Validations Engine Documentation

Release 1.0.0

QuintoAndar

Dec 01, 2022

CONTENTS

Made with by the **Data Engineering** team from QuintoAndar.

Engine for creating and running validation for general purposes

CHAPTER

ONE

PURPOSE

This automation engine was conceived to simulate key communication/integration lines between services that compose the core jobs of the data engineering pipeline at QuintoAndar, to guarantee there is no failures in the pipeline (and **early catch** eventual failures). But it can execute validations for general purposes.

CHAPTER

TWO

STRUCTURE AND THE ENGINE CORE CONCEPTS

This engine is composed by the Executors and the ValidationSuites.

Executors

Executors are responsible for executing all the respective validation suites that inherit them. A suite can inherit a custom executor or the `BaseValidationSuitesExecutor`. The executor may contain default and generic code and tests for a group of suites. For example, we may have one executor with features for validating a group of *Databases* and another for validating some *APIs*.

Validation Suites

A validation suite works like a Python unit test class. Once you define the validation methods (following the name pattern), these methods will be run by the engine.

Inside the validation suite class, you may implement into the validation methods the validations you want to perform. You can implement a custom validation inside your suite or if it is a common validation you may want to generalize it and use a custom executor like `DatabaseValidationSuitesExecutor` for sharing the same validation among other suites that inherits it. Every suite must inherit from an executor, in order to be parsed and run. So it should inherit from the `BaseValidationSuitesExecutor` or from a custom one (that inherits from the base executor).

The `validation_*` methods inside the suite and inside the executor classes will run automatically by the validation engine.

2.1 Navigation

2.1.1 Getting Started

Validations Engine depends on **Python 3.7.6+**.

Python Package Index hosts reference to a pip-installable module of this library, using it is as straightforward as including it on your project's requirements.

```
pip install validations-engine
```

Or after listing `validations-engine` in your `requirements.txt` file:

```
pip install -r requirements.txt
```

Adding a new validation suite

Check *here* the instructions to create a new validation suite.

Discovering Validations Engine

Click on the following links to open the examples:

#1 Example

2.1.2 About

Motivation

During the daily data engineering pipeline execution, may occur some problems related to:

- Network access (like VPC, peering, etc).
- API failures.
- Database connections failures (access, authentication, networking related errors).
- Failures in Python requirements installing (via init scripts) and its dependencies installing.
- DML errors.
- and the list goes on... .

Eventually, internal modifications in service's platform (infra, services, etc) can lead to bugs and failures in the ELT daily runs. Also, eventual external factors (like a lib dependency update) may also lead to such problems. And we definitely cannot expect that everything will run accordingly on the official daily executions.

It is pretty important therefore to have internal processes to guarantee (*as much as we can*) that the pipelines will run without failures.

2.1.3 API Specification

validations_engine package

Submodules

Slack communications module.

class validations_engine.SlackHelper.**SlackHelper**

Bases: object

Slack Helper class.

static build_slack_payload(*error_messages: List[Tuple[str, str]]*) → Dict[str, Dict[str, Any]]

Builds the message payload from the error messages.

static send_slack_errors(*error_messages: List[Tuple[str, str]]*) → bool

Sends errors messages to Slack (channels).

Returns

flag stating if messages were sent or not

Suites base executor.

class validations_engine.base_validation_suites_executor.**BaseValidationSuitesExecutor**(*auth: Optional[Dict[str, Any]] = None*)

Bases: object

Validation suites executors abstract class.

SLACK_MSG_HEADER = ''

get_suite_validation_failures_messages() → List[Tuple[str, str]]

Return the errors list.

get_suite_validation_has_failures() → bool

Returns the bool state indicating if there were failures.

run() → None

Main method executed by the validation suites (E.g.: FooValidationSuite).

It will run every method prefixed with *validation_* defined in the validation suite class. And the default ones defined in the executor class.

Only stops when all validations are finished.

Validations Engine main class.

class validations_engine.validations_engine.**ValidationsEngine**(*validations_suites_root_path: str*)

Bases: object

The Validator engine main file.

This class is responsible for parsing all validation suites and running its validation methods individually.

handle_errors() → None

Ensures (raises) a failure in the end of all validations.

load_validation_suites(*validations_suites_root_path: str*) → List[*BaseValidationSuitesExecutor*]

Import suites' modules and loads a list with their classes.

Get each suite file, imports it as a python module and loads the suite class.

run_validation_suites() → None

The main method.

Runs all validations from Suite classes and raises an error if some validation failed.

set_connections_auth_params(*connections_auth_params: Dict[str, Any]*) → None

Setter of connections_auth_params.

set_suites_have_failures(*param: bool*, *messages: Optional[List[str]] = None*) → None

Merges previous state with new one.

Module contents

Top-level package for validations_engine.

2.1.4 Adding a new validation suite

Creating a new validation context (executor)

If you want to create a validation for a new category (like the categories API or Database), you may first create an executor. The executor must inherit from `BaseValidationSuitesExecutor` and wrap common validation that validation suites may share. And this executor must be inherited by each validation suite you create in this context. E.g.:

```
from validations_engine.base_validation_suites_executor import␣
↪BaseValidationSuitesExecutor

class DatabasesValidationSuitesExecutor(BaseValidationSuitesExecutor):

    def __init__(self, auth):
        """
        :param auth: Authentication dictionary with DB auth needed for connecting in␣
↪your test.
         You should define it after calling instantiating the ValidationsEngine().
         After this, it is automatically filled in this executor by the Validation␣
↪Engine.
        """
        super().__init__()
        self.auth = auth

    def method_one(self):
        # some code you use in this executor
        pass


    def validation_foo(self):
        # some validation which all the suites of this executor will have
        pass
```

Creating a new validation suite

If you want to validate something of an existing context, like add validations for a new database type, you may only create a new validation suite file.

1. Add the new Validation Suite class file in your validation suites' root path.
2. Inherit it from some executor or from the base executor.
3. Create the validations inside each method prefixed with `validate_`.

E.g.:

```
# imports suppressed
class MyCoolValidationSuite(APIValidationSuitesExecutor):
    # Every suite must inherit a validation suites executor in order to be run.

    def validate_auth(self):
```

(continues on next page)

(continued from previous page)

```
        # checking if the API is authenticating and returning data as expected
        conn = self.api_client(self.auth['user'], self.auth['passwd'])
        res = conn.get_some_data()
        assert res.code == 200
```

Make sure you perform an **assert** at the end to be sure of your validation.

Felt something is missing? Open an issue and we shall make it clearer =]

PYTHON MODULE INDEX

V

INDEX