Guide: Implementing Book CRUD Operations (v2)

This guide details how to implement Create, Read, Update, and Delete (CRUD) operations for the Book model, following a layered architecture, reflecting the merge of Seller functionality into the User model.

References:

- Models: book.py, author.py, publisher.py, category.py, rating.py, user.py, location.py, book_category_table.py
- Example Implementation Style: guide_category.md

1. Model Layer (src/app/model/book.py)

- **Existing Model:** The Book model (__tablename__ = 'book') includes fields like title, description, price, quantity, discount_percent, image URLs (image_url_1, image_url_2, image_url_3), rating (to store the calculated average), foreign keys (publisher_id, author_id, user_id), and timestamps (created_at, updated_at). The user_id links a book to the user who listed it for sale. It also includes database-level constraints for quantity, price, and discount_percent.
- **Relationships:**
  - Many-to-One: publisher (to Publisher), author (to Author), user (to User, linking to the user selling the book).
  - Many-to-Many: categories (to Category via book_category_table).
  - One-to-Many: ratings (to Rating with cascade="all, delete-orphan").
- **Responsibilities:** Defines the data structure, relationships, database mapping, and serialization for books. Includes a helper method get_seller_location_info() to retrieve location details of the user selling the book.
- **Serialization:** The to_dict() method provides a detailed view, and to_simple_dict() offers a summarized view.

```
# Inside Book class in book.py (Reflecting new model structure)

# Note: The Book.rating field (Numeric(3, 2)) will be updated by the BookService
# or potentially a RatingService, as described later.

def get_seller_location_info(self):
    """Helper method to safely get seller's location details."""
    if not self.user or not self.user.location or not self.user.location.city:
        return None, None, None

    city = self.user.location.city
    state = city.state
    country = state.country if state else None

    return (
        city.name if city else None,
        state.name if state else None,
        country.name if country else None
    )

def to_dict(self, include_categories=True):
    """Returns a detailed dictionary representation of the book."""
    city_name, state_name, country_name = self.get_seller_location_info()
    data = {
        'id': self.id,
        'title': self.title,
        'author_id': self.author_id,
        'author_name': self.author.full_name if self.author else None,
        'publisher_id': self.publisher_id,
        'publisher_name': self.publisher.name if self.publisher else None,
        'description': self.description,
        'rating': float(self.rating) if self.rating is not None else None,
        'quantity': self.quantity,
        'price': float(self.price) if self.price is not None else None,
        'discount_percent': self.discount_percent,
        'user_name': self.user.full_name if self.user else None,
        'user_id': self.user_id,
        'seller_location': {
                        'city': city_name,
                        'state': state_name,
                        'country': country_name,
                    },
        'image_url_1': self.image_url_1,
        'image_url_2': self.image_url_2,
        'image_url_3': self.image_url_3,
        'created_at': self.created_at.isoformat() if self.created_at else None,
        'updated_at': self.updated_at.isoformat() if self.updated_at else None,
    }
    if include_categories and self.categories:
        # Assuming Category model has a to_dict() method
        data['categories'] = [category.to_dict() for category in self.categories]
    else:
        data['categories'] = []

    return data
```

```
def to_simple_dict(self):
    """Returns a simpler dictionary representation of the book."""
    city_name, _, _ = self.get_seller_location_info() # Only need city for simple view
    return {
        'id': self.id,
        'title': self.title,
        'author_name': self.author.full_name if self.author else None,
        'image_url_1': self.image_url_1,
        'rating': float(self.rating) if self.rating is not None else None,
        'price': float(self.price) if self.price is not None else None,
        'discount_percent': self.discount_percent,
        'user_name': self.user.full_name if self.user else None,
        'seller_city': city_name,
    }

# __table_args__ in the Book model define constraints like:
# CheckConstraint('quantity >= 0', name='book_quantity_non_negative'),
# CheckConstraint('price > 0', name='book_price_positive'),
# CheckConstraint('discount_percent BETWEEN 0 AND 100',
name='book_discount_percent_range'),

# Ensure Author, Publisher, Category, User, Rating models also have
# appropriate to_dict() methods as needed by the Book serialization.
```

2. Utility Layer (src/app/utils/)

- **Validators (validators.py):**
    - Create validate_book_input(data, is_update=False):
        - **Checks (Create):** title (required, string), price (required, positive number), quantity (required, non-negative integer), discount_percent (optional, 0-100 integer), description (optional, string), author_id (optional, integer - check existence), publisher_id (optional, integer - check existence), category_ids (optional, list of integers - check existence). user_id is derived from the logged-in user context (JWT), not input data.
        - **Checks (Update):** Similar checks, but fields are optional. If provided, they must meet the constraints.
        - **Existence Checks:** Verify that provided author_id, publisher_id, and category_ids correspond to existing records in their respective tables.
        - Return a dictionary of errors if validation fails, otherwise None.
- **Response Formatting (response.py):**
    - Use existing success_response, error_response, create_response.

- **Decorators (decorators.py):**
  - Use @jwt_required() for authenticated endpoints (Create, Update, Delete, Get My Books).
  - Implement or use a role/permission checking decorator (e.g., @roles_required or custom logic within the service/route) if needed to restrict actions based on User.role.

3. Service Layer (src/app/services/book_service.py)

- **Create BookService Class:**

```
from decimal import Decimal, ROUND_HALF_UP # For price validation and rating rounding
from flask_jwt_extended import get_jwt_identity # To get current user ID
from sqlalchemy import func # For average calculation
from sqlalchemy.orm import joinedload, subqueryload # For eager loading
from sqlalchemy.exc import IntegrityError
from ..extensions import db
from ..model.book import Book
from ..model.author import Author
from ..model.publisher import Publisher
from ..model.category import Category
from ..model.user import User # Import User model
from ..model.rating import Rating # Needed for average calculation
from ..utils.validators import validate_book_input
from ..utils.response import success_response, error_response
import logging

logger = logging.getLogger(__name__)

class BookService:

    def _get_and_validate_related(self, data):
        """Helper to fetch and validate related entities (Author, Publisher, Categories)."""
        # (Content remains the same as previous version)
        related = {'author': None, 'publisher': None, 'categories': []}
        errors = {}

        author_id = data.get('author_id')
        if author_id:
            related['author'] = Author.query.get(author_id)
            if not related['author']:
                errors['author_id'] = f"Author with ID {author_id} not found."
        publisher_id = data.get('publisher_id')
        if publisher_id:
            related['publisher'] = Publisher.query.get(publisher_id)
            if not related['publisher']:
                errors['publisher_id'] = f"Publisher with ID {publisher_id} not found."

        category_ids = data.get('category_ids', [])
        if category_ids:
            if not isinstance(category_ids, list):
                 errors['category_ids'] = "Category IDs must be a list."
            else:
                categories = Category.query.filter(Category.id.in_(category_ids)).all()
                if len(categories) != len(set(category_ids)): # Check if all provided IDs were
found
                    found_ids = {cat.id for cat in categories}
                    missing_ids = [cid for cid in category_ids if cid not in found_ids]
                    errors['category_ids'] = f"Categories with IDs {missing_ids} not found."
                else:
                     related['categories'] = categories # Assign list of Category objects

        return related, errors

    def _update_book_average_rating(self, book_id):
        """
        Helper function to recalculate and update the average rating for a book.
        This should be called within the same transaction whenever a Rating
        for this book is created, updated, or deleted (likely from a RatingService).
        """
        # (Content remains the same as previous version - uses Book.rating field)
        try:
            book = Book.query.get(book_id)
            if not book:
                logger.warning(f"Attempted to update rating for non-existent book ID:
{book_id}")
                return

            avg_score_result = db.session.query(
                func.coalesce(func.avg(Rating.score), 0)
            ).filter(Rating.book_id == book_id).scalar()

            avg_score_decimal = Decimal(str(avg_score_result))
            rounded_avg_score = avg_score_decimal.quantize(Decimal("0.01"),
rounding=ROUND_HALF_UP)
```

```
            book.rating = rounded_avg_score
            logger.info(f"Updated average rating for Book ID {book_id} to {book.rating}")
        except Exception as e:
            logger.error(f"Error updating average rating for Book ID {book_id}: {e}",
exc_info=True)
            # Let the calling function handle transaction management.

    def create_book(self, data, user_id):
        # 1. Get User (who will be the seller/owner of the book)
        user = User.query.get(user_id)
        if not user:
             # This case might be less likely if jwt_required worked, but good practice
             return error_response("User not found.", error="unauthorized",
status_code=401)

        # Validate if the user has a location_id set
        if not user.location_id:
            logger.warning(f"User ID {user_id} attempted to create a book without a
location_id.")
            return error_response(
                "You must set your location before listing a book for sale. Please update your
profile with a location.",
                error="location_required",
                status_code=400  # Bad Request, as a prerequisite is missing
            )

        # Optional: Check if user role allows creating books (e.g., 'seller', 'admin')
        # if user.role not in ['seller', 'admin']:
        #    return error_response("User role not permitted to create books.",
error="forbidden", status_code=403)

        # 2. Validate Input Data
        errors = validate_book_input(data)
        if errors:
            return error_response("Validation failed", errors=errors, status_code=400)

        # 3. Fetch and Validate Related Entities (Author, Publisher, Categories)
        related_entities, related_errors = self._get_and_validate_related(data)
        if related_errors:
             errors = (errors or {}) | related_errors
             return error_response("Validation failed", errors=errors, status_code=400)
        # 4. Create Book Instance (Initialize rating to None or 0)
        new_book = Book(
            title=data['title'],
            description=data.get('description'),
            quantity=data['quantity'],
            price=Decimal(str(data['price'])), # Ensure conversion to Decimal
            discount_percent=data.get('discount_percent', 0),
            image_url_1=data.get('image_url_1'),
            image_url_2=data.get('image_url_2'),
            image_url_3=data.get('image_url_3'),
            user_id=user.id, # Assign the logged-in user's ID
            author=related_entities['author'],
            publisher=related_entities['publisher'],
            rating=None # Initialize rating
        )

        # 5. Add Categories
        if related_entities['categories']:
            new_book.categories.extend(related_entities['categories'])

        # 6. Add to Session and Commit
        try:
            db.session.add(new_book)
            db.session.commit()
            logger.info(f"Book created: ID {new_book.id}, Title '{new_book.title}', User ID
{user.id}")
            # Use the model's to_dict method for the response
            return success_response("Book created successfully", data=new_book.to_dict(),
status_code=201)
        except IntegrityError as e:
            db.session.rollback()
            logger.error(f"Integrity error creating book '{data['title']}': {e}", exc_info=True)
            return error_response("Failed to create book due to database constraint.",
error="db_integrity_error", status_code=409)
        except Exception as e:
            db.session.rollback()
            logger.error(f"Error creating book '{data['title']}': {e}", exc_info=True)
            return error_response("Failed to create book", error=str(e), status_code=500)

    def get_all_books(self, args):
        page = args.get('page', 1, type=int)
        per_page = args.get('per_page', 12, type=int)
        search_term = args.get('search')
        author_id = args.get('author_id', type=int)
        publisher_id = args.get('publisher_id', type=int)
        category_id = args.get('category_id', type=int)
        user_id_filter = args.get('user_id', type=int)
        min_price = args.get('min_price', type=float)
        max_price = args.get('max_price', type=float)
        sort_by = args.get('sort_by', 'created_at') # Default sort by Book.created_at
        order = args.get('order', 'desc')

        # Eager load related entities. User.location is accessed via Book.user.location
        # which is used by Book.get_seller_location_info() in to_simple_dict().
        query = Book.query.options(
            joinedload(Book.author),
            joinedload(Book.publisher),

joinedload(Book.user).joinedload(User.location).joinedload(Location.city).joinedload(Cit
y.state).joinedload(State.country), # Eager load full location path
            subqueryload(Book.categories)
        )

        # Filtering
        if search_term: query = query.filter(Book.title.ilike(f'%{search_term}%'))
        if author_id: query = query.filter(Book.author_id == author_id)
        if publisher_id: query = query.filter(Book.publisher_id == publisher_id)
        if category_id: query = query.filter(Book.categories.any(Category.id ==
category_id))
        if user_id_filter: query = query.filter(Book.user_id == user_id_filter)
        if min_price is not None: query = query.filter(Book.price >= Decimal(str(min_price)))
        if max_price is not None: query = query.filter(Book.price <=
Decimal(str(max_price)))

        # Sorting
        order_direction = db.desc if order.lower() == 'desc' else db.asc
        if sort_by == 'price':
            query = query.order_by(order_direction(Book.price))
        elif sort_by == 'title':
            query = query.order_by(order_direction(Book.title))
        elif sort_by == 'rating': # Uses Book.rating field
           query = query.order_by(order_direction(Book.rating))
        else: # Default sort by creation date (Book.created_at)
            query = query.order_by(order_direction(Book.created_at))

        try:
            paginated_books = query.paginate(page=page, per_page=per_page,
error_out=False)
            return success_response(
                "Books retrieved successfully",
                data={
                    # Use the model's to_simple_dict method
                    "books": [book.to_simple_dict() for book in paginated_books.items],
                    "total": paginated_books.total,
                    "pages": paginated_books.pages,
                    "current_page": paginated_books.page
                },
                status_code=200
            )
        except Exception as e:
            logger.error(f"Error retrieving books: {e}", exc_info=True)
            return error_response("Failed to retrieve books", error=str(e), status_code=500)

    def get_book_by_id(self, book_id):
        # Eager load related entities. User.location is accessed via Book.user.location
        # which is used by Book.get_seller_location_info() in to_dict().
        book = Book.query.options(
            joinedload(Book.author),
            joinedload(Book.publisher),

joinedload(Book.user).joinedload(User.location).joinedload(Location.city).joinedload(Cit
y.state).joinedload(State.country), # Eager load full location path
            joinedload(Book.categories),
            # subqueryload(Book.ratings).joinedload(Rating.user) # Optional: if ratings
details are needed
        ).get(book_id)

        if not book:
            return error_response("Book not found", error="not_found", status_code=404)
        # Use the model's to_dict method
        return success_response("Book found", data=book.to_dict(), status_code=200)

    def get_books_by_user(self, owner_user_id, args):
        """Gets books listed by a specific user."""
        args = args.copy()
        args['user_id'] = owner_user_id
        return self.get_all_books(args) # Reuses get_all_books with user_id filter

    def update_book(self, book_id, data, current_user_id):
        # Eager load the user relationship for the authorization check
        book = Book.query.options(joinedload(Book.user)).get(book_id)
        if not book:
            return error_response("Book not found", error="not_found", status_code=404)

        # Authorization Check
        user = User.query.get(current_user_id)
        if not user: return error_response("User not found.", error="unauthorized",
status_code=401)
        is_owner = book.user_id == current_user_id
        is_admin = user.role == 'admin'
        if not (is_owner or is_admin):
            logger.warning(f"Unauthorized attempt to update Book ID {book_id} by User ID
{current_user_id}")
            return error_response("You are not authorized to update this book.",
error="forbidden", status_code=403)

        # Validate Input Data
        errors = validate_book_input(data, is_update=True)
        if errors: return error_response("Validation failed", errors=errors, status_code=400)

        # Fetch and Validate Related Entities
        related_entities, related_errors = self._get_and_validate_related(data)
        if related_errors:
             errors = (errors or {}) | related_errors
             return error_response("Validation failed", errors=errors, status_code=400)

        updated = False
        for key, value in data.items():
            if key in ['category_ids', 'author_id', 'publisher_id', 'user_id']: continue # user_id
cannot be changed here
            if key == 'price' and value is not None: value = Decimal(str(value))
            if hasattr(book, key) and getattr(book, key) != value:
                setattr(book, key, value)
                updated = True

        if 'author_id' in data:
             new_author = related_entities['author'] if data['author_id'] else None
             if book.author != new_author:
                 book.author = new_author
                 updated = True
        if 'publisher_id' in data:
             new_publisher = related_entities['publisher'] if data['publisher_id'] else None
             if book.publisher != new_publisher:
                 book.publisher = new_publisher
                 updated = True
        if 'category_ids' in data:
            current_category_ids = {cat.id for cat in book.categories}
            new_category_ids = set(data.get('category_ids', []))
            if current_category_ids != new_category_ids:
                book.categories = related_entities['categories']
                updated = True

        if not updated:
            return error_response("No changes detected in the provided data.",
error="no_change", status_code=400)

        try:
            db.session.commit()
            logger.info(f"Book updated: ID {book.id}, Title '{book.title}' by User ID
{current_user_id}")
            return success_response("Book updated successfully", data=book.to_dict(),
status_code=200)
        except IntegrityError as e:
            db.session.rollback()
            logger.error(f"Integrity error updating book {book_id}: {e}", exc_info=True)
            return error_response("Failed to update book due to database constraint.",
error="db_integrity_error", status_code=409)
        except Exception as e:
            db.session.rollback()
            logger.error(f"Error updating book {book_id}: {e}", exc_info=True)
            return error_response("Failed to update book", error=str(e), status_code=500)

    def delete_book(self, book_id, current_user_id):
        book = Book.query.options(joinedload(Book.user)).get(book_id)
        if not book: return error_response("Book not found", error="not_found",
status_code=404)

        user = User.query.get(current_user_id)
        if not user: return error_response("User not found.", error="unauthorized",
status_code=401)
        is_owner = book.user_id == current_user_id
        is_admin = user.role == 'admin'
        if not (is_owner or is_admin):
            logger.warning(f"Unauthorized attempt to delete Book ID {book_id} by User ID
{current_user_id}")
```

```
            return error_response("You are not authorized to delete this book.",
error="forbidden", status_code=403)

        try:
            book_title = book.title
            db.session.delete(book) # Ratings are cascade deleted by DB relationship
            db.session.commit()
            logger.info(f"Book deleted: ID {book_id}, Title '{book_title}' by User ID
{current_user_id}")
            return success_response("Book deleted successfully", status_code=200) # Or
204
        except Exception as e:
            db.session.rollback()
            logger.error(f"Error deleting book {book_id}: {e}", exc_info=True)
            return error_response("Failed to delete book", error=str(e), status_code=500)
```

4. Route Layer (src/app/routes/book_route.py)

- **Create Blueprint and Import Services/Utils:**

```
from flask import Blueprint, request, jsonify
from flask_jwt_extended import jwt_required, get_jwt_identity
from ..services.book_service import BookService
from ..utils.response import create_response
# from ..utils.decorators import roles_required
# from ..utils.roles import UserRoles
import logging

logger = logging.getLogger(__name__)
book_bp = Blueprint('books', __name__, url_prefix='/api/v1/books')
book_service = BookService()

@book_bp.route('/', methods=['POST'])
@jwt_required()
# @roles_required(UserRoles.SELLER, UserRoles.ADMIN) # Optional role check
def create_book_route():
    user_id = get_jwt_identity()
    data = request.get_json()
    if not data: return create_response(status="error", message="Request body must be
JSON"), 400
    result = book_service.create_book(data, user_id)
    status_code = result.get('status_code', 500)
    return create_response(**result), status_code
```

```
@book_bp.route('/', methods=['GET'])
def get_books_route():
   args = request.args
   # Import Location, City, State, Country if not already for eager loading path
   # from ..model.location import Location
   # from ..model.city import City
   # from ..model.state import State
   # from ..model.country import Country
   result = book_service.get_all_books(args)
   status_code = result.get('status_code', 500)
   return create_response(**result), status_code

@book_bp.route('/<int:book_id>', methods=['GET'])
def get_book_by_id_route(book_id):
   # Import Location, City, State, Country if not already for eager loading path
   result = book_service.get_book_by_id(book_id)
   status_code = result.get('status_code', 500)
   return create_response(**result), status_code

@book_bp.route('/me', methods=['GET'])
@jwt_required()
def get_my_books_route():
   user_id = get_jwt_identity()
   args = request.args
   result = book_service.get_books_by_user(user_id, args)
   status_code = result.get('status_code', 500)
   return create_response(**result), status_code

@book_bp.route('/<int:book_id>', methods=['PATCH', 'PUT'])
@jwt_required()
def update_book_route(book_id):
   user_id = get_jwt_identity()
   data = request.get_json()
   if not data: return create_response(status="error", message="Request body must be
JSON"), 400
   result = book_service.update_book(book_id, data, user_id)
   status_code = result.get('status_code', 500)
   return create_response(**result), status_code

@book_bp.route('/<int:book_id>', methods=['DELETE'])
@jwt_required()
def delete_book_route(book_id):
    user_id = get_jwt_identity()
    result = book_service.delete_book(book_id, user_id)
    status_code = result.get('status_code', 500)
    if result.get('status') == 'success' and (status_code == 200 or status_code == 204):
        return create_response(**result), status_code # Or return '', 204 for explicit 204
    return create_response(**result), status_code

# Register blueprint in app factory
# from .routes.book_route import book_bp
# app.register_blueprint(book_bp)
```

5. Key Considerations & Error Handling

- **Role-Based Access Control (RBAC):** Use User.role for authorization. The current implementation allows owners or 'admin' users to update/delete.
- **Input Validation:** validate_book_input and the user.location_id check in create_book are crucial.
- **Relationship Management:** Correctly handle related entities (Author, Publisher, Categories) and user_id.
- **Average Rating (Book.rating):**
    - Book.rating stores the average.
    - BookService._update_book_average_rating(book_id) calculates it.
    - **This helper must be called by a RatingService after Rating CRUD operations.**
- **Error Handling:** Use consistent error responses and log errors.
- **Pagination & Filtering:** Robustly implement in get_all_books.
- **Serialization (to_dict, to_simple_dict):** Ensure they reflect the model and provide necessary data. Note the change to seller_location (object) in to_dict and seller_city (string) in to_simple_dict.
- **Eager Loading for Location:** The get_all_books and get_book_by_id service methods now include a more comprehensive eager load path: joinedload(Book.user).joinedload(User.location).joinedload(Location.city).joinedload(City.state).joinedload(State.country) to ensure all necessary data for get_seller_location_info() is fetched efficiently. You'll need to ensure Location, City, State, and Country models are imported in book_service.py if they are not already.