CA2 Individual Report

|  |  |
| --- | --- |
| Name | Lois Poh |
| Student Id | P2429478 |
| Class | DIT/FT/2B/21 |
| Github Repository URL | <https://github.com/soc-DBS/dbs-assignment-crypt-soil> |
| Github Account ID | crypt-soil |

For each criterion, provide links to pull requests/commits/files that demonstrate the completion of the requirement. Replace each “**?**” with your Self Rating.

For Self Rating, you may rate yourself accordingly if you feel that you:

1. Have little or **no** understanding. and did not attempt the requirement.
2. Have **limited** understanding to demonstrate competency for the criterion.
3. Have **basic** understanding and only able to replicate examples from tutorials/practicals.
4. Have **adequate** understanding and can extend from what you have learned to fulfil specifications.
5. Have **solid** understanding in the specific criterion, able work on the requirement without much references.
6. Have **excellent** understanding and implemented the requirement according to latest industry guidelines, best practices and documentations.

**Important**

1. You are required to provide for each criterion:
   * **Documentation** and description of the work done.
   * **One to three** of your best implementations with URL **link** to respective repository files/commits/pull requests.
   * You should also provide **screenshots** where relevant.
2. You are to ensure the hyperlink in this document works. **Failure to do so will result in a 50% deduction of marks.**

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Criterion | Describe What Was Done | Self Rating |
| 1 | Database Design & ORM Modeling | * *Provide a clear screenshot of your ERD. ERD should be diagrammed on Lucid Chart and Screenshot should be based on the corresponding image exported.* * *What are the new models defined in the schema file?* * *Provide link to schema file* * *What are* ***additional*** *assumptions you made and reasons for your design decision(s)?*   A diagram of a computer program  AI-generated content may be incorrect.  <http://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/prisma/schema.prisma> | 5/5 |
| 2 | Cart Management feature Implementation | * *What are the backend functions (eg ORM functions, model, controller, route) & the front-end changes? (eg html, js)?* * *Provide links to relevant files/commits/pull requests of the above* * *Any screenshots of how you test for errors and success conditions?*   Model:   * createCartItem(memberId, productId, quantity)   + Used upsert so that if the same product already exists in the cart, it increments quantity instead of throwing a unique constraint error.   + Added automatic cart creation if it doesn’t exist for the member. * deleteCartItem(memberId, productId) * Checks if the cart exists for the member, then deletes the specific product from that cart. * updateCartItem(memberId, productId, quantity)   + Updates quantity for the given product in the user’s cart.   Controller:   * createCartItems   + Validates request data (memberId, productId, quantity)   + Checks product stock availability using productModel.getProductById   + Calls cartsModel.createCartItem to insert/upsert item into cart * updateCartItems   + Validates and updates quantity in cart via cartsModel.updateCartItem * retrieveCartItems   + Fetches cart items via cartsModel.getCartItems * deleteCartItems   + Validates input and removes an item from the cart via cartsModel.deleteCartItem * getCartSummary   + Aggregates total quantity and price via cartsModel.getCartSummary * getCheckout   + Loads checkout details via cartsModel.getCheckoutData * placeOrder   + Places order via cartsModel.placeOrder after cart validation * orderStatus   + Gets order status via cartsModel.getOrderStatus   Routes:  Corresponding routes were made for create, delete and update  Index.js:   * **public/cart/create/index.js**   + Handles form submission to /carts/create with token authentication   + Validates fields before sending   + Displays server responses in #message * **public/cart/retrieve/all/index.js**   + Fetches cart items and appends them to the table * **Delete** button:   + Confirms action   + Sends DELETE /carts request with productId in JSON body   + Reloads page on success, alerts on failure without reloading * **Update** button:   + Sends PUT /carts with updated quantity   + Alerts on success/failure   + Fetches summary from /carts/summary   + Redirects to checkout page on checkout button click   Link to files:   * Controller: controllers/cartsController.js * <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/controllers/cartsController.js> * Model: models/carts.js * <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/models/carts.js> * Routes: routes/carts.js   <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/routes/carts.js>   * Frontend JS: * public/cart/create/index.js   <https://github.com/soc-DBS/dbs-assignment-crypt-soil/tree/master/public/cart/create>   * public/cart/retrieve/all/index.js   <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/public/cart/retrieve/all/index.js>  Screenshots:  Success:    Error:  User input is more than available stock    Product is out of stock    Cart is empty: | 4.5/5 |
| 3 | Checkout feature Implementation | * *What are the backend functions (eg ORM functions, model, controller, route) & the front-end changes? (eg html, js)?* * *Provide links to relevant files/commits/pull requests of the above* * *Any screenshots of how you test for errors and different conditions/scenarios/business rules?* * *How flexible or extensible is your implementation of the feature?*   **1. Backend Functions**  **a. Routes**   * GET /checkout → Fetches checkout page data (cartsController.getCheckout) * POST /checkout/placeOrder → Places the order after validating cart is not empty (cartsController.placeOrder) * GET /carts/summary → Returns total quantity and total price for checkout summary (cartsController.getCartSummary)   **b. Controllers**   * **getCheckout**   + Reads memberId from JWT.   + Calls cartsModel.getCheckoutData(memberId) to retrieve all cart items, discounts, subtotal, and total.   + Returns { success: true, data: {...} } or error message. * **placeOrder**   + Reads memberId from JWT.   + Checks if cart has items.   + Calls cartsModel.placeOrder(memberId) (executes stored procedure).   + Returns { orderId, message }. * **getCartSummary**   + Aggregates total quantity and price from cartsModel.getCartSummary(memberId).   **c. Models (Prisma ORM)**   * **getCheckoutData(memberId)**   + Joins cart → cartItem → product → discounts.   + Calculates subtotal, total discount, and grand total for each product line. * **placeOrder(memberId)**   + Calls Postgres stored procedure place\_orders(memberId) via prisma.$executeRaw.   + Returns latest order info from saleOrder.   **2. Front-End Changes**  **a. HTML**   * checkout.html   + Table to display checkout items (name, quantity, price, discount, total).   + Checkout summary section showing subtotal, total discount, and grand total.   + "Place Order" button with click handler.   **b. JavaScript**   * **checkout/index.js**   + On page load:     - Calls /checkout API to populate checkout table.     - Calls /carts/summary for summary display.   + On **Place Order** click:     - Calls /checkout/placeOrder.     - Alerts user if:       * Cart is empty ("Your cart is empty" error from backend).       * Order placed successfully ("Order placed successfully").   + Error handling:     - Displays API errors in alert without clearing the table.   Link to files:   * **Routes:**   + <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/routes/carts.js> * **Controllers:**   + <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/controllers/cartsController.js> * **Models:**   + <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/models/carts.js> * **Frontend JS:**   + <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/public/checkout/index.js> | 4/5 |
| 4 | Transaction Management | * *What is the stored procedure to manage transactions for place order?* * *Provide screenshots of how you test that it successfully handles different conditions.* * *What are the backend functions (eg ORM functions, model, controller, route) & the front-end changes (eg html, js)?* * *Provide links to relevant files/commits/pull requests of the above*  1. **Stored Procedure Overview**   **Name:** public.place\_orders(p\_member\_id integer, p\_preview boolean DEFAULT false) **Purpose:** Handles the entire **place order** process in one atomic transaction, including:   * Fetching cart items and any active discounts * Calculating subtotal, discounts, and grand total * Optionally previewing the order without committing changes (p\_preview = TRUE) * Updating stock quantities * Creating a new sale order and order items * Clearing purchased items from the cart  1. **Transaction Management**  * **Single Transaction:** The procedure runs inside a single transaction context, so:   + If any step fails (e.g., insufficient stock, database error), **no changes are committed**.   + Either the entire order is placed successfully, or nothing changes (atomicity). * **Stock Check Before Deduction:** Before updating product.stock\_quantity, the procedure checks if stock\_quantity >= ordered quantity. * **Automatic Cleanup:** Items are removed from cart\_item only if the stock update and order item insert are successful.  1. Testing Scenarios 2. Backend Functions 3. Frontend Changes  * Fetches /checkout data to display cart items. * On "Place Order" click and submit Place Order * Handles empty cart error messages returned from backend.   Success:  Successful order:    Show current order status:    Error:    Stored procedure flexibility:  My stored procedure is flexible because it supports both preview and commit modes, handles various discount types, dynamically validates stock, and adapts to different error scenarios without partial updates. It ensures that no matter the order condition—empty cart, insufficient stock, inactive discount—the database remains consistent, and the user receives an accurate response.  Link to relevant files:  Index.js:   * <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/public/checkout/index.js>   Routes:   * <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/routes/carts.js>   Stored procedure:   * <https://github.com/soc-DBS/dbs-assignment-crypt-soil/blob/master/functions_%26_stored_procedures_ca2.sql>   Stored procedure test scenarios:  Successful checkout:    Preview checkout (doesn’t reduce stock\_quantity)    Percentage discount    Fixed discount:    Inactive discount (no discount applied):    Does not meet minimum quantity: | 4.5/5 |
| 5 | Indexing | *For each of your proposed queries (between 3 to 6)*   1. *State what is the query proposed* 2. *State what* ***type*** *of index is used* ***and*** *what is the SQL statement used to create the index* 3. *Provide proof of improved query performance and* ***explain*** *the* ***key*** *reason(s) for the improvement*   Query 1:  Show all manufacturers from a given country, sorted alphabetically by name  SQL:   * + SELECT id, firm\_name, origin   FROM public.manufacturer  WHERE origin = 'Japan'  ORDER BY firm\_name;  Index type: Composite B-tree index   * *CREATE INDEX IF NOT EXISTS idx\_m\_origin\_firm\_name*   *ON public.manufacturer (origin, firm\_name);*  Before:    After:      Query 2:  Find manufacturers whose name contains a given pattern/string (case-insensitive), and order them A-Z  SELECT id, firm\_name  FROM public.manufacturer  WHERE lower(firm\_name) ILIKE '%' || lower($1) || '%'  ORDER BY firm\_name;  Index type: GIN trigram expression index. Used this instead of normal B-tree because B-tree don’t support/optimise %....% substring searches  CREATE EXTENSION IF NOT EXISTS pg\_trgm;  CREATE INDEX IF NOT EXISTS idx\_srch\_firm\_name\_trgm  ON public.manufacturer  USING GIN (lower(firm\_name) gin\_trgm\_ops);  Before:    After:    Reasons:  GIN Trigram Index stores 3-character chunks of lower(firm\_name), for %’smith’% into trigrams (for eg, ‘smi’, ‘mit’, ‘ith’) and uses the index only to fetch only the rows that contains those trigrams instead of scanning the whole table.  Changes from full table to scan to bitmap index scan of a smaller string, reducing time.  Query 3:   * Get only operational manufacturers and show get the Top 20 biggest (by employee count) companies * SQL query:   *SELECT id, firm\_name, employee\_count*  *FROM public.manufacturer*  *WHERE is\_operational = true*  *ORDER BY employee\_count DESC*  *LIMIT 20;*   * Partial B-tree index with a WHERE clause * CREATE INDEX IF NOT EXISTS idx\_m\_operational\_empcount\_desc   ON public.manufacturer (employee\_count DESC)  WHERE is\_operational = true;  Before:    After:    Reasons:  Partial index, it only stores rows where is\_operational = true, it skips all inactive rows completely at it is\_operational = false is not stored in the index at all.  Pre-sorted on employee\_count DESC, rows are already arranged from largest employee\_count to smallest. So PostgreSQL can just take 20 rows from the top of the index. | ?/5 |
| 9 | Report Quality | *Based on quality of documentation for above criteria.*  *No inputs required here.* | ?/5 |
| 10 | Demonstration & Interview | *Based on assessment during demonstration & interview.*  *No inputs required here.* | - |