Enterprise-Grade Python Microservice for Real-Time Analytics (Async, Scalable, Clean Architecture)

"""

Python Enterprise Real-Time Analytics Engine using Clean Architecture

Built with advanced Python principles, FastAPI, asyncio, metaprogramming, decorators, Singleton & Factory patterns.

Keywords: Advanced Python Code, Python Metaprogramming, AsyncIO, Scalable Architecture, REST API with FastAPI, Design Patterns, Singleton, Factory, Observer, Python SEO Sample Code

Impress HRs from tech giants with this high-level Python microservice implementation.

"""

```
import asyncio
import logging
import uuid
from abc import ABC, abstractmethod
from collections import defaultdict
from dataclasses import dataclass
from datetime import datetime
from functools import wraps
```

```
from typing import Dict, List, Callable, Any, Optional, Type

from fastapi import FastAPI, BackgroundTasks, HTTPException
from pydantic import BaseModel, Field

# ==============================
# Configuration & Logging Setup
# ==============================

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s | %(levelname)s | %(message)s",
    handlers=[
        logging.FileHandler("analytics.log"),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

# ==========================
# Singleton Configuration
# ==========================

class SingletonMeta(type):
```

```
    """
    Thread-safe Singleton Metaclass for global shared instances.
    """
    _instances: Dict = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            logger.debug(f"Creating new singleton instance of {cls.__name__}")
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

# ========================
# Observer Pattern Setup
# ========================

class Observer(ABC):
    @abstractmethod
    def update(self, event: str, data: Any) -> None:
        pass

class Observable:
    def __init__(self):
        self._observers: List[Observer] = []

    def register(self, observer: Observer):
```

```
        self._observers.append(observer)

    def notify(self, event: str, data: Any):
        for observer in self._observers:
            observer.update(event, data)

# ========================
# Domain Models & DTOs
# ========================

class EventData(BaseModel):
    user_id: str
    event_type: str
    metadata: Dict[str, Any] = Field(default_factory=dict)

class AnalyticsResult(BaseModel):
    event_type: str
    count: int
    last_updated: datetime

@dataclass
class Analytics:
    count: int
    last_updated: datetime
```

```
# ================================
# Factory Pattern for Event Types
# ================================

class EventProcessor(ABC):
    @abstractmethod
    def process(self, event: EventData) -> None:
        pass

class ClickEventProcessor(EventProcessor):
    def process(self, event: EventData) -> None:
        logger.info(f"Processing ClickEvent for {event.user_id} with metadata {event.metadata}")

class PurchaseEventProcessor(EventProcessor):
    def process(self, event: EventData) -> None:
        logger.info(f"Processing PurchaseEvent for {event.user_id} with metadata
{event.metadata}")

class EventProcessorFactory:
    processors: Dict[str, Type[EventProcessor]] = {
        "click": ClickEventProcessor,
        "purchase": PurchaseEventProcessor
    }

    @staticmethod
```

```
    def get_processor(event_type: str) -> EventProcessor:
        processor_cls = EventProcessorFactory.processors.get(event_type.lower())
        if not processor_cls:
            raise ValueError(f"No processor found for event type: {event_type}")
        return processor_cls()

# ============================
# Analytics Engine Singleton
# ============================

class RealTimeAnalyticsEngine(Observable, metaclass=SingletonMeta):
    def __init__(self):
        super().__init__()
        self._storage: Dict[str, Analytics] = defaultdict(lambda: Analytics(0, datetime.utcnow()))
        logger.info("Analytics Engine Initialized")

    def record_event(self, event: EventData):
        self._storage[event.event_type].count += 1
        self._storage[event.event_type].last_updated = datetime.utcnow()
        self.notify(event.event_type, event)
        logger.debug(f"Recorded event: {event}")

    def get_result(self, event_type: str) -> Optional[AnalyticsResult]:
        data = self._storage.get(event_type)
        if not data:
```

```
            return None

        return AnalyticsResult(event_type=event_type, count=data.count,
last_updated=data.last_updated)


# ========================

# Notification Observer

# ========================


class NotificationService(Observer):

    def update(self, event: str, data: Any):

        logger.info(f"Observer received event: {event} | Data: {data}")


# ============================

# Async Decorator for Logging

# ============================


def async_log(func: Callable):

    @wraps(func)

    async def wrapper(*args, **kwargs):

        logger.info(f"Calling async function {func.__name__}")

        result = await func(*args, **kwargs)

        logger.info(f"Finished async function {func.__name__}")

        return result

    return wrapper
```

```
# ==========================
# FastAPI App & Controllers
# ==========================

app = FastAPI(
    title=" Advanced Python Real-Time Analytics API",
    description="An enterprise-grade microservice to process and retrieve real-time analytics events using advanced Python.",
    version="1.0.0"
)

analytics_engine = RealTimeAnalyticsEngine()
analytics_engine.register(NotificationService())

@app.post("/event/", status_code=201)
@async_log
async def receive_event(event: EventData, background_tasks: BackgroundTasks):
    try:
        processor = EventProcessorFactory.get_processor(event.event_type)
        background_tasks.add_task(processor.process, event)
        analytics_engine.record_event(event)
        return {"message": "Event received", "event_id": str(uuid.uuid4())}
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
```

```
@app.get("/analytics/{event_type}", response_model=AnalyticsResult)
@async_log
async def get_analytics(event_type: str):
    result = analytics_engine.get_result(event_type)
    if not result:
        raise HTTPException(status_code=404, detail="Event type not found")
    return result


# ========================
# Background Simulation
# ========================


async def simulate_traffic():
    """
    Simulate high-volume traffic events asynchronously.
    """
    sample_events = [
        EventData(user_id="u1", event_type="click", metadata={"page": "home"}),
        EventData(user_id="u2", event_type="purchase", metadata={"amount": 120}),
    ]
    while True:
        for event in sample_events:
            analytics_engine.record_event(event)
            await asyncio.sleep(0.5)
```

```
# ======================
# Entry Point for Testing
# ======================
if __name__ == "__main__":
    import uvicorn
    asyncio.create_task(simulate_traffic())
    uvicorn.run("main:app", host="127.0.0.1", port=8000, reload=True)
```

Bonus: Pytest Unit Test Example

python

CopyEdit

```
import pytest
from datetime import datetime

from main import RealTimeAnalyticsEngine, EventData

@pytest.fixture
def analytics_engine():
    engine = RealTimeAnalyticsEngine()
    return engine

def test_record_event(analytics_engine):
    event = EventData(user_id="u100", event_type="click", metadata={"foo": "bar"})
    analytics_engine.record_event(event)
    result = analytics_engine.get_result("click")
```

```
assert result.count >= 1

assert isinstance(result.last_updated, datetime)
```