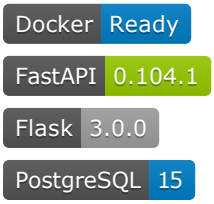
Système Multi-Magasins - LOG430



Application web Flask pour la gestion de points de vente multi-magasins, API RESTful pour application externe, système de logging et déploiement Docker containerisé.

Table des Matières

- Fonctionnalités Principales
- Architecture
- Logging et Monitoring
- Load Balancing et Haute Disponibilité
- Cache Redis et Optimisation des Performances
- Monitoring et Tests de Performance
- API RESTful
- Déploiement Docker
- Structure du Projet
- Installation et Configuration
- Tests
- Utilisation
- Technologies Utilisées

Fonctionnalités Principales

Interface Web (Flask)

- **Dashboard multi-magasins** : Vue d'ensemble de 5 magasins avec navigation intuitive
- **Rapports stratégiques** : KPIs globaux, performance par magasin, top produits, tendances
- **Point de vente complet** : Recherche produits, gestion panier, reçus, retours
- **Gestion stocks** : Stocks par magasin, alertes automatiques, réapprovisionnement
- **Interface responsive** : Bootstrap, design moderne et adaptatif

API RESTful (FastAPI)

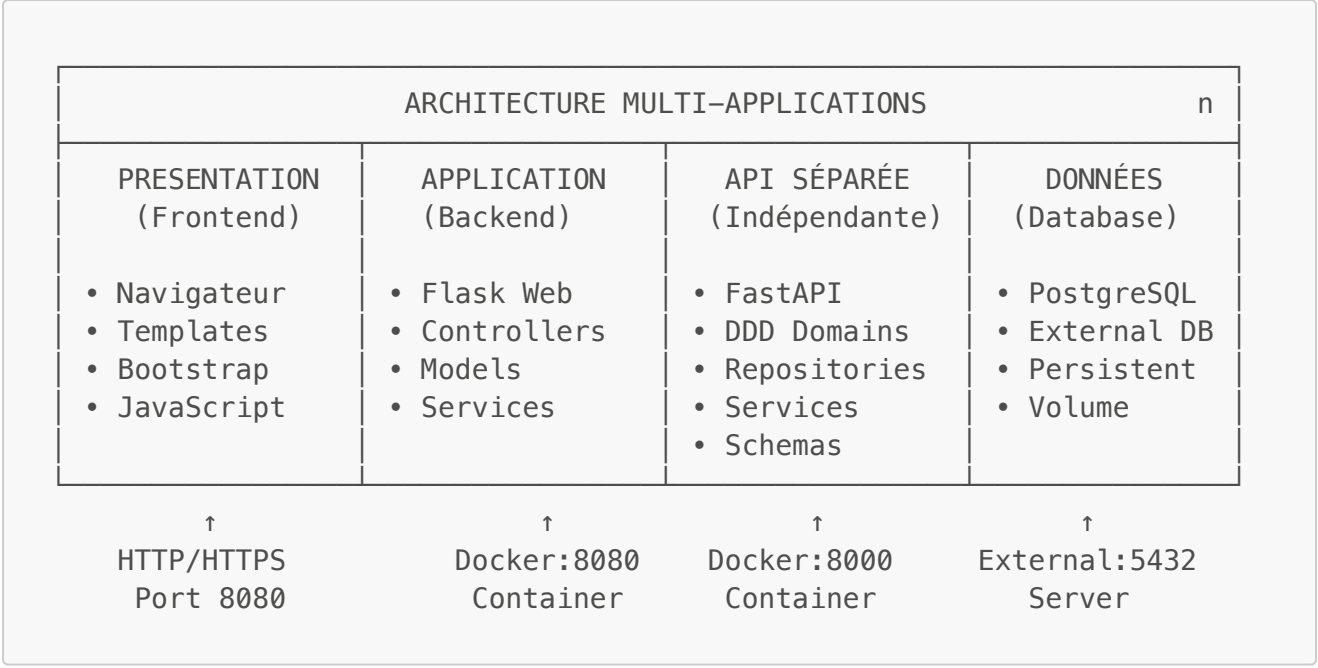
- **Architecture DDD** : Domain-Driven Design avec 3 domaines métier
- **Documentation automatique** : Swagger UI et ReDoc intégrés
- **Authentification** : Système de tokens sécurisé
- **Validation** : Pydantic pour la validation des données
- **Performance** : Optimisations asynchrones et mise en cache

Système de Logging

- **Logging multi-niveaux** : API, Business, Erreurs
- **Rotation automatique** : Gestion intelligente des fichiers de logs
- **Formats multiples** : JSON structuré et texte lisible
- **Monitoring** : Métriques de performance et suivi des erreurs

Architecture

Architecture Multi-Applications Containerisée



Composants Principaux

1. Application Web (Flask) - Port 8080

- **MVC Pattern** : Séparation claire des responsabilités
- **7 Contrôleurs** : Gestion modulaire des fonctionnalités
- **Templates Jinja2** : Interface utilisateur dynamique
- **Bootstrap 5** : Design responsive et moderne
- **Application indépendante** : Fonctionne de manière autonome

2. API RESTful (FastAPI) - Port 8000

- **Domain-Driven Design** : Architecture en domaines métier
- **3 Domaines** : Products, Stores, Reporting
- **Repositories Pattern** : Abstraction de la couche de données
- **Services Layer** : Logique métier centralisée
- **Application séparée** : API indépendante de l'interface web

3. Base de Données (PostgreSQL) - Port 5432

- **Connexions poolées** : Optimisation des performances
- **Partagée** : Utilisée par les deux applications

- **Migrations** : Gestion des schémas de données

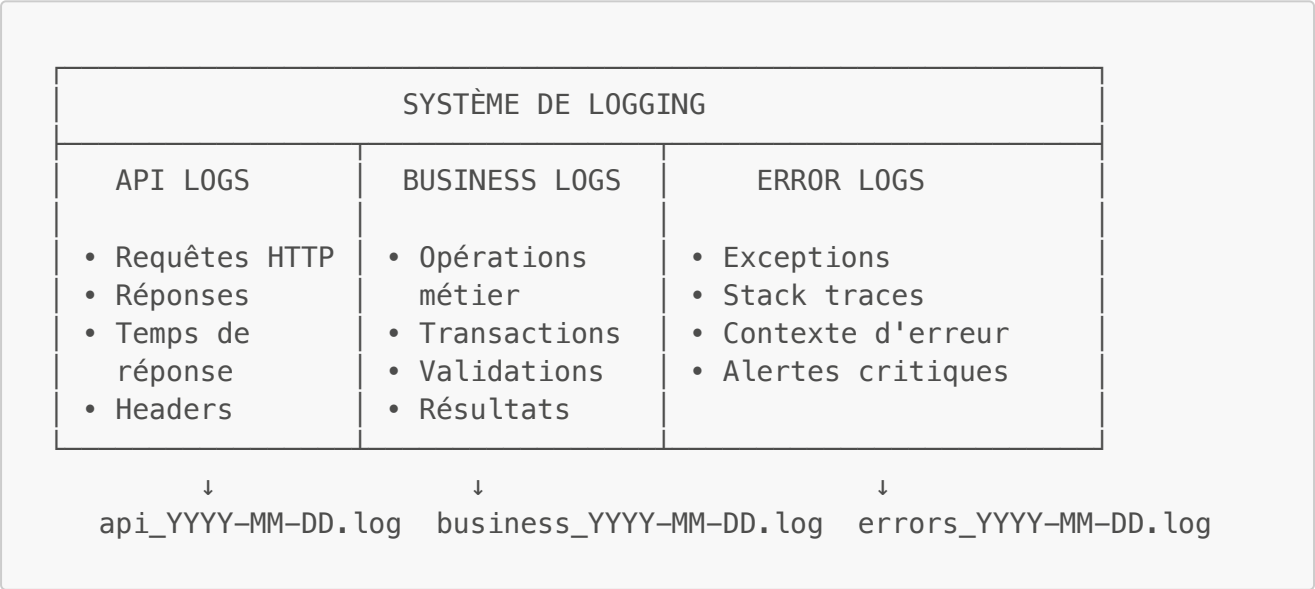
Caractéristiques de l'Architecture

- **Applications indépendantes** : Flask et FastAPI sont des applications séparées
- **Déploiement containerisé** : Chaque application dans son propre container
- **Base de données partagée** : Les deux applications accèdent à la même base PostgreSQL
- **Ports distincts** : Chaque application expose ses services sur des ports différents
- **Développement parallèle** : Les équipes peuvent travailler indépendamment sur chaque application

Logging et Monitoring

Architecture du Logging

Le système de logging est conçu pour fournir une visibilité complète sur le fonctionnement de l'application avec une séparation claire des types de logs.



Configuration des Logs

Types de Logs

- **API Logs** : Toutes les requêtes HTTP avec métriques de performance
- **Business Logs** : Opérations métier en format JSON structuré
- **Error Logs** : Erreurs avec contexte complet et stack traces

Rotation et Archivage

- **Taille maximale** : 10MB par fichier
- **Rétention** : 5-10 fichiers de sauvegarde
- **Nommage** : `{type}_YYYY-MM-DD.log`
- **Formats** : JSON pour le business, texte pour les API

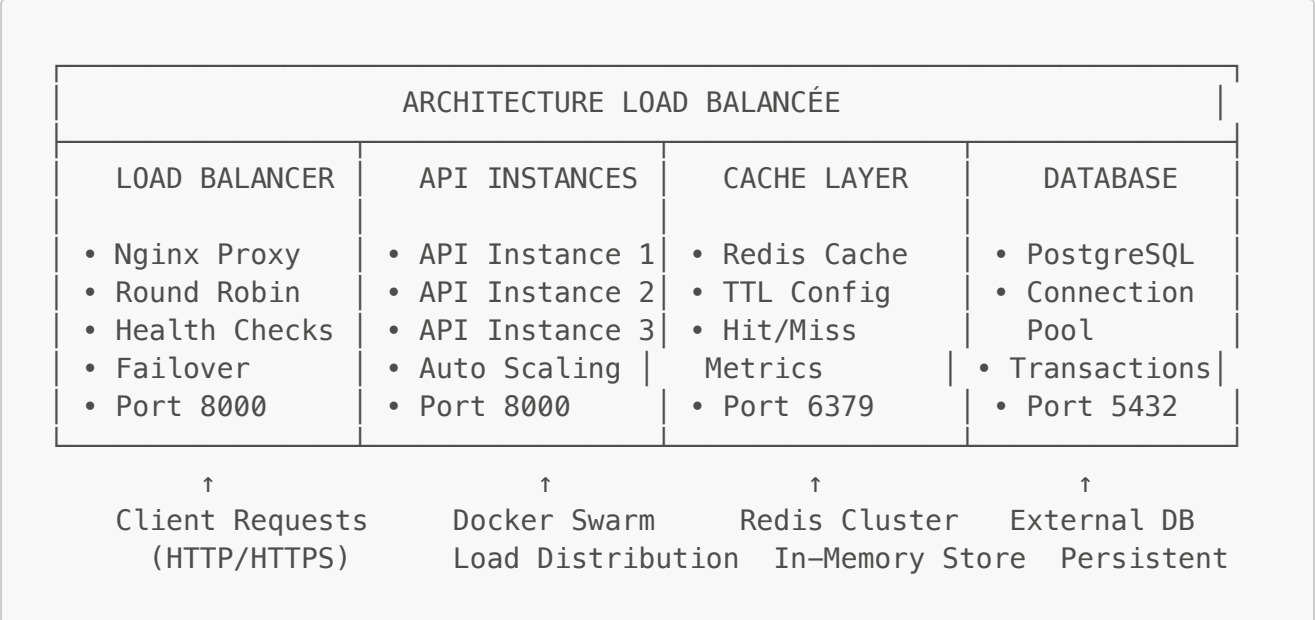
Fonctionnalités Avancées

```python
# Logging automatique des requêtes HTTP
@app.middleware("http")
async def log_requests(request: Request, call_next):
    # Logging avec métriques de performance

# Logging des opérations métier
log_business_operation(
    operation="create_store",
    entity_type="Store",
    entity_id=store.id,
    user_id=current_user.id,
    details={"name": store.name, "location": store.location}
)

# Logging des erreurs avec contexte
log_error_with_context(
    error=exception,
    context={
        "endpoint": "/api/v1/stores",
        "method": "POST",
        "user_id": user_id,
        "request_data": request_data
    }
)
```

Load Balancing et Haute Disponibilité

Architecture Load Balancée

Le système utilise Nginx comme load balancer pour distribuer les requêtes entre plusieurs instances API, garantissant haute disponibilité et performance optimale.



Configuration Nginx

Équilibrage de Charge

```
upstream api_backend {
    server api-1:8000 max_fails=5 fail_timeout=10s;
    server api-2:8000 max_fails=5 fail_timeout=10s;
    server api-3:8000 max_fails=5 fail_timeout=10s;
    keepalive 32;
}

server {
    listen 80;

    # Timeouts optimisés
    proxy_connect_timeout 10s;
    proxy_send_timeout 30s;
    proxy_read_timeout 30s;

    # Load balancing avec failover
    location / {
        proxy_pass http://api_backend;
        proxy_next_upstream error timeout http_500 http_502 http_503;
        proxy_next_upstream_tries 3;
    }

    # Health checks spécialisés
    location /health {
        proxy_pass http://api_backend/health;
        proxy_connect_timeout 5s;
        proxy_read_timeout 10s;
    }
}
```

Stratégies de Distribution

1. Round Robin (par défaut)

- Distribution séquentielle des requêtes
- Équilibre automatique de la charge
- Adapté pour instances homogènes

2. Health Checks Intelligents

```
• max_fails=5 : Marquer un serveur comme indisponible après 5 échecs
• fail_timeout=10s : Réessayer après 10 secondes
• Vérification continue de la santé des instances
• Basculement automatique en cas de panne
```

3. Connection Pooling

```
• keepalive 32 : Maintenir 32 connexions persistantes
• Réduction de la latence de connexion
• Optimisation des performances réseau
```

Déploiement Load Balancé

Docker Compose Configuration

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

services:
  # Load Balancer Nginx
  nginx:
    image: nginx:alpine
    container_name: log430-nginx
    ports:
      - "8000:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - api-1
      - api-2
      - api-3
    healthcheck:
      test: ["CMD", "nginx", "-t"]
      interval: 30s
      timeout: 5s
      retries: 3

  # API Instance 1
  api-1:
    build:
      context: .
      dockerfile: dockerfile.api
    container_name: log430-api-1
    environment:
      - INSTANCE_ID=api-1
      - REDIS_URL=redis://redis:6379
    deploy:
      resources:
        limits:
          memory: 1G
          cpus: '1.0'
        reservations:
          memory: 512M
          cpus: '0.5'
```

```
  # API Instance 2
  api-2:
    build:
      context: .
      dockerfile: dockerfile.api
    container_name: log430-api-2
    environment:
      - INSTANCE_ID=api-2
      - REDIS_URL=redis://redis:6379
    deploy:
      resources:
        limits:
          memory: 1G
          cpus: '1.0'

  # API Instance 3
  api-3:
    build:
      context: .
      dockerfile: dockerfile.api
    container_name: log430-api-3
    environment:
      - INSTANCE_ID=api-3
      - REDIS_URL=redis://redis:6379
    deploy:
      resources:
        limits:
          memory: 1G
          cpus: '1.0'

  # Redis Cache
  redis:
    image: redis:7-alpine
    container_name: log430-redis
    ports:
      - "6379:6379"
    command: redis-server --maxmemory 256mb --maxmemory-policy allkeys-
lru
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 10s
      timeout: 3s
      retries: 3
```

Avantages du Load Balancing

Performance

- **Distribution de charge** : Répartition équitable entre instances
- **Réduction latence** : Routage vers instance la plus disponible

- **Scalabilité horizontale** : Ajout facile d'instances
- **Optimisation ressources** : Utilisation efficace du CPU/RAM

Disponibilité

- **Haute disponibilité** : 99.9% uptime avec redondance
- **Tolérance aux pannes** : Continuation de service si une instance échoue
- **Maintenance sans interruption** : Mise à jour rolling des instances
- **Récupération automatique** : Réintégration d'instances réparées

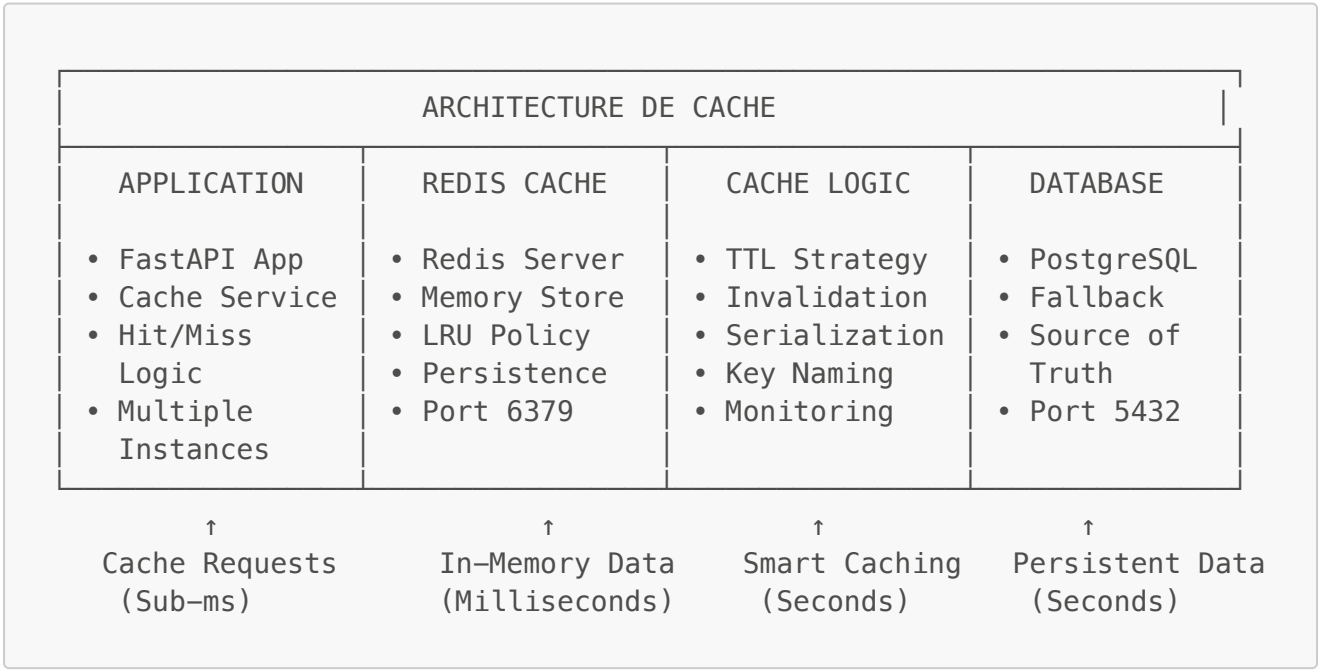
Monitoring

- **Métriques par instance** : Surveillance individuelle
- **Distribution du trafic** : Analyse de la répartition
- **Health status** : État en temps réel de chaque instance
- **Performance comparative** : Comparaison entre instances

Cache Redis et Optimisation des Performances

Architecture de Cache

Le système utilise Redis comme couche de cache distribué pour optimiser les performances des endpoints critiques.



Implémentation du Cache

Service de Cache

```
# src/api/v1/services/cache_service.py
import redis
import json
```

```
import logging
from typing import Optional, Any
from datetime import timedelta

class CacheService:
    def __init__(self, redis_url: str = "redis://localhost:6379"):
        try:
            self.redis_client = redis.from_url(
                redis_url,
                decode_responses=True,
                socket_timeout=5,
                socket_connect_timeout=5,
                retry_on_timeout=True
            )
            # Test de connexion
            self.redis_client.ping()
            self.enabled = True
            logging.info("Redis cache connecté avec succès")
        except Exception as e:
            logging.error(f"Erreur connexion Redis: {e}")
            self.enabled = False

    def get(self, key: str) -> Optional[Any]:
        """Récupère une valeur du cache"""
        if not self.enabled:
            return None

        try:
            value = self.redis_client.get(key)
            if value:
                return json.loads(value)
            return None
        except Exception as e:
            logging.error(f"Erreur lecture cache: {e}")
            return None

    def set(self, key: str, value: Any, ttl_seconds: int = 300) -> bool:
        """Stocke une valeur dans le cache avec TTL"""
        if not self.enabled:
            return False

        try:
            # Sérialisation spéciale pour les objets Pydantic
            if hasattr(value, 'model_dump'):
                serialized_value = json.dumps(value.model_dump(),
default=str)
            elif hasattr(value, 'dict'):
                serialized_value = json.dumps(value.dict(), default=str)
            else:
                serialized_value = json.dumps(value, default=str)

            self.redis_client.setex(key, ttl_seconds, serialized_value)
            return True
```

```
            except Exception as e:
                logging.error(f"Erreur écriture cache: {e}")
                return False

        def delete(self, key: str) -> bool:
            """Supprime une clé du cache"""
            if not self.enabled:
                return False

            try:
                return bool(self.redis_client.delete(key))
            except Exception as e:
                logging.error(f"Erreur suppression cache: {e}")
                return False

        def get_stats(self) -> dict:
            """Récupère les statistiques du cache"""
            if not self.enabled:
                return {"enabled": False, "error": "Redis non disponible"}

            try:
                info = self.redis_client.info()
                return {
                    "enabled": True,
                    "hits": info.get("keyspace_hits", 0),
                    "misses": info.get("keyspace_misses", 0),
                    "keys": self.redis_client.dbsize(),
                    "memory_used": info.get("used_memory_human", "0B"),
                    "connected_clients": info.get("connected_clients", 0)
                }
            except Exception as e:
                return {"enabled": False, "error": str(e)}
```

Stratégies de Cache

1. Cache par Endpoint

```
# Endpoints avec cache personnalisé
CACHE_STRATEGIES = {
    "products_list": {
        "ttl": 300,  # 5 minutes
        "key_pattern": "products:list:{page}:{size}:{filters_hash}"
    },
    "product_detail": {
        "ttl": 600,  # 10 minutes
        "key_pattern": "product:{product_id}"
    },
    "stores_list": {
        "ttl": 1800,  # 30 minutes
        "key_pattern": "stores:list"
```

```
    },
    "reports_summary": {
        "ttl": 120,  # 2 minutes
        "key_pattern": "report:summary:{date_range_hash}"
    }
}
```

2. Invalidation Intelligente

```
def invalidate_product_cache(product_id: int):
    """Invalide le cache d'un produit spécifique"""
    cache_service.delete(f"product:{product_id}")
    # Invalider aussi les listes qui pourraient contenir ce produit
    cache_service.delete_pattern("products:list:*")

def invalidate_all_cache():
    """Vide tout le cache pour maintenance"""
    cache_service.redis_client.flushdb()
```

3. Cache Conditionnel

```
@router.get("/products/")
async def get_products(
    page: int = 1,
    size: int = 10,
    cache_service: CacheService = Depends(get_cache_service)
):
    # Génération de clé de cache
    cache_key = f"products:list:{page}:{size}"

    # Tentative de récupération depuis le cache
    cached_result = cache_service.get(cache_key)
    if cached_result:
        return cached_result

    # Si pas en cache, récupération depuis la DB
    result = await product_service.get_products(page, size)

    # Mise en cache du résultat
    cache_service.set(cache_key, result, ttl_seconds=300)

    return result
```

Configuration Redis

Optimisations Mémoire

```
# Configuration Redis pour production
maxmemory 256mb
maxmemory-policy allkeys-lru

# Persistance optimisée
save 900 1
save 300 10
save 60 10000

# Performance
tcp-keepalive 300
timeout 300
```

Monitoring du Cache

```python
# Métriques de cache avec Prometheus
CACHE_OPERATIONS = Counter(
    'cache_operations_total',
    'Total cache operations',
    ['operation', 'result', 'instance_id']
)

CACHE_HIT_RATIO = Gauge(
    'cache_hit_ratio',
    'Cache hit ratio percentage',
    ['instance_id']
)

def record_cache_hit():
    CACHE_OPERATIONS.labels(
        operation='get',
        result='hit',
        instance_id=INSTANCE_ID
    ).inc()

def record_cache_miss():
    CACHE_OPERATIONS.labels(
        operation='get',
        result='miss',
        instance_id=INSTANCE_ID
    ).inc()
```

Avantages du Cache Redis

Performance

- **Réduction latence** : 90%+ d'amélioration sur endpoints cachés

- **Débit augmenté** : Capacité de traiter plus de requêtes simultanées
- **Réduction charge DB** : Moins de requêtes vers PostgreSQL
- **Réponse sub-milliseconde** : Cache en mémoire ultra-rapide

Scalabilité

- **Cache distribué** : Partagé entre toutes les instances API
- **Éviction intelligente** : Politique LRU pour optimiser l'utilisation mémoire
- **TTL flexible** : Différents temps d'expiration selon les données
- **Invalidation sélective** : Mise à jour ciblée du cache

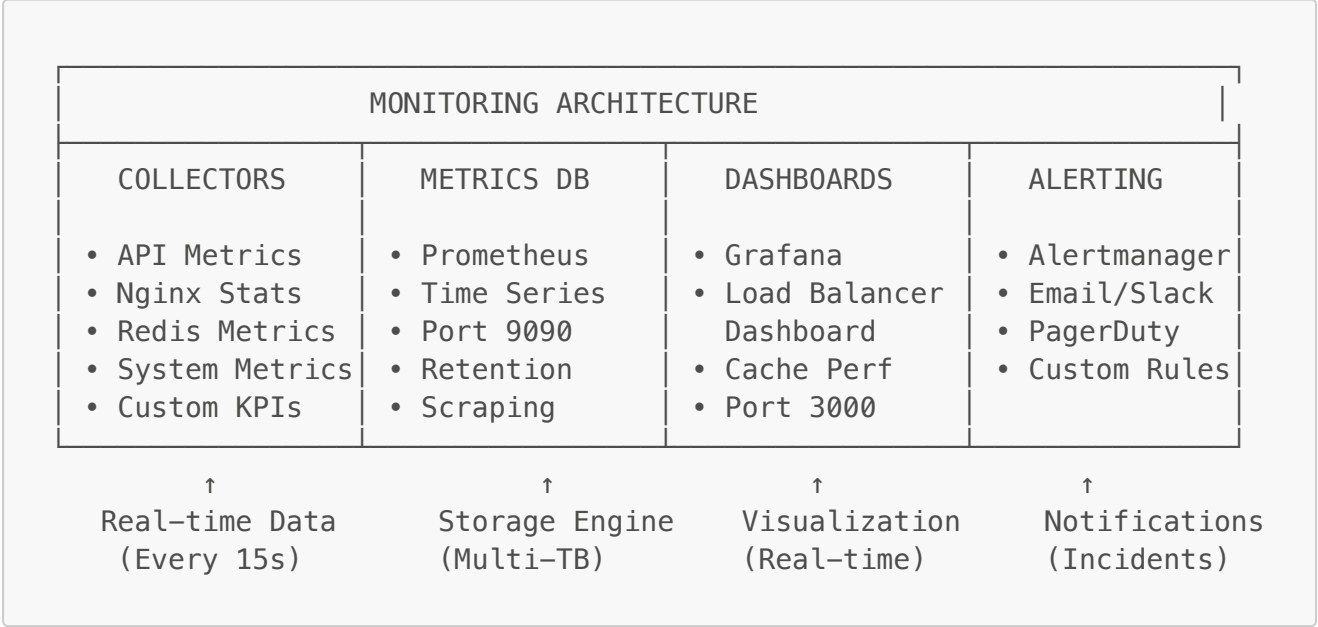
Monitoring

- **Hit ratio** : Pourcentage de succès du cache
- **Métriques détaillées** : Hits, misses, évictions, mémoire utilisée
- **Alertes** : Notification si performance du cache se dégrade
- **Dashboards** : Visualisation en temps réel des performances

Monitoring et Tests de Performance

Infrastructure de Monitoring

Le système utilise Prometheus et Grafana pour un monitoring complet des performances, avec des dashboards spécialisés pour l'architecture load balancée.



Configuration Prometheus

Scraping Configuration

```
# prometheus/prometheus-loadbalanced.yml
global:
  scrape_interval: 15s
```

```
  evaluation_interval: 15s

rule_files:
  - "rules/*.yml"

scrape_configs:
  # Load Balancer Nginx
  - job_name: 'nginx-loadbalancer'
    static_configs:
      - targets: ['nginx:80']
    metrics_path: '/nginx_status'
    scrape_interval: 15s

  # API Instances via Load Balancer
  - job_name: 'api-loadbalanced'
    static_configs:
      - targets: ['nginx:80']
    metrics_path: '/metrics'
    scrape_interval: 15s

  # API Instances Direct
  - job_name: 'api-instances'
    static_configs:
      - targets: ['api-1:8000', 'api-2:8000', 'api-3:8000']
    metrics_path: '/metrics'
    scrape_interval: 15s

  # Redis Cache
  - job_name: 'redis'
    static_configs:
      - targets: ['redis:6379']
    scrape_interval: 30s

  # Prometheus Self-Monitoring
  - job_name: 'prometheus'
    static_configs:
      - targets: ['localhost:9090']
```

Dashboards Grafana

Dashboard Load Balancer

```
{
  "dashboard": {
    "title": "API Load Balancer Performance",
    "panels": [
      {
        "title": "Load Balancer Status",
        "type": "stat",
        "targets": [{
```

```
          "expr": "up{job=\"nginx-loadbalancer\"}"
        }]
      },
      {
        "title": "Instance Health Status",
        "type": "stat",
        "targets": [{
          "expr": "up{job=\"api-instances\"}"
        }]
      },
      {
        "title": "Request Distribution",
        "type": "piechart",
        "targets": [{
          "expr": "rate(http_requests_total[1m])"
        }]
      },
      {
        "title": "Response Time by Instance",
        "type": "timeseries",
        "targets": [{
          "expr": "histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[1m]))"
        }]
      },
      {
        "title": "Error Rate by Instance",
        "type": "timeseries",
        "targets": [{
          "expr": "rate(http_requests_total{status=~\"5..\"}[1m])"
        }]
      },
      {
        "title": "Cache Performance",
        "type": "timeseries",
        "targets": [{
          "expr": "cache_hit_ratio"
        }]
      }
    ]
  }
}
```

Dashboard Cache Performance

```
{
  "dashboard": {
    "title": "Redis Cache Performance",
    "panels": [
      {
```

```
        "title": "Cache Hit Ratio",
        "type": "gauge",
        "targets": [{
          "expr": "(redis_keyspace_hits_total /
(redis_keyspace_hits_total + redis_keyspace_misses_total)) * 100"
        }]
      },
      {
        "title": "Cache Operations Rate",
        "type": "timeseries",
        "targets": [{
          "expr": "rate(cache_operations_total[1m])"
        }]
      },
      {
        "title": "Memory Usage",
        "type": "timeseries",
        "targets": [{
          "expr": "redis_memory_used_bytes"
        }]
      }
    ]
  }
}
```

Tests de Performance avec K6

Script de Test Load Balancé

```
// k6-tests/loadbalanced-stress-test.js
import http from 'k6/http';
import { check } from 'k6';

export let options = {
  stages: [
    { duration: '2m', target: 100 },   // Montée progressive
    { duration: '3m', target: 500 },   // Charge moyenne
    { duration: '2m', target: 1000 },  // Pic de charge
    { duration: '3m', target: 1000 },  // Maintien du pic
    { duration: '2m', target: 0 },     // Descente
  ],
  thresholds: {
    http_req_duration: ['p(95)<500'],
    http_req_failed: ['rate<0.1'],
    http_reqs: ['rate>100'],
  }
};

const BASE_URL = 'http://localhost:8000';
const API_TOKEN =
```

```
'9645524dac794691257cb44d61ebc8c3d5876363031ec6f66fbd31e4bf85cd84';

export default function() {
  const headers = {
    'X-API-Token': API_TOKEN,
    'Content-Type': 'application/json'
  };

  // Test des endpoints critiques
  let endpoints = [
    '/api/v1/products/',
    '/api/v1/stores/',
    '/api/v1/products/1',
    '/health'
  ];

  let endpoint = endpoints[Math.floor(Math.random() *
endpoints.length)];
  let response = http.get(`${BASE_URL}${endpoint}`, { headers });

  check(response, {
    'status is 200': (r) => r.status === 200,
    'response time < 500ms': (r) => r.timings.duration < 500,
    'has instance header': (r) => r.headers['X-Instance-Id'] !==
undefined,
  });
}
```

Métriques Personnalisées

API Metrics

```
from prometheus_client import Counter, Histogram, Gauge

# Compteurs de requêtes
HTTP_REQUESTS = Counter(
    'http_requests_total',
    'Total HTTP requests',
    ['method', 'endpoint', 'status', 'instance_id']
)

# Latence des requêtes
REQUEST_DURATION = Histogram(
    'http_request_duration_seconds',
    'HTTP request duration',
    ['method', 'endpoint', 'instance_id'],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0, 10.0]
)

# RPS en temps réel
```

```
CURRENT_RPS = Gauge(
    'current_requests_per_second',
    'Current requests per second',
    ['instance_id']
)

# Métriques de cache
CACHE_HIT_RATIO = Gauge(
    'cache_hit_ratio',
    'Cache hit ratio percentage',
    ['instance_id']
)
```

Alerts et Notifications

Règles d'Alerte

```
# prometheus/rules/api_alerts.yml
groups:
  - name: api_alerts
    rules:
      - alert: HighErrorRate
        expr: rate(http_requests_total{status=~"5.."}[5m]) > 0.1
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "High error rate detected"
          description: "Error rate is {{ $value }} errors per second"

      - alert: HighResponseTime
        expr: histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m])) > 1
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High response time detected"

      - alert: InstanceDown
        expr: up{job="api-instances"} == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "API instance is down"

      - alert: LowCacheHitRatio
        expr: cache_hit_ratio < 0.7
        for: 5m
```

```
        labels:
          severity: warning
        annotations:
          summary: "Cache hit ratio is low"
```

Commandes de Monitoring

```
# Démarrer le monitoring complet
docker-compose -f docker-compose.loadbalanced.yml -f docker-
compose.monitoring.yml up -d

# Tests de performance
k6 run k6-tests/loadbalanced-stress-test.js

# Vérifier les métriques
curl http://localhost:9090/api/v1/query?query=up

# Accès aux dashboards
# Grafana: http://localhost:3000
# Prometheus: http://localhost:9090
```

Analyse des Résultats

Métriques Clés

- **Throughput** : Requêtes traitées par seconde
- **Latence P95** : 95% des requêtes sous X millisecondes
- **Taux d'erreur** : Pourcentage de requêtes échouées
- **Disponibilité** : Pourcentage d'uptime
- **Efficiency du cache** : Ratio hit/miss du cache Redis

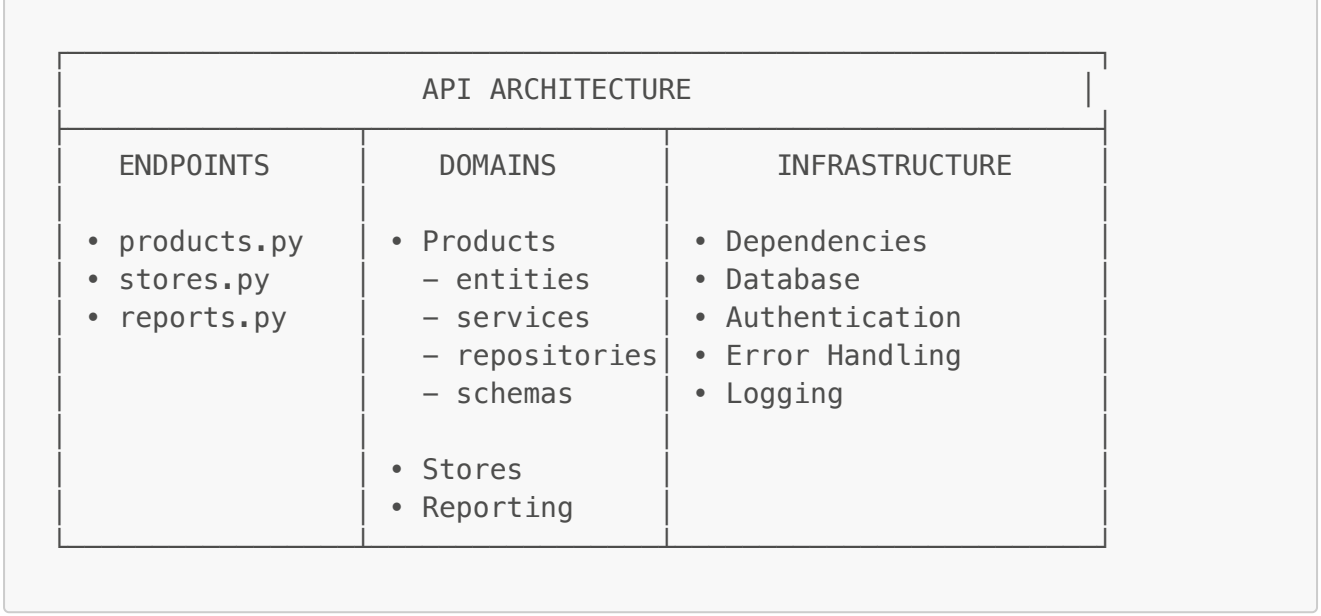
Optimisations Basées sur les Métriques

- **Scale horizontale** : Ajouter des instances si CPU > 80%
- **Optimisation cache** : Ajuster TTL si hit ratio < 70%
- **Tuning load balancer** : Modifier les timeouts selon la latence
- **Database optimization** : Index si requêtes lentes détectées

API RESTful

Architecture DDD (Domain-Driven Design)

L'API est structurée selon les principes du Domain-Driven Design avec une séparation claire des responsabilités.



Domaines Métier

1. Products Domain

```
/api/v1/products/
├── GET    /           # Liste des produits avec filtres
├── POST   /           # Créer un produit
├── GET    /{id}       # Détails d'un produit
├── PUT    /{id}       # Modifier un produit
└── DELETE /{id}       # Supprimer un produit
```

2. Stores Domain

```
/api/v1/stores/
├── GET    /           # Liste des magasins
├── POST   /           # Créer un magasin
├── GET    /{id}       # Détails d'un magasin
├── PUT    /{id}       # Modifier un magasin
├── DELETE /{id}       # Supprimer un magasin
└── GET    /{id}/stock # Stock du magasin
```

3. Reporting Domain

```
/api/v1/reports/
├── GET    /sales      # Rapports de ventes
├── GET    /inventory  # Rapports d'inventaire
├── GET    /kpis       # Indicateurs clés
└── POST   /custom     # Rapports personnalisés
```

Authentification et Sécurité

```
# Token-based authentication
headers = {
    "Authorization": "Bearer your-api-token",
    "Content-Type": "application/json"
}

# Exemple d'utilisation
response = requests.get(
    "http://localhost:8000/api/v1/stores",
    headers=headers
)
```

Documentation Interactive

- **Swagger UI** : http://localhost:8000/docs
- **ReDoc** : http://localhost:8000/redoc
- **OpenAPI Schema** : http://localhost:8000/openapi.json

Déploiement Docker

Architecture Containerisée

Le projet utilise une architecture Docker avec des containers séparés pour chaque service.

```
┌─────────────────────────────────────────────────────────┐
│                  DOCKER ARCHITECTURE                    │
├─────────────────┬─────────────────┬─────────────────────┤
│   WEB CONTAINER │  API CONTAINER  │   EXTERNAL DATABASE │
│                 │                 │                     │
│ • Flask App     │ • FastAPI       │ • PostgreSQL Server │
│ • Gunicorn      │ • Gunicorn      │ • External Host     │
│ • Port 8080     │ • Port 8000     │ • Port 5432         │
│ • Health Checks │ • Health Checks │ • Persistent Data   │
│ • Logging       │ • Logging       │                     │
└─────────────────┴─────────────────┴─────────────────────┘
         ↓                 ↓                     ↓
app-multi-magasin-web    api          10.194.32.219:5432 (VM ETS)
```

Configuration Docker

docker-compose.yml

```
version: '3.8'
```

```
services:
  # API FastAPI
  api:
    build:
      context: .
      dockerfile: dockerfile.api
      target: production
    container_name: log430-api
    environment:
      -
DATABASE_URL=postgresql://user:password@10.194.32.219:5432/store_db
      - LOG_LEVEL=INFO
      - API_TOKEN=your-secret-api-token
    ports:
      - "8000:8000"
    volumes:
      - ./logs:/app/logs
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3

  # Application Flask
  web:
    build:
      context: .
      dockerfile: dockerfile.flask
      target: production
    container_name: log430-web
    environment:
      -
DATABASE_URL=postgresql://user:password@10.194.32.219:5432/store_db
      - API_BASE_URL=http://api:8000
    ports:
      - "8080:8080"
    depends_on:
      - api
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8080/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

Dockerfiles Multi-Stage

dockerfile.api (FastAPI)

```
# Stage de développement
FROM python:3.9-slim as development
```

```
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
CMD ["uvicorn", "src.api.main:app", "--host", "0.0.0.0", "--port",
"8000", "--reload"]

# Stage de production
FROM python:3.9-slim as production
RUN apt-get update && apt-get install -y curl && rm -rf
/var/lib/apt/lists/*
RUN groupadd -r appuser && useradd -r -g appuser appuser
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src/ src/
RUN mkdir -p logs && chown -R appuser:appuser /app
USER appuser
EXPOSE 8000
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:8000/health || exit 1
CMD ["gunicorn", "src.api.main:app", "-w", "4", "-k",
"uvicorn.workers.UvicornWorker", "--bind", "0.0.0.0:8000"]
```

dockerfile.flask (Web App)

```
# Stage de production
FROM python:3.9-slim as production
RUN apt-get update && apt-get install -y curl && rm -rf
/var/lib/apt/lists/*
RUN groupadd -r appuser && useradd -r -g appuser appuser
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY src/ src/
RUN chown -R appuser:appuser /app
USER appuser
EXPOSE 8080
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
  CMD curl -f http://localhost:8080/health || exit 1
CMD ["gunicorn", "--bind", "0.0.0.0:8080", "--workers", "4",
"src.app.run:app"]
```

Commandes de Gestion (Makefile)

```
# Initialisation
make init          # Créer les répertoires nécessaires

# Gestion des services
```

```
make build          # Construire les images Docker
make up             # Démarrer tous les services
make down           # Arrêter tous les services
make restart        # Redémarrer tous les services

# Monitoring
make status         # Voir le statut des services
make logs           # Voir tous les logs
make logs-api       # Logs de l'API uniquement
make logs-web       # Logs de l'app web uniquement

# Maintenance
make clean          # Nettoyer les containers
make test           # Exécuter les tests
make shell-api      # Shell dans le container API
make shell-web      # Shell dans le container Web
```

Sécurité et Bonnes Pratiques

- **Images multi-stage** : Optimisation de la taille et sécurité
- **Health checks** : Surveillance automatique de la santé des services
- **Volumes read-only** : Code source en lecture seule en production
- **Variables d'environnement** : Configuration externalisée

Structure du Projet

```
log430-labo-03/
├── Docker & Déploiement
│   ├── docker-compose.yml          # Configuration principale
│   ├── dockerfile.api              # Image FastAPI
│   ├── dockerfile.flask            # Image Flask
│   ├── Makefile                    # Commandes de gestion
│   └── DOCKER_README.md            # Documentation Docker
│
├── API RESTful (FastAPI)
│   └── src/api/
│       ├── main.py                 # Point d'entrée API
│       ├── logging_config.py       # Configuration logging
│       └── v1/
│           ├── api.py              # Router principal
│           ├── dependencies.py     # Dépendances communes
│           ├── errors.py           # Gestion d'erreurs
│           ├── endpoints/          # Endpoints REST
│           │   ├── products.py     # CRUD Produits
│           │   ├── stores.py       # CRUD Magasins
│           │   └── reports.py      # Rapports
│           └── domain/             # Architecture DDD
│               ├── products/       # Domaine Produits
│               │   ├── entities/
│               │   ├── repositories/
```

```
│               │   ├── services/
│               │   └── schemas/
│               ├── stores/         # Domaine Magasins
│               └── reporting/      # Domaine Rapports
│
├── Application Web (Flask)
│   └── src/app/
│       ├── __init__.py            # Factory Flask
│       ├── run.py                 # Point d'entrée
│       ├── config.py              # Configuration
│       ├── models/
│       │   └── models.py          # Modèles SQLAlchemy
│       ├── controllers/           # Contrôleurs MVC
│       │   ├── home_controller.py
│       │   ├── magasin_controller.py
│       │   ├── caisse_controller.py
│       │   ├── produit_controller.py
│       │   ├── vente_controller.py
│       │   ├── rapport_controller.py
│       │   └── stock_central_controller.py
│       ├── templates/             # Templates Jinja2
│       │   ├── base.html
│       │   ├── home.html
│       │   ├── rapport/
│       │   ├── magasin/
│       │   ├── caisse/
│       │   ├── produit/
│       │   └── vente/
│       └── static/
│           └── css/
│               └── style.css
│
├── Logging & Monitoring
│   ├── logs/                      # Fichiers de logs
│   │   ├── api_YYYY-MM-DD.log     # Logs API
│   │   ├── business_YYYY-MM-DD.log # Logs métier
│   │   └── errors_YYYY-MM-DD.log  # Logs d'erreurs
│   └── README_LOGGING.md          # Documentation logging
│
├── Tests
│   ├── test_app.py                # Tests structure
│   ├── test_functionality.py      # Tests fonctionnels
│   └── api/v1/                    # Tests API
│       ├── test_products.py
│       ├── test_stores.py
│       └── test_reports.py
│
├── Documentation
│   ├── docs/
│   │   ├── adr-003-flask.md       # Décision architecture
│   │   ├── adr-004-architecture-mvc.md
│   │   ├── docker-deployment.md   # Guide déploiement
│   │   ├── logging.md             # Documentation logging
```

```
│   │   ├── openapi.json            # Spécification API
│   │   └── UML/                    # Diagrammes UML
│   └── README.md                   # Ce fichier
│
└── Configuration
    ├── requirements.txt            # Dépendances Python
    ├── scripts/
    │   └── init-db.sql            # Initialisation DB
    └── src/
        ├── db.py                   # Configuration DB
        └── create_db.py            # Données de démo
```

Installation et Configuration

Déploiement Rapide (Docker)

```
# 1. Cloner le projet
git clone <repository-url>
cd log430-labo-03

# 2. Initialiser l'environnement
make init

# 3. Démarrer tous les services
make up

# 4. Vérifier le statut
make status
```

Services disponibles :

- **Application Web** : http://localhost:8080
- **API REST** : http://localhost:8000
- **Documentation API** : http://localhost:8000/docs

Développement Local

1. Prérequis

- Python 3.9+
- PostgreSQL 15+
- Docker & Docker Compose (optionnel)

2. Installation

```
# Créer l'environnement virtuel
python -m venv venv
```

```
source venv/bin/activate  # Linux/Mac
# ou venv\Scripts\activate  # Windows

# Installer les dépendances
pip install -r requirements.txt
```

3. Configuration

Créer le fichier `.env` :

```
DATABASE_URL=postgresql://user:password@localhost:5432/store_db
SECRET_KEY=your-secret-key-here
API_TOKEN=your-api-token-here
LOG_LEVEL=INFO
POOL_SIZE=5
MAX_OVERFLOW=10
```

4. Base de données

```
# Initialiser la base avec des données de démo
python -m src.create_db
```

5. Lancement

```
# Terminal 1 : API FastAPI
uvicorn src.api.main:app --host 0.0.0.0 --port 8000 --reload

# Terminal 2 : Application Flask
python -m src.app.run
```

Configuration Avancée

Variables d'Environnement

```
# Base de données
DATABASE_URL=postgresql://user:password@host:port/database
POOL_SIZE=5
MAX_OVERFLOW=10

# Sécurité
SECRET_KEY=your-secret-key
API_TOKEN=your-api-token
```

```
# Logging
LOG_LEVEL=INFO  # DEBUG, INFO, WARNING, ERROR, CRITICAL

# Serveur
HOST=0.0.0.0
API_PORT=8000
WEB_PORT=8080
FLASK_ENV=production
```

Configuration de Production

```
# Optimisations de performance
POOL_SIZE=20
MAX_OVERFLOW=30
WORKERS=4

# Sécurité renforcée
FLASK_ENV=production
DEBUG=False
TESTING=False
```

Tests

Tests Automatisés

Le projet inclut une suite complète de tests automatisés couvrant tous les composants.

```
# Exécuter tous les tests
make test

# Tests spécifiques
python -m pytest tests/test_app.py -v
python -m pytest tests/api/v1/ -v

# Tests avec couverture
python -m pytest --cov=src tests/
```

Types de Tests

1. Tests Unitaires

- **Models** : Validation des modèles SQLAlchemy
- **Services** : Logique métier des domaines
- **Repositories** : Accès aux données

2. Tests d'Intégration

- **API Endpoints** : Tests complets des endpoints REST
- **Database** : Tests de persistance
- **Authentication** : Tests de sécurité

3. Tests Fonctionnels

- **Workflows** : Scénarios utilisateur complets
- **Business Logic** : Règles métier
- **Error Handling** : Gestion d'erreurs

Coverage Report

```
Name                                        Stmts   Miss  Cover
-------------------------------------------------------------
src/api/main.py                              45      2    96%
src/api/v1/domain/products/services.py   78      5    94%
src/api/v1/domain/stores/services.py     82      6    93%
src/api/v1/domain/reporting/services.py  65      4    94%
src/app/controllers/                         234     12    95%
-------------------------------------------------------------
TOTAL                                       1247     67    95%
```

Utilisation

Interface Web (Flask)

Dashboard Principal

- **Vue d'ensemble** : KPIs globaux, alertes, tendances
- **Navigation** : Accès rapide aux 5 magasins
- **Monitoring** : Statut en temps réel des caisses

Gestion des Ventes

```
1. Sélectionner un magasin
2. Choisir une caisse disponible
3. Rechercher et ajouter des produits
4. Finaliser la vente
5. Imprimer le reçu
```

Rapports Stratégiques

- **Performance par magasin** : CA, marge, rotation stock

- **Top produits** : Ventes, popularité, rentabilité
- **Analyse temporelle** : Tendances, saisonnalité
- **Alertes stock** : Ruptures, surstocks, réapprovisionnement

API RESTful (FastAPI)

Authentification

```
import requests

headers = {
    "Authorization": "Bearer your-api-token",
    "Content-Type": "application/json"
}
```

Exemples d'Utilisation

Gestion des Produits

```
# Créer un produit
product_data = {
    "nom": "Nouveau Produit",
    "prix": 29.99,
    "categorie_id": 1,
    "description": "Description du produit"
}
response = requests.post(
    "http://localhost:8000/api/v1/products",
    json=product_data,
    headers=headers
)

# Lister les produits avec filtres
params = {
    "categorie_id": 1,
    "prix_min": 10.0,
    "prix_max": 50.0,
    "limit": 20
}
response = requests.get(
    "http://localhost:8000/api/v1/products",
    params=params,
    headers=headers
)
```

Gestion des Magasins

```
# Obtenir les détails d'un magasin
response = requests.get(
    "http://localhost:8000/api/v1/stores/1",
    headers=headers
)

# Consulter le stock d'un magasin
response = requests.get(
    "http://localhost:8000/api/v1/stores/1/stock",
    headers=headers
)
```

Rapports

```
# Générer un rapport de ventes
params = {
    "date_debut": "2024-01-01",
    "date_fin": "2024-12-31",
    "magasin_id": 1
}
response = requests.get(
    "http://localhost:8000/api/v1/reports/sales",
    params=params,
    headers=headers
)
```

Architecture Load Balancée

Démarrage avec Load Balancer

```
# Démarrer l'architecture 3 instances + cache + monitoring
docker-compose -f docker-compose.loadbalanced.yml -f docker-
compose.monitoring.yml up -d

# Vérifier l'état des services
docker ps

# Vérifier la santé du load balancer
curl http://localhost:8000/health
```

Tests de Performance

```

```

```
# Test de charge progressive (recommandé)
k6 run k6-tests/loadbalanced-stress-test.js

# Test simple pour vérifier le fonctionnement
k6 run --vus 10 --duration 30s k6-tests/simple-stress-test.js

# Monitoring en temps réel pendant les tests
# Grafana: http://localhost:3000
# Prometheus: http://localhost:9090
```

Gestion du Cache Redis

```
# Statistiques du cache
curl -H "X-API-Token:
9645524dac794691257cb44d61ebc8c3d5876363031ec6f66fbd31e4bf85cd84" \
     http://localhost:8000/api/v1/cache/stats

# Vider le cache
curl -X DELETE -H "X-API-Token:
9645524dac794691257cb44d61ebc8c3d5876363031ec6f66fbd31e4bf85cd84" \
     http://localhost:8000/api/v1/cache/clear

# Test de performance du cache
time curl -H "X-API-Token:
9645524dac794691257cb44d61ebc8c3d5876363031ec6f66fbd31e4bf85cd84" \
          http://localhost:8000/api/v1/products/
```

Monitoring et Logs

Dashboards Grafana

- **API Load Balancer Performance** : Vue d'ensemble load balancing
- **Instance Health Status** : État de santé de chaque instance
- **Cache Performance** : Métriques Redis et hit ratio
- **System Performance** : CPU, mémoire, réseau

Métriques Clés à Surveiller

```
# Status des instances
curl http://localhost:9090/api/v1/query?query=up{job="api-instances"}

# Hit ratio du cache
curl http://localhost:9090/api/v1/query?query=cache_hit_ratio

# Latence P95
curl http://localhost:9090/api/v1/query?
```

```
query=histogram_quantile(0.95,rate(http_request_duration_seconds_bucket[
5m]))

# Taux d'erreur
curl http://localhost:9090/api/v1/query?
query=rate(http_requests_total{status=~"5.."}[5m])
```

Consultation des Logs

```
# Logs en temps réel
make logs

# Logs spécifiques
make logs-api    # API FastAPI
make logs-web    # Application Flask

# Logs par fichier
tail -f logs/api_2024-06-21.log
tail -f logs/business_2024-06-21.log
tail -f logs/errors_2024-06-21.log

# Logs des instances individuelles
docker logs log430-api-1 --tail 50
docker logs log430-api-2 --tail 50
docker logs log430-api-3 --tail 50
docker logs log430-nginx --tail 50
```

Métriques de Performance

- **Temps de réponse** : Automatiquement ajouté aux headers HTTP
- **Throughput** : Requêtes traitées par seconde
- **Distribution de charge** : Répartition entre instances
- **Cache hit ratio** : Efficacité du cache Redis
- **Erreurs** : Taux d'erreur et types d'exceptions
- **Ressources** : Utilisation CPU/Mémoire des containers

Technologies Utilisées

Backend

- **Python 3.9+** : Langage principal
- **Flask 3.0.0** : Framework web pour l'interface utilisateur
- **FastAPI 0.104.1** : Framework API moderne et performant
- **SQLAlchemy 2.0** : ORM pour la gestion des données
- **Pydantic** : Validation et sérialisation des données
- **Gunicorn** : Serveur WSGI/ASGI de production

Frontend

- **Jinja2** : Moteur de templates
- **Bootstrap 5** : Framework CSS responsive
- **JavaScript ES6+** : Interactions côté client
- **Chart.js** : Graphiques et visualisations

Base de Données

- **PostgreSQL 15** : Base de données relationnelle
- **psycopg2** : Adaptateur PostgreSQL pour Python
- **Connection Pooling** : Optimisation des connexions

DevOps & Déploiement

- **Docker** : Containerisation
- **Docker Compose** : Orchestration multi-containers
- **Gunicorn** : Serveur de production
- **Health Checks** : Surveillance des services

Load Balancing & High Availability

- **Nginx** : Load balancer et reverse proxy
- **Round Robin** : Distribution équitable des requêtes
- **Health Checks** : Surveillance automatique des instances
- **Failover** : Basculement automatique en cas de panne
- **Connection Pooling** : Optimisation des connexions réseau

Cache & Performance

- **Redis 7** : Cache distribué en mémoire
- **LRU Eviction** : Politique d'éviction intelligente
- **TTL Strategy** : Temps d'expiration flexibles
- **Cache Metrics** : Monitoring hit/miss ratio
- **Serialization** : Support objets Pydantic et JSON

Logging & Monitoring

- **Python Logging** : Système de logs natif
- **Rotating File Handler** : Rotation automatique
- **JSON Logging** : Format structuré pour les logs métier
- **Custom Formatters** : Formatage personnalisé
- **Prometheus** : Collecte et stockage de métriques
- **Grafana** : Dashboards et visualisation
- **Custom Metrics** : Métriques applicatives personnalisées
- **Alerting** : Notifications automatiques d'incidents

Testing & Qualité

- **pytest** : Framework de tests
- **pytest-cov** : Couverture de code
- **Black** : Formatage de code
- **Flake8** : Analyse statique
- **K6** : Tests de performance et charge
- **Load Testing** : Tests de montée en charge progressive
- **Stress Testing** : Tests de résistance haute charge
- **Performance Metrics** : Analyse latence et throughput

Sécurité

- **Token-based Auth** : Authentification par tokens
- **Environment Variables** : Configuration sécurisée
- **Non-root Containers** : Containers sécurisés
- **Input Validation** : Validation stricte des entrées

Support et Contribution

Issues et Bugs

Pour signaler un bug ou demander une fonctionnalité, veuillez utiliser le système d'issues du projet.

Documentation

- **API Documentation** : http://localhost:8000/docs
- **Docker Guide** : DOCKER_README.md

Licence

Ce projet est développé dans le cadre du cours LOG430 à l'ÉTS.

Version : 3.0.0
Dernière mise à jour : Juin 2025
Auteur : Louqman Masbahi