Молдавский государственный университет

Факультет математики и информатики

Департамент Информатики

Курсовая работа

Управление транзакциями в Spring Framework

444.3 Informatica Aplicată

Выполнена студенткой II курса,
специальности Прикладная
Информатика

Анной Доцен

Руководитель, преподаватель

Департамента Информатики

Чорней Олег

Кишинев, 2025

Оглавление

1. Введение

С развитием технологий и увеличением объёмов обрабатываемых данных возрастает значение надёжности и устойчивости информационных систем. Одной из важнейших задач при разработке таких систем является обеспечение целостности данных, особенно в условиях высокой нагрузки и многопоточности. Транзакции становятся ключевым механизмом, позволяющим гарантировать согласованность операций и защиту от потери данных.

Spring Framework предоставляет разработчику мощный инструментарий для управления транзакциями — как в декларативной, так и в программной форме. Это особенно актуально при построении сложных бизнес-приложений, требующих строгого соблюдения логики и обеспечения отказоустойчивости.

Цель данной курсовой работы — изучить и реализовать принципы управления транзакциями в Spring Framework, исследовать их влияние на поведение системы при различных уровнях изоляции и рассмотреть практические аспекты разработки серверной части приложения с поддержкой транзакционного взаимодействия с базой данных.

2. Набор инструментария

Java является одним из самых популярных языков программирования, на момент июня 2024 года, он занимал 4 место.

Java - это объектно-ориентированный язык программирования, который является многоцелевым и строго типизированным. Имеет широкий спектр применений в мире.

Особенность этого языка в том, что он был разработан на основе C++, его создатели, Джеймс Гослинг, Майк Шеридан и Патрик Ноттон, разработали цифровой пульт дистанционного управления с графическим и анимированным сенсорным экраном. Этот пульт, результат многомесячных интенсивных исследований, отличался впечатляющей способностью управлять всеми устройствами в гостиной. Он использовал новый язык программирования, который не зависел от процессора, что делало его по-настоящему уникальным.

Его характеристики:

- Компилируемый и интерпретируемый
- Платформонезависимый и переносимый
- Объектно-ориентированный
- Надежный и безопасный
- Распределенный
- Знакомый, простой и компактный
- Многопоточный и интерактивный
- Высокопроизводительный
- Динамичный и расширяемый

У Java множество фреймворков, однако, один из самых популярных - это Spring Framework. Spring Framework представляет собой облегченный фреймворк с открытым исходным кодом. Он, в основном, ориентирован на предоставление различных способов управления бизнес-объектами. Это значительно упростило разработку веб-приложений по сравнению с классическими фреймворками Java и интерфейсами прикладного программирования (API), такими как подключение к базе данных Java (JDBC), JavaServer Pages (JSP) и Java Servlet.

Для того, чтобы понять всю значимость JDBC, JSP и Java Servlet, рассмотрим более подробно каждую технологию.

Java Database Connectivity - представляет собой унифицированный способ доступа к базам данных, благодаря чему становится возможным отправлять SQL- запросы из Java - приложения и обрабатывать данные, чтобы прийти к результатам. Данный интерфейс позволяет работать с любыми реляционными базами данных, имеющих JDBC - драйвер.

JavaServer Pages - позволяет встраивать Java - код непосредственно в HTML файл, чтобы создать динамическую страницу. Благодаря этой технологии упрощается серверная часть сайта и становится возможным писать Java и HTML код в одном файле. JSP хорошо подходит для создания и быстрого прототипирования страниц, что является очень важным фактором для использования этой технологии.

Java Servlet - является классом, который запускается на сервере и служит для обработки HTTP - запросов и, соответственно, генерации ответов. С помощью Servlet создаётся динамический контент и происходит обработка пользовательских запросов. Преимущество заключается в том, что Servlet имеет высокую производительность за счёт многопоточности, поскольку все запросы обрабатываются отдельно друг друга, в разных потоках.

Использование этих технологий вместе позволяет создавать мощные веб-приложения на Java, где JDBC отвечает за данные, сервлеты – за логику, а JSP – за представление данных.

Диаграмма связи между клиент сервером приложений и базой данных, с использованием вышеупомянутых сервлетов.

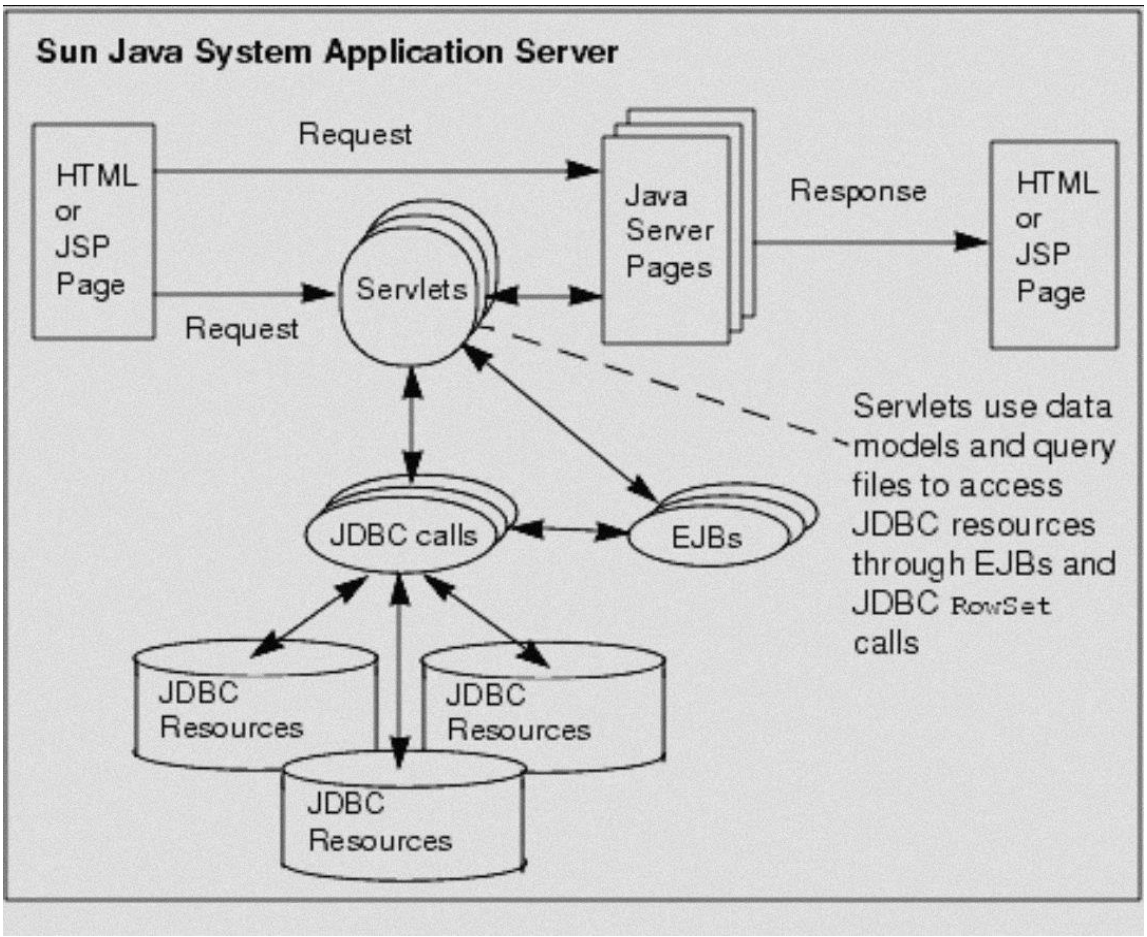

Рис. 1 Обработка запросов в серверном приложении

Эта диаграмма хорошо иллюстрирует общий процесс обработки запросов в серверном приложении на Java и показывает, как компоненты взаимодействуют для выполнения бизнес-логики, работы с базой данных и генерации динамического контента.

Java, благодаря своей экосистеме и поддержке различных технологий, таких как JDBC, JSP и сервлеты, предоставляет разработчикам мощные инструменты для создания надёжных, масштабируемых и высокопроизводительных веб-приложений. А использование современных фреймворков, таких как Spring, значительно упрощает процесс разработки, особенно в контексте управления транзакциями и интеграции с базами данных.

3. Структура проекта

3.1 Структура транзакций

Для анализа транзакций и определения ключевых аспектов следует понять, что они из себя представляют. Транзакции — это набор операций, которые выполняются как единое целое. Их основная задача — сохранить целостность данных. Существует два возможных исхода выполнения транзакций: либо все операции завершаются успешно, либо при возникновении ошибки происходит откат к исходному состоянию, и результат не фиксируется. Это означает, что в случае сбоя система возвращает данные к состоянию, в котором они находились до начала выполнения транзакции, чтобы избежать частично выполненных операций. Для более детального изучения транзакций можно выделить три составляющие: простота интерфейса и функциональность, хранение данных и принцип работы. Данная курсовая работа предусматривает исследование процесса выполнения транзакций и реализацию проекта.

3.2 Функциональность и интерфейс

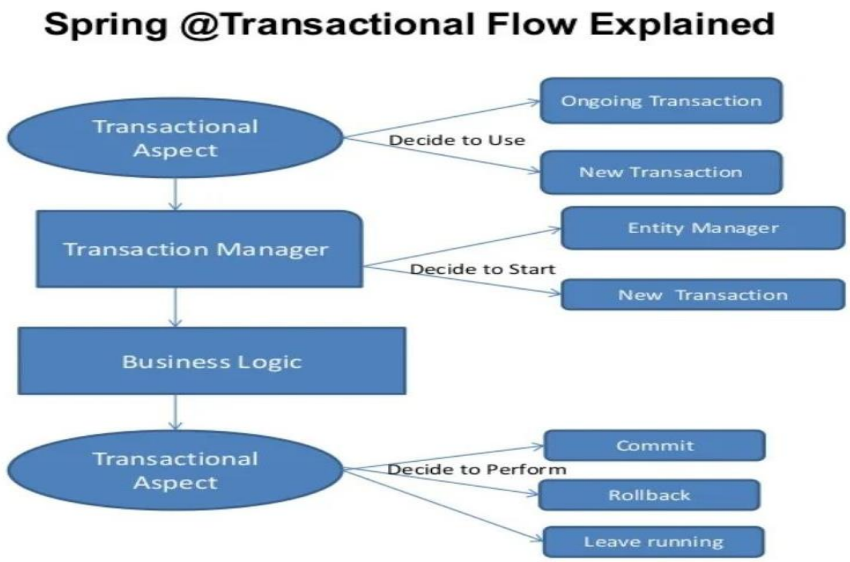

Рис. 1 Поток выполнения транзакций, при помощи "@Transactional"

Интерфейс включает в себя Transactional Aspect, который делится на 2 этапа:

1) Ongoing transaction, отвечает за выполнение и использование уже существующей транзакции, но только в том случае, если она активна. В этом случае она автоматически использует распространение "REQUIRED", благодаря этому предотвращается создание новой транзакции.

2) New Transactional, отвечает за создание новой транзакции. При создании, все изменения, которые будут происходить после, не будут связаны с прошлыми операциями. Этот способ используется, когда транзакции должны быть выполнены независимо друг от друга. Примером использования может служить запись в журнале, где каждая запись фиксируется независимо от другой.

В свою очередь функциональность транзакций в Spring Framework предоставляет пользователю множество гибких и качественных способов управления транзакциями в приложении.

Функциональность явно отображена на картинке, поскольку представлен пример поведения и обработки данных, на разных этапах работы:

1) Transaction Manager, несёт ответственность за выполнение и управление транзакциями, решает начинать новую или продолжить существующую. Подразделяется на "Entity Manager", который отвечает за все операции CRUD (чтение, заполнение, удаление) и на "New Transaction", она отвечает за создание новой транзакции, если того требует ситуация.

2) Business Logic, является основной частью выполнения кода, транзакции позволяют гарантировать, что все действия происходят как единое целое.

3) Transactional Aspect, выполняет завершающий этап, решает, что делать с результатом завершенной операции. Подразделяется на:

- Commit - фиксирует результат, если все успешно завершилось.
- Rollback - производит откат, если произошла ошибка.
- Leave Running - позволяет продолжать работу с данной транзакцией.

Несмотря на функциональность, у транзакций в Spring Framework есть ряд требований, которые проиллюстрированы на изображении:

Рис. 2 Требования ACID

В современном мире значительно возросли требования к безопасности использования транзакций. Необходимо уделять особое внимание возможности гибко управлять и настраивать их. Это можно сделать при помощи ряда функций которые помогут улучшить систему управления.

1) Расширенные параметры управления транзакциями

При использовании расширенных настроек распространения и изоляции можно обрабатывать транзакции в зависимости от специфики задачи. К примеру использование уровней изоляции позволяет снизить риск блокировки и различных конфликтов которые могут возникнут при параллельном выполнении обработки данных программы. Если настроить timeout для транзакций, то можно предотвратить зависание системы при долгих операциях, также это позволит автоматически отменять выполнение операции, если процесс слишком затянулся.

Уровни изоляции транзакций

Уровень изоляции	Черновое чтение	Неповторяемое чтение	Фантомы
Read Uncommited – чтение незавершённых транзакций	да	да	да
Read Commited – чтение завершённых транзакций	нет	да	да
Repeatable Read – повторяемое чтение	нет	нет	да
Serializable – последовательное чтение	нет	нет	нет

Рис. 3 Сравнение уровней изоляции транзакций

2) Поддержка мониторинга и логирования транзакций

Чтобы эффективно управлять транзакциями необходимо предусмотреть системы мониторинга и логирования. Любую из транзакций можно написать с указанием начала и завершения работы, выявлении ошибок, это поможет при диагностике и устранении неполадок. Использование систем мониторинга, например, таких как Prometheus или Grafana, позволит отслеживать состоянии транзакций в режиме реального времени.

Таблица 1. Сравнение инструментов для мониторинга и логирования

Инструмент	**Возможности**	**Совместимость**	**Поддержка Spring Fraemwork**
Prometheus	Мониторинг и анализ системы + сбор данных	Linux, Windows, macOS	Поддерживается

Grafana	Визуализация данных, построение графиков	Linux, Windows, macOS	Поддерживается
ELK Stack (Elasticsearch, Logstash, Kibana)	Сбор и анализ информации, визуализация данных и ошибок	Кроссплатформенный	Поддерживается
Spring Boot Actuator	Мониторинг и метрики специально для приложений на Spring	Java, Spring Framework	Полностью интегрирован

3) Поддержка асинхронных транзакций

При работе с задачами, не требующими мгновенного выполнения, асинхронная обработка транзакций позволяет добиться гибкости и улучшить производительность системы. Асинхронные транзакции не блокируют основное приложение, что особенно важно в условиях высокой нагрузки. Это позволяет продолжать выполнение других операций, пока транзакция обрабатывается в фоновом режиме.

Таблица 2. Сравнение методов асинхронных транзакций

Подход	**Описание**	**Применение**	**Пример**
@Async в Spring	Асинхронное выполнение метода через @Async	Базовые задачи, которые не требуют блокировок	Асинхронное чтение и запись данных, обработка IO операций. (Уведомления и отчеты)
Message Queue (RabbitMQ)	Передача сообщений в очередь для последующей обработки	Масштабные задачи	Выполнение задач с отложенным временем, распределение задач по сервисам (Обработка платежей)
Event-Driven Architecture	Подход, где данные и события обрабатываются как потоки; применяется реактивное программирование	Сложные задачи, которые требуют дополнительных обработок	Обработка данных в реальном времени, системы с высокой нагрузкой. (Уведомление пользователей)

3.3 Организация данных

Эффективное управление транзакциями и данными в приложении на основе Spring Framework невозможно без правильной организации хранения и обработки информации. В современных приложениях особое внимание уделяется не только корректности работы бизнес-логики, но и мониторингу, аудиту и обеспечению безопасности данных.

Для обеспечения целостности и безопасности данных важную роль играют мониторинг и аудит. Spring Framework предоставляет встроенные механизмы для контроля состояния приложения и отслеживания изменений данных:

- Spring Data JPA позволяет реализовать аудит с помощью аннотаций @EntityListeners и AuditingEntityListener, автоматически фиксируя дату создания и изменения сущностей. Это упрощает ведение истории изменений и помогает при отладке и анализе работы приложения.
- Для логирования изменений и общего ведения журналов событий используются такие популярные библиотеки, как SLF4J и Logback. Они обеспечивают гибкость в конфигурации и интеграцию с различными системами хранения логов.

Качественное управление транзакциями в приложении на основе Spring Framework невозможно без правильной и продуманной структуры данных. Проект состоит в основном из файлов расширения .java (Java классы), .properties (конфигурации приложения), и .xml (конфигурация зависимостей), .html (frontend файлы, используемые для отображения пользовательского интерфейса с использованием шаблонизатора Thymeleaf.), файл README.md (указаны необходимые параметры для запуска проекта).

Таблица 1. Основные инструменты мониторинга и аудита

Инструмент	Описание	Назначение
Spring Boot Actuator	Расширение Spring Boot с готовыми эндпоинтами для мониторинга здоровья и метрик приложения	Быстрый доступ к состоянию приложения и журналам
ELK Stack	Набор инструментов: Elasticsearch (хранение), Logstash (сбор), Kibana (визуализация логов)	Централизованный сбор, хранение и анализ логов
Prometheus и Grafana	Prometheus собирает метрики, Grafana визуализирует их с помощью дашбордов	Мониторинг производительности и состояния в реальном времени

Использование таких инструментов позволяет не только собирать данные о работе приложения, но и оперативно реагировать на возможные сбои, а также проводить анализ и оптимизацию производительности.

Архитектура серверной части приложения построена с целью обеспечения максимальной модульности и удобства сопровождения. Основой является разделение приложения на функциональные уровни, что способствует изоляции ответственности и упрощению поддержки кода.

Компонент обработки запросов (аналог контроллеров) отвечает за приём и маршрутизацию входящих запросов от пользователей, обработку параметров и возврат результатов. Благодаря четкому разграничению, изменения в пользовательском интерфейсе или протоколах взаимодействия не затрагивают внутреннюю бизнес-логику.

Слой бизнес-логики (аналог сервисов) реализует основные правила и процессы приложения. Здесь происходит валидация данных, координация действий между разными компонентами и применение бизнес-правил. Такой подход упрощает тестирование и повторное использование кода.

Слой доступа к данным (аналог репозиториев) инкапсулирует взаимодействие с базой данных, обеспечивая абстракцию и защиту от изменений в инфраструктуре хранения. Использование ORM (Object-Relational Mapping) и Spring Data JPA позволяет упростить работу с данными и повысить читаемость кода.

Данный подход архитектуры улучшает масштабируемость приложения — отдельные модули можно развивать, заменять или расширять без существенного влияния на остальные части системы. Кроме того, это облегчает внедрение новых функций и исправление ошибок.

Также важным аспектом является управление транзакциями, которое обеспечивает целостность данных при выполнении сложных операций. В Spring Framework это реализовано с помощью декларативного управления транзакциями, что снижает вероятность ошибок и повышает надёжность системы.

В итоге, использование продуманной архитектуры вместе с современными инструментами мониторинга и аудита позволяет создавать масштабируемые, устойчивые к ошибкам и легко поддерживаемые приложения на основе Spring Framework, соответствующие современным требованиям бизнеса и пользователей.

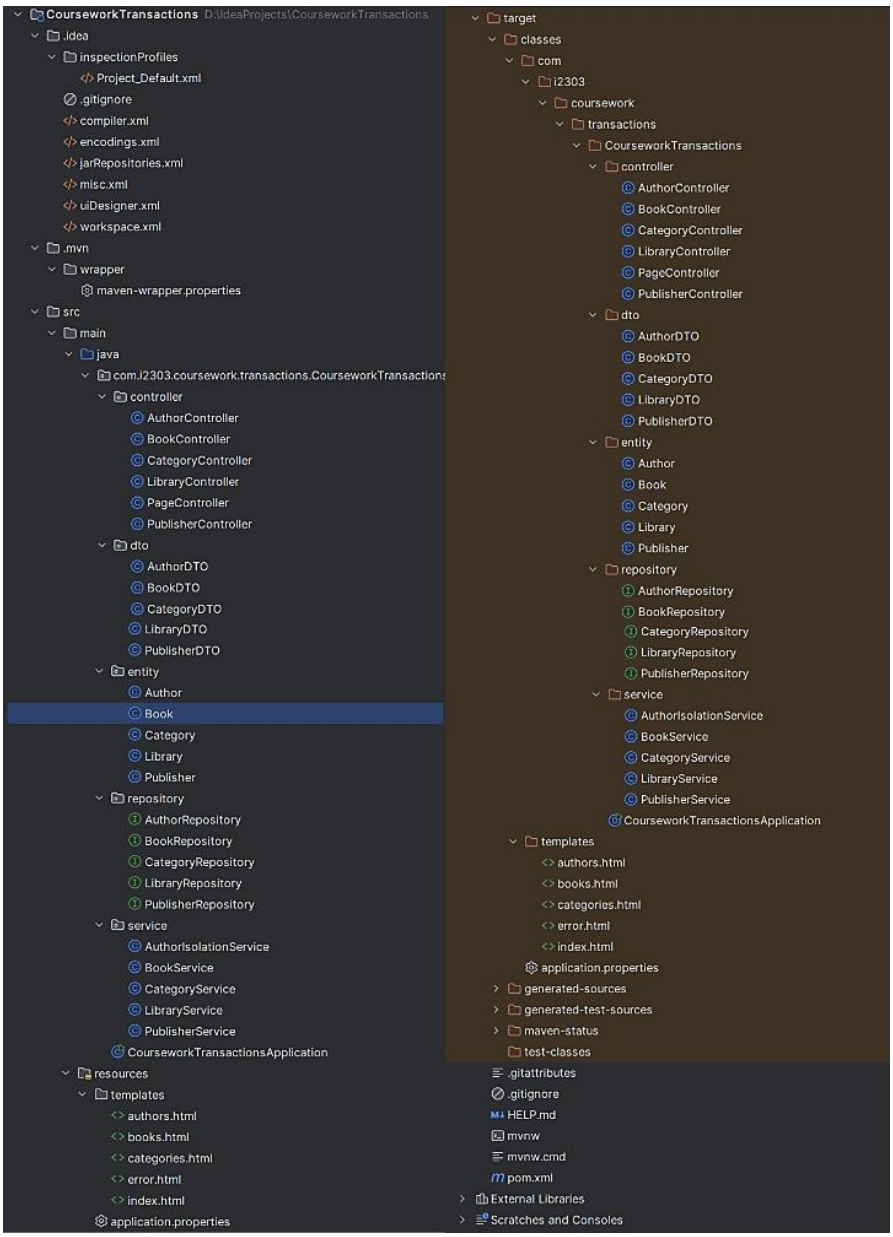

Рис. 1 Подробная организация файлов проекта

В рамках проекта были реализованы основные сущности: Book, Author, Publisher и Category, каждая из которых представляет отдельную таблицу в реляционной базе данных. Связи между сущностями описаны с использованием аннотаций JPA (Java Persistence API), что обеспечивает удобную работу с данными и поддерживает целостность информации на уровне базы данных.

- Book — основная сущность, представляющая книгу. Она содержит такие поля, как id, title, а также связи с сущностями Author, Publisher и множеством Category.

Книга связана с автором и издателем через отношения @ManyToOne, а с категориями — через отношение @ManyToMany.

- Author — сущность, описывающая автора книги. Включает информацию об имени автора и может быть связана с несколькими книгами.

- Publisher — сущность, представляющая издателя. Аналогично автору, издатель может быть ассоциирован с несколькими книгами.
- Category — сущность, представляющая категории, к которым может относиться книга (например, «Научная литература», «Художественная литература», «Учебник» и т.д.). Одна категория может быть связана с множеством книг.

Связи между таблицами позволяют обеспечить гибкую навигацию между данными и эффективное выполнение запросов к базе данных, что особенно важно в условиях транзакционного управления.

Также была реализована схема базы данных, наглядно иллюстрирующая взаимосвязь между сущностями:

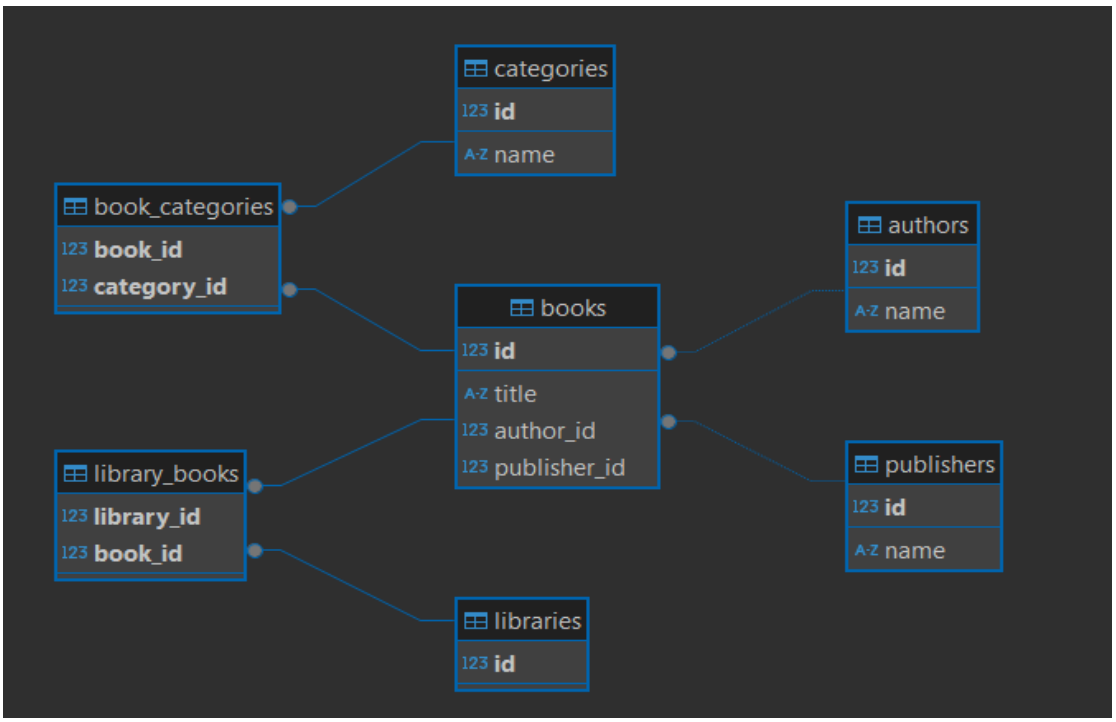

Рис. 2 Схема баз данных

3.4 Структура процессов

Процессы в системе организованы в соответствии с их функциями и взаимодействиями. Основные категории включают:

- Пользовательские процессы — обрабатывают HTTP-запросы от пользователей через контроллеры и сервисы, обеспечивая выполнение бизнес-логики.
- Фоновые процессы — выполняют асинхронные задачи, такие как обработка данных, отправка уведомлений или обновление кэшированных данных (при необходимости можно расширить проект соответствующими сервисами).
- Системные процессы — отвечают за мониторинг состояния приложения, логирование и техническое обслуживание, включая, например, аудит изменений данных.

Процессы структурированы логически: основной поток обработки запроса может координировать выполнение вспомогательных задач через асинхронные вызовы или события. Взаимодействие между компонентами осуществляется через механизм событий в Spring Framework или специализированные очереди сообщений при масштабировании.

Для эффективного управления жизненным циклом процессов применяются методы создания, выполнения и завершения задач.

Оптимизация работы достигается за счёт:

- настройки приоритетов выполнения задач;
- использования многопоточности через @Async в Spring;
- мониторинга состояния компонентов приложения с помощью Spring Boot Actuator и Prometheus/Grafana.

4. Реализация

4.1 Описание реализации серверной части

Серверная часть приложения реализована с использованием Spring Framework и построена по принципам трёхслойной архитектуры (Controller–Service–Repository). В данном разделе рассматриваются ключевые компоненты, их взаимодействие и применяемые технологии, а также особенности обработки данных, работы с транзакциями и интерфейса.

Если учитывать, что это разработка серверной части, значит, чтобы пользователь получил доступ, необходимо использовать визуальный интерфейс приложения

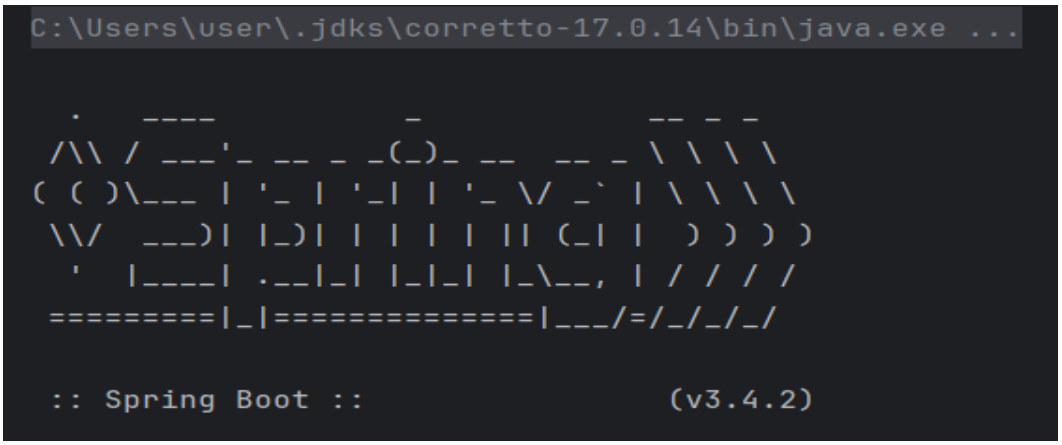

Рис. 1 Запуск проекта через терминал

В ходе выполнения работы была реализована структурированная архитектура проекта с разделением на отдельные директории, каждая из которых выполняет строго определённую задачу. В частности, были созданы директории для контроллеров, сервисов и репозиториев.

Контроллеры принимают HTTP-запросы, связанные с книгами, авторами и категориями. После получения запроса контроллеры передают данные в сервисный слой, где реализована основная бизнес-логика приложения — например, валидация данных, выполнение преобразований и принятие решений. Далее сервис взаимодействует с репозиторием, который служит промежуточным звеном между сервисным слоем и базой данных. Репозиторий отвечает за непосредственное извлечение, сохранение, обновление и удаление данных в базе.

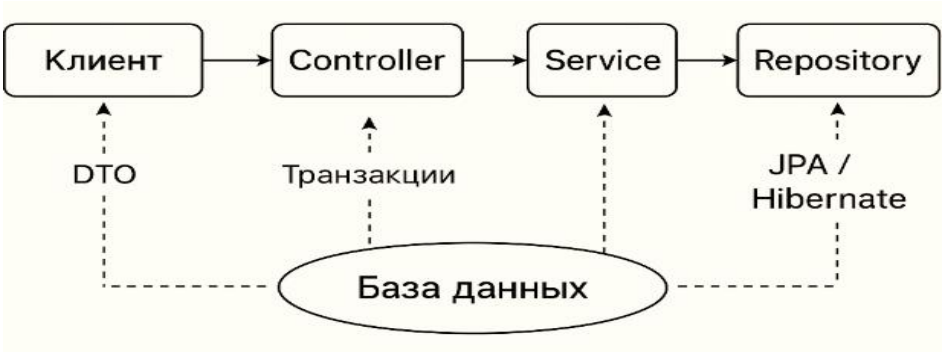

Рис. 2 Демонстрация архитектуры проекта

В проекте использованы следующие аннотации Spring Framework:

- *@Controller* — указывает, что класс является контроллером, обрабатывающим HTTP-запросы.
- *@RequestMapping, @GetMapping, @PostMapping* — используются для связывания методов контроллера с определёнными маршрутами и типами HTTP-запросов.
- *@Service* — обозначает класс как компонент сервисного слоя.
- *@Transactional* — управляет транзакциями базы данных, обеспечивая их целостность и согласованность.
- *@Autowired* — применяется для автоматического внедрения зависимостей через Spring-контейнер.

В текущей реализации зависимость между компонентами (контроллером, сервисом, репозиториями) устанавливается вручную через явно объявленные конструкторы.

Также в проекте используются аннотации, относящиеся к Spring Data JPA, такие как @Entity, @Table, @Id, @GeneratedValue, @Column, а также аннотации валидации @NotBlank, @Size (из jakarta.validation). Эти аннотации позволяют фреймворку автоматически связывать Java-объекты с соответствующими строками и столбцами в базе данных. Например, класс Author аннотирован как сущность с таблицей authors, а поле name имеет ограничения уникальности и обязательности:

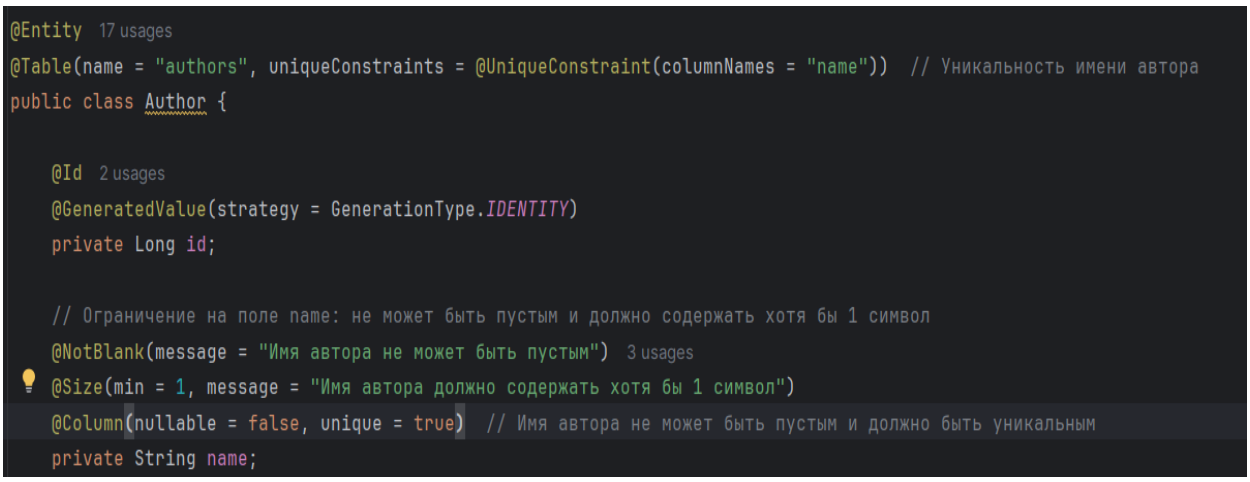

Рис. 3 Связь Java-объектов с соответствующими строками и столбцами в БД

Все сущности (Book, Author, Category, Publisher) используют JPA (Jakarta Persistence API) для отображения в таблице БД.

Связь между сущностями реализована через аннотацию @OneToMany. В приведённом примере один автор может быть связан с несколькими книгами:

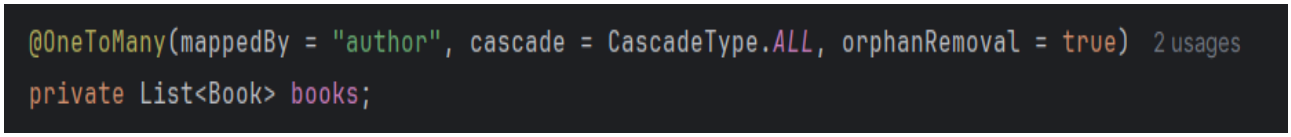

Рис. 4 Связь через аннотацию @OneToMany

Такие связи позволяют ORM автоматически управлять зависимостями и каскадными операциями между таблицами.

Важно отметить, что в приложении не используются Entity напрямую при передаче данных пользователю. Вместо этого применяются DTO-классы (Data Transfer Object), которые содержат только необходимые поля. Они применяются для передачи данных между слоями, которые исключают избыточные поля и рекурсивные зависимости. Например, AuthorDTO включает лишь id и name, в отличие от сущности Author, которая содержит также список книг. Это позволяет обезопасить данные, исключить избыточную информацию и избежать рекурсивных зависимостей при сериализации.

```
public class AuthorDTO {  17 usages
    private Long id;  3 usages
    private String name;  3 usages

    public AuthorDTO() {}  2 usages

    public AuthorDTO(Long id, String name) {  3 usages
        this.id = id;
        this.name = name;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Рис. 5 Применение DTO-классов

4.2 Форма работы с авторами и уровнями изоляции транзакций

Для повышения безопасности и корректной работы с конкурентными запросами была реализована поддержка транзакций с разными уровнями изоляции. Это позволяет обеспечить согласованность данных при параллельном доступе к системе, особенно при массовом добавлении или обновлении записей.

```
@Transactional(isolation = Isolation.SERIALIZABLE)  1 usage
public void addMultipleAuthors(List<AuthorDTO> authors) {
    for (AuthorDTO dto : authors) {
        if (dto.getName() == null || dto.getName().isBlank()) {
            throw new IllegalArgumentException("Имя автора не может быть пустым");
        }
        Author author = new Author();
        author.setName(dto.getName());
        authorRepository.save(author);  // Это будет транзакционная операция
    }
}
```

Рис. 6 Уровень изоляции **SERIALIZABLE** для добавления авторов

Поскольку SERIALIZABLE является самым строгим уровнем изоляции, следовательно, при параллельном добавлении одинаковых авторов СУБД обнаруживает потенциальный конфликт.

Итогом этого будет, что одна из транзакций будет отменена или заблокирована до завершения другой.

```
@Transactional(isolation = Isolation.READ_COMMITTED)  1 usage
public AuthorDTO updateAuthor(Long id, AuthorDTO authorDTO) {
    Author author = authorRepository.findById(id)
            .orElseThrow(() -> new RuntimeException("Автор с ID " + id + " не найден"));
    author.setName(authorDTO.getName());
    Author updatedAuthor = authorRepository.save(author);  // Сохраняем обновление
    return new AuthorDTO(updatedAuthor.getId(), updatedAuthor.getName());
}
```

Рис. 7 Уровень изоляции **READ_COMMITTED** для обновления авторов

При применении READ_COMMITTED, который позволяет читать только зафиксированные данные происходит то, что он не предотвращает проблему "потерянного обновления" (lost update). Если две транзакции одновременно обновляют одну запись, последняя фиксация перезапишет изменения первой, СУБД не блокирует вторую транзакцию, а просто применяет изменения последовательно, именно поэтому вывода ошибок не будет.

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)  1 usage
public void deleteAuthor(Long id) {
    if (!authorRepository.existsById(id)) {
        throw new RuntimeException("Автор с ID " + id + " не найден");
    }
    authorRepository.deleteById(id);  // Удаляем автора
}
```

Рис. 8 Уровень изоляции **READ_UNCOMMITTED** для удаления авторов

READ_UNCOMMITTED - самый слабый уровень изоляции. Он позволяет читать "грязные" данные (незафиксированные изменения из других транзакций). При параллельном выполнении с SERIALIZABLE транзакцией может возникнуть непредсказуемое поведение.

В рамках проекта реализована возможность управления сущностью автор. Пользователь может:

- добавлять нескольких авторов одновременно,
- редактировать существующих,

- удалять авторов из базы данных.

Форма представлена на Thymeleaf-шаблоне, что позволяет создавать интерактивный интерфейс.

```
<h2>Добавить нескольких авторов</h2>
<form action="/authors/multiple" method="post">
    <label for="names">Введите имена авторов (по одному на строку):</label>
    <textarea id="names" name="names" rows="5" required></textarea>
    <button type="submit">Добавить авторов</button>
</form>
```

Рис. 9 HTML разметка для добавления авторов

4.3 Управление книгами

Система управления книгами позволяет пользователям:

- добавлять новые книги,
- указывать одного автора и издателя,
- назначать категорий из предложенного списка.

В ходе выполнения работы создаем класс для добавления книги, ищем ID автора, издателя и нужную категорию, затем создаем объект книги и сохраняем ее

```
@Transactional 1 usage
public BookDTO addBook(BookDTO bookDTO) {
    try {
        // Находим автора по ID
        Author author = authorRepository.findById(bookDTO.getAuthorId())
                .orElseThrow(() -> new RuntimeException("Автор с ID " + bookDTO.getAuthorId() + " не найден"));

        // Найти издателя по ID
        Publisher publisher = publisherRepository.findById(bookDTO.getPublisherId())
                .orElseThrow(() -> new RuntimeException("Издатель с ID " + bookDTO.getPublisherId() + " не найден"));

        // Получить категории книги
        Set<Category> categories = bookDTO.getCategoryIds().stream() Stream<Long>
                .map(id -> categoryRepository.findById(id)
                        .orElseThrow(() -> new RuntimeException("Категория с ID " + id + " не найдена"))) Stream<Category>

                .collect(Collectors.toSet());

        // Создаем объект книги
        Book book = new Book(bookDTO.getTitle(), author, publisher);
        book.setCategories(categories);

        // Сохраняем книгу
        Book savedBook = bookRepository.save(book);
```

Рис. 10 Класс для добавления книги

Все данные связаны между собой через внешние ключи. Используется форма с выпадающими списками, автоматически заполняемыми с помощью контроллера.

```
<form th:action="@{/books}" th:object="${book}" method="post">
    <label for="title">Название книги:</label>
    <input type="text" id="title" th:field="*{title}" placeholder="Введите название книги" required>

    <label for="author">Автор:</label>
    <select id="author" th:field="*{authorId}">
        <option th:each="author : ${authors}" th:value="${author.id}" th:text="${author.name}">Автор</option>
    </select>

    <label for="publisher">Издатель:</label>
    <select id="publisher" th:field="*{publisherId}">
        <option th:each="publisher : ${publishers}" th:value="${publisher.id}" th:text="${publisher.name}">Издатель</option>
    </select>

    <label for="editCategoryIds">Категории:</label>
    <select id="editCategoryIds" th:field="*{categoryIds}" required>
        <option th:each="category : ${categories}" th:value="${category.id}" th:text="${category.name}"></option>
    </select>
    <button type="submit">Добавить книгу</button>
</form>
```

Рис. 11 Форма для добавления книги

4.4 Система категорий

В проекте реализовано управление категориями через базу данных PostgreSQL, что позволяет эффективно добавлять, удалять и извлекать категории.

Реализованы следующие операции с категориями:

- *Добавление нескольких категорий в базу данных:*

```
public List<String> addMultipleCategories(List<CategoryDTO> categoryDTOList) {
        List<String> errors = new ArrayList<>();
        for (CategoryDTO categoryDTO : categoryDTOList) {
            try {
                addCategory(categoryDTO); // если ошибка — логируем, но продолжаем
            } catch (Exception e) {
                String name = categoryDTO.getName();
                errors.add("Категория '" + name + "' не добавлена: " + e.getMessage());
                logger.warn("Категория '{}' не добавлена: {}", name, e.getMessage());
            }
        }
        return errors;
    }
```

- *Извлечение всех категорий из базы данных для отображения:*

```
public List<CategoryDTO> getAllCategories() {
    return categoryRepository.findAll().stream()
        .map(CategoryDTO::new)
        .collect(Collectors.toList());
}
```

- *Удаление категории из базы данных:*

```
public void deleteCategory(Long id) {
    categoryRepository.deleteById(id);
}
```

Каждая операция логируется, что позволяет отслеживать успешные действия и ошибки. Также реализована обработка ошибок, например, при попытке добавления категории с уже существующим названием.

Также, в проекте, при подключении Thymeleaf-шаблонов, был реализован процесс вывода ошибок, в случае неполадок с сервисом. Выводится статус ошибки, сообщение о том, что из себя представляет сбой работы, а также указывает где именно произошла неполадка.

```
    <h2>Информация об ошибке:</h2>
    <!-- Если сообщение пришло из контроллера -->
    <p th:if="${errorMessage}" th:text="${errorMessage}">Произошла неизвестная ошибка.</p>

    <!-- Если ошибка обрабатывается глобально (Spring Boot) -->
    <p th:if="${status}" th:text="'Статус ошибки: ' + ${status}"></p>
    <p th:if="${error}" th:text="'Ошибка: ' + ${error}"></p>
    <p th:if="${message}" th:text="'Сообщение: ' + ${message}"></p>
    <p th:if="${path}" th:text="'Путь: ' + ${path}"></p>
</div>
```

Рис. 12 Вывод ошибок

5. Заключение

В рамках курсовой работы была разработана серверная часть веб-приложения с использованием Spring Framework, в которой особое внимание уделено механизму управления транзакциями. Применение трёхслойной архитектуры (Controller–Service–Repository) обеспечило чёткую структуризацию проекта и позволило гибко управлять бизнес-логикой и доступом к данным.

Ключевым аспектом реализации стала интеграция аннотации @Transactional и настройка уровней изоляции транзакций. Это позволило обеспечить надёжную обработку конкурентных запросов, целостность данных и избежать типичных проблем многопоточности, таких как «грязные чтения», «потерянные обновления» и конфликты при массовых операциях. Были рассмотрены и реализованы различные уровни изоляции: READ_UNCOMMITTED, READ_COMMITTED и SERIALIZABLE, что позволило адаптировать поведение системы под конкретные сценарии работы.

Каждая транзакционная операция была логически обоснована и протестирована на предмет корректного поведения при параллельной нагрузке. Использование Spring Data JPA в сочетании с транзакциями обеспечило высокую надёжность при взаимодействии с базой данных и минимизировало вероятность ошибок при сохранении или обновлении сущностей.

В целом, работа показала, что Spring Framework предоставляет мощные и удобные средства для управления транзакциями, позволяющие строить надёжные и устойчивые серверные приложения.

6. Библиография:

1) https://habr.com/ru/companies/ssp-soft/articles/821663/
2) https://habr.com/ru/companies/spring_aio/articles/840336/
3) https://www.geeksforgeeks.org/introduction-to-spring-framework/
4) https://medium.com/edureka/advanced-java-tutorial-f6ebac5175ec
5) https://translated.turbopages.org/proxy_u/en-ru.ru.d2bc93fe-672de34a-e65849ab-74722d776562/https/www.baeldung.com/java-transactions
6) https://reflectoring.io/spring-transactions-and-exceptions/
7) https://javarush.com/quests/lectures/questspring.level03.lecture03?ysclid=m38kyl88nh608358640
8) https://struchkov.dev/blog/ru/transactional-isolation-levels/
9) https://www.javachinna.com/logging-performance-monitoring-security-and-transaction-management-with-spring-aop/