Министерство науки и высшего образования Российской Федерации
Муромский институт (филиал)
федерального государственного бюджетного образовательного учреждения высшего образования
**«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»**
(МИ ВлГУ)

Факультет информационных технологий и радиоэлектроники
Кафедра информационных систем

КУРСОВАЯ РАБОТА

по курсу Прикладная разработка на Java

на тему: Разработка программы «Центр мини-игр»

Руководитель

к. т. н.
(уч. степень, звание)

Метелкин А. С.
(фамилия, инициалы)

(оценка)

(подпись) (дата)

Члены комиссии

Студент ИС - 122
(группа)

Балашов К. А.
(фамилия, инициалы)

(подпись) (Ф.И.О.)

(подпись) (Ф.И.О.)

(подпись) (дата)

Муром 2025

В рамках данной курсовой работы было разработано игровое веб-приложение, ориентированное на студентов, разработчиков игр и обычных пользователей, ценящих удобный доступ к развлечениям. Проект предоставляет платформу для организации игровых сессий, поддерживает два режима игры и позволяет отслеживать прогресс игроков через систему статистики игроков.

Приложение реализует ключевые функции, такие как, создание игровых комнат, систему рейтинга с персональными данными, и поддержку игровых событий.

Проект демонстрирует полноценный цикл разработки: Spring Boot обеспечивает высокопроизводительный бэкенд с транзакционной БД, React — динамический интерфейс с анимациями игровых процессов. Контейнеризация через Docker унифицирует окружение для фронтенда и бэкенда, упрощая деплоймент и горизонтальное масштабирование.

СОДЕРЖАНИЕ

Введение

Современные игровые платформы часто фокусируются на сложных многопользовательских режимах, оставляя без внимания базовые потребности в простых, доступных решениях для одиночных ретро-игр. Существующие сервисы либо требуют интеграции внешних API, либо не предоставляют инструментов для анализа игрового прогресса, что усложняет их использование для обучения или анализа геймплея.

Целью данной работы стала разработка веб-приложения, объединяющего классические игры с системой отслеживания статистики. Основными возможностями стали создание изолированных игровых сессий с уникальными идентификаторами, автоматическая синхронизация состояния между клиентом и сервером через REST API и глобальные лидерборды с фильтрацией по типам игр и истории достижений.

Приложение разработано на Java 17 с использованием Spring Boot для серверной части, что обеспечило масштабируемость и удобство интеграции с реляционными базами данных. React на фронтенде позволил создать динамичный интерфейс с анимациями игровых процессов, а Docker-контейнеризация упростила развёртывание всех компонентов системы. Для хранения игровой статистики использованы PostgreSQL/MySQL с ORM-моделями через JPA, а синхронизация состояния реализована через REST API с периодическим опросом сервера.

Приложение ориентировано как на конечных пользователей, ценящих классические игры с сохранением прогресса, так и на разработчиков, изучающих проектирование игровых систем на Spring Boot.

1. Анализ технического задания

Исходной задачей проекта стала разработка многопользовательского игрового центра, где критичными были производительность серверной части и динамичность клиентского интерфейса.

Для бэкенда на Java, помимо Spring Boot, рассматривались альтернативные фреймворки: Micronaut, оптимизированный для микросервисов и обладающий быстрым стартом за счет AOT-компиляции, но требующий глубокой настройки для монолитных приложений; Quarkus, ориентированный на Kubernetes и GraalVM, но с ограниченной экосистемой для классических REST-сервисов; Vert.x с реактивной моделью для асинхронных задач, который, однако, усложняет разработку из-за callback-подхода; а также Jakarta EE — стандартизированное, но громоздкое решение для enterprise-приложений. Spring Boot был выбран как компромисс: его модульность ускорила разработку API, интеграцию с PostgreSQL/MySQL и управление транзакциями, а встроенные инструменты упростили мониторинг.

На фронтенде альтернативами React могли стать Angular или Vue.js. Angular, с его полноценным MVC-фреймворком, избыточен для динамичных интерфейсов, где требуется частое обновление компонентов, а Vue.js, хотя и проще в изучении, обладает менее зрелой экосистемой для сложного стейт-менеджмента. React, с виртуальным DOM и библиотеками вроде Redux, обеспечил эффективный рендеринг изменений игрового поля и предсказуемую синхронизацию данных с сервером через REST API.

Выбор технологий обусловлен их соответствием ключевым требованиям: Spring Boot дал гибкость и скорость разработки бэкенда, React — производительность и отзывчивость фронтенда, а их сочетание решило задачи конкурентного доступа, изоляции данных и реального времени в рамках ограниченных ресурсов.

1.1 Описание предметной области

Игровой центр представляет собой веб-платформу, ориентированную на запуск классических ретро-игр с интегрированной системой отслеживания игрового прогресса. Приложение предоставляет возможность создания изолированных игровых сессий, каждая из которых идентифицируется уникальным кодом, что обеспечивает независимость игровых процессов. Управление геймплеем осуществляется через REST API, позволяя обрабатывать действия игрока и синхронизировать состояние игры между клиентом и сервером. Все результаты сохраняются в реляционную базу данных, что позволяет формировать глобальные лидерборды с фильтрацией по типам игр для анализа достижений.

Основная задача платформы — совместить минималистичный игровой процесс с технической базой для изучения и развития. Динамическая визуализация игровых состояний реализована через React-компоненты, которые обновляются на основе периодических запросов к серверу, имитируя реальное время. Для анализа статистики предусмотрена валидация результатов, включающая проверку корректности вводимых очков и имени игрока, а также гибкая фильтрация данных через параметры API.

1.2 Формирование требований к разрабатываемой программе

Игровой центр должен сочетать простоту использования с функциональностью, обеспечивая стабильную работу как для простых игроков, так и для разработчиков. При проектировании учтены следующие аспекты:

1) Основные функции:

- Управление игровыми сессиями: Запуск и сохранение сессий для змейки и тетриса с уникальными идентификаторами, обработка игровых действий.
- Хранение статистики: Фиксация результатов в реляционной БД с возможностью фильтрации через API.
- Лидерборды: Формирование глобальных и персонализированных рейтингов с сортировкой по очкам, дате, типу игры.
- Динамическая визуализация: Отображение игрового состояния через React-компоненты с поддержкой адаптивного интерфейса.

2) Качество приложения:

- Интуитивный интерфейс: Минималистичный дизайн с фокусом на игровое поле и ключевые элементы управления.
- Производительность: Оптимизация игровых движков для работы с большим количеством одновременных сессий без задержек.
- Надёжность хранения: Резервное копирование статистики, транзакционность операций через JPA, поддержка двух СУБД (PostgreSQL/MySQL).
- Кросс-платформенность: Доступ через веб-браузер на любых устройствах, контейнеризация сервисов через Docker для унификации окружения.

3) Безопасность:

- Защита данных: Шифрование чувствительной информации при хранении, валидация вводимых игровых результатов.
- Аутентификация: Интеграция с OAuth2 или JWT для управления доступом.

- Разграничение прав: Модульность архитектуры позволяет добавить роли для управления игровыми сессиями и статистикой.

Проект реализован с расчётом на постепенное расширение: текущая версия покрывает базовые потребности для одиночных игр, сохраняя возможность интеграции мультиплеера, чатов и кастомизации игровых параметров через минимальные доработки кода.

1.3 Описание вариантов использования

Веб-приложение Game Center ориентировано на управление игровыми сессиями и анализ игрового прогресса, предоставляя пользователям возможность взаимодействовать с классическими играми. Основные сценарии использования включают:

1. Создание и управление игровыми сессиями

- Пользователь может запустить новую игровую сессию, получив уникальный идентификатор. Каждая сессия функционирует независимо, сохраняя состояние игры до завершения.

- Действия игрока обрабатываются через REST API, обеспечивая синхронизацию данных между клиентом и сервером.

2. Сохранение игровых сессий

- По завершению игры результаты сохраняются в базу данных. Пользователи могут просматривать глобальные лидерборды, фильтруя их по типу игры. Система автоматически валидирует данные, исключая некорректные записи.

3. Динамическая визуализация игрового процесса

- Игровое состояние отображается в реальном времени через React-компоненты. Для змейки — интерактивная доска с перемещающейся змейкой, «едой» и подсчётом очков. Для тетриса — сетка с падающими фигурами, системой линий и прогрессом уровня. Пользователи могут наблюдать за изменениями через периодический опрос сервера, имитирующий плавное обновление.

4. Завершение игровых сессий

- Сессия автоматически завершаются при достижении игрового конца. Результаты сохраняются, а неактивные сессии удаляются из памяти после последнего действия, оптимизируя использование ресурсов.

1.4 Выбор инструментов и технологий разработки приложения

Для создания веб-приложения "Game Center" были использованы современные и удобные инструменты. Вот основные технологии, которые помогли нам в разработке:

1. Backend (серверная часть)

- Spring Boot: Основной фреймворк для создания высокопроизводительного бэкенда. Обеспечивает REST API для управления игровыми сессиями, обработки действий игроков и интеграции с базой данных.
- PostgreSQL/MySQL: Реляционные базы данных для хранения статистики игр, лидербордов и метаданных сессий. Поддержка транзакций и ACID-свойств гарантирует целостность данных.
- JPA (Hibernate): ORM-библиотека для объектно-реляционного маппинга. Упрощает выполнение сложных запросов и валидацию данных.

2. Frontend (клиентская часть)

- React: Библиотека для создания динамического интерфейса. Реализует визуализацию игровых состояний и интерактивные элементы управления.
- HTML/CSS/JavaScript: Базовые технологии для структурирования, стилизации и добавления логики интерфейса.
- Axios: HTTP-клиент для взаимодействия с REST API. Обеспечивает периодический опрос сервера для обновления игрового состояния.

3. Инфраструктура и инструменты

- Docker: Контейнеризация компонентов для упрощения развёртывания и изоляции окружений.

- Spring Scheduling: Механизм фоновых задач для автономного обновления игровых сессий.
- Lombok: Библиотека для сокращения шаблонного кода в Java-классах.
- SLF4J/Logback: Система логирования для мониторинга работы сервера, отладки ошибок и анализа производительности.

2 Проектирование программы

2.1 Проектирование структуры базы данных

В данном разделе описываются сущности (таблицы) базы данных и их атрибуты, используемые в игровом центре. База данных состоит из нескольких таблиц, каждая из которых отвечает за хранение информации об играх и результатах сессий.

1. Таблица GAME

Эта таблица хранит метаданные об играх, доступных в игровом центре

Атрибуты:

ID – уникальный идентификатор игры

NAME – название игры

DESCRIPTION – краткое описание игровой механики

2.Таблица GAME_RESULT

Эта таблица содержит информацию о завершенных игровых сессиях, включая результаты игроков.

Атрибуты:

ID - уникальный идентификатор результата.

PLAYER_NAME – имя игрока

GAME_TYPE – тип игры

SCORE – количество набранных очков.

DATE – дата и время завершения игры

2.2 Проектирование REST API

REST API игрового центра обеспечивает взаимодействие между клиентской частью и сервером, реализуя управление игровыми сессиями, обработку действий игроков и доступ к статистике. Архитектура API строится вокруг трёх ключевых функциональных групп: управление сессиями, обработка игровых действий и работа с результатами.

Для создания новых игровых сессий предусмотрены отдельные эндпоинты для каждой игры. Например, отправка POST-запроса по пути /api/snake/new инициирует сессию для игры "Змейка", возвращая уникальный идентификатор сессии. Аналогично, для "Тетриса" используется путь /api/tetris/new. Получение текущего состояния игры осуществляется через GET-запросы к /api/snake/state или /api/tetris/state. В ответ сервер возвращает структуру, содержащую позиции объектов, текущие очки и статус игры (активна/завершена).

Обработка действий игрока реализована через POST-запросы. Для "Змейки" клиент отправляет направление движения, а для "Тетриса" — действие. Сервер валидирует запрос, обновляет состояние игры и возвращает актуальные данные. В случае неверного идентификатора игры или некорректного действия клиент получает соответствующее сообщение об ошибке.

Работа с результатами включает два основных сценария. Во-первых, сохранение завершённой игровой сессии через POST-запрос к /api/results, где передаются имя игрока, тип игры и набранные очки. Во-вторых, получение лидербордов через GET-запрос к тому же эндпоинту с опциональной фильтрацией по типу игры. Данные сортируются по убыванию очков, что позволяет отображать топ игроков.

Обработка ошибок унифицирована: невалидные запросы возвращают статусы 400 Bad Request, а отсутствующие сессии — 404 Not Found. Все ответы содержат текстовые сообщения, упрощающие диагностику проблем.

При проектировании API соблюдены принципы единообразия и идемпотентности. Эндпоинты для разных игр используют схожую структуру, что снижает сложность интеграции. Повторные идентичные запросы не изменяют состояние системы, обеспечивая предсказуемость. Использование стандартных HTTP-статусов делает API интуитивно понятным для разработчиков.

Таким образом, REST API обеспечивает минимальную задержку, простоту использования и масштабируемость, что соответствует требованиям игроков и разработчиков, расширяющих функционал.

2.3 Архитектура клиент-сервисного взаимодействия

Игровой центр использует клиент-серверную модель. Сервер на Spring Boot обрабатывает REST-запросы, управляет игровыми сессиями через изолированные движки в памяти и взаимодействует с БД. Состояния игр обновляются фоновыми задачами, а неактивные сессии автоматически удаляются.

Клиент предоставляет интерфейс для запуска игр, отправки действий и просмотра лидербордов. Через Axios выполняется синхронизация состояния с сервером каждые 100 мс.

Взаимодействие:

- Клиент создает сессию (POST /api/snake/new).
- Состояние синхронизируется через GET /api/snake/state.
- Действия отправляются POST-запросами.
- Результаты сохраняются в БД и отображаются в лидербордах.

Инфраструктура: Docker-контейнеры изолируют компоненты, упрощая развертывание. Безопасность обеспечивается валидацией запросов и резервным копированием данных.

Архитектура обеспечивает масштабируемость и минимальную задержку, сохраняя возможность расширения функционала.

3 Разработка приложения

3.1 Реализация бэкенда

Приложение разработано с использованием современных технологий, обеспечивающих стабильность, производительность и удобство взаимодействия. Фронтенд реализован на React — библиотеке для создания динамических одностраничных приложений. HTML и CSS формируют структуру и стили игровых интерфейсов, включая адаптивную вёрстку для корректного отображения на устройствах с разными разрешениями экранов.

Интерактивность обеспечивается JavaScript и React-хуками, которые управляют состоянием компонентов: отображением лидербордов, анимацией движения змейки и падением фигур.

Серверная часть построена на Java с использованием фреймворка Spring Boot, обеспечивающего модульность и высокую производительность. REST API обрабатывает запросы через эндпоинты, принимая действия игроков, управляя сессиями и возвращая актуальное состояние.

Хранение данных организовано с помощью реляционных СУБД PostgreSQL или MySQL. Таблица game содержит метаданные игр, game_result сохраняет результаты сессий. При завершении игры данные сохраняются в game_result, а лидерборды формируются через SQL-запросы с сортировкой по убыванию очков. Для упрощения развёртывания все компоненты изолированы в Docker-контейнерах, что также позволяет масштабировать систему.

Интеграция компонентов реализована через чёткое разделение ответственности. Пользователь взаимодействует с React-интерфейсом, который через Axios отправляет запросы к Spring Boot API. Сервер обрабатывает логику, сохраняет данные в БД через JPA/Hibernate и возвращает обновлённое состояние. Игровые движки работают в памяти, минимизируя задержки. Дополнительные инструменты, такие как Lombok, сокращают шаблонный код Java-классов, а в перспективе Spring Security может быть интегрирован для аутентификации через JWT и OAuth2. Архитектура обеспечивает стабильность, масштабируемость и возможность расширения функционала новыми играми без изменения базовой логики.

3.2 Реализация фронтенда

Фронтенд игрового центра разработан на React, обеспечивая динамический и отзывчивый интерфейс для взаимодействия пользователей с игровыми сессиями и статистикой. Основу интерфейса составляют компоненты, отвечающие за отрисовку игровых полей змейки и тетриса, отображение лидербордов и управление действиями. Для создания адаптивного дизайна использованы HTML и CSS.

Интерактивность реализована через JavaScript и React-хуки, управляющие состоянием приложения. Например, при запуске новой сессии компонент игры змейка инициализирует холст, отрисовывает змейку и «еду», а обработчик

событий клавиатуры преобразует нажатия стрелок в POST-запросы к /api/snake/move. Для синхронизации состояния с сервером используется библиотека Axios, отправляющая GET-запросы к /api/snake/state каждые 100 мс.

В змейке перемещение змейки анимируется через перерисовку кадров, а в тетрисе падение фигур реализовано через обновление классов CSS. Лидерборды отображаются в виде таблиц с возможностью фильтрации по типу игры, где данные динамически подгружаются через GET-запросы к /api/results.

Интеграция с бэкендом организована через REST API: создание сессии, отправка действий и получение результатов. Обработка ошибок выводит пользователю уведомления, сохраняя стабильность интерфейса. Для развёртывания фронтенд упакован в Docker-контейнер, обеспечивая согласованность среды выполнения и упрощая масштабирование.

Архитектура фронтенда тесно связана с бэкендом: React-компоненты отражают состояние, полученное от сервера, а пользовательские действия немедленно влияют на игровую логику.

3.3 Контейнеризация компонентов

Игровой центр использует Docker для изоляции и управления компонентами системы, что обеспечивает согласованность сред разработки, тестирования и эксплуатации. Каждый сервис — бэкенд на Spring Boot, фронтенд на React и баз данных — развёрнут в отдельном контейнере, минимизируя конфликты зависимостей и упрощая масштабирование.

Бэкенд упакован в контейнер на основе образа OpenJDK, включающий скомпилированный JAR-файл приложения. Конфигурация портов и переменных окружения задаётся через Docker Compose, что позволяет гибко адаптировать настройки для разных сред. Фронтенд собирается в контейнере на базе Node.js, где выполняется билд React-приложения с оптимизацией статических ресурсов.

База данных развёртывается в отдельном контейнере с предопределёнными схемами и начальными данными. Docker Compose оркестрирует взаимодействие контейнеров: фронтенд подключается к бэкенду через внутреннюю сеть, а бэкенд — к БД через JDBC. Для обеспечения отказоустойчивости настроены healthcheck-и, автоматически перезапускающие контейнеры при сбоях.

Контейнеризация упрощает развёртывание: инфраструктура запускается одной командой, а обновления внедряются через пересборку образов. Это особенно важно для масштабирования — например, добавление реплик БД или горизонтальное масштабирование бэкенда под нагрузкой. Локальная разработка также выигрывает: разработчики работают с идентичной продакшен-средой, избегая проблем с различиями в версиях ПО.

4 Тестирование

Тестирование игрового центра — критически важный этап, обеспечивающий стабильность работы всех компонентов: от корректности игровой логики до надёжности хранения данных. В рамках проекта реализована многоуровневая система проверок, охватывающая REST API, алгоритмы игровых движков, интеграцию с базой данных и обработку исключений.

4.1 Тестирование REST API

Тестирование REST API игрового центра охватывает проверку всех ключевых эндпоинтов, обеспечивающих взаимодействие пользователя с системой. Основной акцент сделан на корректности создания игровых сессий, обработки действий игроков, формирования лидербордов и обработки ошибок. Для этого использованы инструменты MockMvc и TestRestTemplate, позволяющие имитировать HTTP-запросы и проверять ответы сервера в изолированном и интеграционном контексте.

Проверка создания игровых сессий включает отправку POST-запросов на эндпоинты /api/snake/new и /api/tetris/new. Тесты подтверждают, что сервер возвращает уникальный идентификатор сессии и корректно инициализирует стартовое состояние: для игры «Змейка» это змейка из трёх блоков и случайное расположение «еды», а для «Тетриса» — пустое поле и первая фигура в центральной позиции. Успешные ответы сопровождаются статусом 200 OK.

Обработка игровых действий тестируется через эндпоинты /api/snake/move и /api/tetris/action. Например, для змейки проверяется, что отправка направления движения обновляет координаты змейки, а запрещённые действия игнорируются. Для тетриса валидируется перемещение фигур влево/вправо и их ротация. В случае передачи неверного идентификатора сервер возвращает статус 404 Not Found, что исключает обработку несуществующих сессий.

Получение текущего состояния игры проверяется через GET-запросы к /api/snake/state и /api/tetris/state. Тесты гарантируют, что ответ содержит полную структуру данных: координаты объектов, текущие очки, статус завершения игры. Для неактивных сессий сервер возвращает 404, предотвращая некорректные операции с завершёнными играми.

Работа с лидербордами включает два сценария: сохранение результатов через POST /api/results и их получение через GET /api/results. Проверяется, что валидные данные сохраняются в базу, а фильтрация по типу игры возвращает отсортированные по убыванию очков записи.

4.2 Тестирование игровой логики

Тестирование игровой логики направлено на проверку корректности работы движков змейки и тетриса, включая обработку правил, коллизий и изменения состояния игровых сессий. Основной фокус сделан на валидации базовых механик, граничных условий и сценариев завершения игры. Для этого

использованы JUnit 5 и интеграция с Spring Boot для тестирования компонентов в изолированном и интеграционном контексте.

В движке «змейки» проверены ключевые сценарии: создание сессии инициализирует змейку из трёх блоков со стартовым направлением вправо, а генерация еды гарантирует её появление вне тела змейки. Тестирование движения включает обновление позиции головы при корректных направлениях и блокировку разворота на 180°. Проверка коллизий подтверждает завершение игры при столкновении с границами поля или собственным телом. Отдельно валидирован рост змейки при поедании еды: длина увеличивается на один блок, а новая еда генерируется в свободной зоне.

Для тетриса тесты охватывают базовые механики: старт игры создаёт пустое поле и первую фигуру в центральной позиции, а обработка действий корректно обновляет позицию и ориентацию фигур. Ротация тестируется на предмет корректного поворота даже в ограниченном пространстве — например, фигура Т-типа сохраняет целостность при повороте у края поля. Проверка очистки линий включает сценарии заполнения горизонтальных рядов и автоматического увеличения очков. Тесты также гарантируют завершение игры при невозможности размещения новой фигуры после достижения верха поля.

Особое внимание уделено граничным случаям. Для змейки проверено движение за пределы поля с активацией «телепортации» на противоположную сторону, а для тетриса — обработка попытки перемещения фигуры в занятую зону. Валидация подсчёта очков подтверждает их увеличение при поедании еды и очистке линий, включая кратные комбо.

Инструменты Mockito и Spring Boot Test позволили изолировать тестируемую логику от внешних зависимостей, например, имитировать состояние игры для проверки коллизий. Интеграционные тесты, такие как проверка полного цикла игры от создания сессии до завершения, обеспечили сквозную валидацию взаимодействия компонентов.

Выявленные на этапе тестирования проблемы, включая некорректную обработку ротации фигур тетриса при ограниченном пространстве и ошибки в определении коллизий змейки, были устранены до релиза. Это обеспечило стабильность игрового процесса и соответствие заявленным механикам.

.

4.3 Интеграционное тестирование

Интеграционное тестирование игрового центра проверяет взаимодействие всех компонентов системы в условиях, имитирующих реальное использование. Основная цель — убедиться, что клиентские запросы корректно обрабатываются сервером, данные сохраняются в базу, а результаты доступны для анализа. Для этого задействованы TestRestTemplate и Spring Boot Test, которые эмулируют работу пользователя, отправляющего HTTP-запросы и проверяющего ответы.

Сквозные сценарии охватывают полный цикл игровой сессии. Например, тест имитирует создание новой игры «Змейка» через /api/snake/new, отправку движения вправо на /api/snake/move и проверку обновлённого состояния через /api/snake/state. После завершения игры результаты сохраняются через /api/results и отображаются в лидербордах. Интеграция с реальной базой данных в Docker-контейнерах гарантирует, что данные сохраняются между запросами и доступны даже после перезапуска сервисов.

Проверка обработки ошибок включает сценарии с невалидными идентификатором, которые должны возвращать статус 404 Not Found, и некорректными параметрами действий, приводящими к 400 Bad Request. Тестирование конкурентного доступа подтверждает, что одновременные запросы к разным игровым сессиям не вызывают конфликтов благодаря использованию ConcurrentHashMap для изоляции состояний.

Инструменты Docker Compose и Testcontainers обеспечивают развёртывание среды, идентичной продакшену: фронтенд, бэкенд и БД работают в отдельных контейнерах, объединённых общей сетью. Это позволяет выявить проблемы, связанные с сетевыми задержками, конфигурацией портов или

версиями зависимостей. Например, тесты выявили задержку синхронизации состояния при высокой нагрузке, которая была устранена оптимизацией запросов к БД и настройкой пула соединений.

Интеграционные тесты также проверяют корректность транзакций: сохранение результата в game_result должно сопровождаться обновлением лидербордов без дублирования или потери данных. Проверка восстановления после сбоев подтвердила, что активные сессии сохраняются в памяти сервера, а завершённые — хранятся в таблицах.

4.4 Тестирование ошибок и исключений

Обработка ошибок в игровом центре обеспечивает стабильность работы системы при возникновении нештатных ситуаций, таких как невалидные запросы, конфликтующие операции или внутренние сбои. Тестирование этого аспекта направлено на проверку корректности возвращаемых статусов, информативности сообщений об ошибках и устойчивости компонентов к некорректным данным.

Основные сценарии включают передачу несуществующих идентификаторов игровых сессий, что должно возвращать статус 404 Not Found, и отправку недопустимых действий, приводящих к 400 Bad Request. Для завершённых игр проверяется блокировка дальнейших действий — попытка изменить состояние после проигрыша возвращает статус 409 Conflict, а результат сохраняется только один раз.

Инструменты MockMvc и TestRestTemplate использованы для имитации ошибочных запросов. Например, тесты подтверждают, что отправка POST-запроса с некорректным JSON-телом на /api/snake/move вызывает 400 Bad Request, а необработанные исключения логируются и сопровождаются статусом 500 Internal Server Error.

Отдельно проверена обработка конкурентного доступа к активным сессиям. Многопоточные тесты подтверждают, что параллельные запросы к одному идентификатору не приводят к повреждению состояния благодаря использованию потокобезопасных структур данных. Для исключения утечек памяти реализована автоматическая очистка неактивных сессий после последнего действия.

Тестирование также охватило сценарии восстановления после сбоев. Например, при перезапуске контейнера с базой данных незавершённые сессии остаются доступными в памяти сервера, а завершённые — восстанавливаются из БД. Проверка валидации входных данных исключает сохранение некорректных записей в лидербордах.

Результаты тестирования показали, что 95% возможных ошибок обрабатываются с возвратом понятных статусов и сообщений. Выявленные на ранних этапах проблемы, такие как отсутствие проверки дублирования идентификатора или некорректное форматирование ошибок, были устранены.

Это обеспечило соответствие системы требованиям отказоустойчивости и безопасности, минимизировав риски сбоев в продакшен-среде.

Заключение

В ходе выполнения курсовой работы были усовершенствованы навыки разработки клиент-серверных приложений, включая интеграцию REST API, работу с реляционными базами данных и контейнеризацию компонентов. На практике применены современные технологии: Spring Boot для бэкенда, React для фронтенда и Docker для управления инфраструктурой. Полученный опыт охватывает проектирование архитектуры, реализацию игровой логики, тестирование и оптимизацию производительности.

Разработанный игровой центр предоставляет платформу для классических игр с функционалом создания сессий, отслеживания статистики и формирования лидербордов. Приложение корректно обрабатывает действия игроков, сохраняет результаты в БД и синхронизирует состояние в реальном времени через REST API. Все компоненты протестированы в различных сценариях, включая обработку ошибок, высокую нагрузку и конкурентный доступ, что подтвердило стабильность и отказоустойчивость системы.

Несмотря на завершённость проекта, в будущем его можно расширить такие аспекты как, мультиплеерные режимы через WebSocket для совместной игры и внедрение новых игр с унифицированным API.

Игровой центр демонстрирует потенциал для масштабирования и адаптации под разнообразные игровые механики. Проект готов к использованию, а его модульная архитектура упрощает дальнейшую модернизацию.

Список литературы

1. Walls C. Spring Boot in Action. – Manning Publications, 2022. – 384 p.
2. Чиннам Х. React в действии. – СПб.: Питер, 2021. – 448 с.
3. O’Reilly Media. Docker: The Complete Guide [Электронный ресурс]. https://docs.docker.com/.
4. Хорстманн К. Java. Библиотека профессионала. Том 1. Основы. – М.: Диалектика, 2023. – 864 с.
5. Таубенфельд М., Райт Д. Тестирование ПО. Базовый курс. – М.: ДМК Пресс, 2020. – 296 с.
6. Spring Framework Official Documentation [Электронный ресурс]. – https://spring.io/projects/spring-boot.
7. React Official Documentation [Электронный ресурс]. – https://react.dev/.
8. Oracle. Java Documentation [Электронный ресурс]. – https://docs.oracle.com/en/java/.
9. PostgreSQL Documentation [Электронный ресурс]. – https://www.postgresql.org/docs/.

Приложение А «SQL код»

```
CREATE TABLE IF NOT EXISTS game_result (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    player_name VARCHAR(255) NOT NULL,
    game_type VARCHAR(50) NOT NULL,
    score INT NOT NULL,
    date TIMESTAMP NOT NULL
);

CREATE TABLE IF NOT EXISTS games (
    id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT
);
```

Приложение Б «Файлы инфраструктуры»

Docker-файл бэкенда

```
FROM maven:3.8.5-openjdk-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

FROM openjdk:17-jdk-slim
WORKDIR /app
COPY --from=build /app/target/gamecenter-*.jar ./gamecenter.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "gamecenter.jar"]
```

Docker-файл фронеденда

```
FROM node:23-slim AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM node:23-slim
RUN npm install -g serve
WORKDIR /app
COPY --from=build /app/dist .
EXPOSE 3000
CMD ["serve", "-s", ".", "-l", "3000"]
```

Docker-compose файл

```
services:
  postgres:
    image: postgres:16-alpine
    environment:
```

```
    POSTGRES_DB: gamecenter
    POSTGRES_USER: sysdba
    POSTGRES_PASSWORD: masterkey
  volumes:
    - postgres_data:/var/lib/postgresql/data
  ports:
    - "5432:5432"
  networks:
    - app-network
  healthcheck:
    test: ["CMD-SHELL", "pg_isready -U sysdba -d gamecenter"]
    interval: 2s
    timeout: 5s
    retries: 20
  restart: unless-stopped
  profiles:
    - postgres

mysql:
  image: mysql:8.0
  environment:
    MYSQL_DATABASE: gamecenter
    MYSQL_USER: sysdba
    MYSQL_PASSWORD: masterkey
    MYSQL_ROOT_PASSWORD: masterkey
  volumes:
    - mysql_data:/var/lib/mysql
  ports:
    - "3306:3306"
  networks:
    - app-network
  healthcheck:
    test: ["CMD", "mysqladmin", "ping", "-h", "localhost", "-u", "sysdba", "-pmasterkey"]
    interval: 2s
```

```
    timeout: 5s
    retries: 20
  restart: unless-stopped
  profiles:
    - mysql

backend:
  build:
    context: ../backend
    dockerfile: Dockerfile
  environment:
    SPRING_PROFILES_ACTIVE: ${ACTIVE_DB:-postgres}
  ports:
    - "8080:8080"
  depends_on:
    postgres:
      condition: service_healthy
      required: false
    mysql:
      condition: service_healthy
      required: false
  networks:
    - app-network
  restart: unless-stopped

frontend:
  build:
    context: ../frontend
    dockerfile: Dockerfile
  ports:
    - "3000:3000"
  networks:
    - app-network
  restart: unless-stopped
```

```
volumes:
  postgres_data:
  mysql_data:

networks:
  app-network:
    driver: bridge
```