Service Oriented Architecture (SOA)

Перевод статьи: Service Oriented Architecture (SOA) (Herberto Graca)

SOA существует с конца 1980-х годов и основывается на идеях таких подходов как CORBA, DCOM, DCE. Много было сказано о SOA, и есть несколько различных шаблонов её реализации, но, по сути, SOA фокусируется только на нескольких концепциях и не дает никакого рецепта их реализации:

- Совместимость пользовательских приложений;
- Повторное использование бизнес-сервисов;
- Независимость от технологического стэка;
- Автономность (независимое развитие, масштабирование, развертывание).

SOA – это набор архитектурных принципов, независимых от какой-либо технологии или продукта, как полиморфизм или инкапсуляция.

В этой статье я собираюсь осветить следующие шаблоны, связанные с SOA:

- CORBA – Common Object Request Broker Architecture
- Web Services
- Message Queue
- Enterprise Service Bus (ESB)
- Microservices

CORBA – Common Object Request Broker Architecture

В 1980-х годах, в связи с растущим использованием корпоративных сетей и клиент-серверной архитектуры, возникла потребность в общем способе взаимодействия приложений, которые были построены с использованием различных технологий и работали на разных компьютерах с разной ОС. CORBA была разработана для удовлетворения этой потребности. Это стандарт для распределенных вычислений, разработанный в 1980-х годах и достигший своей первой зрелой версии 1991 года.

Стандарт CORBA был реализован несколькими поставщиками и направлен на обеспечение:

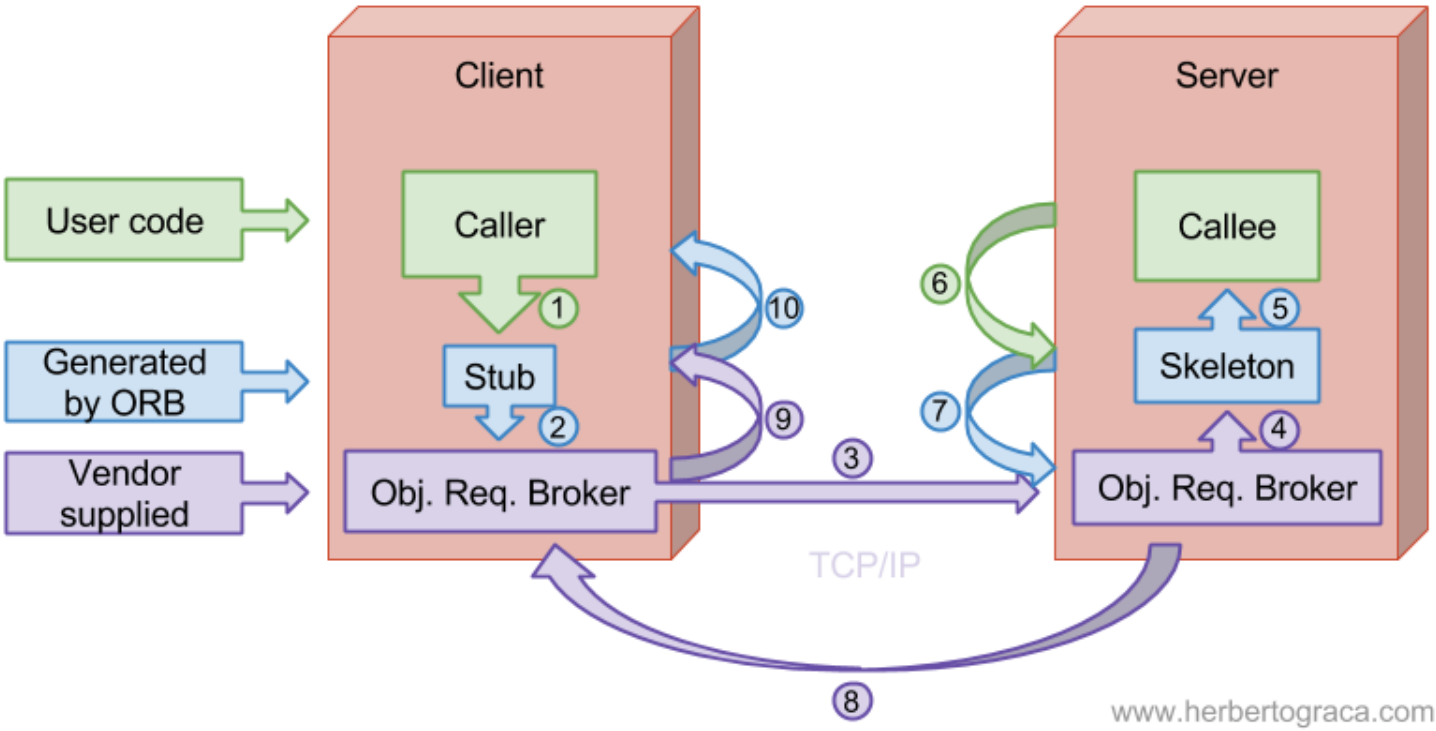
- Платформонезависимого RPC (Remote Procedure Call);
- Транзакций (также распределнных транзакций);
- Безопасности;
- Событии;
- Независимости от языка программирования;
- Независимости от ОС;
- Независимости от оборудования;
- Изоляции от детплей взаимодействия/передачи данных.

На данный момент стандарт CORBA все еще используется для распределенных вычислений. Например, он все еще является частью JAVA EE, несмотря на то, что будет упакован как отдельный модуль в JAVA 9 и далее.

Важно отметить, что я не считаю CORBA шаблоном SOA (хотя я считаю, что оба шаблона и CORBA и SOA подходят для организации распределенных вычислений). Я решил включить CORBA здесь, потому что я чувствую, что именно его недостатки привели к развитию SOA.

Как работает стандарт COBRA

Во-первых, нам нужно получить Object Request Broker (ORB), который соответствует спецификации CORBA. Он реализуется поставщиком и используется для создания заглушек (Stub'ов) и скелетов (Skeleton'ов) на языках клиентского кода. Используя ORB и определение интерфейса, выраженного с помощью IDL (аналогично WSDL), на стороне клиента мы генерируем Stub-классы реальных классов, которые могут быть вызваны удаленно. В тоже время на сервере мы генерируем Skeleton-классы, которые могут обрабатывать входящие запросы и вызывать методы целевого объекта.



1. Объект на стороне клиента (Caller) вызывает локальную процедуру, реализованную Stub'ом;
2. Stub проверяет вызов и создает сообщение запроса, которое передается ORB'у;
3. ORB на строне клиента отправляет сообщение по сети на сервер и блокирует текущий поток выполнения;

4. ORB на строне сервера получает сообщение запроса и создает экземпляр Skeleton’a;
5. Skeleton выполняет процедуру на вызываемом объекте (Callee);
6. Вызываемый объект (Callee) выполняет вычисление и возвращает результат;
7. Skeleton упаковывает выходные аргументы в ответное сообщение и передает его ORB’у;
8. ORB на стороне сервера отправляет сообщение по сети обратно клиенту;
9. ORB на строне клиента получает ответное сообщение, распаковывает его и доставляет Stub’у;
10. Stub передает выходные аргументы вызывающему объекту (Caller), освобождает поток выполнения, и вызывающий продолжает выполнение.

Достоинства:

- Независимость от технологического стека;
- Изоляции от детплей взаимодействия/передачи данных.

Недостатки:

- **Неопределнность расположения вызываемого объекта**: код клиента не знает, является ли вызов локальным или удаленным. Это звучит как что-то хорошее, однако, задержка и типы сбоев совершенно разные, в зависимости от типа вызова. И это делает невозможным для приложений выбор соответствующей стратегии для обработки вызова метода, и в конечном итоге может возникнуть ситуация, когда удаленные вызовы производятся внутри цикла, что значительно замедляет всю систему.
- **Сложная, избыточная и неоднозначная спецификация**: она создавалась как месиво из нескольких существующих версий вендора,

поэтому (в то время) она была неоднозначной и избыточной, что затрудняло ее реализацию.

- **Блокировка каналов связи**: COBRA использовал определенные протоколы передачи данных через TCP/IP и определенные порты (или даже случайные порты). Но корпоративные правила безопасности и межсетевые экраны часто разрешают связь только через HTTP по порту 80, эффективно блокируя каналы связи, необходимые для CORBA.

Web Services

Хотя стандарт CORBA все еще применяется в наше время, его недостатки позволяют понять, что нам необходимо уменьшить количество удаленных вызовов для повышения производительности, получить надежный канал связи и более простую спецификацию.

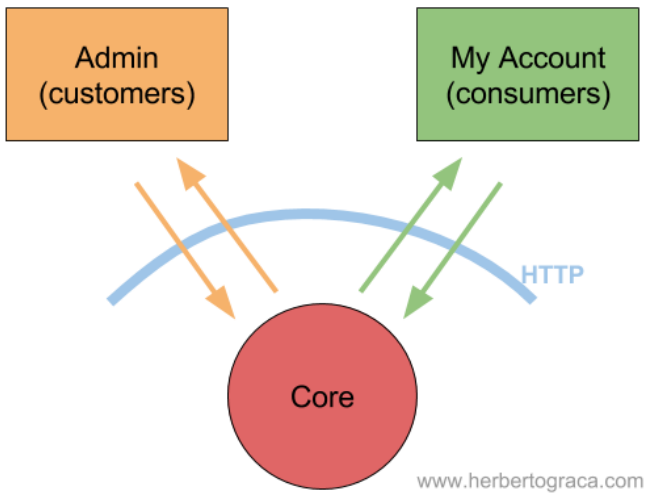
Таким образом, в конце 1990-х годов Web Service'ы начали зарождаться с целью решения вышеупомянутых проблем:

- Нам необходим надежный канал связи, поэтому:
  1. HTTP через 80 порт – это канал связи по-умолчанию;
  2. XML или JSON, как язык взаимодействия.
- Нам необходимо уменьшить количество удаленных вызовов, поэтому:
  1. У нас есть явноопределенные удаленные связи, следовательно, мы точно знаем, когда мы делаем удаленный вызов;
  2. У нас есть крупные вызовы/запросы (одним вызовом запрашивается большое количество различных данных), что позволяет нам реже вызывать удаленную службу.
- Нам необходима более простая спецификация, поэтому появились следующие спецификации:

- Черновой вариант **SOAP** был подготовлен в 1998 году, а в 2003 году был рекомендован W3C, что фактически сделало его стандартом. Он воплотил некоторые идеи CORBA, такие как слой для обработки взаимодействий и "документ", определяющий интерфейсы по средствам языка описания веб-служб (WSDL);
- **REST** был представлен в 2000 году Роем Филдингом в его докторской диссертации "Architectural Styles and the Design of Network-based Software Architectures", и это гораздо более простая спецификация, чем SOAP, что позволило ей быстро получить одобрение в сообществе разработчиков;
- **GraphQL** был разработан Facebook в 2012 году и представлен общественности в 2015 году. Это язык запросов API, который позволяет клиенту точно указать, какие данные сервер должен вернуть в ответ на запрос, что позволяет избежать как чрезмерной, так и неполной выборки данных.

[Web] Service'ы могут быть опубликованы, обнаружены и использованы в технологически нейтральной, стандартной форме.

Microsoft 2004, [Understanding Service-Oriented Architecture]



Однако мы должны понимать, что под эгидой SOA, Web Service – это не просто API общего назначения, который просто предоставляет CRUD-доступ к своей базе данных через HTTP (хотя эта реализация может быть полезна в некоторых случаях). Напротив, спецификация SOA требует, чтобы пользователи понимали базовую модель и соблюдали бизнес-правила, чтобы обеспечить защиту целостности данных. SOA подразумевает, что Web Service'ы разрабатываются как ограниченные контексты для бизнес-поддоменов.

Достоинства:

- Независимость от технологического стека, развертывание и масштабируемость сервисов;
- ***Общий, простой и надежный канал связи (текст через HTTP, порт 80);***
- ***Оптимизированные коммуникации;***
- ***Стабильная спецификация взаимодействия;***
- ***Изоляция контекстов предметной области.***

Недостатки:

- Сложная интеграция различных Web Service'ов, из-за разных языков общения. Например, два Web Service'а, использующие разное представления (например, в формате JSON) одного и того же понятия;
- Синхронные взаимодействия могут привести к снижению производительности системы.

Message Queue

Основная идея состоит в том, чтобы несколько приложений асинхронно взаимодействовали между собой, используя сообщения в каком-то общем для всех взаимодействующих приложений формате. Message Queue способствует масштабируемости и ослаблению зависимости между приложениями, т.к. им не нужно знать, где находятся другие приложения, сколько их и даже кто они. Несмотря на это, все они должны использовать один и тот же язык общения, т.е. заранее определенный текстовый формат для представления данных.

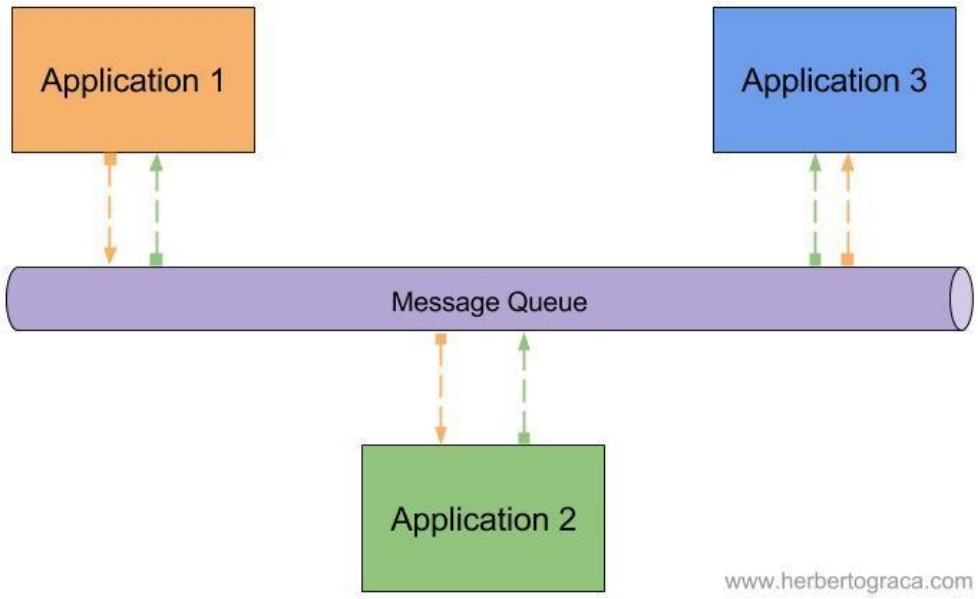
Message Queue использует программное обеспечение брокера сообщений (например, RabbitMQ, Beanstalkd, Kafka и т.д.) как инфраструктурный компонент, который может быть настроен различными способами для реализации связи между приложениями:

- **Request/Response**
  Клиент отправляет сообщение, содержащее идентификатор взаимодейсвия, в Message Queue. Сообщение доставляется определенному узлу, который ответит исходному отправителю другим сообщением с тем же идентификатором взаимодейсвия. На основе идентификатора взаимодейсвия можно понять к катому взаимодействию относится то или иное сообщение. Это очень полезно для средних и длительных бизнес-процессов (sagas).

- **Publish/Subscribe**
  Существует 2 типа реализации:
  1. **List-Based**
     Т.е ведутся списки опубликованных тем и подписчиков на эти темы. Когда Message Queue получает сообщения для какой-то темы, она помещает сообщение в соответствующий список относящийся к этой

теме. Сопоставление сообщения с темой может быть выполнено по типу сообщения или с более сложным набором предопределенных критериев, которые могут включать само содержимое сообщения.

2. **Broadcast-Based**
   Когда Message Queue получает сообщения, он передает их всем узлам, которые прослушивают эту Message Queue. Каждый прослушивающий узел отвечает за фильтрацию и обработку только тех сообщений, которые его интересуют.



Подходы, перечисленные выше, могут быть настроены в режиме **Pull** (polling (опрос)) или **Push**:

- В режиме **Pull** клиент запрашивает сообщения из очереди через определенные промежутки времени. Преимущество в том, что клиент может контролировать свою нагрузку. А недостатком является то, что когда в очередь поступают новые сообщения, то клиент не получает их сразу, а просто ждет определенный промежуток времени, чтобы сделать запрос за этими сообщениями;

- В режиме **Push** новое сообщение доставляются клиенту автоматически, как только это сообщение поступает в очередь. Преимущество здесь состоит в том, что нет никакой задержки, но клиенты не могут самостоятельно управлять своей нагрузкой.

Достоинства:

- Независимость от технологического стека, развертывание и масштабируемость сервисов;
- Общий, простой и надежный канал связи (текст через HTTP, порт 80);
- Оптимизированные коммуникации;
- Стабильная спецификация взаимодействия;
- Изоляция контекстов предметной области;
- ***Простота добавления/удаления служб;***
- ***Асинхронная связь помогает управлять нагрузкой системы.***

Недостатки:

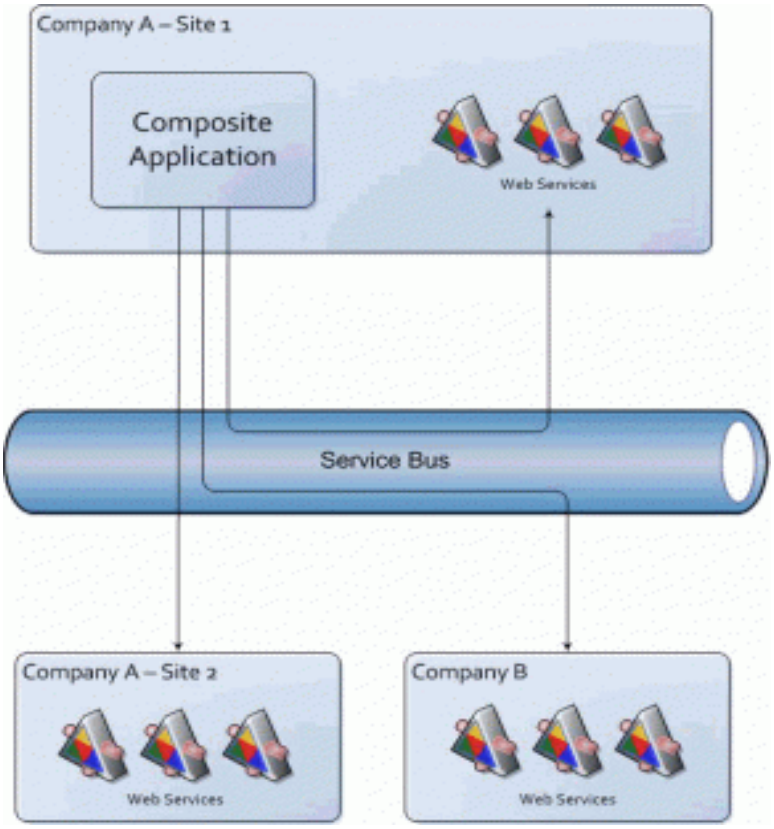
- Сложная интеграция различных Web Service'ов, из-за разных языков общения. Например, два Web Service'а, использующие разное представления (например, в формате JSON) одного и того же понятия;

Enterprise Service Bus (ESB)

ESB был разработан в 1990-х для удовлетворения потребности компаний в интеграции нескольких независимых/разнородных приложений (например, для финансов, учета кадров, управления активами и т. д.), т.е. создать сопособ коммуниуации между такими приложениями. Но дизайн этих приложений не подразумевал возможности интеграции, т.е. не было общего формата для общения между этими приложениями. Поэтому разумным решением для поставщиков приложений было создание конечных точек для отправки и получения данных в определенном формате. Затем компании-клиенты должны будут интегрировать приложения путем создания канала связи, который будет осуществлять перевод сообщений из формата/языка одного приложения в формат/язык другого приложения.

Message Queue может способствовать решению этой проблемы, но по-прежнему не может решить проблему приложений с разными форматами языков коммуникации. Тем не менее, это был шаг, чтобы преобразовать Message Queue из «тупого» канала связи в промежуточное программное обеспечение, которое обрабатывает как доставку сообщений, так и их преобразование в язык/формат, ожидаемый получателем. Таким образом ESB является результатом естественного развития Message Queue.

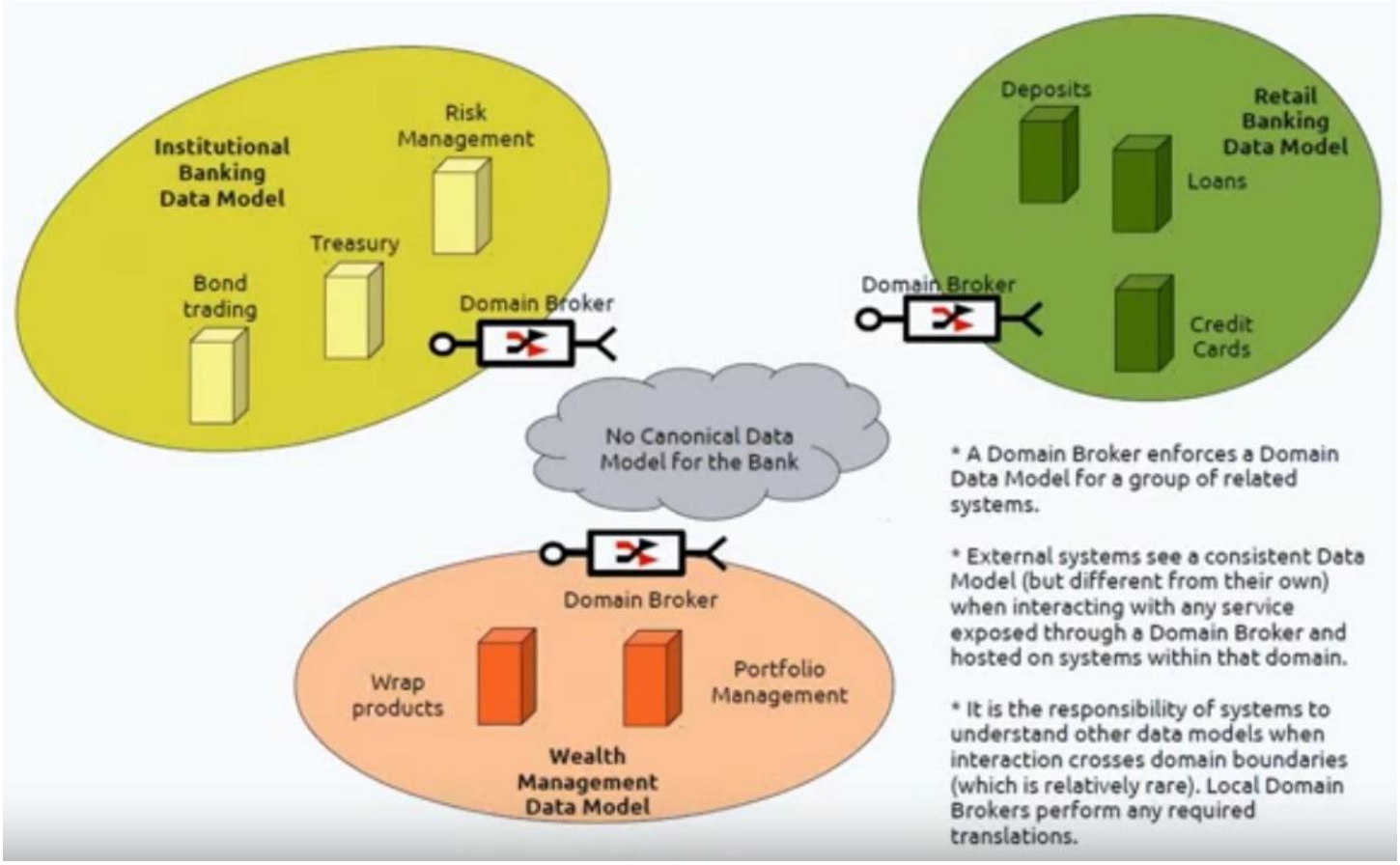
В архитектуре использующей ESB у нас могут быть составные приложения, обычно ориентированные на пользователя (User-facing приложения), которые связываются с Web Service'ми для выполнения определенных операций. Web Service'ы, в свою очередь, также могут обращаться к другим Web Service'ам, и в конечном итоге они могут возвращать некоторые данные обратно User-facing приложению. Однако как User-facing приложение, так и Web Service'ы не имеют никаких сведений друг о друге, а именно об их расположении или протоколах связи. Все что им необходимо знать – это название службы, которая им нужна и расположение Service Bus'а.



Таким образом, клиентское приложение (будь то служба или составное приложение) отправляет свой запрос в Service Bus, которая, в свою очередь, преобразует сообщение в формат, ожидаемый получателем, и направляет запрос к пункту назначения. Важно отметить, что все коммуникации проходят через ESB, что означает, что если ESB отключается, все коммуникации отключаются, и все системы становятся неработоспособными. В конце концов, ESB работает как промежуточное программное обеспечение, где происходит много вещей, что делает его очень сложным и значимым компонентом архитектуры.

Это, конечно, очень поверхностное описание архитектуры с применением ESB. И не смотря на то, что ESB является основным компонентом, в такой архитектуре, могут быть и другие компоненты, такие как брокеры доменов (Domain Brokers), службы данных (Data Services), службы оркестрации процессов (Process Orchestration Services) или Rules Engine'ы. Этот же шаблон архитектуры может быть реализован с использование интеграционного дизайна, в котором система разделена на бизнес-домены, каждый из которых имеет собственную ESB. Это

помогает повысить производительность и решить проблему единой точки отказа (Single Point of Failure), т. е. в случае сбоя одного ESB будет затронут только его бизнес-домен.



Основными обязанностями ESB являются:

- Мониторинг и управление обменом сообщениями между службами;
- Преобразование сообщений между взаимодействующими службами;
- Управление развертыванием и версиями служб;
- Поддержка таких процессов как обработка событий, преобразование и mapping данных, контроль очередности и последовательности сообщений и событий, безопасность, обработка исключительных ситуаций.

У архитектуры с использованием ESB безусловно есть преимущества, но я нахожу её особенно полезной, если мы не “владеем” Web Service’ами и следовательно нуждаемся в промежуточном программном обеспечении для преобразования сообщений между ними.

Также необходимо иметь в виду, что реализации ESB эволюционировали, и сегодня ESB может быть настроена через удобный UI для большинства случаев использования.

Достоинства:

- Независимость от технологического стека, развертывание и масштабируемость сервисов;
- Общий, простой и надежный канал связи (текст через HTTP, порт 80);
- Оптимизированные коммуникации;
- Стабильная спецификация взаимодействия;
- Изоляция контекстов предметной области;
- Простота добавления/удаления служб;
- Асинхронная связь помогает управлять нагрузкой системы;
- ***Единая точка для настройки и контроля версионности компонентов архитектуры и правил преобразования сообщений между этими компонентами.***

Недостатки:

- Более низкая скорость коммуникаций между компонентами (Web Service’ами);
- Централизованная логика:
    1. Единая точка отказа, которая может сделать невозможными коммуникации между компонентами архитектуры;

2. Высокая сложность конфигурирования и обслуживания;
3. Со временем ESB может содержать бизнес-правила;
4. Из-за сложности ESB ей в конечном итоге понадобится отдельная команда для администрирования;
5. Компоненты системы становятся сильно зависимыми от ESB.

Microservices

Архитектура Микросервисов основана на концепциях SOA и имеет те же глобальные цели, что и ESB: ***создание глобального корпоративного приложения из нескольких приложений, каждое из которых относится к определенному бизнес-домену/бизнес-поддомену***.

Ключевое различие заключается в том, что ESB был рожден в контексте автономных приложений, которые должны быть интегрированы для создания корпоративного распределенного приложения, в то время как Архитектура Микросервисов родилась в контексте быстро меняющихся и постоянно меняющихся предприятий, которые создают свои собственные облачные приложения с нуля.

Другими словами, в случае ESB у нас есть существующие приложения, которыми мы не “владеем”, и поэтому мы не можем их изменить. Но с Микросервисами мы имеем полный контроль над приложениями (но это не означает, что в системе не может быть сторонних веб-сервисов).

Архитектура Микросервисов предотвращает высокую потребность в интеграции между микросервисами. Микросервисы должны быть специфичны для ограниченного контекста, они должны сохранять свое собственное состояние,

чтобы не зависеть напрямую от других микросервисов и поэтому нуждаться в меньшей интеграции. Другими словами, высокая внутренняя связность (high cohesion) и низкая внешняя зависимость (low coupling) микросервисов, обеспечиваемая Архитектурой Микросервисов иметь приятный побочный эффект снижения потребности в интеграции.

> *Микросервисы – это небольшие автономные службы, смоделированные вокруг бизнес-домена и работающие вместе.*
>
> *Sam Newman 2015, Principles Of Microservices*

Поскольку самым большим недостатком архитектуры ESB было очень сложное центральное приложение, от которого зависели все другие приложения, Архитектура Микросервисов решает эту проблему, удаляя ее почти полностью.

Несмотря на то, что Архитектура Микросервисов все еще содержит элементы, которые являются общими для всей экосистемы микросервисов, эти элементы не содержат так много обязанностей по сравнению с ESB. Например, имеется Message Queue для асинхронного взаимодействия между микросервисами, но это просто канал передачи сообщений без каких-либо других обязанностей. Другим примером является шлюз экосистемы микросервисов, через который осуществляется вся связь с внешним миром.

Сэм Ньюман, автор книги Building Microservices, выделяет 8 принципов Архитектуры Микросервисов:

- **Микросервисы формируются вокруг бизнес-доменов**
  Это позволяет нам получить стабильные интерфейсы вокруг бизнес-концепции, единицы кода с высокой внутренней связностью и низкой внешней зависимостью, и четко определенные ограниченные контексты;

- **Культура автоматизации**
  Потому что у нас будет много подвижных частей архитектуры, которые нуждаются в развертывании;

- **Сокрытие деталей реализации**
  Т.е. возможность развивать отдельный микросервис независимо от остальных;

- **Децентрализация**
  Децентрализация власти принятия решений и архитектурных концепций, что дает автономию командам разработки. Таким образом организация превращается в систему, которая может быстро адаптироваться к изменениям;

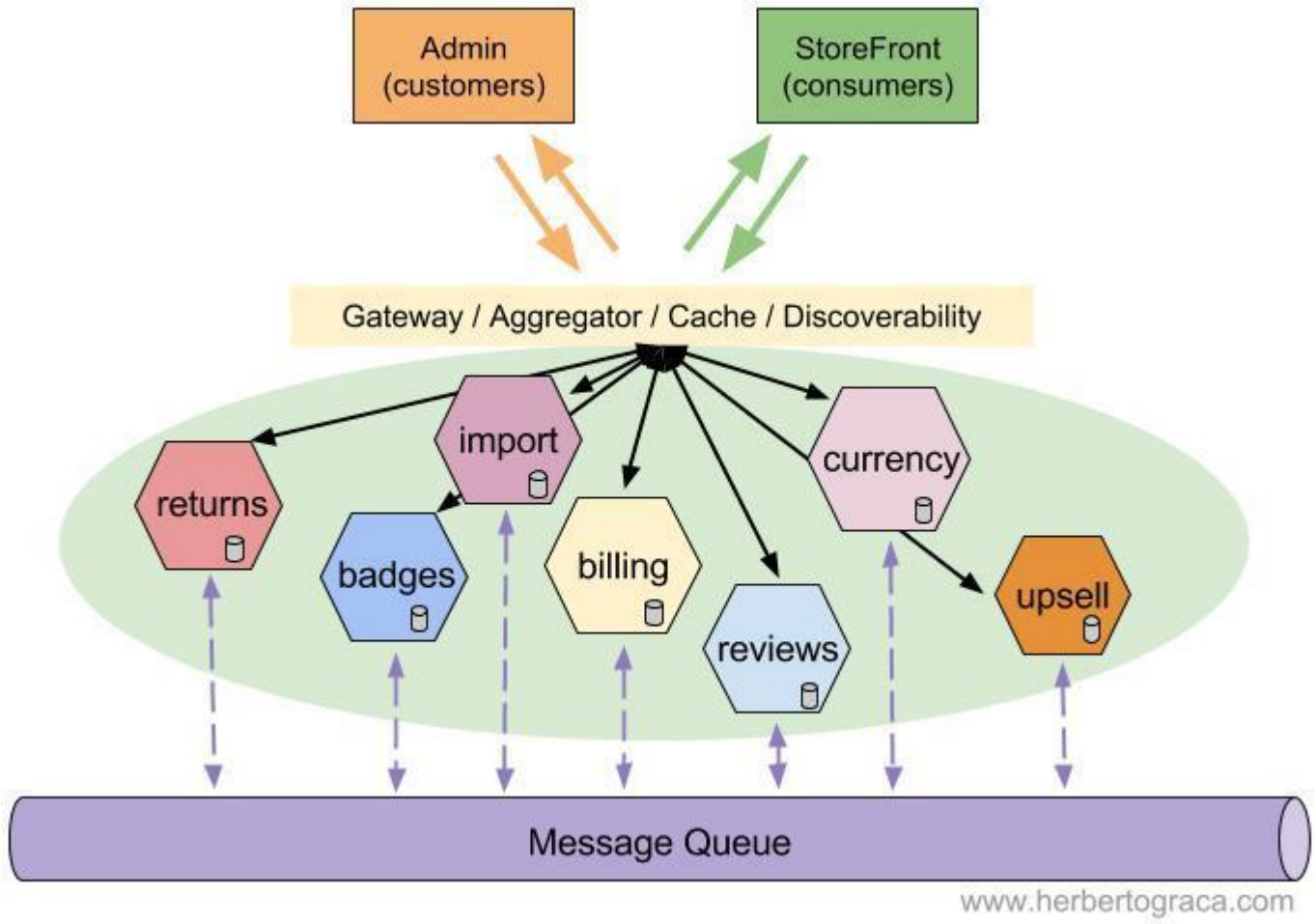
- **Независимое развертывание**
  Т.е. мы можем развернуть новую версию микросервиса без необходимости изменять что-либо еще;

- **Клиент превыше всего**
  Каждый микросервис должен быть прост в использовании для других компонентов архитектуры;

- **Изоляция отказов**
  Т.е. если один микросервис выходит из строя, другие продолжают работать, давая общей системе высокую устойчивость к сбоям;

- **Прозрачность**
  Из-за большого количества частей системы становится сложнее понять все, что происходит, поэтому нам нужны сложные инструменты мониторинга,

которые позволяют нам контролировать, что происходит в каждой части системы и понимать любые цепные реакции.



Достоинства:

- Независимость от технологического стека, развертывание и масштабируемость сервисов;
- Общий, простой и надежный канал связи (текст через HTTP, порт 80);
- Оптимизированные коммуникации;
- Стабильная спецификация взаимодействия;
- Изоляция контекстов предметной области;
- Простота добавления/удаления служб;
- Асинхронная связь помогает управлять нагрузкой системы;
- ***Асинхронная связь помогает управлять производительностью системы;***

- ***Независимость и автономность сервисов;***
- ***Логика предметной области не просачивается за пределы микросервисов;***
- ***Организация превращается в систему, состоящую из определенного количества независимых команд и способную быстро адаптироваться к изменениям бизнеса.***

Недостатки:

- Высокая сложность:
  1. Требует развития DevOps культуры;
  2. Использование множества технологий и библиотек может выйти из-под контроля;
  3. Изменения в API микросервисов должны производиться очень аккуратно, т.к. большое количество частей системы зависят от этого интерфейса;
  4. Тестирование становится более сложным, поскольку изменения интерфейса могут иметь непредсказуемые последствия в других сервисах.

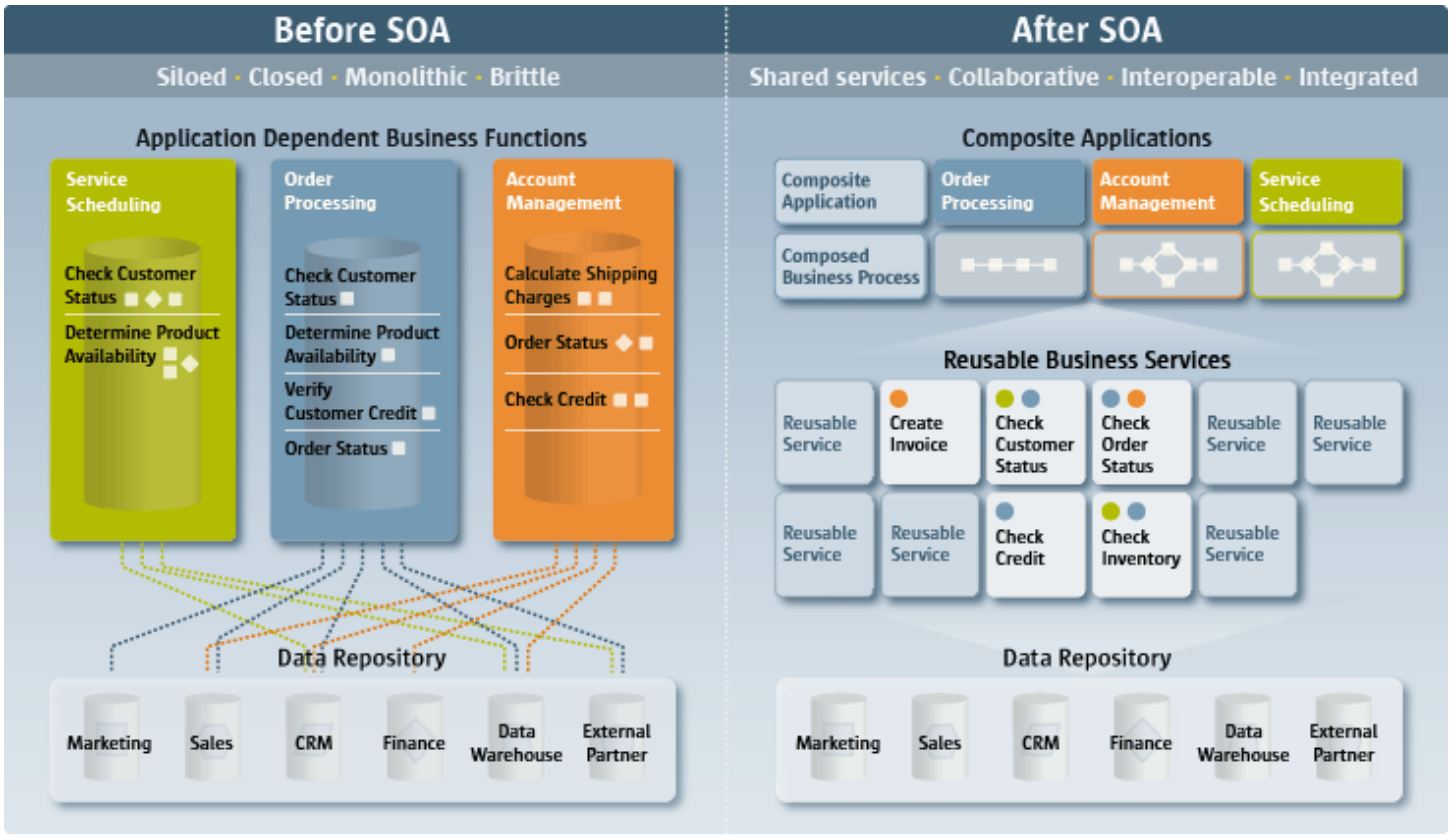
Анти-шаблон: Ravioli-архитектура

Этим понятием обозначается анти-шаблон для архитектуры микросервисов. Ravioli-архитектура – это когда созданная экосистема содержит слишком много очень маленьких микросервисов, которые не отражают концепции предметной области.

Заключение

SOA сильно эволюционировала в последние десятилетия. Опираясь на опыт, достоинства и недостатки предыдущих реализация мы пришли к Архитектуре Микросервисов.

Основной идея этой эволюции всегда была обычная стратегия, используемая для решения сложных задач, а именно *разбить проблему на более мелкие, разрешимые части*.



Решить проблему сложности кода можно таким же образом и в случае, когда у нас есть монолитное приложение, путем разбиения его на ограниченные контексты. Но по мере роста команд и кодовой базы, растет и потребность в независимой эволюции, масштабируемости и развертываемости компонентов. SOA предоставляет инструменты для этой независимости, обозначая более строгие границы между ограниченными контекстами.

И опять же это о высокой внутренней связности (high cohesion) и низкой внешней зависимости (low coupling) только в более крупном масштабе. Также очень важно подходить с прагматизмом к анализу наших потребностей, а именно использовать SOA только тогда, когда нам это действительно нужно, потому что это привносит больше сложности в проект. Но если мы действительно должны использовать SOA давайте создадим Архитектуру Микросервисов с размером и количеством микросервисов, которое будет действительно соответствовать нашим потребностям, ни больше, ни меньше.