System Requirements Specification

Index

For

E-Commerce

Version 1.0

TABLE OF CONTENTS

E-COMMERCE
System Requirements Specification

BACKEND-EXPRESS RESTFUL APPLICATION

1 PROJECT ABSTRACT

"**E-Commerce**" is an express js application designed to provide a seamless online shopping through products, cart and order management APIs with authentication enabled. It leverages the ExpressJs with MongoDB as the database. This platform aims to provide APIs on the management of products, cart and order, allowing users to browse, search for, and purchase a wide range of products.

Following is the requirement specifications:

	E-Commerce
Modules	
1	Product
2	Order
3	Middleware

Product Module Functionalities	
1	Get all products
2	Create a new product
3	Search the products
4	Get top rated products
5	Apply discount on cart
6	View the cart
7	Add item to the cart
8	Checkout the cart
9	Update an item in cart
10	Remove an item in cart
11	Get a product
12	Update a product
13	Delete a product

Order Module Functionalities	
1	Get all orders
2	Create an order
3	Get order by id
4	Update an order it’s id
5	Delete an order by it’s id
6	Get order for user
7	Cancel any order
8	Get the payment details of any order
9	Process the payment for any order
10	Get the analytics of orders with data for totalOrders, average order amount, highest and lowest order amount
11	Generate the invoice for any order with order id, order date and total amount details
12	Track the order shipment by returning status and tracking number
13	Get the revenue analytics which returns the total, highest and lowest revenue
14	Check whether the product is ordered or not

Middleware Module Functionalities	
1	Implement logic to check and validate token in authUserMiddlware file.
2	Implement logic to check whether a logged in user is admin or not in authAdminMiddleware.

2 Assumptions, Dependencies, Risks / Constraints

2.1 Order Constraints

1. When creating an order userId, product, totalAmount, paymentDetails and shipmentDetails are mandatory fields, on failing it should throw a custom exception with message as “Failed to create order.”.
2. When fetching an order by ID, if the order ID does not exist, the operation should throw a custom exception with message as “Order not found.”.
3. When updating an order, if the order ID does not exist, the operation should throw a custom exception with message as “Order not found.”.
4. When removing an order, if the order ID does not exist, the operation should throw a custom exception with message as “Order not found.”.
5. When canceling an order, if the order ID does not exist, it should throw a custom exception with message as “Order not found.”.
6. When fetching payment details for an order, if the order ID does not exist, it should throw a custom exception with message as “Order not found.”.
7. When processing the payment for an order, if the order ID does not exist, it should throw a custom exception with message as “Order not found.”.
8. When generating an invoice for an order, if the order ID does not exist, it should throw a custom exception with message as “Order not found.”.
9. When tracking an order, if the order ID does not exist, it should throw a custom exception with message as “Order not found.”.

2.2 Product Constraints

1. When creating product name and price are mandatory fields, on failing it should throw a custom exception with message as “Failed to create product.”.
2. When fetching a product by ID, if the product ID does not exist, the operation should throw a custom exception with message as “Product not found.”.
3. When updating a product, if the product ID does not exist, the operation should throw a custom exception with message as “Product not found.”.
4. When removing a product, if the product ID does not exist, the operation should throw a custom exception with message as “Product not found.”.

5. When searching a product, if the name or description does not exist, it should throw a custom exception with message as “Failed to search for products.”.
6. When applying some discount on a product, if the discount percentage is invalid (0 > discount or discount > 100), it should throw a custom exception with message as “Invalid discount percentage.”.
7. When applying some discount on a product, if the discount percentage does not exist, it should throw a custom exception with message as “Failed to apply discount.”.
8. When checking out a cart, if the payment method and address does not exist, it should throw a custom exception with message as “Failed to complete checkout.”.
9. When adding a product in cart, if the user, product, quantity and price does not exist, it should throw a custom exception with message as “Failed to add products to the cart.”.
10. When fetching cart details, if the user does not exist, it should throw a custom exception with message as “Failed to view the contents of the cart.”.
11. When updating a cart item, if the user, item and quantity does not exist, it should throw a custom exception with message as “Failed to update item in the cart.”.
12. When removing an item from the cart, if the user and item does not exist, it should throw a custom exception with message as “Failed to remove item from the cart.”.

Common Constraints

- All the database operations must be implemented in serviceImpl file only.
- Do not change, add, remove any existing methods in the service file.
- In the service layer, custom methods can be added as per requirements.
- All RestEndpoint methods and Exception Handlers must return data in json format.
- Any type of authentication and authorisation must be added in routes file only.

3 REST ENDPOINTS

Rest End-points to be exposed in the routes file and attached with controller method along with method details for the same to be created. Please note, that these all are required to be implemented.

3.1 ORDER RESTPOINTS

Note: All order routes must be authenticated and only accessible on valid token.

URL Exposed		Purpose
1. /api/orders/all		Fetches all the orders
Http Method	GET	
Parameter	-	
Return	list all orders	
2. /api/orders/create		Creates a new order
Http Method	POST	
Parameter	-	
Return	newly created order	

3. /api/orders/:id		Fetches an order by id
Http Method	GET	
Parameter	id	
Return	get an order	
4. /api/orders/:id		Updates an order by id
Http Method	PUT	
Parameter	id	
Return	updates an order	

5. /api/orders/:id		Deletes an order by id
Http Method	DELETE	
Parameter	id	
Return	deletes an order	
6. /api/orders/user/:userId		Fetches an order for that user
Http Method	GET	
Parameter	userId	
Return	order	

7. /api/orders/cancel/:id		Cancels an orders
Http Method	DELETE	
Parameter	id	
Return	canceled order	

8. /api/orders/:id/payment

Http Method	GET
Parameter	-
Return	return the payment details

Get the payment details of any order

9. /api/orders/:id/pay

Http Method	GET
Parameter	-
Return	message whether payment is processed or not

Process the payment for any order

10. /api/orders/analytics

Http Method	GET
Parameter	-
Return	order analytics

Get the analytics of orders with data for totalOrders, average order amount, highest and lowest order amount

11. /api/orders/:id/invoice

Http Method	GET
Parameter	-
Return	invoice

Generate the invoice for any order with order id, order date and total amount details

12. /api/orders/:id/shipment

Http Method	GET
Parameter	-
Return	shipment details

Track the order shipment by returning status and tracking number

13. /api/orders/revenue

Http Method	GET
Parameter	-
Return	analytics

Get the revenue analytics which returns the total, highest and lowest revenue

14. /api/orders/ordered/:userId/:productId

Http Method	GET
Parameter	userId, productId
Return	returns order

Check whether the product is ordered or not

3.2 Product Restpoints

Note:

Any route which must be authenticated (there must be valid token) are marked with authMiddleware

Any route which must be authorized only for admin (user must be of type admin) are marked with adminMiddleware.

Any route which must be both authenticated and authorized are marked with authMiddleware and adminMiddleware.

URL Exposed		Purpose
1. /api/products/all		Fetches all the products
Http Method	GET	
Parameter	-	
Return	list of products	
2. /api/products/create		Creates a new product **[authMiddleware, adminMiddleware]**
Http Method	POST	
Parameter	-	
Return	newly created product	
3. /api/products/search		Search the product by name or description
Http Method	GET	
Parameter	-	
Return	searched products	
4. /api/products/top-rated/:limit		Fetches the top rated products **[authMiddleware]**
Http Method	GET	
Parameter	limit	
Return	list of products	
5. /api/products/discount/:userId		Apply discount on cart **[authMiddleware]**
Http Method	POST	
Parameter	userId	
Return	updated cart	
6. /api/products/cart/:userId		View the cart for that user **[authMiddleware]**
Http Method	GET	
Parameter	userId	
Return	returns the cart	

7. /api/products/cart/add/:userId

Http Method	POST
Parameter	userid
Return	updated cart

Adds item in the cart
[authMiddleware]

8. /api/products/cart/checkout/:userId

Http Method	POST
Parameter	userId
Return	return successful message with created order id

Checkout the cart
[authMiddleware]

9. /api/products/cart/update/:useId/:itemId

Http Method	PUT
Parameter	userId, itemId
Return	updated cart

Updates an item in cart
[authMiddleware]

10. /api/products/cart/remove/:userId/:itemId

Http Method	DELETE
Parameter	userId, itemid
Return	removed item

Remove an item in cart
[authMiddleware]

11. /api/products/:id

Http Method	GET
Parameter	id
Return	product

Gets the product by id

12. /api/products/:id

Http Method	PUT
Parameter	id
Return	updated product

Updated the product by id

13. /api/products/:id

Http Method	DELETE
Parameter	id
Return	deleted product

Deletes the product by id
[authMiddleware, adminMiddleware]

4 Template Code Structure

4.1 Products code structure

1) MODULES/PRODUCTS: controller

Resources

ProductController **(Class)**	This is the controller class for the product module.	To be implemented

2) MODULES/PRODUCTS: dao

Resources

File	**Description**	**Status**
models/cart model **models/product model**	Models for cart and product	Already implemented
schemas/cart schema **schemas/product schema**	Schemas for cart and product	Already implemented

3) MODULES/PRODUCTS: routes

Resources

File	**Description**	**Status**
Product routes	Routes for product	Partially implemented.

4) MODULES/PRODUCTS: service

Resources

Class	**Description**	**Status**
ProductService	● Defines ProductService	Already implemented.

5) MODULES/PRODUCTS: service/impl

Resources

Class	Description	Status
ProductServiceImpl	● Implements ProductService.	To be implemented.

.

4.2 Orders code structure

1) MODULES/ORDER: controller

Resources

OrderController **(Class)**	This is the controller class for the order module.	To be implemented

2) MODULES/ORDER: dao

Resources

File	Description	Status
models/order model	Model for order	Already implemented
schemas/order schema	Schema for order	Already implemented

3) MODULES/ORDER: routes

Resources

File	Description	Status
Order routes	Routes for order	Partially implemented.

4) MODULES/ORDER: service

Resources

Class	Description	Status
OrderService	● Defines OrderService	Already implemented.

5) MODULES/ORDER: service/impl

Resources

Class	**Description**	**Status**
OrderServiceImpl	● Implements OrderService.	To be implemented.

5 METHOD DESCRIPTIONS

5.1 PRODUCT - Method Descriptions:

1. ProductController Class - Method Descriptions(Based on Route Mapping):

Method	**Task**	**Implementation Details**
createProduct	To create a new product	- The request type should be POST with URL /api/products/create. - Requires authentication and admin privileges. - Call productService.createProduct(req.body). - Return 201 status with created product. - On error, respond with 500 and message: 'Failed to create product.'
getProduct	To retrieve a product by ID	- The request type should be GET with URL /api/products/:id. - Call productService.getProduct(req.params.id). - Return the product. - On error, respond with 404 and message: 'Product not found.'
updateProduct	To update a product by ID	- The request type should be PUT with URL /api/products/:id. - Call productService.updateProduct(req.params.id, req.body). - Return the updated product. - On error, respond with 404 and message: 'Product not found.'

deleteProduct	To delete a product by ID	- The request type should be DELETE with URL /api/products/:id. - Requires authentication and admin privileges. - Call productService.deleteProduct(req.params.id). - Return the deleted product. - On error, respond with 404 and message: 'Product not found.'
getAllProducts	To fetch all products	- The request type should be GET with URL /api/products/all. - Call productService.getAllProducts(). - Return all products. - On error, respond with 500 and message: 'Failed to retrieve products.'
getTopRatedProducts	To get top-rated products with limit	- The request type should be GET with URL /api/products/top-rated/:limit. - Requires authentication. - Call productService.getTopRatedProducts(limit). - Return top-rated products. - On error, respond with 500 and message: 'Failed to retrieve top rated products.'
searchProduct	To search products by name/description	- The request type should be GET with URL /api/products/search. - Call productService.searchProduct(name, description). - Return matching products. - On error, respond with 500 and message: 'Failed to search products.'
applyDiscount	To apply discount for a user's cart	- The request type should be POST with URL /api/products/discount/:userId. - Requires authentication. - Call productService.applyDiscount(userId, discountPercentage). - Return discount result. - On error, respond with 500 and message: 'Failed to apply discount.'
checkoutCart	To checkout a user's cart	- The request type should be POST with URL /api/products/cart/checkout/:userId. - Requires authentication. - Call productService.checkoutCart(userId, paymentMethod, address). - Return checkout result.

		- On error, respond with 500 and message: 'Failed to checkout cart.'
addToCart	To add a product to a user's cart	- The request type should be POST with URL /api/products/cart/add/:userId. - Requires authentication. - Call productService.addToCart(userId, productId, quantity, price). - Return updated cart. - On error, respond with 500 and message: 'Failed to add to cart.'
viewCart	To view the contents of a user's cart	- The request type should be GET with URL /api/products/cart/:userId. - Requires authentication. - Call productService.viewCart(userId). - Return cart details. - On error, respond with 500 and message: 'Failed to view cart.'
updateCartItem	To update quantity of a cart item	- The request type should be PUT with URL /api/products/cart/update/:userId/:itemId. - Requires authentication. - Call productService.updateCartItem(userId, itemId, quantity). - Return update result. - On error, respond with 500 and message: 'Failed to update cart item.'
removeCartItem	To remove an item from a cart	- The request type should be DELETE with URL /api/products/cart/remove/:userId/:itemId. - Requires authentication. - Call productService.removeCartItem(userId, itemId). - Return removal result. - On error, respond with 500 and message: 'Failed to remove cart item.'

2. ProductServiceImpl Class - Method Descriptions:

Method	Task	Implementation Details
createProduct	To create a new product	- Accept product data and create a new product using Product.create(). - Return the newly created product. - On error, throw: 'Failed to create product.'
getProduct	To retrieve a product by its ID	- Use Product.findById(productId). - If not found, throw: 'Product not found.' - Return the found product. - On error, throw: 'Failed to get product.'
updateProduct	To update an existing product by ID	- Use Product.findByIdAndUpdate(productId, updatedProduct, { new: true }). - If not found, throw: 'Product not found.' - Return the updated product. - On error, throw: 'Failed to update product.'
deleteProduct	To delete a product by ID	- Use Product.findByIdAndDelete(productId). - If not found, throw: 'Product not found.' - Return the deleted product. - On error, throw: 'Failed to delete product.'
searchProduct	To search products based on name and/or description	- Build query object using regex for name and description fields. - Use Product.find(query) to search. - Return the list of matching products. - On error, throw: 'Failed to search for products.'
getTopRatedProducts	To retrieve top-rated products up to a limit	- Use Product.find().sort({ ratings: -1 }).limit(limit). - Return the list of top-rated products. - On error, throw: 'Failed to get top rated products.'
getAllProducts	To retrieve all products	- Use Product.find() to retrieve all products. - Return the list of all products. - On error, throw: 'Failed to get all products.'

applyDiscount	To apply a discount to all products in the user's cart	- Validate the cart and discount percentage. - Apply discount to item prices and save the cart. - Return a success message. - On error, throw: 'Failed to apply discount.'
checkoutCart	To place an order and clear the cart	- Retrieve user's cart and calculate total amount. - Find cart as per userId, if not found throw “Cart not found.”. - Delete the user's cart. - Return a success message and the created order ID. - On error, throw: 'Failed to complete checkout.'
addToCart	To add a product to a user's cart	- If item exists, increment quantity. - Else, add new item to the cart using $addToSet. - Return the updated cart. - On error, throw: 'Failed to add products to the cart.'
viewCart	To view a user's cart and product details	- Use Cart.findOne({ userId }).populate('items.product'). - Return the user's cart with product details. - On error, throw: 'Failed to view the contents of the cart.'
updateCartItem	To update quantity of a cart item	- Use Cart.findOneAndUpdate() with itemId and quantity. - Return the updated cart. - On error, throw: 'Failed to update item in the cart.'
removeCartItem	To remove an item from the cart	- Use $pull to remove item by itemId. - Return the updated cart. - On error, throw: 'Failed to remove item from the cart.'

5.2 ORDER - Method Descriptions:

1. OrderController Class - Method Descriptions(Based on Route Mapping):

Method	**Task**	**Implementation Details**
getAllOrders	To retrieve all orders	- The request type should be GET with URL /api/orders/all. - Requires authentication. - Call orderService.getAllOrders(req.body). - Return list of all orders. - On error, respond with 500 and message: 'Failed to create order.'
createOrder	To create a new order	- The request type should be POST with URL /api/orders/create. - Call orderService.createOrder(req.body). - Return the created order with status 201. - On error, respond with 500 and message: 'Failed to create order.'
getOrder	To get a specific order by ID	- The request type should be GET with URL /api/orders/:id. - Call orderService.getOrder(req.params.id). - Return the order. - On error, respond with 404 and message: 'Order not found.'
updateOrder	To update an existing order by ID	- The request type should be PUT with URL /api/orders/:id. - Call orderService.updateOrder(req.params.id, req.body). - Return the updated order. - On error, respond with 404 and message: 'Order not found.'
deleteOrder	To delete an order by ID	- The request type should be DELETE with URL /api/orders/:id. - Call orderService.deleteOrder(req.params.id). - Return the deleted order. - On error, respond with 404 and message: 'Order not found.'

getUserOrders	To get all orders placed by a specific user	- The request type should be GET with URL /api/orders/user/:userId. - Call orderService.getUserOrders(userId). - Return list of user orders. - On error, respond with 500 and message: 'Failed to retrieve user orders.'
cancelOrder	To cancel a specific order by ID	- The request type should be DELETE with URL /api/orders/cancel/:id. - Call orderService.cancelOrder(orderId). - Return the canceled order. - On error, respond with 404 and message: 'Order not found.'
getPaymentDetails	To retrieve payment details of an order	- The request type should be GET with URL /api/orders/:id/payment. - Call orderService.retrievePaymentDetails(orderId). - Return payment details. - On error, respond with 404 and message: 'Order not found.'
processPayment	To process payment for a specific order	- The request type should be POST with URL /api/orders/:id/pay. - Call orderService.processPayment(orderId). - Return success message “Payment processed successfully.”. - On error, respond with 500 and message: 'Failed to process payment.'
getOrderAnalytics	To retrieve analytics on orders	- The request type should be GET with URL /api/orders/analytics. - Call orderService.getOrderAnalytics(). - Return order analytics data. - On error, respond with 500 and message: 'Failed to retrieve order analytics.'
generateInvoice	To generate invoice for an order	- The request type should be GET with URL /api/orders/:id/invoice. - Call orderService.generateInvoice(orderId). - Return invoice details. - On error, respond with 500 and message: 'Failed to generate invoice.'

trackShipment	To track shipment of an order	- The request type should be GET with URL /api/orders/:id/shipment. - Call orderService.trackShipment(orderId). - Return shipment tracking details. - On error, respond with 500 and message: 'Failed to track shipment.'
getRevenueAnalytics	To retrieve revenue analytics	- The request type should be GET with URL /api/orders/revenue. - Call orderService.getRevenueAnalytics(). - Return revenue analytics data. - On error, respond with 500 and message: 'Failed to retrieve revenue analytics.'
hasOrderedProduct	To check if a user has ordered a specific product	- The request type should be GET with URL /api/orders/ordered/:userId/:productId. - Call orderService.hasOrderedProduct(userId, productId). - Return boolean value or relevant result. - On error, respond with 500 and message: 'Failed to retrieve information about this product for this user'

2. OrderServiceImpl Class - Method Descriptions:

Method	**Task**	**Implementation Details**
createOrder	To create a new order	- Accept order data and create the order using Order.create(). - Return the created order. - On error, throw: 'Failed to create order.'
getAllOrders	To retrieve all orders	- Use Order.find() to fetch all orders. - Return list of all orders. - On error, throw: 'Failed to fetch all orders.'
getOrder	To retrieve a specific order by ID	- Use Order.findById(orderId). - If not found, throw: 'Order not found.' - Return the found order. - On error, throw: 'Failed to get order.'

updateOrder	To update an existing order by ID	- Use Order.findByIdAndUpdate(orderId, updatedOrder, { new: true }). - If not found, throw: 'Order not found.' - Return the updated order. - On error, throw: 'Failed to update order.'
deleteOrder	To delete an order by ID	- Use Order.findByIdAndDelete(orderId). - If not found, throw: 'Order not found.' - Return the deleted order. - On error, throw: 'Failed to delete order.'
getUserOrders	To retrieve all orders placed by a specific user	- Use Order.find({ userId }) to retrieve user's orders. - Return list of user's orders. - On error, throw: 'Failed to retrieve user orders.'
cancelOrder	To cancel an order by ID	- Use Order.findByIdAndDelete(orderId). - If not found, throw: 'Order not found.' - Return the canceled order. - On error, throw: 'Failed to cancel order.'
retrievePaymentDetails	To get payment details of a specific order	- Use Order.findById(orderId, 'payment'). - If not found, throw: 'Order not found.' - Return the payment details. - On error, throw: 'Failed to retrieve payment details.'
processPayment	To mark the order as paid	- Use Order.findById(orderId), set payment status and transactionId. - Save the order. - Return success message as “Payment processed successfully.”. - On error, throw: 'Failed to process payment.'
getOrderAnalytics	To compute analytics based on all orders	- Calculate total, average, highest and lowest order amounts. - Return order analytics object. - On error, throw: 'Failed to retrieve order analytics.'

generateInvoice	To generate invoice data for an order	- Use Order.findById(orderId) to get order details. - If not found, throw: 'Order not found.' - Return basic invoice details. - On error, throw: 'Failed to generate invoice.'
trackShipment	To provide tracking details for an order	- Use Order.findById(orderId). - If not found, throw: 'Order not found.' - Return static shipment tracking information. - On error, throw: 'Failed to track shipment.'
getRevenueAnalytics	To compute revenue statistics from all orders	- Calculate total, highest, and lowest revenue from Order data. - Return revenue analytics object. - On error, throw: 'Failed to retrieve revenue analytics.'
hasOrderedProduct	To check if a user has ever ordered a specific product	- Use Order.findOne with user and product conditions. - Return true if order exists, otherwise false. - On error, return false.

5.3 MIDDLEWARE - Method Descriptions:

1. AuthAdminMiddleware - Method Descriptions:

Middleware	Purpose	Implementation Details
adminMiddleware	To restrict access to routes that require admin privileges	- This middleware checks if req.user.isAdmin equals 1. - If true, it allows the request to proceed by calling next(). - If false, it returns a 403 Forbidden status with the message: 'Access forbidden. Admin privileges required.' - Return: Continues to next middleware or route if authorized; otherwise, returns an error response.

2. AuthUserMiddleware - Method Descriptions:

Middleware	Purpose	Implementation Details
authMiddleware	To authenticate users using a JWT token	- Extracts the token from the 'Authorization' header. - If token is missing, responds with 401 status and message: 'Unauthorized'. - Verifies the token using jwt.verify with a secret key. - If valid, sets req.user to the decoded token's user object and proceeds to the next middleware. - If invalid or expired, responds with 401 status and message: 'Unauthorized'. - Return: Proceeds with the request if token is valid; otherwise, returns an error response.

EXECUTION STEPS TO FOLLOW FOR BACKEND

1. **All actions like build, compile, running application, running test cases will be through Command Terminal.**
2. **To open the command terminal the test takers, need to go to**

   Application menu (Three horizontal lines at left top) -> Terminal ->New Terminal.
3. **This editor Auto Saves the code.**
4. **If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page) then you need to use CTRL+Shift+B-command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.**
5. **These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.**
6. **To test any Restful application, the last option on the left panel of IDE, you can find**

ThunderClient, which is the lightweight equivalent of POSTMAN.

7. **You can follow series of command to setup express environment once you are in your project-name folder:**
    a. **npm install -> Will install all dependencies -> takes 10 to 15 min**
    b. **npm run start -> To compile and run the project.**
    c. **npm run jest -> to run all test cases and see the summary of all passed and failed test cases.**
    d. **npm run test -> to run all test cases and register the result of all test cases. It is mandatory to run this command before submission of workspace -> takes 5 to 6 min**
8. **You need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.**