Архитектура приложений

Основные подходы и виды

Виды архитектур

К проектированию системы важно подходить с точки зрения выбора правильного подхода к типу архитектуры.

Зачастую, выбор архитектуры сводится к совмещению нескольких различных вариантов.

Ниже приведены основные способы построения архитектуры:

1. Событийно-ориентированная архитектура (Event-Driven Architecture, EDA): приложение разбивается на отдельные компоненты, каждый из которых реагирует на определенные события и взаимодействует с другими компонентами через обмен сообщениями.
2. Сервер-клиентская архитектура (Client-Server Architecture): приложение разбивается на клиентскую часть, которая предоставляет пользовательский интерфейс, и серверную часть, которая обрабатывает запросы и управляет данными.
3. Архитектура, основанная на контейнерах (Container-Based Architecture): приложение разбивается на независимые контейнеры, каждый из которых выполняет определенную функцию и может масштабироваться независимо.

Виды архитектур

4. Слоистая архитектура (Layered Architecture): приложение разбивается на слои, каждый из которых выполняет определенную функцию и взаимодействует только с более близкими к нему слоями.

5. Архитектура, основанная на событийных потоках (Event-Streaming Architecture): приложение разбивается на отдельные компоненты, каждый из которых обрабатывает события и передает результаты другим компонентам через событийные потоки.

6. Архитектура, основанная на графах (Graph-Based Architecture): приложение представляет собой граф, в котором узлы представляют компоненты, а ребра - связи между ними.

Виды архитектур. Слоистая архитектура

Слоистая архитектура (Layered Architecture) - это подход к проектированию программных систем, в котором приложение разбивается на слои, каждый из которых отвечает за выполнение конкретной функциональности. Каждый слой является абстракцией и предоставляет определенный интерфейс для взаимодействия с другими слоями.

В слоистой архитектуре обычно выделяют три основных слоя:

1. **Представление (Presentation Layer)** - отвечает за отображение информации пользователю и обработку пользовательского ввода. Этот слой содержит интерфейс пользователя и логику, связанную с пользовательским интерфейсом.
2. **Бизнес-логика (Business Logic Layer)** - отвечает за обработку бизнес-логики приложения. Этот слой содержит логику приложения, которая обрабатывает запросы от пользователей и взаимодействует с базой данных.
3. **Источник данных (Data Access Layer)** - отвечает за доступ к данным и взаимодействие с базой данных. Этот слой содержит код, который получает данные из базы данных и обрабатывает их.

Виды архитектур. Слоистая архитектура

Слоистая архитектура имеет ряд преимуществ, включая:

- Модульность: слоистая архитектура позволяет легко добавлять новые модули и функции в приложение, а также изменять их без влияния на другие слои.
- Удобство тестирования: каждый слой может быть протестирован отдельно, что упрощает процесс тестирования и обеспечивает более высокое качество приложения.
- Легкость сопровождения: изменения в одном слое не влияют на другие слои, что упрощает сопровождение приложения и добавление новых функций.

Однако, слоистая архитектура может иметь и некоторые недостатки, например, если слои слишком тесно связаны между собой, это может привести к затруднениям в разработке и поддержке приложения.

Также слоистая архитектура может не быть подходящей для больших и сложных приложений, где требуется большая гибкость и масштабируемость.

Виды архитектур

Отдельно выделяют два основных подхода при выборе архитектуры.

Это монолитная и микросервисная архитектуры.

Монолитная архитектура - это тип архитектуры программного обеспечения, при котором все компоненты приложения находятся в одной кодовом базе и выполняются как единое приложение.

Микросервисная архитектура - это тип архитектуры программного обеспечения, при котором приложение разбивается на небольшие, автономные сервисы, каждый из которых отвечает за выполнение определенной функции.

Виды архитектур. Монолитная архитектура

В монолитной архитектуре все функциональные блоки приложения (например, пользовательский интерфейс, бизнес-логика, хранение данных) находятся в одной монолитной кодовом базе и работают в одном процессе операционной системы.

Такой подход обеспечивает простоту развертывания и масштабирования приложения, так как все компоненты находятся в одном месте.

Однако, этот подход может приводить к проблемам, связанным с размером и сложностью кодовой базы, так как все компоненты находятся в одном месте.
Это может затруднять поддержку и разработку приложения на долгосрочной основе.

Монолитная архитектура часто используется в небольших и средних проектах, где скорость разработки и простота развертывания являются более приоритетными, чем гибкость масштабирования и поддержки.

Виды архитектур. Монолитная архитектура

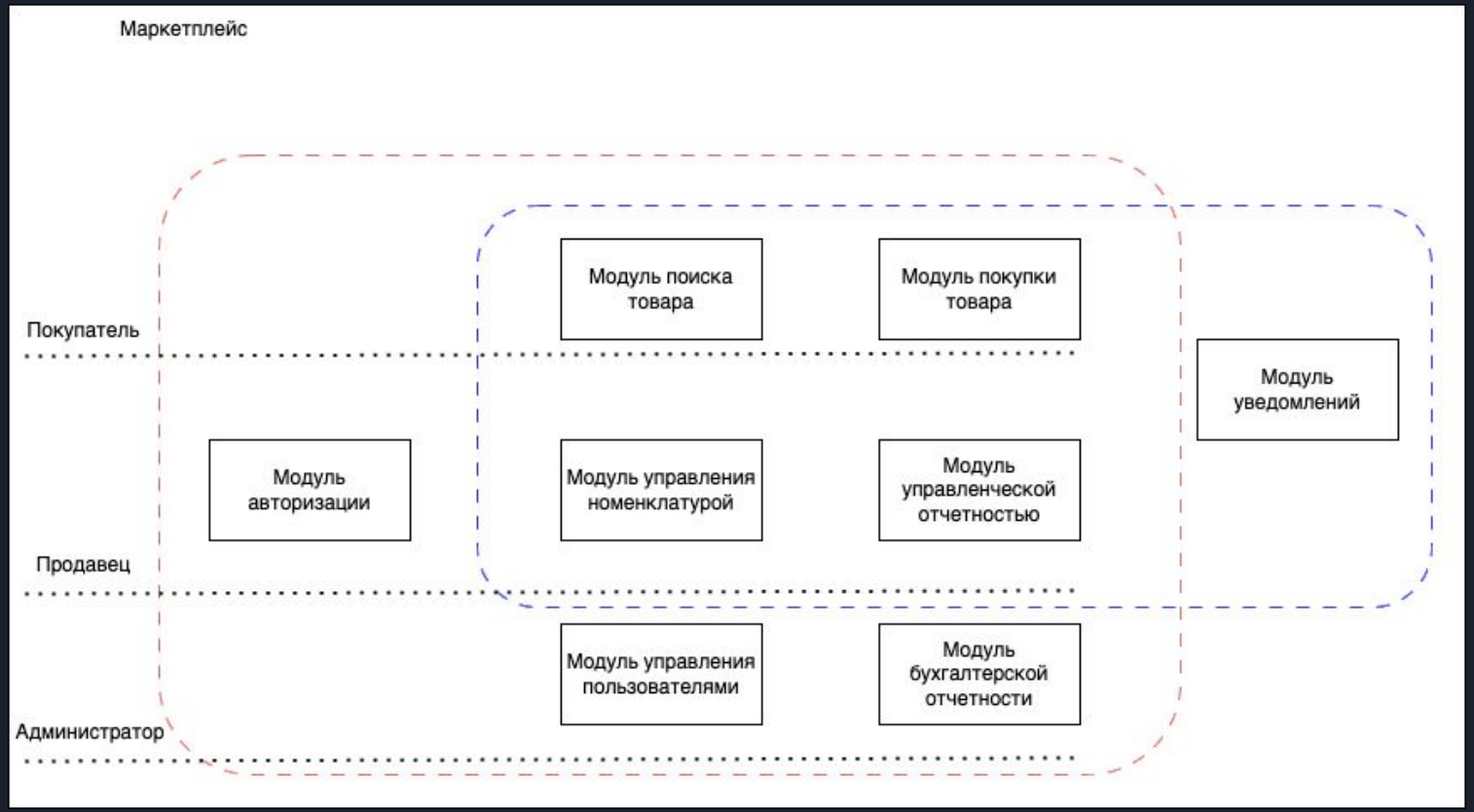
Один из примеров монолитной архитектуры может быть традиционное трехслойное веб-приложение, состоящее из следующих компонентов:

1. Пользовательский интерфейс (UI): это слой, который отображает информацию на веб-странице и взаимодействует с пользователем.
2. Бизнес-логика: это слой, который обрабатывает запросы от пользователей, выполняет логические операции, работает с базами данных и выдает ответы на UI слой.
3. Слой данных: это слой, который содержит базу данных приложения, используемую бизнес-логикой для хранения и получения данных.

В монолитной архитектуре все эти компоненты находятся в одной кодовой базе и работают в одном процессе операционной системы. Это означает, что все компоненты могут обмениваться данными и вызывать друг друга напрямую без использования сетевых протоколов.

Такой подход может быть применен в небольших и средних веб-приложениях, которые не нуждаются в высокой гибкости масштабирования и поддержки.

Виды архитектур. Монолитная архитектура



Виды архитектур. Микросервисная архитектура

В микросервисной архитектуре каждый сервис работает в своем собственном процессе и взаимодействует с другими сервисами через сетевые протоколы, такие как HTTP или Message Queuing.

Такой подход обеспечивает гибкость, масштабируемость и легкость развертывания приложения, так как каждый сервис можно масштабировать и развертывать независимо от других сервисов.

Кроме того, это позволяет разработчикам использовать различные языки программирования и технологии для каждого сервиса, что может увеличить эффективность и гибкость разработки.

Однако, микросервисная архитектура может приводить к сложности в управлении и координации сервисов между собой, а также к проблемам при обработке транзакций и обеспечении целостности данных.

Кроме того, этот подход может быть избыточным для небольших проектов, где простота развертывания и управления являются более важными, чем гибкость масштабирования и поддержки.

Виды архитектур. Микросервисная архитектура

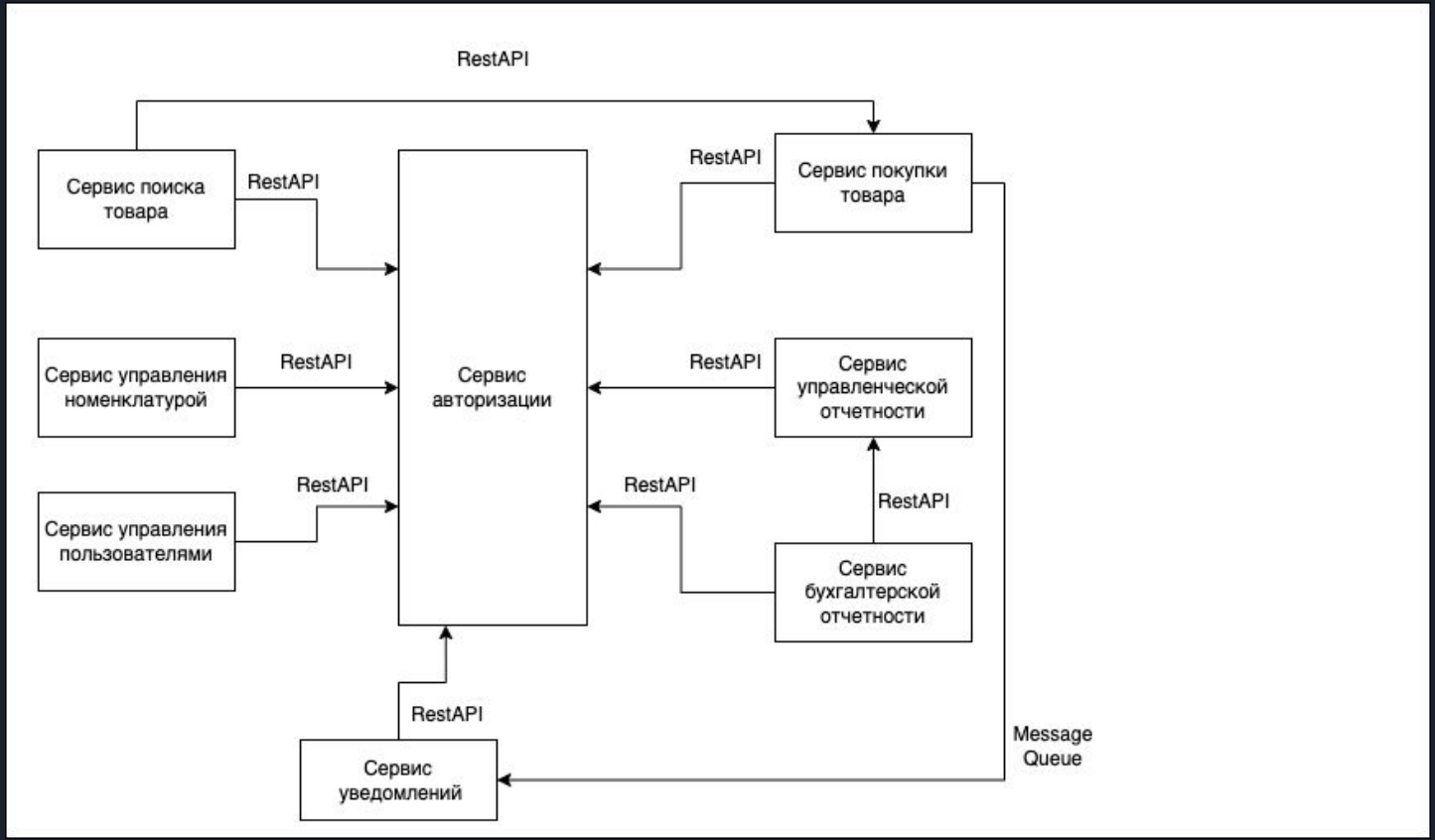
Чтобы преобразовать трехслойное веб-приложение в микросервисную архитектуру, мы можем разбить его на несколько небольших сервисов, каждый из которых будет отвечать за выполнение отдельной функции. Например:

1. Сервис пользовательского интерфейса (UI): этот сервис будет отображать информацию на веб-странице и обрабатывать взаимодействие с пользователем.
2. Сервис бизнес-логики: этот сервис будет обрабатывать запросы от пользователей, выполнять логические операции и взаимодействовать с другими сервисами для получения необходимых данных.
3. Сервис базы данных: этот сервис будет отвечать за хранение данных и обеспечивать доступ к ним для других сервисов.

В микросервисной архитектуре каждый сервис работает в своем собственном процессе и взаимодействует с другими сервисами через сетевые протоколы.

Такой подход позволяет легко масштабировать и развертывать каждый сервис независимо от других, а также использовать различные языки программирования и технологии для каждого сервиса. Однако, это также приводит к сложностям в управлении и координации сервисов между собой.

Виды архитектур. Микросервисная архитектура



Виды архитектур. Монолитный ад

Монолитный ад - это термин, который используется для описания проблем, связанных с монолитными архитектурами. В монолитной архитектуре все компоненты приложения находятся в одной кодовом базе и работают в одном процессе операционной системы. Это может привести к следующим проблемам:

1. **Сложность масштабирования**: монолитные приложения сложно масштабировать, поскольку все компоненты находятся в одном процессе. Это означает, что вы не можете масштабировать каждый компонент независимо, а вынуждены масштабировать всю систему целиком.
2. **Ограниченная гибкость**: монолитные приложения имеют ограниченную гибкость, поскольку все компоненты находятся в одном кодовом базе. Это значит, что вы не можете использовать различные технологии или языки программирования для каждого компонента, что может ограничивать возможности разработки.
3. **Трудность в обновлении**: обновление монолитных приложений может быть сложным, поскольку все компоненты находятся в одном кодовом базе. Это означает, что обновление одного компонента может повлиять на другие компоненты, что может привести к нежелательным последствиям.

Виды архитектур. Монолитный ад

Решение?

Начать выделять из системы компоненты, которые могут и должны существовать, как независимые сервисы и выполнять одну узконаправленную функцию.

Что для этого нужно?

- Документация
- Аналитика
- Ключевые метрики
- Стек технологий

Микросервисная архитектура. REST API

REST (Representational State Transfer) API - это архитектура для создания веб-сервисов, которая предоставляет способ обмена данными между клиентом и сервером. REST API позволяет использовать HTTP-протокол для обмена данными в формате JSON или XML.

Основными принципами архитектуры REST являются:

1. Клиент-серверная архитектура. Это означает, что клиент и сервер разделены, что позволяет им развиваться и изменяться независимо друг от друга.
2. Без состояния. Это означает, что сервер не хранит информацию о состоянии клиента между запросами. Каждый запрос рассматривается как новый запрос.
3. Кэширование. Клиенты могут кэшировать ответы на запросы, что уменьшает количество запросов к серверу.
4. Единообразный интерфейс. Клиенты и серверы взаимодействуют через единообразный интерфейс, что упрощает разработку, улучшает масштабируемость и уменьшает сложность.
5. Слои. Клиенты не знают о слоях, через которые проходят запросы, что повышает устойчивость архитектуры.

Микросервисная архитектура. REST API

Одним из основных преимуществ REST API является его универсальность.

REST API может быть использован на любой платформе и с любым языком программирования, что позволяет создавать гибкие и масштабируемые приложения.

Для создания REST API необходимо определить ресурсы, которые будут доступны клиентам, и методы, которые клиенты могут использовать для работы с этими ресурсами. Например, GET-запрос может использоваться для получения данных о ресурсе, а POST-запрос для создания нового ресурса.

Однако, при разработке REST API есть несколько важных факторов, которые следует учитывать.
В частности, необходимо обеспечить безопасность передаваемых данных, предусмотреть механизмы аутентификации и авторизации, и следить за производительностью и масштабируемостью.