## Authentication & Authorization Approach

#### ****1. Authentication Approach****

* The system will use **JWT-based authentication** to ensure that only registered sellers can manage their products.
* When a seller logs in using their email and password, a **secure JWT token** is generated and returned.
* The JWT token contains:
  + Seller’s **User ID**
  + Seller’s Email
  + **Role (e.g., "Seller", "Admin")**
  + Token **expiration time**
* This token must be included in the Authorization header (Bearer <token>) in all subsequent API requests.
* The server verifies the JWT signature before processing any request using authentication middleware.

#### ****2. Authorization Approach****

* **Role-Based Access Control (RBAC)** is enforced to limit actions based on user roles.
* **The authorization middleware** will check the decoded JWT token and compare the **role** in the request with the role passed during routing.
* Sellers can **only manage their own products** based on the sellerId in the request.

##### ****Alternative Approaches (Optional Enhancements)****

* **OAuth 2.0** with AWS Cognito/Auth0 for external identity management.
* **Multi-Factor Authentication (MFA)** for added security.
* **Session-based authentication** (if using stateful authentication for specific needs).

**Fine-Grained Permissions with AWS IAM (if applicable)**

* Use **AWS IAM policies** to control access based on attributes like seller ID.
* For event-driven services, enforce **resource-based policies** (e.g., SNS topic access).

**Token Expiry & Refresh Mechanism**

* JWT tokens should have **short lifetimes** (e.g., 15 minutes) for security.
* Implement a **refresh token mechanism** to generate new tokens without requiring a new login.

#### ****Security Best Practices****

* Use **HTTPS (TLS/SSL)** for encrypted communication.
* Store **passwords securely** with hashing algorithms like **bcrypt**.
* Implement **rate limiting & IP whitelisting** to prevent abuse.
* Use **CORS policies** to control API access from external sources.
* Regularly **rotate and revoke tokens** if suspicious activity is detected.
* Implement **audit logging** for authentication and authorization events.

## Scalability - Handling 10,000 Requests Per Second

To ensure high throughput and efficient scaling, the system leverages **serverless, containerized, and event-driven architectures** with **asynchronous processing**.

### ****Scalability Approaches****

1. **Horizontally Scalable API Layer**

* Deploy **Node.js APIs** using **AWS ECS (Elastic Container Service) with Fargate** or **AWS Lambda** for auto-scaling.
* Use **Auto Scaling Groups (ASG)** to dynamically adjust capacity based on traffic.
* Implement **Node.js clustering** to utilize multiple CPU cores.

1. **Asynchronous Processing with AWS SNS + SQS**

* Offload heavy workloads (e.g., notifications, analytics) to **AWS SQS** for background processing.
* Use **Lambda or worker nodes** to process events asynchronously, ensuring system responsiveness.

1. **Efficient Data Storage**

* Use **DynamoDB** with **on-demand capacity mode** for auto-scaling storage.
* Optimize query performance with **Global Secondary Indexes (GSI)**.
* Enable **DynamoDB Streams** for real-time data synchronization.

1. **API Gateway & Caching**

* Deploy **AWS API Gateway** with **CloudFront caching** to minimize API load.
* Use **AWS ElastiCache (Redis)** for frequently accessed product data, reducing database queries.

1. **Load Balancing**

* Utilize **AWS Application Load Balancer (ALB)** to distribute traffic across ECS tasks.
* Implement sticky sessions for optimized request routing.

### ****Why This Approach?****

* **Auto-Scalable** – ECS, Lambda, and DynamoDB scale automatically.
* **High-Performance** – Redis caching and async event processing reduce response times.
* **Cost-Efficient** – On-demand scaling prevents unnecessary resource allocation.

**Eventual Consistency for Analytics and Notifications**

In an event-driven system, **eventual consistency** ensures that updates propagate asynchronously across services, preventing bottlenecks and improving scalability.

### **Eventual Consistency **Approaches****

1. **AWS SNS + SQS for Asynchronous Event Processing**

* When a product is created or updated, an event is published to **AWS SNS** (e.g., "ProductCreated", "ProductUpdated").
* Multiple subscribers (e.g., **analytics service, notification service**) consume these events via **AWS SQS**, ensuring non-blocking operations.
* If a subscriber fails, **SQS retains messages** until successfully processed.

1. **Idempotency to Prevent Duplicate Processing**

* Store **event IDs** in a database to avoid processing the same event multiple times.
* Use **DynamoDB conditional writes** to ensure updates only occur if the data has changed.

1. **Event Replay and Failure Handling**

* Store event logs in **AWS S3** for **historical event replay** in case of failures.
* Implement **Dead Letter Queues (DLQ)** in SQS to capture and debug failed messages.

1. **DynamoDB Streams for Change Data Capture (CDC)**

* Enables **real-time updates** by streaming database changes to event consumers.
* Used to keep **analytics and notifications in sync** with product updates.

1. **Handling Latency with Eventual Consistency**

* Implement **caching and read replicas** to improve read consistency while updates propagate.
* Use **cache invalidation** mechanisms to refresh stale product data.
* Implement **retry logic with exponential backoff** to handle transient failures.

### ****Why This Approach?****

* **Ensures Reliability** – No message loss due to retries, DLQs, and event replay.
* **Efficient Event Processing** – Asynchronous updates reduce system load and improve responsiveness.
* **Cost-Effective** – Processing is optimized to handle only necessary updates.

## ****Performance Optimization Using Node.js Advanced Concepts****

To maximize performance, the system leverages **Node.js internals**, including asynchronous programming, clustering, and caching, ensuring efficient handling of high-traffic workloads.

### ****Performance Optimization Approaches****

1. **Asynchronous and Non-Blocking Architecture**

* APIs are fully **async/await** to handle multiple concurrent requests without blocking the event loop.
* Offload **CPU-intensive tasks** to **worker threads** to prevent main thread congestion.

1. **Node.js Clustering for Multi-Core Utilization**

* Use the **Cluster module** to distribute traffic across multiple processes and **fully utilize CPU cores**.
* Implement **PM2 process manager** to monitor and auto-restart processes in case of failures.

1. **Efficient Data Streaming and Compression**

* Use **Node.js Streams** for efficient handling of large datasets (e.g., bulk product updates).
* Implement **zlib compression streams** to reduce payload size, improving response times.

1. **Caching with Redis (ElastiCache)**

* Frequently accessed API responses are stored in **Redis**, reducing redundant database queries.
* Implement **TTL-based caching** and **cache invalidation mechanisms** for freshness.

1. **Database Connection Pooling and Optimization**

* Maintain a **DynamoDB connection pool** using the AWS SDK to manage and reuse database connections efficiently.
* Use **DynamoDB Global Secondary Indexes (GSI)** for optimized query performance.

1. **Rate Limiting & Load Balancing**

* **AWS API Gateway** enforces request limits per user to prevent API abuse.
* **AWS ALB (Application Load Balancer)** distributes traffic across multiple ECS instances.

1. **Memory and Garbage Collection Management**

* Monitor **Node.js V8 garbage collection** to prevent memory leaks.
* Optimize event listeners and **free up memory after processing**.

### ****Why This Approach?****

* **Handles High Traffic** – Asynchronous, non-blocking APIs prevent request bottlenecks.
* **Reduces Database Load** – Cached responses speed up read-heavy operations.
* **Optimized Memory Usage** – Efficient process management and garbage collection prevent memory bloat.
* **Scalable and Resilient** – Clustering, load balancing, and connection pooling enhance reliability.

## ****Failures in Message Broker Consumers & Mitigation Strategies****

To ensure reliable event processing, the system implements **resilient consumer strategies** to handle failures in AWS SNS + SQS.

**Potential Failures & Mitigation Strategies**

| **Failure Type** | **Description** | **Mitigation Approach** |
| --- | --- | --- |
| **Message Loss** | Events may be lost before processing. | Use **AWS SQS with Dead Letter Queues (DLQ)** to store unprocessed messages. Enable **SNS message retries** for transient failures. |
| **Duplicate Processing** | The same event might be processed multiple times due to at-least-once delivery. | Implement **idempotency checks** by tracking event IDs in DynamoDB to prevent duplicate execution. |
| **Poison Messages** | Invalid messages that always fail (e.g., incorrect format) can clog the queue. | Route failing messages to **DLQs** for debugging and manual intervention. |
| **High Latency in Message Processing** | Consumers may not process messages fast enough, causing a backlog. | **Auto-scale consumers dynamically** based on SQS queue depth using AWS Lambda or ECS Service Auto Scaling. |
| **Consumer Crash** | If the consumer service crashes, messages may remain unprocessed. | Use **Auto Scaling and ECS self-healing** to restart failed consumers automatically. |
| **Out-of-Order Event Processing** | Events may arrive out of sequence due to distributed processing. | Implement **event versioning and timestamp checks** to ensure correct ordering in the database. |
| **Slow Message Processing** | Heavy processing tasks may delay new message consumption. | Offload expensive operations to **asynchronous workers** and implement **batch processing** where applicable. |

### ****Why This Approach?****

* **Ensures Reliable Event Processing** – Messages are never lost due to retries and DLQs.
* **Handles Failures Gracefully** – Auto-scaling prevents bottlenecks and congestion.
* **Prevents Data Integrity Issues** – Idempotency and event ordering ensure consistency.
* **Optimized for High-Throughput Systems** – Consumers scale dynamically to handle load spikes.

## ****SNS vs SQS – Choosing the Right Approach****

### ****Current Approach (Single Topic, Multiple Event Types)****

**Advantages**

* Simpler infrastructure (single SNS topic).
* Lower cost with fewer resources.
* Easier to subscribe to all product-related events.

**Disadvantages**

* Less control over filtering.
* All subscribers receive all events, filtering them manually.

### ****Separate SNS Topics Approach****

**Advantages**

* Better separation of concerns.
* Fine-grained access control per event.
* More scalable in the long run.

**Disadvantages**

* More complex infrastructure.
* Higher costs due to multiple SNS topics.

### ****Recommended Approach****

* For now, **keeping the current single-topic approach is better** because:
* Both events are closely related to products.
* Both events have similar subscribers.
* The system scale does not yet require separation.

**S**eparate topics** should be considered when:**

* Different teams need separate event access.
* The number of event types grows significantly.
* Event processing needs vary across services.

## ****When to Use SNS vs SQS vs Both?****

| **Feature** | **SNS (Pub/Sub)** | **SQS (Queue)** | **SNS + SQS** |
| --- | --- | --- | --- |
| **Use Case** | Broadcast messages | Queue-based processing | Scalable, reliable messaging |
| **Subscribers** | Multiple services | One service | Multiple durable subscribers |
| **Processing** | Real-time | Async processing | Both |
| **Durability** | No persistence | Stores messages | Ensures delivery |

* **Use SNS** for **event broadcasting**.
* **Use SQS** for **message durability & retries**.
* **Use SNS + SQS** for **reliable event-driven architecture**.

## ****AWS Hosting, Deployment, and Event Handling****

### ****Best Approaches for a Scalable and Event-Driven System****

1. **Compute & API Hosting**

* **AWS ECS (Fargate) for API Layer** – Deploy **Node.js Express APIs** in a containerized environment with auto-scaling.
* **AWS Lambda for Event Processing** – Handle asynchronous events triggered by **SNS + SQS** for tasks like notifications and analytics.

1. **AWS Infrastructure Components**

* **API Gateway** – Manages external API requests with rate limiting and caching.
* **DynamoDB** – Stores product and event data with on-demand scaling.
* **S3** – Stores event logs and backups for auditing and recovery.
* **CloudWatch & X-Ray** – Monitors API performance, logs errors, and traces request flows.

1. **Event-Driven Architecture**

* **AWS SNS + SQS** – Ensures asynchronous event processing with retries and message durability.
* **DynamoDB Streams** – Captures real-time product updates for event-driven workflows.

1. **CI/CD Pipeline for Automated Deployments**

* **AWS CodePipeline + CodeBuild** – Automates build, testing, and deployment.
* **Infrastructure-as-Code (IaC)** – Use **Terraform** or **AWS CDK** for consistent and repeatable deployments.

### ****Why This Approach?****

* **Highly Scalable** – ECS & Lambda auto-scale based on traffic.
* **Resilient & Fault-Tolerant** – SNS/SQS ensures reliable message delivery.
* **Optimized for Performance** – API Gateway + CloudFront caching reduces latency.
* **Automated Deployments** – CI/CD pipelines improve efficiency and reliability.