API:CONF 20 Lagos 26

From Monolith to Microservices

Building APIs that survive production

API Conference Lagos 2026

Want to follow along?

Scan for the setup guide

Everything you need to run the demos on your own machine

Prerequisites

Docker (Compose v2), 8 GB RAM with 4 GB given to Docker, 10 GB free disk.

Pull the images

Setup the repository and pull the images

One command

docker compose up — then a checklist to prove your stack actually works.

Can't run it? Watch instead — nothing today depends on it.

Usman Soliu

Senior Backend Engineer & Tech Lead @ HCMatrix

It's 2 a.m. Checkout is down.

One bad deploy took out the whole API; payments, auth, orders, all in the same process. This workshop is about making sure one sick module can never sink the whole ship again.

Where we're going today

The monolith

A real NestJS app: why it was the right call on day one, and where it hurts at scale.

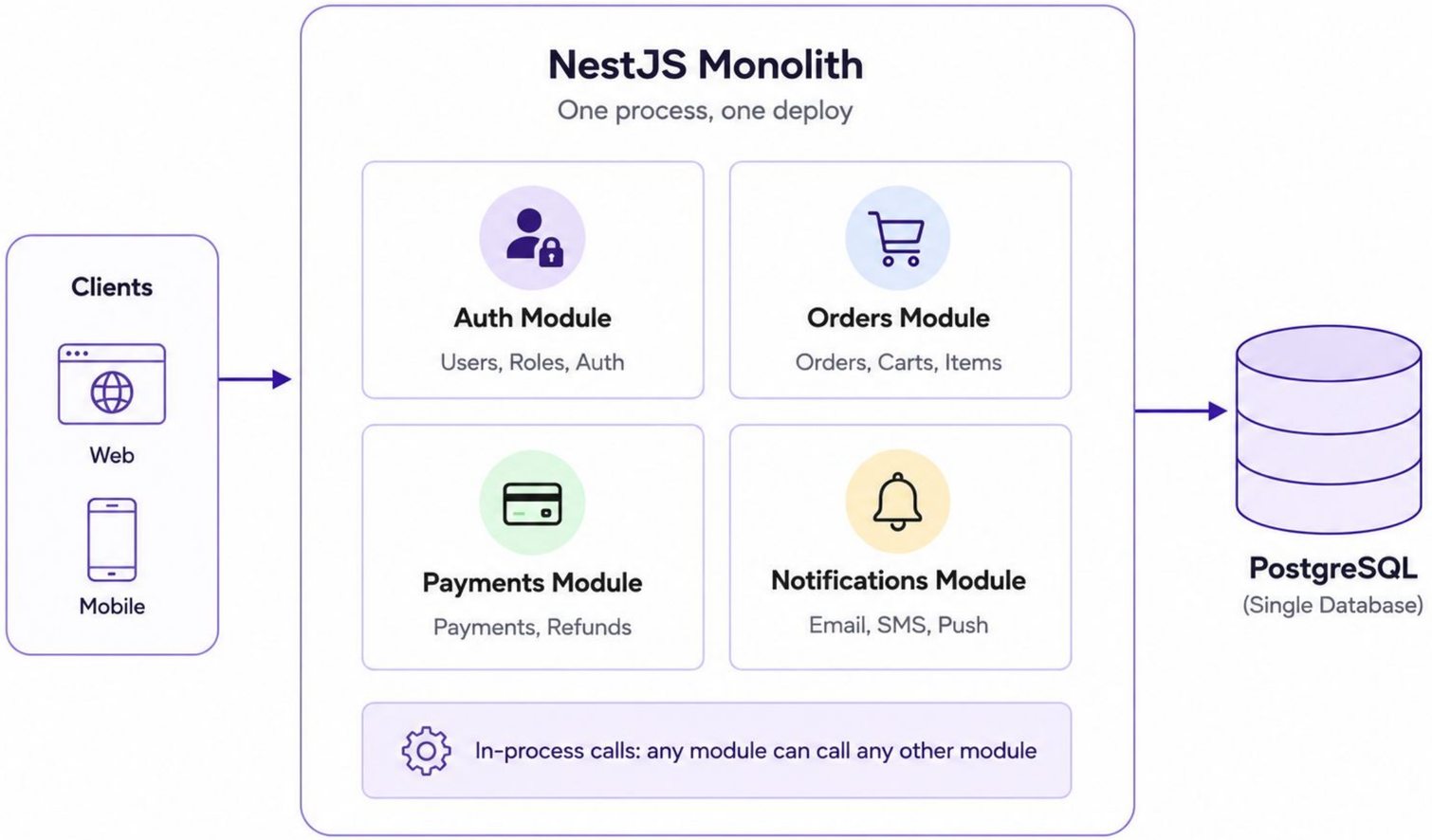
The split

Boundaries, gRPC, RabbitMQ — services that talk to each other without falling over.

Production

Resilience patterns, correlation IDs, tracing, dashboards — then we break it live.

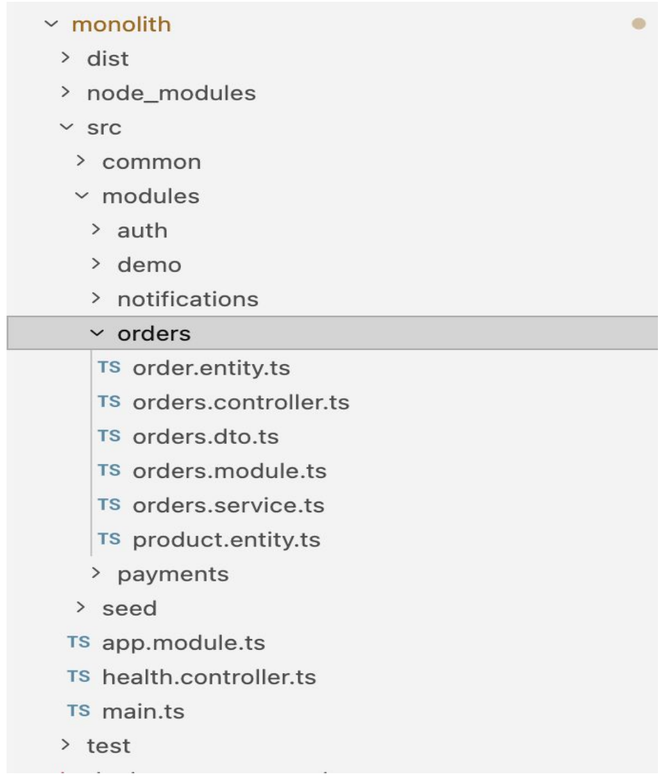
Meet the monolith



One blast radius: any module can take the whole process down.

One AppModule to rule them all

```ts
// app.module.ts — everything lives together
@Module({
  imports: [
    TypeOrmModule.forRoot(dbConfig),   // ONE database
    AuthModule,
    OrdersModule,
    PaymentsModule,
    NotificationsModule,
  ],
})
export class AppModule {}

// orders.service.ts — "just inject it"
@Injectable()
export class OrdersService {
  constructor(
    private payments: PaymentsService,        // direct call
    private notifier: NotificationsService,   // direct call
  ) {}
}
```



Convenient. Fast to build.
Every module can reach into every other module — and given enough time, it will.

Where it starts to hurt

Deploys are all-or-nothing

A typo in the email template redeploys payments. Friday deploys become a personality trait.

Scaling is blunt

Checkout needs 10x capacity; and you pay to scale the PDF generator along with it.

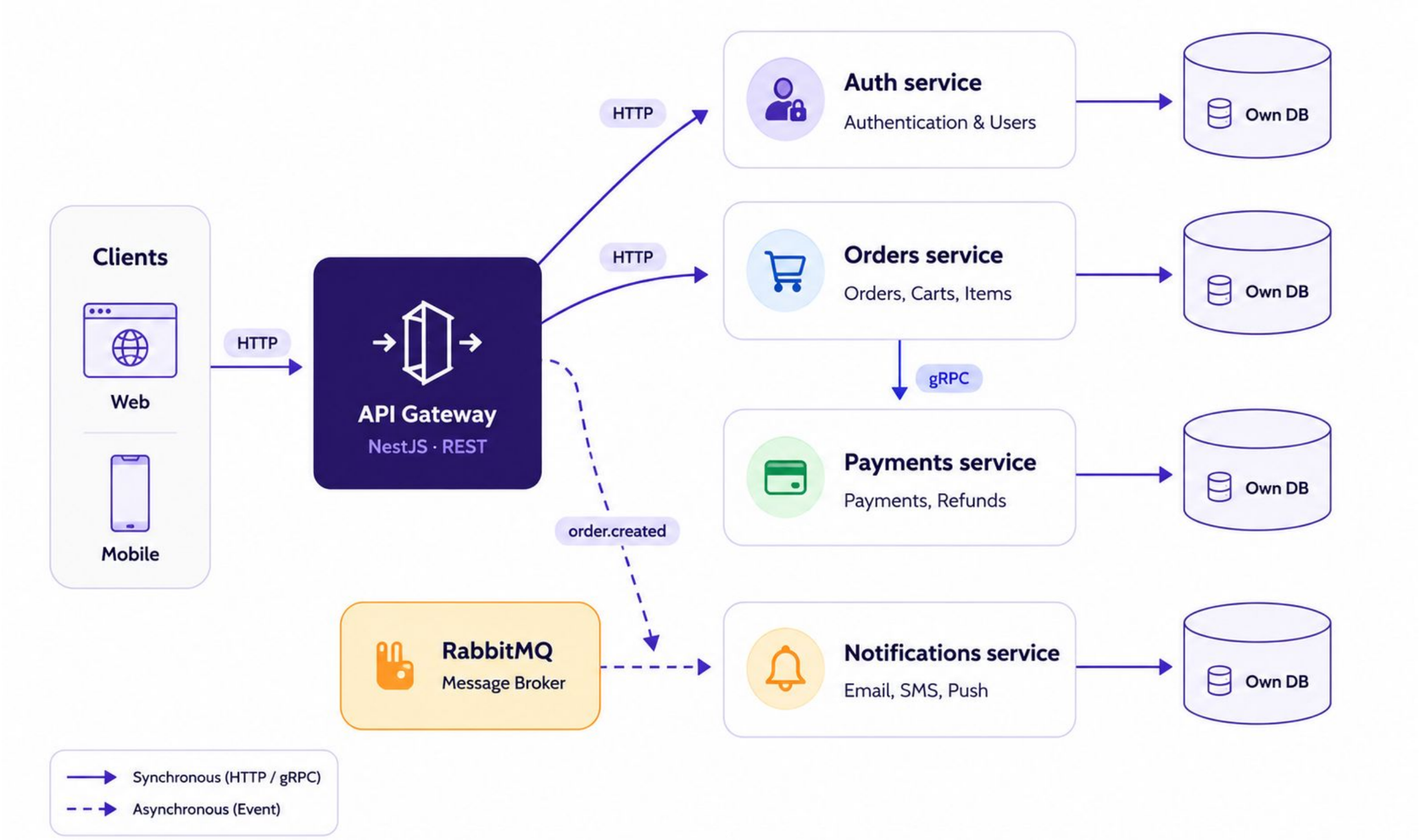
One failure is total failure

A slow report query exhausts the connection pool and takes checkout down with it.

Don't split because a conference told you to

Microservices trade code complexity for operational complexity. Split when teams block each other, when parts scale differently, or when blast radius keeps you up at night; not before. Yes, I am aware I am saying this at a conference.

The target: small services, clear contracts



How to carve the seams

Follow the domain

Split along business capabilities; orders, payments, identity; never technical layers.

Own your data

One service, one database. Shared tables are a distributed monolith in disguise.

Strangle, don't rewrite

Route one capability at a time out of the monolith. Big-bang rewrites die in staging.

Sync calls: gRPC between services

```
// payments.proto — the contract
syntax = 'proto3';
package payments;

service Payments {
  rpc Charge (ChargeRequest)
      returns (ChargeReply);
}
```

The .proto file IS the API.
Typed, versioned, reviewed in PRs; no more guessing what payments expects.

```
// orders: client side
@Client({
  transport: Transport.GRPC,
  options: {
    package: 'payments',
    protoPath: 'proto/payments.proto',
    url: 'payments:50051',
  },
})
client: ClientGrpc;

onModuleInit() {
  this.payments = this.client
    .getService<PaymentsGrpc>('Payments');
}

// payments: server side
@GrpcMethod('Payments', 'Charge')
charge(req: ChargeRequest): ChargeReply {
  return this.charges.execute(req);
}
```

Async events: RabbitMQ

```
// orders.service.ts — publish and move on
async placeOrder(dto: PlaceOrderDto) {
  const order = await this.repo.save(dto);
  await this.amqp.publish(
    'orders',                  // exchange
    'order.created',           // routing key
    { orderId: order.id,
      userId: order.userId,
      total: order.total },
  );
  return order;                // don't wait for email
}

// notifications — consumer
@RabbitSubscribe({
  exchange: 'orders',
  routingKey: 'order.created',
  queue: 'notifications.order-created',
})
async onOrderCreated(evt: OrderCreatedEvent) {
  await this.mailer.sendReceipt(evt);
}
```

Rule of thumb:

Need the answer to respond? → gRPC
Just announcing a fact? → event

An email can be 5 seconds late.
Checkout cannot.

Assume the network hates you

Timeouts everywhere

No call waits forever. A hung dependency should fail fast; not quietly queue up requests.

Retries, with backoff

Retry only idempotent calls, with jitter; or you will DDoS yourself during recovery.

Circuit breakers

Stop calling a dying service. Fail fast, serve a fallback, give it room to recover.

Resilience in ·12 lines

```
// orders → payments, armoured
charge(dto: ChargeDto): Observable<ChargeReply> {
  return this.payments.charge(dto).pipe(
    timeout(800),                          // latency budget (ms)
    retry({
      count: 2,
      delay: (err, attempt) =>
        timer(200 * 2 ** attempt),         // exponential backoff
    }),
    catchError(err =>
      this.breaker.open(err, () =>
        of(this.markPendingReview(dto)))),  // fallback
  );
}
```

NestJS speaks RxJS natively — timeouts and retries are one pipe() away.

The fallback keeps checkout alive: orders land as “pending review” instead of failing.

Correlation IDs: one request, one story

```ts
// correlation-id.middleware.ts
@Injectable()
export class CorrelationIdMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    const id = (req.headers['x-correlation-id'] as string)
            ?? randomUUID();
    req.headers['x-correlation-id'] = id;
    res.setHeader('x-correlation-id', id);
    ctx.run({ correlationId: id }, next);   // AsyncLocalStorage
  }
}

// logger.service.ts — every log line carries the id
log(msg: string, meta = {}) {
  const { correlationId } = ctx.getStore() ?? {};
  console.log(JSON.stringify({ correlationId, msg, ...meta }));
}
```

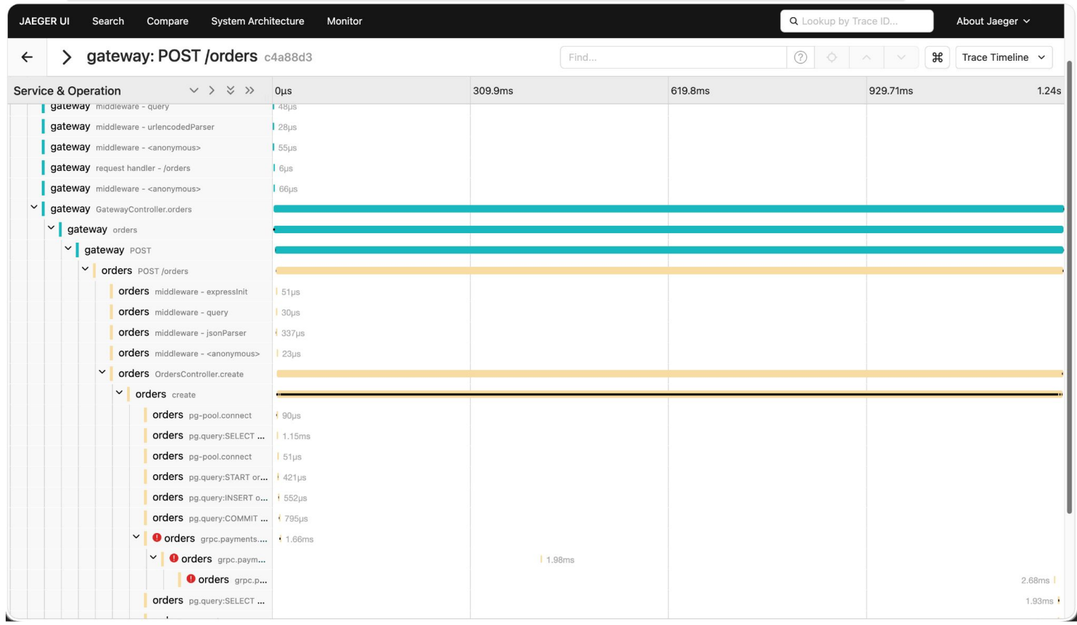
Forward the header on EVERY hop:

- gRPC metadata
- AMQP message headers
- outbound HTTP

One grep, whole request.

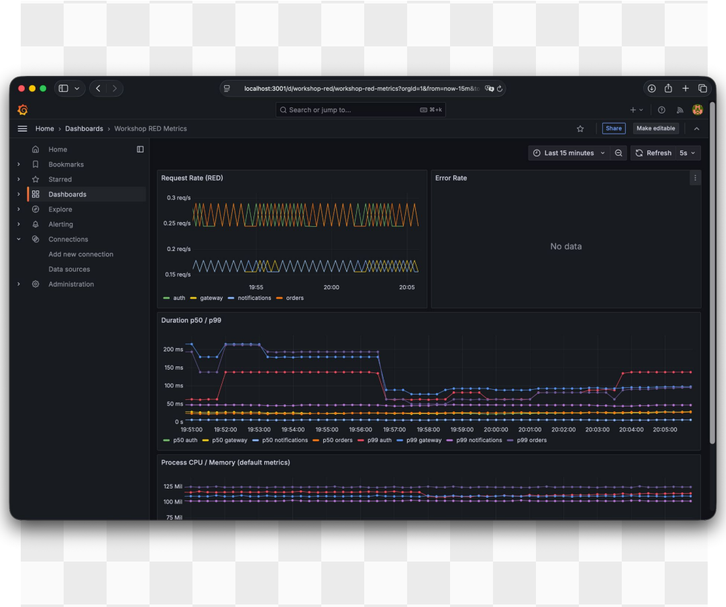
Tracing: OpenTelemetry → Jaeger

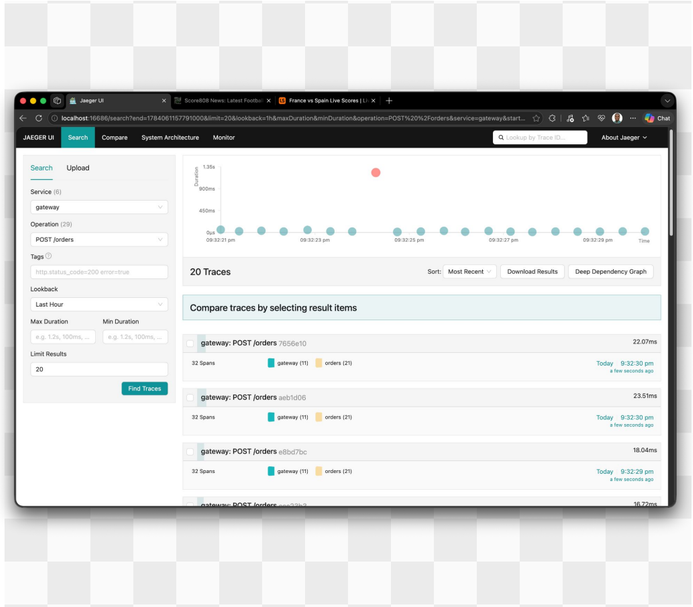
```ts
// tracing.ts — runs BEFORE Nest bootstraps
const sdk = new NodeSDK({
  serviceName: process.env.OTEL_SERVICE_NAME,
  traceExporter: new OTLPTraceExporter({
    url: 'http://jaeger:4318/v1/traces',
  }),
  instrumentations: [getNodeAutoInstrumentations()],
});
sdk.start();

// main.ts
import './tracing';   // first import — order matters
import { NestFactory } from '@nestjs/core';
```

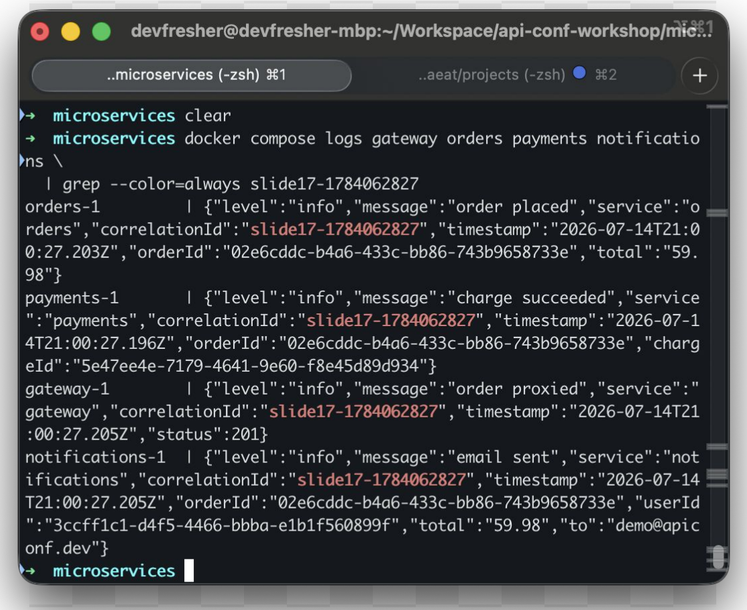


Auto-instrumentation covers HTTP, gRPC, amqplib and pg out of the box — you get cross-service traces before writing a single custom span.

If you can't see it, it's down







Grafana: RED metrics

Rate, errors, duration per service; the first dashboard you build.

Jaeger: trace search

Every slow checkout, findable by correlation ID in seconds.

Terminal: structured logs

JSON lines with correlation IDs; greppable end to end.

Before you ship

Health checks & graceful shutdown

Readiness probes, SIGTERM handling, connection draining. Kubernetes is polite exactly once.

Config from the environment

Twelve-factor config, secrets in a vault — no credentials in the repo. Yes, even that one.

Idempotent consumers + DLQs

Messages WILL arrive twice, late, or broken. Design for redelivery; park the poison in a dead-letter queue.

Let's break it live

Two repos: monolith/ and microservices/, both docker-compose up and ready. We kill the payments service mid-checkout and watch the timeout, the retries, the breaker — and the full trace in Jaeger. Step-by-step walkthrough.

Thank you

usman-soliu-website.onrender.com/apiconf

Connect with me