Bigdata_cw2_Jacaranda

Release 1.0

Roger

May 02, 2025

CONTENTS:

1 modules package **3**
- 1.1 Subpackages . . . 3
- 1.2 Module contents . . . 14

2 my_fastapi package **15**
- 2.1 Submodules . . . 15
- 2.2 my_fastapi.main module . . . 15
- 2.3 Module contents . . . 18

3 test package **19**
- 3.1 Subpackages . . . 19
- 3.2 Module contents . . . 34

Python Module Index **35**

Index **37**

Welcome to the Bigdata_cw2_Jacaranda project documentation! This site contains technical documentation for all modules, FastAPI backend, and tests.

CHAPTER

ONE

MODULES PACKAGE

1.1 Subpackages

1.1.1 modules.data_storage package

Submodules

modules.data_storage.create_table module

Database setup module for CSR reporting system.

This module connects to a PostgreSQL database, creates necessary schema and tables for storing CSR indicators and associated data, and populates the indicators table with predefined environmental indicators and keywords.

modules.data_storage.create_table.**create_table_and_insert_data**()

Create the schema and tables required for CSR data reporting and insert predefined indicator metadata into the *CSR_indicators* table.

This includes: - Creating the *csr_reporting* schema if it does not exist. - Creating the *CSR_indicators* table to store CSR indicator metadata. - Creating the *CSR_Data* table to store extracted CSR data. - Inserting predefined indicators and associated keywords.

Raises

psycopg2.Error – If any error occurs during schema/table creation or data insertion.

modules.data_storage.data_export module

This module exports tables from a PostgreSQL database to CSV files.

It uses *psycopg2* for database connectivity and *pandas* to execute SQL queries and export the results to CSV files. The module supports configuration for the database connection, output file paths, and default sorting columns for the queries.

Functions: - *export_table_to_csv*: Exports a specified table from the PostgreSQL database to a CSV file. - *main*: Orchestrates the connection to the database and exports predefined tables to CSV files.

modules.data_storage.data_export.**export_table_to_csv**(*table_name*, *output_path*, *conn*)

Exports a specific table from the PostgreSQL database to a CSV file.

This function fetches all data from the specified table, optionally ordering it by a default column (if configured), and saves the results to a CSV file at the provided output path.

Parameters

- **table_name** (*str*) – The name of the table to export.
- **output_path** (*str*) – The path where the CSV file will be saved.

- **conn** (*psycopg2.extensions.connection*) – A psycopg2 connection object used to interact with the database.

Returns
None

Raises
Any exceptions raised by the *pandas.read_sql_query* or *pandas.DataFrame.to_csv* functions.

modules.data_storage.data_export.**main**()

Main function to connect to the PostgreSQL database and export multiple tables to CSV files.

This function establishes a connection to the database, iterates over a predefined list of tables, and calls *export_table_to_csv* to export each table to a CSV file.

Returns
None

Raises
Any exceptions raised during the database connection or table export process.

modules.data_storage.llm_analyse module

This module processes CSR (Corporate Social Responsibility) data by interacting with a PostgreSQL database and an OpenAI API. It performs tasks such as fetching pending CSR data from the database, building prompts for an LLM (Large Language Model), processing the data, and updating the results in the database.

The workflow involves the following key steps: 1. Establishing a connection to the PostgreSQL database. 2. Fetching pending rows that need processing. 3. Creating prompts for the LLM based on CSR report data and indicators. 4. Sending these prompts to an OpenAI-powered LLM for processing. 5. Extracting and updating relevant values (numeric or goal-related data) in the database.

Configuration: - PostgreSQL: Used for storing and retrieving CSR data. - OpenAI API: Used for processing CSR report content.

Dependencies: - psycopg2: PostgreSQL database adapter. - dotenv: Used to load environment variables. - openai: Python client for interacting with the OpenAI API. - tqdm: For progress bars. - concurrent.futures: For concurrent processing using threads.

modules.data_storage.llm_analyse.**build_prompt**(*indicator_name*, *description*, *is_target*, *source_excerpt*, *report_year*)

Builds a prompt for the LLM based on the CSR report and indicator details.

Parameters

- **indicator_name** (*str*) – The name of the CSR indicator.
- **description** (*str*) – The description of the CSR indicator.
- **is_target** (*bool*) – Flag indicating whether the indicator is a target indicator.
- **source_excerpt** (*list of dict*) – List of matched paragraphs from the CSR report.
- **report_year** (*int*) – The year of the CSR report.

Returns
The formatted prompt for the LLM.

Return type
str

modules.data_storage.llm_analyse.**call_llm**(*prompt*)

Calls the LLM API with the provided prompt and returns the response.

Parameters
prompt (*str*) – The prompt to be sent to the LLM.

Returns
The response content from the LLM.

Return type
str

modules.data_storage.llm_analyse.**fetch_pending_rows**(*conn*)

Fetches rows from the database that have a NULL value for 'value_raw' and are pending processing.

Parameters
conn (*psycopg2.extensions.connection*) – A psycopg2 connection object.

Returns
List of tuples representing the pending rows.

Return type
list of tuples

modules.data_storage.llm_analyse.**get_connection**()

Establishes a connection to the PostgreSQL database using the provided configuration.

Returns
psycopg2 connection object.

Return type
psycopg2.extensions.connection

modules.data_storage.llm_analyse.**is_valid_number**(*value*)

Checks if the given value is a valid number (either integer or float).

Parameters
value (*str*) – The value to check.

Returns
True if the value is a valid number, otherwise False.

Return type
bool

modules.data_storage.llm_analyse.**main**()

Main function to process all pending rows in the database by calling the process_row function concurrently.

modules.data_storage.llm_analyse.**process_row**(*conn*, *row*)

Processes a single row from the database, including calling the LLM API and updating the result in the database.

Parameters

- **conn** (*psycopg2.extensions.connection*) – A psycopg2 connection object.
- **row** (*tuple*) – A tuple representing a single row of CSR data to be processed.

Returns
The ID of the processed data row.

Return type
int

Raises
Exception – If an error occurs during processing.

modules.data_storage.llm_analyse.**update_result**(*conn*, *data_id*, *value_raw*, *unit_raw*, *llm_response_raw*, *pdf_page*)

Updates the processed result in the database for the given data ID.

Parameters

- **conn** (*psycopg2.extensions.connection*) – A psycopg2 connection object.
- **data_id** (*int*) – The ID of the data to be updated.
- **value_raw** (*str*) – The raw value extracted from the CSR report.
- **unit_raw** (*str*) – The unit of the extracted value.
- **llm_response_raw** (*str*) – The raw response from the LLM.
- **pdf_page** (*str*) – The page number(s) from the CSR report where the data was found.

modules.data_storage.llm_standardize module

This module processes and standardizes CSR report data by converting indicator units to a target unit using a machine learning model (LLM). It interacts with a PostgreSQL database to fetch, process, and update the data.

Key functions: - Fetch rows from the database that need unit standardization. - Build prompts for the LLM to convert units based on the CSR indicator. - Call the LLM API to get the conversion results. - Parse the response and update the database with standardized values.

It uses *psycopg2* for PostgreSQL interaction, *dotenv* for environment variable management, *tqdm* for progress tracking, and *concurrent.futures* for concurrent execution of row processing.

modules.data_storage.llm_standardize.**build_conversion_prompt**(*indicator_name*, *description*, *value_raw*, *unit_raw*, *target_unit*)

Constructs a prompt for the LLM to convert a unit of measurement from the raw value to the target unit.

Parameters

- **indicator_name** (*str*) – The name of the CSR indicator.
- **description** (*str*) – The description of the CSR indicator.
- **value_raw** (*str*) – The raw value to be converted.
- **unit_raw** (*str*) – The unit of the raw value.
- **target_unit** (*str*) – The target unit to convert the value to.

Returns
The LLM prompt as a formatted string.

Return type
str

modules.data_storage.llm_standardize.**call_llm**(*prompt*)

Sends a request to the LLM (DeepSeek API) to process the unit conversion based on the given prompt.

Parameters
prompt (*str*) – The conversion prompt to send to the LLM.

Returns
The response content from the LLM.

Return type
str

modules.data_storage.llm_standardize.**fetch_rows_to_standardize**(*conn*)

Fetches rows from the database that require unit standardization.

Parameters
conn (*psycopg2.extensions.connection*) – The psycopg2 connection object to the database.

Returns
List of rows containing data to be standardized.

Return type
list of tuples

modules.data_storage.llm_standardize.**get_connection**()

Establishes and returns a connection to the PostgreSQL database using the specified configuration.

Returns
psycopg2 connection object

Return type
psycopg2.extensions.connection

modules.data_storage.llm_standardize.**is_valid_number**(*value*)

Checks if the given value is a valid numeric value (either integer or float).

Parameters
value (*str*) – The value to check.

Returns
True if the value is valid, otherwise False.

Return type
bool

modules.data_storage.llm_standardize.**main**()

Main function to process all pending rows in the database by calling the process_row function concurrently.

modules.data_storage.llm_standardize.**process_row**(*conn*, *row*)

Processes a single row of CSR data, including calling the LLM API and updating the database.

Parameters

- **conn** (*psycopg2.extensions.connection*) – The psycopg2 connection object to the database.
- **row** (*tuple*) – A tuple representing the CSR data to process.

Returns
The data ID of the processed row, or None if processing failed.

Return type
int or None

modules.data_storage.llm_standardize.**safe_json_parse**(*response_text*)

Cleans and parses the raw response text from the LLM into a structured JSON format.

Parameters
response_text (*str*) – The raw response text from the LLM.

Returns
The parsed JSON object containing the conversion result.

Return type
dict

Raises
ValueError – If the response cannot be parsed into valid JSON.

modules.data_storage.llm_standardize.**update_standardized**(*conn*, *data_id*, *value_standardized*, *unit_standardized*, *unit_conversion_note=None*)

Updates the database with the standardized value and unit for the given data ID.

Parameters

- **conn** (*psycopg2.extensions.connection*) – The psycopg2 connection object to the database.
- **data_id** (*int*) – The ID of the data row to update.
- **value_standardized** (*str*) – The standardized value to be saved.
- **unit_standardized** (*str*) – The unit of the standardized value.
- **unit_conversion_note** (*str, optional*) – Additional notes about the unit conversion (optional).

modules.data_storage.main module

This module defines and manages the execution of a series of Python scripts in a data processing pipeline.

It includes functions for executing shell commands, logging the results, tracking completed modules, and executing the pipeline with progress reporting. The pipeline logs progress and execution times of individual modules, ensuring each module is executed only once.

Modules run by this pipeline: - *create_table.py* - *paragraph_extraction.py* - *retry_failed_reports.py* - *llm_analyse.py* - *llm_standardize.py* - *data_export.py*

Functions: - *read_completed_modules*: Reads a list of completed modules from a file. - *write_completed_module*: Appends a completed module to a record file. - *run_command*: Executes a shell command and logs the result. - *main*: Orchestrates the pipeline execution, including dependency installation, module execution, and logging.

modules.data_storage.main.**main**()

Orchestrates the execution of the full data processing pipeline.

This function installs the necessary dependencies, loads the list of completed modules, and iterates over a predefined set of modules to execute. It tracks the execution progress and logs each step. Each module is only executed once, and the completion is recorded.

Returns
None

Raises
Any exceptions raised during the pipeline execution.

modules.data_storage.main.**read_completed_modules**()

Reads the list of modules that have already been completed from a file.

This function checks for a file containing the names of completed modules and returns a set of these module names. If the file does not exist, an empty set is returned.

Returns
A set of module names that have been completed.

Return type
set

Raises
FileNotFoundError – If the completed modules file is not found.

modules.data_storage.main.**run_command**(*command*)

Executes a shell command and logs the result.

This function runs the specified command in the shell, logging the start and success of the execution. If the command fails, the error is logged, and the script exits with a failure code.

Parameters
command (*str*) – The shell command to execute.

Returns
None

Raises
subprocess.CalledProcessError – If the command fails during execution.

modules.data_storage.main.**write_completed_module**(*module*)

Appends the name of a completed module to the record file.

This function writes the provided module name to a file that tracks which modules have been executed. Each module name is written on a new line.

Parameters
module (*str*) – The name of the module that has been completed.

Returns
None

Raises
Any exceptions raised during file operations.

modules.data_storage.paragraph_extraction module

PDF Paragraph Extraction and CSR Indicator Matching Tool

This module provides functions to extract paragraphs from CSR reports in PDF format stored in MinIO, match them against indicators stored in a PostgreSQL database, and store the matched results.

Modules used: - MinIO for object storage access - pdfplumber for PDF text extraction - fuzzywuzzy for keyword similarity matching - psycopg2 for PostgreSQL interaction - tqdm for progress display - multiprocessing for parallel processing

modules.data_storage.paragraph_extraction.**check_memory_usage**(*threshold_percent=90*)

Check current memory usage and print a warning if it exceeds a given threshold.

Parameters
threshold_percent (*int*) – Memory usage percentage threshold.

Returns
True if memory usage is below threshold, else False.

Return type
bool

modules.data_storage.paragraph_extraction.**extract_paragraphs_from_pdf**(*file_path*)

Extract paragraphs from a PDF file.

Parameters
file_path (*str*) – Path to the local PDF file.

Yield
Tuple containing page number and cleaned paragraph text.

Return type
tuple[int, str]

modules.data_storage.paragraph_extraction.**find_matching_paragraphs**(*paragraphs*, *keywords*, *threshold=80*)

Find paragraphs that match a set of keywords using fuzzy string matching.

Parameters

- **paragraphs** (*list[tuple[int, str]]*) – List of (page number, paragraph) tuples.
- **keywords** (*list[str]*) – List of keyword strings to match.
- **threshold** (*int*) – Matching threshold (0–100), defaults to 80.

Returns
List of matched paragraphs with page numbers.

Return type
list[dict]

modules.data_storage.paragraph_extraction.**insert_matched_data**(*conn*, *security*, *report_year*, *indicator_id*, *indicator_name*, *matched*, *extraction_time*)

Insert matched paragraphs into the csr_reporting.CSR_Data table.

Parameters

- **conn** (*psycopg2.connection*) – PostgreSQL connection object.
- **security** (*str*) – Security identifier.
- **report_year** (*int*) – Year of the report.
- **indicator_id** (*int*) – ID of the matched indicator.
- **indicator_name** (*str*) – Name of the matched indicator.
- **matched** (*list[dict]*) – List of matched paragraph entries.
- **extraction_time** (*datetime.datetime*) – Timestamp of extraction.

modules.data_storage.paragraph_extraction.**load_indicators_from_db**(*conn*)

Load CSR indicators and associated keywords from the database.

Parameters
conn (*psycopg2.connection*) – PostgreSQL database connection.

Returns
List of tuples (indicator_id, indicator_name, keywords).

Return type
list[tuple[int, str, list[str]]]

modules.data_storage.paragraph_extraction.**parse_security_and_year**(*object_name*)

Parse the security identifier and report year from a given object name.

Parameters
object_name (*str*) – The name of the PDF object in MinIO.

Returns
Tuple of (security, year) or (None, None) if parsing fails.

Return type
tuple[str, int | None]

modules.data_storage.paragraph_extraction.**process_all_pdfs**()

Main pipeline to process all PDF files from MinIO: - Downloads each file - Extracts paragraphs - Matches with indicators - Saves matched results to the database - Logs failed files

modules.data_storage.paragraph_extraction.**process_report**(*args*)

Process a single PDF report: download, extract, match indicators, and insert data.

Parameters
args (*tuple*) – Tuple of (conn_config, MinIO object, indicators).

Returns
Object name if failed, else None.

Return type
str | None

modules.data_storage.retry_failed_reports module

This module is responsible for processing CSR (Corporate Social Responsibility) reports. It connects to a MinIO server to download the reports, extracts text paragraphs from PDF files, matches them against CSR indicators loaded from a PostgreSQL database, and stores the matched data back into the database.

The workflow includes the following key steps: 1. Downloading the CSR report from MinIO. 2. Extracting text paragraphs from the report's PDF file. 3. Matching the extracted paragraphs with predefined CSR indicators using fuzzy string matching. 4. Inserting matched data into the PostgreSQL database. 5. Supporting retry logic for processing failed reports.

Configuration: - MinIO: Used for storing and retrieving CSR report PDF files. - PostgreSQL: Used to store CSR indicators and matched data.

Dependencies: - psycopg2: PostgreSQL database adapter. - pdfplumber: PDF text extraction library. - fuzzywuzzy: Fuzzy string matching library. - tqdm: For progress bars. - minio: MinIO Python client for object storage.

modules.data_storage.retry_failed_reports.**extract_paragraphs_from_pdf**(*file_path*)

Extracts paragraphs from a PDF file.

This function opens the specified PDF file, extracts the text from each page, and splits the text into paragraphs based on sentence delimiters. It only retains paragraphs with more than 20 characters.

Parameters
file_path (*str*) – The path to the PDF file.

Returns
A list of tuples, each containing the page number and a paragraph.

Return type
list of tuples (int, str)

modules.data_storage.retry_failed_reports.**find_matching_paragraphs**(*paragraphs*, *keywords*, *threshold=80*)

Finds paragraphs that match the given keywords based on fuzzy string matching.

This function compares each paragraph with the provided keywords using fuzzy string matching (via the fuzzywuzzy library) and returns the paragraphs with matches above a specified threshold.

Parameters

- **paragraphs** (*list of tuples (int, str)*) – A list of paragraphs to search.
- **keywords** (*list of str*) – A list of keywords to match against.
- **threshold** (*int*) – The minimum fuzzy match score to consider a paragraph as a match.

Returns
A list of dictionaries containing the matched paragraphs with their page number.

Return type
list of dict

modules.data_storage.retry_failed_reports.**insert_matched_data**(*conn*, *security*, *report_year*, *indicator_id*, *indicator_name*, *matched*, *extraction_time*)

Inserts the matched CSR data into the database.

This function inserts the matched paragraphs along with metadata into the PostgreSQL database for later retrieval and reporting.

Parameters

- **conn** (*psycopg2.connection*) – The database connection object.
- **security** (*str*) – The security identifier of the company.
- **report_year** (*int*) – The year of the CSR report.
- **indicator_id** (*int*) – The ID of the CSR indicator.
- **indicator_name** (*str*) – The name of the CSR indicator.
- **matched** (*list of dict*) – A list of matched paragraphs.
- **extraction_time** (*datetime.datetime*) – The time when the data was extracted.

modules.data_storage.retry_failed_reports.**load_indicators_from_db**(*conn*)

Loads CSR indicators from the PostgreSQL database.

This function queries the database for CSR indicator data, including indicator names and keywords, which are used later to match against extracted paragraphs.

Parameters
conn (*psycopg2.connection*) – The database connection object.

Returns
A list of tuples containing indicator_id, indicator_name, and keywords.

Return type
list of tuples (int, str, list)

modules.data_storage.retry_failed_reports.**parse_security_and_year**(*object_name*)

Parses the security identifier and report year from the given object name.

This function expects the object name to follow a specific naming convention where the filename contains a security identifier followed by a 4-digit year (e.g., "ABC_2021.pdf").

Parameters
object_name (*str*) – The name of the object (PDF file) stored in MinIO.

Returns
A tuple containing the security identifier and report year, or (None, None) if parsing fails.

Return type
tuple or (None, None)

modules.data_storage.retry_failed_reports.**process_single_report**(*object_name*, *indicators*, *conn_config*)

Processes a single report by downloading it from MinIO, extracting paragraphs, matching them with indicators, and saving the results to the database.

This function downloads the report, extracts its paragraphs, matches them with CSR indicators, and stores the results in the database.

Parameters

- **object_name** (*str*) – The name of the object (PDF file) stored in MinIO.
- **indicators** (*list of tuples (int, str, list)*) – A list of CSR indicators to match against.
- **conn_config** (*dict*) – The configuration for connecting to the PostgreSQL database.

Returns
True if the report was processed successfully, False otherwise.

Return type
bool

Raises
Exception – If any error occurs during the processing of the report.

modules.data_storage.retry_failed_reports.**retry_failed_reports**()

Retries processing of reports that failed during a previous attempt. If retrying fails, the report names are saved in the 'failed_reports.json' file.

This function reads a list of failed report names from a JSON file, retries processing them, and updates the list of failed reports if necessary.

Raises
Exception – If any error occurs during the retry process.

Module contents

1.1.2 modules.frontend package

Module contents

1.1.3 modules.security package

Submodules

modules.security.scan_code module

This module provides a function to run a security scan on a specified source code directory using Bandit.

Bandit is a tool designed to find common security issues in Python code. This module uses the *poetry* package to execute the Bandit scan on a specified directory (*modules/data_storage*).

Main function: - *run_bandit_scan*: Executes Bandit on the source code directory to identify security issues.

To run this module: - Execute it as the main program using the *if __name__ == "__main__":* block.

modules.security.scan_code.**run_bandit_scan**()

Executes a security scan on the source code directory using Bandit.

This function runs the Bandit tool on the *modules/data_storage* directory, scanning the Python source code for common security issues. It uses *poetry* to execute the scan.

The Bandit scan helps identify potential security flaws, such as improper handling of inputs, hardcoded passwords, insecure cryptographic practices, and more.

Returns
None

Return type
None

modules.security.scan_dependencies module

This module provides a function to run a security check on the project dependencies using Safety.

Safety is a tool for checking Python dependencies for known security vulnerabilities. This module uses *poetry* to execute the Safety check command.

Main function: - *run_safety_check*: Executes Safety to check for known security issues in the project dependencies.

To run this module: - Execute it as the main program using the *if __name__ == "__main__":* block.

modules.security.scan_dependencies.**run_safety_check**()

Executes a security check on the project dependencies using Safety.

This function runs the Safety tool, which checks the project's dependencies for known security vulnerabilities. The check is executed using *poetry* as the package manager to run the Safety command.

Returns
None

Return type
None

Module contents

1.2 Module contents

CHAPTER

TWO

MY_FASTAPI PACKAGE

2.1 Submodules

2.2 my_fastapi.main module

This module defines a FastAPI application that serves as an API for accessing CSR (Corporate Social Responsibility) data. It includes endpoints for querying CSR indicators, CSR data, and company reports from a PostgreSQL database. Additionally, it serves static files for frontend development and includes CORS configuration for cross-origin requests.

Main functionality: - CORS setup for allowing specific origins. - Static file serving for frontend deployment (React). - Database connections and query execution using psycopg2. - API endpoints for accessing and querying CSR data, indicators, and reports.

To run the application: - Run *poetry run uvicorn my_fastapi.main:app –host 0.0.0.0 –port 8000 –reload* - Run *ngrok http –url=csr.jacaranda.ngrok.app 8000* - Access Swagger UI at *http://127.0.0.1:8000/docs* - Original JSON schema can be found at *http://127.0.0.1:8000/openapi.json*

`my_fastapi.main.get_all_company_reports()`

Retrieves all company reports from the database.

This endpoint queries the *csr_reporting.company_reports* table and returns all rows ordered by *id*.

Returns
A list of company reports.

Return type
list

`my_fastapi.main.get_all_data()`

Retrieves all CSR data from the database.

This endpoint queries the *csr_reporting.CSR_Data* table and returns the first 100 rows ordered by *data_id*.

Returns
A list of CSR data entries.

Return type
list

`my_fastapi.main.get_all_indicators()`

Retrieves all CSR indicators from the database.

This endpoint queries the *csr_reporting.CSR_indicators* table and returns all rows ordered by *indicator_id*.

Returns
A list of CSR indicators from the database.

Return type
list

my_fastapi.main.**get_company_report_by_id**(*report_id: int*)

Retrieves a specific company report by its ID.

This endpoint queries the *csr_reporting.company_reports* table for a record with the specified *report_id*.

Parameters
report_id (*int*) – The ID of the company report to retrieve.

Returns
The company report with the specified ID.

Return type
dict

Raises
HTTPException – If no company report is found with the given ID.

my_fastapi.main.**get_data_by_id**(*data_id: int*)

Retrieves a specific CSR data entry by its ID.

This endpoint queries the *csr_reporting.CSR_Data* table for a record with the specified *data_id*.

Parameters
data_id (*int*) – The ID of the data entry to retrieve.

Returns
The CSR data entry with the specified ID.

Return type
dict

Raises
HTTPException – If no data entry is found with the given ID.

my_fastapi.main.**get_db**()

Establishes and returns a connection to the PostgreSQL database.

This function connects to the database using the credentials and configuration defined in the *db_config* dictionary. It is used to execute queries against the database.

Returns
A database connection object.

Return type
psycopg2.extensions.connection

my_fastapi.main.**get_indicator_by_id**(*indicator_id: int*)

Retrieves a specific CSR indicator by its ID.

This endpoint queries the *csr_reporting.CSR_indicators* table for a record with the specified *indicator_id*.

Parameters
indicator_id (*int*) – The ID of the indicator to retrieve.

Returns
The CSR indicator with the specified ID.

Return type
dict

Raises
HTTPException – If no indicator is found with the given ID.

my_fastapi.main.**query_db**(*query: str*, *params: tuple | None = None*)

Executes a query on the database and returns the results.

This function executes a given SQL query on the database using the provided parameters and returns the results as a list of dictionaries.

Parameters

- **query** (*str*) – The SQL query to execute.
- **params** (*Optional[tuple]*) – Optional parameters to pass to the query, defaults to None.

Returns
The results of the query as a list of dictionaries.

Return type
list

my_fastapi.main.**search_data**(*security: str | None = Query(None)*, *report_year: int | None = Query(None)*, *indicator_id: int | None = Query(None)*, *indicator_name: str | None = Query(None)*)

Searches CSR data based on optional query parameters.

This endpoint allows filtering CSR data by security, report year, indicator ID, and/or indicator name.

Parameters

- **security** (*Optional[str]*) – The security of the data to search for (optional).
- **report_year** (*Optional[int]*) – The report year of the data to search for (optional).
- **indicator_id** (*Optional[int]*) – The ID of the indicator to search for (optional).
- **indicator_name** (*Optional[str]*) – The name of the indicator to search for (optional).

Returns
A list of filtered CSR data.

Return type
list

my_fastapi.main.**search_indicators**(*indicator_name: str | None = Query(None)*, *theme: str | None = Query(None)*)

Searches CSR indicators based on optional query parameters.

This endpoint allows filtering CSR indicators by name and/or theme. It queries the *csr_reporting.CSR_indicators* table and applies the provided filters.

Parameters

- **indicator_name** (*Optional[str]*) – The name of the indicator to search for (optional).
- **theme** (*Optional[str]*) – The theme of the indicator to search for (optional).

Returns
A list of filtered CSR indicators.

Return type
list

my_fastapi.main.**search_reports**(*symbol: str | None = Query(None)*, *security: str | None = Query(None)*, *report_year: int | None = Query(None)*)

Searches company reports based on optional query parameters.

This endpoint allows filtering company reports by symbol, security, and/or report year.

Parameters

- **symbol** (*Optional[str]*) – The symbol of the report to search for (optional).
- **security** (*Optional[str]*) – The security of the report to search for (optional).
- **report_year** (*Optional[int]*) – The report year of the report to search for (optional).

Returns
A list of filtered company reports.

Return type
list

my_fastapi.main.**serve_frontend**()

Serves the React frontend's *index.html* file.

This endpoint is used to serve the frontend React application after it has been built and deployed to the server.

Returns
The *index.html* file of the React frontend.

Return type
FileResponse

2.3 Module contents

CHAPTER

THREE

TEST PACKAGE

3.1 Subpackages

3.1.1 test.data_storage package

Submodules

test.data_storage.conftest module

This module provides test fixtures used by pytest for testing the *modules.data_storage* functionality.

Test Fixtures: - *mock_db_connection*: Mocks a PostgreSQL database connection for testing purposes. - *monkeypatch_env*: Mocks the environment variable *DEEPSEEK_API_KEY* for tests.

These fixtures help simulate interactions with external dependencies like a database and environment variables.

`test.data_storage.conftest.`**`mock_db_connection`**`()`

Mocks a PostgreSQL database connection for testing.

This fixture patches the *psycopg2.connect* function to return a fake connection and cursor. The cursor's *fetchall*, *fetchone*, and *execute* methods are mocked to simulate query results without needing an actual database connection.

This fixture is scoped to the module, meaning it is created once per test module.

Returns
A mocked database connection.

Return type
MagicMock

`test.data_storage.conftest.`**`monkeypatch_env`**`(`*monkeypatch*`)`

Mocks the *DEEPSEEK_API_KEY* environment variable for testing.

This fixture uses the *monkeypatch* fixture to set the environment variable *DEEPSEEK_API_KEY* to a mock value (*"mock-key"*) during testing. This allows tests to simulate the presence of the environment variable without requiring an actual key.

Parameters
`monkeypatch` – The pytest monkeypatch fixture.

Returns
None

Return type
None

test.data_storage.test_create_table module

Module to simulate database connection and cursor behavior using dummy objects. It includes tests for successful table creation, insert failures, and connection failures.

class test.data_storage.test_create_table.**DummyConnection**

Bases: object

A dummy connection class to simulate database connection behavior for testing purposes.

cursor_obj

The dummy cursor object used for executing queries.

Type
DummyCursor

committed

A flag indicating whether the connection has been committed.

Type
bool

closed

A flag indicating whether the connection has been closed.

Type
bool

close()

Simulates closing the connection by setting the closed flag to True.

commit()

Simulates committing a transaction by setting the committed flag to True.

cursor()

Returns the cursor object associated with the connection.

Returns
The cursor object for executing queries.

Return type
DummyCursor

class test.data_storage.test_create_table.**DummyCursor**

Bases: object

A dummy cursor class to simulate database cursor behavior for testing purposes.

executed_queries

A list that records SQL queries that were executed.

Type
list

execute(*query*)

Simulates executing an SQL query by recording it in the executed_queries list.

Parameters
query (*str*) – The SQL query to be executed.

test.data_storage.test_create_table.**test_create_table_connection_failure**(*monkeypatch*, *capsys*)

Test case for a database connection failure.

Simulates a scenario where the database connection fails, and verifies that the appropriate error message is logged and the connection is handled correctly.

Parameters

- **monkeypatch** (*MonkeyPatch*) – A pytest fixture to modify functions or objects at runtime.
- **capsys** (*CaptureFixture*) – A pytest fixture to capture system outputs.

test.data_storage.test_create_table.**test_create_table_insert_failure**(*monkeypatch*, *capsys*)

Test case for a failure during the data insertion phase.

Simulates a scenario where data insertion fails, and verifies that the appropriate error message is logged and the connection is closed.

Parameters

- **monkeypatch** (*MonkeyPatch*) – A pytest fixture to modify functions or objects at runtime.
- **capsys** (*CaptureFixture*) – A pytest fixture to capture system outputs.

test.data_storage.test_create_table.**test_create_table_success**(*monkeypatch*, *capsys*)

Test case for successfully creating a table and inserting data.

Simulates the scenario where the table creation and data insertion are successful, and verifies the correct output and connection behavior.

Parameters

- **monkeypatch** (*MonkeyPatch*) – A pytest fixture to modify functions or objects at runtime.
- **capsys** (*CaptureFixture*) – A pytest fixture to capture system outputs.

test.data_storage.test_data_export module

Module: data_export_tests

This module contains test functions for testing the functionalities of the data export module in the *data_export* package. It tests the SQL query formation for exporting tables, the export process, and the correct execution of the *main()* function.

The tests mock dependencies like the database connection and pandas DataFrame methods to validate the correct behavior of the export logic.

test.data_storage.test_data_export.**test_export_table_query_with_order**(*monkeypatch*)

Test that the query string correctly appends the ORDER BY clause when there are default order columns.

This test simulates the execution of a query with a default order column (e.g., "ORDER BY id") and verifies that the *export_table_to_csv* function appends the ORDER BY clause in the query. It also checks if the query is correctly passed to the pandas *read_sql_query* function and if the *to_csv* method is invoked with the correct path.

Parameters

monkeypatch – The monkeypatch fixture provided by pytest.

test.data_storage.test_data_export.**test_export_table_query_without_order**(*monkeypatch*)

Test that the query string does not include the ORDER BY clause when there are no default order columns.

This test simulates the execution of a query without a default order column and ensures that the *export_table_to_csv* function does not append the ORDER BY clause to the query. It also checks that the *to_csv* method is still called correctly.

Parameters
monkeypatch – The monkeypatch fixture provided by pytest.

test.data_storage.test_data_export.**test_main_exports_all_tables**(*monkeypatch*)

Test the main() function to ensure it exports all tables in order and closes the connection.

This test simulates the process of exporting all tables by calling the *main* function and verifies that all expected tables are exported. It also ensures that the database connection is closed after the export process is complete.

Parameters
monkeypatch – The monkeypatch fixture provided by pytest.

test.data_storage.test_data_storage_pipeline module

Module: main_tests

This module contains unit tests for the main module of the data storage system. It tests various aspects of the main module's functionality, such as reading and writing completed modules, running shell commands, handling module execution, and ensuring correct pipeline behavior.

Tests use pytest's monkeypatching and mocking techniques to simulate various scenarios and edge cases.

Each function is designed to ensure the main module performs as expected under different conditions, including success and failure cases.

test.data_storage.test_data_storage_pipeline.**test_main_pipeline_failure**(*monkeypatch*)

Test that the main function exits with an error when a module fails.

test.data_storage.test_data_storage_pipeline.**test_main_skip_and_execute**(*monkeypatch*)

Test the main function to ensure that already completed modules are skipped and that the remaining modules are executed in the correct order.

test.data_storage.test_data_storage_pipeline.**test_read_completed_modules**(*tmp_path*, *monkeypatch*)

Test that the read_completed_modules function behaves correctly when the file does or does not exist.

test.data_storage.test_data_storage_pipeline.**test_run_command_success_and_failure**(*monkeypatch*)

Test the run_command function for both success and failure scenarios.

test.data_storage.test_data_storage_pipeline.**test_write_completed_module**(*tmp_path*, *monkeypatch*)

Test that the write_completed_module function correctly appends module names to the completed file.

test.data_storage.test_llm_analyse module

Module for testing the LLM (Large Language Model) analysis functions. This module contains various test cases to ensure the correct behavior of functions like build_prompt, process_row, update_result, and others in the llm_analyse module.

class test.data_storage.test_llm_analyse.**DummyConnection**

Bases: object

A dummy database connection class to simulate database connection behavior.

This class is used to simulate database connection operations during testing.

close()

Simulate closing the connection.

commit()

Simulate committing a transaction.

cursor()

Simulate getting a cursor from the connection.

Returns
A dummy cursor instance.

Return type
DummyCursor

rollback()

Simulate rolling back a transaction.

class test.data_storage.test_llm_analyse.**DummyCursor**

Bases: object

A dummy database cursor class to simulate database cursor behavior.

This class is used to simulate cursor behavior for database operations during testing, specifically for testing the update_result function.

execute(*query*, *params=None*)

Simulate executing a database query.

Parameters

- **query** (*str*) – The SQL query string to be executed.
- **params** (*tuple*) – The parameters to be passed to the query.

test.data_storage.test_llm_analyse.**test_build_prompt_target_vs_non_target**()

Test the build_prompt function for target vs non-target classes.

This test verifies that the output prompt varies based on whether the target class is specified or not.

test.data_storage.test_llm_analyse.**test_is_valid_number**()

Test the is_valid_number function for valid and invalid number inputs.

This test checks if the function correctly identifies valid numeric values and handles invalid inputs like strings, None, or empty values.

test.data_storage.test_llm_analyse.**test_main_concurrent_processing**(*monkeypatch*, *capsys*)

Test the main function with concurrent processing logic, including both successful and failed tasks.

This test simulates a multi-threaded environment where multiple rows are processed concurrently, some successfully and some failing, and verifies the expected output messages.

Parameters

- **monkeypatch** (*pytest.MonkeyPatch*) – The monkeypatch object for mocking.
- **capsys** (*pytest.CaptureFixture*) – The capture fixture to capture the standard output.

test.data_storage.test_llm_analyse.**test_process_row_nontarget_invalid_number**(*monkeypatch*)

Test process_row function with target set to False and invalid number in the LLM output.

This test verifies that the function raises a ValueError and performs a rollback when the output is a non-numeric value for a non-target row.

Parameters
monkeypatch (*pytest.MonkeyPatch*) – The monkeypatch object for mocking.

test.data_storage.test_llm_analyse.**test_process_row_nontarget_json_error**(*monkeypatch*)

Test process_row function with target set to False and invalid JSON format.

This test checks if the function raises a JSONDecodeError when the LLM output is an invalid JSON string and performs a rollback.

Parameters
monkeypatch (*pytest.MonkeyPatch*) – The monkeypatch object for mocking.

test.data_storage.test_llm_analyse.**test_process_row_target_json_and_nonjson**(*monkeypatch*)

Test process_row function with target set to True for both valid and invalid JSON outputs.

This test checks if the function handles cases where the LLM returns valid JSON or invalid JSON, and ensures that the appropriate values are passed for updating the database.

Parameters
monkeypatch (*pytest.MonkeyPatch*) – The monkeypatch object for mocking.

test.data_storage.test_llm_analyse.**test_process_row_target_non_str_output**(*monkeypatch*)

Test process_row function with target set to True when LLM outputs a non-string type.

This test ensures that the function correctly handles non-string types like bytes and processes them appropriately without errors.

Parameters
monkeypatch (*pytest.MonkeyPatch*) – The monkeypatch object for mocking.

test.data_storage.test_llm_analyse.**test_update_result_executes_commit**(*monkeypatch*)

Test that the update_result function executes an UPDATE query and commits.

This test verifies that the update_result function correctly executes an UPDATE statement and commits the changes to the database.

Parameters
monkeypatch (*pytest.MonkeyPatch*) – The monkeypatch object for mocking.

test.data_storage.test_llm_standardize module

This module contains tests for the llm_standardize functionality, which includes the processing and standardization of measurement units using LLM (Large Language Models). It includes dummy database connection and cursor classes to simulate interactions with a database and test various edge cases, such as invalid or non-convertible data, successful standardizations, and handling of JSON errors.

Tests cover the following aspects: - Parsing of JSON wrapped in markdown - Construction of conversion prompts - Execution of database updates for standardized data - Handling of unit mismatches and the conversion process - Proper handling of success and failure cases for standardization

class test.data_storage.test_llm_standardize.**DummyConnection**

Bases: object

A dummy implementation of a database connection, simulating commit, rollback, and cursor functionalities.

close()

Simulates closing the database connection.

commit()

Simulates committing the current transaction.

cursor()

Returns the simulated cursor object.

Returns
The cursor object for executing queries.

Return type
DummyCursor

rollback()

Simulates rolling back the current transaction.

class test.data_storage.test_llm_standardize.**DummyCursor**

Bases: object

A dummy implementation of a database cursor for simulating database interactions.

close()

Simulates closing the cursor.

execute(*query*, *params=None*)

Simulates executing a database query.

Parameters

- **query** (*str*) – The SQL query string to execute.
- **params** (*dict, optional*) – Optional parameters for the SQL query.

test.data_storage.test_llm_standardize.**test_build_conversion_prompt_content**()

Test case for the build_conversion_prompt function to ensure the correct construction of conversion prompts.

test.data_storage.test_llm_standardize.**test_main_standardization_pipeline**(*monkeypatch*, *capsys*)

Test case for the main function in the standardization pipeline, including bulk processing and file output.

test.data_storage.test_llm_standardize.**test_process_row_convertible_false**(*monkeypatch*)

Test case for the process_row function when LLM returns a response indicating that conversion is not possible.

test.data_storage.test_llm_standardize.**test_process_row_invalid_result_value**(*monkeypatch*, *capsys*)

Test case for the process_row function when the LLM returns a non-numeric value after conversion.

test.data_storage.test_llm_standardize.**test_process_row_success**(*monkeypatch*)

Test case for the process_row function when LLM returns a valid numeric conversion result.

test.data_storage.test_llm_standardize.**test_process_row_unit_match**(*monkeypatch*)

Test case for the process_row function where the unit matches and no conversion is needed.

test.data_storage.test_llm_standardize.**test_safe_json_parse_cleanup_and_errors**()

Test case for the *safe_json_parse* function to ensure proper handling of valid and invalid JSON strings.

test.data_storage.test_llm_standardize.**test_update_standardized_execution**(*monkeypatch*)

Test case for the update_standardized function to verify SQL execution and commit behavior.

test.data_storage.test_paragraph_extraction module

Test module for paragraph extraction functionality within the data storage module. This module tests various functions including PDF paragraph extraction, database insertion, memory usage checks, and the processing of multiple PDFs.

test.data_storage.test_paragraph_extraction.**mock_db**()

Fixture to mock database connection and cursor for testing database interactions.

Yields
mock_conn (MagicMock) – The mocked database connection object.

test.data_storage.test_paragraph_extraction.**mock_minio**()

Fixture to mock MinIO client by patching the 'fget_object' method to return None.

Yields
mock_fget (MagicMock) – The mocked MinIO 'fget_object' method.

test.data_storage.test_paragraph_extraction.**test_check_memory_usage**()

Test the memory usage check function.

Verifies that the function returns the correct value based on system memory usage.

test.data_storage.test_paragraph_extraction.**test_extract_paragraphs_from_pdf**(*mock_minio*)

Test the extraction of paragraphs from a PDF file.

Simulates the process of extracting text from a PDF and verifies that paragraphs are correctly extracted.

Parameters
mock_minio (*MagicMock*) – The mocked MinIO client for testing purposes.

test.data_storage.test_paragraph_extraction.**test_find_matching_paragraphs**()

Test the functionality of finding paragraphs that match given keywords.

Verifies that the correct paragraphs are matched based on the provided keywords.

test.data_storage.test_paragraph_extraction.**test_insert_matched_data**(*mock_db*, *mock_minio*)

Test the insertion of matched paragraphs into the database.

Verifies that the matched data is correctly inserted into the database and that the proper SQL statement is executed.

Parameters

- **mock_db** (*MagicMock*) – The mocked database connection object.
- **mock_minio** (*MagicMock*) – The mocked MinIO client for testing purposes.

test.data_storage.test_paragraph_extraction.**test_memory_management**()

Test the memory management functionality by simulating high and low memory usage.

Verifies that the system behaves as expected under varying memory conditions.

test.data_storage.test_paragraph_extraction.**test_parse_security_and_year**()

Test the extraction of security and year information from a file name.

Verifies that the correct security ticker and year are parsed from the file name.

test.data_storage.test_paragraph_extraction.**test_process_all_pdfs**(*mock_fget*, *mock_list_objects*, *mock_db*)

Test the processing of all PDF files from MinIO.

Verifies that all PDFs are processed correctly, including the retrieval and handling of files, database updates, and the execution of relevant queries.

Parameters

- **mock_fget** (*MagicMock*) – The mocked MinIO 'fget_object' method.
- **mock_list_objects** (*MagicMock*) – The mocked MinIO 'list_objects' method.

- **mock_db** (*MagicMock*) – The mocked database connection object.

test.data_storage.test_paragraph_extraction.**test_process_report**(*mock_fget*, *mock_db*)

Test the processing of a report file.

Simulates the entire process of handling a report, including database interactions, memory usage checks, and file handling.

Parameters

- **mock_fget** (*MagicMock*) – The mocked MinIO 'fget_object' method.
- **mock_db** (*MagicMock*) – The mocked database connection object.

test.data_storage.test_retry_failed_reports module

Test module for retrying failed reports in the paragraph extraction module. This module tests the retry logic for handling failed report downloads, matching paragraphs in PDFs, and database interactions.

test.data_storage.test_retry_failed_reports.**test_download_error**(*mock_connect*, *mock_fget*, *mock_open_file*, *mock_exists*)

Test case where the file download fails during the retry process.

Verifies that the function handles download errors appropriately.

Parameters

- **mock_connect** (*MagicMock*) – Mocked psycopg2 connect method for simulating database connection.
- **mock_fget** (*MagicMock*) – Mocked MinIO 'fget_object' method to simulate a download error.
- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading a PDF file name.
- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

test.data_storage.test_retry_failed_reports.**test_empty_failed_json**(*mock_open_file*, *mock_exists*)

Test case where the 'failed_reports.json' file is empty.

Verifies that the function does not attempt any processing when the JSON file is empty.

Parameters

- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading an empty list.
- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

test.data_storage.test_retry_failed_reports.**test_failed_json_removed_after_success**(*mock_unlink*, *mock_remove*, *mock_os_exists*, *mock_pdf*, *mock_connect*, *mock_fget*, *mock_open_file*, *mock_exists*)

Test case where all files are successfully processed and the 'failed_reports.json' is deleted.

Verifies that the function properly removes the 'failed_reports.json' after all files are successfully processed.

Parameters

- **mock_unlink** (*MagicMock*) – Mocked Path.unlink method to simulate file deletion.
- **mock_remove** (*MagicMock*) – Mocked os.remove method to simulate file removal.
- **mock_os_exists** (*MagicMock*) – Mocked os.path.exists method to simulate file existence.
- **mock_pdf** (*MagicMock*) – Mocked pdfplumber.open to simulate PDF page extraction.
- **mock_connect** (*MagicMock*) – Mocked psycopg2 connect method for simulating database connection.
- **mock_fget** (*MagicMock*) – Mocked MinIO 'fget_object' method for simulating file retrieval.
- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading a PDF file name.
- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

test.data_storage.test_retry_failed_reports.**test_invalid_filename_skipped**(*mock_connect*, *mock_open_file*, *mock_exists*)

Test case where 'parse_security_and_year' returns None due to an invalid file name.

Verifies that the function skips processing for invalid filenames.

Parameters

- **mock_connect** (*MagicMock*) – Mocked psycopg2 connect method for simulating database connection.
- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading an invalid file name.
- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

test.data_storage.test_retry_failed_reports.**test_no_failed_json**(*mock_exists*)

Test case where the 'failed_reports.json' file does not exist.

Verifies that the function does not raise an error and returns immediately.

Parameters

mock_exists (*MagicMock*) – Mocked Path.exists method to simulate file absence.

test.data_storage.test_retry_failed_reports.**test_no_matching_paragraphs**(*mock_pdf*, *mock_connect*, *mock_fget*, *mock_open_file*, *mock_exists*)

Test case where no paragraphs match the given keywords.

Verifies that the function skips further processing when no matching paragraphs are found.

Parameters

- **mock_pdf** (*MagicMock*) – Mocked pdfplumber.open to simulate PDF page extraction.
- **mock_connect** (*MagicMock*) – Mocked psycopg2 connect method for simulating database connection.
- **mock_fget** (*MagicMock*) – Mocked MinIO 'fget_object' method for simulating file retrieval.
- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading a PDF file name.

- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

test.data_storage.test_retry_failed_reports.**test_pdf_page_no_text**(*mock_pdf*, *mock_connect*, *mock_fget*, *mock_open_file*, *mock_exists*)

Test case where the PDF page extraction returns None (no text extracted).

Verifies that the function handles the case of missing text on a PDF page appropriately.

Parameters

- **mock_pdf** (*MagicMock*) – Mocked pdfplumber.open to simulate PDF page extraction.
- **mock_connect** (*MagicMock*) – Mocked psycopg2 connect method for simulating database connection.
- **mock_fget** (*MagicMock*) – Mocked MinIO 'fget_object' method for simulating file retrieval.
- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading a PDF file name.
- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

test.data_storage.test_retry_failed_reports.**test_successful_extraction**(*mock_remove*, *mock_os_exists*, *mock_pdf*, *mock_connect*, *mock_fget*, *mock_open_file*, *mock_exists*)

Test case where the paragraph extraction is successful and database updates occur.

Verifies that the extraction is successful, the data is processed, and the database is updated.

Parameters

- **mock_remove** (*MagicMock*) – Mocked os.remove method to simulate file removal.
- **mock_os_exists** (*MagicMock*) – Mocked os.path.exists method to simulate file existence.
- **mock_pdf** (*MagicMock*) – Mocked pdfplumber.open to simulate PDF page extraction.
- **mock_connect** (*MagicMock*) – Mocked psycopg2 connect method for simulating database connection.
- **mock_fget** (*MagicMock*) – Mocked MinIO 'fget_object' method for simulating file retrieval.
- **mock_open_file** (*MagicMock*) – Mocked open function to simulate reading a PDF file name.
- **mock_exists** (*MagicMock*) – Mocked Path.exists method to simulate file existence.

Module contents

3.1.2 test.frontend package

Submodules

test.frontend.test_app module

Module: test_app_structure

This module contains tests to verify the structure and functionality of the frontend *App.js* file located within the *modules/frontend/src/* directory. These tests ensure that the file includes essential elements like React hooks, JSX components, and event handling.

Test cases: - Structural validation of the App.js file (checking for required elements like hooks, JSX structure, and HTML elements). - Functional validation to ensure that the App.js contains necessary event handling logic (e.g., *onClick* event binding and state update logic).

test.frontend.test_app.**test_app_contains_add_functionality**()

Test the functionality of the App.js file to check if it includes add functionality.

This test checks if the *App.js* file contains logic for handling button click events and updating state. Specifically, it verifies that an *onClick* event handler is present and that there is a state update logic, indicated by calls to functions that start with *set*.

It performs the following checks: - Verifies that the file contains an *onClick* event handler. - Verifies that a state update function (e.g., *setState*) is called within the file.

Raises

AssertionError – If any of the above checks fail.

test.frontend.test_app.**test_app_structure**()

Test the structure of the App.js file.

This test verifies that the *App.js* file contains the essential React elements, including the use of the *useState* hook, JSX structure, and required HTML elements such as <*input*> and <*button*>. It ensures the basic structure of the component is set up correctly.

It performs the following checks: - Verifies that *useState* is used in the file. - Verifies that the file contains a return statement with JSX. - Verifies the inclusion of an <*input*> element. - Verifies the inclusion of a <*button*> element.

Raises

AssertionError – If any of the above checks fail.

test.frontend.test_index_css module

Module: test_custom_checkbox_class_exists

This module contains a test function that checks if the necessary CSS classes related to the custom checkbox, loading spinner, and error message elements are defined within the *index.css* file of the frontend project. It also verifies that Tailwind CSS has been properly integrated into the project by checking for the presence of its directives.

Test case: - Verifies the existence of specific CSS classes (*.custom-checkbox*, *.custom-checkbox:checked*, *.loading-spinner*, and *.error-message*). - Verifies that Tailwind CSS directives (*@tailwind base*, *@tailwind components*, and *@tailwind utilities*) are included in the *index.css* file.

test.frontend.test_index_css.**test_custom_checkbox_class_exists**()

Test the existence of essential CSS classes and Tailwind integration in the *index.css* file.

This test function reads the contents of the *index.css* file, which is part of the frontend project, and checks for the presence of the following: - *.custom-checkbox*: A class that should define custom checkbox styles. - *.custom-checkbox:checked*: A pseudo-class for checked custom checkboxes. - *.loading-spinner*: A class to define the loading spinner styles. - *.error-message*: A class to define the error message styles. - Tailwind CSS directives: Verifies that the *@tailwind* directives for *base*, *components*, and *utilities* are present to ensure Tailwind CSS is integrated into the project.

Raises

AssertionError – If any of the CSS classes or Tailwind CSS directives are missing.

test.frontend.test_index_js module

Module: test_index_imports_and_mounts

This module contains a test function that verifies the correct imports and mounting logic within the *index.js* file in a frontend project. Specifically, it ensures that: - The *App* component is imported correctly. - The *ReactDOM.createRoot* method is called for mounting. - The *root.render* method is invoked to render the *App* component.

Since Python cannot directly mock or execute JavaScript files, this test performs static checks on the content of the *index.js* file to confirm the required JavaScript functionality.

Test case: - Ensures proper import and mounting of the *App* component in *index.js*.

`test.frontend.test_index_js.`**`test_index_imports_and_mounts`**`()`

Test the imports and mounting logic in the *index.js* file.

This test checks that the *index.js* file contains the necessary JavaScript code for: - Importing the *App* component with *import App from*. - Calling *ReactDOM.createRoot* to initialize the app mounting point. - Using *root.render* to render the *App* component.

Raises

`AssertionError` – If any of the required imports or method calls are missing.

test.frontend.test_tailwind_config module

Module: test_tailwind_config

This module contains tests for validating the contents and structure of the *tailwind.config.js* file in a frontend project. Specifically, it checks if certain expected keys, values, and settings are present in the configuration file.

Test cases: - Ensures the presence of specific keys and values in the Tailwind CSS configuration.

`test.frontend.test_tailwind_config.`**`test_tailwind_config_contains_expected_keys`**`()`

Test that the *tailwind.config.js* file contains expected keys and values.

This test verifies that the following elements are present in the Tailwind CSS configuration file: - The *module.exports* keyword, indicating that the file exports the configuration. - The *theme:* key for defining theme-related customizations. - The *extend:* key to check for extensions in the theme. - A custom color *#1a8c70* for teal. - The animation name *pulse*.

Raises

`AssertionError` – If any of the expected keys or values are missing from the configuration file.

Module contents

3.1.3 test.my_fastapi package

Submodules

test.my_fastapi.test_main module

Module: test_main

This module contains test cases for the FastAPI application endpoints. It uses pytest and unittest.mock to test various endpoints that interact with a database, returning indicator, data, and report information. The tests cover scenarios such as successful retrieval, search with parameters, and error handling.

Each test mocks the database query (*query_db*) to return predefined values, allowing for consistent and controlled test execution.

Test cases include: - Retrieving all indicators. - Searching for indicators by name or theme. - Retrieving data by ID. - Searching for data based on multiple parameters. - Handling of reports and errors (e.g., not found).

The tests verify the status codes, response structures, and expected behaviors of the API.

test.my_fastapi.test_main.**test_favicon**(*monkeypatch*)

Test the favicon endpoint.

This test ensures that the favicon URL returns a valid response (either 200 or 404) depending on the existence of the favicon file.

Parameters
monkeypatch – A pytest fixture that allows patching functions for testing purposes.

test.my_fastapi.test_main.**test_get_all_data**(*mock_query*)

Test the GET /data endpoint.

This test ensures that the endpoint returns a list of data items with the correct structure. It verifies that the response status is 200 and that the returned data is a list.

Parameters
mock_query – The mocked query_db function used to return predefined data.

test.my_fastapi.test_main.**test_get_company_report_by_id**(*mock_query*)

Test the GET /reports/{id} endpoint by report ID.

This test ensures that the endpoint correctly retrieves a report by its ID. It verifies that the response status is 200 and the report contains the 'symbol' field.

Parameters
mock_query – The mocked query_db function used to return predefined data for the specified report ID.

test.my_fastapi.test_main.**test_get_data_by_id**(*mock_query*)

Test the GET /data/{id} endpoint by ID.

This test ensures that the endpoint correctly retrieves data by its ID. It verifies that the response status is 200 and the data ID is 100.

Parameters
mock_query – The mocked query_db function used to return predefined data for the specified ID.

test.my_fastapi.test_main.**test_get_data_by_id_not_found**(*mock_query*)

Test the GET /data/{id} endpoint when the data is not found.

This test ensures that the endpoint returns a 404 status code when the requested data is not found.

Parameters
mock_query – The mocked query_db function used to simulate no data being found.

test.my_fastapi.test_main.**test_get_indicator_by_id**(*mock_query*)

Test the GET /indicators/{id} endpoint by ID.

This test ensures that the endpoint correctly retrieves an indicator by its ID. It verifies that the response status is 200 and that the indicator name matches "Water".

Parameters
mock_query – The mocked query_db function used to return predefined data for the specified ID.

test.my_fastapi.test_main.**test_get_indicator_not_found**(*mock_query*)

Test the GET /indicators/{id} endpoint when the indicator is not found.

This test ensures that the endpoint returns a 404 status code when the requested indicator does not exist.

Parameters
mock_query – The mocked query_db function used to simulate no data being found.

test.my_fastapi.test_main.**test_get_indicators**(*mock_query*)

Test the GET /indicators endpoint.

This test ensures that the endpoint returns a list of indicators with the correct structure. It verifies that the response status is 200 and that the 'indicator_name' of the first item is "Energy".

Parameters
mock_query – The mocked query_db function used to return predefined data.

test.my_fastapi.test_main.**test_get_report_not_found**(*mock_query*)

Test the GET /reports/{id} endpoint when the report is not found.

This test ensures that the endpoint returns a 404 status code when the requested report is not found.

Parameters
mock_query – The mocked query_db function used to simulate no data being found.

test.my_fastapi.test_main.**test_get_reports**(*mock_query*)

Test the GET /reports endpoint.

This test ensures that the endpoint returns a list of reports with the correct structure. It verifies that the response status is 200 and the returned data is a list.

Parameters
mock_query – The mocked query_db function used to return predefined data.

test.my_fastapi.test_main.**test_root_frontend_redirect**(*monkeypatch*)

Test the root URL for frontend redirect behavior.

This test ensures that the root URL either returns index.html or a 404 error depending on the availability of the frontend build directory. The test mocks the existence of the directory.

Parameters
monkeypatch – A pytest fixture that allows patching functions for testing purposes.

test.my_fastapi.test_main.**test_search_data_all_params**(*mock_query*)

Test the GET /data/search endpoint with all search parameters.

This test ensures that the endpoint returns no data when searching with specific parameters. It verifies that the response status is 200 and the returned data is an empty list.

Parameters
mock_query – The mocked query_db function used to simulate no data matching the search parameters.

test.my_fastapi.test_main.**test_search_data_no_param**(*mock_query*)

Test the GET /data/search endpoint with no search parameters.

This test ensures that the endpoint returns an empty list when no parameters are provided. It verifies that the response status is 200 and the returned data is an empty list.

Parameters
mock_query – The mocked query_db function used to simulate no data when no parameters are provided.

test.my_fastapi.test_main.**test_search_indicators_no_match**(*mock_query*)

Test the GET /indicators/search endpoint with no matching indicators.

This test ensures that when no matching indicators are found, the response is an empty list. It verifies that the response status is 200 and the JSON body is an empty list.

Parameters
mock_query – The mocked query_db function used to return an empty list.

test.my_fastapi.test_main.**test_search_indicators_with_name_and_theme**(*mock_query*)

Test the GET /indicators/search endpoint with matching indicators.

This test ensures that the endpoint returns matching indicators when searching by name and theme. It verifies that the response status is 200 and the returned data is a list.

Parameters
mock_query – The mocked query_db function used to return predefined matching data.

test.my_fastapi.test_main.**test_search_reports**(*mock_query*)

Test the GET /reports/search endpoint with a partial search parameter.

This test ensures that the endpoint returns a list of reports even when only a partial search parameter is provided. It verifies that the response status is 200 and the returned data is a list.

Parameters
mock_query – The mocked query_db function used to return predefined data for the search.

test.my_fastapi.test_main.**test_search_reports_all_params**(*mock_query*)

Test the GET /reports/search endpoint with all search parameters.

This test ensures that the endpoint returns matching reports when searching with specific parameters. It verifies that the response status is 200 and the returned data is a list.

Parameters
mock_query – The mocked query_db function used to return predefined data for the search parameters.

Module contents

3.2 Module contents

PYTHON MODULE INDEX

m

modules, 14
modules.data_storage, 13
modules.data_storage.create_table, 3
modules.data_storage.data_export, 3
modules.data_storage.llm_analyse, 4
modules.data_storage.llm_standardize, 6
modules.data_storage.main, 8
modules.data_storage.paragraph_extraction, 9
modules.data_storage.retry_failed_reports, 11
modules.frontend, 13
modules.security, 14
modules.security.scan_code, 13
modules.security.scan_dependencies, 14
my_fastapi, 18
my_fastapi.main, 15

t

test, 34
test.data_storage, 29
test.data_storage.conftest, 19
test.data_storage.test_create_table, 20
test.data_storage.test_data_export, 21
test.data_storage.test_data_storage_pipeline, 22
test.data_storage.test_llm_analyse, 22
test.data_storage.test_llm_standardize, 24
test.data_storage.test_paragraph_extraction, 25
test.data_storage.test_retry_failed_reports, 27
test.frontend, 31
test.frontend.test_app, 29
test.frontend.test_index_css, 30
test.frontend.test_index_js, 31
test.frontend.test_tailwind_config, 31
test.my_fastapi, 34
test.my_fastapi.test_main, 31

INDEX

B

build_conversion_prompt() (*in module modules.data_storage.llm_standardize*), 6
build_prompt() (*in module modules.data_storage.llm_analyse*), 4

C

call_llm() (*in module modules.data_storage.llm_analyse*), 4
call_llm() (*in module modules.data_storage.llm_standardize*), 6
check_memory_usage() (*in module modules.data_storage.paragraph_extraction*), 9
close() (*test.data_storage.test_create_table.DummyConnection method*), 20
close() (*test.data_storage.test_llm_analyse.DummyConnection method*), 22
close() (*test.data_storage.test_llm_standardize.DummyConnection method*), 24
close() (*test.data_storage.test_llm_standardize.DummyCursor method*), 25
closed (*test.data_storage.test_create_table.DummyConnection attribute*), 20
commit() (*test.data_storage.test_create_table.DummyConnection method*), 20
commit() (*test.data_storage.test_llm_analyse.DummyConnection method*), 23
commit() (*test.data_storage.test_llm_standardize.DummyConnection method*), 24
committed (*test.data_storage.test_create_table.DummyConnection attribute*), 20
create_table_and_insert_data() (*in module modules.data_storage.create_table*), 3
cursor() (*test.data_storage.test_create_table.DummyConnection method*), 20
cursor() (*test.data_storage.test_llm_analyse.DummyConnection method*), 23
cursor() (*test.data_storage.test_llm_standardize.DummyConnection method*), 24
cursor_obj (*test.data_storage.test_create_table.DummyConnection attribute*), 20

D

DummyConnection (*class in test.data_storage.test_create_table*), 20
DummyConnection (*class in test.data_storage.test_llm_analyse*), 22
DummyConnection (*class in test.data_storage.test_llm_standardize*), 24
DummyCursor (*class in test.data_storage.test_create_table*), 20
DummyCursor (*class in test.data_storage.test_llm_analyse*), 23
DummyCursor (*class in test.data_storage.test_llm_standardize*), 25

E

execute() (*test.data_storage.test_create_table.DummyCursor method*), 20
execute() (*test.data_storage.test_llm_analyse.DummyCursor method*), 23
execute() (*test.data_storage.test_llm_standardize.DummyCursor method*), 25
executed_queries (*test.data_storage.test_create_table.DummyCursor attribute*), 20
export_table_to_csv() (*in module modules.data_storage.data_export*), 3
extract_paragraphs_from_pdf() (*in module modules.data_storage.paragraph_extraction*), 9
extract_paragraphs_from_pdf() (*in module modules.data_storage.retry_failed_reports*), 11

F

fetch_pending_rows() (*in module modules.data_storage.llm_analyse*), 5
fetch_rows_to_standardize() (*in module modules.data_storage.llm_standardize*), 7
find_matching_paragraphs() (*in module modules.data_storage.paragraph_extraction*), 10
find_matching_paragraphs() (*in module modules.data_storage.retry_failed_reports*), 11

G

get_all_company_reports() (*in module my_fastapi.main*), 15
get_all_data() (*in module my_fastapi.main*), 15
get_all_indicators() (*in module my_fastapi.main*), 15
get_company_report_by_id() (*in module my_fastapi.main*), 16
get_connection() (*in module modules.data_storage.llm_analyse*), 5
get_connection() (*in module modules.data_storage.llm_standardize*), 7
get_data_by_id() (*in module my_fastapi.main*), 16
get_db() (*in module my_fastapi.main*), 16
get_indicator_by_id() (*in module my_fastapi.main*), 16

I

insert_matched_data() (*in module modules.data_storage.paragraph_extraction*), 10
insert_matched_data() (*in module modules.data_storage.retry_failed_reports*), 12
is_valid_number() (*in module modules.data_storage.llm_analyse*), 5
is_valid_number() (*in module modules.data_storage.llm_standardize*), 7

L

load_indicators_from_db() (*in module modules.data_storage.paragraph_extraction*), 10
load_indicators_from_db() (*in module modules.data_storage.retry_failed_reports*), 12

M

main() (*in module modules.data_storage.data_export*), 4
main() (*in module modules.data_storage.llm_analyse*), 5
main() (*in module modules.data_storage.llm_standardize*), 7
main() (*in module modules.data_storage.main*), 8
mock_db() (*in module test.data_storage.test_paragraph_extraction*), 25
mock_db_connection() (*in module test.data_storage.conftest*), 19
mock_minio() (*in module test.data_storage.test_paragraph_extraction*), 26
module
    modules, 14
    modules.data_storage, 13
    modules.data_storage.create_table, 3
    modules.data_storage.data_export, 3
    modules.data_storage.llm_analyse, 4
    modules.data_storage.llm_standardize, 6
    modules.data_storage.main, 8
    modules.data_storage.paragraph_extraction, 9
    modules.data_storage.retry_failed_reports, 11
    modules.frontend, 13
    modules.security, 14
    modules.security.scan_code, 13
    modules.security.scan_dependencies, 14
    my_fastapi, 18
    my_fastapi.main, 15
    test, 34
    test.data_storage, 29
    test.data_storage.conftest, 19
    test.data_storage.test_create_table, 20
    test.data_storage.test_data_export, 21
    test.data_storage.test_data_storage_pipeline, 22
    test.data_storage.test_llm_analyse, 22
    test.data_storage.test_llm_standardize, 24
    test.data_storage.test_paragraph_extraction, 25
    test.data_storage.test_retry_failed_reports, 27
    test.frontend, 31
    test.frontend.test_app, 29
    test.frontend.test_index_css, 30
    test.frontend.test_index_js, 31
    test.frontend.test_tailwind_config, 31
    test.my_fastapi, 34
    test.my_fastapi.test_main, 31
modules
    module, 14
modules.data_storage
    module, 13
modules.data_storage.create_table
    module, 3
modules.data_storage.data_export
    module, 3
modules.data_storage.llm_analyse
    module, 4
modules.data_storage.llm_standardize
    module, 6
modules.data_storage.main
    module, 8
modules.data_storage.paragraph_extraction
    module, 9
modules.data_storage.retry_failed_reports
    module, 11
modules.frontend

module, 13
modules.security
module, 14
modules.security.scan_code
module, 13
modules.security.scan_dependencies
module, 14
monkeypatch_env() (*in module test.data_storage.conftest*), 19
my_fastapi
module, 18
my_fastapi.main
module, 15

P

parse_security_and_year() (*in module modules.data_storage.paragraph_extraction*), 10
parse_security_and_year() (*in module modules.data_storage.retry_failed_reports*), 12
process_all_pdfs() (*in module modules.data_storage.paragraph_extraction*), 11
process_report() (*in module modules.data_storage.paragraph_extraction*), 11
process_row() (*in module modules.data_storage.llm_analyse*), 5
process_row() (*in module modules.data_storage.llm_standardize*), 7
process_single_report() (*in module modules.data_storage.retry_failed_reports*), 13

Q

query_db() (*in module my_fastapi.main*), 17

R

read_completed_modules() (*in module modules.data_storage.main*), 8
retry_failed_reports() (*in module modules.data_storage.retry_failed_reports*), 13
rollback() (*test.data_storage.test_llm_analyse.DummyConnection method*), 23
rollback() (*test.data_storage.test_llm_standardize.DummyConnection method*), 25
run_bandit_scan() (*in module modules.security.scan_code*), 14
run_command() (*in module modules.data_storage.main*), 9
run_safety_check() (*in module modules.security.scan_dependencies*), 14

S

safe_json_parse() (*in module modules.data_storage.llm_standardize*), 7
search_data() (*in module my_fastapi.main*), 17
search_indicators() (*in module my_fastapi.main*), 17
search_reports() (*in module my_fastapi.main*), 17
serve_frontend() (*in module my_fastapi.main*), 18

T

test
module, 34
test.data_storage
module, 29
test.data_storage.conftest
module, 19
test.data_storage.test_create_table
module, 20
test.data_storage.test_data_export
module, 21
test.data_storage.test_data_storage_pipeline
module, 22
test.data_storage.test_llm_analyse
module, 22
test.data_storage.test_llm_standardize
module, 24
test.data_storage.test_paragraph_extraction
module, 25
test.data_storage.test_retry_failed_reports
module, 27
test.frontend
module, 31
test.frontend.test_app
module, 29
test.frontend.test_index_css
module, 30
test.frontend.test_index_js
module, 31
test.frontend.test_tailwind_config
module, 31
test.my_fastapi
module, 34
test.my_fastapi.test_main
module, 31
test_app_contains_add_functionality() (*in module test.frontend.test_app*), 30
test_app_structure() (*in module test.frontend.test_app*), 30
test_build_conversion_prompt_content() (*in module test.data_storage.test_llm_standardize*), 25
test_build_prompt_target_vs_non_target() (*in module test.data_storage.test_llm_analyse*), 23
test_check_memory_usage() (*in module test.data_storage.test_paragraph_extraction*),

26
test_create_table_connection_failure() (in module test.data_storage.test_create_table), 20
test_create_table_insert_failure() (in module test.data_storage.test_create_table), 21
test_create_table_success() (in module test.data_storage.test_create_table), 21
test_custom_checkbox_class_exists() (in module test.frontend.test_index_css), 30
test_download_error() (in module test.data_storage.test_retry_failed_reports), 27
test_empty_failed_json() (in module test.data_storage.test_retry_failed_reports), 27
test_export_table_query_with_order() (in module test.data_storage.test_data_export), 21
test_export_table_query_without_order() (in module test.data_storage.test_data_export), 21
test_extract_paragraphs_from_pdf() (in module test.data_storage.test_paragraph_extraction), 26
test_failed_json_removed_after_success() (in module test.data_storage.test_retry_failed_reports), 27
test_favicon() (in module test.my_fastapi.test_main), 32
test_find_matching_paragraphs() (in module test.data_storage.test_paragraph_extraction), 26
test_get_all_data() (in module test.my_fastapi.test_main), 32
test_get_company_report_by_id() (in module test.my_fastapi.test_main), 32
test_get_data_by_id() (in module test.my_fastapi.test_main), 32
test_get_data_by_id_not_found() (in module test.my_fastapi.test_main), 32
test_get_indicator_by_id() (in module test.my_fastapi.test_main), 32
test_get_indicator_not_found() (in module test.my_fastapi.test_main), 32
test_get_indicators() (in module test.my_fastapi.test_main), 33
test_get_report_not_found() (in module test.my_fastapi.test_main), 33
test_get_reports() (in module test.my_fastapi.test_main), 33
test_index_imports_and_mounts() (in module test.frontend.test_index_js), 31
test_insert_matched_data() (in module test.data_storage.test_paragraph_extraction), 26
test_invalid_filename_skipped() (in module test.data_storage.test_retry_failed_reports), 28
test_is_valid_number() (in module test.data_storage.test_llm_analyse), 23
test_main_concurrent_processing() (in module test.data_storage.test_llm_analyse), 23
test_main_exports_all_tables() (in module test.data_storage.test_data_export), 22
test_main_pipeline_failure() (in module test.data_storage.test_data_storage_pipeline), 22
test_main_skip_and_execute() (in module test.data_storage.test_data_storage_pipeline), 22
test_main_standardization_pipeline() (in module test.data_storage.test_llm_standardize), 25
test_memory_management() (in module test.data_storage.test_paragraph_extraction), 26
test_no_failed_json() (in module test.data_storage.test_retry_failed_reports), 28
test_no_matching_paragraphs() (in module test.data_storage.test_retry_failed_reports), 28
test_parse_security_and_year() (in module test.data_storage.test_paragraph_extraction), 26
test_pdf_page_no_text() (in module test.data_storage.test_retry_failed_reports), 29
test_process_all_pdfs() (in module test.data_storage.test_paragraph_extraction), 26
test_process_report() (in module test.data_storage.test_paragraph_extraction), 27
test_process_row_convertible_false() (in module test.data_storage.test_llm_standardize), 25
test_process_row_invalid_result_value() (in module test.data_storage.test_llm_standardize), 25
test_process_row_nontarget_invalid_number() (in module test.data_storage.test_llm_analyse), 23
test_process_row_nontarget_json_error() (in module test.data_storage.test_llm_analyse), 24
test_process_row_success() (in module test.data_storage.test_llm_standardize), 25
test_process_row_target_json_and_nonjson() (in module test.data_storage.test_llm_analyse), 24
test_process_row_target_non_str_output() (in module test.data_storage.test_llm_analyse), 24
test_process_row_unit_match() (in module test.data_storage.test_llm_standardize), 25
test_read_completed_modules() (in module test.data_storage.test_data_storage_pipeline), 22
test_root_frontend_redirect() (in module

test.my_fastapi.test_main), 33
test_run_command_success_and_failure() (*in module test.data_storage.test_data_storage_pipeline*), 22
test_safe_json_parse_cleanup_and_errors() (*in module test.data_storage.test_llm_standardize*), 25
test_search_data_all_params() (*in module test.my_fastapi.test_main*), 33
test_search_data_no_param() (*in module test.my_fastapi.test_main*), 33
test_search_indicators_no_match() (*in module test.my_fastapi.test_main*), 33
test_search_indicators_with_name_and_theme() (*in module test.my_fastapi.test_main*), 34
test_search_reports() (*in module test.my_fastapi.test_main*), 34
test_search_reports_all_params() (*in module test.my_fastapi.test_main*), 34
test_successful_extraction() (*in module test.data_storage.test_retry_failed_reports*), 29
test_tailwind_config_contains_expected_keys() (*in module test.frontend.test_tailwind_config*), 31
test_update_result_executes_commit() (*in module test.data_storage.test_llm_analyse*), 24
test_update_standardized_execution() (*in module test.data_storage.test_llm_standardize*), 25
test_write_completed_module() (*in module test.data_storage.test_data_storage_pipeline*), 22

U

update_result() (*in module modules.data_storage.llm_analyse*), 6
update_standardized() (*in module modules.data_storage.llm_standardize*), 8

W

write_completed_module() (*in module modules.data_storage.main*), 9