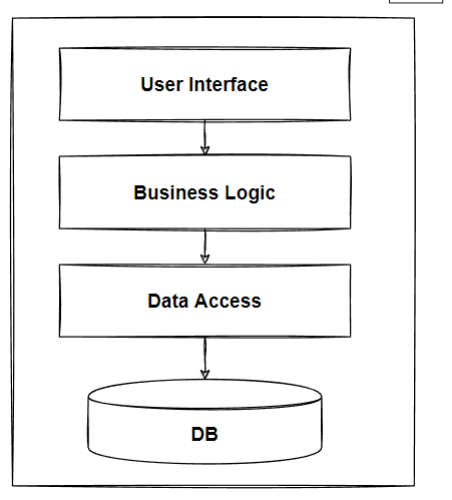
NFR: Non function Requirements

1. Availability – Application should be up 24X7
2. Concurrency
3. Maintainability
4. Scalability – Serve million of users
5. Reliability - Application should be up 24X7
6. Usability – Less than 3 seconds to response

Functional Requirements:

1. List product
2. Filter products by brand and categories
3. Add to cart
4. Apply discount coupons
5. Checkout
6. List order and order item history
7. Login as user and remember my items.

**Monolithic**: Monolithic architecture is an architectural pattern and of designing and developing complete application as a single unit.



Pros:

1. Easy to develop,
2. Easy to debug and test.
3. Easy to deploy.

Cons:

1. Difficult to manage, hard to implement new features.
2. Unable to implement new technologies.
3. Difficult to scale an individual module.

Design principles of monolithic Architecture:

DRY – Don’t Repeat Yourself.

KISS- Keep it Simple Silly.

YAGNI – You ain’t gonna need it.

**Layered (N- LAyer) Architecture**: The main purpose of a layered architecture is to organize the components of an application with similar functionalities into horizontally logical layers.

A diagram of a computer process

Description automatically generated

Pros:

1. Break down application into logical layers
2. Oranising code for **separation of concerns**.
3. Layers can be modified without affecting other layers.
4. Presentation Layer, Business Layer, Database Layer.
5. .Low coupling , High Cohesion.

Cons:

1. Highly couples and dependent on each other
2. Code organization hard toto maintain.
3. Frameworks hard to change.
4. Dependent layers causing complexity and hard to change

Design Principles: SOC and SOLID

SOLID:

1. Single Responsibility – Each module should have one functionality.
2. Open Closed – Open to extension , closed for modification.
3. Liskov Substitution – Abstract the underlying service bus implementation so that it can be replaced from Rabbit MQ to Kafka in future.
4. Interface segregation – Interfaces should be broken into smaller and specific ones.
5. Dependency Inversion – High level modules should not depend on low level modules, both should depend on abstractions.

**Clean Architecture:**  It is a software design practices that separates the elements of design into circle levels. Keep the core business logic and application domain at the center of solution. It is divided into **policies and details**.

1. Policies – Business rules and procedures.
2. Details- Implementation code to carry out the policies.

A diagram of a clean architecture

Description automatically generated

Layes of Clean Architecture:

1. Domain Layer – Plain domain and business rules along with validation logic.
2. Application Layer– Application business rules and decision making logic.
3. Infrastructure Layer - Storage logic for DB, FS and 3rd party libraries
4. Presentation Layer– Implementation layer for infrastructure and external interfaces. (Dependent on UI Frameworks, DB version etc.)

Design Rules of Clean Architecture

Dependency Rule –

1. The dependency of source code should only point inwards ie to the center of the circle.
2. Code dependencies can only move from outer levels to inner layer
3. Code on inner layes should not have any knowledge about outer layer.
4. Class , functions , variables in outer layes should not be mentioned by the code of an inner layer.

Pros:

* Independent of DB and Frameworks.
* Independence of UI.
* Loosely couple independent layers
* Testable

Cons:

1. Growing Traffic
2. Large number of request coming to the app
3. Acceptable latency for users.
4. Scalability Issues – Increased Traffic and Acceptable latency for users (Solution- Scalabilty Vertical and Horizontal)
5. Layers are independent but those are technical layers
6. Vertical business logic implementation required to modify all layers ie(Add to basket , checkout order)

Scalabilty – Number of request an application can handle.

We should increase scalability to **prevent downtime and reduce latency**.

Horizontal scaling (Scale out) – Add more machines

Vertical Scaling (Scale up) – Add more CPU, Ram to existing machine.

State less apps are easy to scale hortizontally.

Load Balancer- Balance the traffic across all the application nodes.

Load Balancer- Fault tolerance and improves availability. (nginx)

Load Balancer sits between client and server

It uses consistent hashing algorithm, round robin algorithm

Modular Monolithic Architecture: The modular monolithic architecture divided our application logic into modules, and each module will be independent and isolated.

Each model should have its own business logic and if necessary, its own database and schema.

Also, each model can follow their own logical separation.

A diagram of a software system

Description automatically generated

Pros:

* Encapsulate Business Logic
* Reusable Codes, Easy to Refactor
* Better-Organized Dependencies
* Update front end independently

Cons:

* Scalability Limits, DB can’t scale
* Can deploy modules independently

Strong Consistency is Mandatory && Independent Scale and Deploy is not Required Modular Monolithic

Strong Consistency is not Mandatory, Eventual Consistency is OK && Independent Scale and Deploy is Required = Microservice

Headless architecture separates the frontend from the backend layer of the application. Separates UI and business logic. (CBRE PACE, Airgain)

▪ Headless architecture emphasizes mainly decoupling frontend and backend layers, and it is the first step before moving to microservices.

▪ It uses APIs to connect the front and backends applications.

**Microservice Architecture:**

1. **Microservices are small, loosely coupled , independent services that can work together.**
2. **Each service is a separate codebase, maintained by small development team.**
3. **Microservices communicate with each other by using well defined API’s.**
4. **Microservice can be deployed independently and autonomously.**
5. **Microservices can work with different technologies stack.**
6. **Each service has its own database.**
7. **Microservices are organized by business capability named bounded context.**
8. **SmartEndpoints and dumb pipes: Decoupled and cohesive.**
9. **Decentralized Governance: Sharing common code as library**
10. **Infrastructure Automation: Deployment to each new environment in automated fashion.**
11. **Design for failures: Self healing (retry and circuit breaking)**

**Cons:**

1. **Add new features and less time to market.**
2. **Identify**

Microservices Decomposition Patterns:

Scale Cube-

1. X axis: Horizontal decomposition (Running multiple copies of an application behind load balancer).
2. Y axis: Functional decomposition (Splitting application into multiple and different services).
3. Z axis: Data Partitioning(Data partitioned across a set of servers)

Decomposing by Business Capabilities:

1. Services must be loosely coupled and cohesive in nature.

Decomposing by subdomain:

1. Define services according to DDD
2. Domain – Ecom . SubDomain: Product Catalog, Inventory, Order, Delivery.

Domain Driven Design:

1. Strategic
   1. Identifying and defining the business Core Domains and Subdomains.
   2. Establishing clear boundaries between different parts of the system (Bounded Contexts).
   3. Developing a shared language (Ubiquitous Language) within each Bounded Context.
   4. Mapping relationships between different Bounded Contexts and Teams (Context Mapping).
   5. Aligning the software architecture with the business strategy and domain structure.
   6. we focus on performing high-level analysis of the domain and its constraints, staying away from technical details.
2. Tactical – Depending

Bounded Context: a bounded context is a design boundary within which a particular model is defined and applicable. It's a way to handle complex systems by dividing them into more manageable parts, each with its own specific model and rules.

In DDD, use context mapping pattern to identify bounded context.

Context Map – It is the legal boundaries between domains.

A microservice can represent one or more bounded context

**1. Bounded Contexts:**

* **Customer Management**
* **Order Processing**
* **Inventory Management**
* **Shipping**

**Customer Management Context:**

* **Purpose:** Manages customer profiles, authentication, and preferences.
* **Entities:** Customer, Address, PaymentMethod.
* **Key Concepts:** Customer ID, Authentication Tokens, Payment Methods

**Order Processing Context:**

* **Purpose:** Handles the creation, management, and tracking of orders.
* **Entities:** Order, OrderItem, OrderStatus.
* **Key Concepts:** Order ID, Order Status, Payment Status.

**Inventory Management Context:**

* **Purpose:** Manages stock levels, product details, and inventory adjustments.
* **Entities:** Product, StockLevel, InventoryAdjustment.
* **Key Concepts:** Product ID, Stock Quantity, Reorder Levels.

**Shipping Context:**

* **Purpose:** Manages the logistics of shipping orders, including tracking and delivery.
* **Entities:** Shipment, TrackingNumber, DeliveryStatus.
* **Key Concepts:** Shipping Method, Delivery Time, Tracking Information.

**Inter-Bounded Context Communication:**

* **Customer Management ↔ Order Processing:** The Order Processing context needs customer details for order fulfillment, so it requests this information from the Customer Management context.
* **Order Processing ↔ Inventory Management:** To ensure orders are processed correctly, the Order Processing context queries the Inventory Management context about stock levels.
* **Order Processing ↔ Shipping:** Once an order is processed, the Shipping context is informed to begin the delivery process.

Checklist for Microservices:

1. Should have single responsibility
2. Should not be chatty services
3. Microservices should not be too big or small
4. No locking dependencies

Nouns- products, brand, categories, cart, stock,

Verbs – list, see , filter, add , login , receive

GraphQL –

gRPC

Kafka

RabbitMQ: Push based Model

* 1. Producer: Which send messages
  2. Exchange: Have routing logic
  3. Bindings: Link between exchange and queue
  4. Queues: The place where messages are stored.
  5. Consumer:

Durable Queue: The queue will persist even if the broker is restarted.

Exclusive: If a queue can be used in other connections

AutoDelete: Queue will be deleted when message is passed to the consumer.

Exchange Type:

* 1. Direct: 1:1 mapping
  2. Topic: When message needs to be sent to a consumer based on routing key (similar to Load balancer)
  3. Fanout: When message needs to be sent to multiple consumers.
  4. Header:

**Microservices Communication**

Problems:

1. Direct client to service communication
2. Chatty calls from client to service.
3. Manage invocations from client app.

Types:

1. Synchronous: request/response
2. Asynchronous: message broker/ event bus (AMQP): pub sub mechanism
   1. One to one: queue
   2. One to many: topic

MS Communication Style:

1. Request/Reply: HTTP and REST based
   1. REST – Public APIs: When exposing from Microservices
   2. gRPC- Backend APIs: When communicating internally within microservices
   3. GraphQL – When flexible data is required in microservices
   4. Websockets- Real time Communication
2. Real Time: HTTP and Web socket based
3. Polling: HTTP and AMQP
4. Event Driven: Publish and subscriber

Problems:

* Wrong approach: /customers/6/orders/22/products

Right approach: 1. /customers/6/orders

2. /orders/22/products

\* To handle N+1 request for retrieving data use **GraphQL**

**\*** To handle network performance issues on inter service communication use **gRPC**

**Ex: Add item into shopping cart after getting discount price from discount services.**

* **Chat support : Web socket Real time communication**

Problem:

1. Direct mapping of backend services exist in client apps.
2. Direct client to service communication.
3. New service needs to be added we need to make changes to the client.

API Gateway:

Purpose:  
1. To decouple client app and microservices.

2.Service discovery and registration is handled by API Gateway/

3. Along with cross cutting concerns like Authn, Authz, logging, Rate limiting,Monitoring, Load balancing, protocol translations.

Patterns:

1. Gateway Routing : Expose multiple backend microservices to a single gateway. (Blue Green or Canary deployment pattern)
   1. It can help with versioning of API
   2. It uses Layer 7 routing to route the request.
2. Gateway Aggregation: Split request to multiple backend services and merge the response of multiple backend services to a single client app.
3. Gateway offloading: In this pattern authen, authz, token management process is handled by Identity server.

Backend for Frontend pattern: It is used when a single gateway is used to serve the response for Web UI as well as mobile UI.

Having 3 separate API gateways for Backend , Mobile and desktop apps.

Drawback of BFF : Increased Latency

Service Aggregator Pattern: In Ecom, if we want to place an order and it uses cart, order , product and shipment services. If any of these services are busy or failing it will impact application performance. In Service aggregator pattern we will create an aggregator service which will split the communicate with all the microservices and merge the response back from the services before sending it to the client.

**Service Discovery is performed by Kubernetes Container orchestrator these days.**

**Asynchronous Microservice Communication:**

* 1. Client Send a request but doesn’t wait for the response.

Benefits:

1. Easy to add new services
2. Easy to scale.
3. Event driven architecture.

Challenges:

1. SPOF with message brokers
2. Debugging is hard to do . (Use correlationId)
3. At least once delivery and message ordering.

1:1 Queue based messaging: Message stays in queue until the consumer is failing/not responding.

1: M Topic based messaging:

Fan out(Pub/Sub) is a messaging pattern in which message broker fans out messages to multiple destinations.

**Data Management in Microservices:**

1. Polyglot persistence: Meaning multiple(SQL,NoSql) databases in an application
2. Relational DB: ACID
3. IT is difficult to implement queries and transactions in microservices architecture.

Database Management Patterns in Microservices:

1. Database per service: It provide isolation to other microservices. If A microservice fails due to DB corruption, it wont impact microservice B and C etc.
   1. Cons: Unable to make complex joins
   2. Need to implement retry and circuit breaking logic for resiliency
   3. Distributed transactions can negatively impact consistency and atomicity.
2. API composition
3. CQRS
4. Event Sourcing
5. SAGA
6. Shared Database Anti pattern : Unable to scale,resilience and independency, Block microservices due to SPOF.

Relation Databases Uses:

1. ACID
2. Predictable Data: In terms structure, size and frequency
3. Complex Join Queries
4. Centrailzed Data store

Non Relational Databases:

1. Flexible schema and dynamic data
2. Unpredictable data and high workload volume
3. Frequently change data and read requirements
4. Data consistency – BASE(Basically Available) (Soft State) (Eventual Consistency)
5. Write Performance Requirements: NoSQL database compromise consistency to achieve faster write operations.
6. Not Good for complex join queries.
7. Decentralized Structure.

 **Basically Available**: The system is designed to be highly available, meaning that it remains operational even if some nodes in the system fail.

 **Soft State**: The system allows for temporary inconsistencies or delays in state updates. The state of the system is not immediately consistent across all nodes.

 **Eventually Consistent**: The system guarantees that all updates will eventually propagate to all nodes, achieving consistency over time.

**Adding an Item to the Cart**

1. **User Action**:
   * A user adds an item to their shopping cart.
2. **Write Operation**:
   * The request is processed by one node in the distributed database. The node updates its local copy of the shopping cart to reflect the new item.
3. **Eventual Propagation**:
   * The update is replicated to other nodes in the distributed database. This replication might not happen immediately due to network latencies or temporary unavailability of some nodes.
   * While the update is in transit, some nodes might still reflect the old state of the cart.
4. **Read Operation**:
   * If the user queries their cart immediately after the update, they might get different results depending on which node handles the request. Some nodes might show the new item, while others might not.
5. **Eventual Consistency**:
   * Over time, all nodes in the distributed database will receive the update, and the state of the cart will become consistent across the system.
   * This is ensured by the system’s replication and synchronization mechanisms, which guarantee that all nodes will eventually converge to the same state.

CAP – Consistency, Availability and Partition Tolerance

Materialized View pattern:

Instead of making complex join across different database we can store the denormalized data in our database. For ex: Shopping cart service can have a denormalized data related to product and pricing microservices.

CQRS: Command Query Responsibility Seggregation

Command: Insert , update, delete

Query: select operation

Commands will write to a database, while queries read data from highly denormalized materialized view created on the database so that complex joins and table locks can be avoided.

Since read and write db are different we need to sync them.

Eventual consistency: Read data may be old and not in sync with write data , hence the application is reflecting incorrect data on client app.

To sync read and write data we can use message broker between the read and write DB.

Event sourcing pattern: Add and deleting items from cart are added to the event source table and then it can trigger a message to update read data and we can replay all the events while inserting data in write db.

Problems of Distributed Transactions:

* 1. Data consistency
  2. Rollback transactions

SAGA pattern: To manage distribute transactions across multiple microservices.

-------------------------------------------------------------------------------------

Microservice Distributed Caching:

Caching: Storing frequently accessed data in a cache that can be quickly accessed from memory. When number of requests are increased, caching provide to handle requests with high availability.

Problems:

1. Slow and low performance communication
2. Latency while publishing and subscribing events
3. Rest API make DB calls that are expensive and low performance.

Types of Caching:

1. In Memory cache: data is lost when machine is restarted.
2. Disk Cache: store in SSD
3. Distributed cache: cache is distributed across multiple machine

Cache hit: when requested data is found in the cache.

Cache miss: When data is not found in the cache.

**Caching Strategies:**

1. Cache Aside: Microservice will check cache first before requesting data from db.
2. Read Through: When cache miss, cache reads the data from db and populate back to cache and the client read it from cache.
3. Write Through: A microservice first write data to cache and then to the database, it is synchronous operation
4. Write Back, Write Behind: The data written to the cache is asynchronously updated in the main database.