**PageRAG**Backend Retrieval Generation Augmentation Architecture

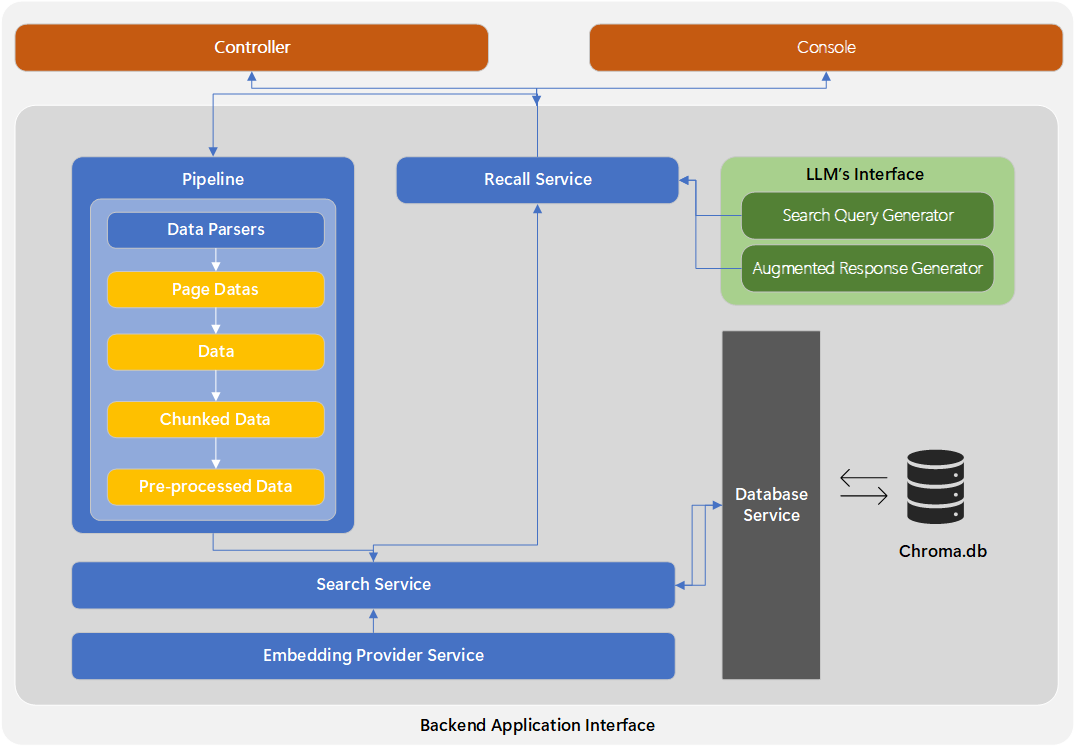
**Overview**

This document will provide detailed overview of the PageRAG application architecture, which handle recall scenarios from various data-sources such as PDF, word-documents etc. The primary responsibility of the application is:

1. Collecting different types of data (text, audio, image etc.) from various data-sources
2. Maintaining and processing the collected data with the help of embedding models
3. Maintaining database for storing the embedding data and other relevant information
4. Indexing and performing semantic search on stored embeddings
5. Using LLM’s to perform contextual search and generate relevant answers to user’s query
6. Exposing APIs from the host-server

**High Level Design**

Below is the diagram which illustrates all the logical modules which are required for PageRAG backend application and its dependencies.

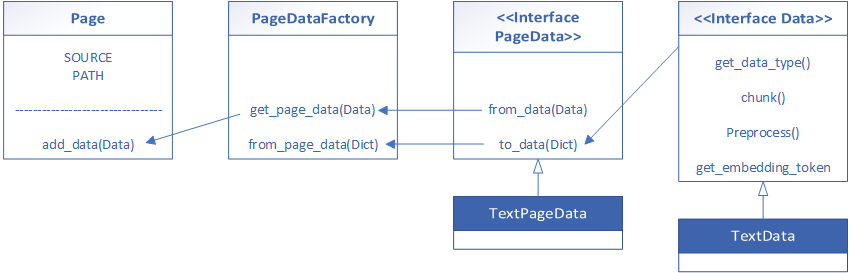


**Low Level Design**

Backend Application is divided into below logical modules

**Models**

The Model module refer to abstract representation and actual implementations of real-world entities that handle data and the operations tightly coupled with them

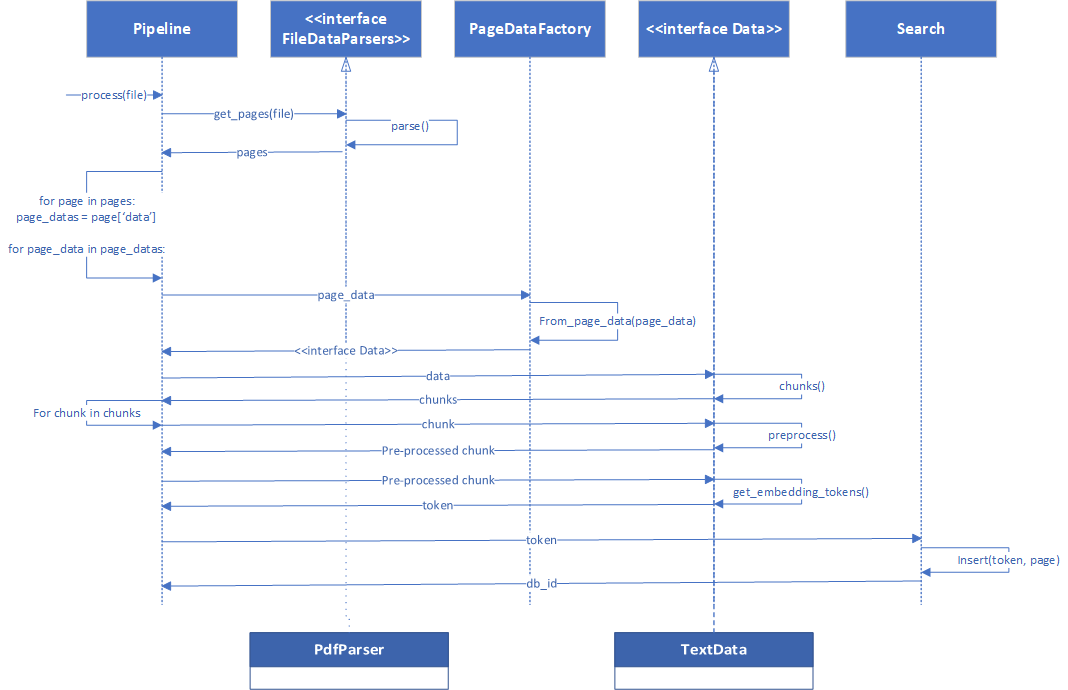


|  |  |
| --- | --- |
| Item | Description |
| Page | A data entity that represents an encapsulated set of *Data* from a data-source. It is a serializable class and contains a list of *PageData’s* depicting the *Data* contained within the *Page*. These *Page* items are meant to be passed onto LLM’s for augmented data-generation.  The concept lies in the fact that each data-source can be interpreted as a *Book*. Each *Book* in turn would contain multiple *Page*’s depicting the data contained within the *Book* |
| PageDataFactory | A concrete factory class that provides methods to convert dictionary onto *Data* and vice-versa |
| PageData | A data entity the represent a serializable form for a *Data* object |
| Data | A concrete class containing all the properties for a specific datatype. E.g Text, Image, Audio etc. They also implement the logic to handle chunking and pre-processing. |

**Backend**

Pipeline (incl. Data Parsers and Pre-processors)

The Pipeline module is responsible for processing files and indexing them onto the storage. It is highly flexible and very scalable in nature



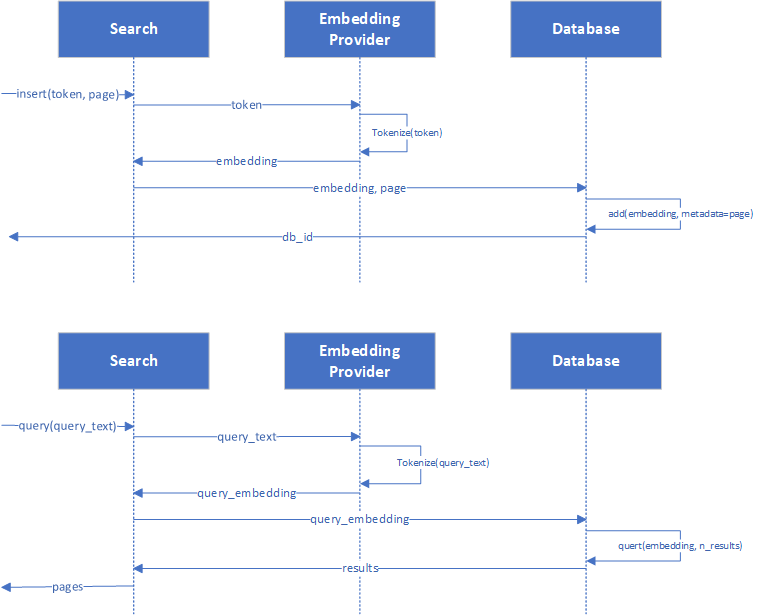
|  |  |
| --- | --- |
| Item | Description |
| Pipeline | This is a concrete class as a singleton service for handling indexing of a file onto the storage. It first parses the file, converting it onto *Page*’s which represents raw data.  The *Data* present in the *Page* is further chunked, processed and is used to fetch the embedding tokens |
| FileDataParsers | An interface representing class for converting a file (*Book*) into corresponding list of *Page*’s. Each *Page* contains the raw data present in the file |
| Search | A concrete class as a singleton service for inserting the embedding onto the storage and return a unique db\_id |

The core functionalities of a Data:

1. Chunk: Chunking the text using a sliding window algorithm so that its maximum character length is length than the size of the tokens required by the embedding provider
2. Pre-processes: Pre-process the data before sending to the embedding function

Search (incl. Embedding Provider or Tokenizer)

The search module is responsible for interfacing with the pipeline to insert embeddings and its corresponding page onto the storage, as well as for returning the right search results for semantic search done over the vector storage.



|  |  |
| --- | --- |
| Item | Description |
| Search | This is a concrete class as a singleton service responsible for inserting token and to perform semantic search based on a query on the storage, and return the right search-results in the form of relevant pages from all over the knowledge base |
| Embedding Provider (or Tokenizer) | This is a concrete class as a singleton service for converting token into embeddings. Currently it uses “BAAI/bge-m3” multi-lingual dense vector model |

Recall

**Database**

The Database module contains the logic to query with the underlying storage. Its main responsibilities are to add embeddings onto the vector-database and retrieve semantically-similar embeddings along with the associated *Page* item & relevant metadata from the database.

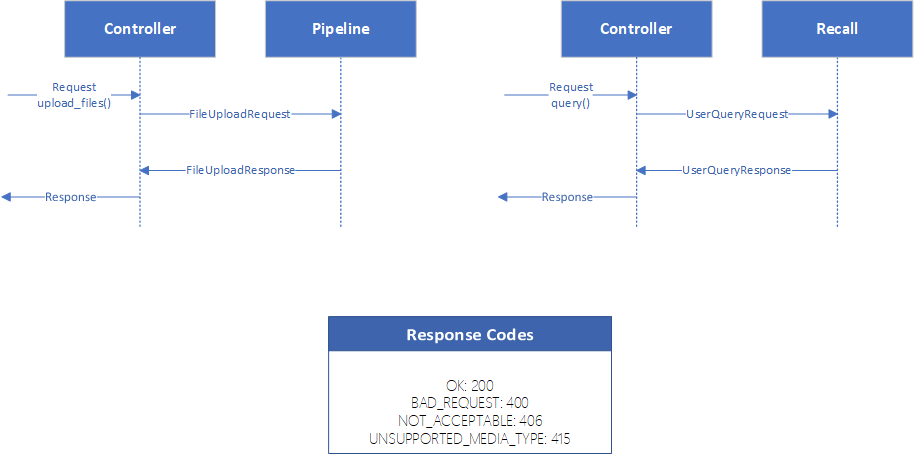
A blue and white box with black text

Description automatically generated

|  |  |
| --- | --- |
| Item | Description |
| Database | This is a concrete class for handling interactions on the underlying vector database. It separated the for accessing, querying, and inserting data from the rest of the application.  In our use-case it will be a singleton service |
| Chroma Vector Database | A vector-database that provides us with API to store embeddings and their metadata’s  The metadata’s is required to be a dictionary with *`key: str`* and *`value: str`* |

**Controllers**

The Controller module contains the code for handling the request made to the hosted server of the application.  
  
Currently, the server is hosted using Flask python framework.

****

|  |  |
| --- | --- |
| Item | Description |
| Controller | This is a concrete class for handling interactions with the requested made to the hosted flask server |
| Request Models | A data-class to hold request models |
| Response Models | A data-class to hold response models |

The goal of clean architecture is to always create Request and Response Model Data-Classes to operate on business logic inside the controller. These data-class must be serializable. Additionally, these data-classes should hold the logic to manipulate and process the data and must be passed by the controller to further services. A data-class should never self-interact with a service.

**References**

1. [Xero-Technologies/gangchain (github.com)](https://github.com/Xero-Technologies/gangchain/tree/main)