News Aggregator Application Documentation

Table of Contents

1. Project Overview
2. System Requirements & Setup
3. Architecture Overview
4. Component Breakdown
5. Implementation Details
6. Frontend Interface
7. Usage Guide

1. Project Overview

The News Aggregator is a full-stack application that fetches, processes, and serves news articles based on user preferences. It combines several modern technologies to create a robust news delivery system with intelligent content processing capabilities.

Key Features: - Multi-topic news aggregation - Vector-based article storage and retrieval - AI-powered content summarization - User-friendly web interface - Real-time news processing

2. System Requirements & Setup

Required Dependencies

```
!pip install --upgrade --quiet langchain langchain-community langchain-pinecone
langchain-huggingface neo4j langchain-core tiktoken yfiles_jupyter_graphs
newsapi-python requests huggingface_hub pinecone-client tqdm pinecone
sentence_transformers py2neo gradio fastapi mistralai
```

API Keys Required

The application requires several API keys for various services:

```
HUGGINGFACE_TOKEN = userdata.get('HUGGINGFACE_TOKEN')
NEWSAPI_KEY = userdata.get('NEWSAPI_KEY')
PINECONE_API_KEY = userdata.get('PINECONE_API_KEY')
NEO4J_PASSWORD = userdata.get('NEO4J_PASSWORD')
MISTRAL_API_KEY = userdata.get('MISTRAL_API_KEY')
NEO4J_URI = userdata.get('NEO4J_URI')
```

3. Architecture Overview

The application follows a three-tier architecture:

1. Data Collection Layer
    - NewsAPI integration for fetching articles

- Article processing and storage

2. Processing Layer
    - Vector embeddings using HuggingFace
    - Pinecone vector database for efficient retrieval
    - Mistral AI for content summarization
3. Presentation Layer
    - FastAPI backend server
    - Gradio frontend interface

4. Component Breakdown

4.1 Article Fetching System

The `fetch_multiple_topics` function handles article collection:

```
def fetch_multiple_topics(api_key: str, topics: List[str], database_folder: str = "database'
```

Key features: - Fetches articles for multiple topics - Implements rate limiting - Saves articles with metadata - Handles error cases - Creates safe filenames for storage

4.2 RAG (Retrieval-Augmented Generation) System

The `ArticleRAG` class manages article processing and retrieval:

```
class ArticleRAG:
    def __init__(self, database_folder: str = "database",
                 index_name: str = "articles-embeddings")
```

Features: - HuggingFace embeddings integration - Pinecone vector database management - Text chunking for better processing - Similarity search capabilities

4.3 Backend API

FastAPI implementation for handling requests:

```
app = FastAPI()

@app.post("/fetch_news/")
def fetch_news(request: FetchNewsRequest)
```

Components: - Request validation using Pydantic models - Document fetching and processing - LLM integration for content generation - Error handling

4.4 Frontend Interface

Gradio-based user interface:

```
with gr.Blocks() as ui:
    user_dropdown = gr.Dropdown(users, label="Select User", value="1")
    category_dropdown = gr.Dropdown(categories, label="Select Category")
```

Features: - User selection - Category filtering - Real-time news fetching - Formatted news display

5. Implementation Details

5.1 Document Processing

The system processes documents in chunks using RecursiveCharacterTextSplitter:

```
self.text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=500,
    chunk_overlap=100,
    length_function=len
)
```

This ensures: - Optimal chunk sizes for processing - Proper context preservation - Efficient vector storage

5.2 Vector Search Implementation

The system uses Pinecone for vector search:

```
self.vector_store = PineconeVectorStore(
    index=self.pc.Index(self.index_name),
    embedding=self.embeddings
)
```

Features: - Cosine similarity search - Efficient vector storage - Serverless architecture

5.3 Content Generation

Content generation uses Mistral AI:

```
def llm_response(prompt: str) -> str:
    model = "mistral-large-latest"
    client = Mistral(api_key = MISTRAL_API_KEY)
```

The system: - Processes user preferences - Combines with retrieved documents - Generates personalized summaries

6. Frontend Interface

The Gradio interface provides: - User selection dropdown - Category selection - News fetch button - News display area

Interface code:

```
with gr.Blocks() as ui:
    user_dropdown = gr.Dropdown(users, label="Select User", value="1")
    category_dropdown = gr.Dropdown(categories, label="Select Category")
    fetch_button = gr.Button("Fetch News")
    news_display = gr.Textbox(label="News")
```

7. Usage Guide

1. Start the application by running all cells in sequence
2. The FastAPI backend will start automatically
3. The Gradio interface will launch with a public URL
4. Select a user and category
5. Click "Fetch News" to retrieve articles

Supported Categories:

- Artificial Intelligence
- Politics
- Business
- Technology
- Sports
- Entertainment
- Health

Error Handling

The system includes comprehensive error handling: - API request failures - Processing errors - Database connection issues - Invalid user inputs

Best Practices and Recommendations

1. API Key Management
   - Store keys securely
   - Use environment variables
   - Implement key rotation
2. Performance Optimization
   - Monitor API rate limits
   - Implement caching where appropriate
   - Use batch processing for vectors
3. Scaling Considerations
   - Implement connection pooling
   - Use async operations where possible
   - Consider implementing caching
4. Maintenance
   - Regular database cleanup

- Monitor vector store size
- Update embeddings periodically

Submitted by - **Giriraj Data Scientist - Trainee**
giriraj@incedoinc.com
