УДК 004.051

СРАВНЕНИЕ ПОДХОДОВ МОНОЛИТНОЙ АРХИТЕКТУРЫ И МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ ПРИ РЕАЛИЗАЦИИ СЕРВЕРНОЙ ЧАСТИ ВЕБ-ПРИЛОЖЕНИЯ

Никитин И.В.

студент 2 курса магистратуры

Московский государственный технический университет им. Н. Э. Баумана

Россия, г. Москва

Гриценко Т.Ю.

студент 2 курса магистратуры

Московский государственный технический университет им. Н. Э. Баумана

Россия, г. Москва

Аннотация

В статье рассматриваются два принципиально разных подхода к верхнеуровневой организации архитектуры современного веб-приложения: монолитная архитектура и микросервисная архитектура. Представлено описание работы каждого подхода, а также приведены плюсы и минусы их использования.

Ключевые слова: веб-приложение, монолитная архитектура, микросервисная архитектура

COMPARISON OF MONOLITHIC ARCHITECTURE AND MICROSERVICE ARCHITECTURE APPROACHES IN IMPLEMENTING THE SERVER PART OF WEB APPLICATION

Nikitin I.V.

master student

Bauman Moscow State Technical University

Russian Federation, Moscow

Gritsenko T.Y.

master student

Bauman Moscow State Technical University

Russian Federation, Moscow

Annotation

In article two different ways of organization modern web application's architectures discovered: monolithic architecture and microservice architecture. Description for both approaches presented, also covered their positive and negative qualities.

Keywords: web-application, monolithic architecture, microservice architecture

Введение

В последний годы люди в своей повседневной жизни все активнее используют веб-приложения, многие из которых значительно упрощают нам жизнь. Например, достаточно простое, но удобное приложение по сохранению заметок. С помощью такого приложения мы можем сохранить необходимую нам информацию и вернуться к ней в любое удобное время и на любом устройстве.

И чем чаще используется такое приложение, тем сильнее возрастает нагрузка на сервера этого приложения. Одной из основных метрик хорошей работоспособности сервера является rps – request per seconds - метрика, показывающая сколько запросов способен обработать сервис за одну секунду. Если этот показатель для какого-то веб-приложения небольшой, то запросы от пользователей не успевают быстро обработаться и пользователю приходиться ждать, когда его запрос дойдет до обработки. Чаще всего в такой ситуации

возникает мысль о том, что приложение как-то неправильно работает и возвращаться к нему в будущем не будет желания.

Увеличение rps-метрики можно добиться двумя способами: улучшением технических характеристик серверов или архитектуры приложения. Первый путь улучшения требует достаточно больших финансовых вложений, но из-за плохой организации инфраструктуры вложения могут не окупиться. Второй же может дать значительный прирост к нагрузочным способностям приложением при сохранении характеристик серверов.

В статье рассматриваются верхнеуровневые способы организации архитектуры современного веб-приложения и проводится сравнение плюсов и минусов двух принципиально разных подходов: монолитная архитектура и микросервисной архитектуры. Помимо этого, будут рассмотрены такие аспекты функционирования веб-приложения, как его тестирование и релиз. Для наглядного сравнения подходов двух архитектур будут рассмотрены такие крупные приложения, как Wix и 2GIS, которые совершили переход от одного вида архитектуры к другому. В конце статьи будет составлен список рекомендаций по применению каждого типа архитектуры.

Обзор монолитной архитектуры

Монолитная архитектура является классическим подход в реализации приложений любой направленности и нагрузки. Она представляет из себя один большой сервис (монолит), который содержит и обрабатывает в себе всю бизнес-логику приложения. Упрощенная схема работы такого сервиса представлена на рис. 1.

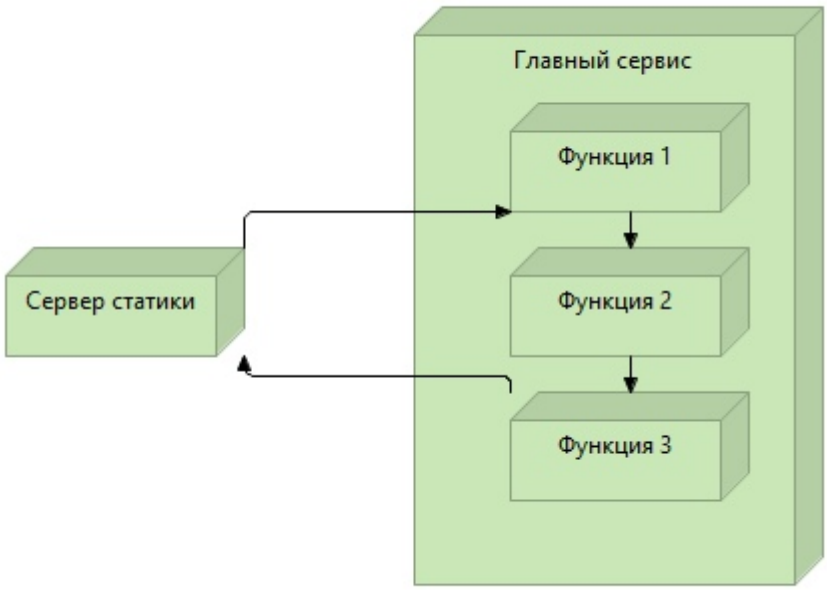

Рис. 1 – Схема работы монолитного приложения

Для понимания особенностей работы монолитной архитектуры проследим путь пользовательского запроса. Допустим, пользователь запрашивает информацию с описанием товара от сервиса. После того, как с клиента был отправлен запрос, он попадет на специальный сервер, который используется для отдачи статического контента пользователю, если пользователь его запросил, или произойдет перенаправление запроса на монолит, если запрос требует работы с данным. Этот сервис призван разгрузить работу основного монолита. Далее запрос попадает непосредственно на монолит. В нем запрос вначале разбирается по параметрам для того, чтобы определить какие функции необходимо будет вызвать и какую информацию из них получить. После этого происходит вызов функций. Сами функции могут представлять из себя, например, запрос к базе данных для получения данных, фильтрация известного набора данных, сбор и слияние данных от нескольких других функций и так далее. У каждой функции своя зона ответственности, поэтому разбор параметров является важной частью работы программы. При

обращении к неправильной функции можно получить ответ, который пользователь не ожидает увидеть. После получения ответа от всех функций формируется ответ, и он отправляется обратно пользователю.

Главное преимущество такой системы – вся бизнес-логика приложения находится в одном месте. При необходимости изменить какой-то параметр, можно достаточно легко (при хорошей организации структуры проекта) найти зависимости всех параметров между собой. К минусам такого подхода можно отнести одну точку входа приложения, которая является единой точкой отказа всей системы. Также в случае большой кодовой базы становится сложно внести в нее изменения.

Однако, при использовании такого подхода на больших высоконагруженных проектах возникает ряд проблем. Допустим несколько сотен пользователей одновременно отправили запрос к такому серверу. У монолита одна точка входа, а значит все запросы должны пройти через нее. Это может значительно увеличить время ожидания ответа от приложения из-за того, что только небольшое количество запросов успевает обработаться. Далее допустим, что необходимо сделать несколько запросов к базе данных. Даже при условии, что сервер работает в многопоточном режиме, а база данных поддерживает большое количество соединений, возникнет ситуация, когда функции придется ждать в очереди на выполнение запроса к базе данных. Однако без данных от базы данных функция не может продолжить работу, так как в монолите функции выполняются последовательно и бывает сложно распараллелить обработку одного запроса, а значит запрос будет простаивать в очереди, ожидая возможности возобновить свою работу. Таким образом время, которое пользователь может потратить на ожидание ответа приложения будет большим, а rps маленьким.

Ситуацию можно несколько улучшить, распараллелив выполнение функций с помощью закупки дополнительного оборудования. Допустим,

параллельно текущему серверу запустить еще один. Для этого необходимо настроить такую же инфраструктуру, как и на первом сервере и явно указать серверу статики о наличии второго сервера, на который можно отправить запросы. После этого общий rps несколько повысится. Однако данный способ с точки зрения финансов не является удачным. Закупка нового оборудования требует больших средств, а увеличение метрик может быть не таким существенным как хотелось бы.

Помимо всего вышеперечисленного, в определенный момент может возникнуть ситуация, когда внедрение нового функционала станет дорогой по времени задачей. Из-за того, что все находится в одном месте, бывает сложно найти нужный кусок приложения, который необходимо исправить.

Более подробную информацию о монолитной архитектуре можно прочитать из источника [6], в котором монолит рассматривается как шаблон проектирования веб-приложения.

Тестирование

Важным критерием разработки любого продукта является возможность обеспечить качество разрабатываемого продукта. Тестирование является частью комплекса мероприятий для проверки и обеспечения качества продукта. В современных веб-приложениях используется несколько различных направлений тестирования: юнит-тестирование, авто-тестирование, интеграционное тестирование и т.д. Рассмотрим чуть более подробно данные виды тестирования при разработке монолитного приложения.

Юнит-тестирование – вид тестирования, в котором проверяется функционал отдельно взятой функции без привязки к окружению. Подход юнит-тестирования заключается в вызове функции со всеми возможными комбинациями входных переменных и последующим сравнении возвращаемого результата с ожидаемым. Независимо от архитектурного подхода на разрабатываемый функционал необходимо писать юнит-тесты. Однако в случае

монолитной архитектуры возникает проблемы с поддержанием и написанием юнит-тестов. На начальных этапах жизни приложения данное тестирование не представляет никаких трудностей, однако по мере роста кодовой базы приложения возникают проблемы с поддержкой. Из-за большой кодовой базы бывает сложно найти, разобраться и поправить определенный юнит-тест или написать новый. В результате в какой-то момент возникает ситуация, когда есть большая кодовая база, которая частично покрыта юнит-тестами, часть из которых может быть не актуализирована под новый функционал, а на часть функционала вообще нет тестов.

Авто-тестирование – вид тестирования, в котором в автоматическом режиме проверяется какой-либо функционал или сценарий работы приложения. Отличается от юнит-тестирования тем, что проверяет не отдельный небольшой функционал, а часть работы бизнес-логики приложения. Также является важной частью обеспечения качества продукта, так как позволяет не тратить время на ручную проверку функционала, а выполнить ее в автоматическом режиме. Данный вид тестирования сталкивается с теми же проблемами, что и юнит-тестирование: сложность разработки при большой кодовой базе. Из-за обилия бизнес-логики в монолите приложения, написать и поддержать автотесты на весь функционал проблематично, так как изменение какой-либо части бизнес-логики потребует исправлений во всех автотестах, в которых использует данный кусок логики.

Интеграционное тестирование – вид тестирования, в котором проверяется взаимодействие сервиса при работе с другими сервисами. Интеграционное тестирование может являться частью авто-тестирования. Примером интеграционного тестирования может быть проверка возможности получения каких-либо данных из базы данных. Так как в монолитной архитектуре все сосредоточенно в одном сервисе, то интеграционное тестирование может сводиться к проверке поведения при работе с серверами баз данных.

Помимо вышеупомянутых видов тестирования рассмотрим еще один вид – нагрузочное. Целью данного тестирования является определение максимально возможной нагрузки на систему, а также проверка поведения при постепенном увеличении нагрузки. Необходимость данного вида тестирования заключается в том, что при неспособности системы выдержать большое количество запросов, запросы от пользователей в лучшем случае будут попадать в очередь и выполняться по мере обработки остальных запросов из очереди. Это приведет к увеличению времени обработки запроса. В худшем случае система вообще перестанет работать при достижении определенной нагрузки. В подавляющем большинстве базы данных являются тем местом, скорость работы которого сильно зависит от нагрузки. Поэтому в первую очередь именно на них необходимо смотреть при запуске нагрузочных тестов.

В результате можно увидеть, что проблема при тестировании монолитной архитектуры сводится к большому количеству элементов, которые необходимо проверить. В определенный момент становится достаточно сложно поддержать и обеспечить необходимое качество приложения, использующего подход монолитной архитектуры.

Микросервисная архитектура

В отличие от монолитной архитектуры, микросервисная архитектура является более современным подходом к организации большого веб-приложения. Упрощенная схема приложения, работающего по принципам микросервисной архитектуры, представлена на рис. 2.

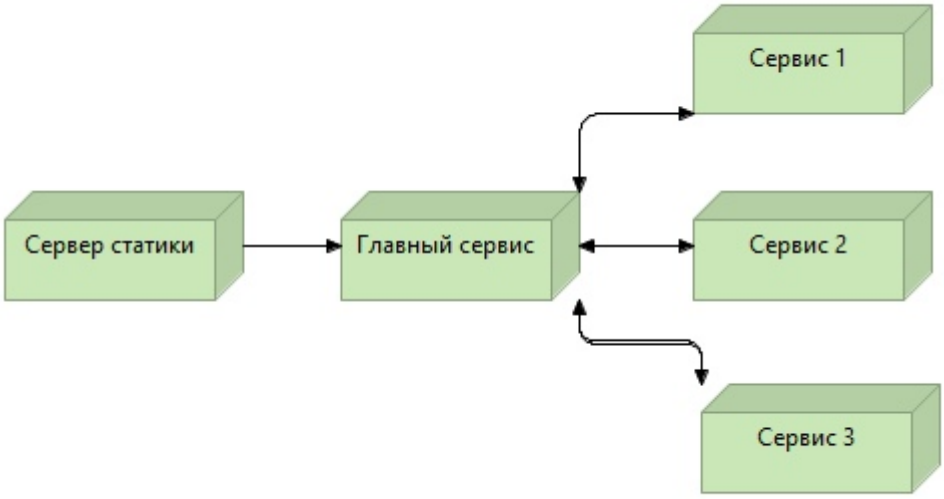

Рис. 2 – Схема работы микросервисной архитектуры

Так же, как и при монолитной архитектуре, при отправке запроса с клиента, он сначала попадает на сервер статики, а затем перенаправляется на основной сервер. На нем также определяются параметры и данные, которые необходимо получить. Однако дальнейший принцип работы значительно отличается от монолита: вместо того, чтобы обрабатывать все запросы в одном месте, основной сервер рассылает подзапросы к микросервисам, которые возвращают определенные данные. Микросервисы похожи на функции монолита: каждый такой сервис имеет свою зону ответственности и чаще всего не имеет доступа к другим микросервисам. После обработки каждым микросервисом подзапроса и отправки ответа основному серверу, данные в нем собираются и отправляются назад клиенту.

Главным преимуществом такого подхода является разбиение функционала приложения на маленькие независимые модули, которые могут работать в параллель. Сами такие модули запущены как отдельные серверы, работа которых с основным сервером происходит параллельно. При таком

подходе к архитектуре приложения сервер может отправить сетевой запрос на получение данных от микросервиса. Сетевой запрос в определенных ситуациях может выполняться дольше, чем вызов обычной функции, так как при отправке запроса необходимо время на установку соединения, обмен данными между сторонами-участниками, обработку непосредственно самого запроса микросервисом. Однако, большинство современных фреймворков и библиотек, использующихся при серверной разработке, умеют выполнять асинхронные запросы или держать открытые соединения для того, чтобы не было необходимости тратить время на установку соединения. Такие запросы не блокируют выполнение основного кода в ожидании ответа сервер. Таким образом можно повысить количество обрабатываемых запросов за счет выполнения большого количества асинхронных действий. К минусам такого подхода стоит отнести инфраструктурные проблемы. Так как при разбиении функционала на микросервисы необходимо четко понимать в какой момент к какому сервису необходимо обратиться, необходимо разработать достаточно стабильную и сложную инфраструктуру, в которой запросы между микросервисами будут выполняться быстро и надежно. Также добавление новых микросервисов, удаление существующих и перераспределение ресурсов между существующими микросервисами не должно вызывать больших сложностей.

Подход с использованием микросервисов очень удобен при разработке большого приложения. Как было сказано ранее, каждый такой микросервис является маленьким и независимым. Это означает, что, как и функции в монолите, у каждого микросервиса есть своя зона ответственности. Но теперь вся бизнес-логика разбита на несколько кусков, каждый из которых расположен отдельно. А значит поддерживать такой код гораздо проще, так как нет необходимости в понимании того, как работает все приложение. Достаточно

понимать логику работы конкретного микросервиса, чтобы внести в него какие-то изменения.

Также стоит отметить повышенную отказоустойчивость по сравнению с монолитным подходом. В случае одного большого монолита при каких-то поломках в аппаратуре сервера запрос просто может быть не обработан, что приведет к невозможности работы с таким приложением. В свою очередь отказ какого-то из микросервисов хоть и может лишить пользователя основного функционала приложения, будет возможно донести до него эту информацию, так как остальные микросервесы продолжают свою работу. А вышедший из строя микросервис гораздо проще заменить, чем полностью все приложение.

При всех своих достоинствах, подход с микросервисами имеет ряд минусов. При запуске приложения необходимо помимо главного сервиса также запустить и микросервесы, а значит необходимо для каждого из них настроить CI (Continuous Integration), развернуть всю необходимую инфраструктуру, удостовериться в том, что запросы доходят до микросервиса. На начальных этапах небольшого веб-приложения временные затраты на все эти действия могут быть критическими, так как могут занять время, которое можно потратить на разработку продукта. Также к минусам можно отнести некоторую сложность в организации тестовой инфраструктуры. Например, микросервис, который использует машинное обучение для составление какого-то набора данных не сможет эффективно работать с маленьким набором тестовых данных. А значит необходимо правильно настроить такие микросервисы, чтобы определенные запросы могли отправляться в боевые микросервисы, а какие-то в тестовые, при этом не должно возникать конфликтов или критических ситуаций, когда тестовые сервера слишком сильно нагружают боевые.

Более подробное описание принципов работы микросервисов можно найти в книге С. Ньюмена «Создание микросервисов» [10].

Тестирование

Как и в случае монолитной архитектуры, возникает необходимость в проверке качества работы отдельных микросервисов. Рассмотрим различные варианты тестирования при работе с микросервисами.

Микросервисы, в отличие от монолита, имеют меньшую кодовую базу, а это значит, что покрыть микросервис юнит-тестами и поддерживать их значительно проще.

Похожая ситуация с авто-тестами. Из-за того, что микросервис содержит только часть бизнес-логики приложения, покрыть эту часть автотестами, которые будут выполняться только в рамках этого микросервиса, гораздо проще.

Однако, в случае покрытия самого сервиса тестами, необходимо также убедиться, что само приложение в связке всех микросервисов работает правильно. Для этого необходимо также написать автотесты, которые повторяют основные сценарии пользователя.

В отличие от монолитной архитектуры, важность интеграционного тестирования становится более весомой. Так как сетевое взаимодействие - единственный вид взаимодействия между микросервисами, интеграционному тестированию необходимо уделить большее внимание, чем в монолите. В тест-план для интеграционного тестирования входит проверка возможности взаимодействия с другими микросервисами, а также результат этого взаимодействия. Это может взаимодействие как между Main Service и любым из трех сервисов (см. рис. 1.2), так и взаимодействие сервисов между собой. По результатам интеграционного тестирования можно судить о доступности микросервиса другим микросервисам.

Помимо возросшей важности интеграционных тестов, также возрастает необходимость в правильно составленных нагрузочных тестах. Кроме того, что микросервисы общаются с базой данных, они могут общаться между собой. А это значит, что может возникнуть ситуация, когда два микросервиса не могут

нормально общаться между собой, так как большая нагрузка между ними не позволяет быстро обрабатывать запросы. В результате, помимо проверки нагрузки при общении микросервисов с базой данных возникает необходимость в проверки их общения между собой.

Анализируя выше описанные виды тестирования, можно увидеть, что там, где возникают трудности в тестировании монолита, в тестировании микросервиса их нет, и наоборот. В связи с этим нельзя однозначно утверждать, что микросервис или монолит лучше в плане тестируемости, так как у каждого подхода свои сложности при подготовке и разработке тестов.

Анализ необходимости перехода веб-приложения с монолитной архитектуры к микросервисной

В настоящее время можно увидеть множество докладов и выступлений на тему перехода крупного веб-приложения с монолитной архитектуры на микросервисы. В качестве примеров таких приложений будут рассмотрены приложения Avito, Wix, 2GIS.

Avito является одним из крупнейших приложений Российского сегмента Интернета и входит в топ 3 классифайдов мира. Каждый день к серверам Avito совершается более одного миллиона запросов, при этом само по себе приложение является быстроразвивающимся и подстраивающимся под текущую ситуацию на рынке [4]. Во время запуска приложения в 2007 году оно представляло собой монолит, однако последние пару лет идет работа по переходу от монолитной архитектуры к микросервисной. Причины такого перехода просты: учитывая большую функциональность всех систем приложения, очень сложно поддерживать такое приложение, что приводит к невозможности быстрого развития, а также к проблемам, связанным с большой нагрузкой на сервера приложения, так как нет распараллеливания обработки запросов. В результате, на данный момент большие усилия прикладываются к тому, чтобы в ближайшее время данный переход был произведен.

Wix представляет собой приложения, являющееся конструктором сайтов. Аудитория приложения составляет более 80 миллионов человек, а функционал самого приложение состоит из двух частей: инструменты для создания сайта и функционал для поддержания ранее созданных сайтов в рабочем состоянии [2]. Аналогично Avito, в момент запуска приложение представляло собой монолит, однако по мере роста аудитории приложения возникла необходимость в децентрализации данных, которая позволить избежать падения всей системы в результате отказа или неправильной работы какой-либо ее части. В результате, на текущий момент в Wix существуют два больших отдельных микросервиса, каждый со своей зоной ответственности, слабосвязанные между собой.

2GIS – международная картографическая компания, предоставляющая электронные справочники городов. Как и выше описанные приложения, изначально приложение представляло собой монолит, но по мере роста приложения возникли проблемы со скоростью работы приложения (как внутренних процедур, так и обработка запросов от пользователей), а также частые и долгие релизы приложения привели к тому, что постепенно монолит превратился в множество микросервисов [7]. В результате такого перехода стало возможно оптимизировать работу кода в некоторых местах.

С похожими проблема столкнулись и в Uber. В начале своего развития приложение представляло собой монолит, однако в процессе развития стало проблематичным добавление нового функционала, исправление ошибок или решение каких-либо технических вопросов. В результате было принято решение перейти на микросервисы. Однако, возник ряд проблем, которые необходимо было решить. К таким проблемам отнесли общение микросервисов (была решена с помощью библиотеки Apache Thrift), безопасность (Apache Thrift тоже помог в решении этой проблемы) и масштабируемость (для решения этой проблемы была написана собственная библиотека). [12]

Учитывая преимущества, которые дают микросервисы, у них есть определенное количество минусов, на которые нельзя не обратить внимание. Так, например, в одной из статей Райффайзен банка [4] ведутся рассуждения о том, что такие вещи, как вход нового разработчика в монолит и в микросервис или возможность частого релиза, ставятся под вопросом. Также затрагиваются темы усложнения инфраструктуры, так как необходимо настраивать инфраструктуру для каждого микросервиса отдельно, а при изменении или добавлении какого-либо функционала необходимо выполнять релиз до нескольких сотен микросервисов прежде чем пользователь увидит что-то новое. В конце высказывается мысль о том, что при правильной организации процессов работы над проектом любой вид архитектуры может быть удачным.

На схожие минусы обращается внимание в статье компании Флант [9]. В ней автор, при всей привлекательности микросервисного подхода, анализирует такие проблемы микросервисов, как возросшие сложности разработки и эксплуатации, сложности взаимодействия и версионирования. В результате в конце статьи дается небольшая рекомендация по применению микросервисов.

Еще один примером, который показывает, что микросервисная архитектура может нанести больше вреда чем пользы, является другой доклад компании Флант [5]. Ссылаясь на графики и вывода Мартина Фоулера [11], а также анализируя опыт работы с различными приложениями, докладчиком был сделан вывод о том, что на ранних этапах жизни приложения использование микросервисов может только нанести вред будущему приложению так как усложнит его. При этом и в [5], и в [11] говорится о том, что начинать разработку надо именно с монолита, так как построенные с самого начала на микросервисах приложения будут иметь серьезные проблемы в будущем, частично связанные с тем, что нет четкого понимания предметной области.

Для компании Segment переход на микросервисную архитектуру принес больше вреда, чем пользы. Первоначально приложение было монолитом и в

какой-то момент было переписано с использованием микросервисов. Однако стали возникать проблемы, связанные с поддержкой каждого отдельного сервиса. В каждом из них находились свои версии библиотек, что приносило трудности при автоматизации процессов. Также каждый микросервис принимал различную нагрузку, и настройка существующих и новых сервисов представляла собой сложный и дорогостоящий процесс. В результате от идеи микросервисов отказались в пользу одного монолита [8].

В результате анализа различных подходов можно увидеть, что на практике нет единственно правильного решения, которое сможет решить все проблемы, так как каждый из подходов обладает своими плюсами и минусами. Однако, на данный момент уже можно попробовать составить список рекомендаций о применении того или иного подхода в различных ситуациях.

Рекомендации по применению монолитной и микросервисной архитектуры

При выборе архитектурного подхода стоит обратить внимание на следующие характеристики будущего приложения: rps, схема базы данных, время до запуска minimum viable product (MVP), технические возможности серверов и дальнейшая масштабируемость.

Монолитную архитектуру выгодно использовать в небольших проектах, расширение которых либо не планируется, либо будет происходит за счет установки новых серверов. Примером таких приложений могут быть небольшие Интернет-магазины или стартапы, которые необходимо быстро запустить, rps которых меньше 1к запросов, время от начала разработки до запуска MVP должно быть минимально возможным (до 1-1,5 месяцев), обращения к основной базе данных происходят редко, количество записей в самой базе меньше одного миллиона, а добавление какого-то нового функционала планируется в виде небольших доработок и не будет сильно

менять сервис. Такого подхода рекомендуется придерживаться на первых этапах жизни приложения.

Микросервисную архитектуру, в свою очередь, удобно использовать в уже существующих проектах, которые постоянно расширяются и дорабатываются, количество запросов и данных увеличивается. Примером таких приложений могут быть приложения-сервисы с гибкой и хорошо развитой инфраструктурой, которая позволяет легко внедрять и запускать новые микросервисы, а также есть четкое понимание того из каких частей будет состоять будущая система и как эти части будут между собой взаимодействовать. Необходимо быть готовым к тому, что полный переход с одной архитектуры на другую может потребовать большого количества времени (до 1 года) и большого количества финансовых вложений. Не рекомендуется использовать данный подход на ранних стадиях жизни проекта.

Данные рекомендации дают общее представление о возможной применимости того или иного архитектурного решения, так как помимо характеристик, рассмотренных выше, имеется еще большое множество вещей, на которые необходимо обратить внимание при выборе решения. Более подробные варианты анализа и применимости архитектурного подхода можно увидеть в работе Л. Бааса [3].

Заключение

Выбор архитектуры при проектировании веб-приложения является важным этапом в его разработке. И как было описано в статье, у каждого из подходов к организации работы сервера свои плюсы и минусы и выбор той или иной архитектуры во многом зависит от задач, которые будут решать будущее веб-приложение.

Библиографический список

1. Где живут объявления? [Электронный ресурс]. – Режим доступа – URL: https://habr.com/company/avito/blog/321796/ (Дата обращения 27.11.2018)

2. Масштабируя до 100 миллионов: архитектура, определяемая уровнем сервиса [Электронный ресурс] – Режим доступа – URL: https://habr.com/company/wix/blog/282045/ (Дата обращения 24.11.2018)
3. Лен Баас и др. Архитектура программного обеспечения на практике // Лен Баас, Пол Клементис, Рик Кацман – М:Питер, 2006 – 576 с.: ил.
4. Микросервисы делают мир проще (а вот и нет) [Электронный ресурс]. – Режим доступа – URL: https://habr.com/company/raiffeisenbank/blog/427953/ (Дата обращения 27.11.2018)
5. Микросервисы: размер имеет значение, даже если у вас Kubernetes [Электронный ресурс]. – Режим доступа – URL: https://habr.com/company/flant/blog/424531/ (Дата обращения 24.11.2018)
6. Паттерн монолитной архитектуры [Электронный ресурс]. – Режим доступа – URL: https://microservices.io/patterns/monolithic.html (Дата обращения 247.11.2018)
7. Путь от монолита на PHP к микросервисам на Scala [Электронный ресурс]. – Режим доступа – URL: https://www.youtube.com/watch?v=rFM-VxI7Cio (Дата обращения 24.11.2018)
8. Прощайте микросервисы: от ста проблемных детей до одной суперзвезды [Электронный ресурс]. – Режим доступа – URL: https://habr.com/post/416819/ (Дата обращения 20.11.2018)
9. Смерть микросервисному безумию в 2018 году [Электронный ресурс]. – Режим доступа – URL: https://habr.com/company/flant/blog/347518/ (Дата обращения 20.11.2018)
10. Создание микросервисов // Сэм Ньюмен. – М:Питер, 2016 – 304 с.: ил.
11. Microservices Trade-Off - [Электронный ресурс]. – Режим доступа – URL: https://martinfowler.com/articles/microservice-trade-offs.html (Дата обращения 20.11.2018)
12. Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow [Электронный ресурс]. – Режим доступа – URL: https://eng.uber.com/soa/ (Дата обращения 24.11.2018)

Оригинальность 97%