Построение архитектуры проекта. Шаблон проекта.

Выбор средства построения ИС

Построение архитектуры проекта – это важный этап в разработке информационной системы, который помогает определить структуру, компоненты, их взаимодействие и обеспечить устойчивость, масштабируемость и безопасность системы.

Выбор средства построения ИС

Шаблон проекта (или архитектурный шаблон) является структурным идеальным образцом, который позволяет стандартизировать архитектурные решения.

Давайте рассмотрим процесс построения архитектуры проекта и несколько популярных архитектурных шаблонов.

Выбор средства построения ИС

Идентификация требований:

- анализ требований к проекту. Определение функциональных требований, бизнес-процессов и особенностей проекта.
- установка приоритеты и фокусировка на ключевых требованиях.

Определение архитектурных принципов:

- разработка архитектурных принципов, которые будут руководить построением системы. Например, принципы безопасности, масштабируемости, гибкости и производительности.

Выбор средства построения ИС

Выбор архитектурных паттернов:

- стоит выбирать архитектурные паттерны, которые соответствуют требованиям проекта. Например, Model-View-Controller (MVC) для веб-приложений или микросервисную архитектуру для распределенных систем.

Определение компонентов и их взаимодействия:

- необходимо разбить систему на компоненты и определть, как они будут взаимодействовать друг с другом. Это включает в себя определение интерфейсов, API и данных, которые будут передаваться между компонентами.

Выбор средства построения ИС

Разработка архитектурных диаграмм:

- создание архитектурных диаграмм, таких как диаграммы классов, диаграммы последовательности и диаграммы компонентов, чтобы визуализировать структуру и взаимодействие компонентов.

Учет нефункциональных требований:

- обеспечение выполнения нефункциональных требований, таких как производительность, безопасность и масштабируемость, через выбор соответствующих архитектурных решений и технологий.

Выбор средства построения ИС

Популярные архитектурные шаблоны:

Model-View-Controller (MVC):

- Используется для построения веб-приложений.
- Разделяет систему на три компонента: модель (хранение данных), представление (отображение данных) и контроллер (обработка запросов и управление моделью и представлением).

Выбор средства построения ИС

MVC (Model-View-Controller) – это популярный архитектурный паттерн, используемый в разработке программных систем, особенно в веб-приложениях. Он предоставляет структуру для организации кода и разделения функциональности в приложении на три основных компонента: **Model**, **View** и **Controller**.

Этот разделительный подход упрощает сопровождение, расширение и тестирование приложения.

Рассмотрим каждый из компонентов более подробно.

Model-View-Controller (MVC):

Модель (Model):

Модель представляет собой компонент, который отвечает за управление данными и бизнес-логикой приложения. Он не зависит от пользовательского интерфейса и представления данных.

В модели содержится информация о состоянии приложения, его бизнес-правила и методы для доступа к данным и их обновления.

Model-View-Controller (MVC):

Представление (View):

Представление отвечает за отображение данных пользователю и предоставляет пользовательский интерфейс. Оно не содержит бизнес-логики и не обрабатывает данные, а только отображает их.

В веб-приложениях представление может быть HTML-страницей, шаблоном, виджетом и т. д. Оно получает данные из модели и отображает их пользователю.

Model-View-Controller (MVC):

Контроллер (Controller):

Контроллер является посредником между моделью и представлением. Он обрабатывает пользовательские запросы, взаимодействует с моделью для получения или обновления данных и управляет представлением для отображения результата.

Контроллер обычно содержит логику обработки запросов, маршрутизацию и взаимодействие с моделью и представлением.

Model-View-Controller (MVC):

Преимущества MVC:

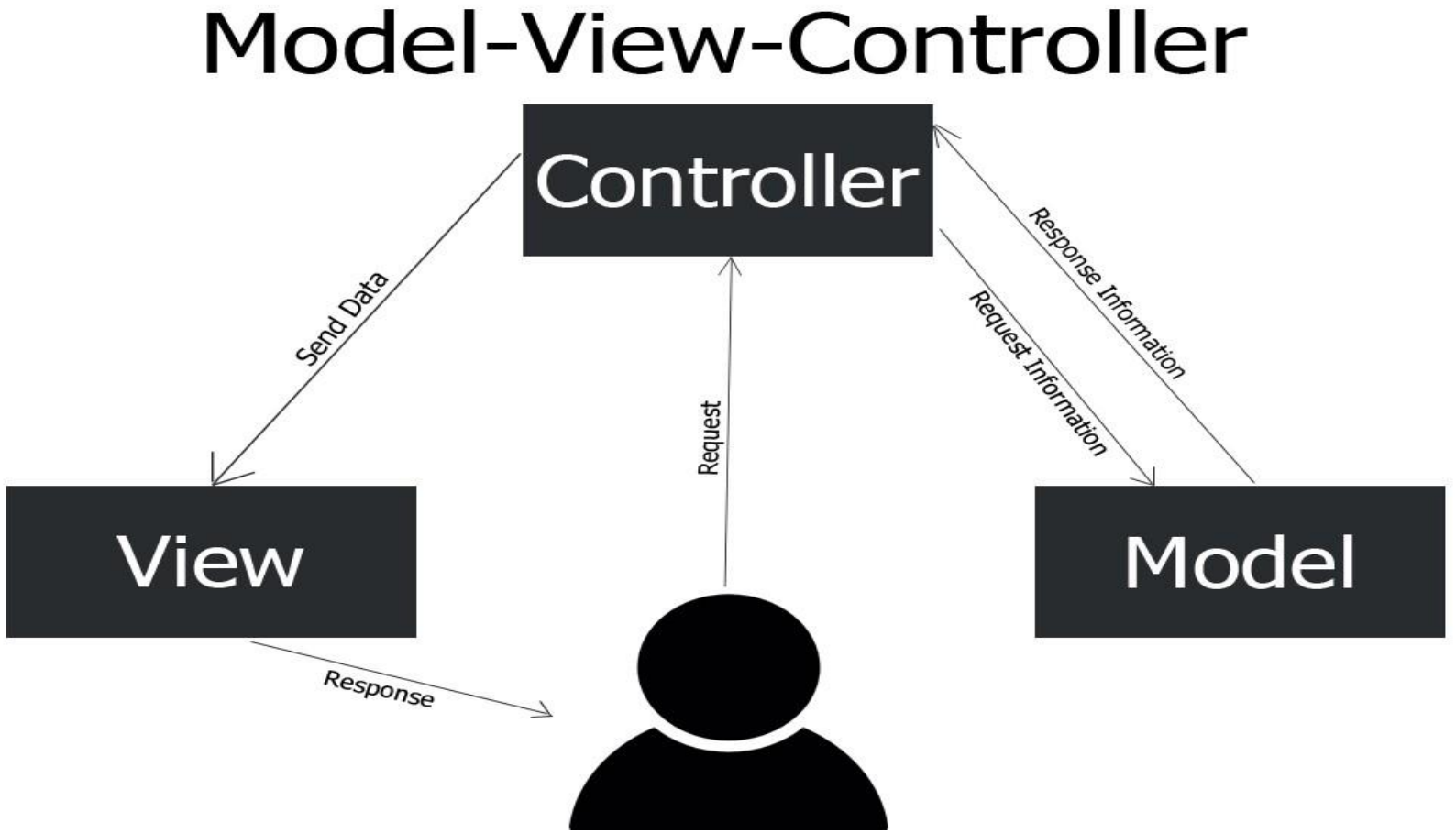
Разделение обязанностей: MVC позволяет разделить обязанности между компонентами приложения, что упрощает сопровождение и повторное использование кода.

Масштабируемость: Каждый компонент может быть изменен или расширен независимо от других, что упрощает масштабирование приложения.

Тестирование: Из-за четкого разделения функциональности, тестирование отдельных компонентов (особенно модели и контроллера) становится более простым.

Гибкость и поддержка: Если требования к пользовательскому интерфейсу меняются, часто можно переиспользовать модель и контроллер, меняя только представление.

Model-View-Controller (MVC):



Микросервисная архитектура

Микросервисная архитектура:

- Используется для построения распределенных систем.
- Разделяет систему на небольшие, независимые микросервисы, каждый из которых отвечает за определенную функциональность.

Микросервисная архитектура

Микросервисная архитектура (Microservices Architecture) – это структура разработки программного обеспечения, в которой приложение разбивается на небольшие, автономные и независимые сервисы, которые взаимодействуют друг с другом через API (Application Programming Interface). Эти микросервисы разрабатываются, развертываются и масштабируются независимо друг от друга, что позволяет командам разработки более гибко управлять приложением и облегчает его развитие и обслуживание.

Микросервисная архитектура

Каждый микросервис разрабатывается и поддерживается независимо. Это позволяет командам быстро вносить изменения в свой сервис без необходимости менять другие части приложения.

Микросервисы общаются между собой через сетевые вызовы по API.

Поскольку каждый микросервис независим, их можно разворачивать и обновлять отдельно. Это позволяет ускорить процесс разработки и внедрения изменений.

Микросервисная архитектура

Приложение, построенное на микросервисной архитектуре, может быть более устойчивым к отказам, так как отказ одного сервиса не обязательно приводит к отказу всего приложения.

Эффективное внедрение микросервисной архитектуры требует хорошего понимания ее принципов и лучших практик, а также правильного выбора инструментов для разработки, развертывания и управления сервисами. Микросервисная архитектура может быть полезной для больших и сложных приложений, но также может повлечь за собой дополнительные вызовы в управлении и обслуживании.

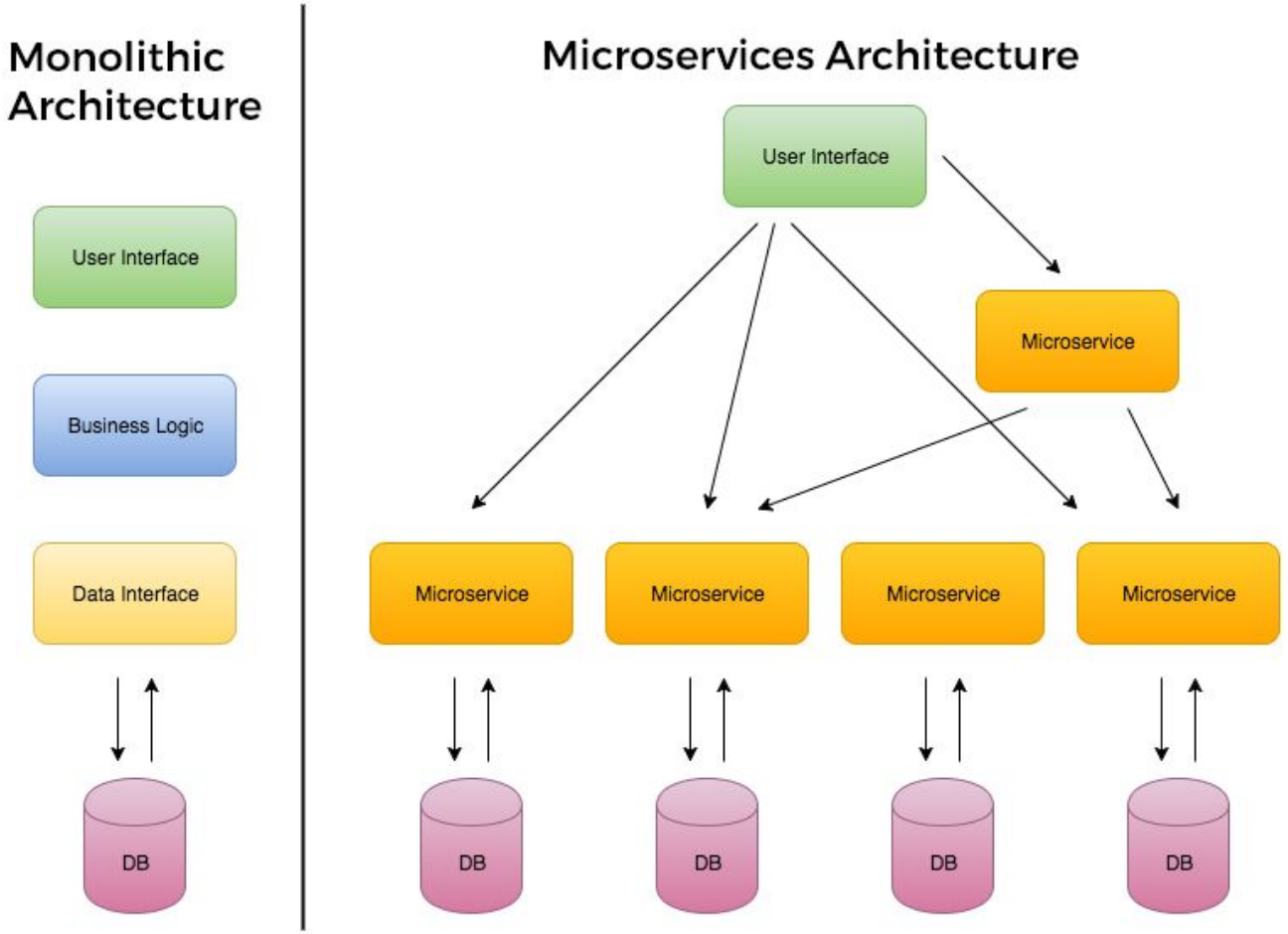
Микросервисная архитектура

Давайте рассмотрим пример микросервисной архитектуры для веб-приложения электронной коммерции.

- каталог товаров;
- корзина покупок;
- аутентификация и управление пользователями;
- оформление заказа;
- сервис оплаты.

Каждый из этих микросервисов может быть разработан, развернут и масштабирован независимо. Они взаимодействуют друг с другом через API, отправляя HTTP-запросы.

Микросервисная архитектура



Микросервисная архитектура

Слоистая архитектура (Layered Architecture):

- Разделяет систему на несколько уровней (слоев), такие как представление, бизнес-логика и доступ к данным.
- Обеспечивает четкое разделение ответственности и упрощает сопровождение и масштабирование системы.

Слоистая архитектура (Layered Architecture)

Слоистая архитектура (Layered Architecture) – это популярный подход к проектированию программных систем, в котором приложение разделяется на несколько логических слоев или уровней. Каждый слой выполняет определенную функцию и взаимодействует только с ближайшими слоями, что способствует организации кода, облегчает его понимание и обеспечивает модульность.

Слои в архитектуре могут варьироваться в зависимости от конкретной системы, но обычно включают следующие основные слои:

Слоистая архитектура (Layered Architecture)

Представление (Presentation Layer): Этот слой отвечает за представление данных пользователю и управление пользовательским интерфейсом. Здесь находятся компоненты, связанные с отображением данных, обработкой пользовательского ввода и взаимодействием с пользователем. Это может включать в себя веб-интерфейсы, графические интерфейсы пользователя (GUI) или API для взаимодействия с клиентскими приложениями.

Слоистая архитектура (Layered Architecture)

Бизнес-логика (Business Logic Layer): Этот слой содержит бизнес-логику приложения. Здесь происходит обработка данных, принятие бизнес-решений и взаимодействие с базой данных или другими источниками данных. Бизнес-логика управляет бизнес-процессами и бизнес-правилами, которые определяют, как приложение должно работать.

Слоистая архитектура (Layered Architecture)

Слой доступа к данным (Data Access Layer): Этот слой отвечает за доступ к данным и взаимодействие с хранилищами данных, такими как базы данных, файлы или внешние сервисы. Он обеспечивает абстракцию от конкретных источников данных и предоставляет бизнес-логике доступ к данным.

Слоистая архитектура (Layered Architecture)

Инфраструктурный слой (Infrastructure Layer): Этот слой содержит общие компоненты и утилиты, необходимые для функционирования приложения, такие как система управления конфигурацией, система логирования, аутентификация, кэширование и другие инфраструктурные аспекты.

Слоистая архитектура (Layered Architecture)

Преимущества слоистой архитектуры включают в себя:

Модульность: Каждый слой является независимым модулем, что упрощает разработку, тестирование и обслуживание.

Чистый дизайн: Разделение приложения на слои помогает соблюдать принципы чистой архитектуры и отделить бизнес-логику от деталей реализации.

Повторное использование: Слои могут быть повторно использованы в разных частях приложения или в разных проектах.

Масштабируемость: Каждый слой может быть масштабирован независимо, что облегчает оптимизацию производительности.

Понимание и обслуживание: Четкая организация слоев делает код более понятным и облегчает обслуживание и доработку приложения.

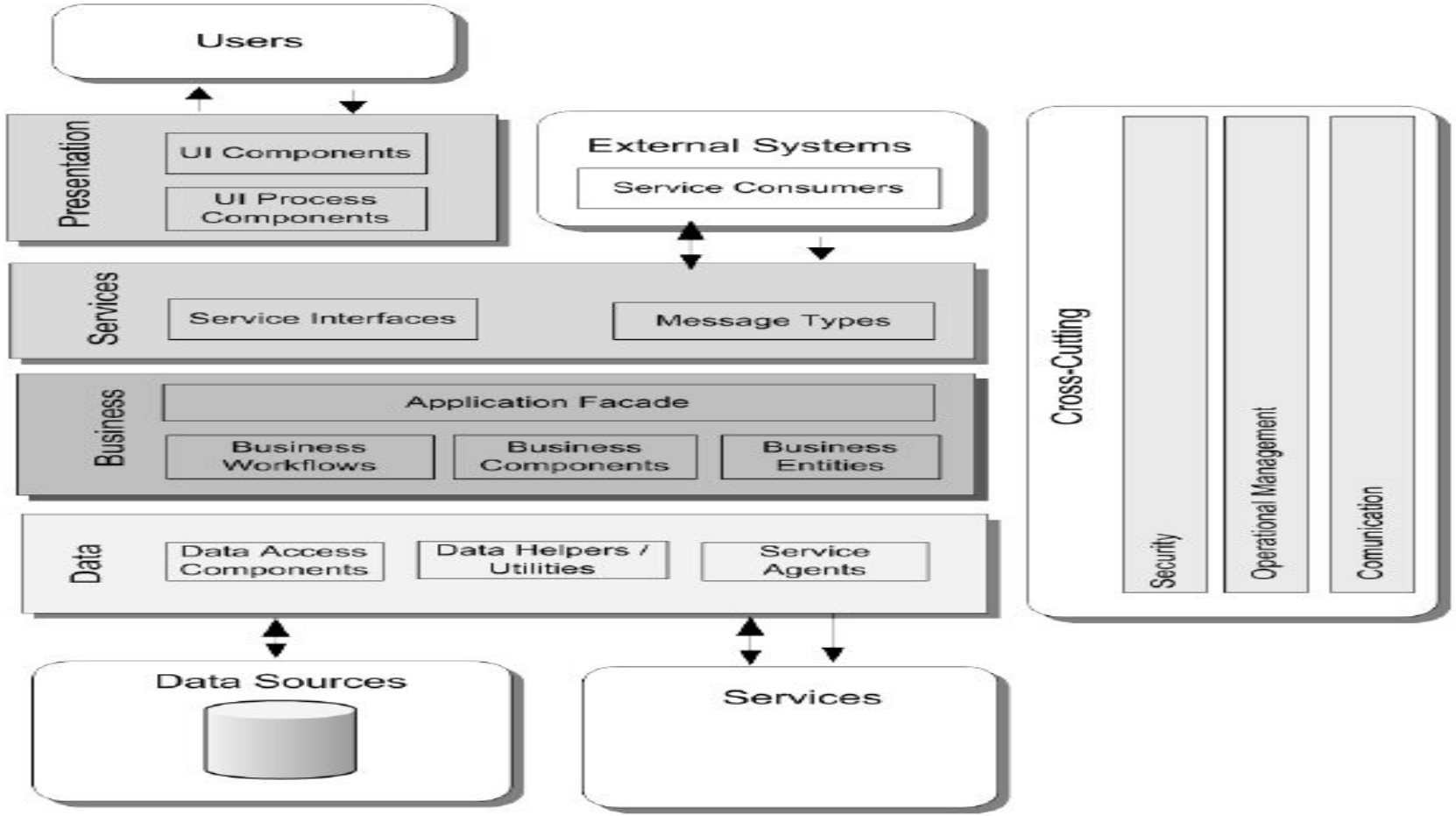
Слоистая архитектура (Layered Architecture)

Когда пользователь открывает веб-интерфейс и делает запрос на просмотр каталога товаров, запрос сначала попадает в представление.

Представление передает запрос на бизнес-логику, которая обрабатывает запрос, консультируясь с данными из слоя доступа к данным.

Бизнес-логика получает необходимые данные о товарах из базы данных, выполняет расчеты и формирует ответ, который отправляется обратно в представление для отображения пользователю.

Слоистая архитектура (Layered Architecture)



Слоистая архитектура (Layered Architecture)

Когда пользователь открывает веб-интерфейс и делает запрос на просмотр каталога товаров, запрос сначала попадает в представление.

Представление передает запрос на бизнес-логику, которая обрабатывает запрос, консультируясь с данными из слоя доступа к данным.

Бизнес-логика получает необходимые данные о товарах из базы данных, выполняет расчеты и формирует ответ, который отправляется обратно в представление для отображения пользователю.

Архитектура "Чистая архитектура" (Clean Architecture)

Архитектура "Чистая архитектура" (Clean Architecture):

- Определяет зависимости между компонентами системы так, чтобы бизнес-логика была независима от фреймворков и внешних библиотек.
- Поощряет высокую степень модульности и тестируемость кода.

Архитектура "Чистая архитектура" (Clean Architecture)

Архитектура "Чистая архитектура" (Clean Architecture) – это архитектурный подход, разработанный Робертом Мартином. Этот подход призван упростить проектирование, разработку и сопровождение программных систем, делая их более модульными, гибкими и тестируемыми. Цель "Чистой архитектуры" - изолировать бизнес-логику от зависимостей от фреймворков, библиотек и инфраструктуры.

Архитектура "Чистая архитектура" (Clean Architecture)

Принципы "Чистой архитектуры" включают:

Разделение ответственности (Separation of Concerns):

Подход "Чистой архитектуры" разделяет систему на различные уровни или слои, каждый из которых отвечает за определенную ответственность. Эти слои включают в себя:

- **Слой представления (Presentation Layer**): Отвечает за пользовательский интерфейс и взаимодействие с пользователем.
- **Слой бизнес-логики (Business Logic Layer)**: Содержит бизнес-правила и бизнес-логику приложения.
- **Слой данных (Data Layer)**: Обеспечивает доступ к данным, таким как базы данных или внешние API.

Архитектура "Чистая архитектура" (Clean Architecture)

Принципы "Чистой архитектуры" включают:

Зависимость наружу (Dependency Inversion):

Принцип инверсии зависимости предполагает, что высокоуровневые модули не должны зависеть от низкоуровневых модулей, а оба должны зависеть от абстракций. Это способствует уменьшению связности между компонентами и увеличивает гибкость системы.

Архитектура "Чистая архитектура" (Clean Architecture)

Принципы "Чистой архитектуры" включают:

Граничные объекты (Boundary Objects):

Архитектура "Чистой архитектуры" уделяет особое внимание граничным объектам, которые служат связующим звеном между внешним миром (например, пользовательским интерфейсом или базой данных) и внутренней бизнес-логикой. Граничные объекты позволяют изолировать внутреннюю логику от конкретных технологий и упрощают тестирование.

Архитектура "Чистая архитектура" (Clean Architecture)

Принципы "Чистой архитектуры" включают:

Использование презентационного шаблона (Presenter):

Для отделения бизнес-логики от пользовательского интерфейса часто используется презентационный шаблон (например, MVP - Model-View-Presenter или MVVM - Model-View-ViewModel). Это позволяет разработчикам создавать интерфейс, который независим от логики и бизнес-правил.

MVP (Model-View-Presenter) и MVVM (Model-View-ViewModel) - это два популярных архитектурных шаблона, используемых для разработки пользовательских интерфейсов в приложениях.

Архитектура "Чистая архитектура" (Clean Architecture)

Принципы "Чистой архитектуры" включают:

Соблюдение принципа единственной ответственности (Single Responsibility Principle):

Каждый модуль, класс или компонент должен иметь только одну причину для изменения. Этот принцип помогает уменьшить сложность и повысить поддерживаемость системы.

Архитектура "Чистая архитектура" (Clean Architecture)

Архитектура "Чистая архитектура" призвана сделать систему более гибкой, легко расширяемой и поддерживаемой, а также обеспечить ее независимость от конкретных технологий.

Архитектура "Чистая архитектура" (Clean Architecture)

Событийно-ориентированная архитектура (Event-Driven Architecture)

Событийно-ориентированная архитектура (Event-Driven Architecture):

- Основана на обмене сообщениями и событиями между компонентами системы.
- Позволяет построить высоко масштабируемые и отзывчивые системы.

Событийно-ориентированная архитектура (Event-Driven Architecture)

Событийно-ориентированная архитектура (Event-Driven Architecture, EDA) – это архитектурный подход, который ориентирован на обработку и передачу событий между компонентами системы. В EDA система строится вокруг генерации, передачи и обработки событий, что позволяет создавать более гибкие, масштабируемые и реактивные приложения.

Рассмотрим основные концепции и компоненты событийно-ориентированной архитектуры:

Событийно-ориентированная архитектура (Event-Driven Architecture)

События (Events):

События представляют собой сигналы или уведомления о произошедших действиях или изменениях в системе. Они могут быть сгенерированы различными компонентами приложения, такими как пользовательский интерфейс, сервисы, базы данных и т. д.

События обычно содержат информацию о событии (например, тип события и данные, связанные с ним) и могут быть асинхронно переданы другим компонентам.

Событийно-ориентированная архитектура (Event-Driven Architecture)

Источники событий (Event Sources):

Источники событий - это компоненты или системы, которые генерируют события. Примерами источников могут быть веб-приложения, базы данных и другие приложения.

Событийно-ориентированная архитектура (Event-Driven Architecture)

Брокеры событий (Event Brokers):

Брокеры событий (или событийные брокеры) - это компоненты, которые принимают события от источников и маршрутизируют их к соответствующим обработчикам событий. Брокеры событий выполняют функцию посредника и позволяют разным компонентам системы общаться между собой, не зная друг о друге напрямую.

Брокеры событий могут быть централизованными (например, системой сообщений) или децентрализованными (например, через публикацию-подписку).

Событийно-ориентированная архитектура (Event-Driven Architecture)

Обработчики событий (Event Handlers):

Обработчики событий - это компоненты или функции, которые реагируют на события и выполняют соответствующие действия. Они могут подписываться на определенные типы событий и реагировать на них.

Обработчики событий могут выполнять различные действия, такие как обновление базы данных, отправка уведомлений, запуск вычислительных процессов и т. д.

Событийно-ориентированная архитектура (Event-Driven Architecture)

Преимущества событийно-ориентированной архитектуры:

Гибкость и реактивность: EDA обеспечивает гибкую и реактивную модель разработки, позволяя системе мгновенно реагировать на события и изменения.

Масштабируемость: Событийная архитектура позволяет легко масштабировать систему, добавляя новые источники, брокеры и обработчики событий.

Отделение компонентов: EDA способствует отделению компонентов, позволяя им взаимодействовать через события, что делает систему более модульной и поддерживаемой.