■ Modular Monolith → Microservices Migration Checklist

1■■ Preparation

- Identify service boundaries - Each feature package becomes a candidate microservice.
- Check dependencies between features - Ensure no feature directly accesses another's domain/infra objects.
- Extract shared utilities - Logging, error handling, base response models, security filters → move to a common library JAR.

2■■ Infrastructure Setup

- Create separate Spring Boot projects for each microservice.
- Database strategy: Each microservice has its own schema (no shared tables).
- Service discovery & gateway: Add API Gateway (Spring Cloud Gateway, Nginx, or Kong).
- Centralized config: Use Spring Cloud Config or HashiCorp Vault for configs and secrets.

3■■ Communication

- Synchronous: Use REST APIs (OpenAPI + Feign clients for calls).
- Asynchronous: Use Kafka/RabbitMQ for events (for decoupling & scalability).
- Error handling: Define standard error response contracts (shared in common lib).

4■■ Observability

- Centralized logging (ELK stack, Loki, or Cloud logs).
- Metrics & monitoring (Micrometer + Prometheus + Grafana).
- Distributed tracing (Jaeger or Zipkin with Sleuth/OpenTelemetry).

5■■ Migration Strategy

- Start with 1 feature (e.g., businesstype).
- Copy its package into a new Spring Boot project.
- Expose the same REST API from that service.
- Replace direct monolith calls → REST call to new service (or Kafka event).
- Gradually migrate more features one by one.

6■■ Deployment

- Containerize each service with Docker.
- Use Kubernetes or Docker Compose for orchestration.
- Automate builds with CI/CD (Jenkins/GitHub Actions/GitLab CI).

7■■ Cleanup

- Remove migrated feature code from the monolith.
- Keep only the gateway + shared libs.
- Monitor performance and scale services independently.

■ Recommended Migration Order

- Pick a less critical feature first → businesstype (pilot).
- Next, extract read-heavy features (e.g., product).
- Finally, extract transaction-heavy features (e.g., order, payment).