Service Oriented Architecture (SOA)

Перевод статьи: [Service Oriented Architecture (SOA)](https://herbertograca.com/2017/11/09/service-oriented-architecture-soa/) ([Herberto Graca](https://herbertograca.com/))

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:

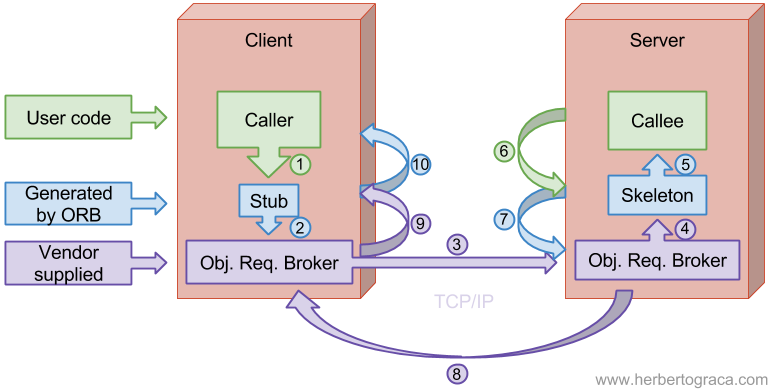
* Платформанезависимого RPC (Remote Procedure Call);
* Транзакций (также распределнных транзакций);
* Безопасности;
* Событии;
* Независимости от языка программирования;
* Независимости от ОС;
* Независимости от оборудования;
* Изоляции от детплей взаимодействия/передачи данных.

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 стандарт COBRA работает**

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’y;
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’а;
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’y;
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 хорошая вещь, 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 произодятся 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.
* **Заблокированные каналы 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 наше время, её 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:

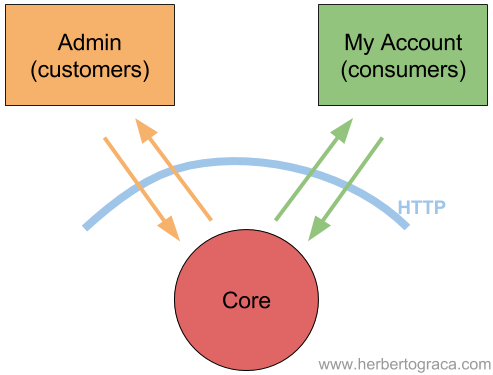
1. HTTP через 80 порт – это канал связи по-умолчанию;
2. XML или JSON, как язык взаимодействия.

* 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 пояились 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, 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](https://msdn.microsoft.com/en-us/library/aa480021.aspx?irgwc=1&OCID=AID681541_aff_7593_1243925&tduid=(ir_1E5S9ATqcSzxx6N3-Kx-QQlfUkjWAMVhnzq61c0)(7593)(1243925)(TnL5HPStwNw-MobCHa37s9s3uKidRvDwwA)()&irclickid=1E5S9ATqcSzxx6N3-Kx-QQlfUkjWAMVhnzq61c0)



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: