WEB Architecture

Уровень представления PL (Presentation Layer):

***PL** отображает пользовательский интерфейс и упрощает взаимодействие с пользователем. Уровень представления имеет компоненты пользовательского интерфейса, которые визуализируют и показывают данные для пользователей.*

Также существуют компоненты пользовательского процесса, которые задают взаимодействие с пользователем. PL предоставляет всю необходимую информацию клиентской стороне.

Основная цель уровня представления - получить входные данные, обработать запросы пользователей, отправить их в службу данных и показать результаты.

Слой бизнес логики BLL (Business Logic Layer):

***BLL** несет ответственность за надлежащий обмен данными. Этот уровень определяет логику бизнес-операций и правил. Вход на сайт - это пример уровня бизнес-логики.*

Уровень службы данных DSL (Data Service Layer):

***DSL** передает данные, обработанные уровнем бизнес-логики, на уровень представления. Этот уровень гарантирует безопасность данных, изолируя бизнес-логику со стороны клиента*.

Уровень доступа к данным DAL (Data Access Layer):

***DAL** предлагает упрощенный доступ к данным, хранящимся в постоянных хранилищах, таких как двоичные файлы и файлы XML. Уровень доступа к данным также управляет операциями CRUD - создание, чтение, обновление, удаление.*

Особенности тестирования монолитных и микросервисных веб-приложений:

Коротко о системах:

Традиционная монолитная архитектура веб-приложения состоит из трех частей - базы данных, клиентской и серверной сторон. Это означает, что внутренняя и внешняя логика, как и другие фоновые задачи, генерируются в одной кодовой базе. Чтобы изменить или обновить компонент приложения, разработчики программного обеспечения должны переписать все приложение.

Что касается микросервисов, этот подход позволяет разработчикам создавать веб-приложение из набора небольших сервисов. Разработчики создают и развертывают каждый компонент отдельно.

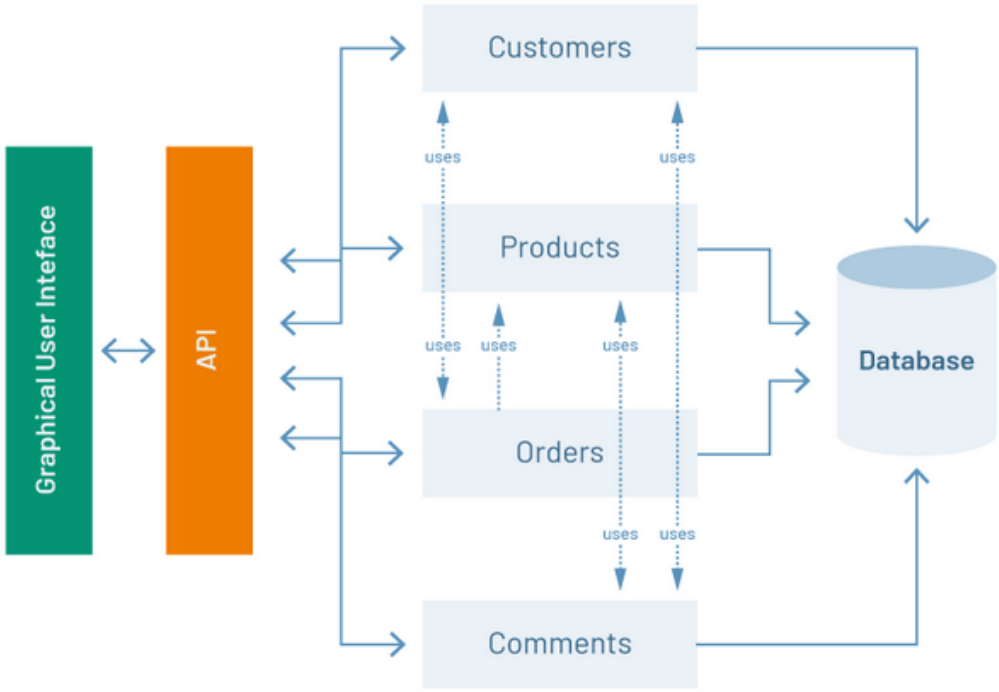
Плюсы монолитной системы:

- ***Простое развертывание**. Использование одного исполняемого файла или каталога упрощает развертывание;*
- ***Разработка**. Приложение легче разрабатывать, когда оно создано с использованием одной базы кода;*
- ***Производительность.** В централизованной базе кода и репозитории один интерфейс API часто может выполнять ту функцию, которую при работе с микросервисами*
  выполняют многочисленные API;
- ***Упрощенное тестирование.** Монолитное приложение представляет собой единый централизованный модуль, поэтому сквозное тестирование можно проводить быстрее, чем при использовании распределенного приложения;*
- ***Удобная отладка.** Весь код находится в одном месте, благодаря чему становится легче выполнять запросы и находить проблемы.*

Минусы монолитной системы:

- ***Снижение скорости разработки**. Большое монолитное приложение усложняет и замедляет разработку;*

- ***Масштабируемость**. Невозможно масштабировать отдельные компоненты;*
- ***Надежность**. Ошибка в одном модуле может повлиять на доступность всего приложения;*
- ***Препятствия для внедрения технологий**. Любые изменения в инфраструктуре или языке разработки влияют на приложение целиком, что зачастую приводит к увеличению стоимости и временных затрат;*
- ***Недостаточная гибкость**. Возможности монолитных приложений ограничены используемыми технологиями;*
- ***Развертывание**. При внесении небольшого изменения потребуется повторное развертывание всего монолитного приложения.*



Плюсы микросервисной системы:

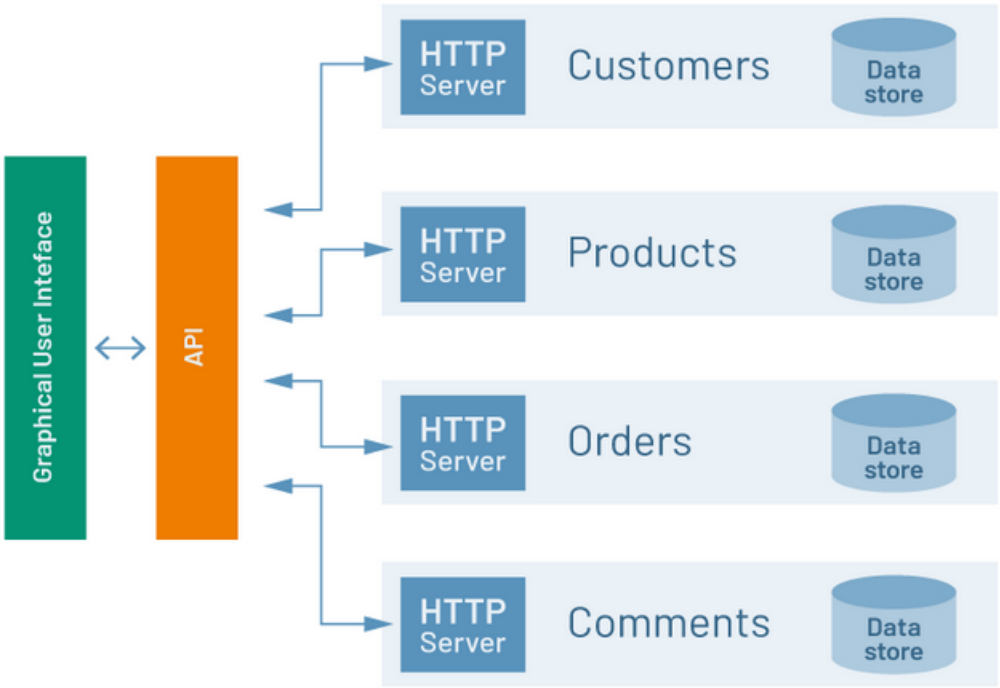
- ***Гибкое масштабирование**. Когда микросервис достигает предельной нагрузки, можно быстро выполнить развертывание новых экземпляров данной службы в сопутствующем кластере и снизить нагрузку;*
- ***Непрерывное развертывание**. Регулярные и ускоренные циклы релиза;*

- ***Легкость обслуживания и тестирования**. Команды могут экспериментировать с новыми функциями и возвращаться к предыдущей версии, если что-то не работает. Это упрощает обновление кода и ускоряет выпуск новых функций на рынок. Кроме того, в отдельных службах легко находить и исправлять ошибки и баги;*
- ***Независимое развертывание**. Микросервисы представляют собой отдельные модули, поэтому с ними можно легко и быстро выполнять независимое развертывание отдельных функций;*
- ***Гибкость технологий**. При использовании архитектуры микросервисов команды могут выбирать инструменты с учетом своих предпочтений;*
- ***Высокая надежность**. Развертывая изменения для конкретной службы, можно не бояться, что приложение выйдет из строя целиком;*
- ***Довольные команды**. Команды, работающие с микросервисами, гораздо лучше*
  отзываются о своей работе благодаря автономности и возможности самостоятельно создавать и развертывать приложения, не дожидаясь одобрения запроса pull в течение нескольких недель.

Минусы микросервисной системы:

- ***Разрастание процесса разработки**. Микросервисы усложняют работу по сравнению с монолитной архитектурой, поскольку в различных местах возникает все больше служб, созданных несколькими командами. Если разрастание не контролируется должным образом, оно приводит к замедлению разработки и снижению операционной эффективности;*
- ***Экспоненциальный рост расходов на инфраструктуру**. У каждого нового микросервиса может быть своя стоимость комплекта тестов, инструкций по развертыванию, инфраструктуры хостинга, инструментов мониторинга и т. д.;*
- ***Дополнительные организационные расходы**. Командам требуется дополнительный уровень коммуникации и сотрудничества, чтобы координировать работу над обновлениями и интерфейсами*;
- ***Проблемы при отладке**. У каждого микросервиса свой набор журналов, что усложняет отладку. Кроме того, дополнительные затруднения могут возникать в том случае, когда один бизнес-процесс выполняется на нескольких машинах;*

- ***Отсутствие стандартизации***. *Без общей платформы может возникнуть ситуация, в которой расширяется список языков, стандартов ведения журналов и средств мониторинга;*
- ***Отсутствие ясности в вопросах владения***. *По мере появления новых служб увеличивается и количество работающих над ними команд. Со временем становится сложнее определить, какие службы команда может использовать и к кому следует обращаться за поддержкой.*



Тестирование микросервисных архитектур:

При работе с микросервисами зачастую используется гибридный подход к тестированию, как такового QA отдела может и не существовать.

Большая часть тестов проводится разработчиками и код пишется соответствующим образом:

- понимание эксплуатационной *семантики* приложения;
- понимание эксплуатационных *характеристик* зависимостей;
- написание отлаживаемого кода.

Эксплуатационная семантика приложения:

Это понятие предполагает написание кода, во время которого
вы озадачиваетесь следующими вопросами:

- как осуществляется деплой сервиса, с помощью каких инструментов;
- сервис забинден на порт 0 или на стандартный порт?
- как приложение обрабатывает сигналы?
- как стартует процесс на выбранном хосте;
- как сервис регистрируется в *service discovery*?
- как сервис обнаруживает апстримы?
- как сервису отключают коннекты, когда он собирается завершиться;
- должен ли производиться *graceful restart* или нет;
- как конфиги — статические и динамические — скармливаются процессу;
- модель конкурентности приложения (многопоточное, или чисто однопоточное, *event driven*, или на акторах, или гибридной модели);
- каким образом реверс-прокси на фронте приложения держит соединения (пре-форк или потоки или процессы).

Отлаживаемый код:

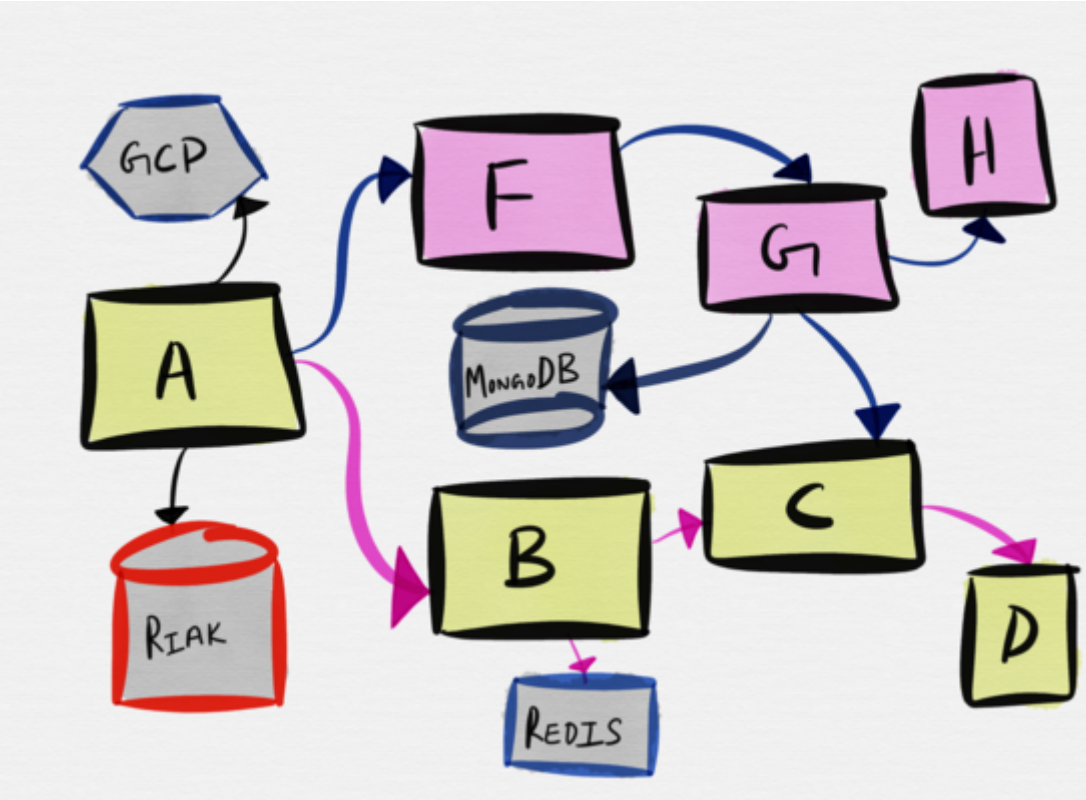
- хорошую степень инструментации кода;
- понимание выбранного формата наблюдаемости — метрик, логов, трекеров ошибок, трейсов и их комбинации, — а также его плюсов и минусов;
- *умение выбрать наилучший формат наблюдаемости, учитывая требования*
  конкретного сервиса, эксплуатационные нюансы зависимости и хорошую инженерную интуицию.

Тестирование в пре-продакшне:

Это наилучшая возможная верификация корректности системы, и вместе с тем - наилучшая возможная симуляция известных видов отказов.

- Unit-тестирование;
- Интеграционные тесты;
- UI тесты;

- End-2-End тесты.



Применяется метод “Двойника” или “Фейка”. Взаимодействие сервиса A с сервисом B включает коммуникацию сервиса B с Redis и сервисом C. Однако, наименьшей тестируемой единицей здесь является взаимодействие сервиса A с сервисом B, и простейший способ простестировать это взаимодействие - это развернуть фейк для сервиса B и тестировать взаимодействие сервиса A с фейком. Тестирование контрактов может быть особенно полезным для тестирования подобных интеграций.

Сервис A также сообщается с Riak. Минимальная тестируемая единица в данном случае включает в себя реальное сообщение между сервисом A и Riak, поэтому имеет смысл развертывание локального инстанса Riak на время тестирования.

Для повышения скорости разработки, при интеграционном тестировании, особое внимание уделяется соблюдению безопасности, целостности данных и т. п. Остальные же интеграционные тесты проводятся на “живом” трафике.

Как строить тестирование микросервисов

Тестирование микросервисов — один из важнейших разделов работы с ними. Вдумчивый подход к тестированию обеспечивает экономию времени и ресурсов при отработке процессов. Важно уделять время юнит-тестированию либо компонентному тестированию, так как не все процессы можно проверить на интеграционном уровне. Разные уровни тестирования должны сочетаться между собой, чтобы покрывать разную логику проверки микросервиса.

Виды тестирования

Модульное тестирование (или юнит-тестирование). Самый простой вид быстрого тестирования, направленный на один компонент за один раз.

Интеграционное тестирование. Программные модули тестируют как группу после объединения для выявления багов взаимодействия.

Регрессионное тестирование. Проверка изменений, сделанных в приложении или окружающей среде, для подтверждения ранее существующей и по-прежнему работающей функциональности.

Тестирование производительности. Для определения скорости работы вычислительной системы или ее части. Проверяет non function requirements (требования, не относящиеся к функциональности). Мы измеряем нагрузку для определения основных показателей ЦПУ и памяти сервиса. Используется и для тестирования каналов связи.

Программные средства для тестирования

PostMan и Swagger. Rest-клиенты протокола общения для ручного тестирования.

JavaScript. Для автоматизации тестирования.

GitLab CI. Автоматический запуск тестирования.

Allure. Фреймворк Яндекса для просмотра красивых отчетов.

Jaeger. Чтобы подружить между собой несколько микросервисов и проверить их полную интеграцию. Показывает точку входа клиента на сайт и всю цепочку его передвижений до конечного сервиса. Тестировщик может оперативно отследить проблемы на этом пути.

Что важно учесть при тестировании микросервисов

Взаимопонимание. Часто некоторые задумки во взаимодействии микросервисов реализуются не так, как хотелось на старте. Оказывается, менеджеры и разработчики просто неправильно поняли друг друга. Крайне важно, чтобы тестировщик и разработчик обсуждали и договаривались, как будет тестироваться задача. Какую логику покроет юнит-тестирование, а какую — более высокоуровневые проверки.

Тестовые данные. Это больная история микросервисов. Их нужно сгенерировать в достаточном количестве, что иногда трудозатратно. К тому же, нужно одновременно генерировать новые и поддерживать старые данные.

Автотестирование. Его нужно внедрить и постоянно совершенствовать. Этот процесс значительно экономит время разработчиков и тестировщиков. Когда мы только запускали микросервисы, на ручное тестирование регресса уходила неделя. Сейчас нужно около 25 минут.

Учитывать цепочки. Особенность тестирования микросервисов в том, что их нужно проверять в цепочке. Это усложняет процесс. Для успешного тестирования нужно знать все контракты, которые прописывают общение между микросервисами.

Здравый смысл. Микросервисы нужны не всем компаниям. Есть и категорические противники микросервисов вроде компании Badoo. Они решили, что раздробились слишком сильно и не сгруппировали новые службы. Поэтому в разработке и тестировании микросервисов важно искать баланс между пользой для бизнеса и затрачиваемыми ресурсами.