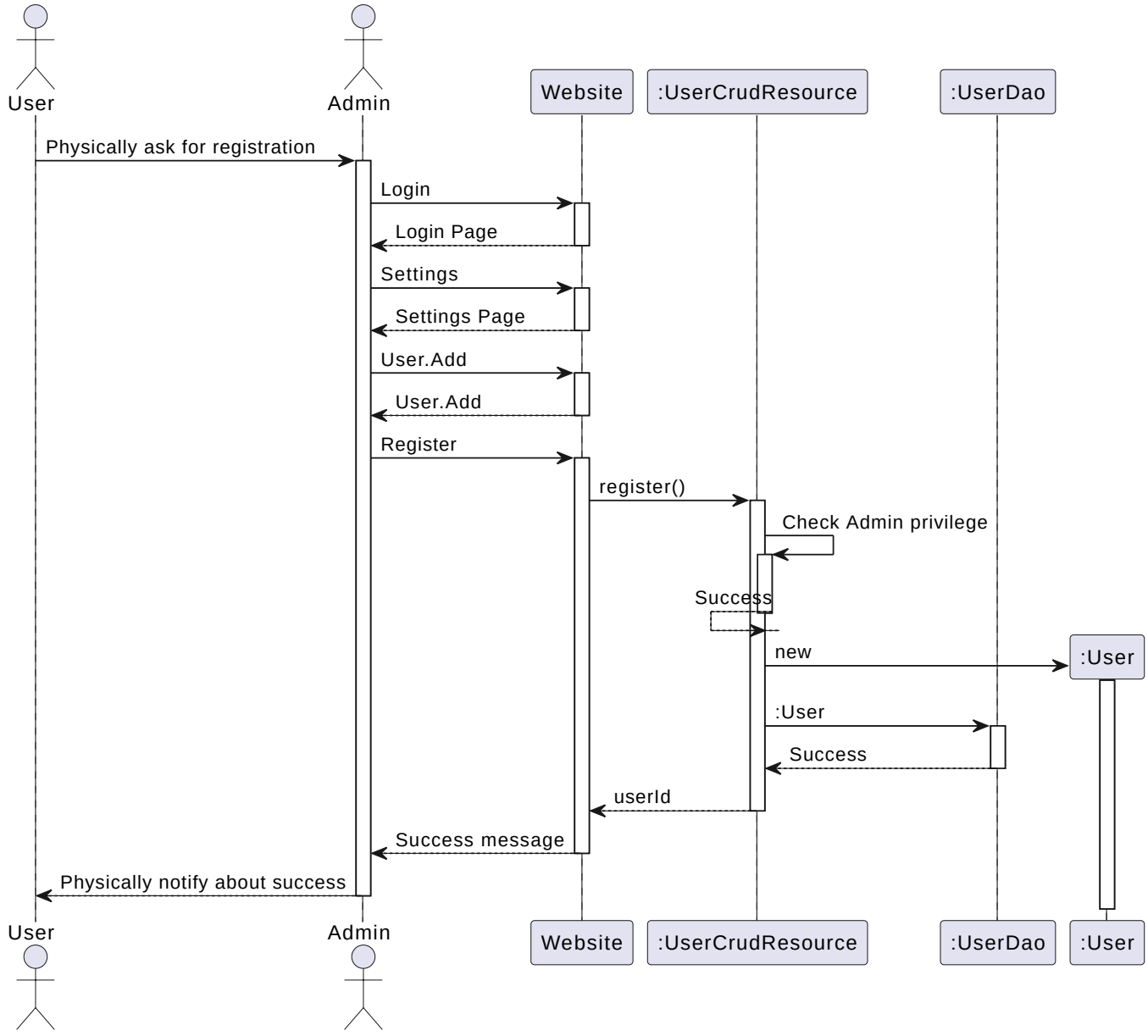
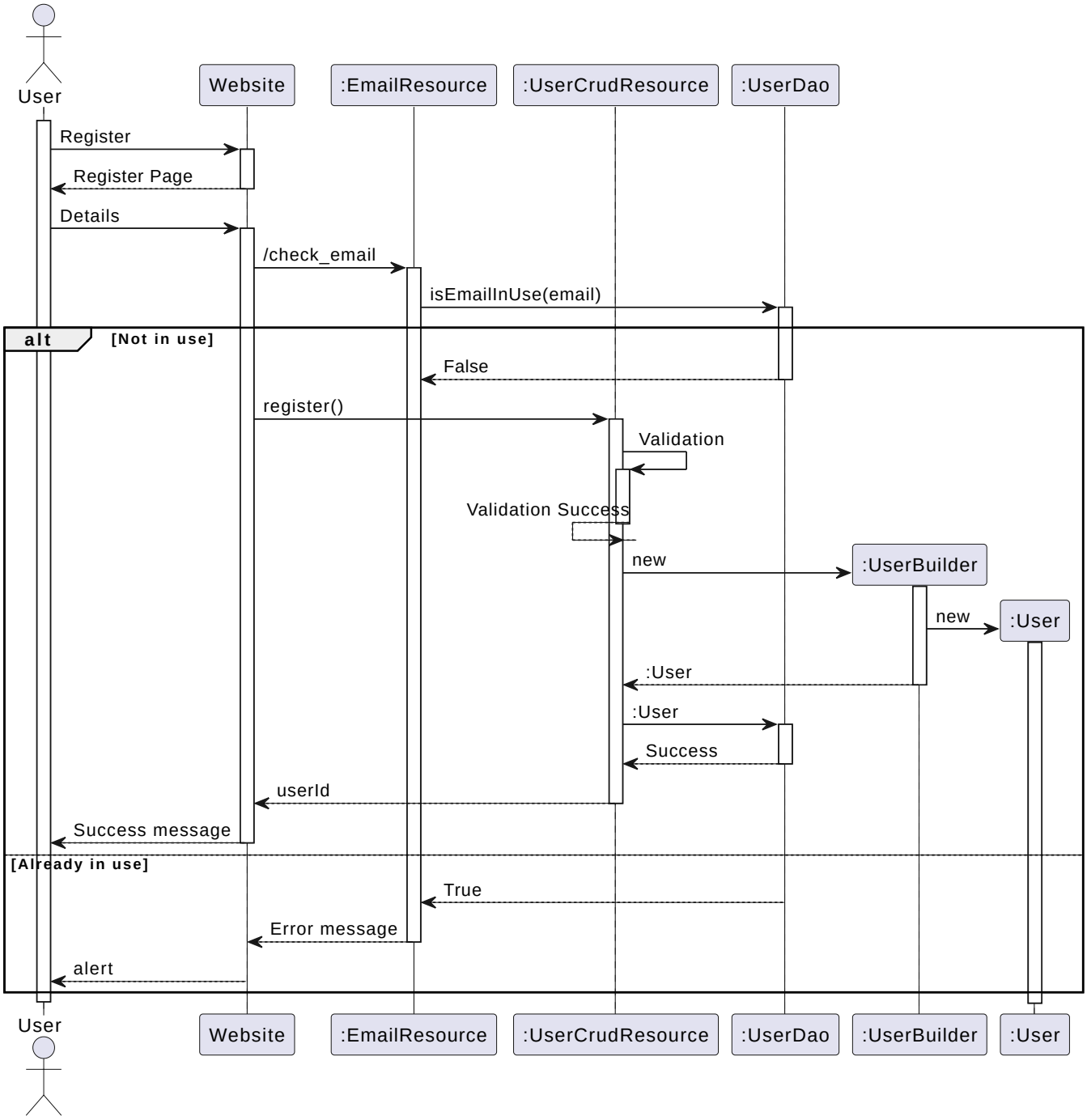
Part 1: Better user management

Existing Flow:



This existing flow requires Admin to act as a mediator between User and the system to create a new User account

New Flow:



New flow completely removes the Admin from the process, there by easing the process for Registration.

Exact Code changes:

1. Adding new function `Boolean isEmailInUse(email)` to check if an email is already in Use.
2. New class `UserBuilder` which helps creation of `User` easier.
3. new Resource `EmailResource` at endpoint `/check_email`.
4. `EmailResource` contains `checkEmail` function at `POST /check_email` that checks if a Email is already used, using `UserDao.isEmailInUse`
5. `UserCrudResource.register` now uses the `UserBuilder` to create instead of directly creating the `User` and setting attributes.
6. Frontend changes:

- new register page
- button in login page

- register controller. which on submit, first `check_email` and then `register`

Design Patterns Used

Builder Pattern

The change introduced in the provided code involves transitioning from a direct instantiation and manual setting of properties in the `User` class to using a builder pattern implemented in the `UserBuilder` class.

Advantages of this refactoring is:

Improved Readability and Conciseness

With the builder pattern, the code for creating a User object is more concise and readable. Instead of multiple setter calls, you have a chain of method calls on the builder, which clearly indicate the properties being set.

Encapsulation

The builder pattern encapsulates the construction process within a separate class (UserBuilder). This separates the construction logic from the User class itself, adhering to the Single Responsibility Principle (SRP). The User class is now only concerned with representing user data, while the construction logic is handled by the builder.

Flexibility and Extendibility

The builder pattern allows for easy addition of optional parameters or variations of User objects without cluttering the User class with numerous constructors or setter methods. You can simply add new methods to the UserBuilder class to support additional properties or configurations.

Defaults and Consistency

The builder pattern allows for setting default values for properties, as seen in the UserBuilder constructor where default values for roleId, localeId, etc., are provided. This ensures consistency in object creation and reduces the likelihood of errors due to missing or incorrect property initialization.

Overall, the transition to the builder pattern offers several advantages including improved readability, encapsulation, flexibility. It enhances maintainability by separating concerns and facilitating future extensions or modifications to the User class. Additionally, it promotes a more fluent and expressive style of object creation, which can lead to cleaner and more maintainable codebases.

Part 2

CommonBook Entity Description

Overview

To create the CommonBook entity, we have designed a model that encompasses essential fields such as its unique identifier, book ID referencing the Book class, and additional entities for related data. These include:

- **CommonBookGenre**: Contains fields such as its unique ID, common book ID referencing the CommonBook class, and genre representing the genre(s) of the common book.
- **CommonBookRating**: Includes fields such as its unique ID, common book ID referencing the CommonBook class, user ID referencing the User class who provided the rating, and the rating value given to the common book.

Book Details

Each book in the library is characterized by the following information:

- Title
- Author
- Genre(s) (supporting multiple genres per book)
- Rating (numerical value, calculated and averaged from user ratings)

Adding Books

Books can be added to the library in two ways:

1. **By ISBN**: Utilizes the ISBN to retrieve book details automatically.
2. **Manual Entry**: Allows users to manually enter details such as title, author, ISBN-10, ISBN-13, etc., to create a book entry.

Rating Books

Users can rate existing books on a defined scale of 1-10. The process involves:

- Selecting a numerical rating from 1 to 10.
- Submitting the rating to record the user's feedback. Subsequent ratings by the same user overwrite previous ratings.

Viewing All Books

The library's complete book collection can be viewed by accessing the "Display Books" section within the Common Library via the navigation bar.

Book Ranking

A list displays the top 10 books based on two criteria chosen by the user:

- Average Rating
- Number of Ratings

Filtering

Users have the ability to filter displayed books based on various criteria:

- **Author(s)**: Supports multi-select functionality; users can supply comma-separated author names.
- **Genre(s)**: Also supports multi-select functionality; users can supply comma-separated genre names.
- **Ratings**: Single-select options available (>6, >7, >8, >9) for filtering books based on rating thresholds.

Details of Implementation and Changes to Existing Code

`CommonBook.java` (com.sismics.books.core.model.jpa)

```
@Entity
@Table(name = "T_COMMON_BOOK")
public class CommonBook {
    @Id
    @Column(name = "COMMON_BOOK_ID", length = 36)
    private String id;

    @Column(name = "BOOK_ID", length = 36)
    private String bookId;

    @Column(name = "CREATE_DATE")
    private Date createDate;

    @Column(name = "DELETE_DATE")
    private Date deleteDate;

    @Column(name = "READ_DATE")
    private Date readDate;

    public String getId() {
        return id;
    }

    // getters and setters
}
```

`CommonBookGenre.java` (com.sismics.books.core.model.jpa)

```
@Entity
@Table(name = "common_book_genre")
public class CommonBookGenre {
    @Id
```

```
    @Column(name = "genre_id",length=36)
    private String id;

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

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

    public CommonBookGenre() {
    }

    public CommonBookGenre(String commonBookId, String genre) {
        this.commonBookId = commonBookId;
        this.genre = genre;
    }

    // getters and setters
}
```

`CommonBookRating.java` (com.sismics.books.core.model.jpa)

```
@Entity
@Table(name = "common_book_rating")
public class CommonBookRating {
    @Id
    @Column(name = "rating_id",length=36)
    private String id;

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

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

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

    public CommonBookRating() {
    }

    public CommonBookRating(String commonBookId, String userId, int rating)
{
        this.commonBookId = commonBookId;
        this.userId = userId;
        this.rating = rating;
    }

    // Getters and setters
}
```

`CommonBookDao.java` (com.sismics.books.core.dao.jpa)

```
public class CommonBookDao {

    private static final Logger logger =
Logger.getLogger(CommonBookDao.class.getName());
    public String create(CommonBook commonBook) {
        commonBook.setId(UUID.randomUUID().toString());
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        em.persist(commonBook);
        return commonBook.getId();
    }
    public CommonBook getById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        try {
            return em.find(CommonBook.class, id);
        } catch (NoResultException e) {
            return null;
        }
    }
    public void deleteById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        CommonBook commonBook = getById(id);
        if (commonBook != null) {
            Date dateNow = new Date();
            commonBook.setDeleteDate(dateNow);
            em.remove(commonBook);
        }
    }

    public CommonBook getByIsbn(String isbn) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query q = em
                .createQuery("SELECT cb FROM CommonBook cb JOIN cb.book b
WHERE b.isbn10 = :isbn OR b.isbn13 = :isbn");
        q.setParameter("isbn", isbn);
        try {
            return (CommonBook) q.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }

    public CommonBook getByBookId(String bookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cb FROM CommonBook cb WHERE
cb.bookId = :bookId");
        query.setParameter("bookId", bookId);
        try {
            return (CommonBook) query.getSingleResult();
        } catch (NoResultException e) {
            return null;
```

```
        }
    }

    public List<CommonBook> getAllCommonBooks() {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cb FROM CommonBook cb");
        return query.getResultList();
    }

    public void findByCriteria(PaginatedList<CommonBookDto> paginatedList,
CommonBookCriteria criteria,
            SortCriteria sortCriteria) throws Exception {
        Map<String, Object> parameterMap = new HashMap<String, Object>();
        List<String> criteriaList = new ArrayList<String>();

        StringBuilder sb = new StringBuilder(
                "select cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE
c2, cb.READ_DATE c3");
        sb.append(" from T_BOOK b ");
        sb.append(" join T_COMMON_BOOK cb on cb.BOOK_ID = b.BOK_ID_C and
cb.DELETE_DATE is null ");
        if (!Strings.isNullOrEmpty(criteria.getSearch())) {
            criteriaList.add(
                    " (b.BOK_TITLE_C like :search or b.BOK_SUBTITLE_C like
:search or b.BOK_AUTHOR_C like :search) ");
            parameterMap.put("search", "%" + criteria.getSearch() + "%");
        }
        if (criteria.getRead() != null) {
            criteriaList.add(" cb.READ_DATE is " + (criteria.getRead() ?
"not" : "") + " null ");
        }
        if (!criteriaList.isEmpty()) {
            sb.append(" where ");
            sb.append(Joiner.on(" and ").join(criteriaList));
        }
        QueryParam queryParam = new QueryParam(sb.toString(),
parameterMap);
        logger.info("Not this");
        List<Object[]> l =
PaginatedLists.executePaginatedQuery(paginatedList, queryParam,
sortCriteria);
        logger.info("Hey No error at all here!!");
        List<CommonBookDto> commonBookDtoList = new
ArrayList<CommonBookDto>();
        for (Object[] o : l) {
            int i = 0;
            CommonBookDto commonBookDto = new CommonBookDto();
            commonBookDto.setId((String) o[i++]);
            commonBookDto.setBookId((String) o[i++]);
            commonBookDto.setCreateDate((Date) o[i++]);
            commonBookDto.setReadDate((Date) o[i++]);
            logger.info("Book ID: " + commonBookDto.getBookId());
            logger.info("ID: " + commonBookDto.getId());
            logger.info("Create Date: " + commonBookDto.getCreateDate());
```

```
            commonBookDtoList.add(commonBookDto);
        }
        paginatedList.setResultList(commonBookDtoList);
    }

    public void findByRatingCriteria(PaginatedList<CommonBookDto>
paginatedList, CommonBookCriteria criteria,
            SortCriteria sortCriteria, String sortBy) throws Exception {
        Map<String, Object> parameterMap = new HashMap<String, Object>();
        List<String> criteriaList = new ArrayList<String>();
        StringBuilder sb = new StringBuilder(
                "select cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE
c2 , cb.READ_DATE c3");
        sb.append(
                ", (SELECT AVG(Cast(cr.RATING as Float)) FROM
COMMON_BOOK_RATING cr WHERE cr.COMMON_BOOK_ID = cb.COMMON_BOOK_ID) c4"); //
Average

// rating
        sb.append(", (SELECT COUNT(*) FROM COMMON_BOOK_RATING cr WHERE
cr.COMMON_BOOK_ID = cb.COMMON_BOOK_ID) c5"); // Count

// of

// ratings
        sb.append(" from T_BOOK b ");
        sb.append(" join T_COMMON_BOOK cb on cb.BOOK_ID = b.BOK_ID_C and
cb.DELETE_DATE is null ");
        QueryParam queryParam = new QueryParam(sb.toString(),
parameterMap);
        List<Object[]> resultList =
PaginatedLists.executePaginatedQuery(paginatedList, queryParam,
sortCriteria);
        List<CommonBookDto> commonBookDtoList = new
ArrayList<CommonBookDto>();
        for (Object[] o : resultList) {
            int i = 0;
            CommonBookDto commonBookDto = new CommonBookDto();
            commonBookDto.setId((String) o[i++]);
            commonBookDto.setBookId((String) o[i++]);
            commonBookDto.setCreateDate((Date) o[i++]);
            commonBookDto.setReadDate((Date) o[i++]);
            commonBookDtoList.add(commonBookDto);
        }
        paginatedList.setResultList(commonBookDtoList);
    }

    public void findByFilteredCriteria(PaginatedList<CommonBookDto>
paginatedList, CommonBookCriteria criteria,
            SortCriteria sortCriteria, String sortBy, List<String> authors,
List<String> genres, Integer ratingFilter)
            throws Exception {
        logger.info("entered CommonBOokDao");
        Map<String, Object> parameterMap = new HashMap<String, Object>();
```

```
        List<String> criteriaList = new ArrayList<String>();
        StringBuilder sb = new StringBuilder(
                "SELECT cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE
c2 , cb.READ_DATE c3, b.BOK_AUTHOR_C c4");
        sb.append(
                ", (SELECT AVG(CAST(cr.RATING as FLOAT)) FROM
COMMON_BOOK_RATING cr WHERE cr.COMMON_BOOK_ID = cb.COMMON_BOOK_ID) c5");
        sb.append(
                ", (SELECT GROUP_CONCAT(genre SEPARATOR ',') FROM
common_book_genre WHERE common_book_id = cb.COMMON_BOOK_ID) AS c6");
        sb.append(" FROM T_BOOK b ");
        sb.append(" JOIN T_COMMON_BOOK cb ON cb.BOOK_ID = b.BOK_ID_C AND
cb.DELETE_DATE IS NULL ");
        QueryParam queryParam = new QueryParam(sb.toString(),
parameterMap);
        List<Object[]> resultList =
PaginatedLists.executePaginatedQuery(paginatedList, queryParam,
sortCriteria);
        List<CommonBookDto> commonBookDtoList = new
ArrayList<CommonBookDto>();
        for (Object[] o : resultList) {
            int i = 0;
            CommonBookDto commonBookDto = new CommonBookDto();
            commonBookDto.setId((String) o[i++]);
            commonBookDto.setBookId((String) o[i++]);
            commonBookDto.setCreateDate((Date) o[i++]);
            commonBookDto.setReadDate((Date) o[i++]);
            String author = (String) o[i++];
            logger.info(author);
            Double rating = (Double) o[i++];
            String genreString = (String) o[i++];
            logger.info(genreString);
            List<String> genresList =
Arrays.asList(genreString.split(","));
            boolean authorFound = authors == null;
            if(!authorFound){
                logger.info("authors is empty");
            }
            else{
                logger.info("authors is not empty");
            }
            boolean genreFound = genres == null;
            if (!authorFound && authors.contains(author)) {
                logger.info("Contains Author");
                authorFound = true;
            }
            if (!genreFound) {
                for (String genre : genresList) {
                    if (genres.contains(genre)) {
                        logger.info("Found genre");
                        genreFound = true;
                        break;
                    }
                }
```

```
            }
            else{
                logger.info("genreFound is true");
            }
            boolean ratingPass = ratingFilter == null || (rating != null &&
rating > ratingFilter);
            if(ratingPass){
                logger.info("greater rating than ratingFIlter");
            }
            if (authorFound && genreFound && ratingPass) {
                logger.info("added something to dto");
                commonBookDtoList.add(commonBookDto);
            }
        }
        paginatedList.setResultList(commonBookDtoList);
    }
}
```

CommonBookGenreDao.java (com.sismics.books.core.dao.jpa)

```
public class CommonBookGenreDao {

    private static final Logger logger =
Logger.getLogger(CommonBookGenreDao.class.getName());

    public String create(CommonBookGenre commonBookGenre) {
        commonBookGenre.setId(UUID.randomUUID().toString());
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        em.persist(commonBookGenre);
        return commonBookGenre.getId();
    }

    public CommonBookGenre getById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        try {
            return em.find(CommonBookGenre.class, id);
        } catch (NoResultException e) {
            return null;
        }
    }

    public void deleteById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        CommonBookGenre commonBookGenre = getById(id);
        if (commonBookGenre != null) {
            em.remove(commonBookGenre);
        }
    }

    public CommonBookGenre getByCommonBookIdAndGenre(String commonBookId,
String genre) {
```

```
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbg FROM CommonBookGenre cbg
WHERE cbg.commonBookId = :commonBookId AND cbg.genre = :genre");
        query.setParameter("commonBookId", commonBookId);
        query.setParameter("genre", genre);
        try {
            return (CommonBookGenre) query.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }

    public List<CommonBookGenre> getByCommonBookId(String commonBookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbg FROM CommonBookGenre cbg
WHERE cbg.commonBookId = :commonBookId");
        query.setParameter("commonBookId", commonBookId);
        return query.getResultList();
    }

    public List<String> getGenresByCommonBookId(String commonBookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbg.genre FROM CommonBookGenre
cbg WHERE cbg.commonBookId = :commonBookId");
        query.setParameter("commonBookId", commonBookId);
        return query.getResultList();
    }
}
```

`CommonBookRatingDao.java` (com.sismics.books.core.dao.jpa)

```
public class CommonBookRatingDao {

    private static final Logger logger =
Logger.getLogger(CommonBookRatingDao.class.getName());

    public String create(CommonBookRating commonBookRating) {
        commonBookRating.setId(UUID.randomUUID().toString());
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        em.persist(commonBookRating);
        return commonBookRating.getId();
    }

    public CommonBookRating getById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        try {
            return em.find(CommonBookRating.class, id);
        } catch (NoResultException e) {
            return null;
        }
    }
```

```
    public void deleteById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        CommonBookRating commonBookRating = getById(id);
        if (commonBookRating != null) {
            em.remove(commonBookRating);
        }
    }

    public void update(CommonBookRating commonBookRating) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        em.merge(commonBookRating);
    }

    public CommonBookRating getByCommonBookIdAndUserId(String commonBookId,
String userId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbr FROM CommonBookRating cbr
WHERE cbr.commonBookId = :commonBookId AND cbr.userId = :userId");
        query.setParameter("commonBookId", commonBookId);
        query.setParameter("userId", userId);
        try {
            return (CommonBookRating) query.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }

    public List<CommonBookRating> getAllCommonBookRatings() {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbr FROM CommonBookRating
cbr");
        return query.getResultList();
    }

    public List<CommonBookRating> getByCommonBookId(String commonBookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbr FROM CommonBookRating cbr
WHERE cbr.commonBookId = :commonBookId");
        query.setParameter("commonBookId", commonBookId);
        return query.getResultList();
    }

    public List<CommonBookRating> getByUserId(String userId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cbr FROM CommonBookRating cbr
WHERE cbr.userId = :userId");
        query.setParameter("userId", userId);
        return query.getResultList();
    }

    public double getAverageRatingByCommonBookId(String commonBookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT AVG(cbr.rating) FROM
```

```
CommonBookRating cbr WHERE cbr.commonBookId = :commonBookId");
        query.setParameter("commonBookId", commonBookId);
        try {
            Object result = query.getSingleResult();
            if (result != null) {
                return (double) result;
            }
        } catch (NoResultException e) {
            return 0; // Default if no result found
        }
        return 0; // Default if no result found
    }

    public long getNumberOfRatingsByCommonBookId(String commonBookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT COUNT(cbr) FROM
CommonBookRating cbr WHERE cbr.commonBookId = :commonBookId");
        query.setParameter("commonBookId", commonBookId);
        try {
            Object result = query.getSingleResult();
            if (result != null) {
                return (long) result;
            }
        } catch (NoResultException e) {
            return 0; // Default if no result found
        }
        return 0; // Default if no result found
    }

}
```

`CommonBookCriteria.java` (com.sismics.books.core.dao.jpa.criteria)

```
public class CommonBookCriteria {
    private String commonBookId;
    private String search;
    private Boolean read;

    public String getCommonBookId() {
        return commonBookId;
    }

    public void setCommonBookId(String commonBookId) {
        this.commonBookId = commonBookId;
    }

    public String getSearch() {
        return search;
    }

    public void setSearch(String search) {
```

```
        this.search = search;
    }

    public Boolean getRead() {
        return read;
    }

    public void setRead(Boolean read) {
        this.read = read;
    }
}
```

`CommonBookDto.java` (com.sismics.books.core.dto)

```
public class CommonBookDto {
    @Id
    private String id;
    private String bookId;
    private Date createDate;
    private Date deleteDate;
    private Date readDate;

    public CommonBookDto(){

    }

    public CommonBookDto(String id, String bookId, Date createDate, Date
deleteDate, Date readDate) {
        this.id = id;
        this.bookId = bookId;
        this.createDate = createDate;
        this.deleteDate = deleteDate;
        this.readDate = readDate;
    }

    // getters and setters
}
```

`PaginatedLists.java` (com/sismics/books/core/util/jpa)

```
public static <E> List<Object[]> executePaginatedQuery(PaginatedList<E>
paginatedList, QueryParam queryParam,
            SortCriteria sortCriteria) {
        StringBuilder sb = new StringBuilder(queryParam.getQueryString());
        sb.append(" order by c");
        sb.append(sortCriteria.getColumn());
        sb.append(sortCriteria.isAsc() ? " asc" : " desc");
        if (sortCriteria.isAsc())
            sb.append(" NULLS FIRST");
```

```
        else
            sb.append(" NULLS LAST");
        QueryParam sortedQueryParam = new QueryParam(sb.toString(),
queryParam.getParameterMap());
        executeCountQuery(paginatedList, sortedQueryParam);
        return executeResultQuery(paginatedList, sortedQueryParam);
    }
```

`dbupdate-000-0.sql` (src/main/resources/db/update)

```
CREATE cached TABLE T_COMMON_BOOK ( COMMON_BOOK_ID VARCHAR(36) PRIMARY KEY,
BOOK_ID VARCHAR(36), CREATE_DATE datetime, DELETE_DATE datetime, READ_DATE
datetime);
CREATE cached TABLE common_book_genre ( genre_id VARCHAR(36) PRIMARY KEY,
common_book_id VARCHAR(36), genre VARCHAR(255));
CREATE cached TABLE common_book_rating (rating_id VARCHAR(36) PRIMARY KEY,
common_book_id VARCHAR(36), user_id VARCHAR(255), rating INT);
```

`CommonBookResource.java`

```
@Path("/commonbook")
public class CommonBookResource extends BaseResource {

    private static final Logger logger =
Logger.getLogger(CommonBookResource.class.getName());

    @PUT
    @Produces(MediaType.APPLICATION_JSON)
    public Response add(
            @FormParam("isbn") String isbn,
            @FormParam("genre") String genre) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }

        ValidationUtil.validateRequired(isbn, "isbn");
        String[] genreArray = genre.split(",");
        List<String> genreList = Arrays.asList(genreArray);
        logger.info(genre);
        BookDao bookDao = new BookDao();
        Book book = bookDao.getByIsbn(isbn);
        if (book == null){
            try {
                book =
AppContext.getInstance().getBookDataService().searchBook(isbn);
            } catch (Exception e) {
                throw new ClientException("BookNotFound",
e.getCause().getMessage(), e);
```

```
            }
            bookDao.create(book);
        }

        CommonBookDao commonBookDao = new CommonBookDao();
        CommonBook commonBook = commonBookDao.getByBookId(book.getId());
        if (commonBook == null) {
            commonBook = new CommonBook();
            commonBook.setBookId(book.getId());
            commonBook.setCreateDate(new Date());
            commonBookDao.create(commonBook);
        } else {
            throw new ClientException("BookAlreadyAdded", "Book already
added");
        }

        CommonBookGenreDao commonBookGenreDao = new CommonBookGenreDao();
        for (String genreName : genreList) {
            CommonBookGenre commonBookGenre =
commonBookGenreDao.getByCommonBookIdAndGenre(commonBook.getId(),
genreName);
            if (commonBookGenre == null) {
                commonBookGenre = new CommonBookGenre();
                commonBookGenre.setCommonBookId(commonBook.getId());
                logger.info(book.getId()+"
"+commonBookGenre.getCommonBookId());
                commonBookGenre.setGenre(genreName);
                commonBookGenreDao.create(commonBookGenre);
            }
        }

        JSONObject response = new JSONObject();
        response.put("id", commonBook.getId());
        return Response.ok().entity(response).build();
    }

    @PUT
    @Path("manual")
    @Produces(MediaType.APPLICATION_JSON)
    public Response add(
            @FormParam("title") String title,
            @FormParam("subtitle") String subtitle,
            @FormParam("author") String author,
            @FormParam("description") String description,
            @FormParam("isbn10") String isbn10,
            @FormParam("isbn13") String isbn13,
            @FormParam("page_count") Long pageCount,
            @FormParam("language") String language,
            @FormParam("publish_date") String publishDateStr,
            @FormParam("genre") String genre) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        String[] genreArray = genre.split(",");
```

```
        List<String> genreList = Arrays.asList(genreArray);
        title = ValidationUtil.validateLength(title, "title", 1, 255,
false);
        subtitle = ValidationUtil.validateLength(subtitle, "subtitle", 1,
255, true);
        author = ValidationUtil.validateLength(author, "author", 1, 255,
false);
        description = ValidationUtil.validateLength(description,
"description", 1, 4000, true);
        isbn10 = ValidationUtil.validateLength(isbn10, "isbn10", 10, 10,
true);
        isbn13 = ValidationUtil.validateLength(isbn13, "isbn13", 13, 13,
true);
        language = ValidationUtil.validateLength(language, "language", 2,
2, true);
        Date publishDate = ValidationUtil.validateDate(publishDateStr,
"publish_date", false);
        if (Strings.isNullOrEmpty(isbn10) && Strings.isNullOrEmpty(isbn13))
{
            throw new ClientException("ValidationError", "At least one ISBN
number is mandatory");
        }
        BookDao bookDao = new BookDao();
        Book bookIsbn10 = bookDao.getByIsbn(isbn10);
        Book bookIsbn13 = bookDao.getByIsbn(isbn13);
        if (bookIsbn10 != null || bookIsbn13 != null) {
            throw new ClientException("BookAlreadyAdded", "Book already
added");
        }
        Book book = new Book();
        book.setId(UUID.randomUUID().toString());

        if (title != null) {
            book.setTitle(title);
        }
        if (subtitle != null) {
            book.setSubtitle(subtitle);
        }
        if (author != null) {
            book.setAuthor(author);
        }
        if (description != null) {
            book.setDescription(description);
        }
        if (isbn10 != null) {
            book.setIsbn10(isbn10);
        }
        if (isbn13 != null) {
            book.setIsbn13(isbn13);
        }
        if (pageCount != null) {
            book.setPageCount(pageCount);
        }
        if (language != null) {
```

```
            book.setLanguage(language);
        }
        if (publishDate != null) {
            book.setPublishDate(publishDate);
        }
        bookDao.create(book);

        CommonBookDao commonBookDao = new CommonBookDao();
        CommonBook commonBook = commonBookDao.getByBookId(book.getId());
        if (commonBook == null) {
            commonBook = new CommonBook();
            commonBook.setBookId(book.getId());
            commonBook.setCreateDate(new Date());
            commonBookDao.create(commonBook);
        } else {
            throw new ClientException("BookAlreadyAdded", "Book already
added");
        }

        CommonBookGenreDao commonBookGenreDao = new CommonBookGenreDao();
        for (String genreName : genreList) {
            CommonBookGenre commonBookGenre =
commonBookGenreDao.getByCommonBookIdAndGenre(book.getId(), genreName);
            if (commonBookGenre == null) {
                commonBookGenre = new CommonBookGenre();
                commonBookGenre.setCommonBookId(commonBook.getId());
                commonBookGenre.setGenre(genreName);
                commonBookGenreDao.create(commonBookGenre);
            }
        }

        JSONObject response = new JSONObject();
        response.put("id", commonBook.getId());
        return Response.ok().entity(response).build();
    }

    @DELETE
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
    public Response delete(
            @PathParam("id") String id) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        CommonBookDao commonBookDao = new CommonBookDao();
        CommonBook commonBook = commonBookDao.getById(id);
        if (commonBook == null) {
            throw new ClientException("BookNotFound", "Book not found");
        }
        commonBookDao.deleteById(commonBook.getId());
        JSONObject response = new JSONObject();
        response.put("status", "ok");
        return Response.ok().entity(response).build();
    }
```

```
    @PUT
    @Path("{id: [a-z0-9\\-]+}/rate")
    @Produces(MediaType.APPLICATION_JSON)
    public Response rate(
            @PathParam("id") String id,
            @FormParam("rating") Integer rating) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        if (rating < 1 || rating > 10) {
            throw new ClientException("ValidationError", "Rating must be
between 1 and 10");
        }
        CommonBookDao commonBookDao = new CommonBookDao();
        CommonBook commonBook = commonBookDao.getById(id);
        if (commonBook == null) {
            throw new ClientException("BookNotFound", "Book not found");
        }
        CommonBookRatingDao commonBookRatingDao = new
CommonBookRatingDao();
        CommonBookRating commonBookRating =
commonBookRatingDao.getByCommonBookIdAndUserId(id, principal.getId());
        if (commonBookRating == null) {
            commonBookRating = new CommonBookRating();
            commonBookRating.setCommonBookId(id);
            commonBookRating.setUserId(principal.getId());
            commonBookRating.setRating(rating);
            commonBookRatingDao.create(commonBookRating);
        } else {
            commonBookRating.setRating(rating);
            commonBookRatingDao.update(commonBookRating);
        }
        JSONObject response = new JSONObject();
        response.put("id", commonBookRating.getId());
        return Response.ok().entity(response).build();
    }

    @GET
    @Path("list")
    @Produces(MediaType.APPLICATION_JSON)
    public Response list(
            @QueryParam("limit") Integer limit,
            @QueryParam("offset") Integer offset,
            @QueryParam("sort_column") Integer sortColumn,
            @QueryParam("asc") Boolean asc,
            @QueryParam("search") String search,
            @QueryParam("read") Boolean read) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        JSONObject response = new JSONObject();
        List<JSONObject> books = new ArrayList<>();
        CommonBookDao commonBookDao = new CommonBookDao();
```

```
        PaginatedList<CommonBookDto> paginatedList =
PaginatedLists.create(limit, offset);
        SortCriteria sortCriteria = new SortCriteria(sortColumn, asc);
        CommonBookCriteria criteria = new CommonBookCriteria();
        criteria.setSearch(search);
        criteria.setRead(read);
        try {
            commonBookDao.findByCriteria(paginatedList, criteria,
sortCriteria);
        } catch (Exception e) {
            throw new ServerException("SearchError", "Error searching in
books", e);
        }
        CommonBookRatingDao commonBookRatingDao = new
CommonBookRatingDao();
        CommonBookGenreDao commonBookGenreDao = new CommonBookGenreDao();
        BookDao bookDao = new BookDao();
        for (CommonBookDto commonBookDto : paginatedList.getResultList()) {
            Book book = bookDao.getById(commonBookDto.getBookId());
            double avgRating =
commonBookRatingDao.getAverageRatingByCommonBookId(commonBookDto.getId());
            long numRatings =
commonBookRatingDao.getNumberOfRatingsByCommonBookId(commonBookDto.getId())
;
            List<String> GenreList =
commonBookGenreDao.getGenresByCommonBookId(commonBookDto.getId());
            String genreString = String.join(",", GenreList);
            JSONObject bookJson = new JSONObject();
            bookJson.put("id", commonBookDto.getId());
            bookJson.put("createDate", commonBookDto.getCreateDate());
            bookJson.put("bookId", commonBookDto.getBookId());
            bookJson.put("book", book);
            bookJson.put("author", book.getAuthor());
            bookJson.put("title", book.getTitle());
            bookJson.put("subtitle", book.getSubtitle());
            bookJson.put("description", book.getDescription());
            bookJson.put("publishTime", book.getPublishDate());
            bookJson.put("pageCount", book.getPageCount());
            bookJson.put("language", book.getLanguage());
            bookJson.put("isbn10", book.getIsbn10());
            bookJson.put("isbn13", book.getIsbn13());
            bookJson.put("avgRating", avgRating);
            bookJson.put("genre",genreString);
            bookJson.put("numRatings", numRatings);
            books.add(bookJson);
        }
        response.put("total", paginatedList.getResultCount());
        response.put("books", books);
        return Response.ok().entity(response).build();
    }

    @GET
    @Path("{id: [a-z0-9\\-]+}")
    @Produces(MediaType.APPLICATION_JSON)
```

```
    public Response get(
            @PathParam("id") String id) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        CommonBookDao commonBookDao = new CommonBookDao();
        CommonBook commonBook = commonBookDao.getById(id);
        if (commonBook == null) {
            throw new ClientException("BookNotFound", "Book not found");
        }
        BookDao bookDao = new BookDao();
        Book book = bookDao.getById(commonBook.getBookId());
        CommonBookRatingDao commonBookRatingDao = new
CommonBookRatingDao();
        CommonBookGenreDao commonBookGenreDao = new CommonBookGenreDao();
        double avgRating =
commonBookRatingDao.getAverageRatingByCommonBookId(id);
        List<String> GenreList =
commonBookGenreDao.getGenresByCommonBookId(commonBook.getId());
        long numRatings =
commonBookRatingDao.getNumberOfRatingsByCommonBookId(id);
        String genreString = String.join(",", GenreList);
        JSONObject response = new JSONObject();
        response.put("id", commonBook.getId());
        response.put("createDate", commonBook.getCreateDate());
        response.put("bookId", commonBook.getBookId());
        response.put("book", book);
        response.put("author", book.getAuthor());
        response.put("title", book.getTitle());
        response.put("subtitle", book.getSubtitle());
        response.put("description", book.getDescription());
        response.put("publishTime", book.getPublishDate());
        response.put("pageCount", book.getPageCount());
        response.put("language", book.getLanguage());
        response.put("isbn10", book.getIsbn10());
        response.put("isbn13", book.getIsbn13());
        response.put("avgRating", avgRating);
        response.put("genre", genreString);
        response.put("numRatings", numRatings);
        return Response.ok().entity(response).build();
    }


    @GET
    @Path("list/genre")
    @Produces(MediaType.APPLICATION_JSON)
    public Response listGenre(
            @QueryParam("limit") Integer limit,
            @QueryParam("offset") Integer offset,
            @QueryParam("sort_column") Integer sortColumn,
            @QueryParam("asc") Boolean asc,
            @QueryParam("search") String search,
            @QueryParam("read") Boolean read,
            @QueryParam("genre") String genre,
```

```
            @QueryParam("author") String author,
            @QueryParam("rating") Integer rating) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        String[] genreArray = null;
        List<String> genrelist = null;
        if(genre != null){
            genreArray = genre.split(",");
            genrelist = Arrays.asList(genreArray);
        }

        String[] authorArray = null;
        List<String> authorlist = null;
        if(author != null){
            authorArray = author.split(",");
            authorlist = Arrays.asList(authorArray);
        }
        JSONObject response = new JSONObject();
        List<JSONObject> filters = new ArrayList<>();
        CommonBookDao commonBookDao = new CommonBookDao();
        PaginatedList<CommonBookDto> paginatedList =
PaginatedLists.create(limit, offset);
        SortCriteria sortCriteria = new SortCriteria(sortColumn, asc);
        CommonBookCriteria criteria = new CommonBookCriteria();
        criteria.setSearch(search);
        criteria.setRead(read);
        try {
            commonBookDao.findByFilteredCriteria(paginatedList, criteria,
sortCriteria, "title", authorlist, genrelist,
                    rating);
        } catch (Exception e) {
            throw new ServerException("SearchError", "Error searching in
books", e);
        }
        CommonBookRatingDao commonBookRatingDao = new
CommonBookRatingDao();
        CommonBookGenreDao commonBookGenreDao = new CommonBookGenreDao();
        BookDao bookDao = new BookDao();
        for (CommonBookDto commonBookDto : paginatedList.getResultList()) {
            Book book = bookDao.getById(commonBookDto.getBookId());
            double avgRating =
commonBookRatingDao.getAverageRatingByCommonBookId(commonBookDto.getId());
            long numRatings =
commonBookRatingDao.getNumberOfRatingsByCommonBookId(commonBookDto.getId())
;
            List<String> GenreList =
commonBookGenreDao.getGenresByCommonBookId(commonBookDto.getId());
            String genreString = String.join(",",GenreList);
            JSONObject bookJson = new JSONObject();
            bookJson.put("id", commonBookDto.getId());
            bookJson.put("createDate", commonBookDto.getCreateDate());
            bookJson.put("bookId", commonBookDto.getBookId());
            bookJson.put("book", book);
```

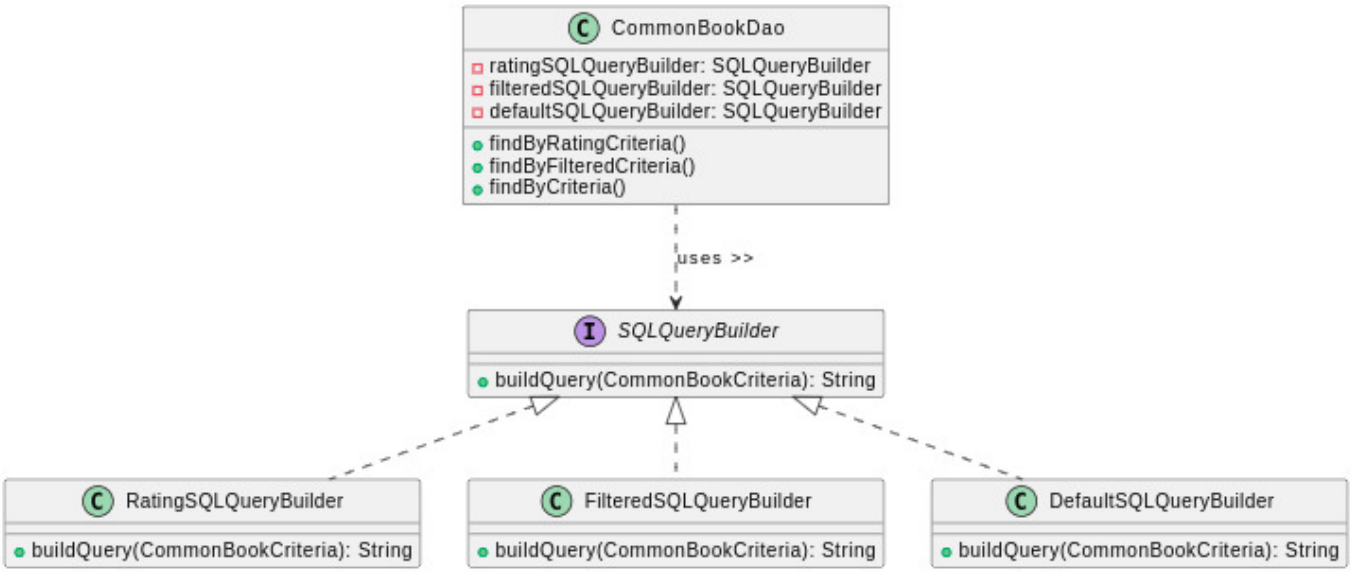
```
            bookJson.put("author", book.getAuthor());
            bookJson.put("title", book.getTitle());
            bookJson.put("subtitle", book.getSubtitle());
            bookJson.put("description", book.getDescription());
            bookJson.put("publishTime", book.getPublishDate());
            bookJson.put("pageCount", book.getPageCount());
            bookJson.put("language", book.getLanguage());
            bookJson.put("isbn10", book.getIsbn10());
            bookJson.put("isbn13", book.getIsbn13());
            bookJson.put("avgRating", avgRating);
            bookJson.put("numRatings", numRatings);
            bookJson.put("genre",genreString);
            filters.add(bookJson);
        }
        response.put("total", paginatedList.getResultCount());
        response.put("books", filters);
        return Response.ok().entity(response).build();
    }

    @GET
    @Path("list/toprated")
    @Produces(MediaType.APPLICATION_JSON)
    public Response listRating(
            @QueryParam("limit") Integer limit,
            @QueryParam("offset") Integer offset,
            @QueryParam("sort_column") Integer sortColumn,
            @QueryParam("asc") Boolean asc,
            @QueryParam("search") String search,
            @QueryParam("read") Boolean read,
            @QueryParam("sortBy") String sortBy) throws JSONException {
        if (!authenticate()) {
            throw new ForbiddenClientException();
        }
        JSONObject response = new JSONObject();
        List<JSONObject> filters = new ArrayList<>();
        CommonBookDao commonBookDao = new CommonBookDao();
        PaginatedList<CommonBookDto> paginatedList =
PaginatedLists.create(limit, offset);
        sortColumn = 4;
        if("numRating".equalsIgnoreCase(sortBy)){
            sortColumn = 5;
        }
        SortCriteria sortCriteria = new SortCriteria(sortColumn, asc);
        CommonBookCriteria criteria = new CommonBookCriteria();
        criteria.setSearch(search);
        criteria.setRead(read);
        try {
            commonBookDao.findByRatingCriteria(paginatedList, criteria,
sortCriteria, sortBy);
        } catch (Exception e) {
            throw new ServerException("SearchError", "Error searching in
books", e);
        }
        CommonBookRatingDao commonBookRatingDao = new
```

```
CommonBookRatingDao();
        BookDao bookDao = new BookDao();
        for (CommonBookDto commonBookDto : paginatedList.getResultList()) {
            Book book = bookDao.getById(commonBookDto.getBookId());
            double avgRating =
commonBookRatingDao.getAverageRatingByCommonBookId(commonBookDto.getId());
            long numRatings =
commonBookRatingDao.getNumberOfRatingsByCommonBookId(commonBookDto.getId())
;
            JSONObject bookJson = new JSONObject();
            bookJson.put("id", commonBookDto.getId());
            bookJson.put("createDate", commonBookDto.getCreateDate());
            bookJson.put("bookId", commonBookDto.getBookId());
            bookJson.put("book", book);
            bookJson.put("author", book.getAuthor());
            bookJson.put("title", book.getTitle());
            bookJson.put("subtitle", book.getSubtitle());
            bookJson.put("description", book.getDescription());
            bookJson.put("publishTime", book.getPublishDate());
            bookJson.put("pageCount", book.getPageCount());
            bookJson.put("language", book.getLanguage());
            bookJson.put("isbn10", book.getIsbn10());
            bookJson.put("isbn13", book.getIsbn13());
            bookJson.put("avgRating", avgRating);
            bookJson.put("numRatings", numRatings);

            filters.add(bookJson);
        }
        response.put("total", paginatedList.getResultCount());
        response.put("books", filters);
        return Response.ok().entity(response).build();
    }
}
```

Strategy Pattern in CommonBookDao



`SQLQueryBuilder.java` (com.sismics.books.core.dao.jpa)

```
package com.sismics.books.core.dao.jpa;
import com.sismics.books.core.dao.jpa.criteria.CommonBookCriteria;
public interface SQLQueryBuilder {
    String buildQuery(CommonBookCriteria criteria);
}
```

`DefaultSQLQueryBuilder.java` (com.sismics.books.core.dao.jpa)

```
public class DefaultSQLQueryBuilder implements SQLQueryBuilder {
    @Override
    public String buildQuery(CommonBookCriteria criteria) {
        StringBuilder sb = new StringBuilder(
            "SELECT cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE c2,
cb.READ_DATE c3");
        sb.append(" FROM T_BOOK b ");
        sb.append(" JOIN T_COMMON_BOOK cb ON cb.BOOK_ID = b.BOK_ID_C AND
cb.DELETE_DATE IS NULL ");

        if (!Strings.isNullOrEmpty(criteria.getSearch())) {
            sb.append(" WHERE (b.BOK_TITLE_C LIKE :search OR
b.BOK_SUBTITLE_C LIKE :search OR b.BOK_AUTHOR_C LIKE :search) ");
        }
        if (criteria.getRead() != null) {
            sb.append(" AND cb.READ_DATE IS " + (criteria.getRead() ? "NOT"
: "") + " NULL ");
        }
        return sb.toString();
    }
}
```

`FilteredSQLQueryBuilder.java` (com.sismics.books.core.dao.jpa)

```
public class FilteredSQLQueryBuilder implements SQLQueryBuilder {
    @Override
    public String buildQuery(CommonBookCriteria criteria) {
        StringBuilder sb = new StringBuilder(
            "SELECT cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE c2
, cb.READ_DATE c3, b.BOK_AUTHOR_C c4");
        sb.append(", (SELECT AVG(CAST(cr.RATING AS FLOAT)) FROM
COMMON_BOOK_RATING cr WHERE cr.COMMON_BOOK_ID = cb.COMMON_BOOK_ID) c5");
        sb.append(", (SELECT GROUP_CONCAT(genre SEPARATOR ',') FROM
common_book_genre WHERE common_book_id = cb.COMMON_BOOK_ID) AS c6");
        sb.append(" FROM T_BOOK b ");
        sb.append(" JOIN T_COMMON_BOOK cb ON cb.BOOK_ID = b.BOK_ID_C AND
cb.DELETE_DATE IS NULL ");
        return sb.toString();
```

```
    }
}
```

`RatingSQLQueryBuilder.java` (com.sismics.books.core.dao.jpa)

```
public class RatingSQLQueryBuilder implements SQLQueryBuilder {
    @Override
    public String buildQuery(CommonBookCriteria criteria) {
        StringBuilder sb = new StringBuilder(
            "SELECT cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE c2
, cb.READ_DATE c3");
        sb.append(", (SELECT AVG(CAST(cr.RATING AS FLOAT)) FROM
COMMON_BOOK_RATING cr WHERE cr.COMMON_BOOK_ID = cb.COMMON_BOOK_ID) c4");
        sb.append(", (SELECT COUNT(*) FROM COMMON_BOOK_RATING cr WHERE
cr.COMMON_BOOK_ID = cb.COMMON_BOOK_ID) c5");
        sb.append(" FROM T_BOOK b ");
        sb.append(" JOIN T_COMMON_BOOK cb ON cb.BOOK_ID = b.BOK_ID_C AND
cb.DELETE_DATE IS NULL ");
        return sb.toString();
    }
}
```

`CommonBookDao.java` (com.sismics.books.core.dao.jpa)

```
public class CommonBookDao {

    private static final Logger logger =
Logger.getLogger(CommonBookDao.class.getName());
    public String create(CommonBook commonBook) {
        commonBook.setId(UUID.randomUUID().toString());
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        em.persist(commonBook);
        return commonBook.getId();
    }
    public CommonBook getById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        try {
            return em.find(CommonBook.class, id);
        } catch (NoResultException e) {
            return null;
        }
    }
    public void deleteById(String id) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        CommonBook commonBook = getById(id);
        if (commonBook != null) {
            Date dateNow = new Date();
            commonBook.setDeleteDate(dateNow);
            em.remove(commonBook);
```

```
        }
    }

    public CommonBook getByIsbn(String isbn) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query q = em
                .createQuery("SELECT cb FROM CommonBook cb JOIN cb.book b
WHERE b.isbn10 = :isbn OR b.isbn13 = :isbn");
        q.setParameter("isbn", isbn);
        try {
            return (CommonBook) q.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }

    public CommonBook getByBookId(String bookId) {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cb FROM CommonBook cb WHERE
cb.bookId = :bookId");
        query.setParameter("bookId", bookId);
        try {
            return (CommonBook) query.getSingleResult();
        } catch (NoResultException e) {
            return null;
        }
    }

    public List<CommonBook> getAllCommonBooks() {
        EntityManager em = ThreadLocalContext.get().getEntityManager();
        Query query = em.createQuery("SELECT cb FROM CommonBook cb");
        return query.getResultList();
    }

    public void findByCriteria(PaginatedList<CommonBookDto> paginatedList,
CommonBookCriteria criteria,
            SortCriteria sortCriteria) throws Exception {
        Map<String, Object> parameterMap = new HashMap<String, Object>();
        List<String> criteriaList = new ArrayList<String>();
        StringBuilder sb = new StringBuilder(
                "select cb.COMMON_BOOK_ID c0, b.BOK_ID_C c1, cb.CREATE_DATE
c2, cb.READ_DATE c3");
        sb.append(" from T_BOOK b ");
        sb.append(" join T_COMMON_BOOK cb on cb.BOOK_ID = b.BOK_ID_C and
cb.DELETE_DATE is null ");
        if (!Strings.isNullOrEmpty(criteria.getSearch())) {
            criteriaList.add(
                    " (b.BOK_TITLE_C like :search or b.BOK_SUBTITLE_C like
:search or b.BOK_AUTHOR_C like :search) ");
            parameterMap.put("search", "%" + criteria.getSearch() + "%");
        }
        if (criteria.getRead() != null) {
            criteriaList.add(" cb.READ_DATE is " + (criteria.getRead() ?
"not" : "") + " null ");
```

```
        }
        if (!criteriaList.isEmpty()) {
            sb.append(" where ");
            sb.append(Joiner.on(" and ").join(criteriaList));
        }
        QueryParam queryParam = new QueryParam(sb.toString(),
parameterMap);
        List<Object[]> l =
PaginatedLists.executePaginatedQuery(paginatedList, queryParam,
sortCriteria);
        List<CommonBookDto> commonBookDtoList = new
ArrayList<CommonBookDto>();
        for (Object[] o : l) {
            int i = 0;
            CommonBookDto commonBookDto = new CommonBookDto();
            commonBookDto.setId((String) o[i++]);
            commonBookDto.setBookId((String) o[i++]);
            commonBookDto.setCreateDate((Date) o[i++]);
            commonBookDto.setReadDate((Date) o[i++]);
            logger.info("Book ID: " + commonBookDto.getBookId());
            logger.info("ID: " + commonBookDto.getId());
            logger.info("Create Date: " + commonBookDto.getCreateDate());
            commonBookDtoList.add(commonBookDto);
        }
        paginatedList.setResultList(commonBookDtoList);
    }

    public void findByRatingCriteria(PaginatedList<CommonBookDto>
paginatedList, CommonBookCriteria criteria,
            SortCriteria sortCriteria, String sortBy) throws Exception {
        Map<String, Object> parameterMap = new HashMap<String, Object>();
        RatingSQLQueryBuilder ratingSQLQueryBuilder = new
RatingSQLQueryBuilder();
        String sqlQuery = ratingSQLQueryBuilder.buildQuery(criteria);
        QueryParam queryParam = new QueryParam(sqlQuery, parameterMap);
        List<Object[]> resultList =
PaginatedLists.executePaginatedQuery(paginatedList, queryParam,
sortCriteria);
        List<CommonBookDto> commonBookDtoList = new
ArrayList<CommonBookDto>();
        for (Object[] o : resultList) {
            int i = 0;
            CommonBookDto commonBookDto = new CommonBookDto();
            commonBookDto.setId((String) o[i++]);
            commonBookDto.setBookId((String) o[i++]);
            commonBookDto.setCreateDate((Date) o[i++]);
            commonBookDto.setReadDate((Date) o[i++]);
            commonBookDtoList.add(commonBookDto);
        }
        paginatedList.setResultList(commonBookDtoList);
    }

    public void findByFilteredCriteria(PaginatedList<CommonBookDto>
paginatedList, CommonBookCriteria criteria,
```

```
            SortCriteria sortCriteria, String sortBy, List<String> authors,
List<String> genres, Integer ratingFilter)
            throws Exception {
        Map<String, Object> parameterMap = new HashMap<String, Object>();
        FilteredSQLQueryBuilder filteredSQLQueryBuilder = new
FilteredSQLQueryBuilder();
        String sqlQuery = filteredSQLQueryBuilder.buildQuery(criteria);
        QueryParam queryParam = new QueryParam(sqlQuery, parameterMap);
        List<Object[]> resultList =
PaginatedLists.executePaginatedQuery(paginatedList, queryParam,
sortCriteria);
        List<CommonBookDto> commonBookDtoList = new
ArrayList<CommonBookDto>();
        for (Object[] o : resultList) {
            int i = 0;
            CommonBookDto commonBookDto = new CommonBookDto();
            commonBookDto.setId((String) o[i++]);
            commonBookDto.setBookId((String) o[i++]);
            commonBookDto.setCreateDate((Date) o[i++]);
            commonBookDto.setReadDate((Date) o[i++]);
            String author = (String) o[i++];
            logger.info(author);
            Double rating = (Double) o[i++];
            String genreString = (String) o[i++];
            logger.info(genreString);
            List<String> genresList =
Arrays.asList(genreString.split(","));
            boolean authorFound = authors == null;
            if(!authorFound){
                logger.info("authors is empty");
            }
            else{
                logger.info("authors is not empty");
            }
            boolean genreFound = genres == null;
            if (!authorFound && authors.contains(author)) {
                logger.info("Contains Author");
                authorFound = true;
            }
            if (!genreFound) {
                for (String genre : genresList) {
                    if (genres.contains(genre)) {
                        logger.info("Found genre");
                        genreFound = true;
                        break;
                    }
                }
            }
            else{
                logger.info("genreFound is true");
            }
            boolean ratingPass = ratingFilter == null || (rating != null &&
rating > ratingFilter);
            if(ratingPass){
```

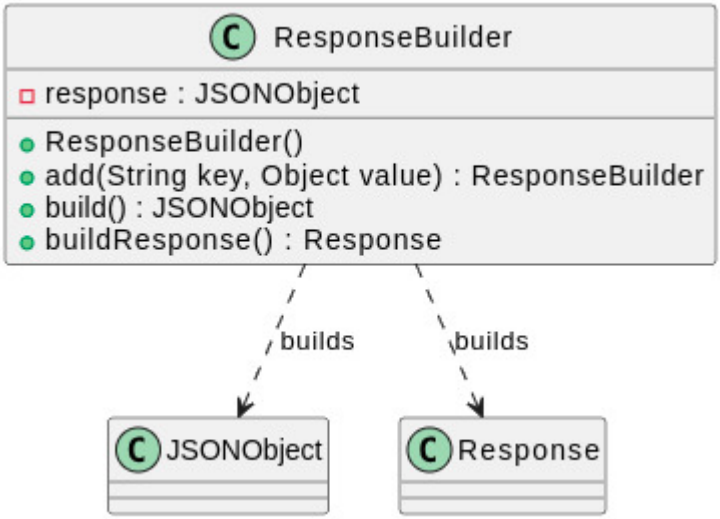
```
                logger.info("greater rating than ratingFIlter");
            }
            if (authorFound && genreFound && ratingPass) {
                logger.info("added something to dto");
                commonBookDtoList.add(commonBookDto);
            }
        }
        paginatedList.setResultList(commonBookDtoList);
    }
}
```

The Strategy Pattern is implemented in the `CommonBookDao` to allow for different strategies in constructing SQL queries for retrieving common books based on various criteria. This pattern promotes flexibility by encapsulating the query building algorithm into separate classes, which can be interchanged based on specific requirements.

- **Components**:
  - `SQLQueryBuilder`: Interface defining the contract for SQL query builder implementations.
  - `DefaultSQLQueryBuilder`: Constructs SQL queries for retrieving common books based on standard criteria.
  - `FilteredSQLQueryBuilder`: Builds SQL queries with additional filtering options like authors, genres, and rating.
  - `RatingSQLQueryBuilder`: Generates SQL queries with rating-related calculations such as average rating and rating count.
- **Benefits**:
  - **Modularity**: Encapsulates query construction logic into separate classes for improved organization.
  - **Flexibility**: Allows for dynamic selection of query building strategies based on runtime requirements.
  - **Extensibility**: Supports easy integration of new query builders to accommodate evolving criteria or requirements.
- **Considerations**:
  - Ensure consistency in naming conventions and query structure across different query builders.
  - Thoroughly test each query builder implementation to validate correctness and performance.
  - Provide clear documentation to facilitate understanding and usage by other developers.

By leveraging the Strategy Pattern, the `CommonBookDao` achieves enhanced versatility and maintainability in constructing SQL queries, enabling efficient retrieval of common books based on diverse criteria.

Builder Pattern Implementation in `ResponseBuilder`



`ResponseBuilder.java` (com.sismics.books.rest.resource)

```
package com.sismics.books.rest.resource;

public class ResponseBuilder {
    private JSONObject response;
    private static final Logger logger =
Logger.getLogger(ResponseBuilder.class.getName());

    public ResponseBuilder() {
        this.response = new JSONObject();
    }

    public ResponseBuilder add(String key, Object value) {
        try {
            this.response.put(key, value);
        } catch (JSONException e) {
            logger.log(Level.SEVERE, "JSONException occurred while adding
key '" + key + "' with value '" + value
                    + "' to JSON response.", e);
        }
        return this;
    }

    public JSONObject build() {
        return this.response;
    }

    public Response buildResponse() {
        return Response.ok(this.response.toString()).build();
    }
}
```

The ResponseBuilder class exemplifies the Builder Pattern, facilitating the incremental construction of a complex JSONObject for REST API responses. This design pattern enhances code readability and maintainability, especially for creating JSON objects with optional parameters.

Example Usage in CommonBookResource

```
@GET
@Path("{id: [a-z0-9\\-]+}")
@Produces(MediaType.APPLICATION_JSON)
public Response get(@PathParam("id") String id) throws JSONException {
    // Authentication and book retrieval logic omitted for brevity
    ResponseBuilder responseBuilder = new ResponseBuilder()
            .add("id", commonBook.getId())
            .add("createDate", commonBook.getCreateDate())
            // Additional data fields added to the response
            .add("avgRating", avgRating)
            .add("genre", genreString)
            .add("numRatings", numRatings);

    return responseBuilder.buildResponse();
}
```

Components:

- **Product (JSONObject):** The complex object being constructed.
- **Builder (ResponseBuilder):** Provides methods to add key-value pairs to the JSON object and methods to retrieve the final product.

Benefits:

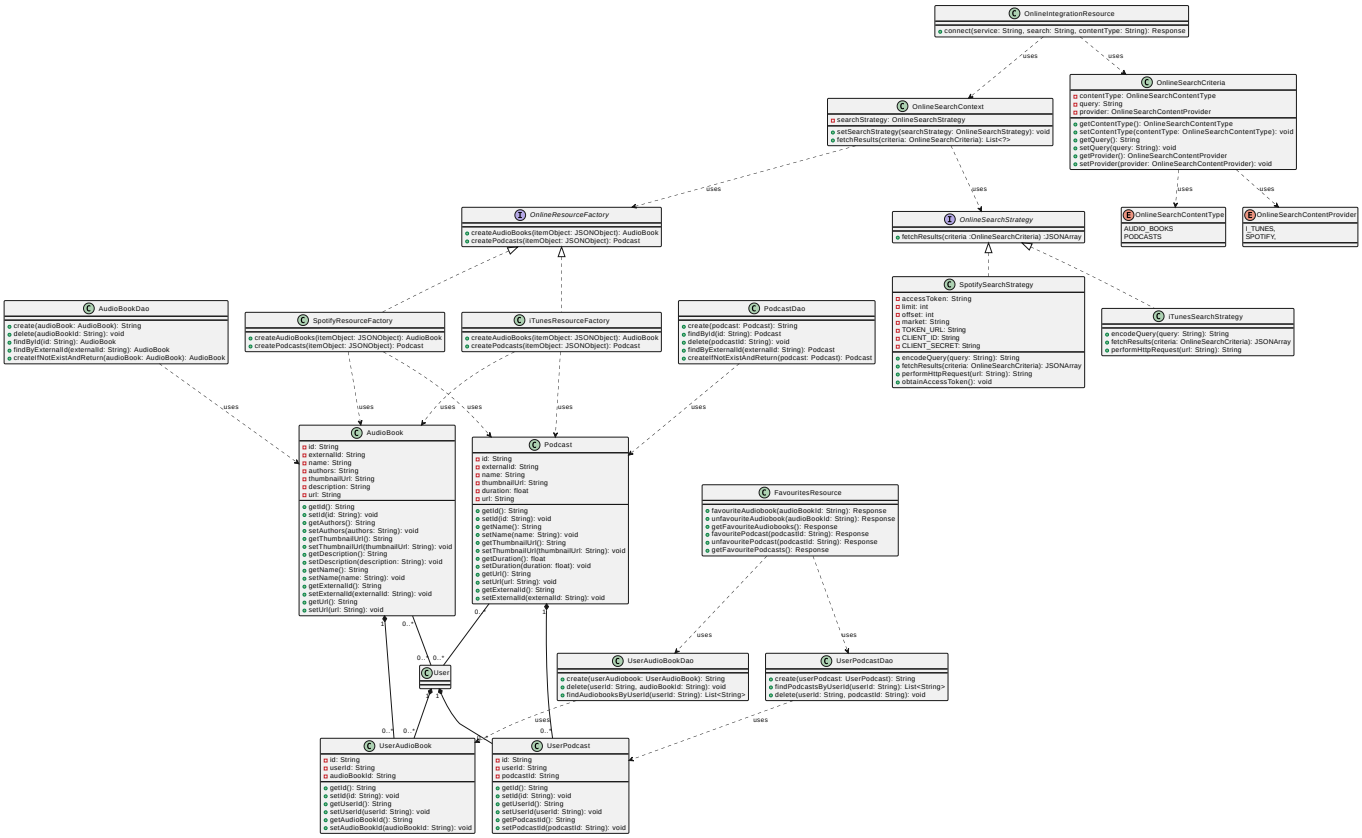
- **Flexibility:** Enables the construction of objects with various configurations, simplifying the creation of complex JSON structures.
- **Fluent Interface:** Supports method chaining, offering a concise and readable way to add data to the JSON object.
- **Separation of Construction and Representation:** Allows for changing the product's internal representation without changing the client code.

Considerations:

- The builder class can become complicated if the product itself is complex, potentially requiring multiple builder classes or methods to cover all use cases.
- Developers should ensure that the builder does not leave the product in an inconsistent state during construction.
- The ResponseBuilder demonstrates the Builder Pattern's power in simplifying the construction of complex objects, enabling clear and maintainable code for building JSON responses in RESTful services.

Part 3: Online Integration

Class diagram of the new classes for this feature:



Exact Code changes:

1. added `OnlineSearchContentProvider` enum data type
2. added `OnlineSearchContentType` enum data type
3. `AudioBook` class that contains data related to an audiobook (common to Spotify and ITunes). along with its DAO (having function for creation, deletion, and different types of retrivals). Uniquely Identified by both `id` and `externId` (externId is the Id provided by the Spotify and ITunes APIs, this remains constant accross multiple API hits)
4. `Podcast` class that contains data related to an podcast (common to Spotify and ITunes). along with its DAO (having function for creation, deletion, and different types of retrivals). Uniquely Identified by both `id` and `externId` (externId is the Id provided by the Spotify and ITunes APIs, this remains constant accross multiple API hits)
5. `UserAudioBook` and `UserPodcast` are classes that hold the relation between `User` (through userId) and `AudioBook`, `Podcast` (through audioBookId and podcast Id respectively). and their corresponding DAO classes.
6. `OnlineSearchCriteria` is a class that holds data of a query done by the user. holds `OnlineSearchContentProvider`, `OnlineSearchContentType` and `query` data.
7. `OnlineResourceFactory` is an interface that is to be used to create `AudioBook` and `Podcast` (using `createAudioBooks` and `createPodcasts` methods) Objects from the `JSONObject` type data provided by the `OnlineSearchStrategy` (i.e from the API). `SpotifyResourceFactory` and `iTunesResourceFactory` are the concrete implementations of this interface.

8. `OnlineSearchStrategy` is an interface that is implemented by `SpotifySearchStrategy` and `iTunesSearchStrategy` to extract `JSONArray` type data from the API endpoints of respective service. Each of it takes input `OnlineSearchCriteria` and uses the `OnlineSearchContentType` and `query` attributes of this class.
9. `OnlineIntegrationResource` acts as an endpoint for the frontend to make a query to the service for the list of AudioBooks and Podcasts. this `OnlineIntegrationResource` class uses `OnlineSearchStrategy` to do the required.
10. `FavouritesResource` class has all endpoints to add/remove AudioBook or Podcast to/from a user's favourite list.

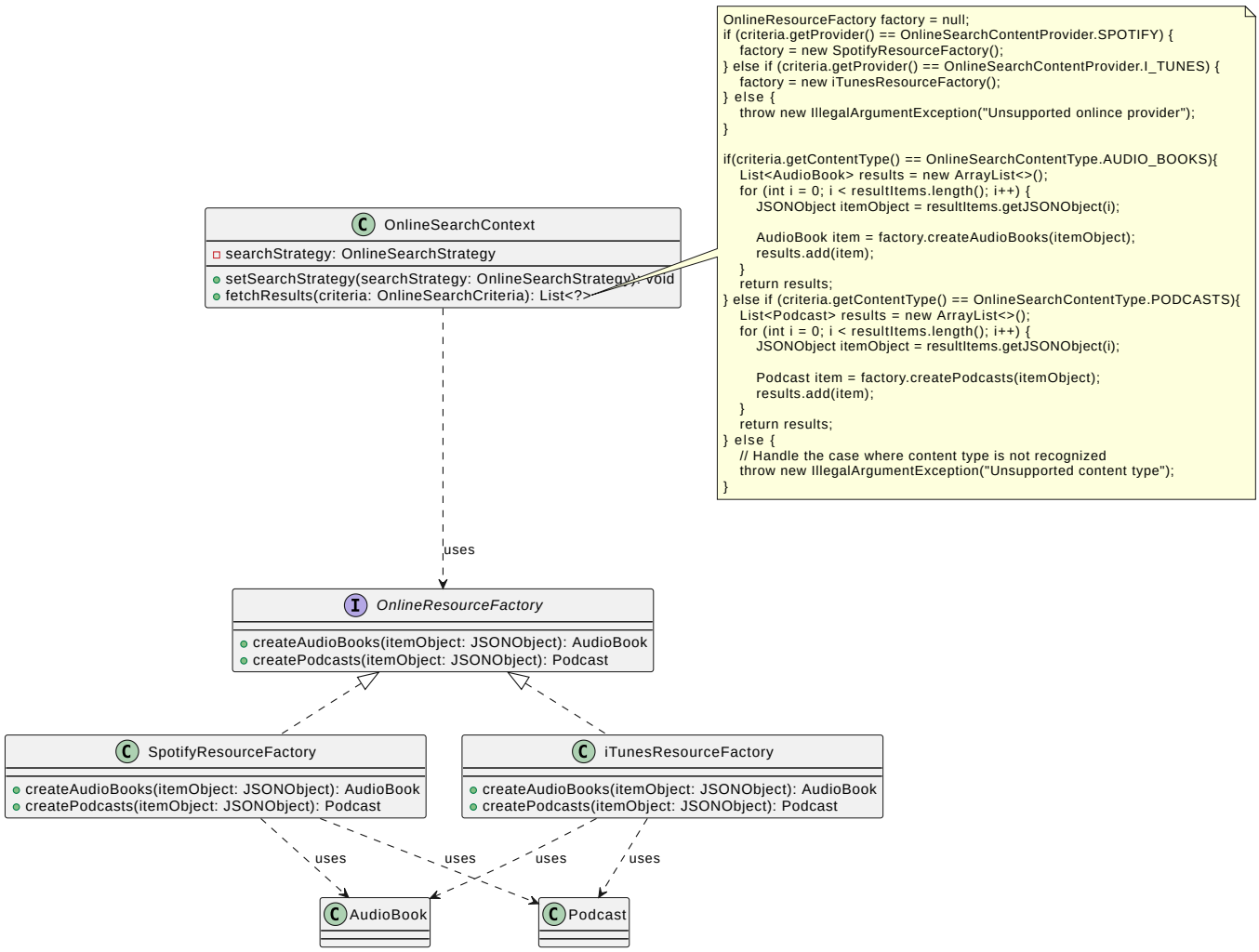
Basic Flow of this new feature:

1. User goes to the respective webpage
2. User selects Service Provider (Spotify or iTunes)
3. User selects Content Type (AudioBooks or Podcasts)
4. User enters the search query and clicks 'search'.
5. On searching, all the results are added to database `AudioBook` and `Podcast`
6. Then the data is returned to user and shown in the website. user gets an option to favourite any `AudioBook` or `Podcast`.
7. When user favourites a `AudioBook` or `Podcast`, the corresponding userId and audioBookId, podcastId is added to `UserAudioBook` or `UserPodcast`.
8. he can view this favourites form "Favourites" tab. where he gets to choose to unfavourite them also. when unfavourite is clicked, the corresponding row from `UserAudioBook` or `UserPodcast` is removed.

Design Patterns Used

Factory Pattern

The Factory Pattern is incredibly useful in this context, especially when dealing with object creation and handling different types of resources from various providers like Spotify and iTunes.



Advantages of this pattern in this context include:

Encapsulation of Object Creation:

The Factory Pattern encapsulates the creation logic for objects like `AudioBook` and `Podcast`, allowing you to hide the complex creation process from the client. This promotes a clean separation of concerns and keeps the codebase more maintainable.

Abstraction:

By using the Factory Pattern, you're abstracting the creation of `AudioBook` and `Podcast` objects behind an interface (`OnlineResourceFactory`). This abstraction allows you to change the implementation details of how these objects are created without affecting the client code.

Code Extensibility:

Adding new types of resources or content providers becomes easier with the Factory Pattern. For example, if you need to add support for a new content provider like SoundCloud in the future, you can simply create a new implementation of `OnlineResourceFactory` without modifying existing code.

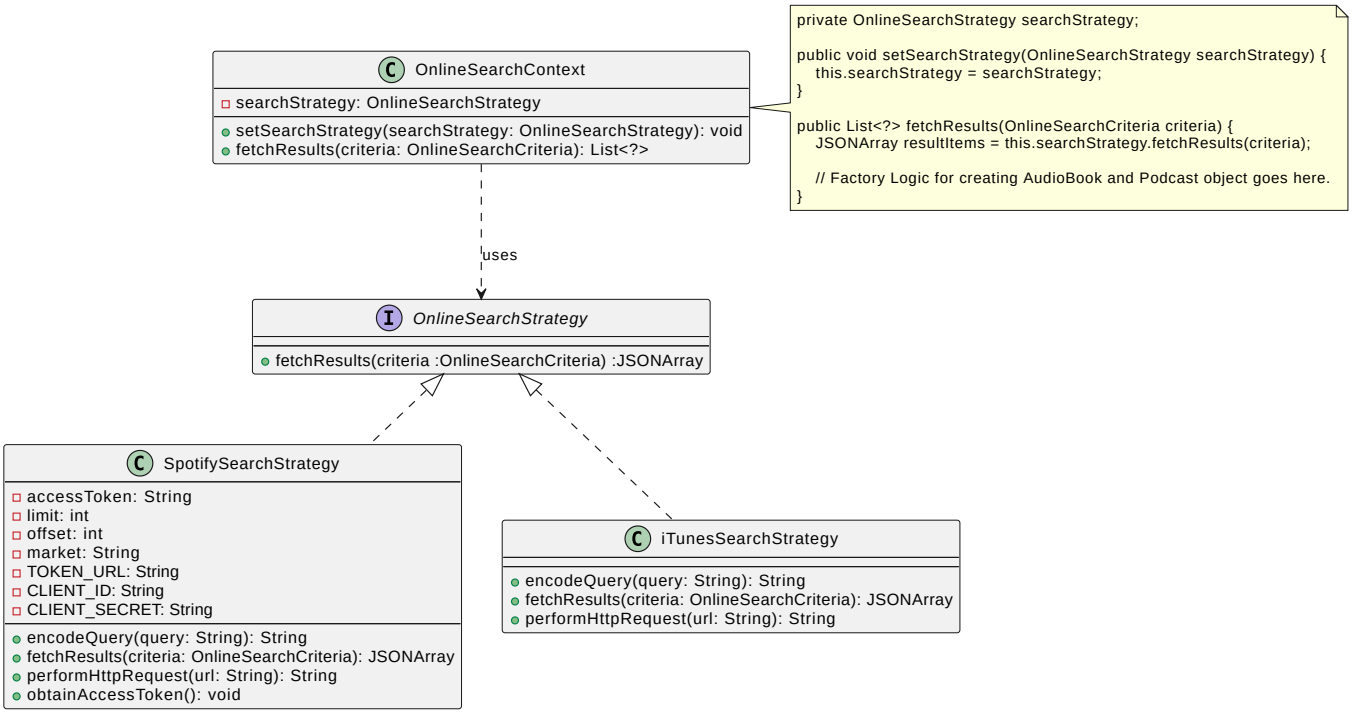
Ease of Testing:

By using the Factory Pattern, you can easily mock the creation process of `AudioBook` and `Podcast` objects during testing. This facilitates unit testing and ensures that your code behaves as expected under different scenarios.

In summary, the Factory Pattern brings several benefits to your application architecture by providing a flexible and maintainable approach to object creation, especially in the context of handling different types of resources from multiple providers.

Strategy Pattern

The Strategy Pattern proves invaluable for managing various online search strategies, such as those for Spotify and iTunes. By abstracting the search algorithms into separate implementations of the OnlineSearchStrategy interface, such as SpotifySearchStrategy and iTunesSearchStrategy, the pattern promotes flexibility, extensibility, and maintainability in the application architecture. With the Strategy Pattern, algorithms can be dynamically selected at runtime, facilitating dynamic behavior, separation of concerns, and easy testing of individual strategies.



Few of the Advantages in this context include:

Encapsulation of Algorithms:

The Strategy Pattern encapsulates algorithms (represented by the `OnlineSearchStrategy` interface and its implementations) independently of the client that uses them. This allows for easy swapping of algorithms at runtime without affecting the client's code.

Flexibility and Extensibility:

By employing the Strategy Pattern, you can introduce new search strategies (e.g., for different providers or content types) by simply implementing the `OnlineSearchStrategy` interface. This promotes code extensibility and accommodates future changes without the need for significant modifications to existing code.

Dynamic Behavior:

Since the Strategy Pattern enables the selection of algorithms at runtime, it facilitates dynamic behavior in your application. This is particularly useful when dealing with user preferences, changing requirements, or different environmental conditions.

Separation of Concerns:

The Strategy Pattern separates the concerns related to search algorithms from the classes that use them (e.g., OnlineResourceFactory). This separation promotes a cleaner codebase, improves maintainability, and makes it easier to understand and modify the behavior of each component independently.

Testability:

By using the Strategy Pattern, you can easily test each search strategy in isolation, independently of the client code. This promotes unit testing and ensures that each strategy behaves as expected under different scenarios.