BOSCH
Invented for life

inside.Docupedia Export

Author: Venkatesan Kannan (MS/EDB7-XC)
Date: 07-Jan-2025 06:36

Table of Contents

1. Introduction.
2. Configuration Layer.
3. Management Layer.
4. Concrete Checkers: Statement Checker, File Checker, Link Checker, Content Checker
5. Validation Layer.
6. Reporting Layer.

Introduction

This document describes the software architecture for the **Checkers Framework**, a system developed to automate the evaluation and validation of questions in the **Quality Check** (QC) process. The framework helps categorize questions into **Green**, **Yellow**, or **Red** statuses based on predefined criteria, ensuring a consistent and objective approach to quality assessments.

The **Checkers Framework** is designed to be flexible, allowing users to adjust the validation process according to the needs of specific projects. By using configuration settings, users can choose which checkers to activate or deactivate, tailoring the system to the requirements of each project.

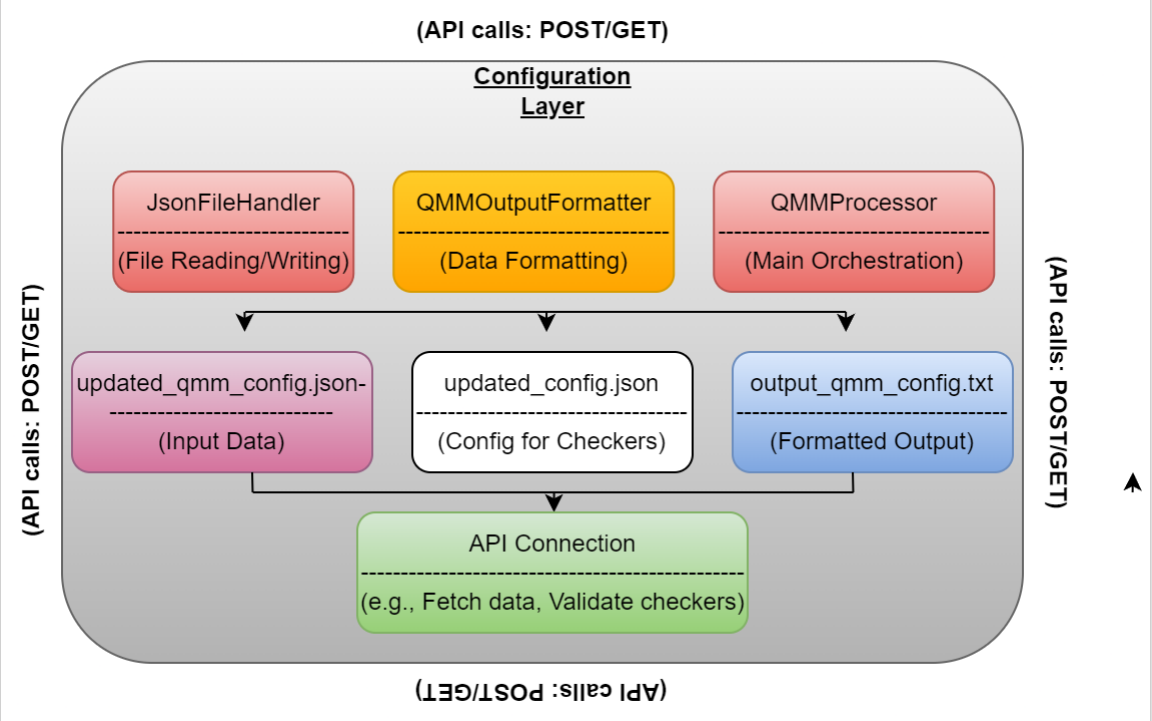
The framework also incorporates guidelines from the **QMM Team**, such as statement fields, reference files, and links for each question, to ensure the integration of standardized rules while still allowing customization. This balance between standardization and flexibility makes the **Checkers Framework** adaptable to different project needs and helps ensure that quality checks are accurate and reliable.

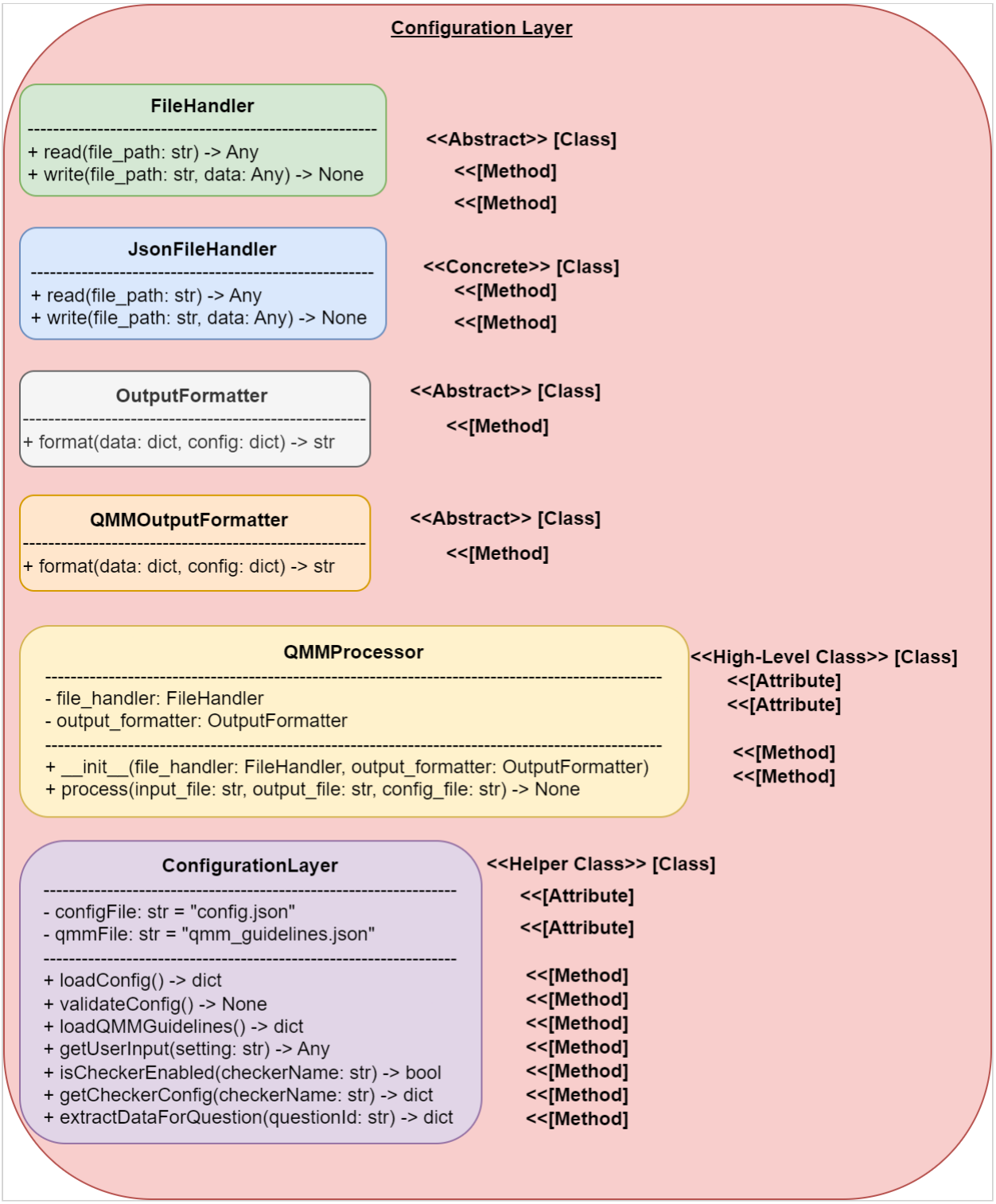
Quick Overview:

Checkers Framework Layer	Checkers Framework Action
Configuration Layer	**Action**: Load configuration settings like file paths, validation rules, and environment settings to ensure the process runs as expected.
Management Layer	**Action**: Initiate and control the execution of checkers based on the configuration. Organize the execution flow, managing dependencies between different tasks.
Concrete Checkers	**Action**: Perform actual checks on the data, such as validating files, comparing values, or checking if a condition is met (e.g., "Does file X exist?").
Validation Layer	**Action**: Assess the results of each checker and categorize them (e.g., a check result could be "Green" if everything is fine, "Yellow" for warnings, or "Red" for errors).
Reporting Layer	**Action**: Generate a report summarizing the check results and send it to stakeholders or save it for future reference.

****

1) Configuration Layer:





Configuration Layer Overview:

The **Configuration Layer** is focused on managing configurations based on user settings and QMM guidelines. By isolating this responsibility, the framework ensures a clear separation of concerns, modularity, and efficient management of configuration and guideline data

Key Considerations for the Configuration Layer:

1) Configuration File Availability:

Requirement: The `config.json` file must exist in the specified location and be accessible.

Handling:

- If the file is missing or cannot be read, the system will:
- Raise a `FileNotFoundError` with a detailed message specifying the expected file location.
- Log the error in a centralized log file for debugging purposes.

Path:

- The path to the `config.json` file should be configurable (e.g., via an environment variable or default hardcoded location).
- The user must ensure the file is present in the expected directory with proper read permissions.

2) Configuration File Completeness:

Requirement: The `config.json` file must include all required settings, such as checkers, file paths, and other mandatory fields.

Schema:

- A schema should define:
- Required keys: e.g., `checkers`, `paths`, `settings`.
- Data types for each key (e.g., `string`, `array`, `boolean`).
- Constraints: e.g., non-empty arrays, valid paths.

Handling:

- The `validateConfig()` method will:
- Parse the `config.json` file.
- Verify its structure adheres to the predefined schema.
- Raise a `ValueError` with specific details if validation fails.

Functionality:

Provide a default configuration file as a template for users.

Include descriptive error messages to guide users on fixing validation issues.

3) QMM Guidelines File is Mandatory:

Requirement: The `qmm_guidelines.json` file must exist and be accessible.

Handling:

- If the file is missing or unreadable:
- Raise a `FileNotFoundError` with the expected file location and detailed instructions for providing the file.
- Log the error and notify the user.

Functionality:

- The `loadQMMGuidelines()` method will attempt to load the file and validate its structure against a schema.
- If the file is missing, include fallback logic to continue running with limited functionality while alerting the user.

4) User Input Retrieved:

Requirement: The `getUserInput(setting)` method retrieves user-defined settings, such as enabling/disabling checkers or file preferences.

Handling:

- Validate user input to ensure:
- It matches allowed values or formats (e.g., boolean for enabling/disabling checkers, valid file paths for preferences).

- Invalid input raises a `ValueError` with clear guidance.

Functionality:

- Fallback defaults for optional settings

5) **Checker Configuration Is Well-Defined:**

Requirement: Each checker has a unique name, and its configuration must be present in the `config.json` file.

Handling:

- The `isCheckerEnabled()` method will:
- Verify if a specific checker is enabled in the configuration.
- Return a default state (e.g., `False`) or raise a warning if the checker's configuration is missing.
- The `getCheckerConfig()` method will:
- Fetch configurations for individual checkers.
- Provide a default configuration if missing, along with a warning message.

Functionality:

- Maintain a predefined list of valid checkers and their default configurations.
- Ensure missing or invalid checker configurations do not cause system failure.

6) **Question IDs Are Valid:**

Requirement: The `extractDataForQuestion(questionId)` method assumes the `questionId` exists in the `qmm_guidelines.json` file.

Handling:

- Validate the `questionId` against the data in `qmm_guidelines.json`.
- If the `questionId` is missing or invalid:
- Raise a `KeyError` with a descriptive message.
- Log the issue for debugging.

Functionality:

- Implement logging for invalid `questionId` usage.
- Provide a list of valid `questionId`s for user reference.

7) **Interaction with Other Layers:**

Requirement: Methods like `loadConfig()` and `extractDataForQuestion()` must provide data in a format compatible with the Management Layer and Validation Layer.

Handling:

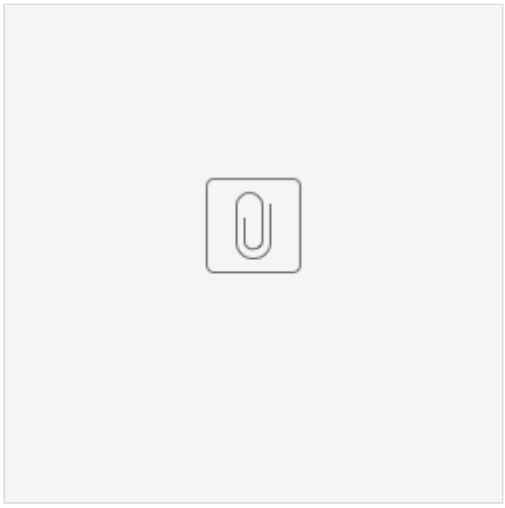
- Define clear output formats:
- Use dictionaries or structured JSON for data transfer.
- Include field names, data types, and constraints in documentation.
- Ensure dependent layers can handle the outputs without requiring additional conversions**.**

Functionality:

- Unit tests will verify compatibility between the Configuration Layer and other layers.
- Regularly review and update interfaces to accommodate new requirements.

Output from Configuration Layer:

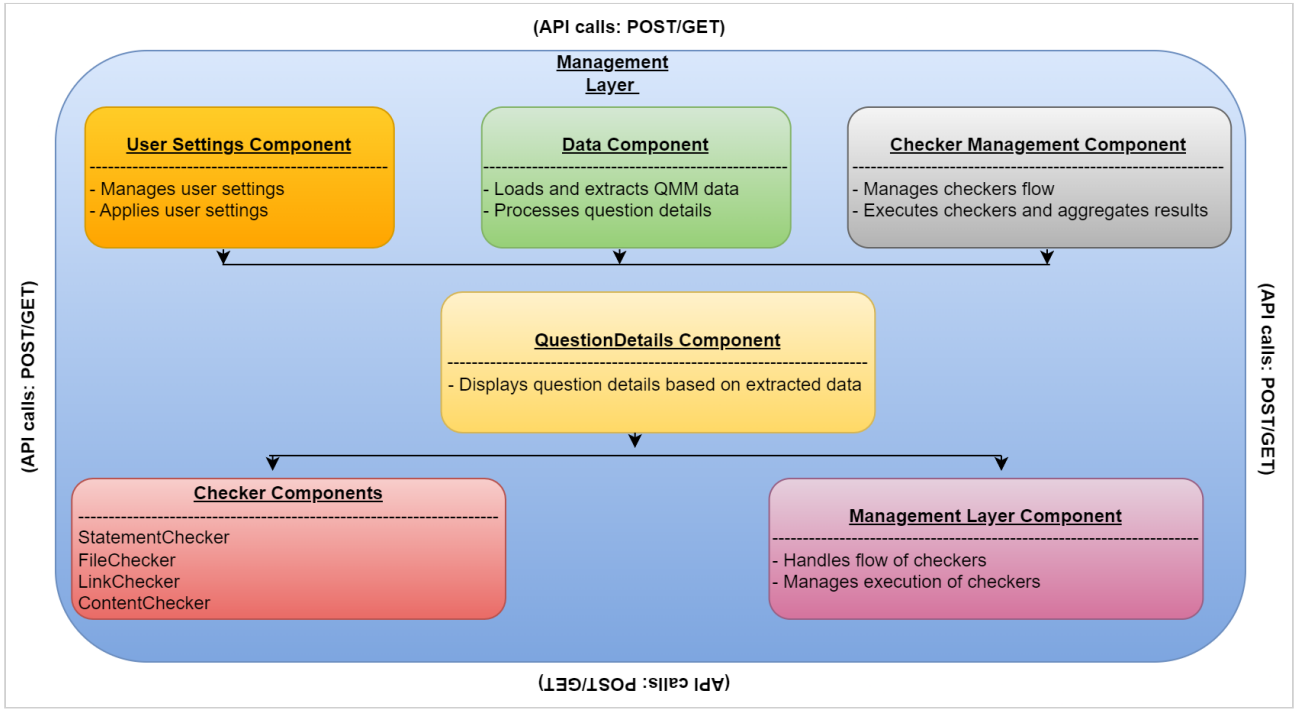
```
Formatted Output:
 ### Main Question: Tools Capability ###

Enabled Checkers:
StatementChecker: ready to Execute
FileChecker: ready to Execute
LinkChecker: ready to Execute
ContentChecker: Skipped (disabled)
Question ID: Q1
Sub Question ID: Q1.1
Question: Are the technical and non-technical internal, external stakeholder, market and legal requirements, identified, available, allocated, analyzed and agreed upon with the stakeholders?

Details:
 - Upload to Splunk
 - Upload Splunk image
 - Analyze document
 - Check Splunk Query
 - Digitization of the document
 - Download SharePoint file
 - Capture T&R data

Tools Capability:
 - Splunk
 - SharePoint

### Responses ###
Response ID: 1.1
  SafetyGap: All safety-requirements are in final state.
  SecurityGap: All security requirements are in final state.
  ConformityGap: All conformity requirements are in final state.
  AdditionalInfo: Gaps shown in Splunk please see attached ppt.
  NonSafetyRequirements: All non-safety, non-security and non-conformity requirements are in final state or are evaluated in the risk evaluation list.
Links:
  InternalStRS: https://sites.inside-share3.bosch.com/sites/105130/Documents/Management/Software_Releases/CAT5%20Series%20Releases/MRRPrem%20Cluster3%20CAT5/MRRprem_RA_SOP_Cluster3_Internal%20Stakeholder%20req.
CAT5.xlsx(1)?d=w138b1d6fcb234a18bd615b0182a21ab4
  ExternalStRSNotlinked: https://sites.inside-share3.bosch.com/sites/105130/Documents/Management/Software_Releases/CAT5%20Series%20Releases/MRRPrem%20Cluster3%20CAT5/MRRprem_RA_SOP_Cluster3_StakeholderReq_gaps_
AT5.xlsx(1)?d=wda7965875012492898c5bedfbc85f72a
  ExternalStRSOpenStatus: https://sites.inside-share3.bosch.com/sites/105130/Documents/Management/Software_Releases/CAT5%20Series%20Releases/MRRPrem%20Cluster3%20CAT5/MRRprem_RA_SOP_Cluster3_StakeholderReq_gaps
CAT5.xlsx(1)?d=wda7965875012492898c5bedfbc85f72a
  Summary: All requirements are available, up-to-date and analyzed.
  Conclusion: No gaps, all deviations evaluated in the risk evaluation list, stakeholder requirements discussed and agreed by the customer.


Configuration extracted and saved to output_qmm_config.txt
Framework created by: Kannan Venkatesan
```

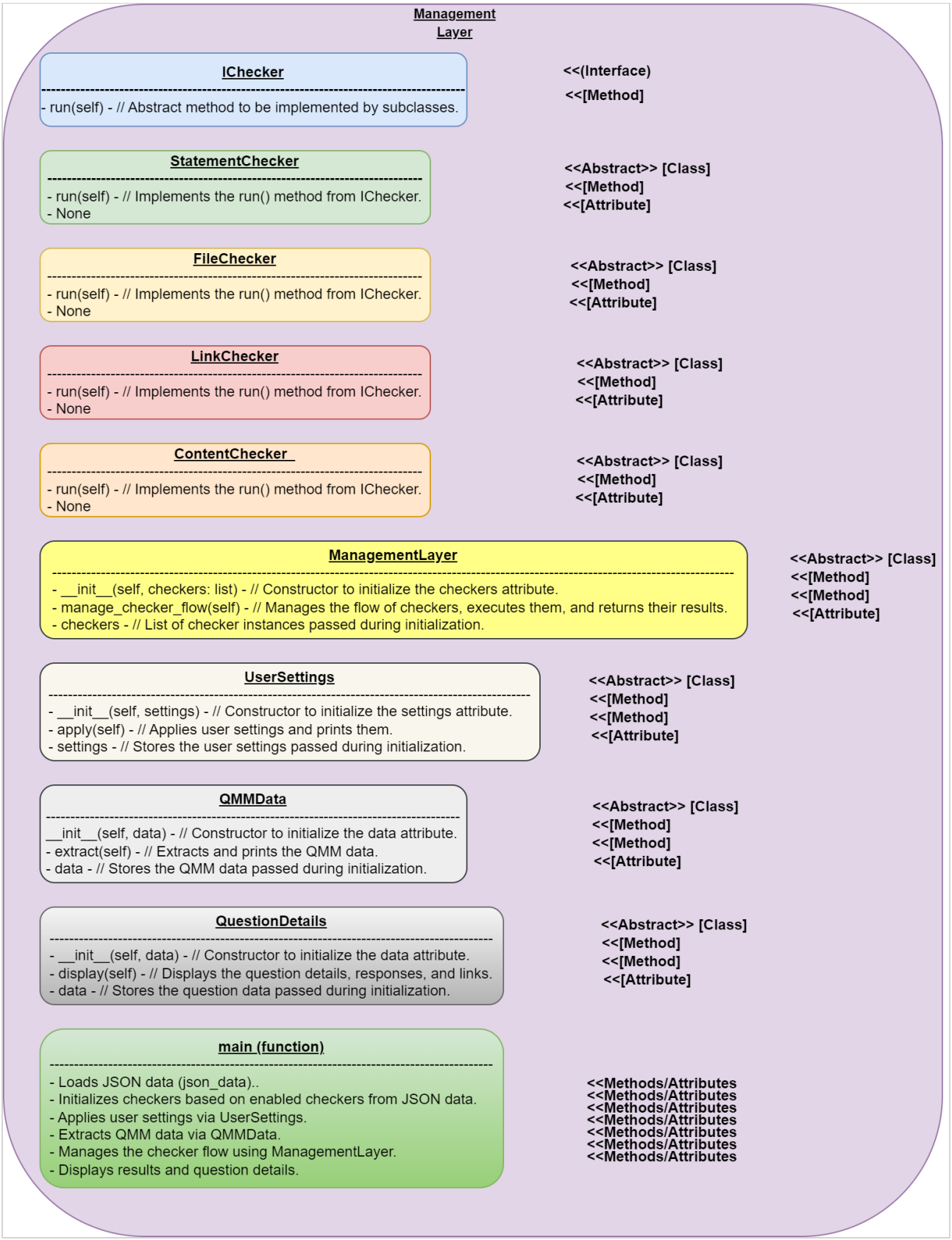


****

****

2) Management Layer:





Management Layer Overview:

The **Management Layer** class is the central orchestrator responsible for managing and executing checkers based on user settings and QMM guidelines. By isolating this functionality into its own layer, the framework ensures modularity, scalability, and seamless interaction between the **Configuration Layer** and **Validation Layer**. The Management Layer processes inputs, initializes checkers, handles their interdependencies, and aggregates results for further validation.

This layer ensures that user inputs and QMM data are effectively integrated into the system's execution flow, maintaining flexibility and robustness.

Key Considerations for the Management Layer:

1) Checker Initialization:

Requirement: Checkers must be initialized and configured according to the settings provided by the Configuration Layer.

Handling: Use the `initializeCheckers()` method to parse user input and QMM data, and load the appropriate configurations for each checker

Functionality: Ensure that all necessary checkers are properly initialized before execution. If a checker fails to initialize, raise an error and log details for debugging.

2) Checker Flow Management:

Requirement: Checkers must be executed in the correct order, considering dependencies between them

Handling: The `manageCheckerFlow()` method manages the execution sequence by resolving interdependencies and aggregating results.

Functionality: Ensure that the checker flow remains uninterrupted and dependencies are handled dynamically. Provide fallback mechanisms for optional checkers to prevent system-wide failures.

3) Integration with QMM Data:

Requirement: Data extracted from the QMM guidelines must be utilized for checker configurations and execution.

Handling: The `extractQMMDataForCheckers()` method parses the QMM guidelines and validates the data's completeness and correctness.

Functionality: Ensure that QMM data is readily accessible to all checkers and aligned with their input requirements. Log errors for any missing or invalid data.

4) User Input Application:

Requirement: User-provided settings must override default configurations for checkers as needed.

Handling: The `applyUserSettings()` method validates and applies user inputs received via the **Configuration Layer.**

Functionality: Support dynamic updates to checker configurations based on user preferences. Notify users of invalid inputs and maintain default values when necessary.

5) Aggregated Results for Validation:

Requirement: Results from all executed checkers must be aggregated and formatted for use by the **Validation Layer**.

Handling: The `manageCheckerFlow()` method compiles the outputs from all checkers into a structured format (e.g., JSON or dictionary).

Functionality: Ensure that outputs are consistent and compatible with the **Validation Layer**. Provide clear error reports for any failed checkers or discrepancies.

6) Dependency on Configuration Layer:

Requirement: The Management Layer relies on inputs from the Configuration Layer, including checker configurations, user settings, and QMM data

Handling: Validate that all required inputs are available before executing any checkers. If inputs are incomplete or missing, raise appropriate errors and notify the user.

Functionality: Maintain seamless integration with the **Configuration Layer** and ensure compatibility with its outputs.

7) Interaction with Other Layers:

Requirement: Methods like `initializeCheckers()`, `manageCheckerFlow()`, and `applyUserSettings()` must handle data in a format compatible with the **Configuration Layer**, **Concrete Checkers**, and **Validation Layer** to ensure seamless integration across components.

Handling:

Configuration Layer: Provide user input and QMM data from the Configuration Layer into well-structured formats like dictionaries or JSON to initialize and configure checkers effectively. Raise appropriate warnings or errors for missing or invalid configurations.

Concrete Checkers: Provide each checker with its specific configuration and QMM data in a consistent format. Ensure execution flow between checkers is maintained without any dependency issues.

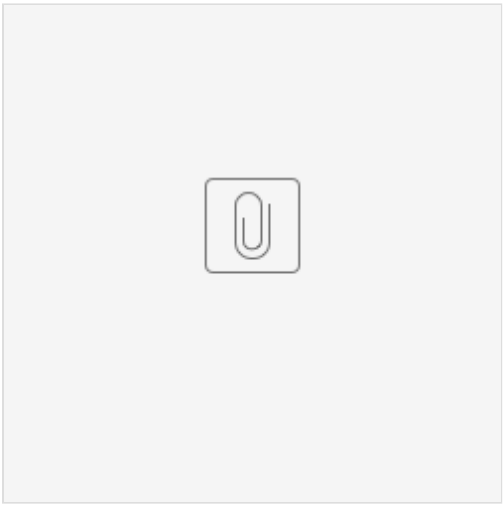
Validation Layer: Aggregate results from all checkers into a unified structure before passing them to the Validation Layer for further analysis and validation. Ensure no additional transformations are needed.

Functionality:

- We will create integration tests to verify that the data received from the Configuration Layer aligns with the Management Layer's expected input formats.
- Validate that the outputs of `manageCheckerFlow()` meet the requirements of the Validation Layer, ensuring compatibility and correctness.
- Simulate interactions with Concrete Checkers to confirm proper initialization, dependency handling, and execution flow.

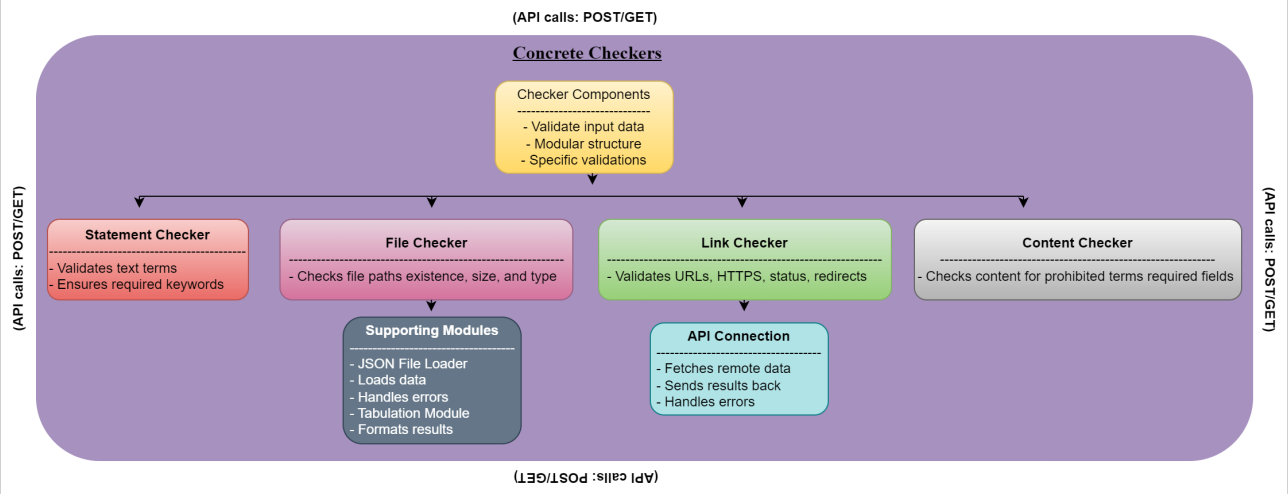
Output from Management Layer:

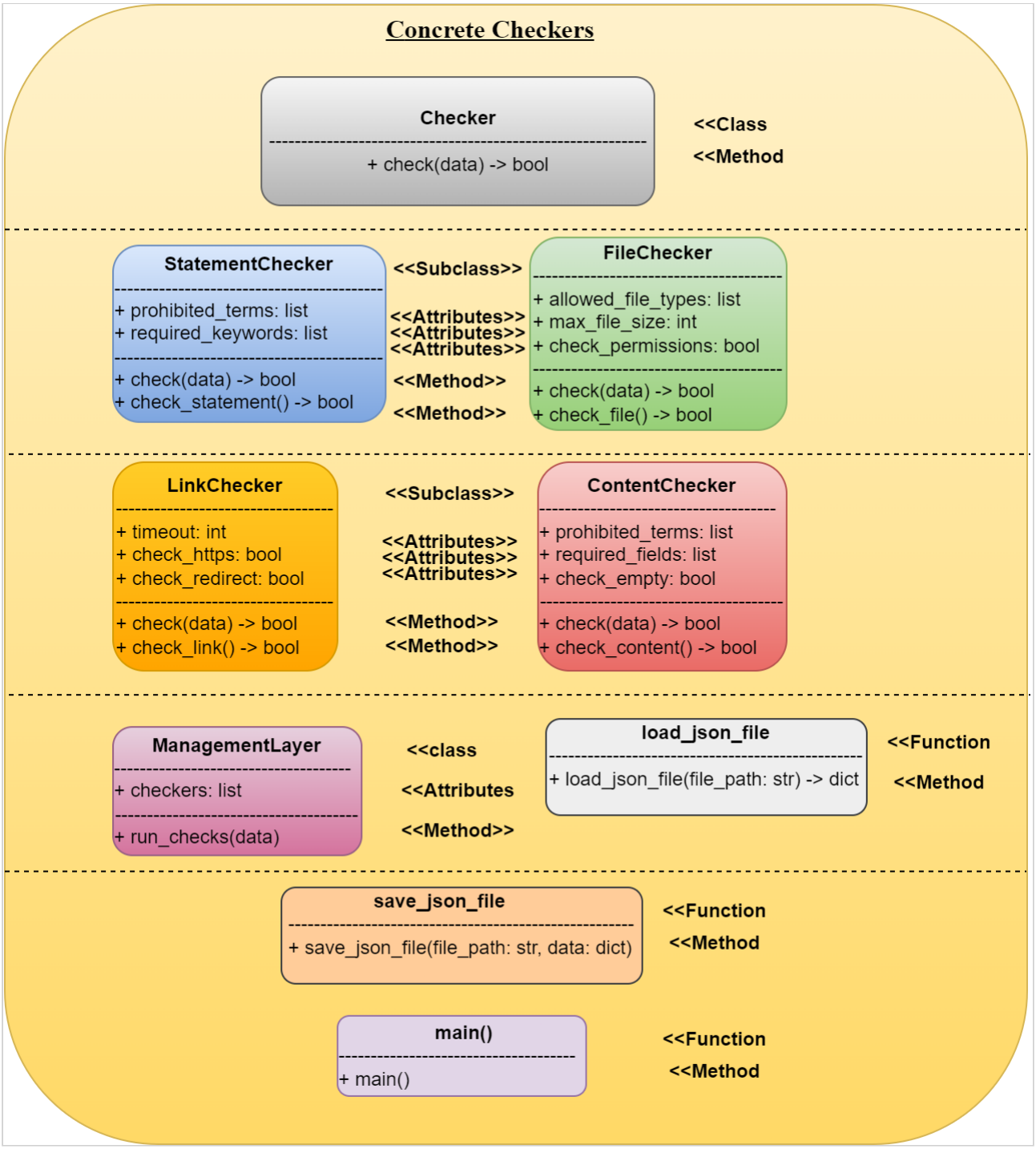
```
ekc3kor@KOR-C-0051G | internet: ok+docker:~/framework/Management_Layer$ python3 management_solid_2.py
Framework created by: Kannan Venkatesan
User settings applied: ['StatementChecker', 'FileChecker', 'LinkChecker', 'ContentChecker']
QMM data extracted: {'StatementChecker': {}, 'FileChecker': {}, 'LinkChecker': {}, 'ContentChecker': {}}
### Checker Results ###
StatementChecker: StatementChecker ready to Execute
FileChecker: FileChecker skipped (disabled)
LinkChecker: LinkChecker ready to Execute
ContentChecker: ContentChecker ready to Execute
```



****
****
****

Concrete Checkers:





The **Concrete Checkers**—**Statement Checker**, **File Checker**, **Link Checker**, and **Content Checker**—are specialized components designed to **validate** different types of data. The **Statement Checker** ensures that statements contain **required keywords** and avoid **prohibited terms**. The **File Checker** verifies **file existence**, **type**, **size**, and **permissions**. The **Link Checker** validates **URLs** for **accessibility** and **proper protocol**. Lastly, the **Content Checker** ensures that structured content meets the necessary **field** and **term requirements**. Together, these checkers collaborate to maintain **data accuracy** and provide **flexible validation** based on **configurable criteria**.

Key Considerations for the Checker Layer:

1) Statement Checker Class (`StatementChecker`):

Requirement:

- Validating the structure and content of statements based on prohibited terms and required keywords
- prohibited_terms = ["password", "123456", "admin", "confidential", "forbidden"]

Handling:

- The statement must be a non-empty string
- The statement should not contain any prohibited terms.
- The statement must include all required keywords.

Functionality:

- **check_statement**: This method checks whether the provided statement complies with the configuration (i.e., doesn't contain prohibited terms and includes required keywords).

Attributes:

- `prohibited_terms` : A list of terms that, if found in the statement, result in a failure.
- `required_keywords` : A list of keywords that must be present in the statement for it to pass.
- required_keywords = ["safety, quality, performance"] for each software release.

Edge Cases:

- Empty or non-string input should fail.
- If the required keywords are missing or if prohibited terms are present, the check fails.

2) File Checker Class (`FileChecker`):

Requirement:

- Ensure that the provided file meets various criteria, such as the file type, size, and permissions.

Handling:

- The file must exist, be accessible, and meet the allowed file types and size constraints.
- If `check_permissions` is enabled, the file must be writable.

Functionality:

- **check_file**: This method checks whether the file exists, whether it is of an allowed type, whether its size is within the limit, and whether the necessary permissions are in place.

Attributes:

- `allowed_file_types` : A list of file extensions that are allowed.
- `max_file_size` : The maximum allowable file size.
- `check_permissions` : A boolean that determines if file write permissions should be checked.

Edge Cases:

- Missing file, wrong file type, file size exceeding the limit, or permission issues will cause failure.

3) Link Checker Class (`LinkChecker`)

Requirement:

- Verify the validity of a URL (checking status code, HTTPS protocol, etc.).

Handling:

- The URL must be reachable and return a successful status code (200 OK).
- The URL must start with HTTPS (if specified).
- check redirect behavior.

Functionality:

- **check_link**: This method sends an HTTP GET request to the URL and checks its status code.

Attributes:

- `timeout` : The timeout duration for the HTTP request.
- `check_https` : A boolean to enforce HTTPS protocol.
- `check_redirect` : A boolean that determines whether redirects are allowed during the request.

Edge Cases:

- Invalid URLs or unreachable URLs will cause the check to fail.
- If the URL does not start with HTTPS (when `check_https` is enabled), the check will fail.

4) Content Checker Class (`ContentChecker`)

Requirement:

- Validate content by checking prohibited terms, required fields, and ensuring it is not empty.

Handling:

- The content must not contain any prohibited terms.
- The content must include all required fields no missing field (e.g., version, author).
- The content must be non-empty.

Functionality:

- **check_content**: This method verifies whether the content adheres to the requirements (e.g., no prohibited terms, includes required fields).

Attributes:

- `prohibited_terms` : A list of terms that are not allowed in the content.
- `required_fields` : A list of fields that must appear in the content.
- `check_empty` : A boolean indicating whether to check if the content is empty.

Edge Cases:

- If the content is empty or contains prohibited terms, the check will fail.
- If the required fields are missing, the check will fail.

Handling and Interactions Across Checker Classes:

Inter-Class Communication:

Requirement:

- The checker classes should be independent, but able to work together within the management layer (e.g., `ManagementLayer`) to perform comprehensive validation across different data types (statements, files, links, content).

Handling:

- The `ManagementLayer` class should coordinate the execution of individual checkers based on the type of data being validated.
- When the `ManagementLayer` runs the checks, it dynamically selects the appropriate checker based on the data type (e.g., "statement", "file", "link", "content").
- Each checker returns a boolean indicating whether the data passed or failed the validation.

Functionality:

- `ManagementLayer` uses a collection of checkers and applies the correct ones based on the data being validated.
- It aggregates results and generates a summary of overall success or failure.

Edge Cases:

- The checkers should handle various edge cases gracefully, including:
- Invalid or missing input (e.g., missing file paths, invalid URLs, or incomplete statements).
- Inconsistent configurations or missing attributes in the configuration files.
- HTTP errors (e.g., timeouts, 404 not found, etc.).
- Incorrect content formats (e.g., JSON with missing required fields).

Interaction with Other Layers (Management Layer):

Requirement:

- The checker layer must interact seamlessly with other layers such as the **Management Layer** and **Validation Layer.**

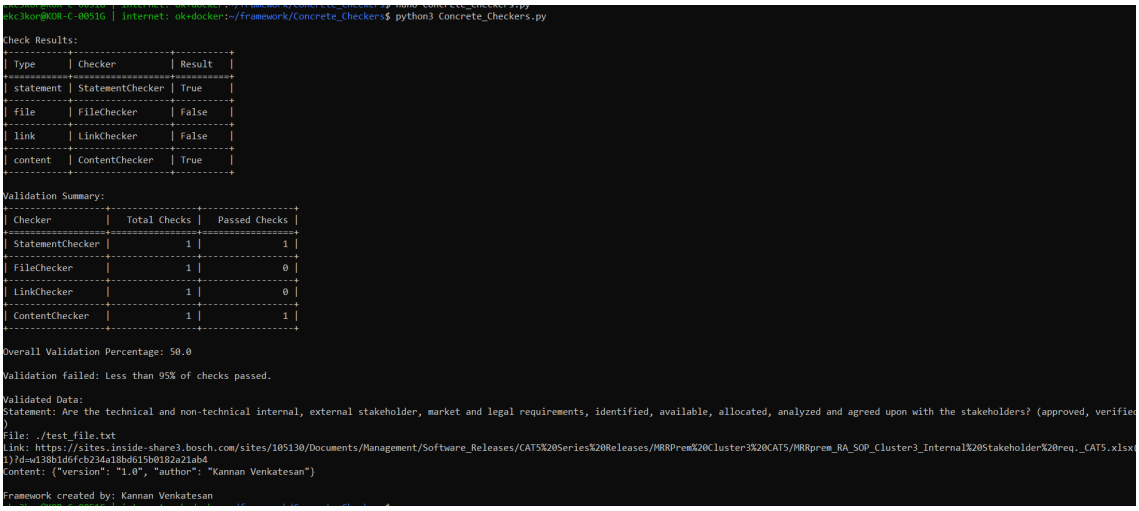
Handling:

- The `ManagementLayer` is responsible for orchestrating the individual checkers and consolidating their results into a summary format.
- The output of the checker layer (whether a check passes or fails) must be in a structured format that the other layers (e.g., reporting) can process.

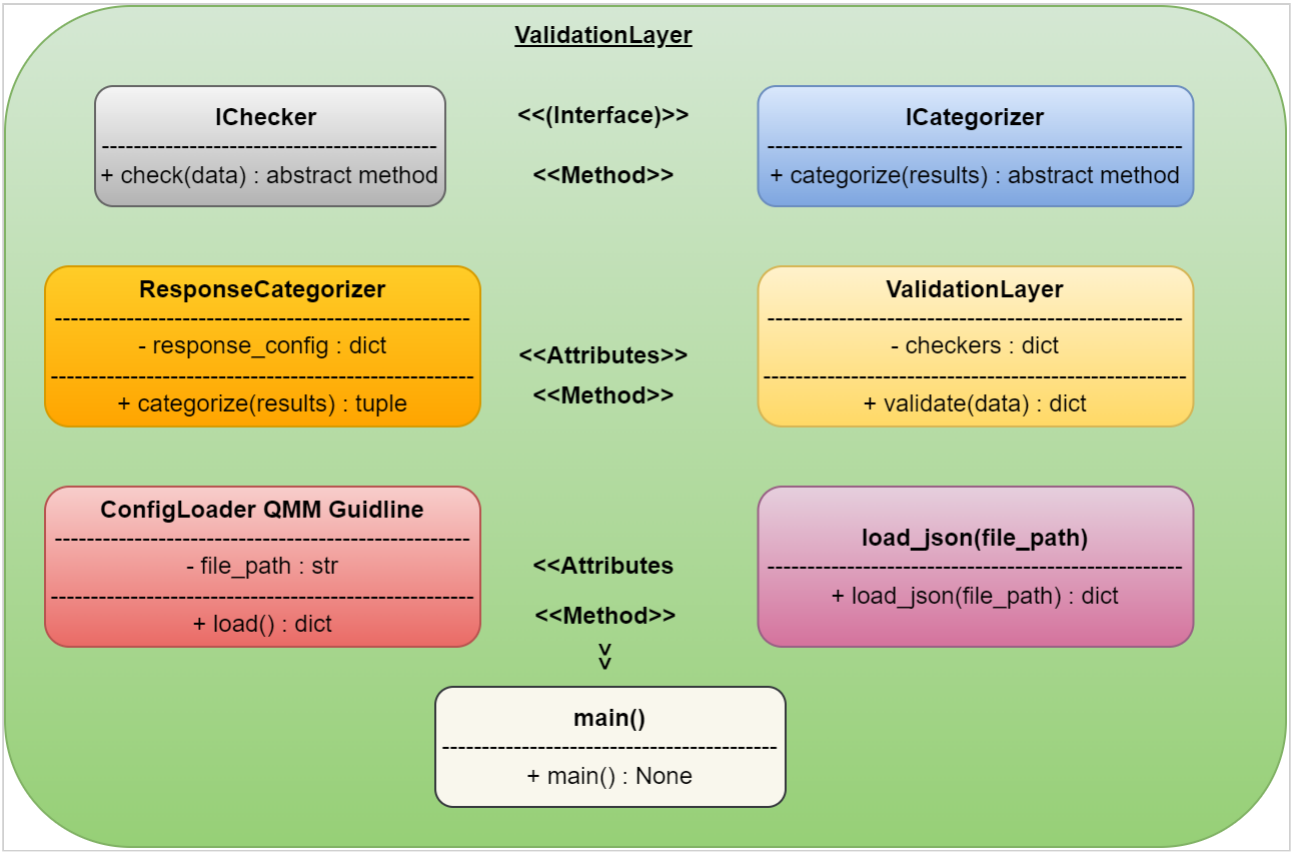
Functionality:

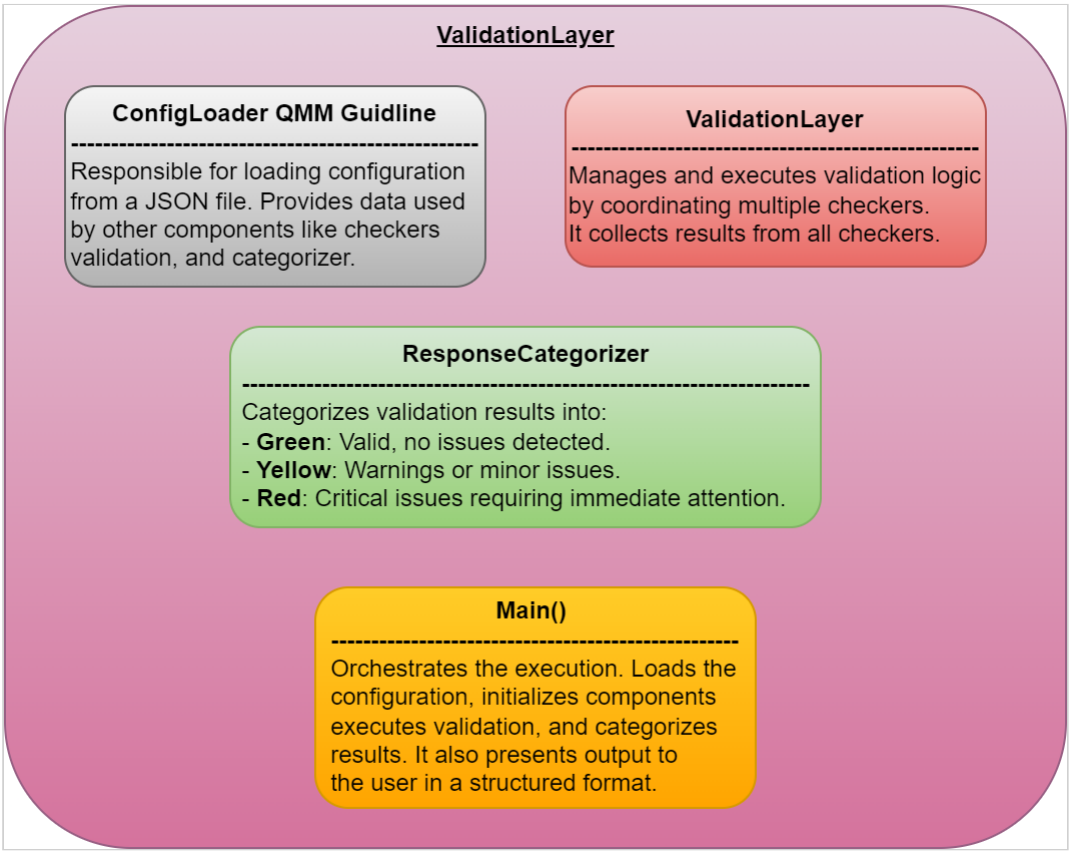
- The check results are returned as a dictionary with keys representing the types of checks (e.g., "statement", "file", "link", "content").
- The validation summary provides an aggregated view of the checks, including the percentage of checks passed.

Output from Concrete Checkers Layer:



****
****





The **ValidationLayer** is responsible for ensuring that the **Concrete Checkers**—such as **StatementChecker**, **FileChecker**, **LinkChecker**, and **ContentChecker**—run correctly according to the user's input. It validates that each checker executes as per the defined requirements.

After running the checks, the **ValidationLayer** categorizes the results into three statuses—**Red**, **Yellow**, or **Green**—based on predefined rating guidelines. These categories reflect the validity of the input, with **Green** indicating valid input, **Yellow** indicating potential issues, and **Red** signaling critical errors.

	Green	Yellow	Red
Traceability Links	Established links for all data	Link available but not accessible	Links missing or incomplete.
Requirement Status	All requirements are final and up-to-date.	Some non-critical requirements are missing agreement.	Critical requirements are missing or outdated.
Mandatory Attributes	All attributes are filled for all requirements.	Some attributes are missing or incomplete.	Many mandatory attributes are missing.
File Availability	All files are accessible and correct.	Some files are inaccessible or have minor issues.	Key files missing, corrupted, or not opening, or not the correct name of the file

Key Considerations for the Validation Layer:

Checker Execution:

Requirement:

- Ensure each checker runs according to the defined validation criteria.

Handling:

- The `validate()` method loops over the checkers, checking if the required data exists, then calls the `check()` method for each checker.

Result Aggregation and Categorization:

Requirement:

- Aggregate results from all checkers and categorize them into defined statuses.

Handling:

- After validation, results are categorized into Red (critical), Yellow (warning), or Green (valid), based on predefined levels.

Functionality:

- This enables the system to provide a clear view of all checks, allowing users to identify critical issues and address them promptly.

Error Handling:

Requirement:

- Ensure proper error handling when validation fails.

Handling:

- The `validate()` method checks if the necessary data for validation exists, providing a mechanism for early termination if issues are detected.

Functionality:

- Helps in preventing runtime errors by handling missing or malformed data early in the process.

Mapping Results to Categories:

Requirement:

- Transform validation results into human-readable categories

Handling:

- The `categorize()` method maps the results from Red, Yellow, and Green to categories like "Critical", "Warning", and "Valid", respectively.

Validate Workflow:

Requirement:

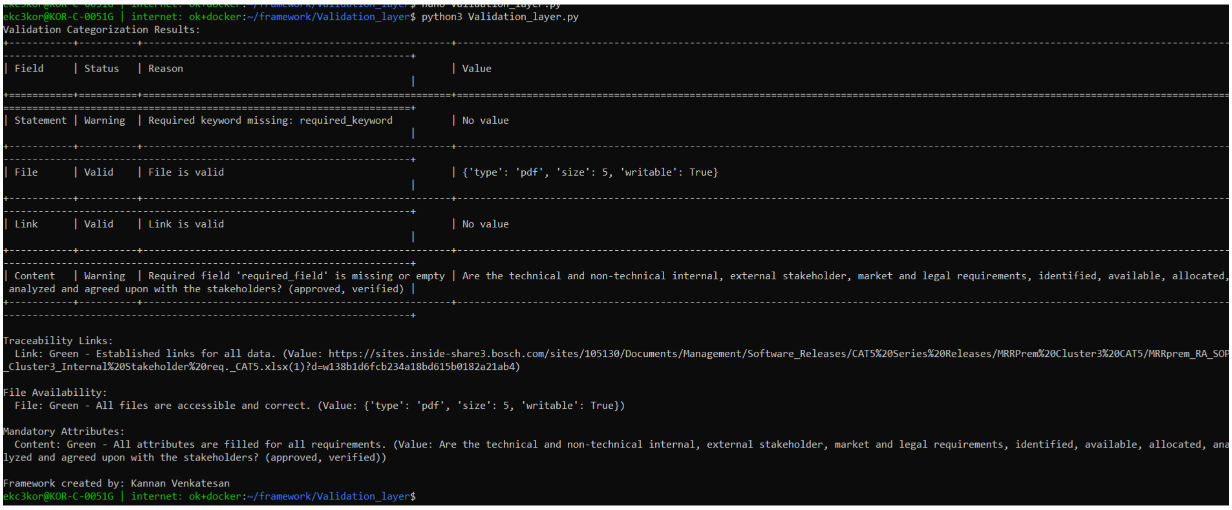
- Ensure the validation process follows the correct sequence and steps.

Handling:

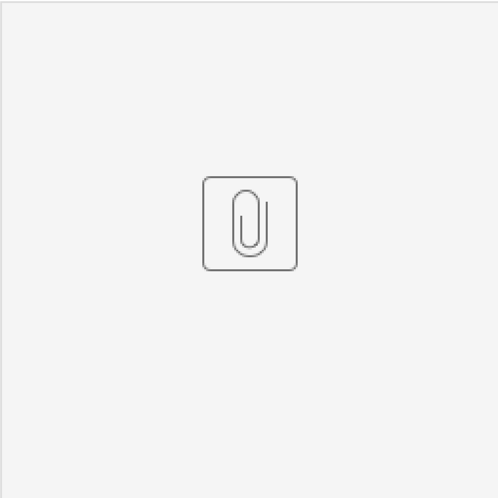
- The main function validates the flow of the process by ensuring that validation is executed before categorization and that all necessary steps are completed in the right order.

Functionality:

- Acts as the checker for the validation workflow, ensuring that the system follows the intended sequence of operations and that no steps are skipped.

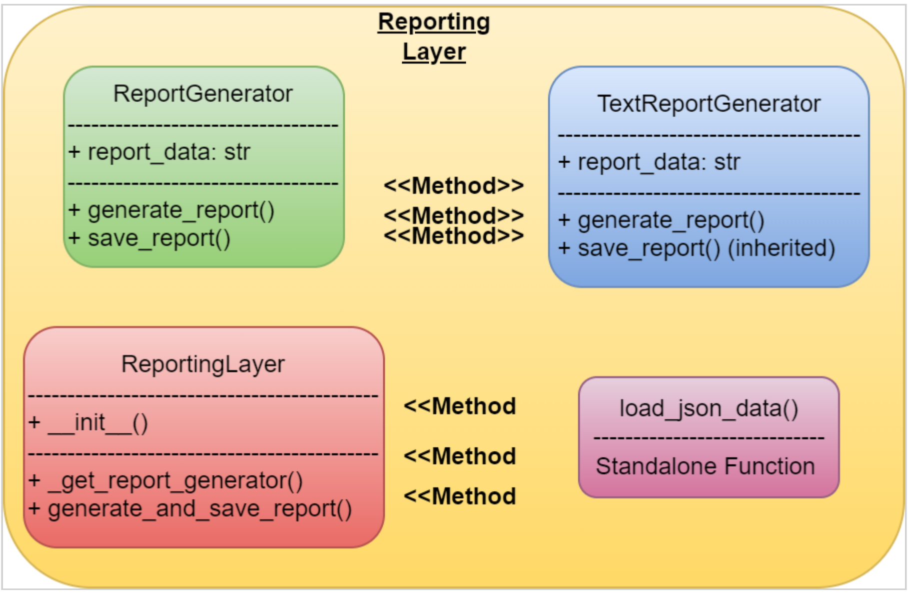


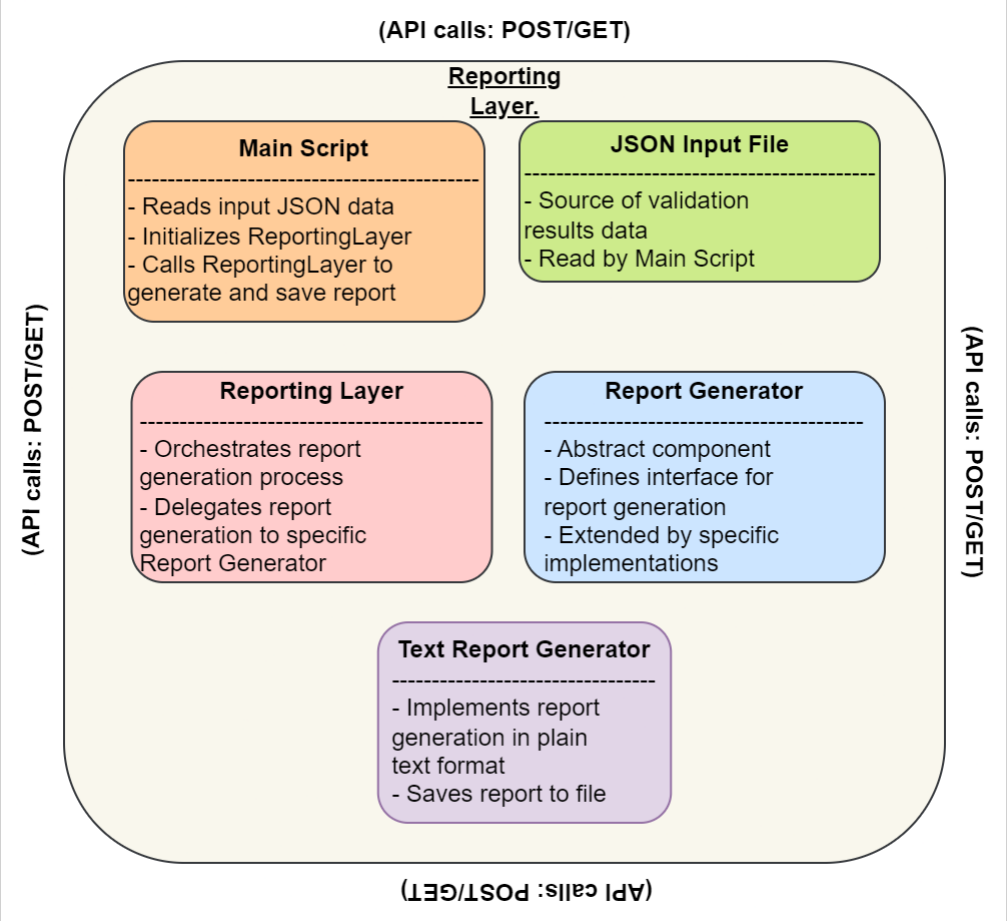
```
ekc3kor@KOR-C-0051G | internet: ok+docker:~/framework/Validation_layer$ python3 Validation_layer.py
Validation Categorization Results:
+-----------+----------+-------------------------------------------------------+----------------------------------------
| Field     | Status   | Reason                                                | Value
+===========+==========+=======================================================+========================================
| Statement | Warning  | Required keyword missing: required_keyword            | No value
+-----------+----------+-------------------------------------------------------+----------------------------------------
| File      | Valid    | File is valid                                         | {'type': 'pdf', 'size': 5, 'writable': True}
+-----------+----------+-------------------------------------------------------+----------------------------------------
| Link      | Valid    | Link is valid                                         | No value
+-----------+----------+-------------------------------------------------------+----------------------------------------
| Content   | Warning  | Required field 'required_field' is missing or empty | Are the technical and non-technical internal, external stakeholder, market and legal requirements, identified, available, allocated,
 analyzed and agreed upon with the stakeholders? (approved, verified) |
+-----------+----------+-------------------------------------------------------+----------------------------------------

Traceability Links:
  Link: Green - Established links for all data. (Value: https://sites.inside-share3.bosch.com/sites/105130/Documents/Management/Software_Releases/CAT5%20Series%20Releases/MRRPrem%20Cluster3%20CAT5/MRRprem_RA_SOP
_Cluster3_Internal%20Stakeholder%20req._CAT5.xlsx(1)?d=w138b1d6fcb234a18bd615b0182a21ab4)

File Availability:
  File: Green - All files are accessible and correct. (Value: {'type': 'pdf', 'size': 5, 'writable': True})

Mandatory Attributes:
  Content: Green - All attributes are filled for all requirements. (Value: Are the technical and non-technical internal, external stakeholder, market and legal requirements, identified, available, allocated, ana
lyzed and agreed upon with the stakeholders? (approved, verified))

Framework created by: Kannan Venkatesan
ekc3kor@KOR-C-0051G | internet: ok+docker:~/framework/Validation_layer$
```



****

Reporting Layer:





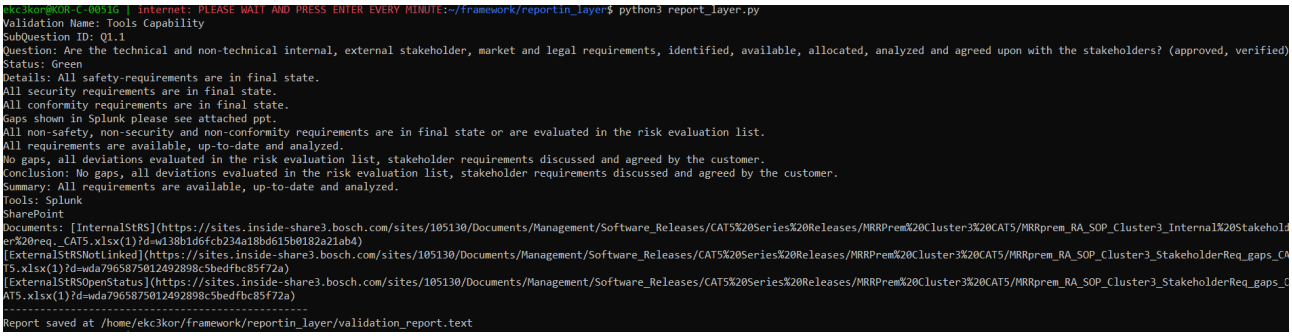
The **Reporting Layer** serves as the final layer in the system, responsible for collecting and processing results from previous layers. It ensures that raw data and intermediate results are transformed into a human-readable format, making them easier to interpret and analyze. This layer orchestrates the report generation process by delegating tasks to appropriate report generators and consolidating the information into a structured and meaningful report.

Key responsibilities of the Reporting Layer include:

- **Data Aggregation**: Collecting results from upstream layers for further processing.
- **Report Generation**: Transforming raw results into a well-organized, human-readable format, making it suitable for stakeholders.
- **Output Logs**: Displaying the generated report in the console for immediate review.
- **Report Storage**: Saving the generated report locally on the customer's system to facilitate further analysis or archival

This layer acts as the final touchpoint for ensuring the data is presented effectively, serving both operational and analytical purposes. It combines flexibility with clarity, adapting to different report types while maintaining a streamlined and consistent output process.

Final output from Reporting Layer:



Conclusion:

This framework is a **complete solution** designed to **automate** the **quality process**, reducing the need for **manual effort** and ensuring **accurate results**. Each **layer** in the framework is carefully designed to work **independently**, with a clear purpose in performing **quality checks**. From setting up **configurations** to generating **reports**, every layer plays a specific role in organizing results according to predefined **rating guidelines**, providing a clear and structured assessment of **quality**.

The framework follows the **SOLID principles**, which are fundamental for creating well-structured and **reliable systems**. By adhering to these principles, the framework ensures that its components are simple to **maintain**, **extend**, and adapt to future needs. The **modular design** allows each layer to function on its own while smoothly connecting with others, making the system **flexible** and **easy to understand**.

In summary, this framework makes the **quality assurance process** efficient, **reliable**, and **straightforward**. It has been thoughtfully designed to meet current needs while being adaptable for **future improvements**, reflecting a **practical** and **user-friendly approach** to **quality management**.

****
****
****