OrderFlow Backend – Day-by-Day Plan (Always Orders Domain)

This document contains the complete "Day-by-Day Plan (always Orders, no task → order rename)" response, including all explanations, steps, and code snippets, formatted for reference.

You will not rename or switch domain mid-way. The project starts with Orders and stays with Orders all the way to the final application.

This is a linear, step-by-step plan from zero → full OrderFlow app. Each day says exactly what to build: endpoints, DTOs, validation, and data.

Final Architecture You'll End Up With (so you know where this is going):

```
orderflow/
  services/
    orders-svc/         # NestJS + Postgres + Kafka producer + consumer
    inventory-svc/      # NestJS + Postgres + Kafka consumer + producer
    notifications-svc/  # NestJS + Redis + Bull queue + Kafka consumer
  infra/
    docker-compose.yml  # Postgres, Kafka, Redis, Kong, services
    k8s/                # Kubernetes manifests (later)
```

Main components:

- orders-svc: REST API for /orders, SQL DB, publishes & consumes Kafka events.
- inventory-svc: listens to order.created, checks stock, publishes order.reserved / order.failed.
- notifications-svc: listens to order.reserved / order.failed, pushes jobs to Redis/Bull.
- Kong: front door for /orders, JWT + rate limits.
- GitHub Actions: basic CI (lint, test, build).

Day-by-Day Plan (always Orders, no task → order rename)

Day 1 – Create repo & first Nest app

- Goal: Have a NestJS project running with a health endpoint.
- Create folder: orderflow/.
- Inside it, init git: git init.
- Create first service with Nest CLI: npm i -g @nestjs/cli, then nest new services/orders-svc.
- In orders-svc/src/app.controller.ts (or new health controller), add a GET /health endpoint returning { status: 'ok' }.
- Run: npm run start:dev and hit GET http://localhost:3000/health.

```
@Get('health')
getHealth() {
  return { status: 'ok' };
}
```

Day 2 – Orders module + in-memory orders list

- Goal: Have /orders returning fake data.
- Generate module/controller/service:
- nest g module orders
- nest g controller orders
- nest g service orders
- Define a Order type file with OrderItem, OrderStatus, and Order interfaces.
- In OrdersService, create an in-memory orders array.
- In OrdersController, inject OrdersService and implement GET /orders to return all orders.

```
export interface OrderItem {
  productId: string;
  quantity: number;
}

export type OrderStatus = 'pending' | 'reserved' | 'failed' | 'cancelled';

export interface Order {
  id: number;
  userId: string;
  items: OrderItem[];
  status: OrderStatus;
  totalAmount: number;
  createdAt: string;
```

```
  updatedAt: string;
}

@Injectable()
export class OrdersService {
  private orders: Order[] = [];

  findAll() {
    return this.orders;
  }
}

@Controller('orders')
export class OrdersController {
  constructor(private readonly ordersService: OrdersService) {}

  @Get()
  getOrders() {
    return this.ordersService.findAll();
  }
}
```

Day 3 – CreateOrder DTO + POST /orders (in-memory)

- Goal: Create orders in memory, with validation.
- Install validation libs: npm i class-validator class-transformer.
- Enable global ValidationPipe in main.ts with whitelist, forbidNonWhitelisted, transform.
- Create CreateOrderDto and OrderItemDto with appropriate validation decorators.
- In OrdersService, implement a create method that generates an auto-incremented id and pushes the new order into the in-memory array.
- In OrdersController, implement POST /orders to call OrdersService.create.
- Test POST /orders with a valid body and confirm it appears in GET /orders.

```
export class OrderItemDto {
  @IsString()
  productId: string;

  @IsInt()
  @Min(1)
  quantity: number;
}

export class CreateOrderDto {
  @IsString()
  userId: string;

  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => OrderItemDto)
  items: OrderItemDto[];
}
```

```
private nextId = 1;

create(dto: CreateOrderDto): Order {
  const now = new Date().toISOString();
  const totalAmount = 0; // keep simple for now

  const order: Order = {
    id: this.nextId++,
    userId: dto.userId,
    items: dto.items,
    status: 'pending',
    totalAmount,
    createdAt: now,
    updatedAt: now,
  };

  this.orders.push(order);
  return order;
}

@Post()
createOrder(@Body() dto: CreateOrderDto) {
  return this.ordersService.create(dto);
}
```

Day 4 – GET /orders/:id, update & delete (still in-memory)

- Goal: All basic CRUD routes working.
- In OrdersService add methods: findOne, updateStatus, remove.
- In OrdersController add endpoints: GET /orders/:id, PATCH /orders/:id/status, DELETE /orders/:id.
- Ensure NotFoundException is thrown when a requested order id does not exist.
- Test: create, get by id, update status, delete.

```
findOne(id: number) {
  const order = this.orders.find(o => o.id === id);
  if (!order) throw new NotFoundException('Order not found');
  return order;
}

updateStatus(id: number, status: OrderStatus) {
  const order = this.findOne(id);
  order.status = status;
  order.updatedAt = new Date().toISOString();
  return order;
}

remove(id: number) {
  const index = this.orders.findIndex(o => o.id === id);
  if (index === -1) throw new NotFoundException('Order not found');
  this.orders.splice(index, 1);
}
```

```
@Get(':id')
getOrder(@Param('id', ParseIntPipe) id: number) {
  return this.ordersService.findOne(id);
}

@Patch(':id/status')
updateStatus(
  @Param('id', ParseIntPipe) id: number,
  @Body('status') status: OrderStatus,
) {
  return this.ordersService.updateStatus(id, status);
}

@Delete(':id')
deleteOrder(@Param('id', ParseIntPipe) id: number) {
  this.ordersService.remove(id);
  return { message: 'Order deleted' };
}
```

Day 5 – Connect Postgres, move orders to DB

- Goal: Same endpoints, but real DB table.
- Add Postgres via Docker:
- docker run --name orderflow-postgres -e POSTGRES_PASSWORD=postgres -e POSTGRES_DB=orderflow -p 5432:5432 -d postgres:15
- Install TypeORM and configuration packages: npm i @nestjs/typeorm typeorm pg @nestjs/config.
- In AppModule, add ConfigModule.forRoot({ isGlobal: true }) and TypeOrmModule.forRoot using environment variables.
- Create .env file with DB_HOST, DB_PORT, DB_USER, DB_PASS, DB_NAME.
- Create an Order entity matching your in-memory type, and register TypeOrmModule.forFeature([Order]) in OrdersModule.
- Refactor OrdersService to use Repository instead of the in-memory array. Endpoints stay identical.

Day 6 – Better validation + pagination on /orders

- Goal: More realistic API.
- Extend CreateOrderDto to validate items.length > 0 and optionally compute or validate totalAmount.
- Add query params for GET /orders: /orders?status=pending&page;=1&limit;=10.
- Add a service method using findAndCount for pagination.
- Return a response structure that includes items and total count.

Day 7 – Clean up & minimal tests

- Goal: Stabilize orders-svc before adding other services.
- Use Nest Logger to log order creation and any failures.
- Write unit tests for OrdersService basic flows (create, find, update, delete).
- Write one e2e test for POST /orders and GET /orders using @nestjs/testing and Supertest.
- Update README for orders-svc: how to run, list of endpoints.

Day 8 – Split repo structure, introduce inventory-svc project

- Goal: Prepare for microservices without renaming anything.
- Reorganize root structure as orderflow/services/orders-svc and orderflow/services/inventory-svc.
- Use Nest CLI to create inventory-svc inside services/inventory-svc.
- Add InventoryModule, InventoryService, InventoryController.
- Implement GET /inventory/health that returns { status: 'ok' }.

Day 9 – Introduce Kafka infra (docker-compose) & test producer

- Goal: Kafka cluster running, orders-svc can send a test message.
- Create infra/docker-compose.yml with services for Postgres, Zookeeper, and Kafka.
- Stop any standalone Postgres container and use docker-compose instead.
- Install kafkajs in orders-svc.
- Create KafkaProducerService in orders-svc with @Injectable that connects to Kafka and exposes emit(topic, payload).
- Call emit('order.test', { hello: 'world' }) once on app bootstrap to confirm connectivity.

Day 10 – Publish real order.created events from orders-svc

- Goal: Every created order leads to order.created event.
- Define an OrderCreatedEvent TypeScript interface.
- In OrdersService.create, after saving the order to DB, publish an order.created event via KafkaProducerService.
- Log any failures when publishing events but do not break the API response.
- Confirm events appear in Kafka using a consumer or logs.

Day 11 – Kafka consumer in inventory-svc, log events

- Goal: inventory-svc prints order.created events.
- Install kafkajs in inventory-svc.
- Create KafkaConsumerService that connects to Kafka and subscribes to order.created topic.
- For each message, parse JSON and log "Received order.created for orderId X".
- No DB changes yet, just logging.

Day 12 – Inventory DB & inventory entity

- Goal: inventory-svc has its own DB table for stock.
- Reuse the same Postgres instance via compose, but create tables for inventory items.
- Connect TypeORM in inventory-svc.
- Create InventoryItem entity with productId, availableQuantity, reservedQuantity, updatedAt.
- Seed some inventory data manually using a script or startup logic.

Day 13 – On order.created, check and reserve stock

- Goal: inventory-svc processes events and decides if order can be reserved.
- In the Kafka consumer handler in inventory-svc, for each order item, check if availableQuantity >= quantity.
- If all items have enough stock, mark as reservable and update the DB.
- If any item is short, plan to fail the order.
- For now, update inventory quantities in the DB and log whether the order would be reserved or failed.

Day 14 – Have inventory-svc publish order.reserved / order.failed

- Goal: Real events flowing out of inventory-svc.
- Add a Kafka producer to inventory-svc similar to orders-svc.
- When reservation succeeds, publish order.reserved with orderId and items.
- When it fails, publish order.failed with orderId and reason.
- Log every published event with enough detail to debug.

Day 15 – orders-svc consumes inventory results, updates status

- Goal: Completed order lifecycle in DB.
- In orders-svc, add a Kafka consumer for order.reserved and order.failed topics.

- On order.reserved, update the order's status to reserved.
- On order.failed, update the order's status to failed and record the failure reason.
- Ensure GET /orders/:id now shows the latest status and optional failReason.

Day 16 – notifications-svc skeleton + Redis/Bull

- Goal: Third service ready to accept jobs.
- Create notifications-svc using Nest CLI in services/notifications-svc.
- Add Redis service to docker-compose.
- Install Bull or BullMQ and ioredis in notifications-svc.
- Set up a simple notifications queue and a worker that logs any job it processes.

Day 17 – notifications-svc consumes Kafka & enqueues jobs

- Goal: events → jobs in queue.
- Install Kafka client in notifications-svc.
- Subscribe to order.reserved and order.failed topics.
- For each event, create a job in the notifications queue with orderId, userId, eventType, and message.
- Make the worker log simulated email or log notification messages.

Day 18 – Add retry logic & small stats endpoint

- Goal: more realistic notifications.
- Configure job retry behavior (e.g. 3 attempts with exponential backoff).
- Mark jobs as dead-letter or permanently failed after max retries.
- Add GET /notifications/health to confirm Redis connectivity.
- Add GET /notifications/stats to return simple counters for processed and failed jobs.
- Document how to use these endpoints during debugging.

Day 19 – Dockerize each service

- Goal: whole system runs via docker-compose.
- Add Dockerfile into orders-svc, inventory-svc, and notifications-svc.
- Build images and ensure each service runs correctly with environment variables in containers.
- Update infra/docker-compose.yml to run 3 services + Postgres + Kafka + Redis.

- Run docker compose up and verify main flows still work.

Day 20 – Add Kong to docker-compose

- Goal: use Kong as the external entry point.
- Add Kong and (optionally) a database or DB-less configuration to docker-compose.
- Configure Kong routes so that /orders points to orders-svc.
- Test POST /orders via Kong's public URL instead of calling orders-svc directly.

Day 21 – JWT auth via Kong + simple auth in orders-svc

- Goal: basic security over the Orders API.
- Implement minimal JWT validation in orders-svc (an AuthGuard or simple middleware).
- Configure Kong JWT plugin to require a token on /orders routes.
- Test calls without token (should be 401) and with a valid token (should be 200).

Day 22 – Basic rate limiting in Kong

- Goal: protect Orders API against abuse.
- Configure Kong rate limiting plugin on /orders (e.g. 100 requests/min per consumer).
- Run a script or repeated calls to /orders to confirm that rate limiting is enforced.
- Adjust limits as necessary for demo vs development.

Day 23 – Kubernetes manifests (optional but recommended)

- Goal: have YAML ready for real cluster deployments.
- Create infra/k8s/ directory with deployments and services for each NestJS service.
- Add ConfigMaps and Secrets for shared configuration and secrets.
- Configure readiness and liveness probes for each deployment hitting /health.
- Apply manifests to a local cluster (minikube, kind, or similar) and verify pods are running.

Day 24 – GitHub Actions CI

- Goal: automated tests and build on every push.
- Add .github/workflows/ci.yml at repo root.

- Configure steps to install dependencies, run lint, run tests, and build each service.
- Ensure the pipeline runs successfully for the current codebase.

Day 25 – Documentation & diagrams

- Goal: clear explanation for others (and future you).
- Create an architecture diagram showing services, Kafka, Redis, Postgres, and Kong.
- Write a high-level technical overview explaining how an order flows through the system.
- Ensure each service has a README describing its purpose, endpoints, and environment variables.
- Document event contracts and DB schemas.

Day 26 – End-to-end manual testing

- Goal: validate the full system behavior.
- Run the full stack and perform: POST /orders via Kong.
- Observe order.created in Kafka, inventory decision, order.reserved/order.failed, and notifications being enqueued and processed.
- Verify final order status and logs for notifications.
- Fix any bugs discovered during this test.

Day 27 – Final cleanup & preparation

- Goal: make the project presentable and interview-ready.
- Remove unused code and commented-out blocks.
- Ensure all required environment variables are documented and/or there is an .env.example file.
- Tag a v1.0.0 release in version control.
- Write final notes for how you will explain this project in an interview: architecture, trade-offs, lessons learned.