Задание:

Проанализируйте ключевые требования системы ZooTicket
Выберите подходящий архитектурный стиль для реализации системы.

Обоснуйте свой выбор:
- Почему выбранный стиль подходит для системы ZooTicket.
- Какие преимущества он даёт для пользователей и администраторов.
- Как выбранный стиль влияет на масштабируемость, интеграции и отказоустойчивость.

Сравните с другими стилями:
- Какие альтернативные архитектуры могли бы быть использованы.
- В чем их преимущества и недостатки по сравнению с выбранным стилем.

Анализ ключевых требований системы ZooTicket

На основе типичных требований для подобных систем можно выделить следующие ключевые характеристики:

1. **Различные группы пользователей:** Система должна обслуживать разных пользователей через разные интерфейсы:
    - **Покупатели (Посетители зоопарка):** Мобильное приложение (iOS/Android) и Веб-сайт для покупки билетов(браузеры Google, Яндекс, Safary, Firefox, Opera).
    - **Администраторы, специалисты технической поддержки, кассиры:** Десктопные или веб-приложения для управления билетами, проверки на входе, аналитики.
2. **Высокая пиковая нагрузка:** Продажи билетов могут резко возрастать в выходные дни, праздники или в период проведения акций. Система должна быть готова к скачкам трафика.
3. **Интеграции:** Требуется интеграция с:
    - Платежными шлюзами (банковские карты, MirPay, СБП).
    - Системами email/SMS-рассылок для отправки квитанций и QR-кодов.
    - С системами аналитики и CRM.
4. **Отказоустойчивость и доступность:** Падение системы в час пик означает прямые финансовые потери и недовольство клиентов. Критически важные компоненты (например, процесс покупки) должны быть максимально доступны.
5. **Масштабируемость:** Система должна иметь возможность расти вместе с бизнесом (открытие новых филиалов, увеличение потока посетителей) без полной перестройки архитектуры.
6. **Безопасность:** Необходима безопасная обработка платежных данных и защита от мошенничества.

Выбор архитектурного стиля: Микросервисная архитектура (Microservices)

Обоснование выбора:

Микросервисная архитектура идеально подходит для ZooTicket, так как позволяет разбить систему на небольшие, слабосвязанные сервисы, каждый из которых отвечает за свою бизнес-область.

Предполагаемый набор микросервисов для ZooTicket:

- **User Service:** Управление пользователями, аутентификация и авторизация.
- **Ticket Catalog Service:** Управление типами билетов, ценами, акциями.
- **Order Service:** Оформление заказов, управление корзиной.
- **Payment Service:** Взаимодействие с внешними платежными шлюзами.
- **Notification Service:** Отправка email и SMS с билетами.
- **Validation Service:** Проверка QR-кодов на входе.

Почему этот стиль подходит для системы ZooTicket:

1. **Разработка и развертывание:** Разные команды могут параллельно разрабатывать и независимо развертывать сервисы. Например, можно обновить Payment Service, не трогая Ticket Catalog Service.
2. **Технологическая гетерогенность:** Каждый сервис может быть написан на языке, наиболее подходящем для его задач (например, Python для аналитики, Go для высоконагруженного Validation Service, Java для основного бизнес-логика).
3. **Изоляция сбоев:** Если сервис уведомлений временно недоступен, это не должно блокировать процесс покупки билета. Заказ будет создан, а уведомление отправлено позже, когда сервис восстановится.

Преимущества для пользователей и администраторов:

- **Для пользователей:**
    - **Высокая доступность:** Система реже "падает" полностью.
    - **Быстрый отклик:** Критически важные сервисы (покупка, проверка билета) можно масштабировать отдельно для обеспечения скорости.
    - **Постоянное улучшение:** Новые функции и исправления появляются быстрее и без полного обновления системы.
- **Для администраторов:**
    - **Гибкость управления:** Легко добавлять новые типы билетов, акции или подключать новые платежные системы.
    - **Детальный мониторинг:** Можно отслеживать производительность и ошибки каждого сервиса в отдельности, быстро выявляя "узкие места".
    - **Упрощенное обслуживание:** Можно "выключать" некритичные сервисы для технического обслуживания, не затрагивая всю систему.

Влияние на масштабируемость, интеграции и отказоустойчивость:

- **Масштабируемость: Горизонтальное масштабирование.** Если сервис проверки билетов на входе испытывает высокую нагрузку утром, мы можем добавить дополнительные экземпляры только этого сервиса, а не всей системы. Это экономически эффективно.

- **Интеграции: Гибкость.** Новые интеграции (например, с системой лояльности) реализуются в виде нового микросервиса или как часть существующего (например, Order Service). Внешние системы взаимодействуют с API-шлюзом (API Gateway), который является единой точкой входа и маршрутизирует запросы к нужным сервисам.
- **Отказоустойчивость: Повышенная.** Благодаря изоляции сервисов, сбой в одном из них (например, Notification Service) не приводит к каскадному отказу всей системы.

Сравнение с другими архитектурными стилями

Сравнение 1: Монолитная архитектура

- **Описание:** Вся система (UI, бизнес-логика, доступ к данным) развертывается как единое целое.
- **Преимущества:**
  - **Простота:** На начальном этапе разработки проще спроектировать, разработать и развернуть.
  - **Производительность:** Внутримонолитные вызовы обычно быстрее, чем сетевые вызовы между микросервисами.
- **Недостатки по сравнению с микросервисами:**
  - **Сложность изменений:** Небольшое изменение в коде требует пересборки и повторного развертывания всего приложения.
  - **Масштабируемость:** Можно масштабировать только всю систему целиком ("вертикальное масштабирование" или клонирование монолита), что менее гибко и дорого.
  - **Технологический стек:** Привязанность к одному стеку технологий.
  - **Надежность:** Ошибка в небольшом модуле может "положить" все приложение.
  - **Для ZooTicket:** Монолит быстро станет неуправляемым при необходимости быстрых изменений и интеграций, а пиковые нагрузки будут проблемой.

Сравнение 2: Архитектура на основе событий (EDA)

- **Описание:** Компоненты системы общаются друг с другом путем асинхронной отправки и получения событий через шину сообщений (Kafka, RabbitMQ).
- **Преимущества:**
  - **Слабая связанность:** Сервисы ничего не знают друг о друге, общаясь только через события.
  - **Высокая отказоустойчивость и эластичность:** Сервисы могут работать и обрабатывать события в своем темпе.
- **Недостатки по сравнению с микросервисами:**
  - **Сложность:** Значительно сложнее в проектировании, отладке и обеспечении согласованности данных.
  - **Латентность:** Асинхронная обработка может быть неприемлема для операций, требующих немедленного ответа (например, проверка билета).

- **Для ZooTicket:** EDA может быть отличным **дополнением** к микросервисам. Например, после успешной оплаты Order Service публикует событие OrderConfirmed, которое слушают Notification Service (отправляет билет) и Analytics Service (обновляет статистику). Но для синхронных сценариев (поиск билетов, оплата) лучше подходит прямой API-вызов.

Итоговое решение

Для системы **ZooTicket** рекомендуется использовать **Микросервисную архитектуру**, усиленную элементами **EDA** для асинхронных сценариев.

Эта гибридная модель позволяет:

- Использовать **синхронные API-вызовы** для взаимодействия с клиентскими приложениями, где требуется немедленный ответ (просмотр каталога, оплата).
- Использовать **асинхронные события** для фоновых и слабосвязанных задач (отправка уведомлений, обновление аналитики, синхронизация данных между сервисами).

Такой подход обеспечивает наилучший баланс между гибкостью, производительностью, масштабируемостью и надежностью, что важно для развития системы продаж билетов.