MeshHub

Build Specification

Version 1.0 | February 2026

Stack: Python 3.12 · Django 5.x · PostgreSQL 16 · Redis 7 · Celery 5

Scope: Supplier Layer · Developer API / MCP · Payment Layer (TrueLayer)

Classification: Internal Technical Specification · MVP

Complete backend build specification for the MeshHub execution infrastructure layer. Covers project structure, data models, API endpoints, authentication, pricing engine, supplier connectors, payment integration, async jobs, MCP protocol, error contracts, test requirements, Docker setup, and environment variable reference. A developer or Claude coding session should be able to build the MVP backend from this document without referencing any prior conversation.

Version: 1.0 | **Stack:** Python 3.12 · Django 5.x · PostgreSQL 16 · Redis 7 · Celery 5

Classification: Internal Technical Specification

Scope: MVP Backend — Supplier Layer + Developer API/MCP + Payment Layer

TABLE OF CONTENTS

1. PROJECT SETUP & STRUCTURE

Django Application Layout

```
meshhub/
■■■ manage.py
■■■ requirements.txt
■■■ requirements-dev.txt
■■■ .env.example
■■■ docker-compose.yml
■■■ Dockerfile
■
■■■ config/                          # Django project config
■   ■■■ __init__.py
■   ■■■ settings/
■   ■   ■■■ base.py
■   ■   ■■■ local.py
■   ■   ■■■ production.py
■   ■■■ urls.py
■   ■■■ celery.py
■   ■■■ wsgi.py
■
■■■ apps/
■   ■■■ suppliers/                   # Supplier registry, connectors, pricing
```

```
■   ■   ■■■ models.py
■   ■   ■■■ serializers.py
■   ■   ■■■ views.py
■   ■   ■■■ urls.py
■   ■   ■■■ connectors/              # Per-supplier HTTP client wrappers
■   ■   ■   ■■■ base.py             # Abstract connector interface
■   ■   ■   ■■■ rest_connector.py   # Enterprise REST connector
■   ■   ■   ■■■ handon_connector.py # HandOn app virtual connector
■   ■   ■■■ tests/
■   ■
■   ■■■ builders/                   # Developer/app-builder accounts
■   ■   ■■■ models.py
■   ■   ■■■ serializers.py
■   ■   ■■■ views.py
■   ■   ■■■ urls.py
■   ■   ■■■ tests/
■   ■
■   ■■■ execution/                  # Core: receive → validate → execute → respond
■   ■   ■■■ models.py
■   ■   ■■■ serializers.py
■   ■   ■■■ views.py
■   ■   ■■■ urls.py
■   ■   ■■■ engine.py               # Main execution orchestrator
■   ■   ■■■ idempotency.py          # Duplicate detection
■   ■   ■■■ tests/
■   ■
■   ■■■ pricing/                    # Buy/sell price calculation
■   ■   ■■■ models.py
■   ■   ■■■ engine.py               # Price calculation logic
■   ■   ■■■ cache.py                # Redis pricing cache
■   ■   ■■■ tests/
■   ■
■   ■■■ payments/                   # TrueLayer integration + billing
■   ■   ■■■ models.py
■   ■   ■■■ serializers.py
■   ■   ■■■ views.py                # Webhook handler
■   ■   ■■■ urls.py
■   ■   ■■■ truelayer.py            # TrueLayer API client
■   ■   ■■■ settlement.py           # Fund distribution logic
■   ■   ■■■ tests/
■   ■
■   ■■■ contracts/                  # Supplier contracts and KYB
■   ■   ■■■ models.py
■   ■   ■■■ serializers.py
■   ■   ■■■ views.py
■   ■   ■■■ urls.py
■   ■   ■■■ tests/
■   ■
■   ■■■ billing/                    # Ledger, invoicing, payouts
■   ■   ■■■ models.py
■   ■   ■■■ tasks.py                # Celery billing tasks
■   ■   ■■■ tests/
■   ■
■   ■■■ mcp/                        # MCP protocol server
■       ■■■ server.py               # MCP tool definitions
■       ■■■ views.py
```

```
■       ■■■ urls.py
■       ■■■ tests/
■
■■■ common/                          # Shared utilities
■   ■■■ auth.py                      # API key authentication backend
■   ■■■ middleware.py                # Rate limiting, request ID
■   ■■■ pagination.py
■   ■■■ exceptions.py                # Custom exception handlers
■   ■■■ cache.py                     # Redis client wrapper
■   ■■■ circuit_breaker.py           # Supplier call protection
■
■■■ migrations/                      # One migrations/ folder per app (Django standard)
```

Django Apps Registration

```
# config/settings/base.py
INSTALLED_APPS = [
    "django.contrib.contenttypes",
    "django.contrib.auth",
    # Third party
    "rest_framework",
    "drf_spectacular",
    "django_celery_results",
    "django_celery_beat",
    # MeshHub apps
    "apps.suppliers",
    "apps.builders",
    "apps.execution",
    "apps.pricing",
    "apps.payments",
    "apps.contracts",
    "apps.billing",
    "apps.mcp",
]
```

---

2. DEPENDENCIES & ENVIRONMENT

requirements.txt

```
# Core
Django==5.1.*
djangorestframework==3.15.*
psycopg[binary]==3.2.*          # PostgreSQL driver — NOT psycopg2
redis==5.2.*
celery==5.4.*
django-celery-results==2.5.*
django-celery-beat==2.7.*
# API
drf-spectacular==0.27.*          # OpenAPI schema auto-generation
djangorestframework-simplejwt==5.3.*
# Config
django-environ==0.11.*
# HTTP client (outbound supplier calls)
httpx==0.27.*                    # Async-capable, replaces requests
```

```
# Observability
opentelemetry-sdk==1.27.*
opentelemetry-instrumentation-django==0.48b0
opentelemetry-exporter-otlp==1.27.*
# Security
cryptography==43.*               # API key hashing
```

requirements-dev.txt

```
pytest==8.*
pytest-django==4.*
pytest-asyncio==0.24.*
factory-boy==3.*
faker==30.*
coverage==7.*
fakeredis==2.*
responses==0.25.*                # Mock outbound HTTP calls
```

---

3. DATA MODELS (POSTGRESQL)

Critical Implementation Rules

- Use `psycopg3` directly for any query involving `JSONB` operators — do NOT use Django ORM for these
- All primary keys are `UUIDField(default=uuid.uuid4, primary_key=True)`
- All timestamps use `DateTimeField(auto_now_add=True)` or explicit `DateTimeField`
- Never store raw API keys, secrets, or tokens — store SHA-256 hashes only
- Use `db_table` on every model Meta to control exact table name

---

apps/suppliers/models.py

```
import uuid
from django.db import models

class Category(models.Model):
    """Service categories that group suppliers and scope developer access."""
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    slug = models.SlugField(unique=True)           # e.g. "food", "transport"
    name = models.CharField(max_length=100)
    description = models.TextField(blank=True)
    is_active = models.BooleanField(default=True)
    created_at = models.DateTimeField(auto_now_add=True)
    class Meta:
        db_table = "categories"
        verbose_name_plural = "categories"
    def __str__(self):
        return self.slug

class Supplier(models.Model):
    """Any business or individual that provides services through MeshHub."""
    class Status(models.TextChoices):
        PENDING   = "pending",   "Pending"
        ACTIVE    = "active",    "Active"
        SUSPENDED = "suspended", "Suspended"
```

```
        REJECTED  = "rejected",  "Rejected"

    class SupplierType(models.TextChoices):
        ENTERPRISE = "enterprise", "Enterprise (REST API)"
        HANDON     = "handon",     "HandOn App (Individual/SMB)"

    id           = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name         = models.CharField(max_length=255)
    supplier_type = models.CharField(max_length=20, choices=SupplierType.choices)
    status       = models.CharField(max_length=20, choices=Status.choices, default=Status.PENDING)
    category     = models.ForeignKey(Category, on_delete=models.PROTECT, related_name="suppliers")
    # KYB fields
    legal_name        = models.CharField(max_length=255, blank=True)
    registration_number = models.CharField(max_length=100, blank=True)
    country_code      = models.CharField(max_length=2)              # ISO 3166-1 alpha-2
    kyb_verified_at   = models.DateTimeField(null=True, blank=True)
    # Contact
    contact_email     = models.EmailField()
    contact_phone     = models.CharField(max_length=50, blank=True)
    # Settlement bank details (encrypted at rest via field-level encryption)
    bank_account_encrypted = models.TextField(blank=True)           # AES-256 encrypted JSON
    # Metadata
    created_at  = models.DateTimeField(auto_now_add=True)
    updated_at  = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = "suppliers"

    def __str__(self):
        return f"{self.name} ({self.supplier_type})"

class Connector(models.Model):
    """
    Defines HOW MeshHub calls a supplier's API for a specific capability.
    Each supplier-capability pair has one active connector.
    """

    class Protocol(models.TextChoices):
        REST    = "rest",    "REST API"
        WEBHOOK = "webhook", "Webhook (push)"
        HANDON  = "handon",  "HandOn App"
        MCP     = "mcp",     "MCP Server"

    id         = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    supplier   = models.ForeignKey(Supplier, on_delete=models.CASCADE, related_name="connectors")
    capability = models.CharField(max_length=100)    # e.g. "place_order", "get_menu", "track"
    protocol   = models.CharField(max_length=20, choices=Protocol.choices)
    version    = models.CharField(max_length=20, default="1.0")
    is_active  = models.BooleanField(default=True)

    # JSONB: full connector specification
    # Schema varies by protocol — see Connector Spec section below
    spec = models.JSONField()

    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    class Meta:
        db_table = "connectors"
        unique_together = [("supplier", "capability", "version")]

    def __str__(self):
        return f"{self.supplier.name} / {self.capability}"

class PricingSchema(models.Model):
    """
```

```
    Supplier's published pricing for a specific capability.
    buy_price = what MeshHub pays to supplier.
    Developers never see this directly.
    """
    class PricingModel(models.TextChoices):
        PER_CALL        = "per_call",        "Per API Call"
        PER_TRANSACTION = "per_transaction", "Per Transaction"
        FIXED           = "fixed",           "Fixed Price"
        DYNAMIC         = "dynamic",         "Dynamic (supplier API)"
    id         = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    supplier   = models.ForeignKey(Supplier, on_delete=models.CASCADE, related_name="pricing_schemas")
    capability = models.CharField(max_length=100)
    model      = models.CharField(max_length=30, choices=PricingModel.choices)
    currency   = models.CharField(max_length=3, default="EUR")       # ISO 4217
    valid_from = models.DateTimeField()
    valid_until = models.DateTimeField(null=True, blank=True)        # NULL = indefinite
    is_active   = models.BooleanField(default=True)
    # JSONB pricing parameters — see Pricing Schema Spec below
    params = models.JSONField()
    created_at = models.DateTimeField(auto_now_add=True)
    class Meta:
        db_table = "pricing_schemas"
        indexes = [
            models.Index(fields=["supplier", "capability", "is_active"]),
        ]
```

apps/builders/models.py

```
import uuid
import hashlib
import secrets
from django.db import models

class Builder(models.Model):
    """Developer/app-builder accounts that consume the MeshHub API."""
    class Status(models.TextChoices):
        ACTIVE    = "active",    "Active"
        SUSPENDED = "suspended", "Suspended"
        PENDING   = "pending",   "Pending KYC"
    id     = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name   = models.CharField(max_length=255)
    email  = models.EmailField(unique=True)
    status = models.CharField(max_length=20, choices=Status.choices, default=Status.PENDING)
    # Stripe customer ID for billing
    stripe_customer_id = models.CharField(max_length=100, blank=True)
    # KYC
    kyc_verified_at = models.DateTimeField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    class Meta:
        db_table = "builders"
    def __str__(self):
        return self.email

class APIKey(models.Model):
```

```
    """
    Builder API keys. Raw key shown ONCE on creation, never stored.
    Only SHA-256 hash stored in DB.
    """
    id          = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    builder     = models.ForeignKey(Builder, on_delete=models.CASCADE, related_name="api_keys")
    name        = models.CharField(max_length=100)              # human label e.g. "production"
    key_hash    = models.CharField(max_length=64, unique=True)  # SHA-256 hex digest
    key_prefix  = models.CharField(max_length=8)                # First 8 chars for identification
    is_active   = models.BooleanField(default=True)
    last_used_at = models.DateTimeField(null=True, blank=True)
    created_at  = models.DateTimeField(auto_now_add=True)
    expires_at  = models.DateTimeField(null=True, blank=True)   # NULL = no expiry

    class Meta:
        db_table = "api_keys"

    @staticmethod
    def generate():
        """Generate a new API key. Returns (raw_key, hash, prefix)."""
        raw = f"mh_{secrets.token_urlsafe(32)}"
        hashed = hashlib.sha256(raw.encode()).hexdigest()
        return raw, hashed, raw[:8]

    @staticmethod
    def hash_key(raw_key: str) -&gt; str:
        return hashlib.sha256(raw_key.encode()).hexdigest()

class CategoryAccess(models.Model):
    """
    Which categories a builder's API key is permitted to access.
    Enforced at execution time.
    """
    id       = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    api_key  = models.ForeignKey(APIKey, on_delete=models.CASCADE, related_name="category_access")
    category = models.ForeignKey("suppliers.Category", on_delete=models.CASCADE)
    # Optional: restrict to specific suppliers only
    supplier_whitelist = models.JSONField(default=list, blank=True)  # list of supplier UUIDs
    granted_at = models.DateTimeField(auto_now_add=True)

    class Meta:
        db_table = "category_access"
        unique_together = [("api_key", "category")]
```

apps/execution/models.py

```
import uuid
from django.db import models

class Execution(models.Model):
    """
    Immutable log of every API call routed through MeshHub.
    Written once. Never updated — append-only audit trail.
    """
    class Status(models.TextChoices):
        PENDING   = "pending",   "Pending"
        EXECUTING = "executing", "Executing"
        SUCCESS   = "success",   "Success"
        FAILED    = "failed",    "Failed"
        RETRYING  = "retrying",  "Retrying"
```

```
        CANCELLED = "cancelled", "Cancelled"

    id              = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    builder         = models.ForeignKey("builders.Builder", on_delete=models.PROTECT)
    api_key         = models.ForeignKey("builders.APIKey",  on_delete=models.PROTECT)
    supplier        = models.ForeignKey("suppliers.Supplier",  on_delete=models.PROTECT)
    connector       = models.ForeignKey("suppliers.Connector", on_delete=models.PROTECT)
    capability      = models.CharField(max_length=100)
    idempotency_key = models.CharField(max_length=255, unique=True)
    status          = models.CharField(max_length=20, choices=Status.choices, default=Status.PENDING)

    # Pricing snapshot at execution time (immutable record of what was charged)
    buy_price       = models.DecimalField(max_digits=12, decimal_places=4, null=True)
    sell_price      = models.DecimalField(max_digits=12, decimal_places=4, null=True)
    currency        = models.CharField(max_length=3, default="EUR")

    # Payloads (store for debugging and audit)
    request_payload  = models.JSONField(null=True, blank=True)
    response_payload = models.JSONField(null=True, blank=True)
    error_detail     = models.JSONField(null=True, blank=True)

    # Performance
    latency_ms      = models.IntegerField(null=True)
    retry_count     = models.IntegerField(default=0)

    executed_at     = models.DateTimeField(auto_now_add=True)
    completed_at    = models.DateTimeField(null=True, blank=True)

    class Meta:
        db_table = "executions"
        indexes = [
            models.Index(fields=["builder", "executed_at"]),
            models.Index(fields=["supplier", "executed_at"]),
            models.Index(fields=["idempotency_key"]),
            models.Index(fields=["status", "executed_at"]),
        ]
```

apps/payments/models.py

```
import uuid
from django.db import models

class Payment(models.Model):
    """TrueLayer payment initiation record."""

    class Status(models.TextChoices):
        PENDING   = "pending",   "Pending"
        INITIATED = "initiated", "Initiated (redirect sent)"
        CONFIRMED = "confirmed", "Confirmed (bank settled)"
        FAILED    = "failed",    "Failed"
        REFUNDED  = "refunded",  "Refunded"
        EXPIRED   = "expired",   "Expired (user abandoned)"

    id              = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    execution       = models.OneToOneField("execution.Execution", on_delete=models.PROTECT,
                                           related_name="payment")
    truelayer_payment_id = models.CharField(max_length=255, unique=True, null=True, blank=True)
    status          = models.CharField(max_length=20, choices=Status.choices, default=Status.PENDING)

    # Amounts
    gross_amount    = models.DecimalField(max_digits=12, decimal_places=4)  # End user pays
    currency        = models.CharField(max_length=3)

    # Redirect
    redirect_url    = models.URLField(blank=True)  # bank app deeplink returned by TrueLayer
```

```
    return_url      = models.URLField(blank=True)  # where user returns after bank app
    # Timestamps
    initiated_at    = models.DateTimeField(null=True, blank=True)
    confirmed_at    = models.DateTimeField(null=True, blank=True)
    expires_at      = models.DateTimeField(null=True, blank=True)   # 10 min session window
    created_at      = models.DateTimeField(auto_now_add=True)
    class Meta:
        db_table = "payments"
        indexes = [
            models.Index(fields=["truelayer_payment_id"]),
            models.Index(fields=["status", "created_at"]),
        ]
class BillingEvent(models.Model):
    """
    Immutable financial ledger entry. One per confirmed payment.
    Append-only. Never updated.
    """
    id              = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    payment         = models.OneToOneField(Payment, on_delete=models.PROTECT,
                                           related_name="billing_event")
    execution       = models.ForeignKey("execution.Execution", on_delete=models.PROTECT)
    builder         = models.ForeignKey("builders.Builder", on_delete=models.PROTECT)
    supplier        = models.ForeignKey("suppliers.Supplier", on_delete=models.PROTECT)
    # Split amounts (must sum to gross_amount)
    gross_amount    = models.DecimalField(max_digits=12, decimal_places=4)
    supplier_payout = models.DecimalField(max_digits=12, decimal_places=4)  # buy_price
    builder_margin  = models.DecimalField(max_digits=12, decimal_places=4)  # dev margin
    meshhub_fee     = models.DecimalField(max_digits=12, decimal_places=4)  # platform fee
    currency        = models.CharField(max_length=3)
    # Settlement status
    supplier_settled_at = models.DateTimeField(null=True, blank=True)
    builder_settled_at  = models.DateTimeField(null=True, blank=True)
    recorded_at     = models.DateTimeField(auto_now_add=True)
    class Meta:
        db_table = "billing_events"
```

apps/contracts/models.py

```
import uuid
from django.db import models
class Contract(models.Model):
    """Legal agreement between MeshHub and a supplier."""
    class Archetype(models.TextChoices):
        MARKETPLACE   = "marketplace",   "Marketplace"
        BOOKING       = "booking",       "Booking/Reservation"
        DATA_ACCESS   = "data_access",   "Data Access"
        PROFESSIONAL  = "professional",  "Professional Services"
    class Status(models.TextChoices):
        DRAFT    = "draft",    "Draft"
        PENDING  = "pending",  "Pending Signature"
        SIGNED   = "signed",   "Signed"
        EXPIRED  = "expired",  "Expired"
        REVOKED  = "revoked",  "Revoked"
```

```
    id             = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    supplier       = models.ForeignKey("suppliers.Supplier", on_delete=models.PROTECT,
                                       related_name="contracts")
    archetype      = models.CharField(max_length=30, choices=Archetype.choices)
    terms_version  = models.CharField(max_length=20)                # e.g. "v1.2"
    status         = models.CharField(max_length=20, choices=Status.choices, default=Status.DRAFT)
    # Snapshot of full terms at signing (never link to mutable template)
    terms_snapshot = models.JSONField()
    signed_at      = models.DateTimeField(null=True, blank=True)
    expires_at     = models.DateTimeField(null=True, blank=True)
    signed_by_name = models.CharField(max_length=255, blank=True)
    signed_by_email = models.EmailField(blank=True)
    # DocuSign or equivalent envelope ID
    signature_envelope_id = models.CharField(max_length=255, blank=True)
    created_at     = models.DateTimeField(auto_now_add=True)
    class Meta:
        db_table = "contracts"
```

---

4. REDIS KEY MAP

All Redis keys follow: `meshhub:{entity}:{id}:{field}`

Key Pattern	Value	TTL	Operation	Purpose
`meshhub:pricing:{supplier_id}:{capability}`	JSON pricing schema	60s	SET EX / GET	Supplier pricing cache
`meshhub:auth:token:{supplier_id}`	JSON `{token, expires_at}`	token_expiry - 30s	SET EX / GET	Supplier API auth tokens
`meshhub:idempotency:{key}`	execution UUID string	86400s (24h)	SET NX EX	Duplicate execution prevention
`meshhub:ratelimit:builder:{builder_id}:{window}`	integer count	window_size	INCR / EXPIRE	Per-builder rate limiting
`meshhub:ratelimit:supplier:{supplier_id}:{window}`	integer count	window_size	INCR / EXPIRE	Per-supplier rate limiting
`meshhub:apikey:{key_hash}`	JSON `{builder_id, key_id, status, categories}`	300s (5min)	SET EX / GET	API key lookup cache
`meshhub:payment:session:{payment_id}`	JSON payment session state	600s (10min)	SET EX / GET	Active payment redirect session
`meshhub:connector:{supplier_id}:{capability}`	JSON connector spec	300s (5min)	SET EX / GET	Connector spec cache

Redis Client Setup

```
# common/cache.py
import redis
from django.conf import settings
_client = None
def get_redis() -&gt; redis.Redis:
```

```
    global _client
    if _client is None:
        _client = redis.Redis.from_url(
            settings.REDIS_URL,
            decode_responses=True,
            socket_connect_timeout=2,
            socket_timeout=2,
        )
    return _client
```

Idempotency Check Implementation

```
# apps/execution/idempotency.py
from common.cache import get_redis

IDEMPOTENCY_TTL = 86400  # 24 hours

def check_and_set(key: str, execution_id: str) -&gt; str | None:
    """
    Returns existing execution_id if key already seen.
    Sets key and returns None if key is new.
    Atomic: uses SET NX EX.
    """
    r = get_redis()
    redis_key = f"meshhub:idempotency:{key}"
    result = r.set(redis_key, execution_id, nx=True, ex=IDEMPOTENCY_TTL)
    if result is None:
        # Key already existed — return existing execution_id
        return r.get(redis_key)
    return None  # New key — execution should proceed
```

Rate Limit Implementation

```
# common/middleware.py (rate limit logic)
from common.cache import get_redis
import time

def check_rate_limit(entity: str, entity_id: str, limit: int, window: int) -&gt; bool:
    """
    Returns True if rate limit exceeded.
    Uses sliding window approximation (1-minute buckets).
    """
    r = get_redis()
    window_key = int(time.time() / window)
    key = f"meshhub:ratelimit:{entity}:{entity_id}:{window_key}"
    pipe = r.pipeline()
    pipe.incr(key)
    pipe.expire(key, window * 2)
    count, _ = pipe.execute()
    return count &gt; limit
```

---

5. API ENDPOINT SPECIFICATION

Base URL: `/api/v1/`

Authentication: `Authorization: Bearer {api_key}` on all endpoints except webhooks and health

---

5.1 Health

```
GET /api/v1/healthz/
Auth:     None
Response: 200 { "status": "ok", "db": "ok", "redis": "ok", "version": "1.0.0" }
```

5.2 Supplier Endpoints

List Suppliers

```
GET /api/v1/suppliers/
Auth:     API Key (scoped to at least one category)
Query:    ?category=food&amp;country=DE&amp;status=active&amp;page=1&amp;page_size=20
Response: 200
{
  "count": 42,
  "next": "...",
  "results": [
    {
      "id": "uuid",
      "name": "Pizza Roma",
      "supplier_type": "handon",
      "category": { "slug": "food", "name": "Food &amp; Local" },
      "country_code": "DE",
      "capabilities": ["place_order", "get_menu"],
      "status": "active"
    }
  ]
}
Errors:
  401  Invalid or missing API key
  403  Key has no access to requested category
```

Get Supplier Detail

```
GET /api/v1/suppliers/{supplier_id}/
Auth:     API Key (must have access to supplier's category)
Response: 200
{
  "id": "uuid",
  "name": "Pizza Roma",
  "supplier_type": "handon",
  "category": { "slug": "food" },
  "country_code": "DE",
  "capabilities": [
    {
      "name": "place_order",
      "protocol": "handon",
      "version": "1.0"
    },
    {
      "name": "get_menu",
      "protocol": "rest",
      "version": "1.0"
    }
```

```
  ]
}
Errors:
  403  Supplier not in builder's permitted categories
  404  Supplier not found or not active
```

Get Supplier Pricing (sell price only — buy price never exposed)

```
GET /api/v1/suppliers/{supplier_id}/pricing/{capability}/
Auth:     API Key
Response: 200
{
  "supplier_id": "uuid",
  "capability": "place_order",
  "sell_price": "14.28",     // MeshHub sell price (buy + MeshHub margin)
  "currency": "EUR",
  "pricing_model": "per_transaction",
  "surcharges": [
    { "type": "fuel", "amount": "1.00", "description": "Fuel surcharge" },
    { "type": "tax",  "pct": 23,        "description": "VAT" }
  ],
  "upsales": [
    { "id": "extra_bag", "name": "Extra bag", "price": "2.50", "optional": true }
  ],
  "peak_multipliers": {
    "weekend": 1.8,
    "holiday": 2.0
  },
  "cached_at": "2026-02-27T10:00:00Z",
  "ttl_seconds": 60
}
Notes:
  - Response served from Redis cache (60s TTL)
  - Cache miss triggers PostgreSQL fetch and re-cache
  - sell_price includes MeshHub margin but NOT developer margin
Errors:
  404  No active pricing schema for supplier+capability
```

5.3 Execution Endpoints

Execute Capability

```
POST /api/v1/execute/
Auth:     API Key
Headers:  X-Idempotency-Key: {unique_key}   (REQUIRED — reject if missing)
Request:
{
  "supplier_id":  "uuid",
  "capability":   "place_order",
  "payload":      { ... },     // Capability-specific payload
  "upsales":      ["extra_bag"],  // Optional selected upsale IDs
  "metadata":     { "user_id": "..." }  // Builder-defined metadata, stored but not processed
}
Response: 200
{
```

```
  "execution_id":  "uuid",
  "status":        "success",
  "supplier_response": { ... },   // Normalized supplier response
  "execution_time_ms": 342,
  "buy_price":     null,          // NEVER returned
  "sell_price":    "14.28",
  "currency":      "EUR",
  "payment_required": true,
  "payment_initiation_url": "/api/v1/payments/{execution_id}/initiate/"
}
Response (idempotent — key already seen):
{
  "execution_id": "uuid",
  "status": "success",
  "cached": true,
  "supplier_response": { ... }
}
Execution Flow (internal — see engine.py):
  1. Authenticate API key → 401 if invalid
  2. Check idempotency key → return cached response if exists
  3. Check rate limits (builder + supplier) → 429 if exceeded
  4. Validate supplier exists and builder has category access → 403/404
  5. Load connector spec (Redis → PostgreSQL fallback)
  6. Load pricing (Redis → PostgreSQL fallback)
  7. Validate payload against connector spec
  8. Fetch supplier auth token (Redis → Secrets Manager fallback)
  9. Execute supplier API call (httpx, timeout=10s, retries=3)
  10. Write execution record to PostgreSQL
  11. Emit billing_event Celery task (async, non-blocking)
  12. Return normalized response
Errors:
  400  Missing X-Idempotency-Key header
  400  Payload validation failed { "errors": [...] }
  401  Invalid API key
  403  Category access denied
  404  Supplier not found
  422  Connector spec validation failed
  429  Rate limit exceeded { "retry_after": 30 }
  502  Supplier API call failed (after retries)
  503  Supplier circuit breaker open
```

Get Execution Status

```
GET /api/v1/executions/{execution_id}/
Auth:     API Key (must be the builder who created it)
Response: 200
{
  "execution_id": "uuid",
  "status": "success",
  "capability": "place_order",
  "supplier": { "id": "uuid", "name": "Pizza Roma" },
  "sell_price": "14.28",
  "currency": "EUR",
  "executed_at": "2026-02-27T10:00:00Z",
  "completed_at": "2026-02-27T10:00:00.342Z",
  "latency_ms": 342,
```

```
  "payment_status": "confirmed"
}
Errors:
  403  Execution belongs to different builder
  404  Execution not found
```

List Executions

```
GET /api/v1/executions/
Auth:     API Key
Query:    ?status=success&amp;supplier_id=uuid&amp;from=2026-02-01&amp;to=2026-02-28&amp;page=1
Response: 200 { "count": ..., "results": [...] }
```

5.4 Payment Endpoints

Initiate Payment

```
POST /api/v1/payments/{execution_id}/initiate/
Auth:     API Key
Request:
{
  "return_url": "https://yourapp.com/payment/return",
  "end_user_price": "16.21",   // Developer's final price to end user
  "currency": "EUR"
}
Response: 200
{
  "payment_id":    "uuid",
  "redirect_url":  "https://pay.truelayer.com/...",  // Send user here (bank app deeplink)
  "expires_at":    "2026-02-27T10:10:00Z",            // 10 minute window
  "status":        "initiated"
}
Implementation notes:
  - Validate end_user_price &gt;= sell_price (cannot sell below MeshHub cost)
  - Store payment session in Redis (meshhub:payment:session:{payment_id}, TTL 600s)
  - Call TrueLayer Payment Initiation API (see Section 9)
  - Return redirect_url to developer — developer redirects end user to this URL
Errors:
  400  end_user_price below sell_price
  404  Execution not found
  409  Payment already initiated for this execution
  422  Execution not in success status
```

Payment Webhook (TrueLayer → MeshHub)

```
POST /api/v1/webhooks/payment/truelayer/
Auth:     HMAC signature verification (X-TL-Webhook-Timestamp + X-TL-Signature headers)
          Reject if signature invalid — return 401
          Reject if timestamp &gt; 5 minutes old — return 400
Request (from TrueLayer):
{
  "type": "payment_settled",
  "payment_id": "tl_...",
  "amount": 1621,              // In minor units (cents)
  "currency": "EUR",
```

```
  "settled_at": "..."
}
Processing:
  1. Verify HMAC signature
  2. Look up Payment by truelayer_payment_id
  3. Update Payment.status = confirmed, confirmed_at = now
  4. Queue: trigger_order_fulfillment.delay(execution_id)
  5. Queue: record_billing_event.delay(payment_id)
  6. Return 200 immediately (do not block on fulfillment)
Response: 200 { "received": true }
Errors:
  400  Stale timestamp
  401  Invalid signature
  404  Payment not found
  409  Payment already confirmed (idempotent — still return 200)
```

5.5 Builder Management Endpoints

Register Builder

```
POST /api/v1/builders/register/
Auth:     None (public endpoint — used for onboarding)
Request:
{
  "name":  "My App Ltd",
  "email": "dev@myapp.com"
}
Response: 201
{
  "builder_id": "uuid",
  "status": "pending",
  "message": "Account created. KYC verification required before API access."
}
```

Create API Key

```
POST /api/v1/builders/keys/
Auth:     API Key (must be active builder)
Request:
{
  "name": "production",
  "categories": ["food", "transport"],
  "supplier_whitelist": [],    // empty = all suppliers in category
  "expires_at": null           // null = no expiry
}
Response: 201
{
  "key_id":     "uuid",
  "key_prefix": "mh_abc123",
  "raw_key":    "mh_abc123...",   // SHOWN ONCE ONLY — not stored
  "name":       "production",
  "categories": ["food", "transport"],
  "created_at": "..."
}
CRITICAL: raw_key is returned once and never retrievable again.
```

```
          Builder must store it securely immediately.
```

List API Keys

```
GET /api/v1/builders/keys/
Auth:     API Key
Response: 200
{
  "results": [
    {
      "key_id":     "uuid",
      "key_prefix": "mh_abc123",
      "name":       "production",
      "categories": ["food"],
      "is_active":  true,
      "last_used_at": "...",
      "created_at": "..."
      // raw_key NOT included
    }
  ]
}
```

Revoke API Key

```
DELETE /api/v1/builders/keys/{key_id}/
Auth:     API Key
Response: 200 { "revoked": true }
```

5.6 Category Endpoints

```
GET /api/v1/categories/
Auth:     None (public discovery)
Response: 200
{
  "results": [
    {
      "slug": "food",
      "name": "Food &amp;amp; Local",
      "supplier_count": 340,
      "capabilities": ["place_order", "get_menu", "track_delivery"]
    }
  ]
}
```

5.7 Webhook Registration

```
POST /api/v1/webhooks/register/
Auth:     API Key
Request:
{
  "url":    "https://yourapp.com/meshhub/webhook",
  "events": ["execution.completed", "payment.confirmed", "order.fulfilled"],
  "secret": "your_webhook_secret"   // MeshHub will HMAC-sign payloads with this
}
```

```
Response: 201
{
  "webhook_id": "uuid",
  "url": "...",
  "events": [...],
  "created_at": "..."
}
```

---

6. AUTHENTICATION LAYER

API Key Flow

```
# common/auth.py
import hashlib
from rest_framework.authentication import BaseAuthentication
from rest_framework.exceptions import AuthenticationFailed
from apps.builders.models import APIKey, Builder
from common.cache import get_redis
import json

class APIKeyAuthentication(BaseAuthentication):
    """
    Authenticates requests using Bearer API key.
    Cache-first: checks Redis before hitting PostgreSQL.
    """

    def authenticate(self, request):
        auth_header = request.META.get("HTTP_AUTHORIZATION", "")
        if not auth_header.startswith("Bearer "):
            return None  # No credentials provided — let next authenticator try

        raw_key = auth_header[7:]
        if not raw_key.startswith("mh_"):
            raise AuthenticationFailed("Invalid API key format")

        key_hash = hashlib.sha256(raw_key.encode()).hexdigest()

        # 1. Check Redis cache
        cached = self._get_cached_key(key_hash)
        if cached:
            if cached["status"] != "active":
                raise AuthenticationFailed("API key suspended")
            return self._build_auth_tuple(cached)

        # 2. Hit PostgreSQL
        try:
            api_key = (
                APIKey.objects
                .select_related("builder")
                .get(key_hash=key_hash, is_active=True)
            )
        except APIKey.DoesNotExist:
            raise AuthenticationFailed("Invalid API key")

        if api_key.builder.status != Builder.Status.ACTIVE:
            raise AuthenticationFailed("Builder account suspended")

        # 3. Load category access
        categories = list(
            api_key.category_access
            .select_related("category")
```

```
            .values_list("category__slug", flat=True)
        )
        # 4. Cache result (5 min TTL)
        cache_data = {
            "builder_id": str(api_key.builder_id),
            "key_id": str(api_key.id),
            "status": "active",
            "categories": categories,
        }
        self._cache_key(key_hash, cache_data)
        # 5. Update last_used_at (async — don't block request)
        from apps.builders.tasks import update_key_last_used
        update_key_last_used.delay(str(api_key.id))
        return self._build_auth_tuple(cache_data)

    def _get_cached_key(self, key_hash: str) -&gt; dict | None:
        r = get_redis()
        data = r.get(f"meshhub:apikey:{key_hash}")
        return json.loads(data) if data else None

    def _cache_key(self, key_hash: str, data: dict):
        r = get_redis()
        r.set(f"meshhub:apikey:{key_hash}", json.dumps(data), ex=300)

    def _build_auth_tuple(self, data: dict):
        # Returns (user-like object, auth token)
        # DRF requires a 2-tuple from authenticate()
        class AuthContext:
            def __init__(self, d):
                self.builder_id = d["builder_id"]
                self.key_id = d["key_id"]
                self.categories = d["categories"]
                self.is_authenticated = True
        return AuthContext(data), raw_key  # noqa — raw_key in closure

# config/settings/base.py (add to REST_FRAMEWORK)
REST_FRAMEWORK = {
    "DEFAULT_AUTHENTICATION_CLASSES": [
        "common.auth.APIKeyAuthentication",
    ],
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated",
    ],
    "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema",
    "EXCEPTION_HANDLER": "common.exceptions.custom_exception_handler",
}
```

Category Access Check

```
# Used in execution/views.py and suppliers/views.py
def assert_category_access(request, category_slug: str):
    if category_slug not in request.auth.categories:
        from rest_framework.exceptions import PermissionDenied
        raise PermissionDenied(f"API key not authorized for category: {category_slug}")
```

---

7. PRICING ENGINE

Price Calculation Flow

```
buy_price   = base_price + surcharges + peak_delta + tax
sell_price  = buy_price × (1 + meshhub_margin) + meshhub_flat_fee
end_price   = set by developer (must be &gt;= sell_price)
```

Implementation

```
# apps/pricing/engine.py

from decimal import Decimal, ROUND_HALF_UP
from datetime import datetime, timezone
from apps.suppliers.models import PricingSchema
from apps.pricing.cache import get_cached_pricing, set_cached_pricing
from django.conf import settings

MESHHUB_MARGIN_PCT = Decimal(settings.MESHHUB_MARGIN_PCT)   # e.g. "0.08" = 8%
MESHHUB_FLAT_FEE   = Decimal(settings.MESHHUB_FLAT_FEE)     # e.g. "0.20" = €0.20

def calculate_buy_price(supplier_id: str, capability: str, upsale_ids: list = None) -&gt; dict:
    """
    Returns full price breakdown.
    Cache-first. Falls back to PostgreSQL.
    """
    # 1. Try cache
    schema_params = get_cached_pricing(supplier_id, capability)

    # 2. Cache miss — load from DB
    if schema_params is None:
        try:
            schema = PricingSchema.objects.get(
                supplier_id=supplier_id,
                capability=capability,
                is_active=True,
                valid_from__lte=datetime.now(tz=timezone.utc),
            )
            schema_params = schema.params
            set_cached_pricing(supplier_id, capability, schema_params)
        except PricingSchema.DoesNotExist:
            raise ValueError(f"No active pricing for {supplier_id}/{capability}")

    now = datetime.now(tz=timezone.utc)
    base = Decimal(str(schema_params["base_price"]))
    currency = schema_params.get("currency", "EUR")

    # Surcharges
    surcharge_total = Decimal("0")
    surcharges_applied = []
    for s in schema_params.get("surcharges", []):
        if s["type"] == "tax":
            amount = (base * Decimal(str(s["pct"])) / 100).quantize(Decimal("0.0001"))
        else:
            amount = Decimal(str(s.get("amount", "0")))
        surcharge_total += amount
        surcharges_applied.append({"type": s["type"], "amount": str(amount)})

    # Peak multiplier
    peak_mult = Decimal("1.0")
    peak_mults = schema_params.get("peak_multipliers", {})
    if now.weekday() &gt;= 5 and "weekend" in peak_mults:      # Saturday=5, Sunday=6
        peak_mult = Decimal(str(peak_mults["weekend"]))

    base_with_peak = (base * peak_mult).quantize(Decimal("0.0001"), rounding=ROUND_HALF_UP)
```

```
    # Upsales
    upsale_total = Decimal("0")
    upsales_applied = []
    if upsale_ids:
        for u in schema_params.get("upsales", []):
            if u["id"] in upsale_ids:
                price = Decimal(str(u["price"]))
                upsale_total += price
                upsales_applied.append({"id": u["id"], "price": str(price)})
    buy_price = (base_with_peak + surcharge_total + upsale_total).quantize(
        Decimal("0.0001"), rounding=ROUND_HALF_UP
    )
    return {
        "base_price":     str(base),
        "peak_multiplier": str(peak_mult),
        "surcharges":     surcharges_applied,
        "upsales":        upsales_applied,
        "buy_price":      str(buy_price),
        "currency":       currency,
    }
def calculate_sell_price(buy_price_str: str) -&gt; dict:
    """
    Applies MeshHub margin to buy price to produce sell price.
    """
    buy = Decimal(buy_price_str)
    meshhub_fee = (buy * MESHHUB_MARGIN_PCT + MESHHUB_FLAT_FEE).quantize(
        Decimal("0.0001"), rounding=ROUND_HALF_UP
    )
    sell_price = (buy + meshhub_fee).quantize(Decimal("0.0001"), rounding=ROUND_HALF_UP)
    return {
        "buy_price":    str(buy),
        "meshhub_fee":  str(meshhub_fee),
        "sell_price":   str(sell_price),
    }
```

Pricing Cache

```
# apps/pricing/cache.py
import json
from common.cache import get_redis

PRICING_TTL = 60  # seconds

def get_cached_pricing(supplier_id: str, capability: str) -&gt; dict | None:
    r = get_redis()
    key = f"meshhub:pricing:{supplier_id}:{capability}"
    data = r.get(key)
    return json.loads(data) if data else None

def set_cached_pricing(supplier_id: str, capability: str, params: dict):
    r = get_redis()
    key = f"meshhub:pricing:{supplier_id}:{capability}"
    r.set(key, json.dumps(params), ex=PRICING_TTL)

def invalidate_pricing(supplier_id: str, capability: str):
    """Call when supplier updates pricing schema."""
    r = get_redis()
    r.delete(f"meshhub:pricing:{supplier_id}:{capability}")
```

8. SUPPLIER CONNECTOR INTERFACE

Connector Spec Structure (stored in Connector.spec JSONB)

Enterprise REST Connector Spec

```
{
  "protocol": "rest",
  "base_url": "https://api.supplier.com/v1",
  "auth": {
    "type": "oauth2_client_credentials",
    "token_url": "https://api.supplier.com/oauth/token",
    "client_id_secret_key": "suppliers/{supplier_id}/client_id",
    "client_secret_secret_key": "suppliers/{supplier_id}/client_secret",
    "scope": "orders:write"
  },
  "endpoint": "/orders",
  "method": "POST",
  "timeout_seconds": 10,
  "request_schema": {
    "type": "object",
    "required": ["item_id", "quantity"],
    "properties": {
      "item_id":  { "type": "string" },
      "quantity": { "type": "integer" }
    }
  },
  "response_schema": {
    "type": "object",
    "properties": {
      "order_id":  { "type": "string" },
      "status":    { "type": "string" },
      "eta_minutes": { "type": "integer" }
    }
  },
  "retry": {
    "max_attempts": 3,
    "backoff_seconds": [1, 3, 9],
    "retry_on_status": [429, 500, 502, 503, 504]
  }
}
```

HandOn App Connector Spec

```
{
  "protocol": "handon",
  "supplier_device_token_secret_key": "suppliers/{supplier_id}/device_token",
  "fcm_topic": "supplier_{supplier_id}_orders",
  "order_timeout_seconds": 120,
  "capability": "place_order",
  "request_schema": {
    "type": "object",
    "required": ["item_description", "delivery_address"],
    "properties": {
      "item_description":  { "type": "string" },
```

```
      "delivery_address":  { "type": "string" }
    }
  }
}
```

Abstract Connector Base Class

```
# apps/suppliers/connectors/base.py
from abc import ABC, abstractmethod
from dataclasses import dataclass

@dataclass
class ConnectorResult:
    success: bool
    data: dict
    status_code: int | None = None
    error: str | None = None
    latency_ms: int | None = None

class BaseConnector(ABC):
    def __init__(self, connector_spec: dict, supplier_id: str):
        self.spec = connector_spec
        self.supplier_id = supplier_id

    @abstractmethod
    def execute(self, payload: dict) -&gt; ConnectorResult:
        """
        Execute the capability with given payload.
        Returns ConnectorResult. Never raises — catch all exceptions
        and return ConnectorResult(success=False, error=...).
        """
        ...

    @abstractmethod
    def validate_payload(self, payload: dict) -&gt; list[str]:
        """
        Validate payload against connector spec.
        Returns list of error strings (empty = valid).
        """
        ...
```

REST Connector Implementation

```
# apps/suppliers/connectors/rest_connector.py
import time
import httpx
from .base import BaseConnector, ConnectorResult
from apps.suppliers.auth import get_supplier_token

class RESTConnector(BaseConnector):
    def execute(self, payload: dict) -&gt; ConnectorResult:
        start = time.monotonic()
        try:
            token = get_supplier_token(self.supplier_id, self.spec["auth"])
            response = self._call_with_retry(payload, token)
            latency = int((time.monotonic() - start) * 1000)
            return ConnectorResult(
                success=response.is_success,
                data=response.json(),
                status_code=response.status_code,
```

```
                latency_ms=latency,
            )
        except Exception as e:
            latency = int((time.monotonic() - start) * 1000)
            return ConnectorResult(
                success=False,
                data={},
                error=str(e),
                latency_ms=latency,
            )
    def _call_with_retry(self, payload: dict, token: str) -&gt; httpx.Response:
        spec = self.spec
        retry_cfg = spec.get("retry", {"max_attempts": 3, "backoff_seconds": [1, 3, 9]})
        retry_status = set(retry_cfg.get("retry_on_status", [429, 500, 502, 503, 504]))
        last_response = None
        for attempt, backoff in enumerate(retry_cfg["backoff_seconds"]):
            with httpx.Client(timeout=spec.get("timeout_seconds", 10)) as client:
                resp = client.request(
                    method=spec["method"],
                    url=spec["base_url"] + spec["endpoint"],
                    json=payload,
                    headers={"Authorization": f"Bearer {token}"},
                )
            last_response = resp
            if resp.status_code not in retry_status:
                return resp
            if attempt &lt; retry_cfg["max_attempts"] - 1:
                time.sleep(backoff)
        return last_response
    def validate_payload(self, payload: dict) -&gt; list[str]:
        # Basic required-field validation against request_schema
        errors = []
        schema = self.spec.get("request_schema", {})
        for field in schema.get("required", []):
            if field not in payload:
                errors.append(f"Missing required field: {field}")
        return errors
```

HandOn App Connector

```
# apps/suppliers/connectors/handon_connector.py
import time
import uuid
import httpx
from django.conf import settings
from .base import BaseConnector, ConnectorResult
class HandOnConnector(BaseConnector):
    """
    Sends push notification to supplier's mobile app.
    Waits for acceptance webhook (polling Redis or long-poll).
    """
    FCM_URL = "https://fcm.googleapis.com/v1/projects/{project_id}/messages:send"
    def execute(self, payload: dict) -&gt; ConnectorResult:
        start = time.monotonic()
        order_id = str(uuid.uuid4())
```

```
        try:
            # 1. Send push notification to supplier's device
            self._send_push_notification(order_id, payload)
            # 2. Wait for supplier acceptance (via Redis polling)
            accepted = self._wait_for_acceptance(order_id, timeout=self.spec["order_timeout_seconds"])
            latency = int((time.monotonic() - start) * 1000)
            if accepted:
                return ConnectorResult(
                    success=True,
                    data={"order_id": order_id, "status": "accepted"},
                    latency_ms=latency,
                )
            else:
                return ConnectorResult(
                    success=False,
                    data={},
                    error="Supplier did not accept order within timeout",
                    latency_ms=latency,
                )
        except Exception as e:
            return ConnectorResult(
                success=False,
                data={},
                error=str(e),
                latency_ms=int((time.monotonic() - start) * 1000),
            )
    def _send_push_notification(self, order_id: str, payload: dict):
        # FCM push to supplier's device
        # Implementation uses Firebase Admin SDK
        from firebase_admin import messaging
        message = messaging.Message(
            data={"order_id": order_id, "payload": str(payload)},
            topic=self.spec["fcm_topic"],
        )
        messaging.send(message)
    def _wait_for_acceptance(self, order_id: str, timeout: int) -&gt; bool:
        from common.cache import get_redis
        r = get_redis()
        key = f"meshhub:handon:acceptance:{order_id}"
        # Poll Redis every 2 seconds for supplier acceptance
        deadline = time.monotonic() + timeout
        while time.monotonic() &lt; deadline:
            val = r.get(key)
            if val == "accepted":
                return True
            if val == "declined":
                return False
            time.sleep(2)
        return False
    def validate_payload(self, payload: dict) -&gt; list[str]:
        errors = []
        schema = self.spec.get("request_schema", {})
        for field in schema.get("required", []):
            if field not in payload:
                errors.append(f"Missing required field: {field}")
```

```
        return errors
```

HandOn App Acceptance Webhook (supplier's app calls this)

```
POST /api/v1/handon/accept/
Auth:     Supplier token (device token stored in Secrets Manager)
Request:  { "order_id": "uuid", "action": "accepted" | "declined" }
Response: 200 { "received": true }
Implementation: sets Redis key meshhub:handon:acceptance:{order_id} = "accepted" | "declined"
```

---

9. PAYMENT LAYER (TRUELAYER)

TrueLayer API Client

```
# apps/payments/truelayer.py
import httpx
from django.conf import settings
from dataclasses import dataclass

@dataclass
class PaymentInitResult:
    payment_id: str
    redirect_url: str
    expires_at: str
    success: bool
    error: str | None = None

class TrueLayerClient:
    BASE_URL   = "https://pay-api.truelayer.com"
    AUTH_URL   = "https://auth.truelayer.com/connect/token"

    def __init__(self):
        self.client_id     = settings.TRUELAYER_CLIENT_ID
        self.client_secret = settings.TRUELAYER_CLIENT_SECRET

    def _get_access_token(self) -&gt; str:
        from common.cache import get_redis
        import json
        r = get_redis()
        cached = r.get("meshhub:auth:token:truelayer")
        if cached:
            data = json.loads(cached)
            return data["access_token"]
        resp = httpx.post(self.AUTH_URL, data={
            "grant_type":    "client_credentials",
            "client_id":     self.client_id,
            "client_secret": self.client_secret,
            "scope":         "payments",
        })
        resp.raise_for_status()
        token_data = resp.json()
        ttl = token_data["expires_in"] - 30  # Refresh 30s before expiry
        r.set("meshhub:auth:token:truelayer", json.dumps({
            "access_token": token_data["access_token"]
        }), ex=ttl)
        return token_data["access_token"]
```

```
    def initiate_payment(
        self,
        amount_minor: int,       # In minor currency units (cents/pence)
        currency: str,
        reference: str,          # Execution ID — appears on bank statement
        return_url: str,
        idempotency_key: str,
    ) -&gt; PaymentInitResult:
        token = self._get_access_token()
        try:
            resp = httpx.post(
                f"{self.BASE_URL}/v3/payments",
                json={
                    "amount_in_minor": amount_minor,
                    "currency": currency,
                    "payment_method": {
                        "type": "bank_transfer",
                        "provider_selection": { "type": "user_selected" },
                        "beneficiary": {
                            "type":           "merchant_account",
                            "merchant_account_id": settings.TRUELAYER_MERCHANT_ACCOUNT_ID,
                        }
                    },
                    "user": { "name": "MeshHub Customer" },
                    "metadata": { "reference": reference },
                },
                headers={
                    "Authorization":     f"Bearer {token}",
                    "Idempotency-Key":   idempotency_key,
                    "X-TL-Correlation-Id": idempotency_key,
                },
                timeout=15,
            )
            resp.raise_for_status()
            data = resp.json()
            return PaymentInitResult(
                payment_id=data["id"],
                redirect_url=data["resource_token"],   # TrueLayer v3 uses resource token
                expires_at=data.get("expires_at", ""),
                success=True,
            )
        except Exception as e:
            return PaymentInitResult(
                payment_id="",
                redirect_url="",
                expires_at="",
                success=False,
                error=str(e),
            )
    def verify_webhook_signature(self, payload: bytes, signature: str, timestamp: str) -&gt; bool:
        """
        Verify TrueLayer webhook HMAC signature.
        See TrueLayer docs for exact signing algorithm.
        """
        import hmac
        import hashlib
```

```
        import time
        # Reject stale webhooks (&gt; 5 minutes)
        if abs(time.time() - int(timestamp)) &gt; 300:
            return False
        signing_payload = f"{timestamp}.{payload.decode()}"
        expected = hmac.new(
            settings.TRUELAYER_WEBHOOK_SECRET.encode(),
            signing_payload.encode(),
            hashlib.sha256,
        ).hexdigest()
        return hmac.compare_digest(expected, signature)
```

Payment Webhook Handler

```
# apps/payments/views.py
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework.authentication import []  # No auth — verified by HMAC
from rest_framework.permissions import AllowAny
from .models import Payment
from .truelayer import TrueLayerClient
from apps.billing.tasks import record_billing_event
from apps.execution.tasks import trigger_order_fulfillment
import logging
logger = logging.getLogger(__name__)

class TrueLayerWebhookView(APIView):
    authentication_classes = []
    permission_classes = [AllowAny]
    def post(self, request):
        client = TrueLayerClient()
        # 1. Verify signature
        signature = request.META.get("HTTP_X_TL_SIGNATURE", "")
        timestamp = request.META.get("HTTP_X_TL_WEBHOOK_TIMESTAMP", "")
        if not client.verify_webhook_signature(request.body, signature, timestamp):
            return Response({"error": "invalid_signature"}, status=401)
        data = request.data
        event_type = data.get("type")
        if event_type == "payment_settled":
            tl_payment_id = data["payment_id"]
            try:
                payment = Payment.objects.get(truelayer_payment_id=tl_payment_id)
            except Payment.DoesNotExist:
                logger.error(f"Webhook: payment not found {tl_payment_id}")
                return Response({"error": "not_found"}, status=404)
            if payment.status == Payment.Status.CONFIRMED:
                # Idempotent — already processed
                return Response({"received": True})
            # 2. Update payment status
            from django.utils import timezone
            payment.status = Payment.Status.CONFIRMED
            payment.confirmed_at = timezone.now()
            payment.save(update_fields=["status", "confirmed_at"])
            # 3. Queue downstream tasks (non-blocking)
            trigger_order_fulfillment.delay(str(payment.execution_id))
```

```
            record_billing_event.delay(str(payment.id))
        return Response({"received": True})
```

---

10. CELERY ASYNC JOBS

config/celery.py

```
import os
from celery import Celery
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "config.settings.production")
app = Celery("meshhub")
app.config_from_object("django.conf:settings", namespace="CELERY")
app.autodiscover_tasks()
```

config/settings/base.py (Celery config)

```
CELERY_BROKER_URL         = env("REDIS_URL")
CELERY_RESULT_BACKEND     = "django-db"          # django-celery-results
CELERY_TASK_SERIALIZER    = "json"
CELERY_RESULT_SERIALIZER  = "json"
CELERY_ACCEPT_CONTENT     = ["json"]
CELERY_TASK_TRACK_STARTED = True
CELERY_TASK_TIME_LIMIT    = 300                  # 5 min hard limit per task
CELERY_TASK_SOFT_TIME_LIMIT = 240               # 4 min soft limit
CELERY_ACKS_LATE          = True                 # Re-queue on worker crash
CELERY_TASK_REJECT_ON_WORKER_LOST = True
```

Task Definitions

apps/billing/tasks.py

```
from celery import shared_task
import logging
logger = logging.getLogger(__name__)
@shared_task(
    bind=True,
    max_retries=5,
    default_retry_delay=30,
    name="billing.record_billing_event",
)
def record_billing_event(self, payment_id: str):
    """
    Creates immutable BillingEvent record after payment confirmed.
    Triggered by: TrueLayer payment_settled webhook.
    Retry: up to 5 times with 30s delay.
    """
    from apps.payments.models import Payment
    from apps.billing.models import BillingEvent  # add this model if needed
    from apps.pricing.engine import calculate_sell_price
    from decimal import Decimal
    try:
        payment = Payment.objects.select_related(
```

```
            "execution", "execution__builder", "execution__supplier"
        ).get(id=payment_id)
        if hasattr(payment, "billing_event"):
            logger.info(f"BillingEvent already exists for payment {payment_id}")
            return
        execution = payment.execution
        buy_price  = execution.buy_price
        sell_price = execution.sell_price
        gross      = payment.gross_amount
        meshhub_fee    = sell_price - buy_price
        builder_margin = gross - sell_price
        from apps.payments.models import BillingEvent
        BillingEvent.objects.create(
            payment=payment,
            execution=execution,
            builder=execution.builder,
            supplier=execution.supplier,
            gross_amount=gross,
            supplier_payout=buy_price,
            builder_margin=builder_margin,
            meshhub_fee=meshhub_fee,
            currency=payment.currency,
        )
        logger.info(f"BillingEvent created for payment {payment_id}")
    except Exception as exc:
        logger.error(f"record_billing_event failed: {exc}")
        raise self.retry(exc=exc)

@shared_task(
    bind=True,
    max_retries=3,
    default_retry_delay=60,
    name="billing.settle_supplier_payout",
)
def settle_supplier_payout(self, billing_event_id: str):
    """
    Initiates bank transfer to supplier for their share.
    Phase 2: requires MeshHub EMI license.
    At MVP: generates invoice + marks for manual settlement.
    """
    from apps.payments.models import BillingEvent
    from django.utils import timezone
    try:
        event = BillingEvent.objects.select_related("supplier").get(id=billing_event_id)
        # MVP: mark as pending manual settlement
        # Production: trigger TrueLayer outbound payment to supplier
        event.supplier_settled_at = timezone.now()  # Placeholder for MVP
        event.save(update_fields=["supplier_settled_at"])
        logger.info(f"Supplier payout settled for event {billing_event_id}")
    except Exception as exc:
        raise self.retry(exc=exc)
```

apps/execution/tasks.py

```
from celery import shared_task
import logging
```

```
logger = logging.getLogger(__name__)

@shared_task(
    bind=True,
    max_retries=3,
    default_retry_delay=10,
    name="execution.trigger_order_fulfillment",
)
def trigger_order_fulfillment(self, execution_id: str):
    """
    Signals supplier to begin fulfilling the order after payment confirmed.
    Called after payment_settled webhook.
    """
    from apps.execution.models import Execution
    from apps.suppliers.connectors.rest_connector import RESTConnector
    from apps.suppliers.connectors.handon_connector import HandOnConnector
    from apps.suppliers.models import Connector

    try:
        execution = Execution.objects.select_related(
            "supplier", "connector"
        ).get(id=execution_id)

        connector_spec = execution.connector.spec
        protocol = execution.connector.protocol

        if protocol == "rest":
            connector = RESTConnector(connector_spec, str(execution.supplier_id))
        elif protocol == "handon":
            connector = HandOnConnector(connector_spec, str(execution.supplier_id))
        else:
            raise ValueError(f"Unknown protocol: {protocol}")

        # Send fulfillment signal (separate capability: "confirm_order")
        payload = {
            "order_id": str(execution.id),
            "action": "confirm",
        }
        result = connector.execute(payload)

        if not result.success:
            raise ValueError(f"Fulfillment failed: {result.error}")

        logger.info(f"Order {execution_id} fulfillment triggered")

    except Exception as exc:
        logger.error(f"trigger_order_fulfillment failed: {exc}")
        raise self.retry(exc=exc)

@shared_task(name="execution.send_webhook")
def send_webhook(builder_id: str, event_type: str, payload: dict):
    """
    Delivers event webhook to builder's registered endpoint.
    Signs payload with HMAC using builder's webhook secret.
    """
    import hmac
    import hashlib
    import httpx
    import json
    from apps.builders.models import Builder

    try:
        # Load builder's webhook registrations
        # (WebhookRegistration model not shown — add to builders app)
        from apps.builders.models import WebhookRegistration
```

```
        registrations = WebhookRegistration.objects.filter(
            builder_id=builder_id,
            events__contains=[event_type],
            is_active=True,
        )
        body = json.dumps({"event": event_type, "data": payload})
        for reg in registrations:
            sig = hmac.new(
                reg.secret.encode(),
                body.encode(),
                hashlib.sha256,
            ).hexdigest()
            httpx.post(
                reg.url,
                content=body,
                headers={
                    "Content-Type": "application/json",
                    "X-MeshHub-Signature": sig,
                    "X-MeshHub-Event": event_type,
                },
                timeout=10,
            )
    except Exception as exc:
        logger.error(f"send_webhook failed: {exc}")
        # No retry — webhook delivery is best-effort
```

apps/builders/tasks.py

```
from celery import shared_task

@shared_task(name="builders.update_key_last_used")
def update_key_last_used(key_id: str):
    """
    Updates APIKey.last_used_at timestamp asynchronously.
    Fire-and-forget — no retry needed.
    """
    from apps.builders.models import APIKey
    from django.utils import timezone
    APIKey.objects.filter(id=key_id).update(last_used_at=timezone.now())
```

---

11. MCP PROTOCOL LAYER

The MCP (Model Context Protocol) server allows LLMs (Claude, GPT, Gemini) to call MeshHub capabilities as native tools.

Tool Definitions

MeshHub exposes 5 MCP tools:

Tool	Description
`search_suppliers`	Find suppliers by category, country, capability
`get_pricing`	Get sell price for a supplier+capability
`create_order`	Execute a capability (calls `/api/v1/execute/`)

`initiate_payment`	Get payment redirect URL
`check_order_status`	Get execution status

apps/mcp/server.py

```
"""
MeshHub MCP Server.
Exposes MeshHub API as MCP-compatible tools.
Protocol: Model Context Protocol (JSON-RPC 2.0 over HTTP SSE or STDIO)
"""

from dataclasses import dataclass
from typing import Any
import json

MCP_TOOLS = [
    {
        "name": "search_suppliers",
        "description": "Search for service suppliers in a category. Returns available suppliers with capabilities.",
        "inputSchema": {
            "type": "object",
            "required": ["category"],
            "properties": {
                "category":   { "type": "string", "description": "Category slug e.g. food, transport" },
                "country":    { "type": "string", "description": "ISO 3166-1 alpha-2 country code" },
                "capability": { "type": "string", "description": "Filter by capability e.g. place_order" },
            }
        }
    },
    {
        "name": "get_pricing",
        "description": "Get the sell price for a supplier capability before ordering.",
        "inputSchema": {
            "type": "object",
            "required": ["supplier_id", "capability"],
            "properties": {
                "supplier_id": { "type": "string" },
                "capability":  { "type": "string" },
                "upsales":     { "type": "array", "items": { "type": "string" } }
            }
        }
    },
    {
        "name": "create_order",
        "description": "Execute a supplier capability. Creates an order or booking.",
        "inputSchema": {
            "type": "object",
            "required": ["supplier_id", "capability", "payload", "idempotency_key"],
            "properties": {
                "supplier_id":     { "type": "string" },
                "capability":      { "type": "string" },
                "payload":         { "type": "object" },
                "idempotency_key": { "type": "string" },
                "upsales":         { "type": "array", "items": { "type": "string" } }
            }
        }
    },
```

```
    {
        "name": "initiate_payment",
        "description": "Get a payment redirect URL for a completed order. User must visit URL to confirm payment via bank a
        "inputSchema": {
            "type": "object",
            "required": ["execution_id", "end_user_price", "currency", "return_url"],
            "properties": {
                "execution_id":  { "type": "string" },
                "end_user_price":{ "type": "string" },
                "currency":      { "type": "string" },
                "return_url":    { "type": "string" }
            }
        }
    },
    {
        "name": "check_order_status",
        "description": "Check the current status of an order by execution ID.",
        "inputSchema": {
            "type": "object",
            "required": ["execution_id"],
            "properties": {
                "execution_id": { "type": "string" }
            }
        }
    }
]
```

apps/mcp/views.py

```
import json
import httpx
from rest_framework.views import APIView
from rest_framework.response import Response
from common.auth import APIKeyAuthentication
from rest_framework.permissions import IsAuthenticated
from .server import MCP_TOOLS

class MCPToolsListView(APIView):
    """Returns available MCP tools."""
    authentication_classes = [APIKeyAuthentication]
    permission_classes = [IsAuthenticated]

    def get(self, request):
        return Response({"tools": MCP_TOOLS})

class MCPExecuteView(APIView):
    """
    Executes an MCP tool call.
    Internally routes to the appropriate REST API endpoint.
    """
    authentication_classes = [APIKeyAuthentication]
    permission_classes = [IsAuthenticated]

    def post(self, request):
        tool_name = request.data.get("name")
        arguments = request.data.get("arguments", {})

        # Route tool calls to internal API
        handler = getattr(self, f"_handle_{tool_name}", None)
        if not handler:
```

```
            return Response({"error": f"Unknown tool: {tool_name}"}, status=400)
        result = handler(request, arguments)
        return Response({"content": [{"type": "text", "text": json.dumps(result)}]})

    def _call_internal(self, request, method, path, data=None, params=None):
        """Call internal REST API preserving authentication."""
        from django.test import RequestFactory
        from django.urls import reverse
        # Internally dispatch to DRF view — avoids network hop
        # Implementation: use Django's test client pattern or direct view dispatch
        # For MVP: use httpx to call localhost (simple, works)
        import httpx
        base = "http://localhost:8000"
        resp = httpx.request(
            method=method,
            url=f"{base}{path}",
            json=data,
            params=params,
            headers={"Authorization": request.META.get("HTTP_AUTHORIZATION", "")},
            timeout=30,
        )
        return resp.json()

    def _handle_search_suppliers(self, request, args):
        params = {k: v for k, v in args.items() if v}
        return self._call_internal(request, "GET", "/api/v1/suppliers/", params=params)

    def _handle_get_pricing(self, request, args):
        path = f"/api/v1/suppliers/{args['supplier_id']}/pricing/{args['capability']}/"
        return self._call_internal(request, "GET", path)

    def _handle_create_order(self, request, args):
        import uuid
        data = {
            "supplier_id": args["supplier_id"],
            "capability":  args["capability"],
            "payload":     args.get("payload", {}),
            "upsales":     args.get("upsales", []),
        }
        # MCP tool calls must include idempotency key
        # Pass through from args or generate from tool call
        idem_key = args.get("idempotency_key", str(uuid.uuid4()))
        return self._call_internal(
            request, "POST", "/api/v1/execute/",
            data=data,
        )

    def _handle_initiate_payment(self, request, args):
        path = f"/api/v1/payments/{args['execution_id']}/initiate/"
        return self._call_internal(request, "POST", path, data={
            "end_user_price": args["end_user_price"],
            "currency":       args["currency"],
            "return_url":     args["return_url"],
        })

    def _handle_check_order_status(self, request, args):
        path = f"/api/v1/executions/{args['execution_id']}/"
        return self._call_internal(request, "GET", path)

# apps/mcp/urls.py
from django.urls import path
from . import views
```

```
urlpatterns = [
    path("tools/",   views.MCPToolsListView.as_view(),  name="mcp-tools"),
    path("execute/", views.MCPExecuteView.as_view(),    name="mcp-execute"),
]
```

---

12. ERROR HANDLING CONTRACTS

Standard Error Response Format

All errors return this JSON structure:

```
{
  "error": {
    "code":    "rate_limit_exceeded",
    "message": "Too many requests. Builder limit: 100/min",
    "details": {},
    "request_id": "req_abc123"
  }
}
```

Error Code Reference

HTTP Status	Error Code	Trigger
400	`missing_idempotency_key`	`X-Idempotency-Key` header absent on POST /execute
400	`invalid_payload`	Request body fails connector schema validation
400	`stale_webhook_timestamp`	Webhook timestamp > 5 minutes old
400	`price_below_minimum`	`end_user_price` < `sell_price`
401	`invalid_api_key`	Key not found or inactive
401	`invalid_webhook_signature`	HMAC verification failed
403	`category_access_denied`	Builder key not scoped to requested category
403	`supplier_not_permitted`	Supplier not in builder's whitelist
404	`supplier_not_found`	Supplier UUID not found or not active
404	`execution_not_found`	Execution UUID not found or belongs to another builder
404	`pricing_not_found`	No active pricing schema for supplier+capability
409	`payment_already_initiated`	Payment already exists for this execution
409	`idempotent_response`	Execution with this key already completed (returns original response)
422	`connector_validation_failed`	Payload fails connector spec validation
429	`rate_limit_exceeded`	Per-builder or per-supplier rate limit hit

502	`supplier_api_failed`	Supplier API returned error after all retries
503	`circuit_breaker_open`	Supplier circuit breaker is open (too many failures)

Custom Exception Handler

```
# common/exceptions.py
import uuid
import logging
from rest_framework.views import exception_handler
from rest_framework.response import Response

logger = logging.getLogger(__name__)

def custom_exception_handler(exc, context):
    request_id = f"req_{uuid.uuid4().hex[:12]}"

    # Let DRF handle it first
    response = exception_handler(exc, context)

    if response is not None:
        response.data = {
            "error": {
                "code":       _get_error_code(exc),
                "message":    _get_message(response.data),
                "details":    _get_details(response.data),
                "request_id": request_id,
            }
        }
        logger.warning(
            f"API error {response.status_code}",
            extra={"request_id": request_id, "error": str(exc)}
        )
        return response

    # Unhandled exception — 500
    logger.error(f"Unhandled exception", exc_info=exc, extra={"request_id": request_id})
    return Response(
        {"error": {
            "code": "internal_error",
            "message": "An unexpected error occurred",
            "details": {},
            "request_id": request_id,
        }},
        status=500
    )

def _get_error_code(exc) -&gt; str:
    mapping = {
        "AuthenticationFailed": "invalid_api_key",
        "PermissionDenied":     "access_denied",
        "NotFound":             "not_found",
        "ValidationError":      "invalid_payload",
        "Throttled":            "rate_limit_exceeded",
    }
    return mapping.get(type(exc).__name__, "error")

def _get_message(data) -&gt; str:
    if isinstance(data, dict):
        return data.get("detail", str(data))
```

```
    if isinstance(data, list):
        return data[0] if data else "Error"
    return str(data)

def _get_details(data) -&gt; dict:
    if isinstance(data, dict) and "detail" not in data:
        return data
    return {}
```

Rate Limiting Middleware

```
# common/middleware.py

from django.http import JsonResponse
from common.cache import get_redis
import time

BUILDER_RATE_LIMIT  = 100   # requests per minute per builder
SUPPLIER_RATE_LIMIT = 50    # requests per minute per supplier

class RateLimitMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Only rate-limit API paths
        if not request.path.startswith("/api/"):
            return self.get_response(request)

        # Auth must have run first (via DRF) — skip if no auth context yet
        # Rate limiting is enforced inside execution/views.py using request.auth
        return self.get_response(request)

def enforce_rate_limit(builder_id: str, supplier_id: str = None):
    """
    Call inside execution view after authentication.
    Returns (allowed: bool, retry_after: int)
    """
    r = get_redis()
    window = 60
    window_key = int(time.time() / window)

    # Builder limit
    b_key = f"meshhub:ratelimit:builder:{builder_id}:{window_key}"
    pipe = r.pipeline()
    pipe.incr(b_key)
    pipe.expire(b_key, window * 2)
    count, _ = pipe.execute()

    if count &gt; BUILDER_RATE_LIMIT:
        return False, window - (int(time.time()) % window)

    # Supplier limit
    if supplier_id:
        s_key = f"meshhub:ratelimit:supplier:{supplier_id}:{window_key}"
        pipe = r.pipeline()
        pipe.incr(s_key)
        pipe.expire(s_key, window * 2)
        s_count, _ = pipe.execute()
        if s_count &gt; SUPPLIER_RATE_LIMIT:
            return False, window - (int(time.time()) % window)

    return True, 0
```

13. TEST REQUIREMENTS

What Must Be Tested

Every item below must have a test. Implementation detail is developer's choice.

Authentication (apps/builders/tests/test_auth.py)

- ■ Valid API key authenticates successfully
- ■ Invalid API key returns 401 with `invalid_api_key` code
- ■ Expired API key returns 401
- ■ Suspended builder account returns 401
- ■ API key cached in Redis after first lookup (second call must not hit DB)
- ■ Cache invalidation on key revocation

Execution Engine (apps/execution/tests/test_engine.py)

- ■ Idempotency: second call with same key returns cached result, not duplicate execution
- ■ Idempotency: cached result matches original execution ID
- ■ Rate limit: 101st request in 60s window returns 429
- ■ Category access: request for unauthorized category returns 403
- ■ Payload validation: missing required field returns 400 with field name
- ■ Supplier API success: execution created with status=success
- ■ Supplier API 500: retried 3 times, execution created with status=failed
- ■ Supplier API timeout: execution created with status=failed, latency recorded
- ■ Circuit breaker: opens after 5 consecutive failures, returns 503
- ■ Circuit breaker: resets after 60 seconds
- ■ buy_price never appears in API response
- ■ Billing Celery task enqueued after successful execution

Pricing Engine (apps/pricing/tests/test_pricing.py)

- ■ Base price calculation: exact decimal match
- ■ Tax surcharge: calculated as percentage of base, not of total
- ■ Weekend peak multiplier: applied on Saturday/Sunday
- ■ Weekend peak multiplier: NOT applied on Monday–Friday
- ■ Upsale addition: price correctly summed
- ■ Sell price: MeshHub margin applied correctly
- ■ Sell price: flat fee added on top of margin
- ■ Cache hit: second call returns cached value without DB query
- ■ Cache miss: falls back to DB, then caches result
- ■ Stale pricing (valid_until in past): raises ValueError

Payment Layer (apps/payments/tests/test_payments.py)

- ■ TrueLayer payment initiation: correct payload sent to TrueLayer API
- ■ Webhook: valid signature accepted
- ■ Webhook: invalid signature returns 401
- ■ Webhook: stale timestamp (>5min) returns 400

- ■ Webhook: payment_settled updates Payment.status to confirmed
- ■ Webhook: duplicate payment_settled is idempotent (200, no duplicate event)
- ■ Webhook: record_billing_event Celery task enqueued
- ■ Webhook: trigger_order_fulfillment Celery task enqueued
- ■ end_user_price below sell_price: returns 400

Billing (apps/billing/tests/test_billing.py)

- ■ BillingEvent amounts: supplier_payout + builder_margin + meshhub_fee == gross_amount
- ■ BillingEvent idempotency: task running twice does not create duplicate event
- ■ record_billing_event retries on DB failure (up to 5 times)

API Endpoints (integration tests)

- ■ GET /api/v1/suppliers/ returns only suppliers in builder's categories
- ■ GET /api/v1/suppliers/{id}/ returns 403 for out-of-scope supplier
- ■ GET /api/v1/suppliers/{id}/pricing/{cap}/ returns sell_price but not buy_price
- ■ POST /api/v1/execute/ without X-Idempotency-Key returns 400
- ■ POST /api/v1/builders/keys/ returns raw_key in response body
- ■ GET /api/v1/builders/keys/ does NOT include raw_key in response
- ■ DELETE /api/v1/builders/keys/{id}/ invalidates Redis cache for that key

MCP Layer (apps/mcp/tests/test_mcp.py)

- ■ GET /api/v1/mcp/tools/ returns all 5 tool definitions
- ■ POST /api/v1/mcp/execute/ with `search_suppliers` routes correctly
- ■ POST /api/v1/mcp/execute/ with unknown tool returns 400

Test Setup

```
# conftest.py (root)
import pytest
import fakeredis
from unittest.mock import patch
@pytest.fixture(autouse=True)
def fake_redis(monkeypatch):
    """Replace Redis with fakeredis for all tests."""
    fake = fakeredis.FakeRedis(decode_responses=True)
    monkeypatch.setattr("common.cache._client", fake)
    return fake
@pytest.fixture
def active_builder(db):
    from apps.builders.models import Builder
    return Builder.objects.create(
        name="Test Builder",
        email="test@builder.com",
        status=Builder.Status.ACTIVE,
    )
@pytest.fixture
def api_key(active_builder):
    from apps.builders.models import APIKey
    raw, hashed, prefix = APIKey.generate()
    key = APIKey.objects.create(
```

```
        builder=active_builder,
        name="test",
        key_hash=hashed,
        key_prefix=prefix,
    )
    return raw, key

@pytest.fixture
def auth_header(api_key):
    raw, _ = api_key
    return {"Authorization": f"Bearer {raw}"}
```

---

14. DOCKER & LOCAL DEV SETUP

docker-compose.yml

```
version: "3.9"

services:
  db:
    image: postgres:16-alpine
    environment:
      POSTGRES_DB:       meshhub
      POSTGRES_USER:     meshhub
      POSTGRES_PASSWORD: meshhub_dev
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U meshhub"]
      interval: 5s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    ports:
      - "6379:6379"
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s

  api:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/app
    ports:
      - "8000:8000"
    env_file:
      - .env
    environment:
      DJANGO_SETTINGS_MODULE: config.settings.local
    depends_on:
      db:
        condition: service_healthy
      redis:
```

```
        condition: service_healthy
  worker:
    build: .
    command: celery -A config worker --loglevel=info --concurrency=4
    volumes:
      - .:/app
    env_file:
      - .env
    environment:
      DJANGO_SETTINGS_MODULE: config.settings.local
    depends_on:
      - db
      - redis
  beat:
    build: .
    command: celery -A config beat --loglevel=info --scheduler django_celery_beat.schedulers:DatabaseScheduler
    volumes:
      - .:/app
    env_file:
      - .env
    environment:
      DJANGO_SETTINGS_MODULE: config.settings.local
    depends_on:
      - db
      - redis
volumes:
  postgres_data:
```

Dockerfile

```
FROM python:3.12-slim
WORKDIR /app
RUN apt-get update &amp;&amp; apt-get install -y \
    libpq-dev \
    gcc \
    &amp;&amp; rm -rf /var/lib/apt/lists/*
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["gunicorn", "config.wsgi:application", "--bind", "0.0.0.0:8000", "--workers", "4"]
```

Makefile (developer shortcuts)

```
.PHONY: up down migrate shell test
up:
■docker-compose up -d
down:
■docker-compose down
migrate:
■docker-compose exec api python manage.py migrate
shell:
■docker-compose exec api python manage.py shell
test:
```

```
■docker-compose exec api pytest apps/ -v --tb=short

createsuperuser:
■docker-compose exec api python manage.py createsuperuser

logs:
■docker-compose logs -f api worker

seed:
■docker-compose exec api python manage.py loaddata fixtures/seed_data.json
```

First-Time Setup

```
# 1. Clone and configure
cp .env.example .env
# Edit .env with your credentials

# 2. Start services
docker-compose up -d

# 3. Run migrations
make migrate

# 4. Create initial data
make seed

# 5. Run tests
make test

# 6. View API docs
open http://localhost:8000/api/schema/swagger-ui/
```

---

15. ENVIRONMENT VARIABLES REFERENCE

.env.example

```
# ■■ Django ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
DJANGO_SECRET_KEY=change-me-in-production-use-50-random-chars
DEBUG=True
ALLOWED_HOSTS=localhost,127.0.0.1

# ■■ Database ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
DATABASE_URL=postgres://meshhub:meshhub_dev@db:5432/meshhub

# ■■ Redis ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
REDIS_URL=redis://redis:6379/0

# ■■ MeshHub Pricing ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
MESHHUB_MARGIN_PCT=0.08       # 8% margin on buy price
MESHHUB_FLAT_FEE=0.20         # €0.20 flat platform fee per transaction

# ■■ TrueLayer ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
TRUELAYER_CLIENT_ID=your_client_id
TRUELAYER_CLIENT_SECRET=your_client_secret
TRUELAYER_MERCHANT_ACCOUNT_ID=your_merchant_account_uuid
TRUELAYER_WEBHOOK_SECRET=your_webhook_signing_secret
TRUELAYER_ENVIRONMENT=sandbox   # sandbox | production

# ■■ AWS (Secrets Manager for supplier credentials) ■■■■■■■■■■■■■■■■■■■■■■
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
AWS_DEFAULT_REGION=eu-west-1

# ■■ Firebase (HandOn push notifications) ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
FIREBASE_PROJECT_ID=your_firebase_project
FIREBASE_CREDENTIALS_PATH=/app/secrets/firebase-credentials.json
```

```
# ■■ Celery ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
# Uses REDIS_URL — no separate config needed for broker
# ■■ Observability ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
OTEL_EXPORTER_OTLP_ENDPOINT=http://localhost:4317
OTEL_SERVICE_NAME=meshhub-api
# ■■ Stripe (billing — Phase 2) ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
STRIPE_SECRET_KEY=sk_test_...
STRIPE_WEBHOOK_SECRET=whsec_...
# ■■ Sentry (error tracking) ■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
SENTRY_DSN=
```

config/settings/base.py (env loading)

```
import environ

env = environ.Env()
environ.Env.read_env()

SECRET_KEY   = env("DJANGO_SECRET_KEY")
DEBUG        = env.bool("DEBUG", default=False)
DATABASES    = {"default": env.db("DATABASE_URL")}
REDIS_URL    = env("REDIS_URL")

MESHHUB_MARGIN_PCT = env("MESHHUB_MARGIN_PCT", default="0.08")
MESHHUB_FLAT_FEE   = env("MESHHUB_FLAT_FEE",   default="0.20")

TRUELAYER_CLIENT_ID          = env("TRUELAYER_CLIENT_ID")
TRUELAYER_CLIENT_SECRET      = env("TRUELAYER_CLIENT_SECRET")
TRUELAYER_MERCHANT_ACCOUNT_ID = env("TRUELAYER_MERCHANT_ACCOUNT_ID")
TRUELAYER_WEBHOOK_SECRET     = env("TRUELAYER_WEBHOOK_SECRET")

AWS_DEFAULT_REGION = env("AWS_DEFAULT_REGION", default="eu-west-1")
```

APPENDIX A: CONNECTOR SPEC — PRICING SCHEMA JSONB EXAMPLES

Fixed-Price Supplier (pizza)

```
{
  "base_price": "8.00",
  "currency": "EUR",
  "surcharges": [
    { "type": "fuel",    "amount": "1.00" },
    { "type": "tax",     "pct": 23 }
  ],
  "peak_multipliers": {
    "weekend": 1.2
  },
  "upsales": [
    { "id": "extra_cheese", "name": "Extra Cheese", "price": "1.50", "optional": true },
    { "id": "drink",        "name": "Can of Coke",  "price": "2.00", "optional": true }
  ],
  "minimum_order": "5.00"
}
```

Dynamic-Price Supplier (taxi)

```
{
  "pricing_model": "dynamic",
```

```
  "price_api_endpoint": "/estimate",
  "price_field_path":   "data.estimated_fare",
  "currency": "EUR",
  "surcharges": [
    { "type": "tax", "pct": 23 }
  ],
  "peak_multipliers": {
    "weekend": 1.5,
    "holiday": 2.0
  },
  "minimum_price": "5.00",
  "price_ttl_seconds": 30
}
```

APPENDIX B: DATABASE MIGRATION ORDER

Run migrations in this order on first deploy:

```
python manage.py migrate contenttypes
python manage.py migrate auth
python manage.py migrate suppliers
python manage.py migrate contracts
python manage.py migrate builders
python manage.py migrate execution
python manage.py migrate pricing
python manage.py migrate payments
python manage.py migrate billing
python manage.py migrate mcp
python manage.py migrate django_celery_results
python manage.py migrate django_celery_beat
```

APPENDIX C: SECURITY CHECKLIST

Before any deployment (staging or production):

- ■ `DEBUG=False` in production settings
- ■ `SECRET_KEY` is 50+ random characters, not the default
- ■ All supplier credentials stored in AWS Secrets Manager — not in `.env`
- ■ API keys stored as SHA-256 hash only — raw key never in DB, logs, or responses (except at creation)
- ■ TrueLayer webhook signature verified on every webhook request
- ■ HandOn acceptance webhook uses supplier device token authentication
- ■ Rate limiting active on all `/api/v1/execute/` calls
- ■ Database connection uses SSL (`?sslmode=require` in DATABASE_URL for production)
- ■ Redis requires AUTH password in production
- ■ ALLOWED_HOSTS set explicitly — no wildcard in production
- ■ Celery broker uses Redis AUTH in production
- ■ buy_price never serialized into any API response
- ■ CORS configured explicitly — not `*`
- ■ HTTPS only — HTTP redirected to HTTPS at load balancer level
- ■ Bank account details encrypted at field level (AES-256) before storage

End of MeshHub Backend Developer Specification v1.0