API/Back-End Developer Intern

Time Limit: 48 hours
Tech Stack: NestJS + TypeScript + PostgreSQL/MySQL
Deployment: Railway / Render / Any Cloud Platform

Business Problem: MSME Vendor Payment Tracking System

QistonPe works with MSMEs who need to manage payments to their vendors/suppliers. These businesses need to track purchase orders, payments made, and outstanding balances to manage their working capital effectively.

You'll build a **backend API system** that handles vendor management, purchase orders, and payment tracking with proper business logic and data integrity.

What You Need to Build

MUST-HAVE Features (Core Requirements)

1. Vendor Management API

Endpoints:

- `POST /vendors` - Create a new vendor
- `GET /vendors` - List all vendors
- `GET /vendors/:id` - Get vendor details with payment summary
- `PUT /vendors/:id` - Update vendor information

Vendor Fields:

- Vendor name (required, unique)
- Contact person
- Email (required, unique, validated)
- Phone number
- Payment terms (7, 15, 30, 45, 60 days)
- Status (Active, Inactive)

Business Rules:

- Vendor name must be unique
- Email validation required
- Cannot create PO for inactive vendors

2. Purchase Order API

Endpoints:

- `POST /purchase-orders` - Create a new purchase order
- `GET /purchase-orders` - List all POs (basic filtering by vendor, status)
- `GET /purchase-orders/:id` - Get PO details with payment history

- `PATCH /purchase-orders/:id/status` - Update PO status

Purchase Order Fields:

- PO Number (auto-generated: PO-YYYYMMDD-XXX)
- Vendor ID (foreign key)
- PO Date
- Total Amount
- Due Date (auto-calculated from PO date + vendor payment terms)
- Status (Draft, Approved, Partially Paid, Fully Paid)
- Items (array of line items with description, quantity, unit price)

Business Rules:

- PO number must be unique and auto-generated
- Total amount = sum of all line items (quantity × unit price)
- Due date auto-calculated based on vendor's payment terms
- Status transitions: Draft → Approved → Partially Paid → Fully Paid

3. Payment Recording API

Endpoints:

- `POST /payments` - Record a payment against a PO
- `GET /payments` - List all payments
- `GET /payments/:id` - Get payment details

Payment Fields:

- Payment reference number (auto-generated: PAY-YYYYMMDD-XXX)
- Purchase Order ID (foreign key)
- Payment date
- Amount paid
- Payment method (Cash, Cheque, NEFT, RTGS, UPI)
- Notes (optional)

Business Rules:

- Cannot pay more than PO outstanding amount
- Payment amount must be positive
- Auto-update PO status based on payment:
    - If total payments = PO amount → Status: Fully Paid
    - If total payments < PO amount → Status: Partially Paid
- Track payment date for aging analysis

4. Analytics API (Minimum 1 Required)

Must Implement At Least ONE of:

- `GET /analytics/vendor-outstanding` - Outstanding balance by vendor
- `GET /analytics/payment-aging` - Aging report (0-30, 31-60, 61-90, 90+ days)

Required Calculations:

- **Outstanding by Vendor:** Group by vendor, sum (PO amount - payments made)
- **Payment Aging:** Group overdue amounts by age buckets (if choosing this option)

NICE-TO-HAVE Features (Bonus Points)

These are **optional** but will earn you extra points:

Advanced Features

- `DELETE /payments/:id` - Void a payment (soft delete) and recalculate PO status
- Pagination on list endpoints (default 20 items, max 100)
- Advanced filtering (date range, multiple status, search by PO number)
- Both analytics endpoints (instead of just one)
- `GET /analytics/payment-trends` - Monthly payment trends (last 6 months)

Authentication

- JWT-based auth endpoint: `POST /auth/login`
- Protect all endpoints with JWT guard
- (Hardcoded user is fine, no registration needed)

Code Quality & Performance

- Unit tests for critical business logic (Jest)
- Swagger/OpenAPI documentation (auto-generated)
- Database transactions for payment operations
- Soft deletes for vendors/POs
- Audit trail (created_by, updated_by fields)

Intricate Logic & Optimization Requirements

Critical Business Logic (MUST IMPLEMENT)

1. **PO Status Auto-Update**
    - When payment is recorded → recalculate total payments → update PO status
    - Status transitions must follow business rules
    - Must handle multiple partial payments correctly
2. **Outstanding Calculation**
    - Outstanding = PO Total Amount - SUM(all payments for that PO)
    - Must be accurate across the system
3. **Data Integrity**
    - Use database transactions for payment recording (Nice-to-have: explicit transaction blocks)
    - Ensure PO status updates correctly with payment creation
    - Proper foreign key relationships

Performance & Database Requirements

1. **Database Schema Design (CRITICAL)**
    - Proper foreign key relationships
    - Normalized schema (3NF minimum)
    - Use migrations for schema versioning
    - Include seed script with sample data
2. **Query Optimization**
    - Use joins efficiently for vendor + PO + payment queries
    - Avoid N+1 queries where possible
    - Optimize analytics queries with proper SQL aggregations
3. **API Best Practices**
    - Use DTOs for validation and transformation
    - Implement proper error handling and status codes
    - Return meaningful error messages
    - Clean request/response structure
4. **Code Quality (CRITICAL)**
    - Use NestJS modules, services, controllers pattern
    - Use class-validator for DTO validation
    - Proper TypeScript typing (avoid `any` types)
    - No code duplication

Technical Requirements

Validation (MUST HAVE)

- All DTOs must use class-validator decorators
- Validate:
    - Required fields
    - Email format
    - Positive numbers for amounts
    - Valid enum values for status/payment method
    - Date formats

Error Handling (MUST HAVE)

- Use NestJS exception filters
- Return proper HTTP status codes:
    - 200: Success
    - 201: Created
    - 400: Validation error
    - 404: Not found
    - 409: Conflict (e.g., duplicate vendor)
    - 500: Server error
- Meaningful error messages in response

Database (MUST HAVE)

- Use TypeORM or Prisma (your choice)
- Include migration files
- Seed script with at least: 5 vendors, 15 POs, 10 payments
- Proper relationships and foreign keys

Evaluation Criteria

Database Design (30%)

- Proper schema normalization
- Correct relationships and foreign keys
- Migration files included
- Seed data provided
- **This is heavily weighted - get this right!**

Business Logic (25%)

- PO status auto-update works correctly
- Outstanding calculations are accurate
- All core business rules enforced
- Edge cases handled

API Design (20%)

- RESTful endpoint design
- Proper HTTP methods and status codes
- Clean request/response structure
- DTO validation working
- Meaningful error responses

Code Quality (20%)

- Clean NestJS architecture (modules, services, controllers)
- Proper TypeScript usage
- Readable and maintainable
- Follows NestJS best practices

Bonus Features (5%)

- Any nice-to-have features implemented
- Goes beyond minimum requirements
- Shows initiative and depth

Submission Guidelines

What to Submit

1. **GitHub Repository**

- Clean commit history
- Well-documented README.md
- `.env.example` file (don't commit actual .env)
- Migration files included

2. **Live Deployment URL**
    - API deployed and accessible
    - Include API base URL
    - Database hosted (Railway, Render, Supabase, etc.)
3. **API Documentation**
    - Postman collection (JSON export) OR
    - Simple README with endpoint examples OR
    - Swagger docs (if implemented)
4. **README Must Include:**

```
## Setup Instructions
- Prerequisites
- Installation steps
- Database setup and migrations
- Running the application
- Running seed script

## Database Schema
- Brief description of tables and relationships
- (ER diagram is nice-to-have)

## Implemented Features
- List of MUST-HAVE features completed
- List of NICE-TO-HAVE features completed (if any)

## Key Design Decisions
- Why you chose specific approaches
- How you handled business logic
- Any trade-offs you made

## API Endpoints
- List all endpoints with brief description

## Testing the API
- Sample requests for key flows
- How to test the main scenarios

## Time Breakdown
- Database design: X hours
- API development: X hours
- Testing & debugging: X hours
- Total: X hours
```

Submission Include:

- GitHub repo link
- Live API URL
- Postman collection or endpoint documentation
- Database credentials (for testing)
- Brief note on what you completed (3-4 sentences)

Testing Scenarios We'll Check

Core Flows (Must Work):

1. Create vendor → Create PO → Make partial payment → Verify PO status changes to "Partially Paid"
2. Make another payment completing the PO → Verify status changes to "Fully Paid"
3. Try to make payment exceeding outstanding amount → Should fail with proper error
4. Create PO for inactive vendor → Should fail
5. Query analytics endpoint → Should return correct calculations

Nice-to-Have (If Implemented): 6. Void payment → Verify PO status recalculates correctly 7. Test with JWT authentication → Unauthorized access blocked

What We're NOT Looking For

- Perfect UI or frontend (this is backend-only)
- Complex multi-role authentication system
- File uploads or email notifications
- Microservices architecture
- GraphQL or WebSockets
- Docker (nice to have, not required)
- 100% test coverage

What We ARE Looking For

- **Clean, working API** that solves the core business problem
- **Solid database design** with proper relationships
- **Correct business logic** for PO status updates
- **Good NestJS architecture** (modules, services, controllers)
- **Proper validation and error handling**
- **Clear documentation** so we can test easily
- **Quality over quantity** - focus on doing the core features well

Questions?

If you have clarifying questions about requirements, email us within the first 12 hours. We'll respond within 4 hours during business hours.

Good luck! We're excited to see what you build.