**Design Doc for Online Automotive Dealership System**

**Overview:** The **Online Automotive Dealership System** is a scalable platform designed to connect multiple automotive dealers with customers. Rather than being limited to a single dealer's inventory, the platform aggregates inventories from many dealers, enabling customers to browse a wide selection of vehicles based on dealer location and other criteria. Dealers can opt to join the platform, providing access to their inventory for customers to view and purchase. This system enhances the online car-buying experience for customers and helps streamline the sales process for dealers by automating inventory and order management.

**Clients or Target audience:**

* **Automotive Dealers**: dealers who can list and manage their inventory on the platform
* **Customers:** Buyers who can browse list of vehicles from various dealers and filter results based on proximity and place orders
* **Administrators:** Platform administrators who manage user accounts, monitor dealer data, and ensure the platform’s smooth operation.

**Features / Functionalities / Services:**

1. **User Management Service**

* Login Module

Admin Login functionality, Authentication Mechanism, Role Based Access control

* Logout Functionality

Admin logout to terminate session and clear session data.

1. **Admin Dashboard Service**

* Car Overview**:** Display statistics of available vehicles, orders, and customer activities
* Car Details**:** Admin can access detailed information for each vehicle in the system.
* Order Overview: View of a summary of customer orders, payment statuses and deliveries.

1. **Vehicle Management Service Module**

* Adding New Vehicle Details**:** Admin can add new vehicles to the platform with specified details.
* Edit Existing Details**:** Admin can edit or update the information of existing inventory.
* View All Vehicle Info: Admin can view the complete list of vehicle inventory.

1. **Order Management Service Module**

* Adding New Order Details: Admin can add new orders when customers make a purchase or request a test drive
* Edit Existing Order Details:Admin can modify the existing order details.
* View All Order Details:Admin can view all the list of orders, their progress , maintain statuses such as delivery and payment.

1. **Vehicle History Report Service**

* Allow users to input either the VIN number or License Plate number to request a vehicle's history report.
* Integrate with external official APIs or services to pull vehicle history data.
* Maintain a record of past vehicle history searches for both admin and customer users.
* Ensure secure transmission of sensitive input data using encryption protocols (e.g., HTTPS, SSL/TLS).

1. **Notification Service**

* Send alerts or updates to users via email, SMS, or push notifications.
* Support for multiple channels (email, SMS, push).
* Templates for common notifications.
* Admin dashboard for managing notification settings.

1. **Geolocation Service**

* Helps users find dealerships based on their location or preferred search radius.
* Integration with mapping APIs (e.g., Google Maps).
* Dealer search by distance or region.
* Real-time location updates for delivery tracking (optional).

**Compliance standards for dealers and customers/users**

1. **Compliance for Dealers**

* Dealers must provide accurate and up-to-date information about their inventory.
* Vehicles listed must meet local and national regulations (e.g., emissions standards, safety requirements).
* Dealers must authorize the system to pull inventory data through approved APIs or manual uploads.
* Dealers must comply with data privacy laws such as GDPR (if applicable).

1. **Compliance for Customers**

* Customers must provide accurate personal details during registration (e.g., name, email, phone number).
* The system will not store sensitive financial information like credit card numbers.
* Customers must agree to the platform's privacy policy before registering.

1. **Platform-Wide Standards**

* Regular audits will ensure that dealers comply with the platform’s standards for inventory and sales.
* All data exchanges between customers, dealers, and the platform will use secure protocols (e.g., HTTPS, encrypted APIs).

**Database Schema**

**Users**

* UserId (Primary Key)
* Name
* Email
* PasswordHash
* Role (Admin, Dealer, Customer)
* PhoneNumber
* Address
* CreatedAt

**Vehicles**

* VehicleId (Primary Key)
* DealerId (Foreign Key → Users)
* VIN
* LicensePlate
* Make
* Model
* Year
* Price
* Category
* Description
* Status (Available, Sold)
* CreatedAt

**Orders**

* OrderId (Primary Key)
* CustomerId (Foreign Key → Users)
* VehicleId (Foreign Key → Vehicles)
* OrderDate
* TotalAmount
* PaymentStatus (Pending, Paid, Failed)
* DeliveryStatus (Pending, Shipped, Delivered)

**VehicleHistoryReports**

* ReportId (Primary Key)
* UserId (Foreign Key → Users)
* VIN
* LicensePlate
* ReportData
* RequestedAt

**Notifications**

* NotificationId (Primary Key)
* UserId (Foreign Key → Users)
* Message
* DeliveryChannel (Email, SMS, Push)
* SentAt
* Status (Pending, Sent, Failed)

**DealershipLocations**

* LocationId (Primary Key)
* DealerId (Foreign Key → Users)
* Latitude
* Longitude
* Address

**Logs**

* LogId (Primary Key)
* LogType (Error, Warning, Info)
* Message
* Timestamp
* RelatedUserId (Foreign Key → Users)

**High Level Architecture**

**A diagram of a service

Description automatically generated**

**Message Queue Story**

In the Online Automotive Dealership System, Apache Kafka is implemented as the backbone for inter-service communication, enabling seamless data exchange among microservices. For instance, when a customer places an order, the Order Management Service publishes an "Order Created" event to a Kafka topic. The Notification Service consumes this event to send order confirmation to the customer, while the Dashboard Service consumes the same event to update real-time statistics. Similarly, when the Vehicle History Report Service completes fetching vehicle data, it publishes a "Report Generated" event, which triggers the Notification Service to inform the user. The Geolocation Service also uses Kafka for sending location updates to the dashboard whenever a dealer’s location is updated.

By using Apache Kafka, the system ensures loose coupling between microservices, supports high-throughput event processing, and provides reliability through message retention and replication. The key challenges include handling message schema evolution and ensuring idempotency for message processing, addressed by implementing Avro schemas and distributed locks, respectively. This event-driven architecture not only enhances scalability but also ensures real-time responsiveness across the platform.

**Biggest Technical Challenge: Maintaining data consistency across services**

In a microservices architecture like the Online Automotive Dealership System, where services like Login, Dashboard, and Car Management operate independently yet share interdependent data, maintaining data consistency becomes the most significant challenge. For example, when a dealer updates vehicle details, the Dashboard and other consumers need to reflect these updates accurately and in a timely manner.

**Solution Approach**

Instead of storing only the current state of data, every change in the system is recorded as an immutable event. For instance, when a vehicle is added, an event like vehicle. added is stored in a log (Kafka topic). Services consuming this log can build their state incrementally.

**AWS Architecture**

|  |  |  |
| --- | --- | --- |
| Layer | AWS Service | Type |
| Compute | ECS/EKS, Lambda | Containerized, Serverless |
| Messaging | MKS, SQS, SNS | Event streaming, Messaging |
| Storage | RDS, S3, DynamoDB | Relational, Object, NoSQL |
| Security | IAM, Cognito, KMS | Identity, Authentication |
| API Management | API Gateway | API Management |
| Monitoring | CloudWatch, X-Ray | Observability |
| Caching | ElastiCache | In-Memory caching |
| Content Delivery | CloudFront | CDN |
| Load Balancing | ALB, Auto Scaling Groups | Traffic Distribution |
| CI/CD | CodePipeline, CodeBuild | DevOps Automation |

**Monitoring**

**API Monitoring:** Track API performance, availability and usage. Used built-in API gateway metrics in CloudWatch to monitor request count and latencies, Integration errors and 4xx/5x errors. Used Spring Boot Actuator to expose application health and performance metrics. Integrate Actuator endpoints with tools like Prometheus and Grafana for real-time API monitoring. Used tools like AWS CloudWatch Synthetics or third-party services (Postman, Pingdom) to simulate user interactions and measure API availability.

**Log Monitoring:** Used AWS CloudWatch to aggregate logs from all services. Forward logs from AWS CloudWatch to AWS OpenSearch Service for log indexing and search.

**JVM Monitoring:** Enable JMX Monitoring in your Spring Boot applications. Use tools like **VisualVM**, **JConsole**, or **Prometheus JMX Exporter** to collect JVM metrics. Publish JVM metrics (e.g., heap memory usage, garbage collection, thread count) from services to CloudWatch.

**Production Support Story**

The production support workflow for the Online Automotive Dealership System ensures seamless operation and quick issue resolution. When a dealer reported inventory updates not appearing and customers faced stuck orders, monitoring tools like CloudWatch and Grafana alerted the support team to high Kafka consumer lag and API errors in the Order Management Service. Logs revealed that a burst of inventory updates overwhelmed Kafka consumers, delaying message processing. The team scaled up consumer instances, applied producer rate limits, and resolved the issue within 90 minutes, restoring normal operations. Post-incident, dynamic scaling for Kafka consumers, prioritization of critical messages, and improved alert thresholds were implemented to prevent recurrence.

**Performance Tuning**

Added indexes to optimize queries, implemented Redis caching for frequent data, and refactored APIs to use asynchronous communication via Kafka. Also scaled ECS tasks dynamically using AWS Application Auto Scaling. These changes cut response times by 40%, reduced query delays by 60%, and restored smooth performance, ensuring the system could handle growth effortlessly.

**REST APIs**

* **Login**

API Endpoint:POST /api/auth/login

Request Body: {“username": "user", "password": "pass" }

Response body: {“token": "JWT\_TOKEN" }

* **View All Cars**

API Endpoint: GET /api/cars

Query Parameters: ?category=suv&priceRange=20000-40000

Response Body: [ { "id": 1, "make": "Toyota", "model": "Camry", "price": 30000 }, ... ]

* **Get Vehicle History**

API Endpoint: GET /api/vehicle-history

Query parameters: ?vin=1HGCM82633A123456

Response: { "vin": "1HGCM82633A123456", "accidents": 2, "owners": 3, "status": "Clean" }

* **Send Notification**

API Endpoint: POST /api/notifications

Request body: {“userId": 123, "message": "Your order has been shipped!" }

Response: {“status": "Notification sent" }

**Recent API developed**

**GET:** /api/v1/cars/search

**Description:** Searches for cars based on various criteria as query string parameters.

Example: GET/api/v1/cars/search?category=SUV&min\_price=20000&max\_price=40000&availability=true.

**Data Flow Diagrams**

1. **Data Flow for Order Placement**

**A diagram of a car sharing process

Description automatically generated**

1. **Authentication and Access flow**

**A diagram of a service

Description automatically generated**

**Prepare stories based on your resume: example, where did you use multi-threading in your last project? Where did you use builder design patterns in your last project?**

Calculating insurance premiums involves complex rules, formulas, and possibly external API calls for real-time data (e.g., risk scores). Multi-threading is used to handle multiple calculations concurrently, reducing response time.

When constructing quotes with multiple optional attributes (e.g., coverage type, deductible amount, policy period, customer-specific discounts), the Builder pattern provided a clean and flexible way to assemble quote objects incrementally.

**Come up team size**: 4 Full Stack developers, 2 DevOps (CI/CD), 2 Testing.

**Daily Active User**: let's assume, 10 agents \* 20 companies + 200 customers \* 20% customers log to view policies = 240

**TPS (Transactions Per Second):** Assume each user generates 20 transactions/day on average, Spread evenly over 24 hours: TPS = 240 \* 20 / 24 \* 60 \* 60 = 0.55 TPS

**QPS (Queries Per Second):** For every transactional request, assume 2-3 read operations (e.g., fetching user data, policy details, etc.).

Average QPS = TPS × 3 ≈ 1.11 QPS.

Peak QPS = Peak TPS × 3 ≈ 11.1 QPS. (assuming peak traffic 10x average)

**Design a ci/cd pipeline flow to (AWS / local) depends on your resume project (if you want to keep AWS). Yes**

GitHub/GitLab

↓

[Source Stage: Pull Code]

↓

[Build Stage: Compile → Unit Tests → Build Docker Images]

↓

[Test Stage: Deploy to Staging → Integration & API Tests]

↓

[Deploy Stage: Deploy to AWS EKS/ECS or Local Kubernetes]

↓

[Monitor Stage: Monitor Logs, Metrics, Alerts]

**Frontend Story:** In the Dealership Platform Service, I used Angular to build a dynamic and responsive UI, developing reusable components for key features like vehicle listings, dashboards, and booking forms. I integrated RESTful APIs to fetch real-time data, optimized performance using lazy loading and OnPush change detection. The UI was enhanced with Angular Material, while form validations, real-time notifications via AWS SNS, and smooth routing with Angular Router improved user experience. This resulted in a highly interactive, scalable, and efficient frontend for seamless user engagement.**Monitor tools**

Prometheus + Grafana: For JVM, API, and infrastructure monitoring.

AWS CloudWatch + X-Ray: For alerting and distributed tracing.

ELK Stack: For centralized log aggregation and analysis.

New Relic or Datadog: For a unified view of application and database performance.