МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«Тольяттинский государственный университет»
Институт математики, физики и информационных технологий
(наименование института полностью)

Кафедра «Прикладная математика и информатика»
(наименование)

09.04.03 Прикладная информатика
(код и наименование направления подготовки)

Информационные системы и технологии корпоративного управления
(направленность (профиль))

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА (МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ)

на тему Архитектурное решение при построении компонентной модели пользовательской части корпоративных систем

Студент М.А. Логинова
(И.О. Фамилия) (личная подпись)

Руководитель к.т.н., доцент, Э.В. Егорова
(ученая степень, звание, И.О. Фамилия)

Тольятти 2020

Содержание

Введение 3
1 Анализ существующих архитектурных решений 8
1.1 Анализ научных работ, связанных с монолитным подходом 8
1.2 Анализ научных работ, связанных с модульным подходом 11
1.3 Анализ научных работа, связанных с микросервисным подходом 16
2 Формулирование компонентной модели пользовательского интерфейса системы 24
2.1 Описание обнаруженных проблем и формулирование требований 24
2.2 Суть компонентной модели пользовательского интерфейса системы, её терминология 26
2.3 Теоретическое описание компонентной модели пользовательского интерфейса системы 30
2.4 Взаимодействие микросервисов и компонентов 35
3 Реализация компонентной модели пользовательского интерфейса системы 37
3.1 Описание технологий реализации модели 37
3.2 Процесс реализации компонентной модели 38
4 Доказательство эффективности компонентной модели пользовательской части системы 46
4.1 Экономические показатели эффективности работы разработчиков 46
4.2 Доказательство эффективности компонентной модели 50
4.3 Тестирование компонентной модели 54
Заключение 60
Список используемой литературы 66

Введение

Первоначально все системы строились без каких-либо архитектурных принципов, множество строк следовали друг за другом, без наследований, классовых зависимостей и других возможностей языков программирования. С течением времени, они стремительно эволюционировали, стали увеличивать функциональность, расширяться. Схемы базы данных, интерфейсы, структура, архитектура программного кода и технологии разработки, реализующиеся и модернизирующиеся с их помощью, сильно изменились, системы стали распределёнными. Таким образом, без грамотно подобранной архитектуры уже не получается обходиться.

Архитектурой системы называют логическую структуру, описывающую отдельные компоненты, их свойства и связи как единую систему. Грамотно выбранная архитектура позволит реализовывать и поддерживать системы более простым и эффективным способом, их станет легче расширять и изменять, а также тестировать, отлаживать и понимать разработчикам. Архитектурный подход будет осуществлять выбор структурных элементов, их интерфейсы, а также взаимодействие между собой.

Существующие архитектурные решения хороши в применении, так, например, монолитный подход требует наименьших затрат на начальных этапах; модульный подход дает возможность реализовывать более сложные системы или приложения, которые состоят из множества различных модулей; микросервисный подход позволит создавать отказоустойчивые системы, которые легко можно расширять, не ориентируясь на конкретную платформу. Однако **научная проблема** заключается в том, что архитектурные решения дублирует некоторую функциональность, что приводит к повторному написанию кода, то есть потери времени и нерациональному использованию ресурсов команды, а следовательно, и компании.

Целью исследования является разработка компонентной модели, направленной на сокращение количества затраченного на разработку времени и эффективное использование ресурсов команды разработчиков.

Объектом исследования является архитектурные подходы реализации пользовательского интерфейса системы.

Предметом исследования являются моделирование пользовательского интерфейса корпоративной системы на основе архитектурных решений.

Гипотеза: сформулированная компонентная модель пользовательского интерфейса системы будет более эффективной, если

1) будут рассмотрены существующие модели пользовательского интерфейса системы;
2) будут определены положительные стороны существующих архитектурных решений, а также найдены их недостатки;
3) во время формулирования модели будут учтены ранее найденные достоинства и недостатки существующих решений;
4) будут описаны и рассмотрены новые термины и определения модели;
5) будут выбраны технологии, с помощью которых она будет реализована;
6) она будет реализована.

Для достижения поставленной цели были сформулированы и решены следующие **задачи**:

– изучение научной литературы, связанной с архитектурными подходами и технологиями их реализации;
– выявление достоинств и недостатков существующих архитектурных подходов;
– выявление достоинств и недостатков существующих технологий реализации;
– формулирование принципов компонентной модели пользовательского интерфейса системы;

– описание новых терминов и определений компонентной модели пользовательского интерфейса системы;

– построение и реализация компонентой модели;

– вычисление показателей, отвечающих за эффективное использование ресурсов команды;

– проведение функционального тестирования, показывающее корректность модели.

Научная новизна состоит в том, что будет реализована новая архитектурная модель пользовательского интерфейса системы.

Теоретические основы исследования включают использование трудов (работ) зарубежных и отечественных авторов по исследованию архитектурных решений.

Теоретическая значимость заключается в определении концептуального подхода нового архитектурного решения.

Практическая значимость состоит в том, что будет реализовано архитектурное решение, используемое при построении пользовательского интерфейса системы в проектах ИТ-компаний.

Достоверность и обоснованность научных положений и выводов, которые были сделаны в магистерской диссертации, следуют из адекватности архитектурной модели, сформулированной и реализованной в рамках исследования, подтверждаются экономическими показателями и результатами функционального тестирования.

Основные положения, выносимые на защиту:

1. Компонентная модель пользовательского интерфейса корпоративной системы, направленная на исключение дублирования ранее написанного функционала при эффективном использовании ресурсов команды разработчиков.

2. Результаты функционального тестирования реализованной компонентной модели пользовательского интерфейса системы, показывающие эффективность и корректность ее реализации.

Диссертация состоит из введения, четырех разделов, заключения и списка литературы. Объем диссертации составляет 72 страницы и содержит 32 рисунка, список литературы включает 62 наименований источников зарубежных и отечественных авторов.

Во введении обосновывается актуальность выбранной темы исследования, определяется объект, предмет и цель исследования, выдвигается гипотеза и формулируются задачи работы, рассматриваются научная новизна, практическая и теоретическая значимость результатов данного исследования.

В первой части диссертации описываются архитектурные решения, достоинства и недостатки, обнаруженные различными авторами научных работ, а также описывается найденная проблема, которая не рассматривалась авторами.

Во второй части формулируется компонентная модель пользовательского интерфейса системы, которая позволит решить обнаруженную в первой части проблему. Также описываются различные термины данной модели.

В третьей части описанная модель реализовываться с помощью выбранных технологий, приводятся скриншоты интерфейсов, с внедренными компонентами.

В четвертой части рассматривается доказательство эффективности реализованной модели пользовательского интерфейса с помощью различных показателей. Также проводится функциональное тестирование реализованной модели, которое доказывает корректность реализованной модели.

В заключении представлены основные результаты поставленных задач и сделаны следующие выводы:

1. В процессе изучения научной литературы была найдена проблема, которая не рассматривалась авторами в своих работах;

2. Обнаруженная проблема затрагивает актуальную тему, так как в рамках неё может быть решены насущные проблемы;

3. При формулировании принципов модели пользовательского интерфейса системы были учтены выявленные достоинства и недостатки существующих архитектурных решений;

4. Компонентная модель эффективна согласно экономическим показателям.

5. Реализованная модель работает корректно, что доказано с помощью функционального тестирования.

6. Используя данную модель, можно эффективнее использовать ресурсы команды.

Основные результаты исследования представлены в следующих публикациях:

1. Логинова М.А. Исследование и анализ архитектурных решений при построении пользовательской части распределенных корпоративных систем // сборник V международной научно-практической конференции (школы-семинара) молодых ученых «Прикладная математика и информатика: современные исследования в области естественных и технических наук».

2. Логинова М.А. Формирование новой модели пользовательской части корпоративного приложения // сборник VI международной научно-практической конференции (школы-семинара) молодых ученых «Прикладная математика и информатика: современные исследования в области естественных и технических наук».

1 Анализ существующих архитектурных решений

Анализ архитектурных подходов пользовательской части системы является актуальной темой, и рассматривается многими как в рамках диссертаций, так и в рамках научных статей.

Грамотно выбранная архитектура системы позволяет разработать и поддерживать её проще и эффективнее. В такой системе легко увеличивать функциональность, её проще изменять, тестировать, отлаживать и понимать. От хорошо продуманной архитектуры зависит, выживет ли она на рынке или нет, будет ли конкурентоспособной. Поэтому необходимо внимательно и подробно изучить процесс создания её архитектуры, уделяя внимание решаемым задачам и используемым критериям.

Самыми распространенными подходами к разработке системы являются монолитный, модульный и микросервисный, эволюционирующий из SOA - сервис-ориентированного подхода.

1.1 Анализ научных работ, связанных с монолитным подходом

Богданенько Д. А. в статье «Подходы к архитектурному проектированию веб-приложений» отмечает, что монолитный подход является самой ранней моделью проектирования приложения, у него отсутствует сложная структура приложения, база данных хранит данные необходимые серверу для работы, а он содержит всю бизнес-логику [7].

Автор, рассматривая данный подход, заметил, что такие приложения не характеризуются большой трудоемкостью в разработке и её значительной стоимостью на начальном этапе. Перечень требуемого функционала не является слишком большим, а ошибки достаточно просто исправить на начальных этапах. Новый функционал можно добавить достаточно быстро и легко. Но с течением времени возможен рост совершения ошибки, то есть сопровождать подобное приложение на протяжении долгого времени дорогостояще. Также автор обращает внимание на проблему способности

подобных приложений справляться с увеличением нагрузки, так как все составляющие системы расположены в одной точке. Следовательно, мощность самой точки должна быть соответствующей для поддержки базы данных и сервера в рабочем состоянии [7].

Зяблов Д. В. рассматривая данный подход в статье «Применение микросервисной архитектуры при разработке корпоративных веб-приложений» обращает внимание на следующие причины, из-за которых приложения, написанные с помощью монолитного подхода, становятся неудобными в разработке: трудность в тестировании, большие задержки при вводе в использовании, увеличивающиеся запросы к современным веб-приложениям. К ним относятся способность взаимодействия с другими веб-приложениями через различные веб-службы, предоставления программного интерфейса, обработка большого количество запросов, предоставление требуемой скорости доступа к данным, обеспечение высокой информационной безопасности, исключение и уменьшение вероятности потери корпоративной информации. Также он, как и предыдущий автор, обращает внимание на проблему с масштабируемостью.

Автор отмечает, что корпоративные веб-приложения быстро развиваются, они становятся распределенными и могут предоставлять определенную функциональность. Также они могут быть использованы в составе другого приложения. Это может привести к тому, что корпоративные веб-приложения, реализованные с применением монолитного решения, являются небезопасными из-за участия в процессах множества систем, они испытывают сложность с реализацией асинхронной связи между приложениями и потребность в сложных механизмах управления транзакциями при взаимодействиях между логически разделенными системами и их уровнями [13].

Основными недостатками монолитной архитектуры, отмеченные Зябловым, являются:

– сложность или невозможность изменения технологического стека веб-приложения во время разработки;

– необходимость полного обновления системы, если изменяешь даже незначительные детали приложения;

– отсутствие возможности быстрого реагирования на изменения требований к бизнес-логике приложения;

– недоступность функционала для пользователя при аварийном завершении работы;

– сложность масштабирования.

На рисунке 1.1 указана структурная схема монолитной архитектуры, указанная Зябловым Д. В. в статье [13].

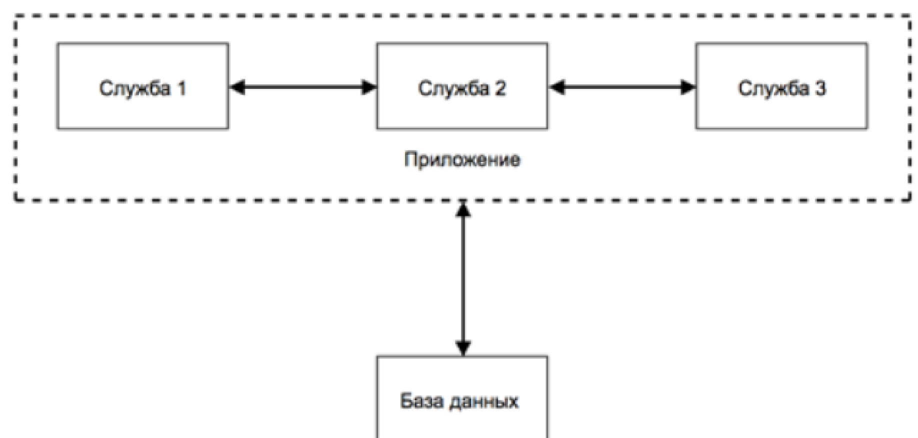

Рисунок 1.1 – Структурная схема монолитной архитектуры

В рамках статьи «Разработка микросервиса интеграции системы самообслуживания абонентов сотовой связи и центра нотификаций в рамках перехода от монолитной архитектуры приложения к микросервисной» Осипова Н.Д. выделяет следующие существенные проблемы монолитного подхода – недостаток ресурсов для растущих сервисов, сложность поддержки написанного функционала, полная пересборка и развертывание новой версии серверной части приложения при любых изменениях. Автор описывает данный подход как трехзвенный. Он представляет собой следующее: база данных по запросу сервера возвращает или обновляет

данные, сервер приложений обрабатывает серверную логику и http-запросы, а затем заполненные им html-страницы отправляются клиенту [31]. Так как монолитное приложения является большим связанным модулем, в котором все компоненты спроектированы для работы друг с другом, то у компонентов общие ресурсы, общая память, и основная сложность заключается в правильной организации доступа каждого из компонентов к памяти и ресурсам.

Таким образом, несмотря на то, что монолитное решение обеспечивает хорошее и быстрое начало, требует наименьших расходов, позволяет начинать с наименьшими затратами на разработки, позволяет допускать ошибки на ранних этапах, так как можно все быстро исправить. Стоимость поддержки систем, реализованных данным подходом, увеличивается достаточно быстро, другими словами, увеличение стоимости обслуживания у данного подхода выше, чем у аналогов.

Данный подход трудно использовать, так как любое изменение вынуждает пересобирать серверную часть и развертывать новый экземпляр продукта, при нарушении работоспособности одной части системы возникают проблемы во всех её связанных частях. Однако, некоторые компании не могут позволить себе для малейшего изменения останавливать всю инфраструктуру. Также данный подход имеет ряд других существенных недостатков, которые авторы описали в своих статьях.

1.2 Анализ научных работ, связанных с модульным подходом

Следующим архитектурным решением, рассматриваемым авторами статей и диссертаций, а в частность Богданенько Д. А., является модульный подход, который по сравнению с монолитом делит всю функциональность системы на отдельные модули, отвечающие за конкретный функционал системы [7]. Другими словами, модульной архитектурой называют архитектуру с множеством монолитных модулей внутри одной системы.

Как отмечает автор, каждый модуль системы является функционально независимым от другого, следовательно, его использование в различных частях кода не будет вызывать проблем с работоспособностью, что позволяет упростить улучшение кода и перенос границ модуля. Все модули взаимодействуют друг с другом в синхронном порядке, так как находятся на одной аппаратной части.

Также автор обращает внимание на то, что разработка системы на основе данного подхода на начальных этапах уже выше по стоимости, чем разработка монолитного аналога, так как при внесении нового функционала необходимо решить, какой именно модуль должен отвечать за данный функционал. Но увеличение стоимости поддержки модульных систем не такой резкий, что позволяет поддерживать их достаточно долго. На рисунке 1.2 отображена структурная схема модульного архитектуры, предоставленная Богданенько [7].

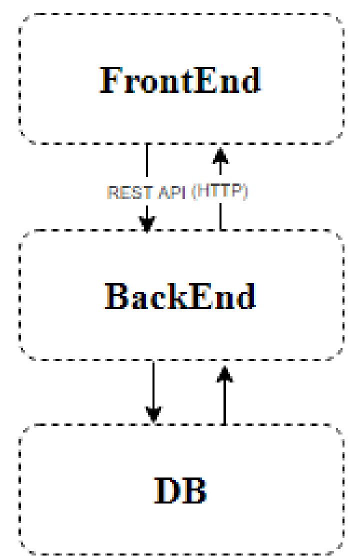

Рисунок 1.2 – Структурная схема модульной архитектуры

Автор считает, что данный подход позволяет реализовывать более сложные системы, которые состоят из множества различных модулей. Развитие данного подхода привело к вынесению некоторых модулей на отдельные аппаратные части, что спровоцировало появление целых

отдельных сервисов, и, как следствие, привело к возникновению нового подхода – сервис-ориентированному.

М. Камарузман в статье «Действительно ли модульная архитектура программного обеспечения мертва?» рассматривает модульную систему, как систему, которая может состоять из слоев, которые затем разделяются на слабо связанные модули [50]. На рисунке 1.3 изображен предоставленный автором пример модульной архитектуры в большом сложном веб-приложении.

В отображенном примере каждый уровень разбивается на слабосвязанные модули, которые в зависимости от языка внутренне связаны посредством вызовов методов или вызовов функций.

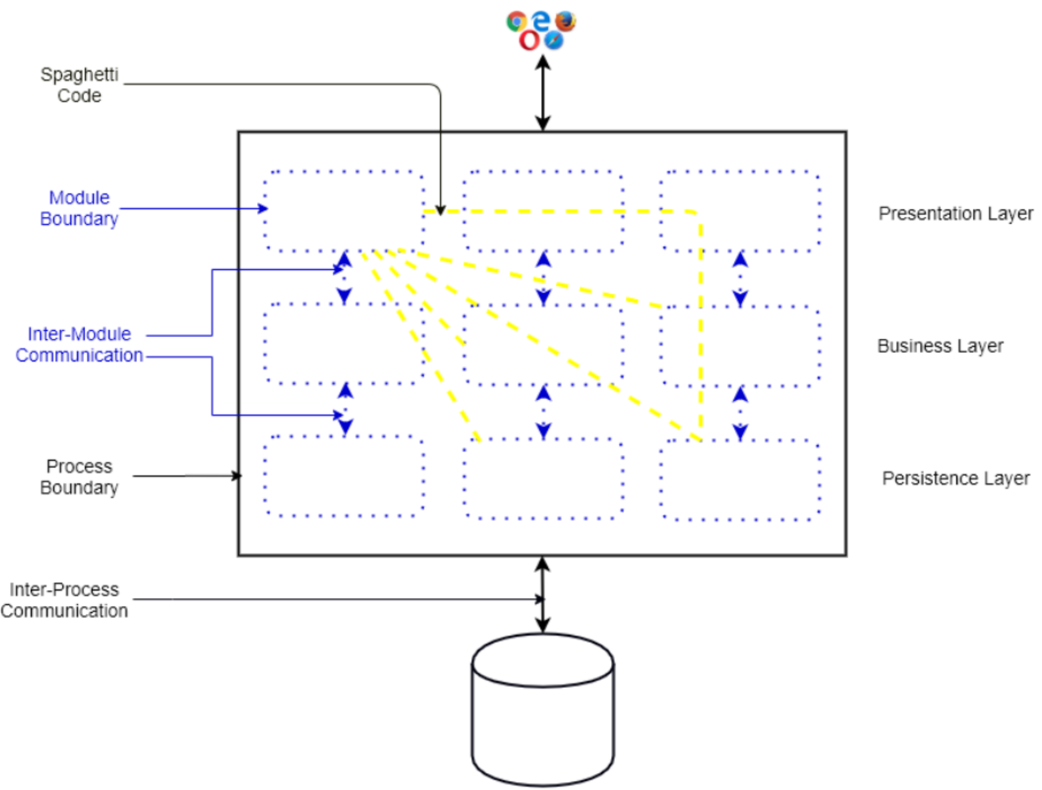

Рисунок 1.3 – Модульная архитектура веб-приложения

В своей работе автор выделил следующие характеристики модульного подхода:

– система развернута как единое целое;

– модульная граница является внутренней и может быть легко пересечена;

– система работает как единый процесс;

– нет строго владения данными между модулями.

Достоинствами данного решения автор считает упрощенную разработку, управление транзакциями и обмен данными, возможность развертывания и тестирования подобных систем в отличие от монолита, низкую эксплуатационную сложность, меньшая уязвимость при атаках [50].

Также автор выделил и недостатки данного решения, например, такие как:

– трудность распараллеливания работ между несколькими командами из-за общей кодовой базы;

– масштабирование разработки желает оставлять лучшего;

– скорость развития и улучшения подобных систем низкая;

– невозможность детального масштабирования, то есть масштабирования определенной части приложения;

– мало нововведений за последнее время;

– для крупных приложений модульный подход не подходит.

Однако, автор утверждает, что данное решение не следует списывать со счетов, потому что существует множество статей, в которых компании описывают, что их попытки внедрить микросервисную архитектуру потерпели неудачи, и в результате они перешли на модульную альтернативу.

М. Камарузман размышляет, что хотя и существует большое количество крупных корпораций и компаний, занимающиеся веб-масштабированием и использующие микросервисное решение, есть также и другие компании, где модульный подход является более оптимальным [50]. Он считает, что одной из самых сильных сторон данного решения является его надежность, проверенная временем, в отличие от микросервисов, где различные факторы, могут негативно повлиять, потому что если микросервисы не разрабатывать тщательно, то они могут быстро стать

распределенным монолитом со всеми недостатками монолита и всеми сложностями микросервисов.

А. Даффи в своей статье «Поддержка дизайна для «повторного использования» с модульным решением» говорит, что данный подход является естественным продолжением принципов структурирования архитектуры. Он включает в себя создание вариантов артефактов на основе конфигурации определенного набора модулей. Модули обычно представляют из себя функционально или структурно независимые компоненты сгруппированные так, чтобы взаимодействия были локализованы внутри каждого модуля, а взаимодействия между модулями сведены к минимуму [47].

Он считает, что подход направлен на создание разнообразия, уменьшение сложностей и максимизацию сходства в проектах. Однако, из-за того, что отдельные функции модулей или структуры должны в конечном итоге объединяться для реализации общей функции или структуры артефакта, модули никогда не могут быть по-настоящему независимыми и должны быть определены вместе с системой, к которой они принадлежат. То есть, различные ограничения модулей должны быть учтены, они должны быть успешно настроены, корректно работать, для удовлетворения общих системных требований.

Даффи выделил следующие преимущества данного подхода:

– эффективное обновление, потому что в данном подходе ограничивается переход в более мелкие области разработки системы, и тем самым обновления общей функциональности системы могут быть достигнуты без полной переделки всей системы;

– улучшенное понимание работы системы, так как необходимо понимать, как и почему отдельные модули взаимодействуют в общей системе, таким образом данный подход повышает понимание посредством накопления знаний по проектированию на основе решений, прошлого опыта и документирования рабочих процессов;

– быстрая разработка системы облегчается благодаря возможности быстрой перенастройке существующих модулей и внедрения новых модулей;

– данный подход может быть использован для поддержки быстрой адаптации на развивающихся рынках [47].

Таким образом, для реализации систем модульным решением необходимы большие расходы на начальной стадии, однако увеличение цены поддержки подобных систем менее резок. Данный факт обеспечивает поддержку подобных систем дольше и проще, чем монолита. Однако некоторые недостатки монолитного подхода еще не решены, такие как трудность распараллеливания работ между несколькими командами, масштабирование разработки желает оставлять лучшего, скорость развития и улучшения подобных систем низкая.

1.3 Анализ научных работа, связанных с микросервисным подходом

Другим часто рассматриваемым архитектурным решением является микросервисный подход. Так, например, в статье «Проектирование микросервисной архитектуры программного обеспечения» Шитько А. описывает, что при данном подходе к созданию приложения происходит «отказ от единой, монолитной структуры, сохраняя при этом модульность и независимость развёртывания компонентов». Он считает, что данный подход «избавляет от необходимости использовать все ограниченные контексты системы на сервере с помощью внутрипроцессных взаимодействий, использует несколько небольших приложений, каждое из которых соответствует какому-то ограниченному контексту» [41]. Причем, эти системы работают на разных серверах и взаимодействуют друг с другом по сети, используя различные протоколы.

В отличие от монолитного решения, в рамках которого существенная часть бизнес-задач имеет единую кодовую базу, данное решение

подразумевает механизм разделения общей бизнес-задачи на отдельные части, каждая из которых имеет отдельное приложение или микросервис со своей кодовой базой, причем изменения в одном сервисе не влекут за собой изменения в другом. В рамках подхода сервисы разрабатываются исключительно отдельными командами, и каждый из них решает конкретную бизнес-задачу. Кроме этого, компонент сервиса содержит все нужные методы для решения поставленной задачи только в этом компоненте.

Как считает автор, микросервисный подход является наиболее актуальным решением в настоящее время. Автор отмечает, что при микросервисном подходе структура приложения должна повторять структуру команды, в отличие от монолитного подхода, где разработчики объединяются по активностям: группы внутренней реализации, внешнего представления и проектирования базы данных. При проектировании и реализации должны придерживаться принципов, что сервисы должны функционировать при выходе из строя отдельных сервисов, каждый сервис должен иметь свою базу данных, вся бизнес-логика должна находиться в сервисах, а по каналу передачи отсылаются только данные. На рисунке 1.4 автор отображает схему экземпляров баз данных микросервисов [41].

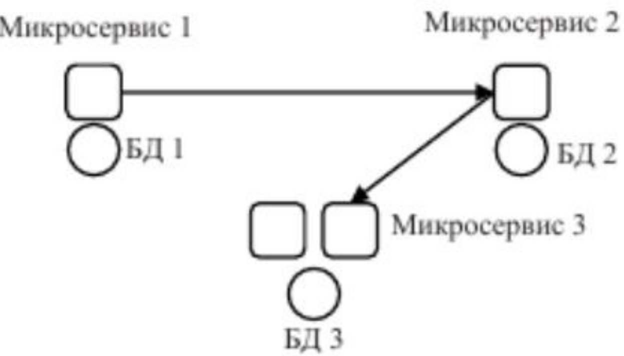

Рисунок 1.4 – Схема экземпляров баз данных микросервисов

Таким образом, можно указать следующие достоинства данного подхода, выделенные автором в статье:

– улучшенная эффективность разработки, достигающаяся разбиением команды по бизнес-задачам;

– высокая доступность, так как при отказе одного из сервисов остальные продолжают работать;

– широкий выбор технологий, то есть для определенной задачи есть возможность подобрать подходящую технологию и протестировать ее на каком-нибудь сервисе, что не повлияет на другие сервисы;

– независимое развертывание, так как простые сервисы проще разворачивать.

Но также автор выделил и недостатки данного подхода: поддержка конечной согласованности, следовательно, необходимость работать с отложенными данными, что требует постоянной доступности приложения, сложность операционной поддержки непрерывного развертывания, непрерывной интеграции и автоматического мониторинга [41].

Богданенько Д. А. в своей статье «Подходы к архитектурному проектированию веб-приложений» описывает сервис как «отдельный полностью самодостаточный модуль, который обладает собственной аппаратной базой», располагающейся на отдельном сервере, и собственной базой данных. А так как сервисы располагаются на отдельных аппаратных устройствах, то взаимодействуют они между собой асинхронно, что как считает автор, может быть, как преимуществом, так и недостатком [7].

Автор выделяет одним из главных плюсов данного подхода возможность независимого увеличения мощностей компонентов. Это предполагает увеличение мощности только того сервиса, для которого это необходимо, не влияя при этом на аппаратную часть остальных сервисов. Также им отмечается возможность реализации сервисов с помощью различных языков программирования и при этом отдельно друг от друга, так как изменения в коде одного сервиса влияют только на него.

Однако, как считает автор, разработка систем данным подходом сложна, так как возникает необходимость в ясной оговоренности границ функционала каждого сервиса. Также возникают проблемы с проверкой передаваемых данных сервису из другой части системы, так как их

разработка происходит на отдельных аппаратных частях. Отладка становится невозможной, потому что схемы взаимодействия между сервисами системы в открытом виде нет, а сервис знает только о себе. Следовательно, обработка ошибок является дополнительной проблемой данного подхода [7].

Д.В. Зяблов рассматривал целесообразность применения микросервисной архитектуры при разработке корпоративных веб-приложений, изучал причины возникновения, актуальность применения и основные особенности микросервисного подхода в статье «Применение микросервисной архитектуры при разработке корпоративных веб-приложений». Главным отличием данного решения от монолитного автор выделил использование специализированных программ или модулей для выполнения бизнес-логики корпоративного веб-приложения – микросервисов. На рисунке 1.5 представлена структурная схема микросервисной архитектуры указанная автором в данной статье [13].

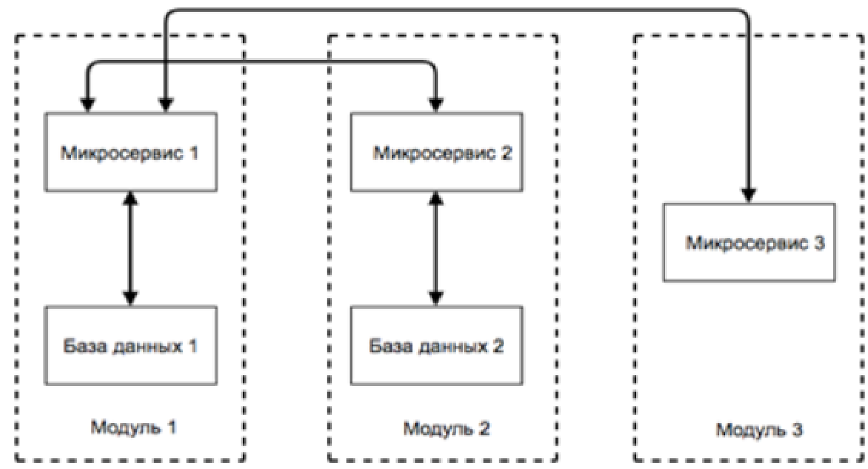

Рисунок 1.5 – Структурная схема микросервисной архитектуры

Каждый микросервис содержит конкретную функциональность и может иметь свою базу данных, он создается исходя из особенностей предметной области приложения. Как отмечает Зяблов Д. В. микросервисы являются видоизменением сервис-ориентированного решения, которое используется для формирования распределенных программных систем при разработке корпоративных веб-приложений [13].

Автор выделяет следующие преимущества данного подхода:

– доступность выбора подходящих для разработки всех сервисов различных языков программирования и программных средств;

– каждый микросервис может быть развернут независимо от других служб;

– возможность замены микросервисов друг другом;

– возникновение вокруг отдельных функций микросервисов;

– упрощение масштабирования;

– повышение гибкости;

– слабосвязанность компонентов.

Однако, как считает автор, система на базе данной архитектуры наиболее склонна к ошибкам из-за использования сети передачи данных для взаимодействия между микросервисами. Это может привести особенно при обработке пользовательских запросов и сложностям реализации общего поведения для всех микросервисов к дополнительным затратам [13].

Авторами статьи «Разработка распределенных приложений сбора и анализа данных на базе микросервисной архитектуры» Артамоновым и Востокиным выделяются следующие проблемы микросервисного подхода, на которые следует обращать внимание при построении систем на:

– учет сетевых задержек;

– сумму, требуемых вычислительных ресурсов для работы большого количества сервисов;

– общий формат использования сервисов и сообщений, а также на протоколы взаимодействия;

– обеспечение отказоустойчивости и балансировку нагрузки распределенных систем;

– сложность разворачивания и тестирования приложений [4].

Однако авторы утверждают, что минусы данного решения значительны для бизнес-приложений, но часть из них можно игнорировать при

построении приложений. На рисунке 1.6 указан пример микросервисной архитектуры, предоставленной авторами [4].

Сервис-ориентированный подход, из которого эволюционировал микросервисный подход, хорошо соотносится с теорией процессного управления в менеджменте отмечается Артамоновым И.В. в статье «Исторические аспекты появления микросервисной архитектуры», и, следовательно, является основой современных корпоративных информационных систем. Также можно реализовать сложные и иерархические системы распределённого типа, используя данные сервисы. Они не будут привязаны к закрытым протоколам и стандартам, обеспечивая хороший уровень повторного использования, асинхронности и автономности отдельных частей. Таким образом, сохраняется возможность привлечения к работе различных технологических и организационно независимых участников в разных отраслях деятельности, начиная от интеграции облачных приложений, выполнения бизнес-процессов в корпоративной среде, построения много агентных систем и заканчивая организацией совместной деятельности в научном сообществе [1].

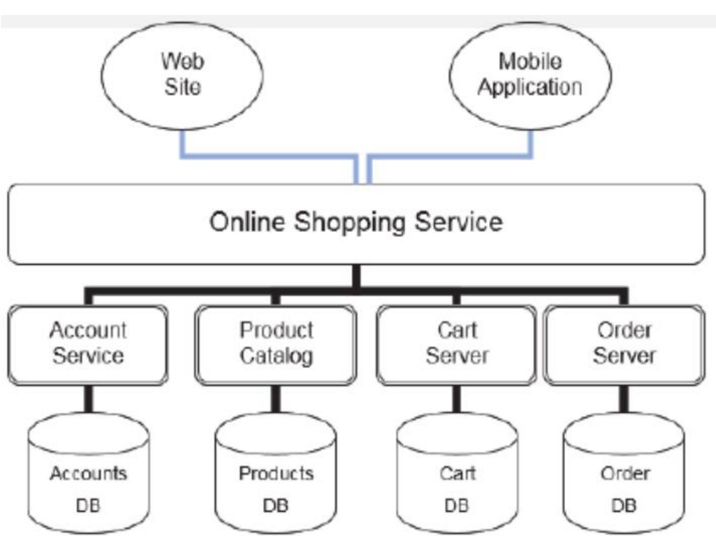

Рисунок 1.6 – Микросервисная архитектура

Сервис-ориентированный подход объединяет в себе все свойства объектных систем. Отличие заключается только в вопросах наследования, иерархии, инкапсуляции, которые в данном подходе вынесены на

организационный уровень [2]. Сервисом является завершенная, прошедшая компиляцию программа, регулирующая доступ к своим данным и функциям через конкретный интерфейс. Как отмечает автор, в отличие от компонентно-ориентированных систем они не привязаны к определенному языку программирования или конкретной технологии, они были направлены на применение в интернете, позволяя таким образом быстро разрабатывать сложные распределенные системы или облегчать объединение с существующими [1].

Артамонов рассматривает в своей статье реализацию сервис-ориентированного подхода в виде микросервисов. Микросервисом он называет самостоятельную программу, выполняющую определенный набор функций и взаимодействующую с другими приложениями через интерфейс, используя стандартные, общепринятые и легковесные протоколы. Основными принципами данного подхода, по мнению автора, являются высокий процент повторного использования кода, независимость от окружающей среды либо слабая с ней связанность.

Автор отмечает, что основное отличие микросервисного подхода от сервис-ориентированного является отсутствие ограничений в виде набора специальных протоколов. Автор затрудняется точно ответить является ли это преимуществом или недостатком [2]. Так как уход от стандартов адресации веб-служб предоставляет разнообразные возможности разработчикам, но с другой стороны наличие установленных стандартов давало возможность взаимодействия сервис-ориентированного подхода.

Н.Д. Осипова считает, что микросервисная архитектура способна решить многие проблемы, так как микросервисы можно разворачивать по частям, потому что они работают в независимом процессе, взаимодействуя с остальными при помощи различных легковесных механизмов (например, HTTP), отсутствует необходимость перезапускать все микросервисы в случае изменений в одном из них [31]. Автор говорит, что в случае выхода из строя одного из сервисов, он остановится, но остальные продолжат работу.

Таким образом, было показано, что технология микросервисов не является абсолютно новой, она стала следующим этапом развития идей построения сложных программных систем.

Использование микросервисного подхода позволит создавать отказоустойчивые приложения, которые без труда можно расширять без тесной ориентации на конкретную платформу, а также гибко настраивать сервисы и развёртывать их в различных средах. Подход уменьшает время непрерывного развертывания, добавления или исправления функционала и поставки по сравнению с монолитной архитектурой. Однако, в данном подходе возникают сложности с организацией взаимодействия сервисов между собой, что приводит к уменьшению общей производительности и пропускной способности канала. Также возникает необходимость явного обозначения границ каждого сервиса. Кроме того, возникают проблемы с отладкой передаваемых сервису данных из другой части системы, потому что они разрабатываются на отдельных аппаратных частях, а схемы взаимодействия между сервисами системы в открытом виде нет, сервис знает только о самом себе. Еще одной проблемой данного подхода является обработка ошибок.

Микросервисный подход достаточно дорогостоящий на стадии начальной разработки, но при правильном подходе позволяет уменьшить расходы на следующих этапах. Однако, данный подход больше пригоден для больших компаний, обладающих достаточным количеством ресурсов для проведения точных расчетов и планирования того как будет развиваться разработка.

Таким образом, были рассмотрены научные работы различных авторов на тему исследования. В ходе изучения статей и научных работ выяснилось, что каждый архитектурный подход, считают авторы, дает свои преимущества и недостатки. Однако, были обнаружены проблемы, которые не рассматривались авторами в своих работах.

2 Формулирование компонентной модели пользовательского интерфейса системы

2.1 Описание обнаруженных проблем и формулирование требований

Изучив научные статьи и диссертации на тему архитектурных решений, можно сделать вывод, что наиболее популярные архитектурные подходы к построению системы это монолитный, модульный и микросервисный, который эволюционировал из сервис-ориентированного подхода.

Монолитный подход разработки в данный момент является редко используемым в крупных современных системах, так как подобные системы сложно масштабируемы, их необходимо полностью обновлять при изменении незначительных деталей, написанный функционал сложно поддерживать. Также существуют и другие причины, рассматриваемые авторами [14][20].

Модульный подход построения приложения также имеет ряд проблем, например, взаимодействие между компонентами происходит лишь синхронно, трудность распараллеливания работ между несколькими командами, масштабирование разработки желает оставлять лучшего, скорость развития и улучшения подобных систем низкая.

Микросервисный подход является набирающим популярность и наиболее используемым в последнее время, так как он удовлетворяет современным требованиям, таким, например, как слабая связанность, устойчивость к сбоям, постепенность выхода в продакшн и независимое версионирование компонентов.

Принято считать, что модульный подход избавлен от некоторых недостатков монолитного подхода, так, например, подход предоставляет возможность реализовывать более сложные системы, которые состоят из множества различных модулей. Микросервисный подход развился из сервис-

ориентированного подхода, который являлся улучшенной версией модульной архитектуры.

Однако при микросервисном подходе у многих микросервисов имеются похожий функционал или UI-части, например, такие как сервис локализации, который хранит переводы, шапка, подвал, логин/логаут [37]. И каждая микросервисная команда разрабатывает самостоятельно подобный функционал. Но подобное не является рациональным, так как команда дублирует функционал, который ранее уже был написан другой командой разработчиков. Кроме того, если каждый микросервис в отдельности будет запрашивать одинаковые данные на сервере, то запросов при инициализации будет много, что приведет к «проседанию» производительности системы.

Таким образом, для решения обнаруженных проблем необходимо смоделировать модель пользовательского интерфейса системы, которая бы учитывала достоинства и недостатки существующих архитектурных решений.

При формулировании модели пользовательского интерфейса системы необходимо учитывать, что она должна:

– позволять создавать системы, которые смогут решать поставленные задачи и хорошо выполнять возложенные на них функции в разных условиях, они будут надежны, производительны, безопасны, масштабируемы, способные справляться с увеличением нагрузки;

– позволять изменять систему согласно новым требованиям и добавлять новый функционал с меньшим количеством ошибок и затрат;

– предусматривать возможность наращивания предоставляемых возможностей по мере возникновения необходимости, таким образом, чтобы на это тратилось минимальное количество сил;

– позволять добавлять новые функции и сущности, не нарушая при этом основную структуру;

– предусматривать возможность обеспечения параллельности процесса разработки, чтобы увеличить количество людей, которые работают над процессом разработки приложения;

– позволять организовывать процесс тестирования, так как код, который легко проверять содержит меньшее количество ошибок и надежнее работает;

– позволять проектировать систему так, чтобы ее отдельные фрагменты можно было повторно использовать, избегая дублирования [3][5][6][8][10][12][15][16][35].

Для удовлетворения описанных требований будет использоваться модель пользовательского интерфейса системы, называемая компонентной, которая будет описана далее.

2.2 Суть компонентной модели пользовательского интерфейса системы, её терминология

Компонентная модель пользовательского интерфейса системы предполагает изолированность UI-частей, которые используется многими микросервисами в микросервисном подходе. Причем разрабатывать изолированную UI-часть необходимо как отдельное приложение. Подобные «отдельные приложения» будут представлять собой некий фрагмент, состоящий из js+html+css+id фрагмента. Фрагменты компонентной модель будут хранится во фронтендном микросервисе.

Данная модель называется компонентной, потому что весь пользовательский интерфейс системы разделен на компоненты или фрагменты. То есть они встраиваются во фронтенд (UI, интерфейс) различных микросервисов, причем не важно с помощью каких языков написаны эти микросервисы или фрагменты. На рисунке 2.1 наглядно продемонстрирован данных факт.

Компонент или фрагмент является независимой UI-частью, он должен соблюдать набор правил разработки для того, чтобы его можно было использовать в общей системе [15][30][44]. Кроме того, стили должны быть максимально специфичны для компонента, прямого взаимодействия с другими фрагментами быть не должно [45]. Благодаря id фрагмента можно сохранить информацию обо всех встраиваемых фрагментах, а затем обращаться к нему по id.

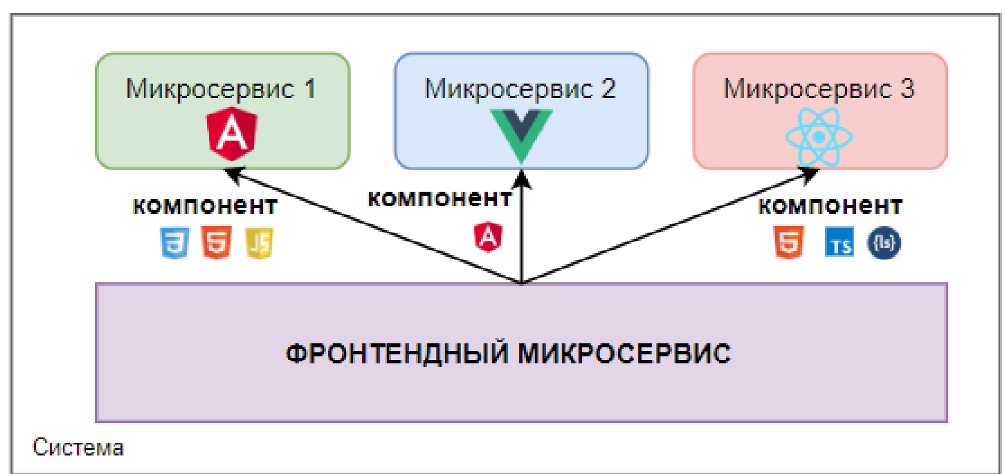

Рисунок 2.1 – Компонентная модель

Фрагменты будут отвечать за некий набор бизнес-функций, то есть они будут полностью функциональными от начала и до конца [30][40]. Фрагменты фронтендного микросервиса подобно обычным микросервисам могут быть написаны на различных языках.

Необходимо вынести во фрагменты компонентной модели те UI- части, которые используются в большинстве микросервисов. Так, например, это может быть шапка, подвал и меню. На рисунке 2.2 показано схематическое представление интерфейса микросервиса, где зеленым цветом выделены общие части, используемые различными микросервисами, а оранжевым – уникальные для конкретных микросервисов.

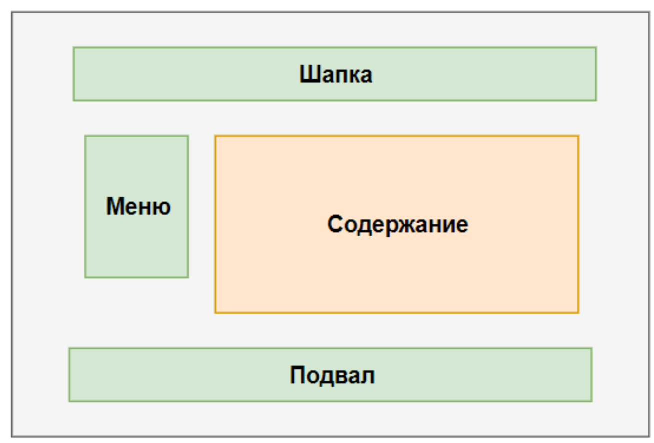

Рисунок 2.2 – Схематическое представление интерфейса микросервиса

Можно выделить следующие характеристики компонента:

– он может представлять собой как небольшую по размеру UI-часть, так и содержимое табы или страницы;

– их могут разрабатывать каждый микросервис по отдельности, какими они будут, какой функционал будут реализовывать решает команда разработчиков или их лид;

– фрагменты, разработанные одним микросервисом могут быть использованы другими микросервисами для избежание дублирования написанного функционала;

– он может как встраиваться в микросервис, так и представлять собой контейнер для встраивания других фрагментов.

На рисунке 2.3 схематично отображены различные фрагменты, так, например, один из фрагментов – это шапка, а другой – содержимое табы.

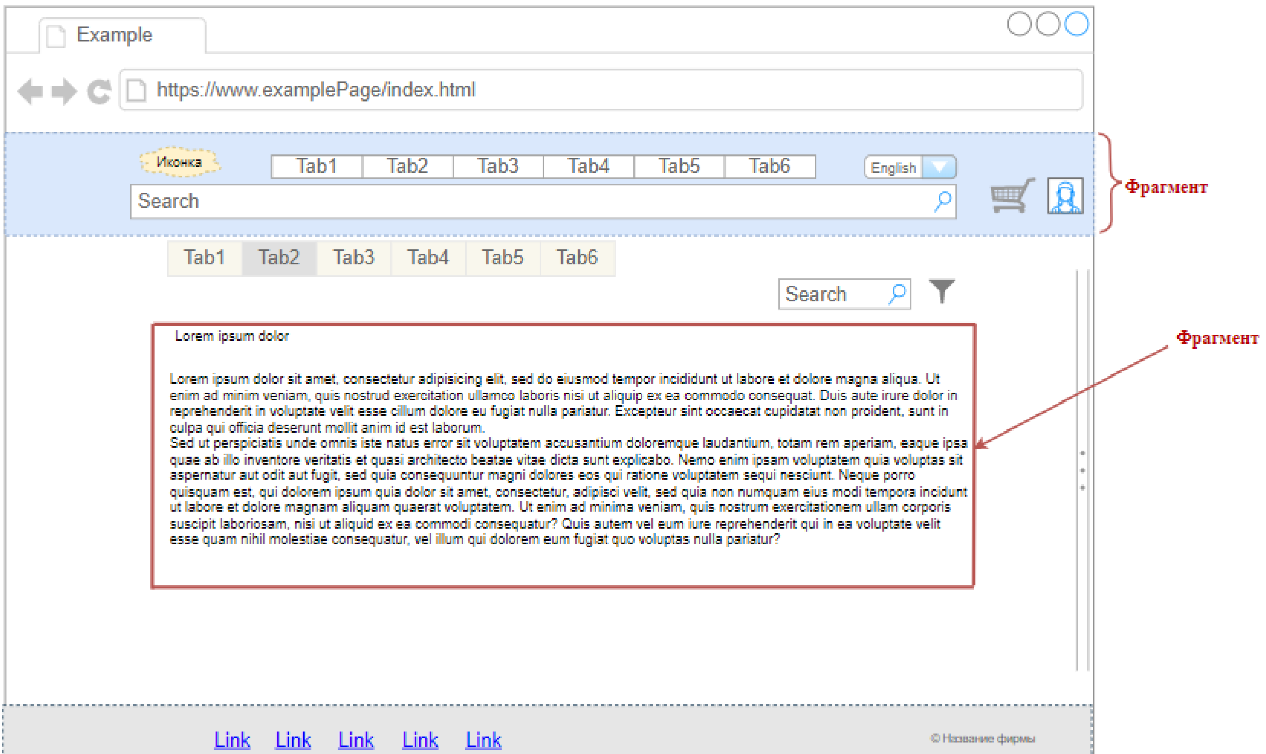

Рисунок 2.3 – Схематическое отображение фрагментов

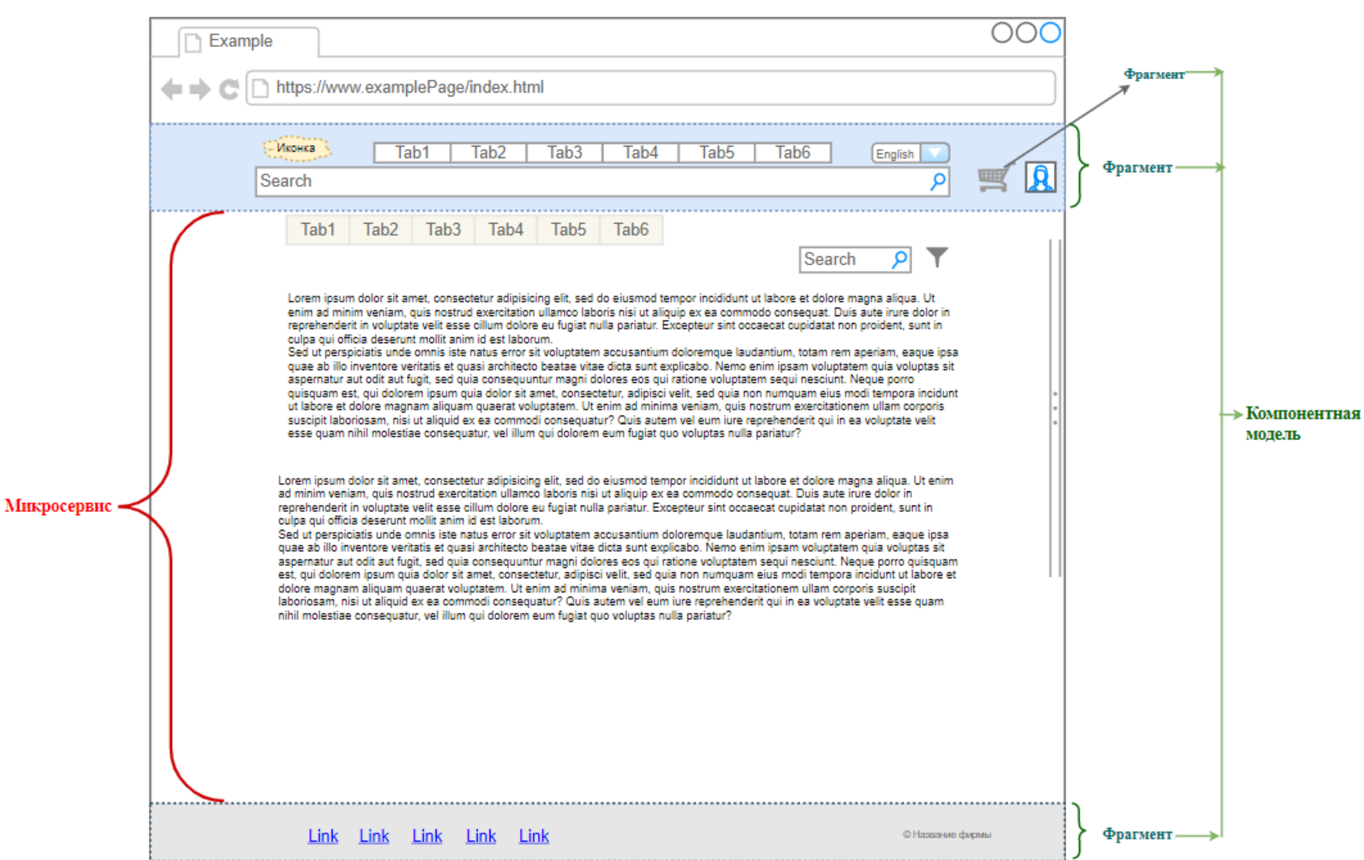

Рисунок 2.4 – Структурная схема компонентной модели

Описанная модель пользовательского интерфейса системы позволяет разместить фрагменты и микросервисы, написанные на разных языках, на одной странице, также она позволяет написать универсальный код, который позволит динамически подгружать нужные фрагменты на страницу, инициализировать их и управлять жизненным циклом. На рисунке 2.4 показана структурная схема системы, написанная с использованием описанной модели.

Для отображения фрагмента необходимо предоставить лишь DOM-элемент, в который он должен встроиться. Микросервису доступен метод, благодаря которому он сможет разместить любой фрагмент внутри своего DOM, для этого необходимо передать контейнер, где он должен отобразиться.

В рамках компонентной модели во фронтендный микросервис на ряду с фрагментами можно вынести различные сервисы, которые являются базовыми и часто используются большинством микросервисов [38][40]. В микросервисной архитектуре появления подобных сервисов неизбежно. Это могут быть сервисы локализации/интернационализации, авторизационный сервис. Если каждый микросервис в отдельности будет запрашивать эти данные на сервере, то количество запросов будет огромным.

Таким образом, преобразуя элементы системы, которые используются многими микросервисами во фрагменты компонентной модели, можно избежать дублирования функционала, так как написаны данные части будут один раз. Кроме того, можно разработать любой уровень вложенности этих фрагментов. Таким образом, данная модель позволит рациональнее использовать ресурсы команды.

2.3 Теоретическое описание компонентной модели пользовательского интерфейса системы

Для того, чтобы микросервисы могли встраивать необходимые фрагменты у себя на странице, необходимо, инициализировать в микросервисе Контроллер, который отвечает за хранение общих функций, за механизм обращения к фронтендному микросервису, в котором хранятся фрагменты, а также за механизм их встраивания в микросервисы [11][17][29]. Данную инициализацию необходимо выполнить в глобальном объекте *window*. Для этого при загрузке микросервиса, необходимо отправить запрос на инициализацию Контроллера, где в качестве параметра будут передаваться название микросервиса и конфигурационный флаг. Во время инициализации на стороне Контроллера будет формироваться конфигурация для микросервиса. Если конфигурационный флаг не будет передан, то считается что конфигурация стандартная. В дальнейшем данная конфигурация будет использоваться при формировании правильной конфигурации фрагмента.

После того как Контроллер инициализировался, микросервису будут доступны все его методы и поля посредством обращения через глобальный объект. Таким образом при необходимости встраивания на страницу фрагмента, микросервис будет использовать метод Контроллера *createFragment()*, который будет отправлять запрос за «собиранием» правильного бандла, отвечающего за встраивание фрагмента в необходимое место на стороне микросервиса. В качестве передаваемого параметра необходимо указать название фрагмента. Результатом данного метода будет js-файл, который будет встраивать фрагмент на страницу, используя возможности JavaScript.

Микросервис должен указать «маркер» в своей верстке, где необходимо встроиться фрагменту. Маркером служит некий DOM-элемент с определённым классом. На рисунке 2.5 схематически показано как будет выглядеть механизм встраивания фрагмента на страницу микросервиса.

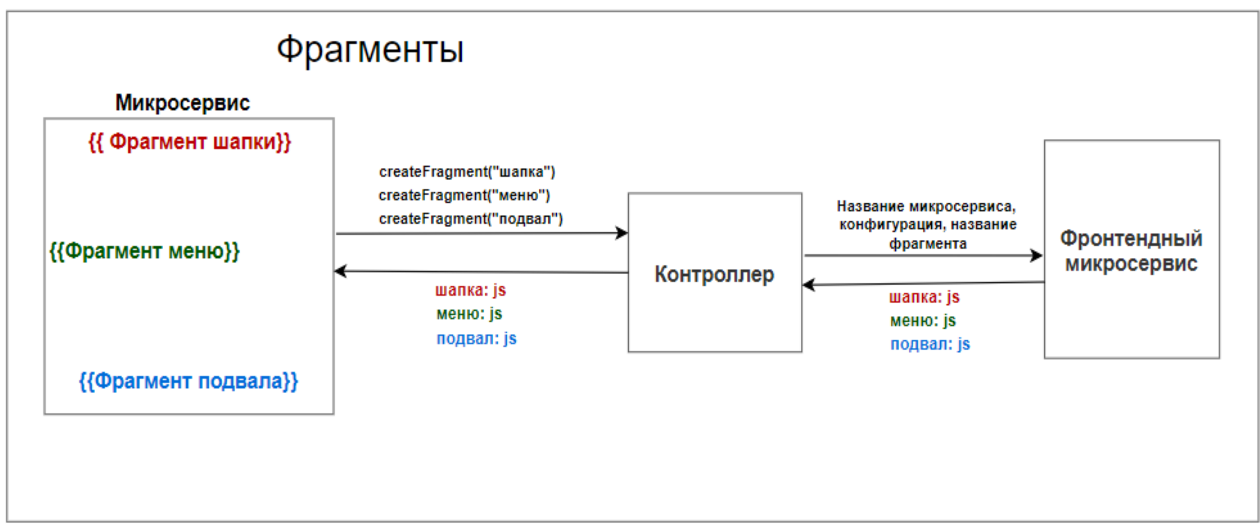

Рисунок 2.5 – Схематическое представление встраивания фрагментов

В рамках данной модели реализации пользовательского интерфейса системы возможна также и конфигурация фрагментов. Микросервис может отправлять фронтендному микросервису некие требования, относящиеся к фрагментам с помощью конфигурационного флага. Так, например, как показано на рисунке 2.6, микросервису необходимо отобразить у себя проектную шапку, при инициализации Контроллера он указывает данное требование, и в дальнейшем на стороне Контроллера формируется конфигурация фрагмента «шапки» с учётом данного параметра.

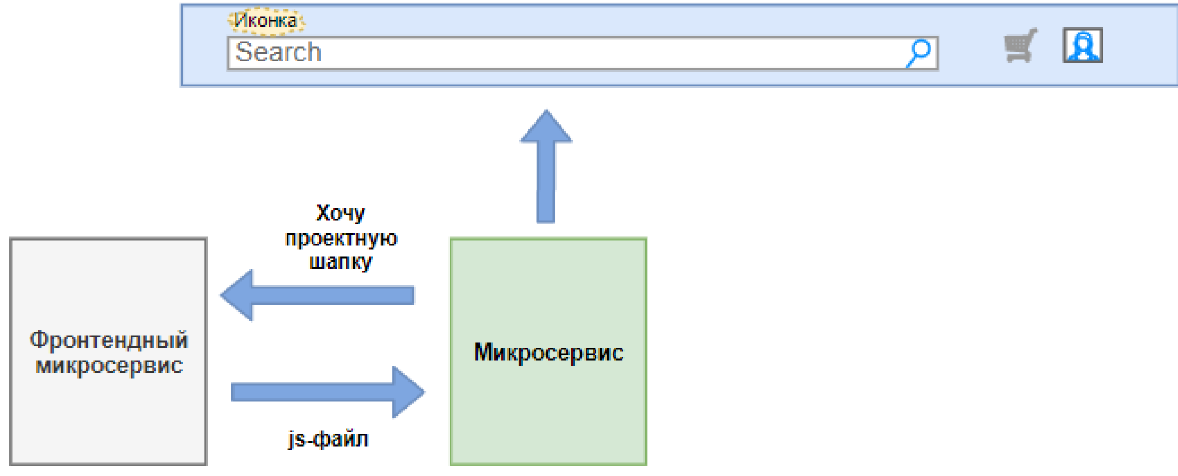

Рисунок 2.6 – Отображение проектной шапки

На рисунке 2.7 показан механизм инициализации Контроллера, а на рисунке 2.8 – процесс создания фрагмента.

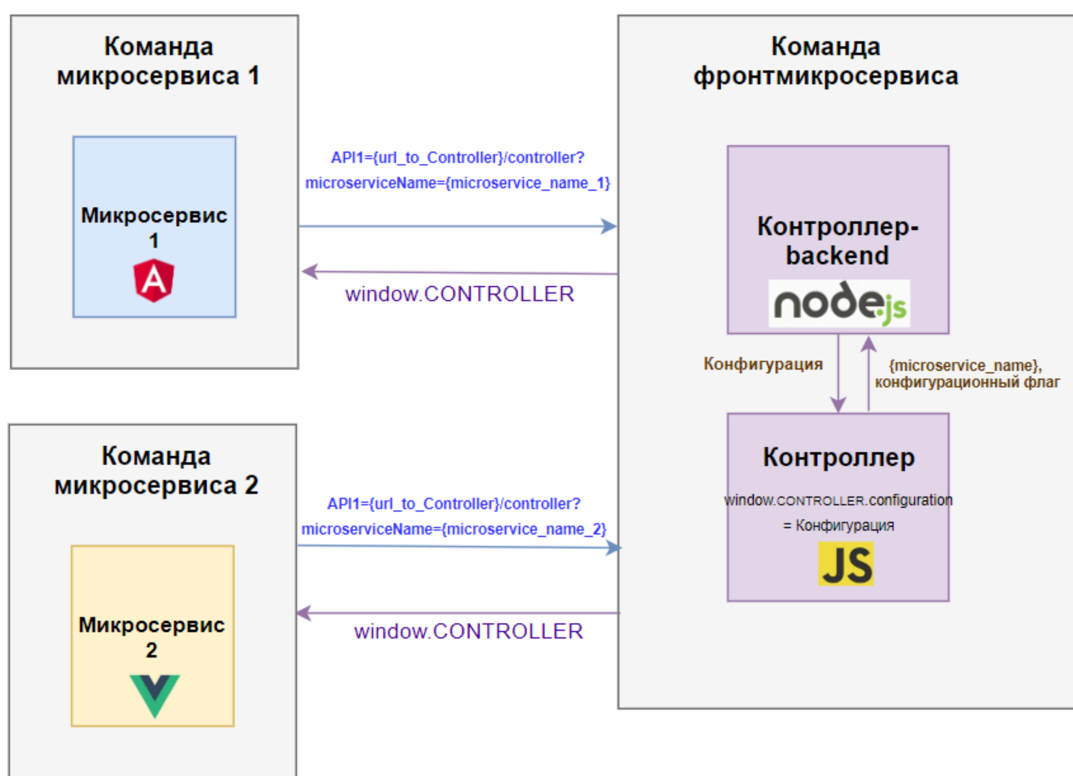

Рисунок 2.7 – Механизм инициализации контроллера

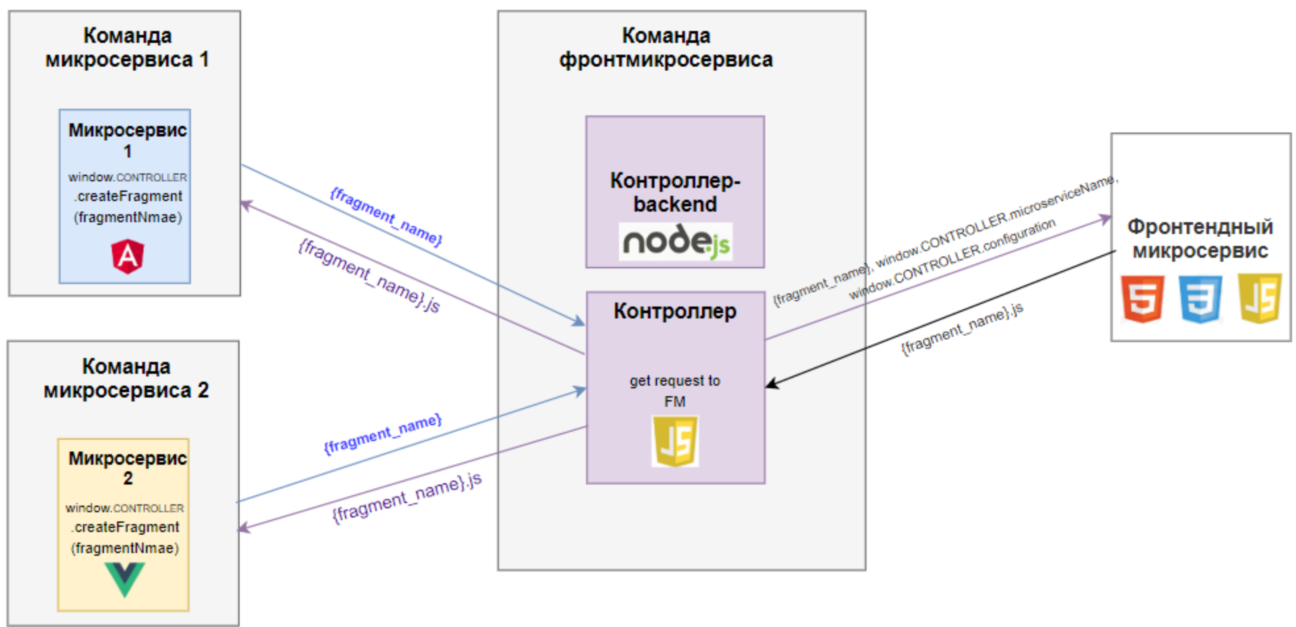

Рисунок 2.8 – Механизм создания фрагмента

Таким образом, при инициализации Контроллера будет собираться необходимая конфигурация фрагментов для конкретных микросервисов согласно указанному параметру – конфигурационному флагу, что позволит гибко регулировать конфигурацию фрагментов.

Одним из достоинств использования фрагментов является то, что при нахождении каких-то ошибок в коде или в функциональности в вынесенной UI-части, при добавлении нового функционала будет правиться только конкретный фрагмент, который затем обновится автоматически во всех микросервисах, в которые он встроен, как показано на рисунке 2.9. То есть нет необходимости заходить в каждый микросервис и править данную UI-часть, необходимо лишь подтянуть изменения с данного фрагмента.

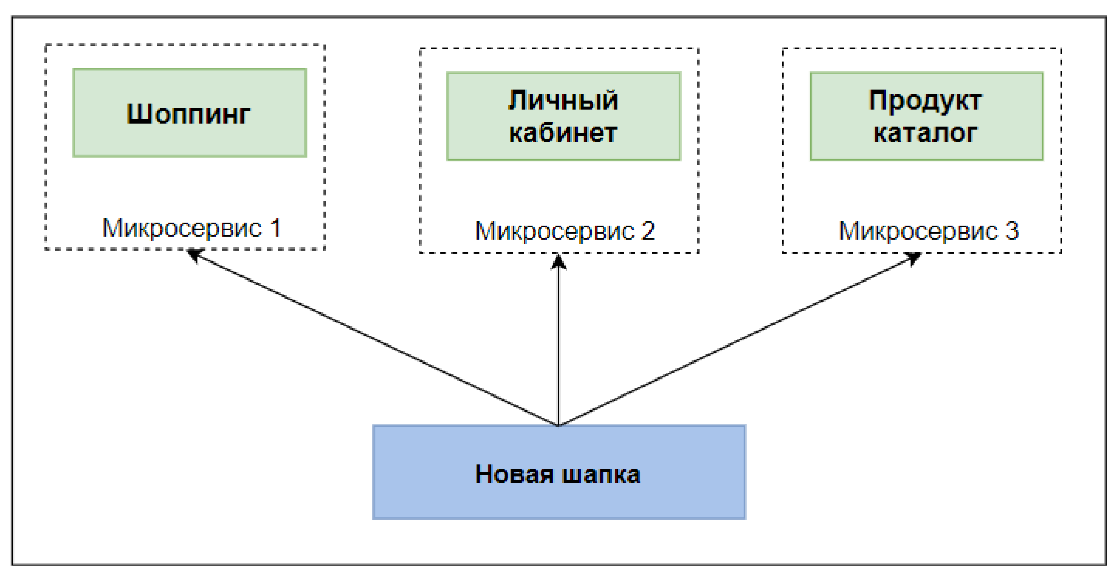

Рисунок 2.9 – Автоматическое обновление фрагментов во всех микросервисах

Таким образом, в данном пункте были рассмотрены механизмы инициализации Контроллера и встраивания фрагментов, далее будет рассматриваться взаимодействие микросервисов и фрагментов.

2.4 Взаимодействие микросервисов и компонентов

Взаимодействие между микросервисами будет происходить с помощью API, а взаимодействие между компонентами и микросервисами с использованием EventBus.

API Gateway является сервисом, собирающим бизнес-вызовы к целевым сервисам. Другими словами, клиент, обращаясь к шлюзу, который переадресовывает запрос к необходимому микросервису, минует необходимость обращаться к каждому сервису по отдельности. Также данный сервис предоставляет возможность наличия разных API для каждого клиента [19][55]. Модель API Gateway представлена на рисунке 2.10.

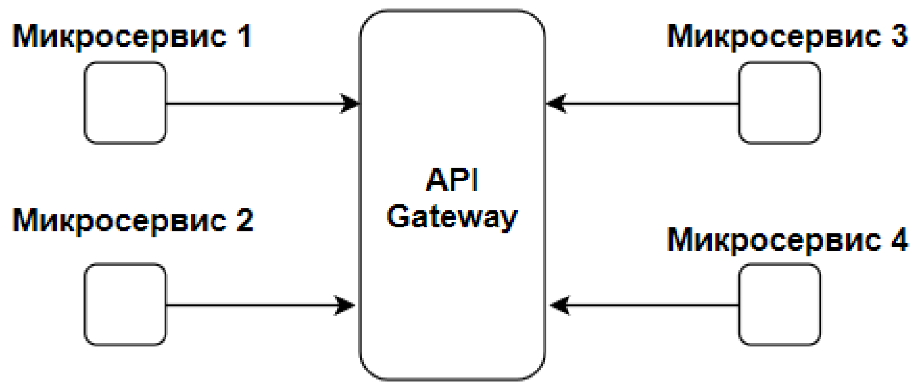

Рисунок 2.10 – Модель API Gateway

В рамках модели Event Bus («Шина событий») микросервисы подписываются на конкретные события, при возникновении которых они отреагируют на это определенным образом [30]. Данный способ необходим, чтобы компонент мог сообщить микросервису, что у него изменилось состояние, чтобы он смог среагировать на это. На одно событие может среагировать не только один микросервис.

Шина событий позволяет осуществлять обмен данными между фрагментом и микросервисами с помощью механизма публикации и подписки, при котором не происходит явного взаимодействия друг с другом как показано на рисунке 2.11.

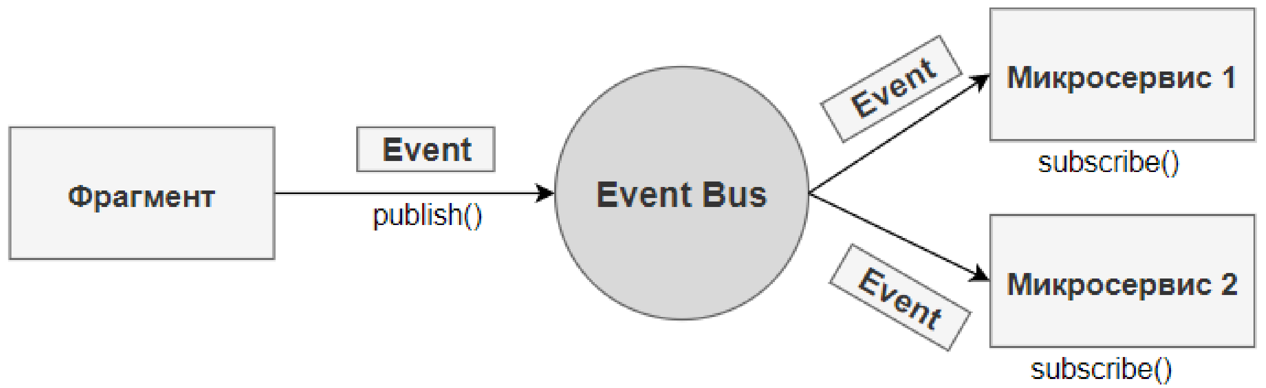

Рисунок 2.11 – Модель Event Bus

Event Bus имеет следующие методы:

– publish(eventName, data) – публикация события с данными;

– subscribe(eventName, callback) – подписка на событие;

– removeEvent(eventName, callback) – удаление обработчика события, зарегистрированного с помощью publish().

Взаимодействие фрагментов и микросервисов будет происходить следующим образом: при совершении каких-либо изменений во фрагменте, он сообщает Event Bus, что его состояние изменилось, используя метод *publish()*, если данные изменения важны для каких-то микросервисов, то они используя метод *subscribe()*, подписываются на эти изменения. Тем самым при изменении состояния фрагмента, микросервис своевременно реагирует на данные изменения необходимым ему образом.

Таким образом, были определены требования к компонентной модели пользовательского интерфейса системы, была рассмотрена суть этой модели, были описаны новые термины и определения данной модели, а также были описаны способы взаимодействия микросервисов и фрагментов.

3 Реализация компонентной модели пользовательского интерфейса системы

3.1 Описание технологий реализации модели

Фрагменты компонентной модели подобно обычным микросервисам могут быть написаны на различных языках. Кроме того, они могут использовать разные фронтенд технологии и иметь свои уникальные инструменты для программирования серверной части. Например, фрагмент шапки может использовать Angular и Go, а фрагмент подвала может быть написан на React и Java. Также у некоторых фрагментов может отсутствовать серверная часть [48]. Таким образом, языки программирования, с помощью которых будут разработаны фрагменты и микросервисы, определяются командой разработчиков.

Часто при работе с микросервисной архитектурой используется также Докер. Данное программное обеспечение автоматизирует управление приложениями и развертывание, также позволяет собрать приложение вместе с его зависимостями и окружением в контейнер. Это освободит от необходимости запускать каждый микросервис по отдельности, а позволит объединить их всех в один контейнер, который и будет запускаться [25][27][28][30]. То есть докер выступает в качестве переносного механизма контейнеров для «упаковки» приложений и зависимостей в контейнеры, легко развёртываемые [43].

Для формирования единого бандла будет использоваться Webpack. Он является сборщиком модулей или бандлером, то есть инструментом, который позволяет упаковывать, скомпилировать JavaScript модули в единый js-файл, тем самым он помогает собрать воедино все необходимые ресурсы, следит за изменениями и повторно выполняет задачи, позволяет добиться лучшей организации проекта [28][48].

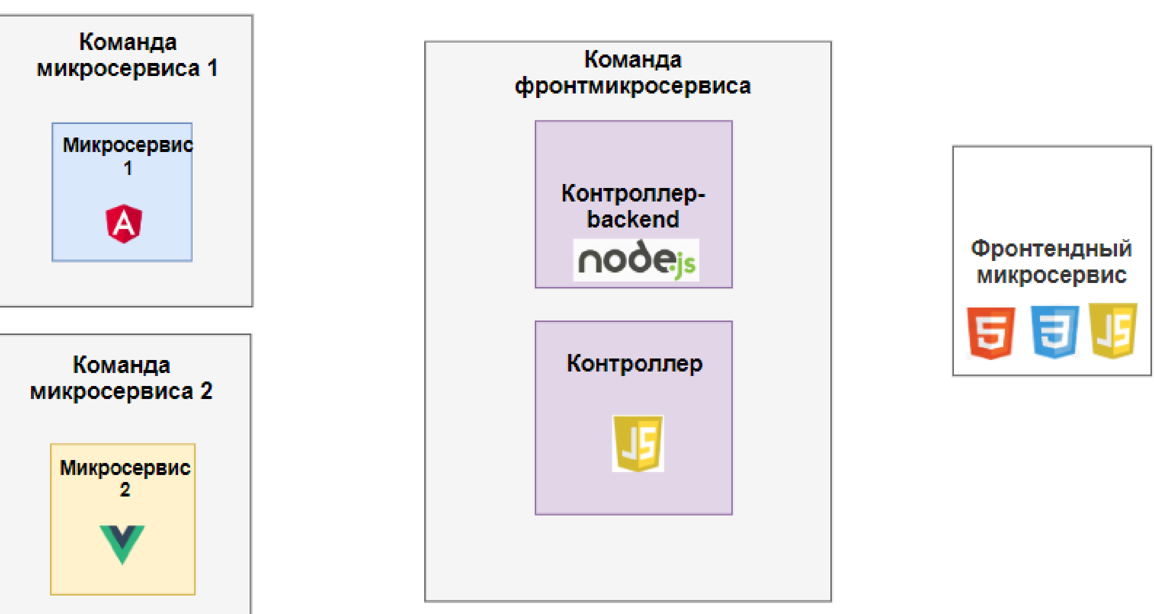

Рисунок 3.1 – Используемые технологии реализации

Согласно рисунку 3.1 в рамках данной работы микросервисы, в которые будут встраиваться фрагменты, будут написаны средствами Angular и Vue.js, Контроллер будет реализован на нативном js, фрагменты будут использовать html+ jQuery+css.

3.2 Процесс реализации компонентной модели

В рамках данной работы будут реализованы 4 проекта: Customer-cabinet и Store – микросервисы, в которые будут встраиваться фрагменты; Controller – Контроллер, отвечающий за реализацию и хранение общих функций и за механизм встраивания фрагментов в микросервис; Frontend-microservice – микросервис, в котором будут хранится все необходимые фрагменты.

Для того, чтобы был доступен механизм встраивания фрагментов на стороне микросервисов, необходимо инициализировать Контроллер, для этого при загрузке микросервисов Customer-cabinet и Store необходимо отправить запрос на инициализацию Controller, где в качестве параметра

необходимо указать, как отображено на рисунке 2.7 на примере одного из микросервисов, название микросервиса и конфигурационный флаг, если конфигурация отличается от стандартной. На рисунке 3.2 показан запрос, отвечающий за вызов инициализации Контроллера, где предполагается стандартная конфигурация, так как конфигурационный флаг не указан. А на рисунке 3.3 отображен запрос на его инициализацию с конфигурацией, отличающейся от дефолтной.

```
<script
  src="{URL_TO_CONTROLLER}/controller?microserviceName=customer-cabinet">
</script>
```

Рисунок 3.2 – Вызов инициализации Контроллера

```
<script
  src="{URL_TO_CONTROLLER}/controller?microserviceName=customer-cabinet&configFlag=test-project">
</script>
```

Рисунок 3.3 – Вызов инициализации Контроллера с нестандартной конфигурацией

Во время выполнения скриптов, указанных на рисунках 3.2, 3.3, будет происходить «предзагрузка», во время которой происходит инициализация Контроллера, формирование конфигурации для микросервиса, реализация общих методов, а также регистрация контроллера в *window*. Подобная регистрация позволит обращаться к методам Controller любым микросервисам, так как *window* является глобальным объектом, представляющим собой окно, содержащее DOM элементы и предоставляющим переменные и функции, доступные в любом месте приложения [35][36][38].

В скрипте на инициализацию Контроллера в качестве параметра обязательно необходимо передать название микросервиса, так как именно благодаря этому параметру Контроллер понимает для какого микросервиса формировать конфигурацию [49]. На рисунке 3.4 показано как на стороне Controller backend происходит обработка параметров.

```
'use strict';

const express = require('express');
const app = express();
const path = require('path');
var fs = require('fs');

app.get('/controller', function (req, resp) {
    const microserviceName = req.query.microserviceName;
    const configFlag = req.query.configFlag ? req.query.configFlag : "default";

    resolveMacros(microserviceName, configFlag).then((result) => {
        resp.send(result);
    });
});

async function resolveMacros(microserviceName, configFlag) {
    let filePath = path.join(__dirname, 'app/configurations/preload.js');
    let data = await fs.readFileSync(filePath, 'utf8');

    return data
        .replace('{MICROSERVICE_NAME}', microserviceName)
        .replace('{CONFIGURATION_FLAG}', configFlag);
}
```

Рисунок 3.4 – Обработка передаваемых параметров

Переданные в скрипте параметры в дальнейшем заменяются в файле инициализации Controller. То есть, передавая из customer-cabinet название микросервиса и конфигурационный флаг, как показано на рисунках 3.2 и 3.3, бэкенд Контроллера понимает, что необходимо формировать конкретную – стандартную или кастомную – конфигурацию для customer-cabinet в зависимости от переданного конфигурационного флага.

Таким образом, в результате отрабатывания данного скрипта Controller backend вернет js-файл, в котором отработает самовызывающаяся функция, вызывающая инициализацию Контроллера и регистрирование ее в *window*. На рисунке 3.5 отображен участок кода самовызывающейся функции.

```
(function () {
    document.body.onload = function () {
        var config = {
            microserviceName: '{MICROSERVICE_NAME}',
            configFlag: '{CONFIGURATION_FLAG}'
        };

        function init() {
            if (!controller.initialized) {
                controller.setConfig();
            }
            window.CONTROLLER = controller;
            console.info('CONTROLLER is initialized');
            let event = new Event("controllerInit", {bubbles: true});
            window.dispatchEvent(event);
        }

        var controller = {microserviceName: config.microserviceName...};
        init();
    }
})();
```

Рисунок 3.5 – Участок кода, отвечающий за инициализацию Контроллера

После того как отработает самовызывающаяся функция, отображенная на рисунке 3.5, в процессе которой произойдет регистрация Контроллера в глобальном объекте window, микросервис сможет использовать методы и поля Controller, обращаясь к ним через глобальный объект. На рисунке 3.6 показаны доступные для микросервиса поля и методы Контроллера.

```
> window.CONTROLLER
<· ▼{configuration: {…}, microserviceName: "customer-cabinet", initialized: true, setConfig: ƒ, getCo
    nfig: ƒ, …}
   ▶configuration: {microserviceName: "customer-cabinet", header: true, footer: true, menu: true, …
    microserviceName: "customer-cabinet"
    initialized: true
   ▶setConfig: ƒ ()
   ▶getConfig: ƒ ()
   ▶getMicroserviceName: ƒ ()
   ▶get: ƒ (e)
   ▶sendGetRequest: ƒ (e,t)
   ▶createFragment: ƒ (fragmentName)
   ▶generateUrlForFragment: ƒ (e,t)
   ▶publish: ƒ (e,t)
   ▶subscribe: ƒ (e,t)
   ▶eventBus: o {bus: fakeelement}
   ▶__proto__: Object
```

Рисунок 3.6 – Доступная из микросервиса информация о Контроллере

Для встраивания фрагмента на странице микросервиса необходимо вызвать метод Controller – *createFragment()*, где в качестве параметра передается название встраиваемого фрагмента. В рамках данного метода на стороне Контроллера отправляется запрос во Frontend-microservice, в котором хранятся фрагменты. Передавая в качестве параметров название фрагмента и его конфигурацию, Frontend-microservice возвращает Controller фрагмент с шаблоном и стилями, затем он возвращает Customer-cabinet бандл, в котором прописан механизм встраивания фрагмента в определенный DOM-элемент на стороне микросервиса. На рисунке 3.7 отображен вызов метода Контроллера на стороне микросервиса, а на рисунке 3.8 реализация данного метода на стороне Контроллера.

```
<script>
  window.addEventListener("controllerInit", function() {
    window.CONTROLLER.createFragment("header");
    window.CONTROLLER.createFragment("footer");
  });
</script>
```

Рисунок 3.7 – Вызов метода Контроллера на стороне микросервиса

```
createFragment: function (fragmentName) {
    return new Promise(function (resolve, reject) {
        if (window.CONTROLLER) {
            const urlFragment = window.CONTROLLER.generateUrlForFragment(fragmentName);
            window.CONTROLLER.get(urlFragment)
                .then((data) => {
                    resolve(data);
                })
                .catch((err) => {
                    reject(err);
                });
        } else {
            init();
        }
    });
}
```

Рисунок 3.8 – Участок кода, отвечающий за вызов фрагмента на стороне Контроллера

На рисунке 3.9 представлен участок кода, который отвечает за добавление фрагмента «подвал» на страницу, используя возможности JavaScript.

```
(function () {
    document.body.onload = function () {
        var footerContainer = document.getElementsByClassName("{{fragment name}}")[0];

        var footer = document.createElement("div");
        footer.setAttribute("id", "{{fragment name}}");
        footerContainer.appendChild(footer);
        $("#{{fragment name}}").load("{{URL to frontend microservice}}/footerTemplate");

        var footerCssId = '{{fragment name style}}';
        if (!document.getElementById(footerCssId)) {
            var head = document.getElementsByTagName('head')[0];
            var link = document.createElement('link');
            link.id = footerCssId;
            link.rel = 'stylesheet';
            link.type = 'text/css';
            link.href = '{{URL to frontend microservice}}/footerStyle';
            link.media = 'all';
            head.appendChild(link);
        }
    }
})();
```

Рисунок 3.9 – Реализация js-файла, добавляющего фрагмент на страницу

Для взаимодействия фрагмента и микросервиса используется EventBus, реализация которого представлена на рисунке 3.10 [42][46]. То есть, микросервис с помощью метода *subscribe()* подписывается на событие, которое публикует фрагмент с помощью метода *publish()*. Тем самым микросервис своевременно обновляет информацию, которая возможна была изменена во фрагменте.

На рисунке 3.11 можно увидеть, как будет выглядеть страница после встраивания в нее фрагмента «шапка» и фрагмента «подвал», если не будет указан конфигурационный флаг, то есть при дефолтной конфигурации. А на рисунке 3.12 отображена страница с указанным проектным конфигурационным флагом.

```
export default class EventBus {
    constructor() {}

    subscribe(event, callback) {
        this.bus.addEventListener(event, callback);
    }

    removeEvent(event, callback) {
        this.bus.removeEventListener(event, callback);
    }

    publish(event, detail = {}) {
        this.bus.dispatchEvent(new CustomEvent(event, { detail }));
    }
}
```

Рисунок 3.10 – Реализация EventBus

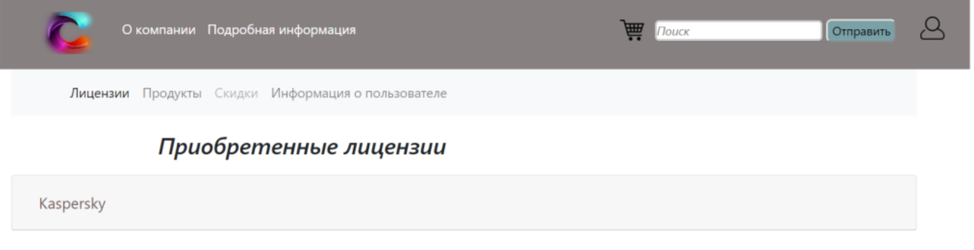

Рисунок 3.11 – Страница после встраивания фрагментов с дефолтной конфигурацией

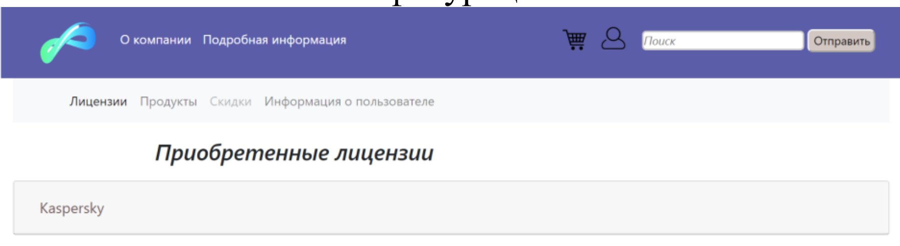

Рисунок 3.12 – Страница после встраивания фрагментов с проектной конфигурацией

Кроме вынесения UI-частей, а также методов, отвечающих за их создание, в рамках компонентной модели можно вынести различные сервисы, которые являются базовыми и часто используются большинством микросервисов. В микросервисной архитектуре появления подобных сервисов неизбежно. Это могут быть сервисы локализации/интернационализации, авторизационный сервис. Кроме того, если каждый микросервис в отдельности будет запрашивать одинаковые данные на сервере, то запросов при инициализации будет много, что приведет к «проседанию» производительности системы.

Таким образом, была реализована компонентная модель пользовательского интерфейса системы. В компоненты были вынесены шапка и подвал. Также они были встроены в микросервис как со стандартной конфигурацией, так и с проектной. Тем самым было показано, что реализованная модель работает корректно.

4 Доказательство эффективности компонентной модели пользовательской части системы

4.1 Экономические показатели эффективности работы разработчиков

Продуктивное использование ресурсов команды разработчиков, её время, предназначенное для определенного процесса разработки, позволяет оптимизировать трудовую деятельность каждого разработчика команды.

Используя определенные показатели и их формулы для вычисления, можно вывести временные затраты, необходимые разработчику для решения поставленной задачи. Выполнив необходимые расчёты, можно выявить оперативность разработчиков команды, а также существенно увеличить показатели эффективности процесса разработки [9].

Таким образом, знание определенных экономических показателей позволит увеличить производительность команды. Для этого необходимо воспользоваться формулами «человеко-часы» и «человеко-дни».

Для оптимизации процесса разработки, сокращения времени разработки и получения большей выгоды необходимо рассчитывать количество затраченных на разработку одной «единицы» человеко-часов [24][39]. Под термином «человеко-часы» понимают единицу учёта рабочего времени, соответствующую одному часу работы одного разработчика.

Термин «человеко-час» применяется для:

– планирования рабочего времени на ту или иную работу;

– определения количество разработчиков, необходимых для выполнения того или иного процесса.

При подсчете учитывается только время фактического труда. То есть игнорируется время больничных, отпусков, другого неоплачиваемого или оплачиваемого времени, в течение которого работа не выполнялась. Другими словами, данный показатель обозначает один полностью отработанный час одним конкретным разработчиком.

Учетная единица просто необходима при планировании объема задач для команды разработчиков на спринт или итерацию [33]. Руководителю команды разработчиков с помощью данного показателя удобно планировать рабочее время членов команды, определять их необходимое количество для выполнения работы, а также устанавливать сроки для выполнения задачей.

Таким образом, используя данный показатель можно вычислить временной отрезок или количество разработчиков, которые требуются для осуществления определенной задачи, направленной на разработку или багофикс.

Также показатель применяют для выявления сроков выполнения обязательств во время реализации определённых задач, ставящихся руководством в жесткие ограничения по времени [24].

Для определения данного показателя, необходимо сложить время трудовой деятельности определенного разработчика. Однако, учитываются как часы, потраченные на процесс разработки на территории компании, так и за её пределами. То есть, при вычислении данного показателя в расчет включается время, затраченное в командировках, сверхурочные, а также трудовая занятность на совмещаемой должности.

Для вычисления показателя «человеко-час» используется формула (1).

$$Ч = КР * РВ, \qquad (1)$$

где Ч – показатель «человеко-час»;

КР – общее количество разработчиков команды;

РВ – фактическое время, которое было потрачено на выполнение работы.

При расчет данного показателя стоит игнорировать:

- период болезни разработчика;
- время простоев;
- период отпусков;

– время, на которое был сокращен трудовой день разработчика определённых категорий в соответствии с указаниями законодательства РФ [39].

Формула (2) используется при вычислении стоимости человеко-часа конкретного разработчика.

$$Ц = ЗП * РЧ, \quad (2)$$

где Ц – стоимость человеко-часа;

ЗП – заработная плата конкретного разработчика за месяц (чистая);

РЧ – количество рабочих часов в месяц, исключая часы перерывов (обеденное время, длительные перерывы, связанные с не работой проекта или компании), отпусков (ежегодных, за свой счёт, дополнительных и прочие), кратковременных отлучений от работы (телефонных звонков, не связанных с работой и т.п.).

Для вычисления человеко-часов всей команды используется формула (3), где необходимо сложить индивидуальные человеко-часы разработчиков.

$$ЧЧ = ЧЧ_1 + ЧЧ_2 + \cdots + ЧЧ_n, \quad (3)$$

где ЧЧ – общее количество отработанного времени;

$ЧЧ_1$ – время, отработанное первым разработчиком;

$ЧЧ_2$ – время, отработанное вторым разработчиком;

$ЧЧ_n$ – время, отработанное n-м разработчиком.

Термин «человеко-день» описывает единицу измерения рабочего времени, которая соответствует рабочему дню одного разработчика независимо от числа отработанных им часов [9][39]. Таким образом, данный показатель характеризует дни, отработанные разработчиками команды, независимо от фактической протяженности трудового дня и количества отработанного времени. То есть, явился ли разработчик на работу вовремя и своевременно начал исполнение своих должностных обязанностей или нет, человеко-день должен быть засчитан.

Другими словами, данный показатель является единицей времени трудовой деятельности, представляющей собой один стандартный рабочий день. При использовании его в расчетах не учитывается средняя длина рабочего дня, даже тогда, когда цифры значительно превышают установленные законодательством 8 часов.

В отличии от показателя «человеко-час», данный показатель является менее точным. Так, например, при его вычислении неуважительные причины неявки на рабочее место, такие как прогулы, неявки и отсутствие на своем рабочем месте дольше трёх часов не учитываются [24].

Человеко-дни используются для расчёта таких показателей как

- отработанные дни;
- явки;
- неявки;
- простои (если это день и более).

К отработанным дням в данном случает относятся:

- дни исполнения своих обязанностей на основном рабочем месте, то есть фактическое нахождения на работе;
- дни, проведенные в служебных командировках;
- дни, когда из-за вынужденного простоя разработчик задействован в другой деятельности команды [39].

Для расчёта показателя человеко-дней используется формула (4).

$$ЧД = \frac{(Ч1+Ч2+\cdots+Чn)*КДМ}{8}, \quad (4)$$

где ЧД – показатель «человеко-дней»;

$Ч_1$, $Ч_2$, $Ч_n$ – время, отработанное каждым разработчиком;

КДМ – количество календарных дней в месяце.

Таким образом, в данном пункте были рассмотрены показали, с помощью которых можно вычислить продуктивность разработчиков. Далее они будут высчитаны на примере конкретных задач.

4.2 Доказательство эффективности компонентной модели

Воспользовавшись показателями, указанными ранее докажем эффективность компонентной модели пользовательского интерфейса системы.

Допустим, имеется три команды разработчиков – команда микросервиса «Личный кабинет», команда микросервиса «Галерея» и команда «Продуктовый каталог». Каждая команда состоит из 10 разработчиков.

Команде «Личного кабинета» необходимо выполнить следующие задачи в спринте:

1) добавить новый функционал в шапку страницы, например, при открытии окошка для выхода из аккаунта, отображать временя, когда пользователь зашел в аккаунт;

2) исправить ошибку в подвале, например, изменить шрифт отображаемой информации и сделать отцентровку данной информации для браузера Explorer;

3) добавить отображение аватара пользователя на вкладке «Информация о пользователе».

Команде «Галереи» необходимо выполнить следующие задачи в спринте:

1) добавить новый функционал в шапку страницы, например, при открытии окошка для выхода из аккаунта, отображать временя, когда пользователь зашел в аккаунт;

2) исправить ошибку в подвале, например, изменить шрифт отображаемой информации и сделать отцентровку данной информации для браузера Explorer;

3) добавить отображение фотографии продуктов при их отображении.

Команде «Продуктовый каталог» необходимо выполнить следующие задачи в спринте:

1) добавить новый функционал в шапку страницы, например, при открытии окошка для выхода из аккаунта, отображать временя, когда пользователь зашел в аккаунт;

2) добавить возможность создавать копии сконфигурированных продуктов вместе со всеми зависимостями.

Команды выделяют для каждой задачи одного разработчика. При вычислении «человеко-часа» будет учитываться только время работы по факту, часы работы, проведенные в служебных командировках, переработки, сверхурочные, праздничные часы труда и ночные. Разработчики «Личного кабинета» выполнят первую задачу за 2 дня, то есть за 16 часов, вторую и третью задачу за день – за 8 часов. Разработчики «Галереи» выполнят первую задачу за 2,5 дня, то есть за 20 часов, вторую за день – за 8 часов, а третью за 1.5 дня – за 12 часов. Команда «Продуктовый каталог» выполнит первую задачу за 2 дня, то есть за 16 часов, вторую задачу за 4 дня – 32 часа.

Воспользовавшись формулой (1) вычислим показатель «человеко-час» для каждой задачи.

Формулы (5) – (7) отображают данные вычисления для команды личного кабинета.

$$Ч_1 = КР_1 * РВ_1 = 1 * 16 = 16, \quad (5)$$

$$Ч_2 = КР_2 * РВ_2 = 1 * 8 = 8, \quad (6)$$

$$Ч_3 = КР_3 * РВ_3 = 1 * 8 = 8, \quad (7)$$

где $Ч_1$ – показатель «человеко-час» для первой задачи;

$КР_1$ – количество разработчиков для первой задачи;

$РВ_1$ – время, фактически затраченное для решения первой задачи;

$Ч_2$ – показатель «человеко-час» для второй задачи;

$КР_2$ – количество разработчиков для второй задачи;

$РВ_2$ – время, фактически затраченное для решения второй задачи;

$Ч_3$ – показатель «человеко-час» для третьей задачи;

$КР_3$ – количество разработчиков для третьей задачи;

$РВ_3$ – время, фактически затраченное для решения третьей задачи.

Формулы (8) – (10) отображают данные вычисления для команды галереи.

$$Ч_1 = КР_1 * РВ_1 = 1 * 20 = 20, \quad (8)$$

$$Ч_2 = КР_2 * РВ_2 = 1 * 8 = 8, \quad (9)$$

$$Ч_3 = КР_3 * РВ_3 = 1 * 12 = 12, \quad (10)$$

где $Ч_1$ – показатель «человеко-час» для первой задачи;

$КР_1$ – количество разработчиков для первой задачи;

$РВ_1$ – время, фактически затраченное для решения первой задачи;

$Ч_2$ – показатель «человеко-час» для второй задачи;

$КР_2$ – количество разработчиков для второй задачи;

$РВ_2$ – время, фактически затраченное для решения второй задачи;

$Ч_3$ – показатель «человеко-час» для третьей задачи;

$КР_3$ – количество разработчиков для третьей задачи;

$РВ_3$ – время, фактически затраченное для решения третьей задачи.

Формулы (11), (12) отображают данные вычисления для команды каталога продуктов.

$$Ч_1 = КР_1 * РВ_1 = 1 * 16 = 16, \quad (11)$$

$$Ч_2 = КР_2 * РВ_2 = 1 * 32 = 32, \quad (12)$$

где $Ч_1$ – показатель «человеко-час» для первой задачи;

$КР_1$ – количество разработчиков для первой задачи;

$РВ_1$ – время, фактически затраченное для решения первой задачи;

$Ч_2$ – показатель «человеко-час» для второй задачи;

$КР_2$ – количество разработчиков для второй задачи;

$РВ_2$ – время, фактически затраченное для решения второй задачи.

Воспользуемся формулой (3) для вычисления человеко-часов всей команды. Для этого необходимо сложить индивидуальные человеко-часы

разработчиков. Формулы (13) – (15) отображают вычисление данного показателя для каждой команды.

$$ЧЧ_1 = 16 + 8 + 8 = 32, \quad (13)$$

$$ЧЧ_2 = 20 + 8 + 12 = 40, \quad (14)$$

$$ЧЧ_3 = 16 + 32 = 48, \quad (15)$$

где $ЧЧ_1$ – общее количество отработанного времени команды личного кабинета;

$ЧЧ_2$ – общее количество отработанного времени команды галереи;

$ЧЧ_3$ – общее количество отработанного времени команды каталога продуктов.

Таким образом, для решения данных трех задач команда личного кабинета потратит 32 «человеко-час», команда галереи – 40 «человеко-час», а команда каталога продуктов – 48 часов.

Используя компонентную модель данные показатели можно сократить. Вынося в отдельные фрагменты – компоненты, например, шапку и подвал, можно избежать необходимости выделять разработчиков каждой команды на задачи данных фрагментов, так как команды будут встраивать фрагменты на своей стороне с уже готовыми исправлениями или новым функционалом.

То есть команда, отвечающая за данные фрагменты, потратит определенное количество «человеко-часов», в то время как команды микросервисов не будут затрачивать их.

Другими словами, при использовании компонентной модели в спринте команда личного кабинета потратит на данные задачи 8 «человеко-часа», команда галереи - 12 «человеко-часа», а команда каталога продуктов – 32 «человеко-часа».

Таким образом, команды микросервисов могут потрать свои ресурсы на решения задач, которые являются уникальными для них. Команда личного кабинета и каталога продуктов может добрать на спринт задач на 16 «человеко-часов», а команда галереи – на 20 «человеко-часов», тем самым улучшая свой микросервис, не тратя при этом лишние ресурсы всего проекта.

Стоимость человеко-часа также может быть уменьшена, если, например, для решения повторяющихся задач разработчики каждой команды выходили в сверхурочные.

На показатель «человеко-день» данная модель не влияет, так как при вычислении данного показателя не учитывается фактически затраченное время, а считается рабочие дни. То есть данный показатель не будет высчитывается, так как реализованная модель не влияет на него.

Таким образом, было доказано, что использование компонентной модели пользовательского интерфейса позволит эффективнее использовать ресурсы и время команды разработчиков.

4.3 Тестирование компонентной модели

В данном пункте будет рассмотрено тестирование компонентной модели. Тестирование будет происходит с учетом вынесения шапки и подвала во фрагменты. Также при тестировании будет учитываться, что для личного кабинета и галереи у данных фрагментов была задана одинаковая конфигурация.

Функциональным тестированием называют тестирование системы, приложения, программного обеспечения в целях проверки реализуемости функциональных требований. Другими словами, в процессе тестирования происходит проверка способности при определённых условиях решать задачи необходимые пользователю. Данное тестирование проводится на основании тест кейса, то есть последовательности действий, направленной на проверку какого-либо функционала и описывающей как добиться ожидаемого результата [18][21][23][26].

В рамках данной работы тест кейс будет выглядеть следующим образом:

1. Заходим в «Личный кабинет»

ER = Шапка и подвал отобразились, в шапке доступна информация о компании, её логотип, отображаются ссылка на подробная информацию о ней, иконка корзинки, поиск и иконка пользователя, в подвале отображается ссылки на информацию о компании и контактный телефон.

2. Переходим на вкладку «Продукты»

ER = Показаны карточки с продуктами.

3. Находим иконку пользователя

ER = Иконка пользователя отображается в шапке в крайнем правом верхнем углу.

4. Кликаем на иконку пользователя

ER = Открывается вкладка «Информация о пользователе».

5. Находим иконку корзины

ER = Иконка корзины находится в шапке левее поля поиска

6. Кликаем на иконку корзины

ER = Переходим на главную страницу галереи.

7. Заходим в галерею

ER = Шапка и подвал отобразились, в шапке доступна информация о компании, её логотип, отображаются ссылка на подробную информацию о ней, иконка корзинки, поиск и иконка пользователя, в подвале отображается ссылки на информацию о компании и контактный телефон, на основной странице отображаются доступные для покупки продукты.

8. Находим иконку корзины

ER = Иконка корзины находится в шапке левее поля поиска

9. Кликаем на иконку корзины

ER = Переходим на главную страницу галереи

10. Находим иконку пользователя

ER = Иконка пользователя отображается в шапке в крайнем правом верхнем углу.

11. Кликаем на иконку пользователя

ER = Открывается вкладка «Информация о пользователе».

Выполнив данный test case получаем следующее:

1. Заходим в «Личный кабинет»

AR = ER = Шапка и подвал отобразились, в шапке доступна информация о компании, её логотип, отображаются ссылка на подробную информацию о ней, иконка корзинки, поиск и иконка пользователя, в подвале отображается ссылки на информацию о компании и контактный телефон, что можно увидеть на рисунке 4.1.

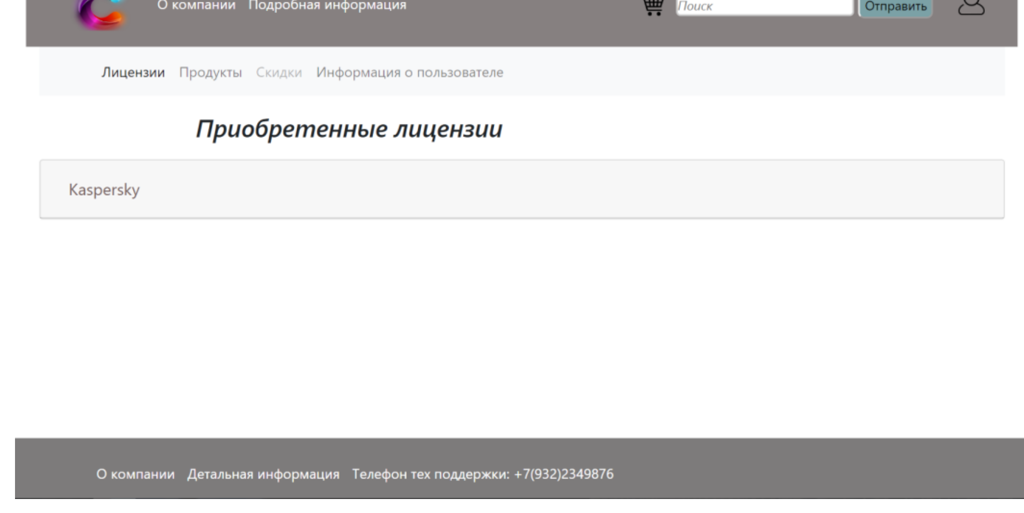

Рисунок 4.1 – Личный кабинет

2. Переходим на вкладку «Продукты»

AR = ER = Показаны карточки с продуктами, что отображается на рисунке 4.2.

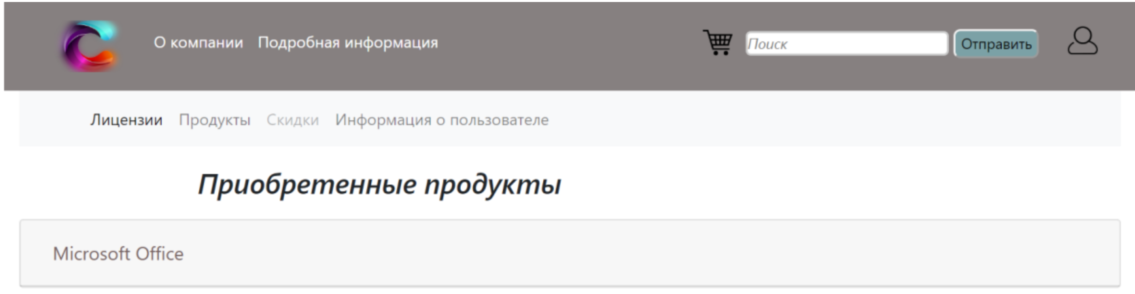

Рисунок 4.2 – Вкладка «Продукты»

3. Находим иконку пользователя.

AR = ER = Иконка пользователя отображается в шапке в крайнем правом верхнем углу.

4. Кликаем на иконку пользователя.

AR = ER = Открывается вкладка «Информация о пользователе», что показано на рисунке 4.3.

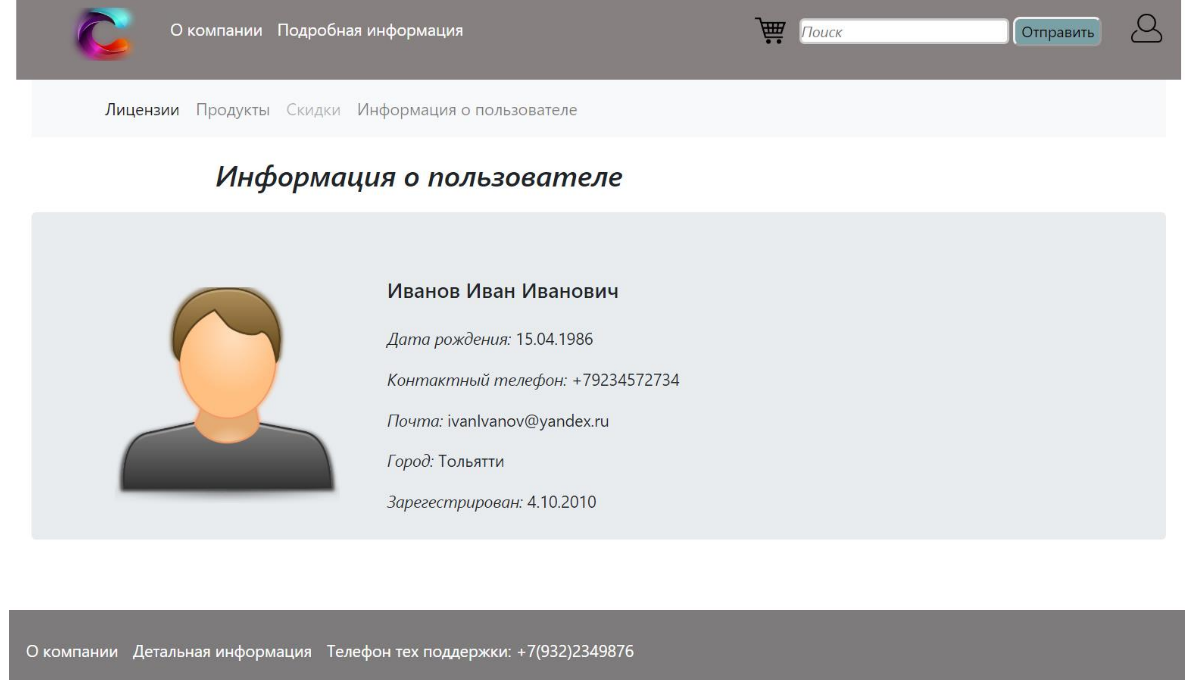

Рисунок 4.3 – Вкладка «Информация о пользователе»

5. Находим иконку корзины

AR = ER = Иконка корзины находится в шапке левее поля поиска

6. Кликаем на иконку корзины

AR = ER = Переходим на главную страницу галереи.

7. Заходим в галерею

AR = ER = Шапка и подвал отобразились, в шапке доступна информация о компании, её логотип, отображаются ссылка на подробную информацию о ней, иконка корзинки, поиск и иконка пользователя, в подвале отображается ссылки на информацию о компании и контактный телефон, на

основной странице отображаются доступные для покупки продукты, что можно увидеть на рисунке 4.4.

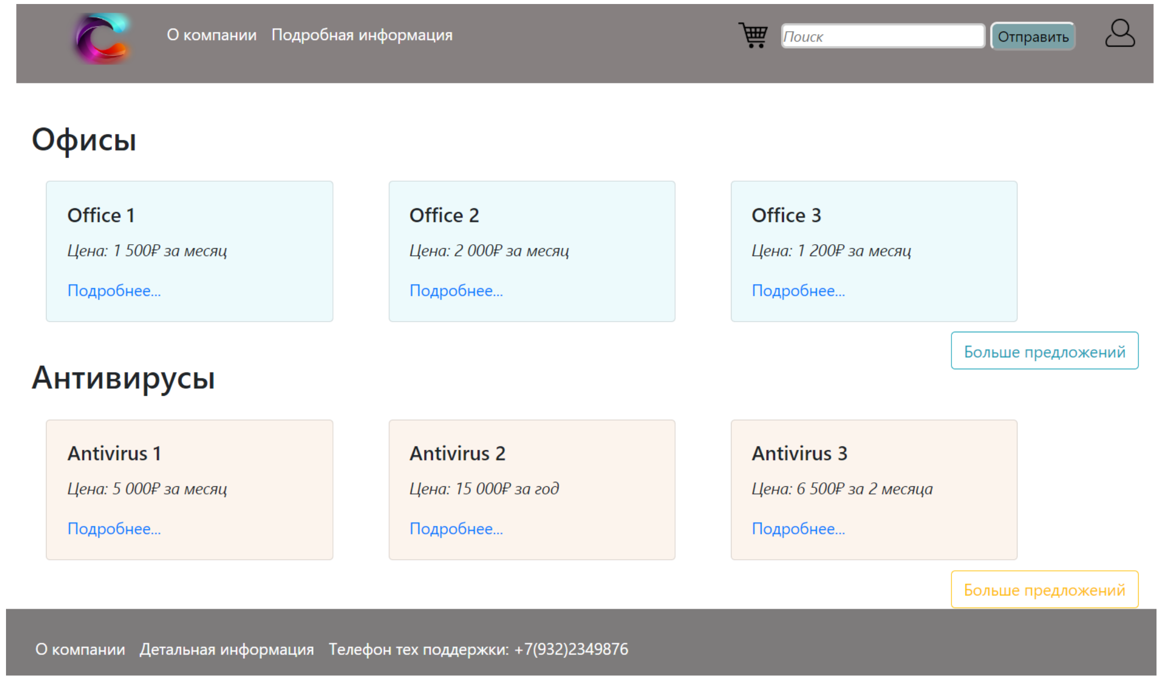

Рисунок 4.4 – Главная страница галереи

8. Находим иконку корзины

AR = ER = Иконка корзины находится в шапке левее поля поиска

9. Кликаем на иконку корзины

AR = ER = Переходим на главную страницу галереи.

10. Находим иконку пользователя

AR = ER = Иконка пользователя отображается в шапке в крайнем правом верхнем углу.

11. Кликаем на иконку пользователя

AR = ER = Открывается вкладка «Информация о пользователе».

Описанный тест кейс был пройден, то есть после встраивания фрагментов шапки и подвала микросервисы работают ожидаемым образом. Другими словами, в данной пункте было показано, что компонентная модель работает корректно.

Таким образом, было доказано, что при использовании компонентной модели можно разумнее и эффективнее использовать ресурсы и время команды разработчиков, а также то, что данная модель работает корректно.

Заключение

В рамках данного исследования были рассмотрены различные научные статьи и диссертации на тему архитектурных подходов разработки системы. В первой части данной работы были описаны достоинства и недостатки архитектурных решений, которые авторы выделяли в своих работах.

Монолит обеспечивает достаточно легкое начало и несет наименьшее количество расходов, но цена поддержки подобных систем увеличивается быстрее по сравнению с аналогами, реализованными с помощью других подходов. Также данные системы сложно масштабируемы и имеют ряд проблем, связанных с тем, что все их модули находятся на одной аппаратной части. Однако существует возможность допуска ошибки на его начальных этапах, потому что их можно быстро исправить.

Для реализации систем модульным решением необходимы большие расходы на начальной стадии, однако увеличение цены поддержки подобных систем менее резок. Данный факт обеспечивает поддержку подобных систем дольше и проще, чем монолита. В рамках данного решения происходит синхронное взаимодействие модулей. Однако, различие данного подхода и монолита весьма неявное, так как изначально подобные системы пишутся модулями, с явно описанной функциональностью каждого и четкими границами между ними. Но и монолит может включать в себя разные компоненты и классы, зависящие от данных других модулей и друг от друга. Другой проблемой модульного решения является реализация функциональности, существующей по другой схеме в другой части системы.

При использование микросервисного подхода необходимо затрачивать большие ресурсы на начальных этапах, потому что необходимо достаточно обширное проектирование развития системы. Однако подход имеет ряд преимуществ и в большей степени удовлетворяет современным требованиям. Микросервисный подход необязательно подходит для каждой системы, требующий иного подхода к проектированию, так как необходимо понимать принципы работы с ними, учитывать возможные побочные эффекты при

работе с асинхронной связью, продумывать четкие границы ответственности команд разработчиков.

На основании выше рассмотренного был сделан вывод, заключающееся в том, что каждое рассмотренное архитектурное решение предоставляет свои плюсы и минусы. Однако в рамках исследования была найдена проблема, которая авторами не рассматривалась – дублирование функционала. Таким образом, в рамках работы необходимо было сформулировать модель, которая могла бы исключить нерациональное использование ресурсов команды разработчиков, убрав дублирование функционала. Другими словами, было доказано, что тема диссертации является актуальной, так как в рамках неё может быть решены насущные проблемы.

Модель, способная решить обнаруженную проблему, предполагает изолированность UI-частей, которые используется многими микросервисами в микросервисном подходе. Она называется компонентной моделью, потому что весь фронтенд (UI, интерфейс) различных микросервисов разбит на компоненты-фрагменты.

В рамках данной работы были сформулированы требования к компонентной модели пользовательского интерфейса системы, введены новые термины и определения, подробно описана суть модели, было рассмотрено определение «фрагмент» и его разновидности, а также были выделены UI-части, дублируемые в различных микросервисах. Кроме того, были выбраны технологии реализации, более того сформулированная компонентная модель пользовательского интерфейса системы была реализована.

Фрагментом является независимая UI-часть, которая соблюдает набор правил разработки, для того, чтобы его можно было использовать в общем интерфейсе системы, его стили должны быть максимально специфичны, прямого взаимодействия с другими фрагментами быть не должно.

Фрагменты компонентной модели необходимо разрабатывать как отдельные приложения, они будут отвечать за некий набор бизнес-функций,

то есть будет являться полностью функциональным от начала и до конца. Также, фрагменты подобно обычным микросервисам могут быть написаны на различных языках. Кроме того, они могут встраиваться не только в микросервисы, некоторые фрагменты могут служить контейнерами для встраивания других фрагментов.

Для того, чтобы микросервисы могли встраивать фрагменты у себя на странице, им необходимо инициализировать Контроллер, который будет предоставлять микросервисам методы, для встраивания фрагментов. Для это микросервисам необходимо указать их название и конфигурационный флаг, а также поставить некий маркер – определенный класс – у себя в шаблоне. Если при инициализации конфигурационный флаг не был указан, считается что у фрагментов будет дефолтовая конфигурация, а следовательно, дефолтные стили и поведение.

Также в рамках данной работы были рассмотрены механизмы взаимодействия микросервисов между собой, а также микросервисов и встроенных в них компонентов. В рамках компонентной модели микросервисы будут взаимодействовать между собой посредством API. То есть клиент, обращаясь к шлюзу, который переадресовывает запрос к нужному микросервису, минует обращение к каждому из них по отдельности. А фрагменты и микросервисы в рамках данной модели взаимодействуют, используя Event Bus. То есть при изменении состояния компонента он сообщает об этом, используя метод *publish()*, в то время как микросервис, которому важны данные изменения подписывается на любые изменения компонента, используя метод *subscribe()*. Таким образом, микросервис своевременно обновляет данные, тем самым владея актуальной информацией.

Кроме того, в рамках данной работы была доказана эффективность реализованной модели, так как важным аспектом любой трудовой деятельности является продуктивность, потому что рациональное и продуктивное использование ресурсов команды разработчиков, её времени,

предназначенное для определенного процесса разработки, позволяет оптимизировать трудовую деятельность этой команды.

Используя определенные экономические показатели, можно вывести временные затраты, необходимые разработчику для решения поставленной задачи. Другими словами, выполнив необходимые расчёты этих показателей, можно выявить оперативность разработчиков команды, а также существенно увеличить показатели эффективности процесса разработки.

Таким образом, знание определенных экономических показателей позволит увеличить производительность команды. Например, для оптимизации процесса разработки, сокращения времени разработки и получения большей выгоды необходимо было рассчитать количество затраченных на разработку одной «единицы» человеко-часов.

Учетная единица просто необходима при планировании объема задач для команды разработчиков на спринт или итерацию. Руководителю команды разработчиков с помощью данного показателя удобно планировать рабочее время членов команды, определять их необходимое количество для исполнения задач, а также устанавливать сроки для завершения работы. Нередко человеко-часы используются для установки сроков выполнения обязательств во время проектирования определённых задач, ставящихся руководством в жесткие ограничения по времени.

То есть, используя данный показатель можно вычислить временной отрезок или количество разработчиков, которые требуются для осуществления определенной задачи, направленной на разработку или багофикс.

Для этого в рамках данной работы были вычислены показатели «человеко-часы» команд при выполнении определенных задач с и без использования компонентной модели. А затем были вычислены человеко-часы всей команды, где необходимо было сложить индивидуальные человеко-часы разработчиков.

Под термином «человеко-часы» понимают единицу учёта рабочего времени, соответствующую одному часу работы одного разработчика. Данный показатель используется для планирования рабочего времени на ту или иную работу и вычисления количества разработчиков, необходимых для выполнения того или иного процесса. При вычислении данного показателя учитывалось только время фактического труда. Не учитывалось время больничных, отпусков, и другого неоплачиваемого или оплачиваемого времени, в течение которого работа не выполнялась. Другими словами, данный показатель обозначал один полностью отработанный час одним конкретным разработчиком.

В итоге, было вычислено, что, для решения описанных трех задач команда личного кабинета потратит 32 «человеко-час», команда галереи – 40 «человеко-час», а команда каталога продуктов – 48 часов.

Используя компонентную модель данные показатели можно было сократить. Вынося в отдельные компоненты, например, шапку и подвал, можно избежать необходимости выделять разработчиков каждой команды на задачи, связанные с ними, так как команды будут встраивать фрагменты на своей стороне с уже готовыми исправлениями или новым функционалом.

То есть команда, отвечающая за данные фрагменты, потратит определенное количество «человеко-часов», в то время как команды микросервисов не будут затрачивать их.

Другими словами, при использовании компонентной модели в спринте команда личного кабинета потратит на данные задачи 8 «человеко-часа», команда галереи - 12 «человеко-часа», а команда каталога продуктов – 32 «человеко-часа».

Таким образом, команды микросервисов могут потрать свои ресурсы на решения задач, которые являются уникальными для них. Команда личного кабинета и каталога продуктов может добрать на спринт задач на 16 «человеко-часов», а команда галереи – на 20 «человеко-часов», тем самым улучшая свой микросервис, не тратя при этом лишние ресурсы всего проекта.

Стоимость человеко-часа также может быть уменьшена, если, например, для решения повторяющихся задач разработчики каждой команды выходили в сверхурочные.

Таким образом, было доказано, что использование компонентной модели пользовательского интерфейса позволит эффективнее использовать ресурсы и время команды разработчиков.

Затем, в четвертой части данной работы было проведено функциональное тестирование системы, которая была реализована с помощью компонентной модели.

Функциональным тестированием называют тестирование системы, приложения, программного обеспечения в целях проверки реализуемости функциональных требований, то есть во время тестирования происходит проверка способности п определённых условиях решать задачи необходимые пользователю. Данное тестирование проводится на основании тест кейса, то есть последовательности действий, направленной на проверку какого-либо функционала и описывающей как добиться ожидаемого результата.

В рамках данной системы в компоненты были вынесены шапка и подвал. Также при тестировании было учтено, что микросервисами была задана одинаковая конфигурация данным фрагментам. Описанный тест кейс был пройдет, то есть после встраивания фрагментов шапки и подвала микросервисы работали ожидаемым образом. Другими словами, было показано, что компонентная модель работает корректно.

Таким образом, в рамках данного исследования была найдена актуальная проблема, было описано и реализовано её решение. Также было доказано, что при использовании компонентной модели можно разумнее и эффективнее использовать ресурсы и время команды разработчиков, а также то, что данная модель работает корректно.

Список используемой литературы

1. Артамонов И.В. Исторические аспекты появления микросервисной архитектуры – Текст: электронный – URL: https://cyberleninka.ru/article/v/istoricheskie-aspekty-poyavleniya-mikroservisnoy-arhitektury (дата обращения: 18.09.2018)

2. Артамонов И.В. Разработка распределенных сервисно-ориентированных программных средств: учеб. пособие / И.В. Артамонов. — Иркутск: Изд-во БГУЭП, 2016. — 129 с.

3. Артамонов И.В. Слабое связывание как фактор эволюции технологий интеграции распределенных систем / И.В. Артамонов // Материалы конференции «Инновации на основе информационных и коммуникационных технологий»

4. Артамонов Ю.С. Разработка распределенных приложений сбора и анализа данных на базе микросервисной архитектуры / Ю.С. Артамонов, С.В. Востокин // Известия Самарского научного центра Российской академии наук, 2016 – т. 18

5. Асанович А. Компьютерные средства и эволюция методологии архитектурного проектирования – Текст: электронный – URL: http://www.dissercat.com/content/kompyuternye-sredstva-i-evolyutsiya-metodologii-arkhitekturnogo-proektirovaniya (дата обращения: 10.10.2018)

6. Аунг Аунг Хейн Моделирование и разработка сервис-ориентированных приложений – Текст: электронный – URL: http://www.dissercat.com/content/modelirovanie-i-razrabotka-servis-orientirovannykh-prilozhenii (дата обращения: 23.10.2018)

7. Богдаренко Д.А. Подходы к архитектурному проектированию веб-приложений – Текст: электронный – URL: https://moluch.ru/archive/195/48609/ (дата обращения: 16.11.2018)

8. Валиков А.Н. Модели и методы разработки крупномасштабных веб-приложений – Текст: электронный – URL:

http://www.dissercat.com/content/modeli-i-metody-razrabotki-krupnomasshtabnykh-veb-prilozhenii (дата обращения: 21.09.2018)

9. Генкин Б.М. Анализ производительности труда / Б.М. Генкин. М.: НОРМАИНФРАМ, 2016. – 506 с.

10. Григорьева Т.Е. Моделирование систем массового обслуживания на примере очереди в банке // Научная сессия ТУСУР-2016: материалы Международной научно-технической конференции студентов, аспирантов и молодых ученых

11. Грученков В.В. Микросервисная архитектура Web приложений / В.В. Грученков // Компьютерные системы и сети: материалы 52-й научной конференции аспирантов, магистрантов и студентов (Минск, 25-30 апреля 2016 года) / [редколлегия: В.А. Прытков и др.] – Минск: БГУИТ, 2016 – 174, [4] с.

12. Дмитриев В.М. Формализм сетей Петри с транзакциями для моделирования бизнес-процессов / В.М. Дмитриев, Т.Е. Григорьева, С.А. Панов, Е.В. Истигечева, И.В. Дмитриев // Информатика и системы управления

13. Зяблов Д.В. Применение микросервисной архитектуры при разработке корпоративных веб-приложений – Текст: электронный – URL: https://sibac.info/journal/student/18/87616 (дата обращения: 15.12.2018)

14. Ильиных Г. Микросервисы или монолит: в поисках золотой середины – Текст: электронный – URL: https://blog.gramant.ru/2017/05/17/microservices-vs-monolith-compromise/ (дата обращения: 26.09.2018)

15. Истигечева Е.В. Моделирование бизнес-процессов на примере модели «Хранение» / Е.В. Истигечева, Т.Е. Григорьева, С.А. Панов – Текст: непосредственный // сборник «Научная дискуссия: вопросы технических наук»: Сборник статей по материалам XL международной заочной научно-практической конференции, место издания Изд. «Интернаука» Москва, том 29, с. 17-22

16. Истигечева Е.В. Моделирование бизнес-процессов на примере модели «Сбыт» / Е.В. Истигечева, Т.Е. Григорьева, С.А. Панов // Текст: непосредственный // журнал «Таврический научный обозреватель», том 2, № 3, с. 55-59

17. Истигечева Е.В. Моделирование логических схем бизнес-процессов / Е.В. Истигечева, Т.Е. Григорьева // Текст: непосредственный // журнал «Информатика и системы управления», том 2, № 48, с. 36-47

18. Канер Сэм Тестирование программного обеспечения. Фундаментальные концепции менеджмента бизнес-приложений: Пер. с англ./Сэм Канер, Джек Фолк, Енг Кек Нгуен. — К.: Издательство «ДиаСофт», 2015. — 544 с. ISBN 966-7393-87-9

19. Ким К. Микросервисы, SOA и API: друзья или враги? – Текст: электронный – URL: https://www.ibm.com/developerworks/ru/library/1601_clark-trs/index.html (дата обращения: 25.12.2018)

20. Когда использовать монолиты, а когда микросервисы – Текст: электронный – URL: https://eax.me/microservices-vs-monolithic/ (дата обращения: 22.10.2018)

21. Криспин Л. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд / Л. Криспин, Д. Грегори – К.: Издательство: «Вильямс», 2016. – 464с. ISBN 978-5-8459-1625-9

22. Крупный Н.В. Оптимизация получения и отображения данных с геоинформационного сервера на уровне клиента – Текст: электронный – URL: http://elib.spbstu.ru/dl/2/v16-2290.pdf/download/v16-2290.pdf (дата обращения: 20.10.2018)

23. Куликов, С. С. Тестирование программного обеспечения. Базовый курс / С. С. Куликов. — Минск: Четыре четверти,2017. — 312 с. ISBN 978-985-581-125-2

24. Кутафьева Л. В. Анализ использования рабочего времени // Молодой ученый. — 2015. — №3. — С. 242-244. — Текст: электронный – URL https://moluch.ru/archive/50/6315/ (дата обращения: 02.04.2020)

25. Логинов А.С. Разработка Web приложения для работы с аудиоматериалами – Текст: электронный – URL: http://elib.spbstu.ru/dl/2/7046.pdf/en/info (дата обращения: 17.02.2019)

26. Майерс Г. Искусство тестирования программ // Г. Майерс, Т. Баджетт, К. Сандлер, 2016. – 272с. ISBN 978-5-907144-37-8

27. Марков Е. Архитектура распределенных систем – Текст: электронный – URL: https://www.itweek.ru/infrastructure/article/detail.php?ID=66147 (дата обращения: 20.09.2018)

28. Марков Е. Архитектура распределенных систем – Текст: электронный – URL: https://www.itweek.ru/infrastructure/article/detail.php?ID=66147 (дата обращения: 22.03.2019)

29. Ньюмен С. Создание микросервисов / С. Ньюман – Текст: непосредственный // СПб.: Питер, 2016. — 304 с.: ил. — (Серия «Бестселлеры O’Reilly»)

30. Оборин А.О. Компонентный подход к разработке крупномасштабных веб-проектов при использовании гибких методологий разработки – Текст: электронный – URL: http://elib.spbstu.ru/dl/2/v17-2105.pdf/download/v17-2105.pdf (дата обращения: 20.04.2019)

31. Осипова Н.Д. Разработка микросервиса интеграции системы самообслуживания абонентов сотовой связи и центра нотификаций в рамках перехода от монолитной архитектуры приложения к микросервисной – Текст: электронный – URL: https://sibac.info/conf/naturscience/liii/75901 (дата обращения: 19.09.2018)

32. Панов С.А Интерактивное документирование бизнес-процессов в среде моделирования МАРС / С.А. Панов, Т.Е. Григорьева // Электронные

средства и системы управления: Материалы докладов XI Международной научно-практической конференции.

33. Райзберг Б.А. Современный экономический словарь // Б.А. Райзберг, Л.Ш. Лозовский, Е.Б. Стародубцева – Текст: электронный // Москва: ИНФРА-М, 2017 – 512 с. – ISBN 978-5-16-105386-7.

34. Серрано Н. Сервисы, архитектура и унаследованные системы – Текст: электронный – URL: https://www.osp.ru/os/2014/08/13043486/ (дата обращения: 8.09.2018)

35. Токарчук А.М. Повышение эффективности методов и алгоритмов разработки, взаимодействия и хранения веб-приложений – Текст: электронный – URL: http://www.dissercat.com/content/povyshenie-effektivnosti-metodov-i-algoritmov-razrabotki-vzaimodeistviya-i-khraneniya-veb-pr (дата обращения: 16.01.2019)

36. Файлер М. Архитектура корпоративных программных приложений – Текст: электронный – URL: http://www.ooart.ru/uploads/book/arhitektura_korporativnyh_programmnyh_prilozhenij_fauler_m.pdf (дата обращения: 26.05.2019)

37. Федоров А., Елманова Н. Архитектура современных Web-приложений – Текст: электронный – URL: https://compress.ru/article.aspx?id=10951 (дата обращения: 15.02.2019)

38. Хориков В. Микросервисы – Текст: электронный – URL: https://habr.com/post/249183/ (дата обращения: 27.11.2018)

39. Шаркова А.В. Словарь финансово-экономических терминов // А.В. Шаркова, А.А. Килячкова, Е.В. Маркина, С.П. Солянникова, Л.А. Чалдаева – Текст: электронный // Москва, Издательско-торговая корпорация «Дашков и К», 2016. – 1167 с.

40. Шинь Н.Г. Методы и модели проектирования бизнес-приложений – Текст: электронный – URL: http://www.dissercat.com/content/metody-i-modeli-proektirovaniya-biznes-prilozhenii-v-arkhitekture-klient-server-obektno-orie (дата обращения: 20.03.2019)

41. Шитько А.М. Проектирование микросервисный архитектуры программного обеспечения – Текст: электронный – URL: https://cyberleninka.ru/article/v/proektirovanie-mikroservisnoy-arhitektury-programmnogo-obespecheniya (дата обращения: 15.03.2019)

42. A Quck Primer on Microservices – Текст: электронный – URL: http://www.cloudcomputingexpo.com/node/3675288 (дата обращения: 24.09.2018)

43. Babak B.Introduction to Docker and Analysis of its Performance// B. Babak, J. Harrison, M. Ahmadi – Текст: электронный – URL: https://www.researchgate.net/publication/318816158_An_Introduction_to_Docker_and_Analysis_of_its_Performance (дата обращения: 20.05.2019)

44. Bhagwati M. Microservices vs Monolithic architecture – Текст: электронный – URL: https://medium.com/startlovingyourself/microservices-vs-monolithic-architecture-c8df91f16bb4 (дата обращения: 27.11.2018)

45. Bhaskaran S. Migration Monolithic Applications to Microservices – Текст: электронный – URL: https://dzone.com/articles/migrating-monolithic-applications-to-microservices (дата обращения: 28.10.2018)

46. Ciceri C. Many Small Monoliths: Microservices vs. Monolithic Architecture– Текст: электронный – URL: https://dzone.com/articles/many-small-monoliths-microservices-vs-monolithic-a (дата обращения: 28.11.2018)

47. Duffy A. Supporting «Design for reuse» with modular design – Текст: электронный – URL: https://www.researchgate.net/publication/241008753_Supporting_Design_for_Reuse'_with_Modular_Design (дата обращения: 29.11.2018)

48. Farrukh S. A Review of Latest Web Tools and Libraries for State-of-the-art Visualization – Текст: электронный – URL: https://www.researchgate.net/publication/308494000_A_Review_of_Latest_Web_Tools_and_Libraries_for_State-of-the-art_Visualization (дата обращения: 29.04.2019)

49. Hezbullah S Node.js Challenges in Implementation // S. Hezbullah, R. Tariq – Текст: электронный – URL: https://www.researchgate.net/publication/318310544_Nodejs_Challenges_in_Implementation (дата обращения: 17.11.2019)

50. Kamaruzzaman Md Is Modular Monolithic Software Architecture Really Dead? - Текст: электронный – URL: https://towardsdatascience.com/looking-beyond-the-hype-is-modular-monolithic-software-architecture-really-dead-e386191610f8 (дата обращения: 10.11.2018)

51. Kong Pattern: Microservices Architecture – Текст: электронный – URL: https://microservices.io/patterns/microservices.html (дата обращения: 20.10.2018)

52. Lewis J., Fowler M. Microservices – a definition of this new architectural term – Текст: электронный – URL: https://martinfowler.com/articles/microservices.html (дата обращения: 2.12.2018)

53. Martin Fowler How to break a Monolith into Microservices – Текст: электронный – URL: https://martinfowler.com/articles/break-monolith-into-microservices.html (дата обращения: 24.01.2019)

54. PengCheng Xiong, YuShun Fan, MengChu Zhou Approach to Analysis and Composition of Web Services – Текст: электронный – URL: http://www.cc.gatech.edu/~pxiong3/Papers/TSMCA2009.pdf (дата обращения: 4.04.2019)

55. Richardson C. Building Microservices: Using an API Gateway – Текст: электронный – URL: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/ (дата обращения: 27.04.2019)

56. Richardson C. Introduction to Microservices – Текст: электронный – URL: https://www.nginx.com/blog/introduction-to-microservices/ (дата обращения: 3.10.2018)

57. Richardson C. Service Discovery in a Microservices Architecture – Текст: электронный – URL: https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/ (дата обращения: 27.09.2019)

58. Rick E. Osowski Microservice in action – Текст: электронный – URL: https://www.ibm.com/developerworks/cloud/library/cl-bluemix-microservices-in-action-part-1-trs/index.html (дата обращения: 21.03.2019)

59. Shupring R., Canberra, Australia, A Model for Web Services Discovery with QoS, ACM SIGecom Exchanges, Volume 4, Issue 1. March 2015, p 1-10

60. Web Services Architecture / W3C Working Group Note 1. – Текст: электронный – URL: http://www.w3.org/TR/ws-arch/ (дата обращения: 2.05.2019)

61. Web Services Architecture – Текст: электронный – URL: http://www.w3.org/TR/ws-arch/ (дата обращения: 27.10.2019)

62. Web services business process execution language version 2.0– Текст: электронный – URL: http://docs.oasis-open.0rg/wsbpel/2.O/OS/wsbpel-v2.O-OS.pdf (дата обращения: 19.10.2019)