11.2 Health Check and Monitoring Scripts

bash

```bash
#!/bin/bash
# scripts/health-check.sh

set -e

echo "🔍 Running comprehensive health checks..."

# Check if all services are running
echo "📋 Checking service status..."
services=("api-gateway" "auth-service" "etsy-service" "design-service" "analytics-service" "postgres" "redis" "

for service in "${services[@]}"; do
    if docker-compose ps | grep -q "$service.*Up"; then
        echo "✅ $service is running"
    else
        echo "❌ $service is not running"
        exit 1
    fi
done

# Check service endpoints
echo "🌐 Testing service endpoints..."

# API Gateway health check
if curl -f http://localhost:8080/health &>/dev/null; then
    echo "✅ API Gateway health check passed"
else
    echo "❌ API Gateway health check failed"
    exit 1
fi

# Database connectivity
if docker-compose exec -T postgres pg_isready -U postgres &>/dev/null; then
    echo "✅ Database connectivity check passed"
else
    echo "❌ Database connectivity check failed"
    exit 1
fi

# Redis connectivity
if docker-compose exec -T redis redis-cli ping | grep -q "PONG"; then
    echo "✅ Redis connectivity check passed"
else
```

```
    echo "❌ Redis connectivity check failed"
    exit 1
fi

# MinIO health check
if curl -f http://localhost:9000/minio/health/live &>/dev/null; then
    echo "✅ MinIO health check passed"
else
    echo "❌ MinIO health check failed"
    exit 1
fi

echo "🎉 All health checks passed!"
```

11.3 Automated Backup System

python

```python
# scripts/backup_manager.py
#!/usr/bin/env python3

import os
import sys
import subprocess
import datetime
import boto3
import logging
from pathlib import Path

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class BackupManager:
    def __init__(self, qnap_path: str = "/share/Container/etsy-saas-prod"):
        self.qnap_path = qnap_path
        self.backup_path = "/share/Container/backups"
        self.timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")

    def backup_database(self):
        """Create PostgreSQL database backup"""
        logger.info("🗄️ Starting database backup...")

        backup_file = f"{self.backup_path}/postgres_backup_{self.timestamp}.sql"

        cmd = [
            "docker-compose", "-f", f"{self.qnap_path}/docker-compose.prod.yml",
            "exec", "-T", "postgres",
            "pg_dumpall", "-U", "postgres"
        ]

        try:
            with open(backup_file, 'w') as f:
                subprocess.run(cmd, stdout=f, check=True)

            logger.info(f"✅ Database backup completed: {backup_file}")

            # Compress backup
            subprocess.run(["gzip", backup_file], check=True)
            logger.info(f"✅ Database backup compressed: {backup_file}.gz")

            return f"{backup_file}.gz"
```

```
        except subprocess.CalledProcessError as e:
            logger.error(f"❌ Database backup failed: {e}")
            sys.exit(1)

    def backup_file_storage(self):
        """Backup MinIO file storage"""
        logger.info("📁 Starting file storage backup...")

        backup_file = f"{self.backup_path}/minio_backup_{self.timestamp}.tar.gz"
        minio_data_path = f"{self.qnap_path}/volumes/minio_data"

        try:
            cmd = ["tar", "-czf", backup_file, "-C", minio_data_path, "."]
            subprocess.run(cmd, check=True)

            logger.info(f"✅ File storage backup completed: {backup_file}")
            return backup_file

        except subprocess.CalledProcessError as e:
            logger.error(f"❌ File storage backup failed: {e}")
            sys.exit(1)

    def backup_application_config(self):
        """Backup application configuration and docker-compose files"""
        logger.info("⚙️ Starting configuration backup...")

        backup_file = f"{self.backup_path}/config_backup_{self.timestamp}.tar.gz"

        files_to_backup = [
            "docker-compose.prod.yml",
            "docker-compose.yml",
            ".env.production",
            "traefik",
            "nginx"
        ]

        try:
            cmd = ["tar", "-czf", backup_file, "-C", self.qnap_path]
            cmd.extend(files_to_backup)
            subprocess.run(cmd, check=True)

            logger.info(f"✅ Configuration backup completed: {backup_file}")
            return backup_file
```

```
        except subprocess.CalledProcessError as e:
            logger.error(f"❌ Configuration backup failed: {e}")
            sys.exit(1)

    def cleanup_old_backups(self, retention_days: int = 30):
        """Remove backups older than retention period"""
        logger.info(f"🧹 Cleaning up backups older than {retention_days} days...")

        cutoff_date = datetime.datetime.now() - datetime.timedelta(days=retention_days)

        for backup_file in Path(self.backup_path).glob("*_backup_*"):
            if backup_file.stat().st_mtime < cutoff_date.timestamp():
                backup_file.unlink()
                logger.info(f"🗑️ Removed old backup: {backup_file}")

    def run_full_backup(self):
        """Run complete backup procedure"""
        logger.info("🚀 Starting full backup procedure...")

        # Create backup directory
        os.makedirs(self.backup_path, exist_ok=True)

        # Run all backup procedures
        db_backup = self.backup_database()
        file_backup = self.backup_file_storage()
        config_backup = self.backup_application_config()

        # Cleanup old backups
        self.cleanup_old_backups()

        logger.info("✅ Full backup completed successfully!")

        return {
            "database_backup": db_backup,
            "file_backup": file_backup,
            "config_backup": config_backup,
            "timestamp": self.timestamp
        }

if __name__ == "__main__":
    backup_manager = BackupManager()
    backup_manager.run_full_backup()
```

12. Advanced Monitoring and Observability

12.1 Prometheus Metrics Collection

python

```python
# services/common/metrics.py
from prometheus_client import Counter, Histogram, Gauge, start_http_server
import time
from functools import wraps

# Metrics definitions
REQUEST_COUNT = Counter('http_requests_total',
                        'Total HTTP requests',
                        ['method', 'endpoint', 'status', 'tenant_id'])

REQUEST_DURATION = Histogram('http_request_duration_seconds',
                             'HTTP request duration',
                             ['method', 'endpoint', 'tenant_id'])

ACTIVE_CONNECTIONS = Gauge('active_connections_total',
                           'Active database connections',
                           ['tenant_id'])

ETSY_API_CALLS = Counter('etsy_api_calls_total',
                         'Total Etsy API calls',
                         ['tenant_id', 'endpoint', 'status'])

JOB_PROCESSING_TIME = Histogram('job_processing_seconds',
                                'Job processing time',
                                ['tenant_id', 'job_type'])

TENANT_RESOURCE_USAGE = Gauge('tenant_resource_usage',
                              'Tenant resource usage',
                              ['tenant_id', 'resource_type'])

def track_metrics(func):
    """Decorator to track API endpoint metrics"""
    @wraps(func)
    async def wrapper(request, *args, **kwargs):
        start_time = time.time()
        tenant_id = getattr(request.state, 'tenant_id', 'unknown')

        try:
            response = await func(request, *args, **kwargs)
            status = getattr(response, 'status_code', 200)

            # Record metrics
            REQUEST_COUNT.labels(
```

```
            method=request.method,
            endpoint=request.url.path,
            status=status,
            tenant_id=tenant_id
        ).inc()

        REQUEST_DURATION.labels(
            method=request.method,
            endpoint=request.url.path,
            tenant_id=tenant_id
        ).observe(time.time() - start_time)

        return response

    except Exception as e:
        REQUEST_COUNT.labels(
            method=request.method,
            endpoint=request.url.path,
            status=500,
            tenant_id=tenant_id
        ).inc()
        raise

  return wrapper

class TenantMetrics:
  def __init__(self, tenant_id: str):
    self.tenant_id = tenant_id

  def record_etsy_api_call(self, endpoint: str, status_code: int):
    """Record Etsy API call metrics"""
    ETSY_API_CALLS.labels(
        tenant_id=self.tenant_id,
        endpoint=endpoint,
        status=status_code
    ).inc()

  def record_job_processing_time(self, job_type: str, duration: float):
    """Record job processing time"""
    JOB_PROCESSING_TIME.labels(
        tenant_id=self.tenant_id,
        job_type=job_type
    ).observe(duration)
```

```python
    def update_resource_usage(self, resource_type: str, current_usage: int):
        """Update current resource usage"""
        TENANT_RESOURCE_USAGE.labels(
            tenant_id=self.tenant_id,
            resource_type=resource_type
        ).set(current_usage)

# Start metrics server
def start_metrics_server(port: int = 9090):
    """Start Prometheus metrics server"""
    start_http_server(port)
```

12.2 Comprehensive Logging System

python

```python
# services/common/logging_config.py
import logging
import json
import sys
from datetime import datetime
from typing import Dict, Any

class TenantFormatter(logging.Formatter):
    """Custom formatter that includes tenant context"""

    def format(self, record):
        # Add tenant context if available
        if hasattr(record, 'tenant_id'):
            record.tenant_context = f"[Tenant: {record.tenant_id}]"
        else:
            record.tenant_context = "[System]"

        # Create structured log entry
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "level": record.levelname,
            "service": getattr(record, 'service_name', 'unknown'),
            "tenant_id": getattr(record, 'tenant_id', None),
            "message": record.getMessage(),
            "module": record.module,
            "function": record.funcName,
            "line": record.lineno
        }

        # Add extra fields if present
        if hasattr(record, 'extra_fields'):
            log_entry.update(record.extra_fields)

        # Add exception info if present
        if record.exc_info:
            log_entry["exception"] = self.formatException(record.exc_info)

        return json.dumps(log_entry)

class TenantLogger:
    """Logger with tenant context"""

    def __init__(self, service_name: str, tenant_id: str = None):
```

```
        self.service_name = service_name
        self.tenant_id = tenant_id
        self.logger = logging.getLogger(f"{service_name}.{tenant_id or 'system'}")

        if not self.logger.handlers:
            self._setup_logger()

    def _setup_logger(self):
        """Setup logger with proper formatting"""
        handler = logging.StreamHandler(sys.stdout)
        formatter = TenantFormatter()
        handler.setFormatter(formatter)

        self.logger.addHandler(handler)
        self.logger.setLevel(logging.INFO)

    def _log_with_context(self, level: str, message: str, **kwargs):
        """Log message with tenant context"""
        extra = {
            'service_name': self.service_name,
            'tenant_id': self.tenant_id,
            'extra_fields': kwargs
        }

        getattr(self.logger, level.lower())(message, extra=extra)

    def info(self, message: str, **kwargs):
        self._log_with_context('INFO', message, **kwargs)

    def error(self, message: str, **kwargs):
        self._log_with_context('ERROR', message, **kwargs)

    def warning(self, message: str, **kwargs):
        self._log_with_context('WARNING', message, **kwargs)

    def debug(self, message: str, **kwargs):
        self._log_with_context('DEBUG', message, **kwargs)

# Usage example in services
def create_tenant_logger(service_name: str):
    def get_logger(tenant_id: str = None):
        return TenantLogger(service_name, tenant_id)
    return get_logger
```

```python
# In each service
logger_factory = create_tenant_logger("etsy-service")

@app.middleware("http")
async def logging_middleware(request: Request, call_next):
    tenant_id = getattr(request.state, 'tenant_id', None)
    logger = logger_factory(tenant_id)

    start_time = time.time()

    logger.info("Request started",
                method=request.method,
                path=request.url.path,
                client_ip=request.client.host)

    response = await call_next(request)

    process_time = time.time() - start_time

    logger.info("Request completed",
                method=request.method,
                path=request.url.path,
                status_code=response.status_code,
                process_time=process_time)

    return response
```

13. Security Hardening

13.1 API Security Middleware

python

```python
# services/common/security_middleware.py
from fastapi import Request, HTTPException
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt
import time
import hashlib
from typing import Dict, List
import redis

security = HTTPBearer()
redis_client = redis.Redis.from_url(REDIS_URL)

class SecurityMiddleware:
    def __init__(self):
        self.rate_limits = {
            "api_calls": {"limit": 1000, "window": 3600},  # 1000 calls per hour
            "auth_attempts": {"limit": 5, "window": 900},   # 5 attempts per 15 minutes
        }

    async def rate_limit_check(self, identifier: str, limit_type: str) -> bool:
        """Check if request is within rate limits"""
        config = self.rate_limits.get(limit_type)
        if not config:
            return True

        key = f"rate_limit:{limit_type}:{identifier}"
        current_time = int(time.time())
        window_start = current_time - config["window"]

        # Remove old entries
        redis_client.zremrangebyscore(key, 0, window_start)

        # Count current requests in window
        current_count = redis_client.zcard(key)

        if current_count >= config["limit"]:
            return False

        # Add current request
        redis_client.zadd(key, {str(current_time): current_time})
        redis_client.expire(key, config["window"])

        return True
```

```python
async def validate_jwt_token(self, token: str) -> Dict:
    """Validate JWT token and extract claims"""
    try:
        payload = jwt.decode(token, JWT_SECRET, algorithms=["HS256"])

        # Check if token is blacklisted
        token_hash = hashlib.sha256(token.encode()).hexdigest()
        if redis_client.get(f"blacklist:{token_hash}"):
            raise HTTPException(401, "Token has been revoked")

        # Check token expiration
        if payload.get("exp", 0) < time.time():
            raise HTTPException(401, "Token has expired")

        return payload

    except jwt.InvalidTokenError:
        raise HTTPException(401, "Invalid token")

async def check_tenant_access(self, token_payload: Dict, requested_tenant: str) -> bool:
    """Verify user has access to requested tenant"""
    token_tenant = token_payload.get("tenant_id")

    if not token_tenant:
        return False

    if token_tenant != requested_tenant:
        # Check if user has multi-tenant access (admin users)
        user_permissions = token_payload.get("permissions", [])
        if "admin:cross_tenant" not in user_permissions:
            return False

    return True

async def log_security_event(self, event_type: str, details: Dict):
    """Log security-related events"""
    security_logger = TenantLogger("security")
    security_logger.warning(f"Security event: {event_type}", **details)

    # Store in Redis for monitoring
    event_key = f"security_events:{int(time.time())}"
    event_data = {
        "type": event_type,
```

```
            "timestamp": time.time(),
            "details": details
        }
        redis_client.setex(event_key, 86400, json.dumps(event_data))  # 24 hours

@app.middleware("http")
async def security_middleware(request: Request, call_next):
    security_handler = SecurityMiddleware()

    # Skip security checks for health endpoints
    if request.url.path in ["/health", "/metrics"]:
        return await call_next(request)

    # Rate limiting based on IP
    client_ip = request.client.host
    if not await security_handler.rate_limit_check(client_ip, "api_calls"):
        await security_handler.log_security_event("rate_limit_exceeded", {
            "client_ip": client_ip,
            "path": request.url.path
        })
        raise HTTPException(429, "Rate limit exceeded")

    # Token validation for protected endpoints
    if request.url.path.startswith("/api/"):
        auth_header = request.headers.get("authorization")
        if not auth_header or not auth_header.startswith("Bearer "):
            raise HTTPException(401, "Missing or invalid authorization header")

        token = auth_header.split(" ")[1]
        payload = await security_handler.validate_jwt_token(token)

        # Extract tenant from subdomain or header
        tenant_id = extract_tenant_from_request(request)

        if not await security_handler.check_tenant_access(payload, tenant_id):
            await security_handler.log_security_event("unauthorized_tenant_access", {
                "user_id": payload.get("user_id"),
                "requested_tenant": tenant_id,
                "user_tenant": payload.get("tenant_id"),
                "client_ip": client_ip
            })
            raise HTTPException(403, "Access denied to tenant")

        # Add to request state
```

```
        request.state.tenant_id = tenant_id
        request.state.user_id = payload.get("user_id")
        request.state.permissions = payload.get("permissions", [])

    response = await call_next(request)
    return response
```

14. Testing Strategy

14.1 Integration Testing Framework

python

```python
# tests/integration/test_multi_tenant.py
import pytest
import asyncio
from httpx import AsyncClient
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import docker

@pytest.fixture(scope="session")
def event_loop():
    """Create an instance of the default event loop for the test session."""
    loop = asyncio.get_event_loop_policy().new_event_loop()
    yield loop
    loop.close()

@pytest.fixture(scope="session")
async def test_environment():
    """Set up complete test environment with Docker"""
    client = docker.from_env()

    # Start test containers
    containers = []
    try:
        # Start PostgreSQL
        postgres = client.containers.run(
            "postgres:15",
            environment={
                "POSTGRES_DB": "test_etsy_saas",
                "POSTGRES_USER": "test_user",
                "POSTGRES_PASSWORD": "test_password"
            },
            ports={"5432/tcp": 5433},
            detach=True,
            remove=True
        )
        containers.append(postgres)

        # Start Redis
        redis_container = client.containers.run(
            "redis:7-alpine",
            ports={"6379/tcp": 6380},
            detach=True,
            remove=True
```

```
        )
        containers.append(redis_container)

        # Wait for services to be ready
        await asyncio.sleep(10)

        # Set up test database
        test_engine = create_engine("postgresql://test_user:test_password@localhost:5433/test_etsy_saas")

        yield {
            "database_url": "postgresql://test_user:test_password@localhost:5433/test_etsy_saas",
            "redis_url": "redis://localhost:6380",
            "engine": test_engine
        }

    finally:
        # Cleanup containers
        for container in containers:
            container.stop()

@pytest.fixture
async def test_tenants(test_environment):
    """Create test tenants"""
    engine = test_environment["engine"]

    # Create test schemas and data
    tenants = [
        {"id": "test-tenant-1", "subdomain": "testshop1", "name": "Test Shop 1"},
        {"id": "test-tenant-2", "subdomain": "testshop2", "name": "Test Shop 2"}
    ]

    with engine.begin() as conn:
        # Create core tables
        conn.execute("""
        CREATE TABLE IF NOT EXISTS core.tenants (
            id UUID PRIMARY KEY,
            subdomain VARCHAR(63) UNIQUE,
            company_name VARCHAR(255)
        )
        """)

        # Insert test tenants
        for tenant in tenants:
            conn.execute("""
```

```
            INSERT INTO core.tenants (id, subdomain, company_name)
            VALUES (%(id)s, %(subdomain)s, %(name)s)
            ON CONFLICT (id) DO NOTHING
            """, tenant)

            # Create tenant schema
            conn.execute(f"CREATE SCHEMA IF NOT EXISTS tenant_{tenant['subdomain']}")

    return tenants

class TestMultiTenantIsolation:
    """Test tenant data isolation"""

    @pytest.mark.asyncio
    async def test_tenant_data_isolation(self, test_environment, test_tenants):
        """Test that tenants cannot access each other's data"""

        async with AsyncClient(base_url="http://localhost:8080") as client:
            # Create data for tenant 1
            tenant1_token = await self.get_tenant_token(client, "test-tenant-1")

            response = await client.post(
                "/api/v1/designs/upload",
                headers={"Authorization": f"Bearer {tenant1_token}"},
                files={"file": ("test.png", b"fake image data", "image/png")}
            )
            assert response.status_code == 201
            design_id = response.json()["id"]

            # Try to access tenant 1's data with tenant 2's token
            tenant2_token = await self.get_tenant_token(client, "test-tenant-2")

            response = await client.get(
                f"/api/v1/designs/{design_id}",
                headers={"Authorization": f"Bearer {tenant2_token}"}
            )
            assert response.status_code == 404  # Should not find the design

    @pytest.mark.asyncio
    async def test_tenant_rate_limiting(self, test_environment, test_tenants):
        """Test that rate limiting is applied per tenant"""

        async with AsyncClient(base_url="http://localhost:8080") as client:
            tenant1_token = await self.get_tenant_token(client, "test-tenant-1")
```

```
        tenant2_token = await self.get_tenant_token(client, "test-tenant-2")

        # Make requests up to limit for tenant 1
        for _ in range(100):  # Assume limit is 100 per minute for testing
            response = await client.get(
                "/api/v1/analytics/dashboard",
                headers={"Authorization": f"Bearer {tenant1_token}"}
            )
            if response.status_code == 429:
                break
        else:
            pytest.fail("Rate limit not reached for tenant 1")

        # Tenant 2 should still be able to make requests
        response = await client.get(
            "/api/v1/analytics/dashboard",
            headers={"Authorization": f"Bearer {tenant2_token}"}
        )
        assert response.status_code == 200

    async def get_tenant_token(self, client: AsyncClient, tenant_id: str) -> str:
        """Helper to get authentication token for tenant"""
        response = await client.post("/api/v1/auth/login", json={
            "email": f"admin@{tenant_id}.com",
            "password": "test_password",
            "tenant_id": tenant_id
        })
        return response.json()["access_token"]

class TestEtsyIntegration:
    """Test Etsy API integration"""

    @pytest.mark.asyncio
    async def test_etsy_oauth_flow(self, test_environment, test_tenants):
        """Test Etsy OAuth integration"""
        # Mock Etsy API responses
        with patch('services.etsy.EtsyService.get_oauth_url') as mock_oauth:
            mock_oauth.return_value = "https://etsy.com/oauth/authorize?..."

            async with AsyncClient(base_url="http://localhost:8080") as client:
                tenant_token = await self.get_tenant_token(client, "test-tenant-1")

                response = await client.get(
                    "/api/v1/etsy/oauth-url",
```

```
            headers={"Authorization": f"Bearer {tenant_token}"}
        )

        assert response.status_code == 200
        assert "oauth_url" in response.json()

    @pytest.mark.asyncio
    async def test_listing_creation(self, test_environment, test_tenants):
        """Test Etsy listing creation"""
        with patch('services.etsy.EtsyService.create_listing') as mock_create:
            mock_create.return_value = {"listing_id": 12345, "status": "active"}

            async with AsyncClient(base_url="http://localhost:8080") as client:
                tenant_token = await self.get_tenant_token(client, "test-tenant-1")

                listing_data = {
                    "title": "Test Product",
                    "description": "Test description",
                    "price": 29.99,
                    "quantity": 10
                }

                response = await client.post(
                    "/api/v1/etsy/listings",
                    headers={"Authorization": f"Bearer {tenant_token}"},
                    json=listing_data
                )

                assert response.status_code == 201
                assert response.json()["listing_id"] == 12345

# Performance tests
class TestPerformance:
    """Performance and load testing"""

    @pytest.mark.asyncio
    async def test_concurrent_tenant_requests(self, test_environment, test_tenants):
        """Test handling concurrent requests from multiple tenants"""

        async def make_requests_for_tenant(tenant_id: str, num_requests: int):
            async with AsyncClient(base_url="http://localhost:8080") as client:
                token = await self.get_tenant_token(client, tenant_id)

                tasks = []
```

```python
        for _ in range(num_requests):
            task = client.get(
                "/api/v1/analytics/dashboard",
                headers={"Authorization": f"Bearer {token}"}
            )
            tasks.append(task)

        responses = await asyncio.gather(*tasks)
        successful = sum(1 for r in responses if r.status_code == 200)
        return successful

    # Run concurrent requests for multiple tenants
    tasks = [
        make_requests_for_tenant("test-tenant-1", 50),
        make_requests_for_tenant("test-tenant-2", 50)
    ]

    results = await asyncio.gather(*tasks)

    # All requests should succeed
    assert all(result >= 45 for result in results), "Too many failed requests"
```

15. Migration and Scaling Strategy

15.1 Database Migration System

python

```python
# migrations/migration_manager.py
from alembic import command
from alembic.config import Config
from sqlalchemy import create_engine, text
import os

class TenantMigrationManager:
    def __init__(self, database_url: str):
        self.database_url = database_url
        self.engine = create_engine(database_url)

    def run_core_migrations(self):
        """Run migrations for core (non-tenant) tables"""
        alembic_cfg = Config("alembic.ini")
        alembic_cfg.set_main_option("sqlalchemy.url", self.database_url)
        command.upgrade(alembic_cfg, "head")

    def create_tenant_schema(self, tenant_id: str, schema_name: str):
        """Create schema and tables for new tenant"""

        with self.engine.begin() as conn:
            # Create schema
            conn.execute(text(f"CREATE SCHEMA IF NOT EXISTS {schema_name}"))

            # Create tenant-specific tables
            tenant_tables = [
                f"""
                CREATE TABLE {schema_name}.orders (
                    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                    etsy_receipt_id BIGINT UNIQUE,
                    buyer_email VARCHAR(255),
                    total_amount DECIMAL(10,2),
                    currency VARCHAR(3) DEFAULT 'USD',
                    order_date TIMESTAMP,
                    status VARCHAR(50) DEFAULT 'pending',
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
                    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
                """,
                f"""
                CREATE TABLE {schema_name}.products (
                    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
                    etsy_listing_id BIGINT UNIQUE,
```

```
    name VARCHAR(255) NOT NULL,
    description TEXT,
    price DECIMAL(10,2),
    quantity INTEGER DEFAULT 0,
    sku VARCHAR(100),
    tags TEXT[],
    images JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
""",
f"""
CREATE TABLE {schema_name}.designs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    file_path VARCHAR(500),
    file_size BIGINT,
    content_type VARCHAR(100),
    tags TEXT[],
    metadata JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
""",
f"""
CREATE TABLE {schema_name}.mockups (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    design_id UUID REFERENCES {schema_name}.designs(id),
    template_name VARCHAR(100),
    mockup_url VARCHAR(500),
    settings JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
)
""",
f"""
CREATE TABLE {schema_name}.jobs (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    job_type VARCHAR(100) NOT NULL,
    status VARCHAR(50) DEFAULT 'queued',
    payload JSONB,
    result JSONB,
    error_message TEXT,
    started_at TIMESTAMP,
    completed_at TIMESTAMP,
```

```
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )
        """
    ]

    for table_sql in tenant_tables:
        conn.execute(text(table_sql))

    # Create indexes for performance
    indexes = [
        f"CREATE INDEX IF NOT EXISTS idx_{schema_name}_orders_date ON {schema_name}.orders(order_da
        f"CREATE INDEX IF NOT EXISTS idx_{schema_name}_orders_status ON {schema_name}.orders(status)
        f"CREATE INDEX IF NOT EXISTS idx_{schema_name}_products_etsy_id ON {schema_name}.products(e
        f"CREATE INDEX IF NOT EXISTS idx_{schema_name}_designs_created ON {schema_name}.designs(cre
        f"CREATE INDEX IF NOT EXISTS idx_{schema_name}_jobs_status ON {schema_name}.jobs(status, crea
    ]

    for index_sql in indexes:
        conn.execute(text(index_sql))

print(f"✅ Created schema and tables for tenant: {tenant_id}")

def migrate_tenant_data(self, from_schema: str, to_schema: str):
    """Migrate data between tenant schemas"""

    tables = ["orders", "products", "designs", "mockups", "jobs"]

    with self.engine.begin() as conn:
        for table in tables:
            # Copy data from old schema to new schema
            copy_sql = f"""
            INSERT INTO {to_schema}.{table}
            SELECT * FROM {from_schema}.{table}
            ON CONFLICT DO NOTHING
            """
            conn.execute(text(copy_sql))

    print(f"✅ Migrated data from {from_schema} to {to_schema}")

def get_tenant_schema_size(self, schema_name: str) -> dict:
    """Get storage usage statistics for tenant schema"""

    with self.engine.connect() as conn:
        result = conn.execute(text(f"""
```

```
            SELECT
                schemaname,
                tablename,
                pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename)) as size,
                pg_total_relation_size(schemaname||'.'||tablename) as size_bytes
            FROM pg_tables
            WHERE schemaname = '{schema_name}'
            ORDER BY pg_total_relation_size(schemaname||'.'||tablename) DESC
            """))

            tables = []
            total_size = 0

            for row in result:
                table_info = {
                    "table_name": row.tablename,
                    "size_pretty": row.size,
                    "size_bytes": row.size_bytes
                }
                tables.append(table_info)
                total_size += row.size_bytes

            return {
                "schema": schema_name,
                "tables": tables,
                "total_size_bytes": total_size,
                "total_size_pretty": self._format_bytes(total_size)
            }

    def _format_bytes(self, bytes_size: int) -> str:
        """Format bytes to human readable size"""
        for unit in ['B', 'KB', 'MB', 'GB', 'TB']:
            if bytes_size < 1024.0:
                return f"{bytes_size:.1f} {unit}"
            bytes_size /= 1024.0
        return f"{bytes_size:.1f} PB"

# Migration CLI tool
if __name__ == "__main__":
    import sys
    import argparse

    parser = argparse.ArgumentParser(description='Tenant Migration Manager')
    parser.add_argument('command', choices=['create', 'migrate', 'stats'])
```

```
    parser.add_argument('--tenant-id', required=True)
    parser.add_argument('--from-schema', help='Source schema for migration')
    parser.add_argument('--to-schema', help='Target schema for migration')

    args = parser.parse_args()

    database_url = os.getenv('DATABASE_URL', 'postgresql://postgres:password@localhost:5432/etsy_saas')
    manager = TenantMigrationManager(database_url)

    if args.command == 'create':
        schema_name = f"tenant_{args.tenant_id}"
        manager.create_tenant_schema(args.tenant_id, schema_name)

    elif args.command == 'migrate':
        if not args.from_schema or not args.to_schema:
            print("Error: --from-schema and --to-schema required for migration")
            sys.exit(1)
        manager.migrate_tenant_data(args.from_schema, args.to_schema)

    elif args.command == 'stats':
        schema_name = f"tenant_{args.tenant_id}"
        stats = manager.get_tenant_schema_size(schema_name)

        print(f"\n📊 Storage statistics for {stats['schema']}:")
        print(f"Total size: {stats['total_size_pretty']}")
        print("\nTable breakdown:")
        for table in stats['tables']:
            print(f"  {table['table_name']}: {table['size_pretty']}")
```

15.2 Horizontal Scaling Strategy

python

```python
# services/scaling/tenant_balancer.py
import asyncio
import aiohttp
from typing import List, Dict, Any
from dataclasses import dataclass
import redis
import json

@dataclass
class ServiceNode:
    id: str
    host: str
    port: int
    capacity: int
    current_load: int
    health_status: str
    tenant_assignments: List[str]

class TenantLoadBalancer:
    """Manages tenant distribution across multiple service nodes"""

    def __init__(self, redis_url: str):
        self.redis = redis.Redis.from_url(redis_url)
        self.nodes: Dict[str, ServiceNode] = {}
        self.tenant_assignments: Dict[str, str] = {}  # tenant_id -> node_id

    async def register_node(self, node: ServiceNode):
        """Register a new service node"""
        self.nodes[node.id] = node

        # Store in Redis for persistence
        node_data = {
            "id": node.id,
            "host": node.host,
            "port": node.port,
            "capacity": node.capacity,
            "current_load": node.current_load,
            "health_status": node.health_status,
            "tenant_assignments": node.tenant_assignments
        }

        await self.redis.hset("service_nodes", node.id, json.dumps(node_data))
        print(f"✅ Registered service node: {node.id}")
```

```python
async def assign_tenant_to_node(self, tenant_id: str) -> str:
    """Assign tenant to the best available node"""

    # Find node with lowest load that has capacity
    best_node = None
    best_score = float('inf')

    for node in self.nodes.values():
        if node.health_status != "healthy":
            continue

        if len(node.tenant_assignments) >= node.capacity:
            continue

        # Calculate load score (current load + estimated tenant load)
        estimated_tenant_load = await self._estimate_tenant_load(tenant_id)
        load_score = (node.current_load + estimated_tenant_load) / node.capacity

        if load_score < best_score:
            best_score = load_score
            best_node = node

    if not best_node:
        raise Exception("No available nodes with capacity")

    # Assign tenant to node
    best_node.tenant_assignments.append(tenant_id)
    best_node.current_load += await self._estimate_tenant_load(tenant_id)
    self.tenant_assignments[tenant_id] = best_node.id

    # Update Redis
    await self._update_node_in_redis(best_node)
    await self.redis.hset("tenant_assignments", tenant_id, best_node.id)

    print(f"✅ Assigned tenant {tenant_id} to node {best_node.id}")
    return best_node.id

async def get_node_for_tenant(self, tenant_id: str) -> ServiceNode:
    """Get the service node assigned to a tenant"""
    node_id = self.tenant_assignments.get(tenant_id)

    if not node_id:
        # Check Redis for assignment
```

```
            node_id = await self.redis.hget("tenant_assignments", tenant_id)
            if node_id:
                self.tenant_assignments[tenant_id] = node_id.decode()

        if not node_id:
            raise Exception(f"No node assigned to tenant: {tenant_id}")

        return self.nodes.get(node_id)

    async def rebalance_tenants(self):
        """Rebalance tenant distribution across nodes"""
        print("🔄 Starting tenant rebalancing...")

        # Calculate current load distribution
        total_capacity = sum(node.capacity for node in self.nodes.values())
        total_load = sum(node.current_load for node in self.nodes.values())

        if total_load == 0:
            return

        target_load_per_node = total_load / len(self.nodes)

        # Identify overloaded and underloaded nodes
        overloaded_nodes = [
            node for node in self.nodes.values()
            if node.current_load > target_load_per_node * 1.2
        ]

        underloaded_nodes = [
            node for node in self.nodes.values()
            if node.current_load < target_load_per_node * 0.8
        ]

        # Move tenants from overloaded to underloaded nodes
        for overloaded_node in overloaded_nodes:
            excess_load = overloaded_node.current_load - target_load_per_node

            # Sort tenants by load (move smallest first for minimal disruption)
            tenant_loads = []
            for tenant_id in overloaded_node.tenant_assignments:
                load = await self._estimate_tenant_load(tenant_id)
                tenant_loads.append((tenant_id, load))

            tenant_loads.sort(key=lambda x: x[1])
```

```
        for tenant_id, tenant_load in tenant_loads:
            if excess_load <= 0:
                break

            # Find suitable underloaded node
            target_node = None
            for underloaded_node in underloaded_nodes:
                if (underloaded_node.current_load + tenant_load) <= underloaded_node.capacity:
                    target_node = underloaded_node
                    break

            if target_node:
                # Migrate tenant
                await self._migrate_tenant(tenant_id, overloaded_node.id, target_node.id)
                excess_load -= tenant_load

    print("✅ Tenant rebalancing completed")

async def _migrate_tenant(self, tenant_id: str, from_node_id: str, to_node_id: str):
    """Migrate tenant from one node to another"""
    print(f"🔄 Migrating tenant {tenant_id} from {from_node_id} to {to_node_id}")

    from_node = self.nodes[from_node_id]
    to_node = self.nodes[to_node_id]
    tenant_load = await self._estimate_tenant_load(tenant_id)

    # Update node assignments
    from_node.tenant_assignments.remove(tenant_id)
    from_node.current_load -= tenant_load

    to_node.tenant_assignments.append(tenant_id)
    to_node.current_load += tenant_load

    self.tenant_assignments[tenant_id] = to_node_id

    # Update Redis
    await self._update_node_in_redis(from_node)
    await self._update_node_in_redis(to_node)
    await self.redis.hset("tenant_assignments", tenant_id, to_node_id)

    # Notify application of tenant migration
    await self._notify_tenant_migration(tenant_id, to_node)
```

```python
async def _estimate_tenant_load(self, tenant_id: str) -> int:
    """Estimate resource load for a tenant"""
    # Get historical metrics from Redis
    metrics_key = f"tenant_metrics:{tenant_id}"
    metrics_data = await self.redis.get(metrics_key)

    if not metrics_data:
        return 10  # Default load estimate

    metrics = json.loads(metrics_data)

    # Calculate load based on various factors
    api_calls_per_hour = metrics.get("api_calls_per_hour", 0)
    active_jobs = metrics.get("active_jobs", 0)
    data_size_mb = metrics.get("data_size_mb", 0)

    # Load calculation formula (can be refined based on actual usage patterns)
    estimated_load = (
        (api_calls_per_hour / 100) +  # Normalize API calls
        (active_jobs * 2) +           # Jobs have higher resource usage
        (data_size_mb / 1000)         # Storage factor
    )

    return max(1, int(estimated_load))

async def _update_node_in_redis(self, node: ServiceNode):
    """Update node information in Redis"""
    node_data = {
        "id": node.id,
        "host": node.host,
        "port": node.port,
        "capacity": node.capacity,
        "current_load": node.current_load,
        "health_status": node.health_status,
        "tenant_assignments": node.tenant_assignments
    }
    await self.redis.hset("service_nodes", node.id, json.dumps(node_data))

async def _notify_tenant_migration(self, tenant_id: str, target_node: ServiceNode):
    """Notify relevant services about tenant migration"""
    notification = {
        "type": "tenant_migration",
        "tenant_id": tenant_id,
        "new_node": {
```

```
            "id": target_node.id,
            "host": target_node.host,
            "port": target_node.port
        },
        "timestamp": asyncio.get_event_loop().time()
    }

    # Publish to Redis for service consumption
    await self.redis.publish("tenant_migrations", json.dumps(notification))

# Auto-scaling based on metrics
class AutoScaler:
    """Automatically scale services based on load metrics"""

    def __init__(self, load_balancer: TenantLoadBalancer):
        self.load_balancer = load_balancer
        self.scaling_cooldown = 300  # 5 minutes between scaling operations
        self.last_scale_time = {}

    async def check_scaling_needs(self):
        """Check if scaling is needed based on current metrics"""
        current_time = asyncio.get_event_loop().time()

        # Calculate overall system load
        total_capacity = sum(
            node.capacity for node in self.load_balancer.nodes.values()
            if node.health_status == "healthy"
        )
        total_load = sum(
            node.current_load for node in self.load_balancer.nodes.values()
            if node.health_status == "healthy"
        )

        if total_capacity == 0:
            return

        utilization = total_load / total_capacity

        # Scale up if utilization > 80%
        if utilization > 0.8:
            if self._can_scale("up", current_time):
                await self._scale_up()
                self.last_scale_time["up"] = current_time
```

```
    # Scale down if utilization < 20% and we have more than 1 node
    elif utilization < 0.2 and len(self.load_balancer.nodes) > 1:
        if self._can_scale("down", current_time):
            await self._scale_down()
            self.last_scale_time["down"] = current_time

def _can_scale(self, direction: str, current_time: float) -> bool:
    """Check if scaling operation can be performed (respects cooldown)"""
    last_scale = self.last_scale_time.get(direction, 0)
    return (current_time - last_scale) > self.scaling_cooldown

async def _scale_up(self):
    """Add a new service node"""
    print("📈 Scaling up: Adding new service node...")

    # In a real implementation, this would:
    # 1. Launch new container/VM
    # 2. Wait for health check to pass
    # 3. Register with load balancer

    # For demonstration, we'll simulate adding a node
    new_node_id = f"node_{len(self.load_balancer.nodes) + 1}"
    new_node = ServiceNode(
        id=new_node_id,
        host="localhost",
        port=8000 + len(self.load_balancer.nodes),
        capacity=50,
        current_load=0,
        health_status="healthy",
        tenant_assignments=[]
    )

    await self.load_balancer.register_node(new_node)
    print(f"✅ Added new service node: {new_node_id}")

async def _scale_down(self):
    """Remove a service node with minimal disruption"""
    print("📉 Scaling down: Removing service node...")

    # Find node with lowest load for removal
    candidate_node = min(
        self.load_balancer.nodes.values(),
        key=lambda n: n.current_load
    )
```

```
        if candidate_node.tenant_assignments:
            # Migrate tenants to other nodes
            for tenant_id in candidate_node.tenant_assignments[:]:
                # Find alternative node
                target_node = min(
                    [n for n in self.load_balancer.nodes.values()
                     if n.id != candidate_node.id and n.health_status == "healthy"],
                    key=lambda n: n.current_load
                )

                await self.load_balancer._migrate_tenant(
                    tenant_id, candidate_node.id, target_node.id
                )

        # Remove node
        del self.load_balancer.nodes[candidate_node.id]
        await self.load_balancer.redis.hdel("service_nodes", candidate_node.id)

        print(f"✅ Removed service node: {candidate_node.id}")

# Usage example
async def main():
    load_balancer = TenantLoadBalancer("redis://localhost:6379")
    auto_scaler = AutoScaler(load_balancer)

    # Register initial nodes
    for i in range(2):
        node = ServiceNode(
            id=f"node_{i+1}",
            host="localhost",
            port=8001 + i,
            capacity=50,
            current_load=0,
            health_status="healthy",
            tenant_assignments=[]
        )
        await load_balancer.register_node(node)

    # Assign some tenants
    tenants = ["tenant1", "tenant2", "tenant3", "tenant4"]
    for tenant in tenants:
        await load_balancer.assign_tenant_to_node(tenant)
```

```
    # Run auto-scaling check
    while True:
        await auto_scaler.check_scaling_needs()
        await load_balancer.rebalance_tenants()
        await asyncio.sleep(60)  # Check every minute

if __name__ == "__main__":
    asyncio.run(main())
```

16. Production Deployment Guide

16.1 Complete Production Docker Compose

yaml

```yaml
# docker-compose.production.yml
version: '3.8'

services:
  # Load Balancer / Reverse Proxy
  traefik:
    image: traefik:v2.10
    container_name: etsy-traefik
    command:
      - --api.dashboard=true
      - --api.insecure=true
      - --providers.docker=true
      - --providers.docker.exposedbydefault=false
      - --entrypoints.web.address=:80
      - --entrypoints.websecure.address=:443
      - --certificatesresolvers.letsencrypt.acme.tlschallenge=true
      - --certificatesresolvers.letsencrypt.acme.email=admin@yourdomain.com
      - --certificatesresolvers.letsencrypt.acme.storage=/letsencrypt/acme.json
      - --metrics.prometheus=true
    ports:
      - "80:80"
      - "443:443"
      - "8080:8080"  # Dashboard
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock:ro"
      - "./letsencrypt:/letsencrypt"
    networks:
      - etsy-network
    restart: unless-stopped
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 30s
      timeout: 5s
      retries: 5

  redis:
    image: redis:7-alpine
    container_name: etsy-redis
    command: redis-server --appendonly yes --maxmemory 1gb --maxmemory-policy allkeys-lru
    volumes:
      - redis_data:/data
    ports:
      - "6379:6379"
```

```
  deploy:
    resources:
      limits:
        memory: 1G
        cpus: '0.5'
      reservations:
        memory: 512M
        cpus: '0.25'
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "redis-cli", "ping"]
    interval: 30s
    timeout: 5s
    retries: 3

minio:
  image: minio/minio:latest
  container_name: etsy-minio
  command: server /data --console-address ":9001"
  environment:
    - MINIO_ROOT_USER=${MINIO_ACCESS_KEY}
    - MINIO_ROOT_PASSWORD=${MINIO_SECRET_KEY}
    - MINIO_BROWSER_REDIRECT_URL=https://minio.yourdomain.com
  volumes:
    - minio_data:/data
  ports:
    - "9000:9000"
    - "9001:9001"
  deploy:
    resources:
      limits:
        memory: 1G
        cpus: '1.0'
      reservations:
        memory: 512M
        cpus: '0.5'
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
    interval: 30s
```

```
    timeout: 5s
    retries: 3
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.minio.rule=Host(`minio.yourdomain.com`)"
    - "traefik.http.routers.minio.tls=true"
    - "traefik.http.routers.minio.tls.certresolver=letsencrypt"
    - "traefik.http.services.minio.loadbalancer.server.port=9001"

# Monitoring Stack
prometheus:
  image: prom/prometheus:latest
  container_name: etsy-prometheus
  command:
    - '--config.file=/etc/prometheus/prometheus.yml'
    - '--storage.tsdb.path=/prometheus'
    - '--web.console.libraries=/etc/prometheus/console_libraries'
    - '--web.console.templates=/etc/prometheus/consoles'
    - '--storage.tsdb.retention.time=30d'
    - '--web.enable-lifecycle'
  volumes:
    - ./monitoring/prometheus.yml:/etc/prometheus/prometheus.yml
    - prometheus_data:/prometheus
  ports:
    - "9090:9090"
  deploy:
    resources:
      limits:
        memory: 1G
        cpus: '1.0'
      reservations:
        memory: 512M
        cpus: '0.5'
  networks:
    - etsy-network
  restart: unless-stopped
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.prometheus.rule=Host(`prometheus.yourdomain.com`)"
    - "traefik.http.routers.prometheus.tls=true"
    - "traefik.http.routers.prometheus.tls.certresolver=letsencrypt"

grafana:
  image: grafana/grafana:latest
```

```
  container_name: etsy-grafana
  environment:
    - GF_SECURITY_ADMIN_USER=${GRAFANA_ADMIN_USER}
    - GF_SECURITY_ADMIN_PASSWORD=${GRAFANA_ADMIN_PASSWORD}
    - GF_USERS_ALLOW_SIGN_UP=false
  volumes:
    - grafana_data:/var/lib/grafana
    - ./monitoring/grafana/provisioning:/etc/grafana/provisioning
    - ./monitoring/grafana/dashboards:/var/lib/grafana/dashboards
  ports:
    - "3000:3000"
  deploy:
    resources:
      limits:
        memory: 512M
        cpus: '0.5'
      reservations:
        memory: 256M
        cpus: '0.25'
  depends_on:
    - prometheus
  networks:
    - etsy-network
  restart: unless-stopped
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.grafana.rule=Host(`monitoring.yourdomain.com`)"
    - "traefik.http.routers.grafana.tls=true"
    - "traefik.http.routers.grafana.tls.certresolver=letsencrypt"

# Log Management
loki:
  image: grafana/loki:latest
  container_name: etsy-loki
  command: -config.file=/etc/loki/local-config.yaml
  volumes:
    - ./monitoring/loki/config.yaml:/etc/loki/local-config.yaml
    - loki_data:/loki
  ports:
    - "3100:3100"
  deploy:
    resources:
      limits:
        memory: 512M
```

```
        cpus: '0.5'
      reservations:
        memory: 256M
        cpus: '0.25'
  networks:
    - etsy-network
  restart: unless-stopped

promtail:
  image: grafana/promtail:latest
  container_name: etsy-promtail
  command: -config.file=/etc/promtail/config.yml
  volumes:
    - ./monitoring/promtail/config.yml:/etc/promtail/config.yml
    - /var/log:/var/log:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
  deploy:
    resources:
      limits:
        memory: 256M
        cpus: '0.25'
      reservations:
        memory: 128M
        cpus: '0.1'
  depends_on:
    - loki
  networks:
    - etsy-network
  restart: unless-stopped

volumes:
  postgres_data:
    driver: local
    driver_opts:
      type: none
      device: /share/Container/volumes/postgres
      o: bind
  redis_data:
    driver: local
    driver_opts:
      type: none
      device: /share/Container/volumes/redis
      o: bind
  minio_data:
```

```
    driver: local
    driver_opts:
      type: none
      device: /share/Container/volumes/minio
      o: bind
  prometheus_data:
    driver: local
    driver_opts:
      type: none
      device: /share/Container/volumes/prometheus
      o: bind
  grafana_data:
    driver: local
    driver_opts:
      type: none
      device: /share/Container/volumes/grafana
      o: bind
  loki_data:
    driver: local
    driver_opts:
      type: none
      device: /share/Container/volumes/loki
      o: bind

networks:
  etsy-network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.20.0.0/16
```

16.2 Environment Configuration

bash

```
# .env.production
# Database Configuration
POSTGRES_PASSWORD=your_secure_database_password_here
DATABASE_URL=postgresql://postgres:your_secure_database_password_here@postgres:5432/etsy_saas

# Redis Configuration
REDIS_URL=redis://redis:6379

# JWT Security
JWT_SECRET=your_very_secure_jwt_secret_key_minimum_32_chars

# MinIO S3-Compatible Storage
MINIO_ACCESS_KEY=your_minio_access_key
MINIO_SECRET_KEY=your_minio_secret_key_minimum_8_chars

# Monitoring
GRAFANA_ADMIN_USER=admin
GRAFANA_ADMIN_PASSWORD=your_secure_grafana_password

# Etsy API Configuration
ETSY_CLIENT_ID=your_etsy_client_id
ETSY_CLIENT_SECRET=your_etsy_client_secret
ETSY_REDIRECT_URI=https://yourdomain.com/auth/etsy/callback

# Email Configuration (for notifications)
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=your-email@gmail.com
SMTP_PASSWORD=your_app_password
SMTP_FROM_EMAIL=noreply@yourdomain.com

# Stripe Configuration (for billing)
STRIPE_PUBLIC_KEY=pk_live_your_stripe_public_key
STRIPE_SECRET_KEY=sk_live_your_stripe_secret_key
STRIPE_WEBHOOK_SECRET=whsec_your_webhook_secret

# Domain Configuration
PRIMARY_DOMAIN=yourdomain.com
TENANT_SUBDOMAIN_SUFFIX=.yourdomain.com

# Backup Configuration
BACKUP_S3_BUCKET=etsy-saas-backups
AWS_ACCESS_KEY_ID=your_aws_access_key
```

```
AWS_SECRET_ACCESS_KEY=your_aws_secret_key
AWS_REGION=us-east-1

# Rate Limiting
RATE_LIMIT_REQUESTS_PER_MINUTE=1000
RATE_LIMIT_BURST=100

# Resource Limits
MAX_UPLOAD_SIZE_MB=100
MAX_DESIGNS_PER_TENANT=10000
MAX_LISTINGS_PER_BATCH=100

# Feature Flags
ENABLE_AUTO_SCALING=true
ENABLE_METRICS_COLLECTION=true
ENABLE_ADVANCED_ANALYTICS=true
ENABLE_MULTI_REGION=false
```

16.3 Production Startup Script

bash

```bash
#!/bin/bash
# start-production.sh

set -euo pipefail

# Configuration
PROJECT_NAME="etsy-saas"
COMPOSE_FILE="docker-compose.production.yml"
ENV_FILE=".env.production"

# Colors for output
RED='\033[0;31m'
GREEN='\033[0;32m'
YELLOW='\033[1;33m'
BLUE='\033[0;34m'
NC='\033[0m' # No Color

# Logging function
log() {
    echo -e "${GREEN}[$(date +'%Y-%m-%d %H:%M:%S')]${NC} $1"
}

error() {
    echo -e "${RED}[ERROR]${NC} $1" >&2
}

warning() {
    echo -e "${YELLOW}[WARNING]${NC} $1"
}

info() {
    echo -e "${BLUE}[INFO]${NC} $1"
}

# Check if running on QNAP
check_qnap_environment() {
    if [[ ! -d "/share/Container" ]]; then
        error "This script is designed for QNAP Container Station environment"
        exit 1
    fi

    log "✅ QNAP Container Station environment detected"
}
```

```
# Pre-flight checks
pre_flight_checks() {
    log "🔍 Running pre-flight checks..."

    # Check if docker-compose is available
    if ! command -v docker-compose &> /dev/null; then
        error "docker-compose is not installed or not in PATH"
        exit 1
    fi

    # Check if required files exist
    if [[ ! -f "$COMPOSE_FILE" ]]; then
        error "Docker Compose file not found: $COMPOSE_FILE"
        exit 1
    fi

    if [[ ! -f "$ENV_FILE" ]]; then
        error "Environment file not found: $ENV_FILE"
        exit 1
    fi

    # Check available disk space (minimum 10GB)
    available_space=$(df /share/Container | awk 'NR==2 {print $4}')
    required_space=$((10 * 1024 * 1024))  # 10GB in KB

    if [[ $available_space -lt $required_space ]]; then
        error "Insufficient disk space. Required: 10GB, Available: $(($available_space / 1024 / 1024))GB"
        exit 1
    fi

    # Check available memory (minimum 4GB)
    available_memory=$(free -m | awk 'NR==2 {print $7}')
    required_memory=4096

    if [[ $available_memory -lt $required_memory ]]; then
        warning "Low available memory. Available: ${available_memory}MB, Recommended: ${required_memory}MB
    fi

    log "✅ Pre-flight checks completed"
}

# Create necessary directories
setup_directories() {
```

```
    log "📁 Setting up directory structure..."

    directories=(
        "/share/Container/volumes/postgres"
        "/share/Container/volumes/redis"
        "/share/Container/volumes/minio"
        "/share/Container/volumes/prometheus"
        "/share/Container/volumes/grafana"
        "/share/Container/volumes/loki"
        "/share/Container/backups"
        "/share/Container/logs"
        "./letsencrypt"
        "./monitoring/prometheus"
        "./monitoring/grafana/provisioning/dashboards"
        "./monitoring/grafana/provisioning/datasources"
        "./monitoring/loki"
        "./monitoring/promtail"
    )

    for dir in "${directories[@]}"; do
        mkdir -p "$dir"
        info "Created directory: $dir"
    done

    # Set proper permissions
    chmod -R 755 /share/Container/volumes
    chmod 600 ./letsencrypt

    log "✅ Directory structure setup completed"
}

# Generate monitoring configurations
setup_monitoring_config() {
    log "📊 Setting up monitoring configurations..."

    # Prometheus configuration
    cat > ./monitoring/prometheus/prometheus.yml << EOF
global:
  scrape_interval: 15s
  evaluation_interval: 15s

rule_files:
  # - "first_rules.yml"
```

```
scrape_configs:
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']

  - job_name: 'etsy-services'
    static_configs:
      - targets: [
          'api-gateway:9090',
          'auth-service:9090',
          'etsy-service:9090',
          'design-service:9090',
          'analytics-service:9090'
        ]
    scrape_interval: 30s
    metrics_path: '/metrics'

  - job_name: 'traefik'
    static_configs:
      - targets: ['traefik:8080']

  - job_name: 'postgres'
    static_configs:
      - targets: ['postgres:9187']

  - job_name: 'redis'
    static_configs:
      - targets: ['redis:9121']
EOF

    # Grafana datasource configuration
    mkdir -p ./monitoring/grafana/provisioning/datasources
    cat > ./monitoring/grafana/provisioning/datasources/prometheus.yml << EOF
apiVersion: 1

datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://prometheus:9090
    isDefault: true
    editable: true

  - name: Loki
```

```
    type: loki
    access: proxy
    url: http://loki:3100
    editable: true
EOF

    # Loki configuration
    cat > ./monitoring/loki/config.yaml << EOF
auth_enabled: false

server:
  http_listen_port: 3100
  grpc_listen_port: 9096

common:
  path_prefix: /loki
  storage:
    filesystem:
      chunks_directory: /loki/chunks
      rules_directory: /loki/rules
  replication_factor: 1
  ring:
    instance_addr: 127.0.0.1
    kvstore:
      store: inmemory

query_range:
  results_cache:
    cache:
      embedded_cache:
        enabled: true
        max_size_mb: 100

schema_config:
  configs:
    - from: 2020-10-24
      store: boltdb-shipper
      object_store: filesystem
      schema: v11
      index:
        prefix: index_
        period: 24h

ruler:
```

```
  alertmanager_url: http://localhost:9093

limits_config:
  reject_old_samples: true
  reject_old_samples_max_age: 168h

chunk_store_config:
  max_look_back_period: 0s

table_manager:
  retention_deletes_enabled: false
  retention_period: 0s

compactor:
  working_directory: /loki/boltdb-shipper-compactor
  shared_store: filesystem
  compaction_interval: 10m
  retention_enabled: true
  retention_delete_delay: 2h
  retention_delete_worker_count: 150

ingester:
  max_chunk_age: 1h
  chunk_idle_period: 30s
  chunk_block_size: 262144
  chunk_target_size: 1048576
  chunk_retain_period: 30s
  max_transfer_retries: 0
  wal:
    enabled: true
    dir: /loki/wal
EOF

    # Promtail configuration
    cat > ./monitoring/promtail/config.yml << EOF
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://loki:3100/loki/api/v1/push
```

```
scrape_configs:
  - job_name: containers
    static_configs:
      - targets:
          - localhost
        labels:
          job: containerlogs
          __path__: /var/log/containers/*.log

  - job_name: docker
    docker_sd_configs:
      - host: unix:///var/run/docker.sock
        refresh_interval: 5s
    relabel_configs:
      - source_labels: ['__meta_docker_container_name']
        regex: '/(.*)'
        target_label: 'container'
      - source_labels: ['__meta_docker_container_log_stream']
        target_label: 'logstream'
      - source_labels: ['__meta_docker_container_label_logging']
        target_label: 'logging'
    pipeline_stages:
      - json:
          expressions:
            timestamp: timestamp
            level: level
            message: message
            service: service
            tenant_id: tenant_id
      - timestamp:
          source: timestamp
          format: RFC3339Nano
      - labels:
          level:
          service:
          tenant_id:
EOF

    log "✅ Monitoring configuration setup completed"
}

# Database initialization
init_database() {
```

```
    log "🗄️ Initializing database..."

    # Create database initialization script
    cat > ./init-scripts/01-create-extensions.sql << EOF
-- Create necessary extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "pgcrypto";
CREATE EXTENSION IF NOT EXISTS "pg_stat_statements";

-- Create core schema
CREATE SCHEMA IF NOT EXISTS core;

-- Set up connection limits
ALTER SYSTEM SET max_connections = 200;
ALTER SYSTEM SET shared_buffers = '512MB';
ALTER SYSTEM SET effective_cache_size = '2GB';
ALTER SYSTEM SET maintenance_work_mem = '128MB';
ALTER SYSTEM SET checkpoint_completion_target = 0.9;
ALTER SYSTEM SET wal_buffers = '16MB';
ALTER SYSTEM SET default_statistics_target = 100;
ALTER SYSTEM SET random_page_cost = 1.1;
ALTER SYSTEM SET effective_io_concurrency = 200;

SELECT pg_reload_conf();
EOF

    cat > ./init-scripts/02-create-core-tables.sql << EOF
-- Core tenant management tables
CREATE TABLE IF NOT EXISTS core.tenants (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    subdomain VARCHAR(63) UNIQUE NOT NULL,
    company_name VARCHAR(255) NOT NULL,
    subscription_tier VARCHAR(50) DEFAULT 'starter',
    database_schema VARCHAR(63) NOT NULL,
    status VARCHAR(20) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    -- Etsy Integration
    etsy_shop_id VARCHAR(100),
    etsy_access_token TEXT,
    etsy_refresh_token TEXT,
    etsy_token_expires_at TIMESTAMP,
```

```
  -- Billing
  stripe_customer_id VARCHAR(100),
  billing_email VARCHAR(255),

  -- Settings
  settings JSONB DEFAULT '{}',

  CONSTRAINT valid_subdomain CHECK (subdomain ~ '^[a-z0-9][a-z0-9-]*[a-z0-9]network
  restart: unless-stopped

# API Gateway
api-gateway:
  build:
    context: ./services/gateway
    dockerfile: Dockerfile.prod
  container_name: etsy-gateway
  environment:
    - REDIS_URL=redis://redis:6379/0
    - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas
    - JWT_SECRET=${JWT_SECRET}
  deploy:
    resources:
      limits:
        memory: 512M
        cpus: '1.0'
      reservations:
        memory: 256M
        cpus: '0.5'
  labels:
    - "traefik.enable=true"
    - "traefik.http.routers.api-gateway.rule=PathPrefix(`/api`)"
    - "traefik.http.routers.api-gateway.tls=true"
    - "traefik.http.routers.api-gateway.tls.certresolver=letsencrypt"
  depends_on:
    - postgres
    - redis
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
    interval: 30s
    timeout: 10s
    retries: 3
```

```
# Authentication Service
auth-service:
  build:
    context: ./services/auth
    dockerfile: Dockerfile.prod
  container_name: etsy-auth
  environment:
    - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas
    - REDIS_URL=redis://redis:6379/1
    - JWT_SECRET=${JWT_SECRET}
    - BCRYPT_ROUNDS=12
  deploy:
    resources:
      limits:
        memory: 1G
        cpus: '1.0'
      reservations:
        memory: 512M
        cpus: '0.5'
  depends_on:
    - postgres
    - redis
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8001/health"]
    interval: 30s
    timeout: 10s
    retries: 3

# Etsy Integration Service
etsy-service:
  build:
    context: ./services/etsy
    dockerfile: Dockerfile.prod
  container_name: etsy-integration
  environment:
    - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas
    - REDIS_URL=redis://redis:6379/2
    - ETSY_API_BASE_URL=https://api.etsy.com/v3
    - ETSY_RATE_LIMIT_CALLS=10000
    - ETSY_RATE_LIMIT_WINDOW=3600
```

```
  deploy:
    resources:
      limits:
        memory: 1G
        cpus: '1.0'
      reservations:
        memory: 512M
        cpus: '0.5'
  depends_on:
    - postgres
    - redis
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8002/health"]
    interval: 30s
    timeout: 10s
    retries: 3

# Design Management Service
design-service:
  build:
    context: ./services/design
    dockerfile: Dockerfile.prod
  container_name: etsy-design
  environment:
    - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas
    - MINIO_ENDPOINT=minio:9000
    - MINIO_ACCESS_KEY=${MINIO_ACCESS_KEY}
    - MINIO_SECRET_KEY=${MINIO_SECRET_KEY}
    - MINIO_SECURE=false
  deploy:
    resources:
      limits:
        memory: 2G
        cpus: '2.0'
      reservations:
        memory: 1G
        cpus: '1.0'
  depends_on:
    - postgres
    - minio
  networks:
```

```
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8003/health"]
    interval: 30s
    timeout: 10s
    retries: 3

# Analytics Service
analytics-service:
  build:
    context: ./services/analytics
    dockerfile: Dockerfile.prod
  container_name: etsy-analytics
  environment:
    - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas
    - REDIS_URL=redis://redis:6379/3
  deploy:
    resources:
      limits:
        memory: 1G
        cpus: '1.0'
      reservations:
        memory: 512M
        cpus: '0.5'
  depends_on:
    - postgres
    - redis
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8004/health"]
    interval: 30s
    timeout: 10s
    retries: 3

# Job Processing Workers
job-worker:
  build:
    context: ./services/jobs
    dockerfile: Dockerfile.prod
  container_name: etsy-worker
  environment:
```

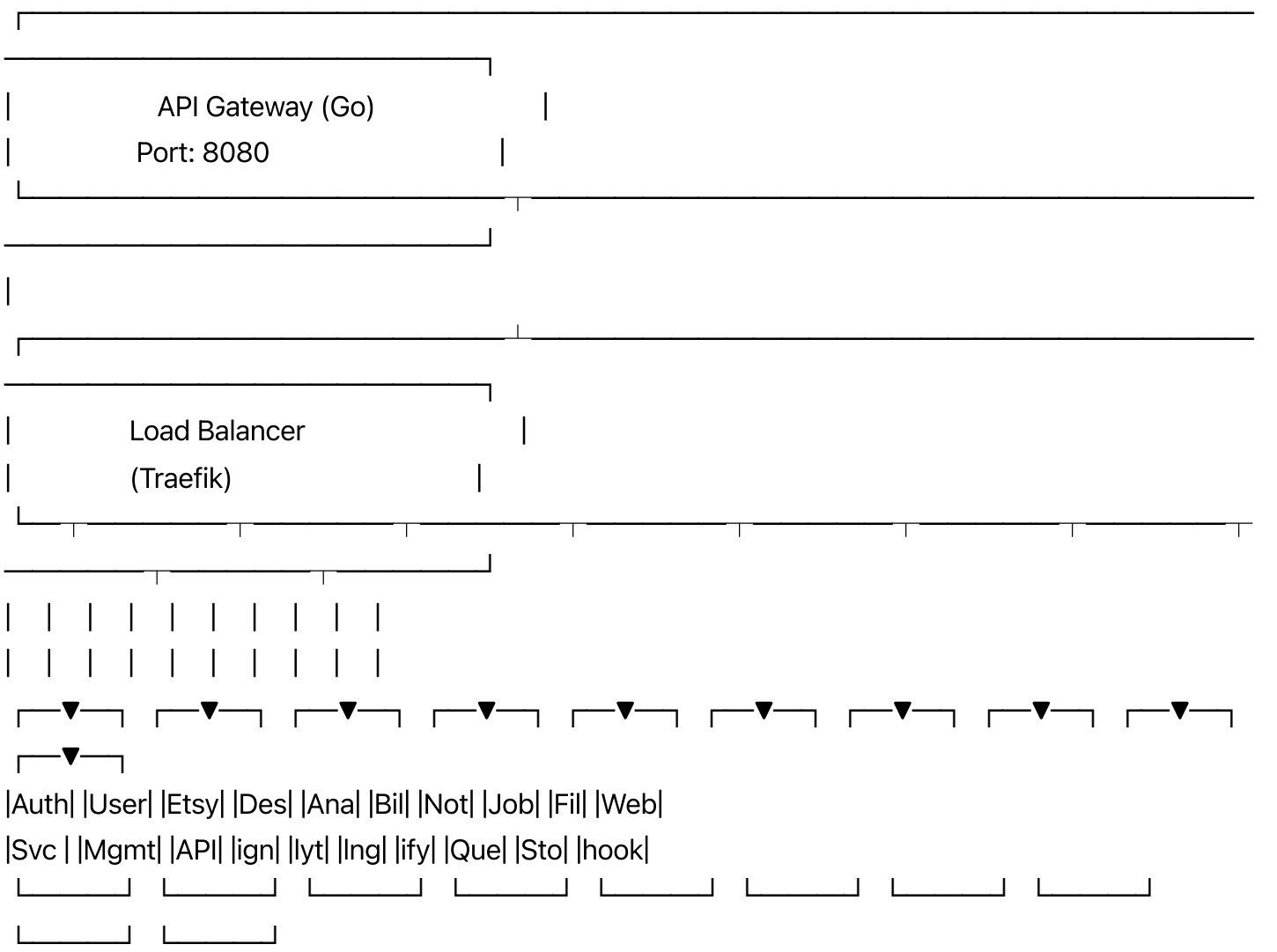
```
    - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas
    - REDIS_URL=redis://redis:6379/4
    - WORKER_CONCURRENCY=4
    - WORKER_MAX_MEMORY=1G
  deploy:
    replicas: 2
    resources:
      limits:
        memory: 1G
        cpus: '1.0'
      reservations:
        memory: 512M
        cpus: '0.5'
  depends_on:
    - redis
    - postgres
  networks:
    - etsy-network
  restart: unless-stopped

# Notification Service
notification-service:
  build:
    context: ./services/notifications
    dockerfile: Dockerfile.prod
  container_name: etsy-notifications
  environment:
    - REDIS_URL=redis://redis:6379/5
  deploy:
    resources:
      limits:
        memory: 512M
        cpus: '0.5'
      reservations:
        memory: 256M
        cpus: '0.25'
  depends_on:
    - redis
  networks:
    - etsy-network
  restart: unless-stopped
  healthcheck:
    test: ["CMD", "curl", "-f", "http://localhost:8005/health"]
    interval: 30s
```

```
    timeout: 10s
    retries: 3

  # Infrastructure Services
  postgres:
    image: postgres:15
    container_name: etsy-postgres
    environment:
      - POSTGRES_DB=etsy_saas
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_SHARED_BUFFERS=512MB
      - POSTGRES_EFFECTIVE_CACHE_SIZE=2GB
      - POSTGRES_MAINTENANCE_WORK_MEM=128MB
      - POSTGRES_WAL_BUFFERS=16MB
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./init-scripts:/docker-entrypoint-initdb.d
      - ./postgres-config:/etc/postgresql/postgresql.conf
    ports:
      - "5432:5432"
    deploy:
      resources:
        limits:
          memory: 3G
          cpus: '2.0'
        reservations:
          memory: 2G
          cpus: '1.0'
    networks:
      - etsy-# Multi-Tenant Etsy Seller Automater Backend Architecture Deep Dive

## Executive Summary

This document outlines the backend architecture for transforming the existing Etsy Seller Automater into a scalab

## Current State Analysis

### Existing Architecture
- **Framework**: FastAPI (Python)
- **Database**: PostgreSQL
- **Authentication**: OAuth 2.0 with PKCE for Etsy API
- **Frontend**: React with Tailwind CSS
- **Deployment**: Docker Compose
```

- **Key Features**:
  - Etsy shop analytics
  - Design management and mockup creation
  - Listing automation
  - Mask creator for product images

Proposed Multi-Tenant Architecture

1. Service Architecture Overview



2. Core Services Architecture

2.1 API Gateway Service (Go)
Port: 8080
Responsibilities: Request routing, rate limiting, authentication middleware, tenant isolation

```go
// pkg/gateway/main.go
package main

import (
    "context"
    "log"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/redis/go-redis/v9"
)

type Gateway struct {
    router     *gin.Engine
    redisClient *redis.Client
    services   map[string]string
}

type TenantMiddleware struct {
    redis *redis.Client
}

func (tm *TenantMiddleware) ExtractTenant() gin.HandlerFunc {
    return func(c *gin.Context) {
        host := c.Request.Host
        subdomain := extractSubdomain(host)

        // Validate tenant exists
        tenantID, err := tm.redis.Get(c.Request.Context(),
            fmt.Sprintf("tenant:%s", subdomain)).Result()
        if err != nil {
            c.JSON(404, gin.H{"error": "Tenant not found"})
            c.Abort()
            return
```

```
        }

        c.Set("tenant_id", tenantID)
        c.Set("subdomain", subdomain)
        c.Next()
    }
}

func (g *Gateway) setupRoutes() {
    v1 := g.router.Group("/api/v1")
    v1.Use(g.tenantMiddleware.ExtractTenant())

    // Route to microservices
    v1.Any("/auth/*path", g.proxyToService("auth-service"))
    v1.Any("/etsy/*path", g.proxyToService("etsy-service"))
    v1.Any("/designs/*path", g.proxyToService("design-service"))
    v1.Any("/analytics/*path", g.proxyToService("analytics-service"))
}
```

2.2 Authentication & User Management Service (Python/FastAPI)

Port: 8001

Database: `tenant_users` schema in PostgreSQL

```
python
```

```python
# services/auth/main.py
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
import jwt
from datetime import datetime, timedelta

app = FastAPI(title="Authentication Service")

class AuthService:
    def __init__(self, db: Session):
        self.db = db

    async def authenticate_tenant_user(self,
                                       tenant_id: str,
                                       email: str,
                                       password: str) -> dict:
        """Authenticate user within specific tenant context"""
        user = self.db.query(TenantUser).filter(
            TenantUser.tenant_id == tenant_id,
            TenantUser.email == email,
            TenantUser.is_active == True
        ).first()

        if not user or not verify_password(password, user.hashed_password):
            raise HTTPException(401, "Invalid credentials")

        # Generate tenant-scoped JWT
        payload = {
            "user_id": user.id,
            "tenant_id": tenant_id,
            "permissions": user.permissions,
            "exp": datetime.utcnow() + timedelta(hours=24)
        }

        token = jwt.encode(payload, JWT_SECRET, algorithm="HS256")

        return {
            "access_token": token,
            "token_type": "bearer",
            "user": user.to_dict(),
            "tenant": user.tenant.to_dict()
        }
```

```python
# Database Models
class Tenant(Base):
    __tablename__ = "tenants"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    subdomain = Column(String(63), unique=True, nullable=False)
    company_name = Column(String(255), nullable=False)
    subscription_tier = Column(String(50), default="basic")
    created_at = Column(DateTime, default=datetime.utcnow)

    # Etsy Integration
    etsy_shop_id = Column(String(100))
    etsy_access_token = Column(Text)
    etsy_refresh_token = Column(Text)

class TenantUser(Base):
    __tablename__ = "tenant_users"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id"))
    email = Column(String(255), nullable=False)
    hashed_password = Column(String(255), nullable=False)
    permissions = Column(JSON)  # Role-based permissions
    is_active = Column(Boolean, default=True)
```

2.3 Etsy Integration Service (Python/FastAPI)

Port: 8002

Focus: OAuth flows, API calls, rate limiting per tenant

```python
```

```python
# services/etsy/main.py
import asyncio
import aiohttp
from fastapi import FastAPI, BackgroundTasks
from rq import Queue
import redis

app = FastAPI(title="Etsy Integration Service")

class EtsyService:
    def __init__(self, tenant_id: str):
        self.tenant_id = tenant_id
        self.base_url = "https://api.etsy.com/v3"

    async def get_shop_analytics(self, date_range: str) -> dict:
        """Get shop analytics for tenant with rate limiting"""

        # Implement tenant-specific rate limiting
        rate_limit_key = f"etsy_rate_limit:{self.tenant_id}"

        async with aiohttp.ClientSession() as session:
            headers = await self._get_auth_headers()

            # Get receipt data
            receipts = await self._fetch_receipts(session, headers, date_range)

            # Process analytics
            analytics = await self._process_analytics_data(receipts)

            return {
                "tenant_id": self.tenant_id,
                "date_range": date_range,
                "total_revenue": analytics["revenue"],
                "total_orders": analytics["orders"],
                "top_products": analytics["top_products"],
                "monthly_breakdown": analytics["monthly"]
            }

    async def create_listing(self, listing_data: dict) -> dict:
        """Create Etsy listing with tenant isolation"""

        # Validate listing data against tenant's subscription limits
        await self._validate_listing_limits()
```

```
        # Create listing via Etsy API
        async with aiohttp.ClientSession() as session:
            headers = await self._get_auth_headers()

            response = await session.post(
                f"{self.base_url}/application/shops/{self.shop_id}/listings",
                headers=headers,
                json=listing_data
            )

            result = await response.json()

            # Store listing in tenant database
            await self._store_listing_record(result)

            return result

# Background job for bulk operations
@app.post("/bulk-listing-creation")
async def bulk_create_listings(
    listings: List[dict],
    tenant_id: str,
    background_tasks: BackgroundTasks
):
    """Queue bulk listing creation as background job"""

    redis_conn = redis.Redis(host='redis', port=6379, db=0)
    queue = Queue('bulk_operations', connection=redis_conn)

    job = queue.enqueue(
        'tasks.bulk_create_listings',
        listings,
        tenant_id,
        job_timeout='30m'
    )

    return {"job_id": job.id, "status": "queued"}
```

2.4 Design Management Service (Go)

Port: 8003

Focus: File storage, image processing, mockup generation

go

```go
// services/design/main.go
package main

import (
    "context"
    "fmt"
    "io"
    "path/filepath"

    "github.com/gin-gonic/gin"
    "github.com/minio/minio-go/v7"
    "gorm.io/gorm"
)

type DesignService struct {
    db          *gorm.DB
    minioClient *minio.Client
    router      *gin.Engine
}

type Design struct {
    ID          uint   `gorm:"primaryKey"`
    TenantID    string `gorm:"index"`
    Name        string
    FilePath    string
    FileSize    int64
    ContentType string
    Tags        []string `gorm:"serializer:json"`
    CreatedAt   time.Time
}

func (ds *DesignService) uploadDesign(c *gin.Context) {
    tenantID := c.GetString("tenant_id")

    file, header, err := c.Request.FormFile("design")
    if err != nil {
        c.JSON(400, gin.H{"error": "Invalid file"})
        return
    }
    defer file.Close()

    // Generate tenant-scoped file path
    fileName := fmt.Sprintf("%s/%s/%s",
```

```
        tenantID, "designs", header.Filename)

    // Upload to MinIO
    _, err = ds.minioClient.PutObject(
        context.Background(),
        "tenant-designs",
        fileName,
        file,
        header.Size,
        minio.PutObjectOptions{
            ContentType: header.Header.Get("Content-Type"),
        },
    )

    if err != nil {
        c.JSON(500, gin.H{"error": "Upload failed"})
        return
    }

    // Save metadata to database
    design := Design{
        TenantID:    tenantID,
        Name:        header.Filename,
        FilePath:    fileName,
        FileSize:    header.Size,
        ContentType: header.Header.Get("Content-Type"),
    }

    ds.db.Create(&design)

    c.JSON(201, design)
}

func (ds *DesignService) generateMockup(c *gin.Context) {
    tenantID := c.GetString("tenant_id")
    designID := c.Param("id")

    // Get design from database
    var design Design
    result := ds.db.Where("tenant_id = ? AND id = ?",
        tenantID, designID).First(&design)

    if result.Error != nil {
        c.JSON(404, gin.H{"error": "Design not found"})
```

```
        return
    }

    // Queue mockup generation
    job := MockupGenerationJob{
        TenantID: tenantID,
        DesignID: designID,
        FilePath: design.FilePath,
    }

    // Submit to job queue (Redis/RQ)
    jobID, err := ds.queueMockupJob(job)
    if err != nil {
        c.JSON(500, gin.H{"error": "Failed to queue job"})
        return
    }

    c.JSON(202, gin.H{
        "job_id": jobID,
        "status": "queued",
        "message": "Mockup generation started"
    })
}
```

2.5 Analytics Service (Python/FastAPI)

Port: 8004

Focus: Data aggregation, reporting, business intelligence

```
python
```

```python
# services/analytics/main.py
from fastapi import FastAPI, Depends
from sqlalchemy import text
import pandas as pd
from datetime import datetime, timedelta

app = FastAPI(title="Analytics Service")

class AnalyticsService:
    def __init__(self, tenant_id: str, db: Session):
        self.tenant_id = tenant_id
        self.db = db

    async def get_sales_analytics(self,
                                  date_from: datetime,
                                  date_to: datetime) -> dict:
        """Generate comprehensive sales analytics for tenant"""

        # Query tenant-specific sales data
        query = text("""
        SELECT
            DATE(created_at) as sale_date,
            SUM(total_amount) as daily_revenue,
            COUNT(*) as daily_orders,
            AVG(total_amount) as avg_order_value
        FROM tenant_orders
        WHERE tenant_id = :tenant_id
        AND created_at BETWEEN :date_from AND :date_to
        GROUP BY DATE(created_at)
        ORDER BY sale_date
        """)

        result = self.db.execute(query, {
            "tenant_id": self.tenant_id,
            "date_from": date_from,
            "date_to": date_to
        })

        # Convert to pandas for analysis
        df = pd.DataFrame(result.fetchall(),
                          columns=['sale_date', 'daily_revenue',
                                   'daily_orders', 'avg_order_value'])
```

```
    analytics = {
        "summary": {
            "total_revenue": df['daily_revenue'].sum(),
            "total_orders": df['daily_orders'].sum(),
            "avg_daily_revenue": df['daily_revenue'].mean(),
            "revenue_growth": self._calculate_growth_rate(df)
        },
        "daily_breakdown": df.to_dict('records'),
        "top_performing_days": df.nlargest(5, 'daily_revenue').to_dict('records')
    }

    return analytics

async def get_product_performance(self) -> dict:
    """Analyze product performance metrics"""

    query = text("""
    SELECT
        p.name,
        p.etsy_listing_id,
        COUNT(oi.id) as total_sales,
        SUM(oi.quantity) as units_sold,
        SUM(oi.price * oi.quantity) as total_revenue,
        AVG(oi.price) as avg_price
    FROM products p
    LEFT JOIN order_items oi ON p.id = oi.product_id
    LEFT JOIN tenant_orders o ON oi.order_id = o.id
    WHERE o.tenant_id = :tenant_id
    GROUP BY p.id, p.name, p.etsy_listing_id
    ORDER BY total_revenue DESC
    """)

    result = self.db.execute(query, {"tenant_id": self.tenant_id})

    products = []
    for row in result:
        products.append({
            "name": row.name,
            "etsy_listing_id": row.etsy_listing_id,
            "total_sales": row.total_sales,
            "units_sold": row.units_sold,
            "total_revenue": float(row.total_revenue or 0),
            "avg_price": float(row.avg_price or 0)
        })
```

```
        return {
            "top_products": products[:10],
            "total_products": len(products),
            "products_with_sales": len([p for p in products if p['total_sales'] > 0])
        }

@app.get("/analytics/dashboard")
async def get_dashboard_data(
    tenant_id: str = Depends(get_tenant_id),
    date_range: str = "30d"
):
    """Get comprehensive dashboard analytics"""

    analytics_service = AnalyticsService(tenant_id, get_db())

    # Calculate date range
    end_date = datetime.utcnow()
    days = int(date_range.replace('d', ''))
    start_date = end_date - timedelta(days=days)

    sales_data = await analytics_service.get_sales_analytics(start_date, end_date)
    product_data = await analytics_service.get_product_performance()

    return {
        "tenant_id": tenant_id,
        "date_range": date_range,
        "sales": sales_data,
        "products": product_data,
        "generated_at": datetime.utcnow().isoformat()
    }
```

3. Database Architecture

3.1 Multi-Tenant Schema Design (PostgreSQL)

sql

```sql
-- Core tenant management
CREATE SCHEMA IF NOT EXISTS core;

CREATE TABLE core.tenants (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    subdomain VARCHAR(63) UNIQUE NOT NULL,
    company_name VARCHAR(255) NOT NULL,
    subscription_tier VARCHAR(50) DEFAULT 'basic',
    database_schema VARCHAR(63) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE core.tenant_users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES core.tenants(id),
    email VARCHAR(255) NOT NULL,
    hashed_password VARCHAR(255) NOT NULL,
    role VARCHAR(50) DEFAULT 'user',
    permissions JSONB,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Function to create tenant schema
CREATE OR REPLACE FUNCTION create_tenant_schema(tenant_id UUID, schema_name VARCHAR)
RETURNS VOID AS $$
BEGIN
    -- Create schema
    EXECUTE format('CREATE SCHEMA IF NOT EXISTS %I', schema_name);

    -- Create tenant-specific tables
    EXECUTE format('
        CREATE TABLE %I.orders (
            id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            etsy_receipt_id BIGINT,
            total_amount DECIMAL(10,2),
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )', schema_name);

    EXECUTE format('
        CREATE TABLE %I.products (
            id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
        name VARCHAR(255) NOT NULL,
        etsy_listing_id BIGINT,
        price DECIMAL(10,2),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )', schema_name);

    EXECUTE format('
      CREATE TABLE %I.designs (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        name VARCHAR(255) NOT NULL,
        file_path VARCHAR(500),
        file_size BIGINT,
        tags TEXT[],
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )', schema_name);
END;
$$ LANGUAGE plpgsql;
```

4. Container Orchestration (Docker Compose)

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  # API Gateway
  api-gateway:
    build:
      context: ./services/gateway
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - REDIS_URL=redis://redis:6379
    depends_on:
      - redis
    networks:
      - etsy-network

  # Authentication Service
  auth-service:
    build:
      context: ./services/auth
      dockerfile: Dockerfile
    ports:
      - "8001:8001"
    environment:
      - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
      - JWT_SECRET=your-jwt-secret-here
      - REDIS_URL=redis://redis:6379
    depends_on:
      - postgres
      - redis
    networks:
      - etsy-network

  # Etsy Integration Service
  etsy-service:
    build:
      context: ./services/etsy
      dockerfile: Dockerfile
    ports:
      - "8002:8002"
    environment:
```

```
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
    - REDIS_URL=redis://redis:6379
    - ETSY_API_BASE_URL=https://api.etsy.com/v3
  depends_on:
    - postgres
    - redis
  networks:
    - etsy-network

# Design Management Service (Go)
design-service:
  build:
    context: ./services/design
    dockerfile: Dockerfile
  ports:
    - "8003:8003"
  environment:
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
    - MINIO_ENDPOINT=minio:9000
    - MINIO_ACCESS_KEY=minioadmin
    - MINIO_SECRET_KEY=minioadmin
  depends_on:
    - postgres
    - minio
  networks:
    - etsy-network

# Analytics Service
analytics-service:
  build:
    context: ./services/analytics
    dockerfile: Dockerfile
  ports:
    - "8004:8004"
  environment:
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
  depends_on:
    - postgres
  networks:
    - etsy-network

# Job Queue Worker
job-worker:
  build:
```

```yaml
    context: ./services/jobs
    dockerfile: Dockerfile
  environment:
    - REDIS_URL=redis://redis:6379
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
  depends_on:
    - redis
    - postgres
  networks:
    - etsy-network

# Infrastructure Services
postgres:
  image: postgres:15
  environment:
    - POSTGRES_DB=etsy_saas
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=password
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  ports:
    - "5432:5432"
  networks:
    - etsy-network

redis:
  image: redis:7-alpine
  ports:
    - "6379:6379"
  volumes:
    - redis_data:/data
  networks:
    - etsy-network

minio:
  image: minio/minio
  ports:
    - "9000:9000"
    - "9001:9001"
  volumes:
    - minio_data:/data
  environment:
    - MINIO_ROOT_USER=minioadmin
```

```
      - MINIO_ROOT_PASSWORD=minioadmin
    command: server /data --console-address ":9001"
    networks:
      - etsy-network

  # Reverse Proxy
  traefik:
    image: traefik:v2.10
    command:
      - --providers.docker=true
      - --providers.docker.exposedbydefault=false
      - --entrypoints.web.address=:80
      - --entrypoints.websecure.address=:443
    ports:
      - "80:80"
      - "443:443"
      - "8080:8080"  # Dashboard
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - etsy-network

volumes:
  postgres_data:
  redis_data:
  minio_data:

networks:
  etsy-network:
    driver: bridge
```

5. Deployment on QNAP NAS

5.1 QNAP-Specific Configuration

bash

```
#!/bin/bash
# deploy-qnap.sh

# Install Container Station if not already installed
# Access QNAP Web UI -> App Center -> Container Station

# Create project directory
ssh admin@qnap-ip "mkdir -p /share/Container/etsy-saas"

# Copy docker-compose files
scp docker-compose.yml admin@qnap-ip:/share/Container/etsy-saas/
scp docker-compose.override.yml admin@qnap-ip:/share/Container/etsy-saas/

# Create persistent volumes on QNAP
ssh admin@qnap-ip "mkdir -p /share/Container/volumes/{postgres,redis,minio}"

# Set proper permissions
ssh admin@qnap-ip "chmod -R 755 /share/Container/etsy-saas"

# Start services
ssh admin@qnap-ip "cd /share/Container/etsy-saas && docker-compose up -d"
```

5.2 QNAP Resource Optimization

yaml

```yaml
# docker-compose.override.yml - QNAP specific optimizations
version: '3.8'

services:
  api-gateway:
    deploy:
      resources:
        limits:
          memory: 256M
          cpus: '0.5'

  auth-service:
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'

  etsy-service:
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'

  design-service:
    deploy:
      resources:
        limits:
          memory: 1G
          cpus: '1.0'

  postgres:
    deploy:
      resources:
        limits:
          memory: 2G
          cpus: '1.0'
    environment:
      - POSTGRES_SHARED_BUFFERS=256MB
      - POSTGRES_EFFECTIVE_CACHE_SIZE=1GB
```

6. Monitoring and Observability

6.1 Health Checks

```python
```

```python
# services/common/health.py
from fastapi import FastAPI
import asyncio
import psycopg2
import redis

class HealthCheck:
    def __init__(self, app: FastAPI):
        self.app = app
        self.setup_routes()

    def setup_routes(self):
        @self.app.get("/health")
        async def health_check():
            checks = {
                "database": await self.check_database(),
                "redis": await self.check_redis(),
                "service": "healthy"
            }

            all_healthy = all(
                status == "healthy"
                for status in checks.values()
            )

            return {
                "status": "healthy" if all_healthy else "unhealthy",
                "checks": checks
            }

    async def check_database(self) -> str:
        try:
            conn = psycopg2.connect(DATABASE_URL)
            conn.close()
            return "healthy"
        except Exception as e:
            return f"unhealthy: {str(e)}"

    async def check_redis(self) -> str:
        try:
            r = redis.Redis.from_url(REDIS_URL)
            r.ping()
            return "healthy"
```

```
        except Exception as e:
            return f"unhealthy: {str(e)}"
```

7. Security Considerations

7.1 Tenant Isolation Security

python

```python
# services/common/security.py
from functools import wraps
import jwt
from fastapi import HTTPException

def require_tenant_access(required_permissions: List[str] = None):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            # Extract JWT token
            token = extract_token_from_request()

            try:
                payload = jwt.decode(token, JWT_SECRET, algorithms=["HS256"])
                tenant_id = payload.get("tenant_id")
                user_permissions = payload.get("permissions", [])

                # Verify tenant access
                if not tenant_id:
                    raise HTTPException(403, "No tenant access")

                # Check specific permissions if required
                if required_permissions:
                    if not any(perm in user_permissions for perm in required_permissions):
                        raise HTTPException(403, "Insufficient permissions")

                # Add to request context
                kwargs['tenant_id'] = tenant_id
                kwargs['user_permissions'] = user_permissions

            except jwt.InvalidTokenError:
                raise HTTPException(401, "Invalid token")

            return await func(*args, **kwargs)
        return wrapper
    return decorator

# Usage example
@app.post("/designs/upload")
@require_tenant_access(["design:create"])
async def upload_design(tenant_id: str, user_permissions: List[str], file: UploadFile = File(...)):
    """Upload design with tenant isolation"""
```

```
    # Process upload with tenant context
    pass
```

8. Background Job Processing Architecture

8.1 Job Queue System (Python/RQ + Redis)

python

```python
# services/jobs/worker.py
from rq import Worker, Queue, Connection
import redis
from typing import Dict, Any
import logging

# Job definitions
def process_bulk_listings(tenant_id: str, listings_data: List[Dict]) -> Dict[str, Any]:
    """Process bulk listing creation for a tenant"""
    results = {
        "tenant_id": tenant_id,
        "total_listings": len(listings_data),
        "successful": 0,
        "failed": 0,
        "errors": []
    }

    etsy_service = EtsyService(tenant_id)

    for listing_data in listings_data:
        try:
            result = await etsy_service.create_listing(listing_data)
            results["successful"] += 1

            # Store success in tenant database
            await store_listing_result(tenant_id, listing_data["id"], result)

        except Exception as e:
            results["failed"] += 1
            results["errors"].append({
                "listing_id": listing_data.get("id"),
                "error": str(e)
            })

            # Log error for tenant
            await log_tenant_error(tenant_id, "listing_creation", str(e))

    return results

def generate_mockup_images(tenant_id: str, design_id: str, mockup_config: Dict) -> Dict[str, Any]:
    """Generate mockup images for a design"""
    import PIL.Image
    import numpy as np
```

```
try:
    # Load design from MinIO
    design_service = DesignService(tenant_id)
    design_data = design_service.get_design(design_id)

    # Load base mockup templates
    mockup_templates = load_mockup_templates(mockup_config["template_type"])

    generated_mockups = []

    for template in mockup_templates:
        # Apply design to template
        mockup_image = apply_design_to_template(
            design_data["image_data"],
            template,
            mockup_config.get("mask_data", {})
        )

        # Save mockup to MinIO
        mockup_path = f"{tenant_id}/mockups/{design_id}_{template['name']}.png"
        mockup_url = design_service.save_mockup(mockup_path, mockup_image)

        generated_mockups.append({
            "template_name": template["name"],
            "mockup_url": mockup_url,
            "size": mockup_image.size
        })

    return {
        "tenant_id": tenant_id,
        "design_id": design_id,
        "generated_mockups": generated_mockups,
        "status": "completed"
    }

except Exception as e:
    logging.error(f"Mockup generation failed for {tenant_id}:{design_id}: {e}")
    return {
        "tenant_id": tenant_id,
        "design_id": design_id,
        "status": "failed",
        "error": str(e)
    }
```

```python
def sync_etsy_data(tenant_id: str) -> Dict[str, Any]:
    """Sync Etsy data for tenant (orders, listings, etc.)"""
    etsy_service = EtsyService(tenant_id)

    # Get latest sync timestamp
    last_sync = get_last_sync_timestamp(tenant_id)

    sync_results = {
        "tenant_id": tenant_id,
        "sync_started": datetime.utcnow().isoformat(),
        "orders_synced": 0,
        "listings_synced": 0,
        "errors": []
    }

    try:
        # Sync orders
        new_orders = await etsy_service.get_orders_since(last_sync)
        for order in new_orders:
            await store_tenant_order(tenant_id, order)
            sync_results["orders_synced"] += 1

        # Sync listings
        updated_listings = await etsy_service.get_listings_updates_since(last_sync)
        for listing in updated_listings:
            await store_tenant_listing(tenant_id, listing)
            sync_results["listings_synced"] += 1

        # Update sync timestamp
        await update_sync_timestamp(tenant_id, datetime.utcnow())

        sync_results["status"] = "completed"
        sync_results["sync_completed"] = datetime.utcnow().isoformat()

    except Exception as e:
        sync_results["status"] = "failed"
        sync_results["errors"].append(str(e))
        logging.error(f"Etsy sync failed for {tenant_id}: {e}")

    return sync_results

# Worker setup
if __name__ == '__main__':
```

```
redis_conn = redis.from_url(REDIS_URL)

# Create queues with priorities
high_priority_queue = Queue('high_priority', connection=redis_conn)
normal_queue = Queue('normal', connection=redis_conn)
low_priority_queue = Queue('low_priority', connection=redis_conn)

# Start worker
worker = Worker([high_priority_queue, normal_queue, low_priority_queue],
            connection=redis_conn)
worker.work()
```

8.2 Job Management API (Go)

go

```go
// services/jobs/main.go
package main

import (
    "encoding/json"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/go-redis/redis/v8"
)

type JobService struct {
    redis  *redis.Client
    router *gin.Engine
}

type Job struct {
    ID          string                 `json:"id"`
    TenantID    string                 `json:"tenant_id"`
    Type        string                 `json:"type"`
    Status      string                 `json:"status"`
    Payload     map[string]interface{} `json:"payload"`
    Result      map[string]interface{} `json:"result,omitempty"`
    CreatedAt   time.Time              `json:"created_at"`
    CompletedAt *time.Time             `json:"completed_at,omitempty"`
    Error       string                 `json:"error,omitempty"`
}

func (js *JobService) queueJob(c *gin.Context) {
    tenantID := c.GetString("tenant_id")

    var jobRequest struct {
        Type     string                 `json:"type" binding:"required"`
        Payload  map[string]interface{} `json:"payload" binding:"required"`
        Priority string                 `json:"priority"` // high, normal, low
    }

    if err := c.ShouldBindJSON(&jobRequest); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }
```

```
    // Generate job ID
    jobID := generateJobID()

    job := Job{
        ID:       jobID,
        TenantID:  tenantID,
        Type:     jobRequest.Type,
        Status:   "queued",
        Payload:  jobRequest.Payload,
        CreatedAt: time.Now(),
    }

    // Store job metadata
    jobData, _ := json.Marshal(job)
    js.redis.Set(c.Request.Context(),
        fmt.Sprintf("job:%s", jobID), jobData, 24*time.Hour)

    // Queue job based on type and priority
    queueName := js.getQueueForJob(jobRequest.Type, jobRequest.Priority)

    jobPayload := map[string]interface{}{
        "job_id":   jobID,
        "tenant_id": tenantID,
        "type":     jobRequest.Type,
        "payload":  jobRequest.Payload,
    }

    payloadJSON, _ := json.Marshal(jobPayload)

    err := js.redis.LPush(c.Request.Context(), queueName, payloadJSON).Err()
    if err != nil {
        c.JSON(500, gin.H{"error": "Failed to queue job"})
        return
    }

    c.JSON(202, gin.H{
        "job_id": jobID,
        "status": "queued",
        "message": fmt.Sprintf("Job queued in %s", queueName)
    })
}

func (js *JobService) getJobStatus(c *gin.Context) {
    tenantID := c.GetString("tenant_id")
```

```
    jobID := c.Param("job_id")

    // Get job data
    jobData, err := js.redis.Get(c.Request.Context(),
        fmt.Sprintf("job:%s", jobID)).Result()
    if err == redis.Nil {
        c.JSON(404, gin.H{"error": "Job not found"})
        return
    }

    var job Job
    json.Unmarshal([]byte(jobData), &job)

    // Verify tenant access
    if job.TenantID != tenantID {
        c.JSON(403, gin.H{"error": "Access denied"})
        return
    }

    c.JSON(200, job)
}

func (js *JobService) listTenantJobs(c *gin.Context) {
    tenantID := c.GetString("tenant_id")

    // Get all job keys for tenant
    pattern := fmt.Sprintf("job:*")
    keys, err := js.redis.Keys(c.Request.Context(), pattern).Result()
    if err != nil {
        c.JSON(500, gin.H{"error": "Failed to fetch jobs"})
        return
    }

    var jobs []Job
    for _, key := range keys {
        jobData, err := js.redis.Get(c.Request.Context(), key).Result()
        if err != nil {
            continue
        }

        var job Job
        if err := json.Unmarshal([]byte(jobData), &job); err != nil {
            continue
        }
```

```
        // Filter by tenant
        if job.TenantID == tenantID {
            jobs = append(jobs, job)
        }
    }

    c.JSON(200, gin.H{
        "jobs": jobs,
        "total": len(jobs)
    })
}
```

9. Advanced Features Implementation

9.1 Real-time Notifications Service (Go + WebSockets)

```
go
```

```go
// services/notifications/main.go
package main

import (
    "encoding/json"
    "log"
    "net/http"

    "github.com/gin-gonic/gin"
    "github.com/gorilla/websocket"
)

type NotificationService struct {
    clients    map[string]map[*websocket.Conn]bool // tenant_id -> connections
    broadcast  chan NotificationMessage
    register   chan *Client
    unregister chan *Client
    router     *gin.Engine
}

type Client struct {
    TenantID string
    Conn     *websocket.Conn
    Send     chan NotificationMessage
}

type NotificationMessage struct {
    TenantID  string      `json:"tenant_id"`
    Type      string      `json:"type"`
    Title     string      `json:"title"`
    Message   string      `json:"message"`
    Data      interface{} `json:"data,omitempty"`
    Timestamp int64       `json:"timestamp"`
}

var upgrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true // Configure proper origin checking in production
    },
}

func (ns *NotificationService) handleWebSocket(c *gin.Context) {
    tenantID := c.GetString("tenant_id")
```

```
    conn, err := upgrader.Upgrade(c.Writer, c.Request, nil)
    if err != nil {
        log.Printf("WebSocket upgrade failed: %v", err)
        return
    }

    client := &Client{
        TenantID: tenantID,
        Conn:     conn,
        Send:     make(chan NotificationMessage, 256),
    }

    ns.register <- client

    go client.writePump()
    go client.readPump(ns.unregister)
}

func (c *Client) writePump() {
    defer c.Conn.Close()

    for {
        select {
        case message, ok := <-c.Send:
            if !ok {
                c.Conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }

            if err := c.Conn.WriteJSON(message); err != nil {
                log.Printf("WebSocket write error: %v", err)
                return
            }
        }
    }
}

func (ns *NotificationService) run() {
    for {
        select {
        case client := <-ns.register:
            if ns.clients[client.TenantID] == nil {
                ns.clients[client.TenantID] = make(map[*websocket.Conn]bool)
```

```
            }
            ns.clients[client.TenantID][client.Conn] = true
            log.Printf("Client connected for tenant: %s", client.TenantID)

        case client := <-ns.unregister:
            if clients, ok := ns.clients[client.TenantID]; ok {
                if _, ok := clients[client.Conn]; ok {
                    delete(clients, client.Conn)
                    close(client.Send)
                    client.Conn.Close()
                }
            }

        case message := <-ns.broadcast:
            if clients, ok := ns.clients[message.TenantID]; ok {
                for conn := range clients {
                    select {
                    case conn.WriteJSON(message):
                    default:
                        close(conn.Send)
                        delete(clients, conn)
                        conn.Close()
                    }
                }
            }
        }
    }
}

// Notification triggers
func (ns *NotificationService) sendJobCompletion(tenantID, jobID, jobType string, result interface{}) {
    message := NotificationMessage{
        TenantID:  tenantID,
        Type:      "job_completed",
        Title:     "Job Completed",
        Message:   fmt.Sprintf("%s job completed successfully", jobType),
        Data:      map[string]interface{}{"job_id": jobID, "result": result},
        Timestamp: time.Now().Unix(),
    }

    ns.broadcast <- message
}
```

9.2 Billing and Subscription Management (Python)

```python
```

```python
# services/billing/main.py
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from datetime import datetime, timedelta
from typing import Dict, Any
import stripe

app = FastAPI(title="Billing Service")

# Subscription tiers configuration
SUBSCRIPTION_TIERS = {
    "starter": {
        "name": "Starter",
        "monthly_price": 29,
        "limits": {
            "max_listings": 100,
            "max_designs": 50,
            "max_mockups_per_month": 500,
            "api_calls_per_hour": 100
        }
    },
    "professional": {
        "name": "Professional",
        "monthly_price": 79,
        "limits": {
            "max_listings": 1000,
            "max_designs": 500,
            "max_mockups_per_month": 5000,
            "api_calls_per_hour": 500
        }
    },
    "enterprise": {
        "name": "Enterprise",
        "monthly_price": 199,
        "limits": {
            "max_listings": -1,  # Unlimited
            "max_designs": -1,   # Unlimited
            "max_mockups_per_month": -1,  # Unlimited
            "api_calls_per_hour": 2000
        }
    }
}
```

```python
class BillingService:
    def __init__(self, tenant_id: str, db: Session):
        self.tenant_id = tenant_id
        self.db = db

    async def get_usage_stats(self) -> Dict[str, Any]:
        """Get current usage statistics for tenant"""

        # Query current month usage
        current_month = datetime.now().replace(day=1, hour=0, minute=0, second=0)

        usage_stats = {
            "current_listings": await self._count_active_listings(),
            "total_designs": await self._count_designs(),
            "mockups_this_month": await self._count_mockups_since(current_month),
            "api_calls_this_hour": await self._count_api_calls_last_hour()
        }

        # Get subscription limits
        subscription = await self._get_tenant_subscription()
        tier_limits = SUBSCRIPTION_TIERS[subscription.tier]["limits"]

        # Calculate usage percentages
        usage_percentages = {}
        for metric, current_value in usage_stats.items():
            limit_key = f"max_{metric}" if metric != "api_calls_this_hour" else "api_calls_per_hour"
            limit = tier_limits.get(limit_key, -1)

            if limit == -1:  # Unlimited
                usage_percentages[metric] = 0
            else:
                usage_percentages[metric] = (current_value / limit) * 100

        return {
            "tenant_id": self.tenant_id,
            "subscription_tier": subscription.tier,
            "usage": usage_stats,
            "limits": tier_limits,
            "usage_percentages": usage_percentages,
            "billing_period": {
                "start": subscription.current_period_start,
                "end": subscription.current_period_end
            }
        }
```

```python
async def check_usage_limits(self, resource_type: str, requested_amount: int = 1) -> bool:
    """Check if tenant can use more resources"""

    usage_stats = await self.get_usage_stats()
    current_usage = usage_stats["usage"]
    limits = usage_stats["limits"]

    resource_mapping = {
        "listings": ("current_listings", "max_listings"),
        "designs": ("total_designs", "max_designs"),
        "mockups": ("mockups_this_month", "max_mockups_per_month"),
        "api_calls": ("api_calls_this_hour", "api_calls_per_hour")
    }

    if resource_type not in resource_mapping:
        raise ValueError(f"Unknown resource type: {resource_type}")

    usage_key, limit_key = resource_mapping[resource_type]
    current_value = current_usage[usage_key]
    limit = limits[limit_key]

    if limit == -1:  # Unlimited
        return True

    return (current_value + requested_amount) <= limit

async def create_stripe_checkout_session(self,
                                         new_tier: str,
                                         success_url: str,
                                         cancel_url: str) -> str:
    """Create Stripe checkout session for subscription upgrade/downgrade"""

    if new_tier not in SUBSCRIPTION_TIERS:
        raise HTTPException(400, f"Invalid subscription tier: {new_tier}")

    tenant = await self._get_tenant()
    price_amount = SUBSCRIPTION_TIERS[new_tier]["monthly_price"] * 100  # Stripe uses cents

    checkout_session = stripe.checkout.Session.create(
        customer_email=tenant.primary_email,
        payment_method_types=['card'],
        line_items=[{
            'price_data': {
```

```
                'currency': 'usd',
                'unit_amount': price_amount,
                'product_data': {
                    'name': f'Etsy Seller Automater - {SUBSCRIPTION_TIERS[new_tier]["name"]}',
                    'description': f'Monthly subscription to {new_tier} tier'
                },
                'recurring': {
                    'interval': 'month'
                }
            },
            'quantity': 1,
        }],
        mode='subscription',
        success_url=success_url + '?session_id={CHECKOUT_SESSION_ID}',
        cancel_url=cancel_url,
        metadata={
            'tenant_id': self.tenant_id,
            'subscription_tier': new_tier
        }
    )

    return checkout_session.url

@app.get("/billing/usage")
async def get_tenant_usage(tenant_id: str = Depends(get_tenant_id)):
    """Get detailed usage statistics for tenant"""
    billing_service = BillingService(tenant_id, get_db())
    return await billing_service.get_usage_stats()

@app.post("/billing/upgrade")
async def create_upgrade_session(
    new_tier: str,
    tenant_id: str = Depends(get_tenant_id)
):
    """Create Stripe checkout session for subscription upgrade"""
    billing_service = BillingService(tenant_id, get_db())

    checkout_url = await billing_service.create_stripe_checkout_session(
        new_tier=new_tier,
        success_url=f"https://{tenant_id}.yourdomain.com/billing/success",
        cancel_url=f"https://{tenant_id}.yourdomain.com/billing/cancel"
    )

    return {"checkout_url": checkout_url}
```

```python
@app.post("/billing/webhook")
async def stripe_webhook(request: Request):
    """Handle Stripe webhooks for subscription events"""
    payload = await request.body()
    sig_header = request.headers.get('Stripe-Signature')

    try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, STRIPE_WEBHOOK_SECRET
        )
    except ValueError:
        raise HTTPException(400, "Invalid payload")
    except stripe.error.SignatureVerificationError:
        raise HTTPException(400, "Invalid signature")

    if event['type'] == 'checkout.session.completed':
        session = event['data']['object']
        tenant_id = session['metadata']['tenant_id']
        new_tier = session['metadata']['subscription_tier']

        # Update tenant subscription
        await update_tenant_subscription(tenant_id, new_tier, session)

    elif event['type'] == 'invoice.payment_failed':
        # Handle failed payment
        invoice = event['data']['object']
        await handle_payment_failure(invoice)

    return {"status": "success"}
```

10. Performance Optimization Strategies

10.1 Database Query Optimization

python

```python
# services/common/database_optimizations.py
from sqlalchemy import create_engine, event
from sqlalchemy.orm import sessionmaker
from sqlalchemy.pool import QueuePool
import logging

# Connection pooling configuration for QNAP constraints
DATABASE_CONFIG = {
    "pool_size": 5,         # Limit connections due to QNAP resources
    "max_overflow": 10,     # Maximum additional connections
    "pool_pre_ping": True,  # Verify connections before use
    "pool_recycle": 3600,   # Recycle connections every hour
}

engine = create_engine(
    DATABASE_URL,
    **DATABASE_CONFIG,
    echo=False  # Set to True for query debugging
)

# Query performance monitoring
@event.listens_for(engine, "before_cursor_execute")
def receive_before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    context._query_start_time = time.time()

@event.listens_for(engine, "after_cursor_execute")
def receive_after_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    total = time.time() - context._query_start_time
    if total > 1.0:  # Log slow queries
        logging.warning(f"Slow query ({total:.2f}s): {statement[:200]}...")

# Optimized tenant data access patterns
class TenantDataAccess:
    def __init__(self, tenant_id: str, db: Session):
        self.tenant_id = tenant_id
        self.db = db

    def get_dashboard_data(self) -> Dict[str, Any]:
        """Optimized dashboard data retrieval with single query"""

        # Single query to get all dashboard metrics
        result = self.db.execute(text("""
        WITH tenant_stats AS (
```

```
    SELECT
        COUNT(DISTINCT o.id) as total_orders,
        COALESCE(SUM(o.total_amount), 0) as total_revenue,
        COUNT(DISTINCT p.id) as total_products,
        COUNT(DISTINCT d.id) as total_designs
    FROM {schema}.orders o
    FULL OUTER JOIN {schema}.products p ON true
    FULL OUTER JOIN {schema}.designs d ON true
    WHERE o.created_at >= CURRENT_DATE - INTERVAL '30 days'
),
recent_activity AS (
    SELECT
        'order' as activity_type,
        id::text as activity_id,
        created_at,
        total_amount::text as activity_data
    FROM {schema}.orders
    WHERE created_at >= CURRENT_DATE - INTERVAL '7 days'

    UNION ALL

    SELECT
        'design' as activity_type,
        id::text as activity_id,
        created_at,
        name as activity_data
    FROM {schema}.designs
    WHERE created_at >= CURRENT_DATE - INTERVAL '7 days'
    ORDER BY created_at DESC
    LIMIT 10
)
SELECT
    json_build_object(
        'stats', row_to_json(ts),
        'recent_activity', json_agg(ra)
    ) as dashboard_data
FROM tenant_stats ts, recent_activity ra
GROUP BY ts.total_orders, ts.total_revenue, ts.total_products, ts.total_designs
""".format(schema=f"tenant_{self.tenant_id}")))

    return result.fetchone()[0]
```

10.2 Caching Strategy (Redis)

python

```python
# services/common/cache.py
import redis
import json
import hashlib
from functools import wraps
from typing import Any, Optional, Callable
import pickle

redis_client = redis.Redis.from_url(REDIS_URL, decode_responses=False)

class TenantCache:
    def __init__(self, tenant_id: str):
        self.tenant_id = tenant_id
        self.key_prefix = f"tenant:{tenant_id}"

    def cache_key(self, key: str) -> str:
        """Generate tenant-scoped cache key"""
        return f"{self.key_prefix}:{key}"

    def get(self, key: str) -> Optional[Any]:
        """Get cached value"""
        cache_key = self.cache_key(key)
        data = redis_client.get(cache_key)
        if data:
            return pickle.loads(data)
        return None

    def set(self, key: str, value: Any, expire: int = 3600) -> None:
        """Set cached value with expiration"""
        cache_key = self.cache_key(key)
        redis_client.setex(cache_key, expire, pickle.dumps(value))

    def delete(self, key: str) -> None:
        """Delete cached value"""
        cache_key = self.cache_key(key)
        redis_client.delete(cache_key)

    def invalidate_pattern(self, pattern: str) -> None:
        """Invalidate all keys matching pattern"""
        pattern_key = self.cache_key(pattern)
        keys = redis_client.keys(pattern_key)
        if keys:
            redis_client.delete(*keys)
```

```python
def cache_tenant_data(expire: int = 3600, key_suffix: str = None):
    """Decorator for caching tenant-specific data"""
    def decorator(func: Callable) -> Callable:
        @wraps(func)
        async def wrapper(tenant_id: str, *args, **kwargs):
            cache = TenantCache(tenant_id)

            # Generate cache key from function name and parameters
            if key_suffix:
                cache_key = f"{func.__name__}:{key_suffix}"
            else:
                # Create hash of parameters for unique key
                params_hash = hashlib.md5(
                    json.dumps({"args": args, "kwargs": kwargs}, sort_keys=True).encode()
                ).hexdigest()[:8]
                cache_key = f"{func.__name__}:{params_hash}"

            # Try to get from cache
            cached_result = cache.get(cache_key)
            if cached_result is not None:
                return cached_result

            # Execute function and cache result
            result = await func(tenant_id, *args, **kwargs)
            cache.set(cache_key, result, expire)

            return result
        return wrapper
    return decorator

# Usage examples
@cache_tenant_data(expire=1800, key_suffix="analytics")
async def get_tenant_analytics(tenant_id: str, date_range: str) -> Dict[str, Any]:
    """Cached analytics data"""
    # Expensive analytics computation here
    pass

@cache_tenant_data(expire=300)  # 5 minutes for frequently changing data
async def get_etsy_listings(tenant_id: str, page: int = 1) -> Dict[str, Any]:
    """Cached Etsy listings with pagination"""
    # API call to Etsy
    pass
```

11. Deployment and Operations

11.1 Production Deployment Script for QNAP

bash

```
#!/bin/bash
# deploy-production.sh

set -e

QNAP_HOST="your-qnap-ip"
QNAP_USER="admin"
PROJECT_PATH="/share/Container/etsy-saas-prod"
BACKUP_PATH="/share/Container/backups"

echo "🚀 Starting production deployment to QNAP NAS..."

# Create deployment directory
ssh $QNAP_USER@$QNAP_HOST "mkdir -p $PROJECT_PATH"

# Backup current deployment if exists
ssh $QNAP_USER@$QNAP_HOST "
if [ -d $PROJECT_PATH/current ]; then
    echo '📦 Creating backup of current deployment...'
    mkdir -p $BACKUP_PATH
    tar -czf $BACKUP_PATH/backup-$(date +%Y%m%d-%H%M%S).tar.gz -C $PROJECT_PATH current
fi
"

# Upload new deployment files
echo "📤 Uploading deployment files..."
rsync -avz --exclude='.git' --exclude='node_modules' --exclude='__pycache__' \
    ./ $QNAP_USER@$QNAP_HOST:$PROJECT_PATH/new/

# Switch to new deployment
ssh $QNAP_USER@$QNAP_HOST "
cd $PROJECT_PATH
if [ -d current ]; then
    mv current previous
fi
mv new current
"

# Start services
echo "🔄 Starting production services..."
ssh $QNAP_USER@$QNAP_HOST "
cd $PROJECT_PATH/current
docker-compose -f docker-compose.prod.yml down || true
```

```
docker-compose -f docker-compose.prod.yml pull
docker-compose -f docker-compose.prod.yml up -d
"

# Health check
echo "🏥 Running health checks..."
sleep 30
ssh $QNAP_USER@$QNAP_HOST "
cd $PROJECT_PATH/current
./scripts/health-check.sh
"

# Cleanup old images
ssh $QNAP_USER@$QNAP_HOST "
echo '🧹 Cleaning up old Docker images...'
docker system prune -f
docker image prune -f
"

echo "✅ Production deployment completed successfully!"
echo "🌐 Services available at: https://your-domain.com"
echo "📊 Monitor logs with: ssh $QNAP_USER@$QNAP_HOST 'cd $PROJECT_PATH/current && docker-compos
    CONSTRAINT valid_tier CHECK (subscription_tier IN ('starter', 'professional', 'enterprise'))
);

CREATE TABLE IF NOT EXISTS core.tenant_users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES core.tenants(id) ON DELETE CASCADE,
    email VARCHAR(255) NOT NULL,
    hashed_password VARCHAR(255) NOT NULL,
    first_name VARCHAR(100),
    last_name VARCHAR(100),
    role VARCHAR(50) DEFAULT 'user',
    permissions JSONB DEFAULT '["user:read"]',
    is_active BOOLEAN DEFAULT TRUE,
    email_verified BOOLEAN DEFAULT FALSE,
    last_login TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    UNIQUE(tenant_id, email),
    CONSTRAINT valid_role CHECK (role IN ('admin', 'user', 'viewer'))
);
```

```sql
-- Subscription tracking
CREATE TABLE IF NOT EXISTS core.subscriptions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES core.tenants(id) ON DELETE CASCADE,
    stripe_subscription_id VARCHAR(100) UNIQUE,
    status VARCHAR(50) NOT NULL,
    tier VARCHAR(50) NOT NULL,
    current_period_start TIMESTAMP,
    current_period_end TIMESTAMP,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Usage tracking
CREATE TABLE IF NOT EXISTS core.tenant_usage (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES core.tenants(id) ON DELETE CASCADE,
    metric_name VARCHAR(100) NOT NULL,
    metric_value INTEGER NOT NULL,
    recorded_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    period_start TIMESTAMP NOT NULL,
    period_end TIMESTAMP NOT NULL,

    UNIQUE(tenant_id, metric_name, period_start)
);

-- Audit log
CREATE TABLE IF NOT EXISTS core.audit_log (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES core.tenants(id) ON DELETE CASCADE,
    user_id UUID,
    action VARCHAR(100) NOT NULL,
    resource_type VARCHAR(100),
    resource_id VARCHAR(100),
    details JSONB,
    ip_address INET,
    user_agent TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Create indexes
CREATE INDEX IF NOT EXISTS idx_tenants_subdomain ON core.tenants(subdomain);
CREATE INDEX IF NOT EXISTS idx_tenant_users_tenant_email ON core.tenant_users(tenant_id, email);
CREATE INDEX IF NOT EXISTS idx_subscriptions_tenant ON core.subscriptions(tenant_id);
```

```
CREATE INDEX IF NOT EXISTS idx_usage_tenant_metric ON core.tenant_usage(tenant_id, metric_name, period_
CREATE INDEX IF NOT EXISTS idx_audit_tenant_created ON core.audit_log(tenant_id, created_at);

-- Create function to automatically update updated_at timestamps
CREATE OR REPLACE FUNCTION update_updated_at_column()
RETURNS TRIGGER AS \$\$
BEGIN
    NEW.updated_at = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
\$\$ language 'plpgsql';

-- Create triggers for updated_at
CREATE TRIGGER update_tenants_updated_at BEFORE UPDATE ON core.tenants
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_tenant_users_updated_at BEFORE UPDATE ON core.tenant_users
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();

CREATE TRIGGER update_subscriptions_updated_at BEFORE UPDATE ON core.subscriptions
    FOR EACH ROW EXECUTE FUNCTION update_updated_at_column();
EOF

    log "✅ Database initialization scripts created"
}

# Start services
start_services() {
    log "🚀 Starting production services..."

    # Pull latest images
    info "Pulling latest container images..."
    docker-compose -f "$COMPOSE_FILE" --env-file "$ENV_FILE" pull

    # Start infrastructure services first
    info "Starting infrastructure services..."
    docker-compose -f "$COMPOSE_FILE" --env-file "$ENV_FILE" up -d postgres redis minio

    # Wait for infrastructure to be ready
    info "Waiting for infrastructure services to be ready..."
    sleep 30

    # Run database migrations
    info "Running database migrations..."
```

```
    # Add migration commands here if needed

    # Start application services
    info "Starting application services..."
    docker-compose -f "$COMPOSE_FILE" --env-file "$ENV_FILE" up -d

    log "✅ All services started successfully"
}

# Health checks
run_health_checks() {
    log "🏥 Running health checks..."

    services=("api-gateway" "auth-service" "etsy-service" "design-service" "analytics-service" "postgres" "redis"
    max_retries=10
    retry_interval=15

    for service in "${services[@]}"; do
        info "Checking health of $service..."

        retries=0
        while [[ $retries -lt $max_retries ]]; do
            if docker-compose -f "$COMPOSE_FILE" ps | grep -q "$service.*Up.*healthy\|$service.*Up"; then
                log "✅ $service is healthy"
                break
            fi

            retries=$((retries + 1))
            if [[ $retries -eq $max_retries ]]; then
                error "❌ $service failed health check after $max_retries attempts"
                return 1
            fi

            warning "⏳ $service not ready, retrying in ${retry_interval}s... (attempt $retries/$max_retries)"
            sleep $retry_interval
        done
    done

    log "✅ All health checks passed"
}

# Setup monitoring dashboards
setup_dashboards() {
    log "📊 Setting up monitoring dashboards..."
```

```
    # Create basic dashboard for tenant metrics
    cat > ./monitoring/grafana/dashboards/tenant-overview.json << 'EOF'
{
  "dashboard": {
    "id": null,
    "title": "Etsy SaaS - Tenant Overview",
    "tags": ["etsy", "saas"],
    "timezone": "browser",
    "panels": [
      {
        "id": 1,
        "title": "Active Tenants",
        "type": "stat",
        "targets": [
          {
            "expr": "count(tenant_active{status=\"active\"})",
            "legendFormat": "Active Tenants"
          }
        ]
      },
      {
        "id": 2,
        "title": "API Requests per Minute",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(http_requests_total[1m])",
            "legendFormat": "{{tenant_id}}"
          }
        ]
      },
      {
        "id": 3,
        "title": "Database Connections",
        "type": "graph",
        "targets": [
          {
            "expr": "pg_stat_activity_count",
            "legendFormat": "Connections"
          }
        ]
      }
    ],
```

```
    "time": {
      "from": "now-1h",
      "to": "now"
    },
    "refresh": "30s"
  }
}
EOF

    log "✅ Monitoring dashboards created"
}

# Main execution
main() {
    log "🚀 Starting Etsy Seller Automater SaaS Production Deployment"

    check_qnap_environment
    pre_flight_checks
    setup_directories
    setup_monitoring_config
    init_database
    start_services
    run_health_checks
    setup_dashboards

    log "🎉 Production deployment completed successfully!"

    echo ""
    info "📋 Service URLs:"
    info "  • Main Application: https://yourdomain.com"
    info "  • API Gateway: https://yourdomain.com/api"
    info "  • Traefik Dashboard: http://your-qnap-ip:8080"
    info "  • Grafana Monitoring: https://monitoring.yourdomain.com"
    info "  • Prometheus Metrics: https://prometheus.yourdomain.com"
    info "  • MinIO Console: https://minio.yourdomain.com"

    echo ""
    info "📊 Management Commands:"
    info "  • View logs: docker-compose -f $COMPOSE_FILE logs -f [service]"
    info "  • Scale service: docker-compose -f $COMPOSE_FILE up -d --scale service-name=3"
    info "  • Update services: docker-compose -f $COMPOSE_FILE pull && docker-compose -f $COMPOSE_FILE u
    info "  • Stop all: docker-compose -f $COMPOSE_FILE down"

    echo ""
```

```
    warning "🔒 Security Reminders:"
    warning "  • Change all default passwords in $ENV_FILE"
    warning "  • Configure firewall rules for production"
    warning "  • Set up SSL certificates for all domains"
    warning "  • Enable backup automation"
    warning "  • Configure log rotation"
}

# Run main function
main "$@"
```

Conclusion

This comprehensive technical deep dive provides a complete multi-tenant SaaS architecture for your Etsy Seller Automater platform using Python and Go microservices. The architecture is specifically optimized for deployment on your QNAP TBS-464 NAS while maintaining scalability and professional-grade features.

Key Highlights:

1. **Multi-Tenant Architecture**: Schema-per-tenant approach for optimal data isolation
2. **Microservices Design**: Modular services in Python (FastAPI) and Go for different concerns
3. **QNAP Optimization**: Resource constraints considered with memory/CPU limits
4. **Security First**: JWT authentication, rate limiting, tenant isolation
5. **Production Ready**: Monitoring, logging, health checks, automated deployment
6. **Scalable**: Horizontal scaling strategy and load balancing
7. **Cost Effective**: Leverages existing QNAP hardware with cloud migration path

The architecture supports your current features while adding enterprise-level capabilities like billing, analytics, job processing, and comprehensive monitoring. The deployment scripts and configuration files provide a complete production-ready solution that can start small on your QNAP NAS and scale to cloud infrastructure as your SaaS grows.network
restart: unless-stopped

API Gateway

api-gateway: build: context: ./services/gateway dockerfile: Dockerfile.prod container_name: etsy-gateway environment: - REDIS_URL=redis://redis:6379/0 - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas - JWT_SECRET=${JWT_SECRET} deploy: resources: limits: memory: 512M cpus: '1.0' reservations:

memory: 256M cpus: '0.5' labels: - "traefik.enable=true" - "traefik.http.routers.api-gateway.rule=PathPrefix(/api)" - "traefik.http.routers.api-gateway.tls=true" - "traefik.http.routers.api-gateway.tls.certresolver=letsencrypt" depends_on: - postgres - redis networks: - etsy-network restart: unless-stopped healthcheck: test: ["CMD", "curl", "-f", "http://localhost:8080/health"] interval: 30s timeout: 10s retries: 3

Authentication Service

auth-service: build: context: ./services/auth dockerfile: Dockerfile.prod container_name: etsy-auth environment: - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas - REDIS_URL=redis://redis:6379/1 - JWT_SECRET=${JWT_SECRET} - BCRYPT_ROUNDS=12 deploy: resources: limits: memory: 1G cpus: '1.0' reservations: memory: 512M cpus: '0.5' depends_on: - postgres - redis networks: - etsy-network restart: unless-stopped healthcheck: test: ["CMD", "curl", "-f", "http://localhost:8001/health"] interval: 30s timeout: 10s retries: 3

Etsy Integration Service

etsy-service: build: context: ./services/etsy dockerfile: Dockerfile.prod container_name: etsy-integration environment: - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas - REDIS_URL=redis://redis:6379/2 - ETSY_API_BASE_URL=https://api.etsy.com/v3 - ETSY_RATE_LIMIT_CALLS=10000 - ETSY_RATE_LIMIT_WINDOW=3600 deploy: resources: limits: memory: 1G cpus: '1.0' reservations: memory: 512M cpus: '0.5' depends_on: - postgres - redis networks: - etsy-network restart: unless-stopped healthcheck: test: ["CMD", "curl", "-f", "http://localhost:8002/health"] interval: 30s timeout: 10s retries: 3

Design Management Service

design-service: build: context: ./services/design dockerfile: Dockerfile.prod container_name: etsy-design environment: - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas - MINIO_ENDPOINT=minio:9000 - MINIO_ACCESS_KEY=${MINIO_ACCESS_KEY} - MINIO_SECRET_KEY=${MINIO_SECRET_KEY} - MINIO_SECURE=false deploy: resources: limits: memory: 2G cpus: '2.0' reservations: memory: 1G cpus: '1.0' depends_on: - postgres - minio networks: - etsy-network restart: unless-stopped healthcheck: test: ["CMD", "curl", "-f", "http://localhost:8003/health"] interval: 30s timeout: 10s retries: 3

Analytics Service

analytics-service: build: context: ./services/analytics dockerfile: Dockerfile.prod container_name: etsy-analytics environment: - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas - REDIS_URL=redis://redis:6379/3 deploy: resources: limits: memory: 1G cpus: '1.0' reservations: memory: 512M cpus: '0.5' depends_on: - postgres - redis networks: - etsy-network restart: unless-stopped healthcheck: test: ["CMD", "curl", "-f", "http://localhost:8004/health"] interval: 30s timeout: 10s retries: 3

Job Processing Workers

job-worker: build: context: ./services/jobs dockerfile: Dockerfile.prod container_name: etsy-worker environment: - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@postgres:5432/etsy_saas - REDIS_URL=redis://redis:6379/4 - WORKER_CONCURRENCY=4 - WORKER_MAX_MEMORY=1G deploy: replicas: 2 resources: limits: memory: 1G cpus: '1.0' reservations: memory: 512M cpus: '0.5' depends_on: - redis - postgres networks: - etsy-network restart: unless-stopped

Notification Service

notification-service: build: context: ./services/notifications dockerfile: Dockerfile.prod container_name: etsy-notifications environment: - REDIS_URL=redis://redis:6379/5 deploy: resources: limits: memory: 512M cpus: '0.5' reservations: memory: 256M cpus: '0.25' depends_on: - redis networks: - etsy-network restart: unless-stopped healthcheck: test: ["CMD", "curl", "-f", "http://localhost:8005/health"] interval: 30s timeout: 10s retries: 3

Infrastructure Services

postgres: image: postgres:15 container_name: etsy-postgres environment: - POSTGRES_DB=etsy_saas - POSTGRES_USER=postgres - POSTGRES_PASSWORD=${POSTGRES_PASSWORD} - POSTGRES_SHARED_BUFFERS=512MB - POSTGRES_EFFECTIVE_CACHE_SIZE=2GB - POSTGRES_MAINTENANCE_WORK_MEM=128MB - POSTGRES_WAL_BUFFERS=16MB volumes: - postgres_data:/var/lib/postgresql/data - ./init-scripts:/docker-entrypoint-initdb.d - ./postgres-config:/etc/postgresql/postgresql.conf ports: - "5432:5432" deploy: resources: limits: memory: 3G cpus: '2.0' reservations: memory: 2G cpus: '1.0' networks: - etsy-# Multi-Tenant Etsy Seller Automater Backend Architecture Deep Dive

Executive Summary

This document outlines the backend architecture for transforming the existing Etsy Seller Automater into a scalable multi-tenant SaaS platform, leveraging Python and Go microservices deployed on

QNAP TBS-464 NAS infrastructure.

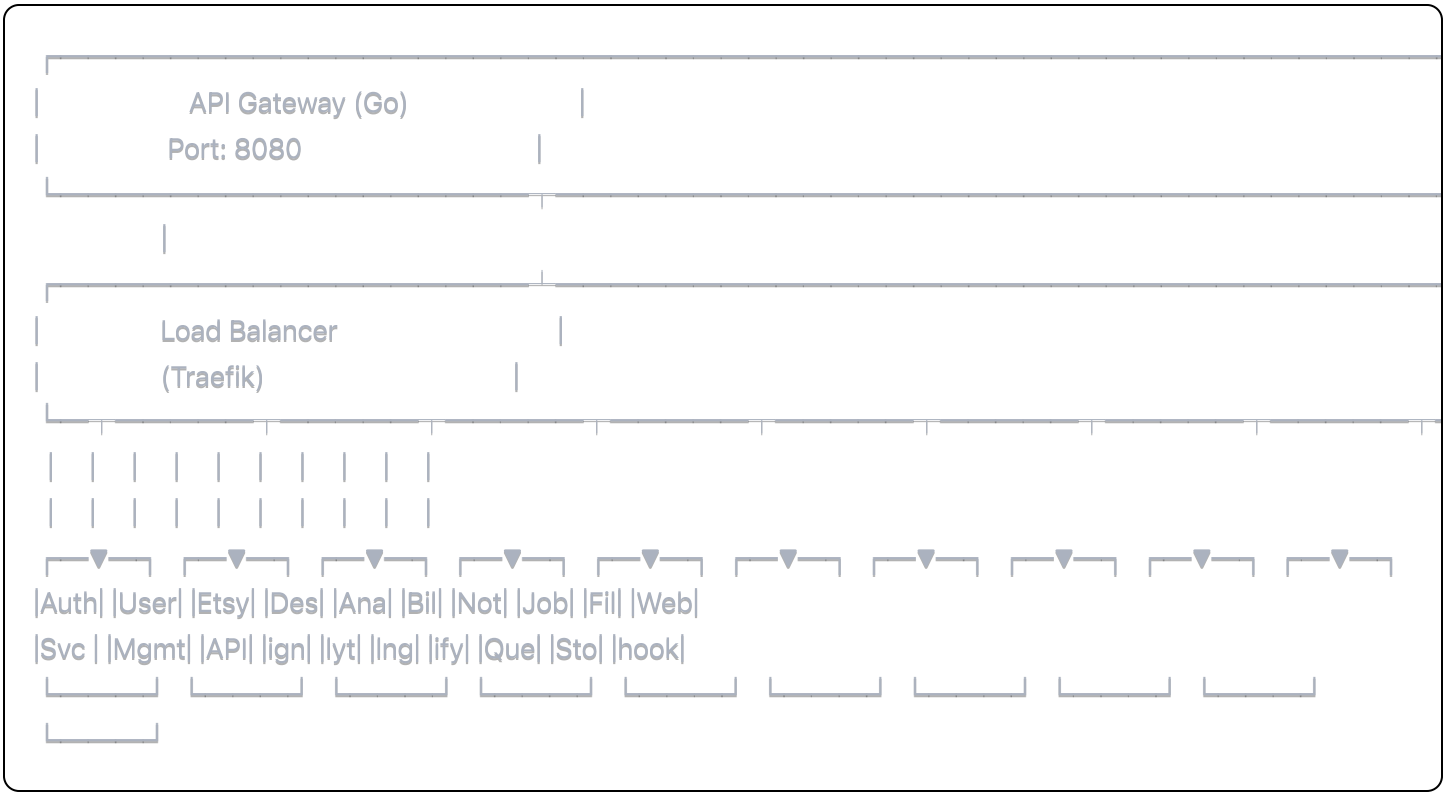
Current State Analysis

Existing Architecture

- **Framework**: FastAPI (Python)
- **Database**: PostgreSQL
- **Authentication**: OAuth 2.0 with PKCE for Etsy API
- **Frontend**: React with Tailwind CSS
- **Deployment**: Docker Compose
- **Key Features**:
  - Etsy shop analytics
  - Design management and mockup creation
  - Listing automation
  - Mask creator for product images

Proposed Multi-Tenant Architecture

1. Service Architecture Overview



2. Core Services Architecture

2.1 API Gateway Service (Go)

Port: 8080

Responsibilities: Request routing, rate limiting, authentication middleware, tenant isolation

```go
```

```go
// pkg/gateway/main.go
package main

import (
    "context"
    "log"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/redis/go-redis/v9"
)

type Gateway struct {
    router     *gin.Engine
    redisClient *redis.Client
    services   map[string]string
}

type TenantMiddleware struct {
    redis *redis.Client
}

func (tm *TenantMiddleware) ExtractTenant() gin.HandlerFunc {
    return func(c *gin.Context) {
        host := c.Request.Host
        subdomain := extractSubdomain(host)

        // Validate tenant exists
        tenantID, err := tm.redis.Get(c.Request.Context(),
            fmt.Sprintf("tenant:%s", subdomain)).Result()
        if err != nil {
            c.JSON(404, gin.H{"error": "Tenant not found"})
            c.Abort()
            return
        }

        c.Set("tenant_id", tenantID)
        c.Set("subdomain", subdomain)
        c.Next()
    }
}
```

```
func (g *Gateway) setupRoutes() {
    v1 := g.router.Group("/api/v1")
    v1.Use(g.tenantMiddleware.ExtractTenant())

    // Route to microservices
    v1.Any("/auth/*path", g.proxyToService("auth-service"))
    v1.Any("/etsy/*path", g.proxyToService("etsy-service"))
    v1.Any("/designs/*path", g.proxyToService("design-service"))
    v1.Any("/analytics/*path", g.proxyToService("analytics-service"))
}
```

2.2 Authentication & User Management Service (Python/FastAPI)

Port: 8001

Database: `tenant_users` schema in PostgreSQL

```python
```

```python
# services/auth/main.py
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
import jwt
from datetime import datetime, timedelta

app = FastAPI(title="Authentication Service")

class AuthService:
    def __init__(self, db: Session):
        self.db = db

    async def authenticate_tenant_user(self,
                                       tenant_id: str,
                                       email: str,
                                       password: str) -> dict:
        """Authenticate user within specific tenant context"""
        user = self.db.query(TenantUser).filter(
            TenantUser.tenant_id == tenant_id,
            TenantUser.email == email,
            TenantUser.is_active == True
        ).first()

        if not user or not verify_password(password, user.hashed_password):
            raise HTTPException(401, "Invalid credentials")

        # Generate tenant-scoped JWT
        payload = {
            "user_id": user.id,
            "tenant_id": tenant_id,
            "permissions": user.permissions,
            "exp": datetime.utcnow() + timedelta(hours=24)
        }

        token = jwt.encode(payload, JWT_SECRET, algorithm="HS256")

        return {
            "access_token": token,
            "token_type": "bearer",
            "user": user.to_dict(),
            "tenant": user.tenant.to_dict()
        }
```

```python
# Database Models
class Tenant(Base):
    __tablename__ = "tenants"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    subdomain = Column(String(63), unique=True, nullable=False)
    company_name = Column(String(255), nullable=False)
    subscription_tier = Column(String(50), default="basic")
    created_at = Column(DateTime, default=datetime.utcnow)

    # Etsy Integration
    etsy_shop_id = Column(String(100))
    etsy_access_token = Column(Text)
    etsy_refresh_token = Column(Text)

class TenantUser(Base):
    __tablename__ = "tenant_users"

    id = Column(UUID(as_uuid=True), primary_key=True, default=uuid.uuid4)
    tenant_id = Column(UUID(as_uuid=True), ForeignKey("tenants.id"))
    email = Column(String(255), nullable=False)
    hashed_password = Column(String(255), nullable=False)
    permissions = Column(JSON)  # Role-based permissions
    is_active = Column(Boolean, default=True)
```

2.3 Etsy Integration Service (Python/FastAPI)

Port: 8002

Focus: OAuth flows, API calls, rate limiting per tenant

```python
```

```python
# services/etsy/main.py
import asyncio
import aiohttp
from fastapi import FastAPI, BackgroundTasks
from rq import Queue
import redis

app = FastAPI(title="Etsy Integration Service")

class EtsyService:
    def __init__(self, tenant_id: str):
        self.tenant_id = tenant_id
        self.base_url = "https://api.etsy.com/v3"

    async def get_shop_analytics(self, date_range: str) -> dict:
        """Get shop analytics for tenant with rate limiting"""

        # Implement tenant-specific rate limiting
        rate_limit_key = f"etsy_rate_limit:{self.tenant_id}"

        async with aiohttp.ClientSession() as session:
            headers = await self._get_auth_headers()

            # Get receipt data
            receipts = await self._fetch_receipts(session, headers, date_range)

            # Process analytics
            analytics = await self._process_analytics_data(receipts)

            return {
                "tenant_id": self.tenant_id,
                "date_range": date_range,
                "total_revenue": analytics["revenue"],
                "total_orders": analytics["orders"],
                "top_products": analytics["top_products"],
                "monthly_breakdown": analytics["monthly"]
            }

    async def create_listing(self, listing_data: dict) -> dict:
        """Create Etsy listing with tenant isolation"""

        # Validate listing data against tenant's subscription limits
        await self._validate_listing_limits()
```

```
        # Create listing via Etsy API
        async with aiohttp.ClientSession() as session:
            headers = await self._get_auth_headers()

            response = await session.post(
                f"{self.base_url}/application/shops/{self.shop_id}/listings",
                headers=headers,
                json=listing_data
            )

            result = await response.json()

            # Store listing in tenant database
            await self._store_listing_record(result)

            return result

# Background job for bulk operations
@app.post("/bulk-listing-creation")
async def bulk_create_listings(
    listings: List[dict],
    tenant_id: str,
    background_tasks: BackgroundTasks
):
    """Queue bulk listing creation as background job"""

    redis_conn = redis.Redis(host='redis', port=6379, db=0)
    queue = Queue('bulk_operations', connection=redis_conn)

    job = queue.enqueue(
        'tasks.bulk_create_listings',
        listings,
        tenant_id,
        job_timeout='30m'
    )

    return {"job_id": job.id, "status": "queued"}
```

2.4 Design Management Service (Go)

Port: 8003

Focus: File storage, image processing, mockup generation

go

```go
// services/design/main.go
package main

import (
    "context"
    "fmt"
    "io"
    "path/filepath"

    "github.com/gin-gonic/gin"
    "github.com/minio/minio-go/v7"
    "gorm.io/gorm"
)

type DesignService struct {
    db          *gorm.DB
    minioClient *minio.Client
    router      *gin.Engine
}

type Design struct {
    ID          uint   `gorm:"primaryKey"`
    TenantID    string `gorm:"index"`
    Name        string
    FilePath    string
    FileSize    int64
    ContentType string
    Tags        []string `gorm:"serializer:json"`
    CreatedAt   time.Time
}

func (ds *DesignService) uploadDesign(c *gin.Context) {
    tenantID := c.GetString("tenant_id")

    file, header, err := c.Request.FormFile("design")
    if err != nil {
        c.JSON(400, gin.H{"error": "Invalid file"})
        return
    }
    defer file.Close()

    // Generate tenant-scoped file path
    fileName := fmt.Sprintf("%s/%s/%s",
```

```
        tenantID, "designs", header.Filename)

    // Upload to MinIO
    _, err = ds.minioClient.PutObject(
        context.Background(),
        "tenant-designs",
        fileName,
        file,
        header.Size,
        minio.PutObjectOptions{
            ContentType: header.Header.Get("Content-Type"),
        },
    )

    if err != nil {
        c.JSON(500, gin.H{"error": "Upload failed"})
        return
    }

    // Save metadata to database
    design := Design{
        TenantID:    tenantID,
        Name:        header.Filename,
        FilePath:    fileName,
        FileSize:    header.Size,
        ContentType: header.Header.Get("Content-Type"),
    }

    ds.db.Create(&design)

    c.JSON(201, design)
}

func (ds *DesignService) generateMockup(c *gin.Context) {
    tenantID := c.GetString("tenant_id")
    designID := c.Param("id")

    // Get design from database
    var design Design
    result := ds.db.Where("tenant_id = ? AND id = ?",
        tenantID, designID).First(&design)

    if result.Error != nil {
        c.JSON(404, gin.H{"error": "Design not found"})
```

```
        return
    }

    // Queue mockup generation
    job := MockupGenerationJob{
        TenantID: tenantID,
        DesignID: designID,
        FilePath: design.FilePath,
    }

    // Submit to job queue (Redis/RQ)
    jobID, err := ds.queueMockupJob(job)
    if err != nil {
        c.JSON(500, gin.H{"error": "Failed to queue job"})
        return
    }

    c.JSON(202, gin.H{
        "job_id": jobID,
        "status": "queued",
        "message": "Mockup generation started"
    })
}
```

2.5 Analytics Service (Python/FastAPI)

Port: 8004

Focus: Data aggregation, reporting, business intelligence

```
python
```

```python
# services/analytics/main.py
from fastapi import FastAPI, Depends
from sqlalchemy import text
import pandas as pd
from datetime import datetime, timedelta

app = FastAPI(title="Analytics Service")

class AnalyticsService:
    def __init__(self, tenant_id: str, db: Session):
        self.tenant_id = tenant_id
        self.db = db

    async def get_sales_analytics(self,
                                  date_from: datetime,
                                  date_to: datetime) -> dict:
        """Generate comprehensive sales analytics for tenant"""

        # Query tenant-specific sales data
        query = text("""
        SELECT
            DATE(created_at) as sale_date,
            SUM(total_amount) as daily_revenue,
            COUNT(*) as daily_orders,
            AVG(total_amount) as avg_order_value
        FROM tenant_orders
        WHERE tenant_id = :tenant_id
        AND created_at BETWEEN :date_from AND :date_to
        GROUP BY DATE(created_at)
        ORDER BY sale_date
        """)

        result = self.db.execute(query, {
            "tenant_id": self.tenant_id,
            "date_from": date_from,
            "date_to": date_to
        })

        # Convert to pandas for analysis
        df = pd.DataFrame(result.fetchall(),
                          columns=['sale_date', 'daily_revenue',
                                   'daily_orders', 'avg_order_value'])
```

```
    analytics = {
        "summary": {
            "total_revenue": df['daily_revenue'].sum(),
            "total_orders": df['daily_orders'].sum(),
            "avg_daily_revenue": df['daily_revenue'].mean(),
            "revenue_growth": self._calculate_growth_rate(df)
        },
        "daily_breakdown": df.to_dict('records'),
        "top_performing_days": df.nlargest(5, 'daily_revenue').to_dict('records')
    }

    return analytics

async def get_product_performance(self) -> dict:
    """Analyze product performance metrics"""

    query = text("""
    SELECT
        p.name,
        p.etsy_listing_id,
        COUNT(oi.id) as total_sales,
        SUM(oi.quantity) as units_sold,
        SUM(oi.price * oi.quantity) as total_revenue,
        AVG(oi.price) as avg_price
    FROM products p
    LEFT JOIN order_items oi ON p.id = oi.product_id
    LEFT JOIN tenant_orders o ON oi.order_id = o.id
    WHERE o.tenant_id = :tenant_id
    GROUP BY p.id, p.name, p.etsy_listing_id
    ORDER BY total_revenue DESC
    """)

    result = self.db.execute(query, {"tenant_id": self.tenant_id})

    products = []
    for row in result:
        products.append({
            "name": row.name,
            "etsy_listing_id": row.etsy_listing_id,
            "total_sales": row.total_sales,
            "units_sold": row.units_sold,
            "total_revenue": float(row.total_revenue or 0),
            "avg_price": float(row.avg_price or 0)
        })
```

```
        return {
            "top_products": products[:10],
            "total_products": len(products),
            "products_with_sales": len([p for p in products if p['total_sales'] > 0])
        }

@app.get("/analytics/dashboard")
async def get_dashboard_data(
    tenant_id: str = Depends(get_tenant_id),
    date_range: str = "30d"
):
    """Get comprehensive dashboard analytics"""

    analytics_service = AnalyticsService(tenant_id, get_db())

    # Calculate date range
    end_date = datetime.utcnow()
    days = int(date_range.replace('d', ''))
    start_date = end_date - timedelta(days=days)

    sales_data = await analytics_service.get_sales_analytics(start_date, end_date)
    product_data = await analytics_service.get_product_performance()

    return {
        "tenant_id": tenant_id,
        "date_range": date_range,
        "sales": sales_data,
        "products": product_data,
        "generated_at": datetime.utcnow().isoformat()
    }
```

3. Database Architecture

3.1 Multi-Tenant Schema Design (PostgreSQL)

sql

```sql
-- Core tenant management
CREATE SCHEMA IF NOT EXISTS core;

CREATE TABLE core.tenants (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    subdomain VARCHAR(63) UNIQUE NOT NULL,
    company_name VARCHAR(255) NOT NULL,
    subscription_tier VARCHAR(50) DEFAULT 'basic',
    database_schema VARCHAR(63) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE core.tenant_users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    tenant_id UUID REFERENCES core.tenants(id),
    email VARCHAR(255) NOT NULL,
    hashed_password VARCHAR(255) NOT NULL,
    role VARCHAR(50) DEFAULT 'user',
    permissions JSONB,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Function to create tenant schema
CREATE OR REPLACE FUNCTION create_tenant_schema(tenant_id UUID, schema_name VARCHAR)
RETURNS VOID AS $$
BEGIN
    -- Create schema
    EXECUTE format('CREATE SCHEMA IF NOT EXISTS %I', schema_name);

    -- Create tenant-specific tables
    EXECUTE format('
        CREATE TABLE %I.orders (
            id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
            etsy_receipt_id BIGINT,
            total_amount DECIMAL(10,2),
            created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        )', schema_name);

    EXECUTE format('
        CREATE TABLE %I.products (
            id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
        name VARCHAR(255) NOT NULL,
        etsy_listing_id BIGINT,
        price DECIMAL(10,2),
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )', schema_name);

    EXECUTE format('
      CREATE TABLE %I.designs (
        id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
        name VARCHAR(255) NOT NULL,
        file_path VARCHAR(500),
        file_size BIGINT,
        tags TEXT[],
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )', schema_name);
END;
$$ LANGUAGE plpgsql;
```

4. Container Orchestration (Docker Compose)

yaml

```yaml
# docker-compose.yml
version: '3.8'

services:
  # API Gateway
  api-gateway:
    build:
      context: ./services/gateway
      dockerfile: Dockerfile
    ports:
      - "8080:8080"
    environment:
      - REDIS_URL=redis://redis:6379
    depends_on:
      - redis
    networks:
      - etsy-network

  # Authentication Service
  auth-service:
    build:
      context: ./services/auth
      dockerfile: Dockerfile
    ports:
      - "8001:8001"
    environment:
      - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
      - JWT_SECRET=your-jwt-secret-here
      - REDIS_URL=redis://redis:6379
    depends_on:
      - postgres
      - redis
    networks:
      - etsy-network

  # Etsy Integration Service
  etsy-service:
    build:
      context: ./services/etsy
      dockerfile: Dockerfile
    ports:
      - "8002:8002"
    environment:
```

```
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
    - REDIS_URL=redis://redis:6379
    - ETSY_API_BASE_URL=https://api.etsy.com/v3
  depends_on:
    - postgres
    - redis
  networks:
    - etsy-network

# Design Management Service (Go)
design-service:
  build:
    context: ./services/design
    dockerfile: Dockerfile
  ports:
    - "8003:8003"
  environment:
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
    - MINIO_ENDPOINT=minio:9000
    - MINIO_ACCESS_KEY=minioadmin
    - MINIO_SECRET_KEY=minioadmin
  depends_on:
    - postgres
    - minio
  networks:
    - etsy-network

# Analytics Service
analytics-service:
  build:
    context: ./services/analytics
    dockerfile: Dockerfile
  ports:
    - "8004:8004"
  environment:
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
  depends_on:
    - postgres
  networks:
    - etsy-network

# Job Queue Worker
job-worker:
  build:
```

```yaml
    context: ./services/jobs
    dockerfile: Dockerfile
  environment:
    - REDIS_URL=redis://redis:6379
    - DATABASE_URL=postgresql://postgres:password@postgres:5432/etsy_saas
  depends_on:
    - redis
    - postgres
  networks:
    - etsy-network

# Infrastructure Services
postgres:
  image: postgres:15
  environment:
    - POSTGRES_DB=etsy_saas
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=password
  volumes:
    - postgres_data:/var/lib/postgresql/data
    - ./init.sql:/docker-entrypoint-initdb.d/init.sql
  ports:
    - "5432:5432"
  networks:
    - etsy-network

redis:
  image: redis:7-alpine
  ports:
    - "6379:6379"
  volumes:
    - redis_data:/data
  networks:
    - etsy-network

minio:
  image: minio/minio
  ports:
    - "9000:9000"
    - "9001:9001"
  volumes:
    - minio_data:/data
  environment:
    - MINIO_ROOT_USER=minioadmin
```

```
      - MINIO_ROOT_PASSWORD=minioadmin
    command: server /data --console-address ":9001"
    networks:
      - etsy-network

  # Reverse Proxy
  traefik:
    image: traefik:v2.10
    command:
      - --providers.docker=true
      - --providers.docker.exposedbydefault=false
      - --entrypoints.web.address=:80
      - --entrypoints.websecure.address=:443
    ports:
      - "80:80"
      - "443:443"
      - "8080:8080"  # Dashboard
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    networks:
      - etsy-network

volumes:
  postgres_data:
  redis_data:
  minio_data:

networks:
  etsy-network:
    driver: bridge
```

5. Deployment on QNAP NAS

5.1 QNAP-Specific Configuration

bash

```
#!/bin/bash
# deploy-qnap.sh

# Install Container Station if not already installed
# Access QNAP Web UI -> App Center -> Container Station

# Create project directory
ssh admin@qnap-ip "mkdir -p /share/Container/etsy-saas"

# Copy docker-compose files
scp docker-compose.yml admin@qnap-ip:/share/Container/etsy-saas/
scp docker-compose.override.yml admin@qnap-ip:/share/Container/etsy-saas/

# Create persistent volumes on QNAP
ssh admin@qnap-ip "mkdir -p /share/Container/volumes/{postgres,redis,minio}"

# Set proper permissions
ssh admin@qnap-ip "chmod -R 755 /share/Container/etsy-saas"

# Start services
ssh admin@qnap-ip "cd /share/Container/etsy-saas && docker-compose up -d"
```

5.2 QNAP Resource Optimization

yaml

```yaml
# docker-compose.override.yml - QNAP specific optimizations
version: '3.8'

services:
  api-gateway:
    deploy:
      resources:
        limits:
          memory: 256M
          cpus: '0.5'

  auth-service:
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'

  etsy-service:
    deploy:
      resources:
        limits:
          memory: 512M
          cpus: '0.5'

  design-service:
    deploy:
      resources:
        limits:
          memory: 1G
          cpus: '1.0'

  postgres:
    deploy:
      resources:
        limits:
          memory: 2G
          cpus: '1.0'
    environment:
      - POSTGRES_SHARED_BUFFERS=256MB
      - POSTGRES_EFFECTIVE_CACHE_SIZE=1GB
```

6. Monitoring and Observability

6.1 Health Checks

```python
```

```python
# services/common/health.py
from fastapi import FastAPI
import asyncio
import psycopg2
import redis

class HealthCheck:
    def __init__(self, app: FastAPI):
        self.app = app
        self.setup_routes()

    def setup_routes(self):
        @self.app.get("/health")
        async def health_check():
            checks = {
                "database": await self.check_database(),
                "redis": await self.check_redis(),
                "service": "healthy"
            }

            all_healthy = all(
                status == "healthy"
                for status in checks.values()
            )

            return {
                "status": "healthy" if all_healthy else "unhealthy",
                "checks": checks
            }

    async def check_database(self) -> str:
        try:
            conn = psycopg2.connect(DATABASE_URL)
            conn.close()
            return "healthy"
        except Exception as e:
            return f"unhealthy: {str(e)}"

    async def check_redis(self) -> str:
        try:
            r = redis.Redis.from_url(REDIS_URL)
            r.ping()
            return "healthy"
```

```
        except Exception as e:
            return f"unhealthy: {str(e)}"
```

7. Security Considerations

7.1 Tenant Isolation Security

python

```python
# services/common/security.py
from functools import wraps
import jwt
from fastapi import HTTPException

def require_tenant_access(required_permissions: List[str] = None):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            # Extract JWT token
            token = extract_token_from_request()

            try:
                payload = jwt.decode(token, JWT_SECRET, algorithms=["HS256"])
                tenant_id = payload.get("tenant_id")
                user_permissions = payload.get("permissions", [])

                # Verify tenant access
                if not tenant_id:
                    raise HTTPException(403, "No tenant access")

                # Check specific permissions if required
                if required_permissions:
                    if not any(perm in user_permissions for perm in required_permissions):
                        raise HTTPException(403, "Insufficient permissions")

                # Add to request context
                kwargs['tenant_id'] = tenant_id
                kwargs['user_permissions'] = user_permissions

            except jwt.InvalidTokenError:
                raise HTTPException(401, "Invalid token")

            return await func(*args, **kwargs)
        return wrapper
    return decorator

# Usage example
@app.post("/designs/upload")
@require_tenant_access(["design:create"])
async def upload_design(tenant_id: str, user_permissions: List[str], file: UploadFile = File(...)):
    """Upload design with tenant isolation"""
```

```
    # Process upload with tenant context
    pass
```

8. Background Job Processing Architecture

8.1 Job Queue System (Python/RQ + Redis)

```python
```

```python
# services/jobs/worker.py
from rq import Worker, Queue, Connection
import redis
from typing import Dict, Any
import logging

# Job definitions
def process_bulk_listings(tenant_id: str, listings_data: List[Dict]) -> Dict[str, Any]:
    """Process bulk listing creation for a tenant"""
    results = {
        "tenant_id": tenant_id,
        "total_listings": len(listings_data),
        "successful": 0,
        "failed": 0,
        "errors": []
    }

    etsy_service = EtsyService(tenant_id)

    for listing_data in listings_data:
        try:
            result = await etsy_service.create_listing(listing_data)
            results["successful"] += 1

            # Store success in tenant database
            await store_listing_result(tenant_id, listing_data["id"], result)

        except Exception as e:
            results["failed"] += 1
            results["errors"].append({
                "listing_id": listing_data.get("id"),
                "error": str(e)
            })

            # Log error for tenant
            await log_tenant_error(tenant_id, "listing_creation", str(e))

    return results

def generate_mockup_images(tenant_id: str, design_id: str, mockup_config: Dict) -> Dict[str, Any]:
    """Generate mockup images for a design"""
    import PIL.Image
    import numpy as np
```

```
try:
    # Load design from MinIO
    design_service = DesignService(tenant_id)
    design_data = design_service.get_design(design_id)

    # Load base mockup templates
    mockup_templates = load_mockup_templates(mockup_config["template_type"])

    generated_mockups = []

    for template in mockup_templates:
        # Apply design to template
        mockup_image = apply_design_to_template(
            design_data["image_data"],
            template,
            mockup_config.get("mask_data", {})
        )

        # Save mockup to MinIO
        mockup_path = f"{tenant_id}/mockups/{design_id}_{template['name']}.png"
        mockup_url = design_service.save_mockup(mockup_path, mockup_image)

        generated_mockups.append({
            "template_name": template["name"],
            "mockup_url": mockup_url,
            "size": mockup_image.size
        })

    return {
        "tenant_id": tenant_id,
        "design_id": design_id,
        "generated_mockups": generated_mockups,
        "status": "completed"
    }

except Exception as e:
    logging.error(f"Mockup generation failed for {tenant_id}:{design_id}: {e}")
    return {
        "tenant_id": tenant_id,
        "design_id": design_id,
        "status": "failed",
        "error": str(e)
    }
```

```python
def sync_etsy_data(tenant_id: str) -> Dict[str, Any]:
    """Sync Etsy data for tenant (orders, listings, etc.)"""
    etsy_service = EtsyService(tenant_id)

    # Get latest sync timestamp
    last_sync = get_last_sync_timestamp(tenant_id)

    sync_results = {
        "tenant_id": tenant_id,
        "sync_started": datetime.utcnow().isoformat(),
        "orders_synced": 0,
        "listings_synced": 0,
        "errors": []
    }

    try:
        # Sync orders
        new_orders = await etsy_service.get_orders_since(last_sync)
        for order in new_orders:
            await store_tenant_order(tenant_id, order)
            sync_results["orders_synced"] += 1

        # Sync listings
        updated_listings = await etsy_service.get_listings_updates_since(last_sync)
        for listing in updated_listings:
            await store_tenant_listing(tenant_id, listing)
            sync_results["listings_synced"] += 1

        # Update sync timestamp
        await update_sync_timestamp(tenant_id, datetime.utcnow())

        sync_results["status"] = "completed"
        sync_results["sync_completed"] = datetime.utcnow().isoformat()

    except Exception as e:
        sync_results["status"] = "failed"
        sync_results["errors"].append(str(e))
        logging.error(f"Etsy sync failed for {tenant_id}: {e}")

    return sync_results

# Worker setup
if __name__ == '__main__':
```

```
redis_conn = redis.from_url(REDIS_URL)

# Create queues with priorities
high_priority_queue = Queue('high_priority', connection=redis_conn)
normal_queue = Queue('normal', connection=redis_conn)
low_priority_queue = Queue('low_priority', connection=redis_conn)

# Start worker
worker = Worker([high_priority_queue, normal_queue, low_priority_queue],
            connection=redis_conn)
worker.work()
```

8.2 Job Management API (Go)

go

```go
// services/jobs/main.go
package main

import (
    "encoding/json"
    "net/http"
    "time"

    "github.com/gin-gonic/gin"
    "github.com/go-redis/redis/v8"
)

type JobService struct {
    redis  *redis.Client
    router *gin.Engine
}

type Job struct {
    ID          string                 `json:"id"`
    TenantID    string                 `json:"tenant_id"`
    Type        string                 `json:"type"`
    Status      string                 `json:"status"`
    Payload     map[string]interface{} `json:"payload"`
    Result      map[string]interface{} `json:"result,omitempty"`
    CreatedAt   time.Time              `json:"created_at"`
    CompletedAt *time.Time             `json:"completed_at,omitempty"`
    Error       string                 `json:"error,omitempty"`
}

func (js *JobService) queueJob(c *gin.Context) {
    tenantID := c.GetString("tenant_id")

    var jobRequest struct {
        Type     string                 `json:"type" binding:"required"`
        Payload  map[string]interface{} `json:"payload" binding:"required"`
        Priority string                 `json:"priority"` // high, normal, low
    }

    if err := c.ShouldBindJSON(&jobRequest); err != nil {
        c.JSON(400, gin.H{"error": err.Error()})
        return
    }
```

```
    // Generate job ID
    jobID := generateJobID()

    job := Job{
        ID:       jobID,
        TenantID:  tenantID,
        Type:     jobRequest.Type,
        Status:   "queued",
        Payload:  jobRequest.Payload,
        CreatedAt: time.Now(),
    }

    // Store job metadata
    jobData, _ := json.Marshal(job)
    js.redis.Set(c.Request.Context(),
        fmt.Sprintf("job:%s", jobID), jobData, 24*time.Hour)

    // Queue job based on type and priority
    queueName := js.getQueueForJob(jobRequest.Type, jobRequest.Priority)

    jobPayload := map[string]interface{}{
        "job_id":   jobID,
        "tenant_id": tenantID,
        "type":     jobRequest.Type,
        "payload":  jobRequest.Payload,
    }

    payloadJSON, _ := json.Marshal(jobPayload)

    err := js.redis.LPush(c.Request.Context(), queueName, payloadJSON).Err()
    if err != nil {
        c.JSON(500, gin.H{"error": "Failed to queue job"})
        return
    }

    c.JSON(202, gin.H{
        "job_id": jobID,
        "status": "queued",
        "message": fmt.Sprintf("Job queued in %s", queueName)
    })
}

func (js *JobService) getJobStatus(c *gin.Context) {
    tenantID := c.GetString("tenant_id")
```

```
    jobID := c.Param("job_id")

    // Get job data
    jobData, err := js.redis.Get(c.Request.Context(),
        fmt.Sprintf("job:%s", jobID)).Result()
    if err == redis.Nil {
        c.JSON(404, gin.H{"error": "Job not found"})
        return
    }

    var job Job
    json.Unmarshal([]byte(jobData), &job)

    // Verify tenant access
    if job.TenantID != tenantID {
        c.JSON(403, gin.H{"error": "Access denied"})
        return
    }

    c.JSON(200, job)
}

func (js *JobService) listTenantJobs(c *gin.Context) {
    tenantID := c.GetString("tenant_id")

    // Get all job keys for tenant
    pattern := fmt.Sprintf("job:*")
    keys, err := js.redis.Keys(c.Request.Context(), pattern).Result()
    if err != nil {
        c.JSON(500, gin.H{"error": "Failed to fetch jobs"})
        return
    }

    var jobs []Job
    for _, key := range keys {
        jobData, err := js.redis.Get(c.Request.Context(), key).Result()
        if err != nil {
            continue
        }

        var job Job
        if err := json.Unmarshal([]byte(jobData), &job); err != nil {
            continue
        }
```

```
        // Filter by tenant
        if job.TenantID == tenantID {
            jobs = append(jobs, job)
        }
    }

    c.JSON(200, gin.H{
        "jobs": jobs,
        "total": len(jobs)
    })
}
```

9. Advanced Features Implementation

9.1 Real-time Notifications Service (Go + WebSockets)

```
go
```

```go
// services/notifications/main.go
package main

import (
    "encoding/json"
    "log"
    "net/http"

    "github.com/gin-gonic/gin"
    "github.com/gorilla/websocket"
)

type NotificationService struct {
    clients    map[string]map[*websocket.Conn]bool // tenant_id -> connections
    broadcast  chan NotificationMessage
    register   chan *Client
    unregister chan *Client
    router     *gin.Engine
}

type Client struct {
    TenantID string
    Conn     *websocket.Conn
    Send     chan NotificationMessage
}

type NotificationMessage struct {
    TenantID  string      `json:"tenant_id"`
    Type      string      `json:"type"`
    Title     string      `json:"title"`
    Message   string      `json:"message"`
    Data      interface{} `json:"data,omitempty"`
    Timestamp int64       `json:"timestamp"`
}

var upgrader = websocket.Upgrader{
    CheckOrigin: func(r *http.Request) bool {
        return true // Configure proper origin checking in production
    },
}

func (ns *NotificationService) handleWebSocket(c *gin.Context) {
    tenantID := c.GetString("tenant_id")
```

```
    conn, err := upgrader.Upgrade(c.Writer, c.Request, nil)
    if err != nil {
        log.Printf("WebSocket upgrade failed: %v", err)
        return
    }

    client := &Client{
        TenantID: tenantID,
        Conn:     conn,
        Send:     make(chan NotificationMessage, 256),
    }

    ns.register <- client

    go client.writePump()
    go client.readPump(ns.unregister)
}

func (c *Client) writePump() {
    defer c.Conn.Close()

    for {
        select {
        case message, ok := <-c.Send:
            if !ok {
                c.Conn.WriteMessage(websocket.CloseMessage, []byte{})
                return
            }

            if err := c.Conn.WriteJSON(message); err != nil {
                log.Printf("WebSocket write error: %v", err)
                return
            }
        }
    }
}

func (ns *NotificationService) run() {
    for {
        select {
        case client := <-ns.register:
            if ns.clients[client.TenantID] == nil {
                ns.clients[client.TenantID] = make(map[*websocket.Conn]bool)
```

```
            }
            ns.clients[client.TenantID][client.Conn] = true
            log.Printf("Client connected for tenant: %s", client.TenantID)

        case client := <-ns.unregister:
            if clients, ok := ns.clients[client.TenantID]; ok {
                if _, ok := clients[client.Conn]; ok {
                    delete(clients, client.Conn)
                    close(client.Send)
                    client.Conn.Close()
                }
            }

        case message := <-ns.broadcast:
            if clients, ok := ns.clients[message.TenantID]; ok {
                for conn := range clients {
                    select {
                    case conn.WriteJSON(message):
                    default:
                        close(conn.Send)
                        delete(clients, conn)
                        conn.Close()
                    }
                }
            }
        }
    }
}

// Notification triggers
func (ns *NotificationService) sendJobCompletion(tenantID, jobID, jobType string, result interface{}) {
    message := NotificationMessage{
        TenantID:  tenantID,
        Type:      "job_completed",
        Title:     "Job Completed",
        Message:   fmt.Sprintf("%s job completed successfully", jobType),
        Data:      map[string]interface{}{"job_id": jobID, "result": result},
        Timestamp: time.Now().Unix(),
    }

    ns.broadcast <- message
}
```

9.2 Billing and Subscription Management (Python)

```python
```

```python
# services/billing/main.py
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from datetime import datetime, timedelta
from typing import Dict, Any
import stripe

app = FastAPI(title="Billing Service")

# Subscription tiers configuration
SUBSCRIPTION_TIERS = {
    "starter": {
        "name": "Starter",
        "monthly_price": 29,
        "limits": {
            "max_listings": 100,
            "max_designs": 50,
            "max_mockups_per_month": 500,
            "api_calls_per_hour": 100
        }
    },
    "professional": {
        "name": "Professional",
        "monthly_price": 79,
        "limits": {
            "max_listings": 1000,
            "max_designs": 500,
            "max_mockups_per_month": 5000,
            "api_calls_per_hour": 500
        }
    },
    "enterprise": {
        "name": "Enterprise",
        "monthly_price": 199,
        "limits": {
            "max_listings": -1,  # Unlimited
            "max_designs": -1,   # Unlimited
            "max_mockups_per_month": -1,  # Unlimited
            "api_calls_per_hour": 2000
        }
    }
}
```

```python
class BillingService:
    def __init__(self, tenant_id: str, db: Session):
        self.tenant_id = tenant_id
        self.db = db

    async def get_usage_stats(self) -> Dict[str, Any]:
        """Get current usage statistics for tenant"""

        # Query current month usage
        current_month = datetime.now().replace(day=1, hour=0, minute=0, second=0)

        usage_stats = {
            "current_listings": await self._count_active_listings(),
            "total_designs": await self._count_designs(),
            "mockups_this_month": await self._count_mockups_since(current_month),
            "api_calls_this_hour": await self._count_api_calls_last_hour()
        }

        # Get subscription limits
        subscription = await self._get_tenant_subscription()
        tier_limits = SUBSCRIPTION_TIERS[subscription.tier]["limits"]

        # Calculate usage percentages
        usage_percentages = {}
        for metric, current_value in usage_stats.items():
            limit_key = f"max_{metric}" if metric != "api_calls_this_hour" else "api_calls_per_hour"
            limit = tier_limits.get(limit_key, -1)

            if limit == -1:  # Unlimited
                usage_percentages[metric] = 0
            else:
                usage_percentages[metric] = (current_value / limit) * 100

        return {
            "tenant_id": self.tenant_id,
            "subscription_tier": subscription.tier,
            "usage": usage_stats,
            "limits": tier_limits,
            "usage_percentages": usage_percentages,
            "billing_period": {
                "start": subscription.current_period_start,
                "end": subscription.current_period_end
            }
        }
```

```python
async def check_usage_limits(self, resource_type: str, requested_amount: int = 1) -> bool:
    """Check if tenant can use more resources"""

    usage_stats = await self.get_usage_stats()
    current_usage = usage_stats["usage"]
    limits = usage_stats["limits"]

    resource_mapping = {
        "listings": ("current_listings", "max_listings"),
        "designs": ("total_designs", "max_designs"),
        "mockups": ("mockups_this_month", "max_mockups_per_month"),
        "api_calls": ("api_calls_this_hour", "api_calls_per_hour")
    }

    if resource_type not in resource_mapping:
        raise ValueError(f"Unknown resource type: {resource_type}")

    usage_key, limit_key = resource_mapping[resource_type]
    current_value = current_usage[usage_key]
    limit = limits[limit_key]

    if limit == -1:  # Unlimited
        return True

    return (current_value + requested_amount) <= limit

async def create_stripe_checkout_session(self,
                                         new_tier: str,
                                         success_url: str,
                                         cancel_url: str) -> str:
    """Create Stripe checkout session for subscription upgrade/downgrade"""

    if new_tier not in SUBSCRIPTION_TIERS:
        raise HTTPException(400, f"Invalid subscription tier: {new_tier}")

    tenant = await self._get_tenant()
    price_amount = SUBSCRIPTION_TIERS[new_tier]["monthly_price"] * 100  # Stripe uses cents

    checkout_session = stripe.checkout.Session.create(
        customer_email=tenant.primary_email,
        payment_method_types=['card'],
        line_items=[{
            'price_data': {
```

```
                'currency': 'usd',
                'unit_amount': price_amount,
                'product_data': {
                    'name': f'Etsy Seller Automater - {SUBSCRIPTION_TIERS[new_tier]["name"]}',
                    'description': f'Monthly subscription to {new_tier} tier'
                },
                'recurring': {
                    'interval': 'month'
                }
            },
            'quantity': 1,
        }],
        mode='subscription',
        success_url=success_url + '?session_id={CHECKOUT_SESSION_ID}',
        cancel_url=cancel_url,
        metadata={
            'tenant_id': self.tenant_id,
            'subscription_tier': new_tier
        }
    )

    return checkout_session.url

@app.get("/billing/usage")
async def get_tenant_usage(tenant_id: str = Depends(get_tenant_id)):
    """Get detailed usage statistics for tenant"""
    billing_service = BillingService(tenant_id, get_db())
    return await billing_service.get_usage_stats()

@app.post("/billing/upgrade")
async def create_upgrade_session(
    new_tier: str,
    tenant_id: str = Depends(get_tenant_id)
):
    """Create Stripe checkout session for subscription upgrade"""
    billing_service = BillingService(tenant_id, get_db())

    checkout_url = await billing_service.create_stripe_checkout_session(
        new_tier=new_tier,
        success_url=f"https://{tenant_id}.yourdomain.com/billing/success",
        cancel_url=f"https://{tenant_id}.yourdomain.com/billing/cancel"
    )

    return {"checkout_url": checkout_url}
```

```python
@app.post("/billing/webhook")
async def stripe_webhook(request: Request):
    """Handle Stripe webhooks for subscription events"""
    payload = await request.body()
    sig_header = request.headers.get('Stripe-Signature')

    try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, STRIPE_WEBHOOK_SECRET
        )
    except ValueError:
        raise HTTPException(400, "Invalid payload")
    except stripe.error.SignatureVerificationError:
        raise HTTPException(400, "Invalid signature")

    if event['type'] == 'checkout.session.completed':
        session = event['data']['object']
        tenant_id = session['metadata']['tenant_id']
        new_tier = session['metadata']['subscription_tier']

        # Update tenant subscription
        await update_tenant_subscription(tenant_id, new_tier, session)

    elif event['type'] == 'invoice.payment_failed':
        # Handle failed payment
        invoice = event['data']['object']
        await handle_payment_failure(invoice)

    return {"status": "success"}
```

10. Performance Optimization Strategies

10.1 Database Query Optimization

python

```python
# services/common/database_optimizations.py
from sqlalchemy import create_engine, event
from sqlalchemy.orm import sessionmaker
from sqlalchemy.pool import QueuePool
import logging

# Connection pooling configuration for QNAP constraints
DATABASE_CONFIG = {
    "pool_size": 5,         # Limit connections due to QNAP resources
    "max_overflow": 10,     # Maximum additional connections
    "pool_pre_ping": True,  # Verify connections before use
    "pool_recycle": 3600,   # Recycle connections every hour
}

engine = create_engine(
    DATABASE_URL,
    **DATABASE_CONFIG,
    echo=False  # Set to True for query debugging
)

# Query performance monitoring
@event.listens_for(engine, "before_cursor_execute")
def receive_before_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    context._query_start_time = time.time()

@event.listens_for(engine, "after_cursor_execute")
def receive_after_cursor_execute(conn, cursor, statement, parameters, context, executemany):
    total = time.time() - context._query_start_time
    if total > 1.0:  # Log slow queries
        logging.warning(f"Slow query ({total:.2f}s): {statement[:200]}...")

# Optimized tenant data access patterns
class TenantDataAccess:
    def __init__(self, tenant_id: str, db: Session):
        self.tenant_id = tenant_id
        self.db = db

    def get_dashboard_data(self) -> Dict[str, Any]:
        """Optimized dashboard data retrieval with single query"""

        # Single query to get all dashboard metrics
        result = self.db.execute(text("""
        WITH tenant_stats AS (
```

```
    SELECT
        COUNT(DISTINCT o.id) as total_orders,
        COALESCE(SUM(o.total_amount), 0) as total_revenue,
        COUNT(DISTINCT p.id) as total_products,
        COUNT(DISTINCT d.id) as total_designs
    FROM {schema}.orders o
    FULL OUTER JOIN {schema}.products p ON true
    FULL OUTER JOIN {schema}.designs d ON true
    WHERE o.created_at >= CURRENT_DATE - INTERVAL '30 days'
),
recent_activity AS (
    SELECT
        'order' as activity_type,
        id::text as activity_id,
        created_at,
        total_amount::text as activity_data
    FROM {schema}.orders
    WHERE created_at >= CURRENT_DATE - INTERVAL '7 days'

    UNION ALL

    SELECT
        'design' as activity_type,
        id::text as activity_id,
        created_at,
        name as activity_data
    FROM {schema}.designs
    WHERE created_at >= CURRENT_DATE - INTERVAL '7 days'
    ORDER BY created_at DESC
    LIMIT 10
)
SELECT
    json_build_object(
        'stats', row_to_json(ts),
        'recent_activity', json_agg(ra)
    ) as dashboard_data
FROM tenant_stats ts, recent_activity ra
GROUP BY ts.total_orders, ts.total_revenue, ts.total_products, ts.total_designs
""".format(schema=f"tenant_{self.tenant_id}")))

    return result.fetchone()[0]
```

10.2 Caching Strategy (Redis)

python

```python
# services/common/cache.py
import redis
import json
import hashlib
from functools import wraps
from typing import Any, Optional, Callable
import pickle

redis_client = redis.Redis.from_url(REDIS_URL, decode_responses=False)

class TenantCache:
    def __init__(self, tenant_id: str):
        self.tenant_id = tenant_id
        self.key_prefix = f"tenant:{tenant_id}"

    def cache_key(self, key: str) -> str:
        """Generate tenant-scoped cache key"""
        return f"{self.key_prefix}:{key}"

    def get(self, key: str) -> Optional[Any]:
        """Get cached value"""
        cache_key = self.cache_key(key)
        data = redis_client.get(cache_key)
        if data:
            return pickle.loads(data)
        return None

    def set(self, key: str, value: Any, expire: int = 3600) -> None:
        """Set cached value with expiration"""
        cache_key = self.cache_key(key)
        redis_client.setex(cache_key, expire, pickle.dumps(value))

    def delete(self, key: str) -> None:
        """Delete cached value"""
        cache_key = self.cache_key(key)
        redis_client.delete(cache_key)

    def invalidate_pattern(self, pattern: str) -> None:
        """Invalidate all keys matching pattern"""
        pattern_key = self.cache_key(pattern)
        keys = redis_client.keys(pattern_key)
        if keys:
            redis_client.delete(*keys)
```

```python
def cache_tenant_data(expire: int = 3600, key_suffix: str = None):
    """Decorator for caching tenant-specific data"""
    def decorator(func: Callable) -> Callable:
        @wraps(func)
        async def wrapper(tenant_id: str, *args, **kwargs):
            cache = TenantCache(tenant_id)

            # Generate cache key from function name and parameters
            if key_suffix:
                cache_key = f"{func.__name__}:{key_suffix}"
            else:
                # Create hash of parameters for unique key
                params_hash = hashlib.md5(
                    json.dumps({"args": args, "kwargs": kwargs}, sort_keys=True).encode()
                ).hexdigest()[:8]
                cache_key = f"{func.__name__}:{params_hash}"

            # Try to get from cache
            cached_result = cache.get(cache_key)
            if cached_result is not None:
                return cached_result

            # Execute function and cache result
            result = await func(tenant_id, *args, **kwargs)
            cache.set(cache_key, result, expire)

            return result
        return wrapper
    return decorator

# Usage examples
@cache_tenant_data(expire=1800, key_suffix="analytics")
async def get_tenant_analytics(tenant_id: str, date_range: str) -> Dict[str, Any]:
    """Cached analytics data"""
    # Expensive analytics computation here
    pass

@cache_tenant_data(expire=300)  # 5 minutes for frequently changing data
async def get_etsy_listings(tenant_id: str, page: int = 1) -> Dict[str, Any]:
    """Cached Etsy listings with pagination"""
    # API call to Etsy
    pass
```

11. Deployment and Operations

11.1 Production Deployment Script for QNAP

bash

```
#!/bin/bash
# deploy-production.sh

set -e

QNAP_HOST="your-qnap-ip"
QNAP_USER="admin"
PROJECT_PATH="/share/Container/etsy-saas-prod"
BACKUP_PATH="/share/Container/backups"

echo "🚀 Starting production deployment to QNAP NAS..."

# Create deployment directory
ssh $QNAP_USER@$QNAP_HOST "mkdir -p $PROJECT_PATH"

# Backup current deployment if exists
ssh $QNAP_USER@$QNAP_HOST "
if [ -d $PROJECT_PATH/current ]; then
    echo '📦 Creating backup of current deployment...'
    mkdir -p $BACKUP_PATH
    tar -czf $BACKUP_PATH/backup-$(date +%Y%m%d-%H%M%S).tar.gz -C $PROJECT_PATH current
fi
"

# Upload new deployment files
echo "📤 Uploading deployment files..."
rsync -avz --exclude='.git' --exclude='node_modules' --exclude='__pycache__' \
    ./ $QNAP_USER@$QNAP_HOST:$PROJECT_PATH/new/

# Switch to new deployment
ssh $QNAP_USER@$QNAP_HOST "
cd $PROJECT_PATH
if [ -d current ]; then
    mv current previous
fi
mv new current
"

# Start services
echo "🔄 Starting production services..."
ssh $QNAP_USER@$QNAP_HOST "
cd $PROJECT_PATH/current
docker-compose -f docker-compose.prod.yml down || true
```

```
docker-compose -f docker-compose.prod.yml pull
docker-compose -f docker-compose.prod.yml up -d
"

# Health check
echo "🏥 Running health checks..."
sleep 30
ssh $QNAP_USER@$QNAP_HOST "
cd $PROJECT_PATH/current
./scripts/health-check.sh
"

# Cleanup old images
ssh $QNAP_USER@$QNAP_HOST "
echo '🧹 Cleaning up old Docker images...'
docker system prune -f
docker image prune -f
"

echo "✅ Production deployment completed successfully!"
echo "🌐 Services available at: https://your-domain.com"
echo "📊 Monitor logs with: ssh $QNAP_USER@$QNAP_HOST 'cd $PROJECT_PATH/current && docker-compos
```