МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РЕСПУБЛИКИ КАЗАХСТАН

СӘТБАЕВ УНИВЕРСИТЕТІ

Институт кибернетики и информационных технологий
Кафедра «Программной инженерии»

Сман Нұрсұлтан

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

На соискание академической степени магистра

Название диссертации	Разработка приложения для SMS рассылок на микросервисной архитектуре
Направление подготовки	6М100200 – Системы информационной безопасности

Научный руководитель
кандидат тех. наук, проф.
_____ А. А. Мухамедгалиев
«27» июня 2020 г.

Оппонент
канд. тех. наук, профессор
_____ Н.Т. Дузбаев
«02» июля 2020 г.

Нормоконтроль
лектор
_____ Ж.М.Алибиева
«__» _____ 2020 г.

ДОПУЩЕН К ЗАЩИТЕ
Заведующий кафедрой ПИ
Доктор PhD
_____ М. Тұрдалыұлы
«__»_____ 2020 г.

Алматы 2020

Институт кибернетики и информационных технологий

Кафедра Программная инженерия

Специальность: 6М100200 – Системы информационной безопасности

УТВЕРЖДАЮ
Заведующий кафедрой ПИ
Доктор PhD
_____ М. Тұрдалыұлы
«___» _____ 2020 г.

ЗАДАНИЕ
на выполнение магистерской диссертации

магистранту Сман Нұрсұлтан

Тема диссертации: «Разработка приложения для SMS рассылок на микросервисной архитектуре»

Срок сдачи законченной диссертации «07» июня 2020г.

Исходные данные к магистерской диссертации. Дана задача на разработку приложения на основе микросервисной архитектуры. Поставленные цели и задачи диссертационной работы направлены на проектирование и на создание прототипа приложения для SMS рассылок.

Перечень подлежащих разработке в магистерской диссертации вопросов или краткое содержание магистерской диссертации: а) определение требований целей, задач и назначения разработки; б) исследование микросервисной архитектуры; в) анализ инструментов и технологий для разработки; г) проектирование способа взаимодействия микросервисов; д) проектирование и разработка системы на микросервисной архитектуре.

ГРАФИК
подготовки магистерской диссертации

Наименование разделов, перечень разрабатываемых вопросов	Сроки представления научному руководителю	Примечание
Раздел 1. Общее описание	15.01.2020ж. – 08.03.2020ж.	
Раздел 2. Основная часть	09.03.2020ж. – 28.04.2020ж.	

Консультации по проекту с указанием относящихся к ним разделов проекта

Раздел	Консультант, (уч. степень, звание)	Сроки	Подпись
Нормоконтроль	Ж.М.Алибиева, Лектор кафедры программная инженерия		

Дата выдачи задания "____"_____ 2020 г.

Заведующий кафедрой _____ М. Тұрдалыұлы

Научный руководитель _____ А. А. Мухамедгалиев

Задание принял к исполнению магистрант _____ Н. Сман

Дата "_____"_____ 2020 г.

АННОТАЦИЯ

Основными задачами данной работы являются разработка серверного приложения для массовой отправки SMS сообщений. Программное обеспечение будет реализовано с использованием микросервисно-ориентированного подхода. А также рассмотрим разбиение большой системы на части, точнее на микросервисы.

Разработанная система будет распределено на несколько микросервисов. Микросервисы будут взаимодействовать между собой используя протокол обмена сообщений AMQP. В данной работе описывается процесс моделирования сервисов, способы интеграции и общения микросервисов между собой и их основные свойства бизнес логики.

Разработка данной системы будет реализовываться с применением языка программирования Java и технологии Spring Framework.

АҢДАТПА

Бұл жұмыстың негізгі міндеттері – SMS-хабарламаларды жаппай жіберуге арналған серверлік қосымшаларды құрастыру болып табылады. Бағдарламалық жасақтама микросервиске бағытталған әдісті қолдана отырып жүзеге асырылады. Сондай-ақ, біз микросервистік қызметтерге дәл келетін үлкен өлшемді жүйені ұсынамыз.

Әзірленген жүйе бірнеше микросервистерге таратылатын болады. AMQP хабарлама хаттамасы. Бұл жұмыста модельдеу қызметтері, микросервистердің өзара әрекеттесу және өзара байланысу әдістері, олардың бизнес-логиканың негізгі қасиеттері сипатталған.

Бұл жүйе Java бағдарламалау тілі мен Spring Framework технологиясын қолдану арқылы жүзеге асырылады.

ABSTRACT

The main objectives of this work are the development of server applications for mass sending SMS messages. The software will be implemented using a microservice-oriented approach. We also introduce a standard large-size system that is accurate for microservices.

The developed system will be distributed over several microservices. AMQP messaging protocol. This paper describes the process of modeling services, the methods of integration and communication of microservices among themselves, and their basic properties of business logic.

The development of this system will be implemented using the Java programming language and Spring Framework technology.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Разработка системы для массовой отправки SMS сообщений на мобильные устройства является очень актуальным на рынке бизнеса. На сегодняшний день существует много споособов, в том числе и SMS рассылка. SMS рассылка является быстрым и эффективным способом информирования. Люди в течение всего дня держат мобильное устройство при себе. Поэтому SMS рассылка является очень хорошим способом для информирования.

Разработанное программное обеспечение предоставит функциональность быстрой и массовой отправки SMS коротких сообщении. Клиентам будет предоставлен простой и удобный REST API интерфейс для SMS рассылки. Чтобы гарантировать стабильную работу системы будем рассматривать разные подходы архитектуры приложения. Одной из самой популярной и удобной архитектурой является микросервисная архитектура. Система будет разработана на микросервисной архитектуре.

Архитектурный стиль микросервисов используется для создания абсолютно новых приложений, в частности приложений, которые можно разбить на несколько задач или бизнес-возможностей. Большинство приложений могут быть разбиты таким образом, что приложения крупной системы разделены по архитектуре микросервисов. Поскольку задачи могут быть разделены на отдельные микросервисы, основная функциональность этих сервисов может быть легко составлена в соответствии с потребностями приложения. Это очень сильно облегчает повторное использование базового кода и делает код понятным и управляемым.

1 Общее описание

1.1 Перечень терминов и сокращения

В таблице 1 показаны перечень терминов и сокращения, которые использованы в предметно области разрабатываемой системы. Так же тут сформулированы определения терминов используемых технологии при разработке данного приложения.

Таблица 1.

Сокращения и их определения

Термины и сокращения	Определение
API (Application Programming Interface)	Программный интерфейс приложения
HTTP (HyperText Transfer Protocol)	Протокол передачи гипертекста
БД	База данных
JSON (JavaScript Object Notation)	Текстовый формат для обмена данными
SQL (Structured Query Language)	Язык запросов
REST (Representational State Transfer)	Передача состояния представления
СУБД	Система управления базами данных
NoSQL (not only SQL)	Не только SQL
CLI (Command Line Interface)	Интерфейс командной строки
AMQP (Advanced Message Queue Protocol)	Протокол очереди сообщений
SMS (Short Message Service)	Служба коротких сообщений

1.2 Микросервисная архитектуры приложения

Микросервис – это процесс, который отвечает за исполнение конкретно одной независимой задачи. Микросервисы обычно разрабатываются для выполнения определенных бизнес-функций. Хотя микросервисы создаются и существуют независимо друг от друга, в конечном итоге они составляются вместе, чтобы обеспечить общую функциональность приложения. Каждый микросервис часто не подчиняется полнослойному архитектурному стилю. Поскольку микросервисы состоят из других микросервисов и не всегда предназначены для конечных пользователей, уровни представления и приложений могут не всегда присутствовать. Однако обычно каждый микросервис контролирует и управляет своими собственными данными. Каждый микросервис имеет четко определенный интерфейс или API, который информирует другие микросервисы о том, как его можно использовать и обмениваться данными. Связь обычно осуществляется через стандарты и протоколы, такие как HTTP и XML. Интерфейсы REST используются для

поддержания связи между микросервисами без сохранения состояния. Желательно, чтобы каждый запрос-ответ был независимым от любого другого запроса-ответа.

1.2.1 Использование микросервисов

Микросервисная архитектура может быть использован для создания совершенно различных новых приложений, в частности приложений, которые могут быть разбиты на набор задач или бизнес-возможностей. Так как большинство приложений можно разбить таким образом, особенно крупные приложения, микросервисы имеют широкий диапазон применимости. Поскольку задачи могут быть разделены на отдельные независимые микросервисы, функциональность этих сервисов может быть легко перекомпозирована в соответствии с потребностями системы. Это очень облегчает повторное использование базового кода и делает его управляемым и понятным.

Архитектурный стиль микросервиса имеет одно исключение в своей широкой применимости, которая предназначена для более старых монолитных приложений, использующих предварительно скомпилированные двоичные файлы. Код для таких приложений становится жестким и хрупким, потому что легко вносить ошибки при внесении небольших или простых изменений. Модернизация такой кодовой базы может потребовать, чтобы большая часть приложения была обернута.

Микросервисы могут быть локальными, удаленными или какой-то их комбинацией. При использовании микросервисов важно учитывать объем связи, который потребуется между микросервисами в приложении. Обмен сообщениями между микросервисами требует дополнительных затрат независимо от того, какие стандарты связи и протоколы (HTTP, XML и т.д.) используются. Связь между микросервисами также не имеет состояния. Однако в зависимости от приложения может быть желательно отслеживать поведение пользователя и его взаимодействие. Веб-приложения могут делать это с помощью файлов cookie, но это потенциально увеличивает объем данных, передаваемых между микросервисами. Накладные расходы на связь должны быть приняты во внимание.

Архитектурный стиль микросервиса позволяет разрабатывать приложения таким образом, чтобы их было легко обновлять и масштабировать. Это помогает предприятиям оставаться актуальными и адаптироваться к быстро меняющемуся технологическому ландшафту.

1.2.2 Преимущества микросервисной архитектуры

Микросервисы – это небольшие автономные сервисы, которые работают вместе. Давайте немного разберем это определение и рассмотрим характеристики. Многие организации обнаружили, что, используя тонкодисперсные микросервисные архитектуры, они могут быстрее поставлять программное обеспечение и использовать новые технологии. Микросервисы дают нам значительно больше свободы реагировать и принимать различные решения, что позволяет нам быстрее реагировать на неизбежные изменения, которые влияют на всех нас.

Небольшой и сосредоточен на том, чтобы хорошо справиться только с одной задачей. Кодовые базы растут по мере того, как мы пишем код для добавления новых функций. Со временем может быть трудно узнать, где нужно внести изменения, потому что база кода очень велика. Несмотря на стремление к четким, модульным монолитным кодовым базам, слишком часто эти произвольные границы в процессе разрушаются. Код, связанный с похожими функциями, начинает распространяться повсеместно, что затрудняет исправление ошибок или реализаций. Микросервисы используют подход к независимым сервисам. Мы фокусируем наши границы обслуживания на границах бизнеса, делая очевидным, где код живет для данной части функциональности. И удерживая этот сервис сфокусированным на явной границе, мы избегаем искушения, чтобы он вырос слишком большим, со всеми сопутствующими трудностями, которые это может привести.

Автономность. Каждый микросервис – это как отдельная сущность. Он может быть развернут как изолированная служба на платформе (PAAS) или это может быть его собственный процесс операционной системы. Мы стараемся избегать упаковки нескольких услуг на одну и ту же машину, хотя определение машины в современном мире довольно туманно! Как мы обсудим позже, хотя эта изоляция может добавить некоторые накладные расходы, результирующая простота делает нашу распределенную систему намного проще для размышления, а новые технологии способны смягчить многие проблемы, связанные с этой формой развертывания. Все коммуникации между самими сервисами осуществляются через сетевые вызовы, чтобы обеспечить разделение между сервисами и избежать опасностей тесной связи.

Сервисы должны иметь возможность меняться независимо друг от друга и развертываться сами по себе, не требуя изменений у потребителей. Нам нужно подумать о том, что должны предоставлять наши сервисы, и что они должны позволять скрывать. Если существует слишком много общего доступа, наши потребительские услуги становятся связаны с нашими внутренними представительствами. Это уменьшает нашу автономию, так как требует дополнительной координации с потребителями при внесении изменений. Сервис предоставляет интерфейс прикладного программирования (API), и сотрудничающие сервисы взаимодействуют с нами через эти API. Нам также нужно подумать о том, какая технология подходит для того, чтобы сама по себе она не соединяла потребителей. Это может означать выбор независимых от

технологий API, чтобы гарантировать, что мы не ограничиваемся в выборе технологий.

Разнородность технологии. С системой, состоящей из нескольких сотрудничающих сервисов, мы можем решить использовать различные технологии внутри каждой из них. Это позволяет нам выбрать правильный инструмент для каждой работы, а не выбирать более стандартизированный, универсальный подход, который часто оказывается самым низким общим знаменателем.

Если какая-то часть нашей системы нуждается в улучшении своей производительности, мы можем решить использовать другой технологический стек, который лучше подходит для достижения требуемых уровней производительности. Мы также можем решить, что способ хранения наших данных должен измениться для различных частей нашей системы. Например, для социальной сети мы могли бы хранить взаимодействия наших пользователей в графически ориентированной базе данных, чтобы отразить сильно взаимосвязанную природу социального графа, но, возможно, сообщения, которые делают пользователи, могут храниться в документально ориентированном хранилище данных, что приводит к гетерогенной архитектуре, подобной той, что показана на рис.1.1.

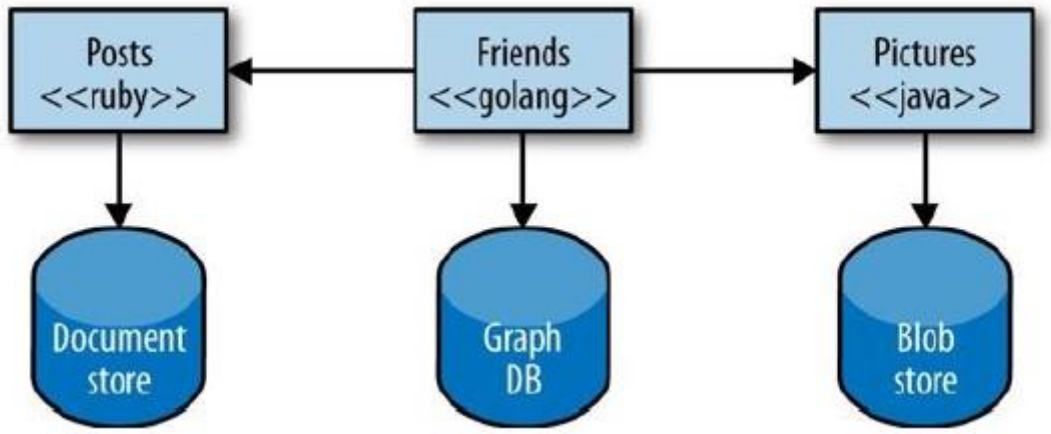


Рисунок 1.1. Микросервисы могут использовать различные технологии

С помощью микросервисов мы также можем быстрее внедрять технологии и понять, как новые достижения могут нам помочь. Одним из самых больших препятствий на пути опробования и внедрения новой технологии являются связанные с ней риски. С монолитным приложением, если я хочу попробовать новый язык программирования, базу данных или фреймворк, любое изменение повлияет на большую часть моей системы. С системой, состоящей из нескольких сервисов, у меня есть несколько новых мест, в которых можно попробовать новую технологию. Я могу выбрать услугу, которая, возможно, является самым низким риском, и использовать технологию там, зная, что я могу ограничить любое потенциальное негативное воздействие. Многие организации считают,

что эта способность быстрее осваивать новые технологии является для них реальным преимуществом.

Устойчивость. Ключевой концепцией в разработке устойчивости является перегородка. Если один из компонентов системы выходит из строя, но этот сбой не происходит каскадно. Вы можете изолировать проблему, и остальная часть системы сможет продолжить работу. Границы обслуживания становятся вашими очевидными перегородками. В монолитном сервисе, если сервис выходит из строя, все перестает работать. С монолитной системой мы можем работать на нескольких машинах, чтобы уменьшить вероятность сбоя, но работая с микросервисами, мы можем создавать системы, способные справляться с общими сбоями в обслуживании.

Однако нужно быть осторожными. Чтобы гарантировать, что наши микросервисные системы смогут должным образом использовать эту улучшенную устойчивость, нам необходимо понять новые источники сбоев, с которыми приходится иметь дело в распределенных системах. Сети могут и будут отказывать, как и машины. Мы должны знать, как справиться с этим, и какое влияние (если таковое имеется) это должно оказать на конечного пользователя нашего программного обеспечения.

Масштабируемость. В большом монолитном сервисе мы должны масштабировать все вместе сразу. Одна небольшая часть нашей общей системы ограничена в производительности, но если это поведение тормозится в гигантском монолитном приложении, мы должны масштабировать всю систему. С помощью небольших микросервисов мы можем просто масштабировать те сервисы, которые нуждаются в масштабировании, что позволяет нам запускать другие части системы на менее мощном оборудовании, как показано на рисунке 1.2.

При использовании систем предоставления по требованию, подобных тем, которые предоставляются Amazon Web Services, мы можем даже применять это масштабирование по требованию для тех компонентов, которые в этом нуждаются. Это позволяет нам более эффективно контролировать наши расходы. Не часто архитектурный подход может быть настолько тесно связан с почти немедленной экономией средств.

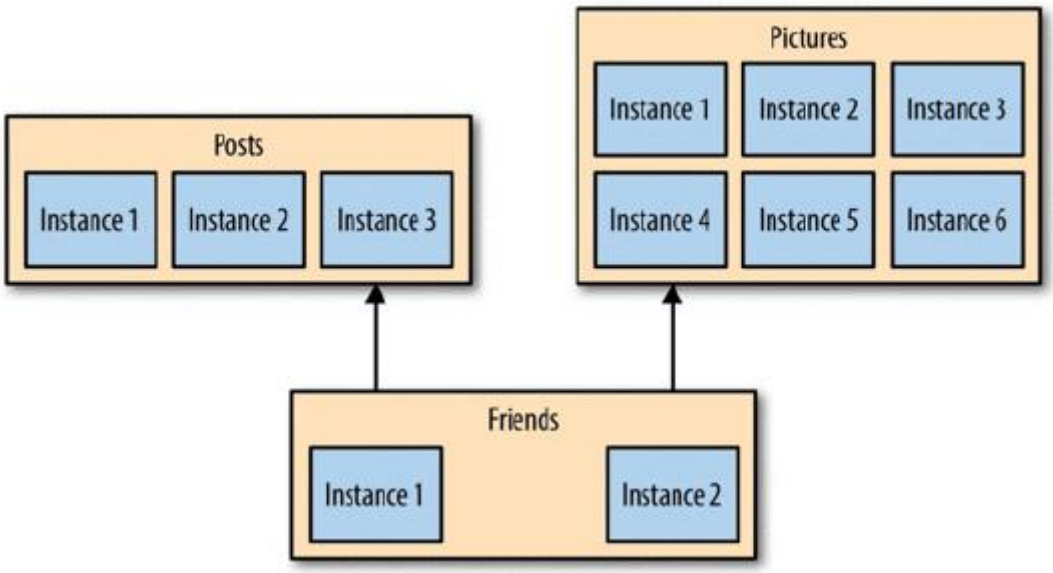


Рисунок 1.2. Масштабирование микросервисов

Простота развертывания. Для изменения одной строки монолитного приложения длиной в миллион строк необходимо развернуть все приложение, чтобы залить изменения в продакшн. Это может быть большое рискованное развертывание. На практике масштабные развертывания с высоким уровнем риска в конечном итоге из-за понятных опасений происходят нечасто. К сожалению, это означает, что наши изменения накапливаются и накапливаются между выпусками до тех пор, пока новая версия нашего приложения, выходящая в производство, не будет иметь массы изменений. И чем больше разница между выпусками, тем выше риск, что мы что-то не так сделаем!

При использовании микросервисов мы можем внести изменения в один сервис и развернуть его независимо от остальной системы. Это позволяет нам быстрее развернуть наш код. Если проблема все же возникает, ее можно быстро изолировать от отдельной службы, что облегчает быстрый откат. Это также означает, что мы можем быстрее донести нашу новую функциональность до клиентов. Это одна из основных причин, по которой такие организации, как Amazon и Netflix, используют эти архитектуры - чтобы обеспечить устранение как можно большего количества препятствий на пути к выпуску программного обеспечения.

1.3 Технологии и инструменты для разработки

1.3.1 Spring Framework

До появления Enterprise Java Beans (EJB) разработчики Java должны были использовать JavaBeans для создания веб-приложений. Хотя JavaBeans помогли в разработке компонентов пользовательского интерфейса (UI), они не смогли

предоставить такие услуги, как управление транзакциями и безопасность, которые были необходимы для разработки надежных и безопасных корпоративных приложений. Появление EJB рассматривалось как решение этой проблемы. EJB расширяет компоненты Java, такие как веб-компоненты и компоненты предприятия, и предоставляет сервисы, которые помогают в разработке корпоративных приложений. Однако разработка корпоративного приложения с EJB была непростой задачей, так как разработчику нужно было выполнять различные задачи, такие как создание интерфейсов Home и Remote и реализация методов обратного вызова жизненного цикла, которые приводят к сложности предоставления кода для EJB. Из-за этого усложнения разработчики начали искать более простой способ разработки корпоративных приложений.

Spring Framework появилась как решение всех этих сложностей. В этой платформе используются различные новые методы, такие как Аспектно-Ориентированное программирование (AOP), Plain Old Java Objecy (POJO) и внедрение зависимостей (DI), для разработки корпоративных приложений, тем самым удаляя сложности, возникающие при разработке корпоративных приложений с использованием EJB. Spring Framework – это легковесная платформа с открытым исходным кодом, позволяющая разработчикам Java создавать простые, надежные и масштабируемые корпоративные приложения. Эта структура в основном ориентирована на предоставление различных способов помочь вам управлять вашими бизнес-объектами. Это значительно облегчило разработку веб-приложений по сравнению с классическими средами Java и интерфейсами прикладного программирования (API), такими как подключение к базе данных Java (JDBC), страницы JavaServer Pages (JSP) и Java Servlets.

Spring framework можно рассматривать как набор фреймворков, также называемых под проектами, таких как Spring AOP. Spring Object-Relational Mapping (Spring ORM). Spring Web Flow и Spring Web MVC. Вы можете использовать любой из этих модулей отдельно при создании веб-приложения. Модули также могут быть сгруппированы вместе, чтобы обеспечить лучшие функциональные возможности в веб-приложении.

1.3.1.1 Особенности Spring Framework

Особенности Spring framework, такие как IoC, AOP и управление транзакциями, делают его уникальным среди всего списка фреймворков. Некоторые из важных особенностей Spring Framework:

– контейнер IoC: Относится к базовому контейнеру, который использует шаблон DI или IoC для неявного предоставления ссылки на объект в классе во время выполнения. Этот шаблон действует как альтернатива шаблону поиска службы. Контейнер IoC содержит код ассемблера, который обрабатывает управление конфигурацией объектов приложения. Spring Framework предоставляет два пакета, а именно org.springframework.beans и

org.springframework.context, которые помогают обеспечить функциональность контейнера IoC;

– Data Access Layer: Позволяет разработчикам использовать API персистентности, такие как JDBC и Hibernate, для хранения данных персистентности в базе данных. Он помогает в решении различных проблем разработчика, таких как взаимодействие с подключением к базе данных, как убедиться, что соединение закрыто, как бороться с исключениями и как реализовать управление транзакциями он также позволяет разработчикам легко писать код для доступа к данным персистентности во всем приложении;

– Spring MVC Framework: Позволяет создавать веб-приложения на основе архитектуры MVC. Все запросы, сделанные пользователем, сначала проходят через контроллер, а затем отправляются в разные представления, то есть на разные страницы JSP или сервлеты. Функции обработки и проверки форм в среде Spring MVC могут быть легко интегрированы со всеми популярными шаблонизаторами, такими как ThymeLeaf, FreeMarker и Mustache;

– управление транзакциями: Помогает в обработке управления транзакциями приложения, не затрагивая его код. Эта платформа предоставляет Java Transaction API (JTA) для глобальных транзакций, управляемых сервером приложений, и локальных транзакций, управляемых с помощью JDBC Hibernate, Java Data Objects (JDO) или других API доступа к данным. Это позволяет разработчику моделировать широкий спектр транзакций на основе декларативного и программного управления транзакциями Spring.

1.3.1.2 Spring Boot

У Spring Boot много преимуществ, включая зависимости от стартера и автоконфигурацию. Исполбзуя Spring Boot на протяжении разработки приложения можно избежать явной формы конфигурации, если это не является абсолютно необходимым. Но в дополнение к стартовым зависимостям и автоматической настройке Spring Boot также предлагает несколько других полезных функций:

– Actuator обеспечивает мониторинг среды выполнения о внутренней работе приложения, включая метрики, сведения о дампе потока, работоспособность приложения и свойства среды, доступные приложению;

– гибкая спецификация свойств окружающей среды;

– дополнительная поддержка тестирования в дополнение к помощи тестирования, уже существующей в базовой структуре.

Более того, Spring Boot предлагает альтернативную модель программирования на основе скриптов Groovy, которая называется Spring Boot CLI (интерфейс командной строки). С помощью Spring Boot CLI вы можете писать целые приложения в виде набора скриптов Groovy и запускать их из командной строки. Spring Boot стал такой неотъемлемой частью развития Spring.

Невозможно представить бысрую разработку приложения на Spring без Spring Boot.

1.3.2 RabbitMQ

RabbitMQ – это брокер обмена сообщениями с открытым исходным кодом, который реализует протокол AMQP. В последние несколько лет его популярность растет. Первоначально используемые наиболее смелыми компаниями, многие теперь открывают для себя не только особые достоинства RabbitMQ, но и положительное влияние использования обмена сообщениями в разработке программного обеспечения. Действительно, с появлением облачных вычислений потребность в архитектуре и создании систем, которые одновременно масштабируются и постепенно ухудшаются, стала более насущной. Выбирая слабосвязанные архитектуры, связанные сообщением, передаваемым через таких брокеров, как RabbitMQ, инженеры-программисты смогли удовлетворить потребности современной разработки приложений.

1.3.2.1 Брокеры сообщений и очередь сообщений

В последнее время программные системы значительно эволюционировали. Приложения должны взаимодействовать с другими приложениями, эти приложения могут быть внутренними и внешними по отношению к самому приложению. Для одного и того же приложения у нас могут быть разные типы клиентов, такие как браузеры, мобильные клиенты и так далее. Следовательно, нам абсолютно необходим коммуникационный уровень между внутренними приложениями и между приложениями и клиентами. Нам нужно доставлять разные сообщения различным приложениям или клиентам. Доставка сообщений может быть узким местом, если уровень связи не масштабируется. Поиск масштабируемых систем для коммуникационного уровня приводит нас к брокерам сообщений и очередям сообщений. Давайте теперь обсудим, что такое брокеры сообщений и очереди сообщений.

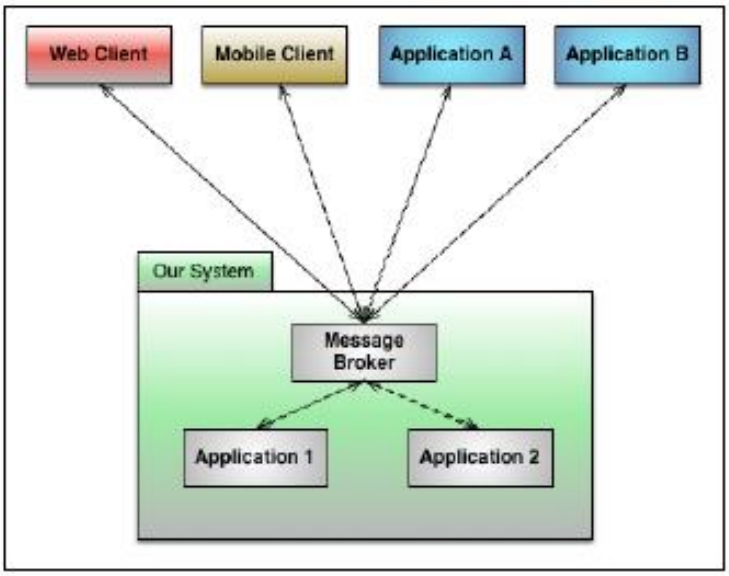


Рисунок 1.3. Брокер сообщений

Брокер сообщений – это архитектурный шаблон, который может принимать сообщения из нескольких назначений, определять правильное назначение и направлять сообщение по правильному маршруту. Брокеры сообщений позволяют системам работать с сообщениями и маршрутизацией, опосредуя связь между компонентами. Как только приложения реализуют шаблон брокера сообщений, он уменьшает связь между компонентами приложения.

Брокеры сообщений централизованы в архитектурном смысле, чтобы контролировать и управлять всеми сообщениями. Таким образом, все входящие и исходящие сообщения отправляются через брокеров сообщений, которые анализируют и доставляют сообщения в нужное место назначения.

1.3.2.2 Очереди сообщений

Очередь сообщений – это, вкратце, очередь для обмена сообщениями. Очередь-это базовая структура данных, лежащая в основе функционирования очереди сообщений. Операции очереди сообщений аналогичны операциям структуры данных очереди, таким как операции enqueue и dequeue. Операция enqueue приводит к добавлению элемента в заднюю часть очереди. Операция dequeue приводит к удалению элемента из передней части очереди. Очереди сообщений обеспечивают параллельные и асинхронные операции для масштабирования приложений. В очереди сообщений сообщения ждут до тех пор, пока другое принимающее приложение не получит сообщение.

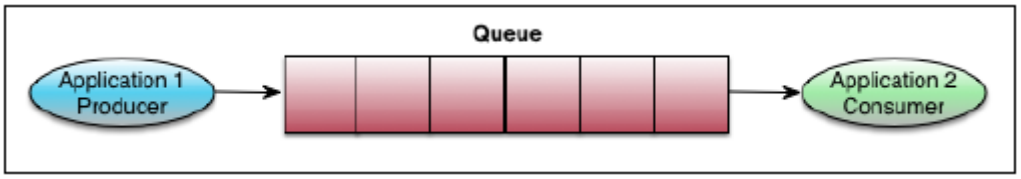


Рисунок 1.4. Очередь сообщений

1.3.2.3 Протокол AMQP

Спецификации очереди сообщений определяются различными типами стандартов и протоколов. RabbitMQ использует расширенный протокол очереди сообщений (AMQP), который определяет политику очередей сообщений.

Джон О'Хара из Дж. П. Моргана основал протокол AMQP в 2003 году. Он вложил в него невероятное количество работы. Затем Дж. П. Морган обратился к другим фирмам с просьбой создать организацию по созданию открытых стандартов в сфере обмена сообщениями. Согласно официальному сайту AMOP (http://www.amqp.org), AMQP является открытым стандартом для передачи сообщений между приложениями или организациями. Таким образом, AMQP просто определяет свойства обмена сообщениями, свойства очереди, как сообщения маршрутизируются между приложениями и клиентами, как броке сообщений гарантируют, что сообщение будет получено или отправлено, А также определяет возможности такие как надежность и безопасность.

Функциональная совместимость и надежность очень важны для современных задач разработки программного обеспечения. Сила AMQP заключается в его функциональных возможностях, таких как функциональная совместимость, надежность и т.д. В частности, с возможностью взаимодействия мы можем использовать разные типы технологий в отправителе и получателе. Основной проблемой для большинства интернет-гигантов является масштабируемость. Масштабируемость имеет прямое отношение к надежности.

Представим некоторые терминологии из мира RabbitMQ, которые мы будем часто использовать:

– обмены (exchanges) – это конечные точки сервера RabbitMQ, к которым клиенты будут подключаться и отправлять сообщения. Каждая конечная точка идентифицируется уникальным ключом;

– очереди (queues) – это компоненты сервера RabbitMQ, которые буферизуют сообщения, поступающие с одного или нескольких обменов, и отправляют их соответствующим получателям сообщений. Сообщения в очереди также могут быть выгружены в постоянное хранилище (такие очереди также называются длительными очередями), что обеспечивает более высокую степень надежности в случае сбоя сервера обмена сообщениями. Как только сервер снова запускается, сообщения из постоянного хранилища помещаются обратно в соответствующие очереди для передачи получателям. Каждая очередь идентифицируется уникальным ключом;

– привязки (bindings) – это логическая связь между обменами и очередями. Каждая привязка - это правило, которое определяет, как обмены должны направлять сообщения в очереди. Привязка может иметь ключ маршрутизации, который может использоваться клиентами для указания семантики маршрутизации сообщения;

– виртуальные хосты (virtual hosts) – логические блоки, которые разделяют компоненты сервера RabbitMQ (такие как обмены, очереди и пользователи) на отдельные группы для лучшего администрирования и контроля доступа. Каждое клиентское соединение AMQP привязано к конкретному виртуальному хосту.

Протокол AMQP позволяет клиенту установить одностороннюю логическую связь для отправки сообщений для обмена. Каждая логическая ссылка представляет собой отдельный канал AMQP, который может предоставлять дополнительные параметры для надежной передачи сообщений. В связи с этим, одно клиентское TCP-соединение с сервером RabbitMQ позволяет использовать несколько каналов связи AMQP. Поскольку AMQP не обеспечивает возможность получения списка очередей, обменов или привязок, которые определены в брокере сообщений RabbitMQ, клиентские приложения должны указывать имя обмена, имена очередей и, необязательно, информацию о маршрутизации посредством ключей маршрутизации для конкретные привязки. AMQP - это программный протокол, который позволяет его клиентам создавать и удалять обмены, очереди и привязки при необходимости. RabbitMQ устраняет некоторые ограничения AMQP, предоставляя пользовательские расширения помимо того факта, что сам протокол AMQP является расширяемым. Для того, чтобы разработать приложение, RabbitMQ предоставляет несколько типов обмена из коробки, а именно:

– Direct exchange – это доставляет сообщение на основе ключа маршрутизации, предоставленного в заголовке сообщения (привязки уже должны быть определены между прямым обменом и очередью). Существует предварительно созданный direct exchange с этим именем .amq.direct. Специализированный тип обмена, называемый обменом по умолчанию с пустой строкой в качестве имени обмена, также предварительно создается в брокере сообщений. Он имеет специальное свойство, в котором ключ привязки, указанный клиентом, должен совпадать с именем очереди, в которую направляется сообщение;

– Fanout exchange – это обеспечивает доставку сообщения во все очереди, связанные с обменом. Он может быть использован для создания широковещательного механизма доставки сообщений в очереди. Существует заранее созданный fanout exchange с именем .amq.fanout;

– Topic exchange – это обеспечивает доставку сообщения в очереди на основе фильтра маршрутизации, заданного между тематическим обменом и очередями; его можно использовать для создания механизма многоадресной рассылки для доставки сообщений. Существует предварительно созданная topic exchange с именем .amq.topic;

– Headers exchange – это может использоваться для доставки сообщений в очереди на основе других атрибутов заголовка сообщения (а не ключа маршрутизации). Существует два предварительно созданных обмена заголовками с именами .amq.headers и .amq.match.

Получатели могут либо подписаться на очередь, чтобы получать сообщения (также называемые обменом данными в push-режиме), либо запрашивать сообщения по запросу из очереди (также называемые обменом в стиле push-сообщений).

1.3.2.4 Преимущества и возможности RabbitMQ

Брокер сообщений RabbitMQ предоставляет ряд функций и инструментов, поддерживающих развертывание, управление и настройку экземпляров сервера RabbitMQ на уровне производства, как показано ниже:

– поддержка нескольких протоколов. Помимо AMQP, RabbitMQ обеспечивает поддержку протоколов STOMP, MQTT и HTTP с помощью плагинов RabbitMQ;

– возможности маршрутизации. Как мы уже видели, мы можем реализовать правила для маршрутизации сообщений между обменами и очередями посредством привязок. Кроме того, можно определить больше пользовательских типов обмена, которые могут предоставить дополнительные возможности маршрутизации;

– поддержка нескольких языков программирования. Существует множество поддерживаемых клиентов для большого разнообразия языков программирования;

– надежная доставка. Это механизм, который гарантирует успешную доставку сообщений посредством подтверждений. Его можно включить между производителем и брокером, а также между брокером и потребителем;

– кластеризация. Это обеспечивает механизм для реализации масштабируемых приложений в терминах брокера сообщений RabbitMQ. Есть альтернативный механизм для реализации масштабируемых приложений с RabbitMQ посредством передачи сообщений между обменником и очередями в разных экземплярах брокера без необходимости создания кластера RabbitMQ;

– высокая доступность. Это гарантирует, что в случае сбоя брокера связь будет перенаправлена на другой экземпляр брокера. Он реализуется посредством зеркального отображения очередей. Сообщения из очереди на экземпляре главного брокера копируются в очередь на экземпляре подчиненного брокера и, как только сообщение подтверждено, сообщения отбрасываются как из главного, так и из подчиненного экземпляров;

– управление и мониторинг. На сервере RabbitMQ есть ряд утилит, обеспечивающих эти возможности;

– подключаемая архитектура. RabbitMQ предоставляет механизм для расширения функциональных возможностей с помощью дополнительных плагинов RabbitMQ.

1.3.3 Redis

Redis – это удаленная база данных в памяти, которая обеспечивает высокую производительность, репликацию и уникальную модель данных для создания платформы для решения проблем. Поддерживая пять различных типов структур данных, Redis вмещает в себя широкий спектр проблем, которые могут быть естественным образом сопоставлены с тем, что предлагает Redis, что позволяет решать ваши проблемы без необходимости выполнять концептуальную гимнастику, требуемую другими базами данных. Дополнительные функции, такие как репликация, персистентность и сегментирование на стороне клиента, позволяют Redis масштабироваться от удобного способа прототипирования системы до сотен гигабайт данных и миллионов запросов в секунду.

Давайте возьмем базу данных контактов клиентов одной компаний. Поиск необходим для поиска контактов по имени, адресу электронной почты, местоположению и номеру телефона. Если использовать базу данных SQL, которая выполняла серию запросов, то эта операция занимала бы 10-15 секунд, чтобы найти совпадения среди 60000 клиентов. Используя Redis, можно построить поисковую систему, которая могла фильтровать и сортировать все эти поля и многое другое, возвращая ответы в течение 50 миллисекунд. Благодаря этому производительность системы повысилась в 200 раз.

Когда говорят, что Redis – это база данных, это только частичная правда. Redis – это очень быстрая нереляционная база данных, которая хранит сопоставление Ключей с пятью различными типами значений. Redis поддерживает постоянное хранение в памяти на диске, репликацию для масштабирования производительности чтения и сегментирование на стороне клиента для масштабирования производительности записи.

Если вы знакомы с реляционными базами данных, то наверняка написали SQL-запросы для связи данных между таблицами. Redis – это тип базы данных, который обычно называют NoSQL или нереляционным. В Redis нет таблиц, и нет никакого определенного базой данных или принудительного способа связывания данных в Redis с другими данными в Redis.

Redis часто сравнивают с memcached, который является очень высокопроизводительным сервером кэширования ключ-значение. Redis также может хранить сопоставление ключей со значениями и даже может достигать таких же уровней производительности, что и memcached. Но сходство быстро заканчивается – Redis поддерживает автоматическую запись данных на диск двумя различными способами и может хранить данные в четырех структурах в дополнение к простым строковым ключам, как это делает memcached. Эти и

другие различия позволяют Redis решать более широкий круг проблем и позволяют использовать Redis либо в качестве основной базы данных, либо в качестве вспомогательной базы данных с другими системами хранения.
Redis обычно используется как для основного, так и для дополнительного носителя данных, поддерживая различные варианты использования и шаблоны запросов. Вообще говоря, многие пользователи Redis предпочитают хранить данные в Redis только тогда, когда необходима производительность или функциональность Redis. Используя другое реляционное или нереляционное хранилище данных для данных, где допустима более низкая производительность или когда данные слишком велики для размещения в памяти.

При использовании базы данных в памяти, такой как Redis, один из первых вопросов, который задается: что происходит, когда мой сервер отключается? Redis имеет две различные формы персистентности, доступные для записи данных из памяти на диск в компактном формате. Первый метод – это дамп в момент времени, когда выполняются определенные условия (количество операций записи за определенный период) или когда вызывается одна из двух команд дампа на диск. Другой метод использует файл только для добавления, который записывает каждую команду, которая изменяет данные в Redis на диск, как это происходит. В зависимости от того, насколько тщательно вы хотите работать со своими данными, вы можете настроить так, чтобы она никогда не синхронизировалась, синхронизировалась один раз в секунду или синхронизировалась по завершении каждой операции.

Несмотря на то, что Redis может работать хорошо, из-за его конструкции в памяти есть ситуации, когда вам может понадобиться Redis для обработки большего количества запросов чтения, чем может обработать один сервер Redis. Для поддержки более высоких скоростей производительности чтения (наряду с обработкой отказа при сбое сервера, на котором работает Redis) Redis поддерживает репликацию master / slave, где ведомые устройства подключаются к ведущему устройству и получают начальную копию полной базы данных. Поскольку записи выполняются на ведущем устройстве, они отправляются всем подключенным ведомым устройствам для обновления подчиненных наборов данных в режиме реального времени. С постоянно обновляемыми данными о ведомых устройствах клиенты могут затем подключаться к любому ведомому устройству для чтения вместо того, чтобы делать запросы к ведущему устройству.

Одним из распространенных видов использования баз данных является хранение долгосрочных отчетных данных в виде агрегатов в фиксированных временных интервалах. Чтобы собрать эти агрегаты, нередко приходится вставлять строки в таблицу отчетов, а затем сканировать эти строки для сбора агрегатов, которые затем обновляют существующие строки в таблице агрегатов. Строки вставляются потому, что для большинства баз данных вставка строк – это очень быстрая операция (вставка записи в конец файла на диске, в отличие от журнала Redis append-only log). Но обновление существующей строки в таблице происходит довольно медленно (это может привести к случайному

чтению и случайной записи). В Redis вы бы вычисляли агрегаты непосредственно с помощью одной из атомарных команд INCR—случайные записи в данные Redis всегда быстры, потому что данные всегда находятся в памяти, и запросы к Redis не нужно проходить через типичный анализатор запросов.

Используя Redis вместо реляционной или другой базы данных, находящейся в основном на диске, вы можете избежать записи ненужных временных данных, избежать необходимости сканировать и удалять эти временные данные и в конечном итоге повысить производительность. Это оба простых примера, но они демонстрируют, как ваш выбор инструмента может сильно повлиять на то, как вы решаете свои проблемы.

1.3.4 Docker

Docker – открытая платформа для разработки, сборки и запуска приложений. Docker позволяет вам отделить ваши приложения от вашей инфраструктуры, чтобы вы могли быстро развернуть программное обеспечение. С помощью Docker вы можете управлять своей инфраструктурой так же, как вы управляете своими приложениями. Воспользовавшись методологиями Docker для быстрой сборки, тестирования и развертывания кода, вы можете значительно сократить задержку между написанием кода и его выполнением в производстве.

Docker – является прорывной технологией, который меняет способ распространения приложений и разработки. Имея много преимуществ, Docker очень прекрасно подходит для разработки на архитектуре микросервисов. Docker является программой командной строки, фоновый демон и набор удаленных сервисов, которые используют логистический подход к решению распространенных программных проблем и упрощают установку, запуск, публикацию и удаление программного обеспечения. Это достигается с помощью технологии UNIX, называемой контейнерами. Он предназначен для ускорения развертывания приложений с помощью облегченной платформы виртуализации контейнеров, окруженной набором инструментов и рабочих процессов, которые помогают разработчикам легче развертывать и управлять приложениями. Однако подход к архитектуре микросервисов также вносит много новых сложностей и требует от разработчиков приложений определенного уровня зрелости, чтобы уверенно применять архитектурный стиль.

1.3.4.1 Docker образ

Создание образов Docker осуществляется с помощью простого, описательного набора шагов, которые называются инструкциями. Файл инструкции называется Dockerfile, в котором инструкции хранятся декларативно. Когда запускается запрос на сборку, Docker читает Dockerfile для

выполнения инструкций и затем возвращает окончательный образ. Каждая инструкция создает новый слой в результирующем образе.

У Docker есть общедоступный реестр образов с названием Docker Hub, представляющий собой большого множество существующих образов, готовых к использованию. Разработчики могут свободно создавать новые образы и добавлять их в реестр образов или делиться ими в частном хранилище, работающем за межсетевым экраном по необходимости.

1.3.4.2 Docker хорошо подходит для микросервисов

Микросервисная архитектура содержит несколько проблем, которые необходимо решить, таких как более высокий уровень сложности от разработки до запуска в производство, критическое требование автоматизации во всех аспектах, изоляция отказов, проблемы тестирования, и так далее. К счастью, Docker представляет ряд ключевых функционала и инструментов, которые могут помочь в решении данных проблем. Давайте обсудим их.

Контейнеры Docker, естественно, очень круто подходят для архитектуры микросервисов, поскольку каждый из них можно использовать в качестве единицы развертывания для детального размещения сервиса. Так как создание и запуск каждого контейнера срабатывает с помощью скриптов. Со временем уже существует множество инструментов, которые ускоряет автоматизацию контейнеров на каждом этапе жизненного цикла проектировани разработки программного обеспечения.

Каждый Docker контейнер представляет собой изолированную среду, которую может содержать среду выполнения для определенного сервиса. Благодаря широкому спектру платформ, поддерживаемых и применяющих Docker, команда разработчиков сервиса может независимо друг от друга работать над внедрением сервиса с использованием любых технологии или языка программирования, процесса, инструментов, которые им удобней.

Docker содержит приложение и все его зависимости в контейнер, который переносим между разными платформами, включая дистрибутивы Linux. Другие заинтересованные стороны приложения, такие как разработчики, тестировщики, системные администраторы могут по желанию быстро запускать одно и то же приложение на разных виртуальных машинах, локальных ноутбуках, серверах с открытым исходным кодом. Это очень хорошо помогает проводить независимое изолированное тестирование определенного сервиса в микросервисной архитектуре. В Docker каждый контейнер состоит только из тех приложений и зависимостей, которые приложение должно иметь для старта, в идеале ни больше, ни меньше. Затем контейнер запускается в виде изолированного процесса в операционной системе хоста, разделяя ядро с остальными другими контейнерами. Таким образом, когда Docker контейнер размещается в среде виртуальных машин, помимо использования преимуществ использования

ресурсов от использования виртуальных машин, метод контейнеризации даже делает это более портативным и эффективным.

Множество из того, что предлагает Docker, очень удобно позволяет разработчикам гибко максимизировать безопасность базового кода на разных уровнях. При сборке кода программисты могут свободно использовать инструменты тестирования для любой части цикла сборки. Так ка источник для создания Docker образов явно, декларативно описан в файлах (Dockerfile, docker-compose), разработчики могут упростить обработку цепочки поставок образов и иметь возможность принудительно назначать политику правил безопасности по мере необходимости. Кроме того, возможность легко укреплять неизменяемые сервисы, помещая их в Docker контейнеры, добавляет очень надежную страховку для сервисов.

1.3.4.3 Контейнеры и виртуализация

Без Docker разработчик обычно используют аппаратную виртуализацию (также называемую виртуальными машинами) для обеспечения изоляции. Виртуальные машины предоставляют виртуальное оборудование, на котором могут быть установлены операционная система и другие программы. На их создание уходит много времени (часто минуты) и требуются значительные затраты ресурсов, поскольку они запускают целую копию операционной системы в дополнение к программному обеспечению, которое вы хотите использовать.

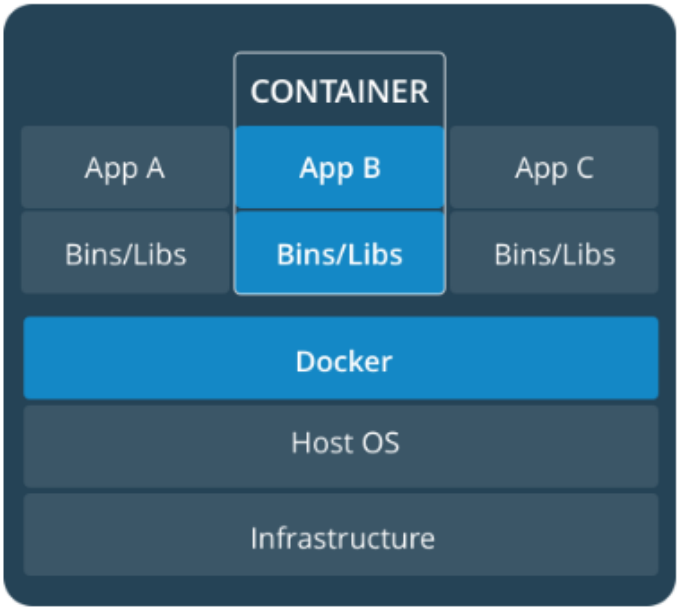


Рисунок 1.5. Архитектура Докер контейнеров

В отличие от виртуальных машин, контейнеры Docker не используют аппаратную виртуализацию. Программы, работающие внутри контейнеров Docker, взаимодействуют непосредственно с ядром Linux хоста. Потому что нет

никакого дополнительного уровня между программой, работающей внутри контейнера, и операционной системой компьютера. Никакие ресурсы не тратятся впустую на запуск избыточного программного обеспечения или имитацию виртуального оборудования. Это очень важное различие. Docker – это не технология виртуализации. Вместо этого он помогает вам использовать контейнерную технологию, уже встроенную в вашу операционную систему. Как уже отмечалось ранее, контейнеры существуют десятилетиями. Docker использует пространства имен Linux и контрольные группы, которые являются частью Linux с 2007 года. Docker не предоставляет контейнерную технологию, но она специально упрощает ее использование.

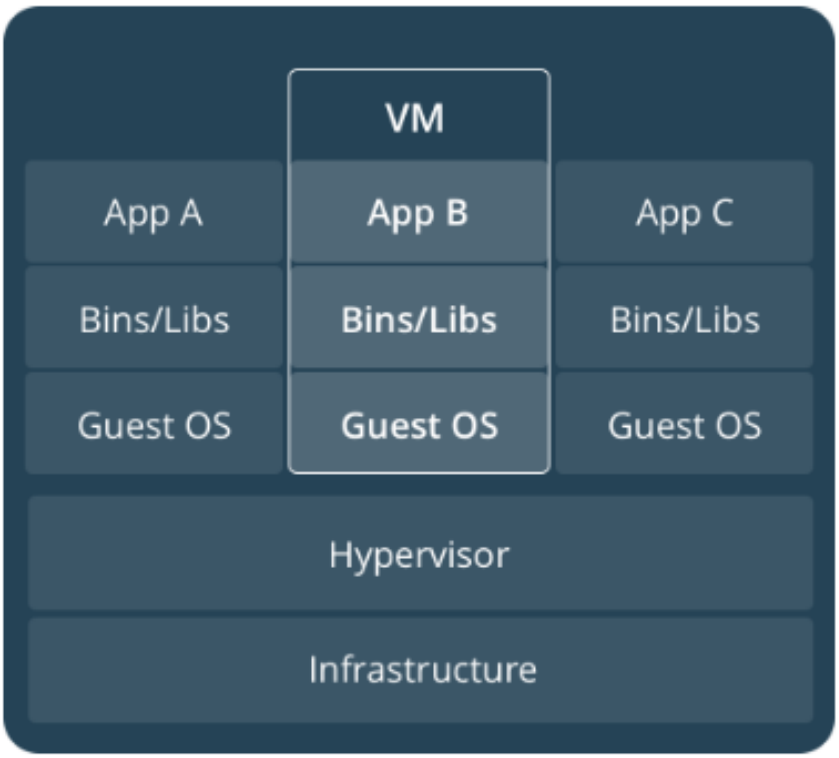


Рисунок 1.6. Архитектура виртуальных машин

Интерфейс командной строки, или CLI, запускается в так называемой пользовательской памяти, как и другие программы, работающие поверх операционной системы. В идеале программы, работающие в пространстве пользователя, не могут изменять память пространства ядра. Вообще говоря, операционная система – это интерфейс между всеми пользовательскими программами и оборудованием, на котором работает компьютер.
На рисунке 5 видно, что запуск Docker означает запуск двух программ в пространстве пользователя. Первый – это демон Docker. При правильной установке этот процесс всегда должен быть запущен. Второй - это Docker CLI. Это программа Docker, с которой пользователи взаимодействуют. Если вы хотите запустить, остановить или установить программное обеспечение, введите

команду с помощью программы Docker. На рисунке 5 показано также запусу трех контейнеров. Каждый из них выполняется как дочерний процесс демона Docker, обернутый контейнером, а процесс делегирования выполняется в своем собственном подпространстве памяти пользовательского пространства. Программы, запущенные внутри контейнера, могут получить доступ только к своей собственной памяти и ресурсам в пределах области действия контейнера.

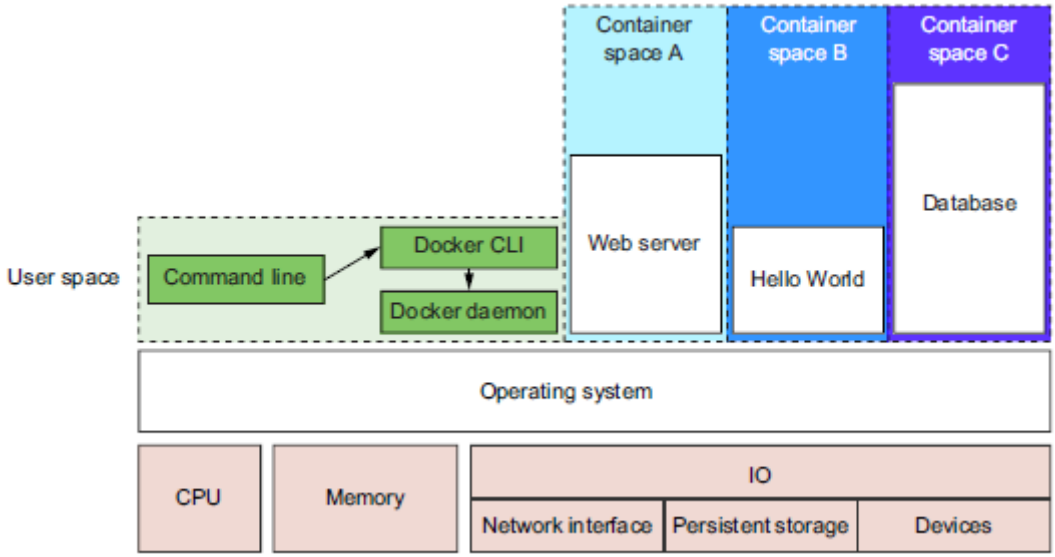


Рисунок 1.7. Запущены три Docker контейнера в системе Linux

2. Основная часть

2.1 Структура проекта

В данной работе разработана система которая предназначена для массовой отправки SMS сообщений. Архитектурой системы является микросервисная. Система разделена систему на следующие микросервисы:

- http-api-service
- routing-service
- sms-sender-service

Все сервисы разработаны с использованием языка программирования Java и его популярный фреймворк Spring Framework (Spring Boot).

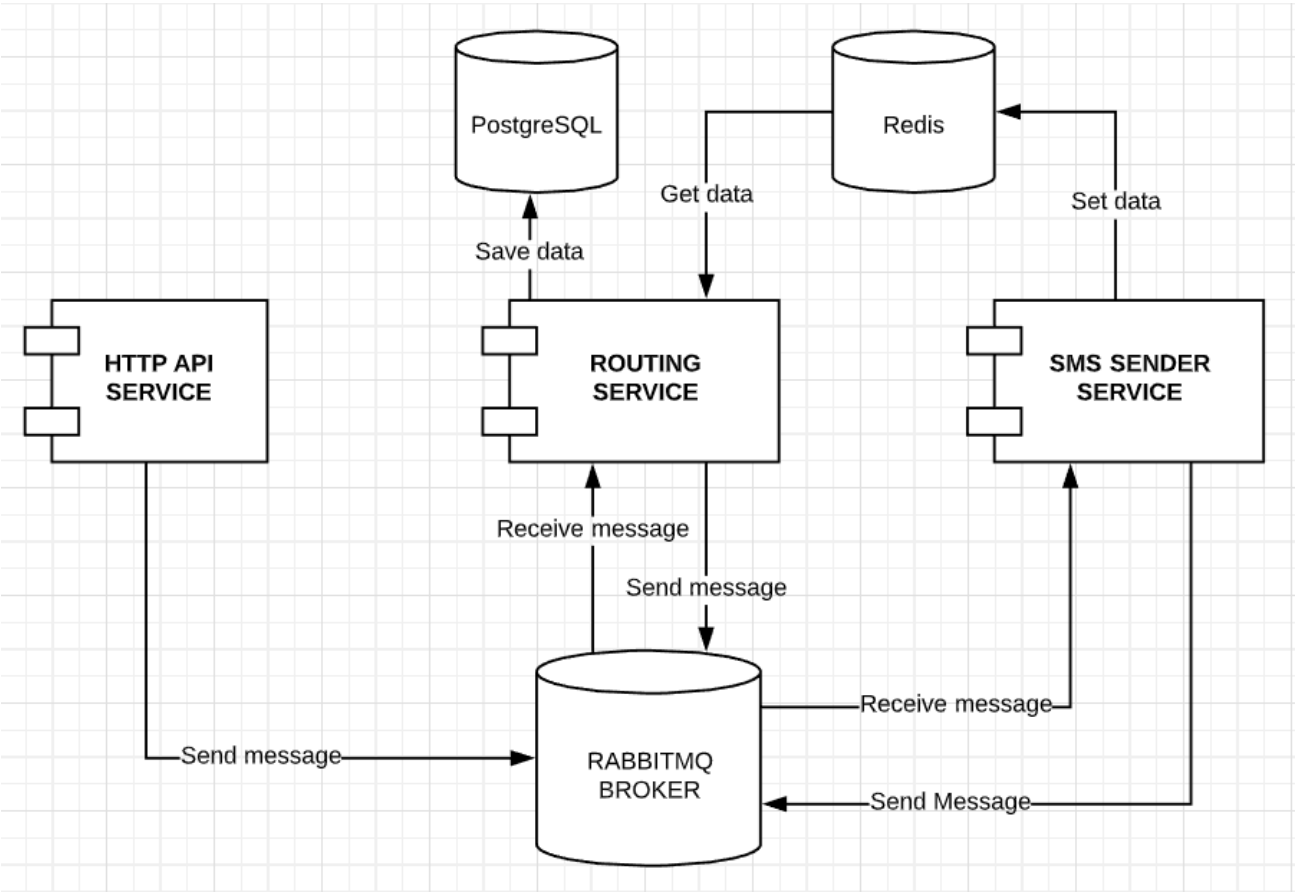


Рисунок 2.1. Структурная схема системы

На рисунке 2.1 можно разглядеть как микросервисы взаимодействуют с друг-другами. Сообщения между микросервисами передаются через брокер сообщений RabbitMQ. Для хранения данных используется СУБД PostgreSQL. Для хранения временных данных используется key-value storage Redis.

На первом этапе перед тем как приступить к разработке нам нужно запустить:

- сервер реляционной базы данных Postgres

– сервер нереляционной базы данных Redis
– сервер брокера обмена сообщениями RabbitMQ

И все эти сервера будут запускаться с помощью Docker.

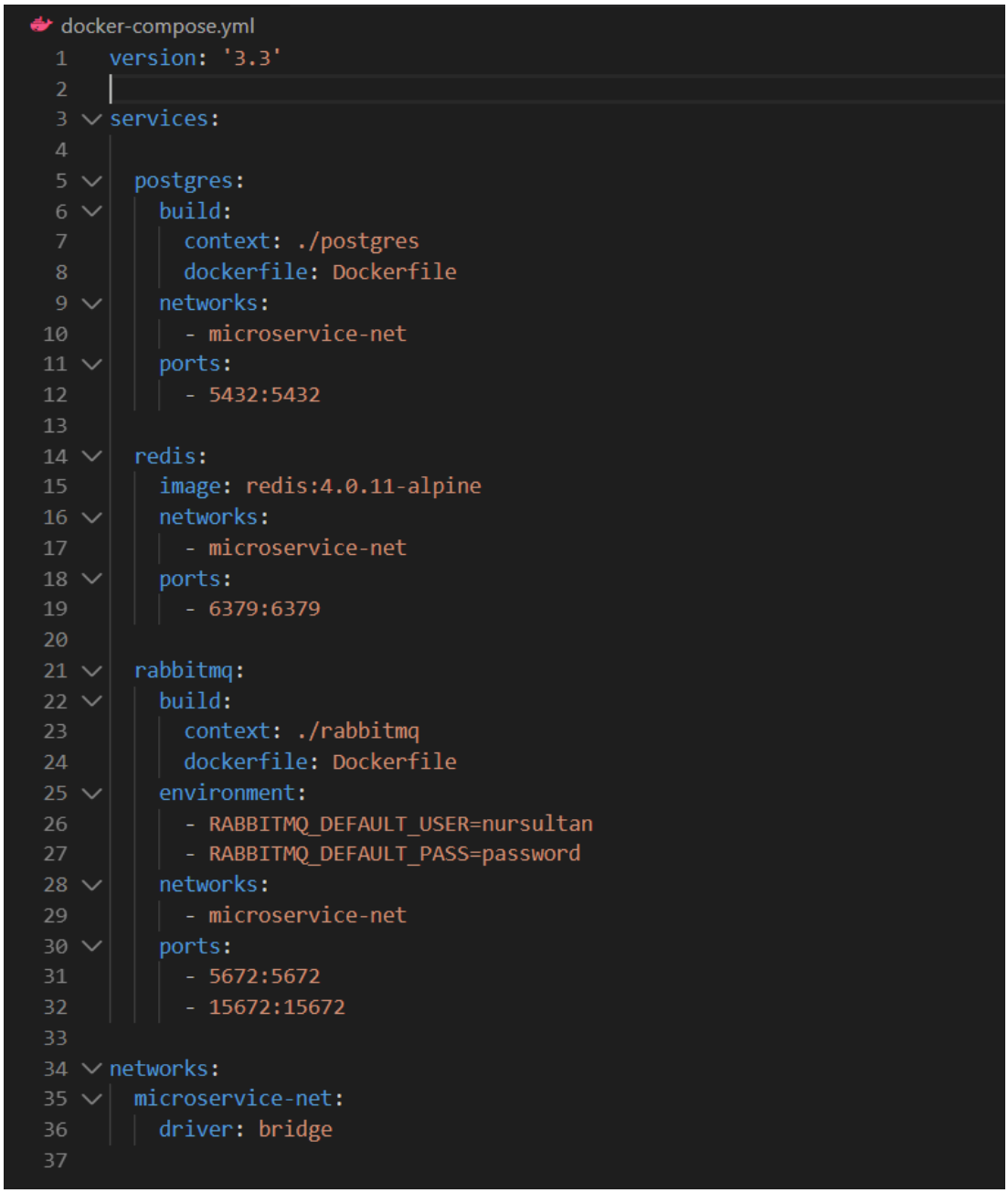


Рисунок 2.2. Файл docker-compose.yml

Docker Compose – это инструмент который входит в состав Docker. В конфигурационном файле docker-compose.yml описаны инструкции для запуска и настройки сервисов. Docker Compose предназначен для одновременного запуска и управления с несколькими контейнерами. В этом инструменте предложены те же возможности, что и Docker. Для работы с Docker Compose нам понадобится всего несколько команд:

– docker ps просмотр запущенных контейнеров

– docker-compose up –d сборка и запуск контейнеров

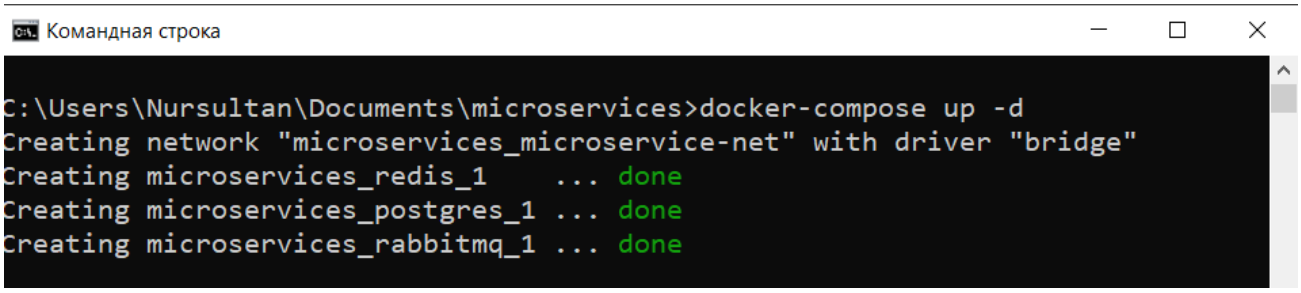


Рисунок 2.3. Запуск контейеров

На командной строке запустили команду docker-compose up –d. В результате собраны три контейнера:

1) microservice_redis_1
2) microservice_postgres_1
3) microservice_rabbitmq_1

```
C:\Users\Nursultan\Documents\microservices>docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
          PORTS
     NAMES
5f673489c392        microservices_rabbitmq   "docker-entrypoint.s…"   21 seconds ago      Up 20 sec
onds       4369/tcp, 5671/tcp, 0.0.0.0:5672->5672/tcp, 15671/tcp, 25672/tcp, 0.0.0.0:15672->15672/t
cp   microservices_rabbitmq_1
08c636950260        redis:4.0.11-alpine      "docker-entrypoint.s…"   21 seconds ago      Up 20 sec
onds       0.0.0.0:6379->6379/tcp
     microservices_redis_1
977ef2e72c35        microservices_postgres   "/entrypoint.sh /run…"   21 seconds ago      Up 20 sec
onds       0.0.0.0:5432->5432/tcp
     microservices_postgres_1
```

Рисунок 2.4 Просмотр запушенных контейнеров

С помощью команды docker ps можно увидеть подробную информацию о контейнерах такие как:

1) container id – идентификатор контейнера
2) image – название Docker образа
3) command – команда которая запущена внутри контейнера
4) created – время создания контейнера
5) status - статус контейнера
6) ports – порты через которую можно достучаться
7) names – названия контейнеров

2.2 Реализация и разработка проекта

Проект состоит из трех микросервисов. Общая бизнес задача проекта разбита на отдельные части. Каждый микросервис имеет свою задачу и кодовую базу. Они должны хорошо справляться с поставленной задачей.

2.2.1 Разработка http-api-service

Http api service – он предназначен для клиентов для того чтобы осуществить SMS отправку через наше приложение. У сервиса есть API который соответсвует архитектурному стилю REST.

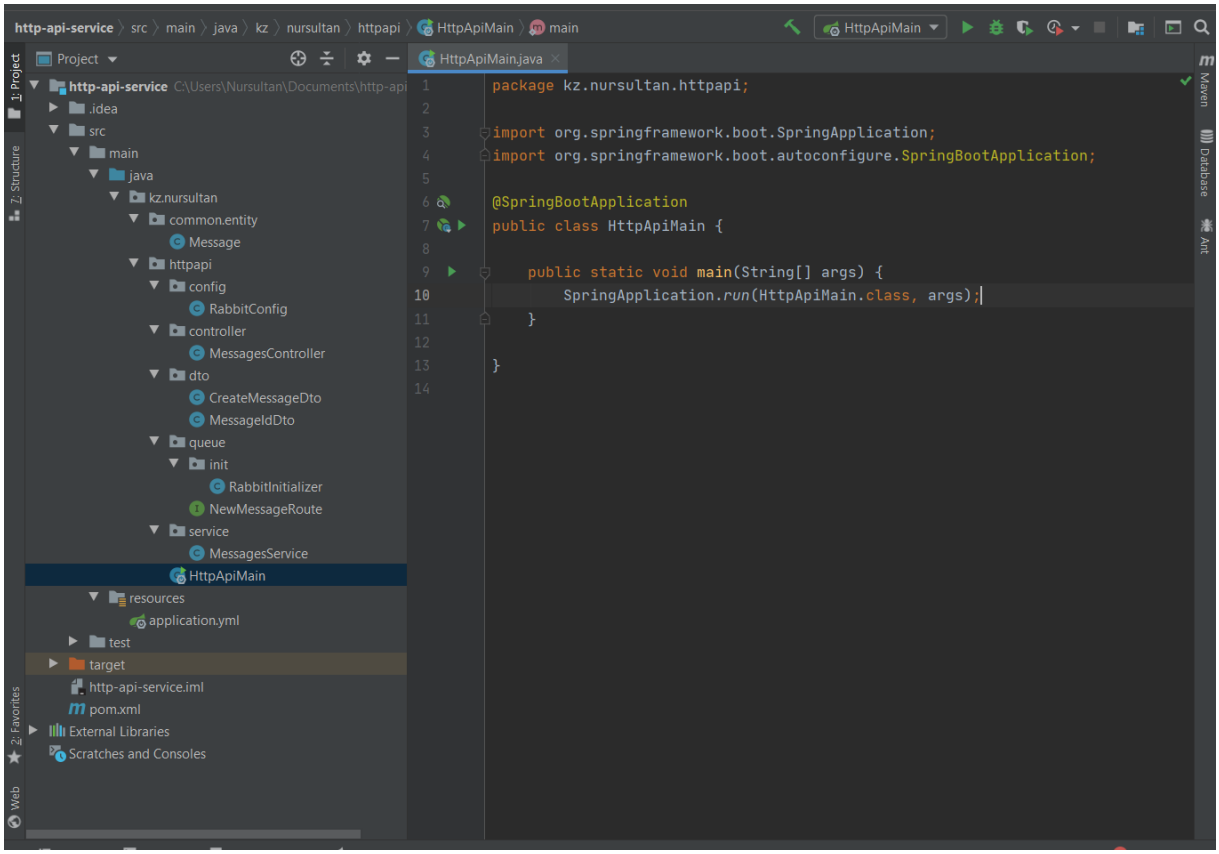


Рисунок 2.5. Структура сервиса http-api-service

Главной задачей этого сервиса является принять SMS сообщение через http запрос. Имеется конечная точка (endpoint) для http запроса по URL адресу http://localhost:8080/messages. Метод данного http запроса является POST. В теле запроса указываем указываем параметры SMS сообщения в формате JSON. Параметры сообщения:

- recipient – номер получатель сообщения
- sender – отправитель сообщения
- message text – содержимое сообщения

Для отправки запроса воспользуемся клиентской программой Postman. Вводим URL адрес, выбираем метод запроса и указываем тело запроса в формате JSON и отпраляем запрос. Пример запроса показано на рисунке 2.7.

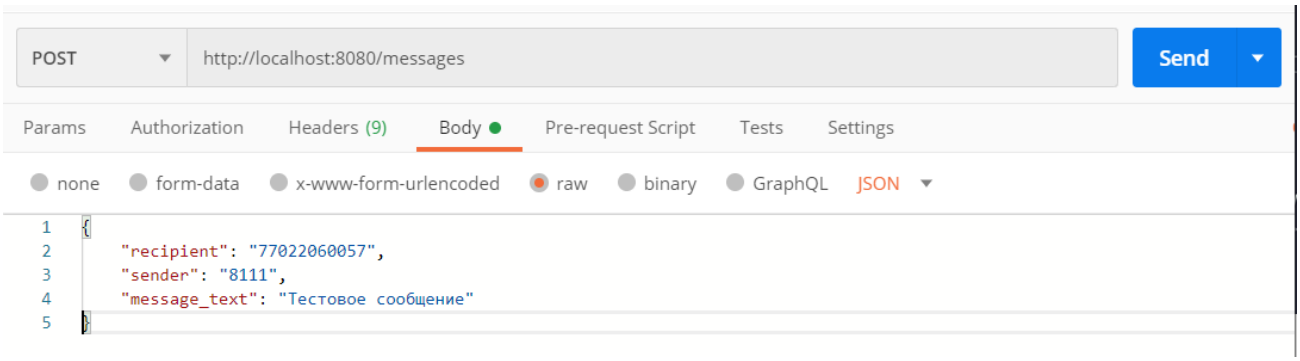


Рисунок 2.6. Отправка запроса на создание сообщения

Для того чтобы принять данный запрос создадим класс контроллер и опишем его код в классе MessagesController как показано на рисунке 2.6.

```
@RestController
@RequestMapping("/messages")
public class MessagesController {

    private MessagesService messagesService;

    public MessagesController(MessagesService messagesService) { this.messagesService = messagesService; }

    @PostMapping
    public ResponseEntity<?> sendMessage(@RequestBody CreateMessageDto messageDto) {
        String messageId = messagesService.createMessage(messageDto);
        MessageIdDto messageIdDto = new MessageIdDto();
        messageIdDto.setMessageId(messageId);
        return ResponseEntity.ok(messageIdDto);
    }

}
```

Рисунок 2.7. Класс MessagesController

В этом классе описано что мы принимаем Post запрос по URI /messages. Он принимает в теле запроса объект класса CreatedMessageDto и отправляет в теле ответа идентификатор созданного сообщения.

Класс CreateMessageDto описывает параметры JSON объекта в теле запроса. Такие параметры как отправитель, получатель и текст сообщения.

```java
@Data
public class CreateMessageDto {
    @JsonProperty("sender")
    private String sender;
    @JsonProperty("recipient")
    private String recipient;
    @JsonProperty("message_text")
    private String messageText;
}
```

Рисунок 2.8. Класс CreateMessageDto

После получения объекта класса CreateMessageDto, оборачиваем в объекта класса Message. Заполняем начальные значения некоторых полей. Этот объект конвертируем в текстовый формат JSON. И затем отправляем это сообщение в брокере обмена сообщениями RabbitMQ. Сообщение попадет в очередь с названием que.msg.new.send. Чтобы отправить сообщение в очередь, очередь нужно создать за ранее.

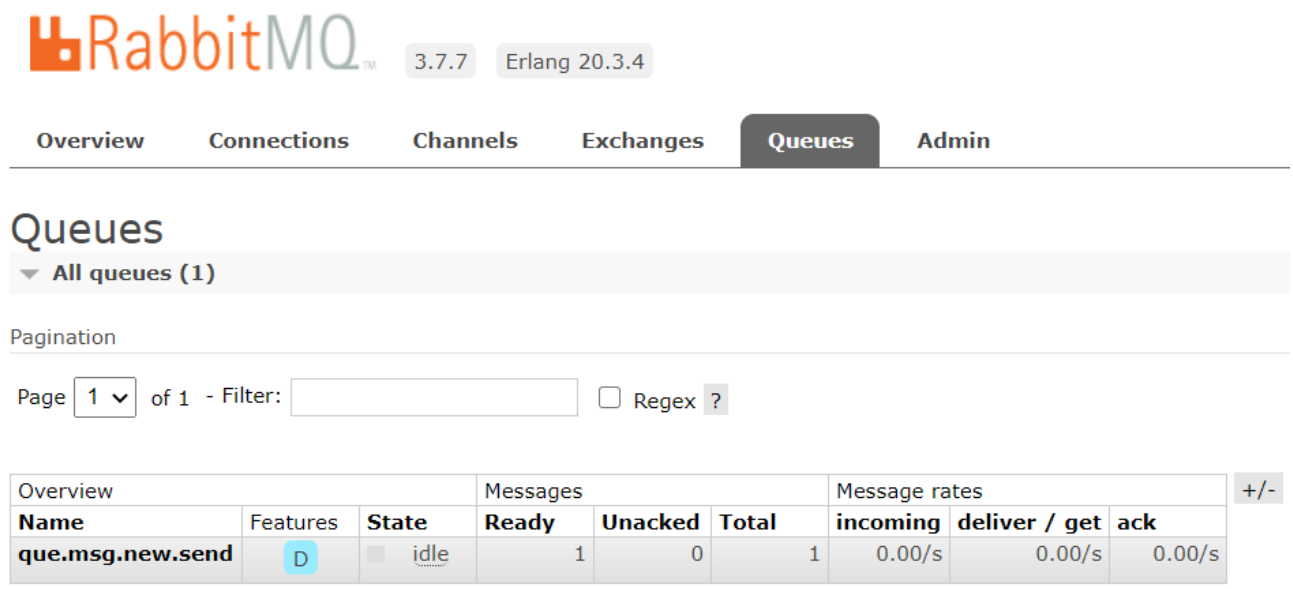


Рисунок 2.9. Очередь que.msg.new.send

Для срабатывания данной логики есть класс сервис MessagesService. У сервиса имеется методы для создания сообщения с начальными значениями и дальше для отправки сообщения в очередь que.msg.new.send. Фрагмент кода приведен на рисунке 2.10

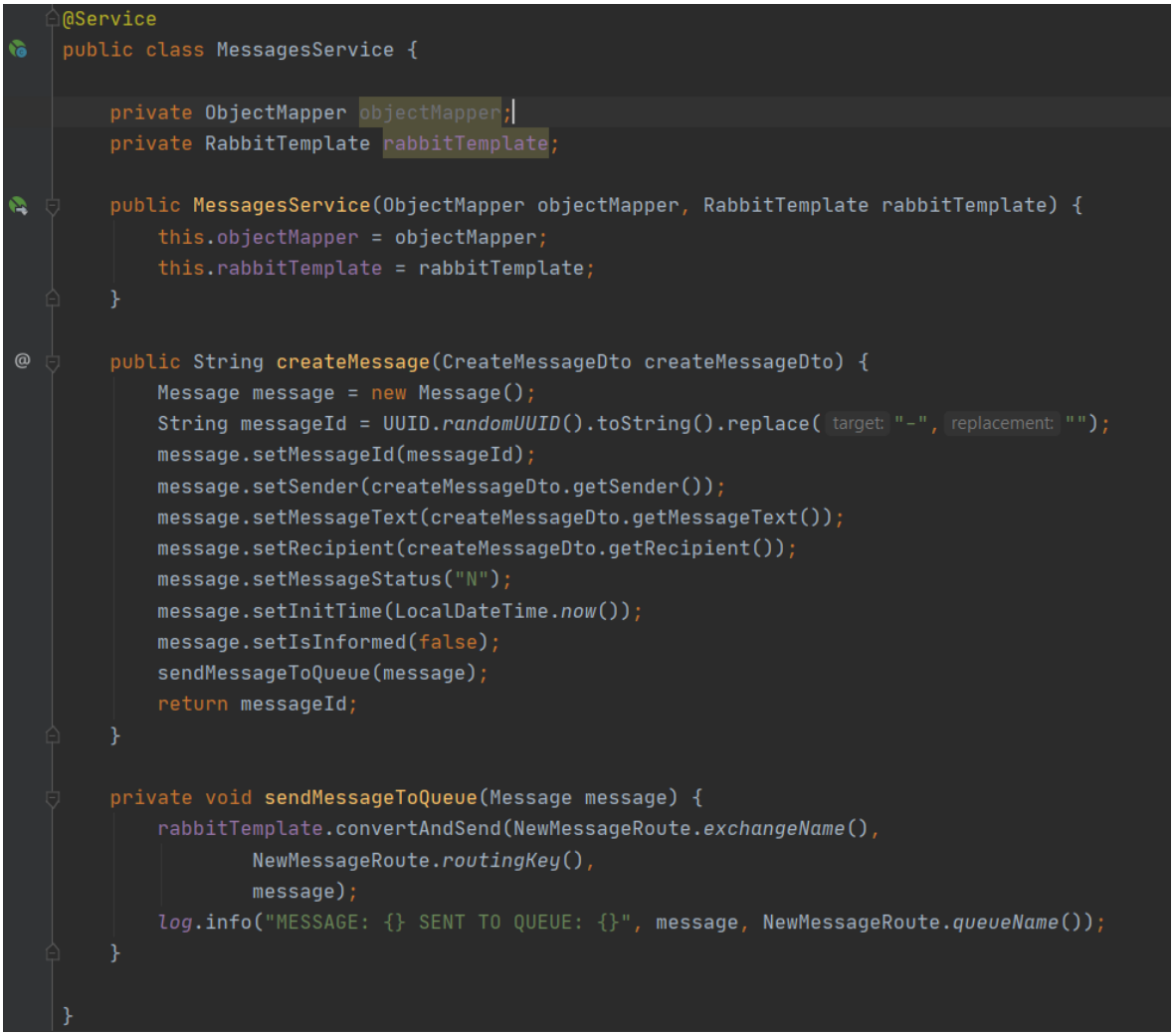


Рисунок 2.10. Методы для отправки сообщения в очередь

После того как сообщение попадет в очередь его сразу же принимает микросервис routing-service. И он его обрабатывает по своей бизнес логике.

2.2.2 Разработка routing-service

Routing service – он преднанзначен для маршрутизации сообщений между сервисами и сохранения данных в базе. В основном он берет сообщения из очереди и ложит их в базу данных. Чтобы отправить сообщения дальше, передает сообщения на SMS sender service через RabbitMQ. Sms sender service отправит сообщение на SMS центр с помощью протокола SMPP. Статусы сообщения обновляются в базе. За это тоже отвечает routing service.

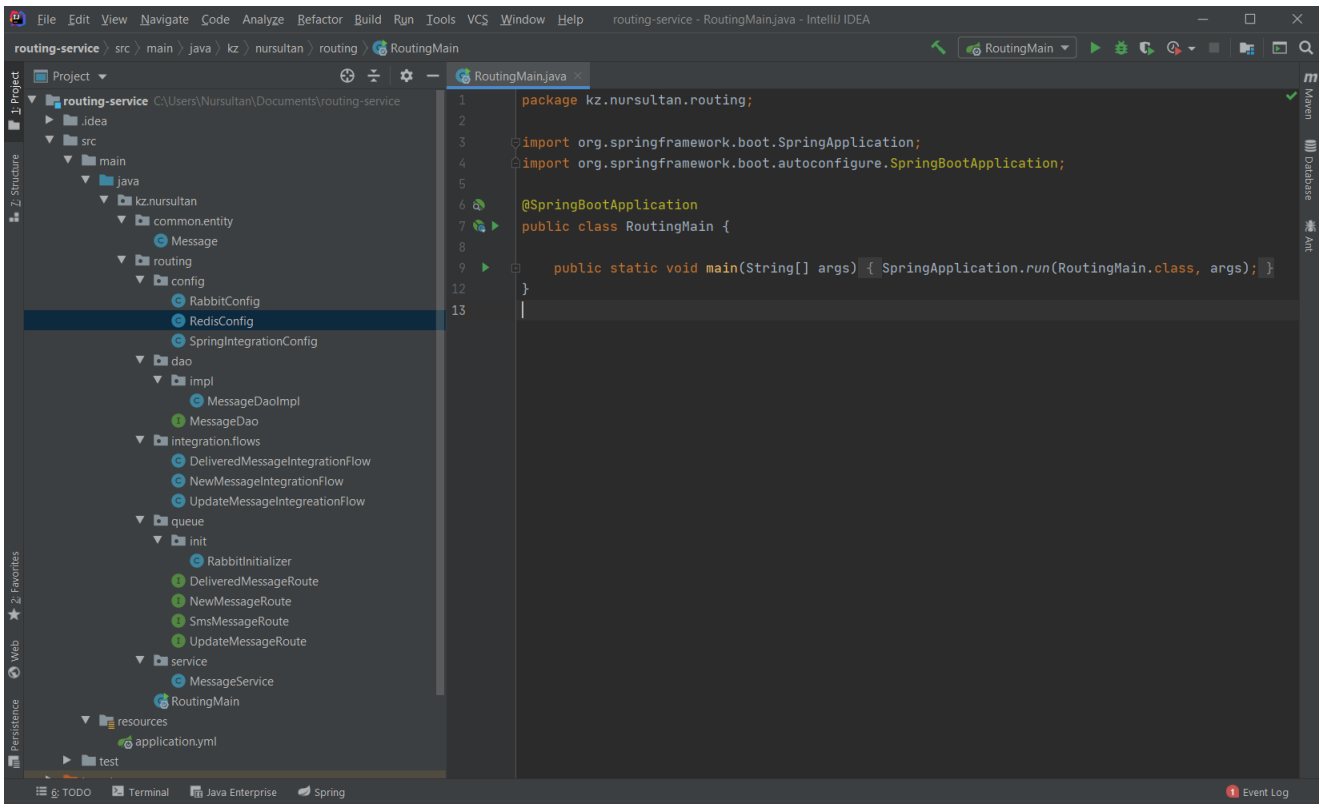


Рисунок 2.11. Структура проекта Routing service

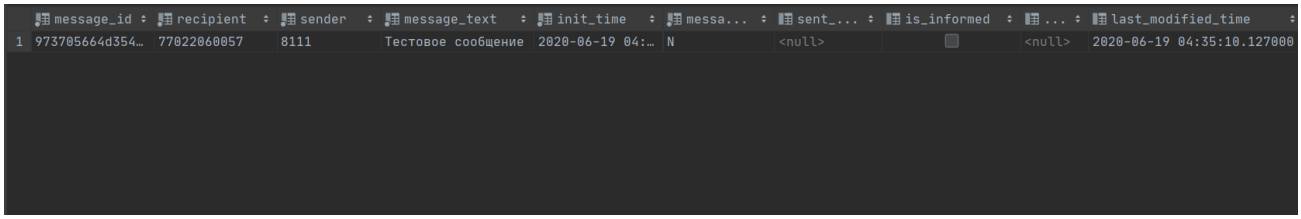
После получения сообщения Routing Service сохраняет данные в базе данных. В качестве СУБД используется PostgreSQL. На текущий момент там содержится всего лишь одна таблица Messages. Таблица имеет следующие столбцы:

1) message_id – идентификатор сообщения
2) recipient – получатель сообщения
3) sender – отправитель сообщения
4) message_text – текст сообщения
5) init_time – время создания сообщения
6) message_status – статус сообщения
7) sent_time – время отрпавки сообщения на SMS центр
8) is_informed – флаг доставки сообщения
9) informed_time – время доставки сообщения
10) last_modified_time – время изменения статуса

Имеется 3 статуса сообщения:

- N (new) – новое сообщение
- S (sent) – сообщение отправлено
- D (delivered) – сообщение доставлено

Routing сервис получает сообщения с брокера сообщения RabbitMQ и сохраняет их в статусе N. Хранение сообщения можно увидеть на рисунке 2.13.



	message_id	recipient	sender	message_text	init_time	messa...	sent_...	is_informed	...	last_modified_time
1	973705664d354…	77022060057	8111	Тестовое сообщение	2020-06-19 04:…	N	<null>	☐	<null>	2020-06-19 04:35:10.127000

Рисунок 2.12. Сообщения в статусе N

Для создания сообщения и сохранения сообщения написано следующий фрагмент кода. Методы данного функционала отображено на рисунке 2.14

```
private static final String CREATE_MESSAGE =
        "insert into nursultan.messages(message_id,          " +
        "                               recipient,           " +
        "                               sender,              " +
        "                               message_text,        " +
        "                               init_time,           " +
        "                               message_status,      " +
        "                               sent_time,           " +
        "                               is_informed,         " +
        "                               informed_time,       " +
        "                               last_modified_time)  " +
        "values(?, ?, ?, ?, ?, ?, ?, ?, ?, ?);              ";

@Override
public void create(Message message) throws SQLException {
    try (Connection connection = dataSource.getConnection();
         PreparedStatement statement = connection.prepareStatement(CREATE_MESSAGE)) {
        statement.setString( parameterIndex: 1, message.getMessageId());
        statement.setString( parameterIndex: 2, message.getRecipient());
        statement.setString( parameterIndex: 3, message.getSender());
        statement.setString( parameterIndex: 4, message.getMessageText());
        statement.setTimestamp( parameterIndex: 5, Timestamp.valueOf(message.getInitTime()));
        statement.setString( parameterIndex: 6, message.getMessageStatus());
        statement.setTimestamp( parameterIndex: 7,
                message.getSentTime() == null ?
                        null : Timestamp.valueOf(message.getSentTime()));
        statement.setBoolean( parameterIndex: 8, message.getIsInformed());
        statement.setTimestamp( parameterIndex: 9, message.getInformedTime() == null ? null : Timestamp.valueOf(message.getInformedTime()));
        statement.setTimestamp( parameterIndex: 10, message.getLastModifiedTime() == null ? null : Timestamp.valueOf(message.getLastModifiedTime()));

        statement.execute();
    }
```

Рисунок 2.13. Метод для сохранения данных в базу

Сервис Routing тесно взаимодействует с Http Api сервис и Sms Sender сервис. Сохраненные сообщения отрпавляются в очередь с названием que.msg.sms.send. Очереди для взаимодействия сервисов можно увидеть на рисунке 2.13. Главной задачей Routing является не только маршрутизация и создания сообщений, но и обновление статусов сообщений. Для обновление статусов сообщений, сервис получает сообщения с очередей que.msg.update.send и que.msg.delivered.send. Сообщения со статусами в очередь приходят с сервиса Sms Sender.

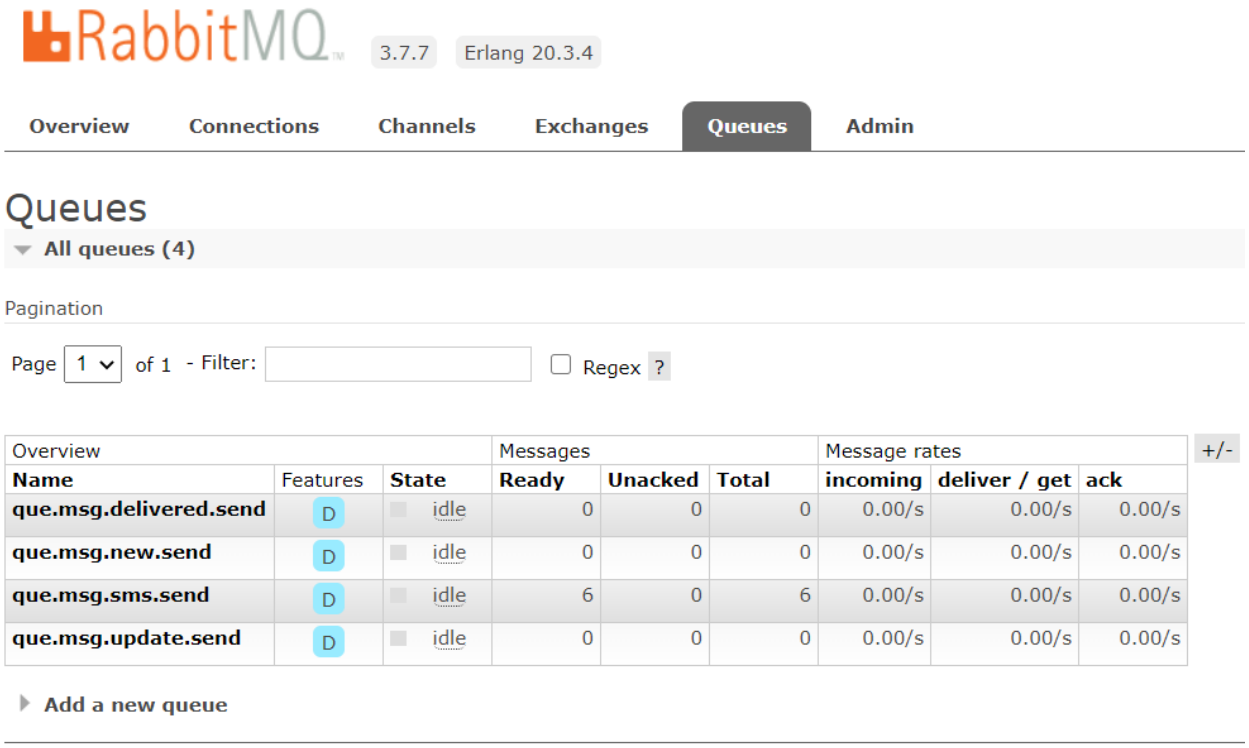
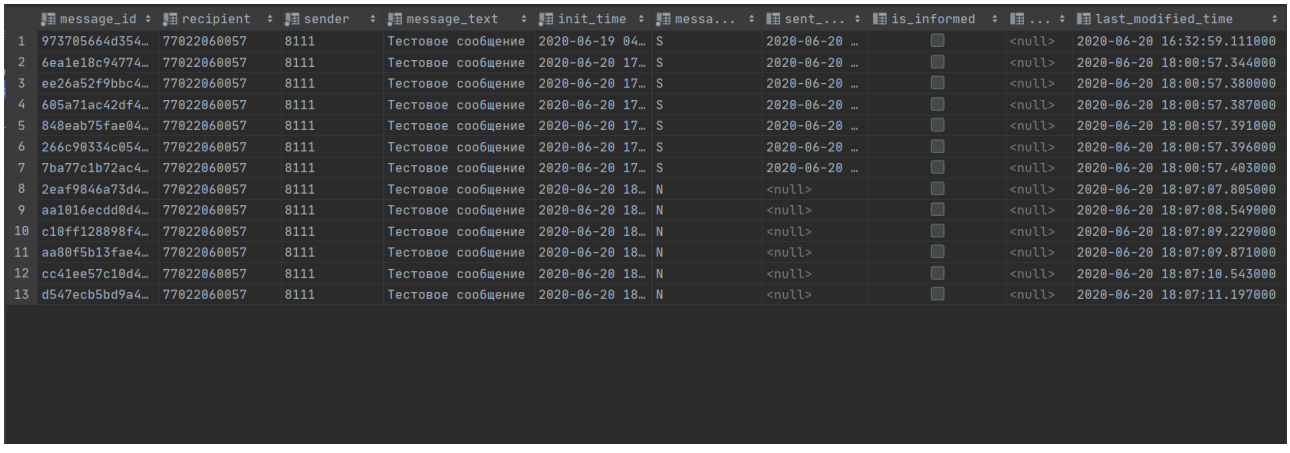


Рисунок 2.14. Сообщения в очереди que.msg.sms.send.

Эти сообщения обрабатаются сервисом Sms Sender. Sms Sender отправят их Sms центру. Статус сообщения изменится на отправлено. Затем сообщения попадут в очередь que.msg.update.send. Сообщения обработаются сервисом Routing. Обновленные сообщения на рисунке 2.14.



	message_id	recipient	sender	message_text	init_time	messa...	sent_...	is_informed	...	last_modified_time
1	973705664d354…	77022060057	8111	Тестовое сообщение	2020-06-19 04…	S	2020-06-20 …		<null>	2020-06-20 16:32:59.111000
2	6ea1e18c94774…	77022060057	8111	Тестовое сообщение	2020-06-20 17…	S	2020-06-20 …		<null>	2020-06-20 18:00:57.344000
3	ee26a52f9bbc4…	77022060057	8111	Тестовое сообщение	2020-06-20 17…	S	2020-06-20 …		<null>	2020-06-20 18:00:57.380000
4	605a71ac42df4…	77022060057	8111	Тестовое сообщение	2020-06-20 17…	S	2020-06-20 …		<null>	2020-06-20 18:00:57.387000
5	848eab75fae04…	77022060057	8111	Тестовое сообщение	2020-06-20 17…	S	2020-06-20 …		<null>	2020-06-20 18:00:57.391000
6	266c90334c054…	77022060057	8111	Тестовое сообщение	2020-06-20 17…	S	2020-06-20 …		<null>	2020-06-20 18:00:57.396000
7	7ba77c1b72ac4…	77022060057	8111	Тестовое сообщение	2020-06-20 17…	S	2020-06-20 …		<null>	2020-06-20 18:00:57.403000
8	2eaf9846a73d4…	77022060057	8111	Тестовое сообщение	2020-06-20 18…	N	<null>		<null>	2020-06-20 18:07:07.805000
9	aa1016ecdd0d4…	77022060057	8111	Тестовое сообщение	2020-06-20 18…	N	<null>		<null>	2020-06-20 18:07:08.549000
10	c10ff128898f4…	77022060057	8111	Тестовое сообщение	2020-06-20 18…	N	<null>		<null>	2020-06-20 18:07:09.229000
11	aa80f5b13fae4…	77022060057	8111	Тестовое сообщение	2020-06-20 18…	N	<null>		<null>	2020-06-20 18:07:09.871000
12	cc41ee57c10d4…	77022060057	8111	Тестовое сообщение	2020-06-20 18…	N	<null>		<null>	2020-06-20 18:07:10.543000
13	d547ecb5bd9a4…	77022060057	8111	Тестовое сообщение	2020-06-20 18…	N	<null>		<null>	2020-06-20 18:07:11.197000

Рисунок 2.15 Статусы сообщений обновлены

Во время обновления данных заполняется поле время отправки (sent_time) сообщения. Значения полей статус (message_status), время последних изменений (last_modified_time) обновляются.

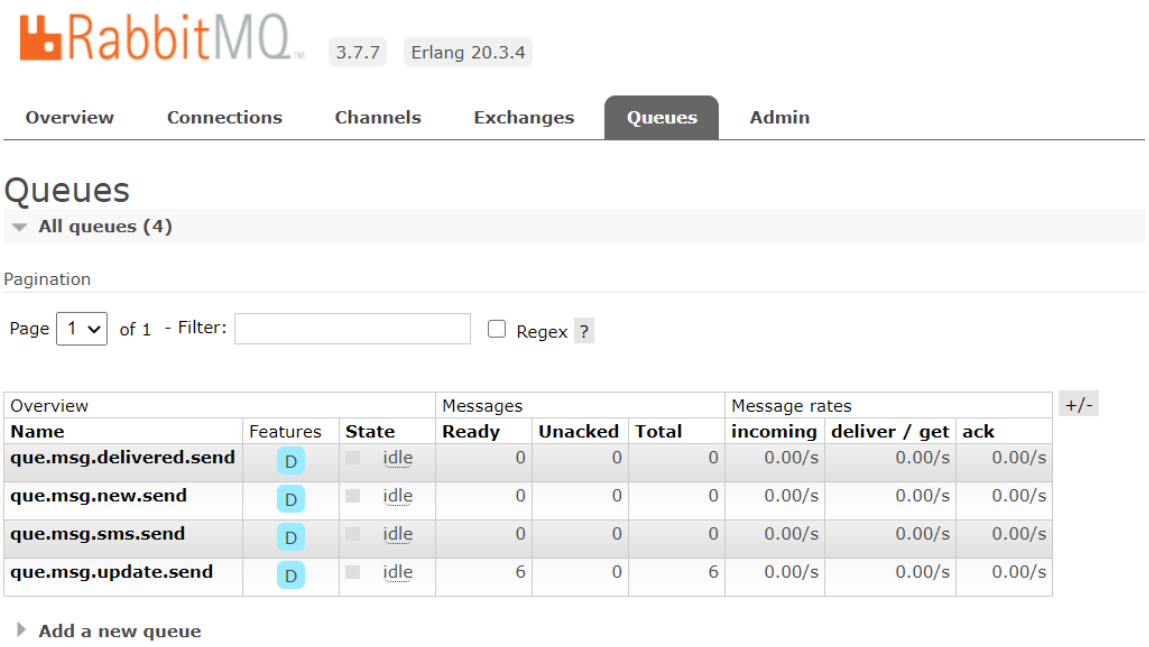


Рисунок 2.16. Сообщения в очереди que.msg.update.send.

Для обновления полей сообщения и срабатывания данной логики написано следующий фрагмент кода. Отображено на рисунке 2.17.

```
    private static final String UPDATE_MESSAGE =
            "update nursultan.messages set message_status = ?,          " +
            "                              sent_time = ?,               " +
            "                              is_informed = ?,             " +
            "                              informed_time = ?,           " +
            "                              last_modified_time = ?       " +
            "                              where message_id = ?         ";

    @Override
    public void update(Message message) throws SQLException {
        try (Connection connection = dataSource.getConnection();
             PreparedStatement statement = connection.prepareStatement(UPDATE_MESSAGE)) {
            statement.setString( parameterIndex: 1, message.getMessageStatus());
            statement.setTimestamp( parameterIndex: 2, message.getSentTime() == null ?
                    null : Timestamp.valueOf(message.getSentTime()));
            statement.setBoolean( parameterIndex: 3, message.getIsInformed());
            statement.setTimestamp( parameterIndex: 4, message.getInformedTime() == null ?
                    null : Timestamp.valueOf(message.getInformedTime()));
            statement.setTimestamp( parameterIndex: 5, Timestamp.valueOf(message.getLastModifiedTime()));
            statement.setString( parameterIndex: 6, message.getMessageId());

            statement.executeUpdate();

        }
    }
```

Рисунок 2.17. Код функционала для обновления

2.2.3 Разработка Sms Sender Service

Sms Sender Service – он предназначен для отправки SMS сообщения на SMS центр или некому провайдеру. SMS центр отвечает за доставку сообщении

получателям или абонентам услуги. Отправка сообщения на SMS центр осуществляется с помощью протокола оправки коротких сообщений SMPP.

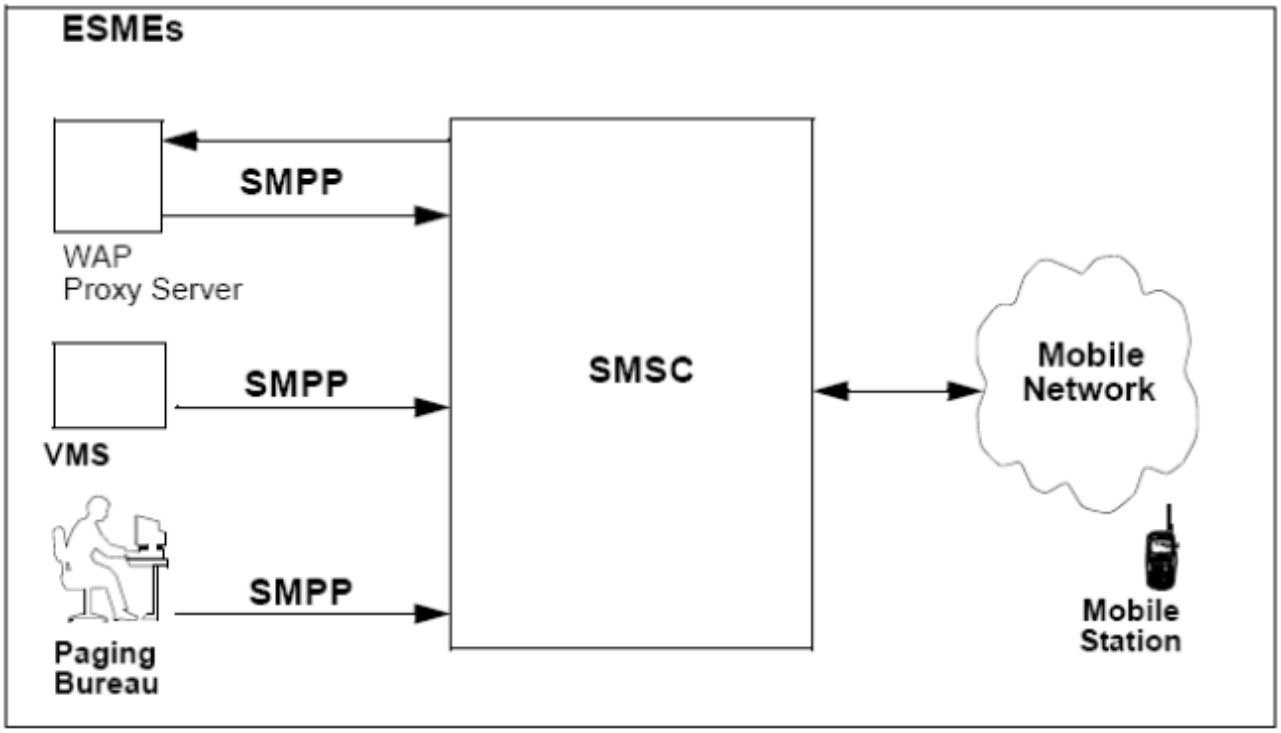


Рисунок 2.18. Схема обмена сообщениями

SMPP – это протокол прикладного уровня одноранговых сообщений. Базируется на обмене PDU (Protocol Data Unit). Построено поверх TCP / IP для передачи SMS сообщений. SMPP работает в режиме постоянного подключения. Используется клиент – серверная архитектура.

Обмен сообщениями через протокол SMPP между отправителем и SMS центром осуществляется тремя способами:

- Transmitter – передача в одну сторону
- Receiver – только прием от SMS центра
- Transreceiver – обмен сообщений между SMS центром и отправителем в оба сторона

PDU операция является парной и она состоит из запрос и ответа. PDU пакеты состоят из двух частей – заголовок и тело запроса.

SMS Sender сервис получает сообщения из очереди que.msg.sms.send. Полученные сообщения отправляется на SMS центр с помощью протокола SMPP. Структура проектного кода можно увидеть на рисунке 2.20.

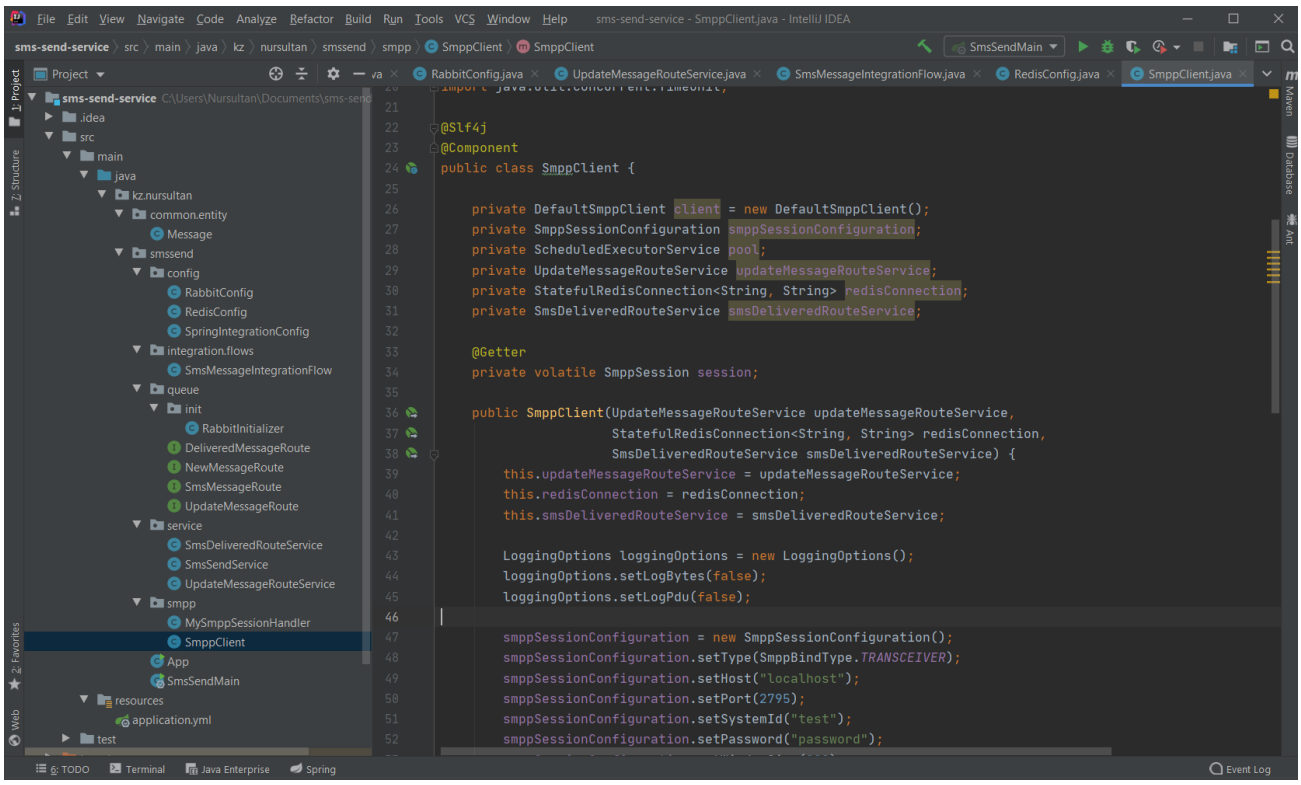


Рисунок 2.19. Структура Sms Sender сервиса

Чтобы отправить короткое сообщение на SMS центр используется SMPP операция PDU Submit_SM. Основные базовые параметры данной операций:

- Message id – идентификатор сообщения
- Source address – отправитель сообщения
- Destination address – получатель сообщения
- Data coding – кодировка сообщения
- Short Message – текст короткого сообщения

В ответ этого сообщения от SMS центра получаем PDU Submit_SM_Resp. Отсюда мы получаем статус сообщения и идентификатор сообщения с SMS центра. Если сообщения доставится нам мы получаем PDU Deliver_SM. Последовательность операции для отправки сообщения отображено на рисунке 2.20.

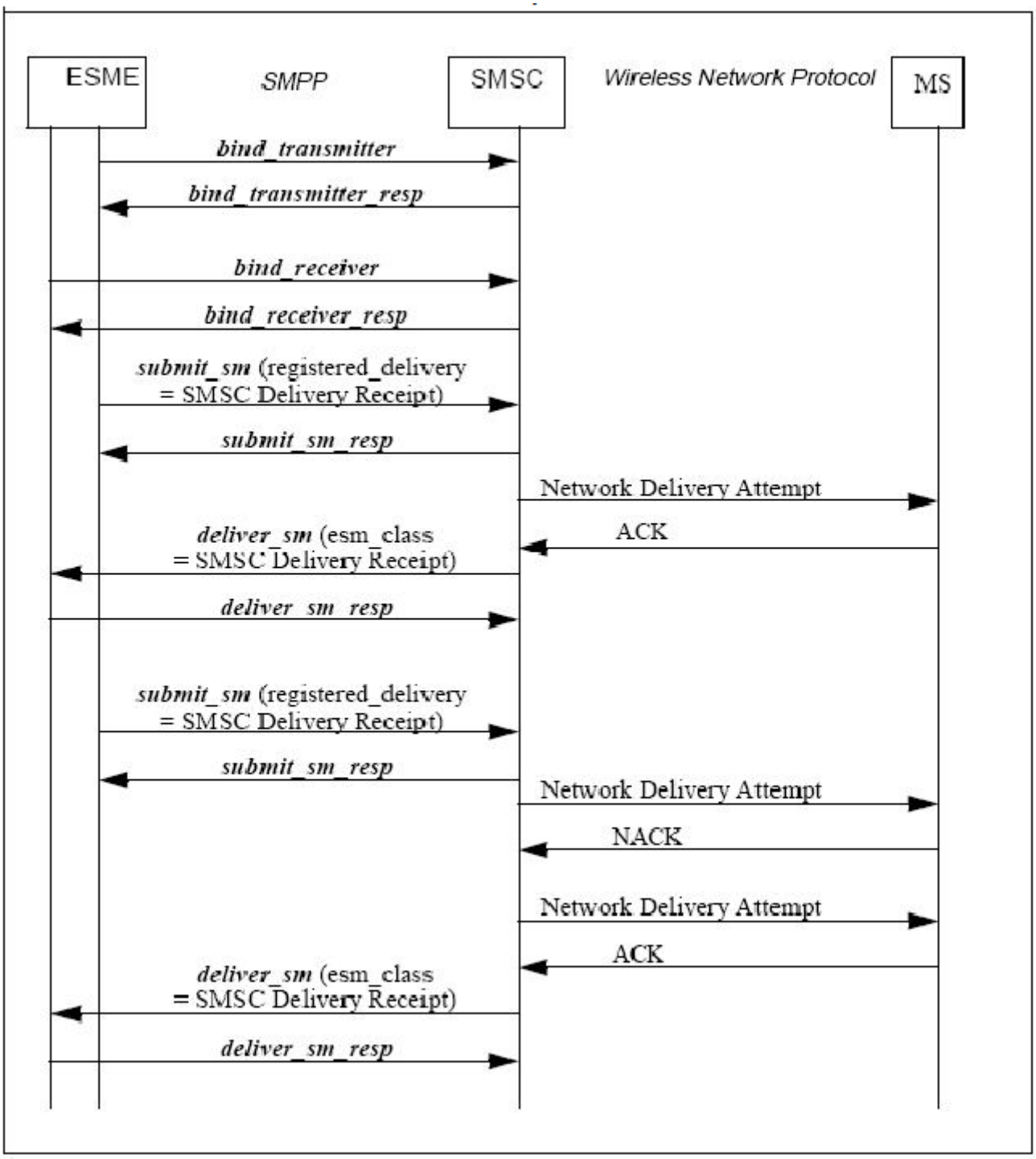


Рисунок 2.20. Последовательность SMPP операции

При доставке SMS сообщения получателю нам приходит отчет о доставки сообщения (Deliver_SM). В теле данной операции можем получить только идентификатор от SMS центра (SMSC ID). Для обновления статуса сообщения в базе данных этого недостаточно. Идентификатор сообщения который сгенерирован нами и идентификатор от SMS центра (SMSC ID) можно получить на теле операции Submit_Sm_Resp. Во время получения ответа Submit_Sm_Resp извлекаем оттуда нужные идентифкаторы и соответственно их временно будем хранить в кэше Redis.

```java
@Override
public void fireExpectedPduResponseReceived(PduAsyncResponse pduAsyncResponse) {
    if (pduAsyncResponse.getResponse().getClass() == SubmitSmResp.class) {
        SubmitSm submitSm = (SubmitSm) pduAsyncResponse.getRequest();
        SubmitSmResp submitSmResp = (SubmitSmResp) pduAsyncResponse.getResponse();

        String smscId = submitSmResp.getMessageId();
        String messageId = (String) submitSm.getReferenceObject();

        redisConnection.sync().hset(REDIS_HASH_KEY, smscId, messageId);
        log.info("TO REDIS ADDED TO HASH:{}, WITH SmscID: {} AND MessageID: {}",
                REDIS_HASH_KEY, smscId, messageId);
        log.info("Got response with MSG ID={}", submitSmResp.getMessageId());
    }
}
```

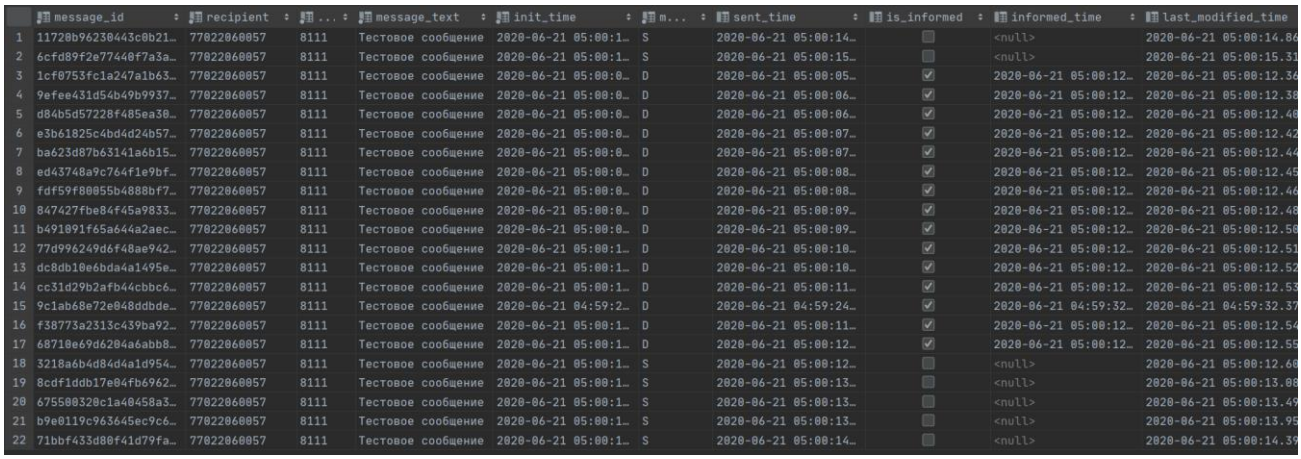
Рисунок 2.21. Код обработки ответа SubmitSMResp

При сохранении идентификаторов (ID) сообщения мы отправляем сообщение в очередь que.msg.delivered.send RabbitMQ. Сервис Routing получает сообщение из очереди. В теле сообщения из очереди получаем SMSC ID. Соответственно с помощью SMS ID извлекаем идентификатор сообщения и обновляем статусы и другие данные в базе данных.

```
127.0.0.1:6379> keys *
1) "message_ids_by_smsc_id"
127.0.0.1:6379> hgetall message_ids_by_smsc_id
1) "976e7117-0a63-4ccc-837f-3a91bbe8d40c"
2) "7b8452c5f5434bc9a8bbdf6ab02262ed"
3) "fb0a5475-095d-46f0-a4c9-2a9296f90b60"
4) "461fd015b16744039d5f52e332323698"
5) "13cf87fe-9c06-4046-b2fa-6f9b297980d4"
6) "a9d8d5682726459daa9a54a3bb5dcf45"
127.0.0.1:6379>
```

Рисунок 2.22. Хранения идентификаторов в кэше

В конечном итоге при доставке сообщения, данные обновляются в базе. Столбцы is_informed, informed time, last_modified_time и message_status обновились. Пример можно увидеть на рисунке 2.22.



	message_id	recipient	...	message_text	init_time	m...	sent_time	is_informed	informed_time	last_modified_time
1	11720b96230443c0b21…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:14…	☐	<null>	2020-06-21 05:00:14.86
2	6cfd89f2e77440f7a3a…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:15…	☐	<null>	2020-06-21 05:00:15.31
3	1cf0753fc1a247a1b63…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:05…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.36
4	9efee431d54b49b9937…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:06…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.38
5	d84b5d57228f485ea30…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:06…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.40
6	e3b61825c4bd4d24b57…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:07…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.42
7	ba623d87b63141a6b15…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:07…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.44
8	ed43748a9c764f1e9bf…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:08…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.45
9	fdf59f80055b4888bf7…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:08…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.46
10	847427fbe84f45a9833…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:09…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.48
11	b491091f65a644a2aec…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:0…	D	2020-06-21 05:00:09…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.50
12	77d996249d6f48ae942…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	D	2020-06-21 05:00:10…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.51
13	dc8db10e6bda4a1495e…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	D	2020-06-21 05:00:10…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.52
14	cc31d29b2afb44cbbc6…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	D	2020-06-21 05:00:11…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.53
15	9c1ab68e72e048ddbde…	77022060057	8111	Тестовое сообщение	2020-06-21 04:59:2…	D	2020-06-21 04:59:24…	☑	2020-06-21 04:59:32…	2020-06-21 04:59:32.37
16	f38773a2313c439ba92…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	D	2020-06-21 05:00:11…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.54
17	68710e69d6204a6abb8…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	D	2020-06-21 05:00:12…	☑	2020-06-21 05:00:12…	2020-06-21 05:00:12.55
18	3218a6b4d84d4a1d954…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:12…	☐	<null>	2020-06-21 05:00:12.60
19	8cdf1ddb17e04fb6962…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:13…	☐	<null>	2020-06-21 05:00:13.08
20	675580320c1a40458a3…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:13…	☐	<null>	2020-06-21 05:00:13.49
21	b9e0119c963645ec9c6…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:13…	☐	<null>	2020-06-21 05:00:13.95
22	71bbf433d80f41d79fa…	77022060057	8111	Тестовое сообщение	2020-06-21 05:00:1…	S	2020-06-21 05:00:14…	☐	<null>	2020-06-21 05:00:14.39

Рисунок 2.22. Обновленные данные

ЗАКЛЮЧЕНИЕ

В результате выполнения данной работы мы рассмотрели как можно разделить приложение на независимых друг от друга микросервисов. Познакомились с инструментами и технологиями для разработки микросервисов такие Spring Framework, RabbitMQ.

Микросервисная архитектура предлагает модульное обслуживание. Так как микросервисы имеют слабую связь, их становиться легче обслуживать. Когда приложение необходимо обновить, настроить или заменить, это легко достигается по каждому микросервису за раз, не затрагивая остальную часть системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ЛИТЕРАТУР

1. Ньюмен С. Создание микросервисов — СПб.: Питер, 2016.
2. Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019.
3. Martin Toshev. Learning RabbitMQ, 2015.
4. Thomas Hunter. Advanced Microservices: A Hands-on Approach to Microservice Infrastructure and Tooling, 2017.
5. M. Hofmann, E. Schnabel, K. Stanley. Microservices Best Practices for Java, 2016.
6. Vinoo Das. Learning Redis, 2015.
7. Харроб Р., Шефер К., Хо К. Spring 5 для профессионалов — СПб.: ООО “Диалектика”, 2019.
8. Craig Walls. Spring in Action Fifth Edition, 2019.
9. Adrian Mouat. Using Docker, 2016.
10. John Carnell. Spring Microservices in Action, 2017.
11. Gavin Roy. RabbitMQ in Depth, 2018.
12. Binildas Christudas. Practical Microservices Architectural Patterns, 2019.
13. Sam Newman, Monolith to Microservices, 2020.
14. Krause, L.: Microservices: Patterns and Applications, 1 edn. Lucas Krause, Paris (2014). 1 April 2015.
15. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. Present and Ulterior Software Engineering, pp. 195–216. Springer, Cham (2017).