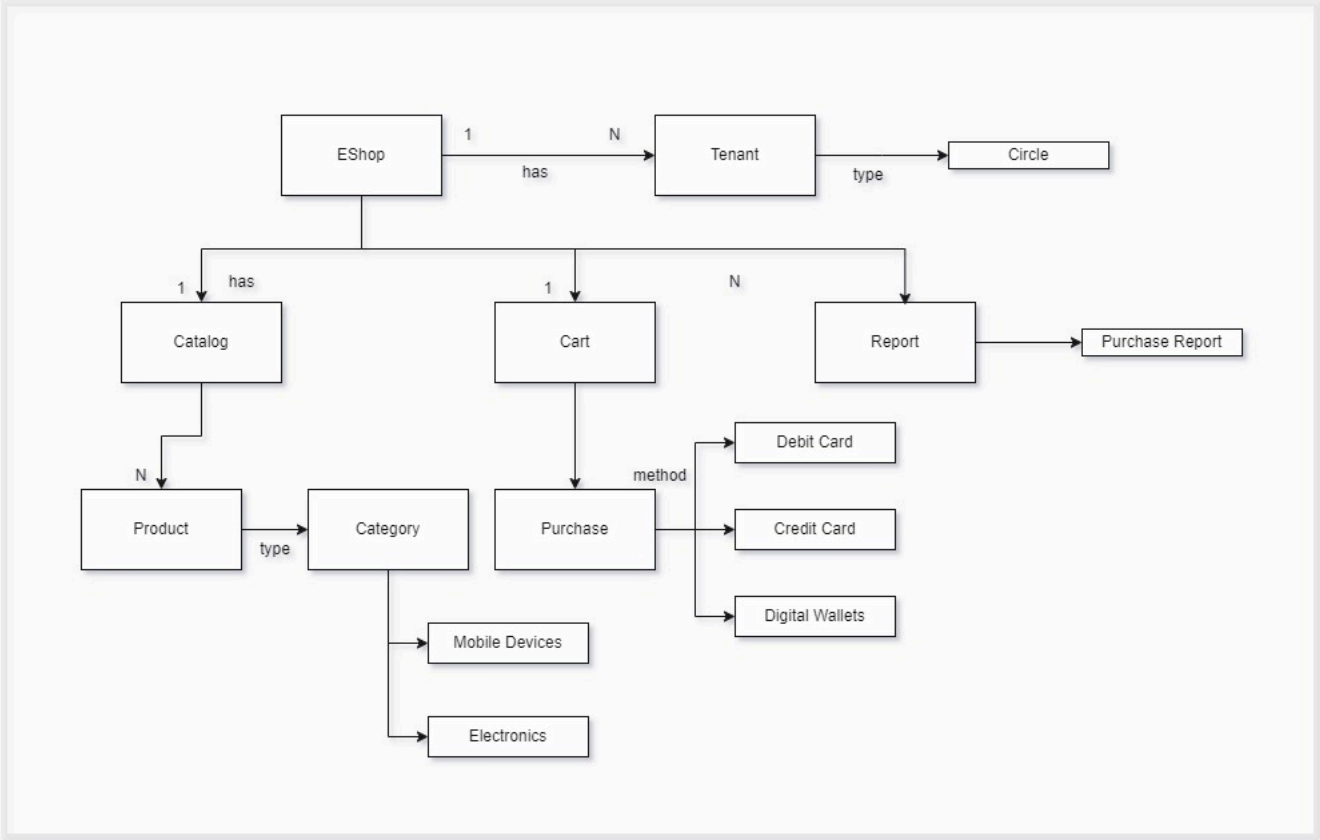
E-Shopping

Last modified by MANISH JHA (/bin/view/XWiki/MANISHJHA) on 2024/04/14 06:23

E-shopping

As an example project the E-shopping Solution SaaS is considered in the context of the MVP for a specific requirement: to develop an eShopping product for an organization or entity wit Circle. This organization in real life context may be completely different and internal or external to Circle or maybe a signed partner, serving as an alternative tenant. This E-Shopping SaaS solution is designed to meet the basic requirements of developing search capabilities and facilitating shopping through the standard practices of the e-commerce domain.

Domain Definitions



Catalog:

The Catalog that provides information on products and services sold by a tenant.

Product:

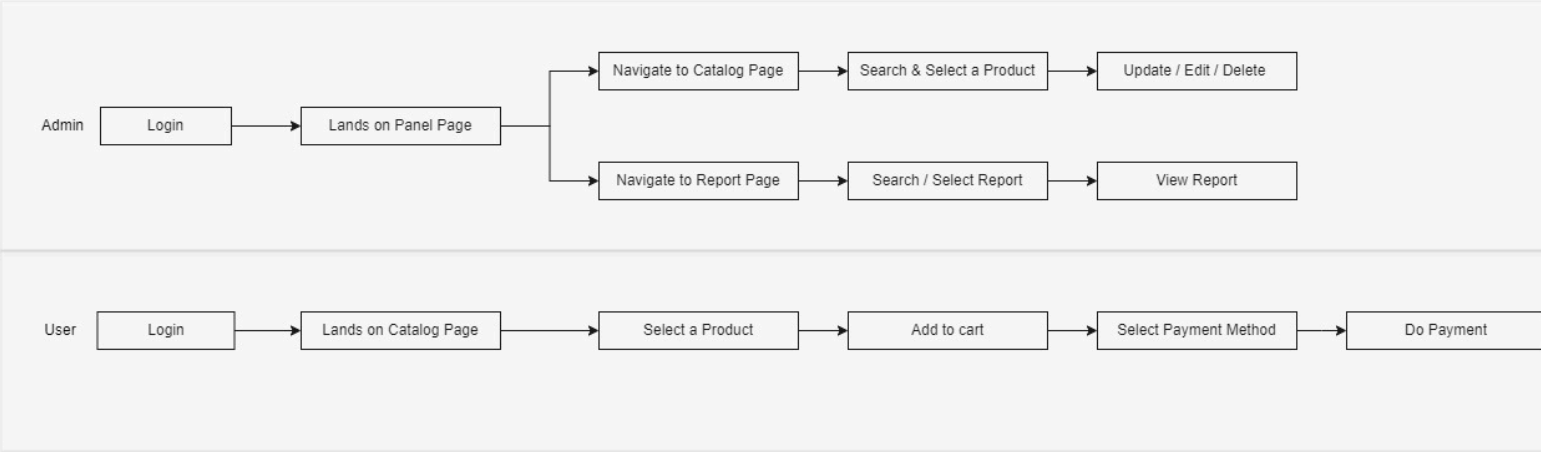
Collection of item type getting listed under Catalog where the tenant can sell it on E-shopping SaaS platform under multiple categories like Mobile devices, Electronic accessories etc

Cart:

E-shopping cart is where product is added by the user with an intent to purchase

Workflow

E-Shopping Workflow



Requirements

Functional Requirements

As per MVP for all main functional PRD are listed below

Non Functional Requirements

Security

- The system needs to be externalised and used by external tenants. For MVP basic User Roles and Authorisation need to be maintained so Data resides locally in the system.

Manageability

- Products, Catalogs Cart etc need to be set and managed by Admin teams. This requires abilities to CRUD services for the same.

Interoperability

- All Services/modules need to provide services/event streams which could be used by other modules to build upon.

Maintainability/Adaptability

- Since this product is trying to map a business e-Shopping requirement which can evolve so process best Engineering practices for a modular architecture.

Performance

- Product catalog and search functionalities are expected to be able to handle requests per second up to 5000 TPS.
- Cart functionality is expected to be able to handle requests per second up to 1000 TPS.
- Purchasing functionality to be able to handle requests per second up to 250 TPS.

Capacity / Scale

- Tenant Count: 1-5
- Catalog Per Tenant: 1
- Category Per Catalog: 1000
- Average Product Category: 10000

User and Load Estimation
Average Daily Usage for Cart: Average Load(1000) * Duration(3600) = 3.6*10^6 approx
Total User for Cart and Product(10% active monthly) Total User = Active User / .1 = 3.6*10^7
Peak TPS User for Cart: 2000-3000
Max Allowed TPS for Cart: 4000
Average Daily User for Search: Average Load * Duration: 2*10^7
Peak TPS User for Search: 7000-8000*
Max Allowed TPS for Cart: 10000

Storage required in Db in primary: Expected Data Per user(5kb) * Max User(3.6*10^7) : 170GB
Storage required Cache: Expecting 10% for active user: 17GB
Shards requirement: 4 (considering 50GB as balance) so provision for 7

Core Capability

Your design should include:
• Clearly defined integration interfaces and integration approach between the various services in your solution with clear justifications for such selections.
• The data stores used in your solution with clear justifications for such selections.

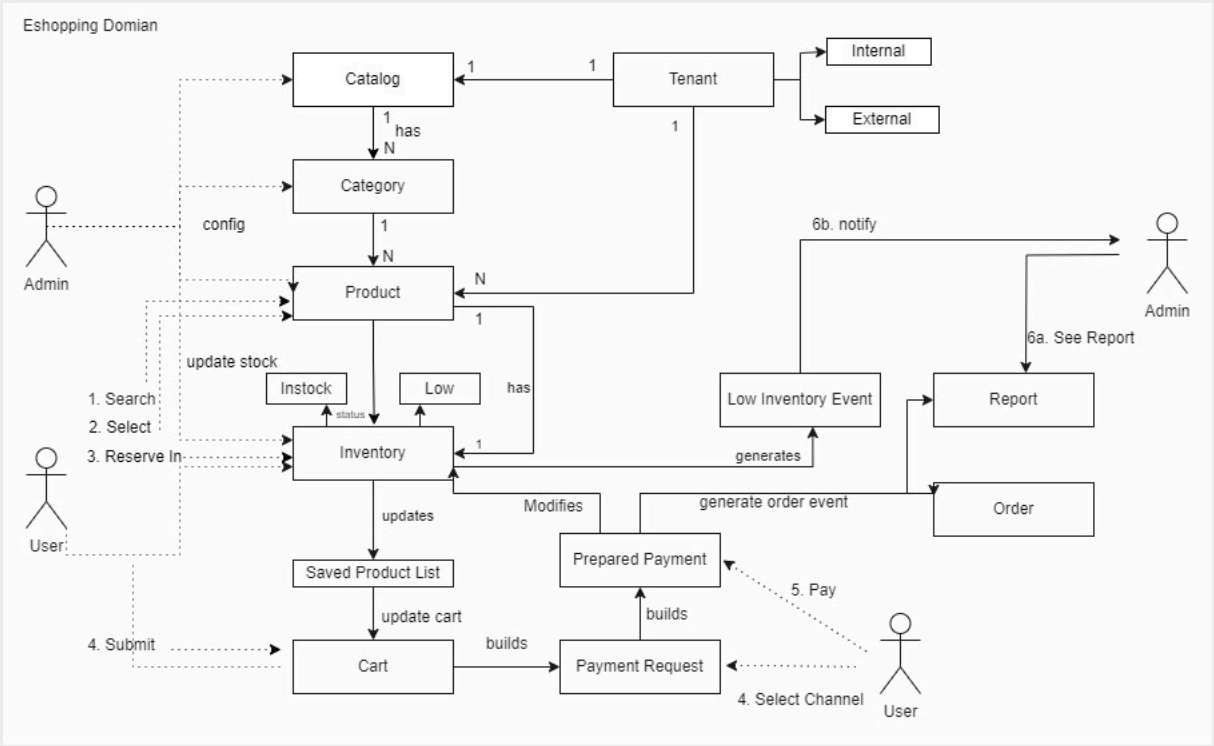
- All required diagrams (including but not limited to context diagrams, flow charts, sequence diagrams).
- All API definitions exposed by the various services in your solution.
- Specific security considerations in your design.
- Specific performance considerations in your design.
- Any technology selections in your solution and clear justifications for such selections.

Proposed Architecture

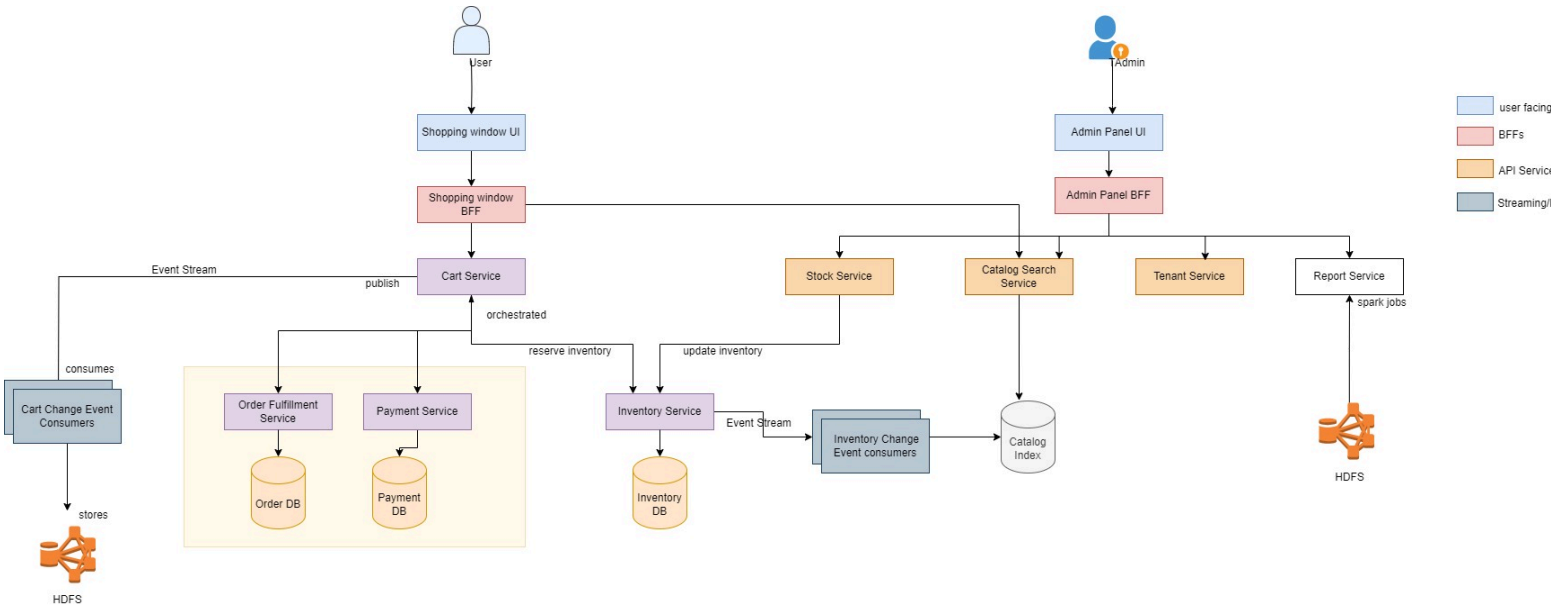
We propose a multi-tenant, event-sourcing-based architecture to fulfill the aforementioned requirements. The system will support two primary user roles

1. User: Responsible for searching, selecting products, adding products to the catalog, and submitting the catalog for payment.
2. Admin: Tasked with editing or configuring catalog, category, product, and inventory information.

Domain Diagram



Component Diagram

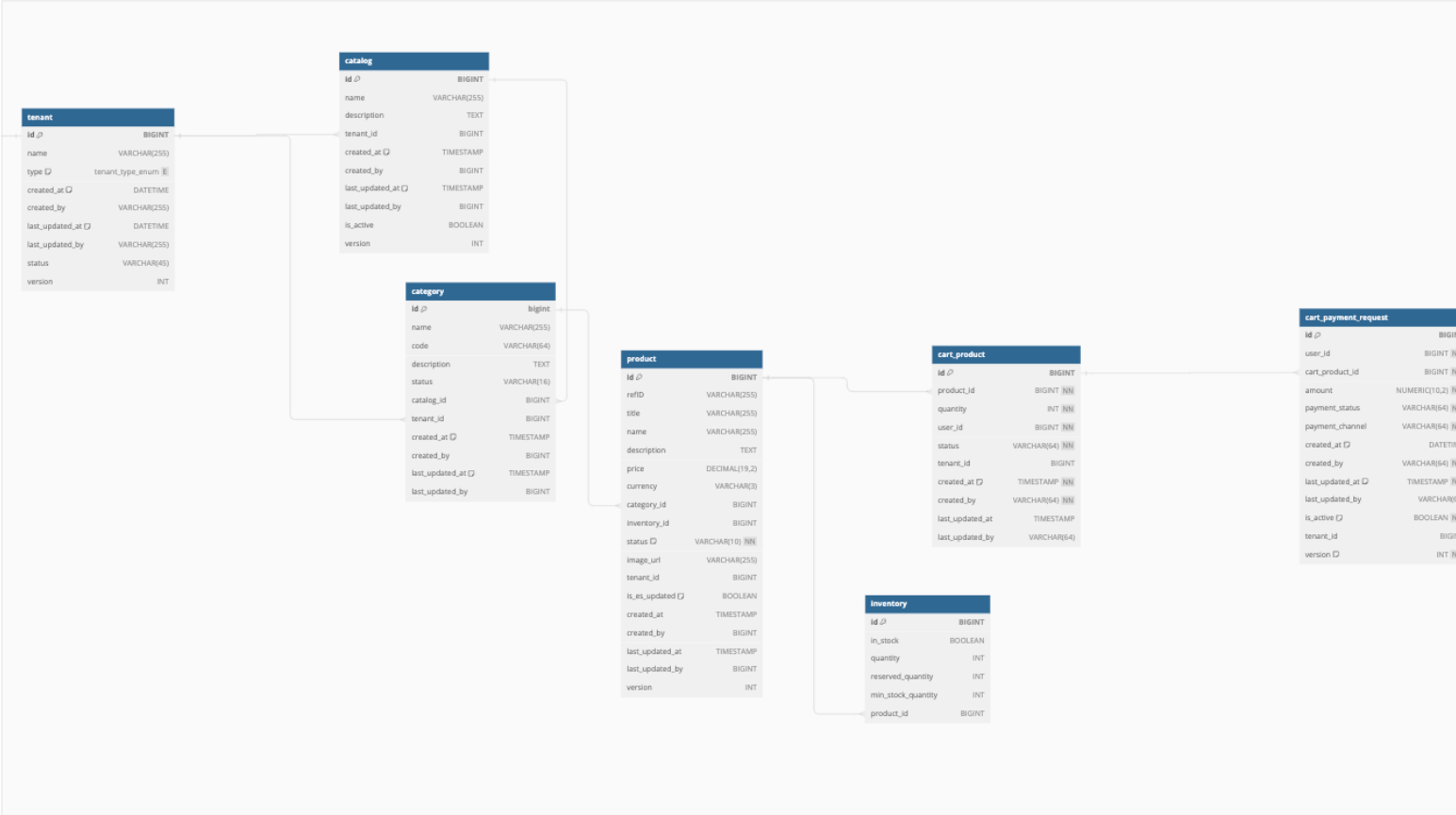


Low Level Design Document

API Contract

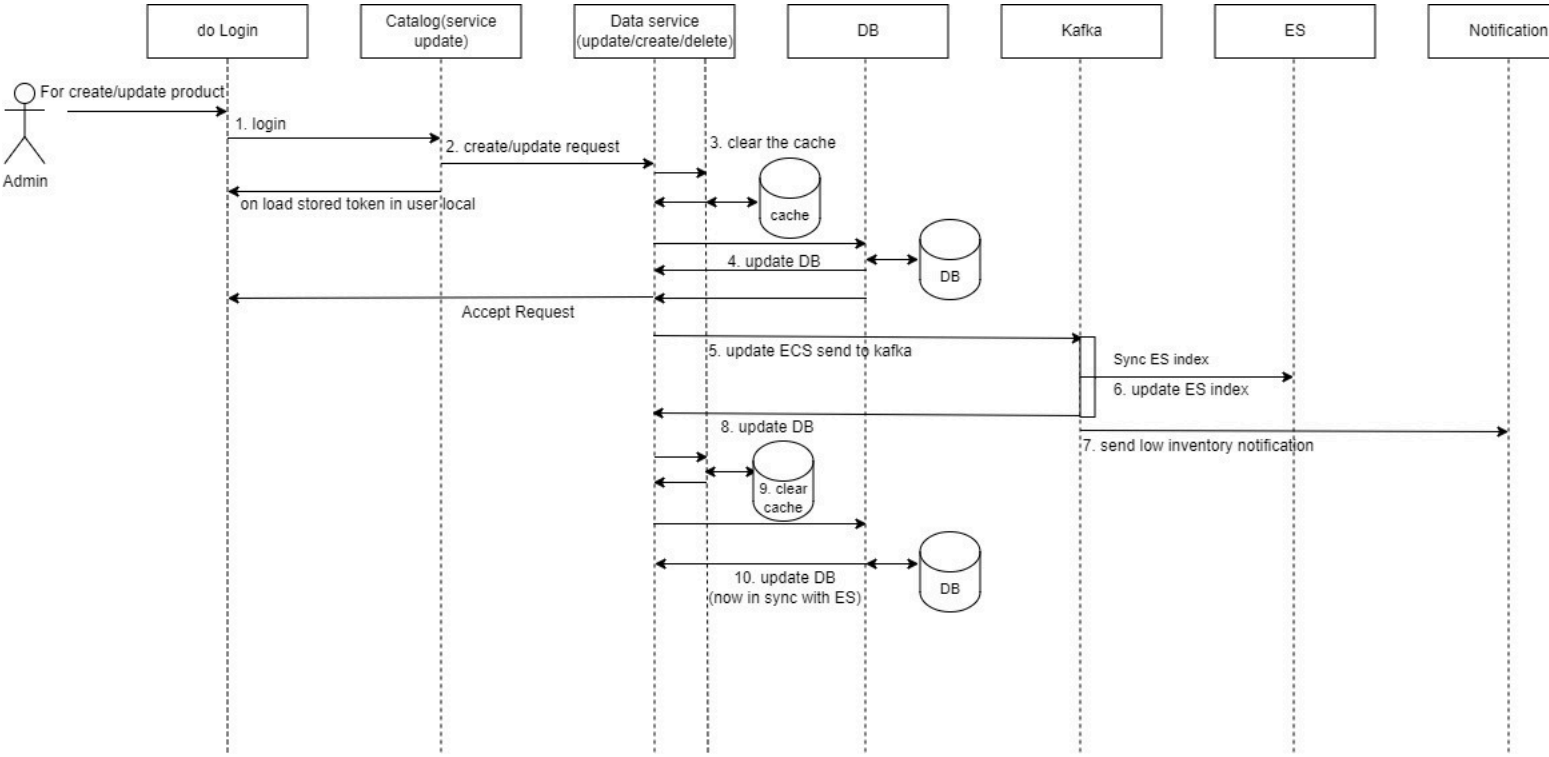
Item	Path	Method	Headers	Context	Request Payload
1	/user/api/v1/products	POST	loginId=test@test&estoken=12345	Create New Product	Same as PUT
2	/admin/api/v1/products/{productid}	PUT	loginId=test@test&estoken=12345	Update Existing Product. During update version must match to avoid stale update	{ "title" : "Example item 50345", "name" : "Example item name 50 "description" : "This is an examp description of 50345", "price" : 4569.99, "currency" : "USD", "inS : true, "quantity" : 90001, "reservedQuantity" : 0, "minStockQuantity" : 2601, "stat "ACTIVE", "categoryId" : 1, "imag : "http//:www.test.test/image5034 "tenantId" : 1, "versionNo":3 }
3	/user/api/v1/products/{productid}	GET	loginId=test@test&estoken=12345	Get Product Details	
4	/catalog/_doc/{refid}	GET	?query_search_key=product-id&query_search_value=12345&status=ACTIVE&login=manish.jha@test.co	Serach Product Details	
5	/user/api/v1/cart/{userId}	GET	loginId=test@test&estoken=12345	GET Cart Detail	
6	/user/api/v1/cart	POST	loginId=test@test&estoken=12345	Create Cart with Product Item	Same as PUT
7	/user/api/v1/cart/{cartId}	PUT	loginId=test@test&estoken=12345	Update Cart with Product Item	
8	/user/api/v1/cart/{cartId}/add	POST	loginId=test@test&estoken=12345	Add to existing Cart with Product Item	
9	/user/api/v1/cart/{cartid}/payment/prepare	POST	loginId=test@test&estoken=12345	Prepare Payment request	
10	/user/api/v1/cart/{cartid}/payment/pay	POST	loginId=test@test&estoken=12345	Pay. creates purchase event	

DB Schema : inventory_service



API Flows

Create / UPDATE Product Sequence Diagram



Stability Pattern Used :

To enhance system reliability and handle failover gracefully, several stability patterns can be implemented:

- **Send Fail Event to DLQ** : When message processing fails, these are getting redirected to a Dead Letter Queue (DLQ). The DLQ acts as a reservoir for such problematic messages, isolating them from the main workflow. This separation is done to prevent clogging of the primary queue and facilitates focused debugging and reprocessing.

- **Circuit Breaker for Sending Notifications**: This pattern halts operations prone to failure, like sending notifications to downstream using third party network, when a set failure thre reached in a sliding window of the Last 100 calls circuit is opened and no overwhelming integration. It temporarily breaks the flow, reducing the risk of repeated failures and conserv system resources. Operations resume gradually, ensuring system stability.
- **Rate Limiter for Kafka Consumers**: Applying rate limits to Kafka consumers controls the message processing rate, preventing system overload. This ensures that the consumptio aligns with the capacity of downstream systems, maintaining overall throughput without triggering failures.
- **Elastic Nodes for Peak Loads** : Elastic nodes dynamically adjust resources based on real-time demand. This scalability allows us to handle increased loads efficiently during peak and reduce resource consumption during low activity periods, optimizing performance and resource utilization.