ChatGPT

Advanced Spring Boot E-Commerce Microservices Project Prompt (Story Map Aligned)

Create a **microservices-based e-commerce backend** with Spring Boot (latest stable version, Java 17+). This system is designed for modularity, security, scalability, real-time monitoring, and user-centric features. The following prompt integrates all the requirements from the story map – including advanced Spring Boot Admin monitoring, AI-driven analytics, enhanced user engagement, PWA/offline support, and other production-ready considerations – into one comprehensive project specification.

Core Business Microservices & Features

User Service

- **Customer and Admin Accounts:** Supports customer and administrator registration & login (with local credentials **and** OAuth2 login via Google/Facebook).
- **Profile Management:** Full user profile management with personal details (name, contact info), **address book** for multiple shipping addresses (like Amazon/Flipkart), and profile updates.
- **Role-Based Access Control:** Enforce roles and permissions (regular customer vs. admin users) across services.
- **JWT Authentication:** A dedicated authentication service issues JWTs upon login, and all other services validate tokens for secure, stateless authentication of requests.

Product Catalog Service

- **Product Browsing & Search:** Browse product listings with advanced search filters (by keyword, category, brand, etc.).
- **Product Management:** Full CRUD functionality for products and their attributes (with admin roles managing product entries).
- **AI-Driven Recommendations:** Provide personalized product recommendations using hybrid approaches (collaborative filtering + content-based). Includes trending products, personalized feeds, and “recently viewed” item tracking for each user.
- **Catalog Scalability:** Efficient querying and indexing for fast searches; ready for large catalog sizes and high read throughput.

Cart & Payment Service

- **Shopping Cart & Wishlist:** Add to cart or wishlist, update item quantities, and remove items. Persist carts per user session and sync across devices if logged in.
- **Discounts & Coupons:** Apply discount codes or coupons at checkout with validation rules.
- **Multiple Payment Options:** Integrate multiple payment methods (UPI, wallet, credit/debit card, net banking) – using mocked third-party payment integrations for demo purposes.

- **Secure Payment Data:** Securely store payment details (e.g. card tokens or masked card info) for quick checkout in demo mode. (In a real production setup, sensitive card data would be handled via a compliant vault or external PCI service.)
- **Invoices & Refunds:** Generate downloadable order invoices. Handle order cancellation and refund processes (with appropriate business rules and status updates to orders).

Order Service

- **Order Placement:** Process checkout to create orders, with all relevant details (items, prices, discounts, user info, shipping address).
- **Real-Time Order Status:** Track order status in real-time (e.g. **Pending**, **Confirmed**, **Shipped**, **Delivered**), with updates broadcast via notifications.
- **Order History:** Allow users to view their past orders and details. Enable administrators to search and manage orders.
- **Cancellations & Returns:** Support order cancellation (if item not yet shipped) and initiation of refund/exchange requests, coordinating with Payment and Inventory services to update stock and issue refunds.

Inventory Service

- **Stock Management:** Maintain product stock levels and reservations. Updates inventory in real-time based on orders placed, cancellations, and returns.
- **Synchronization:** Ensure consistency between inventory and orders – e.g. decrement stock on order placement, restore stock on cancellation/refund.
- **Alerts for Low Stock:** (Optional) trigger notifications or events when inventory for a product falls below a threshold (could integrate with Notification Service or Analytics for reordering insights).

Notification Service

- **Multi-Channel Notifications:** Send out email, SMS, and push notifications for various events (order confirmations, status updates, promotional offers, admin announcements).
- **Event-Driven Architecture:** Uses an event stream (Kafka topics) to decouple notification triggers from the main flow – e.g. an Order Placed event leads to an order confirmation notification. Push notification support is built-in (FCM or web push) for real-time alerts.
- **User Preferences:** Respects user notification preferences (e.g. opt-in/opt-out for marketing emails or SMS) by integrating with the Customer Engagement APIs service.

Offers & Loyalty Service

- **Loyalty Points:** Accrue and manage loyalty/reward points for user activities (purchases, reviews, referrals, etc.). Allow users to redeem points on purchases.
- **Promotions & Campaigns:** Create and track discount campaigns, promo codes, and special offers (e.g. seasonal sales, flash deals). Apply eligible discounts in the Cart/Order flow.
- **Usage Analytics:** Track promotion usage and effectiveness (e.g. how many users used a coupon) and integrate with the Analytics Service for reporting.

Analytics Service

- **KPI Dashboards:** Aggregate key performance indicators across the platform – sales figures, revenue, active users, conversion rates, product views, etc. Also track user activity metrics and product review statistics.
- **Customer Segmentation & Behavior:** Analyze customer data to derive segments (e.g. frequent buyers, high-value customers) and shopping behavior trends. This can feed into personalized recommendations or targeted campaigns.
- **Reporting:** Provide data for dashboards/reports on campaigns, notification delivery success (email/SMS open rates, push notification clicks), and other business insights. Potential integration with third-party analytics or AI tools for advanced insights.

Customer Engagement APIs

- **Preference Management:** Endpoints for users to set and update their notification preferences (channels on/off, frequency, etc.) and personalization settings (e.g. categories of interest for recommendations).
- **Event Tracking:** APIs to log user engagement events (product clicks, wishlist additions, cart abandons) which can be used by the Analytics and Recommendation systems to tailor the user experience.
- **Marketing Hooks:** Provide hooks for marketing systems – for example, an API to trigger a promotional email or in-app message to a specific segment, or to record when a user interacts with a marketing campaign.

PWA & Offline Support

- **Push Notifications:** Full support for web/mobile push notifications (for order status updates, promotional offers, and other marketing messages) via the Notification Service and compatible with service workers in a PWA.
- **Offline Catalog Browsing:** Certain product catalog APIs are optimized to allow caching for offline use. Users can browse previously loaded products/categories even without connectivity (leveraging service worker cache strategies on the frontend).
- **Background Sync:** Supports background synchronization for critical actions. For example, if an order is placed or an item added to cart while offline, the request can be retried automatically when back online. All such write operations are designed to be **stateless** and **idempotent** (so they can safely be retried without duplication).
- **PWA Analytics:** Provide APIs for collecting PWA-specific usage data (like how often users interact offline, sync events, etc.), enabling the Analytics Service to present PWA engagement metrics in dashboards.

Infrastructure & Cross-Cutting Concerns

- **API Gateway:** Use Spring Cloud Gateway as a single entry point to route requests to appropriate services. The gateway handles centralized authentication (JWT validation), rate limiting, and can serve as a point for cross-cutting policies (CORS, required headers, etc.).
- **Service Discovery:** All services register with Spring Cloud Eureka for discovery. This allows the system to locate service instances (for inter-service calls or for the gateway to forward requests) and enables easy scaling of services.

- **Circuit Breaker:** Implement resilience with Resilience4j (circuit breakers, retries, bulkheads) on inter-service calls. This prevents cascading failures – e.g. if the recommendation engine is down, fall back to a default response without impacting the entire app.
- **Transaction Management:** Use Spring's `@Transactional` where appropriate for atomic operations within each service's domain. For cross-service consistency (saga patterns or outbox patterns could be considered), ensure eventual consistency via events (for example, inventory updates on order events).
- **Global Exception Handling:** Implement `@ControllerAdvice` in each service to handle exceptions and return standardized error responses (with proper HTTP status codes and error codes) to the API consumers.
- **Distributed Messaging:** Use Kafka as a message broker for asynchronous communication and event-driven workflows between microservices. Key events (order placed, payment processed, inventory low, etc.) are published to topics that interested services can consume, decoupling direct dependencies.
- **Externalized Configuration:** Manage all configuration using Spring Cloud Config (backed by a central config repository or server) so that environment-specific properties and feature toggles can be managed without altering code. This supports profiles for dev/staging/prod with secure handling of secrets.
- **OpenAPI/Swagger Documentation:** Each microservice provides an OpenAPI (Swagger) specification. This allows easy inspection of endpoints and models for developers or integration partners, and can be aggregated in the gateway or a developer portal.
- **Automated Testing:** Implement comprehensive tests for each service (unit tests with JUnit 5 and Mockito, and integration tests for critical flows). Consider contract testing for APIs between microservices. A CI pipeline will run tests on each service to catch issues early.
- **Logging & Monitoring:** Use SLF4J with Logback (or Log4j2) for structured logging in JSON format. Each service emits logs with correlation IDs (e.g. trace IDs from Sleuth/OpenTelemetry) to tie together requests across services. Monitoring is enabled via Spring Boot Actuator for health, metrics, and info endpoints on each service.
- **Caching (Optional):** Leverage caching (e.g. Redis or Caffeine) for frequently accessed data like product catalog queries, to improve performance and reduce load on databases. Ensure cache is used consistently and invalidated on relevant updates (product changes, inventory changes, etc.).

Advanced Admin & Monitoring

- **Spring Boot Admin Server:** Deploy a Spring Boot Admin server as a centralized dashboard (running on its own, e.g. on port 9090, behind a Kubernetes Ingress). It discovers and monitors all microservices via their Actuator endpoints (health, metrics, etc.) and also integrates with Eureka for service status.
- **System Metrics Monitoring:** Spring Boot Admin (and associated Micrometer/Actuator metrics) collects JVM metrics, memory and thread usage, HTTP call stats, etc., in real time. This gives the ops team visibility into each service's performance and resource usage. Optionally integrate with Prometheus/Grafana for detailed time-series monitoring.
- **Dynamic Log Level & JVM Management:** The admin interface allows on-the-fly changing of log levels for troubleshooting. It can also expose JMX or thread dump info to diagnose issues in real time without restarting services.

- **Alerting Integration:** Configure alert channels for critical health or performance issues. For example, use Slack for real-time alerts (primary channel), send email for audit trails of alerts, and integrate with PagerDuty for escalation of production incidents that require immediate attention.
- **Admin UI Security (RBAC):** Secure the Spring Boot Admin UI with role-based access. For instance, operations team members have full access to all features, developers may have access to view logs or debug-level info, and auditors or stakeholders get read-only access. This prevents unauthorized changes (like log level tweaks) by non-ops personnel.
- **Automated Remediation Hooks:** Set up webhooks and automation scripts triggered by certain alerts or threshold breaches. For example, if memory usage is critically high or a service is unresponsive, a webhook could trigger a runbook via PagerDuty or OpsGenie, or even initiate automated scaling or restart scripts in Kubernetes. This ensures faster recovery and less manual intervention for known issues.

Tech Stack

- **Java & Spring Framework:** Spring Boot (latest stable release) with Java 17+ for all microservices.
- **Spring Cloud Stack:** Spring Cloud Gateway, Spring Cloud Eureka (Discovery), Spring Cloud Config (external config server) to handle infrastructure needs.
- **Database & Persistence:** Spring Data JPA with PostgreSQL in production (each service having its own schema or database as needed), and H2 in-memory databases for development/testing purposes.
- **Messaging & Streaming:** Apache Kafka for messaging. Spring for Apache Kafka to integrate producers/consumers in the microservices.
- **Resilience and Observability:** Resilience4j for circuit breaking/retries, Spring Boot Actuator and Micrometer for metrics, logs through SLF4J/Logback. (Optionally, OpenTelemetry/Zipkin for distributed tracing.)
- **Containerization & Orchestration:** Docker for containerizing each microservice, and Kubernetes for deploying all services (YAML manifests or Helm charts to manage deployments, services, config maps, etc.).
- **Logging & Monitoring Tools:** Centralized logging (ELK stack or cloud logging service) and monitoring dashboards (like Grafana connected to Prometheus or CloudWatch) can be used in conjunction with Spring Boot Admin for a full observability suite.

Exclusions

- **No UI/Frontend:** The project is backend-focused. (No Angular/React or other frontend code is included — this is purely the microservices backend and supporting infrastructure).
- **Mocked Payment Gateway:** The payment integrations are simulated for demonstration and learning purposes. (No real payment gateway integration or PCI compliance scope in this project.)
- **Microservices Only:** The project will not include any monolithic application structure; all features are delivered as independent microservices to align with modern architectural best practices.
- **Focused AI/ML Scope:** AI/ML usage is limited to recommendation engine and analytics insights as described. There is no complex machine learning pipeline beyond the recommendation system and basic analytics – the emphasis is on integration rather than developing new ML algorithms.

Deliverables

- **Modular Codebase:** A complete modular project structure with separate Spring Boot services for each of the above business domains, plus infrastructure services (Gateway, Config, Eureka, Admin Server, etc.). Each service is independently buildable and deployable.
- **Code Samples & Patterns:** Implementation of key patterns and integrations, including sample code for JWT authentication and OAuth2 login flows, Resilience4j circuit breaker configuration, global exception handling, Kafka producers/consumers for one or two critical events, and transactional outbox logic for order and payment workflow. Also include code for the Spring Boot Admin client in each service (to register with Admin Server) and server setup.
- **Configuration Files:** OAuth2 client configuration for Google/Facebook login, Firebase or webpush configuration (if using FCM for notifications), and any social login secrets (in a secure manner). Also, Spring Cloud Config repository files or YAML configurations for different environments.
- **Deployment Scripts/Manifests:** Dockerfiles for each service, and Kubernetes manifest files (or a Helm chart) to deploy the entire suite (including Deployments, Services, Ingress for gateway, ConfigMap/Secret for configuration, etc.). Instructions or scripts to run everything with Docker Compose locally could be included as well.
- **API Documentation:** Full OpenAPI/Swagger definitions for every microservice, detailing all endpoints, request/response schemas, and auth requirements. These can be compiled into a documentation portal or accessed via each service's Swagger UI.
- **Admin & Monitoring Setup:** Configuration for the Spring Boot Admin UI server including user roles and access settings, plus integration details for Slack/PagerDuty alerts (e.g. Slack webhooks, PagerDuty integration keys). Also include any custom scripts or webhook configurations used for automated remediation.

This prompt defines a **feature-complete, scalable, and production-ready e-commerce backend**. It aligns with the story map and incorporates enterprise-grade concerns for monitoring, observability, security, and PWA/mobile readiness. Following this specification will result in a robust backend platform similar to industry leaders, ready for further extension or frontend integration.
