Spidermon Documentation

Zyte

Oct 04, 2021

Contents

Spidermon is a framework to build monitors for Scrapy spiders. It offers the following features:

- It can check the output data produced by Scrapy (or other sources) and verify it against a schema or model that defines the expected structure, data types and value restrictions. It supports data validation based on two external libraries:
  - jsonschema: https://github.com/Julian/jsonschema
  - Schematics: https://github.com/schematics/schematics
- It allows you to define conditions that should trigger an alert based on Scrapy stats.
- It supports notifications via email, Slack and Telegram.
- It can generate custom reports.

CHAPTER 1

Contents

1.1 Installation

Spidermon's core functionality provides some useful features that allow you to build your monitors on top of it. The library depends on six, jsonschema and python-slugify.

If you want to set up any notifications, additional *monitoring* dependencies will help with that.

If you want to use schematics validation, you probably want *validation*.

So the recommended way to install the library is by adding both:

```
pip install "spidermon[monitoring,validation]"
```

1.2 Getting started

Spidermon is a monitoring tool for Scrapy spiders that allows you to write monitors that may validate the execution and the results of your spiders.

This tutorial shows how to set up Spidermon to monitor a spider to check if it extracted a minimum number of items and if they follow a defined data model. The results of the spider execution will be sent to a Slack channel.

It is expected that you have a basic knowledge of Scrapy. If that is not the case read the Scrapy Tutorial and come back later. We will also assume that Spidermon is already installed on your system. If that is not the case case, see *Installation*.

1.2.1 Our sample project

You can find the complete code of our tutorial project here.

We are going to scrape quotes.toscrape.com, a website that lists quotes from famous authors. First we need a regular Scrapy project and create a simple spider:

```
$ scrapy startproject tutorial
$ cd tutorial
$ scrapy genspider quotes quotes.toscrape.com
```

And our spider code:

```
# tutorial/spiders/quotes.py
import scrapy

class QuotesSpider(scrapy.Spider):
    name = 'quotes'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['http://quotes.toscrape.com/']

    def parse(self, response):
        for quote in response.css('.quote'):
            yield {
                'quote': quote.css('.text::text').get(),
                'author': quote.css('.author::text').get(),
                'author_url': response.urljoin(
                    quote.css('.author a::attr(href)').get()),
                'tags': quote.css('.tag *::text').getall(),
            }

        yield scrapy.Request(
            response.urljoin(
                response.css('.next a::attr(href)').get()
            )
        )
```

1.2.2 Enabling Spidermon

To enable Spidermon in your project, include the following lines in your Scrapy project *settings.py* file:

```
SPIDERMON_ENABLED = True

EXTENSIONS = {
    'spidermon.contrib.scrapy.extensions.Spidermon': 500,
}
```

1.2.3 Our first monitor

Monitors are similar to test cases with a set of methods that are executed at well defined moments of the spider execution containing your monitoring logic.

Monitors must be grouped into monitor suites which define a list of monitors that need to be executed and the actions to be performed before and after the suite execute all monitors.

Our first monitor will check whether at least 10 items were returned at the end of the spider execution.

Create a new file called *monitors.py* that will contain the definition and configuration of your monitors.

```
# tutorial/monitors.py
from spidermon import Monitor, MonitorSuite, monitors

```

(continues on next page)

(continued from previous page)

```
@monitors.name('Item count')
class ItemCountMonitor(Monitor):

    @monitors.name('Minimum number of items')
    def test_minimum_number_of_items(self):
        item_extracted = getattr(
            self.data.stats, 'item_scraped_count', 0)
        minimum_threshold = 10

        msg = 'Extracted less than {} items'.format(
            minimum_threshold)
        self.assertTrue(
            item_extracted >= minimum_threshold, msg=msg
        )

class SpiderCloseMonitorSuite(MonitorSuite):

    monitors = [
        ItemCountMonitor,
    ]
```

This suite needs to be executed when the spider closes, so we include it in the SPIDERMON_SPIDER_CLOSE_MONITORS list in your *settings.py* file:

```
# tutorial/settings.py
SPIDERMON_SPIDER_CLOSE_MONITORS = (
    'tutorial.monitors.SpiderCloseMonitorSuite',
)
```

After executing the spider, you should see the following in your logs:

```
$ scrapy crawl quotes
(...)
INFO: [Spidermon] -------------------- MONITORS --------------------
INFO: [Spidermon] Item count/Minimum number of items... OK
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 1 monitor in 0.001s
INFO: [Spidermon] OK
INFO: [Spidermon] ---------------- FINISHED ACTIONS ----------------
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 0 actions in 0.000s
INFO: [Spidermon] OK
INFO: [Spidermon] ----------------- PASSED ACTIONS -----------------
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 0 actions in 0.000s
INFO: [Spidermon] OK
INFO: [Spidermon] ----------------- FAILED ACTIONS -----------------
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 0 actions in 0.000s
INFO: [Spidermon] OK
[scrapy.statscollectors] INFO: Dumping Scrapy stats:
(...)
```

If the condition in your monitor fails, this information will appear in the logs:

```
$ scrapy crawl quotes
```

(continues on next page)

(continued from previous page)

```
(...)
INFO: [Spidermon] -------------------- MONITORS --------------------
INFO: [Spidermon] Item count/Minimum number of items... FAIL
INFO: [Spidermon] --------------------------------------------------
ERROR: [Spidermon]
====================================================================
FAIL: Item count/Minimum number of items
--------------------------------------------------------------------
Traceback (most recent call last):
  File "/tutorial/monitors.py",
    line 17, in test_minimum_number_of_items
    item_extracted >= minimum_threshold, msg=msg
AssertionError: False is not true : Extracted less than 10 items
INFO: [Spidermon] 1 monitor in 0.001s
INFO: [Spidermon] FAILED (failures=1)
INFO: [Spidermon] ---------------- FINISHED ACTIONS ----------------
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 0 actions in 0.000s
INFO: [Spidermon] OK
INFO: [Spidermon] ----------------- PASSED ACTIONS -----------------
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 0 actions in 0.000s
INFO: [Spidermon] OK
INFO: [Spidermon] ----------------- FAILED ACTIONS -----------------
INFO: [Spidermon] --------------------------------------------------
INFO: [Spidermon] 0 actions in 0.000s
INFO: [Spidermon] OK
```

1.2.4 Notifications

Receiving fail notification in the logs may be helpful during the development but not so useful when you are running a huge number of spiders, so you can define actions to be performed when your spider start or finishes (with or without failures).

Spidermon has some built-in actions but you are free to define your own.

1.2.5 Slack notifications

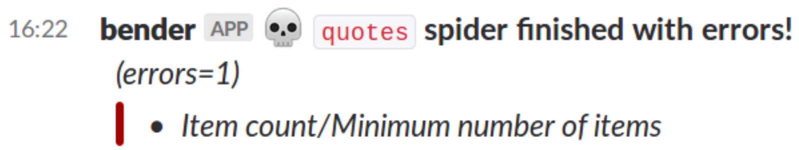
Here we will configure a built-in Spidermon action that sends a pre-defined message to a Slack channel using a bot when a monitor fails.

```python
# tutorial/monitors.py
from spidermon.contrib.actions.slack.notifiers import SendSlackMessageSpiderFinished

# (...your monitors code...)

class SpiderCloseMonitorSuite(MonitorSuite):
    monitors = [
        ItemCountMonitor,
    ]

    monitors_failed_actions = [
        SendSlackMessageSpiderFinished,
    ]
```

After enabling the action, you need to provide the Slack credentials. You can access the required credentials by following these steps to *How do I configure a Slack bot for Spidermon?*. Later, fill the credentials in your *settings.py* as follows:

```
# tutorial/settings.py
(...)
SPIDERMON_SLACK_SENDER_TOKEN = '<SLACK_SENDER_TOKEN>'
SPIDERMON_SLACK_SENDER_NAME = '<SLACK_SENDER_NAME>'
SPIDERMON_SLACK_RECIPIENTS = ['@yourself', '#yourprojectchannel']
```

If a monitor fails, the recipients provided will receive a message in Slack:



16:22 **bender** APP 💀 `quotes` **spider finished with errors!**
(errors=1)

- *Item count/Minimum number of items*

1.2.6 Telegram notifications

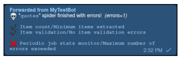
Here we will configure a built-in Spidermon action that sends a pre-defined message to a Telegram use, group or channel using a bot when a monitor fails.

```
# tutorial/monitors.py
from spidermon.contrib.actions.telegram.notifiers import SendTelegramMessageSpiderFinished

# (...your monitors code...)

class SpiderCloseMonitorSuite(MonitorSuite):
    monitors = [
        ItemCountMonitor,
    ]

    monitors_failed_actions = [
        SendTelegramMessageSpiderFinished,
    ]
```

After enabling the action, you need to provide the Telegram bot token and *Recipients*. You can learn more about how to create and configure a bot by following the steps on *How do I configure a Telegram bot for Spidermon?*. Later, fill the required information in your *settings.py* as follows:

```
# tutorial/settings.py
(...)
SPIDERMON_TELEGRAM_SENDER_TOKEN = '<TELEGRAM_SENDER_TOKEN>'
SPIDERMON_TELEGRAM_RECIPIENTS = ['chatid', 'groupid', '@channelname']
```

If a monitor fails, the recipients provided will receive a message in Telegram:



Forwarded from MyTestBot
💀 "quotes" spider finished with errors! *(errors=1)*

✓ Item count/Minimum items extracted
✓ Item validation/No item validation errors

✗ Periodic job stats monitor/Maximum number of errors exceeded

2:32 PM ✓

1.2.7 Item validation

Item validators allows you to match your returned items with predetermined structure ensuring that all fields contains data in the expected format. Spidermon allows you to choose between schematics or JSON Schema to define the structure of your item.

In this tutorial, we will use a schematics model to make sure that all required fields are populated and they are all of the correct format.

First step is to change our actual spider code to use Scrapy items. Create a new file called *items.py*:

```
# tutorial/items.py
import scrapy

class QuoteItem(scrapy.Item):
    quote = scrapy.Field()
    author = scrapy.Field()
    author_url = scrapy.Field()
    tags = scrapy.Field()
```

And then modify the spider code to use the newly defined item:

```
# tutorial/spiders/quotes.py
import scrapy
from tutorial.items import QuoteItem

class QuotesSpider(scrapy.Spider):
    name = 'quotes'
    allowed_domains = ['quotes.toscrape.com']
    start_urls = ['http://quotes.toscrape.com/']

    def parse(self, response):
        for quote in response.css('.quote'):
            item = QuoteItem(
                quote=quote.css('.text::text').get(),
                author=quote.css('.author::text').get(),
                author_url=response.urljoin(
                    quote.css('.author a::attr(href)').get()
                ),
                tags=quote.css('.tag *::text').getall()
            )
            yield item

        yield scrapy.Request(
            response.urljoin(
                response.css('.next a::attr(href)').get()
            )
        )
```

Now we need to create our schematics model in *validators.py* file that will contain all the validation rules:

```
# tutorial/validators.py
from schematics.models import Model
from schematics.types import URLType, StringType, ListType

class QuoteItem(Model):
    quote = StringType(required=True)
    author = StringType(required=True)
    author_url = URLType(required=True)
    tags = ListType(StringType)
```

To allow Spidermon to validate your items, you need to include an item pipeline and inform the name of the model class used for validation:

```
# tutorial/settings.py
ITEM_PIPELINES = {
    'spidermon.contrib.scrapy.pipelines.ItemValidationPipeline': 800,
}

SPIDERMON_VALIDATION_MODELS = (
    'tutorial.validators.QuoteItem',
)
```

After that, every time you run your spider you will have a new set of stats in your spider log providing information about the results of the validations:

```
$ scrapy crawl quotes
(...)
 'spidermon/validation/fields': 400,
 'spidermon/validation/items': 100,
 'spidermon/validation/validators': 1,
 'spidermon/validation/validators/item/schematics': True,
[scrapy.core.engine] INFO: Spider closed (finished)
```

You can then create a new monitor that will check these new statistics and raise a failure when we have a item validation error:

```
# monitors.py

from spidermon.contrib.monitors.mixins import StatsMonitorMixin

# (...other monitors...)

@monitors.name('Item validation')
class ItemValidationMonitor(Monitor, StatsMonitorMixin):

    @monitors.name('No item validation errors')
    def test_no_item_validation_errors(self):
        validation_errors = getattr(
            self.stats, 'spidermon/validation/fields/errors', 0
        )
        self.assertEqual(
            validation_errors,
            0,
            msg='Found validation errors in {} fields'.format(
                validation_errors)
        )

class SpiderCloseMonitorSuite(MonitorSuite):
    monitors = [
        ItemCountMonitor,
        ItemValidationMonitor,
    ]

    monitors_failed_actions = [
        SendSlackMessageSpiderFinished,
    ]
```

You could also set the pipeline to include the validation error as a field in the item (although it may not be necessary, since the validation errors are included in the crawling stats and your monitor can check them once the spiders finishes):

By default, it adds a field called _*validation* to the item when the item doesn't match the schema:

```
# tutorial/settings.py
SPIDERMON_VALIDATION_ADD_ERRORS_TO_ITEMS = True
```

The resulted item will look like this:

```
{
    '_validation': defaultdict(
        <class 'list'>, {'author_url': ['Invalid URL']}),
     'author': 'Mark Twain',
     'author_url': 'not_a_valid_url',
     'quote': 'Never tell the truth to people who are not worthy of it.',
     'tags': ['truth']
}
```

1.3 Monitoring your jobs

1.3.1 Monitors

Monitors are the main class where you include your monitoring logic. After defining them, you need to include them in a *MonitorSuite*, so they can be executed.

As *spidermon.core.monitors.Monitor* inherits from Python *unittest.TestCase*, you can use all existing assertion methods in your monitors.

In the following example, we define a monitor that will verify whether a minimum number of items were extracted and fails if it is less than the expected threshold.

```
from spidermon import Monitor, monitors

@monitors.name('Item count')
class ItemCountMonitor(Monitor):

    @monitors.name('Minimum items extracted')
    def test_minimum_number_of_items_extracted(self):
        minimum_threshold = 100
        item_extracted = getattr(self.data.stats, 'item_scraped_count', 0)
        self.assertFalse(
            item_extracted < minimum_threshold,
            msg='Extracted less than {} items'.format(minimum_threshold)
        )
```

A `Monitor` instance defines a monitor that includes your monitoring logic and has the following properties that can be used to help you implement your monitors:

`data.stats` dict-like object containing the stats of the spider execution

`data.crawler` instance of actual Crawler object

`data.spider` instance of actual Spider object

class `spidermon.core.monitors.`**`Monitor`**(*methodName='runTest'*, *name=None*)

1.3.2 Monitor Suites

A *Monitor Suite* groups a set of *Monitor* classes and allows you to specify which actions must be executed at specified moments of the spider execution.

Here is an example of how to configure a new monitor suite in your project:

```
# monitors.py
from spidermon.core.suites import MonitorSuite

# Monitor definition above...
class SpiderCloseMonitorSuite(MonitorSuite):
    monitors = [
        # (your monitors)
    ]

    monitors_finished_actions = [
        # actions to execute when suite finishes its execution
    ]

    monitors_failed_actions = [
        # actions to execute when suite finishes its execution with a failed monitor
    ]
```

```
# settings.py
SPIDERMON_SPIDER_OPEN_MONITORS = (
    # list of monitor suites to be executed when the spider starts
)

SPIDERMON_SPIDER_CLOSE_MONITORS = (
    # list of monitor suites to be executed when the spider finishes
)
```

class MonitorSuite(*name=None*, *monitors=None*, *monitors_finished_actions=None*, *monitors_passed_actions=None*, *monitors_failed_actions=None*, *order=None*, *crawler=None*)

An instance of `MonitorSuite` defines a set of monitors and actions to be executed after the job finishes its execution.

`name` suite name

`monitors` list of `Monitor` that will be executed if this suite is enabled.

`monitors_finished_actions` list of action classes that will be executed when all monitors finished their execution.

`monitors_passed_actions` list of action classes that will be executed if all monitors passed.

`monitors_failed_actions` list of action classes that will be executed if at least one of the monitors failed.

`order` if you have more than one suite enabled in your project, this integer defines the order of execution of the suites

`crawler` crawler instance

on_monitors_finished(*result*)

Executed right after the monitors finished their execution and before any other action is executed.

`result` stats of the spider execution

on_monitors_passed(*result*)

Executed right after the monitors finished their execution but after the actions defined in *monitors_finished_actions* were executed if all monitors passed.

`result` stats of the spider execution

on_monitors_failed(*result*)

Executed right after the monitors finished their execution but after the actions defined in *monitors_finished_actions* were executed if at least one monitor failed.

`result` stats of the spider execution

1.3.3 The Basic Monitors

Spidermon has some batteries included :)

class `spidermon.contrib.scrapy.monitors.`**ErrorCountMonitor**(*methodName='runTest'*, *name=None*)

Check for errors in the spider log.

You can configure the expected number of ERROR log messages using `SPIDERMON_MAX_ERRORS`. The default is `0`.

class `spidermon.contrib.scrapy.monitors.`**FieldCoverageMonitor**(*methodName='runTest'*, *name=None*)

Validate if field coverage rules are met.

To use this monitor you need to enable the `SPIDERMON_ADD_FIELD_COVERAGE` setting, which will add information about field coverage to your spider statistics.

To define your field coverage rules create a dictionary containing the expected coverage for each field you want to monitor.

As an example, if the items you are returning from your spider are Python dictionaries with the following format:

```
{
    "field_1": "some_value",
    "field_2": "some_value",
    "field_3": {
        "field_3_1": "some_value",
        "field_3_2": "some_value",
    }
}
```

A set of rules may be defined as follows:

```
# project/settings.py
SPIDERMON_FIELD_COVERAGE_RULES = {
    "dict/field_1": 0.4,  # Expected 40% coverage for field_1
    "dict/field_2": 1.0,  # Expected 100% coverage for field_2
    "dict/field_3": 0.8,  # Expected 80% coverage for parent field_3
    "dict/field_3/field_3_1": 0.5,  # Expected 50% coverage for nested field_3_1
}
```

You are not obligated to set rules for every field, just for the ones in which you are interested. Also, you can monitor nested fields if available in your returned items.

Warning: Rules for nested fields will be validated against the total number of items returned.

For the example below, rule for `dict/field_3/field_3_1` will validate if 50% of **all** items returned contains `field_3_1`, not just the ones that contain parent `field_3`.

Note: If you are returning an item type other than a dictionary, replace *dict* by the class name of the item you are returning.

Considering you have an item defined as:

```
class MyCustomItem(scrapy.Item):
    field_1 = scrapy.Field()
    field_2 = scrapy.Field()
```

You must define the field coverage rules as follows:

```
SPIDERMON_FIELD_COVERAGE_RULES = {
    "MyCustomItem/field_1": 0.4,
    "MyCustomItem/field_2": 1.0,
}
```

class `spidermon.contrib.scrapy.monitors.`**`FinishReasonMonitor`**(*methodName='runTest'*, *name=None*)

Check if a job has a expected finish reason.

You can configure the expected reason with the `SPIDERMON_EXPECTED_FINISH_REASONS`, it should be an `iterable` of valid finish reasons.

The default value of this settings is: `['finished', ]`.

class `spidermon.contrib.scrapy.monitors.`**`ItemCountMonitor`**(*methodName='runTest'*, *name=None*)

Check if spider extracted the minimum number of items.

You can configure it using `SPIDERMON_MIN_ITEMS` setting. There's **NO** default value for this setting, if you try to use this monitor without setting it, it'll raise a `NotConfigured` exception.

class `spidermon.contrib.scrapy.monitors.`**`UnwantedHTTPCodesMonitor`**(*methodName='runTest'*, *name=None*)

Check for maximum number of unwanted HTTP codes. You can configure it using `SPIDERMON_UNWANTED_HTTP_CODES_MAX_COUNT` setting or `SPIDERMON_UNWANTED_HTTP_CODES` setting

This monitor fails if during the spider execution, we receive more than the number of `SPIDERMON_UNWANTED_HTTP_CODES_MAX_COUNT` setting for at least one of the HTTP Status Codes in the list defined in `SPIDERMON_UNWANTED_HTTP_CODES` setting.

Default values are:

```
SPIDERMON_UNWANTED_HTTP_CODES_MAX_COUNT = 10
SPIDERMON_UNWANTED_HTTP_CODES = [400, 407, 429, 500, 502, 503, 504, 523, 540, 541]
```

`SPIDERMON_UNWANTED_HTTP_CODES` can also be a dictionary with the HTTP Status Code as key and the maximum number of accepted responses with that code.

With the following setting, the monitor will fail if more than 100 responses are 404 errors or at least one 500 error:

```
SPIDERMON_UNWANTED_HTTP_CODES = {
    400: 100,
    500: 0,
}
```

1.3.4 Is there a Basic Scrapy Suite ready to use?

Of course, there is! We really want to make it easy for you to monitor your spiders ;)

class `spidermon.contrib.scrapy.monitors.`**`SpiderCloseMonitorSuite`**(*name=None, monitors=None, monitors_finished_actions=None, monitors_passed_actions=None, monitors_failed_actions=None, order=None, crawler=None*)

This Monitor Suite implements the following monitors:

- *`ItemCountMonitor`*
- `ItemValidationMonitor`
- *`ErrorCountMonitor`*
- *`FinishReasonMonitor`*
- *`UnwantedHTTPCodesMonitor`*
- *`FieldCoverageMonitor`*

You can easily enable this monitor *after* enabling Spidermon:

```
SPIDERMON_SPIDER_CLOSE_MONITORS = (
    'spidermon.contrib.scrapy.monitors.SpiderCloseMonitorSuite',
)
```

1.3.5 Periodic Monitors

Sometimes we don't want to wait until the end of the spider execution to monitor it. For example, we may want to be notified as soon the number of errors reaches a value or close the spider if the time elapsed is greater than expected.

You define your *Monitors* and *Monitor Suites* the same way as before, but you need to provide the time interval (in seconds) between each of the times the *Monitor Suites* is run.

In the following example, we defined a periodic monitor suite that will be executed every minute and will verify if the number of errors found is lesser than a value. If not, the spider will be closed.

First we define a new action that will close the spider when executed:

```
# tutorial/actions.py
from spidermon.core.actions import Action

class CloseSpiderAction(Action):

    def run_action(self):
        spider = self.data['spider']
        spider.logger.info("Closing spider")
        spider.crawler.engine.close_spider(spider, 'closed_by_spidermon')
```

Then we create our monitor and monitor suite that verifies the number of errors and then take an action if it fails:

```
# tutorial/monitors.py
from tutorial.actions import CloseSpiderAction

@monitors.name('Periodic job stats monitor')
class PeriodicJobStatsMonitor(Monitor, StatsMonitorMixin):

    @monitors.name('Maximum number of errors reached')
    def test_number_of_errors(self):
        accepted_num_errors = 20
        num_errors = self.data.stats.get('log_count/ERROR', 0)

        msg = 'The job has exceeded the maximum number of errors'
        self.assertLessEqual(num_errors, accepted_num_errors, msg=msg)

class PeriodicMonitorSuite(MonitorSuite):
    monitors = [PeriodicJobStatsMonitor]
    monitors_failed_actions = [CloseSpiderAction]
```

The last step is to configure the suite to be executed every 60 seconds:

```
# tutorial/settings.py

SPIDERMON_PERIODIC_MONITORS = {
    'tutorial.monitors.PeriodicMonitorSuite': 60,  # time in seconds
}
```

1.3.6 What to monitor?

These are some of the usual metrics used in the monitors:

- the amount of items extracted by the spider.
- the amount of successful responses received by the spider.
- the amount of failed responses (server-side errors, network errors, proxy errors, etc.).
- the amount of requests that reach the maximum amount of retries and are finally discarded.
- the amount of time it takes to finish the crawl.
- the amount of errors in the log (spider errors, generic errors detected by Scrapy, etc.)
- the amount of bans.
- the job outcome (if it finishes without major issues or if it is closed prematurely because it detects too many bans, for example).
- the amount of items that don't contain a specific field or a set of fields
- the amount of items with validation errors (missing required fields, incorrect format, values that don't match a specific regular expression, strings that are too long/short, numeric values that are too high/low, etc.)

1.4 Item Validation

One useful feature when monitoring a spider is being able to validate your returned items against a defined schema.

Spidermon provides a mechanism that allows you to define an item schema and validation rules that will be executed for each item returned. To enable the item validation feature, the first step is to enable the built-in item pipeline in your project settings:

```
# tutorial/settings.py
ITEM_PIPELINES = {
    'spidermon.contrib.scrapy.pipelines.ItemValidationPipeline': 800,
}
```

After that, you need to choose which validation library will be used. Spidermon accepts schemas defined using schematics or JSON Schema.

1.4.1 With schematics

Schematics is a validation library based on ORM-like models. These models include some common data types and validators, but they can also be extended to define custom validation rules.

Warning: You need to install schematics to use this feature.

```
# Usually placed in validators.py file
from schematics.models import Model
from schematics.types import URLType, StringType, ListType

class QuoteItem(Model):
    quote = StringType(required=True)
    author = StringType(required=True)
    author_url = URLType(required=True)
    tags = ListType(StringType)
```

Check schematics documentation to learn how to define a model and how to extend the built-in data types.

1.4.2 With JSON Schema

JSON Schema is a powerful tool for validating the structure of JSON data. You can define which fields are required, the type assigned to each field, a regular expression to validate the content and much more.

Warning: You need to install jsonschema to use this feature.

This guide explains the main keywords and how to generate a schema. Here we have an example of a schema for the quotes item from the *tutorial*.

```
{
  "$schema": "http://json-schema.org/draft-07/schema",
  "type": "object",
  "properties": {
    "quote": {
      "type": "string"
    },
    "author": {
      "type": "string"
    },
```

(continues on next page)

(continued from previous page)

```
    "author_url": {
      "type": "string",
      "pattern": ""
    },
    "tags": {
      "type"
    }
  },
  "required": [
    "quote",
    "author",
    "author_url"
  ]
}
```

1.4.3 Settings

These are the settings used for configuring item validation:

SPIDERMON_VALIDATION_ADD_ERRORS_TO_ITEMS

Default: `False`

When set to `True`, this adds a field called *_validation* to the item that contains any validation errors. You can change the name of the field by assigning a name to *SPIDERMON_VALIDATION_ERRORS_FIELD*:

```
{
    '_validation': defaultdict(<class 'list'>, {'author_url': ['Invalid URL']}),
    'author': 'C.S. Lewis',
    'author_url': 'invalid_url',
    'quote': 'Some day you will be old enough to start reading fairy tales '
        'again.',
    'tags': ['age', 'fairytales', 'growing-up']
}
```

SPIDERMON_VALIDATION_DROP_ITEMS_WITH_ERRORS

Default: `False`

Whether to drop items that contain validation errors.

SPIDERMON_VALIDATION_ERRORS_FIELD

Default: `_validation`

The name of the field added to the item when a validation error happens and *SPIDERMON_VALIDATION_ADD_ERRORS_TO_ITEMS* is enabled.

SPIDERMON_VALIDATION_MODELS

Default: `None`

A *list* containing the schematics models that contain the definition of the items that need to be validated.

```
# settings.py

SPIDERMON_VALIDATION_MODELS = [
    'tutorial.validators.DummyItemModel'
]
```

If you are working on a spider that produces multiple items types, you can define it as a *dict*:

```
# settings.py

from tutorial.items import DummyItem, OtherItem

SPIDERMON_VALIDATION_MODELS = {
    DummyItem: 'tutorial.validators.DummyItemModel',
    OtherItem: 'tutorial.validators.OtherItemModel',
}
```

SPIDERMON_VALIDATION_SCHEMAS

Default: `None`

A *list* containing the location of the item schema. Could be a local path or a URL.

```
# settings.py

SPIDERMON_VALIDATION_SCHEMAS = [
    '/path/to/schema.json',
    's3://bucket/schema.json',
    'https://example.com/schema.json',
]
```

If you are working on a spider that produces multiple items types, you can define it as a *dict*:

```
# settings.py

from tutorial.items import DummyItem, OtherItem

SPIDERMON_VALIDATION_SCHEMAS = {
    DummyItem: '/path/to/dummyitem_schema.json',
    OtherItem: '/path/to/otheritem_schema.json',
}
```

1.4.4 Validation in Monitors

You can build a monitor that checks the validation problems and raises errors if there are too many. You can base it on `spidermon.contrib.monitors.mixins.ValidationMonitorMixin` which provides methods that can be useful for this. There are 2 groups of methods, for checking all validation errors and specifically for checking `missing_required_field` errors. All of these methods rely on the job stats, reading `spidermon/validation/fields/errors/*` entries.

- `check_missing_required_fields`, `check_missing_required_field` - check that number of `missing_required_field` errors is less than the specified threshold.

- `check_missing_required_fields_percent`, `check_missing_required_field_percent` - check that percent of `missing_required_field` errors is less than the specified threshold.
- `check_fields_errors`, `check_field_errors` - check that the number of specified (or all) errors is less than the specified threshold.
- `check_fields_errors_percent`, `check_field_errors_percent` - check that the percent of specified (or all) errors is less than the specified threshold.

All `*_field` method take a name of one field, while all `*_fields` method take a list of field names.

Warning: The default behavior for `*_fields` methods when no field names is passed is to combine error counts for all fields instead of checking each field separately. This is usually not very useful and inconsistent with the behavior when a list of fields is passed, so you should set the `correct_field_list_handling` monitor attribute to get the correct behavior. This will be the default in some later version.

Note: The `*_percent` methods receive the ratio, not the percent number, so for 15% you need to pass 0.15.

Some examples:

```
# checks that each of field2 and field3 is missing in no more than 10 items
self.check_missing_required_fields(field_names=['field2', 'field3'], allowed_count=10)

# checks that field2 has errors in no more than 15% of items
self.check_field_errors_percent(field_name='field2', allowed_percent=0.15)

# checks that no errors is present in any fields
self.check_field_errors_percent()
```

1.5 Expression Monitors

Expressions Monitors are *monitors* created on-the-fly when Spidermon extension initializes. They can create tests based on simple expressions defined in a dictionary in your settings like:

```
SPIDERMON_SPIDER_OPEN_EXPRESSION_MONITORS = [
    {
        "name": "DumbChecksMonitor",
        "tests": [
            {
                "name": "test_spider_name",
                "expression": "spider.name == 'httpbin'",
            },
        ],
    },
]
```

The definition of each monitor should follow the *Expression monitor schema*.

Use the following settings to configure expression monitors:

- *SPIDERMON_SPIDER_OPEN_EXPRESSION_MONITORS*
- *SPIDERMON_SPIDER_CLOSE_EXPRESSION_MONITORS*

- *SPIDERMON_ENGINE_STOP_EXPRESSION_MONITORS*

You have the following objects available to be used in your *expression*:

- stats
- crawler
- spider
- job
- validation
- responses

Note: Notice that not the full-set of the Python Language features are available to the expressions, only the ones that makes sense for a simple expressions that should evaluate to `True` or `False`.

To have a more deep understand about which features of the language are available please refer to `spidermon.python.interpreter.Interpreter`.

1.5.1 How to create an expression monitor

First you need to choose *when* you want to run your expression monitors.

You can use *SPIDERMON_SPIDER_OPEN_EXPRESSION_MONITORS* to run a monitor when a spider opens, or *SPIDERMON_SPIDER_CLOSE_EXPRESSION_MONITORS* if you want to run a monitor when a spider is closed.

There's also the *SPIDERMON_ENGINE_STOP_EXPRESSION_MONITORS* setting to run a monitor once the engine has stopped.

Here's an example of how to declare two `ExpressionMonitors`.

The first monitor with two tests, one for checking the spider name and other test to check if the crawler is there. The 2nd monitor will check if the spider finished with `finished`:

```
[
    {
        "name": "DumbChecksMonitor",
        "description": "My expression monitor",
        "tests": [
            {
                "name": "test_spider_name",
                "description": "Test spider name",
                "expression": "spider.name == 'httpbin'",
            },
            {
                "name": "test_crawler_exists",
                "description": "Test Crawler exists",
                "expression": "crawler is not None"
            }
        ],
    },
    {
        "name": "FinishedOkMonitor",
        "description": "My expression monitor 2",
        "tests": [
            {
```

(continues on next page)

(continued from previous page)

```
                "name": "test_finish_reason",
                "description": "Test finish reason",
                "expression": 'stats["finish_reason"] == "finished"',
            }
        ],
    }
]
```

1.5.2 Expression monitor schema

Each *expression monitor* should follow this schema:

```
{
   "type": "object",
   "properties": {
       "name": {"type": "string", "minLength": 1},
       "description": {"type": "string", "minLength": 1},
       "tests": {
           "type": "array",
           "items": {
               "type": "object",
               "properties": {
                   "name": {"type": "string", "minLength": 1},
                   "description": {"type": "string", "minLength": 1},
                   "expression": {"type": "string", "minLength": 1},
                   "fail_reason": {"type": "string", "minLength": 1},
               },
               "required": ["name", "expression"],
           },
       },
   },
   "required": ["name", "tests"],
 }
```

1.6 Settings

The Spidermon settings allow you to customize the behaviour of your monitors enabling, disabling and configuring features like enabled monitors, monitor actions, item validation and notifications.

1.6.1 Built-in settings reference

Here's a list of all available Spidermons settings, in alphabetical order, along with their default values and the scope where they apply. These settings must be defined in *settings.py* file of your Scrapy project.

SPIDERMON_ENABLED

Default: `False`

Whether to enable Spidermon.

SPIDERMON_EXPRESSIONS_MONITOR_CLASS

Default: `spidermon.python.monitors.ExpressionMonitor`

A subclass of `spidermon.python.monitors.ExpressionMonitor`.

This class will be used to generate *expression monitors*.

- *SPIDERMON_SPIDER_CLOSE_EXPRESSION_MONITORS*
- *SPIDERMON_SPIDER_OPEN_EXPRESSION_MONITORS*
- *SPIDERMON_ENGINE_STOP_EXPRESSION_MONITORS*

Note: You probably will not change this setting unless you have an advanced use case and needs to change how the context data is build or how the on-the-fly `MonitorSuite` are generated. Otherwise the default should be enough.

SPIDERMON_PERIODIC_MONITORS

Default: `{}`

A dict containing the monitor suites that must be executed periodically as key and the time interval (in seconds) between the executions as value.

For example, the following suite will be executed every 30 minutes:

```
SPIDERMON_PERIODIC_MONITORS = {
    'tutorial.monitors.PeriodicMonitorSuite': 1800,
}
```

SPIDERMON_SPIDER_CLOSE_MONITORS

Default: `[]`

List of monitor suites to be executed when the spider closes.

SPIDERMON_SPIDER_CLOSE_EXPRESSION_MONITORS

Default: `[]`

List of dictionaries describing *expression monitors* to run when a spider is closed.

SPIDERMON_SPIDER_OPEN_MONITORS

Default: `[]`

List of monitor suites to be executed when the spider starts.

SPIDERMON_SPIDER_OPEN_EXPRESSION_MONITORS

Default: `[]`

List of dictionaries describing *expression monitors* to run when a spider is opened.

SPIDERMON_ENGINE_STOP_MONITORS

List of monitor suites to be executed when the crawler engine is stopped.

SPIDERMON_ENGINE_STOP_EXPRESSION_MONITORS

Default: `[]`

List of dictionaries describing *expression monitors* to run when the engine is stopped.

SPIDERMON_ADD_FIELD_COVERAGE

Default: `False`

When enabled, Spidermon will add statistics about the number of items scraped and coverage for each existing field following this format:

```
'spidermon_item_scraped_count/<item_type>/<field_name>':  <item_count>
'spidermon_field_coverage/<item_type>/<field_name>':  <coverage>
```

Note: Nested fields are also supported. For example, if your spider returns these items:

```
[
  {
    "field_1": {
      "nested_field_1_1": "value",
      "nested_field_1_2": "value",
    },
  },
  {
    "field_1": {
      "nested_field_1_1": "value",
    },
    "field_2": "value"
  },
]
```

Statistics will be like the following:

```
'spidermon_item_scraped_count/dict': 2,
'spidermon_item_scraped_count/dict/field_1': 2,
'spidermon_item_scraped_count/dict/field_1/nested_field_1_1': 2,
'spidermon_item_scraped_count/dict/field_1/nested_field_1_2': 1,
'spidermon_item_scraped_count/dict/field_2': 1,
'spidermon_field_coverage/dict/field_1': 1,
'spidermon_field_coverage/dict/field_1/nested_field_1_1': 1,
'spidermon_field_coverage/dict/field_1/nested_field_1_2': 0.5,
'spidermon_item_scraped_count/dict/field_2': 0.5,
```

SPIDERMON_FIELD_COVERAGE_SKIP_NONE

Default: `False`

When enabled, returned fields that have `None` as value will not be counted as fields with a value.

Considering your spider returns the following items:

```
[
  {
    "field_1": None,
    "field_2": "value",
  },
  {
    "field_1": "value",
    "field_2": "value",
  },
]
```

If this setting is set to `True`, spider statistics will be:

```
'spidermon_item_scraped_count/dict': 2,
'spidermon_item_scraped_count/dict/field_1': 1,  # Ignored None value
'spidermon_item_scraped_count/dict/field_2': 2,
'spidermon_field_coverage/dict/field_1': 0.5,  # Ignored None value
'spidermon_item_scraped_count/dict/field_2': 1,
```

If this setting is not provided or set to `False`, spider statistics will be:

```
'spidermon_item_scraped_count/dict': 2,
'spidermon_item_scraped_count/dict/field_1': 2,  # Did not ignore None value
'spidermon_item_scraped_count/dict/field_2': 2,
'spidermon_field_coverage/dict/field_1': 1,  # Did not ignore None value
'spidermon_item_scraped_count/dict/field_2': 1,
```

1.7 “How-to” guides

1.7.1 How do I add required fields coverage validation?

When you enable *item validation* in your project you can use *ValidationMonitorMixin* in your monitor, which allows you to perform some extra checks on your results.

Considering that we have the *validation schema* from the *getting started* section of our documentation, where the **author** field is required, we want to add a new monitor to ensure that no more than 20% of the items returned have the **author** not filled.

Note: The methods that will be presented next only work to check coverage of fields that are defined as **required** in your validation schema.

ValidationMonitorMixin gives you the *check_missing_required_fields_percent* method, which receives a list of field names and the maximum percentage allowed not to be filled. Using that we can create a monitor that enforces our validation rule:

```
from spidermon import Monitor
from spidermon.contrib.monitors.mixins import ValidationMonitorMixin

class CoverageValidationMonitor(Monitor, ValidationMonitorMixin):

    def test_required_fields_with_minimum_coverage(self):
```

(continues on next page)

(continued from previous page)

```
        allowed_missing_percentage = 0.2
        self.check_missing_required_fields_percent(
           field_names=["author"],
           allowed_percent=allowed_missing_percentage
        )
```

We also have the option to set an absolute amount of items that we want to allow not to be filled. This requires us to use the *check_missing_required_fields* method. The following monitor will fail if more than 10 items returned do not have the **author** field filled.

```
class CoverageValidationMonitor(Monitor, ValidationMonitorMixin):

    def test_required_fields_with_minimum_coverage(self):
        allowed_missing_items = 10
        self.check_missing_required_fields(
           field_names=["author"],
           allowed_count=allowed_missing_items
        )
```

Multiple fields

What if we want to validate more than one field? There are two different ways, depending on whether you want to use the same thresholds for both fields or a different one for each field.

Using the same threshold, we just need to pass a list with the field names to the desired validation method as follows:

```
class CoverageValidationMonitor(Monitor, ValidationMonitorMixin):

    def test_required_fields_with_minimum_coverage(self):
       allowed_missing_percentage = 0.2
       self.check_missing_required_fields_percent(
           field_names=["author", "author_url"],
           allowed_percent=allowed_missing_percentage
       )
```

However, if you want a different rule for different fields, you need to create a new monitor for each field:

```
class CoverageValidationMonitor(Monitor, ValidationMonitorMixin):

    def test_min_coverage_author_field(self):
        allowed_missing_percentage = 0.2
        self.check_missing_required_fields_percent(
            field_names=["author"],
            allowed_percent=allowed_missing_percentage
        )

    def test_min_coverage_author_url_field(self):
        allowed_missing_items = 10
        self.check_missing_required_fields(
            field_names=["author_url"],
            allowed_count=allowed_missing_items
        )
```

1.7.2 How do I configure a Slack bot for Spidermon?

What are bots?

A bot is a type of Slack App designed to interact with users via conversation.

To work with Slack Actions, you will need a Slack bot which would send notifications to your Slack workspace from Spidermon.

Steps

1. Create a Slack bot.
2. Once your bot is created, you can find Bot User OAuth Access Token in its settings. This *Bot User OAuth Access Token* is what we use for `SPIDERMON_SLACK_SENDER_TOKEN`.
3. Lastly, add your Slack credentials to your Scrapy project's settings.

```
# settings.py
SPIDERMON_SLACK_SENDER_TOKEN = 'YOUR_BOT_USER_OAUTH_ACCESS_TOKEN'
SPIDERMON_SLACK_SENDER_NAME = 'YOUR_BOT_USERNAME'
SPIDERMON_SLACK_RECIPIENTS = ['@yourself', '#yourprojectchannel']
```

1.7.3 How do I configure a Telegram bot for Spidermon?

What are bots?

A bot is a type of Telegram user designed to interact with users via conversation.

To work with Telegram Actions, you will need a Telegram bot which would send notifications to Telegram from Spidermon.

Steps

1. Create a Telegram bot.
2. Once your bot is created, you will receive its Authorization Token. This *Bot Authorization Token* is what we use for `SPIDERMON_TELEGRAM_SENDER_TOKEN`.
3. Add your Telegram bot credentials to your Scrapy project's settings.
4. Add the recipients to your Scrapy project's settings, getting the *chat_id* or *group_id* is not straightforward, there is an info bot that can help with this. Just forward a message from the user or group that you want to receive Spidermon notifications to *[@GroupIDbot](https://t.me/GroupIDbot)* and it will reply with the id. To forward a message from a group to the bot, you need to add the bot to that group before.
5. For notifications in groups: Add your self-created bot (from step 1.) to your group. For notifications in channels: Add your self-created bot (from step 1.) as administrator to your channel (not possible in browser version).

```
# settings.py
SPIDERMON_TELEGRAM_SENDER_TOKEN = 'YOUR_BOT_AUTHORIZATION_TOKEN'
SPIDERMON_TELEGRAM_RECIPIENTS = ['chat_id', 'group_id', '@channelname']
```

1.7.4 How do I compare Spider executions?

Sometimes it is worthy to compare results from previous executions of your spider. For example, we should be able to track whether the number of items returned is lower than previous executions, which may indicate a problem.

Scrapy allows you to collect stats in the form key/value, but these stats are lost when the spider finishes its execution. Spidermon provides a built-in stats collector that stores the stats of all spider executions in your local file system.

After enabling this stats collector, every spider instance running will have a `stats_history` attribute containing a list of the stats of the previous spider executions that can be easily accessed in your monitors.

To enable it, include the following code in your project settings:

```
# tutorial/settings.py
STATS_CLASS = (
    "spidermon.contrib.stats.statscollectors.LocalStorageStatsHistoryCollector"
)

# Stores the stats of the last 10 spider execution (default=100)
SPIDERMON_MAX_STORED_STATS = 10
```

The following example shows how we can verify whether the number of items returned in the current spider execution is higher than 90% of the mean of items returned in the previous spider executions.

```
# tutorial/monitors.py
from spidermon import Monitor, MonitorSuite, monitors


@monitors.name("History Validation")
class HistoryMonitor(Monitor):

  @monitors.name("Expected number of items extracted")
  def test_expected_number_of_items_extracted(self):
      spider = self.data["spider"]
      total_previous_jobs = len(spider.stats_history)
      if total_previous_jobs == 0:
          return

      previous_item_extracted_mean = (
          sum(
              previous_job["item_scraped_count"]
              for previous_job in spider.stats_history
          )
          / total_previous_jobs
      )
      items_extracted = self.data.stats["item_scraped_count"]

      # Minimum number of items we expect to be extracted
      minimum_threshold = 0.9 * previous_item_extracted_mean

      self.assertFalse(
          items_extracted <= minimum_threshold,
          msg="Expected at least {} items extracted.".format(
              minimum_threshold
          ),
      )

class SpiderCloseMonitorSuite(MonitorSuite):
    monitors = [HistoryMonitor]
```

Warning: Running your spider in Scrapy Cloud requires you to manually change some settings in your project:

1. Enable DotScrapy Persistence Add-on in your project to keep your `.scrapy` directory available between job executions.
2. STATS_CLASS is overriden by default in Scrapy Cloud. You need to manually include `spidermon.contrib.stats.statscollectors.LocalStorageStatsHistoryCollector` in your spider settings. The drawback is that you job stats will not be uploaded to Scrapy Cloud interface and will be available only in the job logs.

1.8 Actions

By default, when a monitor suite finishes, the pass/fail information is included in the spider logs, which would be enough during development but useless when you are monitoring several spiders.

Spidermon allows you to define actions that are ran after the monitors finish. You can define your own actions or use one of the existing built-in actions.

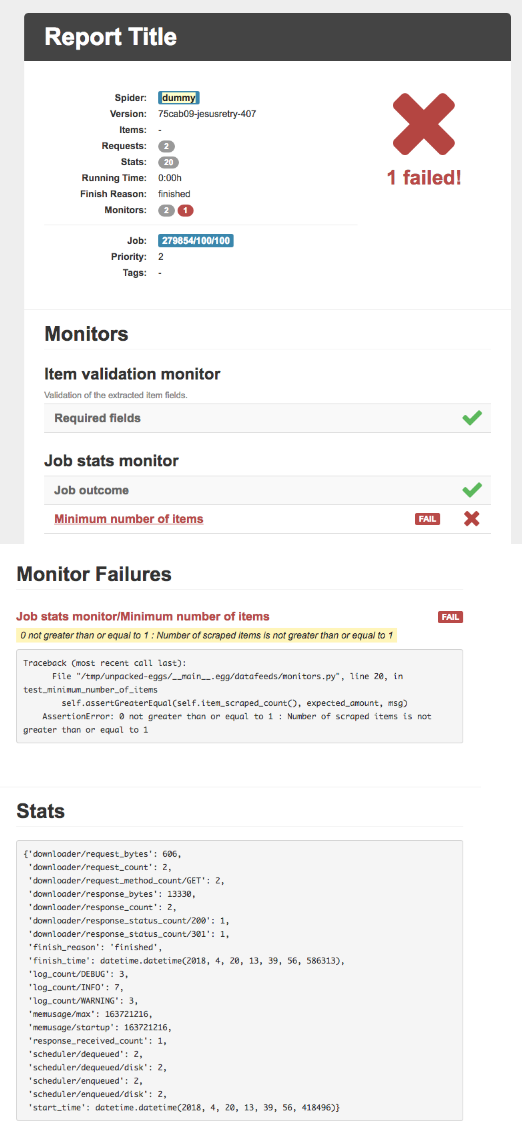
1.8.1 E-mail action

This action relies on Amazon Simple Email Service to send an e-mail after the monitor suite finishes its execution. In this example, an e-mail will be sent when your monitor suite finishes no matter if it passed or failed:

```
from spidermon.contrib.actions.email.ses import SendSESEmail

class DummyMonitorSuite(MonitorSuite):
    monitors = [
        DummyMonitor,
    ]

    monitors_finished_actions = [
        SendSESEmail,
    ]
```

By default, Spidermon uses a HTML template that can be altered in *SPIDERMON_BODY_HTML_TEMPLATE* setting. You can use Jinja2 as your template engine.

The result of a report generated using this default template may be seen next:



Report Title

Spider: dummy
Version: 75cab09-jesusretry-407
Items: -
Requests: 2
Stats: 20
Running Time: 0:00h
Finish Reason: finished
Monitors: 2 1

Job: 279854/100/100
Priority: 2
Tags: -

1 failed!

Monitors

Item validation monitor

Validation of the extracted item fields.

Required fields ✓

Job stats monitor

Job outcome ✓

Minimum number of items FAIL ✗

Monitor Failures

Job stats monitor/Minimum number of items FAIL

0 not greater than or equal to 1 : Number of scraped items is not greater than or equal to 1

```
Traceback (most recent call last):
      File "/tmp/unpacked-eggs/__main__.egg/datafeeds/monitors.py", line 20, in
test_minimum_number_of_items
        self.assertGreaterEqual(self.item_scraped_count(), expected_amount, msg)
    AssertionError: 0 not greater than or equal to 1 : Number of scraped items is not
greater than or equal to 1
```

Stats

```
{'downloader/request_bytes': 606,
 'downloader/request_count': 2,
 'downloader/request_method_count/GET': 2,
 'downloader/response_bytes': 13330,
 'downloader/response_count': 2,
 'downloader/response_status_count/200': 1,
 'downloader/response_status_count/301': 1,
 'finish_reason': 'finished',
 'finish_time': datetime.datetime(2018, 4, 20, 13, 39, 56, 586313),
 'log_count/DEBUG': 3,
 'log_count/INFO': 7,
 'log_count/WARNING': 3,
 'memusage/max': 163721216,
 'memusage/startup': 163721216,
 'response_received_count': 1,
 'scheduler/dequeued': 2,
 'scheduler/dequeued/disk': 2,
 'scheduler/enqueued': 2,
 'scheduler/enqueued/disk': 2,
 'start_time': datetime.datetime(2018, 4, 20, 13, 39, 56, 418496)}
```

You can also define actions for when your monitors fails or passes also including actions to the lists *monitors_passed_actions* and *monitors_failed_actions*.

The following settings are the minimum needed to make this action works:

SPIDERMON_AWS_ACCESS_KEY

Default: `None`

AWS Access Key.

> **Warning:** This setting has been deprecated in preference of `SPIDERMON_AWS_ACCESS_KEY_ID`.

SPIDERMON_AWS_SECRET_KEY

Default: `None`

AWS Secret Key.

Warning: This setting has been deprecated in preference of `SPIDERMON_AWS_SECRET_ACCESS_KEY`.

SPIDERMON_AWS_ACCESS_KEY_ID

Default: `None`

AWS Access Key. If not set, it defaults to AWS_ACCESS_KEY_ID (`scrapy` credentials for AWS S3 storage).

SPIDERMON_AWS_SECRET_ACCESS_KEY

Default: `None`

AWS Secret Key. If not set, it defaults to AWS_SECRET_ACCESS_KEY (`scrapy` credentials for AWS S3 storage).

SPIDERMON_AWS_REGION_NAME

AWS Region.

Default: `us-east-1`

SPIDERMON_EMAIL_SENDER

Default: `None`

Address of the sender of the e-mail notification.

SPIDERMON_EMAIL_TO

Default: `None`

List of all recipients of the e-mail notification.

The following settings can be used to improve the action:

SPIDERMON_BODY_HTML

Default: `None`

SPIDERMON_BODY_HTML_TEMPLATE

String containing the location of the Jinja2 template for the Spidermon email report.

Default `reports/email/monitors/result.jinja`.

SPIDERMON_BODY_TEXT

SPIDERMON_BODY_TEXT_TEMPLATE

SPIDERMON_EMAIL_BCC

Default: `None`

SPIDERMON_EMAIL_CONTEXT

Default: `None`

SPIDERMON_EMAIL_CC

Default: `None`

SPIDERMON_EMAIL_FAKE

Default: `False`

If set *True*, the e-mail content will be in the logs but no e-mail will be sent.

SPIDERMON_EMAIL_REPLY_TO

SPIDERMON_EMAIL_SUBJECT

SPIDERMON_EMAIL_SUBJECT_TEMPLATE

1.8.2 Slack action

This action allows you to send custom messages to a Slack channel (or user) using a bot when your monitor suites finish their execution.

To use this action you need to:

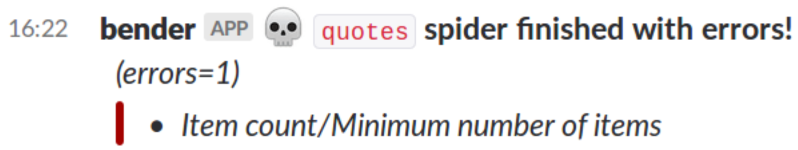
1. Install slackclient 2.0 or higher:

```
$ pip install "slackclient>=2.0"
```

2. Provide the Slack credentials in your `settings.py` file as follows:

```
# settings.py
SPIDERMON_SLACK_SENDER_TOKEN = '<SLACK_SENDER_TOKEN>'
SPIDERMON_SLACK_SENDER_NAME = '<SLACK_SENDER_NAME>'
SPIDERMON_SLACK_RECIPIENTS = ['@yourself', '#yourprojectchannel']
```

A notification will look like the following one:



16:22 **bender** APP 💀 `quotes` **spider finished with errors!**

(errors=1)

- *Item count/Minimum number of items*

Follow *these steps* to configure your Slack bot.

The following settings are the minimum needed to make this action works:

SPIDERMON_SLACK_RECIPIENTS

List of recipients of the message. It could be a channel or an user.

SPIDERMON_SLACK_SENDER_NAME

Username of your bot.

SPIDERMON_SLACK_SENDER_TOKEN

Bot User OAuth Access Token of your bot.

> **Warning:** Be careful when using bot user tokens in Spidermon. Do not publish bot user tokens in public code repositories.

Other settings available:

SPIDERMON_SLACK_ATTACHMENTS

SPIDERMON_SLACK_ATTACHMENTS_TEMPLATE

SPIDERMON_SLACK_FAKE

Default: `False`

If set *True*, the Slack message content will be in the logs but nothing will be sent.

SPIDERMON_SLACK_INCLUDE_ATTACHMENTS

SPIDERMON_SLACK_INCLUDE_MESSAGE

SPIDERMON_SLACK_MESSAGE

SPIDERMON_SLACK_MESSAGE_TEMPLATE

SPIDERMON_SLACK_NOTIFIER_INCLUDE_ERROR_ATTACHMENTS

SPIDERMON_SLACK_NOTIFIER_INCLUDE_OK_ATTACHMENTS

SPIDERMON_SLACK_NOTIFIER_INCLUDE_REPORT_LINK

SPIDERMON_SLACK_NOTIFIER_REPORT_INDEX

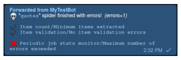
1.8.3 Telegram action

This action allows you to send custom messages to a Telegram channel, group or user using a bot when your monitor suites finish their execution.

To use this action you need to provide the Telegram bot token in your `settings.py` file as follows:

```
# settings.py
SPIDERMON_TELEGRAM_SENDER_TOKEN = '<TELEGRAM_SENDER_TOKEN>'
SPIDERMON_TELEGRAM_RECIPIENTS = ['chatid', 'groupid' '@channelname']
```

A notification will look like the following:



Forwarded from MyTestBot

💀 "quotes" spider finished with errors! *(errors=1)*

✓ Item count/Minimum items extracted
✓ Item validation/No item validation errors

❌ Periodic job stats monitor/Maximum number of errors exceeded

Follow *these steps* to configure your Telegram bot.

The following settings are the minimum needed to make this action work:

SPIDERMON_TELEGRAM_RECIPIENTS

List of recipients of the message. It could be a user id, group id or channel name.

SPIDERMON_TELEGRAM_SENDER_TOKEN

Bot Authorization Token of your bot.

> **Warning:** Be careful when using bot user tokens in Spidermon. Do not publish bot user tokens in public code repositories.

Other settings available:

SPIDERMON_TELEGRAM_FAKE

Default: `False`

If set *True*, the Telegram message content will be in the logs but nothing will be sent.

SPIDERMON_TELEGRAM_MESSAGE

The message to be sent, it supports Jinja2 template formatting.

SPIDERMON_TELEGRAM_MESSAGE_TEMPLATE

Path to a Jinja2 template file to format messages sent by the Telegram Action.

1.8.4 Job tags action

If you are running your spider using the Scrapy Cloud you are able to include tags in your jobs. Spidermon includes two actions that may be used to add or to remove tags to your jobs depending on the result of the monitoring.

In this example, considering that you defined a *running* tag when you start the job in Scrapy Cloud, if the job passes without errors, it will remove this tag. If the job fails the *failed* tag will be added to the job so you can easily look for failed jobs.

```
# monitors.py
from spidermon.contrib.actions.jobs.tags import AddJobTags, RemoveJobTags

class DummyMonitorSuite(MonitorSuite):
    monitors = [
        DummyMonitor,
    ]

    monitors_passed_actions = [
        RemoveJobTags,
    ]

    monitors_failed_actions = [
        AddJobTags,
    ]
```

```
# settings.py
SPIDERMON_JOB_TAGS_TO_ADD = ['failed', ]
SPIDERMON_JOB_TAGS_TO_REMOVE = ['running', ]
```

By default we have the following settings when using these two actions:

SPIDERMON_JOB_TAGS_TO_ADD

List of tags to be included when *AddJobTags* is executed.

SPIDERMON_JOB_TAGS_TO_REMOVE

List of tags to be removed when *RemoveJobTags* is executed.

If you want to have different rules adding or removing tags for different results of the monitoring, you need to create a custom action class including the name of the setting that will contain the list of tags that will be included in the job:

```
# monitors.py
from spidermon.contrib.actions.jobs.tags import AddJobTags

class AddJobTagsPassed(AddJobTags):
    tag_settings = 'TAG_TO_ADD_WHEN_PASS'

class AddJobTagsFailed(AddJobTags):
    tag_settings = 'TAG_TO_ADD_WHEN_FAIL'

class DummyMonitorSuite(MonitorSuite):
    monitors = [
        DummyMonitor,
    ]
```

(continues on next page)

(continued from previous page)

```

    monitors_passed_actions = [
        AddJobTagsPassed,
    ]

    monitors_failed_actions = [
        AddJobTagsFailed,
    ]
```

```
# settings.py
TAG_TO_ADD_WHEN_PASS = ['passed', ]
TAG_TO_ADD_WHEN_FAIL = ['failed', ]
```

1.8.5 File Report action

This action allows to create a file report based on a template. As *E-mail action* you can use Jinja2 as your template engine.

In this example we will create a file called *my_report.html* when the monitor suite finishes:

```
# monitors.py
from spidermon.contrib.actions.reports.files import CreateFileReport

class DummyMonitorSuite(MonitorSuite):
    monitors = [
        DummyMonitor,
    ]

    monitors_finished_actions = [
        CreateFileReport,
    ]
```

```
# settings.py
SPIDERMON_REPORT_TEMPLATE = 'reports/email/monitors/result.jinja'
SPIDERMON_REPORT_CONTEXT = {
    'report_title': 'Spidermon File Report'
}
SPIDERMON_REPORT_FILENAME = 'my_report.html'
```

Settings available:

SPIDERMON_REPORT_CONTEXT

Dictionary containing context variables to be included in your report.

SPIDERMON_REPORT_FILENAME

String containing the path of the generated report file.

SPIDERMON_REPORT_TEMPLATE

String containing the location of the template for the file report.

1.8.6 S3 Report action

This action works exactly like *File Report action* but instead of saving the generated report locally, it uploads it to a S3 Amazon Bucket.

Settings available:

SPIDERMON_REPORT_S3_BUCKET

SPIDERMON_REPORT_S3_CONTENT_TYPE

SPIDERMON_REPORT_S3_FILENAME

SPIDERMON_REPORT_S3_MAKE_PUBLIC

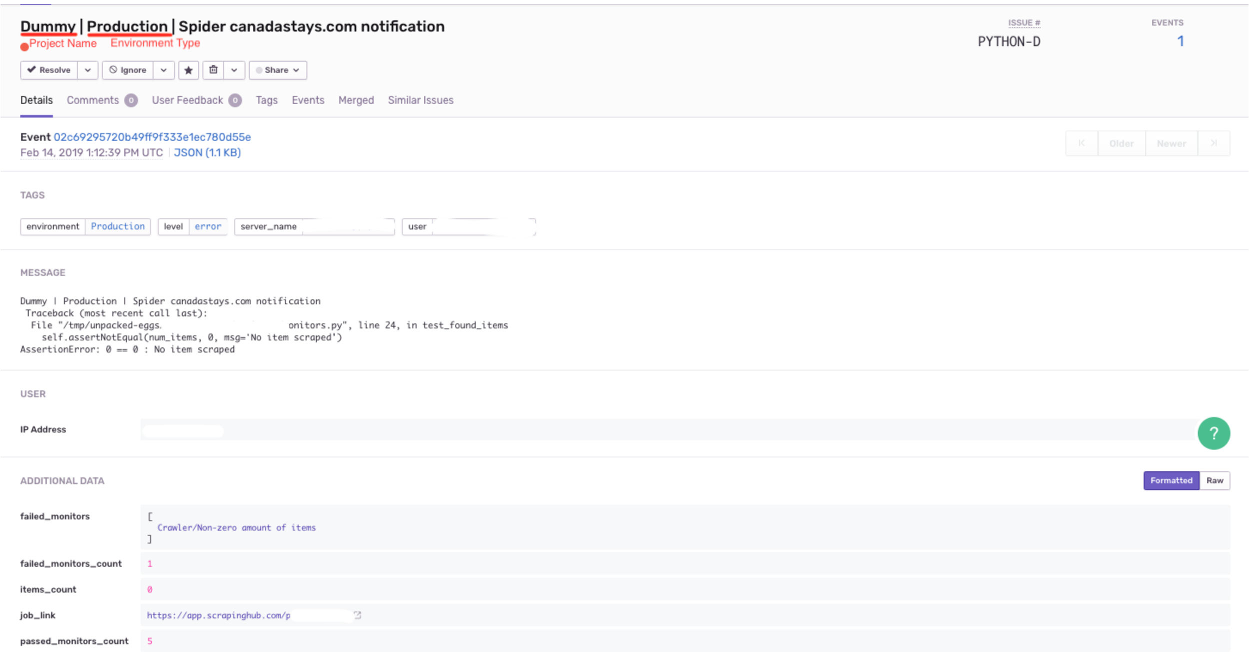
SPIDERMON_REPORT_S3_REGION_ENDPOINT

1.8.7 Sentry action

This action allows you to send custom messages to Sentry when your monitor suites finish their execution. To use this action you need to provide the Sentry DSN in your *settings.py* file as follows:

```
# settings.py
SPIDERMON_SENTRY_DSN = '<SENTRY_DSN_URL>'
SPIDERMON_SENTRY_PROJECT_NAME = '<PROJECT_NAME>'
SPIDERMON_SENTRY_ENVIRONMENT_TYPE = '<ENVIRONMENT_TYPE>'
```

A notification on Sentry will look like the following one:



Dummy | Production | Spider canadastays.com notification

ISSUE # PYTHON-D

EVENTS 1

Event 02c69295720b49ff9f333e1ec780d55e
Feb 14, 2019 1:12:39 PM UTC | JSON (1.1 KB)

TAGS

environment Production level error server_name user

MESSAGE

```
Dummy | Production | Spider canadastays.com notification
 Traceback (most recent call last):
  File "/tmp/unpacked-eggs.                    onitors.py", line 24, in test_found_items
    self.assertNotEqual(num_items, 0, msg='No item scraped')
AssertionError: 0 == 0 : No item scraped
```

USER

IP Address

ADDITIONAL DATA

failed_monitors	[Crawler/Non-zero amount of items]
failed_monitors_count	1
items_count	0
job_link	https://app.scrapinghub.com/p
passed_monitors_count	5

The message will have tags based on the failed monitor names (after replacing whitespace, special symbols etc.), but as the tag length is limited to 32 chars you should use `@monitors.name` to set monitor names that will produce useful tag names.

The following settings are needed to make this action workable:

SPIDERMON_SENTRY_DSN

Data Source Name provided by Sentry, it's a representation of the configuration required by the Sentry SDKs.

SPIDERMON_SENTRY_PROJECT_NAME

Project name to use in notification title.

SPIDERMON_SENTRY_ENVIRONMENT_TYPE

Default: `Development`

Environment type to use in notification title. It could be set to anything like local, staging, development or production.

SPIDERMON_SENTRY_LOG_LEVEL

Default: `error`

It could be set to any level provided by Sentry Log Level

SPIDERMON_SENTRY_FAKE

Default: `False`

If set *True*, the Sentry message will be in the logs but nothing will be sent.

1.8.8 Custom actions

You can define your own custom actions to be executed by your monitor suites. Just create a class that inherits from *spidermon.core.actions.Action* and implement the *run_action* method.

```python
from spidermon.core.actions import Action

class MyCustomAction(Action):
    def run_action(self):
        # Include here the logic of your action
        # (...)
```

1.9 Release notes

1.9.1 1.15.1 (2021-10-04)

- chore: pin *jsonschema* version to 3.2.0 to avoid problems with newest version that has backward incompatible changes
- chore: pin *schematics* version to 2.1.0 to avoid problems with newest version that has backward incompatible changes

1.9.2 1.15.0 (2021-04-06)

- feature: Improve content of Sentry messages (PR#279)
- bug: Replace *boto* with *boto3* for Amazon SES work correctly (issue#285)

1.9.3 1.14.0 (2020-10-05)

- feature: Built-in monitor for field and item coverage (issue#253)
- feature: Add field coverage statistics (PR#262)
- chore: Update required slackclient version (PR#265)
- chore: Add Python 3.8 support (issue#255)
- chore: Drop Python 3.5 support (issue#266)
- chore: Remove test decorator that skips tests if executed in old Python versions (PR#258)
- chore: Fix deprecation warnings (PR#272, PR#274)
- docs: Fix inconsistent examples (PR#273)

1.9.4 1.13.0 (2020-06-23)

- bug: Fix Telegram action error logging
- feature: Disable item validation pipeline when Spidermon is disabled
- feature: Item validation built in monitor
- chore: Removed Python 2.7 support
- docs: Improved documentation organization

1.9.5 1.12.2 (2020-05-07)

- Fixed version 1.12.1 changelog

1.9.6 1.12.1 (2020-05-07)

- bugfix: AttributeError when using ValidationMonitorMixin (issue)
- docs: How-To Guide - Adding required fields coverage validation (pull request)

1.9.7 1.12.0 (2020-01-09)

- Dropped python 3.4 support
- Added action to send monitor reports to Telegram
- Added fallback to scrapy AWS settings
- Logged errors from Slack API calls
- Allowed to define SPIDERMON_SLACK_RECIPIENTS setting as a comma-separated string with the desired recipients

- Read SES settings with getlist
- Added documentation of Expression Monitors
- Improved Slack action documentation
- Fixed sphinx warnings when building docs
- Fixed warnings in docs build
- Validate docs build in CI
- Applied and enforced black formatting on spidermon source code
- Configured test coverage reporting in project

1.9.8 1.11.0 (2019-08-02)

- Allowed per-field checking in ValidationMonitorMixin
- Added option to set AWS Region Name on SES E-Mail action
- Added default value for 'SPIDERMON_BODY_HTML_TEMPLATE' setting
- Fixed bug in logging of Slack messages when fake setting is enabled
- Enforced lxml 4.3.5 or lower for Python 3.4
- Improved stats history documentation

1.9.9 1.10.2 (2019-07-01)

- Version 1.10.1 with CHANGELOG updated

1.9.10 1.10.1 (2019-07-01)

- Allowed to add absolute location for custom templates

1.9.11 1.10.0 (2019-06-12)

- Added new StatsCollector that access stats data from previous spider executions.
- Added new setting to define the max number of unwanted HTTP status codes allowed in built-in monitor.
- Improved validation error messages with JSON Schema when additional fields are found.
- Made possible to retrieve JSON schema files from external locations.
- Included documentation of periodic monitor suites.
- Fixed bug caused by new slackclient release.
- Other small documentation improvements.

1.9.12 1.9.0 (2019-03-11)

- Add set of built-in basic monitors with the most common test methods to allow start monitoring spiders more straightforward.
- Add SendSentryMessage action to send notifications to Sentry containing the results of Spidermon execution.
- Add SPIDERMON_ENGINE_STOP_MONITORS setting to list monitors to be executed when the Scrapy engine is stopped.
- Fix bug that prevented the use of custom model-level validators in schematics models.
- Refactor JSONSchemaValidator to allow select different versions of JSON Schema.
- Refactor requirements in setup.py to include missing required dependencies.
- Fix bug caused by backward incompatible change in jsonschema 3.0.0.
- Fix example code of tutorial.
- Install documentation improvements.

1.9.13 1.8.0 (2019-01-08)

- Remove CreateJobReport action.
- Include new documentation and tutorial code.
- Rename internal method in MonitorRunner to fix typo.

1.9.14 1.7.0 (2018-12-04)

- Support universal wheels.
- Skip authentication and recipient settings when running in fake mode.

1.9.15 1.6.0 (2018-11-09)

- Add SPIDERMON_EMAIL_CONTEXT setting to pass custom contexts to email actions.
- Add support for Schematics 2.1.0.

1.9.16 1.5.0 (2018-09-19)

- Convert the job ID tag into a clickable button.

1.9.17 1.4.0 (2018-08-17)

- Avoid requests to get the amount of lines in the log by default, because they consume too much memory and they are very slow. You can still use the old behavior adding `show_log_count` to the context before creating the email message.
- Refactor the requirements in setup.py.
- Update the Sphinx configuration.

1.9.18 1.3.0 (2018-08-02)

- Add support for periodic monitors in the Scrapy extension.

1.9.19 1.2.0 (2018-04-04)

- Modify ItemValidationPipeline in order to support dict objects in addition to Scrapy.Item objects.
- Refactor ItemValidationPipeline to make it easier to extend this class.

1.9.20 1.1.0 (2018-03-23)

- Add Schematics 2.* support. Note that Schematics 2.0.0 introduced many changes to its API and even some validation rules have a slightly different behaviour in some cases.
- ItemValidationPipeline optimisations for cases where no validators can be applied.

1.9.21 1.0.0 (2018-03-08)

- Add Python 3 support.
- Run tests on Python 2 and Python 3.
- Add dependencies for optional validation features to setup.py.
- Import HubstorageClient from the scrapinghub library if available.
- Replace dash.scrapinghub.com with app.scrapinghub.com.

Backwards Incompatible Changes

- Rename attachements attribute in the SendSlackMessage class to attachments.
- Add the SPIDERMON_ENABLED setting to control if the Scrapy extension should run (note that it is disabled by default).

Python Module Index

Index