти принципов SOLID, разработанных Робертом Мартином (Uncle Bob). Он определяет, что каждый класс или модуль должен иметь только одну причину для изменения и должен быть ответственным только за одну часть функциональности.

Основная идея принципа SRP состоит в том, чтобы разделить функциональность программы на отдельные модули или классы, каждый из которых будет отвечать только за выполнение одной конкретной задачи. Это позволяет улучшить поддерживаемость, расширяемость и повторное использование кода.

Важные аспекты принципа SRP:

Одна причина для изменения: Класс или модуль должен иметь только одну причину для изменения. Если класс имеет несколько причин для изменения, то изменение одной из этих причин может повлиять на другие аспекты функциональности, что затрудняет поддержку и усложняет разработку.
Высокая связность: Класс должен быть связан только с функциональностью, за которую он отвечает. Это означает, что класс должен содержать только связанные методы и данные, не превышая свою ответственность.
Низкая связность: Классы или модули должны быть слабо связаны друг с другом. Изменение в одном классе или модуле не должно требовать изменений в других классах, не имеющих прямого отношения к этой функциональности.

Преимущества принципа SRP:

Улучшение поддерживаемости: Когда каждый класс отвечает только за одну задачу, легче понять его функциональность и вносить изменения.
Улучшение возможностей тестирования: Модули с ясно определенной ответственностью могут быть более легко тестированы, поскольку их поведение более предсказуемо и легко изолировать от других компонентов системы.
Лучшая возможность для повторного использования: Модули, которые являются независимыми и имеют четкие границы ответственности, могут быть легко повторно использованы в других проектах или частях системы.
Улучшение расширяемости: Изменение или добавление новой функциональности может быть выполнено с минимальными изменениями в других модулях.

Пример на языке Python:

```python
# Пример без применения принципа SRP
class Employee:
    def __init__(self, name, id):
        self.name = name
        self.id = id

    def calculate_salary(self):
        # Расчет заработной платы
        pass

    def save_to_database(self):
        # Сохранение сотрудника в базу данных
        pass

    def send_notification(self):
        # Отправка уведомления сотруднику
        pass
```

Класс **Employee** нарушает принцип SRP, поскольку он объединяет несколько разных ответственностей, таких как расчет заработной платы, сохранение в базу данных и отправка уведомлений.

Пример на языке Python:

```python
# Чтобы применить принцип SRP, мы можем разделить
# эти ответственности на отдельные классы:
class Employee:
    def __init__(self, name, id):
        self.name = name
        self.id = id

class SalaryCalculator:
    def calculate_salary(self, employee):
        # Расчет заработной платы
        pass

class DatabaseSaver:
    def save_to_database(self, employee):
        # Сохранение сотрудника в базу данных
        pass

class NotificationSender:
    def send_notification(self, employee):
        # Отправка уведомления сотруднику
        pass
```

Теперь каждый класс отвечает только за одну конкретную задачу. Это улучшает разделение ответственности, делает код более поддерживаемым и упрощает повторное использование кода.

2. (Open/Closed Principle, OCP)

Принцип OCP (Принцип открытости/закрытости) является одним из принципов объектно-ориентированного программирования (ООП) и описывает принцип проектирования программного кода, который способствует его гибкости и расширяемости.

Суть принципа OCP заключается в том, что программные сущности, такие как классы, модули или функции, должны быть открыты для расширения, но закрыты для модификации. Это означает, что код должен быть легко расширяемым путем добавления нового функционала, но не требовать изменения уже существующего кода.

Принцип OCP можно реализовать с помощью использования абстракций, интерфейсов и полиморфизма. Вместо прямой зависимости от конкретных классов или модулей, код должен зависеть от абстракций, которые могут иметь несколько реализаций. Это позволяет добавлять новый функционал, создавая новые классы, которые реализуют абстракции, без изменения существующего кода.

```python
#
class Shape:
    def area(self):
        # Исключение, возникающее в случаях,
        #когда наследник не переопределил метод
        raise NotImplementedError

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2
```

В этом примере класс **Shape** является абстрактным базовым классом, а классы **Rectangle** и **Circle** наследуют его и реализуют метод **area()**.

3. Liskov Substitution Principle, LSP

Принцип OCP (Принцип открытости/закрытости) является одним из принципов объектно-ориентированного программирования (ООП) и описывает принцип проектирования программного кода, который способствует его гибкости и расширяемости.

Суть принципа заключается в том, что программные сущности, такие как классы, модули или функции, должны быть открыты для расширения, но закрыты для модификации. Это означает, что код должен быть легко расширяемым путем добавления нового функционала, но не требовать изменения уже существующего кода.

Принцип OCP можно реализовать с помощью использования абстракций, интерфейсов и полиморфизма. Вместо прямой зависимости от конкретных классов или модулей, код должен зависеть от абстракций, которые могут иметь несколько реализаций. Это позволяет добавлять новый функционал, создавая новые классы, которые реализуют абстракции, без изменения существующего кода.

```python
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        print("Sparrow can fly")

class Penguin(Bird):
    def fly(self):
        print("Penguin cannot fly")

def make_bird_fly(bird):
    bird.fly()

# Создание экземпляров классов
sparrow = Sparrow()
penguin = Penguin()

# Проверка, что принцип LSP соблюдается
make_bird_fly(sparrow)  # Вывод: "Sparrow can fly"
make_bird_fly(penguin)  # Вывод: "Penguin cannot fly"
```

Базовый класс **Bird** имеет метод **fly.** Два производных класса, **Sparrow** и **Penguin** переопределяют метод fly в соответствии с их поведением. Функция make_bird_fly принимает объект класса Bird и вызывает его метод fly. мы можем передать в ф-ю объекты классов Sparrow и Penguin и программа все равно будет работать корректно.

4. Interface Segregation Principle, ISP

Принцип ISP (Принцип разделения интерфейса) является одним из принципов SOLID и предлагает разделить интерфейсы на более мелкие и специфичные, чтобы клиенты могли зависеть только от тех методов, которые им действительно нужны. Это помогает избежать зависимостей от неиспользуемых методов и упрощает разработку, тестирование и поддержку кода.

```python
class Printer:
    def print_document(self, document):
        raise NotImplementedError

class Scanner:
    def scan_document(self):
        raise NotImplementedError

class Photocopier(Printer, Scanner):
    def print_document(self, document):
        # логика печати документа

    def scan_document(self):
        # логика сканирования документа
```

В этом примере классы Printer и Scanner представляют разные интерфейсы, а класс Photocopier реализует оба интерфейса. Таким образом, класс Photocopier может быть использован как принтер и сканер, но клиентский код может использовать только нужные ему методы.

5. Dependency Inversion Principle, DIP

Принцип инверсии зависимостей (Dependency Inversion Principle, DIP) является одним из принципов SOLID, который обеспечивает гибкость и устойчивость программного кода. Он предлагает следующее: "Зависимости должны строиться на абстракциях, а не на конкретных реализациях".

Принцип DIP подразумевает, что модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба уровня должны зависеть от абстракций. Это означает, что классы и функции должны зависеть от интерфейсов или абстрактных классов, а не от конкретных реализаций.

```python
# Пример без применения принципа DIP
class WeatherService:
    def get_temperature(self):
        # Логика получения температуры
        pass

class WeatherNotifier:
    def __init__(self):
        self.weather_service = WeatherService()

    def send_notification(self):
        temperature = self.weather_service.get_temperature()
        if temperature > 25:
            print("It's hot! Stay hydrated.")
        else:
            print("It's cool outside.")

notifier = WeatherNotifier()
notifier.send_notification()
```

В этом примере класс WeatherNotifier жестко зависит от класса WeatherService. Это создает проблему, так как WeatherNotifier прямо связан с конкретной реализацией WeatherService, что затрудняет гибкость и расширение системы

Чтобы применить принцип DIP, мы можем использовать инъекцию зависимостей:

```python
# Пример с применением принципа DIP
class WeatherService:
    def get_temperature(self):
        # Логика получения температуры
        pass

class WeatherNotifier:
    def __init__(self, weather_service):
        self.weather_service = weather_service

    def send_notification(self):
        temperature = self.weather_service.get_temperature()
        if temperature > 25:
            print("It's hot! Stay hydrated.")
        else:
            print("It's cool outside.")

weather_service = WeatherService()
notifier = WeatherNotifier(weather_service)
notifier.send_notification()
```

В этом примере класс WeatherNotifier не зависит напрямую от конкретной реализации WeatherService. Вместо этого, WeatherService передается в конструкторе WeatherNotifier в качестве зависимости (инъекции зависимостей). Теперь WeatherNotifier может работать с любым объектом, который реализует необходимый интерфейс WeatherService, что позволяет гибко менять или расширять функциональность.

Заключение.

Использование принципов SOLID способствует созданию такой системы, которую будет легко поддерживать и расширять в течение долгого времени

Пример архитектуры телеграм-бота для изучения иностранного языка:

Клиентский слой:

Telegram Bot API: Взаимодействие с Telegram API для обработки команд и сообщений от пользователей, а также отправки ответных сообщений и данных.

Интерфейс пользователя: Разработка интерфейса, предоставляющего возможность взаимодействия с ботом через команды, кнопки и текстовые сообщения.

Серверный слой:

Web-сервер: Обработка запросов от Telegram Bot API и маршрутизация их к соответствующим компонентам обработки.

Бизнес-логика: Реализация логики приложения, включая обработку команд, управление учебным материалом, генерацию упражнений и проверку ответов пользователей.

Модуль управления пользователями:

Хранение данных пользователей: Хранение информации о пользователях, их профилях, прогрессе обучения и статистике.

Аутентификация и авторизация: Обеспечение безопасного доступа к боту, аутентификация пользователей и управление их авторизационными данными.

Модуль обучения иностранному языку:

Учебный материал: Хранение и управление учебным контентом, включая тексты, аудио, видео, грамматические правила и словари.

Генерация упражнений: Создание динамических упражнений, заданий на практику грамматики, понимания на слух и т. д.

Оценка и обратная связь: Проверка ответов пользователей на упражнения и предоставление обратной связи о правильности или ошибке.

Внешние сервисы и API:

Интеграция с сервисами машинного обучения и обработки естественного языка для различных функциональностей, таких как распознавание речи, машинный перевод или проверка орфографии.

Использование внешних API для получения дополнительных ресурсов, таких как переводчики, словари или статистические данные.

База данных:

Хранение данных пользователей: Хранение информации о пользователях, их прогрессе, статистике обучения и других связанных данных.

Хранение учебного контента: Хранение текстовых материалов, аудио- и видеофайлов, грамматических правил и других ресурсов для обучения.

Архитектурные подходы, которые были использованы в предложенной архитектуре:

1.Модульная архитектура: Предложенная архитектура имеет модульную структуру, где различные компоненты (клиентский слой, серверный слой, модуль управления пользователями, модуль обучения иностранному языку и т. д.) разделены на отдельные модули с ясно определенными функциональностями. Это позволяет легко добавлять новые модули или изменять существующие без сильной зависимости между ними.

2.Клиент-серверная архитектура: Архитектура основана на принципе клиент-серверного взаимодействия, где клиент (Telegram Bot API) отправляет запросы на серверный слой (Web-сервер), который обрабатывает запросы и взаимодействует с другими компонентами для выполнения логики приложения и обеспечения ответов на запросы.

3.Архитектурный паттерн "Модель-Представление-Контроллер" (Model-View-Controller, MVC): В клиентском слое использован паттерн MVC для разделения логики приложения на модель (бизнес-логика и управление данными), представление (интерфейс пользователя) и контроллер (управление взаимодействием между моделью и представлением). Это обеспечит лучшую организацию кода и упростит его поддержку и расширение.

4.Микросервисная архитектура: Предложенная архитектура организована в соответствии с принципами SOA, где различные компоненты (например, модуль управления пользователями, модуль обучения иностранному языку) предоставляют сервисы с определенными функциональностями через API. Это облегчает масштабирование и повторное использование компонентов.

Список литературы:

Принципы SOLID в картинках / Хабр

- “Чистый код” Роберт Мартин
- “Приемы объектно-ориентированного проектирования. Паттерны проектирования” Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес
- “Рефакторинг. Улучшение существующего кода” Мартин Фаулер
- “Разработка через тестирование. Подход на основе примеров” Кент Бек
- “Принципы, паттерны и методики гибкой разработки на языке C#” Роберт С. Мартин, Мика Мартин