RUFUS DOCUMENTATION

- Kunal Shah
- kunaljshah03@gmail.com

GitHub - https://github.com/kunalshah03/rufus

This documentation provides comprehensive guidance for integrating Rufus into various RAG systems and maintaining high-quality data processing pipelines.

Table of Contents:

Architecture Overview

Rufus consists of three main components working together to prepare data for RAG systems:

```
[Web Scraper] -> [Content Processor] -> [RAG-Ready Documents]
```

Data Flow

1. URLs are crawled asynchronously
2. Content is extracted and cleaned
3. AI processes and structures the content
4. Documents are formatted for RAG systems
5. Output is saved in JSONL format

Core Components

1. Web Scraper

```
from rufus import RufusClient

client = RufusClient()

raw_content = client.scraper.crawl("https://example.com")
```

Features:

- Async crawling
- Dynamic content handling
- Domain restriction
- Configurable depth

2. Content Processor

```
processed_docs = client.processor.process(raw_content,
instructions="Extract product information")
```

Capabilities:

- AI-powered extraction
- Content chunking
- Relevance scoring
- Metadata enrichment

3. Document Structure

```
{
    "id": "doc-123",
    "text": "Main content for embedding",
    "metadata": {
        "title": "Document Title",
        "source_url": "https://example.com",
        "chunk_type": "article",
        "timestamp": "2024-11-15T12:00:00",
        "topics": ["topic1", "topic2"],
        "context": "Additional context",
        "relevance_score": 0.95
    }
}
```

Integration with RAG Systems

1. LangChain Integration

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import FAISS
import json
# Load Rufus documents
documents = []
with open("output/scraped_content.jsonl", "r") as f:
    for line in f:
        doc = json.loads(line)
        documents.append({
            "page_content": doc["text"],
```

```
            "metadata": doc["metadata"]
        })
# Create vector store
embeddings = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(documents, embeddings)
```

2. LlamaIndex Integration

```
from llama_index import Document, VectorStoreIndex
import json
# Load Rufus documents
documents = []
with open("output/scraped_content.jsonl", "r") as f:
    for line in f:
        doc = json.loads(line)
        documents.append(Document(
            text=doc["text"],
            metadata=doc["metadata"]
        ))
# Create index
index = VectorStoreIndex.from_documents(documents)
```

3. Custom RAG Pipeline

```
from rufus import RufusClient
import chromadb
import json
# 1. Scrape and process content
client = RufusClient()
```

```
documents = client.scrape("https://example.com", "Extract
information")
client.processor.save_to_jsonl(documents, "output.jsonl")


# 2. Create ChromaDB collection
chroma_client = chromadb.Client()
collection = chroma_client.create_collection("my_collection")


# 3. Load documents into ChromaDB
with open("output.jsonl", "r") as f:
    for line in f:
        doc = json.loads(line)
        collection.add(
            documents=[doc["text"]],
            metadatas=[doc["metadata"]],
            ids=[doc["id"]]
      )
```

Advanced Usage

1. Custom Content Processing

```
class CustomProcessor(ContentProcessor):
    def generate_rag_prompt(self, content: str, instructions: str) ->
str:
        return """
        Custom prompt for specific use case
        Content: {content}
        Instructions: {instructions}
```

```
        """
```

2. Filtered Crawling

```
client = RufusClient()

client.scraper.max_depth = 2

client.scraper.max_pages = 50

client.scraper.timeout = 20
```

3. Batch Processing

```
urls = ["https://site1.com", "https://site2.com"]

all_documents = []

for url in urls:

    documents = client.scrape(url, "Extract information")

    all_documents.extend(documents)

client.processor.save_to_jsonl(all_documents,
"combined_output.jsonl")
```

Best Practices

1. Content Chunking

- Keep chunks between 500-1000 tokens
- Preserve semantic boundaries
- Include context in metadata

2. Metadata Management

- Include source information
- Add timestamps
- Maintain context
- Track relationships

3. Performance Optimization

```
# Configure for large sites
```

```
client.scraper.max_concurrent = 10

client.processor.rate_limit_delay = 0.5
```

4. Error Handling

```
try:
    documents = client.scrape(url, instructions)
    if not documents:
        logger.warning(f"No content extracted from {url}")
except Exception as e:
    logger.error(f"Error processing {url}: {str(e)}")
```

5. Regular Maintenance

- Update base URLs
- Check content structure
- Monitor relevance scores
- Validate output format

Common Integration Patterns

1. Incremental Updates

```
      def update_rag_system():
    # Get new content
    new_documents = client.scrape(url, instructions)
    # Filter already processed
    existing_ids = set()
    with open("processed_ids.txt", "r") as f:
        existing_ids = set(f.read().splitlines())
    # Process new documents
    new_docs = [doc for doc in new_documents
                if doc["id"] not in existing_ids]
```

```
    # Update RAG system
    if new_docs:
        client.processor.save_to_jsonl(new_docs, "new_content.jsonl")
        # Update processed IDs
        with open("processed_ids.txt", "a") as f:
            for doc in new_docs:
       f.write(f"{doc['id']}\n")
```

2. Quality Control

```
def validate_documents(documents):
    for doc in documents:
        if doc["metadata"]["relevance_score"] < 0.7:
            continue
        if len(doc["text"]) < 100:
            continue
      yield doc
```
