Service architecture

The Kernel Memory architecture can be divided in two main areas: Ingestion and Retrieval.

Ingestion

KM ingestion components leverage an underlying pipeline of sequential steps, accepting in input some data, such as a file, a URL or a string, and progressively turn the input into **Memory Records**. When a client sends a file, the service first saves data in the **Content Storage**, such as Azure Blobs or Local Disk. During this phase, a client stays connected, sending data and waiting for the data to be stored. Only when this operation is complete, KM releases the client request and starts an asynchronous pipeline, to complete the ingestion without blocking the client.

KM ingestion offers a default pipeline, composed by these sequential steps. Each step depends on the previous to complete successfully before starting:

1. Extract text
2. Split text in partitions
3. Generate embedding vectors for each partition
4. Store embedding vectors and metadata in a memory DB.

These steps are implemented by **Handlers**, shipped with the Core library. The list of steps and the logic of handlers can be customized. However, the default set of handlers is designed to work together:

- If a default handler is removed, without providing a replacement, ingestion will not complete
- Handlers store data in content storage using a specific schema, which you should follow when replacing or adding new handlers.

Only when a pipeline is complete data can be considered successfully imported. This process typically takes only a second, but the time required can grow considerably in two particular situations:

- Step 1: the file(s) uploaded requires an external service to be processed, e.g. OCR on large PDF documents.
- Step 3: the LLM used to generate embeddings is throttling requests, generating only few vectors per second.

NOTE
KM uses two storage solutions to hold information:

- Content Storage: raw data uploaded by clients, pipeline status, unique IDs assigned to documents.
- Memory Storage: databases with search capability, where KM stored Memory Records.

NOTE
The ingestion pipeline operates on **Documents**. A document is a collection of one or more files. When asking to import a web page, the system creates a document containing a special file that contains the web page URL. When asking to import raw text, the system creates a document containing a text file.

Ingestion handlers

For each document uploaded, ingestion handlers are called one at a time. Multiple documents can be uploaded in parallel, but within a document the ingestion pipeline steps always wait for the previous step to complete successfully before proceeding.

Here's a list of the default handlers included in the Core library:

1. **TextExtractionHandler**: this handler is typically the first invoked, taking a file and extracting the text. If a client provides a URL, the handler downloads the web page and extracts the text. The output of this handler is saved in content storage, to be processed further by the next handlers. This is also the handler responsible for OCR and detecting the content type.
2. **TextPartitioningHandler**: this handler covers a simple task: taking a text and splitting it into small chunks. The handler looks for a text file created by the previous handler, and manages plain text and markdown slightly differently. The default handler doesn't understand code syntax, chat logs, JSON or other structured data, which is always handled as a string. If you deal with particular formats, this is the first handler you might want to replace with a custom copy, to better split the user input. The default handler splits text in sentences (also called "lines") and aggregates them in paragraphs (also called "partitions"). Size of lines/sentences/paragraphs/partitions is measured in tokens. It can be configured, but is also dependent on the embedding generator used: if a partition is too big:
   - some embedding generators will throw an error when asked to generate an embedding vector;
   - some embedding generators will ignore the tokens in excess, resulting in incomplete memory records. This handler is typically very fast because all the work is done locally and in memory. Once the partitions are ready they are saved to the same content storage, ready for the next step. Currently, partition files are saved as raw text files.
3. **GenerateEmbeddingsHandler**: this handler loads each partition file, takes the text and asks the configured embedding generator to calculate an embedding vector. The vector is serialized to JSON and saved back to content storage. The handler processes one partition at a time, without any parallelism, and depending on the LLM used, it might take from few milliseconds to minutes to complete. This is another handler you could consider swapping with one optimized for your embedding generator, e.g. sending multiple requests in parallel, using cache, etc.
4. **SaveRecordsHandler**: this handler stores the embedding generated by the previous handler in one or more **Memory DBs**,including information about the source, including the partition text, tags and other metadata useful for search. This operation is usually very fast, and takes care of updating existing records, in case the document ID provided matches one previously uploaded.
5. **DeleteIndexHandler**: this is a special handler used by KM when a client asks to delete an index. The handler loops through all the memories, deleting memory records and files from content storage.
6. **DeleteDocumentHandler**: this is a special handler used by KM when a client asks to delete a document. The handler loops through all the memories extracted from this document, deleting memory records and files from content storage.
7. **DeleteGeneratedFilesHandler**: this is an optional handler, used in situations where one doesn't want to keep a copy of files in content storage. It should be invoked only after SaveRecordsHandler.
8. **SummarizationHandler**: this is an optional handler that a client can ask to use in order to generate and store a summary of the files uploaded. Since summarization can be time consuming, it's recommended to run it after SaveRecordsHandler, repeating TextPartitioningHandler and GenerateEmbeddingsHandler to include the summary. See notes below for more details.

Default pipeline

When uploading a document, unless specified, KM will start a pipeline composed by default steps:

1. "extract"
2. "partition"
3. "gen_embeddings"
4. "save_records"

The default list of steps can be configured, e.g. see `KernelMemory.DataIngestion.DefaultSteps` in the configuration file.

The list can also be changed by a client when sending a document, e.g. see the `steps` parameter in the memory API.

By default the strings "extract", "partition", etc. are mapped to the default handlers. You can override the mapping calling `.WithoutDefaultHandlers()` in the memory builder and registering handlers manually. See Core.KernelMemoryBuilder.BuildAsyncClient() for details. Otherwise, you can add new handlers without replacing the defaults. You can also run a handler as a standalone service, separate from KM service. The code varies depending whether you're customizing the service or using the serverless memory, look for these resources for implementation details:

- Core.KernelMemoryBuilder.BuildAsyncClient(): adding handlers to the service
- Example 004: loading custom handlers into ServerlessMemory
- Example 202: running a handler as standalone service

Summarization

The summarization handler included provides an example of "synthetic memory" generation, ie. processing input data and asking LLMs to extract and generate more information. The approach can be used for several scenario, e.g. extracting facts, transforming structured data format, etc.

In order to provide the best user experience, you should consider running this type of extractions after saving memory records:

- so users can start asking questions as soon as records are generated
- because synthetic data generation could take a long time to be generated, e.g. summarizing a book could take several minutes.

The summarization handler adds two special "synth" and "summary" tags to the records generated, allowing to search and retrieve summaries, e.g. using memory filters.

Synthetic data is generated creating new files inside Content Storage, which you should process after SaveRecordsHandler is complete, calling TextPartitioningHandler, GenerateEmbeddingsHandler and SaveRecordsHandler again. Since the default pipeline is **["extract", "partition", "gen_embeddings", "save_records"]**, you just need to overrider (either via configuration or during the client request) with: **["extract", "partition", "gen_embeddings", "save_records", "summarize", "partition", "gen_embeddings", "save_records"]**.

Since you can customize pipelines, you can also consider using KM just as a summarization service, using a pipeline like: **["extract", "summarize", "partition", "gen_embeddings", "save_records"]**.