«Сейфуллин оқулары – 18: « Жастар және ғылым – болашаққа көзқарас» халықаралық ғылыми -практикалық конференция материалдары = Материалы международной научно-практической конференции «Сейфуллинские чтения – 18: « Молодежь и наука – взгляд в будущее» - 2022.- Т.I, Ч.IV. - С. 6-8

СЕРВИСНО ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА SOA

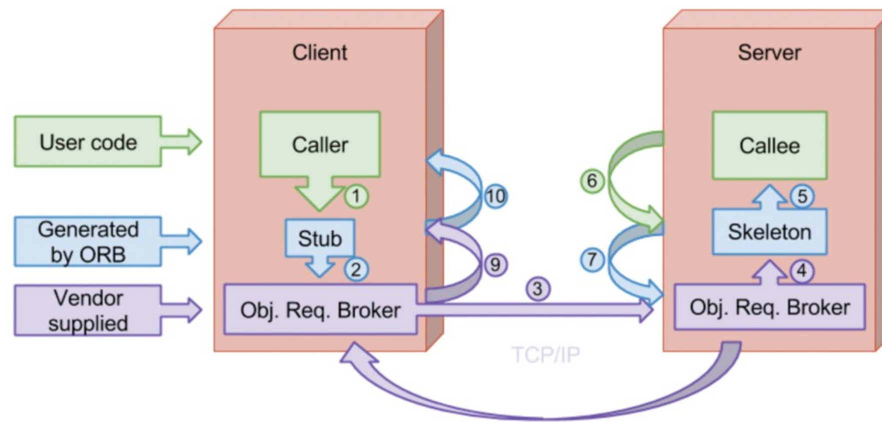
Армантайулы Максат, магистрант 2 курс
Казахский агротехнический университет им. С.Сейфуллина, г. Нур-Султан

Введение. Сервис-ориентированная архитектура (service-oriented architecture, SOA) была создана в конце 1980-х. Истоки начинаются в идеях, которые изложенны в DCOM, CORBA, DCE и других документах. О сервисно-ориентированной архитектуры SOA написано очень много, а также существуют некоторое количество её реализаций. Но, по сути, сервисно-ориентированную архитектуру SOA можно привести к нескольким идеям, при этом данная архитектура не диктует способы их реализации:

1) Сочетаемость приложений, ориентированных на пользователей.
2) Многократное использование бизнес-сервисов.
3) Независимость от набора технологий.
4) Автономность (независимая эволюция, масштабируемость и свертываемость).

Сервисно-ориентированная архитектура SOA — это набор архитектурных принципов, которые не зависят от технологий и продуктов, такие как полиморфизм или инкапсуляция.

Принцип работы: в первую очередь нужно будет получить - брокер объектных запросов (ORB, Object Request Broker), который должен соответствовать спецификации CORBA. Брокер объектных запросов предоставляется вендором, а также использует языковые преобразователи (language mappers) для того, чтобы генерировать «заглушки» (stub) и «скелеты» (skeleton) на языках программирования клиентского кода. С помощью брокера объектных запросов ORB и определений интерфейсов, которые используют IDL (это аналог WSDL), есть возможность на основе реальных классов генерировать в клиенте удалённо вызываемые классы-заглушки (stub classes). А также на сервере возможно генерировать классы-скелеты (skeleton classes), которые обрабатывают входящие запросы и вызывают реальные целевые объекты.



Вызывающая программа (caller) вызывает локальную процедуру, которая реализована заглушкой:

1) Задача заглушки проверять вызов, создавать сообщение-запрос и передавать его в брокер объектных запросов ORB.
2) Клиентский брокер объектных запросов ORB отправляет сообщение по сети на сервер, затем блокирует текущий поток выполнения.
3) Серверный брокер объектных запросов ORB получает сообщение-запрос, и затем создаёт экземпляр скелета.
4) Задача скелета осуществлять процедуру в вызываемом объекте.
5) Задача вызываемого объекта проводить вычисления и возвращать результат.
6) Задача заглушки передать выходные аргументы вызывающему методу, затем разблокировать поток выполнения, и вызвать программу продолжающая свою работу.

Преимущества: 1. Независимость от выбранных технологий, не считая реализации брокер объектных запросов ORB; 2. Независимость от особенностей переди данных и связи.

Недостатки:

1) Независимость от местоположения. Клиентский код не знает, является ли вызов локальным или удалённым. Но длительность задержки и виды сбоев могут очень сильно варьироваться. Если мы не в курсе, какой должен быть у нас вызов, в этом случае приложение не может выбрать подходящую стратегию обработки вызовов методов, таким образом, и генерировать удалённые вызовы внутри самого цикла. В результате всего этого вся система работает очень медленно.
2) Раздутая, сложная и неоднозначная спецификация. Все это собрали из некоторого количества версий спецификаций различных вендоров, в этом случае на тот момент она была раздутой, неоднозначной и очень трудной в реализации.
3) Заблокированные каналы связи (communication pipes). Они используют специфические протоколы поверху TCP/IP, и специфические порты (случайные порты). Но правила корпоративной безопасности и файрволы зачастую позволяют HTTP-соединениям только через 80-й порт, тем самым блокируя обмены данными CORBA.

Очередь сообщений: Существует некоторое количество приложений, которые асинхронно контактируют друг с другом с помощью платформа-независимых сообщений. Улучшает масштабируемость и усиливает изолированность приложений с помощью очереди сообщений. Не обязательно знать, в каком месте хранятся другие приложения, какое их количество и даже то, что они из себя представляют. Впрочем, все данные приложения обязаны использовать один и тот же язык обмена сообщениями, то есть заранее определённый текстовый формат представления данных.

В качестве компонента инфраструктуры программный брокер сообщений (RabbitMO, Beanstalkd, Kafka и другие) используют очередь сообщений. Для реализации связи между приложениями возможно различно настроить очередь:

Клиент отправляет в очередь сообщение, а также ссылку на «разговор» *(«conversation» reference)*. Сообщение получает специальный узел, который отвечает отправителю совсем другим сообщением, где находится ссылка на тот самый «разговор» , таким образом получатель осознает, на какой «разговор» ссылается сообщение, и затем спокойно может продолжать действовать. Для бизнес-процессов средней и большой продолжительности (Sagas цепочек событий) это очень полезно.

По спискам. Списки опубликованных подписок (topics) и их подписчиков поддерживает очередь. Когда очередь получает сообщение для любой темы, затем помещает его в соответствующий список. Сообщение соотносят с темой по типу сообщения или по заранее определённому набору критериев, а также включает и само содержимое сообщения.

На основе вещания. В тот момент, когда очередь получает сообщение, она транслирует его по всем узлам, прослушивающим очередь. Сами узлы должны фильтровать данные и затем обрабатывать только интересующие сообщения.

В концепции SOA лежит основа микро сервисной архитектуры. Назначение у микро сервисной архитектуры такое же, что и у ESB: создавать единое общее корпоративное приложение из некоторого количфества специализированных приложений бизнес-доменов.

Самое основное различие микро сервисов и шины заключается в том, что ESB была основана в контексте интеграции отдельных приложений**,** для того чтобы получилось общее корпоративное распределённое приложение. А микро сервисная архитектура создалась в контексте очень быстро и постоянно изменяющихся бизнесов, которые в основном с нуля производят собственные облачные приложения.

В таком случае с ESB уже были созданы приложения, которые к нам не «относятся»**,** и в этом случае мы не смогли их изменить. А в другом случае с микро сервисами мы полностью контролируем все приложения (при всем этом в системе могут использованы и другие сторонние веб-сервисы).

Принцип построения или проектирования микро сервисов не желает глубокой интеграции. Микро сервисы принуждены соответствовать бизнес-концепции, ограниченному контексту. Они принуждены сохранить своё состояние, а также быть независимыми от всех других микро сервисов, и благодаря тому они меньше необходимы в интеграции. Таким образом низкая взаимозависимость и высокая связность пришли к тому замечательному побочному эффекту — уменьшению потребности в интеграции.
Самым основным недостатком архитектуры ESB есть очень сложное централизованное приложение, от всего этого зависели другие приложения. Приложение почти целиком убрано в микро сервисной архитектуре.

Также остались элементы, которые пронизывают всю остальную экосистему микросервисов. Но у них совсем меньше задач по сравнению с ESB. Например, для асинхронной связи между микросервисами до сих пор используется очередь сообщений, но это всего лишь канал для того, чтобы передать сообщения. В другом случае можно будет вспомнить шлюз экосистемы микросервисов, через который лежит весь внешний обмен данными.

Заключение: SOA очень сильно эволюционировала в последний десятилетия. Благодаря этой неэффективности старых решений и развитию других технологий, на сегодняшний день мы пришли к микро сервисной архитектуре. Эволюция проходила по классическому пути: в первую очередь сложные проблемы разбивались на более мелкие, но очень простые в решении.

Основную проблему сложности кода можно будет решать так же, как мы разбиваем монолитное приложение на совсем отдельные доменные компоненты либо разграниченные контексты. Но в другом случае с разрастанием команд и кодовой базы увеличивается необходимость в независимом развитии, масштабировании и развёртывании. Достичь такой независимости, упрочняя границы контекстов помогает SOA.

Список используемой литературы:

1 Открытый класс национального фонда подготовки кадров. Официальный сайт URL: www.opencalss.ru/blogs/41231
2 Информационные технологии в образовательном процессе Казанского национального университета. Образовательный портал URL: www.edurt.ru/index.php
3 Информационные технологии в образовании. Официальный сайт URL: www.physics.herzen.spb.ru
4 Сервисно-ориентированная архитектура. Официальный сайт URL: www.citforum.ru
5 Википедия SCORM. Свободная энциклопедия URL: www.ru.wikipedia.org
6 Charetter R. Why Software Fails. 2007

7 David Linthicum Information Technology Services Monol this Application Retrieved. 2009