Alarmageddon Documentation

Release 0.1

Tim Stewart, Scott Hellman

Dec 09, 2019

Contents

1 Getting Started **3**
- 1.1 A Short Example . . . 3
- 1.2 Validations . . . 5
- 1.3 Publishers . . . 7
- 1.4 Using the Email Publisher . . . 9

2 Source **13**
- 2.1 alarmageddon package . . . 13

3 Indices and tables **15**

Alarmägeddön

Alarmageddon is a Python monitoring framework for RESTful services, built on top of Requests and Fabric. Alarmageddon supports Python 2.7 and Python 3.4+

The following example GETs www.google.com, and reports to HipChat if the return code is not 200:

```
import alarmageddon
from alarmageddon.validations.http import HttpValidation
from alarmageddon.publishing.hipchat import HipChatPublisher

validations = [HttpValidation.get("http://www.google.com").expect_status_codes([200])]

publishers = [HipChatPublisher("hipchat.route.here","token","stable","hipchat_room")]

alarmageddon.run_tests(validations,publishers)
```

- Verify expectations on the following
  - HTTP requests
  - SSH commands
  - RabbitMQ queue lengths
  - Cassandra status
  - Statistics collected in Graphite
  - The behavior of other Alarmageddon tests
- Report failed verifications to
  - HipChat
  - Slack
  - PagerDuty
  - Graphite
  - Email
  - XML file

CHAPTER 1

Getting Started

1.1 A Short Example

This will walk you through creating and running a basic suite of Alarmageddon validations.

Alarmageddon has two main components: validations and publishers. Validations are the tests that will be run, and publishers handle passing the results of those validations along to an external system (eg, PagerDuty).

1.1.1 Creating Validations

To make sure that the world's search engines are working, let's use HttpValidation:

```
from alarmageddon.validations.http import HttpValidation

validations = []
validations.append(HttpValidation.get("http://www.google.com").expect_status_
↪codes([200]))
validations.append(HttpValidation.get("http://www.bing.com").expect_status_
↪codes([200]))
validations.append(HttpValidation.get("http://www.yahoo.com").expect_status_
↪codes([200]))
```

These validations are constructed to GET the supplied url. We've also set up our expectations about the results of GETting the url - in this case, we expect the status code to be 200. This is the basic structure of Alarmageddon's validations: a validation takes some action and compares the results to the supplied expectations.

1.1.2 Creating Publishers

Of course, if no one knows a validation has failed, it isn't particularly useful. To have Alarmageddon report on failures, we must supply it with at least one publisher:

```
from alarmageddon.publishing.hipchat import HipChatPublisher

publishers = []
hipchat_endpoint = "127.0.0.1"
hipchat_token = "token"
environment = "stable"
room = "hipchat_room"
publishers.append(HipChatPublisher(hipchat_endpoint, hipchat_token, environment,␣
↪room))
```

This publisher will report failures to hipchat. Note that this example won't work - you'll need to supply a valid endpoint and token!

1.1.3 Running Alarmageddon

Given a set of validations and a set of publishers, we can run Alarmageddon:

```
import alarmageddon

alarmageddon.run_tests(validations,publishers)
```

This will run the validations. If any failures occur, a message will be passed along to the designated HipChat room. In this case, the resulting message might look something like:

> 1 failure(s) in stable: (failed) GET http://www.google.com Description: expected status code: 200, actual status code: 504 (Gateway Time-out)

1.1.4 Full Code

Here's the full source of this example:

```
import alarmageddon
from alarmageddon.validations.http import HttpValidation
from alarmageddon.publishing.hipchat import HipChatPublisher

validations = []
validations.append(HttpValidation.get("http://www.google.com").expect_status_
↪codes([200]))
validations.append(HttpValidation.get("http://www.bing.com").expect_status_
↪codes([200]))
validations.append(HttpValidation.get("http://www.yahoo.com").expect_status_
↪codes([200]))

publishers = []
hipchat_endpoint = "127.0.0.1"
hipchat_token = "token"
environment = "stable"
room = "hipchat_room"
publishers.append(HipChatPublisher(hipchat_endpoint, hipchat_token, environment,␣
↪room))

alarmageddon.run_tests(validations,publishers)
```

1.2 Validations

A validation performs some action and then checks the results of that action against a set of expectations. Alarmageddon comes with validations for checking the results of HTTP calls, checking the output of SSH commands, and checking the length of RabbitMQ queues.

All validations accept a `priority` argument. This should be one of `Priority.LOW`, `Priority.NORMAL`, or `Priority.CRITICAL`. This priority level is used to determine whether or not a publisher should publish the results of the validation.

1.2.1 HTTP

You can create HttpValidations for various HTTP methods:

```
HttpValidation.get("http://www.google.com")
HttpValidation.post("http://www.google.com",data={key:value})
HttpValidation.put("http://www.google.com"data={key:value})
HttpValidation.options("http://www.google.com")
HttpValidation.head("http://www.google.com")
```

You can change the timeout length:

```
HttpValidation.get("http://www.google.com", timeout=10)
```

Or designate a number of retry attempts:

```
HttpValidation.get("http://www.google.com", retries=10)
```

You can supply custom headers:

```
header = {"Authorization":"value"}
HttpValidation.get("http://www.google.com", headers=header)
```

If you've created a validation that you would like to apply to multiple hosts:

```
validation = HttpValidation.get("http://www.google.com")
hosts = ["http://www.bing.com","http://www.yahoo.com"]
new_validations = validation.duplicate_with_hosts(hosts)
```

An example of expectations on HttpValidations, where we expect to get either a 200 or 404 status code, and expect the result to contain JSON with the designated value:

```
validation = HttpValidation.get("url")
validation.expect_status_codes([200,404])
validation.expect_json_property_value("json.path.to.value","expected")
```

`expect_json_property_value` accepts query string that allows you to pluck values from json. Consider the following json document

{"abc": "123", "another": {"nested": "entry"}, "alpha": {"array": [1, 2, 3, 4]}}

`abc` will reference `"123"` `another.nested` will reference `"entry"` `array[4]` will reference `4` `array[*]` will reference `[1, 2, 3, 4]`

1.2.2 SSH

To perform validations over SSH, you'll need to supply the appropriate credentials:

```
ctx = SshContext("username","keyfile_path")
```

You can check the average load:

```
LoadAverageValidation(ctx).expect_max_1_minute_load(5, hosts=['127.0.0.1'])
```

You can verify that an upstart service is running:

```
UpstartServiceValidation(ctx, "service_name", hosts=['127.0.0.1'])
```

But ultimately, the above are just convenience classes for common use cases - you can perform arbitrary commands and check the output:

```
validation = SshCommandValidation(ctx, "validation name", "ps -ef | grep python",␣
↪hosts=['127.0.0.1'])
validation.expect_output_contains("python")
```

1.2.3 Cassandra

Cassandra validations are a special case of SSH validations:

```
CassandraStatusValidation(ssh_ctx, hosts=['127.0.0.1'])
```

1.2.4 Kafka

Kafka validations will inspect your kafka partitions and leader elections. If a single partition has multiple leaders the validation will fail:

```
KafkaStatusValidation(ssh_ctx, zookeeper_nodes='127.0.0.1:2181,127.0.0.2:2181,127.0.0.
↪3:2181',hosts=['127.0.0.1'])
```

1.2.5 RabbitMQ

As with SSH, you have to supply credentials for RabbitMQ Validations:

```
ctx = RabbitMqContext("127.0.0.1",80,"username","password")
```

Once you have the context, you can construct validations that check that the number of messages in a queue is less than some value. For example, the following will fail if the queue "queue_name" has more than 1000 messages in it:

```
RabbitMqValidation(ctx, "validation name", "queue_name", 1000)
```

1.2.6 Graphite

You also need a context for Graphite:

```
ctx = GraphiteContext("127.0.0.1")
```

Given the context, you can check statistics on various Graphite readings:

```
validation = GraphiteValidation(ctx, "validation name", "Errors")
validation.expect_average_in_range(1,10)
```

1.2.7 Validation Groups and GroupValidations

You may have a set of tests where individual failures are minor but multiple failures indicate a problem (eg, machines behind an HAProxy). Alarmageddon Validations include the notion of a validation group, which indicate that a set of validations belong together:

```
validations = []
validations.append(HttpValidation.get("http://www.google.com",group="a").expect_
↪status_codes([200]))
validations.append(HttpValidation.get("http://www.yahoo.com",group="a").expect_status_
↪codes([200]))
validations.append(HttpValidation.get("http://www.bing.com",group="a").expect_status_
↪codes([200]))
```

In this case, we have three validations that belong to the validation group "a". Now that we have a group, we can create a GroupValidation that contains expectations about the results of other validations:

```
validations.append(GroupValidation("Group a Validation", "a", normal_threshold=1,␣
↪critical_threshold=2))
```

This new validation does not have an explicit priority level. Rather, it defaults to LOW priority. If the number of failures in group "a" reaches the normal_threshold, the validation will be considered a failure and the priority will become NORMAL. If it reaches the critical_threshold, the priority will become CRITICAL (and the validation will still be a failure).

You can create GroupValidations on groups of GroupValidations. The only difference is that an `order` parameter must be passed, to ensure that the tests are run in the correct order:

```
validations.append(GroupValidation("Group a Validation", "a", normal_threshold=1,␣
↪critical_threshold=2, group="c"))
validations.append(GroupValidation("Group b Validation", "b", normal_threshold=1,␣
↪critical_threshold=2, group="c"))
validations.append(alarmageddon.validation.GroupValidation("Group c Validation", "c",␣
↪normal_threshold=2, order=2))
```

1.3 Publishers

All publishers accept a `priority_threshold` argument. This should be one of `Priority.LOW`, `Priority.NORMAL`, or `Priority.CRITICAL`. A publisher will only publish failing validations if they are at least as critical as the `priority_threshold`. For example, to report on all failures, you should set your publisher's `priority_threshold` to `Priority.LOW`.

1.3.1 JUnit XML

The JUnit XML publisher will write out all validation results to an XML file. This publisher is automatically created when you run the validations, and will write out to results.xml.

1.3.2 HipChat

The HipChat publisher will report failures to your hipchat room:

```
HipChatPublisher("hipchat.route.here","token","stable","hipchat_room")
```

By default, the HipChat publisher alerts on failures of NORMAL priority or higher.

1.3.3 Slack

The Slack publisher will report failures to your slack channel:

```
SlackPublisher("hook.url","stable")
```

`hook.url` should be a slack incoming web hook integration to the channel that should be published to. By default, the Slack publisher alerts on failures of NORMAL priority or higher.

1.3.4 Http

The Http publisher will report failures to an HTTP Server:

```
HttpPubliser(success_url="success.url.here", success_url="failure.url.here")
```

1.3.5 PagerDuty

The PagerDuty publisher will report failures to PagerDuty:

```
PagerDutyPublisher("pagerduty.route.here", "pagerduty_key")
```

By default, the PagerDuty publisher alerts only on CRITICAL failures.

1.3.6 Graphite

The Graphite publisher behaves slightly differently than the other publishers. Instead of only logging failures, it logs both successes and failures, providing you with a way to keep track of how often certain validations are passing or failing:

```
GraphitePublisher("127.0.0.1",8080)
```

The GraphitePublisher will also keep track of how long the validations took, in the case of HttpValidations. By default, GraphitePublisher will publish on all validations.

1.3.7 Email

There are two email publishers. SimpleEmailPublisher provides basic emailing functionality, and will email all test results to the supplied addresses:

```
SimpleEmailPublisher({"real_name": "test", "address": "sender@test.com"},
                     [{"real_name": "test", "address": "recipient@test.com"}],
                     host='127.0.0.1', port=1234)
```

EmailPublisher provides more granular control over the sent messages. For this reason, validations that will be published by the email publisher must be enriched with extra information.

To create an email publisher, you need a config object with the appropriate values in it, and optionally a set of defaults for missing config values:

```
email_pub = EmailPublisher(config, defaults=general_defaults)
```

For enrichment, a convenience method is provided in emailer to ensure that the appropriate value are present:

```
emailer.enrich(validation, email_settings, runtime_context)
```

Note: For the email publisher to publish a failure, the priority threshold must be reached **and** the validation must be enriched.

1.4 Using the Email Publisher

Due to the extensive flexibility allowed by the email publisher, it involves more configuration than the other publishers. This page is intended to be a guide through that process.

1.4.1 Publisher Configuration

Constructing the Email Publisher is similar to other publishers:

```
EmailPublisher(config, priority_threshold=Priority.NORMAL, defaults=general_email_
↪defaults)
```

`config` is the usual Alarmageddon config object, but it must contain the following email specific field:

```
"email_template_directory" : "path/to/email/templates"
```

This specifies where the email jinja templates can be found.

There are a few optional fields as well. The full email settings might look like this:

```
{
    "email_host" : null,
    "email_port" : null,
    "email_template_directory" : "email_templates",
    "email_defaults" : {
        "general" : {
            "email_template" : "default.template",
            "email_subject_template" : "default_subject.template",
```

(continues on next page)

(continued from previous page)

```
            "email_sender" : {"real_name" : "Alarmageddon Monitor", "address" :
↪"noreply@host.com"},
            "email_recipients" : [
                {"real_name" : "Team", "address" : "team@host.com"}
            ],
            "email_custom_message" : ""
        },
    }
}
```

`email_host` and `email_port` can be set programatically or included in conf.json. `email_defaults` provides default information about what templates to use and extra fields that can be used in the template. You will have to programatically assign email_defaults to the publisher, as shown in the example constructor above.

1.4.2 Validation Enrichment

For the email publisher to successfully publish a message, the default information provided to each validation is not enough. To include this extra information, an enrichment function must be called on each validation:

```
emailer.enrich(validation,
               email_settings=validation_settings,
               runtime_context=email_runtime_context)
```

`validation_settings` should be a python dictionary of the form:

```
{
    "email_template" : "alert.template",
    "email_subject_template" : "alert_subject.template",
    "email_recipients" : [
        {"real_name" : "Another Team", "address" : "another@host.com"}
    ],
    "email_custom_message" : """The route located at {{test_name}} failed to respond␣
↪within the alloted time frame.
                                The node may be offline or missing."""
}
```

Note that these fields can also appear in the Email Publisher defaults. If validation-specific fields are present, they will be used instead of the defaults.

`runtime_context` is a dictionary that can contain arbitrary information to be used by the template.

Note: For a validation to be published by the Email Publisher, that validation must both be enriched **and** be of high enough priority.

You can use Alarmageddon's dry run feature to verify that the validations that you intended to be published by the email publisher will actually be published by the email publisher.

1.4.3 Email Templates

The Email Publisher uses jinja2 to create its messages. An example email template is provided below:

```
Validation Failure in environment {{env}}:
{{test_name}} - {{test_description}}

{{email_custom_message}}
```

The email templates are files stored in the `email_template_directory`.

CHAPTER 2

Source

2.1 alarmageddon package

2.1.1 Subpackages

alarmageddon.publishing package

Submodules

alarmageddon.publishing.emailer module

alarmageddon.publishing.exceptions module

alarmageddon.publishing.graphite module

alarmageddon.publishing.hipchat module

alarmageddon.publishing.slack module

alarmageddon.publishing.http module

alarmageddon.publishing.pagerduty module

alarmageddon.publishing.publisher module

Module contents

alarmageddon.validations package

Submodules

alarmageddon.validations.kafka module

alarmageddon.validations.cassandra module

alarmageddon.validations.graphite module

alarmageddon.validations.graphite_expectations module

alarmageddon.validations.http module

alarmageddon.validations.http_expectations module

alarmageddon.validations.json_expectations module

alarmageddon.validations.rabbitmq module

alarmageddon.validations.ssh module

Module contents

2.1.2 Submodules

2.1.3 alarmageddon.config module

2.1.4 alarmageddon.reporter module

2.1.5 alarmageddon.result module

2.1.6 alarmageddon.run module

2.1.7 Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search