ЛУЦЕНКО Д. Ю., ПОЛЯКОВА Л. П. РАЗБИЕНИЕ МОНОЛИТНОГО ПРИЛОЖЕНИЯ НА МИКРОСЕРВИСЫ С ИСПОЛЬЗОВАНИЕМ ПАТТЕРНА STRANGLER

УДК 004.4; 004.42; 339.37, ВАК 05.13.11, ГРНТИ 50.05.03

Разбиение монолитного приложения на микросервисы с использованием паттерна Strangler

Breaking a monolithic application into microservices using the Strangler pattern

Д. Ю. Луценко[1], Л. П. Полякова[2]

[1]Санкт-Петербургский политехнический университет Петра Великого, г. Санкт-Петербург;
[2] Ухтинский государственный технический университет, Воркутинский филиал; г. Воркута

D. Yu. Lutsenko[1], L. P. Polyakova[2]

[1]Peter the Great St. Petersburg Polytechnic University, St. Petersburg;
[2]Ukhta State Technical University, Vorkuta Branch, Vorkuta

В проектах, которые существуют не один год, нередко возникает необходимость работы с устаревшей кодовой базой. С появлением новых технологий и увеличением масштабов продуктов компаниям по всему миру стало сложнее продолжать поддерживать монолитные и старые приложения и в то же время оставаться конкурентоспособными на рынке. Чтобы идти в ногу со временем и сохранять актуальность необходимо постоянно развивать свой продукт. Лучшим подходом в данном случае является преобразование унаследованных монолитных приложений в несколько небольших микросервисов.

Elimination of problems related to outdated database, outdated database. With the advent of new technologies and the scale of food products around the world, it has become more difficult to maintain monolithic and legacy applications. To keep up with the times and the current relevance you need to constantly increase your product. The best approach in this case is to transform legacy monolithic applications into multiple small microservices.

Ключевые слова: *приложение, команда, сервис, паттерн, облачные микросервисы*

Keywords: *application, team, service, pattern, cloud microservices*

Введение

Поддержка устаревшего приложения часто приводит к излишним трудозатратам и дополнительной работе по следующим причинам:

1. Недостаточное количество или полное отсутствие тестов. Нарушение принципа единой ответственности;
2. Высокая сложность;
3. Невозможность масштабирования отдельных компонентов;

4. Тесная связь между компонентами;
5. Накопление технического долга.

Смоделируем ситуацию: несколько команд работают над приложением с монолитной архитектурой (Рисунок 1). Если команда B заносит некорректный код в среду самого низкого уровня, то она автоматически блокирует работу всех оставшихся команд, пока команда B не внесет исправления в проблемный код. Поскольку все команды вносят изменения в одно и то же место, при этом завися друг от друга, то высока вероятность того, что изменения одной команды разработки могут парализовать работу другой. Также невозможно будет произвести деплой всего приложения, что приведет к задержкам и дополнительным затратам.

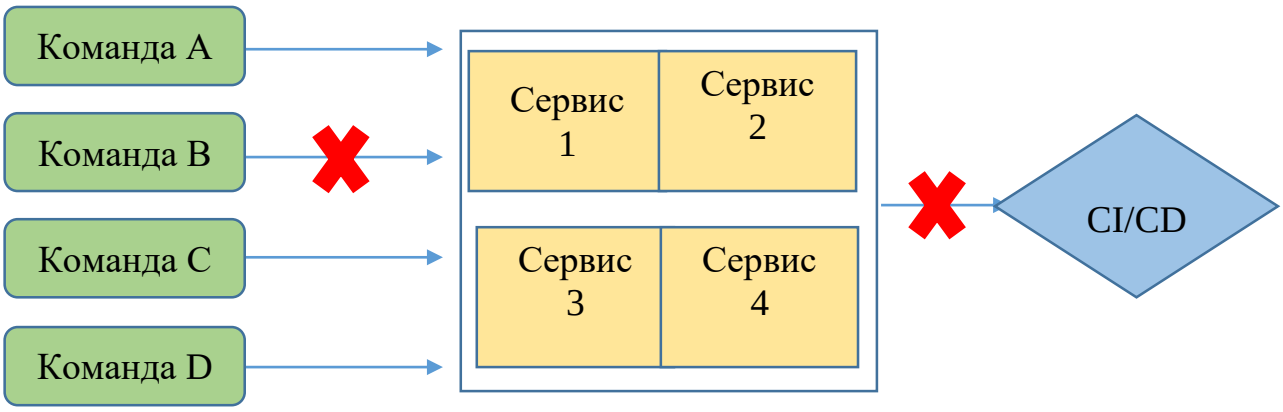

Рисунок 1. Схема деплоя и разработки монолитного приложения

В проектах с большим количеством устаревших технологий и монолитной архитектурой, несмотря на то, что над внесением изменений работает несколько команд разработчиков, зачастую не удается производить регулярное и своевременное развертывание новой версии приложения.

С годами сложность кода подобных приложений только возрастает, а компоненты все более тесно связываются друг с другом, что затрудняет процесс качественного автоматизированного тестирования всего кода. Внесение простого изменения в один из классов может легко нарушить работу существующего функционала другого взаимосвязанного класса.

В современных реалиях данная модель не является надежной и эффективной. Намного целесообразней разделить монолитное приложение на мелкие, слабосвязанные сервисы [2]. Такой подход позволит разработчикам чаще и проще внедрять изменения в свой продукт.

Произвести рефакторинг в уже существующем монолите или переписать все с нуля?

Переписывание большого монолитного приложения с нуля требует немалых усилий и сопряжено с определенными рисками. Одна из самых больших проблем — хорошее понимание унаследованной системы. Также придётся разобраться с техническим долгом устаревшей системы. Кроме того, использование

новой системы будет невозможно пока она не будет завершена. В зависимости от размера и сложности приложения это может занять довольно много времени. В течение этого периода, поскольку идет разработка новой системы, улучшение и поддержка текущей платформы будут приостановлены. Паттерн strangler позволяет снизить вышеуказанные риски. Вместо того, чтобы переписывать все приложение, в него будут постепенно вноситься изменения. Как следствие, ценность новой функциональности будет достигается намного быстрее. Также можно упростить рефакторинг, если следовать принципу единой ответственности и писать слабосвязанный код.

Что такое паттерн strangler?

Strangler - это популярный шаблон проектирования для постепенного преобразования монолитного приложения в микросервисы путем замены определенной функциональности новыми сервисами [3]. Как только новая функциональность готова, то соответствующий сервис начинает работать, а старый компонент полностью выводится из эксплуатации. Любая новая разработка выполняется в рамках новых микросервисов, а не в составе монолита. Данный подход позволяет получить более удобный код для дальнейшей разработки. На приведенной ниже диаграмме (Рисунок 2) один из модулей проекта выносится из монолита и превращается в независимый развертываемый сервис с собственным CI/CD [4]. Команда A теперь не будет зависеть от остальных.

Чтобы упростить переход от монолита к микросервисам, необходимо четко разделить процессы, протекаемые в приложении. При реализации паттерна strangler следует придерживаться трех основных концепций: преобразование, сосуществование и устранение [5].

Сначала разрабатывается новый компонент, затем в течение определенного периода времени существует как новый, так и старый компоненты и, наконец, удаляется старый.

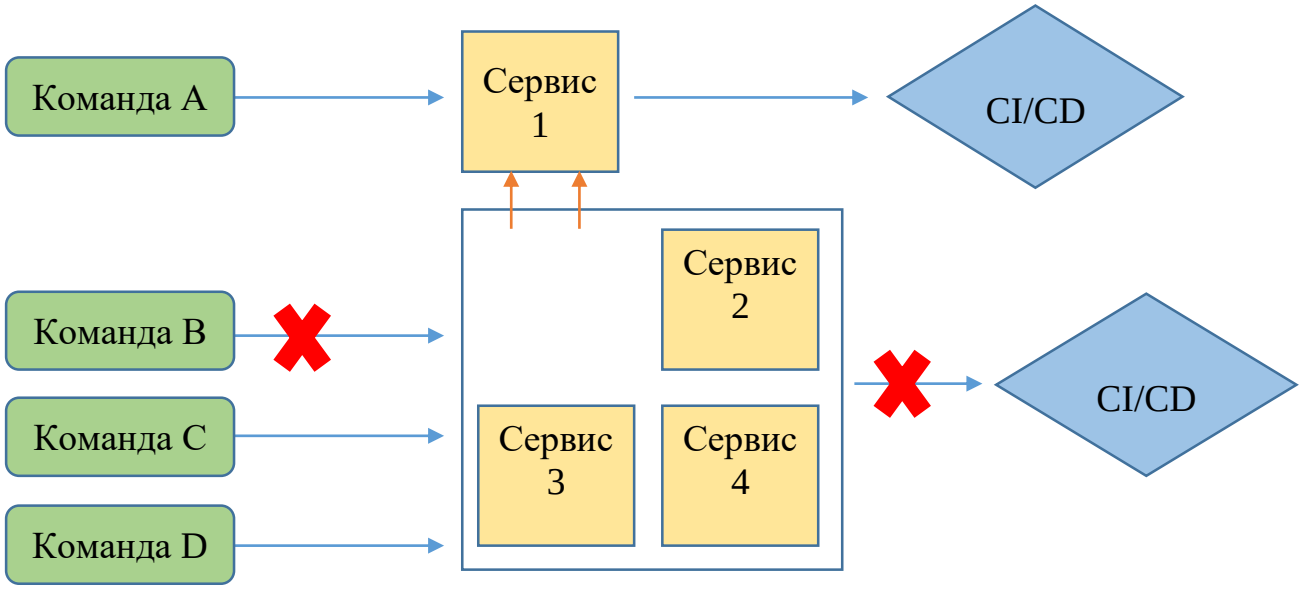

Рисунок 2. Применение паттерна strangler

Как выбрать компоненты, подлежащие рефакторингу в первую очередь?

Первоначально весь трафик направляется в старую версию приложения. Как только новый компонент готов, то его можно параллельно протестировать с уже существующим монолитом кодом. Обе версии компонента должны работать одновременно в течение определенного периода времени. Следует учесть, что переходная фаза может длиться довольно долго. Когда новый компонент будет полностью разработан и протестирован, то можно будет исключить его из монолитного приложения.

Для начала следует выбрать несколько простых компонентов. Это гарантирует, что перед переходом к более сложным частям приложения процесс рефакторинга системы будет достаточно изучен и понятен [5]. После перейти к компонентам, которые обладают следующими особенностями:

- имеют хорошее покрытие тестами и небольшой технический долг;
- лучше всего подходят для переноса в облако и имеют требования к масштабируемости;
- к которым часто предъявляются бизнес-требования и, следовательно, подверженные регулярным изменениям.

Перенос частей приложения в облачный сервис довольно непростая задача. Шаблон проектирования strangler поможет сделать данный процесс более удобным и безопасным, поскольку работа будет вестись с небольшими компонентами [3]. Тогда миграция в облако не составит большого труда, ведь она будет выполняться поэтапно и небольшими частями. Снижение сложности и связанности приложения позволяет быстрее реализовывать новую логику и функционал. В дополнение ко всему повышается масштабируемость приложения. А наличие автоматизированного CI/CD значительно упрощает развертывание микросервисов и может значительно упростить разделение монолита. В итоге условная схема приложения будет выглядеть следующим образом (Рисунок 3):

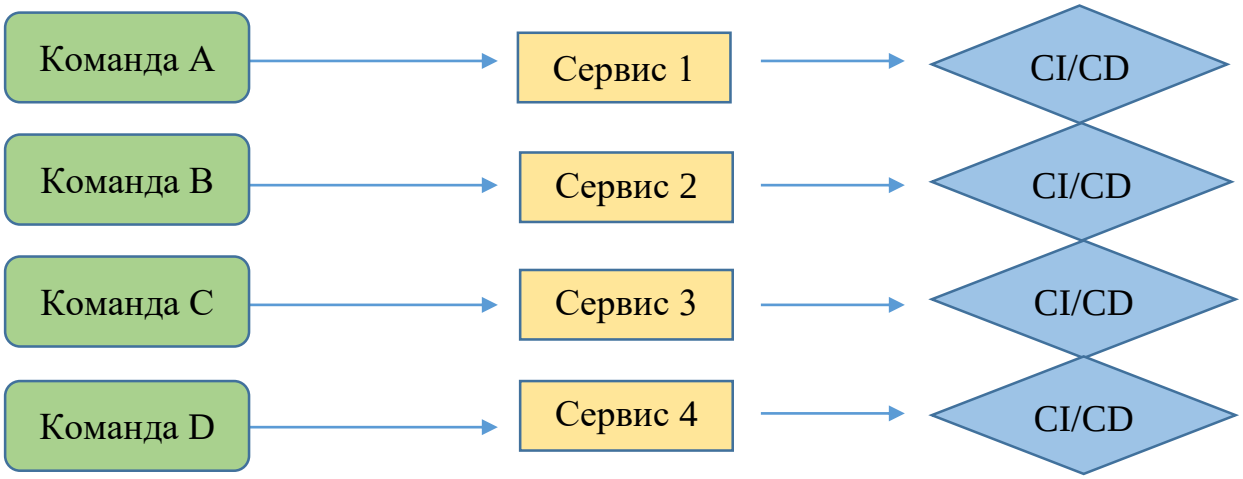

Рисунок 3. Схема приложения после окончательного разбиения на микросервисы

Заключение

Преобразование существующего устаревшего монолитного приложения в облачные микросервисы - полезная современная практика, но это может оказаться не самой тривиальной задачей, для решения которой требуется создание хорошей архитектуры будущего приложения и своевременное планирование. В этой статье был рассмотрен шаблон проектирования под названием strangler, с помощью которого подобные преобразования различных систем можно осуществлять проще и удобнее. Разработчики могут продолжать приносить пользу, выполняя все новые задачи в рамках новых сервисов и постепенно отказываться от монолита.

Список использованных источников и литературы

1. Вольф Э. Гибкая программная архитектура микросервисов. – Эддисон-Уэсли, 2015. – 375 с.

2. Введение в микросервисы // статьи о программных системах [Электронный ресурс]. – Режим доступа: https://specify.io/concepts/microservices (дата обращения: 25.08.2021).

3. Как использовать шаблон душителя для модернизации микросервисов [Электронный ресурс]. – Режим доступа: https://www.castsoftware.com/blog/how-to-use-strangler-pattern-for-microservices-modernization (дата обращения: 26.08.2021).

4. Душитель фиговый узор [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/en/azure/architecture/patterns/strangler-fig (дата обращения: 26.08.2021).

5. Часть 3: выбор правильной стратегии для миграции монолитного приложения на архитектуру на основе микросервисов [Электронный ресурс]. – Режим доступа: https://capgemini-engineering.com/us/en/insight/part-3-choosing-the-right-strategy-to-migrate-your-monolithic-application-to-a-microservices-based-architecture (дата обращения: 27.08.2021).

List of references

1. Wolff E. Microservices Flexible Software Architecture. — Addison-Wesley 2015. — 375 p.

2. Introduction into Microservices // specify.io: Software System Articles. URL: https://specify.io/concepts/microservices (date of the application: 25.08.2021).

3. How to use strangler pattern for microservices modernization // www.castsoftware.com: Software Intelligence Pulse. URL: https://www.castsoftware.com/blog/how-to-use-strangler-pattern-for-microservices-modernization (date of the application: 26.08.2021).

4. Strangler Fig pattern // docs.microsoft.com: документация Microsoft. URL: https://docs.microsoft.com/en/azure/architecture/patterns/strangler-fig (date of the application: 26.08.2021).

5. Part 3: choosing the right strategy to migrate your monolithic application to a microservices-based architecture // capgemini-engineering.com: Capgemini-engineering. URL: https://capgemini-engineering.com/us/en/insight/part-3-choosing-the-right-strategy-to-migrate-your-monolithic-application-to-a-microservices-based-architecture (date of the application: 27.08.2021).