Project Part 10 - Transactions

29/30 Points

8/11/2021

Attempt 1

REVIEW FEEDBACK
8/11/2021

Attempt 1 Score:
29/30

View Feedback

Unlimited Attempts Allowed

Details

Due: Wednesday, August 11
Total Points: 30 points
Deliverables: [Name]BookstoreTransact.war uploaded to the CS5244 server
Resources: server-src-transact.zip (https://drive.google.com/file/d/19KKIduHyWGqxdGztFttpom8qbFXGk8-S/view?usp=sharing)

Setup your project

Establish a new project `[Name]BookstoreTransact` according to our standard setup procedure. Copy the relevant files from your `Order` project and make the relevant name changes. **Run your project.** It should behave the same as your last project.

Modify placeOrder in ApiResource

In ApiResource's placeOrder method replace the exception code

```
    throw new ApiException.InvalidParameter("Transactions have not been implemented yet.");
```

with

```
    if (orderId > 0) {
        return orderService.getOrderDetails(orderId);
    } else {
        throw new ApiException.InvalidParameter("Unknown error occurred");
```

Try Again

Add the DAO classes and OrderDetails to your server code

Unzip the `server-src-transact.zip` in the server folder. The code contains DAO classes and model classes for three tables in the database: `customer`, `customer_order`, and `customer_order_line_item`. Recall that `customer_order` was used instead of just `order` because *order* is a key word in SQL.

- Customer – model class for customer table
- CustomerDao – DAO interface (includes find *and* create methods)
- CustomerDaoJdbc – DAO implementation
- Order – model class for customer_order table
- OrderDao – DAO interface (find *and* create methods)
- OrderDaoJdbc – DAO implementation
- LineItem – model class for customer_order_line_item
- LineItemDao – DAO interface (find *and* create methods)
- LineItemDaoJdbc – DAO implementation
- OrderDetails – returned by REST API placeOrder method

If you don't already have static class BookstoreUpdateDbException inside BookstoreDbException, create it now. It should be similar to BookstoreQueryException.

If you have not connected to your database via IntelliJ's database view, do that now also. It will allow IntelliJ to make addition static checks on the SQL constants in your DAO implementations.

Add `OrderDao`, `LineItemDao` and `CustomerDao` instances to the `ApplicationContext` and `DefaultOrderService`. Add setter methods as well and wire in instances of the classes into `DefaultOrderService` following the pattern we established with the `BookDao` already.

Modify DefaultOrderService

Add getOrderDetails method

Implement DefaultOrderService getOrderDetails method.

Try Again

```
    Order order = orderDao.findByOrderId(orderId);
    Customer customer = customerDao.findByCustomerId(order.getCustomerId());
    List<LineItem> lineItems = lineItemDao.findByOrderId(orderId);
    List<Book> books = lineItems
            .stream()
            .map(lineItem -> bookDao.findByBookId(lineItem.getBookId()))
            .collect(Collectors.toList());
    return new OrderDetails(order, customer, lineItems, books);
}
```

Run your project. Everything should compile, but because we have not implemented transactions yet this code will still fail with an unknown error.

Add code to the placeOrder method

Add the following code after your validation code in the placeOrder method.

```
    try (Connection connection = JdbcUtils.getConnection()) {
        Date date = getDate(
                customerForm.getCcExpiryMonth(),
                customerForm.getCcExpiryYear());
        return performPlaceOrderTransaction(
                customerForm.getName(),
                customerForm.getAddress(),
                customerForm.getPhone(),
                customerForm.getEmail(),
                customerForm.getCcNumber(),
                date, cart, connection);
    } catch (SQLException e) {
        throw new BookstoreDbException("Error during close connection for customer order", e);
    }
```

For this to work, we need the method `getDate` that constructs a java.util.Date object from the month and year in the customer form.

```
private Date getDate(String monthString, String yearString) {
    return new Date(); // TODO Implement this correctly
}
```

The method placeOrder also defers the main transaction code to its own method: `performPlaceOrderTransaction`. To carry out the transaction, we will create rows in three separate tables: `customer`, `customer_order` and `customer_order_line_item`. For a successful

Try Again

1. First, we create a new customer in the customer table. We will create a new customer for every single order, even if the customer information is the same as a customer from a previous order. When we create the customer row in the table, we get back a customer ID that is auto-generated by the database.
2. Second, we create a new order in the customer-order table. Note that we had to call the table customer-order because "order" is a key-word in SQL. An order requires a customer ID, which is why we had to create the customer row first. When we create the order row in the table, we get back an order ID.
3. Finally, we create an order-line-item for each book in the cart. An order-line-item requires a customer-order ID, which is why we had to do the previous step first.

If everything goes well, we can commit the transaction, which means that all the data is officially entered into the database and can be accessed. If anything went wrong, we will NOT commit the transaction, and the data that we made room for in the database will be rolled back.

```
private long performPlaceOrderTransaction(
        String name, String address, String phone,
        String email, String ccNumber, Date date,
        ShoppingCart cart, Connection connection) {
    try {
        connection.setAutoCommit(false);
        long customerId = customerDao.create(
                connection, name, address, phone, email,
                ccNumber, date);
        long customerOrderId = orderDao.create(
                connection,
                cart.getComputedSubtotal() + cart.getSurcharge(),
                generateConfirmationNumber(), customerId);
        for (ShoppingCartItem item : cart.getItems()) {
            lineItemDao.create(connection, customerOrderId,
                               item.getBookId(), item.getQuantity());
        }
        connection.commit();
        return customerOrderId;
    } catch (Exception e) {
        try {
            connection.rollback();
        } catch (SQLException e1) {
            throw new BookstoreDbException("Failed to roll back transaction", e1);
        }
        return 0;
    }
```

Try Again

The method `generateConfirmationNumber` can be implemented by returning `random.nextInt(999999999)`. You will have to declare and instantiate a random field using the type `Random`.

Find the errors in the DAO classes

There are at least 4 errors in the DAO classes we gave you that will keep you from successfully placing an order. The easiest way to fix these is to attend the Q&A session or listen to the recording.

Create a confirmation page using the order details

- **Confirmation.vue** (https://drive.google.com/file/d/1WgJvjbVRG4-MBiVSFt8ILSacLse5SjzI/view?usp=sharing)
- **ConfirmationTable.vue** (https://drive.google.com/file/d/1RH7dlvdFwTp0EuOOA6eOkHt9XPEYOjOJ/view?usp=sharing)

Use the order details to create a confirmation page. The confirmation page should contain the confirmation number, the date-time stamp for the transaction, the cart information (book name, quantity, price), surcharge and total. You should also include customer information including name, email, address, last four of credit-card, and expiration date.

Feel free to use the Dr. K's Confirmation.vue and ConfirmationTable.vue files above as a starting point, but do not submit them with the exact same styling. They should be modified to fit your own site and your own style. Subtotals and totals are not included in the pages, though you **will** need to include them (see below).

The files depend on "orderDetails" being in your Vuex store. Regardless of whether you use the files or not, you should do the following to store.js to save the order details while you are on the checkout page so that you can use them when you are on the confirmation page.

- Create data "orderDetails" with a default value of null
- Create two mutators: CLEAR_ORDER_DETAILS and SET_ORDER_DETAILS. Clearing sets it to null, and setting sets it to a specified value (you'll need a parameter).
- Update your "placeOrder" action, so that the value returned (the order details) are used to set the order detail in the store.

Please note that the full credit card number should not be displayed – only the last four digits. The price for each book should be calculated from the unit price times the quantity. Pay attention to spacing and alignment, and remember that prices should always be aligned on cents. A credit-card expiration date includes only the month and year, not the day.

Try Again

Dr. K will be doing a final style review of your final project. Please see @413 on Piazza for details.

---

http://cs5244.cs.vt.edu:8080/GraceBookstoreTransact/

Try Again