# ReportIQ Wiki

This resource aims to provide in-depth details about reportiq’s inner workings, design principles, and methods for extending functionality, with complementary diagrams for visual reference. Please see [ADO ReportIQ Story](https://dev.azure.com/PSJH/Providence%20Global%20Center/_workitems/edit/2374271/) also for reference to user stories and sprints.

## Table of Contents

1. [Detailed Code Explanation](#detailed-code-explanation)
   * Class and Method Definitions
   * Design Patterns Utilized
   * Exception Handling
2. [Low-Level Design](#low-level-design)
   * Data Flow and Interaction
   * Class Hierarchy
   * Responsibilities of Each Component
3. [Process Flow](#process-flow)
   * Step-by-Step Execution
   * Decision Points
   * Error Paths and Handling
4. [Extending Functionality](#extending-functionality)
   * Designing New Processors
   * Context Customization
   * Integration with Additional APIs
5. [Deployment and Scaling](#deployment-and-scaling)
   * Environment Considerations
   * Optimizing Performance
   * Handling High Loads

## Detailed Code Explanation

### Class and Method Definitions

**FileProcessor (Abstract Class)**

* Purpose: Acts as a blueprint for specific file processors. Enforces implementation of a process() method, which dictates how a file type should be handled.
* Method: process() - Abstract method to be overridden by subclasses like PowerBIProcessor.

**PowerBIProcessor (Subclass of FileProcessor)**

* Purpose: Handles the processing of Power BI report files. Extracts necessary data from .Report and .SemanticModel folders within the uploaded zip file.
* Methods:
  + \_\_init\_\_(zip\_file) - Initializes the processor with the uploaded zip file.
  + process() - Implements extraction logic for report.json and relevant .tmdl files.

**FileProcessorFactory**

* Purpose: Abstracts file type-specific processing logic. Provides a method to retrieve the appropriate processor based on user selection.
* Method: get\_file\_processor(file\_type, \*args) - Returns an instance of the appropriate processor class.

**SetContext (Abstract Class)**

* Purpose: Establishes a contract for context setting, crucial for crafting prompts that guide the completion model.
* Method: set\_context() - Abstract method to be tailored by subclasses.

**PowerBIContext (Subclass of SetContext)**

* Purpose: Constructs Power BI-specific system and user prompts for documentation generation.
* Methods:
  + \_\_init\_\_(file\_contents, user\_context) - Initializes context with file contents and user-provided information.
  + set\_context() - Constructs the prompt, incorporating both file-specific and user-defined context.

**CompletionModel**

* Purpose: Facilitates interaction with the API, sending constructed prompts and handling streaming responses.
* Methods:
  + \_\_init\_\_(prompt) - Initializes the model with system and user prompts.
  + generate\_documentation() - Sends API requests and processes the streamed output for documentation generation.

### Design Patterns Utilized

* **Factory Pattern:** Both FileProcessorFactory and SetContextFactory employ this pattern to abstract the creation logic of objects related to file processing and context setting.
* **Strategy Pattern:** Implementation of file processing and context setting behaviors can be varied by creating different subclasses of FileProcessor and SetContext, allowing strategic adjustment according to file type.

### Exception Handling

The application is designed with robust error handling to manage file processing anomalies:

* **Try-Except Blocks:** Encapsulated within processing methods like PowerBIProcessor.process(), these blocks handle exceptions such as zipfile.BadZipFile and general exceptions, providing user warnings through Streamlit’s interface.

## Low-Level Design

### Data Flow and Interaction

This section dissects how data flows through the application, highlighting interactions between components:

* **Initial User Interaction:** Starts with file type selection and upload. User context is optional but enhances result specificity.
* **File Processing:** The factory retrieves the processor based on file type and executes its process() method to extract content.
* **Context Setting:** Context factories generate file-type-specific prompts, incorporating user input where available.
* **Documentation Generation:** The completion model interacts with APIs using the set prompts, producing streamed outputs.

### Class Hierarchy

* Refer to Low level diagram for more details on the architecture.

FileProcessor (ABC)   
 └── PowerBIProcessor  
SetContext (ABC)  
 └── PowerBIContext  
  
FileProcessorFactory -> SetContextFactory -> CompletionModel

### Responsibilities of Each Component

* **FileProcessor and Subclasses:** Responsible for accurate extraction of pertinent data from uploaded files.
* **Factories:** Maintain modularity, offering an extensible framework for adding new file processors and context setters.
* **CompletionModel:** Interfaces with external APIs, ensuring the transformation of raw data into structured documentation.

## Process Flow

### Step-by-Step Execution

1. **File Type Selection:** User selects desired file type from dropdown.
2. **File Upload:** User uploads a zip file, with instructions provided based on selection.
3. **File Processing:** Processor extracts and processes data using file-type-specific logic.
4. **Context Setting:** Prompts are generated based on processed data and optional user context.
5. **Documentation Generation:** Completion model uses prompts to interact with API, producing streamed documentation output.

### Decision Points

* **File Type Determination:** Managed by factory pattern, allowing differentiation and direction to correct processor.
* **Optional User Context:** Incorporates additional user information, enriching prompt structure for more precise documentation.

### Error Paths and Handling

* **File Processing Errors:** Managed through error handling in processors, returning informative feedback to user interface.

## Extending Functionality

### Designing New Processors

To support additional file types:

1. **Create a Processor Class:** Extend FileProcessor and implement specific logic in the process() method.
2. **Update Processor Factory:** Integrate the new processor class within get\_file\_processor() logic.

### Context Customization

Adding new contexts involves:

1. **Create a Context Class:** Implement specific logic for prompt generation within the set\_context() method.
2. **Update Context Factory:** Incorporate the new context class within get\_context() logic.

### Integration with Additional APIs

To expand API interactions:

1. **Modify CompletionModel:** Adjust parameters and request structure as needed.
2. **Adapt Prompt Structure:** Ensure new API requirements are met within context setting methods.