**1. Общий обзор структуры проекта**

Проект представляет собой веб-приложение, написанное на **Java** с использованием **Spring Boot** и ряда связанных технологий. Оно предназначено для хранения и управления данными о книгах, авторах, издательствах и пр. У приложения много сущностей (таблиц в БД), а также контроллеров для работы с ними через REST API.

**Основной стек**:

1. **Spring Boot** — фреймворк, который упрощает создание веб-приложений (автоматическая конфигурация, встроенный сервер и т. д.).
2. **Spring Data JPA** — модуль, позволяющий работать с базой данных через репозитории и сущности (Entity) без написания большого количества SQL-запросов вручную.
3. **Hibernate** — реализация JPA по умолчанию. Он «общается» с базой данных, переводя объектно-ориентированные сущности в SQL.
4. **Lombok** — библиотека, которая генерирует стандартные методы (геттеры, сеттеры, конструкторы) на этапе компиляции, чтобы не писать их вручную.
5. **OpenCSV** — библиотека для чтения и записи CSV-файлов (экспорт/импорт).
6. **JUnit5 + Mockito** (и Spring Test) — фреймворки для модульного и интеграционного тестирования.
7. **Maven** (вероятно) или **Gradle** — для сборки и управления зависимостями (по коду это похоже на Maven-проект).

Кроме этого в проекте используется:

* **Spring Validation** и **Jakarta Validation** (через аннотации @Valid, @NotBlank, @NotNull и т.д.) для валидации входных данных в контроллерах.
* **SLF4J + Logback** или **SLF4J + Log4j** — для логирования (logger.info, logger.error и др.).
* **Jackson** (библиотека сериализации/десериализации JSON) — для превращения Java-объектов в JSON и обратно.  
  В частности, в конфигурационном классе JacksonTestConfig регистрируются JavaTimeModule и отключение timestamps для дат.

**Архитектурные слои**

1. **Контроллеры (controller/...)**  
   Отвечают за приём HTTP-запросов и возврат HTTP-ответов. Внутри вызывают сервисы.
2. **Сервисы (service/...)**  
   Содержат бизнес-логику и «правила работы» с данными. Вызывают репозитории, обрабатывают результаты, возвращают их контроллерам.
3. **Репозитории (repository/...)**  
   Работают напрямую с базой данных, используя Spring Data JPA. Методы репозиториев позволяют находить, сохранять, удалять и обновлять сущности.
4. **Сущности (dao/...)**  
   Java-классы, соответствующие таблицам базы данных. Аннотированы @Entity, поля связываются с колонками через @Column, @Id, @ManyToOne, @OneToMany и т. д.

Также есть **DTO (Data Transfer Objects)** (пакет dto/), которые упрощают работу контроллеров с данными (не всегда хотим передавать всю сущность наружу).

**2. Состав пакетов и основных классов**

Ниже перечислены ключевые моменты.

**2.1. BookstorageApplication.java**

Это главный класс с аннотацией @SpringBootApplication.

* При запуске (метод main) Spring Boot автоматически поднимает встроенный веб-сервер (Tomcat по умолчанию), сканирует классы, поднимает контекст, создаёт все нужные бины.
* После этого приложение готово принимать HTTP-запросы.

java

Копировать код

@SpringBootApplication

public class BookstorageApplication {

public static void main(String[] args) {

SpringApplication.run(BookstorageApplication.class, args);

}

}

В нём же добавлено логирование начала и конца запуска приложения.

**2.2. Конфигурационный класс JacksonTestConfig.java**

Настраивает объект ObjectMapper от Jackson, который отвечает за сериализацию/десериализацию в JSON.

* Регистрирует модуль JavaTimeModule для работы с LocalDate, LocalDateTime и т.д.
* Отключает запись дат в виде timestamps, чтобы было в виде ISO-строк.

java

Копировать код

@Configuration

public class JacksonTestConfig {

@Bean

public ObjectMapper objectMapper() {

...

}

}

**2.3. Сущности (DAO)**

Находятся в папке dao. Каждая сущность соответствует таблице в базе данных.

* **Author**: таблица authors  
  Поля: id, fio, birthDate, country, nickname.  
  Одному автору может соответствовать несколько авторств (поле Set<Authorship> authorships).
* **Authorship** и AuthorshipId: связь многие-ко-многим между Author и Book, реализуется через промежуточную таблицу authorships.  
  Authorship имеет составной первичный ключ AuthorshipId.  
  AuthorshipId включает bookIsbn и authorId.
* **Book**: таблица books.  
  Поля: isbn (PRIMARY KEY), name, publicationYear (LocalDate), ageLimit (float), publishingCompany (ссылка на издательство), и др.  
  Также содержит Set<Authorship> и Set<BookStyles>.
* **Styles**: таблица styles. Содержит поле id (PK), name (название стиля/жанра).
* **BookStyles** и BookStylesId: аналогично Authorship, промежуточная таблица book\_styles, которая связывает книги и стили.
* **PublishingCompany**: таблица publishing\_companies.  
  Поле name — ключ (PRIMARY KEY), плюс establishmentYear, contactInfo, city и др.  
  Может ссылаться на множество книг через Set<Book> books.

Каждая сущность имеет аннотации JPA:

* @Entity для указания, что это таблица.
* @Id, @Column и т.д.
* @OneToMany, @ManyToOne, @JoinColumn для связей между таблицами.

**2.4. Репозитории (пакет repository)**

Это интерфейсы, которые наследуются от JpaRepository<Сущность, Ключ>. Spring автоматически генерирует их реализации.  
Например, AuthorRepository extends JpaRepository<Author, Integer> — умеет искать, сохранять, удалять Author.

* Также могут содержать методы поиска по полям, например findByFioContainingIgnoreCase(...).

Примеры репозиториев:

* AuthorRepository
* BookRepository
* PublishingCompanyRepository
* и т.д.

**2.5. Сервисы (пакет service)**

Каждая сущность имеет свой сервис, который инкапсулирует бизнес-логику. Пример:

java

Копировать код

@Service

@AllArgsConstructor

@Transactional

public class AuthorServiceImpl implements AuthorService {

private final AuthorRepository authorRepository;

...

@Override

public Author createAuthor(Author author) {

// Логика: например, логирование, сохранение через репозиторий

return authorRepository.save(author);

}

...

}

* Методы сервисов вызывают репозитории, обрабатывают ошибки (например, если не найдено — бросить исключение), и возвращают результаты.
* Аннотация @Transactional говорит о том, что методы выполняются в транзакции на уровне базы данных.
* Существует сервис для авторов, книг, издательств, авторств (Authorship), стилей и т.д.

**2.6. Контроллеры (пакет controller)**

Каждый контроллер обрабатывает запросы, приходящие по определённым URL.

Например, AuthorController:

java

Копировать код

@RestController

@RequestMapping("/api/authors")

@RequiredArgsConstructor

public class AuthorController {

private final AuthorService authorService;

@PostMapping

public ResponseEntity<AuthorDTO> createAuthor(@Valid @RequestBody AuthorDTO authorDTO) {

// 1. Логируем

// 2. mapToEntity(authorDTO)

// 3. Вызываем authorService.createAuthor(author)

// 4. Возвращаем mapToDTO(createdAuthor)

}

@GetMapping("/{id}")

public ResponseEntity<AuthorDTO> getAuthorById(@PathVariable int id) {

// 1. Вызываем service.getAuthorById(id)

// 2. Если автор есть, возвращаем OK + DTO

// 3. Если нет, возвращаем NOT\_FOUND

}

...

}

Ключевые моменты:

* @RestController + @RequestMapping -> URL начинается с /api/authors.
* Методы @GetMapping, @PostMapping, @PutMapping, @DeleteMapping соответствуют HTTP-методам GET, POST, PUT, DELETE.
* Параметры метода:
  + @RequestBody AuthorDTO authorDTO означает, что тело запроса (JSON) будет преобразовано в объект authorDTO.
  + @PathVariable int id означает, что часть URL (например, .../authors/5) берётся как id.
  + @RequestParam берёт параметры из query-string (например, .../search?q=Иван).

Контроллеры занимают весь REST-функционал:

* Создание, чтение (по ID или все), обновление, удаление, поиск и т.д.

**Дополнительные контроллеры:**

* BookController (для книг)
* PublishingCompanyController (для издательств)
* StylesController (для стилей/жанров)
* AuthorshipController (для связи книг и авторов)
* BookStylesController (для связи книг и стилей)
* CsvController (для экспорта/импорта в CSV).

**2.7. DTO (пакет dto)**

**DTO** — объекты, в которые (или из которых) мапятся наши сущности, чтобы не раскрывать наружу все поля или не возвращать громоздкие объекты с лишними полями. В DTO могут быть иные названия полей, своя валидация.  
Например, AuthorDTO, BookDTO, StylesDTO и т.д.

**2.8. Экспорт/Импорт CSV (CsvController, CsvService)**

* **CsvService** содержит методы:
  + exportData() — выкачивает данные из БД в CSV-файлы (authors.csv, books.csv и т.п.).
  + importData() — загружает данные из CSV-файлов в БД.
  + При импорте сначала вызывается метод clearDatabase(), чтобы очистить таблицы.
* **CsvController** даёт эндпоинты /api/csv/export и /api/csv/import, которые вызывают соответствующие методы csvService.

Используется **OpenCSV** для работы с CSV:

* CSVWriter — для записи,
* CSVReader — для чтения.
* Настройки (BOM, разделители, игнор кавычек) прописаны прямо в коде.

**2.9. Тесты**

В папке test/java/org/application/bookstorage/... видны тесты (юнит и интеграционные):

* Контроллеры тестируются при помощи Spring Boot Test (@WebMvcTest), MockMvc для имитации HTTP-запросов.
* Сервисы тестируются Mockito (@ExtendWith(MockitoExtension.class)), чтобы подменять репозитории фейковыми реализациями и проверять бизнес-логику.

**3. Как приложение взаимодействует с базой данных (и какие таблицы используются)**

В конце кода есть SQL-скрипты создания таблиц:

sql

Копировать код

CREATE TABLE IF NOT EXISTS authors(...);

CREATE TABLE IF NOT EXISTS publishing\_companies(...);

CREATE TABLE IF NOT EXISTS books(...);

CREATE TABLE IF NOT EXISTS authorships(...);

CREATE TABLE IF NOT EXISTS styles(...);

CREATE TABLE IF NOT EXISTS book\_styles(...);

CREATE TABLE IF NOT EXISTS administrators(...);

Основные таблицы, которые используются в этом проекте:

1. **authors** (авторы)
2. **publishing\_companies** (издательства)
3. **books** (книги)
4. **authorships** (связь «многие ко многим» автор-книга)
5. **styles** (жанры)
6. **book\_styles** (связь «многие ко многим» книга-жанр)

При запуске Spring Boot, если настроена правильная БД (например, PostgreSQL или H2), и указано spring.jpa.hibernate.ddl-auto=update (или аналогичные настройки), Spring сам создаст таблицы (или обновит схему). Но здесь ещё вручную SQL тоже дан.

**3.1. Сущности в коде и таблицы в БД**

* Author <-> authors
* Book <-> books
* PublishingCompany <-> publishing\_companies
* Authorship <-> authorships
* Styles <-> styles
* BookStyles <-> book\_styles

**Ключи**:

* authors.id (PK), auto-increment (serial).
* books.isbn (PK), строковое поле.
* publishing\_companies.name (PK).
* styles.id (PK), auto-increment (serial).
* authorships (составной PK — book\_isbn + author\_id).
* book\_styles (составной PK — book\_isbn + style).

**4. Как выглядит работа (пример прохождения запроса)**

Допустим, пользователь через UI (страницу) совершает запрос **"Посмотреть список книг"**.  
Но т.к. у нас архитектура REST, предположим, UI просто отправляет GET-запрос на /api/books.

1. **Запрос** приходит на BookController.getAllBooks(...).
2. В методе вызывается bookService.getAllBooks(search, sortColumn, sortOrder).
3. BookServiceImpl внутри метода:
   * Смотрит, есть ли параметр search (текст для поиска).
   * Если есть, вызывает bookRepository.findByNameContainingIgnoreCase(..., sort).
   * Если нет, вызывает bookRepository.findAll(sort).
   * Данные (список Book) возвращаются в сервис.
4. Сервис возвращает эти данные контроллеру, контроллер мапит каждый Book в BookDTO (через метод mapToDTO).
5. Контроллер отдает JSON-список из BookDTO.
6. Фронт (JS в браузере) получает JSON, обрабатывает и рендерит на странице таблицу книг.

По такому же принципу работают операции **"Добавить книгу"** (POST /api/books), **"Обновить книгу"** (PUT /api/books/{isbn}), **"Удалить книгу"** (DELETE /api/books/{isbn}) и так далее.

**5. HTML-страницы в корне (пример: Каталог.html)**

Судя по приведённому HTML коду, у нас есть страница «Каталог»:

* Подключен **Bootstrap** для стилизации таблиц, кнопок и форм.
* Есть меню с ссылками: «Каталог», «Авторы», «Издательства», «Войти», «База данных».
* Кнопки «Загрузить базу данных» / «Выгрузить базу данных» по всей видимости связаны с вызовом CsvController (import/export).
* На странице есть форма поиска, таблица для вывода книг, кнопка «Добавить книгу», кнопка «Удалить выбранные».

Скрипт /js/Каталог.js, вероятно, занимается AJAX-запросами к /api/books (и т. п.), заполняет таблицу данными, обрабатывает сортировку и т.д.

Таким образом, фронтенд часть на базе HTML/Bootstrap + JavaScript, а серверная часть — на базе Spring Boot (REST).

**6. Заключение. Что надо понимать «для преподавателя»**

1. **Принцип работы**:
   * Запросы приходят в контроллеры.
   * Контроллеры вызывают сервисы.
   * Сервисы работают с репозиториями (JPA).
   * Репозитории обращаются к базе данных.
   * Результат возвращается обратно по цепочке → клиент получает JSON-ответ.
   * Фронт на HTML/JS/Bootstrap эти данные рендерит в таблицы, формы и т.д.
2. **Связи в БД**:
   * Один автор может быть связан со многими книгами (через таблицу authorships).
   * Одна книга может иметь много авторов и много стилей.
   * Примерно такие же связи у издательств: одна запись в publishing\_companies может быть связана со многими книгами.
3. **Экспорт/Импорт** в CSV:
   * Позволяет выгружать данные таблиц (авторы, книги, стили и т. д.) в файлы CSV и загружать обратно.
   * При импорте старые данные чистятся (см. clearDatabase()).
4. **Тесты**:
   * Покрывают методы контроллеров (проверка ответов при разных сценариях) и сервисов (проверка бизнес-логики, например, валидации данных).
5. **Конфигурация**:
   * application.properties (не показан здесь, но обычно там лежит). Он указывает настройки подключения к БД (URL, логин-пароль), стратегию генерации схемы, порты и пр.
   * BookstorageApplication — входная точка.

Таким образом, проект **логичен и типичен** для Spring Boot: он разделён на слои, использует JPA для упрощённого доступа к БД, содержит REST-контроллеры для CRUD-операций и дополнительный функционал (поиск, импорт/экспорт).

Если вам нужно «показать преподавателю» понимание, стоит упомянуть:

* **Принцип Inversion of Control** и как Spring «поднимает» все наши бины, внедряя зависимости через @Autowired/@RequiredArgsConstructor.
* **Разницу** между «сущностью (Entity)» и «DTO».
* **Паттерн Репозиторий** и преимущества Spring Data JPA (не нужно писать SQL вручную).
* **Транзакции** (@Transactional) и почему важно, чтобы операции с базой были атомарны.
* **Тестирование**: как создаются заглушки (mock) репозиториев, чтобы тестировать сервисы в изоляции.