

ВВЕДЕНИЕ

В настоящее время в социуме появляется потребность идентифицировать себя как отдельную личность, так и как ячейку какого-либо общественного объединения. В зависимости от общих черт и признаков общественные объединения, возникающие на добровольной основе, в большинстве своём являются объединениями связанными общими целями или общими интересами. В такого рода общественных образованиях могут появиться общие ресурсы, которые объединение получает за счет сбора средств и совместной покупки для дальнейшего свободного использования внутри объединения.

Одно из самых интересных и недооцененных типов объединений – литературные клубы, их специфика заключается в том, что люди стараются путём общего сбора средств получить возможность приобретать книги, систематизировать их и поочередно читать.

Анализируя данные вопросы, можно сделать вывод, что необходима специализированная система, которая позволит делегировать управление ресурсами книжного клуба от администратора сайту, что облегчит участникам возможность выбирать свободные книги и получать по ним отзывы, на основе которых и будет формироваться их выбор.

Целью данного дипломного проекта является создание системы, предоставляющей полный функционал для учета пользователей книжного клуба, их истории чтения, а также учета ресурсов книжного клуба (книг), управления ими, быстрого получения информации о статусе конкретного ресурса (книги) и его местонахождении, составления рейтинга ресурса и возможности просмотра данного рейтинга пользователями. Данная программная система будет иметь клиент серверную архитектуру и использовать три уровня доступа к ресурсам:

- гостевой уровень – возможность просматривать список ресурсов клуба, общую информацию о ресурсах и рейтинг;
- пользовательский уровень – все возможности гостевого уровня, а также аутентификация, оставление заявки на получение ресурса, ведение статуса использования ресурсов, возможность оставлять отзывы и формировать рейтинг ресурса;
- административный уровень – все возможности пользовательского уровня, а также возможность просматривать, редактировать, добавлять и удалять пользователей.

В соответствии с поставленной целью были определены задачи:

- обеспечение удобного пользовательского интерфейса, не требующего затрат времени для изучения;
- обеспечение локализации и интернационализации;
- обеспечение полного функционала, для осуществления поставленных задач системе по делегированию управления ресурсами клуба с администратора на программный модуль.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Аналитический обзор используемых технологий

Разрабатываемая программная система является веб-приложением с клиент-серверной архитектурой. Это значит, что данное приложение по своей структуре распределенное, которое разделяет задачи или рабочие нагрузки между поставщиками ресурса или службы, называемые серверами, и инициаторами запросов на обслуживание [1]. В системе роль клиента играет браузер, а сервером является веб-сервер, который может храниться в облачном хранилище. Логика веб-приложения распределена между сервером и клиентом, хранение данных осуществляется, преимущественно, на сервере, обмен информацией происходит по сети. Данный подход позволяет клиентской части приложения не быть зависимой от реализации серверной части и от платформы, для корректного сообщения между системами достаточно соблюдение одного архитектурного стиля.

При разработке серверной части веб-приложения выбран шаблон проектирования MVC. MVC относится к архитектурным шаблонам, основным принципом которого является разделение программы, приложения, программного модуля или даже системы на три составные и взаимосвязанные части, а именно:

- модель (Model);
- представление (View);
- контроллер (Controller);

Модель – это представление сущностей, передаваемых между представлениями и контроллерами, а также модель включает в себя операции преобразования для управления этими данными.

Представление – это интерфейс приложения, является средством взаимодействия с внешней средой вокруг разрабатываемого программного модуля.

Контроллер – это структура, обрабатывающая поступающие запросы, выполняющая операции с моделью и выбирающая представления для визуализации пользователю.

Для того, чтобы разрабатывать веб-сервер можно использовать следующие языки программирования: Java, JavaScript, C#, Python, Kotlin, Scala и другие. Для написания веб-сервера программной системы будет использоваться язык программирования Java [2] (рисунок 1.1) .

Плюсы Java:

- объектно-ориентированное программирование;
- java — язык высокого уровня с простым синтаксисом и плавной кривой обучения;



Рисунок 1.1 – Логотип Java [3]

- стандарт для корпоративных вычислительных систем;
- безопасность;
- независимость от платформы («написать один раз и использовать везде»);
- язык для распределенного программирования и комфортной удаленной совместной работы;
- автоматическое управление памятью;
- многопоточность;
- стабильность и сообщество.

При выборе языка программирования для написания серверной части рассматривались и другие языки, которые будут вкратце рассмотрены ниже:

Плюсы JavaScript:

1. Быстрый для конечного пользователя: сценарий Java написан для клиентской стороны, для поддержки веб-сервера не требуется поддержка. Он также не нуждается в компиляции на стороне клиента, что дает ему определенные преимущества скорости. Поскольку сценарий выполняется на компьютере пользователя, в зависимости от задачи, результаты выполняются почти мгновенно. Это снижает нагрузку на сервер.

2. Простота: JavaScript относительно прост в освоении и реализации. Он использует модель DOM, которая обеспечивает множество предустановленных функций для различных объектов на страницах, что делает его легким для разработки сценария для решения пользовательской цели.

3. Универсальность: JavaScript отлично работает с другими языками и может использоваться в самых разных приложениях. В настоящее время существует множество способов использования JavaScript через серверы Node.js. Если вы загрузили node.js с помощью Express, используйте базу данных документов, такую как mongodb, и используйте JavaScript в интерфейсе для клиентов, вы можете создать приложение JavaScript полностью из одного окна вперед, используя только JavaScript.

Минусы JavaScript:

1. Безопасность: JavaScript явно добавлен к веб-страницам и

клиентским браузерам, данный факт может потенциально быть использован для кражи или внедрения вредоносного кода на машину пользователя.

2. Поддержка браузера: JavaScript иногда интерпретируется по-разному разными браузерами. Различные механизмы компоновки могут отображать JavaScript по-разному, что приводит к несогласованности с точки зрения функциональности и интерфейса. Большая часть JavaScript зависит от манипуляции элементами DOM браузеров. И разные браузеры предоставляют разные типы доступа к объектам, в частности Internet Explorer.

3. Отключить JavaScript: если вы отключите JavaScript в браузере, весь код JavaScript не запущен.

4. Загрузка файла: файл JavaScript загружается на клиентской машине, чтобы каждый мог прочитать код и повторно использовать его.

Плюсы Python:

1. Логичный, лаконичный и понятный. В сравнении с многими другими языками Python имеет легко читаемый синтаксис.

2. Кроссплатформенный: подходит для разных платформ: и Linux, и Windows.

3. Есть реализация интерпретаторов для мобильных устройств и непопулярных систем.

4. Широкое применение. Используется для разработки веб-приложений, игр, для автоматизации, математических вычислений, машинного обучения, в области интернета вещей. Существует реализация под названием Micro Python, оптимизированная для запуска на микроконтроллерах (можно писать инструкции, логику взаимодействия устройств, организовывать связь, реализовывать умный дом).

5. Сильное комьюнити и много конференций.

6. Мощная поддержка компаний-гигантов IT-индустрии. Такие компании, как Google, Facebook, Dropbox, Spotify, Quora, Netflix, на определенных этапах разработки использовали именно Python.

7. Высокая востребованность на рынке труда.

8. В мире Python много качественных библиотек, так что не нужно изобретать велосипед, если надо срочно решить какую-то коммерческую задачу. Для обучения есть много толковых книг, в первую очередь на английском языке, конечно, но и в переводе также издана достойная литература. Сегодня много обучающих материалов на Youtube: видео блоги, записи вебинаров и конференций. Думаю, что сейчас учиться легче, чем в то время, когда я начинал изучение.

9. Python отличается строгим требованием к написанию кода (требует отступы), что является преимуществом, по моим наблюдениям. Изначально язык способствует писать код организованно и красиво.

Минусы python:

1. Низкая скорость выполнения программ, по сравнению с другими языками. Поначалу, программы на python выполнялись гораздо медленнее, чем аналогичные, написанные на Java или C++, однако эта проблема была

решена созданием инструментов, которые переводят код с питона в байт-код (или код на C) и использованием виртуальных машин.

2. Копирование кода. При копировании кода с другого ресурса, в некоторых случаях, он может скопироваться без сохранения отступов. Поэтому код будет невалидным, а Вам придется долго добавлять табуляцию в каждую строчку. Для решения этой проблемы нужно либо использовать специальные IDE, либо добавлять в Ваш редактор плагины для Python.

3. Конвертация программы на python в *.exe. Программы на python имеют расширение *.py. Для использования без интерпретатора, например, на Windows, его нужно конвертировать в файл с расширением *.exe (для этого можно использовать приложение py2exe). Допустим, у Вас была программа, которое вычисляет три числа и выводит график при помощи Matplotlib. Это программа должна занимать около 30 Кб, однако после конвертации она может весить до 50 Мб. После удаления ненужных библиотек можно уменьшить эту цифру до 10 Мб, но результат будет хуже, чем у аналогичного проекта, сделанного, например на C++.

4. Unicode и русские символы. Другие программисты когда-то шутили про питонщиков, что единственная их проблема - это Unicode. Действительно, была такая проблема - её решили в Python 2, а в Python 3 её почему-то вернули обратно. Поэтому, если программа должна работать с русскими буквами (например, выводить текст «Привет, мир!»), то лучше сначала почитать об этом на ресурсах, посвященных этому замечательному языку, чтобы не видеть нечитаемых символов в выводе программы.

Плюсы C#:

1. Поддержка Microsoft. В отличие от Java, которой не пошел на пользу переход в собственность Oracle, C# хорошо развивается благодаря усилиям Microsoft.

2. В последнее время много совершенствуется. Так как C# был создан позже, чем Java и другие языки, то требовалось очень много доработать. Также это касается популяризации и бесплатности - было обещано открыть исходный код, а инструменты (Visual Studio, Xamarin) стали бесплатными для частных лиц и небольших компаний.

3. Много синтаксического сахара. Синтаксический сахар – это такие конструкции, которые созданы для облегчения написания и понимания кода (особенно если это код другого программиста) и не играют роли при компиляции.

4. Средний порог вхождения. Синтаксис похожий на C, C++ или Java облегчает переход для других программистов. Для новичков это также один из самых перспективных языков для изучения.

5. Xamarin. Благодаря покупке xamarin на C# теперь можно писать под Android и iOS. Это, несомненно, большой плюс, так как их собственная мобильная ОС (Windows Phone) не завоевала большой популярности.

6. Добавлено функциональное программирование (F#).

7. Большое сообщество программистов.
8. Много вакансий на должность C# программиста в любом регионе.

Минусы C#:

1. Ориентированность, в основном, только на .NET (на Windows платформу).
2. Бесплатность только для небольших компаний, учащихся и программистов-одиночек. Для больших команд покупка лицензий обойдется недешево. Поэтому если у вас есть своя фирма, то придется раскошелиться.
3. Сохранён оператор go to.

Для разработки клиентской части было решено использовать JavaScript фреймворк React JS (рисунок 1.2).

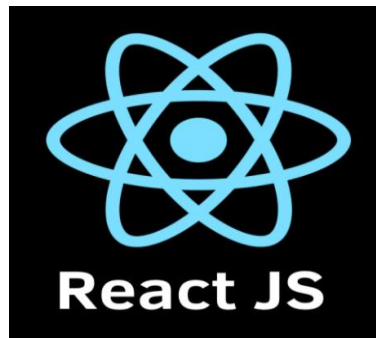


Рисунок 1.2 – Логотип React JS [5]

React – это JavaScript библиотека с открытым исходным кодом, для разработки пользовательских интерфейсов. Данная библиотека разрабатывается и поддерживается компанией Facebook. React может использоваться для разработки одностраничных и мобильных приложений. Его цель – предоставить высокую скорость, простоту и масштабируемость. В качестве библиотеки для разработки пользовательских интерфейсов React часто используется с другими библиотеками, такими как Redux.

1.2 Обзор фреймворков

В ходе анализа фреймворков, используемых для разработки веб-серверов на Java можно выделить только три основных кандидата: Spring Framework, JSF и Vaadin. Рассмотрим эти фреймворки:

Spring Framework – это фреймворк универсального назначения имеющий важную особенность – открытый исходный код, что позволяет ему совершенствоваться и развиваться с огромной скоростью. Данный фреймворк позволяет разрабатывать программы на языке Java или на Java-подобных языках (например, на Kotlin) (рисунок 1.3). Spring способен обеспечить решение многих задач, с которыми сталкиваются Java-разработчики и организации, которые хотят создать информационную систему, основанную на платформе Java.

Из-за широкой функциональности трудно определить наиболее значимые структурные элементы, из которых состоит Spring. Данный фреймворк наиболее известен как источник расширений, нужных для эффективной разработки сложных бизнес-приложений вне тяжеловесных программных моделей. Этот фреймворк предлагает последовательную модель и делает её применимой к большинству типов приложений, которые уже созданы на основе платформы Java. Считается, что Spring реализует модель разработки, основанную на лучших стандартах индустрии, и делает её доступной во многих областях Java.



Рисунок 1.3 – Логотип Spring Framework [6]

JavaServer Faces (JSF) – это Java спецификация для построения компонентно-ориентированных пользовательских интерфейсов для веб-приложений, написанный на Java (рисунок 1.4). Он служит для того, чтобы облегчать разработку пользовательских интерфейсов для Java EE-приложений. В отличие от прочих MVC-фреймворков, которые управляются запросами, подход JSF основывается на использовании компонентов. Состояние компонентов пользовательского интерфейса сохраняется, когда пользователь запрашивает новую страницу и затем восстанавливается, если запрос повторяется. Для отображения данных обычно используется JSP, Facelets, но JSF можно приспособить и под другие технологии, например, XUL.

JavaServer Faces усиливает существующие стандартные концепции пользовательского интерфейса (UI) и концепции Web-уровня без привязки разработчика к конкретному языку разметки, протоколу или клиентскому устройству. Классы компонентов пользовательского интерфейса, поставляемые вместе с технологией JavaServer Faces, содержат функциональность компонент, а не специфичное для клиента отображение, открывая тем самым возможность рендеринга JSF-компонент на различных клиентских устройствах.



Рисунок 1.4 – Логотип JSF [7]

Vaadin – свободно распространяемый фреймворк для создания RIA-веб-приложений, разрабатываемый одноимённой финской компанией. В отличие от библиотек на Javascript и специфических плагинов для браузеров, Vaadin предлагает сервер-ориентированную архитектуру, базирующуюся на Java Enterprise Edition. Использование JEE позволяет выполнять основную часть логики приложения на стороне сервера, тогда как технология AJAX. Для отображения элементов пользовательского интерфейса и взаимодействия с сервером на стороне клиента Vaadin использует Google Web Toolkit (рисунок 1.5).



Рисунок 1.5 – Логотип Vaadin [8]

Для разработки веб-сервера мы будем использовать Spring Framework, так как он реализует важный принцип объектно-ориентированного программирования IoC (Inversion of Control). Также Spring имеет собственный IoC-контейнер, который позволяет упростить и автоматизировать написание кода с использованием данного подхода. Также Spring Framework легко интегрируется с языком программирования Kotlin, который будет

использоваться нами в ходе разработки программной системы.

1.3 Обзор шаблонов проектирования

Одна из главных целей в проектировании – уменьшение трудозатрат на разработку сложного программного обеспечения, предположим, что необходимо использовать готовые унифицированные решения. Ведь шаблонность действий облегчает коммуникацию между разработчиками, позволяет ссылаться на известные конструкции, снижает количество ошибок.

В настоящее время известно большое количество шаблонов проектирования, но самыми распространенными можно назвать шаблоны:

- Model-View-Controller (MVC);
- Model-View-View-Model (MVVM);
- Model-View-Presenter (MVP).

MVC – это именно набор архитектурных идей и принципов для построения сложных систем с пользовательским интерфейсом.

При разработке систем с пользовательским интерфейсом, следуя паттерну MVC нужно разделять систему на три составные части. Их, в свою очередь, можно называть модулями или компонентами: модель, представление и контроллер (рисунок 1.6).

При этом модификация каждого из трех модулей происходит независимо. Данный подход описывает один из принципов SOLID – принцип единой ответственности. Это значит, что каждый объект (модуль) имеет одну ответственность и все поведение объекта должно быть направлено на выполнение только этой одной ответственности.

Модель в шаблоне MVC – основная часть логики, так называемая модель. Она содержит всю бизнес-логику приложения. Модель отвечает за выборку данных из базы, изменение данных, расчеты. Данные в модель приходят из контроллера, затем эти данные обрабатываются, будь то запросы к базе данных, либо же бизнес вычисления. Модель представляет данные приложения и связанную с ними бизнес-логику. Модель может быть представлена одним объектом или сложным графом связанных объектов. В приложении для плат формы Java EE данные инкапсулируются в объектах предметной области, часто развертываемых в EJB-модуле. Данные передаются в БД и из нее в объектах передачи данных (ОТО), и к ним обращаются с помощью объектов доступа к данным (ОАО).

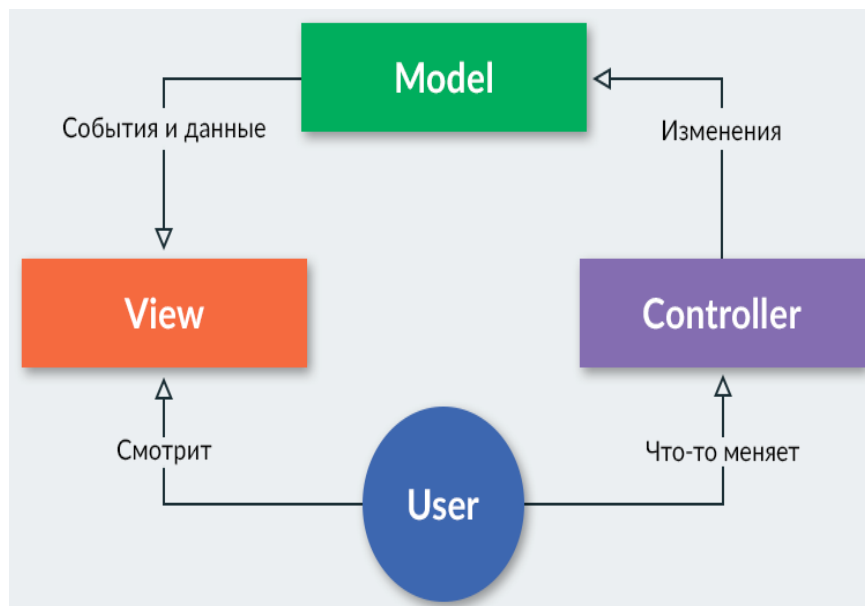


Рисунок 1.6 – Шаблон MVC [9]

Представление отвечает за отображение данных пользователю. Обычно представление содержит в себе мало логики и предназначено только для отображения результата работы модели. Представление – это наглядное отображение содержащихся в модели данных. Подмножество модели содержится в отдельном представлении, таким образом, представление действует в качестве фильтра для данных модели. Пользователь взаимодействует с данными модели с помощью предлагаемого представлением наглядного отображения и обращается к бизнес логике, которая, в свою очередь, воздействует на данные модели. Обычно представление реализуется посредством шаблонов html-разметки, которые заполняются данными. Данные приходят из модели, контроллер заполняет представление и шаблон html документа отправляется клиенту.

Контроллер – блок в шаблоне MVC, который не должен содержать в себе никакой бизнес логики. Вся основная бизнес логика должна содержаться в слое модели. Контроллер выступает связующим звеном между моделью и представлением. Контроллер связывает представление с моделью и управляет потоками данных приложения. Он выбирает, какое представление визуализировать для пользователя в ответ на вводимые им данные и в соответствии с выполняемой бизнес-логикой. Контроллер получает сообщение от представления и пересылает его модели. Модель, в свою очередь, подготавливает ответ и отправляет его обратно контроллеру, где происходит выбор представления и отправка его пользователю.

MVP – Данный подход позволяет создавать абстракцию представления. Для этого необходимо выделить интерфейс представления с определенным набором свойств и методов. Презентер, в свою очередь, получает ссылку на реализацию интерфейса, подписывается на события представления и по запросу изменяет модель.

Реализуется он, когда каждое представление реализует соответствующий интерфейс (рисунок 1.7). Интерфейс представления определяет набор функций и событий, необходимых для взаимодействия с пользователем. Презентер должен иметь ссылку на реализацию соответствующего интерфейса, которую обычно передают в конструкторе. Логика представления должна иметь ссылку на экземпляр презентера. Все события представления передаются для обработки в презентер и практически никогда не обрабатываются логикой представления (в том числе для создания других представлений).

Данный подход позволяет создавать абстракцию представления. Для этого необходимо выделить интерфейс представления с определенным набором свойств и методов. Презентер, в свою очередь, получает ссылку на реализацию интерфейса, подписывается на события представления и по запросу изменяет модель.

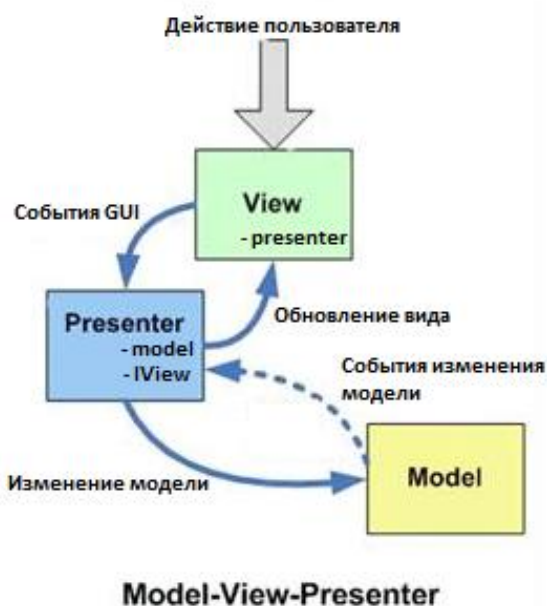


Рисунок 1.7 – Шаблон MVP [10]

MVVM – Данный подход позволяет связывать элементы представления со свойствами и событиями View-модели (рисунок 1.8). Можно утверждать, что каждый слой этого паттерна не знает о существовании другого слоя.

При использовании этого паттерна, представление не реализует соответствующий интерфейс (IView). Представление должно иметь ссылку на источник данных (DataContext), которым в данном случае является View-модель. Элементы представления связаны (Bind) с соответствующими свойствами и событиями View-модели.

В свою очередь, View-модель реализует специальный интерфейс, который используется для автоматического обновления элементов

представления.

Признаки View-модели:

1. Двухсторонняя коммуникация с представлением;
2. View-модель — это абстракция представления. Обычно означает, что свойства представления совпадают со свойствами View-модели модели;
3. View-модель не имеет ссылки на интерфейс представления (IView). Изменение состояния View-модели автоматически изменяет представление и наоборот, поскольку используется механизм связывания данных (Bindings);
4. Один экземпляр View-модели связан с одним отображением.

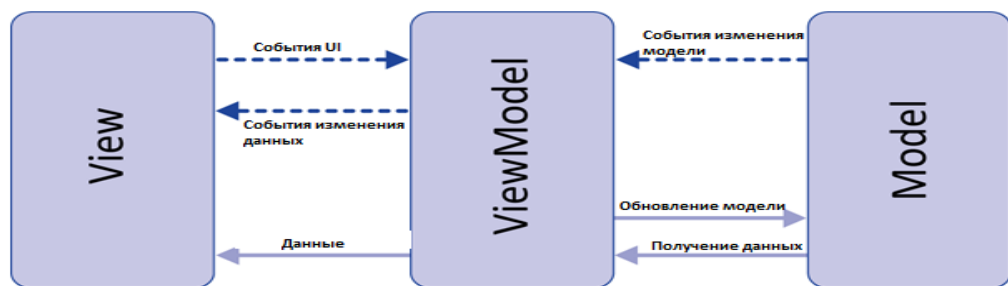


Рисунок 1.8 – Шаблон MVVM [11]

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Проанализировав и изучив теоретические аспекты разрабатываемой системы и выработав список требований необходимых для разработки системы, можно разбить разрабатываемое веб-приложение на следующие функциональные блоки:

- блок базы данных;
- блок авторизации пользователя;
- блок управления ресурсами;
- блок работы сервера;
- блок администрирования;
- блок пользовательского интерфейса.

Структурная схема, иллюстрирующая перечисленные блоки и связи между ними приведена на чертеже ГУИР.400201.013 С1.

Рассмотрим функциональные блоки разрабатываемого веб-приложения.

2.1 Блок базы данных

Блок базы данных включает данные, используемые веб-приложением. При реализации использовалась реляционная база данных PostgreSQL. Это свободно распространяемая объектно-реляционная система управления базами данных наиболее развитая из открытых СУБД в мире и являющаяся реальной альтернативой коммерческим базам данных.

Запросы к базе данных будут осуществляться как посредством стандартных механизмов используемых в Spring JPA для построения SQL запросов, так и используя библиотеку hibernate которая является реализацией стандарта JPA который позволяет связать Java классы с данными в реляционной базе данных. Использование данной библиотеки позволяет избежать работы с типами данных языка SQL и иметь дело с привычными типами данных Java которые в большинстве случаев будут представлять собой Java классы, помеченные аннотацией Entity. Данная библиотека так же позволяет автоматически заполнять некоторые поля классов, которые мы заносим в базу данных. Примерами этих полей могут служить поля, помеченные такими аннотациями как LastModifiedDate и LastModifiedBy которые автоматически заполняют информацию о том когда была последняя модификация данного объекта и кем соответственно.

Задача модели базы данных отразить взаимоотношения сущностей внутри приложения. В базе данных присутствуют 5 основных таблиц:

- user;
- book;
- book_group;
- review;
- booking_club_order.

Их задача вместить себе данные необходимые для корректной

обработки запросов, касающихся работы ситсемы в целом.

Так же существуют побочные таблицы:

- `role`.

Данные таблицы являются придаткам к информации о уже существующих таблицах и выделены по правилам нормальных форм баз данных.

Таблицы:

- `groupbook_book`;
- `groupbook_user`;
- `user_roles`.

Небходимы для обеспечения связей «многие ко многим» между основными и побочными таблицами.

Таблица `user` – служит для хранения информации связанной с активностью пользователей в системе. Данная таблица хранит в себе данные для прохождения успешной аутентификации, авторизации, данные о ролях пользователя, а так же личные идентификационные данные.

Таблица `book` – содержит в себе информацию необходимую для эксплуатации основной единицы ресурсов клуба – книг. Таблица `book` хранит данные о названии книги, краткое описание основного сюжета, автора книги и количество таких книг в клубе.

Таблица `book_group` – содержит в себе информацию необходимую для группировки книг по желанию пользователя, данная таблица содержит описание группировки и её название.

Таблица `review` – содержит необходимую информацию о том как оценили данную книгу, оценка состоит из рецензии.

Таблица `booking_club_order` – содержит информацию с помощью которой ведется учёт использования ресурсов книжного клуба его пользователями.

Таблица `role` – необходима для того чтобы хранить роли пользователей в приложении. Является побочной таблицей для таблицы `users`. Пользователь может одновременно иметь несколько полей.

Таблица `groupbook_book` – является необходимой для связи «многие ко многим» между таблицами `book_group` и `book`. Хранит в себе только первичные ключи обеих таблиц.

Таблица `groupbook_user` – является необходимой для связи «многие ко многим» между таблицами `book_group` и `user`. Хранит в себе только первичные ключи обеих таблиц.

Таблица `user_roles` `user` – является необходимой для связи «многие ко многим» между таблицами `roles` и `user`. Хранит в себе только первичные ключи обеих таблиц.

2.2 Блок аутентификации и авторизации пользователя

Блок аутентификации и авторизации пользователя является частью системы, которая отвечает за проверку существования пользователя и в случае его существования в системе, генерирует авторизационный токен для этого пользователя с данными о его правах к доступам к разного рода ресурсам. Для данных целей был выбран фреймворк Spring Security и тип токена - JWT.

Spring Security это Java/Java EE фреймворк, предоставляющий механизмы построения систем аутентификации и авторизации, а также другие возможности обеспечения безопасности для промышленных приложений, созданных с помощью Spring Framework. Проект был начат Беном Алексом (Ben Alex) в конце 2003 года под именем «Acegi Security» и был публично представлен под лицензией Apache License в марте 2004. Впоследствии был включён в Spring как официальный дочерний проект. Впервые публично представлен под новым именем Spring Security 2.0.0 в апреле 2008 года, что включило официальную поддержку и подготовку от SpringSource. Данный фреймворк, сфокусирован на обеспечение как аутентификации, так и авторизации в Java-приложениях. Как и все Spring проекты, настоящая сила Spring Security в том, что он может быть легко дополнен нужным функционалом.

Данный фреймворк был специально разработан для покрытия задач авторизации и аутентификации, он является представителем семейства Spring фреймворков, отлично зарекомендовал себя в таких категориях как:

- удобство использования;
- поддержка разработчиков;
- высокий уровень безопасности;
- поддержка кастомных исключений;
- интеграция.

Данный блок предоставляет функционал для проверки является ли пользователь тем, за кого он пытается себя выдать на основе JWT токена который сохраняется после у пользователя после первого входа в систему и обновляется по мере надобности или на основе комбинации пароля и логина если пользователь пытается зайти в систему в первый раз.

JWT (JSON Web Token) - это открытый стандарт создания токенов доступа информация в котором содержится в виде зашифрованного JSON, что позволяет хранить внутри токена различную информацию. Сам токен состоит из трех частей:

- **HEADER** – который содержит информацию о том какой алгоритм кодировки был использован;
- **PAYLOAD** – данные которые мы передаем внутри токена;
- **VERIFY SIGNATURE** – часть, которая содержит информацию для проверки подлинности токена;

В данном приложении внутри токена хранится такая информация как роль пользователя, который сделал запрос, его id в системе и время, по истечении которого токен будет считаться недействительным. В случае если

время существования токена подходит к концу сервер вместе с ответом посылает новый токен.

Благодаря тому, что есть возможность хранить роль внутри токена можно распределять функционал приложения в зависимости от роли пользователя в системе. И даже выполнять различные действия в ответ на один и тот же запрос в случае его посылают пользователи, обладающие разными ролями, что в значительной степени, облегчает процесс создания правильных Url для связывания с конечными точками приложения.

Каждый токен, который генерируется в системе по стандарту (RFC 7519), должен начинаться со слова Bearer_.

В данном приложении внутри токена хранится такая информация как роль пользователя, который сделал запрос, его id в системе и время, по истечении которого токен будет считаться недействительным. В случае если время существования токена подходит к концу сервер вместе с ответом посылает новый токен.

Благодаря тому, что есть возможность хранить роль внутри токена можно распределять функционал приложения в зависимости от роли пользователя в системе. И даже выполнять различные действия в ответ на один и тот же запрос в случае его посылают пользователи, обладающие разными ролями, что в значительной степени, облегчает процесс создания правильных Url для связывания с конечными точками приложения.

2.3 Блок классов сущностей

Блок классов сущностей содержит классы, которые являются объектным представлением сущностей из таблиц базы данных. При помощи них реализуется связь между объектами в коде и данными в таблице, которая в значительной степени облегчает работу с наборами данных.

2.4 Блок пользовательского интерфейса

Блок пользовательского интерфейса является клиентской частью веб-приложения. Данный блок представляет собой совокупность средств, при помощи которых пользователь взаимодействует с приложением через браузер. Для построения интерфейса используется технология React JS. React JS - фреймворк для создания интерфейсов, созданный компанией Facebook. Он отвечает за представление данных, получение и обработку ввода пользователя. React JS построен на парадигме реактивного программирования. Этот декларативный подход предлагает описывать данные в виде набора утверждений или формул. Изменение одного из параметров ведёт за собой автоматический пересчёт всех зависимостей.

2.5 Блок работы репозитория

Доступ к базе данных из приложения осуществляется посредством работы через API Spring Data.

Spring Data – это проект SpringSource высокого уровня, цель которого – унифицировать и упростить доступ к различным типам хранилищ постоянного, как системам реляционных баз данных, так и хранилищам данных NoSQL.

В не зависимости от типа постоянного хранилища репозитории (также известные как DAO или объекты доступа к данным) обычно предлагают операции CRUD (создание-чтение-обновление-удаление) для объектов одного домена, методов поиска, сортировки и разбивки на страницы. Spring Data предоставляет общие интерфейсы для этих аспектов (`CrudRepository`, `PagingAndSortingRepository`), а также конкретные реализации хранилища сохраняемости.

В данном проекте используются `CrudRepository` и `JpaRepository`.

2.6 Блок работы сервера

Блок работы сервера является центром, на котором будут происходить основные вычисления, которые необходимы для того чтобы создавать, обновлять и получать книги. Данный блок является RESTful сервисом, написанным на языке программирования Java. Java – это строго типизированный язык который благодаря своей популярности обладает множеством библиотек и фреймворков созданных специально для удобства работы разработчиков программного обеспечения.

Во время разработки данного приложения в качестве фреймворка был использован Spring Framework и его расширение Spring Boot.

Spring Framework это универсальный фреймворк который пользуется огромной популярностью среди разработчиков так как обладает множеством модулей каждый из которых направлен на то чтобы облегчить работу разработчиков и уменьшить количество вручную написанного кода.

Spring Boot как расширение Spring Framework обладает всеми преимуществами фреймворка прародителя однако значительно облегчает процесс конфигурации и вводит такие новые понятия как аннотации и `yaml` документы которые позволяют конфигурировать фреймворк прямо в коде без нужды создавать огромные и сложно читаемые конфигурационные файлы.

2.7 Блок API

Способ передачи данных на сервер является самым важным элементом всего приложения. Он является связующим звеном между пользовательским интерфейсом и сервером. Передача данных происходит при помощи отправления запросов по HTTP протоколу на endpoint контроллеров сервера. Запросы, которые посылает данный блок могут быть нескольких типов самые популярные запросы это:

- GET получить некоторые данные;
- POST создать новую запись;
- PATCH обновить уже существующий данные;
- DELETE удалить данные.

Для запросов типа POST и PATCH недостаточно простой проверки того что пользователь тот, за кого себя выдает и может послать такой запрос, для них важно проверить что данные которые пользователь хочет послать на сервисную часть являются корректными, то есть они соответствуют тому типу, который ожидает увидеть сервис и все их значения следуют логике, указанной на серверной части. Такая проверка называется валидация, и она обязательно нужна для того чтобы не позволять пользователю или злоумышленнику сохранять неверные данные.

2.8 Блок бизнес логики

Блок бизнес-логики отвечает за формирование принципов взаимодействия между данными которые вводит пользователь, данными, прошедшими валидацию и данными которые непосредственно лежат в базе данных, то есть за формирование поведения объектов предметной области.

Блок бизнес логики в контексте разработки приложения является слоем, на котором происходят основные вычисления, которые необходимы для того чтобы создавать, обновлять и получать книги. Данный блок является набором методов, разбитых на классы и написанных на языке Java.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В данном разделе описывается функционирование и структура разрабатываемого программного средства.

Взаимоотношения между классами разрабатываемого программного обеспечения приведены на диаграмме классов ГУИР.400201.013 РР.1.

3.1 Описание модели данных

Стоит отметить что для связи «многие ко многим» таблиц `user` и `role`, `book` и `book_group`, `user` и `book_group` были созданы промежуточные таблицы: `user_roles` для `user` и `role`, `groupbook_book` для `book` и `book_group`, `user_bookgroup` для `user` и `book_group`. Также замечу, что для каждого из сервисов используется общая база данных.

3.1.1 Таблица `booking_club_order`

Данная таблица служит для хранения основных данных об аккаунте.

Поля таблицы `booking_club_order`:

- `id` – первичный ключ;
- `create_time` – дата и время создания брони на книгу;
- `end_time` – дата и время окончания использования книги, завершения чтения и возврата в библиотеку;
- `user_id` – внешний ключ для связи с таблицей `user`. Поле хранит данные о юзере взявшим книгу;
- `book_id` – внешний ключ для связи с таблицей `book`. Поле хранит данные о книге для брони;
- `status_id` – состояния брони, (зарезервированно, на руках, не прочитано, прочитано).

Статус брони может иметь два состояния, а именно:

- `Processed` – книга забронирована, на руках и чтение начато;
- `Opened` – книга свободна и никем не забронированна (может быть на руках у участника клуба).

3.1.2 Таблица `book`

Данная таблица, как описывалось выше, служит для хранения данных сущности «книга».

Поля таблицы `book`:

- `book_id` – первичный ключ;

- `author` – имя и фамилия автора, написавшего данную книгу;
- `count` – количество книг в книжном клубе;
- `date_of_creation` – момент добавления книги в библиотеку;
- `description` – краткое описание книги, содержания, небольшие справки об авторе;
- `name` – полное название книги.

3.1.3 Таблица `user`

Данная таблица служит для хранения данных о пользователях приложения.

Поля таблицы `user`:

- `id` – первичный ключ;
- `email` – электронный адрес пользователя;
- `first_name` – имя пользователя;
- `last_name` – фамилия пользователя;
- `name` – имя пользователя;
- `login` – логин пользователя;
- `password` – пароль пользователя.

3.1.4 Таблица `review`

Данная таблица, как описывалось выше, служит для хранения данных о всех рецензиях, отзывах и оценках, которые делают пользователи по отношению к книгам, которые они прочитали, который есть в базе.

Поля таблицы `review`:

- `id` – первичный ключ;
- `date_of_creation` – момент создания отзыва;
- `user_id` – внешний ключ для связи с таблицей `user`, указывает на пользователя создавшего данный отзыв;
- `description` – рецензия пользователя, краткий отзыв или просто комментарий;
- `rating` – оценка данная книге по 10 балльной шкале;
- `topic` – заглавие темы комментария, рецензии или отзыва пользователя;
- `book_id` – внешний ключ для связи с таблицей `book`, указывает на книгу по отношению к которой был создан комментарий, рецензия или оценка от пользователя.

3.1.5 Таблица **book_group**

Данная таблица служит для хранения данных о созданных администраторами или пользователями списков литературы, которые они могут рекомендовать другим пользователям или назначать себе, редактировать каждую таблицу может только создатель данной таблицы или администратор сайта.

Поля таблицы **book_group**:

- **group_id** – id таблицы;
- **name** – название списка.

3.1.6 Таблица **role**

Данная таблица служит для хранения данных о поддерживаемых ролях пользователя в системе.

Поля таблицы **role**:

1. **id** – первичный ключ;
2. **name** – название роли пользователя в системе;
3. **status** – статус в котором находится юзер. Существует только два типа статусов – **active** и **dismissed**. **Active** – юзер разблокирует. **Dismissed** – юзер больше не имеет доступа к ресурсам клуба.

3.1.7 Таблица **user_roles**

В разрабатываемом приложении имеется два типа ролей пользователя: системная роль и роль пользователя в аккаунте. Данная таблица служит для связи между **user** и **role** таблицами, то есть хранит информацию о системной роли пользователя.

Поля таблицы **user_roles**:

- **user_id** – внешний ключ для связи с таблицей **user**;
- **role_id** – внешний ключ для связи с таблицей **role**.

3.1.8 Таблица **status**

Данная таблица служит для хранения данных о статусах, которые могут быть присвоены каждой брони книги.

Поля таблицы **status**:

- **id** – первичный ключ;
- **name** – имя статуса.

3.1.9 Таблица `groupbook_book`

Данная таблица служит для связи `book_group` и `book` таблиц.

Поля таблицы `groupbook_book`:

- `group_id` – внешний ключ для связи с таблицей `group`, определяющая соответствие между группой и книгой;
- `book_id` – внешний ключ для связи с таблицей `group`, определяющая соответствие между группой и книгой.

3.1.10 Таблица `groupbook_user`

Данная таблица служит для связи `book_group` и `user` таблиц.

Поля таблицы `groupbook_book`:

- `user_id` – внешний ключ для связи с таблицей `user`, определяющая соответствие между группой и книгой;
- `book_id` – внешний ключ для связи с таблицей `user`, определяющая соответствие между группой и книгой.

3.2 Описание валидации

3.2.1 Класс `JwtTokenProvider`

Данный класс предоставляет функционал для обеспечения корректной авторизации и аутентификация пользователя, путём генерации токена, получения данных из токена и проверки валидности токена.

Методы класса:

- `init()` – приватный метод экземпляра класса который инициализирует поля класса и создает зашифрованный ключ для валидации токена;
- `createToken()` – публичный метод экземпляра класса который создает токен для пользователя;
- `getAuthentication()` – публичный метод экземпляра класса который получает `Authentication` из токена необходим для создания экземпляра сущности в рамках `Security Context`;
- `resolveToken()` – публичный метод экземпляра класса который получает токен из `HttpServletRequest`;
- `validateToken()` – публичный метод экземпляра класса который проверяет валидность токена;
- `getUserName()` – приватный метод экземпляра класса который возвращает имя пользователя из токена.

3.2.2 Класс `JwtTokenConfigurer`

Данный класс наследует функциональность абстрактного класса `SecurityConfigurerAdapter` `<DefaultSecurityFilterChain, HttpSecurity>`, необходим для конфигурации токена.

Класс `JwtTokenConfigurer` добавляет `JwtTokenFilter` который был выше в цепочку фильтров внутри `HttpSecurity`.

Методы класса:

- `configure()` – публичный метод экземпляра класса который добавляет экземпляр класса `JwtTokenFilter` в цепочку фильтров класса `HttpSecurity`;

3.2.3 Класс `UserPrincipal`

Данный класс реализует интерфейс `UserDetail` и нужен для того чтобы проверять валидность пользователя.

Методы класса:

- `isAccountNonExpired()` – публичный метод экземпляра класса который проверяет данные пользователя, пытающегося получить доступ на то существует ли он еще;

- `isAccountNonLocked()` – публичный метод экземпляра класса который проверяет что пользователь пытающийся получить доступ является активным;

- `isCredentialsNonExpired()` – публичный метод экземпляра класса который проверяет что время действия данных пользователя полученных из токена еще не истекло;

- `isEnabled()` – публичный метод экземпляра класса который проверяет валидность токена;

- `getAuthorities()` – публичный метод экземпляра класса который перебирает список полномочий которыми обладает пользователь, в связи с которыми ему будет дан доступ к контексту приложения.

3.2.4 Класс `UserPrincipalDetailsService`

Данный класс реализует интерфейс `UserDetailsService` и нужен для того получать объект типа `UserPrincipalDetailsService` при помощи имени пользователя, который хочет войти в систему.

Методы класса:

- `loadUserByUsername()` – публичный метод экземпляра класса который возвращает объект типа `UserPrincipalDetailsService` при помощи имени пользователя, который хочет войти в систему, путём выгрузки его из контекста по логину.

3.2.5 Класс JwtTokenFilter

Класс JwtTokenFilter наследует функциональность от абстрактного класса GenericFilterBean.

JwtTokenFilter это класс типа Filter который проверяет валидность токена и обладает следующими методами;

- doFilter() – публичный метод экземпляра класса который ловит запросы к серверу и добавляет Authentication на основе токена из запроса.

3.3 Описание работы контроллера

3.3.1 Класс UserController

Данный класс-контроллер используется для обработки всех запросов, связанных с пользователями и возвращает ответ в формате JSON.

Методы класса-контроллера:

1. getUser() – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/users/{id}» и вызывает получение данных о конкретном пользователе, рядовому пользователю доступна только информация о самом себе, администратору о любом пользователе. Ответ на запрос приходит в формате JSON.

2. updateUser() – публичный метод экземпляра класса который соответствует типу PUT, REST запроса, вызывается при помощи запроса по URL с путем «/users/{id}» и обновляет данные о существующем пользователе полностью в соответствии с пришедшим форматом данных, для рядовых пользователей доступно редактирование только своих данных, для администратора доступно редактирование данных любого аккаунта. Ответ на запрос приходит в формате JSON.

3. updateUser() – публичный метод экземпляра класса который соответствует типу PATCH, REST запроса, вызывается при помощи запроса по URL с путем «/users/{id}/params» и обновляет данные о существующем пользователе относительно одного конкретного поля, для рядовых пользователей доступно редактирование только своих данных, для администратора доступно редактирование данных любого аккаунта. Ответ на запрос приходит в формате JSON.

4. getAllUsers() – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/users/{id}» от лица клиента с ролью администратор и вызывает постраничное получение всех пользователей. Ответ на запрос приходит в формате JSON.

5. `removeUser()` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «`/users/{id}`» и вызывает удаление пользователя из системы. Ответ на запрос приходит в формате JSON.

6. `getUserOrders()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/users/{id}/orders`» от лица клиента с ролью администратор для всех юзеров и от лица клиента с ролью пользователь только для себя самого и вызывает постраничное получение всех броней произведенным указанным в поле `id` пользователем. Ответ на запрос приходит в формате JSON.

7. `getUserReviews()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/users/{id}/reviews`» от лица клиента с ролью администратор для всех юзеров и от лица клиента с ролью пользователь только для себя самого и вызывает постраничное получение всех отзывов, рецензий или оценок сделанных пользователем. Ответ на запрос приходит в формате JSON.

8. `getBookGroups()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/users/{id}/groups`» от лица клиента с ролью администратор для всех юзеров и от лица клиента с ролью пользователь только для себя самого и вызывает постраничное получение всех списков составленных текущим пользователем. Ответ на запрос приходит в формате JSON.

3.3.2 Класс `OrderController`

Данный класс-контроллер используется для обработки всех запросов, связанных с заказами и возвращает ответ в формате JSON.

Методы класса-контроллера:

1. `getOrder()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/orders/{id}`» и вызывает получение данных о брони ресурса (книги) пользователя который сделал запрос. Ответ на запрос приходит в формате JSON.

2. `updateOrder()` – публичный метод экземпляра класса который соответствует типу PUT, REST запроса, вызывается при помощи запроса по URL с путем «`/orders/{id}`» и вызывает обновление данных о брони. Ответ на запрос приходит в формате JSON.

3. `updateOrder()` – публичный метод экземпляра класса который соответствует типу PATCH, REST запроса, вызывается при помощи запроса по URL с путем «`/orders/{id}/{params}`» и вызывает обновление данных о

брони относительно отдельного поле или полей, указанных в переменной запроса «params». Ответ на запрос приходит в формате JSON.

4. `removeOrder()` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «/orders/{id}» и вызывает удаление заказа пользователя. Ответ на запрос приходит в формате JSON.

5. `getAllOrders()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/orders» и вызывает получение данных о всех заказах пользователя который сделал запрос. Ответ на запрос приходит в формате JSON.

6. `createOrder()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/orders/{id}» и вызывает создание нового заказа от лица пользователя который сделал запрос. Ответ на запрос приходит в формате JSON.

3.3.3 Класс AuthenticationController

Данный класс-контроллер необходим для корректного обеспечения функционала аутентификации на сайте, для дальнейшего доступа к ограниченным ресурсам.

Методы класса-контроллера:

1. `login()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/login» и вызывает процесс авторизации пользователя и в случае успеха возвращает JWT token. Ответ на запрос приходит в формате JSON.

2. `registration()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/registration» и вызывает процесс регистрации пользователя в системе и в случае успеха возвращает JWT token. Ответ на запрос приходит в формате JSON.

3. `logout()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/logout» и вызывает процесс выхода пользователя из своего аккаунта и в случае успеха удаляет JWT token. Ответ на запрос приходит в формате JSON.

3.3.4 Класс BookController

Данный класс-контроллер используется для обработки всех запросов, связанных с книгами и возвращает ответ в формате JSON.

Методы класса-контроллера:

1. `removeBook()` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «/books/{id}» от клиента с ролью администратор и вызывает удаление книги из системы. Ответ на запрос приходит в формате JSON.

2. `getBook()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/orders/{id}» результатом которого является получение всех данных о книге с определенным идентификатором. Ответ на запрос приходит в формате JSON.

3. `updateBook()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/orders/{id}» результатом которого является обновление данных о книге. Ответ на запрос приходит в формате JSON.

4. `getAllBooks()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/books» от клиента с ролью администратор результатом которого является получение данных о всех книгах. Ответ на запрос приходит в формате JSON.

5. `getBookReviews()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/books/{id}/reviews» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является получение данных о рецензиях на книгу, оценках книги и комментариев к книге. Ответ на запрос приходит в формате JSON.

6. `createBook()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/books» от клиента с ролью администратор результатом которого является создание новой книги. Ответ на запрос приходит в формате JSON.

7. `getBookOwner()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «/books/{id}/users» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является получение данных о пользователе владеющем в данный момент книгой. Ответ на запрос приходит в формате JSON.

8. `addBookReview()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «/books/{id}/review» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является обновление данных о книге, добавлением комментария к книге, рецензии на книгу или просто оценки книги. Ответ на запрос приходит в формате JSON.

3.3.5 Класс **BookGroupController**

Используется данный класс-контроллер для обработки всех запросов, связанных со списками книг формируемыми пользователями, а также генерирует ответ в json формате.

Методы класса-контроллера:

1. `getBookGroupByName()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/groups/{name}`» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является получение данных о списке книг. Ответ на запрос приходит в формате JSON.

2. `addBookGroupsToUser()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «`/groups/{id}/user/{id}`» от клиента с ролью администратор и от любого авторизованного пользователя в рамках работы со своим аккаунтом результатом которого является добавление нового списка книг. Ответ на запрос приходит в формате JSON.

3. `deleteBookGroupsFromUser()` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «`/groups/{id}`» от клиента с ролью администратор и от любого авторизованного пользователя в рамках работы со своим аккаунтом результатом которого является удаление списка книг. Ответ на запрос приходит в формате JSON.

4. `addBookToBookGroup()` – публичный метод экземпляра класса который соответствует типу PATCH, REST запроса, вызывается при помощи запроса по URL с путем «`/groups/{id}/book/{id}`» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является обновление данных о списке книг. Ответ на запрос приходит в формате JSON.

5. `getAllBookGroups()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/groups`» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является обновление данных о списке книг. Ответ на запрос приходит в формате JSON.

6. `createBookGroup()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «`/groups`» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является создание нового списка книг. Ответ на запрос приходит в формате JSON.

7. `deleteBookGroup()` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «`/groups/{id}`» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является удаление списка книг. Ответ на запрос приходит в формате JSON.

8. `deleteBookFromBookGroup` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «`/groups/{id}`» от клиента с ролью администратор результатом которого является удаление списка книг. Ответ на запрос приходит в формате JSON.

3.3.6 Класс `ReviewController`

Используется данный класс-контроллер для обработки всех запросов, связанных с рецензиями на книги, комментарии к книгам и оценки книг от пользователей, авторизированных на данном сайте, а также данный контроллер генерирует ответ в json формате.

Методы класса-контроллера:

1. `getReview()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/reviews`» от клиента с ролью администратор и от любого авторизованного пользователя результатом которого является получение рецензии на книгу. Ответ на запрос приходит в формате JSON;

2. `getReviewsByUser()` – публичный метод экземпляра класса который соответствует типу GET, REST запроса, вызывается при помощи запроса по URL с путем «`/reviews/{params}`» от клиента с ролью администратор и от любого авторизованного пользователя в рамках работы со своим аккаунтом результатом которого является получение рецензий на книгу, сделанных конкретным пользователем. Ответ на запрос приходит в формате JSON.

3. `updateReview()` – публичный метод экземпляра класса который соответствует типу PUT, REST запроса, вызывается при помощи запроса по URL с путем «`/reviews/{id}`» от клиента с ролью администратор и от любого авторизованного пользователя в рамках работы со своим аккаунтом результатом которого является обновление рецензии. Ответ на запрос приходит в формате JSON.

4. `createReview()` – публичный метод экземпляра класса который соответствует типу POST, REST запроса, вызывается при помощи запроса по URL с путем «`/reviews`» от клиента с ролью администратор и от любого авторизованного пользователя в рамках работы со своим аккаунтом результатом которого является создание рецензии. Ответ на запрос приходит в формате JSON.

5. `removeReview()` – публичный метод экземпляра класса который соответствует типу DELETE, REST запроса, вызывается при помощи запроса по URL с путем «`/reviews/{id}`» от клиента с ролью администратор и от любого авторизованного пользователя в рамках работы со своим аккаунтом результатом которого является удаление рецензии. Ответ на запрос приходит в формате JSON.

3.3.7 Класс BookDTO

Класс BookDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для создания, обновления, удаления и редактирования книги. А так же для предоставления полей, необходимых клиенту, но не отображаемых в базе данных.

3.3.8 Класс ReviewDTO

Класс ReviewDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для для создания, обновления, удаления и редактирования отзывов.

3.3.9 Класс UserDTO

Класс UserDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для для создания, обновления, удаления и редактирования пользователя.

3.3.10 Класс BookGroupDTO

Класс BookGroupDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для создания, обновления, удаления и редактирования контекстного списка книг.

3.3.11 Класс AuthenticationRequestDTO

Класс AuthenticationRequestDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для аутентификации и регистрации пользователя.

3.3.12 Класс ApiErrorDTO

Класс ApiErrorDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для вывода сообщения об ошибке.

3.3.13 Класс OrderDTO

Класс OrderDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для создания, обновления, удаления и редактирования брони на книгу.

3.3.14 Класс BookListDTO

Класс BookListDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для вывода списка книг, которые подходят под требование запроса.

3.3.15 Класс UserListDTO

Класс UserListDTO является классом типа DTO (Data Transfer Object) и нужен для представления на сервисе данных, который представляет клиент для вывода списка пользователей, которые подходят под требование запроса.

3.3.16 Интерфейс ConverterDtoToPojo<T, V>

Данный интерфейс служит для преобразования объектов из представления на сервисе данных, которые представляет клиент, в тип объектов которые работают непосредственно с базой данных.

3.3.17 Класс BookConverter<T, V>

Данный класс предназначен для конвертации служебных объектов Book и BookDto между слоями, учитывая их особенности. В зависимости от указанных параметров происходит конвертация либо из Pojo формата, необходимого для работы сервиса и репозитория в Dto формат, необходимый для работы контроллера, либо наоборот, из Dto формата в Pojo формат.

3.3.18 Класс UserConverter<T, V>

Данный класс предназначен для конвертации служебных объектов User и UserDto между слоями, учитывая их особенности. В зависимости от указанных параметров происходит конвертация либо из Pojo формата, необходимого для работы сервиса и репозитория в Dto формат, необходимый для работы контроллера, либо наоборот, из Dto формата в Pojo формат.

3.3.19 Класс BookGroupConverter<T, V>

Данный класс предназначен для конвертации служебных объектов BookGroup и BookGroupDto между слоями, учитывая их особенности. В зависимости от указанных параметров происходит конвертация либо из Pojo формата, необходимого для работы сервиса и репозитория в Dto формат, необходимый для работы контроллера, либо наоборот, из Dto формата в Pojo формат.

3.3.20 Класс OrderConverter<T, V>

Данный класс предназначен для конвертации служебных объектов Order и OrderDto между слоями, учитывая их особенности. В зависимости от указанных параметров происходит конвертация либо из Pojo формата, необходимого для работы сервиса и репозитория в Dto формат, необходимый для работы контроллера, либо наоборот, из Dto формата в Pojo формат.

3.3.21 Класс RoleConverter<T, V>

Данный класс предназначен для конвертации служебных объектов Role и RoleDto между слоями, учитывая их особенности. В зависимости от указанных параметров происходит конвертация либо из Pojo формата, необходимого для работы сервиса и репозитория в Dto формат, необходимый для работы контроллера, либо наоборот, из Dto формата в Pojo формат.

3.3.22 Класс UserRegistrationConverter<T, V>

Данный класс предназначен для конвертации служебных объектов User и RegistrationUserDto между слоями, учитывая их особенности. В зависимости от указанных параметров происходит конвертация либо из Pojo формата, необходимого для работы сервиса и репозитория в Dto формат, необходимый для работы контроллера, либо наоборот, из Dto формата в Pojo формат.

3.4 Описание работы сервиса

3.4.1 Класс BookService

Класс BookService предназначен для обработки запросов связанных с управлением ресурсов клуба.

Собственные методы класса:

- `getBook()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения данных о конкретной книге.
- `getBookOrderByBookId()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получение списка всех заказов связанных с конкретной книгой.
- `getUserBooks()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех книгах принадлежащих конкретному пользователю.
- `createBook()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для создания администратором новой книги которую будут использовать члены клуба.
- `updateBook()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления данных о книге.
- `updateBook()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления данных о книге.
- `getBookCount()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о количестве свободных книг в библиотеке.
- `addBookComment()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для добавления комментария к книге.
- `getBookComments()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для просмотра всех комментариев к книге.

3.4.2 Класс AuthenticationService

Класс `AuthenticationService` предназначен для обработки запросов связанных с аутентификацией членов клуба.

Собственные методы класса:

- `createUser()` – приватный метод экземпляра класса который создаёт нового пользователя в системе;

3.4.3 Класс OrderService

Класс `OrderService` предназначен для обработки запросов связанных с аутентификацией членов клуба.

Собственные методы класса:

- `getOrdersByBookId()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех забронированных книг пользователя.

- `getOrderByBookAndOwnerId()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех забронированных книг пользователя.
- `getOrdersByOwnerId()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех забронированных книг пользователя.
- `createOrder()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для создания новой брони для конкретной книги клуба.
- `updateOrder()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления данных бронирования книги клуба.
- `getAllOrders()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения данных о всех бронях происходящих внутри клуба.
- `removeOrder()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для удаления брони на книгу. Доступно только администраторам.
- `getOrders()` – публичный метод экземпляра класса, который постранично возвращает заказы по идентификатору пользователя;
- `getOrder()` – публичный метод экземпляра класса, который возвращает заказ по его идентификатору;

3.4.4 Класс BookGroupService

Класс `BookGroupService` предназначен для обработки запросов связанных с организацией списков книг.

Собственные методы класса:

- `getBookGroupsByUser()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о списках книг принадлежащих пользователю;
- `assignBookGroupToUser()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для добавления списка книг к пользователю;
- `deleteBookGroupFromUser()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для удаления списка книг от пользователя;
- `updateBookGroup()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления информации о списке книг;

3.4.5 Класс ReviewService

Класс ReviewService предназначен для обработки запросов связанных с получением информации о рецензиях, созданием рецензий и их удалением.

Собственные методы класса:

- `getReviews()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех рецензиях ко всем книгам;
- `updateReview()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления информации внутри конкретной рецензии;
- `createReview()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для создания новой рецензии к конкретной книге;
- `removeReview()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для удаления существующей рецензии;
- `getReviewByBookId()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения рецензий для конкретной книги.

3.4.6 Класс UserService

Класс UserService предназначен для обработки запросов, связанных с управлением данными о пользователях системы.

Собственные методы класса:

- `getUser()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о конкретном пользователе;
- `updateUser()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления информации касающейся конкретного пользователя;
- `getUserOrders()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения всех броней пользователя на книги клуба;
- `getUserReviews()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех рецензиях пользователя;
- `getUserBookGroups()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех списках которые добавил себе пользователь.

- `createUser()` – публичный метод экземпляра класса, который создает пользователя;
- `getAll()` – публичный метод экземпляра класса, который возвращает всех пользователей;
- `getAllByPartialName()` – публичный метод экземпляра класса, который возвращает всех пользователей у которых есть частичное совпадение имени;
- `removeUser()` – публичный метод экземпляра класса, который удаляет пользователя;
- `updateUser()` – публичный метод экземпляра класса, который обновляет пользователя.

3.4.7 Класс `OffsetBasedPageRequest`

Для получения сервисом данных из репозитория, который работает используя механизм взаимодействия с сущностями баз данных – Spring Data, необходимо использовать объект интерфейса `Pageable`, так как простое смещение по `id` нам не подходит в связи с тем, что объекты в базе данных имеют свойство быть удалёнными. Объект интерфейса `Pageable` позволяет нам обратиться к репозиторию `JpaRepository` для получения данных из таблицы, ограничиваемых количеством данных для вывода (`limit`) и устанавливаемым начальным значением (`offset`), путём использования метода `findAll(pageable)`.

Для реализации интерфейса `Pageable` разработан класс `OffsetBasedPageRequest`, который принимает значения `limit` и `offset`.

Собственные методы класса:

- `getPageNumber()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о текущей странице;
- `getPageSize()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления информации о размере каждой отображаемой странице;
- `getOffset()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о сдвиге с которого будут выводиться данные;
- `getSort()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о типе сортировки данных;
- `next()` – публичный метод экземпляра класса который вызывается в сервисе, необходим для определения данных на следующей странице.

- `previous()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о предыдущей странице;
- `previousOrFirst()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для перехода на предыдущую страницу, но с проверкой что текущая страница не первая;
- `first()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о том не стоит ли обработка на первой странице;
- `hasPrevious()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления информации о том, были ли страницы до текущей;

3.4.8 Класс `BookServiceJpa`

Класс `BookServiceJpa` является реализацией интерфейса `BookService`. Данный класс обладает как реализованными методами интерфейса, так и собственными приватными методами, необходимыми для обработки запросов.

Собственные методы класса:

- `findBookInRepository()` - приватный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о книге по ее первичному ключу;

3.4.9 Класс `BookGroupServiceJpa`

Класс `GenreServiceImpl` является реализацией интерфейса `GenreService`. Данный класс обладает только реализованными методами интерфейса и подставляется в качестве его реализации в классе `GenreController`.

Собственные методы класса:

- `findGroupInRepository()` - приватный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о списке книг по его первичному ключу;

3.4.10 Класс `ReviewServiceJpa`

Класс `ReviewServiceJpa` является реализацией интерфейса `ReviewService`. Данный класс обладает как реализованными методами интерфейса, так и собственными приватными методами.

Собственные методы класса:

- `find()` - приватный метод экземпляра класса который вызывается в сервисе, необходим для получения информации об отзывах и выбрасывающий исключение при некорректных данных в запросе;

3.4.11 Класс `UserServiceJpa`

Класс `UserServiceJpa` является реализацией интерфейса `UserService`. Данный класс обладает только реализованными методами интерфейса и подставляется в качестве его реализации в классе `UserController`.

Собственные методы класса:

- `loadByUsername()` - приватный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о пользователе по его логину;

3.4.12 Класс `OrderServiceJpa`

Класс `OrderServiceJpa` является реализацией интерфейса `OrderService`. Данный класс обладает как реализованными методами интерфейса, так и собственными приватными методами. Собственные методы класса:

- `findByUserRepository()` - приватный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о списке заказов по первичному ключу указываемого пользователя;

3.4.13 Класс `RoleServiceJpa`

Класс `RoleServiceJpa` является реализацией интерфейса `CustomerService`. Данный класс обладает как реализованными методами интерфейса, так и собственными приватными методами.

Собственные методы класса:

- `findRoleInRepository()` - приватный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о роли пользователя;

3.4.14 Класс `BookGroupValidator`

Класс `BookGroupValidator` проверяет что книга корректность данных, введенных для работы с выборками книг.

3.4.15 Класс BookValidator

Класс BookValidator проверяет что книга существует и ее можно удалить или изменить.

3.4.16 Класс ReviewValidator

Класс ReviewValidator проверяет что такой отзыв существует и его можно удалить или изменить.

3.4.17 Класс OrderValidator

Класс OrderValidator проверяет что заказ существует и его можно удалить или изменить.

Проверяет связь между бронью или заказом и пользователем.

3.4.18 Класс UserValidator

Класс UserValidator проверяет что пользователь существует и его можно удалить или изменить.

Необходим для повторной валидации введенных учетных данных и для внутренних проверок, соответствия полей заданному формату.

3.5 Описание работы репозитория

3.5.1 Класс BookRepository

Класс BookRepository предназначен для обработки запросов к базе данных, связанных с получением из нее данных о книгах как ресурсах системы.

Собственные методы класса:

- `getBookOrderByBookId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о бронях на книги по их id;
- `updateBook()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для обновления информации касающейся конкретной книги;
- `getBookByUserId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения всех броней пользователя на книги клуба;

- `getBookCountByBookId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о всех рецензиях пользователя;
- `createBook()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о всех списках которые добавил себе пользователь.

3.5.2 Класс `UserAuthenticationRepository`

Класс `UserAuthenticationRepository` предназначен для обработки запросов к базе данных, связанных с получением из нее данных.

Собственные методы класса:

- `getUser()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о конкретном пользователе;

3.5.3 Класс `OrderRepository`

Класс `OrderRepository` предназначен для обработки запросов к базе данных, связанных с получением из нее данных об ордерах пользователей на книги клуба.

Собственные методы класса:

- `getOrdersByBookId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о всех забронированных книг пользователя.
- `getOrderByBookAndOwnerId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о всех забронированных книг пользователя.
- `getOrdersByOwnerId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о всех забронированных книг пользователя.
- `createOrder()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для создания новой брони для конкретной книги клуба.
- `updateOrder()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для обновления данных бронирования книги клуба.
- `getAllOrders()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения данных о всех бронях происходящих внутри клуба.
- `removeOrder()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для удаления брони на книгу. Доступно

только администраторам.

3.5.4 Класс **BookGroupRepository**

Класс **BookGroupRepository** предназначен для обработки запросов к базе данных, связанных с получением из нее данных о списках книг, составленных пользователями.

Собственные методы класса:

- `getBookGroupsByUser()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о списках книг принадлежащих пользователю;
- `assignBookGroupToUser()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для добавления списка книг к пользователю;
- `deleteBookGroupFromUser()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для удаления списка книг от пользователя;
- `updateBookGroup()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для обновления информации о списке книг;

3.5.4 Класс **ReviewRepository**

Класс **ReviewRepository** предназначен для обработки запросов к базе данных, связанных с получением из нее данных о рецензиях, написанных пользователями.

Собственные методы класса:

- `getReviews()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения информации о всех рецензиях ко всем книгам;
- `updateReview()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для обновления информации внутри конкретной рецензии;
- `createReview()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для создания новой рецензии к конкретной книге;
- `removeReview()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для удаления существующей рецензии;
- `getReviewByBookId()` – публичный метод экземпляра класса который вызывается в репозитории, необходим для получения рецензий для конкретной книги.

3.5.5 Класс `UserRepository`

Класс `UserRepository` предназначен для обработки запросов к базе данных, связанных с получением из нее данных пользователей, возможности редактировать эти данные, а так же удалять и добавлять их.

- `getUser()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о конкретном пользователе;

- `updateUser()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для обновления информации касающейся конкретного пользователя;

- `getUserOrders()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для получения всех броней пользователя на книги клуба;

- `getUserReviews()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех рецензиях пользователя;

- `getUserBookGroups()` - публичный метод экземпляра класса который вызывается в сервисе, необходим для получения информации о всех списках которые добавил себе пользователь.

3.5.6 Класс `OrderStatus`

Класс `OrderStatus` предоставляет базовый функционал для отслеживания того, в каком состоянии находится текущий заказ пользователя.

3.5.7 Класс `Book`

Класс `Book` является объектным отображением таблицы `book`. Данный класс используется для управления данными о книгах.

3.5.8 Класс `Review`

Класс `Review` является объектным отображением таблицы `review`. Данный класс используется для управления данными о отзывах.

3.5.9 Класс `BookGroup`

Класс `BookGroup` является объектным отображением таблицы `book_group`. Данный класс используется для управления данными о связи

между книгами и их групповыми списками, которые могут настраивать для себя сами пользователи.

3.5.10 Класс Order

Класс `Order` является объектным отображением таблицы `order`. Данный класс используется для управления данными о заказах, сделанных пользователем, с которым они связаны через другую таблицу.

3.5.11 Класс User

Класс `User` наследует функционал класса `Auditable` и является объектным отображением таблицы `user`. Данный класс используется для управления данными о пользователях.

3.5.12 Класс Role

Класс `Role` является объектным отображением таблицы `role`.

4 РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

В данном разделе описываются основные сценарии и алгоритмы, которые используются в веб-приложении. Схема программы представлена на чертеже ГУИР.400201.033 ПД.

4.1 Получение всех пользователей, с учётом пагинации.

Данный алгоритм описывает получение всех пользователей, зарегистрированных в приложении «Booking Club», с учетом разбиения страниц (пагинации). На стороне пользовательского интерфейса пользователю с ролью администратора или со стандартными правами пользователя, необходимо сделать запрос на получение запрашиваемых данных о всех пользователях. Со стороны клиентской части информация о количестве страниц вычисляется на основе ответа сервера о количестве всех пользователей и на основании закодированного жёстко размера страницы, 5 пользователей на каждой странице.

В связи с этим для составления запроса пользователь не должен передавать никаких параметров, всё задано в клиентском приложении.

После составления запроса клиентом, он (запрос) отправляется на сервер где будет обработан специальным классом-контроллером `UserController` который предназначен специально для обработки всех запросов поступающих от пользователя. На уровне двнного контроллера, будет произведена проверка, что пользователь, который отправил запрос, обладает соответствующими правами для получения данной информации.

Так как данное приложение имеет три уровня прав – администратор, пользоатель и клиент, а доступ к такому типу запросов открыт только для администратора и пользователя, в случае если совершивший запрос не является пользователем и администратором то, если запрос происходит на уровне клиента, приложение производит редирект на страницу логгирования, если же пользователь пытается «достучаться» до данной информации используя предоставленные API сервера, то в ответе ему вернется сообщение с кодом ошибки 403. В случае если пользователь является администратором или стандартным пользователем запрос будет передан валидатору уровня контроллера, классу `ParamsChecker`, где методы `isPageParamCorrect()` и `isSizeParamCorrect()` в случае некорректных данных выбросят `exception` контроллеру, после чего контроллер вернет отшибку с кодом 400. В случае корректности пришедших параметров, управление передаётся на уровень сервиса, методу `findAllUsersWithPagination(page, size)`.

Для получения сервисом данных из репозитория, который работает используя механизм взаимодействия с сущностями баз данных – Spring Data, необходимо использовать объект интерфейса `Pageable`, так как простое

смещение по id нам не подходит в связи с тем, что объекты в базе данных имеют свойство быть удалёнными. Объект интерфейса Pageable позволяет нам обратиться к репозиторию JpaRepository для получения данных из таблицы, ограничиваемых количеством данных для вывода (limit) и устанавливаемым начальным значением (offset), путём использования метода findAll(pageable).

Для реализации интерфейса Pageable, разработан класс OffsetBasedPageRequest, который принимает значения limit и offset, где limit:

```
int limit = page * size - size;
```

Формирование объекта Pageable выглядит так:

```
Pageable pageable = new OffsetBasedPageRequest(size, limit);
```

После того как будет сформирован объект Pageable, метод сервиса findAllUsersWithPagination(page, size) вернет список пользователей согласно заданным параметрам:

```
return userRepository.findAll(pageable).getContent();
```

Далее контроллер конвертирует полученный объект List<User> в объект List<UserDto>, используя метод convert класса DtoConverter<UserDto, User>:

```
List<UserDto> searchedUsers = converter.convert  
(service.findAllUsersWithPagination(page, size));
```

Однако, для того чтобы клиент мог грамотно обработать отображение возможных страниц для переходов при просмотре пользователей, ему так же необходимо в ответе получить информацию о числе абсолютно всех пользователей зарегистрированных в книжном клубе. Делается это получением количества возвращаемых элементов в методе сервиса findAllUsers(), количество возвращаемых объектов методом помещается в переменную count:

```
int count = service.findAllUsers().size();
```

Далее формируется объект ответа, в ответ мы помещаем объект класса UserList, который имеет необходимые поля для обработки ответа клиентом - список List<User> items и количество всех пользователей, которые могут быть выведены в рамках данного запроса int usersCount:

```
UserList users = new UserList(items, count);
```

Объект ответа – `ResponseEntity`, принимает в тело объект `UserList` и константное значение HTTP статуса ответа `HttpStatus.OK`, сам принимающий метод контроллера возвращает объект `ResponseEntity` как ответ на запрос передаётся на слой пользовательского интерфейса:

```
return new ResponseEntity<>(users, HttpStatus.OK);
```

4.2 Получение информации о конкретной книге

Данные клуба «BookingClub» принадлежат только участникам, зарегистрированным в данном клубе. Весь алгоритм просмотра детальной информации о ресурсе (книге) клуба состоит из 4-х основных частей – регистрация (для нового пользователя), аутентификация (для пользователя который еще не прошел её), вывод списка всех книг, выбор необходимой книги.

4.2.1 Регистрация пользователя

Так как приложение клуба «BookingClub» является приложением с открытой регистрацией и закрытым доступом для не аутентифицированных пользователей, чтобы получить возможность к аутентификации, новый пользователь должен попасть в `SpringSecurityContext`, чтобы приложение его «знало». Для этого новому пользователю необходимо пройти регистрацию в книжном клубе.

На стороне пользовательского интерфейса, для не аутентифицированных пользователей есть возможность перейти на страницу регистрации, где им будет предложено ввести необходимые данные для последующей регистрации в системе.

Для успешного прохождения регистрации система требует пользователя ввести свои имя, фамилию, почту и пароль, который пользователь будет использовать для входа в систему.

После корректно введенных данных выполняется запрос от клиента к серверу и первым делом объект в теле запроса проходит валидацию на корректность введенных данных средствами библиотеки `javax.validation`.

Если валидация успешно пройдена, то обработка запроса передаётся контроллеру, который обрабатывается классом `AuthenticationController`, в обратном случае на клиентскую сторону возвращается сообщение об ошибке с кодом 400.

На стороне сервера, для передачи объекта регистрации на слой сервиса, необходимо сформировать из объекта типа `RegistrationUserDto` объект типа `User`, для этого используется объект `registrationConverter` класса `DtoConverter<RegistrationUserDto>` и его метод `convert`:

```
User serviceUser = registrationConverter.convert(userDto);
```

Далее методом `create` класса `UserService` запрос передаётся на сервисный слой для итогового получения зарегистрированного пользователя:

```
User user = service.create(serviceUser);
```

Внутри сервисного слоя формируются все данные для помещения нового пользователя в базу данных, первым делом создаётся объект класса `BCryptPasswordEncoder`, для хеширования переданного пользователем пароля, внутри базы данных:

```
BCryptPasswordEncoder encoder = new BCryptPasswordEncoder()
```

Следующим шагом является непосредственно хеширование пароля пользователя. Для этого вытягивается пароль пользователя из объекта, который пришел от слоя контроллера:

```
String actualPassword = user.getPassword();
```

Далее текущий пароль хешируется методом `encode` класса `BCryptPasswordEncoder`:

```
String hashedPassword = encoder.encode(actualPassword);
```

И устанавливается в поле пароля пользователя:

```
user.setPassword(hashedPassword);
```

Данная процедура необходима для сохранения приватности данных пользователя в случае их утечки из базы. Если все данные базы, используемой приложением будут доступны сторонним лицам, то с помощью защиты пароля хешированием, данные аутентификации останутся засекреченными.

Далее пользователь с указанными данными сохраняется в базе данных, путём передачи данных запроса на слой репозитория, средствами базы данных ему назначается свой уникальный ключ `id`, данные о новом сохраненном пользователе возвращаются после работы метода репозитория `save(user)`:

```
User mutableUser = userRepository.save(user);
```

Следующим шагом является создание списка ролей, которые получит новый пользователь с последующим присвоением ему (пользователю) этих ролей. В случае данного приложения, при создании, пользователю присваивается стандартная роль `USER`. Роль находится благодаря

использованию класса `RoleRepository` который перебрасывает запрос на уровень репозитория и достаёт необходимую роль:

```
ArrayList<Role> roles = new ArrayList<>();  
roleRepository.findById(userRole).ifPresent(roles::add);
```

Далее роли устанавливаются в поле `roles`, нового пользователя:

```
mutableUser.setRoles(roles);
```

После чего сервис возвращает обработку запроса на уровень контроллера с данными о новом пользователе. Где создается объект класса `UserDto`, из данных которые вернул сервис:

```
UserDto createdUser = new UserDto(user);
```

Следующим шагом формируется и возвращается клиенту ответ `ResponseEntity`, с кодом 200 и с данными о новом пользователе в теле:

```
Return ResponseEntity.ok().body(createdUser);
```

Клиент принимает ответ, но в текущей реализации приложения, он (клиент) никак не обрабатывает данные о новом пользователе, а только выводит информацию об успешной регистрации, затем новый пользователь может перейти на страницу аутентификации, ввести свои данные и пользоваться ресурсами приложения.

4.2.2 Аутентификация пользователя

На стороне пользовательского интерфейса пользователю который еще не зарегистрирован в приложении `BookingClub` при попытке получить доступ к ресурсам клуба будет отказано в доступе, первыми шагами нового пользователя является регистрация, аутентификация и авторизация. В данном разделе будет рассмотрена аутентификация пользователя.

Пользователь попавший на сайт, но не аутентифицированный имеет возможности перейти на страницу входа. Данная страница содержит два поля – `login` поле и поле ввода пароля (поле `password`). После ввода учетных данных необходимо нажать на кнопку подтверждения. В случае, если пользователь не ввёл данные в какое-либо поле, появится сообщение об ошибке. При вводе корректных данных запрос отправляется на сторону сервера, где обрабатывается приложением.

Данные, которые получает сервер лежат в теле запроса, который отправляет клиентское приложение. Эти данные принимает объект `AuthenticationRequestDto`.

Аутентификацию данных в приложении производит объект из библиотеки Spring Security – `AuthenticationManager`.

`AuthenticationManager` это просто интерфейс, поэтому реализация может быть какой угодно, в зависимости от выбора. Реализация по умолчанию в Spring Security называется `ProviderManager` и вместо того, чтобы самостоятельно обрабатывать аутентификационный запрос, он делегирует это списку настроенных `AuthenticationProvider`'ов, каждый из которых в свою очередь запрашивается, может ли он выполнить аутентификацию. Каждый провайдер либо сгенерирует исключение, либо вернет полностью заполненный объект `Authentication`.

Наиболее распространенный подход к проверке аутентификационного запроса – это, загрузить соответствующий `UserDetails` и сверить загруженный пароль с тем, что был введен пользователем. Это подход используется `DaoAuthenticationProvider`. Загруженный `UserDetails` объект - и в частности `GrantedAuthority` в нем - будут использоваться при создании полностью заполненного объекта `Authentication`, который возвращается в случае успешной аутентификации и сохраняется в `SecurityContext`.

Если же используется конфигурирование с помощью пространства имен, то экземпляр `ProviderManager` создается и поддерживается автоматически, и необходимо только добавить провайдеров аутентификации с помощью элементов пространства имен. В этом случае, не нужно объявлять bean для `ProviderManager` в контексте приложения.

Во время аутентификации используется метод класса `AuthenticationManager` – `authenticate`. Для его корректной работы необходимо передать ему объект класса расширяющего интерфейс `Authentication`.

Для расширения объекта `Authentication` в данном приложении используется класс `UsernamePasswordAuthenticationToken`. Чтобы создать объект данного класса и передать его методу `authenticate` для обработки и проверки, необходимы данные введенные пользователем для аутентификации – его логин и пароль.

Логин и пароль пользователя, который производит аутентификацию достаётся из объекта класса `AuthenticationRequestDto` – `requestDto`:

```
String username = requestDto.getLogin();  
String pass = requestDto.getPassword();
```

Из полученных переменных `username` и `pass` формируется объект класса `UsernamePasswordAuthenticationToken` необходимый для проведения валидации:

```
Authentication authenticationUser =  
    newUsernamePasswordAuthenticationToken(username, pass);
```

Далее объект класса UsernamePasswordAuthenticationToken передаётся методу `authenticate` класса `AuthenticationManager`, где сформированный объект `authenticationUser` состоящий из данных введенных пользователем проходит валидацию учётных данных.

```
authenticationManager.authenticate(authUser);
```

Далее в случае если данные в объекте `Authentication` оказываются не верными или объект с такими данными оказывается по какой-либо причине не загружен в `SecurityContext`, то метод `authenticate` выбрасывает исключение `AuthenticationException`, которое ловится `try-catch` блоком, откуда, в случае возникшего исключения будет возвращён объект `responseMap` с пустыми полями `username` – для обработки клиентом и получения информации о текущем логине данного пользователя, пустое поле возвращается в по контракту клиента и сервера:

```
String parameterUsername = "username";  
responseMap.put(parameterUsername, "");
```

Так же возвращается пустое поле `token`:

```
String parameterToken = "token";  
responseMap.put(parameterToken, "");
```

И поле `isLogged` с `boolean` параметром `false`:

```
String parameterIsLogged = "isLogged";  
responseMap.put(parameterIsLogged, false);
```

В случае когда метод `authenticate` не выбрасывает никаких исключений, он возвращает объект `Authentication`, который в данной реализации приложения не используется. Метод `Authenticate` необходим исключительно для проверки подлинности.

Следующим этапом аутентификации является получение JWT-токена из данных о пользователе, для этого мы, используя класс `UserService` и данные введенные пользователем, а именно – логин, получаем информацию о текущем пользователе из базы данных, путём передачи запроса на уровень сервиса.

```
User user = service.findByLogin(username);
```

На уровне сервиса, в методе класса `UserService` – `findByLogin(String)`, запрос сразу же передаётся на уровень

репозитория, чтобы вернуть ответ из базы данных, далее метод сервиса возвращается заполненный объект User:

```
return userRepository.findByLogin(username);
```

Для последующей конвертации его в объект UserDto через конструктор:

```
UserDto userDto = new UserDto(user);
```

Далее необходимо сформировать сам токен, который в себе будет нести информацию о логине пользователя и о его ролях, для дальнейшей авторизации:

```
String token = jwtToken.createToken(username,  
userDto.getRoles());
```

После успешного формирования токена будет сформирован объект responseMap с заполненным полем username – для обработки клиентом и получения информации о текущем логине пользователя:

```
String parameterUsername = "username";  
responseMap.put(parameterUsername, user.getLogin());
```

Так же заполняется поле токен:

```
String parameterToken = "token";  
responseMap.put(parameterToken, token);
```

И поле isLogged boolean параметром true:

```
String parameterIsLogged = "isLogged";  
responseMap.put(parameterIsLogged, true);
```

После заполнения, объект responseMap отправляется на клиентскую часть приложения внутри объекта ResponseEntity, где обрабатывается, для дальнейшего использования в запросах:

```
return ResponseEntity.ok(responseMap);
```

4.2.3 Вывод списка всех книг, с учётом пагинации

Получение всех книг зарегистрированных в приложении «BookingClub» с учетом разбиения страниц (пагинации) доступно пользователю с ролью администратора или со стандартными правами пользователя. Для получения

списка всех книг пользователю с вышеупомянутыми правами необходимо сделать запрос на получение запрашиваемых данных. Со стороны клиентской части, информация о количестве страниц вычисляется на основе ответа сервера о количестве всех пользователей и на основании закодированного жёстко размера страницы, 5 пользователей на каждой странице.

В связи с этим для составления запроса пользователь не должен передавать никаких параметров, всё задано в клиентском приложении.

После составления запроса клиентом, он (запрос) отправляется на сервер где будет обработан специальным классом-контроллером `BookController`.

На уровне данного контроллера, будет произведена проверка что пользователь, который отправил запрос обладает соответствующими правами для получения данной информации, в случае если совершивший запрос не является пользователем и администратором то, если запрос происходит на уровне клиента, приложение производит перенаправление на страницу входа, если же пользователь пытается достучаться до данной информации используя предоставленные API сервера, то в ответе ему вернется сообщение с кодом ошибки 403. В случае если пользователь является администратором или стандартным пользователем запрос будет передан валидатору уровня контроллера, классу `ParamsChecker`, где методы `isPageParamCorrect()` и `isSizeParamCorrect()` в случае некорректных данных выбросят `exception` контроллеру, после чего контроллер вернет отшибку с кодом 400, в случае корректности пришедших параметров, управление передаётся на уровень сервиса, методу `findAllBooksWithPagination(page, size)`.

Для получения сервисом данных из репозитория, который работает используя механизм взаимодействия с сущностями баз данных – Spring Data, необходимо использовать объект интерфейса `Pageable`, так как простое смещение по `id` нам не подходит в связи с тем, что объекты в базе данных имеют свойство быть удалёнными. Объект интерфейса `Pageable` позволяет нам обратиться к репозиторию `JpaRepository` для получения данных из таблицы, ограничиваемых количеством данных для вывода (`limit`) и устанавливаемым начальным значением (`offset`), путём использования метода `findAll(pageable)`.

Для реализации интерфейса `Pageable` разработан класс `OffsetBasedPageRequest`, который принимает значения `limit` и `offset`, где `limit`:

```
int limit = page * size - size;
```

Формирование объекта `Pageable` выглядит так:

```
Pageable pageable = new OffsetBasedPageRequest(size, limit);
```

После того как будет сформирован объект `Pageable`, метод сервиса `findAllBooksWithPagination(page, size)`, вернет список пользователей согласно заданным параметрам:

```
return bookRepository.findAll(pageable).getContent();
```

Далее контроллер конвертирует полученный объект `List<Book>` в объект `List<BookDto>`, используя метода `convert` класса `DtoConverter<BookDto, Book>`:

```
List<UserDto> searchedbooks = converter.convert  
(service.findAllBooksWithPagination(page, size));
```

Однако для того чтобы клиент мог грамотно обработать отображение возможных страниц для переходов при просмотре пользователей, ему так же необходимо в ответе получить информацию о числе абсолютно всех пользователей зарегистрированных в книжном клубе. Делается это получением количества возвращаемых элементов в методе сервиса `findAllUsers`, количество возвращаемых объектов методом помещается в переменную `count`:

```
int count = service.findAllBooks().size();
```

Далее формируется объект ответа, в ответ мы помещаем объект класса `BookList`, который имеет необходимые поля для обработки ответа клиентом - список `List<Book> items` и количество всех юзеров, которые могут быть выведены в рамках данного запроса `int usersCount`:

```
BookList books = new BookList(items, count);
```

Объект ответа – `ResponseEntity`, принимает в тело объект `BookList` и константное значение HTTP статуса ответа `HttpStatus.OK`, сам принимающий метод контроллера возвращает объект `ResponseEntity` как ответ на запрос передаётся на слой пользовательского интерфейса:

```
return new ResponseEntity<>(books, HttpStatus.OK);
```

4.2.3 Вывод информации о конкретной книге

После того как пользователь получил возможность увидеть список из всех имеющихся книг, у него так же предоставляется возможность детальнее рассмотреть выбранную книгу, раскрыть меню добавления и так далее. Что бы это сделать, ему необходимо нажать по понравившейся книге, клиентское приложение отреагирует на действие и пошлёт запрос с данными на сервер.

На серверном приложении запрос данного типа обрабатывается в первую очередь слоем контроллера, где за такие запросы, как и за все запросы связанные с делегированием ресурсами клуба (книгами), отвечает BookController, метод, который отдаёт конкретную книгу findById(id), принимающий во входных параметрах id объекта с которым идёт работа.

Первым делом запрос переходит от слоя контроллера к слою сервиса, при вызове метода findById() класса BookService, данный метод возвращает объект Book:

```
Book book = service.findById(id);
```

Данный метод сразу вызывает внутренний метод сервиса findBookInRepository:

```
public Book findById(long Id){  
    return findBookInRepository(id);  
}
```

В свою очередь метод findBookInRepository(id) направляет запрос на уровень репозитория, ищет необходимую книгу по переданному id, в случае если таковой книги нет, бросается исключение ServiceBadRequestException, которое перехватывается ExceptionHendler и пользователь видит код ошибки 400 – Bad Request:

```
private Book findBookInRepository(long id) {  
    return bookRepository.findById(id).orElseThrow(() ->  
        new ServiceBadRequestException  
            (new InvalidDataMessage("This ID is does not  
            exist!")));  
}
```

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

В настоящее время, одним из способов проверки работоспособности приложения является тестирование. Тестирование направлено на исследование программного обеспечения, с целью получения информации о качестве продукта и доказательства соответствия программы заявленным требованиям, что позволяет выделить несоответствия разработанного функционала и устранить их.

Тестирование является одним из важнейших этапов разработки приложения, позволяющее при написании программного кода обнаружить сценарии, которые могут привести к ошибкам во время работы программы. Кроме того, грамотно написанный сценарий теста, позволяет обнаружить ошибки в случае внесения дополнительного функционала в программу даже в уже отлаженном рабочем коде. Однако не редка ситуация, когда добавление нового функционала ведет к поломкам старого, и в такой ситуации разработчик может об этом даже не узнать. Такая ситуация имеет свое название – регрессия.

Регрессионное тестирование – это виды тестирования программного обеспечения, позволяющие обнаружить ошибки в уже протестированном ранее функционале. Регрессионное тестирование выполняется не для того, чтобы убедиться в отсутствии ошибок в имеющемся функционале, а для исправления регрессионных ошибок (ошибки, появляющиеся не при написании программы, а при добавлении в исходный код нового функционала или исправление ошибок, что и стало причиной возникновения новых ошибок в уже протестированной программе).

Методы регрессионного тестирования включают в себя многократное выполнение предыдущих тестов и проверку того, что добавление нового функционала не вызвало создание новых регрессионных ошибок. Регрессионное тестирование включает в себя три этапа:

- верификация устранения нового дефекта;
- верификация того, что дефект, который был проверен и исправлен ранее, не воспроизводится вновь
- верификация того, что функциональность приложения не нарушилась после исправления дефектов и внесения нового кода.

Выделяют несколько видов регрессионных тестов:

- верификационные тесты (проводятся для проверки исправления обнаруженной ранее ошибки);
- тестирование верификации версии (проверка работоспособности основной функциональности программы).

При написании исходного кода веб-приложения тестирование проводилась в три этапа:

- тестирование отдельно каждого сервиса в процессе написания программного кода;
- тестирование взаимодействия нескольких сервисов между собой;

- полное тестирование программы после окончания процесса написания программного кода.

Данные этапы являются важными и связанными между собой, ни один из них невозможно исключить. Например, без модульного тестирования, при анализе работы программы в целом, будет происходить достаточное количество сбоев, выявить которые может оказаться достаточно сложным, в то время как при анализе работы одного модуля неисправность оказывается достаточно очевидной. И обратный случай, работоспособность каждого компонента в отдельности не гарантирует корректное поведение всей программы в целом.

При написании веб-приложения проводились юнит-тестирование и сквозное тестирование (end-to-end).

5.1 Юнит-тестирование

Для тестирования сервисов серверной части использовалось юнит-тестирование. Под юнит-тестированием подразумевается процесс, позволяющий проверить на корректность отдельные модули исходного кода. Основная цель юнит-тестирования заключается в том, чтобы писать тесты для каждой функции или метода и проверять как позитивные, так и негативные сценарии. Это позволяет быстро проверить, не привело ли добавление нового функционала к регрессии. Юнит-тестирование было использовано при написании каждого из сервисов.

Для написания юнит-тестов использовался фреймворк для модульного тестирования JUnit, который написан на языке Java. Он лежит в основе TDD (Test-Driven Development) и входит в семейство фреймворков для тестирования xUnit.

Основные преимущества фреймворка JUnit:

- открытый исходный код, который используется для написания и выполнения тестов;
- написание кода выполняется более быстро и качественно;
- прост в использовании;
- поддержка аннотаций для идентификации методов;
- тесты имеют визуальную индикацию;
- тесты могут быть организованы в связки тестов.

Для запуска юнит-тестов на сервисе необходимо выполнить задачу *test* сборщика проектов Gradle, пример показан на рисунке 5.2.

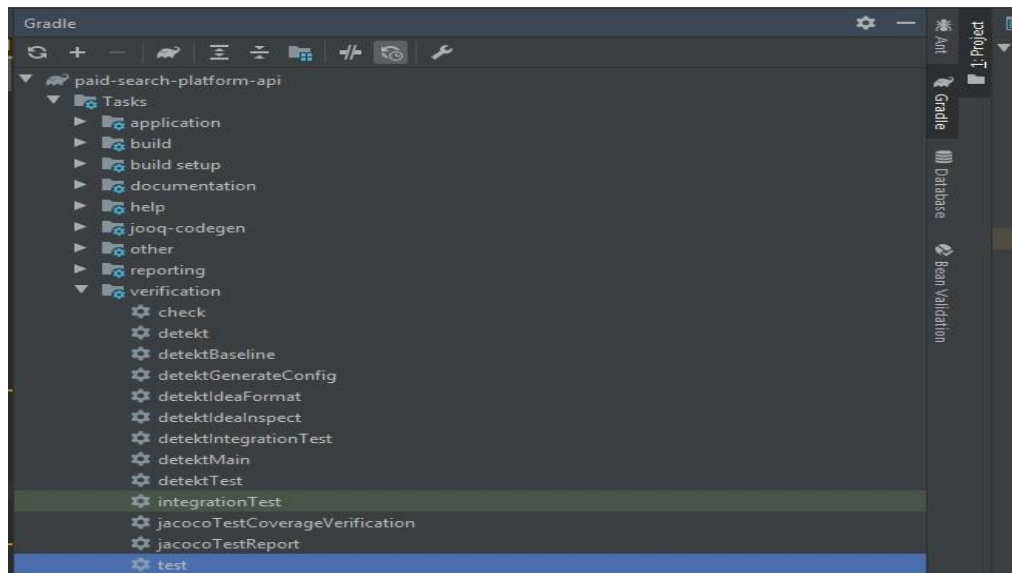


Рисунок 5.2 – Запуск юнит-тестов

В результате написания юнит-тестов было выполнено покрытие основных методов. Процентное соотношение покрытия юнит-тестами достигло отметки 30%, что является нормальным результатом. Результаты тестирования отображены на рисунке 5.3.

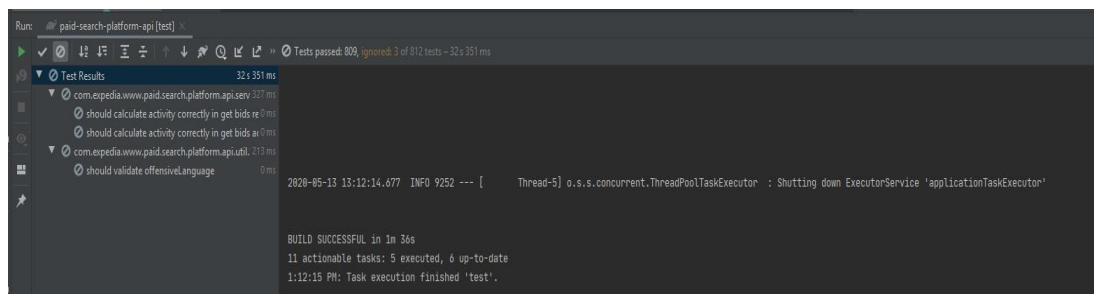


Рисунок 5.3 – Результаты юнит-тестирования

5.2 Сквозное тестирование (end-to-end)

Сквозные тесты выполняют тестирование системы в целом, эмулируя реальную пользовательскую среду. Данные тесты позволяют протестировать полный цикл работы какого-либо сценария, начиная от пользовательского интерфейса и до отправки запроса на сервер. Сквозные тесты проверяют взаимодействие сервисов между собой. В интернете это тесты, запущенные в браузере, имитирующие щелчки мышью и нажатия клавиш.

В разрабатываемом приложении сквозное тестирование осуществлялось двумя способами по методу тестирования, который называется черный ящик.

Метод тестирования черный ящик – это когда тестирующий имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь.

Как говорилось ранее тестирование осуществлялось несколькими способами.

Первый способ при помощи которого производилась проверка работоспособности приложения – это работа с приложением через пользовательский интерфейс. При таком способе тестирования тестирующий проверяет правильность работы приложения начиная с самого верхнего уровня, что позволяет осуществить полное сквозное тестирование, которое затрагивает все слои тестируемой системы.

Второй способ осуществлял тестирование при помощи программы Postman.

Данный инструмент позволяет эмулировать запросы с пользовательского интерфейса без использования самого интерфейса (рисунок 5.4).

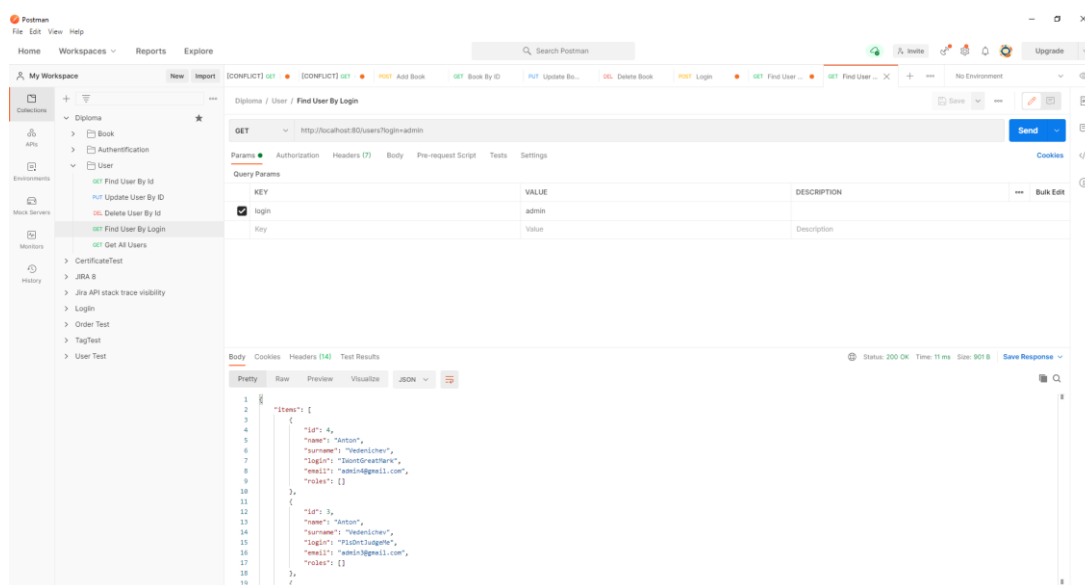


Рисунок 5.4 – Инструмент эмуляции запросов Postman

Основной плюс данного инструмента в том, что с его помощью можно проверить работоспособность каждого метода и класса слоя контроллера по отдельности.

Тестирующий, посылая запросы на сервер и сверяя получаемый ответ с ожидаемым, может увидеть ответ от сервера в удобном для него формате.

Внутри с запросов посылаемых на сервер можно указывать некоторые данные которые будут представлены в теле запроса, и хедер который например будет содержать токен необходимый для авторизации (рисунок 5.5). Такой богатый функционал позволяет проверить все самые разные варианты поведения пользователя начиная с того что он попытается выдать себя за другого пользователя с неверным токеном и заканчивая тем что он прошлет не валидные данные.

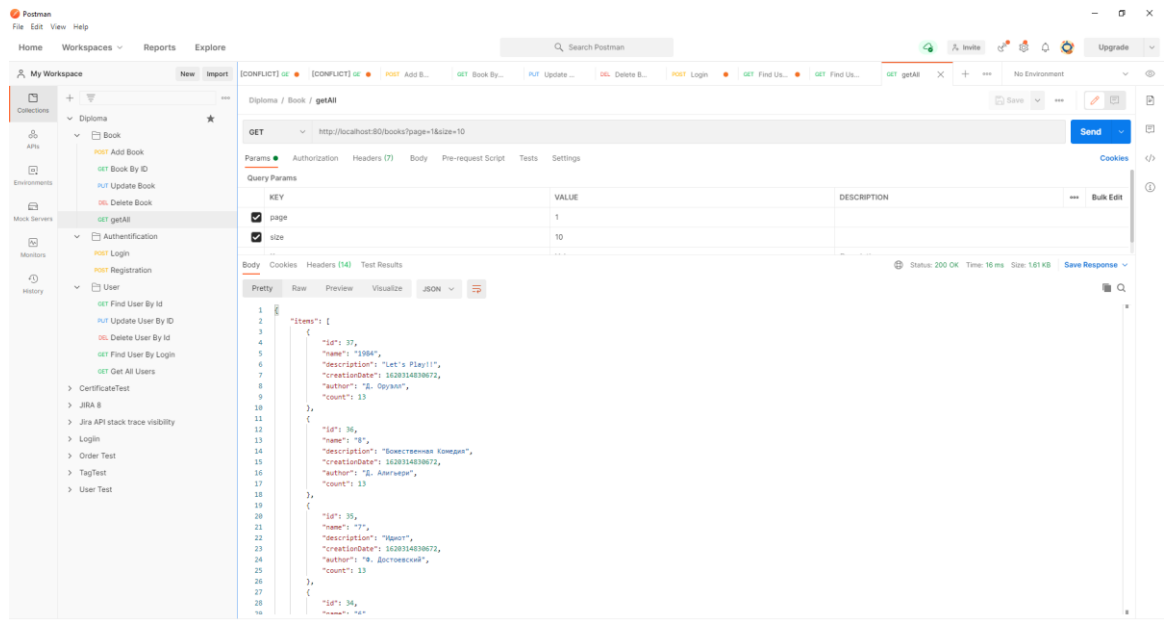


Рисунок 5.6 – Запрос на получение всех книг клуба

Данный запрос выводит книги, принадлежащие клубу «BookingClub», в своих параметрах он содержит информацию о том, сколько книг будет на каждой выведенной странице и какая страница будет выводиться. Как можно заметить, ответ приходит в формате JSON.

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

6.1 Настройка сервера и развертывание приложения

Для удобного запуска, быстрой и корректной работы веб-приложения одну из самых основных ролей роль играет сервер, на котором приложение располагается, так же его аппаратное и программное обеспечение. Настройка сервера и развертывание приложения может выполняться на любой операционной системе.

Аппаратное обеспечение должно соответствовать следующим требованиям:

- Оперативная память: не меньше 4 Гб;
- Операционная система: любая;
- Выход в Интернет: присутствует;
- Свободное место на винчестере: 1 Гб.

После установки операционной системы необходимо установить все необходимые зависимости. Для написания сервисов серверной части, использовалась Java 8, поэтому в первую очередь, необходимо установить JDK 8. Ее можно скачать на официальном сайте Oracle.

Далее необходимо установить среду разработки IntelliJ IDEA. Среда разработки IntelliJ IDEA можно скачать на официальном сайте компании JetBrains. Вам будет предоставлен выбор между двумя типами IDE, это Ultimate и Community издания. Издание Ultimate предоставляет больше возможностей для разработки приложений на Java или Kotlin, но стоит отметить, что данное издание является платным. В нашем случае подойдет любое из двух изданий, но так же предпочтительным и более удобным является Ultimate издание, которое можно получить в бесплатном виде используя почту учебного заведения. На рисунке 6.1 показан пример открытия среды разработки IntelliJ IDEA.



Рисунок 6.1 – Открытие IntelliJ IDEA

После скачивания среды разработки IntelliJ IDEA необходимо настроить ее и скачать все необходимые библиотеки и фреймворки. В первую очередь установим версию Java, которую необходимо использовать. Пример настройки показан на рисунке 6.2.

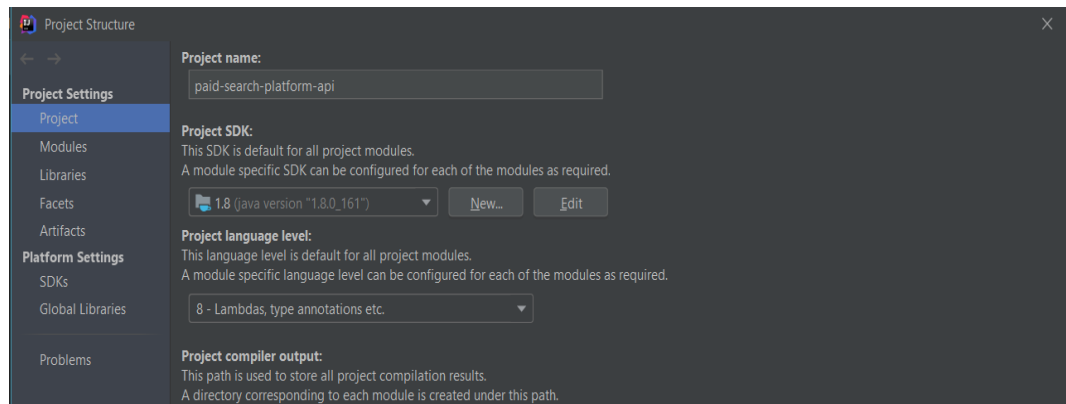


Рисунок 6.2 – Настройка версии SDK

Для работы с базой данных необходимо сконфигурировать подключение к ней через приложение. Spring Boot позволяет сделать это автоматически, не нужно ни настраивать JDBC драйвер, не выбирать поставщика, необходимо только добавить необходимые зависимости как показано на рисунке 6.3.

```
<dependencies>
  <dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>com.epam.esm</groupId>
    <artifactId>module_3_spring_advanced</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>compile</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
  </dependency>
</dependencies>
```

Рисунок 6.3 – Зависимости для работы с базой данных

Для написания сервисов, как обсуждалось в предыдущих разделах, используется фреймворк Spring Boot. В проекте использовался сборщик проектов Maven, который скачивает все необходимые зависимости во время сборки проекта, поэтому отсутствует необходимость скачивать фреймворк вручную, нам просто надо запустить задачу `bootRun`.

Для работы приложения у клиента необходимо наличие браузера. Приложение поддерживает следующие типы браузеров:

- Safari;
- Google Chrome;
- Mozilla Firefox;
- Opera;
- Internet Explorer.

Так как для написания пользовательского интерфейса использовался фреймворк React, необходимо установить среду разработки, которая поддерживает работу с данным фреймворком. Разработку клиентского приложения можно вести так же в IntelliJ IDEA, но для данного проекта была выбрана среда разработки WebStorm, так как она в большей мере подходит для разработки клиентских приложений.

Среду разработки WebStorm можно скачать на официальном сайте компании JetBrains. Как и в случае с IntelliJ IDEA будет предоставлен выбор между двумя типами IDE, это Ultimate и Community издания. Издание Ultimate предоставляет больше возможностей для разработки приложений, однако стоит отметить, что данное издание является платным. В нашем случае подойдет любое из двух изданий. На рисунке 6.4 показан пример официального сайта, где можно скачать среду разработки WebStorm.

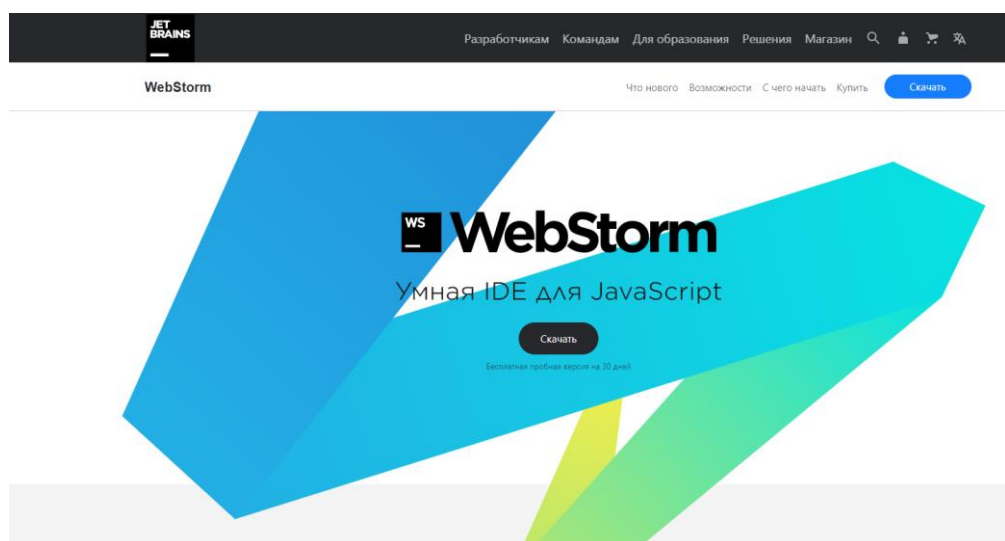
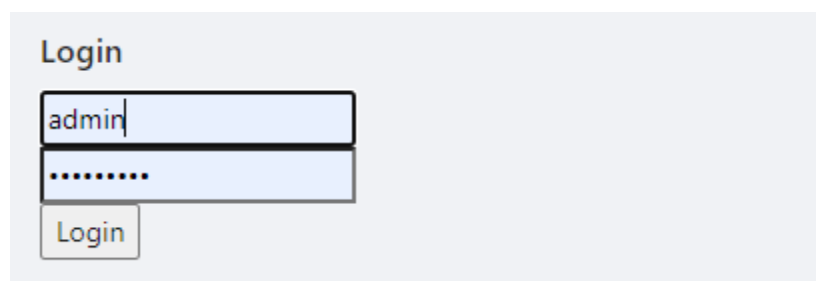


Рисунок 6.4 – Скачивание WebStorm IDE

6.2 Руководство по использованию ПО для администратора

6.2.1 Аутентификация и регистрация пользователя

Перейдя по ссылке приложения впервые пользователь будет перенаправлен на страницу входа в приложение (рисунок 6.5), на которой ему необходимо аутентифицироваться.



The screenshot shows a web form titled "Login". It contains two input fields: the first is a text box with the value "admin", and the second is a password box with masked characters represented by dots. Below these fields is a button labeled "Login".

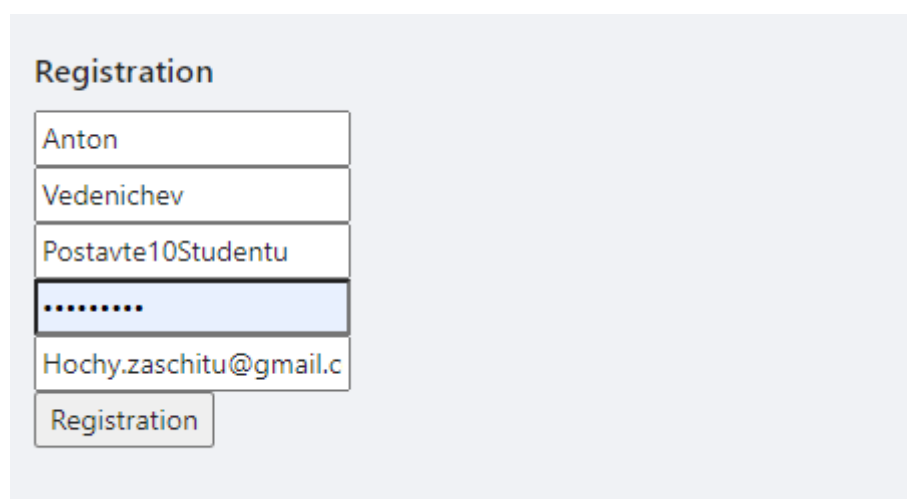
Рисунок 6.5 – Форма аутентификации

В данном окне пользователю необходимо ввести данные для аутентификации, а именно:

- имя пользователя или адрес электронной почты (Login);
- пароль (Password).

Данные поля являются обязательными для заполнения. После ввода пользователем всех данных и нажатия кнопки Login запрос отправится на сервер и в случае успешного прохождения аутентификации он будет перенаправлен на главную страницу приложения.

В случае, если пользователь впервые посетил приложение, ему необходимо будет зарегистрироваться (рисунок 6.6).



The screenshot shows a web form titled "Registration". It contains five input fields stacked vertically: the first three contain the text "Anton", "Vedenichev", and "Postavte10Studentu"; the fourth is a password field with masked dots; the fifth contains the email address "Hochy.zaschitu@gmail.c". Below the fields is a button labeled "Registration".

Рисунок 6.6 – Форма регистрации

В данном окне пользователю необходимо заполнить информацию о

себе, все поля являются обязательными, а именно:

- имя пользователя в системе (Login Name);
- пароль (Password);
- настоящее имя пользователя (name);
- настоящая фамилия пользователя (surname);
- почта пользователя (email);

После нажатия кнопки Registration выполнится проверка уникальности указанного имени и в случае, если в системе не существует пользователя с таким же именем, то будет создан новый аккаунт, а сам пользователь будет перенаправлен на главную страницу приложения, которая является каталогом книг. В случае если пользователь с таким именем уже существует, появится сообщение об ошибке, и просьба сменить имя пользователя так как введенное имя уже занято.

6.2.2 Каталог книг

После успешной аутентификации пользователь будет перенаправлен на главную страницу приложения на которой представлен список книг присутствующих в системе (рисунок 6.5).

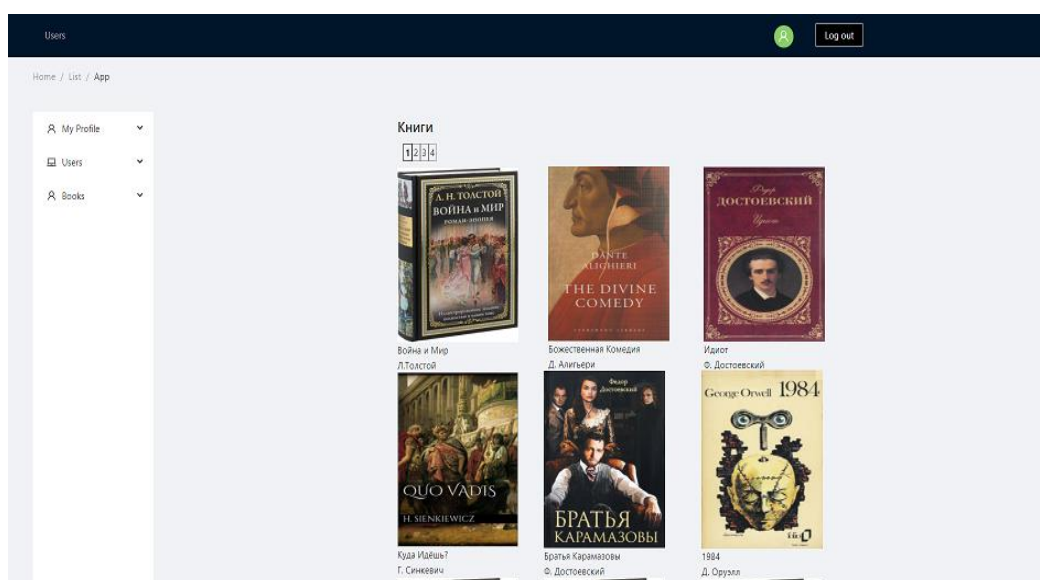


Рисунок 6.5 – Страница каталога книг

На эту страницу могут зайти только авторизованные пользователи. Страница не будет меняться в зависимости от роли пользователя, который зашел на нее. На рисунке 6.5 представлен вид главной страницы каталога книг.

Пользователь имеет возможность просматривать все книги клуба, доступны способы межстраничного перехода, страница состоит из карточек, в каждой карточке присутствует основная информация о книге:

- обложка книги;

- название книги;
- автор книги;

При просмотре книг, так же пользователь может нажать на любую понравившуюся ему книгу и перейти на страницу более подробного отображения информации о ней. Книги в основе своей подбираются по определенному (основному) списку, поэтому на главной странице в основе своей книги будут из данного списка (Норвежский клуб).

Выбрав понравившуюся книгу и перейдя на страницу подробного отображения информации о ней, нам откроется страница представленная на рисунке 6.6:

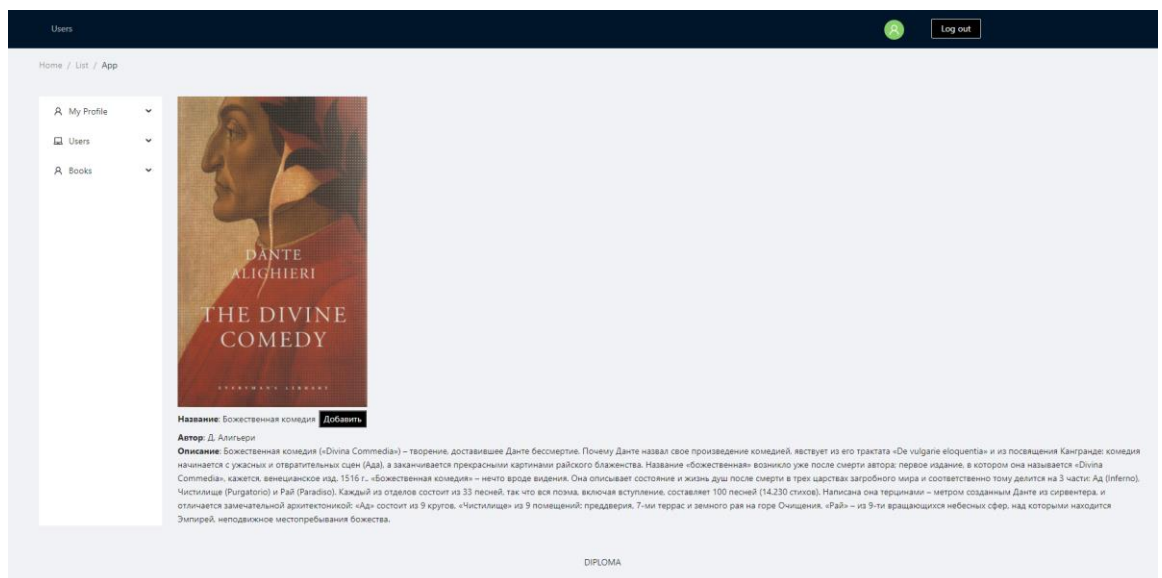


Рисунок 6.6 – Описание книги на главной странице

Как можно увидеть на рисунке 6.6 описание книги состоит из нескольких частей, а именно:

- обложки книги;
- названия книги;
- автора книги;
- описания книги.

Так же мы видим кнопку «Добавить», в случае если на неё нажать, будет произведена попытка сосздать бронь на данную книгу, в случае если в библиотеке находятся в данный момент книги в свободном доступе.

Users

Home / List / App

My Profile

Users

Books

Create Book

Илиада

Гомер

13

«Илиада» начинается с конфликта в стане осаждающих Трои ахейцев (называемых также данайцами). Царь Агамемнон похитил дочь жреца Аполлона. Хрис. Хрис приходит в греческой стан выкупить взятую в плен и доставшуюся в рабыни Агамемнону дочь Хрисеиду.

Гомер

Получив грубый отказ, он обращается с мольбой об отпущении к Аполлону, который насмелся на войско моровую язву. В собрании греков, созванном Ахиллом, Калхант объявляет, что единственное средство умиротворить бога состоит в выдаче Хрисеиды её отцу без выкупа. Агамемнон уступает всеобщему требованию, но, чтобы вознаградить себя за эту потерю, отнимает у Ахилла, которого считал инициатором всей интриги, его любимую рабыню Брисеиду.

В гневе Ахилл удаляется в палатку и просит свою мать Фетиду умолить Зевса, чтобы греки до тех пор терпели поражения от троянцев, пока Агамемнон не даст ему, Ахиллу, полного удовлетворения. Девятилетняя осада на грани срыва, но ситуацию исправляет Одиссей.

Create

DIPLOMA

Рисунок 6.7 – Страница создания книги.

На рисунке 6.7 представлена страница с формой для заполнения информации о новой книге.

Как можно увидеть на рисунке 6.7 данная форма состоит из нескольких частей, а именно:

- форма для ввода названия книги;
- форма для ввода автора книги;
- форма для ввода количества книг;
- форма для ввода описания книги.

Ниже этих форм можно заметить кнопку Create, после нажатия которой, данные о новой книге будут загружены на приложение.

6.2.3 Список пользователей

Сообщество «BookingClub» это открытое сообщество людей, которые любят читать книги и ищут себе единомышленников в этом увлечении, поэтому в данном веб-приложении реализованна возможность просматривать информацию о пользователях, которые так же состоят в данном сообществе, просматривать их профили и читать общую информацию о них.

Перейти на страницу со списком пользователей можно если обратится к навигационной панели слева (рисунок 6.8) там можно заметить 3 основных рабочих вкладки:

- вкладка профиля;
- вкладка пользователей;
- вкладка книг;

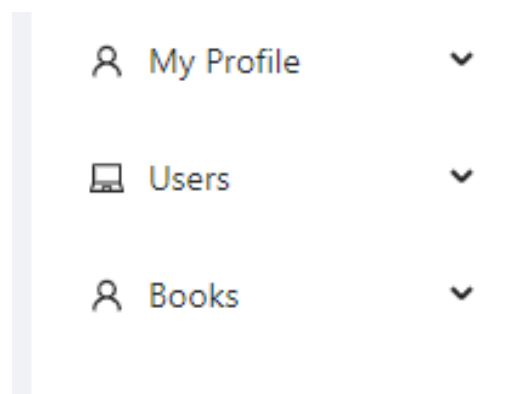


Рисунок 6.8 – Навигационная панель

Чтобы увидеть людей, которые так же состоят в данном клубе, зарегистрированному и аутентифицированному пользователю необходимо обратиться к данной навигационной панели, раскрыть вкладку пользователей, как на рисунке 6.9 и нажать на вкладку `Users`.

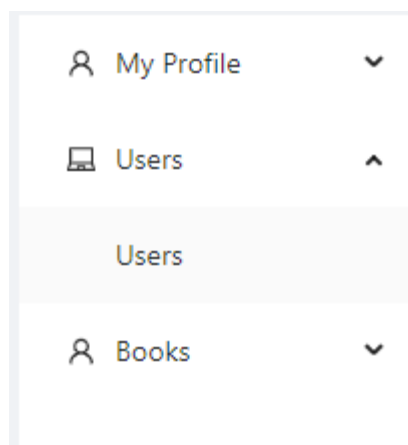


Рисунок 6.9 – Вкладка пользователей

После нажатия на данную вкладку, пользователь переходит на страницу, которая содержит информацию о всех пользователях (рисунок 6.10).

Так как пользователей может быть не ограниченное количество, для данного веб-приложения так же реализованна пагинация для переходов между старницами. Стандартное окно вывода показывает, при первом переходе, первых десятиерых пользователей.

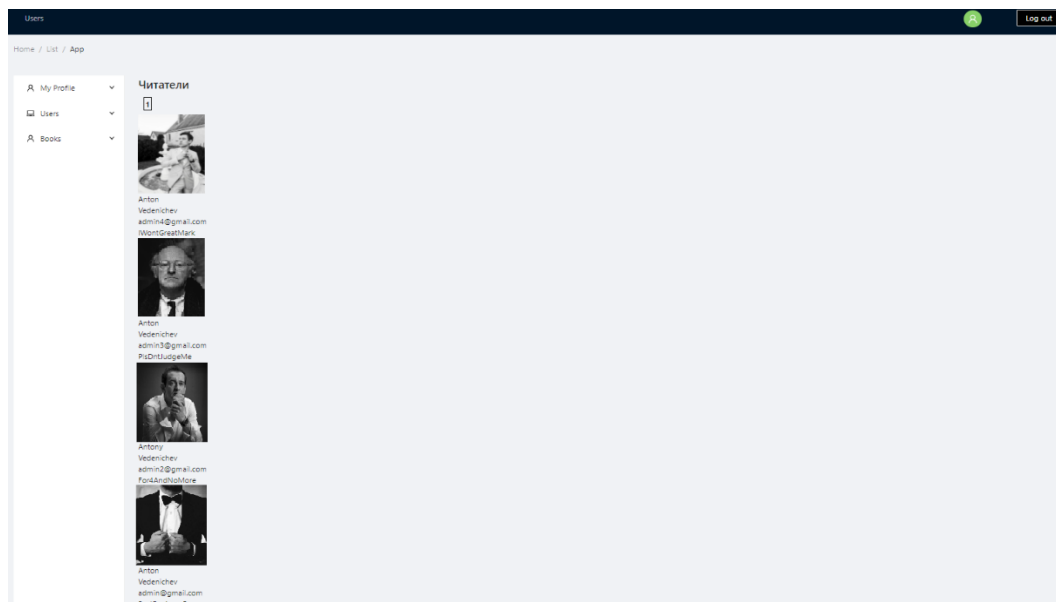


Рисунок 6.10 – Список пользователей книжного клуба

При нажатии на любого из пользователей в списке, можно попасть на его профильную страницу, так же если в навигационной панели открыть вкладку `My Profile` то можно перейти на свою профильную страницу, со всей введенной вами информацией при регистрации (рисунок 6.11). Как можно увидеть на рисунке 6.11 информация о пользователе, в профильной странице, состоит из нескольких частей, а именно:

- профильное фото;
- кнопка перехода на список «Книжного клуба»;
- кнопка перехода на личный список пользователя;
- имя пользователя;
- фамилия пользователя;
- логин пользователя;
- почта пользователя;

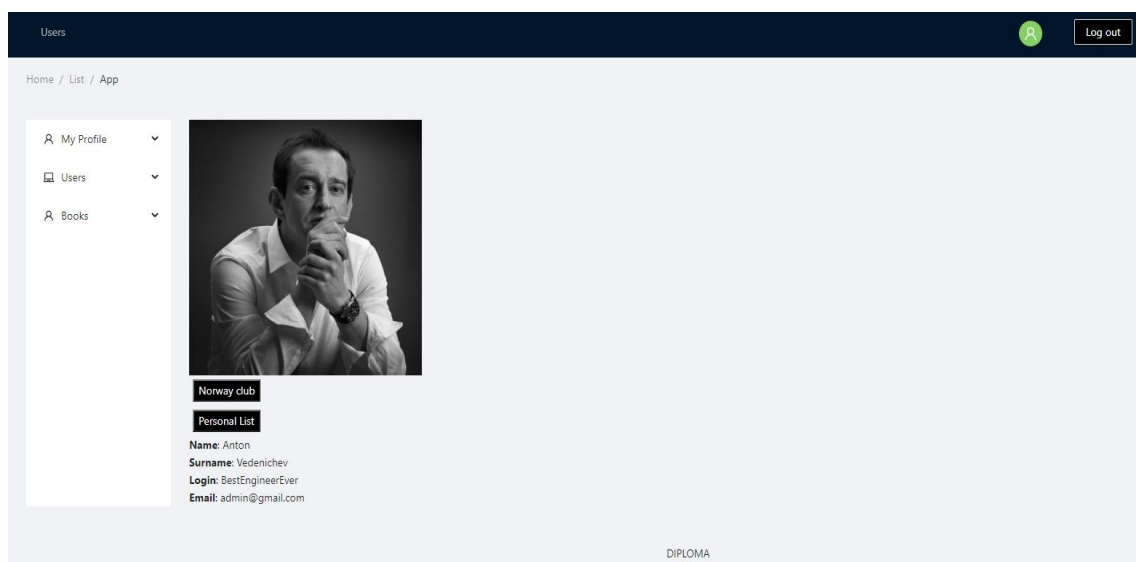


Рисунок 6.11 – Профильная страница

По окончании работы с данным веб-приложением, пользователь имеет возможность выйти из приложения нажатием кнопки «Log out» в правом верхнем углу (рисунок 6.12). После этого данный пользователь будет перенаправлен на страницу аутентификации.

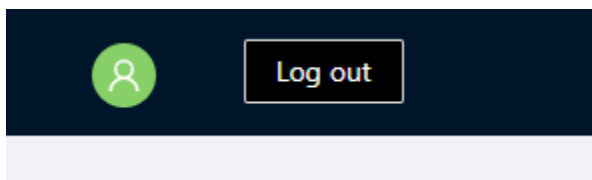


Рисунок 6.11 – Кнопка выхода из приложения

7 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЯ КЛУБА ЧИТАТЕЛЕЙ С КЛИЕНТ-СЕРВЕРНОЙ АРХИТЕКТУРОЙ

7.1 Характеристика разработанного программного средства

Целью данного дипломного проекта является разработка веб-приложения клуба читателей с клиент-серверной архитектурой. Программная часть будет включать в себя клиентскую часть, которая представляет собой удобный интерфейс, благодаря которому пользователь будет взаимодействовать с приложением через браузер, и серверную часть. В процессе разработки программной части будет получен программный продукт, который предоставляет пользователям возможность учета, оценки, комментирования и получения информации о всех ресурсах книжного клуба.

Данный продукт подойдет компаниям, и организациям, которые используют в качестве основного ресурса книги, для передачи их участникам клуба, учета ресурсов и возможности получения оценки о ресурсе.

Данный продукт разрабатывается по индивидуальному заказу, что подразумевает, что пользоваться данным продуктом будут сотрудники организации-заказчика.

7.2 Расчет инвестиций в разработку (модернизацию) программного средства

Для организации-заказчика инвестициями в разработку программного средства является цена программного средства без НДС, которая определяется на основе полных затрат на разработку программного средства организацией-разработчиком и включает в себя следующие статьи затрат: основная заработная плата разработчиков, дополнительная заработная плата разработчиков, отчисления на социальные нужды, прочие расходы, общая сумма затрат на разработку, плановая прибыль, включаемая в цену программного средства, отпускная цена программного средства. Далее рассмотрим каждую статью подробнее.

7.2.1 Затраты на основную заработную плату команды разработчиков

Расчет осуществляется исходя из состава и численности команды, размера месячной заработной платы каждого участника команды, а также трудоемкости работ, выполняемых при разработке программного средства отдельными исполнителями по следующей формуле:

$$З_0 = K_{\text{пр}} \cdot \sum_{i=1}^n З_{\text{чи}} \cdot t_i, \quad (7.1)$$

где n – количество исполнителей, занятых разработкой ПО;
 $K_{пр}$ – коэффициент премий (составляет 1,5);
 $z_{чi}$ – часовая заработная плата i -го исполнителя, руб.;
 t_i – трудоемкость работ i -го исполнителя, ч.

При определении того, какой будет состав команды были проанализированы требования, проведена консультация с разработчиками, работавшими в данной сфере, анализ предметной области. По итогу было принято решение, что на данном проекте будут работать ведущий разработчик, разработчик серверной части продукта, разработчик клиентской части продукта, тестировщик. Команда разработчиков оценила трудоёмкость проекта в 2 месяца, или 336 рабочих часа (по данным Министерства труда и социальной защиты населения, количество рабочих часов в месяце равно 168 часов). Размеры заработных плат сотрудников указаны по состоянию на 09.04.2021. Ведущий разработчик – 5000 рублей, разработчик серверной части – 2750 рублей, разработчик клиентской части – 2500 рублей, тестировщик – 2000 рублей. Размер премии равен 15% от размера основной заработной платы. Исходя из вышеперечисленных данных, рассчитаем затраты на основную заработную плату разработчиков. Все данные по затратам на основную заработную плату, представлены в таблице 7.1.

Таблица 7.1 – Расчет затрат на основную заработную плату команды

№	Участник команды	Вид выполняемой работы	Месячная заработная плата, руб.	Часовая заработная плата, руб.	Трудоёмкость работ, ч.	Зарплата по тарифу, руб.
1	Ведущий разработчик	Разработка задач и ПО	5000	29,76	336	9999,36
2	Разработчик серверной части	Разработка серверной части продукта	3500	16,36	336	5496,96
3	Разработчик клиентской части	Разработка клиентской части продукта	3750	14,88	336	4999,68
4	Тестировщик	Тестирование ПО	2000	11,90	336	3998,40
	Итого					24494,40
	Премия (15%)					3674,16
	Итого затраты на основную заработную плату					28168,56

7.2.2 Затраты на дополнительную заработную плату команды

Дополнительная заработная плата команды включает выплаты, предусмотренные законодательством о труде, и определяется по нормативу в процентах к основной заработной плате по следующей формуле:

$$З_д = \frac{З_о \cdot Н_д}{100\%} \quad (7.2)$$

где $З_о$ – затраты на основную заработную плату;

$Н_д$ – норматив дополнительной заработной платы, 15%.

После подстановки значений в формулу (7.2) затраты на дополнительную заработную плату команды составят:

$$З_д = \frac{28168,56 \cdot 15\%}{100\%} = 4225,28 \text{ руб.}$$

7.2.3 Отчисления на социальные нужды

Расчет размера отчислений в фонд социальной защиты населения и на обязательное страхование определяется в соответствии с действующими законодательными актами Республики Беларусь и вычисляются по формуле:

$$P_{\text{соц}} = \frac{(З_о + З_д) \cdot Н_{\text{соц}}}{100\%}, \quad (7.3)$$

где $Н_{\text{соц}}$ – норматив отчислений на социальные нужды, %.

Согласно законодательству Республики Беларусь, отчисления на социальные нужды составляют: 34% в фонд социальной защиты и 0,6% на обязательное страхование. Сумма отчислений на социальные нужды согласно формуле (7.3) составит:

$$P_{\text{соц}} = \frac{(28168,56 + 4225,28) \cdot 34,6\%}{100\%} = 11208,26 \text{ руб.}$$

Согласно формуле (7.3), размер отчислений в фонд социальной защиты и на обязательное страхование составляет 11208,26 рублей.

7.2.4 Расходы на лицензии программного обеспечения, необходимого для разработки

Расходы по данной статье будут выполнены исходя из стоимости одной лицензии программного обеспечения на месяц и будут определены по

следующей формуле:

$$P_m = C_m \cdot C_p \cdot N, \quad (7.4)$$

где P_m – суммарный расход на лицензии программного обеспечения;

C_m – цена одного месяца использования;

C_p – длительность проекта (месяцев);

N – количество лицензий.

Для работы команды, необходимо наличие лицензии для среды разработки. Цена лицензии на один месяц составляет 180 рублей по состоянию на 09.04.2021. Всего лицензий необходимо в количестве 4 штук. Срок разработки равен 2 месяцам. Рассчитаем расходы на машинное время по формуле 3.4:

$$P_m = 180 \cdot 2 \cdot 4 = 1440 \text{ руб.}$$

7.2.5 Прочие расходы

Расходы по данной статье включают такие затраты, как приобретение специализированной научно-технической литературы, поддержку со стороны производителей устройств, если это необходимо, определяются по следующей формуле:

$$Z_{пз} = \frac{Z_o \cdot N_{пз}}{100\%}, \quad (7.5)$$

где $N_{пз}$ – норматив прочих затрат в целом по организации.

Приняв значение $N_{пз}$ равным 7% и подставив в формулу (7.5) оставшиеся значения, произведём расчёт прочих затрат:

$$Z_{пз} = \frac{28168,56 \cdot 7\%}{100\%} = 1971,79 \text{ руб.}$$

7.2.6 Общая сумма инвестиций (затрат) на разработку

Полная сумма затрат на разработку программного обеспечения находится путем суммирования всех рассчитанных статей затрат. Расчет общей суммы затрат представлен в таблице 7.2.

Таблица 7.2 – Затраты на разработку программного обеспечения

Статья затрат	Сумма, руб.
Основная заработная плата команды разработчиков	28168,56

Дополнительная заработная плата команды разработчиков	4225,28
Отчисления на социальные нужды	11208,26
Машинное время	1440,00
Прочие затраты	1971,79
Общая сумма	47013,89

Полная себестоимость разрабатываемого программного средства составит 47013,89 рублей.

7.2.7 Плановая прибыль, включаемая в цену программного средства

Расчеты по данной статье исходят из нужд на развитие производства и затрат на заработную плату команды.

$$П_{пс} = \frac{З_p \cdot P_{пс}}{100} \quad (7.6)$$

где $P_{пс}$ – рентабельность затрат на разработку программного средства.

Приняв значение $P_{пс}$ равным 49,7% и подставив в формулу (7.6) оставшиеся значения, произведём расчёт плановой прибыли:

$$П_{пс} = \frac{47013,89 \cdot 49,7\%}{100\%} = 23365,90 \text{ руб.}$$

7.2.8 Отпускная цена программного средства

Расчеты по данной статье исходят из затрат на заработную плату и плановой прибыли.

$$Ц_{пс} = З_{пс} + П_{пс} \quad (7.7)$$

Произведя подстановку выше полученных значений в формулу (7.7), получим отпускную цену программного средства.

$$Ц_{пс} = 47013,89 + 23365,90 = 70379,79 \text{ руб.}$$

7.3 Расчет результата от разработки и использования программного средства

7.3.1 Экономический эффект от продажи ПО для организации-разработчика

Экономический эффект для организации-разработчика программного обеспечения заключается в получении прибыли от его реализации, а также его поддержки в будущем, по согласованию с заказчиком. Разрабатываемое программное обеспечение будет продано организации-заказчику в одном экземпляре и будет поддерживаться в будущем по мере необходимости.

В результате расчет инвестиций в разработку программного средства было выявлено, что стоимость данного продукта составляет 70379,79 рублей.

Прибыль от продажи программного продукта можно определить по следующей формуле:

$$П = Ц - НДС - З_p, \quad (7.8)$$

где Ц – цена реализации программного обеспечения заказчику, руб.

НДС – сумма налога на добавленную стоимость, руб.

Z_p – сумма расходов на разработку и реализацию, руб.

Согласно законодательству Республики Беларусь, резиденты Парка высоких технологий освобождены от налога на добавленную стоимость, а так как компания ЗАО «Итранзишен» является резидентом Парка высоких технологий, налог на добавленную стоимость будем брать равным нулю.

Произведя подстановку выше полученных значений в формулу (7.8), получим прибыль от продажи программного продукта:

$$П = 70379,79 - 47013,89 = 23365,90 \text{ руб.}$$

Чистая прибыль от продажи программы рассчитывается по формуле:

$$П_ч = П - \frac{П \cdot Н_п}{100\%}, \quad (7.9)$$

где $H_п$ – ставка налога на прибыль.

Для ЗАО «Итранзишен», как для резидента Парка высоких технологий, ставка налога на прибыль будет равна нулю, следовательно, чистая прибыль от продажи программного продукта составляет 23365,90 рублей.

7.4 Расчёт показателей экономической эффективности разработки и использования программного средства

По причине того, что разработка программного обеспечения занимает менее года, рентабельность инвестиций ($P_{и}$) будет рассчитывается по следующей формуле:

$$P_{и} = \frac{\Pi_{ч}}{З_p} \cdot 100\%, \quad (7.10)$$

В соответствии с формулой (7.10), рентабельность инвестиций составит:

$$P_{и} = \frac{23365,90}{47013,89} \cdot 100\% = 49,7\%$$

Рассчитанный показатель показывает, что разработка имеет положительный экономический эффект.

7.5 Выводы по технико-экономическому обоснованию

В ходе технико-экономического обоснования разрабатываемого программного продукта был произведён расчёт затрат на разработку программного обеспечения, выполнена оценка экономического эффекта от его продажи, произведён расчёт эффективности инвестиций в разработку, а также показана его экономическая целесообразность. Чистая прибыль от реализации программного продукта составит 23365,90 рублей. Исходя из произведённых расчётов и выполненных оценок, можно сделать вывод, что данный проект является экономически выгодным.

ЗАКЛЮЧЕНИЕ

В ходе разработки данного дипломного проекта была изучена теория и предметная область, изучен процесс разработки веб-приложения, его компонентов и подходов к созданию. Были на практике рассмотрены и изучены основные плюсы микросервисной архитектуры перед монолитной.

Функциональность дипломного проекта была реализована на языке Java и Kotlin, в качестве фреймворка использовался Spring Boot.

Были решены все задачи, поставленные перед началом разработки веб-приложения, а именно:

- обеспечение удобного пользовательского интерфейса, не требующего затрат времени для изучения;
- обеспечение локализации и интернационализации;
- обеспечение полного функционала, для осуществления поставленных задач системе по предоставлению рекламы и управления отелями.

Проведя расчёт и анализ экономических показателей эффективности, можно сделать вывод о том, что разработка и реализация данного программного продукта, является целесообразными. Таким образом, разработку данной системы можно считать успешной и экономически выгодной.

Проект разработан в полном объеме в соответствии с поставленной целью. В качестве усовершенствования разработанного веб-приложения можно рассматривать создание мобильного приложения, которое будет предоставлять тот же функционал, что и веб-приложение.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Ruslit [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://ruslit.net/index.php>. – Дата доступа: 09.04.2021.
- [2] Alib [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.alib.ru>. – Дата доступа: 09.04.2021.
- [3] Labitint [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.labirint.ru>. – Дата доступа: 09.04.2021.
- [4] Oz [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://zoon.by/vitebsk>. – Дата доступа: 09.04.2021.
- [5] Typescript [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.typescriptlang.org/>. – Дата доступа: 09.04.2021.
- [6] AngularJs [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://angular.io/docs>. – Дата доступа: 09.04.2021.
- [7] Spring [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://spring.io/>. – Дата доступа: 09.04.2021.
- [8] JWT [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://jwt.io/>. – Дата доступа: 09.04.2021.
- [9] JSON [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://www.json.org/json-en.html>. – Дата доступа: 27.03.2020.
- [10] Регрессионное тестирование [Электронный ресурс]. – Электронные данные. – Режим доступа: <https://quality-lab.ru/blog/how-to-reduce-a-regression-testing-time-from-3-to-1-day/>. – Дата доступа: 11.04.2020.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

Файл BookController.java

```
@RestController
@RequestMapping("/books")
public class BookController implements
BookingClubController<BookDto> {

    private final BookService service;
    private final BookConverter converter;

    @Autowired
    public BookController(BookService service, BookConverter
converter) {
        this.service = service;
        this.converter = converter;
    }

    @CrossOrigin(origins = "http://localhost:3000")
    @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<BookList> findAll(
        @RequestParam(value =
ControllerParamNames.PAGE_PARAM_NAME) int page,
        @RequestParam(value =
ControllerParamNames.SIZE_PARAM_NAME) int size) {

        List<BookDto> book =
converter.convert(service.findAllBooksWithPagination(page,
size));
        int count = service.getAllBooksCount();

        return new ResponseEntity<>( new BookList(book, count),
HttpStatus.OK);
    }

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
produces = MediaType.APPLICATION_JSON_VALUE)//+
    public ResponseEntity<?> create(@RequestBody @Valid BookDto
book) {
        BookDto newBook = new
BookDto(service.create(converter.convert(book)));

        return
ResponseEntity.created(linkTo(methodOn(BookController.class).fin
dById(newBook.getId())) .toUri())
            .body(newBook.getModel());
    }
}
```



```

        @GetMapping(path =("/{id}", produces =
MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<EntityModel<BookDto>>
findById(@PathVariable long id) {//+
            Book book = service.findById(id);
            BookDto bookDTO = new BookDto(book);

            return new ResponseEntity<>(bookDTO.getModel(),
HttpStatus.OK);
        }

        @DeleteMapping(path =("/{id}")
        public ResponseEntity<Void> delete(@PathVariable Long id)
{//+
            service.delete(id);

            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }

        @PutMapping(path =("/{id}", consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)//+
        public ResponseEntity<EntityModel<BookDto>>
update(@RequestBody @Valid BookDto book, @PathVariable int id) {

            return new ResponseEntity<>(new
BookDto(service.update(converter.convert(book), id)).getModel(),
HttpStatus.OK);
        }

        @GetMapping(
            consumes = MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<BookList> find(@RequestParam
Map<String, String> params,

        @RequestBody(required = false) List<reviewDTO> reviews) {
            int page = ControllerUtils

            .getValidPaginationParam(params.get(ControllerParamNames.PAGE_PA
RAM_NAME),

                ControllerParamNames.PAGE_PARAM_NAME);
            int size = ControllerUtils

            .getValidPaginationParam(params.get(ControllerParamNames.SIZE_PA
RAM_NAME),

                ControllerParamNames.SIZE_PARAM_NAME);

            List<reviewPOJO> reviewsPojo = null;
            if (reviews != null) {

```

```

        reviewsPojo = reviews
            .stream()
            .map(reviewConverter::convert)
            .collect(Collectors.toList());
    }

    List<BookPOJO> Books = service
        .findAll(params, reviewsPojo, page, size);
    int BooksCount = service.getBooksCount(params,
reviewsPojo);

    BookList BookList = new BookList
        .BookListBuilder(Books, converter)
        .page(page)
        .size(size)
        .parameters(params)
        .resultCount(BooksCount)
        .build();

    return new ResponseEntity<>(BookList, HttpStatus.OK);
}

@DeleteMapping(path =("/{id}")
public ResponseEntity<Void> deleteBook(@PathVariable Integer
id) {
    service.delete(id);

    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}

@PutMapping(path =("/{id}",
    consumes = MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<EntityModel<BookDTO>> updateBook(
    @RequestBody @Valid BookDTO Book,
    @PathVariable int id) {
    service.update(id, converter.convert(Book));
    BookDTO updatedBook = new BookDTO(service.find(id));

    return new ResponseEntity<>(updatedBook.getModel(), HttpStatus.OK);
}

@PatchMapping(path =("/{id}",
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<EntityModel<BookDTO>> updateBookPrice(
    @RequestBody BookDTO BookDTO,
    @PathVariable long id) {
    service.updatePath(id, converter.convert(BookDTO));
    BookDTO updatedBook = new BookDTO(service.find(id));

    return new ResponseEntity<>(updatedBook.getModel(), HttpStatus.OK);
}

@PostMapping(path = "{id}/reviews", consumes = MediaType.APPLICATION_JSON_VALUE,
    produces = MediaType.APPLICATION_JSON_VALUE)
public ResponseEntity<EntityModel<BookDTO>> addreviewToBook(
    @PathVariable Integer id, @RequestBody @Valid reviewDTO review) {
    service.addreview(id, reviewConverter.convert(review));
    BookDTO editBook = new BookDTO(service.find(id));

    return new ResponseEntity<>(editBook.getModel(), HttpStatus.OK);
}

```

```

    }

    @PatchMapping(path = "{id}/reviews", consumes = MediaType.APPLICATION_JSON_VALUE,
        produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<EntityModel<BookDTO>> addreviewToBook(
        @PathVariable Integer id,
        @Valid @RequestBody AddedreviewDTO reviewDTOId) {
        service.addreview(id, reviewDTOId.getAddedreview());
        BookDTO editBook = new BookDTO(service.find(id));

        return new ResponseEntity<>(editBook.getModel(), HttpStatus.OK);
    }
}

```

Файл UserController.java

```

@RestController()
@RequestMapping("/users")
public class UserController implements
    BookingClubController<UserDto> {

    private final UserService service;
    private final AuthenticationManager authenticationManager;
    private final JwtTokenProvider jwtTokenProvider;
    private final DtoConverter<UserDto, User> converter;

    public UserController(UserService service,
        AuthenticationManager authenticationManager,
        JwtTokenProvider jwtTokenProvider,
        DtoConverter<UserDto, User> converter)
    {
        this.service = service;
        this.authenticationManager = authenticationManager;
        this.jwtTokenProvider = jwtTokenProvider;
        this.converter = converter;
    }

    @CrossOrigin(origins = "http://localhost:3000")
    @GetMapping( produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<UserList> findAll(
        @RequestParam(value =
            ControllerParamNames.PAGE_PARAM_NAME, defaultValue =
            ControllerParamNames.DEFAULT_PAGE_STRING) int page,
        @RequestParam(value =
            ControllerParamNames.SIZE_PARAM_NAME, defaultValue =
            ControllerParamNames.DEFAULT_SIZE_STRING) int size
        ) {

        List<UserDto> searchedUsers =
            converter.convert(service.findAllUsers(page, size));
        int count = service.findAllUsers().size();
        return new ResponseEntity<>(new UserList(searchedUsers,
            count), HttpStatus.OK);
    }

    @GetMapping(path = "{id}", produces =

```

```

MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<EntityModel<UserDto>>
findById(@PathVariable long id) {

        //ControllerUtils.checkUserRulesById(request, id);
        UserDto searchedUser = new
UserDto(service.findById(id));

        return new ResponseEntity<>(searchedUser.getModel(),
HttpStatus.OK);
    }

    @CrossOrigin(origins = "http://localhost:3000")
    @GetMapping(path = "/find", produces =
MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<EntityModel<UserDto>>
findByLogin(@RequestParam(value="login") String login) {

        //ControllerUtils.checkUserRulesById(request, id);
        UserDto searchedUser = new
UserDto(service.findByLogin(login));

        return new ResponseEntity<>(searchedUser.getModel(),
HttpStatus.OK);
    }

    @DeleteMapping(path =("/{id}")
    public ResponseEntity<Void> delete(@PathVariable Long id) {
        service.delete(id);

        return new ResponseEntity<>(HttpStatus.NO_CONTENT);
    }

    @PutMapping(path =("/{id}", consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE) //+
    public ResponseEntity<EntityModel<UserDto>>
update(@RequestBody @Valid UserDto userDTO, @PathVariable int
id) {

        return new ResponseEntity<>(new
UserDto(service.update(id,
converter.convert(userDTO))).getModel(), HttpStatus.OK);
    }

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<EntityModel<UserDto>> create(@Valid
@RequestBody UserDto userDTO) {

        User user =
service.create(converter.convert(userDTO));

```

```

        UserDto createdUser = new UserDto(user);
        return ResponseEntity

.created(linkTo(methodOn(UserController.class).findById(createdU
ser.getId()))).toUri())
        .body(createdUser.getModel());
    }

    @PatchMapping(path = "{id}/orders")
    public ResponseEntity<EntityModel<bookOrderDTO>>
addOrder(

@PathVariable long id, HttpServletRequest request,
                                                    @RequestBody
bookOrderDTO orderDTO) {
        ControllerUtils.checkUserRulesById(request, id);
        UserDTO createdUserDTO = new
UserDTO(service.find(id));
        UserPOJO createdUserPOJO =
converter.convert(createdUserDTO);

        bookOrderDTO resultOrder = new bookOrderDTO(

orderService.create(orderConverter.convert(orderDTO),
        createdUserPOJO));

        return
ResponseEntity.created(linkTo(methodOn(OrderController.class)
        .findOrderById(resultOrder.getId()))
        .toUri()).body(resultOrder.getModel());
    }

    @GetMapping(produces = MediaType.APPLICATION_JSON_VALUE)
    public ResponseEntity<UserList> findAll(
        @RequestParam(value =
ControllerParamNames.PAGE_PARAM_NAME,
        defaultValue =
ControllerParamNames.DEFAULT_PAGE_STRING) int page,
        @RequestParam(value =
ControllerParamNames.SIZE_PARAM_NAME,
        defaultValue =
ControllerParamNames.DEFAULT_SIZE_STRING) int size) {
        List<UserPOJO> users = service.findAll(page, size);
        int usersCount = service.getUsersCount();

        return new ResponseEntity<>(
            new UserListBuilder(users, converter)
                .resultCount(usersCount)
                .page(page).size(size)
                .build(), HttpStatus.OK);
    }

```

```

        @GetMapping(path =("/{id}/orders", produces =
MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<OrderList> findOrders(@PathVariable
long id,
            @RequestParam(value =
ControllerParamNames.PAGE_PARAM_NAME,
                defaultValue =
ControllerParamNames.DEFAULT_PAGE_STRING) int page,
            @RequestParam(value =
ControllerParamNames.SIZE_PARAM_NAME,
                defaultValue =
ControllerParamNames.DEFAULT_SIZE_STRING) int size,
HttpServletRequest request) {
            ControllerUtils.checkUserRulesById(request, id);
            List<bookOrderPOJO> ordersByUser =
orderService.findAllByOwner(id, page, size);
            int ordersCount = orderService.ordersCountByOwner(id);

            return new ResponseEntity<>(
                new OrderListBuilder(ordersByUser, orderConverter)
                    .resultCount(ordersCount)
                    .page(page)
                    .size(size)
                    .build(), HttpStatus.OK);
        }

        @DeleteMapping(path =("/{userId}/orders/{id}")
        public ResponseEntity<Void> deleteOrder(@PathVariable Long
id, @PathVariable Long userId,
                                                    HttpServletRequest
request) {

ControllerUtils.checkIsCurrentUserHaveRulesForEditThisOrder(user
Id, id);
            ControllerUtils.checkUserRulesById(request, userId);

            orderService.delete(id);

            return new ResponseEntity<>(HttpStatus.NO_CONTENT);
        }

        @PatchMapping(path = "{userId}/orders/{id}", consumes =
MediaType.APPLICATION_JSON_VALUE,
            produces = MediaType.APPLICATION_JSON_VALUE)
        public ResponseEntity<EntityModel<bookOrderDTO>> addbooks(
@PathVariable long userId,
@PathVariable long id,

```

```

@RequestBody AddedbooksListDTO

booksId,

HttpServletRequest request) {

ControllerUtils.checkIsCurrentUserHaveRulesForEditThisOrder(user
Id, id);
        ControllerUtils.checkUserRulesById(request, userId);

        bookOrderDTO bookOrderDTO =
            new bookOrderDTO(orderService.addbooks(id, booksId

.getAddedbooks())));
        return new ResponseEntity<>(bookOrderDTO.getModel(),
HttpStatus.OK);
    }

}

```

Файл AuthenticationController.java

```

@RestController()
@RequestMapping()
public class AuthenticationController {

    private final UserService service;
    private final AuthenticationManager authenticationManager;
    private final JwtTokenProvider jwtTokenProvider;
    private final DtoConverter<RegistrationUserDto, User>
registrationConverter;

    public AuthenticationController(UserService service,
AuthenticationManager authenticationManager,
                                JwtTokenProvider
jwtTokenProvider,

DtoConverter<RegistrationUserDto, User> registrationConverter) {
        this.service = service;
        this.authenticationManager = authenticationManager;
        this.jwtTokenProvider = jwtTokenProvider;
        this.registrationConverter = registrationConverter;
    }

    @CrossOrigin(origins = "http://localhost:3000")
    @PostMapping(path = "/login")
    public ResponseEntity<Map<Object, Object>>
login(@RequestBody AuthenticationRequestDto requestDto,

HttpServletRequest request, HttpServletResponse response) {

```

```

        String invalid = "Invalid username or password";
        String parameterUsername = "username";
        String parameterIsLogged = "isLogged";
        String parameterToken = "token";

        try {
            String username = requestDto.getLogin();
            authenticationManager
                .authenticate(new
UsernamePasswordAuthenticationToken(username,
requestDto.getPassword()));
            UserDto user = new
UserDto(service.findByLogin(username));

            String token =
jwtTokenProvider.createToken(username, user.getRoles());

            Map<Object, Object> responseMap = new HashMap<>();
            responseMap.put(parameterUsername, username);
            responseMap.put(parameterToken, token);
            responseMap.put(parameterIsLogged, true);
            /*String path =
linkTo(methodOn(UserController.class)
            .findUserById(user.getId(), request)).toString();
            response.setHeader("Location", path);*/

            return ResponseEntity.ok(responseMap);
        } catch (AuthenticationException e) {
            Map<Object, Object> responseMap = new HashMap<>();
            responseMap.put(parameterUsername, "");
            responseMap.put(parameterToken, "");
            responseMap.put(parameterIsLogged, false);
            return ResponseEntity.ok(responseMap);
        }
    }

    @CrossOrigin(origins = "http://localhost:3000")
    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,
produces = MediaType.APPLICATION_JSON_VALUE, path =
"/registration")
    public ResponseEntity<EntityModel<UserDto>>
registration(@Valid @RequestBody RegistrationUserDto userDTO,

HttpServletRequest request) {
        String invalid = "Invalid data";

        try {
            User user =
service.create(registrationConverter.convert(userDTO));
            UserDto createdUser = new UserDto(user);
            return ResponseEntity

```



```

.created(linkTo(methodOn(UserController.class).findById(createdUser.getId()))).toUri())
        .body(createdUser.getModel());
    } catch (AuthenticationException e) {
        throw new BadCredentialsException(invalid);
    }
}
}

```

Файл ControllerBadRequestException.java

```

@Component
public class ControllerBadRequestException extends
    RuntimeException {

    private final static String MESSAGE = "Controller
exception";
    private List<InvalidControllerOutputMessage> messages;
    private InvalidControllerOutputMessage message;

    public
    ControllerBadRequestException(List<InvalidControllerOutputMessag
e> messages) {
        this.messages = messages;
    }

    public ControllerBadRequestException() {
    }

    public
    ControllerBadRequestException(InvalidControllerOutputMessage
message) {
        this.message = message;
        messages = new ArrayList<>();
        messages.add(message);
    }

    public List<InvalidControllerOutputMessage> getMessages() {
        return messages;
    }

    public InvalidControllerOutputMessage getErrorMessage() {
        return message;
    }

    @Override
    public String getMessage() {
        return MESSAGE;
    }

    public String getStatus(){

```

```

        return HttpStatus.BAD_REQUEST.toString();
    }
}

```

Файл JwtTokenFilter.java

```

public class JwtTokenFilter extends GenericFilterBean {

    private final JwtTokenProvider jwtTokenProvider;

    public JwtTokenFilter(JwtTokenProvider jwtTokenProvider) {
        this.jwtTokenProvider = jwtTokenProvider;
    }

    @Override
    public void doFilter(ServletRequest req, ServletResponse
res, FilterChain filterChain)
        throws IOException, ServletException{
        String token =
jwtTokenProvider.resolveToken((HttpServletRequest) req);
        if (token != null &&
jwtTokenProvider.validateToken(token)) {
            Authentication auth =
jwtTokenProvider.getAuthentication(token);
            if (auth != null) {

SecurityContextHolder.getContext().setAuthentication(auth);
            }
            filterChain.doFilter(req, res);
        }
    }
}

```

Файл JwtTokenProvider.java

```

@Component
public class JwtTokenProvider {

    private UserDetailsService userDetailsService;

    @Value("${jwt.token.secret}")
    private String secret;

    @Value("${jwt.token.expired}")
    private long validityInMilliseconds;

    public JwtTokenProvider(
        @Qualifier("jwtUserDetailsService") UserDetailsService
userDetailsService) {
        this.userDetailsService = userDetailsService;
    }
}

```

```

    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @PostConstruct
    protected void init() {
        secret =
Base64.getEncoder().encodeToString(secret.getBytes());
    }

    public String createToken(String username, List<Role> roles)
{
        String rolesNameParameter = "roles";
        Claims claims = Jwts.claims().setSubject(username);
        claims.put(rolesNameParameter, getRoleNames(roles));

        Date now = new Date();
        Date validity = new Date(now.getTime() +
validityInMilliseconds);

        return Jwts.builder()
            .setClaims(claims)
            .setIssuedAt(now)
            .setExpiration(validity)
            .signWith(SignatureAlgorithm.HS256, secret)
            .compact();
    }

    public Authentication getAuthentication(String token) {
        UserDetails userDetails =
this.userDetailsService.loadUserByUsername(getUsername(token));
        return new
UsernamePasswordAuthenticationToken(userDetails, "",
userDetails.getAuthorities());
    }

    public String getUsername(String token) {
        return
Jwts.parser().setSigningKey(secret).parseClaimsJws(token).getBod
y().getSubject();
    }

    public String resolveToken(HttpServletRequest req) {
        String headerName = "Authorization";
        String prefixHeader = "Bearer_";

        String token = req.getHeader(headerName);
        if (token != null && token.startsWith(prefixHeader)) {

```

```

        return token.substring(7);
    }
    return null;
}

public boolean validateToken(String token){
    String invalidToken = "JWT token is expired or invalid";
    try {
        Jws<Claims> claims =
Jwts.parser().setSigningKey(secret).parseClaimsJws(token);

        return !claims.getBody().getExpiration().before(new
Date());
    } catch (JwtException | IllegalArgumentException e) {
        throw new JwtAuthenticationException(invalidToken);
    }
}

private List<String> getRoleNames(List<Role> userRoles) {
    List<String> result = new ArrayList<>();

    userRoles.forEach(role -> result.add(role.getName()));
    return result;
}
}

```

Файл JwtUser.java

```

@Getter
@Setter
public class JwtUser implements UserDetails {

    private Long id;
    private String login;
    private String name;
    private String surname;
    private String password;
    private String email;
    private Collection<? extends GrantedAuthority> authorities;

    public JwtUser(Long id, String login, String name, String
surname, String email, String password,
Collection<? extends GrantedAuthority>
authorities) {
        this.id = id;
        this.login = login;
        this.name = name;
        this.surname = surname;
        this.email = email;
        this.password = password;
        this.authorities = authorities;
    }
}

```

```

    }

    @JsonIgnore
    public Long getId() {
        return id;
    }

    @Override
    public String getUsername() {
        return login;
    }

    @JsonIgnore
    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @JsonIgnore
    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @JsonIgnore
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @JsonIgnore
    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public Collection<? extends GrantedAuthority>
getAuthorities() {
        return authorities;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}

```

Файл JwtUserFactory.java

```

public final class JwtUserFactory {

    public JwtUserFactory() {
    }

    public static JwtUser create(UserDto user) {
        return new JwtUser(user.getId(), user.getLogin(),
            user.getName(), user.getSurname(),
            user.getEmail(), user.getPassword(),
            mapToGrantedAuthorities(new
                ArrayList<>(user.getRoles())))
    };

    private static List<GrantedAuthority>
        mapToGrantedAuthorities(List<Role> userRoles) {
        return userRoles.stream()
            .map(role ->
                new SimpleGrantedAuthority(role.getName()))
            .collect(Collectors.toList());
    }
}

```

Файл JwtDetailsService.java

```

@Service
public class JwtUserDetailsService implements UserDetailsService
{

    private final UserService userService;

    @Autowired
    public JwtUserDetailsService(UserService userService) {
        this.userService = userService;
    }

    @Override
    public UserDetails loadUserByUsername(String login) throws
        UsernameNotFoundException {
        UserDto userDTO = new
            UserDto(userService.findByLogin(login));
        return JwtUserFactory.create(userDTO);
    }
}

```

Файл BookingClubController.java

```

public interface BookingClubController<T> {

    @PostMapping(consumes = MediaType.APPLICATION_JSON_VALUE,

```

```

produces = MediaType.APPLICATION_JSON_VALUE)
    ResponseEntity<?> create(@RequestBody @Valid T t);

    @GetMapping(path =("/{id}", produces =
MediaType.APPLICATION_JSON_VALUE)
    ResponseEntity<EntityModel<T>> findById(@PathVariable long
id);

    @DeleteMapping(path =("/{id}")
    ResponseEntity<Void> delete(@PathVariable Long id);

    @PutMapping(path =("/{id}", consumes =
MediaType.APPLICATION_JSON_VALUE, produces =
MediaType.APPLICATION_JSON_VALUE)
    ResponseEntity<EntityModel<T>> update(@RequestBody @Valid T
t, @PathVariable int id);
}

```

Файл BookConverter.java

```

@Component
public class BookConverter implements DtoConverter<BookDto,
Book> {

    @Override
    public List<BookDto> convert(List<Book> books) {
        return
books.stream().map(BookDto::new).collect(Collectors.toList());
    }

    @Override
    public Book convert(BookDto bookDTO) {
        return new Book(bookDTO.getName(),
bookDTO.getDescription(), bookDTO.getAuthor(),
bookDTO.getCount(),
        bookDTO.getCreationDate());
    }
}

```

Файл UserConverter.java

```

@Component
public class UserConverter implements DtoConverter<UserDto,
User> {

    @Override
    public List<UserDto> convert(List<User> users) {
        return users
            .stream()
            .map(UserDto::new)
            .collect(Collectors.toList());
    }
}

```

```

    }

    @Override
    public User convert(UserDto user) {
        return new User(user.getId(), user.getName(),
            user.getSurname(), user.getLogin(), user.getPassword(),
            user.getEmail(), user.getRoles(), null, null);
    }
}

```

Файл UserRegistrationConverter.java

```

@Component
public class UserRegistrationConverter implements
    DtoConverter<RegistrationUserDto, User> {

    @Override
    public List<RegistrationUserDto> convert(List<User> users) {
        return users
            .stream()
            .map(RegistrationUserDto::new)
            .collect(Collectors.toList());
    }

    @Override
    public User convert(RegistrationUserDto user) {
        return new User(user.getId(), user.getName(),
            user.getSurname(), user.getLogin(), user.getPassword(),
            user.getEmail(), user.getRoles(), null, null);
    }
}

```

Файл BookService.java

```

@Transactional
@Service
public class BookServiceJpa implements BookService {

    private final BookRepository bookRepository;
    private final ReviewRepository reviewRepository;

    @Autowired
    public BookServiceJpa(BookRepository bookRepository,
        ReviewRepository reviewRepository) {
        this.bookRepository = bookRepository;
        this.reviewRepository = reviewRepository;
    }

    @Override
    public int getBooksCount(Map<String, String> request,

```



```

List<Review> reviews) {
    return bookRepository.getBookCount();
}

/**
 * @param params
 * @return Book list
 */
@Override
public List<Book> findAllBooksWithPagination(int page, int
size) {
    int limit = page*size-size;
    Pageable pageable = new OffsetBasedPageRequest(size,
limit);
    Page<Book> page1 = bookRepository.findAll(pageable);
    return page1.getContent();
}

@Override
public int getBookCount(long id) {
    Book book = findBookInRepository(id);
    return book.getCount();
}

@Override
public Book findById(long id) {
    return findBookInRepository(id);
}

@Override
public void delete(long id) {
    bookRepository.deleteById(id);
}

@Override
public Book update(Book book, long id) {
    Book mutableBook = findBookInRepository(id);

    if (isBlank(book.getAuthor())) {
        book.setAuthor(mutableBook.getAuthor());
    }
    if (isBlank(book.getName())) {
        book.setName(mutableBook.getName());
    }
    if (isBlank(book.getDescription())) {
        book.setDescription(mutableBook.getDescription());
    }

    bookRepository.update(id, book.getName(),
book.getDescription(), book.getAuthor());
    return this.findById(id);
}

```

```

@Override
public int getAllBooksCount() {
    return bookRepository.findAll().size();
}

@Override
public Book create(Book book) {
    book.setCreationDate(new Date());
    return bookRepository.save(book);
}

private Book findBookInRepository(long id) {
    return bookRepository.findById(id)
        .orElseThrow(() -> new
ServiceBadRequestException(new InvalidDataMessage("This ID is
does not exist!")));
}
}

```

Файл UserService.java

```

@Transactional
@Service
public class UserServiceJpa implements UserService {

    private final UserRepository userRepository;
    private final RoleRepository roleRepository;

    @Autowired
    public UserServiceJpa(UserRepository userRepository,
RoleRepository roleRepository) {
        this.userRepository = userRepository;
        this.roleRepository = roleRepository;
    }

    @Override
    public List<User> findAllUsers(int page, int size) {

        int limit = page*size-size;
        Pageable pageable = new OffsetBasedPageRequest(size,
limit);
        return userRepository.findAll(pageable).getContent();
    }

    @Override
    public List<User> findAllUsers() {
        return userRepository.findAll();
    }

    @Override

```

```

    public User findById(long id) {
        return userRepository.findById(id)
            .orElseThrow(() -> new
ServiceBadRequestException(new InvalidDataMessage("Invalid
Id!")));
    }

    @Override
    public void delete(long id) {
        userRepository.deleteById(id);
    }

    @Override
    public User create(User user) {

        long userClassicRole = 1L;
        BCryptPasswordEncoder bCryptPasswordEncoder = new
BCryptPasswordEncoder();
        String actualPassword = user.getPassword();
        String hashedPassword =
bCryptPasswordEncoder.encode(actualPassword);
        user.setPassword(hashedPassword);
        User changeUser = userRepository.save(user);
        ArrayList<Role> roles = new ArrayList<>();

        roleRepository.findById(userClassicRole).ifPresent(roles::add);
        changeUser.setRoles(roles);
        return userRepository.save(changeUser);
    }

    @Override
    public User update(long id, User user) {
        User mutableUser = this.findById(id);

        if (isBlank(user.getName())) {
            user.setName(mutableUser.getName());
        }
        if (isBlank(user.getSurname())) {
            user.setSurname(mutableUser.getSurname());
        }
        if (isBlank(user.getEmail())) {
            user.setEmail(mutableUser.getEmail());
        }
        if (isBlank(user.getLogin())) {
            user.setLogin(mutableUser.getLogin());
        }

        userRepository.update(id, user.getName(),
user.getSurname(), user.getEmail(), user.getLogin());
        return this.findById(id);
    }

```

```

@Override
public User findByLogin(String login) {
    return userRepository.findByLogin(login);
}

@Override
public int getUsersCount() {
    return 0;
}
}

```

Файл OffsetBasedRequest.java

```

public class OffsetBasedPageRequest implements Pageable {
    private int limit;
    private int offset;

    private final Sort sort = by(Sort.Direction.DESC, "id");
    public OffsetBasedPageRequest(int limit, int offset) {
        if (limit < 1) {
            throw new IllegalArgumentException("Limit must not
be less than one!");
        }
        if (offset < 0) {
            throw new IllegalArgumentException("Offset index
must not be less than zero!");
        }
        this.limit = limit;
        this.offset = offset;
    }
    @Override
    public int getPageNumber() {
        return offset / limit;
    }
    @Override
    public int getPageSize() {
        return limit;
    }
    @Override
    public long getOffset() {
        return offset;
    }
    @Override
    public Sort getSort() {
        return sort;
    }
    @Override
    public Pageable next() {
        return new OffsetBasedPageRequest(getPageSize(), (int)
(getOffset() + getPageSize()));
    }
}

```

```

    }
    public Pageable previous() {
        return hasPrevious() ?
            new OffsetBasedPageRequest(getPageSize(), (int)
(getOffset() - getPageSize())): this;
    }
    @Override
    public Pageable previousOrFirst() {
        return hasPrevious() ? previous() : first();
    }
    @Override
    public Pageable first() {
        return new OffsetBasedPageRequest(getPageSize(), 0);
    }
    @Override
    public boolean hasPrevious() {
        return offset > limit;
    }
}

```

Файл BookRepository.java

```

@Repository
public interface BookRepository extends JpaRepository<Book,
Long> {

    @Override
    List<Book> findAll();

    List<Book> findByIdBetween(Long offset, Long lastBook);

    @Override
    List<Book> findAll(Sort sort);

    @Override
    List<Book> findAllById(Iterable<Long> iterable);

    int getBookById(long id);

    @Modifying(clearAutomatically = true)
    @Query("update book b set b.name = :name, b.description =
:description, b.author = :author where b.id = :id")
    void update(@Param("id") long id, @Param("name") String name,
@Param("description") String description,
@Param("author") String
author);

    @Modifying
    @Query("select b.count from book b where b.id = :id")

```

```

        int getBookCount();
    }

```

Файл RoleRepository.java

```

public interface RoleRepository extends CrudRepository<Role,
Long> {

    @Override
    Optional<Role> findById(Long aLong);
}

```

Файл UserRepository.java

```

public interface UserRepository extends JpaRepository<User,
Long> {

    User findByLogin(String login);

    @Override
    <S extends User> S save(S s);

    void deleteById(long id);

    List<User> findAll();

    @Modifying(clearAutomatically = true)
    @Query("update bk_user u set u.name = :name, u.surname =
:surname, u.email = :email, u.login = :login where u.id = :id")
    void update(@Param("id") long id, @Param("name") String
name, @Param("surname") String surname,
                @Param("email") String email, @Param("login")
String login);
}

```

Файл RepositoryNotFoundException.java

```

public class RepositoryNotFoundException extends
RuntimeException {

    private final static String MESSAGE = "Repository
exception";
    private List<InvalidDataOutputMessage> messages;
    private InvalidDataOutputMessage message;

    public
RepositoryNotFoundException(List<InvalidDataOutputMessage>
messages) {
        this.messages = messages;
    }
}

```

```

    }

    public RepositoryNotFoundException() {
    }

    public RepositoryNotFoundException(InvalidDataOutputMessage
message) {
        this.message = message;
        messages = new ArrayList<>();
        messages.add(message);
    }

    public List<InvalidDataOutputMessage> getMessages() {
        return messages;
    }

    public InvalidDataOutputMessage getErrorMessage() {
        return message;
    }

    @Override
    public String getMessage() {
        return MESSAGE;
    }

    public String getStatus(){
        return HttpStatus.NOT_FOUND.toString();
    }
}

```

Файл Book.java

```

@Entity(name = "book")
@Table(name = "book")
@Data
@NoArgsConstructor
public class Book implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "count")
    private int count;

    @Column(name = "description")

```

```

        private String description;

        @Column(name = "date_of_creation")
        private Date creationDate;

        @Column(name = "author")
        private String author;

        public Book(String name, String description, String author,
int count, Date creationDate) {
            this.name = name;
            this.count = count;
            this.description = description;
            this.creationDate = creationDate;
            this.author = author;
        }
    }
}

```

Файл BookGroup.java

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "book_group")
@Table(name = "book_group")
public class BookGroup {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "name")
    private String name;

    @Column(name = "description")
    private String description;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "user_roles",
        joinColumns = @JoinColumn(name = "group_id"),
        inverseJoinColumns = @JoinColumn(name = "book_id"))
    private List<Book> books;
}

```

Файл BookClubOrder.java

```

@Data
@Entity(name = "bookingClubOrder")
@Table(name = "bookingClubOrder")

```



```

@NoArgsConstructor
@AllArgsConstructor
public class BookingClubOrder {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "create_time")
    private Date createTime;

    @Column(name = "end_time")
    private Date endTime;

    @ManyToOne
    @JoinColumn(name="status_id", insertable = false, updatable
= false)
    private OrderStatus orderStatus;

    @ManyToOne
    @JoinColumn(name = "user_id", insertable = false, updatable
= false)
    private User author;

    @ManyToOne
    @JoinColumn(name = "book_id", insertable = false, updatable
= false)
    private Book book;
}

```

Файл OrderStatus.java

```

@Data
@Entity(name = "status")
@Table(name = "status")
@NoArgsConstructor
@AllArgsConstructor
public class OrderStatus {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long status_id;

    @Column(name = "user_name")
    private String name;
}

```

Файл Review.java

```

@Entity(name = "review")
@Table(name = "review")
@Data
public class Review implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "topic")
    private String topic;

    @Column(name = "description")
    private String description;

    @Column(name = "date_of_creation")
    private Date creationDate;

    @Column(name = "rating")
    private Integer rating;

    @ManyToOne
    @JoinColumn(name = "user_id", nullable = false, insertable =
false, updatable = false)
    private User owner;

    @ManyToOne
    @JoinColumn(name = "book_id", nullable = false, insertable =
false, updatable = false)
    private Book book;

    public Review() {
    }
}

```

Файл Role.java

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "role")
@Table(name = "role")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "role_id")
    private Long id;

    @Column(name = "name")

```

```

private String name;

@Enumerated(EnumType.STRING)
@Column(name = "status")
private Status status;

@Override
public String toString() {
    return "Role{" + "id: " + this.getId() + ", " + "name: "
+ name + "}";
}
}

```

Файл User.java

```

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity(name = "bk_user")
@Table(name = "bk_user")
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id")
    private Long id;

    @Column(name = "user_name")
    private String name;

    @Column(name = "user_surname")
    private String surname;

    @Column(name = "login")
    private String login;

    @Column(name = "user_password")
    private String password;

    @Column(name = "email")
    private String email;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name = "user_roles",
        joinColumns = @JoinColumn(name = "user_id"),
        inverseJoinColumns = @JoinColumn(name = "role_id"))
    private List<Role> roles;

    @ManyToMany(fetch = FetchType.LAZY)
    @Fetch(value = FetchMode.SUBSELECT)

```

```

@JoinTable(name = "user_books",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "book_id"))
private List<Book> book;

@ManyToMany(fetch = FetchType.LAZY)
@Fetch(value = FetchMode.SUBSELECT)
@JoinTable(name = "user_booksGroups",
    joinColumns = @JoinColumn(name = "user_id"),
    inverseJoinColumns = @JoinColumn(name = "bookGroup_id"))
private List<BookGroup> bookGroups;

public User(String name, String surname, String login,
String password, String email, List<BookGroup> bookGroups) {
    this.name = name;
    this.surname = surname;
    this.login = login;
    this.password = password;
    this.email = email;
    this.bookGroups = bookGroups;
}
}

```

Файл ApiErrorDto.java

```

@Data
@Getter
@Setter
@JsonInclude(JsonInclude.Include.NON_NULL)
public class ApiErrorDto {

    private String status;
    private String message;
    private String path;

    @JsonFormat(pattern = "YYYY-MM-dd HH:mm")
    private Date date;

    public ApiErrorDto(String status, String message, String
path) {
        this.path = path;
        this.message = message;
        this.status = status;
        this.date = new Date();
    }
}

```

Файл AuthenticationRequestDto.java

```

@Data
@Getter
@Setter
@JsonInclude(JsonInclude.Include.NON_NULL)
public class ApiErrorDto {

    private String status;
    private String message;
    private String path;

    @JsonFormat(pattern = "YYYY-MM-dd HH:mm")
    private Date date;

    public ApiErrorDto(String status, String message, String
path) {
        this.path = path;
        this.message = message;
        this.status = status;
        this.date = new Date();
    }
}

```

Файл BookDto.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class BookDto {

    @ToString.Exclude
    @NotNull(message = "{validation.book.id}")
    private Long id;

    //@NotNull(message = "{validation.user.name}")
    //@Size(min = 2, max = 70, message =
"{validation.user.name.not.null}")
    private String name;

    //@NotNull(message = "{validation.user.surname}")
    //@Size(min = 3, max = 170, message =
"{validation.user.surname}")
    private String description;

    private Date creationDate;

    private String author;

    private int count;

    @JsonIgnore
    private EntityModel<BookDto> model;
}

```

```

public BookDto(Book book) {
    this.id = book.getId();
    this.name = book.getName();
    this.description = book.getDescription();
    this.creationDate = book.getCreationDate();
    this.author = book.getAuthor();
    this.count = book.getCount();
}

public EntityModel<BookDto> getModel() {
    String deleteRelName = "delete";
    String orderRelName = "orders";
    String methodTypeDELETE = "DELETE";
    String methodTypeGET = "GET";
    int defaultPage = 1;
    int defaultSize = 5;

    model = EntityModel
        .of(this,
linkTo(methodOn(BookController.class).findById(id)).withSelfRel(
).withType(methodTypeGET),

linkTo(methodOn(BookController.class).delete(id)).withRel(delete
RelName)
        .withType(methodTypeDELETE));
    return model;
}
}

```

Файл BookGroupDto.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class BookGroupDto {

    @ToString.Exclude
    @Null(message = "{validation.book.id}")
    private Long id;

    private String name;

    private String description;

    private List<Book> books;

    @JsonIgnore
    private EntityModel<BookGroupDto> model;

    public BookGroupDto(BookGroup group) {

```

```

        this.id = group.getId();
        this.name = group.getName();
        this.description = group.getDescription();
        this.books = getBooks();
    }

    public EntityModel<BookGroupDto> getModel() {
        String deleteRelName = "delete";
        String groupRelName = "groups";
        String methodTypeDELETE = "DELETE";
        String methodTypeGET = "GET";
        int defaultPage = 1;
        int defaultSize = 5;

        model = EntityModel
            .of(this,
linkTo(methodOn(BookGroupController.class).findById(id)).withSelfRel().withType(methodTypeGET),

linkTo(methodOn(BookGroupController.class).delete(id)).withRel(deleteRelName)
            .withType(methodTypeDELETE));
        return model;
    }
}

```

Файл BookGroupDto.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class BookGroupDto {

    @ToString.Exclude
    @Null(message = "{validation.book.id}")
    private Long id;

    private String name;

    private String description;

    private List<Book> books;

    @JsonIgnore
    private EntityModel<BookGroupDto> model;

    public BookGroupDto(BookGroup group) {
        this.id = group.getId();
        this.name = group.getName();
        this.description = group.getDescription();
        this.books = getBooks();
    }
}

```

```

    public EntityModel<BookGroupDto> getModel() {
        String deleteRelName = "delete";
        String groupRelName = "groups";
        String methodTypeDELETE = "DELETE";
        String methodTypeGET = "GET";
        int defaultPage = 1;
        int defaultSize = 5;

        model = EntityModel
            .of(this,
linkTo(methodOn(BookGroupController.class).findById(id)).withSelfRel().withType(methodTypeGET),

linkTo(methodOn(BookGroupController.class).delete(id)).withRel(deleteRelName)
                .withType(methodTypeDELETE));
        return model;
    }
}

```

Файл BookList.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class BookList {

    private List<BookDto> items;
    private int booksCount;

    public BookList(List<BookDto> items, int booksCount) {
        this.items = items;
        this.booksCount = booksCount;
    }
}

```

Файл OrderDto.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class OrderDto {

    @ToString.Exclude
    @Null(message = "{validation.book.id}")
    private Long id;

    private Date createTime;

    private Date endTime;
}

```



```

private OrderStatus orderStatus;

private User author;

private Book book;

@JsonIgnore
private EntityModel<OrderDto> model;

public OrderDto(BookingClubOrder order) {
    this.id = order.getId();
    this.createTime = order.getCreateTime();
    this.endTime = order.getEndTime();
    this.orderStatus = order.getOrderStatus();
    this.author = order.getAuthor();
    this.book = order.getBook();
}

public EntityModel<OrderDto> getModel() {
    String deleteRelName = "delete";
    String orderRelName = "orders";
    String methodTypeDELETE = "DELETE";
    String methodTypeGET = "GET";
    int defaultPage = 1;
    int defaultSize = 5;

    model = EntityModel
        .of(this,
linkTo(methodOn(OrderController.class).findById(id)).withSelfRel
().withType(methodTypeGET),

linkTo(methodOn(OrderController.class).delete(id)).withRel(deleteRelName).withType(methodTypeDELETE));
    return model;
}
}

```

Файл RegistrationUserDto.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class RegistrationUserDto {

    @ToString.Exclude
    @NotNull(message = "{validation.user.id}")
    private Long id;

    @NotNull(message = "{validation.user.name}")
    @Size(min = 2, max = 70, message =
"{validation.user.name.not.null}")

```

```

    private String name;

    @NotNull(message = "{validation.user.surname}")
    @Size(min = 3, max = 170, message =
"{validation.user.surname}")
    private String surname;

    @NotNull(message = "{validation.user.login}")
    @Size(min = 5, max = 30, message =
"{validation.user.login.not.null}")
    private String login;

    @NotNull(message = "{validation.user.password.not.null}")
    @Size(min = 4, max = 30, message =
"{validation.user.password}")
    private String password;

    @Email(message = "{validation.user.email}")
    @NotNull(message = "{validation.user.email.not.null}")
    private String email;

    @Null(message = "{validation.user.roles.null}")
    private List<Role> roles;

    @JsonIgnore
    private EntityModel<RegistrationUserDto> model;

    public RegistrationUserDto(User user) {
        this.id = user.getId();
        this.name = user.getName();
        this.surname = user.getSurname();
        this.login = user.getLogin();
        this.password = user.getPassword();
        this.roles = user.getRoles();
        this.email = user.getEmail();
    }

    public EntityModel<RegistrationUserDto> getModel(int page,
int size) {
        String deleteRelName = "delete";
        String orderRelName = "orders";
        String methodTypeDELETE = "DELETE";
        String methodTypeGET = "GET";

        model = EntityModel.of(this,
linkTo(methodOn(UserController.class).findById(id)).withSelfRel(
),
linkTo(methodOn(UserController.class).delete(id)).withRel(delete
RelName).withType(methodTypeDELETE));
        return model;
    }

```

```
}  
}
```

Файл UserDto.java

```
@Data  
@NoArgsConstructor  
@JsonInclude(JsonInclude.Include.NON_NULL)  
public class UserDto {  
  
    @ToString.Exclude  
    @Null(message = "{validation.user.id}")  
    private Long id;  
  
    @NotNull(message = "{validation.user.name}")  
    @Size(min = 2, max = 70, message =  
"{validation.user.name.not.null}")  
    private String name;  
  
    @NotNull(message = "{validation.user.surname}")  
    @Size(min = 3, max = 170, message =  
"{validation.user.surname}")  
    private String surname;  
  
    @NotNull(message = "{validation.user.login}")  
    @Size(min = 5, max = 30, message =  
"{validation.user.login.not.null}")  
    private String login;  
  
    @NotNull(message = "{validation.user.password.not.null}")  
    @Size(min = 4, max = 30, message =  
"{validation.user.password}")  
    @JsonIgnore  
    private String password;  
  
    @Email(message = "{validation.user.email}")  
    @NotNull(message = "{validation.user.email.not.null}")  
    private String email;  
  
    @Null(message = "{validation.user.roles.null}")  
    private List<Role> roles;  
  
    @JsonIgnore  
    private EntityModel<UserDto> model;  
  
    public UserDto(User user) {  
        this.id = user.getId();  
        this.name = user.getName();  
        this.surname = user.getSurname();  
    }  
}
```

```

        this.login = user.getLogin();
        this.password = user.getPassword();
        this.email = user.getEmail();
        this.roles = user.getRoles();
    }

    public EntityModel<UserDto> getModel(int page, int size) {
        String deleteRelName = "delete";
        String orderRelName = "orders";
        String methodTypeDELETE = "DELETE";
        String methodTypeGET = "GET";

        model = EntityModel.of(this,
            linkTo(methodOn(UserController.class)

.findById(id)).withSelfRel().withType(methodTypeGET),
            linkTo(methodOn(UserController.class)

.delete(id)).withRel(deleteRelName).withType(methodTypeDELETE)
        );
        return model;
    }

    public EntityModel<UserDto> getModel() {
        String deleteRelName = "delete";
        String orderRelName = "orders";
        String methodTypeDELETE = "DELETE";
        String methodTypeGET = "GET";
        int defaultPage = 1;
        int defaultSize = 5;

        model = EntityModel.of(this,
            linkTo(methodOn(UserController.class)

.findById(id)).withSelfRel().withType(methodTypeGET),
            linkTo(methodOn(UserController.class)

.delete(id)).withRel(deleteRelName).withType(methodTypeDELETE));
        return model;
    }
}

```

Файл UserList.java

```

@Data
@NoArgsConstructor
@JsonInclude(JsonInclude.Include.NON_NULL)
public class UserList {

    private List<UserDto> items;
    private int usersCount;
}

```

```
public UserList(List<UserDto> items) {  
    this.items = items;  
}  
  
public UserList(List<UserDto> items, int usersCount) {  
    this.items = items;  
    this.usersCount = usersCount;  
}  
}
```

ПРИЛОЖЕНИЕ Б

(обязательное)

Спецификация программного дипломного проекта

ПРИЛОЖЕНИЕ В
(обязательное)
Модель данных

ПРИЛОЖЕНИЕ Г
(обязательное)
Ведомость документов