Test Plan - Unit Test

1. Introduction

1.1. Objective

This Test Plan outlines the strategy and scope for unit testing within the project. It is intended for software developers, QA engineers, and project stakeholders involved in the development and quality assurance processes. The document specifically addresses unit testing activities and does not serve as the main test plan for the entire project. Instead, it focuses on the early-stage verification of individual software components—such as functions, methods, or classes—ensuring that each unit performs as expected in isolation.

The primary objectives of this unit testing effort are to:

- Detect and resolve defects at an early development stage.
- Validate the correctness of each module independently of others.
- Establish a reliable foundation for subsequent testing phases, including integration and system testing.

Security and privacy considerations include ensuring that no sensitive data is hardcoded or exposed during unit test execution and that all test data used complies with relevant data protection standards.

1.2. Scope

The software product to be developed is **AIMS (An Internet Media Store)**, a digital platform that enables users to browse, purchase, and download a wide range of media content, including applications, music, videos, and e-books. AIMS serves as a centralized marketplace designed to deliver a seamless and secure media consumption experience across web and mobile platforms.

AIMS will allow users to create accounts, search and filter media content, manage their purchases, and access their media library anytime. The platform will also support content uploads and management for verified publishers. Its primary goals include enhancing media accessibility, streamlining digital content distribution, and supporting user-friendly interactions.

AIMS will not handle content creation or moderation beyond basic automated checks; these responsibilities lie with the content providers and external policies. This overview is consistent with the system’s Software Requirements Specification (SRS) and outlines the scope, benefits, and intended use of the application without detailing individual requirements.

1.3. Glossary

No	*Term*	*Explanation*	*Example*	*Note*
1	token	A piece of data created by server, and contains the user's information, as well as a special token code that user can pass to the server with every method that supports authentication, instead of passing a username and password directly.	JSON Web Token (JWT)	Compact, URL-safe and usable especially in web browser single sign-on (SSO) context.
2	API (Application Programming Interface)	A set of rules that allows different software applications to communicate with each order	VNPay API	Includes online shopping, digital payments, and logistics
3	E-Commerce (Electronic Commerce)	The buying and selling of goods and services over the internet	Amazon, Shopee	Includes online shopping, digital payment, and logistics
4	VAT (Value Added Tax	Value-added tax, a consumption tax added to goods and services.	10% VAT in Vietnam	Applied at each stage of production and distribution

5	Rush Order	An order that requires expedited processing and delivery.	Express shipping, Same-day delivery.	May involve additional fees for faster service.
6	Payment Gateway	A service that authorizes and processes online payments securely.	VNPay	Ensures secure transactions between customers and merchants.

1.4. Reference

Centers for Medicare & Medicaid Services. (n.d.). Test Case Specification. Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/TestCaseSpecification.docx

Centers for Medicare & Medicaid Services. (n.d.). Test Plan. Retrieved from Centers for Medicare & Medicaid Services: https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/TestPlan.docx

guru99. (n.d.). Download Sample Test Case Template: Example Excel, Word Formats. Retrieved from guru99: https://www.guru99.com/download-sample-test-case-template-with-explanation-of-important-fields.html

2. Overall Description

2.1. General Overview

2.1.1. System Context and Design Philosophy

The design of AIMS (An Internet Media Store) centers around delivering a robust and scalable e-commerce platform for physical media products, tailored to meet both business requirements and user expectations. At its core, the system is built to be reliable,

responsive, and easy to use — even under demanding conditions. Key design objectives include:

- **High Performance**: The system is expected to handle user requests swiftly, ensuring responsive interactions even during peak traffic periods.

- **Scalability**: AIMS aims to support up to 1,000 concurrent users, providing a seamless experience regardless of user volume.

- **Reliability**: The system should remain operational for at least 300 hours continuously and be able to recover within 1 hour following any downtime.

- **Security**: Strong security practices are integrated, including encrypted data storage and secure user authentication, to protect user privacy and transaction safety.

2.1.2. System and Software Architecture

- **Client Application**: A modern, web-based interface built using technologies like React, HTML, CSS, and JavaScript. It enables users to browse products, manage shopping carts, and complete purchases with ease.

- **Server Application**: The server-side logic is implemented using the SpringBoot framework, which handles business operations, database management, and integration with third-party services.

- **Database**: A relational database serves as the backbone for storing user accounts, product listings, order details, and transaction history.

- **External Integration**: The system integrates with **VNPay**, a third-party payment gateway, to securely process credit card transactions.

2.2. Assumptions, Constraints, and Risks

2.2.1. Assumptions

- **Platform**: Users will interact with AIMS via up-to-date web browsers such as Chrome, Firefox, Safari, or Edge.

- **User Capabilities**: It is assumed that users have basic computer literacy and stable internet access.

- **Third-party Services**: VNPay’s APIs are assumed to be stable and consistently available.

- **Scope**: The system will exclusively handle physical media items; digital goods and subscription services are not included.

2.2.2. Constraints

- **Hardware**: The application must operate smoothly on standard web-enabled devices, without requiring high-performance hardware.
- **User Interface**: The UI should be intuitive enough for users with minimal technical background to navigate and complete tasks.
- **Resource Usage**: Efficient resource management is essential to ensure stable performance under varying loads.
- **Performance Metrics**: Response times must not exceed 2 seconds under normal load, and should remain under 5 seconds during peak usage.
- **Network Reliability**: Frontend and backend communications must be consistent and low-latency to ensure real-time feedback and reliability.

2.2.3. Risks

Risk	Description	Mitigation Strategy
Downtime	System outages may affect the shopping experience.	Employ redundancy and automatic failover systems.
Security Vulnerabilities	Risks of data breaches and unauthorized access.	Perform regular security audits, updates, and enforce best practices.
Scalability Limitations	Inability to support increasing traffic.	Optimize queries and backend logic, and implement load balancing.
Payment Integration Issues	Failures in VNPay integration could block payments.	Conduct continuous integration testing and maintain communication with VNPay support.
Performance Bottlenecks	Slow responses under heavy usage.	Conduct load testing, enable caching, and optimize backend processes.
Data Loss	Potential data corruption or loss due to bugs or hardware failure.	Schedule regular backups and implement integrity checks.

3. Testing Approach/Strategy

3.1. Overview

To ensure high reliability and maintainability of the AIMS software, we adopt a comprehensive unit testing strategy grounded in modern software engineering practices:

- **Test-Driven Development (TDD):**
  Developers are expected to write unit test cases *before* implementing functionalities. This helps clarify the expected behavior and guides clean, modular design.
- **Behavior-Driven Testing (BDT):**
  Each unit is tested according to its *expected behavior* as described in the AIMS requirement document. For example, when a product manager updates a price, the system must validate that it stays within 30–150% of the product's value.
- **Automation First:**
  All unit tests are to be integrated into an automated test suite, enabling continuous execution through CI/CD pipelines.
- **Mocking and Isolation:**
  To isolate each unit under test:
  - External systems like **VNPay**, databases, and email servers will be replaced with **mocks** or **stubs**.
  - This allows testing of units like payment validation or order confirmation without real transactions.

Example:
For a `login()` function:

- Test cases should validate:
  a) correct credentials → success
  b) incorrect credentials → failure
  c) locked accounts → proper error message
  → **Database access must be mocked** so that the test behavior is predictable and isolated.

3.2. Scope of Unit Testing

In-Scope

- Core business logic:
  - Product price constraints (30%–150%)
  - Product type-specific data validation (books, CDs, DVDs, etc.)
  - Delivery fee calculation (standard vs. rush order)
  - VAT and order total calculations
  - Cart management and update operations
  - Inventory checks and insufficient stock warnings

- Utility functions:
    - Email format validation
    - Weight-based shipping fee calculation
    - VAT application and rounding
- Error Handling and Edge Cases:
    - Null/invalid inputs
    - Maximum allowed product operations (e.g., delete ≤ 30 products)

Out-of-Scope

- Integration with:
    - **VNPay Sandbox** (use mock endpoints)
    - **Actual databases** (use stubs/fake repositories at the moment)
    - **Email servers**
    - **User Interface**

3.3. Strategy for Designing Unit Tests

Approach

We use a hybrid of **Black-Box Testing** and **White-Box Testing**, tailored to the type of function or module.

A. Black-Box Testing

Used for testing behavior against expected outputs (especially for business logic modules).

Technique	Use Case	Example Unit
Equivalence Partitioning	Input classification	Book, DVD, CD
Boundary Value Analysis	Edge conditions	Price limits at 30% and 150% of product value
Decision Table Testing	Complex rules	Delivery fee with standard vs. rush orders, VAT rules, free shipping conditions

B. White-Box Testing

Used for internal structure-based testing of algorithms and control flow-heavy units.

Technique	Use Case	Example Unit
Statement Coverage	Ensure all lines are executed	Email validation, VAT application
Branch Coverage	All logical branches tested	Conditional pricing and validation logic
Path Coverage	Full control path traversal	Delivery flow with/without rush delivery

3.4. Tools and Frameworks

To facilitate effective and automated unit testing, the following tools will be employed depending on the language and platform:

Tool	Description	Use
JUnit	Java unit testing framework	Main testing tool for Java-based AIMS implementation
Mockito	Mocking framework for Java	Mocking database access, VNPay integration

4. Unit Testing Summary

4.1. Traceability from Test Cases to Use Cases

USE CASE ID			UC001	UC002	UC003	UC004	UC005	UC006
Test Case ID	Test Case Title	Totals	10	5	4	2	6	
UT001	Add Update Product - Test Processing Payment	1	x					
UT002	Add Update Product - Add Product Duplicate Product ID	1	x					

UT003	Add Update Product - Add Product Invalid Category	1	x					
UT004	Add Update Product - Update DVD Successfully	1	x					
UT005	Add Update Product - Update CD Successfully	1	x					
UT006	Add Update Product - Update Book Successfully	1	x					
UT007	Add Update Product - Update Non-existent Product	1	x					
UT008	Add Update Product - Create Product - Manager Not Found	1	x					
UT009	Add Update Product -Update Save Error	1	x					
UT010	Add Update Product - Update Product Invalid Category	1	x					
UT011	Pay Order - Successful Payment	1		x				
UT012	PayOrder - Unsuccessful Payment, Order Not in "PENDING" State	1		x				
UT013	PayOrder - Payment Transaction Found by Order ID	1		x				
UT014	PayOrder - Payment Transaction Not Found by Order ID	1		x				
UT015	PayOrder - Unsuccessful Payment, Order	1		x				

	Not Found							
UT016	Rush Order - All Products Eligible	1			x			
UT017	Rush Order - Some Products Eligible	1			x			
UT018	Rush Order - Address Not Supported	1			x			
UT019	Rush Order - No Eligible Products	1			x			
UT020	Get Product - Success	1				x		
UT021	Get Product - Not Found	1				x		
UT022	Add Product to Cart - Success	1					x	
UT023	Add Product to Cart - Invalid Quantity	1					x	
UT024	Update Cart Item - Success	1					x	
UT025	Update Cart Item - Insufficient Stock	1					x	
UT026	Remove Cart Item - Success	1					x	
UT027	Remove Cart Item - Not Found	1					x	
UT028	Create Order - Null Invoice	1						x
UT029	Create Order - Null Delivery Info	1						x
UT030	Create Order - Null Cart	1						x
UT031	Create Order - Success	1						x
UT032	Create Order - Product Not Found in DB	1						x
UT033	Create Order - Simulate DB Access Error	1						x

4.2. Test Suite for UC001-"AddUpdateProductToStore"

Test Suite ID	Test Suite Title	Description	Test Cases
US001	Test Add Product To Store	Test the AddprodCTectore () method to ensure the right type of product (book, CD, DVD) to the warehouse, associate with manager via shopitem, and handle errors when the product is invalid.	UT001, UT002, UT003, UT008
US002	Test Update Product To Store	Test the updateProductore () method to ensure the correct update of the product information by type and handle errors when the product type is not valid.	UT004, UT005, UT006, UT007, UT009, UT010

4.3. Test Suite for UC002-"Pay Order"

Test Suite ID	Test Suite Title	Description	Test Cases
US003	Test Processing Payment	Test the ability processing a PaymentTransaction Object, and throw exception when needed.	UT011, UT012, UT015
US004	Test method get the PaymentTransaction by OrderID	Test getPaymentTransactionByOrderId method in PayOrderService	UT013, UT014

4.4. Test Suite for UC003-"Place Rush Order"

Test Suite ID	Test Suite Title	Description	Test Cases
US005	PlaceRushOrderServiceTest	Verify rush order logic including eligibility, delivery area support, and fee calculation in PlaceRushOrderService	UT016, UT017, UT018, UT019

4.5. Test Suite for UC004-"View Product Details"

Test Suite ID	Test Suite Title	Description	Test Cases
US006	ViewProductDetailsTest	Tests retrieving product details by ID and handling the case when the product is not found.	UT020, UT021

4.6. Test Suite for UC005-"Manage Cart"

Test Suite ID	Test Suite Title	Description	Test Cases
US007	Test Add Product to Cart	Tests adding a product to the cart, including normal cases and invalid quantities.	UT022, UT023
US008	Test Update Cart Item	Tests updating the quantity of a product in the cart and checks for stock constraints.	UT024, UT025
US009	Test Remove Cart Item	Tests removing a product from the cart and handling non-existent cart items.	UT026, UT027

4.7. Test Suite for UC006-"Place Order"

Test Suite ID	Test Suite Title	Description	Test Cases
US010	Test Create Order – Validations	Tests for input validation errors when creating an order.	UT028, UT029, UT030
US011	Test Create Order – Success Case	Tests successful order creation with complete and valid input.	UT031
US012	Test Create Order – Error Handling	Tests how the system handles internal errors during order creation.	UT032, UT033

5. Test Case Details

5.1. Test Case Details - UC001 “AddUpdateProductToStore”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass/ Fail	Notes
UT001	Add Product Successfully	Verify product is added successfully when valid book data is provided.	ProductService.createProduct	Book DTO with productID=”P001”, title=”Java Basics”, quantity=5, value=120000d, price=150000d, authors=”John Doe”, publisher=”Tech Books”, genre=”Programming”	DTO with productID=”P001”, category=”book”, title=”Java Basics”	DTO with productID=”P001”, category=”book”, title=”Java Basics”	Pass	Valid product
UT002	Add Product Duplicate Product ID	Verify error is thrown when product with duplicate ID is added.	ProductService.createProduct	Book DTO with productID=”P001” already exists.	Throws RuntimeException: already exists	Throws RuntimeException: already exists	Pass	Negative test case
UT003	Add Product Invalid Category	Verify error is thrown for unsupported product category.	ProductService.createProduct	ProductDTO with category = 'toy'	Throws RuntimeException: invalid category	Throws RuntimeException: invalid category	Pass	Invalid input case
UT004	Update DVD Successfully	Ensure DVD product is updated with correct new data.	ProductService.updateProduct	DVD DTO with productID=”P001”, category=”DVD”, title=”Avengers”, quantity=8, value=150000d, price=200000d, director=”Russo”, , genre=”Action”, runtime=”120 minutes”, studio=”Marvel Studios”	Updated DTO with productID=”P001”	Updated DTO with productID=”P001”	Pass	Valid update case
UT005	Update CD Successfully	Ensure CD product updates reflect all fields.	ProductService.updateProduct	CD DTO with productID = “C001”, category=”CD”, title=”Greatest Hits”, quantity=7, value=80000d, price=100000d, artist=”Queen”, recordLabel=”EMI”, trackList=”Bohemian Rhapsody,	Updated CD info with tracklist, artist	Updated CD info with tracklist, artist	Pass	Valid update case

				Don’t Stop Me Now”, musicType=”Rock ”				
UT006	Update Book Successfully	Ensure Book product updates reflect all fields.	ProductService.updateProduct	Book DTO with productID = “B001”’, category=”Book”, title=”Clean Code”, quantity=10, value=120000d, price=150000d, coverType=”Hardcover”, authors=”Robert C. Martin”, publisher=”Prentice Hall”, numberOfPages=464, language=”English”, genre=”Software Engineering”	Updated book with correct author, genre	Updated book with correct author, genre	Pass	Valid update case
UT007	Update Non-existent Product	Verify update fails if product not found.	ProductService.updateProduct	ProductDTO with productID=”P404”, category=”DVD”, title=”Ghost”, quantity=1, value=10000d, price=12000d	Throws RuntimeException: not found	Throws RuntimeException: not found	Pass	Negative test case
UT008	Create Product - Manager Not Found	Verify error is thrown if manager not found.	ProductService.createProduct	DTO with non-existent managerID	Throws RuntimeException: manager not found	Throws RuntimeException: manager not found	Pass	Missing dependency case
UT009	Update Save Error	Simulate DB save failure during update.	ProductService.updateProduct	Book DTO with ID = 'P001'	Throws RuntimeException: Database error	Throws RuntimeException: Database error	Pass	Simulated DB failure
UT010	Update Product Invalid Category	Ensure error when updating with unsupported category.	ProductService.updateProduct	DTO with category = 'toy'	Throws RuntimeException: Invalid category	Throws RuntimeException: Invalid category	Pass	Invalid update input

5.2. Test Case Details - UC002 “Pay Order”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass/Fail	Notes

UT001	Pay Order-Successful Payment	This scenario tests the payment process when everything goes smoothly	processPayment(orderId, content) is called	status = “PENDING” orderId=”ORD001” Content=” Visa ****-****-****-1234”	-paymentTransaction is not null -testOrder.getstatus() = “CONFIRMED” -testOrder= paymentTransaction.getOrder() -content, paymentTransaction.gettcontent() -paymentTransaction.getdatetime() is not null	TBD	TBD	Valid input case
UT002	PayOrder - Unsuccessful Payment, Order Not in "PENDING" State	This scenario tests the exception handling when an order is not in the "PENDING" state when payment is attempted.	processPayment(orderId, content) is called	status = “APPROVED” orderId=”ORD001” Content=” Visa ****-****-****-1234”	exception.getMessage()= “Order is not in PENDING state for payment. Current status: APPROVED"	TBD	TBD	Invalid input case
UT003	PayOrder - Payment Transaction Found by Order ID	This scenario tests retrieving a payment transaction when it exists.	getPaymentTransactionByOrderId(orderId) is called	orderId=”ORD001”	-paymentTransaction is not null -testOrder= paymentTransaction.getOrder() -paymentTransaction.getdatetime() is not null	TBD	TBD	Valid search case
UT004	PayOrder - Payment Transaction Not Found by Order ID	This scenario tests retrieving a payment transaction when it exists.	getPaymentTransactionByOrderId(orderId) is called	orderId=”ORD002”	paymentTransaction is null	TBD	TBD	Invalid search case
UT005	PayOrder - Unsuccessful Payment, Order Not Found	This scenario tests the exception handling when an order with the provided ID does not exist for payment processing.	processPayment(orderId, content) is called	status = “PENDING” orderId=”ORD002” Content=” Visa ****-****-****-1234”	exception.getMessage()= “Order not found with ID: ORD002"	TBD	TBD	Invalid input case

5.3. Test Case Details - UC003 “Place Rush Order”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass /Fail	Notes
UT016	Rush Order - All Products Eligible	Verify rush order when all products are eligible and address is supported	evaluateRushOrder(info, products)	province = "Ba Dinh", all products `isRushEligible = true`	province = "Ba Dinh", all products `isRushEligible = true`	TBD	TBD	Valid full rush case
UT017	Rush Order - Some Products Eligible	Verify rush order when some products are eligible	evaluateRushOrder(info, products)	province = "Hoan Kiem", 1 product rush-eligible, 1 not	`isSupported = true`, no prompt	TBD	TBD	Mixed product case
UT018	Rush Order - Address Not Supported	Address outside inner city → rush order not supported	evaluateRushOrder(info, products)	province = "Hung Yen", all products rush-eligible	`isSupported = false`,, prompt shown	TBD	TBD	Address edge case
UT019	Rush Order - No Eligible Products	No products eligible → rush not supported even if address valid	evaluateRushOrder(info, products)	province = "Ba Dinh", all products `isRushEligible = false`	`isSupported = false`,, prompt shown	TBD	TBD	Product filter case

5.4 Test Case Details - UC004 “View Products Details”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass/ Fail	Notes
UT020	Get Product - Success	Verify the result when the product ID exists	ProductService.getProductById()	productId = "abc123"	Returns `ProductDTO` containing the correct product info	`ProductDTO` with `productId = "abc123"`	Pass	Simple valid case
UT021	Get Product - Not Found	Verify behavior when the product ID is missing	ProductService.getProductById()	productId = "nonexistent"	Throws `RuntimeException` with appropriate message	RuntimeException: Product not found with id: nonexistent	Pass	Not found case

5.5 Test Case Details - UC005 “Manage Cart”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass /Fail	Notes
UT022	Add Product to Cart - Success	Add a new product to the cart successfully	CartService.addToCart	customerId='customer123', productId='product456', quantity=2	CartItemDTO with correct details	CartItemDTO with 'product456', quantity=2	Pass	Valid input
UT023	Add Product to Cart - Invalid Quantity	Add product with quantity = 0 should throw BadRequestException	CartService.addToCart	customerId='customer123', productId='product456', quantity=0	Throws BadRequestException	BadRequestException with message 'Quantity must be greater than zero'	Pass	Invalid quantity
UT024	Update Cart Item - Success	Update cart item with valid quantity and product stock	CartService.updateCartItem	customerId='customer123', productId='product456', quantity=3	CartItemDTO updated with new quantity	CartItemDTO with quantity=3	Pass	Stock available
UT025	Update Cart Item - Insufficient Stock	Updating cart item with quantity exceeding stock throws exception	CartService.updateCartItem	customerId='customer123', productId='product456', quantity=15	Throws BadRequestException	BadRequestException with message 'Not enough stock available. Available: 10'	Pass	Exceeds stock
UT026	Remove Cart Item - Success	Remove an existing cart item	CartService.removeFromCart	customerId='customer123', productId='product456'	Cart item removed	Repository.deleteById called	Pass	Item exists
UT027	Remove Cart Item - Not Found	Try to remove a non-existing cart item should throw exception	CartService.removeFromCart	customerId='customer123', productId='product456'	Throws ResourceNotFoundException	ResourceNotFoundException with message 'Cart item not found'	Pass	Item not found

5.5 Test Case Details - UC006 “Manage Cart”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass /Fail	Notes
UT022	Add Product to Cart - Success	Add a new product to the cart successfully	CartService.addToCart	customerId='customer123', productId='product456', quantity=2	CartItemDTO with correct details	CartItemDTO with 'product456', quantity=2	Pass	Valid input
UT023	Add Product to Cart - Invalid Quantity	Add product with quantity = 0 should throw BadRequestException	CartService.addToCart	customerId='customer123', productId='product456', quantity=0	Throws BadRequestException	BadRequestException with message 'Quantity must be greater than zero'	Pass	Invalid quantity
UT024	Update Cart Item - Success	Update cart item with valid quantity and product stock	CartService.updateCartItem	customerId='customer123', productId='product456', quantity=3	CartItemDTO updated with new quantity	CartItemDTO with quantity=3	Pass	Stock available
UT025	Update Cart Item - Insufficient Stock	Updating cart item with quantity exceeding stock throws exception	CartService.updateCartItem	customerId='customer123', productId='product456', quantity=15	Throws BadRequestException	BadRequestException with message 'Not enough stock available. Available: 10'	Pass	Exceeds stock
UT026	Remove Cart Item - Success	Remove an existing cart item	CartService.removeFromCart	customerId='customer123', productId='product456'	Cart item removed	Repository.deleteById called	Pass	Item exists
UT027	Remove Cart Item - Not Found	Try to remove a non-existing cart item should throw exception	CartService.removeFromCart	customerId='customer123', productId='product456'	Throws ResourceNotFoundException	ResourceNotFoundException with message 'Cart item not found'	Pass	Item not found

5.6 Test Case Details - UC006 “Place Order”

Test Case ID	Test Case Name	Description	Unit Under Test	Input Data	Expected Output	Actual Output	Pass /Fail	Notes
UT028	Create Order - Null Invoice	Should throw exception when invoice is null	PlaceOrderService.createOrder	invoice = null	Throw `IllegalArgumentException`	`IllegalArgumentException` thrown	Pass	Validates null input
UT029	Create Order - Null Delivery Info	Should throw exception when delivery info is null	PlaceOrderService.createOrder	`invoice.deliveryInfo = null`	Throw `IllegalArgumentException`	`IllegalArgumentException` thrown	Pass	Validates delivery info required
UT030	Create Order - Null Cart	Should throw exception when cart is null	PlaceOrderService.createOrder	invoice.cart = null	Throw `IllegalArgumentException`	`IllegalArgumentException` thrown	Pass	Validates cart required
UT031	Create Order - Success	Create order successfully with valid invoice	PlaceOrderService.createOrder	`invoice.deliveryInfo` + `invoice.cart` with valid data	Order object created, `totalAmount` calculated correctly	Order created with expected data	Pass	Full flow success case
UT032	Create Order - Product Not Found	Product in cart not found in database	PlaceOrderService.createOrder	`invoice.cart` contains product with invalid `productId`	Throw `RuntimeException("Product not found with id: ...")`	RuntimeException with correct message	Pass	Tests product existence
UT033	Create Order - Simulate DB Access Error	Simulates DB error when saving order	PlaceOrderService.createOrder	Mock `orderRepository.save()` to throw `DataAccessException`	Throw `RuntimeException("Database error: ...")`	RuntimeException with correct message	Pass	Simulated DB failure