УДК 004.051

ИСПОЛЬЗОВАНИЕ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ ПРИ РАЗРАБОТКЕ ПРИЛОЖЕНИЯ «АВТОМАТИЗИРОВАННАЯ СИСТЕМА КОММУНИКАЦИЙ СТУДЕНТОВ, АДМИНИСТРАЦИИ И ПОТЕНЦИАЛЬНЫХ РАБОТОДАТЕЛЕЙ»

В.И. БАКЛАН
(Представлено: канд. техн. наук, доц. И.Б. БУРАЧЁНОК)

В работе рассмотрены особенности микросервисной архитектуры, а также основные практики и подходы распределённой обработки запросов на примере разработанного приложения «Автоматизированная система коммуникаций студентов, администрации и потенциальных работодателей». Показаны преимущества использования микросервисной архитектуры и подходы к разделению сервисов.

Сегодня проблема трудоустройства выпускников после окончания ВУЗов чрезвычайно актуальна, так как молодой специалист сталкивается с довольно жёсткими условиями рынка труда. Проблема трудоустройства выпускников имеет место и для нашего ВУЗа, поэтому создание автоматизированной системы коммуникаций студентов, администрации и потенциальных работодателей, позволяющей установить связь между ними является одной из актуальнейших задач нашего региона.

Для реализации автоматизированной системы коммуникаций студентов, администрации и потенциальных работодателей использовался большой стек современных, пользующихся популярностью среди разработчиков технологий. Так как приложение подразумевает собой несколько бизнес-моделей с практически отсутствующими зависимостями, было принято решение разработать приложение, используя основные архитектурные приемы микросервисной архитектуры.

Микросервисная архитектура – распространенный подход к разработке современного программного обеспечения. Данная архитектура представляет собой два связанных между собой паттерна High Cohesion и Low Coupling, рассматривать которые имеет смысл только вместе. Иначе разработчика ждут большие проблемы. Причем, микросервис обязан быть независимым компонентом, некой небольшой монолитной программой, которая выполняет свою конкретную функцию, например, он может быть сосредоточен на обработке запросов в рамках одной сущности – доменной области. Безусловно основываясь на тех же принципах с разбиением на компоненты, сегодня пишется любая мало-мальски серьезная программа.

Так как каждый микросервис работает в собственном процессе, необходимо реализовать программный интерфейс приложения API (англ. Application Programming Interface), с помощью которого будет осуществляться взаимодействие с этим микросервисом. API для взаимодействия может работать с разными технологиями, например, если микросервис может работать асинхронно, есть смысл поддерживать even-driven API, например, с помощью программного обеспечения с открытым исходным кодом Kafka. Если необходимо часто читать информацию из миросервиса, может быть полезен GraphQL – язык запросов с открытым исходным кодом, более эффективная альтернатива Rest. Однако, если сторонним сервисам удобнее работать с Rest API, это тоже не должно стать проблемой. Таким образом, разделение приложения не маленькие независимые сервисы дает явный выигрыш с точки зрения независимого развития компонентов. Главное – своевременно обновлять API и избегать изменений в существующие контракты API.

Обозначение границ микросервиса – это один из самых сложных и важных шагов. От этого будет зависеть вся дальнейшая жизнь микросервиса и его целесообразность. Кроме того, в случае неправильного определения границ микросерсиса, команда, которая его поддерживает, столкнется с проблемами частого редактирования в попытках подстроиться под клиента. Как же определяется зона ответственности микросервиса? Основной принцип – это сформировать микросервис вокруг какой-либо бизнес-потребности. И чем компактнее зона ответственности, чем более формализовано её взаимоотношение с другими бизнес-компонентами, тем значительно проще процесс создания и поддержания самого микросервиса.

Следующей проблемой является развитие и доработка микросервиса. Во время каких-либо изменений очень важно не нарушить изначальную цель микросервиса и не зайти за границы бизнес-компонента. Когда границы микросервиса заданы, и он выделен в отдельную кодовую базу, защитить эти границы от постороннего влияния не сложно. Достаточно соблюдать единственное правило – доступ ко всем ресурсам микросервиса должен быть предоставлен исключительно с помощью API.

В микросервисной архитектуре есть несколько критичных мест, главное из которых – наличие сети между микросервисами. Сеть ненадежна по своей природе. Она может просто отказать, может работать плохо или фильтровать траффик. Причин может быть много. Поэтому микросервисы могут перестать обрабатывать запросы, возвращать ответ с задержкой и т.п. Для решений этой проблемы важно предусмотреть возможность подобных отказов и обрабатывать их корректно.

Безусловно микросервисный подход несет довольно много проблем, и их не сложно обнаружить и в дальнейшем учесть при разработке. В современном мире у команд разработчиков уже есть готовые

решений, в том числе, как на стороне dev, так и на стороне ops. Поэтому перед началом разработки микросервисмной архитектуры необходимо анализировать обе стороны проблемы – сторону разработки и сторону развертывания.

Одним из неоспоримых преимуществ микросервисной архитектуры является экономия денег. Так как компоненты изолированы друг от друга, то и нагрузка на компоненты распределяется по-разному. Например, какой-то микросервис может принимать больше запросов и выполнять более сложные операции над базами данных, какой-то – меньше запросов и отсутствие базы данных. Поэтому, для оптимизации трат, на каждый сервис нужно выделять разное количество ресурсов и увеличивать или уменьшать количество реплик независимо друг от друга.

Еще одной немаловажной темой в процессе разделения приложения на микросервисы является база данных. Хорошим решением является создание отдельной базы данных для каждого микросервиса. Это позволяет изолировать данные и быть уверенным в том, что модификации данных будет произведена исключительно с помощью определенного API. Конечно, в этом случае, возникает следующая проблема – транзакционность. Если у каждого сервиса разные базы данных, то запись в разные базы не будет транзакционной по определению. Например, если при оплате надо записать лог транзакции и оформленный заказ. В этом случае лог транзакции будет записал в базу данных сервиса, который работает с транзакциями, а лог заказа в базу данных сервиса, который работает с заказами. Для решения этой проблемы существует несколько паттернов распределенных транзакций. Так же имеется несколько способов реализации распределенных транзакций, основными из которых являются:

- двухфазная фиксация,
- согласованность в конечном счете и компенсация/SAGA.

Основной недостаток этого подхода заключается в том, что не обеспечивается изоляция при чтении. В вышеприведенном примере клиент может увидеть запись сформированного заказа, а уже через секунду запись будет удалена в ходе компенсирующей транзакции из-за того, что платежная система перестала принимать запросы.

Таким образом, использование микросервисной архитектуры позволяет значительно оптимизировать расходы на инфраструктуру, сократить время онбординга новых сотрудников, разделить приложение на изолированные бизнес-компоненты, но в это же время создает и дополнительные проблемы, которые необходимо решать. Выбор, использовать ли микросервисную архитектуру, должен приниматься осознанно с прицелом на дальнейшее развитие приложения. Возможно, в каких-то случаях, стоит реализовать гибридную систему, либо отказаться от микросервисов. В разработанном приложении «Автоматизированная система коммуникаций студентов, администрации и потенциальных работодателей» используется гибридная модель микросервисной архитектуры, в которой один основной backend-сервис, взаимодействующий с рядом сторонних сервисов, например, для работы с сообщениями. Данная архитектура позволяет в короткие сроки реализовать минимально жизнеспособный продукт MVP (англ. Minimum Viable Product), и в дальнейшем при успехе MVP развить систему до полноценной микросервисной архитектуры.

ЛИТЕРАТУРА

1. Building Microservices: Designing Fine-Grained Systems // Sam Newman. – O'Reilly Media, Inc, 2015. – 280 с.
2. Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith // Sam Newman. – O'Reilly Media, Inc, 2019. – 310 с.