API Description Tool: the requirements specification V2

Document Status: in development
Version: 2.0
Last Updated: @August 20, 2025
Author: @Volodymyr Turovets

1. Introduction

1.1 Purpose and Scope

The purpose of the document is to provide high-level requirements for a tool for business analysts

(a) to build a business-oriented description of a given application programming interface.

In scope

- API specifications performed according to Openapi: 3.0.x family (developer-level description, any 3.0.x); data source: YAML file; media type - JSON only.

Out of scope

- any description other than provided in YAML file;
- OpenAPI 3.1.x.
- integration with Confluence;
- any media type other than JSON.

1.2 Definitions and Acronyms

This section provides definitions of terms and acronyms used throughout the document.

Term/Acronym	Definition
API specification	The technical specification of an API that meets Openapi 3.0.x notation
API description	A tabular API description for both technical and non-technical users, providing detailed explanations (e.g. formulas), references, and examples; always keep the current version of the API
Change log	A Confluence table containing documented changes to the API in a predefined format.

1.3 System Overview

The application provides a capability to

(a) create the tabular API description based on the API technical specification.

2. Overall Description

2.1 Product Perspective

2.1 The application is primarily considered as a standalone portable software tool. The output files are compatible with MS Excel/Word 365. The basic capability is to create the tabular API description based on the API technical specification provided (YAML file).

2.2. An integration with Confluence is considered as a second step.

2.3 An advanced Graphical User Interface (GUI) is developed as a third step.

2.4 A web service with a natural language interface is developed as a fourth step.

2.2 User Classes and Characteristics

Description of the users who will use the system:

User Class	Description	Responsibilities
Customer	a person who runs the app	provides input data, change configuration settings
Developer	a person who develops the application	builds/debugs the application by using AI tools, fixes reported problems, and makes amendments to meet new requirements

2.3. Operating Environment

2.3.1 The application is a desktop application running on Windows 11 (or higher) operating system.

3. Functional Requirements

3.1. User Stories: customer’s perspective

3.1.1. As a Customer, I want to specify an input YAML file (see the example alt-trans-anonym.yml), so that the application can read the input API description.

3.1.2. As a Customer, I want to get a meaningful error message in case the application malfunctions, so that I can understand the root cause. The examples of error messages:

[Error] Cannot read data in <filename>.

In the event of a technical error, all relevant information is copied into a log file.

3.1.3. As a Customer, I want to check that the input data meets the OpenAPI specification of the predefined version, so that further processing will be possible and correct.

If the validation **fails** on unresolved `$ref` /schema errors, the app stops processing and return a meaningful error message.

3.1.4. As a Customer, I want to have the tabular API description based on the provided YAML file (see the attached example: alt-trans-anonym.yml)as depicted below, so that I can publish it on a Confluence page (Confluence Cloud 1000.0.0-84c2e1173b9a).

The tabular description includes the following sections:

request description;

response description.

The request description contains the following tables

request parameters (a separate output file);

request body (a separate output file).

The **request parameter table** contains the following fields:

‘Name’ - the request parameter name.

‘Mandatory’ - if the parameter is mandatory.

‘Expected value(s)’ - contains either prescribed values or the type of the property.

‘In’ - where the parameter is located: ‘header’, ‘path’, ‘query’, ‘cookie’.

The **request body description** (a separate output file) contains the following fields:

‘Path’ contains the full reference to a given property, which includes the names of upper-level objects.

‘Property’ contains the name of a given property/object; if the property is an array, then it is denoted as ‘<name>[0]’.

‘Mandatory’ indicates that the property/object is mandatory; possible values: True/False.

‘Expected Value(s)’ contains either prescribed values or the type of the property (e.g. ‘object’, ‘integer’); include all found (min/max, pattern, etc.); show up to N values inline (N is configurable), else write “see Description”.

‘Description’ - an optional field containing the description of a given property provided in YAML file; the customer may request to include this field in the table by relevant setting in configuration (see 5.2.2).

‘Examples’ - an optional field containing the examples of a given property provided in YAML file; the customer may request to include this field in the table by relevant setting in the configuration.

The application mark non-JSON bodies as `binary` and doesn’t deep-flatten.

The snippet of the description is shown below.

Path	Property	Mandatory	Type	Description	Examples
/	pageNumber	False	integer	Page number to get paginated results → If not specified we	1

				will send default as 1	
/	departurePoints[0]	True	array	List of departure airports from which we need the alternate flights from ...	
departurePoints[0]	departurePoint	True	string	Departure airport code	KBP
offer	numberOfNights	True	number	Duration of the package	7

The **response body description** is similar to the request one, though there is a difference: status code responses are to be specified. Hence, an additional column ‘Status’ is assumed.

The whole spec is expected to be exported.

3.2. User Stories: developer’s perspective

3.2.1. As a Developer, I want to have a modular application, so that the changes in a given module do not cause a malfunction of the other modules.

3.2.2. As a Developer, I want to have a flexible application architecture, so that I can add new features if needed.

3.2.3. As a Developer, I want to have a technical log file, so that I can understand the reason(s) of abnormal operations. The log file is created every time the application runs if ‘create_log’ parameter is set to ‘True’ in the configuration file. The file is created in the same folder where the executable file is located.

5. Interface Requirements

5.1. User stories: Customer’s perspective

5.1.1. As a Customer, I want to specify one YAML file, so that the application can build the descriptions (3.1.4) and save the result as either a single Microsoft Excel file (xlsx, params sheet, request body sheet, response body sheet) or CSV files (params CSV file, request body CSV file and response body CSV file).

5.1.4 As a Customer, I want to have an indicator that the application is running, so that I can understand the current stage and status of the operation.

Example: “Parsing... 20% / Flattening... 60% / Writing... 90%”.

5.2 User stories: Developer’s perspective

5.2.1. As a Developer, I want to implement a command-line interface (CLI), so that the application can support input/output operations.

5.2.2. As a Developer, I want to keep the set of the application configuration parameters as a separate text file, so that I can view and check the settings if needed, without running the

application.

The configuration file is a text file containing grouping parameters ([input], [output], etc), variables with the relevant values, and comments. Here is an example.

[input]

#API specification and version

spec=openapi: 3.0.1

#input format

input_format=YAML

[output]

#Output format: xlsx, csv

format=xlsx

file_name = api_tab_desc

#indicate if the provided description is to be included in the output tables

include_provided_description = True

include_examples = True

#Create log file

create_log=False

5.2.3 As a Developer, I want to implement a CLI in a way, so that I can develop an advanced GUI as the next logical step.

List CLI arguments planned: <input_file1> [<output_file>].

<output_file> may not be presented. If missing, derive the output filename from the input filename

as <input_filename>_param, <input_filename>_req_body, <input_filename>_res_body.

The GUI provides a possibility to select input files.

6. **Security and Data Privacy**

There are no specific security and/or data privacy requirements.

7. Non-Functional Requirements

7.1. Scalability

7.1.1. It is expected that the maximum number of lines in an input YAML file does not exceed 100000.

7.2. Performance

7.2.1. The expected runtime under 1 minute for 100 000 YAML lines in each input file on standard hardware (e.g. Intel i5, 16GB RAM).

7.3. Portability

7.3.1. The application runs from a USB stick or redistributable package and does not require any additional installation on the user workstation.

7.4. Maintainability

7.4.1. The source code is documented for further maintenance and development by third parties.

7.5. Testability and Verification

The application shall be testable via manual tests. The following types of tests are expected:

7.5.1. Unit Tests:

- for core processing (e.g. nested array description, allowed values, mandatory properties)
- for configuration file parsing and validation routines.

7.5.2. Data Validation Tests:

- input data checking routines should be tested against malformed or inconsistent data.

7.5.3. CLI Tests (Smoke & Functional):

- verify correct execution with valid/invalid file paths, parameters, and missing arguments;
- ensure helpful error messages appear for user mistakes.

7.5.4. Output Tests:

- confirm structure and content of generated xlsx or csv outputs match the expected format;

7.5.5. Performance Benchmarking (optional):

- run time measurement on typical dataset sizes (e.g. 10k, 50k, 100k rows) to check for acceptable performance.

7.5.6. Regression Testing:

- any future changes to the processing logic or configuration parsing should include regression tests to maintain correctness.

8. Solution requirements

8.1. **Edge cases**

8.1.1. **Combinators** (`oneOf/anyOf/allOf`): default is **not** expanding branches; summarize alternatives in “Expected values”. Toggle in config to expand (more rows).

8.1.2. **Binary bodies / non-JSON media types**: include rows but mark `type=binary|octet-stream` and skip deep flattening; no media type recorded in a separate column is expected.

8.1.3. **Read/Write only**: include both; use the prefix to **Property** with tags, e.g., `[RO] totalPrice`, `[WO] password` optional flags `include_read_only` / `include_write_only`.

8.1.4. **Security schemes/headers**: not in scope for the tables of first release unless present as explicit parameters; can add a fourth “Auth & Headers” sheet later.

8.1.5. **Descriptions, formulas, references**: read vendor/provider extensions, e.g. `x-formula`, `x-reference`, `x-business-note`, and place them in **Description**.

8.1.6. If both `schema.example(s)` and `content.example(s)` exist, the schema.example(s) takes precedence.

8.1.7. **Large specs**: external `$ref` files/URLs expected in 3.0.x are left as references, i.e. as unresolved strings.

8.1.8. **Excel aesthetics**: there are no specific requirements for the first release. freeze top row, auto-filter, autosize columns are not required.

8.1.9. **Response scope**: 2xx + default.

8.1.10. **Constraints in Expected Value(s):** all constraints defined in the schema (e.g., min/max, pattern, length) must be included in the output. No extra derived or reformatted constraints are required for the first release. Manual editing is expected if further refinement is needed.

8.1.11. **CLI arguments**: `<input_file1> [<output_file>]`. no **wildcards** (batch mode) is suppored.

8.2. **Filtering**

8.2.1. The filter capability is supported through configuration as optional parameters. There are two parameters: path and method. Both contain strings that specify the required path and method.

Example:

[Filtering]

path = /search/package/v1/alternative/flights

method = post

This setting means that the output will cover only the specified path and method.

8.2.2. By default YAML file contains a single path and a single method, and the filter parameters are not required, though they can be specified.

8.2.3. If filter settings are not specified, and YAML file contains more than one path or method, the application stops and returns a relevant error.

8.2.4. If filter settings are specified, and YAML file does not contain either the required path or the method, the application stops and returns a relevant error.

8.3. **Error handling**

8.3.1. The application stops on the first error and returns a meaningful error message.

8.3.2. For large specs, partial output with warnings might be useful in further releases (but not in the first one).

8.4. Other requirements

8.4.1. Descriptions/examples are included if explicitly set in config.

9. Appendices

9.1. Assumptions and Dependencies

8.1.1. The input YAML files have unique names.

8.1.2. The latest version of Python and the relevant libraries are to be used.

8.1.3. The application does not assume batch operations (e.g., multiple file pairs).

8.1.4. The localization of the decimal format is not assumed.