ПРОТОТИП РОЗПОДІЛЕНОЇ СИСТЕМИ ІНТЕРНЕТ-МАГАЗИНУ

ВИКОНАВ ОЛЕГ РИСНЮК

Архітектура Системи

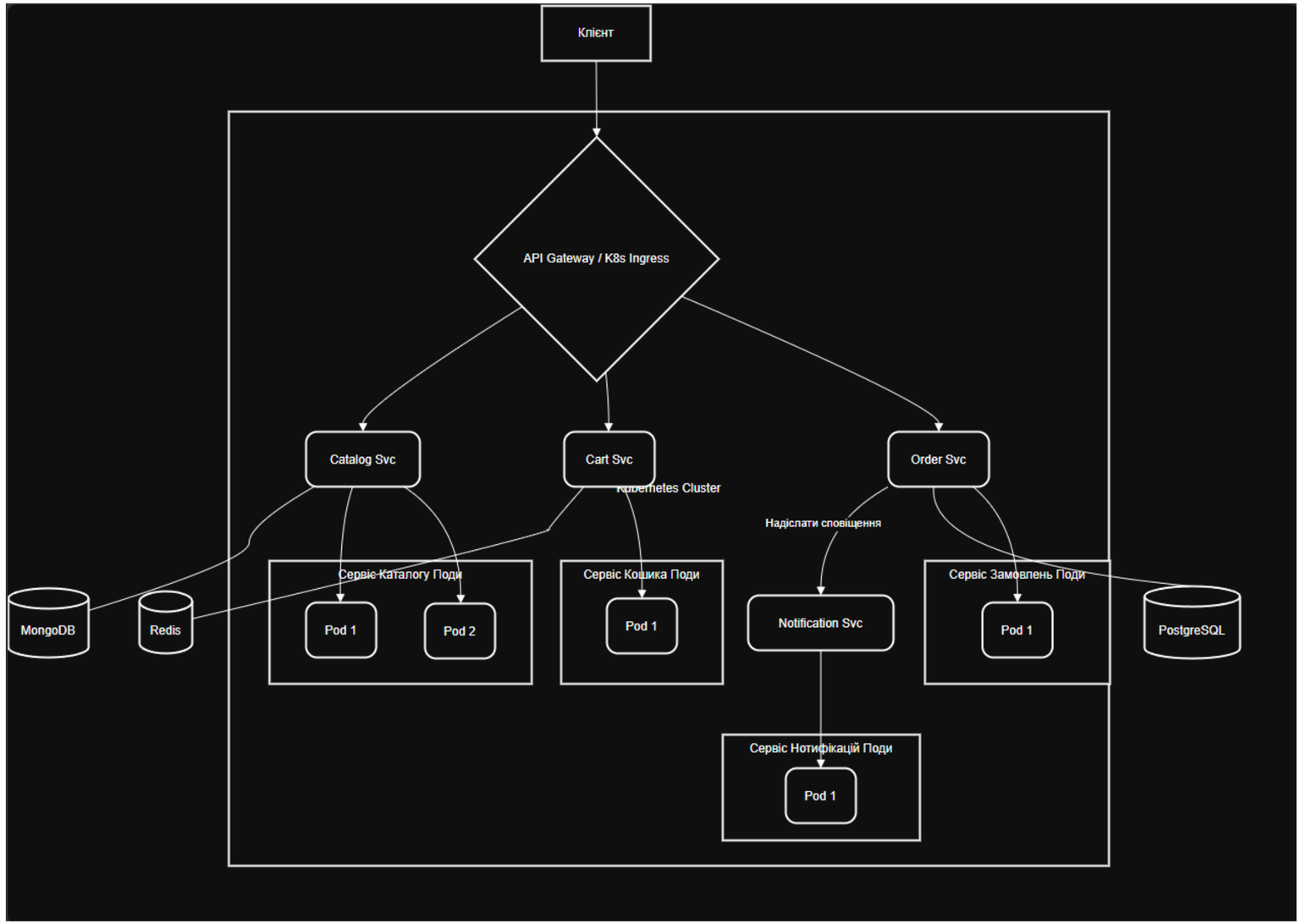
Система побудована на основі мікросервісної архітектури. Кожен ключовий функціональний блок виділено в окремий, незалежний сервіс. Це забезпечує гнучкість, масштабованість та надійність.

Основні компоненти:

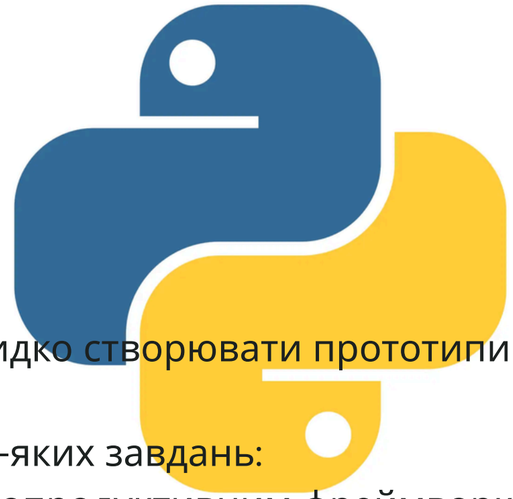
- Сервіс Каталогу Товарів:
  - Функції: Зберігання інформації про товари (назви, описи, ціни, зображення, категорії), надання API для пошуку та отримання даних про товари.
  - Технології: **Python (FastAPI)**, **MongoDB** (для гнучкої структури даних товарів).
- Сервіс Кошика:
  - Функції: Управління кошиками користувачів (додавання, видалення товарів, отримання вмісту). Кошик є тимчасовим сховищем.
  - Технології: **Python (FastAPI)**, **Redis** (для швидкого доступу до даних сесій/кошиків).
- Сервіс Замовлень:
  - Функції: Створення та обробка замовлень, імітація обробки платежів, зберігання історії замовлень.
  - Технології: **Python (FastAPI)**, **PostgreSQL** (для транзакційності та надійного зберігання структурованих даних замовлень).
- Сервіс Нотифікацій:
  - Функції: Відправка сповіщень користувачам (наприклад, підтвердження замовлення). У прототипі — імітація через логування.
  - Технології: **Python (FastAPI).**

Тип комунікації: REST API. Всі сервіси надають HTTP-ендпоїнти. Взаємодія між сервісами відбувається через ці API. Це забезпечує слабку зв'язаність та незалежність сервісів.

Схема Взаємодії

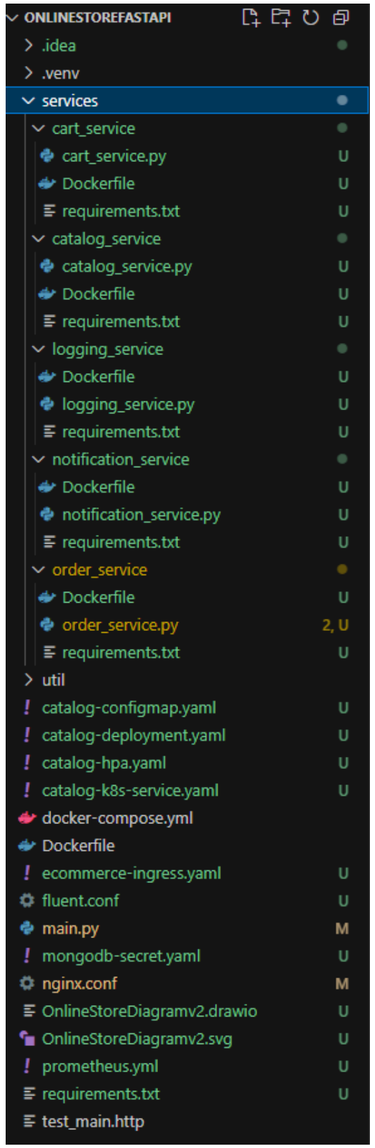


Чому Python



- **Швидкість Розробки:** Чистий синтаксис та динамічна типізація дозволяють швидко створювати прототипи та ітеративно розвивати функціонал.
- **Багата Екосистема:** Існує величезна кількість бібліотек та фреймворків для будь-яких завдань:
  - Веб-фреймворки: FastAPI (обраний тут), Flask, Django. FastAPI є сучасним, високопродуктивним фреймворком, ідеальним для створення API, з вбудованою асинхронністю та документацією.
  - Робота з Базами Даних: Драйвери для PostgreSQL (psycopg2), Redis (redis-py), MongoDB (pymongo), а також ORM (SQLAlchemy).
  - HTTP-клієнти: requests для легкої комунікації між сервісами.
- **Масштабованість**: Хоча Python є інтерпретованою мовою, сучасні фреймворки (особливо асинхронні, як FastAPI) та архітектурні підходи (мікросервіси) дозволяють будувати високопродуктивні та масштабовані системи. Кожен сервіс можна масштабувати незалежно.
- **Спільнота та Підтримка**: Велика та активна спільнота, що означає легкий доступ до документації, навчальних матеріалів та готових рішень.
- **Універсальність**: Python підходить не тільки для веб-розробки, але й для аналізу даних, машинного навчання (що може бути корисним для майбутніх функцій, як-от рекомендації товарів), автоматизації та іншого.

Структура проекту



- Ключові Компоненти (Мікросервіси):
  - Сервіс Каталогу Товарів
  - Сервіс Кошика
  - Сервіс Замовлень
  - Сервіс Нотифікацій
- Тип Комунікації: REST API (HTTP/JSON) між сервісами.
- Точка Входу: API Gateway / Kubernetes Ingress (для маршрутизації, балансування, безпеки).

Код Сервісів

Будемо використовувати FastAPI для всіх сервісів.

Сервіс Каталогу (catalog_service)
База даних: MongoDB

- Основні API Ендпоїнти :GET /products – отримати список товарів.
- GET /products/{product_id} – отримати деталі товару.
- POST /products – додати новий товар (адмін).
- GET /search?q={query} – пошук товарів.

```python
from fastapi import FastAPI, HTTPException, Query, Path, Body, status
from pymongo import MongoClient, ASCENDING, DESCENDING
from pymongo.errors import ConnectionFailure, OperationFailure, DuplicateKeyError
from pydantic import BaseModel, Field, HttpUrl
from typing import List, Optional, Dict, Any, Union
import os
import httpx
import datetime
import logging
from prometheus_fastapi_instrumentator import Instrumentator # Для Prometheus
from tenacity import retry, stop_after_attempt, wait_exponential, RetryError, retry_if_exception_type # Для Retry
import pybreaker # Для Circuit Breaker

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__) # Логер для цього модуля

# --- Моделі Pydantic ---
class Price(BaseModel):
    amount: float = Field(..., gt=0, description="Сума ціни", example=129.99)
    currency: str = Field(..., min_length=3, max_length=3, description="Код валюти (ISO 4217)", example="UAH")

class Attribute(BaseModel):
    key: str = Field(..., description="Назва атрибуту", example="Колір")
    value: Union[str, int, float, bool] = Field(..., description="Значення атрибуту", example="Чорний")

class ProductBase(BaseModel):
    name: str = Field(..., min_length=1, max_length (parameter) max_length: Any ", example="Смартфон SuperX 12 Pro")
    description_short: Optional[str] = Field(None, max_length=500, description="Короткий опис товару", example="Потужний смартфон з неймовірною к
    description_long: Optional[str] = Field(None, description="Повний опис товару з усіма деталями.")
    price: Price = Field(..., description="Ціна товару")
    categories: List[str] = Field(default_factory=list, description="Список ID або назв категорій, до яких належить товар", example=["електроніка
    brand: Optional[str] = Field(None, max_length=100, description="Бренд товару", example="TechCorp")
    stock_quantity: int = Field(0, ge=0, description="Кількість товару на складі", example=150)
    images: List[HttpUrl] = Field(default_factory=list, description="Список URL зображень товару", example=["https://example.com/image1.jpg"])
    attributes: List[Attribute] = Field(default_factory=list, description="Список атрибутів товару")
    is_active: bool = Field(True, description="Чи активний товар (доступний для продажу)")
    tags: List[str] = Field(default_factory=list, description="Теги для пошуку та фільтрації", example=["новинка", "топ продажів"])
```

Код Сервісів

Будемо використовувати FastAPI для всіх сервісів.

Сервіс Кошика (cart_service)
База даних: Redis

- Основні API Ендпоїнти :POST /cart/{user_id}/items – додати товар до кошика.
- GET /cart/{user_id} – отримати вміст кошика.
- DELETE /cart/{user_id}/items/{product_id} – видалити товар з кошика.
- DELETE /cart/{user_id} – очистити кошик.

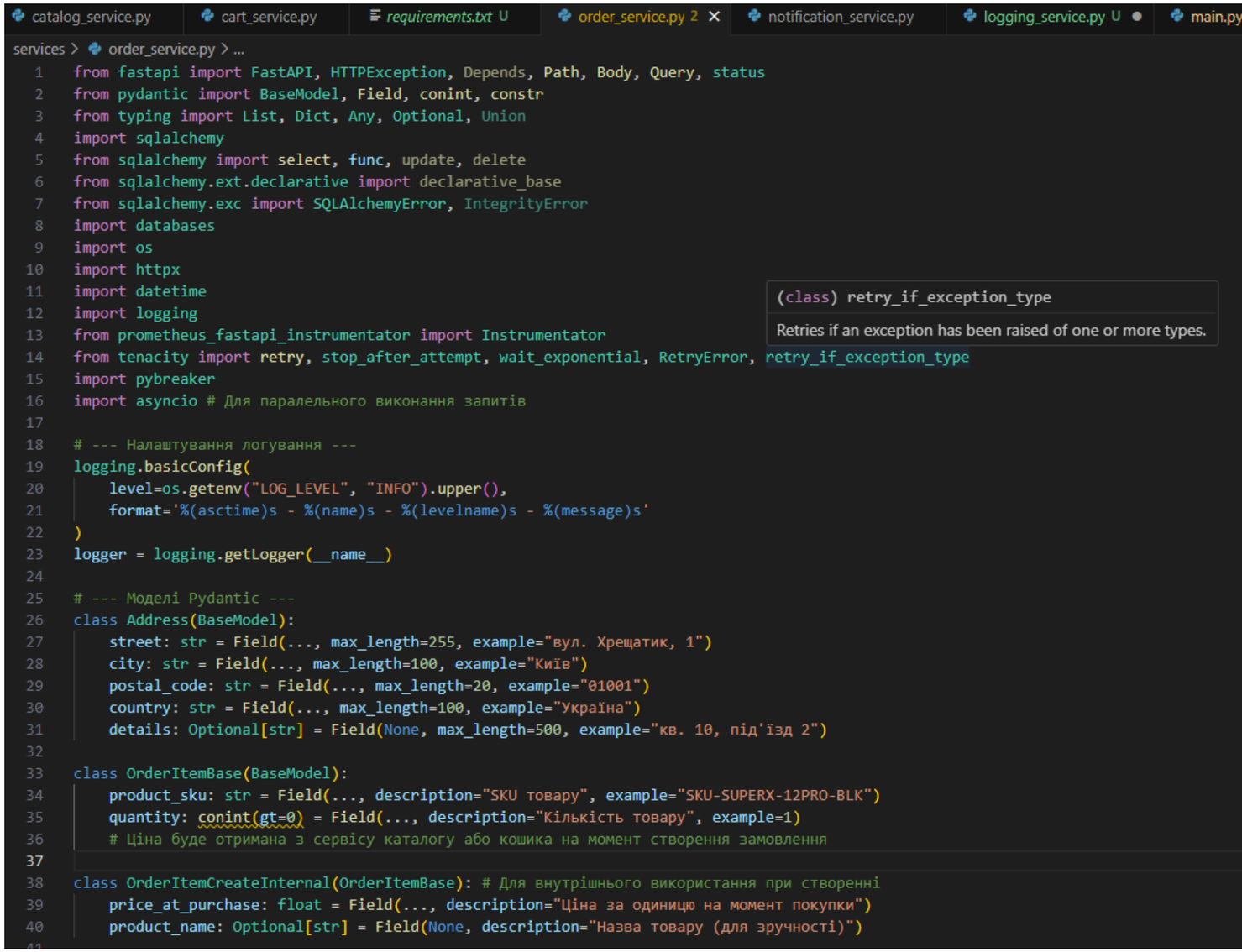
```python
from fastapi import FastAPI, HTTPException, Path, status, Depends, Body, Query
from pydantic import BaseModel, Field, validator
import redis
import redis.asyncio as aioredis # Для асинхронної роботи з Redis
import json
from typing import List, Dict, Any, Optional
import os
import httpx
import datetime
import logging
from prometheus_fastapi_instrumentator import Instrumentator
from tenacity import retry, stop_after_attempt, wait_exponential, RetryError, retry_if_exception_type
import pybreaker

# --- Налаштування логування ---
logging.basicConfig(
    level=os.getenv("LOG_LEVEL", "INFO").upper(),
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# --- Моделі Pydantic ---
class CartItemBase(BaseModel):
    product_sku: str = Field(..., description="SKU товару", example="SKU-SUPERX-12PRO-BLK")
    quantity: int = Field(..., gt=0, le=100, description="Кількість товару (від 1 до 100)", example=2)

class CartItemCreate(CartItemBase):
    pass

class CartItemUpdate(BaseModel):
    quantity: int = Field(..., gt=0, le=100, description="Нова кількість товару (від 1 до 100)", example=3)

class CartItemResponse(CartItemBase):
    name: Optional[str] = Field(None, description="Назва товару (з каталогу)", example="Смартфон SuperX 12 Pro")
    price_per_item: Optional[float] = Field(None, description="Ціна за одиницю товару (з каталогу)", example=34999.99)
    item_total_price: Optional[float] = Field(None, description="Загальна вартість цієї позиції (кількість * ціна)", example=699
```

Код Сервісів

Будемо використовувати FastAPI для всіх сервісів.

Сервіс Замовлень (order_service)
База даних: PostgreSQL

- Основні API Ендпоїнти (Приклади):POST /orders (з user_id у тілі/токені) – створити нове замовлення.
- GET /orders/{user_id} – отримати історію замовлень користувача.
- GET /orders/details/{order_id} – отримати деталі конкретного замовлення.




```python
from fastapi import FastAPI, HTTPException, Depends, Path, Body, Query, status
from pydantic import BaseModel, Field, conint, constr
from typing import List, Dict, Any, Optional, Union
import sqlalchemy
from sqlalchemy import select, func, update, delete
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.exc import SQLAlchemyError, IntegrityError
import databases
import os
import httpx
import datetime
import logging
from prometheus_fastapi_instrumentator import Instrumentator
from tenacity import retry, stop_after_attempt, wait_exponential, RetryError, retry_if_exception_type
import pybreaker
import asyncio # Для паралельного виконання запитів

# --- Налаштування логування ---
logging.basicConfig(
    level=os.getenv("LOG_LEVEL", "INFO").upper(),
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# --- Моделі Pydantic ---
class Address(BaseModel):
    street: str = Field(..., max_length=255, example="вул. Хрещатик, 1")
    city: str = Field(..., max_length=100, example="Київ")
    postal_code: str = Field(..., max_length=20, example="01001")
    country: str = Field(..., max_length=100, example="Україна")
    details: Optional[str] = Field(None, max_length=500, example="кв. 10, під'їзд 2")

class OrderItemBase(BaseModel):
    product_sku: str = Field(..., description="SKU товару", example="SKU-SUPERX-12PRO-BLK")
    quantity: conint(gt=0) = Field(..., description="Кількість товару", example=1)
    # Ціна буде отримана з сервісу каталогу або кошика на момент створення замовлення

class OrderItemCreateInternal(OrderItemBase): # Для внутрішнього використання при створенні
    price_at_purchase: float = Field(..., description="Ціна за одиницю на момент покупки")
    product_name: Optional[str] = Field(None, description="Назва товару (для зручності)")
```

Код Сервісів

Будемо використовувати FastAPI для всіх сервісів.

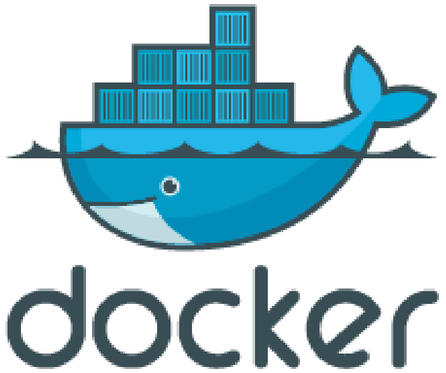
Сервіс Нотифікацій (notification_service)

catalog_service.py | cart_service.py | requirements.txt U | order_service.py 2 | notification_service.py | logging_service.py U | main.py M | catalog-deployment.yaml U

services > notification_service.py > ...

```python
from fastapi import FastAPI, HTTPException, Body, status
from pydantic import BaseModel, EmailStr, Field, validator
from typing import Dict, Any, Optional, Literal, Tuple
import os
import logging
import httpx # Для відправки логів та, можливо, SMS
import datetime
from prometheus_fastapi_instrumentator import Instrumentator
from tenacity import retry, stop_after_attempt, wait_exponential, RetryError, retry_if_exception_type
import pybreaker
import smtplib # Для імітації відправки email
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import asyncio

# --- Налаштування логування ---
# Логи цього сервісу будуть йти в stdout/stderr,
# а також він буде надсилати свої операційні логи до центрального Сервісу Логування.
logging.basicConfig(
    level=os.getenv("LOG_LEVEL", "INFO").upper(),
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__) # Логер для внутрішніх потреб цього сервісу

# --- Моделі Pydantic ---
class NotificationPayload(BaseModel):
    type: Literal['email', 'sms'] = Field(..., description="Тип сповіщення: 'email' або 'sms'")
    recipient: str = Field(..., description="Отримувач: email-адреса для 'email', номер телефону для 'sms'")
    template_id: Optional[str] = Field(None, description="Ідентифікатор шаблону для повідомлення (наприклад, 'order_confirmation', 'password_reset')", example="order_confi
    context: Dict[str, Any] = Field(default_factory=dict, description="Дані для заповнення шаблону (ключ-значення)", example={"order_id": "123XYZ", "user_name": "Іван"})
    subject: Optional[str] = Field(None, description="Тема повідомлення (обов'язково для email, якщо не використовується шаблон з темою)", example="Ваше замовлення підтвер
    body: Optional[str] = Field(None, description="Тіло повідомлення (використовується, якщо немає template_id або шаблон не знайдено)")

    @validator('recipient')
    def recipient_validator(cls, v, values):
        type_ = values.get('type')
```

Контейнеризація з Docker



Для кожного сервісу створюється Dockerfile

```dockerfile
# Використовуємо офіційний, легкий образ Python версії 3.9.
FROM python:3.9-slim AS builder

# Встановлення змінних середовища
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Встановлюємо робочу директорію всередині контейнера.
WORKDIR /app

COPY ./requirements.txt /app/

RUN pip install --upgrade pip && \
    pip install --no-cache-dir --trusted-host pypi.python.org -r requirements.txt

COPY ./order_service.py /app/

RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser
# USER appuser # Переключення на користувача без root-прав (можна розкоментувати, якщо потрібно)

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

```

Оркестрація з Kubernetes (K8s)

Kubernetes дозволяє автоматизувати розгортання, масштабування та управління контейнеризованими додатками.
Основні кроки:

1. **Збірка та Публікація Образів:** Зібрати Docker-образи для кожного сервісу та завантажити їх у реєстр (наприклад, Docker Hub, Google Container Registry, AWS ECR).
2. **Створення K8s Маніфестів (YAML)**: Для кожного сервісу та бази даних потрібні маніфести.

catalog-configmap.yaml U
catalog-deployment.yaml U
catalog-hpa.yaml U
catalog-k8s-service.yaml U

catalog-k8s-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: product-catalog-service # Назва Service (буде DNS-іменем всеред
  labels:
    app: product-catalog-service
spec:
  type: ClusterIP
  selector:
    app: product-catalog-service # Service направляє трафік на Pod'и з
  ports:
    - protocol: TCP
      port: 80 # Порт, на якому Service слухає всередині кластера
      targetPort: 8000 # Порт контейнера, на який перенаправляється тра
```

catalog-hpa.yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: product-catalog-hpa
spec:
  scaleTargetRef: # Вказує на Deployment, який потрібно масштабувати
    apiVersion: apps/v1
    kind: Deployment
    name: product-catalog-service
  minReplicas: 2 # Мінімальна кількість реплік
  maxReplicas: 5 # Максимальна кількість реплік
  metrics: # Метрики, на основі яких приймається рішення про масштабування
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization # Цільове значення - відсоток використання запитаних ресурсів
        averageUtilization: 70 # Масштабувати, якщо середнє використання CPU перевищує 70%
  - type: Resource
    resource:
      name: memory
      target:
        type: AverageValue # Цільове значення - абсолютне середнє значення
        averageValue: 200Mi # Масштабувати, якщо середнє використання пам'яті перевищує 200Mi
```

Балансування Навантаження та Масштабованість в Kubernetes

Балансування Навантаження:

- **K**ubernetes Service типу ClusterIP (для внутрішньої комунікації) або LoadBalancer (для зовнішнього трафіку) автоматично розподіляє запити між доступними подами сервісу.
- Алгоритм за замовчуванням: Round Robin.

Масштабованість:

- Ручне масштабування: kubectl scale deployment <name> ––replicas=N.
- Автоматичне масштабування (Horizontal Pod Autoscaler – HPA):
  - Автоматично збільшує або зменшує кількість подів на основі метрик (CPU, пам'ять, кастомні метрики).
  - Приклад: Якщо середнє завантаження CPU подами Сервісу Каталогу перевищує 75% протягом 1 хвилини, HPA додає нову репліку.

Моніторинг Системи

Інструменти:

- Prometheus: Збір та зберігання часових рядів метрик.
- Grafana: Візуалізація метрик, створення дашбордів.
- Alertmanager: Налаштування сповіщень про критичні події.

Ключові Метрики для Моніторингу:

- **Загальносистемні** (K8s/Інфраструктура):
  - Використання CPU/RAM подами та нодами.
  - Мережевий трафік (вхідний/вихідний).
  - Стан дисків (для баз даних).
- **Метрики Додатків** (APM – Application Performance Monitoring):
  - Час відповіді API (Latency): p50, p90, p95, p99 для кожного ендпоінту.
  - Кількість запитів (Throughput): RPM/RPS для кожного сервісу.
  - Рівень помилок: Кількість HTTP 4xx, 5xx помилок.
- **Сервіс-специфічні**:
  - Каталог: Час пошуку, кількість запитів до MongoDB.
  - Кошик: Кількість активних кошиків, час операцій з Redis.
  - Замовлення: Кількість створених замовлень/хв, % успішних "платежів".
  - Нотифікації: Черга сповіщень, % успішно відправлених.
- **Метрики Баз Даних**: Навантаження, кількість підключень, час виконання запитів, реплікація.

Тестування Системи – Стратегія

Unit-тести (PyTest):

- **Мета**: Перевірка коректності окремих функцій, класів та модулів кожного сервісу.
- **Приклад:** Тестування логіки валідації даних товару в Сервісі Каталогу.

Інтеграційні тести:

- **Мета:** Перевірка взаємодії між мікросервісами.
- **Приклад**: Тестування повного циклу створення замовлення (Кошик -> Каталог -> Замовлення -> Нотифікації). Використовується docker-compose для підняття всіх сервісів.

API-тести (End-to-End):

- **Мета:** Перевірка роботи системи з точки зору клієнта через публічні API ендпоїнти.
- **Інструменти:** Postman, Newman, або кастомні скрипти на Python з requests.

Навантажувальні тести:

- **Мета:** Визначення продуктивності та стабільності системи під високим навантаженням.
- **Інструменти:** k6, Locust, Apache JMeter.
- **Сценарії:** Симуляція великої кількості користувачів, що переглядають товари, додають в кошик, роблять замовлення.

Висновки

- **Розроблений прототип** успішно демонструє реалізацію розподіленої системи інтернет–магазину з використанням сучасних підходів.

- Мікросервісна архітектура на Python (FastAPI) з Docker та Kubernetes забезпечує:
  - **Гнучкість для подальшого розвитку.**
  - **Масштабованість для обробки зростаючого навантаження.**
  - **Керованість та надійність завдяки оркестрації.**

- **Система готова до впровадження додаткового функціоналу та оптимізації на основі результатів тестування та моніторингу.**

Дякую за увагу!

ОЛЕГ РИСНЮК