СТАТЬЯ

УДК 004.056.5

АНТИФРОД-СИСТЕМА: РАЗРАБОТКА ПРОГРАММНОГО КОМПЛЕКСА ДЛЯ ПРЕДОТВРАЩЕНИЯ МОШЕННИЧЕСКИХ ТРАНЗАКЦИЙ

Дубровский А. А., Малёшина Л. М. ORCID ID 0000-0002-4876-462X

Федеральное государственное автономное образовательное учреждение высшего образования «Российский университет транспорта» (МИИТ), Москва, Российская Федерация, e-mail: docentmlm@mail.ru

Предотвращение мошенничества в онлайн-платежах является сложным процессом, но он имеет решающее значение для успеха компаний по всему миру. Внедрение антифрод-систем позволяет выявлять угрозы на разных уровнях, снижать риски и делать транзакции более безопасными. В России такие системы получили широкое распространение благодаря усилению регуляторных требований и активному развитию онлайн-платежей. Цель исследования – создать систему, состоящую из нескольких взаимодействующих сервисов, для автоматической проверки транзакций на возможные признаки мошенничества. В ходе разработки системы была построена архитектура, состоящая из нескольких взаимодействующих сервисов, каждый из которых выполняет конкретную задачу в рамках общей системы. Были использованы такие технологии, как gRPC для обмена сообщениями между сервисами и RabbitMQ для асинхронной обработки транзакций. Валидация транзакций и проверки на мошенничество реализованы на gRPC сервере с использованием собственной логики валидации и интеграции с базой данных, обеспечивающей хранение черного списка пользователей. Главный сервис системы предоставляет пользователям два ключевых интерфейса для взаимодействия с системой: REST API с поддержкой HATEOAS и GraphQL API. REST API с HATEOAS обеспечивает гибкость и удобство, позволяя пользователям получать не только данные о транзакциях и результатах проверок, но и ссылки на возможные дальнейшие действия, что упрощает навигацию и интеграцию с другими сервисами.

Ключевые слова: онлайн-транзакции, мошенничество в онлайн-платежах, защита данных, антифрод-система

ANTIFROD SYSTEM: DEVELOPMENT OF A SOFTWARE COMPLEX TO PREVENT FRAUDULENT TRANSACTIONS

Dubrovskiy A. A., Maleshina L. M. ORCID ID 0000-0002-4876-462X

Federal State Autonomous Educational Institution of Higher Education "Russian University of Transport (MIIT)", Moscow, Russian Federation, e-mail: docentmlm@mail.ru

Preventing online payment fraud is a complex process, yet it is crucial to the success of companies worldwide. Implementing anti-fraud systems allows for the detection of threats at multiple levels, mitigating risks, and making transactions more secure. In Russia, such systems have become widespread due to tightened regulatory requirements and the rapid growth of online payments. The aim of the research is to create a system consisting of several interacting services to automatically check transactions for possible signs of fraud. During the system's development, an architecture was built consisting of several interacting services, each performing a specific task within the overall system. Technologies such as gRPC for messaging between services and RabbitMQ for asynchronous transaction processing were used. Transaction validation and fraud checks were implemented on the gRPC server using custom validation logic and integration with a database that stores a user blacklist. The system's main service provides users with two key interfaces for interacting with the system: a REST API with HATEOAS support and a GraphQL API. The REST API with HATEOAS provides flexibility and convenience, allowing users to access not only transaction data and audit results but also links to possible further actions, simplifying navigation and integration with other services.

Keywords: online transactions, online payment fraud, data protection, anti-fraud system

Введение

«Мошенничество в онлайн-платежах стало одной из главных сложностей для компаний, которые работают в сфере финансовых услуг. Потери средств от мошенничества могут исчисляться в миллиардах долларов, и с каждым годом способы мошенников становятся все более сложными» [1]. Мошенники продолжают придумывать все более изощренные способы обмана, используя социальную инженерию, психологическое давление и современные технологии [2].

Антифрод-системы (от англ. anti-fraud – борьба с мошенничеством) представляют собой программные комплексы для предотвращения мошеннических транзакций. Антифрод-система обнаруживает и идентифицирует банковскую операцию, присваивая ей определенную метку в зависимости

от степени риска. И далее принимает решение: пропустить платеж, запросить дополнительную проверку или заблокировать.

Борьба с мошенничеством – это совместная работа технологий и бдительности людей. Важно не только внедрить систему, но и поддерживать ее в актуальном состоянии, а также ответственно относиться к своей цифровой безопасности. Антифрод стал неотъемлемой частью финансовой экосистемы. Он позволяет компаниям защищать клиентов и минимизировать их потери. Благодаря постоянному развитию и обучению, такие системы остаются ключевым инструментом противодействия мошенникам [3, 4].

Цель исследования – создать систему, состоящую из нескольких взаимодействующих сервисов, для автоматической проверки транзакций на возможные признаки мошенничества. Система должна предоставлять конечным пользователям интерфейсы для взаимодействия, обеспечивать высокую производительность и масштабируемость, а также поддерживать актуальность данных в реальном времени.

Материалы и методы исследования

В результате исследования было выявлено, что антифрод-система должна включать четыре основных компонента: (1) главный сервис: предоставляет REST и GraphQL API [5] для взаимодействия с пользователем; отвечает за создание и обновление транзакций; отправляет сообщения о создании транзакций в очередь RabbitMQ [6, 7]; читает сообщения из RabbitMQ для обновления статуса транзакций на основании решения от других компонентов системы; (2) gRPC клиент [8]: получает сообщения о транзакциях из RabbitMQ; обращается к gRPC серверу для вынесения вердикта о подтверждении или отклонении транзакции; передает результат обработки обратно в очередь RabbitMQ; (3) gRPC сервер [8]: реализует логику проверки транзакций на основе заданных правил и данных; возвращает результат проверки в виде решения: подтверждена или отклонена транзакция; (4) Notifier сервис: получает сообщения из RabbitMQ об обновлении статуса транзакций; пересылает обновления статусов через WebSocket [9] для обеспечения отображения актуальной информации в реальном времени на стороне клиента.

Функциональные требования к системе: пользователь должен иметь возможность создавать транзакции через REST/GraphQL API; система должна проверять каждую транзакцию на признаки мошенничества; результат проверки транзакции должен отображаться в реальном времени с использованием WebSocket; все компоненты системы должны быть модульными и легко масштабируемыми.

Нефункциональные требования: использование RabbitMQ для обмена сообщениями между сервисами; высокая отказоустойчивость за счет распределенной архитектуры; логирование всех событий для последующего анализа и отладки; время обработки транзакции не должно превышать 2 с.

Технологический стек: Backend: Kotlin [10], Spring Boot [11]; API: REST, GraphQL; сообщения: RabbitMQ; логика проверки: gRPC; обновления в реальном времени: WebSocket.

Проектирование безопасного API требует комплексного подхода, включающего надежную аутентификацию (OAuth 2.0, API-ключи), шифрование (HTTPS, TLS), лимиты запросов (Rate Limiting), защиту от атак и постоянный мониторинг.

Результаты исследования и их обсуждение

Для реализации данного проекта был выбран подход contract-first, что подразумевает создание API-контракта на начальном этапе разработки. В рамках данного подхода была разработана отдельная библиотека, которая содержит четко определенный контракт API, а также описания основных сущностей системы. Этот контракт был документирован с использованием таких инструментов, как Swagger и OpenAPI, что позволило получить интерактивную веб-документацию (рис. 1). Такой подход позволяет добиться согласованности между различными компонентами системы и упрощает последующую разработку, тестирование и поддержку.

Данный обработчик отрабатывает только на исключения типа ResourceNotFoundException. Предоставляет пользователю API значащий статус код и сообщение с подсказкой о причине ошибки. Для API в контракте определены два контроллера: user-controller и transaction-controller.

Разработка GraphQL API. Была описана схема запросов graphql, содержащая описания типов для взаимодействия с GraphQL API. Существуют два основных типа операций: запросы (Query) и мутации (Mutation). Запросы позволяют получать данные о пользователях. Например, с помощью запроса user можно получить информацию об одном пользователе по его идентификатору, включая транзакции и данные учетной записи [12, 13]. Запрос users возвращает список всех пользователей с их данными.

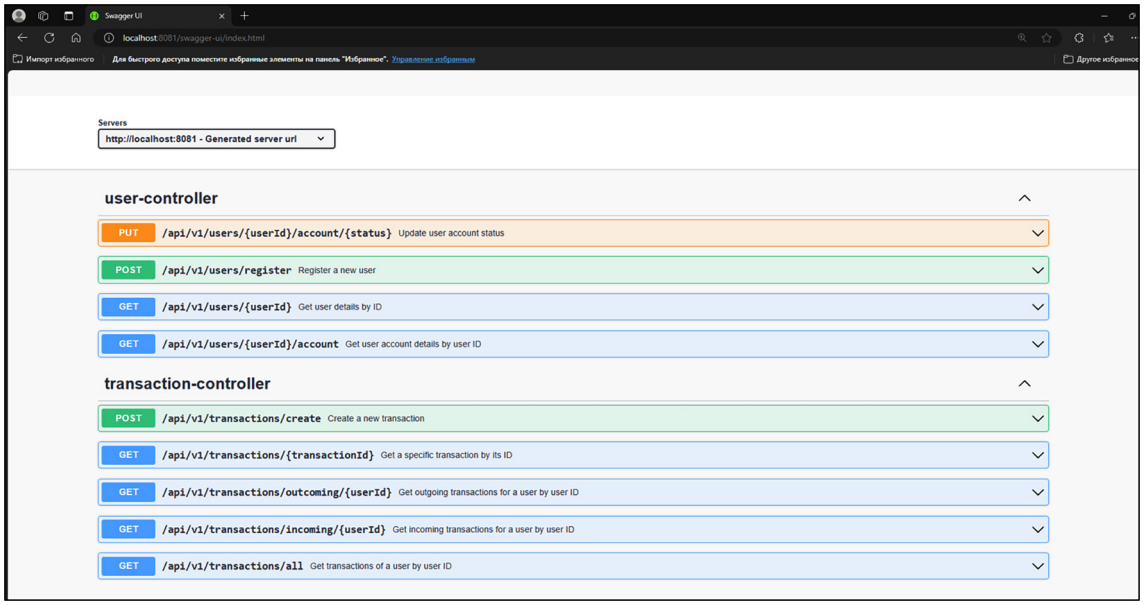

Рис. 1. Документация конечных точек API
Примечание: составлен авторами по результатам данного исследования

Отметим, что библиотека определяет общие классы ошибок и handler`ы для их обработки. Пример обработчика ошибок:

```
@ExceptionHandler(ResourceNotFoundException::class)
@ResponseStatus(HttpStatus.NOT_FOUND)
@ApiResponse(
  responseCode = "404",
  description = "Resource Not Found",
  content = [Content(schema = Schema(implementation =
ExceptionResponse::class))]
)
fun handleResourceNotFoundException(ex: ResourceNotFoundException): Re
sponseEntity<ExceptionResponse> {
  val response = ExceptionResponse(
    errorCode = "RESOURCE_NOT_FOUND",
    message = "Resource not found: ${ex.message}",
    details = "The resource with the given ID was not found in
the system."
  )
  return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response)
}
```

Мутации используются для внесения изменений в систему. Мутация createUser позволяет создать нового пользователя, предоставив такие данные, как имя, электронная почта, код валюты и тип учетной записи. Для работы с транзакциями предусмотрены мутации addIncomingTransaction и addOutcomingTransaction, которые добавляют входящие и исходящие транзакции для конкретного пользователя. Каждая транзакция содержит информацию о сумме, статусе, идентификаторах участников и способе оплаты.

Основной тип данных, с которым работает API, – это UserAggregate, который объединяет информацию о пользователе, его транзакциях и учетной записи. Каждая транзакция представлена типом Transaction, включающим такие поля, как идентификатор, сумма, статус, отправитель, получатель и способ оплаты. Учетная запись пользователя описывается типом Account, который содержит баланс, статус, тип учетной записи и код валюты.

В схеме предусмотрены перечисления для обозначения фиксированных значений. Например, TransactionStatus определяет статус транзакции как «в ожидании» или «завершена», AccountStatus описывает

активность учетной записи, а AccountType указывает на тип учетной записи, который может быть сберегательным или расчетным. Перечисление PaymentMethod задает способы оплаты, такие как кредитная карта или банковский перевод.

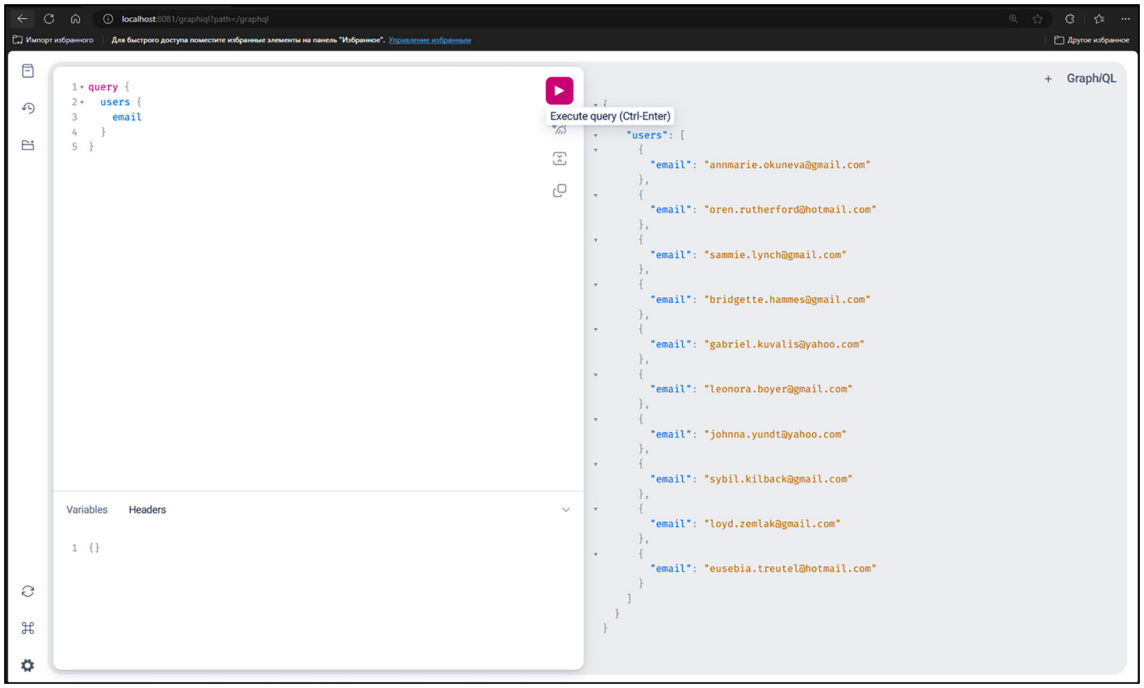

Рис. 2. Пример и результат выполнения запроса
Примечание: составлен авторами по результатам данного исследования

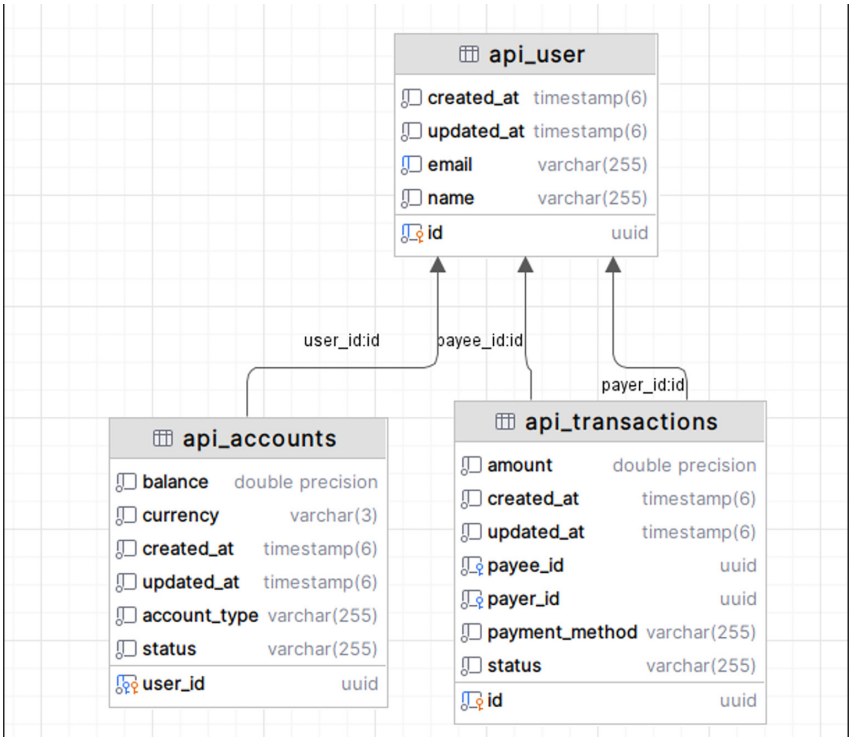

Рис. 3. Схема базы данных
Примечание: составлен авторами по результатам данного исследования

Такое API позволяет пользователю гибко запрашивать только нужные поля и в некоторой степени самому определять формат ответа от API. Как пример можно привести запрос на все e-mail`ы пользователей. Сам запрос и результат его выполнения в веб-оболочке graphiql продемонстрирован на рис. 2. Благодаря гибкой имплементации можно запросить и поля вложенной сущности.

Имплементация основного сервиса. Основной сервис разработан на базе Spring Boot, Spring Data и утилит для взаимодействия с RabbitMQ. Сервис хранит данные о пользователях в PostgreSQL [14] в схеме, представленной на рис. 3.

Как можно видеть, таблица api_user, представленная на рисунке 3, имеет связь один к одному с таблицей api_accounts, а с api_transactions – один ко многим. В данном сервисе и определенном ранее API реализованы CRUD операции для этих сущностей.

Отметим, что при имплементации REST API был использован принцип HATEOAS (Hypermedia as the Engine of Application State) [15]. Все сущности имеют ссылки на связанные с ними сущности, чтобы облегчить пользователям взаимодействие с API. Наглядным примером может служить постраничный список транзакций. Запросим постраничный ответ с размером страницы равным 1 с помощью следующего url - /api/v1/transactions/all?size=1. Ответ будет иметь следующий формат:

```
{«_embedded»: {
  «transactionRepresentationList»: [
    {  «id»: «131addb3-8686-4a7b-b431-6b3cdc2c2811»,
      "amount": 0.8500924491606185,
      "status": "PENDING",
      "_links": {
        "self": {
          "href": "http://localhost:8081/api/v1/transactions/131addb3-8686-
4a7b-b431-6b3cdc2c2811" },
        "payee": {
        "href": "http://localhost:8081/api/v1/users/3be2f90b-d4ac-4102-
889e-872ccf7326e6" },
        "payer": {
        "href": "http://localhost:8081/api/v1/users/243d5f2e-32ca-40bb-
a41c-e0998fd4eb3a" }}}},
"_links": {
  "self": {
    "href": "http://localhost:8081/api/v1/transactions/all?page=0&size=1" },
  "next": {
   "href": "http://localhost:8081/api/v1/transactions/all?page=1&size=1" }},
"page": {
  "size": 1,
  "totalElements": 10,
  "totalPages": 10,
  "number": 0 }}
```

Можно видеть, что ответ содержит ссылки на следующую страницу, payer`а и payee. Данное расширение ответа позволяет значительно упростить работу с API.

Также в проект была добавлена интеграция с очередью сообщений RabbitMQ. Данный сервис будет отправлять уведомления в очередь о создании новых сообщений, а также потреблять сообщения об обновлении статуса транзакций. Взаимодействие с RabbitMQ осуществляется с помощью RabbitTemplate.

Для отправки сообщений реализован RMQTransactionSenderService:

```
@Service
class RMQTransactionSenderService(private val amqpTemplate:
AmqpTemplate) : TransactionSenderService {
  override fun sendTransaction(transaction: TransactionMessageDTO) {
    amqpTemplate.convertAndSend(RMQConstants.Transactions.TRANSACTION_
EXCHANGE_NAME, RMQConstants.Transactions.ROUTING_KEY, transaction)
  }}
```

Данный сервис используется непосредственно в JpaTransactionService, метод sendTransaction вызывается непосредственно при создании транзакции.

Для обновления статуса транзакции используется TransactionStatusUpdateListener:

```
@Service
class TransactionStatusUpdateListener(
  private val transactionService: TransactionService
) {
  @RabbitListener(queues = ["update-queue"])
  fun updateTransactionStatus(transactionCheckDto: TransactionCheckResultDto) {
    transactionService.updateTransactionStatus(UUID.
fromString(transactionCheckDto.transactionId), transactionCheckDto.
status)
  }}
```

При получении сообщения об изменении статуса транзакции слушатель обращается к transactionService, который и обновляет значение статуса в базе данных.

Имплементация gRPC клиента и сервера. В данной архитектуре gRPC клиент будет отвечать за отправку и получение сообщений через RabbitMQ, обеспечивая надежную и асинхронную коммуникацию между различными сервисами [16]. Основная логика валидации данных, выполнение бизнес-правил и условий, будет сосредоточена на gRPC сервере. Важно отметить, что как клиент, так и сервер будут реализованы без использования Spring Boot, что дает большую гибкость в выборе технологий и уменьшает зависимость от сторонних фреймворков. Такой подход обеспечит высокую производительность и расширяемость системы, сохраняя при этом простоту в реализации и поддержку.

В первую очередь также была реализована библиотека с gRPC контрактом на основе protobuf. Проект состоит из .proto файла с объявлением сообщений и gRPC сервисов.

Стоит отметить, что proto файл настроен таким образом, чтобы сгенерированные классы находились com.example.antifraud пакете и были доступны из внешнего класса AntiFraudService. Структура сгенерированных классов представлена на рис. 4.

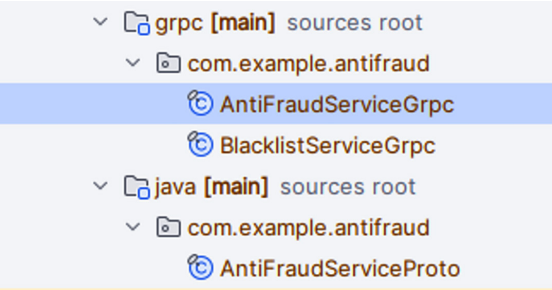

Рис. 4. Структура сгенерированных классов
Примечание: составлен авторами
по результатам данного исследования

Далее был реализован gRPC клиент без использования Spring Boot. Для удобства работы с общими значениями в проекте был создан объект Constants в пакете constants, который содержит константы, используемые по всему клиенту. Основная логика клиента реализована в классе grpcClient, который предоставляет базовую имплементацию для общения с сервером через сгенерированные stub классы. Этот клиент служит посредником между приложением и сервером, обеспечивая корректную отправку запросов и получение ответов. В файле Main.kt объявляются необходимые очереди с использованием стандартного клиента RabbitMQ, после чего регистрируется callback, который будет вызываться при поступлении сообщения в очередь TRANSACTION_CREATION_QUEUE. Внутри этого callback выполняется вызов gRPC сервера, что обеспечивает взаимодействие между RabbitMQ и gRPC сервисом для дальнейшей обработки сообщений. Такой подход позволяет легко интегрировать асинхронные очереди с сервисом. Структура проекта представлена на рис. 5.

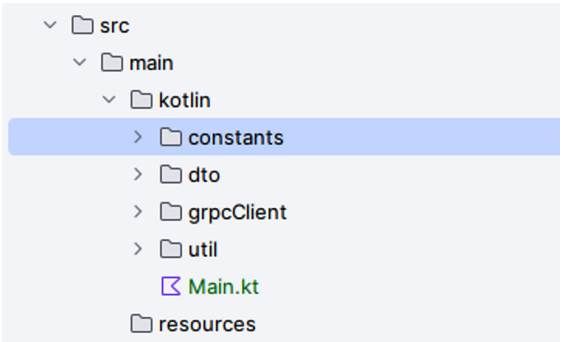

Рис. 5. Структура gRPC-клиента
Примечание: составлен авторами
по результатам данного исследования

gRPC сервер, в свою очередь, также не использует Spring Boot, поскольку сам

по себе io.grpc.Server является HTTP сервером. Для реализации gRPC сервера применяется веб-сервер Netty, который является стандартной имплементацией для gRPC. Сервер взаимодействует с базой данных, которая используется для ведения черного списка пользователей. Для работы с базой данных применяется библиотека Ktorm, которая обеспечивает удобное взаимодействие с базой данных и выполняет операции с объектами, используя функциональные подходы Kotlin. В качестве базы данных используется in-memory база данных H2.

Пакет, отвечающий за работу с базой данных, включает в себя объявление объектов, которые обеспечивают доступ к данным, а также функции для их инициализации и выполнения CRUD операций. Для валидации транзакций реализованы различные условия, которые включают в себя проверку размера переводимых средств в контексте выбранного способа перевода, проверку наличия одного из пользователей в черном списке и другие необходимые проверки.

Имплементация Notifier-service. Notifier-service является Spring Boot приложением, которое использует стандартные механизмы работы с WebSocket, предоставляемые Spring. Основная цель сервиса – обеспечить асинхронное уведомление пользователей через WebSocket при получении сообщений из RabbitMQ. Проект включает несколько ключевых компонентов:

1. конфигурация очередей RabbitMQ – определена очередь для приема сообщений, где реализован класс для парсинга сообщений. Этот класс отвечает за корректную обработку данных, поступающих из RabbitMQ, перед их отправкой на WebSocket;

2. конфигурация WebSocket – настроен WebSocket сервер с определением endpoint-а и топика, который будет использоваться для отправки сообщений клиентам;

3. NotifierService – основной сервис, который слушает очередь RabbitMQ, получает сообщения и перенаправляет их через WebSocket на подключенные клиентские приложения. Этот сервис выполняет роль посредника, получая данные из очереди и обеспечивая их доставку в нужное место.

Для демонстрации работы WebSocket был создан статический ресурс index.html, который включает вложенный JavaScript скрипт. Этот скрипт позволяет устанавливать соединение с сервером через WebSocket и в реальном времени обновлять содержимое HTML страницы.

Скрипт использует библиотеки SockJS и STOMP.js для организации взаимодействия с сервером. После подключения к WebSocket серверу клиент подписывается на определенный топик и начинает получать сообщения, связанные с транзакциями. Каждый раз, когда сервер отправляет новое сообщение, скрипт обрабатывает его и обновляет соответствующий элемент на странице, отображая информацию о транзакции, такую как ее идентификатор, статус и описание. Это позволяет пользователю наблюдать за происходящими событиями в реальном времени без необходимости вручную обновлять страницу.

Заключение

В ходе разработки системы для автоматической проверки транзакций на возможные признаки мошенничества была построена архитектура, состоящая из нескольких взаимодействующих сервисов, каждый из которых выполняет конкретную задачу в рамках общей системы. Были использованы такие технологии, как gRPC для обмена сообщениями между сервисами и RabbitMQ для асинхронной обработки транзакций. Валидация транзакций и проверки на мошенничество реализованы на gRPC сервере с использованием собственной логики валидации и интеграции с базой данных, обеспечивающей хранение черного списка пользователей.

Главный сервис системы предоставляет пользователям два ключевых интерфейса для взаимодействия с системой: REST API с поддержкой HATEOAS и GraphQL API. REST API с HATEOAS обеспечивает гибкость и удобство, позволяя пользователям получать не только данные о транзакциях и результатах проверок, но и ссылки на возможные дальнейшие действия, что упрощает навигацию и интеграцию с другими сервисами. В свою очередь, GraphQL API позволяет пользователям запрашивать только необходимые данные.

Для уведомлений пользователей о результатах проверки транзакций был создан Notifier-service на Spring Boot, который использует RabbitMQ и WebSocket для отправки уведомлений в реальном времени. Это решение позволило пользователям получать актуальные данные без задержек, что важно для эффективной работы с системами, основанными на обработке финансовых транзакций.

Разработан прототип архитектуры системы, которая обеспечивает автоматическую проверку транзакций на возможные признаки мошенничества и предоставляет интерфейсы для взаимодействия с конечными пользователями. Эффективность антифрод-проверок требует экспериментальной валидации.

Для обеспечения безопасности архитектуры необходимо применять комплексный подход.

Список литературы

1. Васильев Т. И. Разработка и внедрение систем для борьбы с мошенничеством в финансовых транзакциях // Экономика. Информатика. 2024. Т. 51. № 4. С. 919–925. URL: https://econom-inform-journal.ru/index.php/journal/article/view/408 (дата обращения: 26.06.2026). DOI: 10.52575/2687-0932-2024-51-4-919-925.

2. Брехов Д. А. Способы совершения мошеннических действий с использованием сети Интернет, средств подвижной связи и систем дистанционного банковского обслуживания // Вестник Поволжского института управления. 2021. Т. 21. № 4. С. 52–60. URL: https://vestnik-piu.ru/arkhiv/2021/4/Brekhov.pdf (дата обращения: 26.06.2026). DOI: 10.22394/1682-2358-2021-4-52-60.

3. Сенцова А. Ю., Тимергазин В. Э., Ильясова Р. И. Антифрод-система как инструмент предотвращения мошенничества // Информационные технологии. Проблемы и решения. 2021. № 4 (17). С. 101–107. EDN: JMFOVP.

4. Копнин А. А., Соколова Е. В., Долгополов А. А. Методика обеспечения безопасности банковских интернет-транзакций на основе антифрод системы // International Journal of Professional Science. 2022. № 10. С. 149–157. URL: https://cyberleninka.ru/article/n/metodika-obespecheniya-bezopasnosti-bankovskih-internet-tranzaktsiy-na-osnove-antifrod-sistemy/viewer (дата обращения: 26.06.2026). DOI: 10.54092/25421085_2022_10_149.

5. Бэнкс А., Порселло Е. GraphQL. Язык запросов для современных веб-приложений. СПб.: Питер, 2019. 240 с. (Серия «Бестселлеры O'Reilly»). ISBN 978-5-4461-1143-5.

6. Videla A., Williams J. J. W. RabbitMQ in Action / 1st Edition. Manning Publishing. 2012. 312 p. ISBN 9781935182979.

7. Официальная документация RabbitMQ. [Электронный ресурс]. URL: https://www.rabbitmq.com/documentation.html (дата обращения: 26.06.2026).

8. Индрасири К., Курупу Д. gRPC: запуск и эксплуатация облачных приложений. Go и Java для Docker и Kubernetes. СПб.: Питер, 2021. 224 с. (Серия «Бестселлеры O'Reilly»). ISBN 978-5-4461-1737-6.

9. Хабаров С. П., Шилкина М. Л. Построение распределенных систем на базе WebSocket: учебное пособие для вузов. 3 е изд., стер. СПб.: Лань, 2026. 216 с. ISBN 978-5-507-54698-5.

10. Сошин А. Kotlin. Паттерны проектирования и лучшие практики. 3-е изд. Sprint book. 2025. 416 с. (Серия «Библиотека программиста»). ISBN 978-601-09-9694-6.

11. Хеклер М. Spring Boot по-быстрому. СПб.: Питер, 2022. 352 с. (Серия «Бестселлеры O'Reilly»). ISBN 978-5-4461-3942-2.

12. Гумерова Г. Р., Мансурова Т. Г. Моделирование требований к программному обеспечению // Вестник Алтайской академии экономики и права. 2023. № 12–1. С. 42–52. URL: https://vaael.ru/ru/article/view?id=3131 (дата обращения: 26.06.2026). DOI: 10.17513/vaael.3131.

13. Патент № 2388053 C1 Российская Федерация, МПК G06Q 20/00. Способ проверки транзакций, автоматическая система для проверки транзакций и узел для проверки транзакций (варианты): № 2008143735/09: заявл. 06.11.2008: опубл. 27.04.2010 / А. Г. Рожков.

14. Жао Э. SQL. Pocket guide. 4-е изд. Астана: Спринт Бук, 2024. 320 с. ISBN 978-601-08-3728-7.

15. Aydemir F., Basciftci F. Application of HATEOAS Principle in RESTful API Design // IEEE 22nd International Symposium on Computational Intelligence and Informatics and 8th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics (CINTI-MACRo), Budapest, Hungary. 2022. P. 000051–000056. URL: https://ieeexplore.ieee.org/document/10029427 (дата обращения: 26.06.2026). DOI: 10.1109/CINTI-MACRo57952.2022.10029427.

16. Баженов А. Э., Райков А. В., Нехаев М. В., Ореховский Н. В. Оптимизация взаимодействия микросервисов с использованием асинхронных вызовов и брокеров сообщений (Kafka, RabbitMQ) // Программные системы и вычислительные методы. 2025. № 4. С. 77–93. URL: https://nbpublish.com/library_read_article.php?id=77228 (дата обращения: 26.06.2026). DOI: 10.7256/2454-0714.2025.4.77228.

Конфликт интересов: Авторы заявляют об отсутствии конфликта интересов.

Conflict of interest: The authors declare that there is no conflict of interest.

Финансирование: Авторы заявляют об отсутствии внешнего финансирования.

Financing: The research was performed without external funding.