Big Picture (before details)

This diagram represents an **E-commerce system built with Microservices**, using:

- **Spring Boot 3**
- **Spring Cloud**
- **Microservices architecture**
- **Synchronous communication (HTTP/REST)**
- **Asynchronous communication (Kafka)**
- **API Gateway**
- **Service Discovery (Eureka)**
- **Centralized Configuration (Config Server)**
- **Distributed Tracing (Zipkin)**
- **One database per service (MongoDB)**

Now let's follow the flow **step by step**, as if we were tracking a real request.

STEP 1 — Client request (Frontend)

On the left side, you see the **Angular icon**.

This represents:

- A frontend application (Angular / React / etc.)
- The user sends requests like:
  - `/customers`
  - `/products`
  - `/orders`

Important rule:

> The frontend **never talks directly to microservices**.

There is **only one entry point**.

STEP 2 — API Gateway (single entry point)

```
Frontend → API Gateway
```

The **API Gateway** works like a **building gatekeeper**.

Its responsibilities:

- Route requests (`/customers → customer-service`)
- Security (JWT / Keycloak)

- Rate limiting
- Logging
- Centralized access

Example:

```
GET /customers → Customer Service
GET /products  → Product Service
POST /orders   → Order Service
```

Without a gateway, the frontend would need to know every service (bad design).

STEP 3 — Core Microservices

Inside the **Private Network** (dashed box) are the microservices.

Customer Service

Responsible for:

- Customer data
- Registration
- Queries

Database:

- **MongoDB (exclusive to this service)**

Typical endpoints:

```
GET /customers
POST /customers
```

Microservices rule:

Each service owns its own database.

Product Service

Responsible for:

- Products
- Prices
- (Sometimes stock control)

Database:

- **MongoDB**

Receives requests from:

- API Gateway

- Order Service

Order Service (the brain of the system)

This is the **most important service**.

It:

- Creates orders
- Talks to:
  - Customer Service
  - Product Service
- Starts the payment process

Typical flow:

```
Frontend → Gateway → Order Service
```

Then:

- Validate customer
- Validate products
- Save the order

STEP 4 — Synchronous communication (HTTP)

The **blue dashed lines** represent **REST calls** between services.

Examples:

```
Order → Product
Order → Customer
Order → Payment
```

This communication is:

- Synchronous
- Blocking
- One service waits for the other

That’s why we use:

- Timeouts
- Retries
- Circuit breakers

STEP 5 — Payment Service

The **Payment Service**:

- Processes the payment
- Confirms approval or failure
- Does NOT send notifications directly

Bad practice:

```
Payment → Notification (direct call)
```

Instead, we use **events**.

STEP 6 — Kafka (asynchronous messaging)

Kafka is the **message broker**.

What happens:

- Payment Service **publishes events**
- Example events:

```
PaymentConfirmed
OrderConfirmed
```

This is **asynchronous**:

- Payment does not wait
- It just publishes the event

Benefits:

- Loose coupling
- High scalability
- Fault tolerance

STEP 7 — Notification Service

The **Notification Service**:

- Subscribes to Kafka topics
- Listens for events

Examples:

```
PaymentConfirmed → send email
OrderConfirmed → send notification
```

Important:

- Notification does NOT know Payment
- Notification does NOT know Order

- It only knows the **event**

This is **event-driven architecture**.

STEP 8 — Zipkin (Distributed Tracing)

Zipkin is used to:

- Trace a request across all services
- Example trace:

```
Gateway → Order → Product → Payment → Kafka → Notification
```

It helps to:

- Measure latency
- Find bottlenecks
- Debug production issues

Without tracing, microservices become a nightmare to debug.

STEP 9 — Eureka Server (Service Discovery)

Eureka acts as a **service registry**.

Each service registers itself:

```
order-service
product-service
customer-service
```

When one service calls another:

- No fixed IPs
- Uses **service name**

Example:

```
http://PRODUCT-SERVICE/products
```

STEP 10 — Config Server (Centralized configuration)

The **Config Server**:

- Stores all configuration in one place
- Usually backed by Git

Examples:

```
spring.datasource.url
kafka.bootstrap.servers
server.port
```

Benefits:

- No rebuild needed for config changes
- Environment-based configs (dev / prod)

FULL REQUEST FLOW (order creation)

```
1. User creates an order
2. Frontend → API Gateway
3. Gateway → Order Service
4. Order calls Customer and Product
5. Order calls Payment
6. Payment confirms payment
7. Payment publishes event to Kafka
8. Notification consumes the event
9. Email / notification is sent
10. Zipkin traces the whole flow
```

Why this is “real” microservices architecture

Because it includes:

- API Gateway
- Service Discovery
- Centralized configuration
- Database per service
- Async communication
- Observability
- Loose coupling

Microservices Architecture in Node.js + TypeScript

This is the **Node.js equivalent** of the Spring Boot + Spring Cloud architecture you saw.

Core Tech Stack (Node.js world)

Runtime & Language

- **Node.js**
- **TypeScript** (mandatory for serious microservices)

Framework (choose one)

- **NestJS** (closest to Spring Boot)
    - Dependency Injection

- Modules
- Controllers
- Interceptors

- *(Express is possible, but NestJS is the industry favorite for this)*

I'll assume **NestJS**.

API Gateway (Node.js version)

Tooling

- **NestJS Gateway** or
- **Kong / Nginx** (production-grade)

Responsibilities

- Single entry point
- Routing
- Authentication (JWT / OAuth2 / Keycloak)
- Rate limiting
- Request validation

Example

```
/api/customers → customer-service
/api/products  → product-service
/api/orders    → order-service
```

In NestJS:

```
@Controller('api')
export class GatewayController {}
```

Service Discovery (Eureka equivalent)

Node.js Options

- **Consul** (most common)
- Kubernetes DNS (if using k8s)
- Hardcoded URLs (only for local dev)

In Node.js, **Kubernetes replaces Eureka** in most real systems.

Example:

```
http://product-service:3000
```

Config Server (Spring Config → Node.js)

Node.js equivalents

- **dotenv**
- **@nestjs/config**
- HashiCorp **Vault**
- ConfigMaps (Kubernetes)

Example:

```
ConfigModule.forRoot({
  isGlobal: true,
});
```

Core Microservices (NestJS + TypeScript)

Each microservice is **its own project**.

Customer Service

Responsibilities

- Customer CRUD

Tech

- NestJS
- MongoDB (Mongoose / Prisma)

```
@Module({
  imports: [MongooseModule.forFeature([{ name: 'Customer', schema }])]
})
export class CustomerModule {}
```

Product Service

Responsibilities

- Products
- Prices

Tech

- NestJS
- MongoDB / PostgreSQL

```
@Injectable()
export class ProductService {
  findById(id: string) {}
}
```

Order Service (central service)

Responsibilities

- Create orders
- Call Customer & Product services
- Trigger payment

Communication

- HTTP (Axios / Fetch)
- Async events (Kafka)

```
await this.httpService.get('http://product-service/products');
```

Inter-service Communication (HTTP)

Tools

- **Axios**
- **node-fetch**
- **@nestjs/axios**

Add:

- Timeout
- Retry
- Circuit breaker

Use:

- **opossum** (circuit breaker)
- **axios-retry**

Payment Service

Responsibilities

- Process payments
- Publish events

No direct calls to Notification

```
this.kafkaProducer.send({
  topic: 'payment-confirmed',
  messages: [{ value: JSON.stringify(event) }]
});
```

Kafka (Message Broker)

Node.js Kafka Clients

- **kafkajs**
- **node-rdkafka**

Topics

```
payment-confirmed
order-confirmed
```

Notification Service

Responsibilities

- Consume Kafka events
- Send emails / push notifications

```
consumer.subscribe({ topic: 'payment-confirmed' });
consumer.run({
  eachMessage: async ({ message }) => {
    // send email
  }
});
```

Database per Service

Service	Database
Customer	MongoDB
Product	MongoDB
Order	PostgreSQL
Payment	PostgreSQL
Notification	MongoDB

Never share databases between services

Distributed Tracing (Zipkin equivalent)

Node.js Tools

- **OpenTelemetry**
- Jaeger / Zipkin

```
NodeSDK({
  traceExporter: new ZipkinExporter(),
});
```

Logging & Observability

Logging

- **Winston**
- **Pino**

Metrics

- **Prometheus**
- **Grafana**

Security

Authentication

- **JWT**
- **Keycloak**
- OAuth2

Tools

- `passport`
- `@nestjs/passport`

Containerization

Docker

Each service:

```
FROM node:20
WORKDIR /app
COPY . .
RUN npm install
CMD ["npm", "run", "start"]
```

Docker Compose (local dev)

```
services:
  order-service:
  product-service:
  kafka:
  mongodb:
```

FULL FLOW (Node.js version)

```
1. User → API Gateway
2. Gateway → Order Service
3. Order → Customer & Product (HTTP)
```

```
4. Order → Payment
5. Payment → Kafka (event)
6. Notification ← Kafka
7. Email sent
8. Traces sent to Zipkin
```

Why NestJS fits microservices so well

Because it gives you:

- Dependency Injection
- Clean architecture
- Middleware / Interceptors
- Easy Kafka, HTTP, gRPC support
- Type safety with TypeScript

It feels like Spring Boot for Node.js