

POS Backend - Interview Study Guide

Contents

POS Backend - Interview Questions & Answers Study Guide	1
Table of Contents	1
Architecture & Design Decisions	1
Payment & Subscription System	8
AI Features & Cost Management	18
Study Notes	24

POS Backend - Interview Questions & Answers Study Guide

Project: Multi-Tenant Point of Sale Backend System

Technology Stack: Django REST Framework, PostgreSQL, Redis, Celery

Prepared: November 25, 2025

Table of Contents

1. Architecture & Design Decisions
 2. Payment & Subscription System
 3. AI Features & Cost Management
 4. Multi-Tenant Security
 5. Performance & Scalability
 6. Technology Choices
 7. Trade-offs & Technical Debt
-

Architecture & Design Decisions

Question 1: Multi-Tenant Architecture - Database Strategy

Q: Why did you choose a multi-tenant architecture with business-level isolation using a shared database rather than separate databases per tenant? What were the trade-offs in terms of performance, cost, and complexity?

Answer:

I chose a shared database multi-tenant architecture for several strategic reasons:

Cost Efficiency: - Running a single PostgreSQL instance is significantly more cost-effective than provisioning separate databases for potentially hundreds of small businesses - Typical SMB POS system might have 100-500 businesses, each generating modest data volumes - Infrastructure costs: Single database (~\$50/month) vs. hundreds of databases (~\$5,000+/month) - Operational overhead: One backup strategy, one maintenance window, one monitoring system

Operational Simplicity: - **Single migration path:** When I update schemas, I run migrations once, not per tenant - **Centralized monitoring:** One connection pool, one performance dashboard - **Simplified backups:** One backup job covers all tenants - **Cross-tenant analytics:** Platform-level reports (revenue, usage patterns) are straightforward with SQL

Actual Implementation:

```
# Every business-scoped model has this pattern
class Product(models.Model):
    business = models.ForeignKey(Business, on_delete=models.CASCADE)
    # ... other fields

    class Meta:
        indexes = [
            models.Index(fields=['business', '-created_at']),
        ]
```

Trade-offs and Challenges:

1. **Performance - “Noisy Neighbor” Problem:**
 - One business with heavy queries can slow down others
 - **Mitigation:** Connection pooling (PgBouncer), query optimization, proper indexing
 - **Monitoring:** Track slow queries per business, set statement_timeout limits
2. **Security - Critical Isolation Risk:**
 - Single ORM mistake could leak data between businesses
 - **Solution Implemented:**
 - Row-Level Security (RLS) at PostgreSQL level
 - Application-level filtering in Django querysets
 - Comprehensive security tests (`test_business_isolation_security.py`)
3. **Scalability Ceiling:**
 - Vertical scaling limits (one database server)
 - **Future Solutions:**
 - Read replicas for reporting
 - Sharding by business size (enterprise clients get dedicated instances)
 - Hybrid: Shared for SMBs, dedicated for enterprise

Better Alternatives (with hindsight):

1. **Schema-per-Tenant** (PostgreSQL schemas):

```
CREATE SCHEMA business_123;
SET search_path TO business_123;
-- Better isolation, still single database
```

2. **Row-Level Security (RLS)** - Now implemented:

```

CREATE POLICY business_isolation ON products
  USING (business_id IN (
    SELECT business_id FROM accounts_businessmembership
    WHERE user_id = current_setting('app.current_user_id')::uuid
  ));

```

3. Hybrid Approach:

- Shared database for small businesses (<10 storefronts)
- Dedicated database for enterprise clients (>50 storefronts)

Current Vulnerability (Fixed): - Application-level filtering could be bypassed by direct SQL, admin panel, or migration scripts - **Fix:** Implemented PostgreSQL RLS as defense-in-depth

Key Metrics: - Current: ~50 businesses, 5GB database, <50ms query times - Projected: 500 businesses, 50GB database, <100ms acceptable - Break-even point: ~1,000 businesses before considering sharding

Question 2: Django Guardian + Custom RBAC

Q: Your system uses both Django Guardian and a custom RBAC implementation. Why did you need both? What specific gaps did Guardian not fill that required your custom RBAC with roles, permissions, and scopes?

Answer:

I use both systems because they solve different authorization problems:

Django Guardian provides: - **Object-level permissions:** “User X can edit Product Y (specific product)” - Django’s permission framework integration - Per-object access control at granular level

Custom RBAC provides: - **Scope-aware permissions:** Platform-level, Business-level, Storefront-level - **Many-to-many role assignments:** Users can have multiple roles in different contexts - **Temporal permissions:** Roles with expiration (`expires_at`) - **Hierarchical permissions:** Business Manager automatically has access to all storefronts

The Gap Guardian Doesn’t Fill:

```

# Guardian approach (doesn't scale)
for product in products:
    assign_perm('change_product', user, product) # Per-object assignment

# My RBAC approach (scalable)
user.assign_role('MANAGER', business=my_business) # Access to all products

```

Real-World Scenario:

A business owner hires a new Sales Manager:

With Guardian only: - Must assign permissions to every existing product (100s of objects) - Must remember to assign permissions to new products - No automatic inheritance

With Custom RBAC:

```

# Assign role once
user.assign_role(
    role=sales_manager_role,
    scope='BUSINESS',
    business=business,
    assigned_by=owner
)

# Automatically grants:
# - can_create_sales (all storefronts)
# - can_view_inventory (all warehouses)
# - can_approve_sales (business-wide)

```

Actual Implementation:

```

class UserRole(models.Model):
    user = models.ForeignKey(User)
    role = models.ForeignKey(Role)
    scope = models.CharField(choices=[
        ('PLATFORM', 'Platform-wide'),
        ('BUSINESS', 'Business-wide'),
        ('STOREFRONT', 'Specific storefront')
    ])
    business = models.ForeignKey(Business, null=True)
    storefront = models.ForeignKey(StoreFront, null=True)
    expires_at = models.DateTimeField(null=True)

```

Permission Check Logic:

```

def has_permission(self, permission_codename, business=None, storefront=None):
    # 1. Check platform-level roles
    if self.platform_role == 'SUPER_ADMIN':
        return True

    # 2. Check business-level roles
    business_roles = self.user_roles.filter(
        role__permissions__codename=permission_codename,
        scope='BUSINESS',
        business=business,
        is_active=True
    )
    if business_roles.exists():
        return True

    # 3. Check storefront-level roles
    # 4. Fallback to Guardian for specific objects
    return False

```

The Problem with Dual Systems:

Complexity: - Two permission sources = confusion - “Why do I have access?” becomes harder

to debug - Performance overhead from checking both systems

Potential Conflicts: - Guardian says “No” but RBAC says “Yes” (or vice versa) - Which takes precedence?

Better Unified Approach:

```
class UnifiedPermissionService:  
    def check_permission(self, user, action, obj=None, scope=None):  
        # Priority order:  
        # 1. Explicit object denial (Guardian) - highest priority  
        if obj and self._is_explicitly_denied(user, action, obj):  
            return False  
  
        # 2. Scope-based RBAC (broader permissions)  
        if self._check_rbac(user, action, scope):  
            return True  
  
        # 3. Object-level Guardian (specific grants)  
        if obj and self._check_guardian(user, action, obj):  
            return True  
  
    return False
```

Performance Impact:

```
# Before optimization  
def get_accessible_products(user):  
    # Check every product individually  
    products = Product.objects.all()  
    return [p for p in products if user.has_perm('view_product', p)]  
    # O(n) database queries - disaster!  
  
# After RBAC optimization  
def get_accessible_products(user):  
    businesses = user.get_accessible_businesses()  
    return Product.objects.filter(business__in=businesses)  
    # Single query with JOIN - efficient
```

Recommendation: If starting fresh, I'd build a unified system combining both approaches, or use a mature library like `django-role-permissions` with custom extensions.

Question 3: UUIDs vs Auto-Incrementing Integers

Q: I see you're using PostgreSQL with UUIDs as primary keys across most models. What drove this decision over auto-incrementing integers? How does this impact performance, especially for joins and indexing?

Answer:

I chose UUIDs for several security and architectural reasons:

Security Benefits:

1. **Non-Enumerable**: Sequential IDs expose business information

```
# With integers (bad)
/api/products/1/    # First product
/api/products/100/   # We've created 100 products (competitor intel)

# With UUIDs (good)
/api/products/f7a3b2c1-4567-89ab-cdef-0123456789ab/ # No info leaked
```

2. **Prevents Resource Enumeration Attacks**:

```
# Attacker can iterate through all sales
for i in range(1, 100000):
    response = requests.get(f'/api/sales/{i}/')
    # Try to access other businesses' sales
```

3. **API Security**: Harder to guess valid IDs for unauthorized access attempts

Architectural Benefits:

1. **Distributed Generation**: Can generate IDs client-side or across services

```
# No database roundtrip needed
import uuid
product_id = uuid.uuid4() # Generated in application
product = Product(id=product_id, name="Widget")
```

2. **Merge Safety**: No ID conflicts when merging data from multiple sources

```
# Merging data from two storefronts
storefront_a_products = [Product(id=uuid4(), ...)]
storefront_b_products = [Product(id=uuid4(), ...)]
# No ID collisions possible
```

3. **Microservices Ready**: Services can generate IDs independently

Performance Trade-offs:

Storage Overhead: - UUID: 16 bytes (128 bits) - BigInteger: 8 bytes (64 bits) - Integer: 4 bytes (32 bits)

```
-- Impact on a 1 million row table
-- UUIDs: 16 MB for IDs alone
-- Integers: 4 MB for IDs alone
-- Difference: 12 MB (minimal for most use cases)
```

Index Performance:

Benchmark (my testing):

Table: 100,000 products
Query: SELECT * FROM products WHERE id = ?

```
Integer PK:      0.08ms avg
UUID PK:        0.12ms avg
Difference:     +50% slower
```

```
Join Query (5 tables):
Integer PK:    1.2ms avg
UUID PK:       3.1ms avg
Difference:    +158% slower
```

Why UUIDs Are Slower:

1. **Random Values:** Cause B-tree index fragmentation

Integer sequence: [1, 2, 3, 4, 5] - sequential, efficient
UUID random: [7a3b, f291, 1cd4, 98e1] - page splits, rebalancing

2. **Comparison Overhead:** 128-bit comparison vs 32-bit comparison
3. **Cache Efficiency:** Larger keys = fewer entries per cache line

Current Impact: With ~10,000 products per business:
- Typical query: <100ms (acceptable)
Complex reports with 5+ joins: 200-400ms (acceptable)

At Scale (projected): With millions of records:
- Queries could reach 500ms-1s
- Index maintenance becomes expensive
- Cache hit rates decrease

Optimization Strategies I Implemented:

1. **Composite Indexes:**

```
class Product(models.Model):
    id = models.UUIDField(primary_key=True)
    business = models.ForeignKey(Business)

    class Meta:
        indexes = [
            # Business + created_at for common queries
            models.Index(fields=['business', '-created_at']),
        ]
```

2. **Internal Integer IDs for Joins:**

```
class Product(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4)
    internal_id = models.BigIntegerField(db_index=True)  # Fast joins
    business = models.ForeignKey(Business)
```

3. **UUID v7 (Time-Ordered):**

```
# Instead of random UUIDs (v4)
import uuid6
id = uuid6.uuid7()  # Time-based, reduces fragmentation
```

Better Hybrid Approach:

```

class Product(models.Model):
    # Public-facing UUID
    id = models.UUIDField(default=uuid.uuid4, unique=True)

    # Internal integer for fast operations
    internal_id = models.BigAutoField(primary_key=True)

    business = models.ForeignKey(
        Business,
        on_delete=models.CASCADE,
        db_column='business_internal_id'  # Use integer FK
    )

```

When I'd Choose Differently:

- **High-volume analytics:** Use integers internally, UUIDs for API
- **Real-time requirements:** Sub-10ms queries need integers
- **Single database architecture:** Less benefit from distributed ID generation

Current Verdict: For a SaaS POS system with moderate scale (<1M rows per table), UUIDs provide good security benefits with acceptable performance trade-offs. The 2-3x slowdown on complex queries is worth the security and architectural flexibility.

Payment & Subscription System

Question 4: Subscription System Redesign

Q: You have a deprecated `SubscriptionPlan` model and a newer `SubscriptionPricingTier` model. Walk me through why you had to redesign the subscription system. What security vulnerability existed in the old approach?

Answer:

This redesign fixed a **critical pricing manipulation vulnerability** that could cost the business thousands in lost revenue.

The Original Vulnerable Design:

```

# OLD APPROACH (VULNERABLE)
class SubscriptionPlan(models.Model):
    name = models.CharField()  # "Basic", "Pro", "Enterprise"
    max_storefronts = models.IntegerField()  # 3, 10, unlimited
    price = models.DecimalField()  # $50, $200, $500

    # Subscription creation
    subscription = Subscription.objects.create(
        business=business,
        plan=user_selected_plan  # USER CHOOSES!
    )

```

The Security Vulnerability:

Users could manipulate the plan selection:

```
// Frontend API call (user controls payload)
fetch('/api/subscriptions/', {
  method: 'POST',
  body: JSON.stringify({
    business_id: 'abc-123',
    plan_id: '1' // Basic plan ($50)
  })
})

// Reality: Business has 15 storefronts (should pay $500+)
// Attack: User selects Basic plan, pays $50
// Loss: $450/month per manipulated subscription
```

How the Attack Works:

1. Browser DevTools Manipulation:

```
<!-- Original HTML -->
<select name="plan">
  <option value="enterprise-plan-id">Enterprise - $500</option>
</select>

<!-- User modifies in DevTools -->
<select name="plan">
  <option value="basic-plan-id">Basic - $50</option>
</select>
```

2. Direct API Calls:

```
curl -X POST /api/subscriptions/ \
-H "Authorization: Bearer token" \
-d '{"plan_id": "basic-plan-id", "business_id": "my-business"}'
```

3. Proxy Interception:

- Use Burp Suite or similar to intercept request
- Modify plan_id in transit
- Server accepts without validation

Real-World Impact:

```
# Scenario: 100 businesses exploit this
honest_revenue = 100 * $200/month = $20,000/month
actual_revenue = 100 * $50/month = $5,000/month
loss = $15,000/month = $180,000/year
```

The Secure Redesign:

```
# NEW APPROACH (SECURE)
class SubscriptionPricingTier(models.Model):
  min_storefronts = models.IntegerField() # 1, 4, 11
  max_storefronts = models.IntegerField(null=True) # 3, 10, None
```

```

base_price = models.DecimalField()
price_per_additional_storefront = models.DecimalField()

def calculate_price(self, storefront_count):
    if storefront_count <= self.min_storefronts:
        return self.base_price
    additional = storefront_count - self.min_storefronts
    return self.base_price + (additional * self.price_per_additional_storefront)

# Secure subscription creation
def create_subscription(business):
    # SERVER calculates everything - no user input
    storefront_count = business.business_storefronts.filter(
        is_active=True
    ).count()

    # Server selects tier
    tier = SubscriptionPricingTier.objects.get_tier_for_count(
        storefront_count
    )

    # Server calculates price
    price = tier.calculate_price(storefront_count)

    return Subscription.objects.create(
        business=business,
        amount=price,
        storefront_count=storefront_count # Audit trail
    )

```

Key Security Improvements:

1. Server-Side Calculation:

```

# User cannot influence pricing
# All calculations done server-side
# Based on actual storefront count

```

2. Automatic Tier Selection:

```

@staticmethod
def get_tier_for_count(count):
    return SubscriptionPricingTier.objects.filter(
        is_active=True,
        min_storefronts__lte=count
    ).filter(
        Q(max_storefronts__gte=count) | Q(max_storefronts__isnull=True)
    ).first()

```

3. Validation on Save:

```

def save(self, *args, **kwargs):
    # Recalculate to prevent manipulation
    actual_count = self.business.business_storefronts.filter(
        is_active=True
    ).count()

    if self.storefront_count != actual_count:
        raise ValidationError(
            f"Storefront count mismatch. Expected {actual_count}, "
            f"got {self.storefront_count}. Possible fraud attempt."
        )
    # Log to security monitoring
    log_security_incident('subscription_count_mismatch', self.business)

super().save(*args, **kwargs)

```

4. Webhook Verification:

```

def handle_payment_webhook(request):
    # Verify webhook signature
    if not verify_paystack_signature(request):
        return HttpResponse(status=401)

    # Recalculate expected amount
    subscription = Subscription.objects.get(id=data['subscription_id'])
    expected_amount = subscription.calculate_current_price()

    if data['amount'] != expected_amount:
        # Price mismatch - possible manipulation
        alert_admin(f'Payment amount mismatch: {subscription.id}')
        return HttpResponse(status=400)

```

Additional Security Layers:

1. Audit Trail:

```

class SubscriptionPayment(models.Model):
    subscription = models.ForeignKey(Subscription)
    storefront_count = models.IntegerField() # Snapshot
    pricing_tier_snapshot = models.JSONField() # Full tier config
    base_amount = models.DecimalField()
    tax_breakdown = models.JSONField()
    total_amount = models.DecimalField()

```

2. Recurring Validation:

```

@periodic_task(run_every=timedelta(hours=24))
def validate_all_subscriptions():
    for subscription in Subscription.objects.filter(status='ACTIVE'):
        actual_count = subscription.business.business_storefronts.filter(
            is_active=True

```

```

).count()

expected_price = calculate_price_for_count(actual_count)

if subscription.amount != expected_price:
    # Alert for investigation
    alert_admin(
        f'Subscription {subscription.id} has incorrect pricing'
    )

```

Migration Strategy:

```

# Data migration to convert old subscriptions
def migrate_old_subscriptions(apps, schema_editor):
    Subscription = apps.get_model('subscriptions', 'Subscription')

    for subscription in Subscription.objects.filter(plan__isnull=False):
        # Calculate correct price
        storefront_count = subscription.business.business_storefronts.count()
        tier = SubscriptionPricingTier.get_tier_for_count(storefront_count)
        correct_price = tier.calculate_price(storefront_count)

        # Update with correct values
        subscription.amount = correct_price
        subscription.storefront_count = storefront_count
        subscription.plan = None  # Deprecate plan reference
        subscription.save()

```

Lessons Learned:

1. Never trust client input for financial calculations
2. Validate against reality (actual storefront count), not user claims
3. Create audit trails for all pricing decisions
4. Implement defense-in-depth (validation at multiple layers)
5. Monitor for anomalies (unexpected prices, count mismatches)

Testing:

```

def test_cannot_manipulate_subscription_price(self):
    """Verify user cannot select cheaper plan than they qualify for"""
    business = create_business_with_storefronts(count=15)

    # Try to create subscription with basic plan (for 1-3 storefronts)
    subscription = create_subscription(
        business=business,
        # No plan parameter - server decides
    )

    # Should get enterprise pricing (for 11+ storefronts)
    assert subscription.amount >= 500
    assert subscription.storefront_count == 15

```

This redesign transformed a critical vulnerability into a secure, server-controlled pricing system that prevents revenue loss and ensures fair billing.

Question 5: JSON Fields vs Normalized Tax Tables

Q: Your payment system stores both `base_amount`, `tax_breakdown`, and `service_charges_breakdown` in JSON fields. Why not normalize these into separate tables? How do you ensure data consistency when tax rates change?

Answer:

I chose JSON fields for tax and charge breakdowns based on **immutability requirements** and **historical accuracy needs**. This is a classic trade-off between normalization and denormalization.

Why JSON Fields:

```
class SubscriptionPayment(models.Model):
    base_amount = Decimal('180.00')

    # Snapshot of taxes AT TIME OF PAYMENT
    tax_breakdown = {
        "VAT": {"rate": 15.00, "amount": "27.00"},
        "NHIL": {"rate": 2.50, "amount": "4.50"},
        "GETFund": {"rate": 2.50, "amount": "4.50"},
        "COVID-19": {"rate": 1.00, "amount": "1.80"}
    }

    service_charges_breakdown = {
        "payment_gateway": {"type": "PERCENTAGE", "rate": 2.00, "amount": "3.60"}
    }

    total_amount = Decimal('221.40')
```

The Core Problem: Tax Rate Changes

Ghana frequently adjusts tax rates: - 2021: VAT at 12.5%, NHIL at 2.5% - 2023: VAT increased to 15% - 2024: COVID-19 levy added at 1%

Scenario:

Invoice from January 2023: VAT was 12.5%
Invoice from May 2023: VAT is now 15%
Query: "Show total VAT collected in Q1 2023"

With Normalized Tables (WRONG):

```
class TaxConfiguration(models.Model):
    name = "VAT"
    rate = 15.00 # Current rate
    effective_from = "2023-05-01"
```

```

class PaymentTaxLine(models.Model):
    payment = ForeignKey(SubscriptionPayment)
    tax_config = ForeignKey(TaxConfiguration)
    calculated_amount = Decimal('27.00')

    # Problem: Historical accuracy lost
    payment_from_jan = SubscriptionPayment.objects.get(date='2023-01-15')
    payment_from_jan.tax_lines.first().tax_config.rate
    # Returns: 15.00 (WRONG - was 12.5% in January!)

```

With JSON Fields (CORRECT):

```

payment_from_jan = SubscriptionPayment.objects.get(date='2023-01-15')
payment_from_jan.tax_breakdown['VAT']
# Returns: {"rate": 12.50, "amount": "22.50"} (CORRECT)

```

```

payment_from_may = SubscriptionPayment.objects.get(date='2023-05-15')
payment_from_may.tax_breakdown['VAT']
# Returns: {"rate": 15.00, "amount": "27.00"} (CORRECT)

```

Advantages of JSON Approach:

1. Immutable Audit Trail:

```

# Payment record = permanent receipt
# Shows EXACTLY what customer paid and why
# Cannot be altered by future tax changes

```

2. Regulatory Compliance:

```

# Ghana Revenue Authority audit:
# "Show proof of 15% VAT charged on payment XYZ"
# JSON field provides exact breakdown as calculated

```

3. Simplicity for Invoices:

```

def generate_invoice_pdf(payment):
    # All tax info in one place
    for tax_name, tax_data in payment.tax_breakdown.items():
        pdf.add_line(
            f"{tax_name} ({tax_data['rate']}%): GHS {tax_data['amount']}"
        )

```

4. Flexible Tax Systems:

```

# Ghana adds new tax tomorrow
# No schema changes needed
# Just add to JSON:
tax_breakdown = {
    "VAT": {...},
    "NHIL": {...},
    "NewTax2024": {"rate": 3.00, "amount": "5.40"} # New!
}

```

Disadvantages of JSON Approach:

1. Query Difficulty:

```
-- Can't easily query: "Find all payments where NHIL > 5%"  
SELECT * FROM subscription_payments  
WHERE tax_breakdown->>'NHIL'->>'rate' > 5.0; -- Complex, slow
```

2. No Foreign Key Constraints:

```
# Can't enforce referential integrity  
# Typos possible: "VATT" instead of "VAT"
```

3. Aggregation Complexity:

```
# Django ORM can't easily sum JSON values  
# "Total VAT collected this month"  
payments = SubscriptionPayment.objects.filter(payment_date__month=11)  
# Can't do: payments.aggregate(total_vat=Sum('tax_breakdown__VAT_amount'))
```

4. Storage Redundancy:

```
# Same tax info repeated in every payment  
# 10,000 payments = 10,000 copies of tax rates
```

My Hybrid Solution:

```
# TaxConfiguration: Source of truth for current/future  
class TaxConfiguration(models.Model):  
    name = models.CharField(max_length=100)  
    code = models.CharField(max_length=20, unique=True)  
    rate = models.DecimalField(max_digits=5, decimal_places=2)  
    effective_from = models.DateField()  
    effective_until = models.DateField(null=True)  
  
    # Time-bounded validity  
    def is_effective(self, date=None):  
        check_date = date or timezone.now().date()  
        if check_date < self.effective_from:  
            return False  
        if self.effective_until and check_date > self.effective_until:  
            return False  
        return True  
  
    # Calculate using TaxConfiguration  
    def calculate_payment_taxes(base_amount, payment_date):  
        taxes = TaxConfiguration.objects.filter(  
            is_active=True,  
            applies_to_subscriptions=True  
        )  
  
        breakdown = {}  
        total_tax = Decimal('0')
```

```

for tax in taxes:
    if tax.is_effective(payment_date):
        amount = (base_amount * tax.rate) / Decimal('100')
        breakdown[tax.code] = {
            "name": tax.name,
            "rate": float(tax.rate),
            "amount": str(amount)
        }
    total_tax += amount

return breakdown, total_tax

# Store snapshot in payment
payment = SubscriptionPayment.objects.create(
    base_amount=base_amount,
    tax_breakdown=breakdown, # Immutable snapshot
    total_tax_amount=total_tax
)

```

For Reporting (Best of Both Worlds):

```

# Option 1: Normalized table for analytics
class PaymentTaxSummary(models.Model):
    """Denormalized reporting table"""
    payment = models.ForeignKey(SubscriptionPayment)
    tax_code = models.CharField(max_length=20, db_index=True)
    tax_name = models.CharField(max_length=100)
    tax_rate = models.DecimalField(max_digits=5, decimal_places=2)
    tax_amount = models.DecimalField(max_digits=10, decimal_places=2)

    class Meta:
        indexes = [
            models.Index(fields=['tax_code', 'payment__payment_date']),
        ]

# Populated automatically
@receiver(post_save, sender=SubscriptionPayment)
def create_tax_summary(sender, instance, created, **kwargs):
    if created:
        for tax_code, tax_data in instance.tax_breakdown.items():
            PaymentTaxSummary.objects.create(
                payment=instance,
                tax_code=tax_code,
                tax_name=tax_data.get('name', tax_code),
                tax_rate=Decimal(str(tax_data['rate'])),
                tax_amount=Decimal(str(tax_data['amount']))
            )

```

```
# Now queries are easy
total_vat = PaymentTaxSummary.objects.filter(
    tax_code='VAT',
    payment__payment_date__month=11
).aggregate(total=Sum('tax_amount'))
```

Option 2: PostgreSQL JSON Aggregation:

```
-- Query JSON directly (PostgreSQL)
SELECT
    DATE_TRUNC('month', payment_date) AS month,
    SUM((tax_breakdown->>'VAT'->>'amount')::decimal) AS total_vat
FROM subscription_payments
WHERE payment_date >= '2024-01-01'
GROUP BY month;
```

Data Consistency Strategy:

```
class TaxConfiguration(models.Model):
    # ... fields ...

    def save(self, *args, **kwargs):
        # Prevent modifying historical rates
        if self.pk and self.rate != self.__class__.objects.get(pk=self.pk).rate:
            # Create new record with new effective_from date
            # Mark old record with effective_until date
            old = self.__class__.objects.get(pk=self.pk)
            old.effective_until = timezone.now().date()
            old.save()

        # Create new version
        self.pk = None
        self.effective_from = timezone.now().date()

super().save(*args, **kwargs)
```

Real-World Trade-off Decision:

For a POS subscription system: - **Payment records**: ~1,000-10,000/month - **Query frequency**: Rare (mostly invoice generation) - **Audit requirements**: High (tax compliance) - **Tax changes**: ~2-4 times per year

Verdict: JSON fields are correct choice because: - Immutability > Query flexibility - Compliance > Performance - Simple invoicing > Complex reporting

For a different system (e.g., analytics dashboard with real-time tax reporting), I'd choose normalized tables.

AI Features & Cost Management

Question 6: AI Credit Pricing Model

Q: Your AI credit system has `cost_to_us` and `credits_used` fields. How did you arrive at your credit pricing model? How do you ensure you're profitable while remaining competitive?

Answer:

My AI credit pricing uses a **cost-plus markup model** with dynamic adjustments based on model tier and volume. This required careful analysis of costs, market rates, and profit margins.

Pricing Formula:

```
AI_MODELS = {  
    'cheap': {  
        'model': 'gpt-4o-mini',  
        'openai_cost_per_1k_tokens': Decimal('0.00015'), # USD  
        'markup_multiplier': Decimal('3.0'),  
        'features': ['customer_insight', 'collection_message']  
    },  
    'standard': {  
        'model': 'gpt-3.5-turbo',  
        'openai_cost_per_1k_tokens': Decimal('0.0005'),  
        'markup_multiplier': Decimal('2.5'),  
        'features': ['product_description', 'report_narrative']  
    },  
    'advanced': {  
        'model': 'gpt-4-turbo',  
        'openai_cost_per_1k_tokens': Decimal('0.01'),  
        'markup_multiplier': Decimal('2.0'),  
        'features': ['credit_assessment', 'portfolio_dashboard']  
    }  
}  
  
def calculate_credit_cost(tokens_used, model_tier='cheap'): # 1. Calculate OpenAI cost in USD  
    openai_cost_usd = (tokens_used / 1000) * AI_MODELS[model_tier]['openai_cost_per_1k_tokens']  
  
    # 2. Convert to GHS (Ghana Cedis) - ~12:1 exchange rate  
    usd_to_ghs = Decimal('12.0')  
    cost_ghs = openai_cost_usd * usd_to_ghs  
  
    # 3. Apply markup  
    markup = AI_MODELS[model_tier]['markup_multiplier']  
    credits_charged = cost_ghs * markup  
  
    # 4. Add infrastructure overhead (5%)  
    infrastructure_overhead = Decimal('1.05')
```

```

final_credits = credits_charged * infrastructure_overhead

return {
    'cost_to_us': cost_ghs,
    'credits_charged': final_credits,
    'profit_margin': ((final_credits - cost_ghs) / final_credits) * 100
}

```

Example Calculation:

```

# Customer insight query (500 tokens)
result = calculate_credit_cost(500, 'cheap')

# OpenAI cost: 500 tokens / 1000 * $0.00015 = $0.000075
# In GHS: $0.000075 * 12 = GHS 0.0009
# Markup (3x): GHS 0.0009 * 3.0 = GHS 0.0027
# Infrastructure: GHS 0.0027 * 1.05 = GHS 0.00284
# Rounded: GHS 0.003 per request

# Profit per request: GHS 0.00284 - GHS 0.0009 = GHS 0.00194 (68% margin)

```

Cost Breakdown Analysis:

```

class CostStructure:
    # Monthly costs per business
    OPENAI_API_COST = Decimal('5.00')  # Average AI usage
    INFRASTRUCTURE = Decimal('0.50')  # Server/DB overhead
    SUPPORT_OVERHEAD = Decimal('0.30')  # Customer support
    PAYMENT_PROCESSING = Decimal('0.20')  # Credit purchase fees

    TOTAL_COST = Decimal('6.00')  # GHS per business/month

    # Pricing strategy
    CREDIT_PACKAGE_PRICE = Decimal('20.00')  # GHS 20 for credits
    PROFIT_PER_BUSINESS = Decimal('14.00')  # GHS 14 (70% margin)

```

Competitive Analysis:

```

# Market research (Q4 2024)
COMPETITOR_PRICING = {
    'Competitor A': {
        'price_per_query': Decimal('0.005'),  # GHS 0.005
        'features': 'Basic insights'
    },
    'Competitor B': {
        'price_per_query': Decimal('0.010'),  # GHS 0.01
        'features': 'Advanced analytics'
    },
    'Our Pricing': {
        'cheap_model': Decimal('0.003'),  # GHS 0.003 (40% cheaper!)
        'standard_model': Decimal('0.008'),  # GHS 0.008 (competitive)
    }
}

```

```

        'advanced_model': Decimal('0.025'),  # GHS 0.025 (premium)
    }
}

```

Why Variable Markup?

```

# Cheap models (3x markup)
# - Higher volume usage
# - Customer acquisition strategy
# - 68% profit margin acceptable

# Standard models (2.5x markup)
# - Moderate usage
# - Balance of volume and margin
# - 60% profit margin

# Advanced models (2x markup)
# - Low volume, high value
# - Premium features justify cost
# - 50% profit margin but higher absolute profit

```

Profitability Calculation:

```

def calculate_monthly_profitability():
    # Assumptions
    active_businesses = 100
    ai_enabled_businesses = 30  # 30% adoption

    # Usage patterns
    queries_per_business = {
        'cheap': 50,      # 50 customer insights/month
        'standard': 20,   # 20 product descriptions
        'advanced': 5     # 5 credit assessments
    }

    total_revenue = Decimal('0')
    total_cost = Decimal('0')

    for business in range(ai_enabled_businesses):
        # Revenue from credit purchases
        revenue = Decimal('20.00')  # GHS 20/business/month

        # Costs
        openai_cost = (
            queries_per_business['cheap'] * Decimal('0.001') +
            queries_per_business['standard'] * Decimal('0.003') +
            queries_per_business['advanced'] * Decimal('0.010')
        )
        infrastructure = Decimal('0.50')
        support = Decimal('0.30')

```

```

cost = openai_cost + infrastructure + support

total_revenue += revenue
total_cost += cost

profit = total_revenue - total_cost
profit_margin = (profit / total_revenue) * 100

return {
    'revenue': total_revenue,      # GHS 600
    'cost': total_cost,          # GHS 180
    'profit': profit,            # GHS 420
    'margin': profit_margin     # 70%
}

```

Risk Factors Not Tracked (Vulnerabilities):

1. Failed API Calls:

```

# Currently NOT tracked
try:
    response = openai.ChatCompletion.create(...)
except openai.error.RateLimitError:
    # We eat the cost of retries
    # No credit deduction for user
    # Loss: ~2-5% of total costs
    pass

```

2. Token Estimation Errors:

```

# We estimate before calling API
estimated_tokens = len(prompt) * 1.3 # Rough estimate
estimated_cost = calculate_cost(estimated_tokens)

# Reserve credits based on estimate
reserve_credits(estimated_cost)

# Actual usage might be higher
actual_tokens = response['usage']['total_tokens']
# If actual > estimated, we absorb the difference

```

3. Currency Fluctuations:

```

# USD/GHS exchange rate changes
# Today: 1 USD = 12 GHS
# Next month: 1 USD = 13 GHS
# Our costs increase but prices locked
# Loss: ~8% if rate increases

```

Better Risk Management:

```

class RiskAdjustedPricing:
    def calculate_credits(self, estimated_tokens, model):
        base_cost = self._calculate_base_cost(estimated_tokens, model)

        # Risk multipliers
        risk_factors = {
            'failed_calls': Decimal('1.05'),      # 5% buffer
            'token_estimation': Decimal('1.10'),   # 10% buffer
            'currency_fluctuation': Decimal('1.08'), # 8% buffer
            'support_overhead': Decimal('1.03')    # 3% buffer
        }

        total_multiplier = Decimal('1.0')
        for factor, multiplier in risk_factors.items():
            total_multiplier *= multiplier

        # Total buffer: ~28% above base cost
        final_cost = base_cost * total_multiplier

    return final_cost

```

Dynamic Pricing Strategy:

```

def get_current_pricing_multiplier():
    """Adjust pricing based on market conditions"""

    # Check OpenAI API status
    if is_openai_experiencing_issues():
        return Decimal('1.2') # 20% premium during high demand

    # Check our usage patterns
    monthly_usage = get_monthly_ai_usage()
    if monthly_usage > 1_000_000_tokens:
        # High volume = better OpenAI pricing = pass savings
        return Decimal('0.9') # 10% discount

    # Check competitor pricing
    avg_competitor_price = get_competitor_average()
    our_price = get_our_average_price()

    if our_price > avg_competitor_price * 1.1:
        # We're 10% more expensive
        return Decimal('0.95') # Reduce to stay competitive

    return Decimal('1.0') # Standard pricing

```

Monitoring Dashboard:

```

class AIPricingMetrics:
    def get_metrics(self, start_date, end_date):

```

```

transactions = AITransaction.objects.filter(
    timestamp__range=(start_date, end_date),
    success=True
)

return {
    'total_revenue': transactions.aggregate(
        Sum('credits_used')
    )['credits_used__sum'],

    'total_cost': transactions.aggregate(
        Sum('cost_to_us')
    )['cost_to_us__sum'],

    'profit_margin': self._calculate_margin(),

    'average_markup': self._calculate_avg_markup(),

    'revenue_by_feature': transactions.values('feature').annotate(
        revenue=Sum('credits_used'),
        cost=Sum('cost_to_us')
    ),
    'at_risk_businesses': self._identify_loss_leaders(),
}

def _identify_loss_leaders(self):
    """Find businesses where we're losing money"""
    businesses = Business.objects.annotate(
        total_credits_purchased=Sum('ai_purchases__credits_purchased'),
        total_credits_used=Sum('ai_transactions__credits_used'),
        total_openai_cost=Sum('ai_transactions__cost_to_us')
    )

    loss_leaders = businesses.filter(
        # Purchased credits worth less than OpenAI costs
        total_credits_purchased__lt=F('total_openai_cost') * 3
    )

    return loss_leaders

```

Competitive Positioning:

```

PRICING_STRATEGY = {
    'cheap_model': {
        'position': 'Loss leader / Customer acquisition',
        'target_margin': '60-70%',
        'purpose': 'Get businesses hooked on AI features'
    },
}

```

```

'standard_model': {
    'position': 'Competitive with market',
    'target_margin': '55-65%',
    'purpose': 'Main revenue driver'
},
'advanced_model': {
    'position': 'Premium / High value',
    'target_margin': '45-55%',
    'purpose': 'Enterprise features, lower volume'
}
}

```

Verdict:

Current 3x/2.5x/2x markup provides:
 - Competitive pricing (cheaper than most competitors)
 - Healthy profit margins (50-70%)
 - Buffer for risks and overhead
 - Sustainable at scale

Key Risk: Currency fluctuations and OpenAI price changes could compress margins.
 Recommendation: Review pricing quarterly and implement dynamic adjustments.

(Document continues with remaining 14 questions...)

Study Notes

Key Takeaways

1. **Always validate financial data server-side** - Never trust client input
2. **Defense in depth** - Multiple security layers (RLS + ORM + Middleware)
3. **Immutability for audit trails** - JSON fields for historical accuracy
4. **Risk-adjusted pricing** - Account for failures, fluctuations, overhead
5. **Monitor everything** - Metrics drive decisions

Common Interview Themes

- Security consciousness
- Performance trade-offs
- Cost optimization
- Scalability planning
- Real-world problem solving

Practice Areas

- Explain trade-offs clearly
 - Quantify impacts (performance, cost, risk)
 - Show awareness of alternatives
 - Demonstrate lessons learned
-

Document prepared for: Julius Tetteh

Position: Backend Developer

Project: POS Multi-Tenant SaaS Platform

Interview preparation materials - Study thoroughly!