Exemple de code

mardi 18 février 2025 13:48

```
# Structure du Projet : Plateforme de Test d'IAs
## 1. Architecture des Dossiers
```
ai-testing-platform/
├── backend/
│   ├── src/
│   │   ├── api/   # FastAPI routes
│   │   │   ├── __init__.py
│   │   │   ├── campaigns.py
│   │   │   ├── questions.py
│   │   │   ├── results.py
│   │   │   └── ai_providers.py
│   │   ├── core/   # Logique métier
│   │   │   ├── __init__.py
│   │   │   ├── config.py
│   │   │   ├── security.py
│   │   │   └── logging.py
│   │   ├── db/   # Gestion base de données
│   │   │   ├── __init__.py
│   │   │   ├── models.py
│   │   │   ├── schemas.py
│   │   │   └── session.py
│   │   ├── services/   # Services métier
│   │   │   ├── __init__.py
│   │   │   ├── campaign_service.py
│   │   │   ├── question_service.py
│   │   │   └── ai_service.py
│   │   ├── ai_providers/   # Intégration IAs
│   │   │   ├── __init__.py
│   │   │   ├── base.py
│   │   │   ├── chatgpt.py
│   │   │   ├── mistral.py
│   │   │   └── deepseek.py
│   │   └── utils/   # Utilitaires
│   │   ├── __init__.py
│   │   ├── token_counter.py
│   │   └── csv_handler.py
│   ├── tests/   # Tests unitaires et intégration
│   ├── alembic/   # Migrations DB
│   ├── requirements.txt
│   └── main.py
│
├── frontend/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── services/
│   │   └── utils/
│   ├── public/
│   └── package.json
│
└── docker/
    ├── docker-compose.yml
    ├── Dockerfile.backend
    └── Dockerfile.frontend
```
```

## 2. Modèles de Base de Données
```python
# backend/src/db/models.py
from sqlalchemy import Column, Integer, String, DateTime, JSON, ForeignKey
from sqlalchemy.orm import relationship
from sqlalchemy.ext.declarative import declarative_base
Base = declarative_base()
class AIProvider(Base):
    __tablename__ = "ai_providers"

    id = Column(Integer, primary_key=True)
    name = Column(String, unique=True)
    description = Column(String)
    api_url = Column(String)
    module_path = Column(String)  # Chemin vers le module Python
    class_name = Column(String)   # Nom de la classe à instancier
    api_key = Column(String)      # Stocké de manière cryptée
    config = Column(JSON)         # Configuration spécifique à l'IA
    is_active = Column(Boolean, default=True)
    created_at = Column(DateTime)
    updated_at = Column(DateTime)
class Question(Base):
    __tablename__ = "questions"

    id = Column(Integer, primary_key=True)
    content = Column(String)
    category = Column(String)
    tags = Column(JSON)
    created_at = Column(DateTime)
    updated_at = Column(DateTime)
class Campaign(Base):
    __tablename__ = "campaigns"

    id = Column(Integer, primary_key=True)
    name = Column(String)
    description = Column(String)
    status = Column(String)  # pending, running, completed, failed
    config = Column(JSON)    # Configuration spécifique de la campagne
    created_at = Column(DateTime)
    started_at = Column(DateTime)
    completed_at = Column(DateTime)

    results = relationship("CampaignResult")
class CampaignResult(Base):
    __tablename__ = "campaign_results"

    id = Column(Integer, primary_key=True)
    campaign_id = Column(Integer, ForeignKey("campaigns.id"))
    question_id = Column(Integer, ForeignKey("questions.id"))
    ai_provider_id = Column(Integer, ForeignKey("ai_providers.id"))
    response = Column(String)
    tokens_count = Column(Integer)
    response_time = Column(Float)
    error = Column(String, nullable=True)
    created_at = Column(DateTime)
```
## 3. Gestion Dynamique des IAs
```python
# backend/src/ai_providers/base.py
from abc import ABC, abstractmethod
class BaseAIProvider(ABC):
    def __init__(self, api_key: str, config: dict):
        self.api_key = api_key
        self.config = config
    @abstractmethod
    async def generate_response(self, prompt: str) -> dict:
        """
        Doit retourner un dict avec:
        {
            'response': str,
            'tokens': int,
            'time': float
        }
        """
        pass
    @abstractmethod
    async def count_tokens(self, text: str) -> int:
        pass
# backend/src/ai_providers/chatgpt.py
from .base import BaseAIProvider
import openai
class ChatGPTProvider(BaseAIProvider):
    def __init__(self, api_key: str, config: dict):
        super().__init__(api_key, config)
        openai.api_key = api_key
    async def generate_response(self, prompt: str) -> dict:
        start_time = time.time()
        response = await openai.ChatCompletion.create(
            model=self.config.get('model', 'gpt-3.5-turbo'),
            messages=[{"role": "user", "content": prompt}]
        )
        end_time = time.time()

        return {
            'response': response.choices[0].message.content,
            'tokens': response.usage.total_tokens,
            'time': end_time - start_time
        }
```
## 4. Service de Gestion des IAs
```python
# backend/src/services/ai_service.py
import importlib
from src.db.models import AIProvider
from src.core.security import decrypt_api_key
class AIProviderService:
    def __init__(self, db_session):
        self.db_session = db_session
        self._providers_cache = {}
    async def get_provider(self, provider_id: int):
        if provider_id in self._providers_cache:
            return self._providers_cache[provider_id]
        provider_data = await
self.db_session.query(AIProvider).get(provider_id)
        if not provider_data:
            raise ValueError(f"Provider {provider_id} not found")
        # Import dynamique du module
        module = importlib.import_module(provider_data.module_path)
        provider_class = getattr(module, provider_data.class_name)

        # Décryptage de la clé API
        api_key = decrypt_api_key(provider_data.api_key)
        # Instanciation du provider
        provider = provider_class(api_key=api_key,
config=provider_data.config)
        self._providers_cache[provider_id] = provider

        return provider
    async def register_provider(self, provider_data: dict):
        """Enregistre un nouveau provider dans la base de données"""
        provider = AIProvider(
            name=provider_data['name'],
            description=provider_data['description'],
            api_url=provider_data['api_url'],
            module_path=provider_data['module_path'],
            class_name=provider_data['class_name'],
            api_key=encrypt_api_key(provider_data['api_key']),
            config=provider_data['config']
        )
        self.db_session.add(provider)
        await self.db_session.commit()
        return provider
```

## 5. Gestionnaire de Campagne

```python
# backend/src/services/campaign_service.py
class CampaignService:
    def __init__(self, db_session, ai_service):
        self.db_session = db_session
        self.ai_service = ai_service
    async def run_campaign(self, campaign_id: int):
        campaign = await self.db_session.query(Campaign).get(campaign_id)
        questions = await self.get_campaign_questions(campaign_id)

        for question in questions:
            for provider_id in campaign.config['providers']:
                try:
                    provider = await self.ai_service.get_provider(provider_id)
                    result = await
provider.generate_response(question.content)

                    # Enregistrement du résultat
                    campaign_result = CampaignResult(
                        campaign_id=campaign_id,
                        question_id=question.id,
                        ai_provider_id=provider_id,
                        response=result['response'],
                        tokens_count=result['tokens'],
                        response_time=result['time']
                    )
                    self.db_session.add(campaign_result)

                except Exception as e:
                    # Gestion des erreurs
                    campaign_result = CampaignResult(
                        campaign_id=campaign_id,
                        question_id=question.id,
                        ai_provider_id=provider_id,
                        error=str(e)
                    )
                    self.db_session.add(campaign_result)

        await self.db_session.commit()
```

## 6. API Endpoints
```python
# backend/src/api/ai_providers.py
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from src.services.ai_service import AIProviderService
router = APIRouter()
@router.post("/providers/")
async def create_provider(
    provider_data: ProviderCreate,
    db: Session = Depends(get_db)
):
    service = AIProviderService(db)
    return await service.register_provider(provider_data.dict())
@router.get("/providers/")
async def list_providers(db: Session = Depends(get_db)):
    return await db.query(AIProvider).all()
# backend/src/api/campaigns.py
@router.post("/campaigns/{campaign_id}/run")
async def run_campaign(
    campaign_id: int,
    db: Session = Depends(get_db)
):
    service = CampaignService(db, AIProviderService(db))
    return await service.run_campaign(campaign_id)
```
Cette structure permet :
1. L'ajout facile de nouvelles IAs via l'API
2. La gestion sécurisée des clés API
3. Le chargement dynamique des providers
4. Une séparation claire des responsabilités
5. Une extensibilité future
Voulez-vous que je détaille davantage certains aspects comme :
- La gestion des erreurs et retries
- Le système de cache des réponses
- Le monitoring et logging
- La sécurisation des données
- Les tests unitaires et d'intégration ?