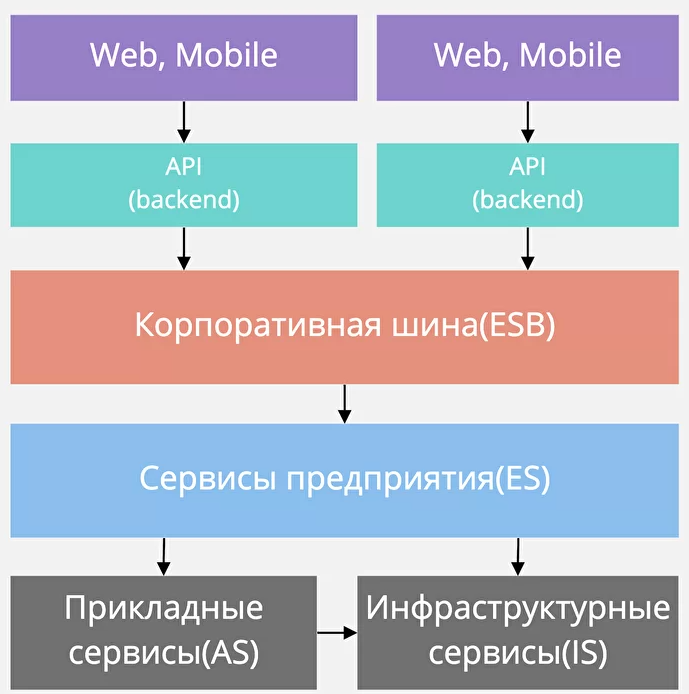
**SOA (***service-oriented architecture***). Сервисно-ориентированная архитектура - модульный** подход к разработке программного обеспечения, основанный на использовании **распределенных, слабо связанных (loose coupling)** **заменяемых компонентов**, оснащенных стандартизированными интерфейсами для взаимодействия по **стандартизированным протоколам**.

По сути SOA – это когда в архитектуре есть и микросервисы и монолиты

**SOA vs MSA (Microservices)**

**SOA**

В его основе лежат несколько основных идей – переиспользование сервисов и корпоративная шина. Разработчики стремятся разбить систему на сервисы таким образом, чтобы их можно было использовать повторно. Взаимодействие и маршрутизация осуществляется через корпоративную шину ESB. Типичная SOA архитектура показана на рисунке ниже.

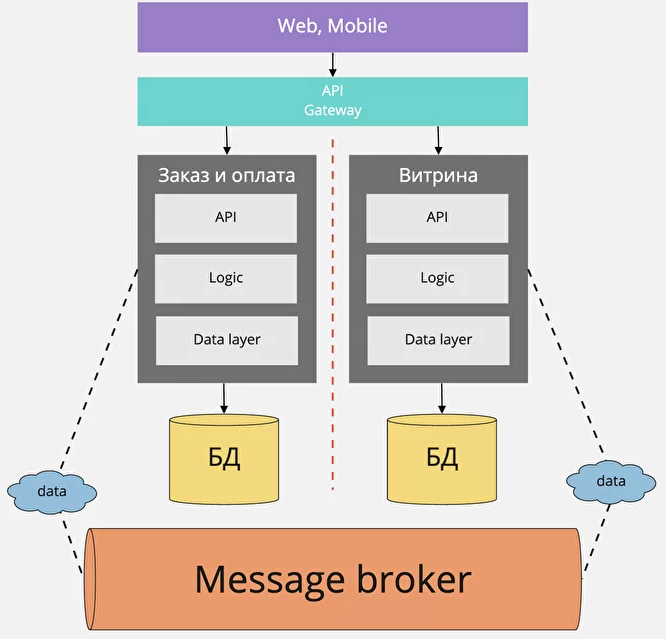


* **Инфраструктурные сервисы (infrastructure services)**: группа легко переиспользуемых сервисов, таких как аутентификация/авторизация, отправка смс и прочее.
* **Прикладные сервисы (application services)**: не могут быть переиспользованны под разные задачи, так как ограничены определенным прикладным контекстом, но их можно встраивать в более высокоуровневые сервисы.
* **Сервисы предприятия (Enterprise services)**: эти сервисы отвечают за реализацию крупных частей бизнес процессов компании, они потребляют более низкоуровневые сервисы.
* **Шина(ESB)**: в случае взаимодействия сложных событий действует как посредник и управляет различными рутинными операциями, такими как передача сообщений и координация вызовов.
* **API**: по сути это бэкенды, предоставляющие API, доступное в интернет, для сайтов и мобильных приложений компании. Они взаимодействуют с ESB и раскрывают функциональность для конечных потребителей.

Как вы могли заметить, проектируя архитектуру в данном стиле, мы имеем очень большое количество слоев и как следствие команд, которые ими владеют. Любой запрос пронизывает все слои системы, в большей степени напоминая монолитную архитектуру. Но данный тип архитектуры намного сложнее, потому что является распределенной архитектурой

**MSA**

Микросервисы в отличие от SOA, наоборот, избегают повторного использования, применяя философию - предпочтительнее дублирование, а не зависимость от других сервисов. Повторное использование предполагает связанность, а архитектура микросервисов в значительной степени старается ее избегать. Это достигается за счет разбиения системы на сервисы по ограниченным контекстам (бизнес областям). Типичная MSA архитектура показана на рисунке ниже.



В отличие от SOA каждый сервис обладает всеми необходимыми для функционирования частями – имеет свою собственную базу данных и существует как независимый процесс. Такая архитектура делает каждый сервис физически разделенным, самодостаточным, что ведет с технической точки зрения к архитектуре без разделения ресурсов.

Сервисы раскрываются для потребителей также через слой API, но его стараются проектировать с полным отсутствием какой-либо логики. Это фактически просто проксирование API сервисов во вне.

Взаимодействие между сервисами сводится к обмену данными, используя брокер сообщений (или REST / GraphQL, gRPC). Именно к обмену данными, а не вызову методов из других сервисов.

**SOA и MSA с точки зрения развития и поддержки систем**

* **Внесение изменений**

**SOA**

Выполнять изменения в архитектуре SOA чрезвычайно трудно, зачастую для внесения одного изменения требуется изменения сразу в нескольких сервисах. Так как отдельными сервисами владеют разные команды, то внесение элементарных изменений превращается в сущий ад из бесконечных совещаний, согласований и документов.

Если две разные команды используют один общий сервис, то при изменении этого сервиса нужно будет учитывать влияние на сервисы обеих команд. Когда таких зависимостей много, развивать сервис становиться очень трудно.

**MSA**.

Так как в микросервисной архитектуре зависимости от других сервисов отсутствуют либо минимальны, то необходимость взаимодействия с другими сервисами и командами пропадает. Внесение изменений осуществляется командой, которая владеет сервисом. Изменения происходят легче и быстрее.

* **Переиспользование/Дублирование**

**SOA**

Стремление к переиспользованию. Но, к сожалению, разработчики часто тратят много времени, пытаясь интегрировать повторно используемые сервисы, которые, как оказывается, мало используются повторно. В конечном итоге мы приходим к ситуации - чем больше у сервиса возможностей для повторного применения, тем менее пригодным к применению он становится.

**MSA**

Может возникнуть ситуация, когда два разных сервиса используют одну и ту же сущность, к примеру «Customer». В микросервисной архитектуре предпочтительнее использовать дублирование. То есть создать в каждом сервисе свою реализацию «Customer», что позволит развивать оба сервиса независимо.

Есть большая вероятность, что спустя время в одном из сервисов изменятся бизнес правила или структура объекта «Customer», но так как в двух сервисах сущность дублируется, то на другой сервис это не окажет никакого влияния. Оба сервиса все так же могут развиваться независимо.

* **Команды/Поддержка**

**SOA**

Большинство команд архитектуры SOA разделены так же, как архитектура, и требуются колоссальные усилия координации для простых изменений. В основном это связано с тем, что большинство из команд не владеет процессом от начала до конца, а лишь развивает сервисы, которые являются его частями. Как следствие они не понимают и не отвечают за процесс целиком.

**MSA**

Команда кроссфункциональна, то есть включает в себя всех специалистов, необходимых для развития сервиса. Так как сервис реализует процесс от и до, то команда владеет процессом от начала до конца. Как следствие они понимают и отвечают за процесс целиком. И да, за часть фронтенда, в рамках сервиса, так же отвечает команда.

* **Взаимодействие**

**SOA**

Взаимодействие осуществляется через корпоративную шину. Если в ней со временем появляется много логики, то она легко становится бутылочным горлышком.

**MSA**

Каждый сервис обладает всеми частями своего ограниченного контекста и осуществляет связь с другими ограниченными контекстами с помощью обмена данными, используя брокер сообщений. Просто обмен, никакой сложной логики.

* **Автоматизация и развертывание**

**SOA**

Архитектуру SOA на основе событий сложно ввести в эксплуатацию. Эта архитектура состоит из множества развертываемых модулей, что затрудняет процесс автоматизации и координации.

**MSA**

Сервис может быть развернут независимо от других сервисов (и другой инфраструктуры), что отражает ограниченный контекст. Возможность для разработчика развертывать один сервис, не оказывая влияния на другой, является одним из преимуществ архитектуры этого типа.

* **Тестирование**

**SOA**

Тестирование архитектуры SOA затруднено. Ни одна из частей архитектуры не является завершенной — все они являются частью более крупного рабочего потока и обычно не предназначены для изолированного тестирования. Провести end-to-end тестирование такой системы, практически нереально.

**MSA**

Так как каждый сервис имеет хорошо определенную границу и минимум зависимостей, это позволяет легко тестировать сервис в изоляции от других частей системы. Подготовка тестовых данных и управление ими так же облегчена, так как они находятся под контролем той же команды, что разрабатывает сам сервис. Не многочисленные зависимости сервиса (БД, брокер сообщений) легко мокируются.

**Итоги**

Архитектуры программного обеспечения создаются не в вакууме, а как ответ на окружающую среду, учитывая различные ограничения среды. 10 лет назад не были так популярны системы с открытым исходным кодом, не было Docker, приложения разворачивались на дорогих железных серверах. Вся инфраструктура была коммерческой, лицензионной и дорогостоящей. SOA была оптимальна для оптимизации и утилизации инфраструктуры, отсюда и такая тяга к переиспользованию.

Сейчас большая часть программного стека имеет открытый исходный код, лицензирование и другие проблемы больше не влияют на архитектуру. Системы непрерывного развертывания и движение DevOps позволяют легко управлять и создавать сотни независимых сервисов.

Важно понимать, что различие между двумя подходами не столько техническое, сколько концептуальное. Если вы убрали ESB, но при этом все еще стремитесь к переиспользованию, и у вас много зависимостей между сервисами, то такую архитектуру нельзя назвать микросервисной.