МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ПЕРМСКИЙ ГОСУДАРСТВЕННЫЙ АГРАРНО-ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА Д.Н. ПРЯНИШНИКОВА»

Факультет экономики и информационных технологий

Кафедра информационных технологий и программной инженерии

Направление подготовки 09.04.03 Прикладная информатика

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

Оптимизация процесса передачи данных в клиент-серверных приложениях с использованием бинарной сериализации

Автор работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*\* \*\*\*\*\*\*\*\*\*\*

Группа ПИм-21 (подпись, дата)

Научный руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*\* \*\*\*\*\*\*\*\*\*\*

(подпись, дата)

Заведующий кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \*\* \*\*\*\*\*\*\*\*\*\*

(подпись, дата)

Пермь-20\_\_ г.

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc203384580)

[1 Анализ существующих проблем передачи данных в клиент-серверных приложениях 6](#_Toc203384581)

[1.1 Анализ микросервисной архитектуры 6](#_Toc203384582)

[1.2 Анализ клиент-серверного взаимодействия 9](#_Toc203384583)

[1.3 Способы улучшения производительности клиент-серверных систем 11](#_Toc203384584)

[1.4 Анализ форматов данных в клиент-серверном взаимодействии 13](#_Toc203384585)

[1.5 Модель AS-IS процесса получения данных от сервера 15](#_Toc203384586)

[1.6 Модель AS-IS процесса обработки входящего запроса 20](#_Toc203384587)

[1.7 Анализ Java-библиотек для сериализации и десериализации объектов 26](#_Toc203384588)

[2 Проектирование клиент-серверного взаимодействия с применением бинарного формата 28](#_Toc203384589)

[2.1 Модель TO-BE процесса обработки входящего запроса 28](#_Toc203384590)

[2.2 Модель TO-BE процесса обработки входящего запроса 32](#_Toc203384591)

[2.3 Выбор технологий реализации библиотеки 37](#_Toc203384592)

[3 Экспериментальные исследования применения бинарного формата 39](#_Toc203384593)

[3.1 Программная реализация библиотеки для сериализации и десериализации 39](#_Toc203384594)

[3.1.1 Сериализация 39](#_Toc203384595)

[3.1.2 Десериализация 47](#_Toc203384596)

[3.2 Описание процесса апробации 52](#_Toc203384597)

[3.2.1 Определение критериев и показателей оценки эффективности 52](#_Toc203384598)

[3.2.2 Определение технологий тестирования 53](#_Toc203384599)

[3.2.3 Описание тестового Spring Boot приложения 54](#_Toc203384600)

[3.2.4 Описание тестового плана JMeter 56](#_Toc203384601)

[3.3 Анализ результатов тестирования 58](#_Toc203384602)

[ЗАКЛЮЧЕНИЕ 61](#_Toc203384603)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 62](#_Toc203384604)

[ПРИЛОЖЕНИЕ А 66](#_Toc203384605)

[ПРИЛОЖЕНИЕ Б 81](#_Toc203384606)

[ПРИЛОЖЕНИЕ В 91](#_Toc203384607)

**Перечень условных обозначений, терминов и сокращений**

*Сериализация* – процесс преобразования объекта или структуры данных в последовательность байтов для хранения или передачи.

*Десериализация* – обратный процесс восстановления объекта из последовательности байтов.

*Клиент-серверное приложение* – архитектура, в которой клиент отправляет запросы серверу, а сервер их обрабатывает и возвращает результат.

*Микросервис* – независимый сервис, выполняющий одну бизнес-функцию и взаимодействующий с другими сервисами через API.

*JSON (JavaScript Object Notation)* – текстовый формат обмена данными, основанный на синтаксисе JavaScript.

*Java* – объектно-ориентированный язык программирования, используемый для разработки клиент-серверных приложений.

*Spring Boot* – фреймворк для быстрой разработки standalone-приложений на основе Spring с минимальной конфигурацией.

*POJO (Plain Old Java Object)* – простой Java-объект без сложной логики, часто используемый для хранения данных.

*Пропускная способность* – объём данных, передаваемых за единицу времени.

*Парсинг* – процесс преобразования одного типа данных в другой с сохранением смысловой нагрузки данных.

ВВЕДЕНИЕ

Клиент-серверные приложения, построенные на микросервисной архитектуре, требуют эффективных механизмов передачи данных между компонентами системы. Такие механизмы помогают достичь малого времени отклика между компонентами таких приложений. Традиционно для обмена информацией широко используется текстовый формат JSON благодаря своей простоте и человекочитаемости. Текстовые форматы влекут за собой дополнительные накладные расходы на используемую память и дополнительными вычислительными затратами на парсинг типов данных.

Исходя из этого можно вынести одну общую проблему таких систем – неизбежным увеличением затрачиваемого времени на передачу данных между микросервисами по мере развития микросервисного приложения.

Учитывая, что подавляющее большинство веб приложений на текущий момент построены именно с использованием микросервисной или сервис-ориентированной архитектур, то проблема значительной задержки во взаимодействии между компонентами такого приложения становится все более актуальной с каждым днем.

С ростом популярности микросервисной архитектуры возникает потребность в достижении наилучшего быстродействия всей системы.

Объектом исследования является процесс передачи данных в клиент-серверных приложениях, построенных на микросервисной архитектуре.

Предметом исследования является бинарный формат данных.

Целью работы является применения бинарного формата данных для оптимизации взаимодействия между компонентами микросервисных приложений.

Чтобы достичь данную цель, необходимо также поставить ряд задач, выполнение которых приведет к достижения поставленной цели. Задачами данной работы будут являться:

* исследовать наиболее популярные форматы данных, используемых в клиент-серверных приложениях;
* выбрать наиболее эффективный формат данных для оптимизации процесса передачи данных между микросервисами;
* произвести проектирование системы взаимодействия между компонентами микросервисного приложения в виде моделей «как есть» и «как будет»;
* сформировать критерии оценки эффективности проектируемой системы;
* провести тестирование взаимодействия с использованием как традиционных форматов данных, так и выбранных для оптимизации в рамках данной работы;
* произвести оценку эффективности применяемого подхода.

Результатом данной работы является подтверждение эффективности использования двоичного формата данных в микросервисных приложениях.

1 Анализ существующих проблем передачи данных в клиент-серверных приложениях

* 1. Анализ микросервисной архитектуры

Информационные системы, в особенности веб приложения, значительно изменились с момента своего появления. Когда-то каждая система представляла из себя одно единственное приложение, выполняющее ряд определенных задач. Такие приложения были построены на классической монолитной архитектуре, а разрабатывала это приложение одна большая команда разработчиков, но на данный момент такой подход считается устаревшим.

Сейчас же информационная система чаще всего представляет из себя распределенную систему из нескольких приложений, которые взаимодействуют между собой и каждое такое приложение выполняет одну или несколько функций во всей системе. Данный подход в построении информационных систем принято называть микросервисной архитектурой, которая является частным случаем сервис-ориентированной архитектуры.

*Микросервисная архитектура* — вариант сервис-ориентированной архитектуры программного обеспечения, направленный на взаимодействие насколько это возможно небольших, слабо связанных и легко изменяемых модулей — микросервисов, получивший распространение в середине 2010-х годов [1].

Такой переход от одного большого приложения ко многим маленьким по большей части связан развитием практик гибкой разработки и DevOps.

Agile – это разновидность процессов разработки, который включает в себя несколько различных методологий – Feature Driven Development (FDD), Agile Modeling, Agile Unified Process, Scrum и др.

DevOps (Development & Operations) – методология, главная цель которой максимально сократить и автоматизировать процессы сборки, настройки и развертывания программного обеспечения. То есть сократить время после написания кода до его прямого использования на промышленном стенде.

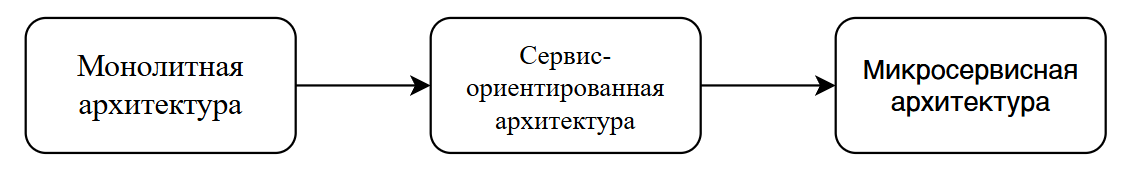
Таким образом, можно выделить общий вектор развития в построении информационных систем – от единого большого к маленькому декомпозированному, то есть от монолитной архитектуры к микросервисной (рис. 1).

Рисунок 1 – Развитие архитектур информационных систем

Каждый из этих подходов призван решить ряд проблем его предшественника, но при этом привнося и новые трудности, специфичные для конкретного подхода. Далее будут рассмотрены преимущества и недостатки микросервисной архитектуры.

К преимуществам микросервисной архитектуры можно отнести:

* масштабируемость;
* гибкость и независимость разработки;
* устойчивость к отказам;
* быстрое развертывание;
* четкое разделение ответственности;
* хорошая поддержка распределенных систем.

К недостаткам данной архитектуры можно отнести;

* сложность управления;
* задержки при взаимодействии и накладные расходы;
* сложность отладки и мониторинга;
* высокие требования к согласованности данных;
* большой объем начальных затрат;
* сложность тестирования.

Особое внимание стоит уделить возрастающему времени взаимодействия и накладным расходам при взаимодействии между компонентами такой системы. Данный недостаток напрямую влияет на успешность клиентского опыта эксплуатации таких систем, а все остальные можно максимально сократить на этапе налаживания процессов разработки и отладки таких систем. То есть чем больше компонентов в системе, тем медленнее она начинает работать внутри себя, так как все больше компонентов внутри неё начинают взаимодействовать между собой.

Следовательно, одной из неизбежно появляющихся проблем таких систем становится постоянно увеличивающееся время работы всей системы в целом, а как следствие и ухудшение клиентского опыта, а значит и упущение потенциального дохода от создания и внедрения такой системы.

* 1. Анализ клиент-серверного взаимодействия

*Клиент-серверная архитектура* — это модель организации вычислительных систем, в которой задачи распределены между клиентами и серверами. В такой архитектуре клиент, обычно являющийся пользователем или программой, запрашивает услуги или ресурсы у сервера, который отвечает на запросы, предоставляя необходимые данные или функциональность. При этом взаимодействие может происходить не только между фронтовым приложением и серверным, но и между двумя серверными приложениями. В обоих случаях взаимодействие происходит через API [2].

*API (Application Programming Interface)* – набор правил и спецификаций, определяющих контракт взаимодействия между приложениями для обмена данными. Обычно взаимодействие происходит согласно архитектурному стилю REST.

*REST (Representational State Transfer)* – это архитектурный стиль взаимодействия между компонентами распределенного приложения, согласованный набор ограничений.

Взаимодействие между клиентом и сервером происходит с использованием протокола HTTP.

HTTP (Hypertext Transfer Protocol) – сетевой протокол, являющийся средством взаимодействия между узлами как сети Интернет, так и узлами в изолированных веб-инфраструктурах.

Данный сетевой протокол широко распространен в большинстве существующих на данный момент информационных системах и позволяет удобно, быстро и понятно настраивать взаимодействие между компонентами системы как с точки зрения проектирования, так и с точки зрения разработки [3].

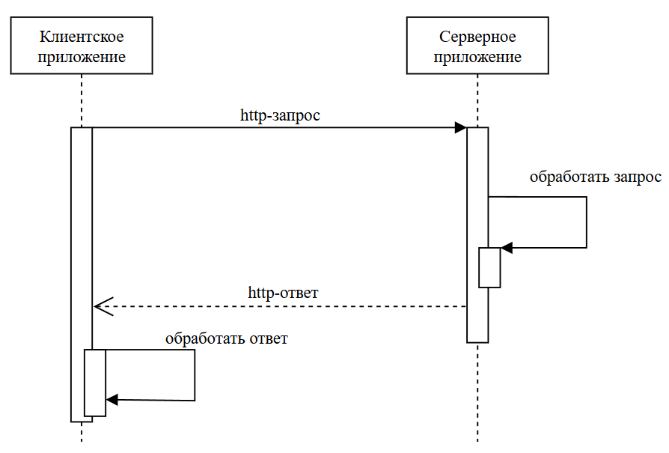
На рисунке 2 представлена схема типичного взаимодействия клиента и сервера.

Рисунок 2 – Взаимодействие между клиентом и сервером по HTTP

К преимуществам клиент-серверной архитектуры можно отнести:

* централизованное хранение данных и управление ими;
* масштабируемость;
* безопасность;
* высокая производительность;
* высокая производительность клиентской части (тонкий клиент);
* совместный доступ к данным (для нескольких клиентов).

К недостаткам такой архитектуры можно отнести:

* высокая степень зависимости клиента от сервера;
* зависимость от работы сети;
* высокие затраты на сервер;
* узкие места в системе (например, низкая производительность сервера);
* сложность развертывания и обслуживания.

Не смотря на все свои недостатки, клиент-серверная архитектура заслуженно имеет высокую популярность и широкую применимость практически во всех современных системах.

* 1. Способы улучшения производительности клиент-серверных систем

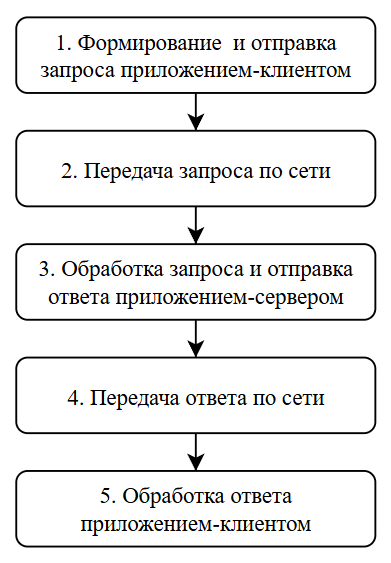
В любом клиент-серверном взаимодействии можно выделить несколько ключевых этапов. Данные этапы являются последовательными и представлены на рисунке 3.

Рисунок 3 – Этапы взаимодействия между клиентом и сервером

Если рассматривать рисунок 3 с точки зрения временных затрат на выполнение каждого этапа, то можно разделить этапы взаимодействия на две группы по основному фактору, влияющему на время выполнения данного этапа.

В первую группу можно отнести этапы №1, №3 и №5. На данных этапах время выполнения зависит от сложности обработки этих данных. Такая обработка состоит из двух частей – преобразование полученного ответа к runtime-объекту и обработка этого объекта согласно бизнес-требованиям конкретно исполняемой в данный момент функции приложения. Чем сложнее алгоритм – тем дольше обработка.

Для повышения эффективности обработки возможно применить более эффективные технологии. Но такой подход требует значительных компетенций (языке программирования, фреймворке). Это может замедлить дальнейшую разработку и увеличить затрат на дальнейшее развитие системы.

Во вторую группу можно отнести этапы №2 и №4. На этих этапах время выполнения зависит только от размера передаваемых данных и сетевых задержек. Чем больше данных – тем дольше передача.

Эти этапы можно оптимизировать, изменив, в первую очередь, размер передаваемых данных. Так как от размера данных напрямую зависит как долго они буду передаваться от отправителя до адресата. Сделать это можно двумя способами:

* исключением «лишних» данных, которые не нужны в рамках выполняемой функции – зависит от частных случаев реализации взаимодействия, невозможно найти универсальное решение без применения точечных изменений API, влияющих на системы-потребители;
* изменением формата передаваемых данных – не зависит от конкретной реализации взаимодействия.

Оптимальным решением для уменьшения затрат времени на выполнение функций системы является изменение формата передаваемых данных. Такой подход абстрагирован от состава передаваемых данных и бизнес-логики для их обработки. Изменение формата не требует точечных изменений на уровне API, так как схема взаимодействия между компонентами системы остается прежней.

* 1. Анализ форматов данных в клиент-серверном взаимодействии

В классической реализации REST-сервисов наиболее часто встречаемым форматом данных является JSON [4]. Но REST-взаимодействие поддерживает несколько форматов данных. Поэтому для оптимизации клиент-серверного взаимодействия посредством важно выбрать наиболее подходящий. Такие форматы данных и их основные особенности приведены в таблице 1.

Таблица 1 – Характеристики форматов данных

|  |  |  |  |
| --- | --- | --- | --- |
| № пп | Наименование формата | Достоинства | Недостатки |
| 1 | JSON | Человекочитаемость, компактность, высокая степень поддержки языками программирования | Нет комментариев, ограничен в типах данных, слабая валидация |
| 2 | XML | Строгая структура, поддержка пространств имен, доступны комментарии и метаданные | Сложен в парсинге, избыточен |
| 3 | Form Data | Простота, гибкость, стандартизирован | Нет вложенных структур, частые проблемы с кодировкой, не подходит для сложных данных |
| 4 | Multipart Form Data | Поддержка файлов, гибкость, стандартизирован | Большой объем данных, сложен в обработке, нечитаемый в сыром виде |
| 5 | Plain Text | Простота, универсальность, минимум накладных расходов | Нет структур и типизации |
| 6 | YAML | Лучшая читаемость, гибкость | Сложен в парсинге, медленная обработка |
| 7 | Binary | Высокая производительность, компактность, строгая типизация | Нечитаемый без парсинга, сложность отладки, дополнительные зависимости |

Каждый из приведенных форматов данных в таблице 1 имеет как достоинства, так и недостатки, исходя из которых формируется понимание целесообразности применения того или иного формата.

Так, форматы JSON, XML и YAML наиболее удобные для чтения человеком, чем все остальные форматы, но это достоинство форматов нивелируется большим размером, который в итоге занимают данные в таком формате [5]. Чем больше дополнительной информации для удобства чтения, тем больший размер будут иметь данные.

Форматы Form Data и Plain Text не поддерживают вложенные структуры или достаточное количество типов данных. Переход от классического JSON, предоставляющего вложенные структуры и типы данных, к таким форматам будет сложным, требующим внесения изменений не только в API, но и в реализацию этого API непосредственно в кодовой базе систем, а иногда и вовсе невозможным.

Формат Multipart Form Data имеет главный недостаток – большой объем данных, представленных в таком формате. Остальные особенности данного формата в таком случае не играют никакой роли, так как от размера данных зависит скорость их передачи. Такой формат данных также не подходит для оптимизации клиент-серверного взаимодействия.

Бинарный формат данных одной стороны вовсе не читается человеком в сыром виде, но несмотря на это бинарный формат данных наиболее легковесный из всех приведенных форматов, а также поддерживает строгую типизацию, к которой стремятся все современные и развивающиеся языки программирования. Бинарный формат данных отлично подойдет для использования в клиент-серверном взаимодействии с целью оптимизации процесса передачи данных.

Таким образом, для небольших систем, не требующих оптимизации, отлично подойдут форматы JSON или XML. Для максимальной скорости передачи следует использовать бинарный формат данных.

* 1. Модель AS-IS процесса получения данных от сервера

Для описания текущего состояния процесса получения данных была построена модель «как есть». При построении модели использована нотация моделирования IDEF0 [6]. Это позволит рассмотреть процесс в целом и понять масштаб изменений, а также в каком месте всего процесса эти изменения будут добавлены.

В рассматриваемой ниже диаграмме процесс продемонстрирован с точки зрения backend сервиса, реализованного на Java с использованием фреймворка Spring Boot [7].

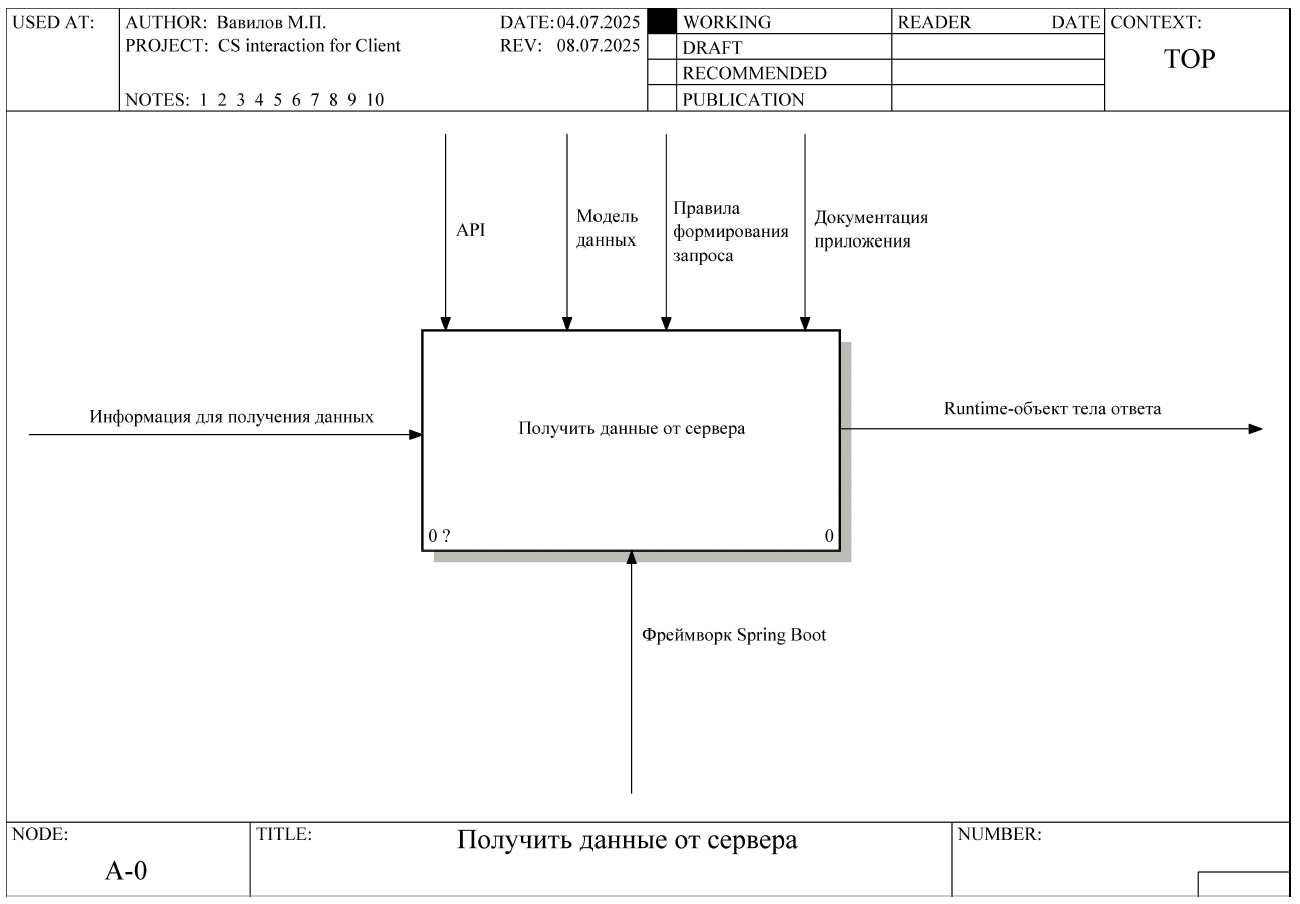
На рисунке 4 представлена контекстная диаграмма всего процесса AS-IS получения данных от сервера.

Рисунок 4 – Контекстная диаграмма процесса «Получить данные от сервера» AS-IS

Контекстная диаграмма представлена в виде одного блока «Получить данные от сервера».

На вход приходит информация для получения бизнес-данных. Она будет использована для составления HTTP-запроса.

На выходе отдается runtime-объект тела ответа. Это бизнес-данные и финальный результат отправки HTTP-запроса.

Стрелки управления на данном уровне являются:

* API (спецификация взаимодействия между клиентом и сервером);
* модель данных (структура передаваемых данных);
* правила формирования запроса (требования к данным запроса);
* документация приложения (описание программного кода сервиса).

Стрелкой «механизм» для диаграммы является фреймворк Spring Boot, так как именно фреймворк управляет отправкой запроса и получением ответа, а также управляет зависимостями приложения, с помощью которых построена вся логика обработки данных.

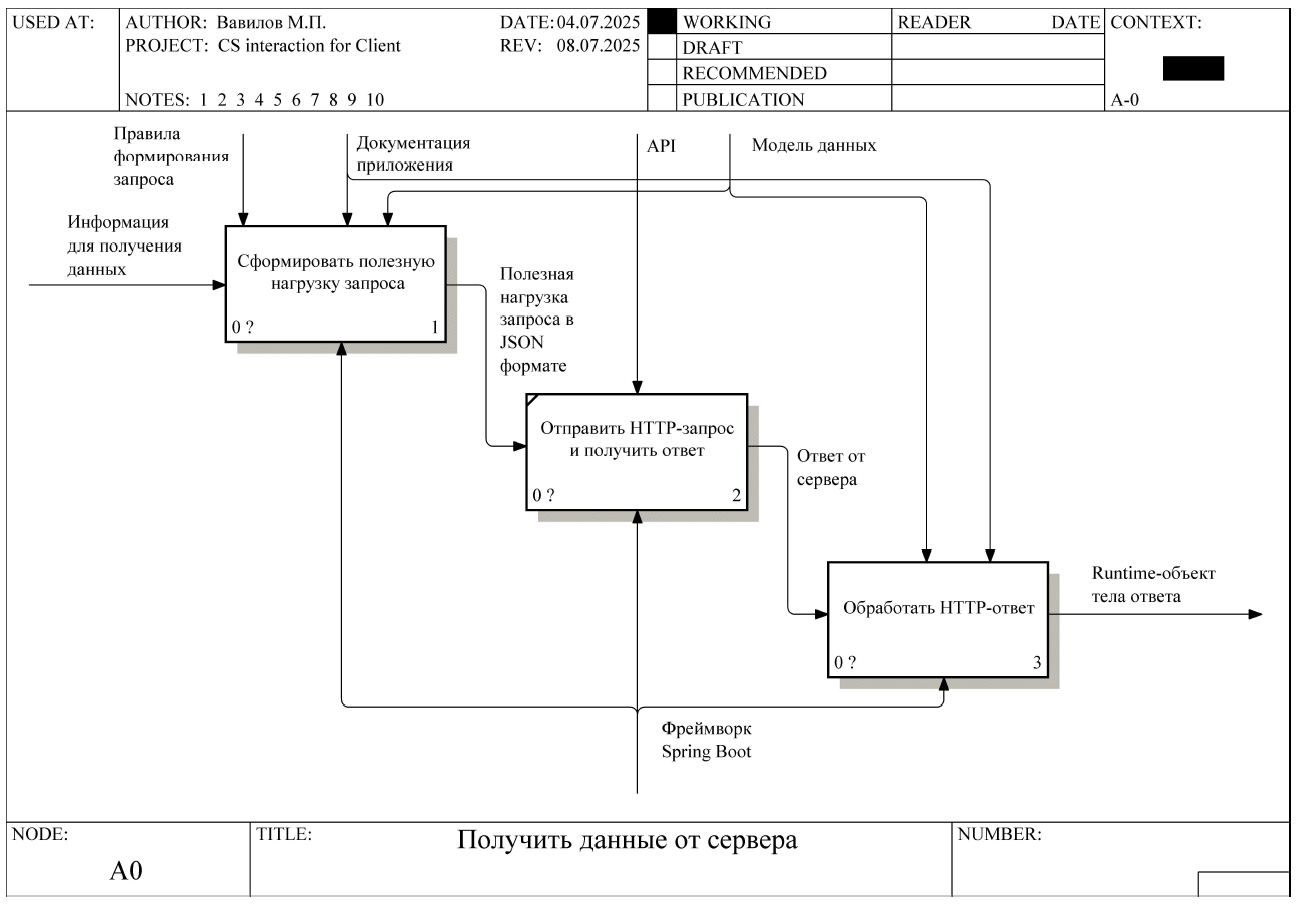
Далее необходимо рассмотреть процесс получения данных от сервера более детально на рисунке 5.

Рисунок 5 – Диаграмма декомпозиции процесса «Получить данные от сервера» AS-IS

Весь процесс можно разделить на три условных этапа.

В первую очередь с помощью данных для запроса формируется полезная нагрузка запроса – тело запроса. Позднее этот этап будет рассмотрен более детально.

Во вторую очередь происходит отправка запроса и получение на него ответа. Как правило, весь этот процесс берет на себя фреймворк Spring Boot, предоставляя ту или иную реализацию HTTP-клиента, но в рамках данной работы это играет роли [8].

В третью очередь происходит обработка полученного HTTP-ответа. На данном этапе происходит извлечение полезных бизнес-данных из ответа от сервера [9].

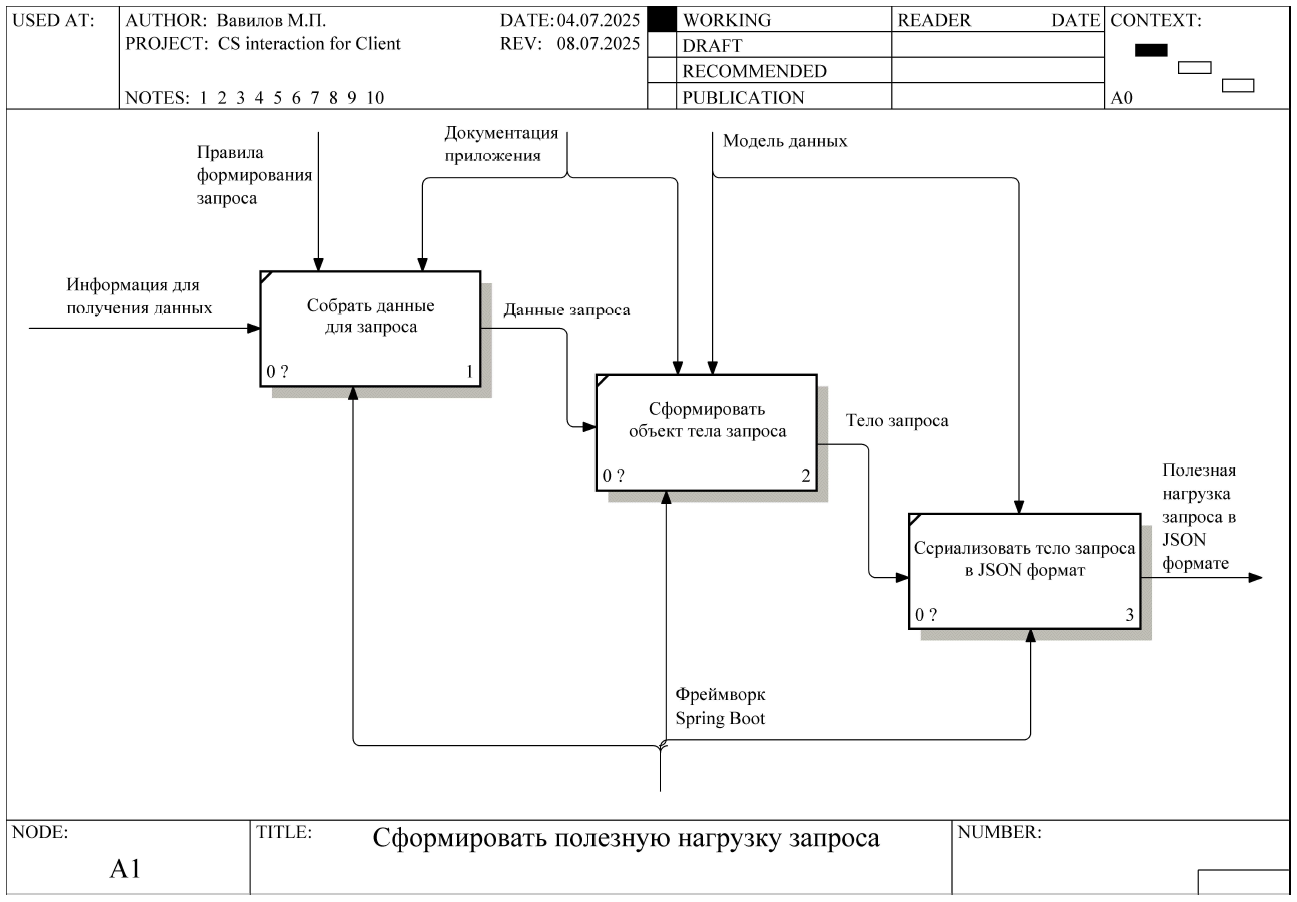
Теперь необходимо рассмотреть более детально процесс формирования тела запроса на рисунке 6.

Рисунок 6 – Диаграмма декомпозиции процесса «Сформировать полезную нагрузку запроса» AS-IS

Данный процесс также разделен на три этапа.

Сначала происходит сбор нужной информации для составления тела запроса. Источники этих данных могут быть разные – интегрированные с текущим сервисом системы, смежные сервисы текущей системы, константы, база данных и т.д.

Далее происходит процесс формирования тела запроса. На данном этапе формируется один объект, описывающий тело запроса со всеми необходимыми данными, приведенные к требуемому API формату (в данном случае и в качестве примера – JSON). Объект передается на выход данного процесса.

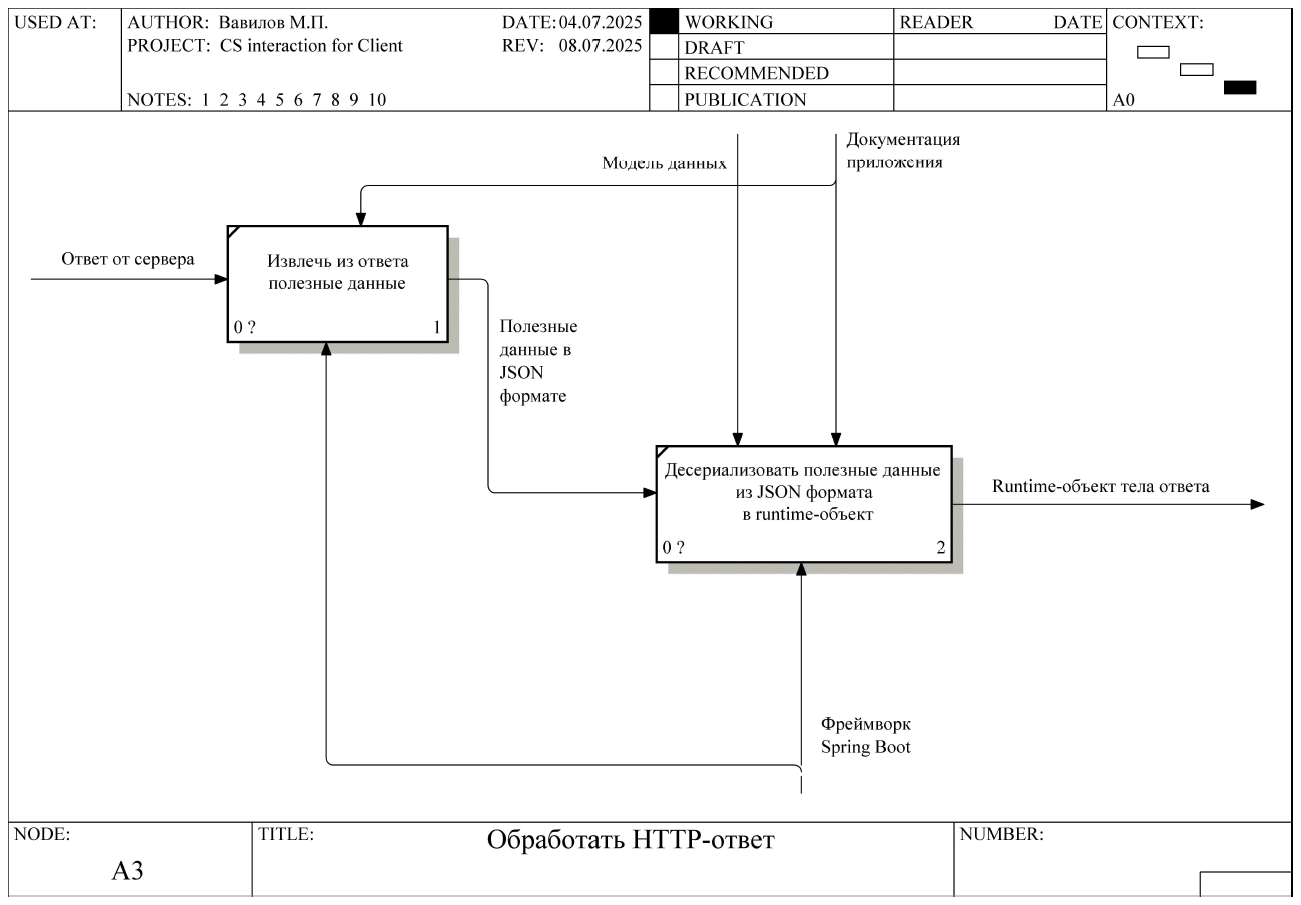
И последней в процессе получения данных от сервера диаграммой является декомпозиция процесса обработки HTTP-ответа, представленная на рисунке 7.

Рисунок 7 – Диаграмма декомпозиции процесса «Обработать HTTP-ответ» AS-IS

Последний этап процесса получения данных состоит из двух шагов.

Первый шаг – извлечение из полученного ответа полезных данных (тело ответа), которые являются JSON строкой.

Второй шаг – десериализация полезных данных из JSON формата в runtime-объект. Такой объект позволит работать удобно обрабатывать ответ как обычный экземпляр Java-класса, описывающего своим состоянием ответ от сервера [10].

Рассмотрев детально процесс получения данных от сервера, стоит отметить, что использование JSON формата данных происходит дважды – при формировании тела HTTP-запроса и при обработке полученного ответа на этот запрос. Внутри бизнес-логики приложения объект представлен в виде конкретного экземпляра класса.

* 1. Модель AS-IS процесса обработки входящего запроса

Изменение используемого формата данных затрагивает не только отправителя, но и получателя HTTP-запроса в клиент-серверном взаимодействии между двумя микросервисами в системе. Поэтому также необходимо детально рассмотреть процесс обработки входящего HTTP-запроса со стороны сервера.

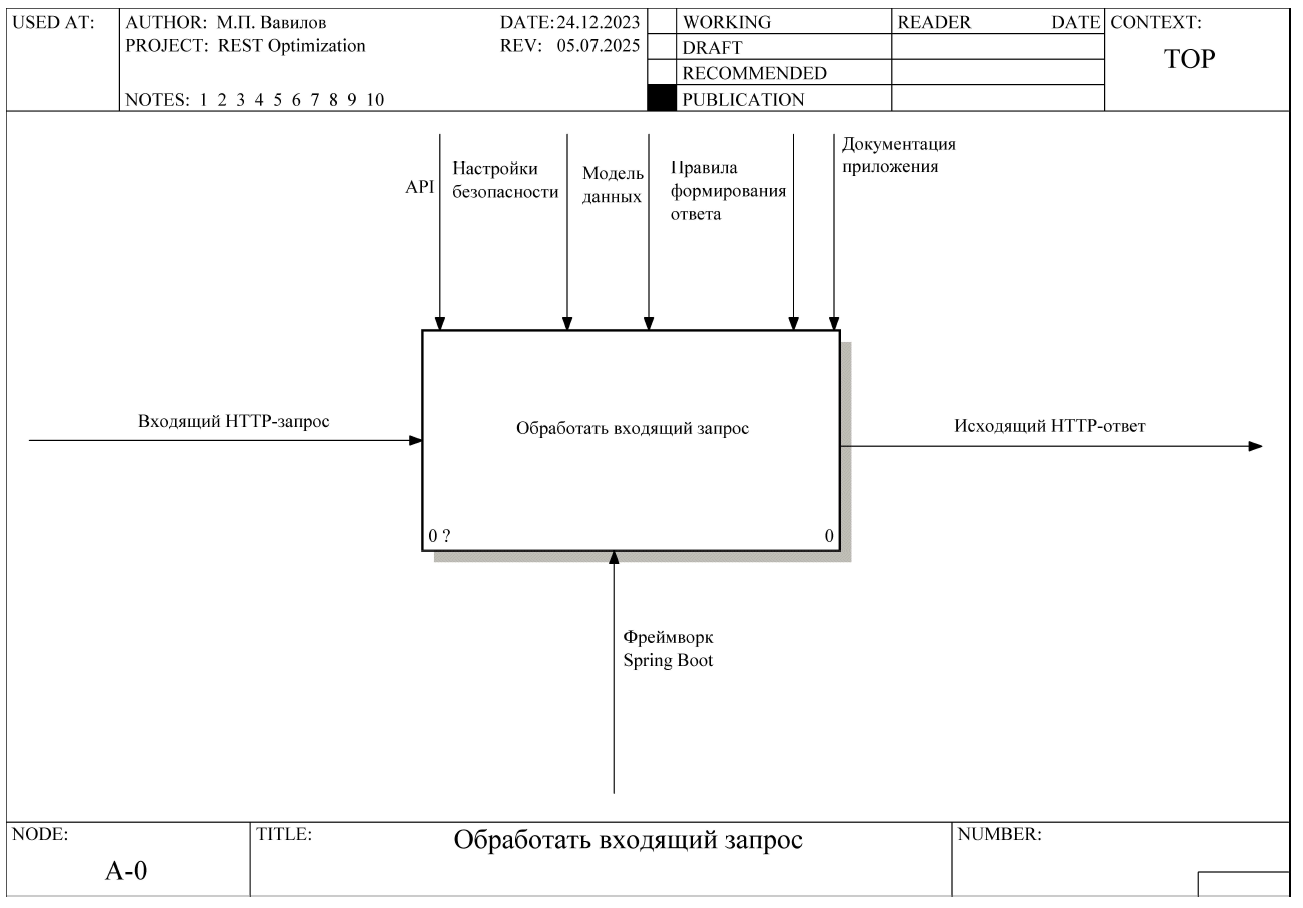
На рисунке 8 представлена контекстная диаграмма процесса обработки входящего запроса.

Рисунок 8 – Контекстная диаграмма процесса «Обработать входящий запрос» AS-IS

Контекстная диаграмма содержит в себе единственный блок «Обработать входящий запрос».

На входе данный процесс имеет входящий HTTP-запрос с полезными данными в теле [11].

На выходе процесс содержит исходящий HTTP-ответ от сервиса, в котором содержатся либо полезные данные, которые запрашивал клиент, либо сообщение об ошибке, которая могла произойти в ходе обработки запроса.

Стрелками управления для данного процесса являются:

* API (спецификация взаимодействия между клиентом и сервером);
* настройки безопасности (правила доступа к ресурсам сервера);
* модель данных (структура передаваемых данных);
* правила формирования ответа (требования к данным ответа);
* документация приложения (описание программного кода сервиса).

Механизмом для всего процесса также, как и для отправки запроса, будет являться фреймворк Spring Boot, управляющий принятием запроса и отправкой ответа.

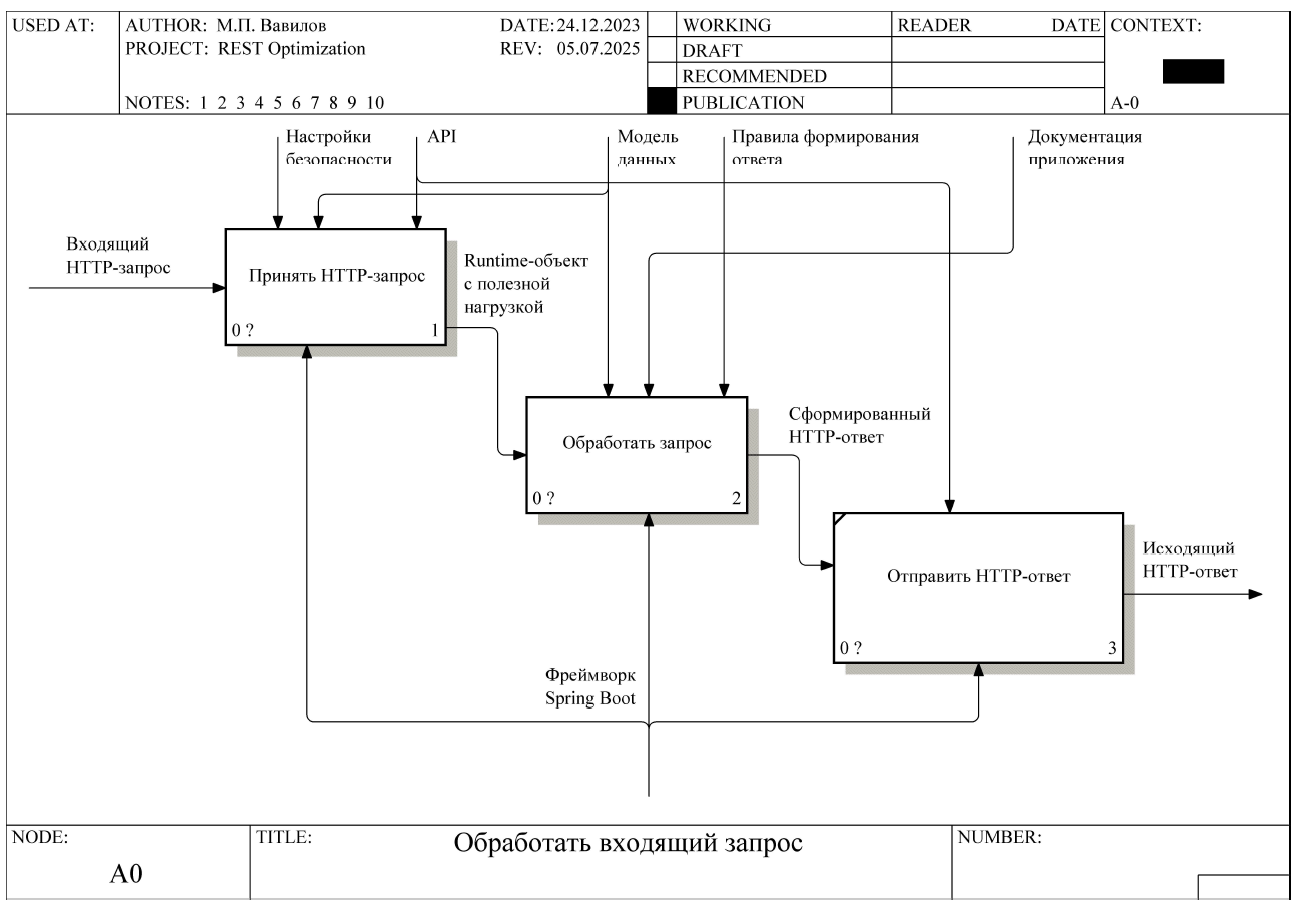
На рисунке 9 представлен первый уровень декомпозиции процесса обработки входящего запроса.

Рисунок 9 – Диаграмма декомпозиции процесса «Обработать входящий запрос» AS-IS

Весь процесс обработки запроса разбит на три последовательных блока.

Первый блок «Принять HTTP-запрос» отвечает за наличие ресурса по указанному пути запроса и проверку наличия доступа к этому ресурсу. Также на данном этапе извлекается полезная нагрузка из запроса. Процесс извлечения будет рассмотрен далее.

Вторым блоком диаграммы декомпозиции процесса обработки входящего запроса является непосредственно обработка запроса. На данном над данными производятся бизнес-операции.

Последний блок производит отправку HTTP-ответа на запрос. В рамках данной работы детальное рассмотрение данного этапа не требуется.

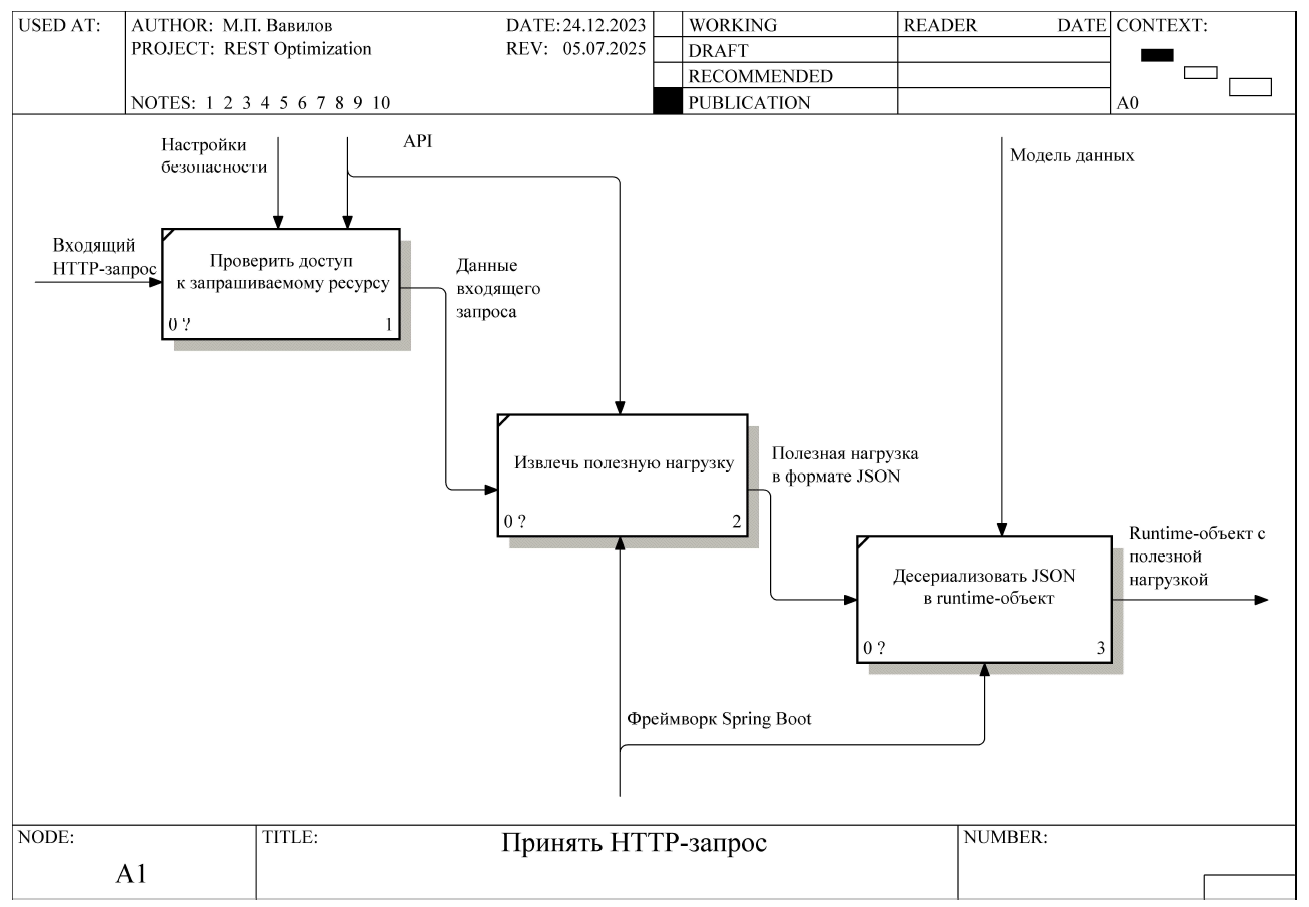
Далее необходимо рассмотреть процесс принятия HTTP-запроса, который представлен на рисунке 10.

Рисунок 10 – Диаграмма декомпозиции процесса «Принять HTTP-запрос» AS-IS

Диаграмма принятия HTTP-запроса разделена на три блока.

Первым блоком данной диаграммы является «Проверить доступ к запрашиваемому ресурсу». На этом этапе происходит проверка наличия доступа к ресурсу согласно настройкам безопасности, то есть производится авторизация.

Как только проверка доступа была успешно пройдена, то данные запроса передаются в блок «Извлечь полезную нагрузку». На данном этапе извлекается тело запроса в формате JSON из всего запроса и передаются в третий блок.

Третий блок отвечает за преобразование полезной нагрузка в формате JSON к runtime-объекту для дальнейшего использования такого объекта в бизнес-операциях по текущему процессу во всей системе.

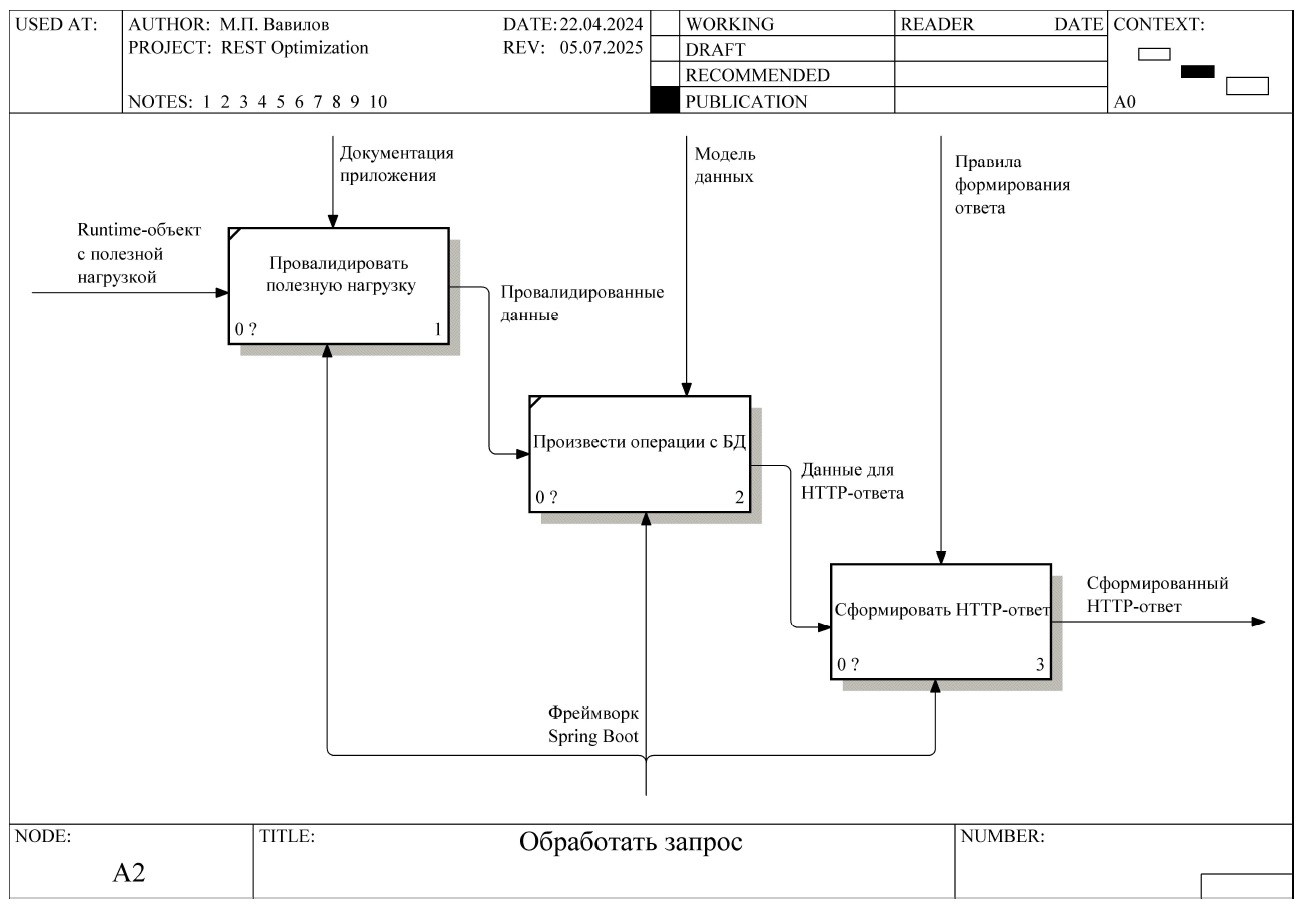
После получения Java-объекта [12] происходит обработка этих данных по текущей бизнес-операции. Данный процесс представлен на рисунке 11.

Рисунок 11 – Диаграмма декомпозиции процесса «Обработать запрос» AS-IS

На диаграмме декомпозиции процесса обработки запроса присутствует три блока. Сначала производится валидация полученных из запроса данных на наличие, формат и так далее. После чего производятся операции с базой данных. Операции с БД могут и отсутствовать, если этого не предусматривает бизнес-процесс. После того, как вся бизнес-обработка была произведена, приложение переходит к формированию HTTP-ответа на запрос.

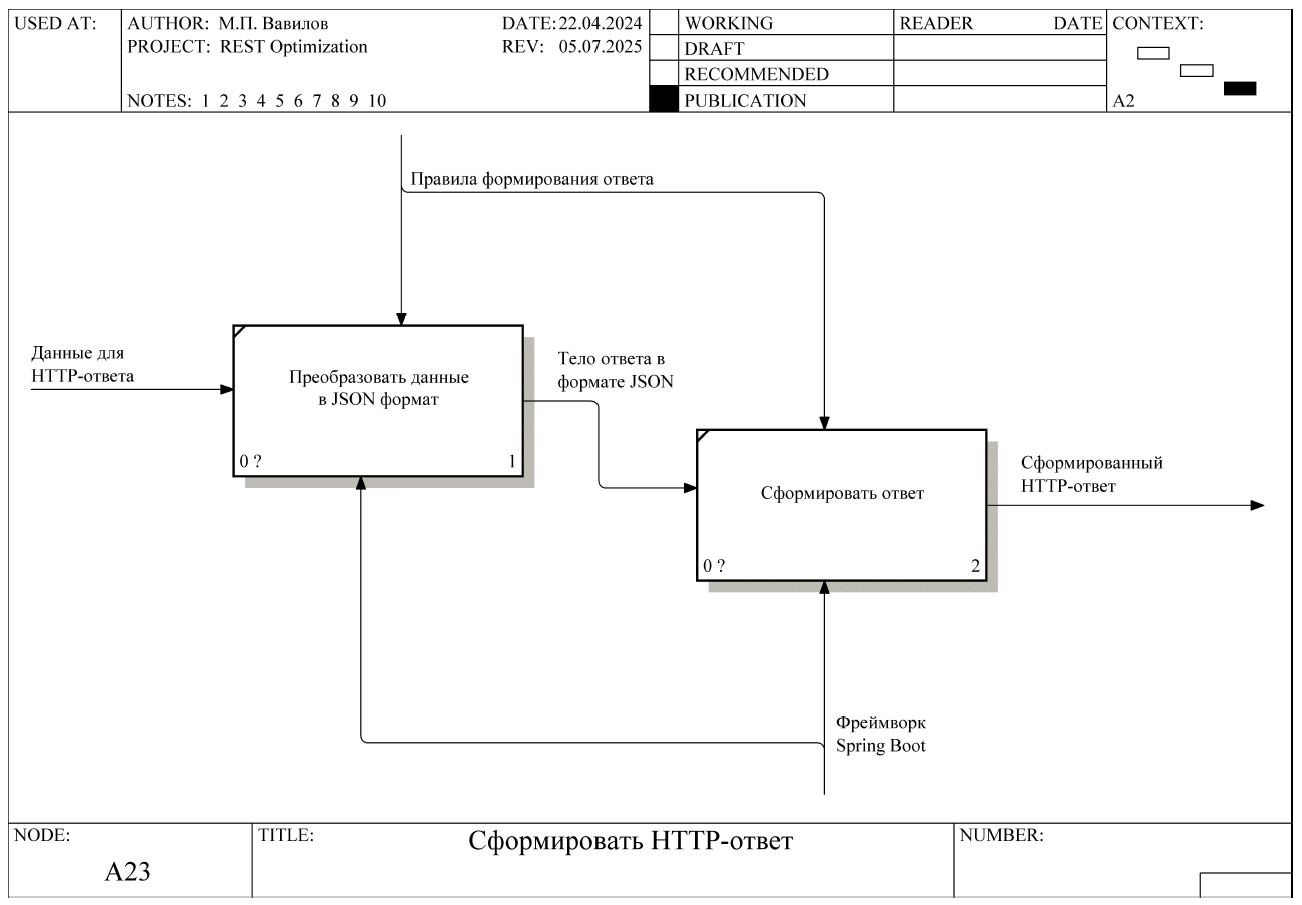
Процесс формирования ответа детально представлен на рисунке 12.

Рисунок 12 – Диаграмма декомпозиции процесса «Сформировать HTTP-ответ» AS-IS

Процесс формирования важен в рамках данной работы и содержит в себе два блока.

Первый блок отвечает за преобразование данных ответа (как правило, тело ответа) в используемый по API формату. Чаще всего таким форматом является JSON. За преобразование отвечает тот же механизм, что и преобразовывал данные запроса из JSON в runtime-объект.

После формирования тела ответа создается ответ на HTTP-запрос. Данный процесс происходит на втором блоке рассматриваемой формирования HTTP-ответа. На данном этапе добавляется различная мета информация к ответу. Затем ответ передается на отправку клиенту.

Таким образом, детально проанализировав весь процесс обработки запроса сервером можно отметить два момента, в которых происходит использование JSON формата. Первый раз это происходит при извлечении полезных данных из тела запроса. Второй раз использование JSON формата происходит в момент формирования ответа на входящий запрос, когда к данному формату приводится тело ответа.

* 1. Анализ Java-библиотек для сериализации и десериализации объектов

В рамках исследования возможных форматов данных был выбран бинарный формат данных. Выбор обусловлен высокой скоростью передачи данных в таком формате между сервисами.

Для реализации RESTful сервисов популярен язык программирования Java. На Java написан фреймворк Spring Boot [13]. Он позволяет быстро поднимать RESTful-сервисы за счет высокого уровня абстракции.

Для сериализации и десериализации данных используются специальные Java библиотеки. В первую очередь необходимо рассмотреть наиболее популярные [14]. К таким библиотекам можно отнести:

* Java Serialization API;
* Jackson;
* Apache Avro.

В таблице 2 приведены преимущества и недостатки данных библиотек.

Таблица 2 – Сравнение библиотек для сериализации

|  |  |  |
| --- | --- | --- |
| Библиотека | Преимущества | Недостатки |
| Java Serialization API | * встроена в Java: * простота в использовании * поддержка графов. | * низкая производительность; * большой размер данных; * уязвимость к подменам. |
| Jackson | * высокая скорость работы; * поддержка многих форматов данных; * аннотации; * поддержка разных версий данных. | * ручное описание структур данных; * низкая производительность для бинарного формата; * избыточность для простых задач. |
| Apache Avro | * быстрая сериализация; * совместимость версий. | * сложность настройки; |

Библиотеки Java Serialization API и Jackson имеют низкую производительность при работе с двоичным форматом. Библиотека Apache Avro требует сложной настройки при использовании. Реализация собственной библиотеки для сериализации и десериализации будет наилучшим решением.

Преимуществами собственной реализации являются:

* контроль над процессом сериализации;
* производительность (добавление только нужных данных);
* возможность адаптации под специфичные структуры данных;
* обратная совместимость;
* возможность добавлять собственные механизмы миграции;
* минимум подключаемых зависимостей;
* легкая отладка за счет опыта реализации.

Собственная библиотека упростит переход на бинарный формат и позволит добиться максимальной производительности от библиотеки. Использование готового решения не позволяет этого сделать.

В рамках аналитического раздела была рассмотрена проблематика микросервисной архитектуры. Основной проблемой такой архитектуры является увеличение времени работы всей системы с ростом количества микросервисов в такой системе.

Было проанализировано клиент-серверное взаимодействие, а также приведены преимущества и недостатки такого подхода. Были определены два способа по улучшению производительности в такой системе. Из них был выбран переход к другому формату данных при передаче.

Также были составлены две AS-IS модели процессов отправки клиентского запроса и обработки входящего запроса. Это позволило определить места в системе, в которых потребуется внести изменения для оптимизации системы посредством замены формата данных на бинарный.

Были исследованы основные библиотеки для сериализации и десериализации данных на языке программирования Java. Реализация собственной библиотеки для сериализации позволит полностью контролировать процесс сериализации и добавиться максимальной производительности.

2 Проектирование клиент-серверного взаимодействия с применением бинарного формата

Проектный раздел содержит в себе описание процессов системы с применением бинарного формата данных, выбор технологий реализации, а также описание реализуемой в рамках данной работы библиотеки.

2.1 Модель TO-BE процесса обработки входящего запроса

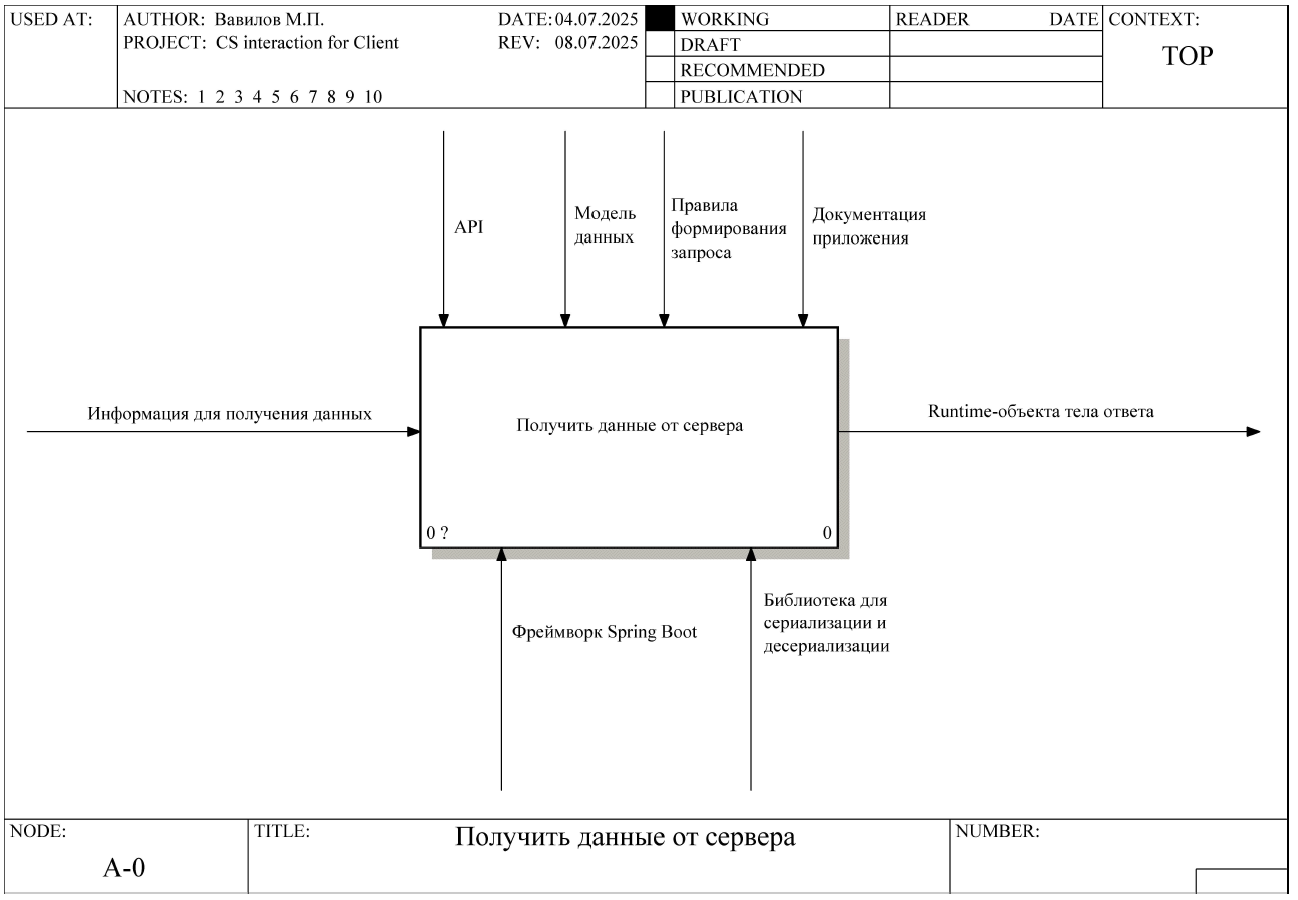
Модель TO-BE процесса получения данных с сервера содержит изменения, которые связаны с формата данных на бинарный. На рисунке 13 приведена контекстная диаграмма данного процесса.

Рисунок 13 – Контекстная диаграмма процесса «Получить данные от сервера» TO-BE

Ключевое изменение на самом верхнем уровне диаграммы описания процесса получения данных от сервера заключается в добавлении нового механизма. Новым механизмом является библиотека для сериализации и десериализации объектов. Данный механизм будет использоваться вместо стандартного сериализатора, поставляемый фреймворком Spring Boot.

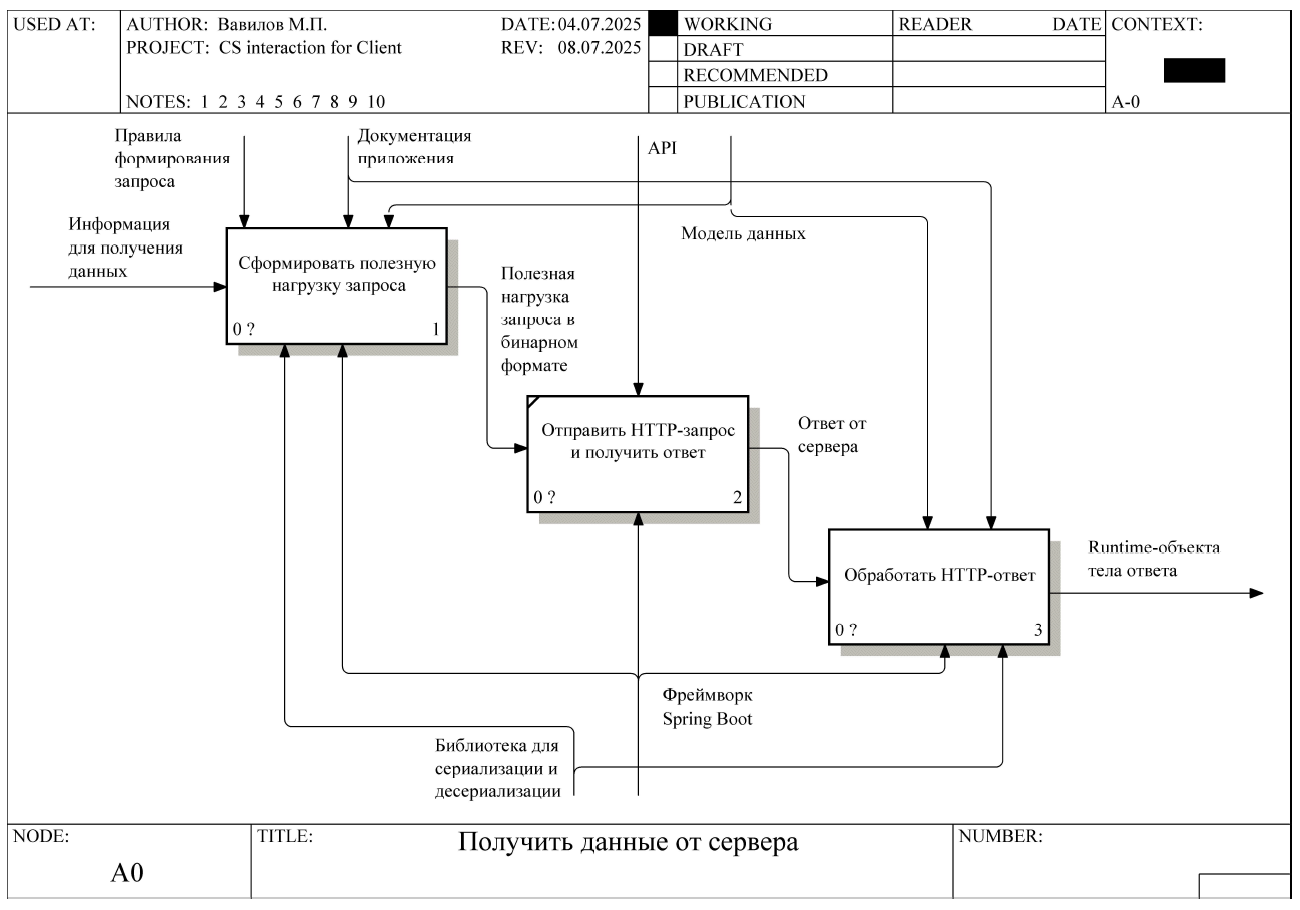
На рисунке 14 представлена диаграмма декомпозиции процесса получения данных с сервера.

Рисунок 14 - Диаграмма декомпозиции процесса «Получить данные от сервера» TO-BE

Первым изменением, в сравнении с аналогичной диаграммой AS-IS процесса, является добавление новой стрелки механизма – библиотеки для сериализации и десериализации. Данный механизм будет использоваться внутри блоков формирования полезной нагрузки запроса и обработки HTTP-ответа. Помимо этого, на вход второго блока «Отправить HTTP-запрос и получить ответ» теперь приходит полезная нагрузка в бинарном формате, в AS-IS диаграмме декомпозиции этого процесса использовался формат JSON.

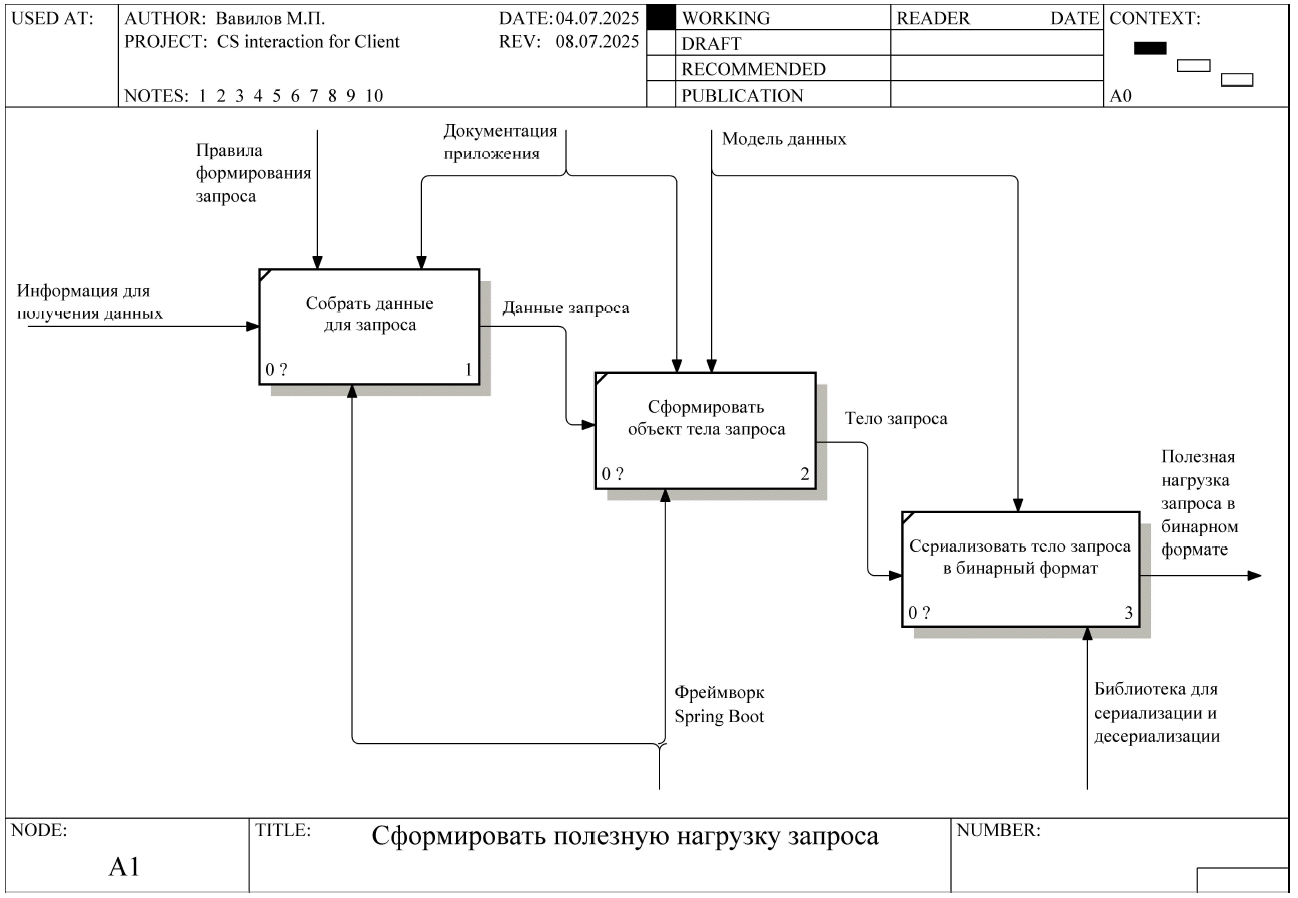
На рисунке 15 представлена диаграмма декомпозиции процесса формирования полезной нагрузки запроса.

Рисунок 15 - Диаграмма декомпозиции процесса «Сформировать полезную нагрузку запроса» TO-BE

На данном уровне диаграммы получения данных с сервера был изменен последний блок, отвечающий за сериализацию тела запроса в используемый формат данных. В TO-BE процессе происходит приведение к бинарному формату, а не к JSON. По этой причине на выходе из последнего блока полезная нагрузка представлена в бинарном формате.

Также для последнего блока был изменен механизм, используемый в нем. Ранее использовался стандартный сериализатор Jackson из фреймворка Spring Boot, он был заменен на библиотеку для сериализации и десериализации.

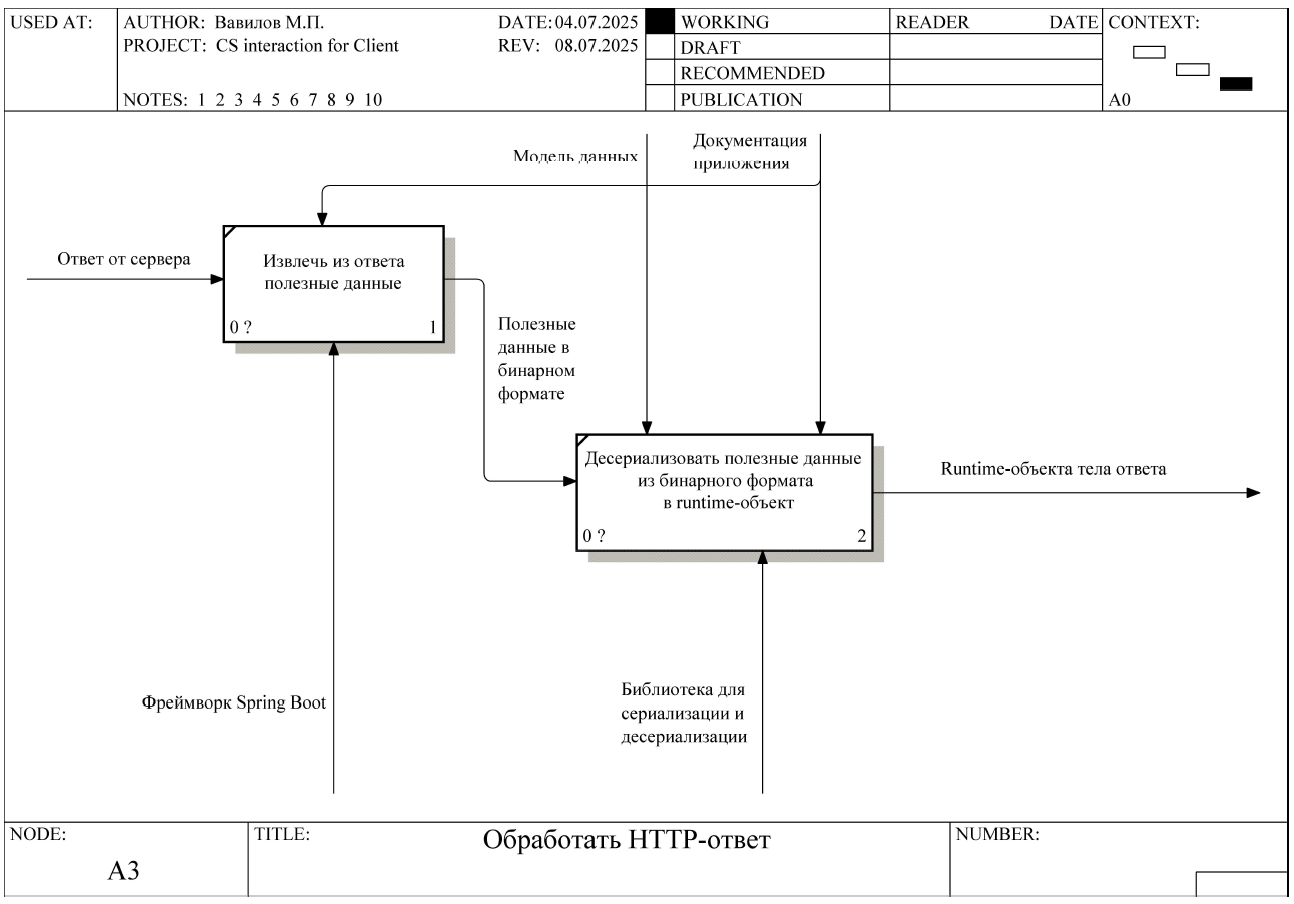
Рисунок 16 содержит описание процесса обработки HTTP-ответа.

Рисунок 16 - Диаграмма декомпозиции процесса «Обработать HTTP-ответ» TO-BE

Был изменен второй блок, отвечающий за десериализацию в runtime-объект. В AS-IS модели производился парсинг из JSON формата в Java-объект, теперь производится парсинг из массива байт в такой же Java-объект. Объект остался без изменений. Изменились и данные, которые этот блок принимает на вход – их формат был изменен с JSON на бинарный. Также для второго блока был изменен механизм со Spring Boot фреймворка на библиотеку для сериализации.

В результате перехода к бинарному формату данных на всех диаграммах процесса получения данных от сервера были модифицированы блоки работы с JSON на блоки работы с бинарным форматом. При этом и изменился используемый ими механизм, которым теперь является библиотека для сериализации и десериализации объектов.

2.2 Модель TO-BE процесса обработки входящего запроса

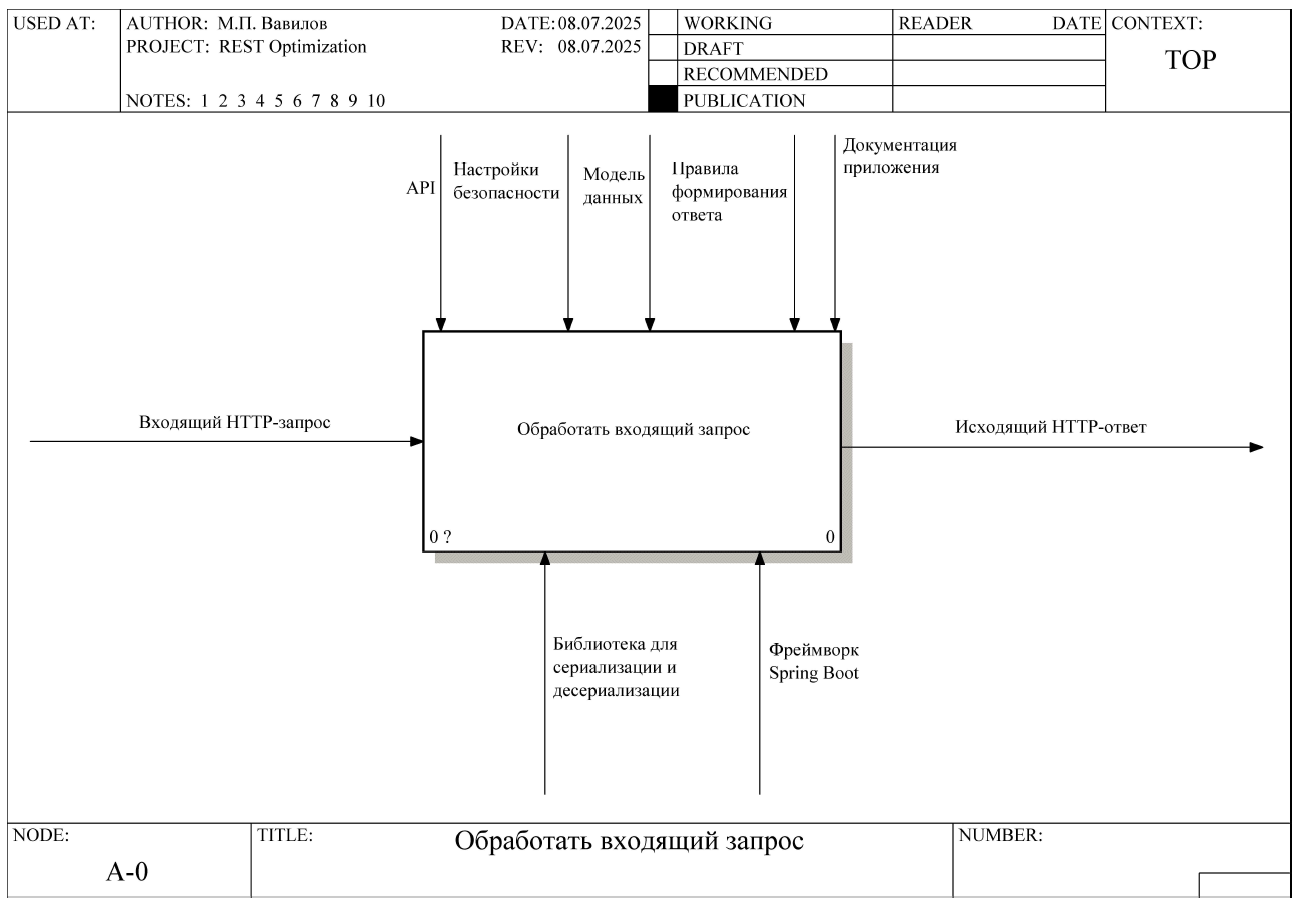
Модель TO-BE процесса обработки входящего запроса содержит изменения, связанные с заменой формата на бинарный. На рисунке 17 представлена контекстная диаграмма данного процесса.

Рисунок 17 – Контекстная диаграмма процесса «Обработать входящий запрос» TO-BE

Главное изменение на данном уровне – добавление новой стрелки механизма «Библиотека для сериализации и десериализации». По умолчанию преобразованием данных управляет фреймворк Spring Boot. Для сериализации он использует библиотеку Jackson. При использовании бинарного формата данных преобразование происходит в ручном режиме – то есть преобразование ложится на плечи разработчиков.

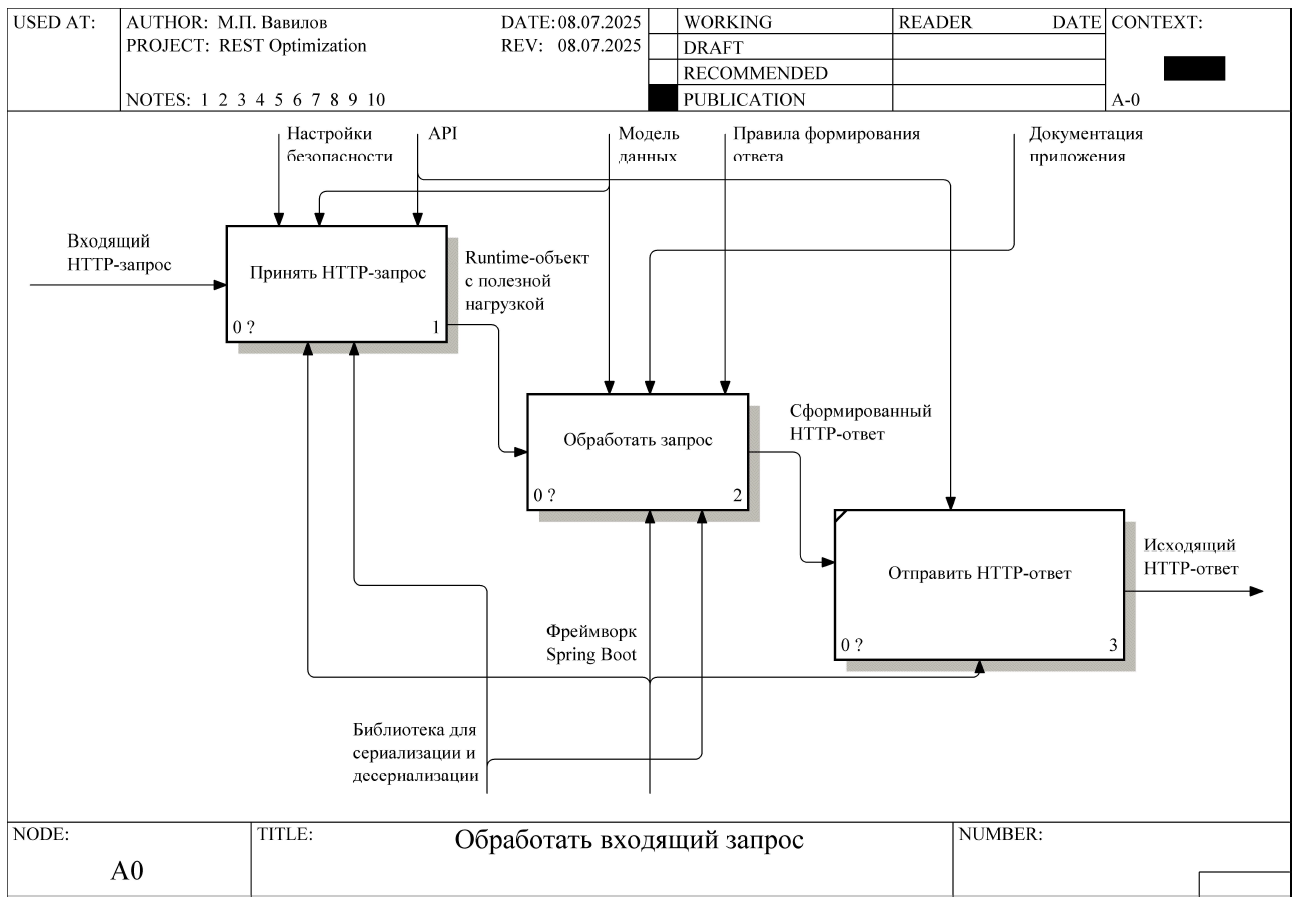
На рисунке 18 представлен следующий уровень диаграммы – декомпозиция процесса обработки входящего запроса.

Рисунок 18 – Диаграмма декомпозиции процесса «Обработать входящий запрос» TO-BE

Изменения на этом уровне связаны с изменениями на предыдущем уровне. Был добавлен механизм в виде библиотеки, который используется в двух блоках данной диаграммы. Первым блоком, использующим рассматриваемый механизм, является блок «Принять HTTP-запрос», вторым блоком является «Отправить HTTP-ответ». Использование библиотеки на данных этапах обусловлено тем, что во время принятии запроса массив байт преобразуется в runtime-объект тела запроса, а при отправке ответа происходит преобразование runtime-объекта тела ответа в массив байт.

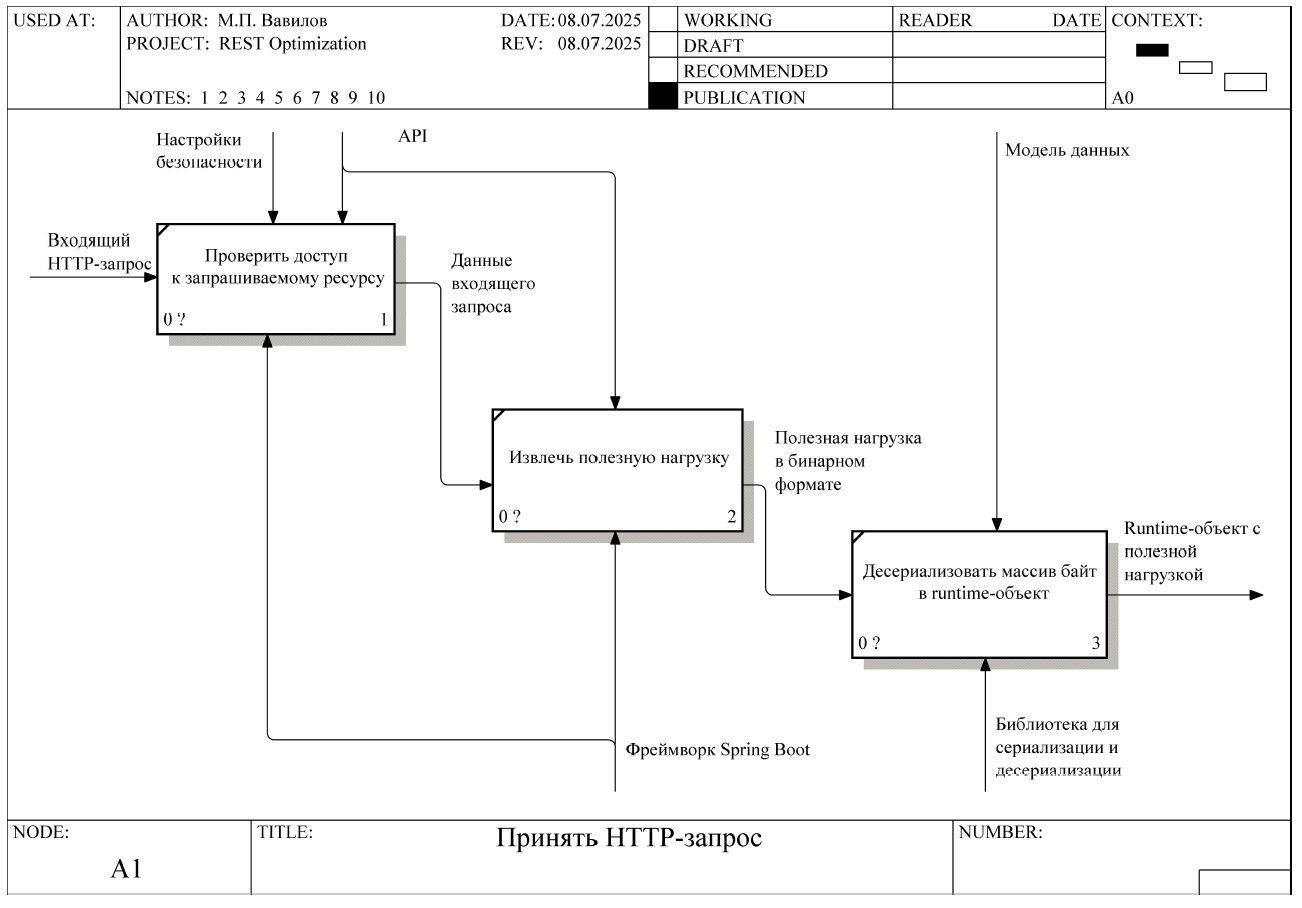
На рисунке 19 представлена диаграмма декомпозиции процесса принятия HTTP-запроса.

Рисунок 19 – Диаграмма декомпозиции процесса «Принять HTTP-запрос» TO-BE

В аналогичном уровне AS-IS модели процесса принятия запроса этот блок назывался «Десериализовать JSON в runtime-объект». Теперь этот блок называется «Десериализовать массив байт в runtime-объект», то есть происходит тот же самый процесс преобразования данных используемого формата в Java-объект, однако в качестве используемого формата выступает бинарный. Также был заменен механизм для данного блока. Ранее механизмом был «Фреймворк Spring Boot», он был заменен на библиотеку для сериализации, так процесс десериализации будет происходить под управлением библиотеки, а не внутренних механизмов Spring Boot.

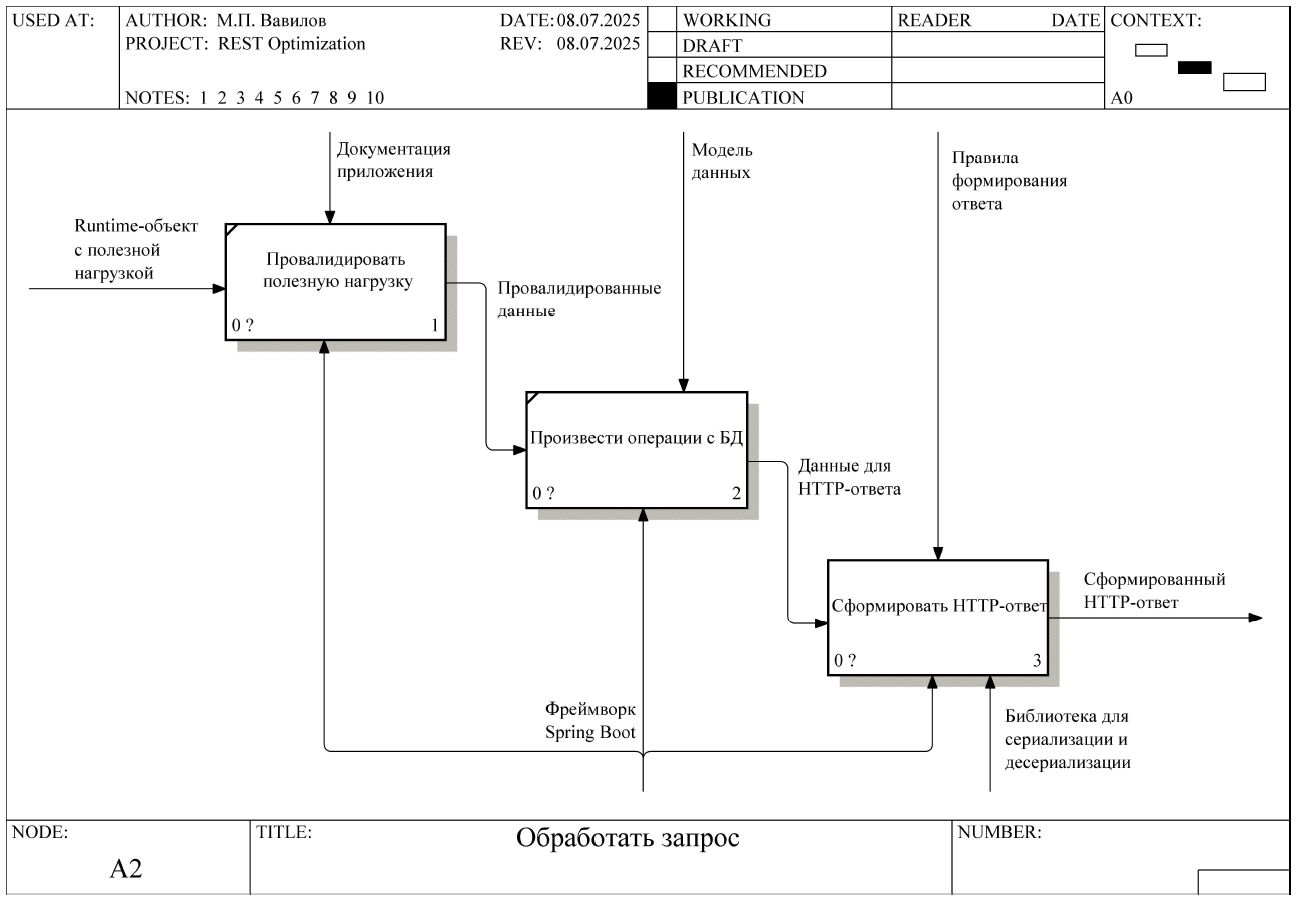
На рисунке 20 представлен процесс обработки запросы, в котором также присутствуют изменения из-за смены используемого формата.

Рисунок 20 – Диаграмма декомпозиции процесса «Обработать запрос» TO-BE

В процессе обработки запроса был добавлен механизм в виде библиотеки для сериализации и десериализации, который будет использован внутри блока «Сформировать HTTP-ответ». При изменении формата данных в теле ответа также отправляется массив байт, а не JSON строка.

В блоках валидации нагрузки и произведения операций с базой данных процесса обработки запроса изменения отсутствуют, так как на этих этапах используется runtime-объект, который был ранее получен из массива байт при принятии клиентского запроса.

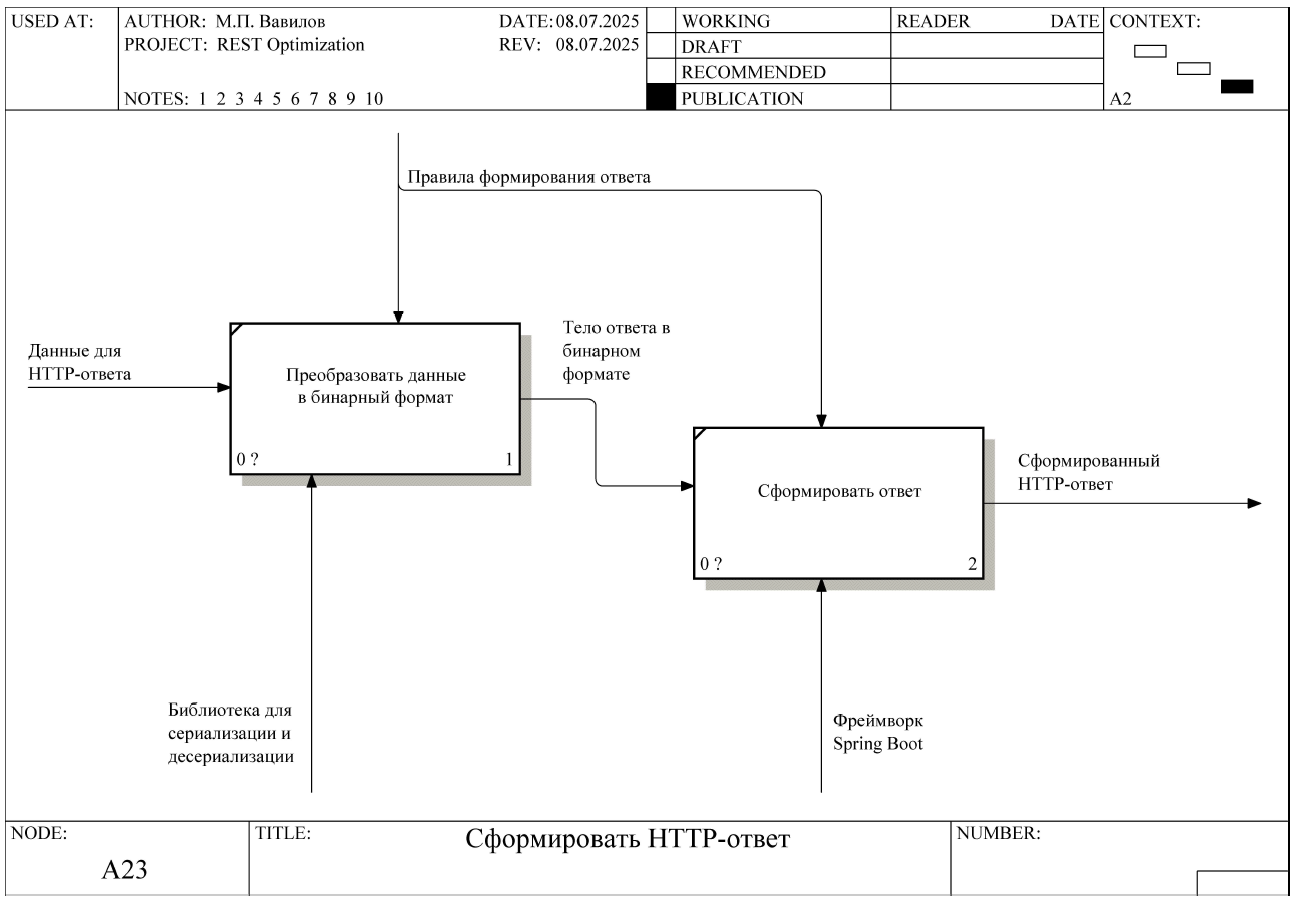
Далее на рисунке 21 представлена детализация процесса формирования HTTP-ответа.

Рисунок 21 – Диаграмма декомпозиции процесса «Сформировать HTTP-ответ» TO-BE

Изменения данного уровня декомпозиции заключаются в изменении первого блока и замены механизма для него.

Ранее первый блок данного уровня назывался отвечал за преобразование тела ответа в JSON строку. После замены формата процесс остался тот же, однако данные преобразуются в бинарный формат.

Этот блок использует механизм библиотеки для сериализации и десериализации объекта в массив байт, а не стандартный инструмент внутри фреймворка Spring Boot.

В результате перехода к бинарному формату данных на всех диаграммах процесса обработки клиентского запроса были изменены блоки работы с JSON на блоки работы с бинарным форматом. Также все эти блоки используют не механизм Spring Boot, а механизм библиотеки сериализации.

2.3 Выбор технологий реализации библиотеки

В качестве языка программирования был выбран Java версии 21, которая является одной из версий с длительной поддержкой со стороны разработчиков языка. Выбор обусловлен высокой популярностью в качестве средства создания REST-сервисов.

Преимуществами Java являются:

* кроссплатформенность;
* популярность в сообществе;
* наличие механизмов управления памятью;
* строгая типизация;
* следование принципам объектно-ориентированного программирования;
* наличие сборщиков проекта;
* легкое подключение написанного кода в качестве зависимости в другие проекты.

Основным инструментов языка для реализации библиотеки является рефлексия.

*Рефлексия* — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов.

Для сборки реализуемой библиотеки в jar архив и последующего подключения к проектам в качестве зависимостей необходимо использовать сборщик. Для Java приложений традиционно используются сборщики Maven или Gradle. Maven считается более классическим и распространенным инструментом, поэтому для сборки библиотеки будет использован именно он.

Maven – это инструмент для управления проектами на JVM языках. Он позволяет автоматизировать процессы компиляции, упаковки и развертывания кода, а также берет на себя управление зависимостями приложения [15].

Maven отличается высокой степенью стандартизации проектов, автоматизацию типовых задач, большую экосистему плагинов и интеграцию с популярными средами разработок.

В качестве среды разработки будет использована свободно распространяемая IntelliJ IDEA Community Edition версии 2023.3.5. Средство предоставляет основной набор средств разработки, которых будет достаточно для реализации библиотеки [16].

Выбор технологий разработки – важный этап в реализации библиотеки. Выбранные инструменты позволят быстро реализовать и отладить написанный код. Выбранные инструменты заслуженно занимают место самых популярных инструментов среди Java разработчиков.

1. Экспериментальные исследования применения бинарного формата

3.1 Программная реализация библиотеки для сериализации и десериализации

### **3.1.1 Сериализация**

Для потребителей библиотеки доступны два перегруженных метода *getBytes* для сериализации объекта в массив байт. Методы поставляются классом *ByteConverter*.

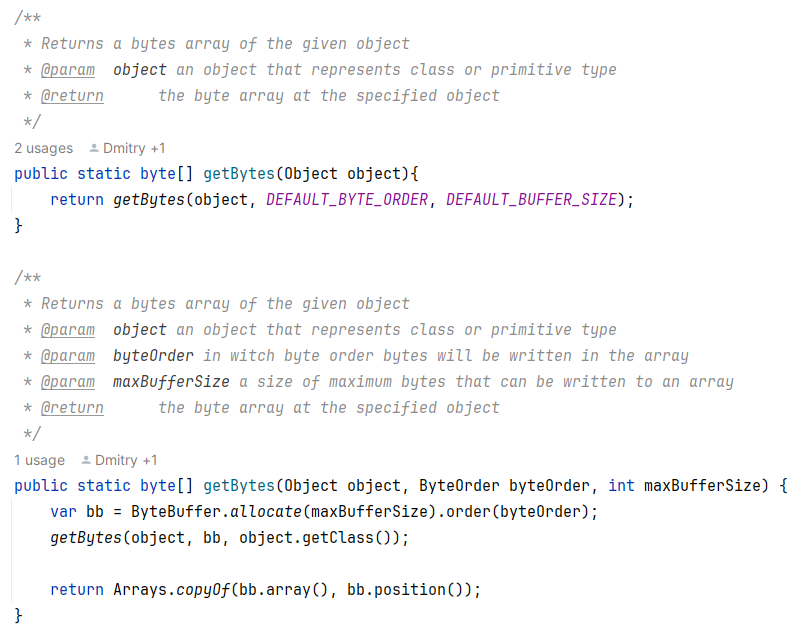
На рисунке 22 представлен исходный код этих методов.

Рисунок 22 – Клиентские методы для сериализации объектов

Метод *getBytes(Object object)* выполняет преобразование переданного ему объекта в массив байт. При выполнении используются настройки по умолчанию. Метод на вход принимает объект, который нужно преобразовать в массив байт. Возвращает рассматриваемый метод массив байт. Внутри метода используется две константы *DEFAULT\_BYTE\_ORDER* и *DEFAULT\_BUFFER\_SIZE*. Первая константа отвечает за порядок байтов по умолчанию (в разных операционных системах он может различаться), по умолчанию используется *ByteOrder.LITTLE\_ENDIAN* из пакета java.nio. Вторая константа определяет начальный размер буфера в байтах. Он необходим для временного хранения данных в процессе сериализации. Метод нужен для проведения сериализации с размером буфера и порядком байт по умолчанию. Рассматриваемый метод является перегруженной версией основного метода с идентичным названием, который будет рассмотрен далее.

Назначение метода *getBytes(Object object, ByteOrder byteOrder, int maxBufferSize)* аналогично первому методу. Однако данный метод принимает на вход три параметра. Первым параметром является object – объект, который нужно сериализовать в байты. Вторым параметров является *byteOrder* – порядок байт, определяющий как будут записаны в итоговый массив числовые значения. И третьим параметром является *maxBufferSize* – максимальный размер буфера в байтах, выделяемого под сериализацию. Рассматриваемый метод сначала создает буфер для работы с байтами, затем получает из переданного в параметрах объекта массив байт, после чего записывает и возвращает заполненную часть буфера. Заполнение только части буфера нужно, чтобы избежать лишних нулевых байтов в конце массива. Метод позволяет вручную настроить сериализацию объекта.

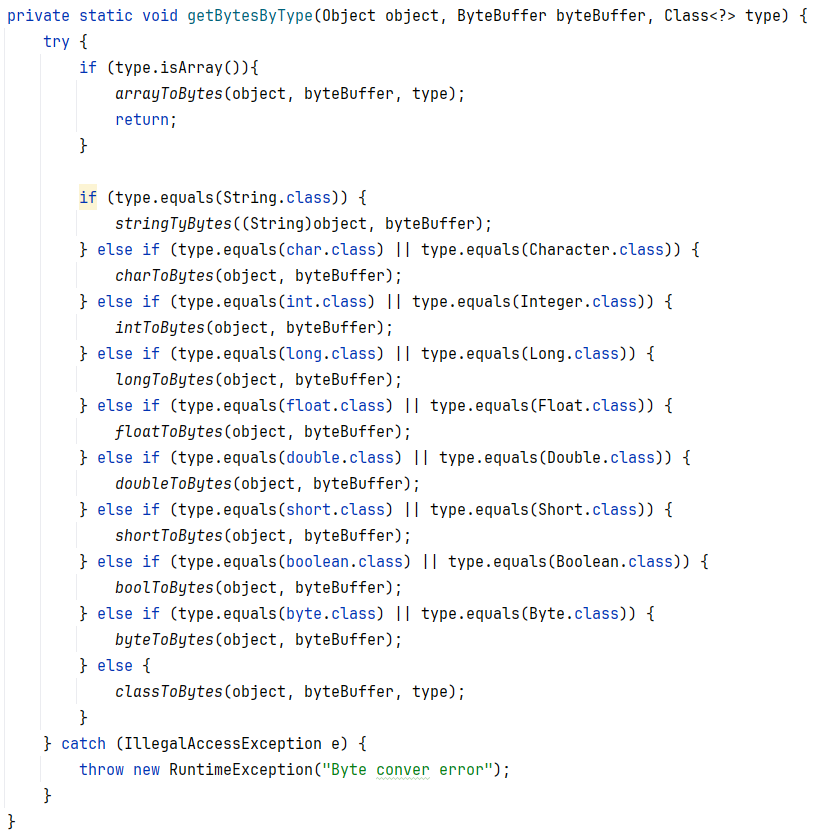
Далее на рисунке 23 представлен метод *getBytesByType*.

Рисунок 23 – Внутренний метод по обработке типов

Метод, представленный на рисунке, выполняет сериализацию объекта в байты в зависимости от его типа. Метод определяет, является ли объект примитивом или оберткой над примитивом, строкой, массивом или пользовательским классом. Затем вызывается соответствующий метод записи в буфер.

Метод принимает на вход три параметра. Первый параметр *object* – объект, который нужно сериализовать. Второй параметр *byteBuffer* – временный буфер для записи. Третий параметр *type* – класс объекта или его примитивный тип.

Методы, используемые для сериализации отдельных типов, поставляются отдельным классом *ByteBufferExtension*. В данном классе содержатся методы для сериализации всех примитивных типов, присутствующих в языке Java: Integer, Long, Float, Double, Short, Character, Boolean и Byte.

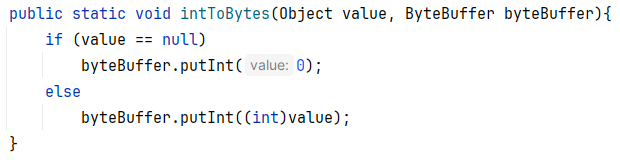
На рисунке 24 представлен один из таких методов.

Рисунок 24 – Метод для сериализации целочисленного типа

Метод *intToBytes* на рисунке выше приведен в качестве примера одного из методов для сериализации примитивов. Методы для других типов выглядят аналогично, но отличаются вызываемым методом у объекта *byteBuffer*.

На вход метод *intToBytes* принимает объект для сериализации и буфер, в который нужно записать байты. Если объект отсутствует (является null), то в буфер помещается значение 0.

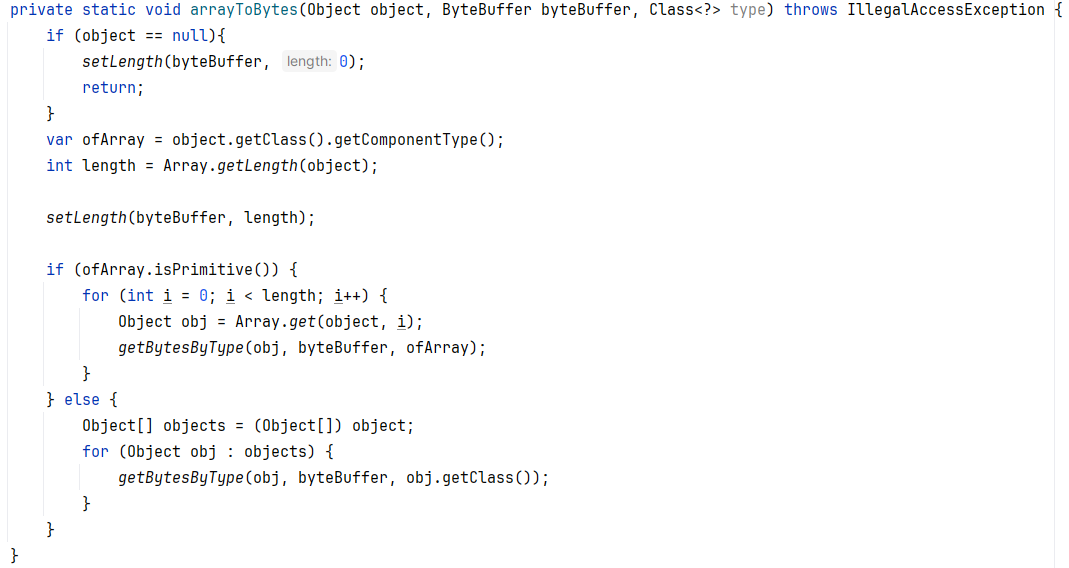
Для преобразования массивов или коллекций в массив байт используется отдельный метод *arrayToBytes*, представленный на рисунке 25.

Рисунок 25 – Метод для сериализации массивов

Метод для сериализации массивов поддерживает как массив примитивов, так и массив объектов. Сначала данный метод производит проверку сериализуемого объекта на null. Далее он определяет тип массива и его длину. После чего происходит запись длины массива во временный буфер. Производится проверка типа объектов в массиве. Если в массиве находятся примитивы, то производится перебор элементов и для каждого объекта вызывается метод *getBytesByType* для сериализации. Если в массиве находятся объекты, то массив приводится к типу массива Object, являющийся базовым классом для всех классов в Java. После преобразования также производится перебор и для каждого объекта вызывается метод *getBytesByType*. Стоит отметить, что рассматриваемый метод поддерживает и вложенные массивы.

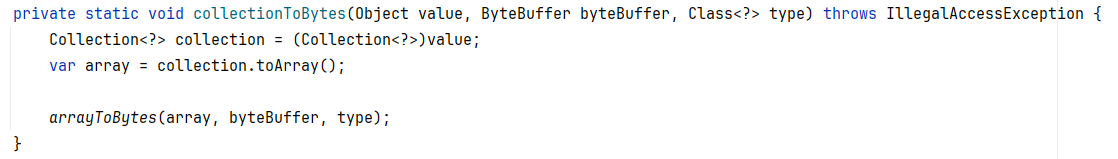
Для сериализации в байты коллекций существует отдельный внутренний метод *collectionToBytes*, представленный на рисунке 26.

Рисунок 26 – Метод для сериализации коллекций

Метод *collectionToBytes* полученный в параметрах объект к типу Collection, затем преобразует коллекцию в массив с помощью поставляемого JDK метода *toArray*. После чего коллекция в виде массива обрабатывается как обычный массив. Это позволит поддерживать все существующие типы коллекций. При сериализации учитывается и тип объектов, лежащих внутри коллекции.

Еще одним внутренним методом для сериализации является метод *classToBytes*. Данный метод необходим для сериализации пользовательским классов, которыми на практике являются полезные данные – тело запроса и тело ответа, которые также могут содержать в себе другие пользовательские классы.

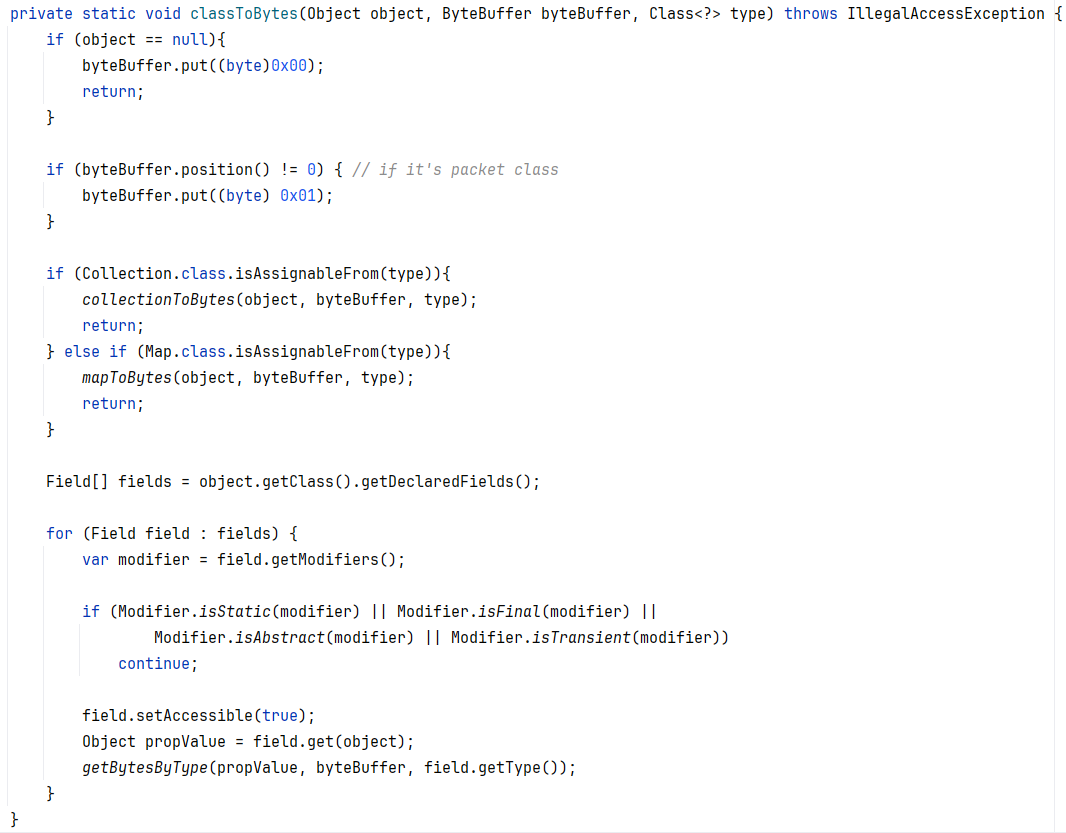
На рисунке 27 представлен исходный код данного метода.

Рисунок 27 – Метод для сериализации пользовательских классов

На вход метод принимает объект для сериализации, буфер для записи и тип объекта.

Сначала метод *classToBytes* обрабатывает null значения, затем маркирует пакет, если позиция в буфере не нулевая. После чего происходит проверка и в случае успешной проверки обработка ассоциативных массивов. Если тип является коллекцией, то вся дальнейшая обработка делегируется методу *collectionToBytes*, а если словарем, то методу *mapToBytes*, который будет рассмотрен позднее. Если сериализуемый объект не является словарем или коллекцией, то происходит извлечение всех его полей. Затем берутся все значения полученных полей и записываются в буфер с помощью метода *getBytesByType*.

Рассматриваемый метод является ключевой частью механизма сериализации.

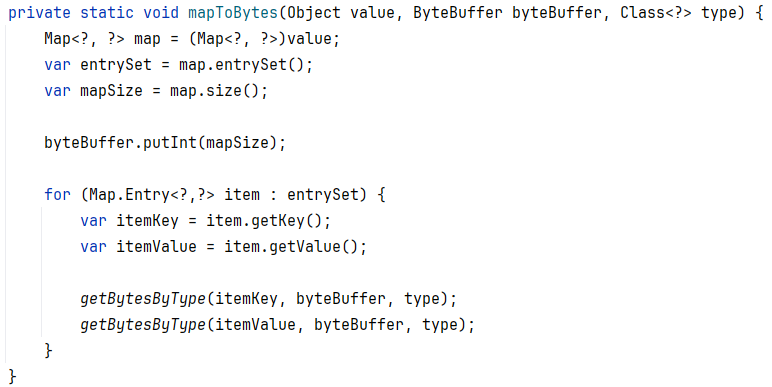
Также необходимо рассмотреть метод *mapToBytes*, исходный код которого представлен на рисунке 28.

Рисунок 28 – Метод для сериализации словарей

Данный метод выполняет сериализацию объекта ассоциативный массив. Он принимает три параметра: объект типа Map для сериализации, буфер и тип объектов, лежащих в Map.

Сначала метод производит приведение типа к Map, затем подготавливает данные – пустые хранилища для ключей, значений, а также размер ассоциативного массива. После чего происходит запись размера. Затем происходит перебор и сериализация (зависимости от типа объектов) пар ключ-значение из словаря и в первую очередь записывается ключ, а во вторую очередь записывается значение.

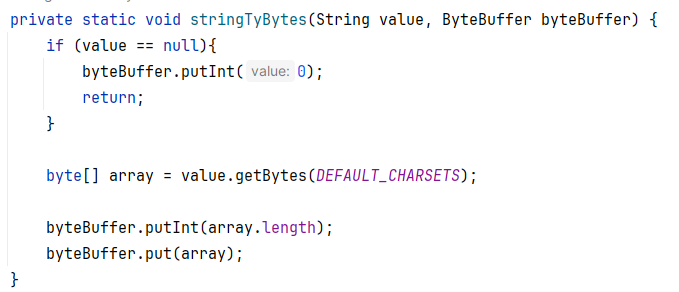
Также необходимо рассмотреть метод для сериализации строк, представленный на рисунке 29.

Рисунок 29 – Метод для сериализации строк

Особенность данного метода заключается в том, что строка с помощью поставляемого Java методом преобразуется в байты и при этом используется кодировка UTF-8. Байты помещаются во временный буфер.

После рассмотрения основных частей исходного кода для десериализации в реализуемой библиотеки можно отметить:

* поддерживаются все типы значений в Java, в том числе примитивы и их обертки;
* поддерживаются пользовательские классы, в том вложенные;
* обрабатываются null значения;
* можно задать настройки сериализации, к таковым относятся – порядок байт и размер временного буфера.

Данная библиотека позволит удобно преобразовывать в байты основные структуры данных, предоставляемые языком Java и его пакетом java.util.Collections.

### **3.1.2 Десериализация**

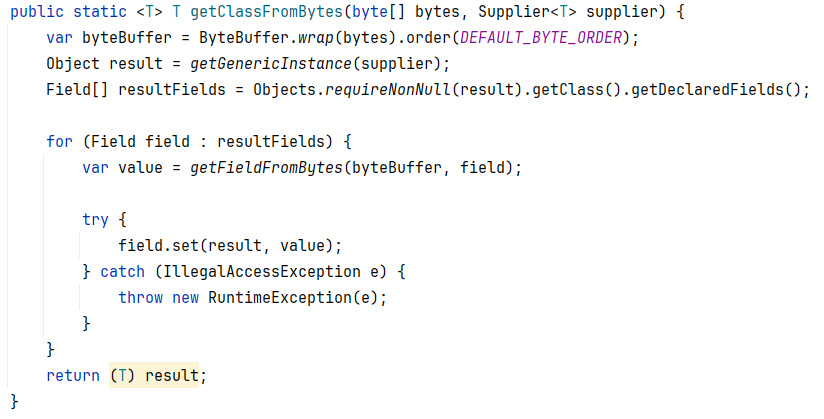
За преобразование массива байт в класс в библиотеке отвечает класс *getClassFromBytes*. Данный класс представлен на рисунке 30.

Рисунок 30 – Клиентский метод для десериализации

Метод *getClassFromBytes* принимает на вход два параметра. Первым параметров является массив байт *bytes*. Вторым параметром является поставщик *supplier* типа Supplier. Поставщик нужен для объекта, в который будут записаны восстановленные значения [17].

Первым шагом для десериализации является подготовка буфера, в которой используется тот же порядок байт, который был задан при сериализации.

С помощью метода *getGenericInstance*. Данный метод будет рассмотрен позднее.

После создания экземпляра нового объекта у него достаются все его поля с помощью метода рефлексии *getDeclaredFields*, поставляемого пакетом java.lang из Java.

Далее полученные поля перебираются в цикле и для каждого из полей получается значение из буфера с помощью метода *getFieldBytes*, а затем устанавливается значение этого поля.

В конце работы возвращается результат, которым является класс, описывающий структуру данных в массиве байт [18].

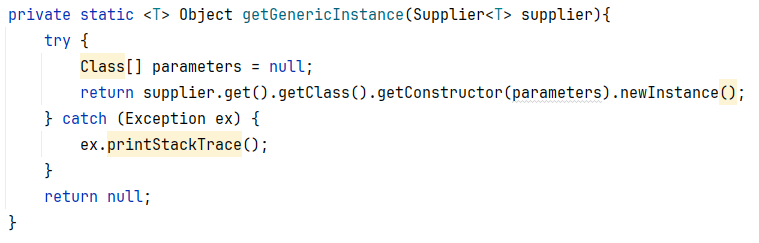
Одним из важнейших внутренних методов для десериализации является *getGenericInstance*, представленный на рисунке 31.

Рисунок 31 – Метод создания нового экземпляра класса

Данный метод создает новый экземпляр класса с использованием механизма рефлексии, основываясь на переданном поставщике *supplier*. При создании нового объекта используется конструктор по умолчанию.

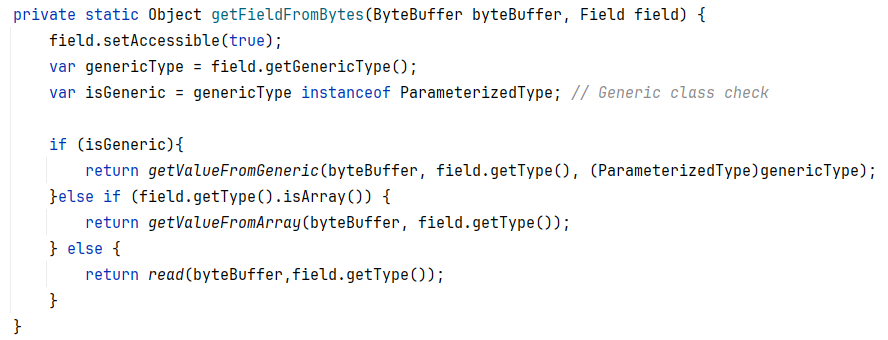
Другим важным методом при десериализации является *getFieldFromBytes*. Метод приведен на рисунке 32.

Рисунок 32 – Метод десериализации поля из массива байт

Метод выполняет восстановления значения поля из буфера. На метод вход принимает буфер с сериализованными данными и поле объекта, которое нужно восстановить. Сначала метод подготавливает поле, затем анализирует поле и после этого выбирает стратегию десериализации.

Метод *getFieldFromBytes* имеет гибкую обработку, поддерживая разные типы полей, для каждого из которых в зависимости от типа вызывается соответствующий внутренний метод.

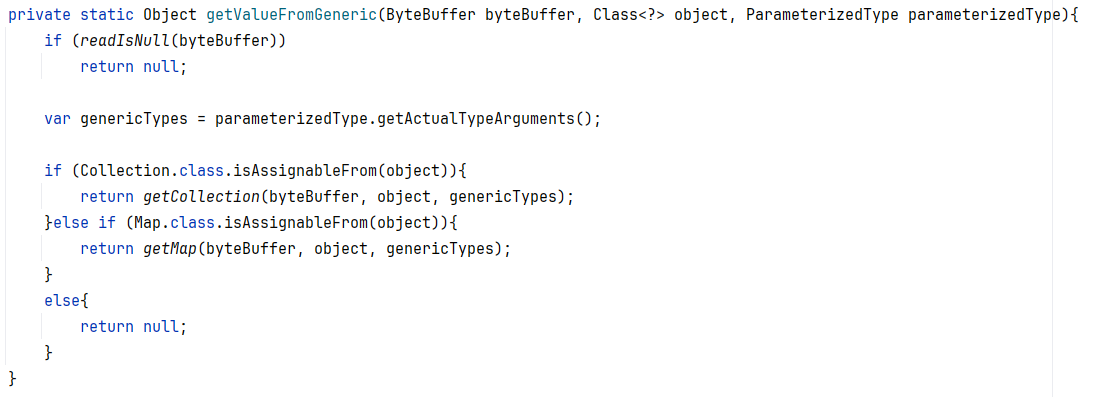
Внутренним методом для десериализации параметризированных типов является *getValueFromGeneric*, представленный на рисунке 33.

Рисунок 33 – Метод десериализации параметризированных типов

Метод *getValueFromGeneric* восстанавливает значения для коллекций, ассоциированных массивов.

Данный метод на вход принимает буфер с байтами, класс восстанавливаемого объекта и информацию о параметризированном типе.

Сначала он производит проверку на null, затем получает тип параметра, после чего определяет, является ли объект коллекцией или ассоциативным массивом. В зависимости от типа объекта вызывается соответствующий внутренний метод *getCollection* или *getMap*.

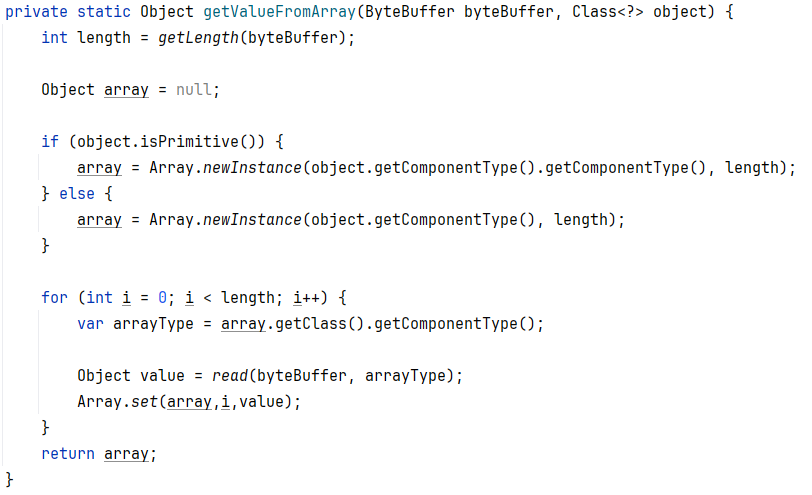
На рисунке 34 представлен метод для десериализации массива.

Рисунок 34 – Метод десериализации массива

Метод *getValueFromArray* выполняет десериализацию массива из байтового буфера, корректно обрабатывая массивы разных типов.

На вход данный метод принимает буфер с байтами и тип объектов в массиве.

Сначала происходит чтение длины массива, затем на её основа создается новый массив. Массив заполняется значениями. Полученный результат возвращается в точку вызова метода *getValueFromArray*.

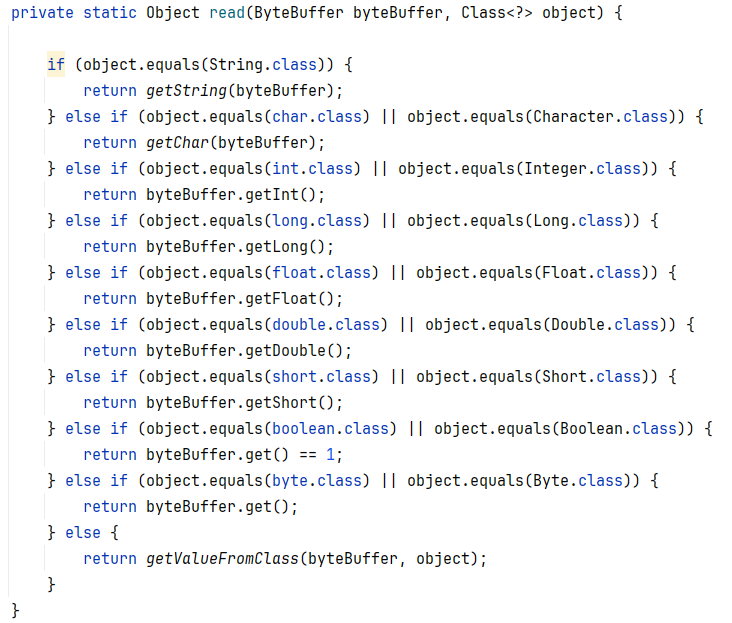
Последним методом для десериализации является метод read, приведенный на рисунке 35.

Рисунок 35 – Метод десериализации примитивов

Метод read является центральным десериализатором примитивных типов. Он восстанавливает объект из буфера с байтами.

На вход метод принимает байтовый буфер и класс целевого типа десериализации.

В первую очередь выполняется определение типа через каскад if-else проверок. После чего, в зависимости от типа значения, вызывается соответствующий этому типу метод восстановления значения. При этом обработка сложных типов делегируется методу *getValueFromClass* [19].

Поставляемые библиотекой методы для десериализации позволяют удобно с помощью вызова одного метода восстановить всю структуру сериализованных данных. Поэтому необходимо соблюдать согласованность контракта между клиентом и сервером.

Библиотека поддерживает большое количество типов в Java.

При этом библиотека разработана на одной из поддерживаемых на данный момент версий Java и использует для своей работы современные механизмы самого языка.

Исходный код библиотеки для бинарной сериализации приведен в приложении А.

3.2 Описание процесса апробации

### **3.2.1 Определение критериев и показателей оценки эффективности**

Системы обработки данных предъявляют высокие требования к скорости передачи и эффективности хранения информации [20]. Текстовые форматы обеспечивают удобство чтения и гибкость структуры данных. Они могут становиться узким местом при высокой [21].

Данный раздел рассматривает процесс тестирования применения бинарного формата. Главной целью использования данного формата является оптимизация производительности системы.

Для тестирования предлагаемого решения необходимо определить:

* критерии оценивания;
* показатели оценки;
* технологии, применяемые в тестировании.

Применение бинарного формата должно привести к следующим результатам:

* система должна быстрее получать ответ на запрос;
* размер получаемого ответа должен уменьшиться;
* система должна обрабатывать больше запросов за то же время;

Исходя из этих критериев оценивания можно выделить ряд показателей оценки эффективности предлагаемого решения. К таким показателям можно отнести:

* среднее время ответа на запрос;
* средний размер получаемых данных;
* количество запросов в секунду (пропускная способность).

В результате тестирования решения ожидается, что среднее время ответа на запрос и средний размер получаемых данных должны уменьшиться, а количество запросов в секунду должно возрасти.

### **3.2.2 Определение технологий тестирования**

Основным инструментом проверки является инструмент нагрузочного тестирования. Данное программное обеспечение позволяет поместить систему в условия реальной нагрузки с большим количеством как пользователей, так и посылаемых ими запросов. Большинство из таких инструментов предоставляют нужные в рамках исследования показатели, поэтому в длительном и тщательном выборе конкретного решения нет необходимости. Одним из таких решений является JMeter версии 5.6.3.

JMeter – это инструмент для нагрузочного тестирования. В программе реализованы механизмы авторизации виртуальных пользователей, поддерживаются пользовательские сеансы. Организовано логирование результатов теста и разнообразная визуализация результатов в виде диаграмм, таблиц и т. п. [22].

JMeter позволяет проводить нагрузочные тесты, стресс-тесты, тестирование стабильности, поддерживает различные сетевые протоколы, достаточно гибкий за счет плагинов и применения пользовательских скриптов, а также позволяет анализировать результаты тестирования в виде графиков, таблиц, отчетов и различного набора метрик [23].

Для тестирования было создано тестовое приложение. Оно создано с помощью языка Java версии 21 и фреймворк Spring Boot [24]. К приложению подключена библиотека для сериализации.

### **3.2.3 Описание тестового Spring Boot приложения**

Приложение имеет две конечные точки (эндпоинта), которые отправляют один и тот же объект, но в разных форматах [25]. Так, по пути «/server/json» в ответ на запрос будет отправляться объект в формате json, а по пути «/server/binary» будет отправляться тот же самый объект, но в бинарном формате.

В качестве отправляемого объекта был создан DTO-класс. Он содержит в себе поля разных типов [26]. Исходный код класса представлен на рисунке 36.

Рисунок 36 – Класс тела ответа от тестового приложения

Отправляемый объект содержит как простые значения (числовые, строковые), так и вложенные сложные объекты со своей структурой [27].

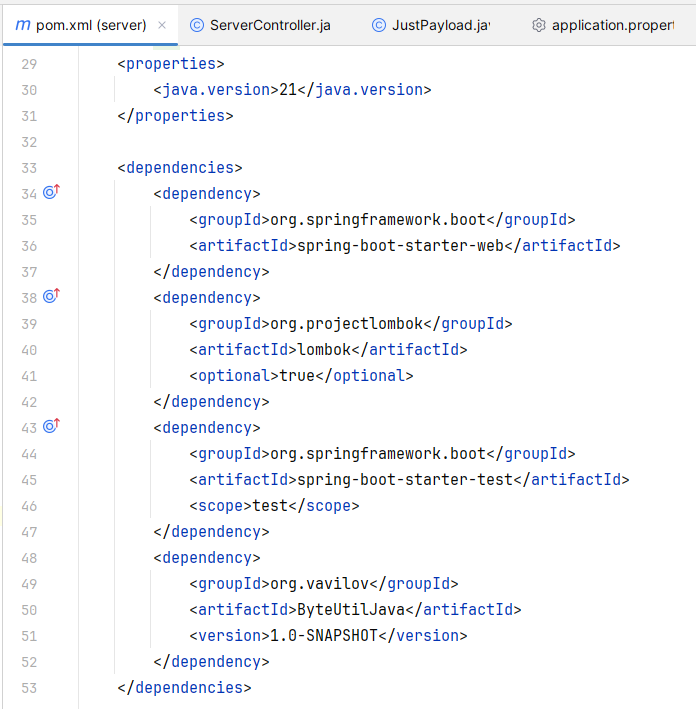
К тестовому приложению с помощью сборщика Maven был подключен Spring Boot Starter, библиотека Lombok, Spring Boot Starter и реализованная ранее библиотека для сериализации. Список зависимостей в файле pom.xml представлен на рисунке 37.

Рисунок 37 – Список зависимостей тестового приложения

Реализованное тестовое приложение моделирует поведение приложений на Spring Boot и Java. При этом приложение отправляет один и тот же объект как в формате JSON, так и в бинарном формате, что позволит произвести параллельное нагрузочное тестирование с одними и теми же данными и выявить связь между показателями эффективности и применяемым форматом передачи данных. Исходный код тестового приложения представлен в приложении Б.

### **3.2.4 Описание тестового плана JMeter**

Для имитации нагрузки на приложение составлен тестовый план. Он описывает показатели нагрузки и HTTP-запросы к приложению. Запросы отправляются на веб-приложение в рамках нагрузочного тестирования [28].

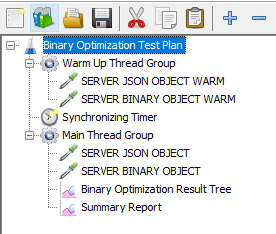
На рисунке 38 продемонстрирован состав объектов тестового плана.

Рисунок 38 – Тестовый план для нагрузочного тестирования

Представленный на рисунке выше тестовый план состоит из двух основных частей.

К первой части относится группа потоков с названием Warm Up Thread Group. Данная группа потоков содержит в себе два HTTP-запроса: один на получение данных в JSON, другой на получение данных в бинарном формате. Каждый из запросов отправляется один раз, после чего выполнение тестового плана продолжается в рамках второй группы потоков. Первая группа потоков необходима для исключения из итоговых показателей первых запросов к ресурсам, так как Spring Boot приложение инициализирует ряд внутренних объектов из-за ленивой инициализации, поэтому выполнение таких запросов обычно занимает больше времени, чем все последующие аналогичные запросы. Результаты первых запросов в общем статистике будут являться аномалиями и исказят общую картину происходящего [29].

Ко второй части относится группа потоков Main Thread Group. Данная группа потоков содержит в себе два HTTP-запросы. Первый запрос (SERVER JSON OBJECT) получает данные от сервера в JSON формате, второй запрос (SERVER BINARY OBJECT) в бинарном формате. В рассматриваемой группе потоков содержатся и два слушателя, которые аккумулируют статистику запросов и их показателей во время выполнения тестового плана. Первым слушателем является объект Binary Optimization Result Tree, в который в процессе выполнения тестового плана записываются данные каждого из запросов, отправляемых в рамках тестирования. Вторым слушателем является Summary Report. Он формирует таблицу с агрегированными результатами тестирования. В таблице содержатся показатели, необходимые для определения эффективности решения.

Также в настройках этой группы потоков был конфигурирован ряд параметров, определяющих нагрузку на приложение [30]. В таблице 3 приведены показатели нагрузки тестового плана и их значения.

Таблица 3 – Показатели нагрузки для нагрузочного тестирования

|  |  |  |
| --- | --- | --- |
| Показатель | Описание | Значение |
| Number of Threads (users) | Количество виртуальных пользователей, шт. | 100 |
| Ramp-Up Period | Время подключения пользователей, сек. | 30 |
| Loop Count | Количество повторений, шт. | 10 |

Показатели нагрузки представляют из себя довольно высокую единовременную нагрузку на приложение с плавным ростом количества запросов на довольно длительном промежутке времени.

Сформированный тестовый план определяет размер нагрузки на тестовое приложение, необходимый для определения эффективности оптимизации нагрузкой, а также предоставит результат тестирования в виде таблицы. Содержание исходных файлов тестового плана в формате JMX представлены в приложении В.

Нагрузочное тестирование проводилось на машине с центральным процессором Intel Core i3-10100F. Размер оперативной памяти составляет 16 Гб DDR4 и частотой 2666 МГц. Операционной системой является Windows 10 Pro 64 бит.

3.3 Анализ результатов тестирования

После определения необходимых показателей оценки эффективности и описания условий эксперимента было проведено нагрузочное тестирование. В результате тестирования сформирован отчет с таблицей, содержащей необходимые критерии оценки эффективности.

Тестирование состояло из двух последовательных сценариев нагрузки на тестовое приложение. В ходе первого сценария была отправлена одна тысяча запросов на эндпоинт, отправляющий ранее описанный ответ в формате JSON. В ходе второго сценария также была отправлена одна тысяча запросов на другой эндпоинт, отправляющий в ответе тот же объект, но в бинарном формате.

Результаты нагрузочного тестирования представлены в таблице 4.

Таблица 4 – Результаты нагрузочного тестирования

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Используемый в передаче данных формат | Среднее время выполнения запроса, мс. | Минимальное время выполнения запроса, мс. | Максимальное время выполнения запроса, мс. | Пропускная способность, КБ/сек | Средний размер ответа, байт. |
| Binary | 31 | 4 | 83 | 25,6 | 33935 |
| JSON | 45 | 5 | 286 | 23,8 | 83841,9 |

При анализе среднего времени выполнения запросов было установлено, что эндпоинт, отправляющий данные в бинарном формате, работает примерно на 45% быстрее.

Минимальное время выполнения запроса для разных форматов практически не отличается, а разница граничит на уровне погрешности. При этом максимальное время выполнения запроса для JSON формата намного выше, чем для бинарного. Разница составляет 203 миллисекунды.

На рисунке 39 представлена диаграмма с различными показателями времени сравниваемых форматов.

Рисунок 39 – Сравнение среднего, минимального и максимального времени

Двоичный формат обеспечивает более предсказуемую и быструю обработку запросов.

На рисунке 40 представлена диаграмма со сравнением показателей пропускной способности и среднего размера ответа сравниваемых форматов.

Рисунок 40 – Сравнение пропускной способности и размера ответа

Сравнивая результаты пропускной способности, можно отметить, что применение бинарного формата эффективнее на 7%. В рамках подаваемой на приложение нагрузки это довольно небольшая разница, но при кратно большем количестве запросов разница в 7% была бы более существенной.

Средний размер равен примерно 33,9 КБ, а для бинарного формата и 83,8 КБ для JSON, что приблизительно в 2,5 раза больше. Можно сделать вывод, что бинарный формат значительно компактнее и его использование снижает нагрузку на сеть.

Применение бинарного формата уменьшает время на выполнение запроса. Обработка запросов становится более плавной и предсказуемой. Данные в ответе на запрос становятся более компактными. Размер данных значительно отличается от данных в формате JSON. Также немного увеличивается пропускная способность сети, в рамках которой происходит передача данных.

Эффект стал достижим за счет применения Java библиотеки для сериализации объектов. Библиотека может работать со списками, ассоциативными словарями и вложенными объектами сложной структуры, что крайне важно в рамках продуктовой разработки, в которой зачастую используются все перечисленные структуры данных.

Дальнейшие доработки библиотеки могут включать в сбея поддержку других структур данных. Помимо этого, можно добавить в библиотеку поддержку её конфигурации через файл настроек приложения, а также разместить библиотеку в открытом Maven репозитории.

ЗАКЛЮЧЕНИЕ

При анализе клиент-серверного взаимодействия между компонентами системы, построенной на микросервисной архитектуре, была выявлена проблема возврастающего времени работы всей системы при увеличении количества взаимодействуюших между собой микросервисов. Обнаружение данной особенности определило тему ВКР, а также составить для работы соответствующие цели и задачи.

В ходе исследования вышеописанной проблемы были выявлены способы увеличения произвоидтельности клиент-серверных приложений и проведен анализ популярных форматов данных при передаче данных. Процессы получения данных с сервера и обработки входящего запроса сервером были описаны в соответствующих AS-IS моделях. Был проведен анализ существующих Java-библиотек для сериализации и десериализации данных. Дополнительно были составлен ряд моделей для более детального представления взаимодействия между клиентом и сервером.

На этапе проектирования были составлены TO-BE модели процессов получения данных с сервера и обработки входящего запроса. Были выбраны необходимые технологии для реализации и отладки библиотеки.

Была реализована Java-библиотека для сериализации данных. Проведено нагрузочное тестирование на тестовом Spring Boot приложении.

Применение бинарного формата данных сокращает размер передаваемых данных, увеличивает пропускную способность сервисов и уменьшает время получения ответа на запрос.

Итогом данной работы стали реализованная Java-библиотека для сериализации данных. Использование библиотеки и переход к двоичному формату уменьшают время получения данных и экономят ресурсы системы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кочер, П. С. Микросервисы и контейнеры Docker : руководство / П. С. Кочер ; перевод с английского А. Н. Киселева. — Москва : ДМК Пресс, 2019. — 240 с. — ISBN 978-5-97060-739-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/123710 (дата обращения: 01.07.2025).
2. Боженко, В. В. Клиент-серверные информационные системы : учебное пособие / В. В. Боженко. — Санкт-Петербург : ГУАП, 2022. — 80 с. — ISBN 978-5-8088-1706-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/340925 (дата обращения: 01.07.2025).
3. Сычев, О. А. Лабораторный практикум по дисциплине «Операционные системы». Клиент-серверные приложения : учебно-методическое пособие / О. А. Сычев, Е. Д. Беришева. — Волгоград : ВолгГТУ, 2019. — 64 с. — ISBN 978-5-9948-3440-4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/157227 (дата обращения: 02.07.2025).
4. Романова, М. В. Информатика : учебное пособие / М. В. Романова, Е. П. Романов. — 3-е изд., стер. — Москва : ФЛИНТА, 2023. — 190 с. — ISBN 978-5-9765-3791-0. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/348287 (дата обращения: 02.07.2025).
5. Поллард, Б. HTTP/2 в действии : руководство / Б. Поллард ; перевод с английского П. М. Бомбаковой. — Москва : ДМК Пресс, 2021. — 424 с. — ISBN 978-5-97060-925-5. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/241037 (дата обращения: 02.07.2025).
6. Миндалёв, И. В. Моделирование бизнес-процессов с помощью IDEF0, DFD, BPMN за 7 дней : учебное пособие / И. В. Миндалёв. — Красноярск : КрасГАУ, 2016. — 123 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/103833 (дата обращения: 03.07.2025).
7. Васюткина, И. А. Разработка серверной части web-приложений на Java : учебное пособие / И. А. Васюткина. — Новосибирск : НГТУ, 2021. — 83 с. — ISBN 978-5-7782-4394-1. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/216155 (дата обращения: 03.07.2025).
8. Разработка серверной части web-приложения на базе Spring : методические указания / составители С. А. Коваленко [и др.]. — Воронеж : ВГТУ, 2023. — 35 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/383252 (дата обращения: 04.07.2025).
9. Заяц, А. М. Основы WEB технологий. Разработка WEB-приложений современными инструментальными средствами : учебно-методическое пособие / А. М. Заяц. — Санкт-Петербург : СПбГЛТУ, 2021. — 116 с. — ISBN 978-5-9239-1269-2. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/191164 (дата обращения: 04.07.2025).
10. Пономарчук, Ю. В. Программирование на языке Java : учебное пособие / Ю. В. Пономарчук, И. В. Кузнецов. — Хабаровск : ДВГУПС, 2021. — 103 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/259451 (дата обращения: 05.07.2025).
11. Смоленцева, Т. Е. Проектирование и разработка WEB-приложений: Практикум : учебное пособие / Т. Е. Смоленцева. — Москва : РТУ МИРЭА, 2023. — 68 с. — ISBN 978-5-7339-1759-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/368954 (дата обращения: 06.07.2025).
12. Курбатова, И. В. Основы программирования на языке Java : учебное пособие для спо / И. В. Курбатова, А. В. Печкуров. — Санкт-Петербург : Лань, 2024. — 348 с. — ISBN 978-5-507-48516-1. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/385925 (дата обращения: 06.07.2025).
13. Карнелл, Д. Микросервисы Spring / Д. Карнелл, И. У. Санчес ; перевод с английского А. Н. Киселева. — Москва : ДМК Пресс, 2022. — 490 с. — ISBN 978-5-97060-971-2. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/241172 (дата обращения: 07.07.2025).
14. Вязовик, Н. А. Программирование на Java : учебное пособие / Н. А. Вязовик. — 2-е изд. — Москва : ИНТУИТ, 2016. — 603 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/100405 (дата обращения: 07.07.2025).
15. Куликова, И. В. Программное обеспечение облачных и распределённых вычислительных систем : учебное пособие / И. В. Куликова. — Москва : РТУ МИРЭА, 2025 — Часть 1 — 2025. — 86 с. — ISBN 978-5-7339-2497-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/493544 (дата обращения: 08.07.2025).
16. Альтман, Е. А. Платформа IntelliJ IDEA для разработки программ : учебно-методическое пособие / Е. А. Альтман, А. В. Александров, Т. В. Васеева. — Омск : ОмГУПС, 2020. — 37 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/165620 (дата обращения: 08.07.2025).
17. Объектно-ориентированное программирование : учебное пособие / В. В. Лозовский, Е. Н. Штрекер, Е. С. Данилович [и др.]. — Москва : РТУ МИРЭА, 2025. — 484 с. — ISBN 978-5-7339-2498-4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/493547 (дата обращения: 08.07.2025).
18. Давыдова, Н. А. Программирование : учебное пособие / Н. А. Давыдова, Е. В. Боровская. — 5-е изд. (эл.). — Москва : Лаборатория знаний, 2025. — 241 с. — ISBN 978-5-93208-831-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/451607 (дата обращения: 09.07.2025).
19. Свердлов, С. З. Языки программирования и методы трансляции : учебное пособие для вузов / С. З. Свердлов. — 5-е изд., стер. — Санкт-Петербург : Лань, 2025. — 564 с. — ISBN 978-5-507-50570-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/447398 (дата обращения: 09.07.2025).
20. Баланов, А. Н. Построение микросервисной архитектуры и разработка высоконагруженных приложений : учебное пособие для СПО / А. Н. Баланов. — 2-е изд., стер. — Санкт-Петербург : Лань, 2025. — 80 с. — ISBN 978-5-507-52412-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/450911 (дата обращения: 09.07.2025).
21. Тонкович, И. Н. Технологии проектирования сложных информационных систем : учебно-методическое пособие / И. Н. Тонкович, А. В. Шелест. — БГУИР : БГУИР, 2025. — 167 с. — ISBN 978-985-543-779-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/479630 (дата обращения: 09.07.2025).
22. Баланов, А. Н. Бэкенд-разработка веб-приложений: архитектура, проектирование и управление проектами : учебное пособие для вузов / А. Н. Баланов. — 2-е изд., стер. — Санкт-Петербург : Лань, 2025. — 312 с. — ISBN 978-5-507-52472-3. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/451820 (дата обращения: 10.07.2025).
23. Турнецкая, Е. Л. Программная инженерия. Интеграционный подход к разработке : учебник для вузов / Е. Л. Турнецкая, А. В. Аграновский. — 2-е изд., стер. — Санкт-Петербург : Лань, 2025. — 216 с. — ISBN 978-5-507-50848-8. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/480161 (дата обращения: 10.07.2025).
24. Федоричев, Л. А. Реализация многопоточности в языке Java : учебное пособие для вузов / Л. А. Федоричев, О. В. Букунова. — 2-е изд., стер. — Санкт-Петербург : Лань, 2025. — 72 с. — ISBN 978-5-507-52722-9. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/457502 (дата обращения: 10.07.2025).
25. Золкин, А. Л. Проектирование и конфигурация компьютерных сетей с внедрением микросервисной архитектуры : учебное пособие для СПО / А. Л. Золкин, В. Д. Мунистер. — Санкт-Петербург : Лань, 2025. — 120 с. — ISBN 978-5-507-50268-4. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/447200 (дата обращения: 10.07.2025).
26. Баланов, А. Н. Комплексное руководство по разработке: от мобильных приложений до веб-технологий : учебное пособие для вузов / А. Н. Баланов. — 2-е изд., стер. — Санкт-Петербург : Лань, 2025. — 412 с. — ISBN 978-5-507-53193-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/478178 (дата обращения: 11.07.2025).
27. Programming, Development, Software: An English Reader (Программирование, разработка, программное обеспечение) : хрестоматия / О. С. Абайдуллина, Н. Е. Иоффе, К. У. Мамаева [и др.]. — Москва : РТУ МИРЭА, 2025. — 185 с. — ISBN 978-5-7339-2465-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/493409 (дата обращения: 11.07.2025).
28. Игнатьев, А. В. Тестирование программного обеспечения : учебное пособие для вузов / А. В. Игнатьев. — 4-е изд., стер. — Санкт-Петербург : Лань, 2025. — 56 с. — ISBN 978-5-507-50858-7. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/481331 (дата обращения: 11.07.2025).
29. Баланов, А. Н. DevOps: интеграция и автоматизация : учебное пособие для вузов / А. Н. Баланов. — 2-е изд., стер. — Санкт-Петербург : Лань, 2025. — 240 с. — ISBN 978-5-507-50491-6. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/440162 (дата обращения: 11.07.2025).
30. Турнецкая, Е. Л. Программная инженерия. Тестирование и контроль качества программного обеспечения : учебное пособие для вузов / Е. Л. Турнецкая, А. В. Аграновский. — Санкт-Петербург : Лань, 2025. — 172 с. — ISBN 978-5-507-51677-3. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/455672 (дата обращения: 12.07.2025).

ПРИЛОЖЕНИЕ А

(обязательное)

Программный код библиотеки для бинарной сериализации

ПРИЛОЖЕНИЕ Б

(обязательное)

Программный код тестового Spring Boot приложения

ПРИЛОЖЕНИЕ В

(обязательное)

Описание тестового плана для нагрузочного тестирования