flask-error-monitor Documentation

Release 1.0

Sonu Kumar

Nov 23, 2019

Contents

CHAPTER 1

Quick start

1.1 Installation

To install Flask Error Monitor, open an interactive shell and run:

```
pip install flask-error-monitor
```

1.2 User Guide

Note:

- It will mask all the variables which contain *password* and *secret* in their name.
- Recorded exceptions will be visible to *http://example.com/dev/error*

Using **Flask Error Monitor** as simple as plugging any flask extension. An instance of instance of *AppErrorMonitor* needs to be created and have to be configured with the correct data. Monitoring feature can be configured either using object based configuration or app-based configuration, the only important thing here is we should have all the required key configs in the app.config otherwise it will fail.

1.2.1 Recording exception/error

An error/exception can be recorded using decorator as well function call. - To record the error using decorator, decorate a function with *track_exception* - Where as to record error using function call use *record_exception* function.

All the data will be stored in the configured data store and these data will be available at */dev/error/* or at configure URL path.

1.3 Example setup

For object based configuration add **settings.py**

```
...
APP_ERROR_SEND_NOTIFICATION = True
APP_ERROR_RECIPIENT_EMAIL = ('example@example.com',)
APP_ERROR_SUBJECT_PREFIX = "Server Error"
APP_ERROR_EMAIL_SENDER = 'user@example.com'
```

app.py

```
from flask import Flask
from flask_mail import Mail
import settings
from flask_error_monitor import AppErrorMonitor
from flask_sqlalchemy import SQLAlchemy
...
app = Flask(__name__)
app.config.from_object(settings)
db = SQLAlchemy(app)
class MyMailer(Mail, NotificationMixin):
    def notify(self, message, exception):
        self.send(message)
mail = MyMailer(app=app)
error_monitor = AppErrorMonitor(app=app, db=db, notifier=mail)

....

....
# Record exception when 404 error code is raised
@app.errorhandler(403)
def error_403(e):
    error_monitor.record_exception()
    # any custom logic

# Record error using decorator
@app.errorhandler(500)
@error_monitor.track_exception
def error_500(e):
    # some custom logic
....
```

Here, app, db and notifier parameters are optional. Alternatively, you could use the init_app() method.

If you start this application and navigate to http://localhost:5000/dev/error, you should see an empty page.

1.4 Configure at the end

Use error_monitor.init_app method to configure

```
error_monitor = AppErrorMonitor()
...
error_monitor.init_app(app=app, db=db, notifier=mail)
```

1.5 Config details

- **Enable or disable notification sending feature**

```
APP_ERROR_SEND_NOTIFICATION = False
```

- **Email recipient list**

```
APP_ERROR_RECIPIENT_EMAIL = None
```

- **Email subject prefix to be used by email sender**

```
APP_ERROR_SUBJECT_PREFIX = ""
```

- **Mask value with following string**

```
APP_ERROR_MASK_WITH = "**************"
```

- **Masking rule** App can mask all the variables whose lower case name contains one of the configured string .. code:

```
APP_ERROR_MASKED_KEY_HAS = ("password", "secret")
```

Above configuration will mask the variable names like

```
password
secret
PassWord
THis_Is_SEcret
```

Note: Any variable names whose lower case string contains either *password* or *secret*

- **Browse link in your service app** List of exceptions can be seen at */dev/error*, but you can have other prefix as well due to some securities or other reasons.

```
APP_ERROR_URL_PREFIX = "/dev/error"
```

- **Email address used to construct Message object**

```
APP_ERROR_EMAIL_SENDER = "prod-issue@example.com"
```

1.6 Ticketing

Ticketing interface can be used to create tickets in the systems like Jira, Bugzilla etc, ticketing can be enabled using ticketing interface.

1.6.1 Using TicketingMixin

implement raise_ticket method of TicketingMixin interface

```
from flask_error_monitor import TicketingMixin
class Ticketing(TicketingMixin):
    def raise_ticket(self, exception):
        # Put your logic here

# create app as
app = Flask(__name__)
db = SQLAlchemy(app)
error_monitor = AppErrorMonitor(app=app, db=db, ticketing=Ticketing() )
db.create_all()
...
```

1.7 Custom Context

Having more and more context about failure always help in debugging, by default this app captures HTTP headers, URL parameters, any post data. More data can be included like data-center name, server details and any other, by default these details are not captured. Nonetheless these details can be captured using ContextBuilderMixin.

1.7.1 Using ContextBuilderMixin

implement get_context method of ContextBuilderMixin, default context builders capture *request body*, *headers* and URL parameters

```
class DefaultContextBuilder(ContextBuilderMixin):
    """
    Default request builder, this records, form data, header and URL parameters and␣
↪mask them if necessary
    """

    def get_context(self, request, masking=None):
        form = dict(request.form)
        headers = dict(request.headers)
        if masking:
            for key in form:
                masked, value = masking(key)
                if masked:
                    form[key] = value
            for key in headers:
                masked, value = masking(key)
                if masked:
                    headers[key] = value

        request_data = str({
            'headers': headers,
            'args': dict(request.args),
            'form': form
        })
        return request_data
```

Implementing custom context builder, in fact we can extend the default one to add more details as well

```
from flask_error_monitor.defaults import DefaultContextBuilder
class ContextBuilder(DefaultContextBuilder):
```

(continues on next page)

(continued from previous page)

```
    def get_context(self, request, masking=None):
        context = super().get_context(request, masking=masking)
        # add logic to update context
        return context
```

```
from flask_error_monitor import ContextBuilderMixin
class ContextBuilder(TicketingMixin):
    def get_context(self, request, masking=None):
        # add logic here
```

This custom context builder can be supplied as parameter of AppErrorMonitor constructor.

```
...
app = Flask(__name__)
db = SQLAlchemy(app)
error_monitor = AppErrorMonitor(app=app, db=db, ticketing=Ticketing() )
db.create_all()
return app, db, error_monitor
...
```

Using Mongo or other data store

Using any data store as easy as implementing all the methods from **ModelMixin**

```
from flask_error_monitor import ModelMixin
class CustomModel(ModelMixin):
    objects = {}

    @classmethod
    def delete(cls, rhash):
        ...

    @classmethod
    def create_or_update(cls, rhash, host, path, method, request_data, exception_
↪name, traceback):
        ...

    @classmethod
    def get_exceptions_per_page(cls, page_number=1):
        ...

    @classmethod
    def get(cls, rhash):
        ...


# create app with our own model
error_monitor = AppErrorMonitor(app=app, model=CustomModel)
```

Notification notify feature

Notifications are very useful in the case of failure, in different situations notification can be used to notify users using different channels like Slack, Email etc. Notification feature can be enabled by providing a *NotificationMixin* object.

```
from flask_error_monitor import NotificationMixin
class TestMail(NotificationMixin):
     def notify(self, message, exception):
         # what ever logic you want here

...
# create app
error_monitor = AppErrorMonitor(app=app, db=db, notifier=TestMail())
...
```

Masking Rule

Masking is essential for any system so that sensitive information can't be exposed in plain text form. Flask error monitor provides masking feature, that can be disabled or enabled.

- Disable masking rule: set `APP_ERROR_MASKED_KEY_HAS = None`
- To set other mask rule add following lines

```
#Mask all the variables or dictionary keys which contains from one of the following␣
↪tuple
APP_ERROR_MASKED_KEY_HAS = ( 'secret', 'card', 'credit', 'pass' )
#Replace value with  `##########`
APP_ERROR_MASK_WITH = "##########"
```

Note:

- Masking is performed for each variable like dict, list, set and all and it's done based on the *variable name*
- Masking is performed on the dictionary key as well as e.g. *ImmutableMultiDict*, standard dict or any object whose super class is dict.

Custom masking rule

Using MaskingMixin

implement __call__ method of

```
from flask_error_monitor import MaskingMixin
class MyMaskingRule(MaskingMixin):
    def __call__(self, key):
        # Put any logic
        # Do not mask return False, None
        # To mask return True, Value

# create app as
...
app = Flask(__name__)
db = SQLAlchemy(app)
error_monitor = AppErrorMonitor(app=app, db=db, masking=MyMaskingRule("#########", (
↪'pass', 'card') ) )
db.create_all()
return app, db, error_monitor
...
```

Using function

```
def mask(key):
    # Put any logic
    # Do not mask return False,None
    # To mask return True, Value

# create app as
...
app = Flask(__name__)
db = SQLAlchemy(app)
error_monitor = AppErrorMonitor(app=app, db=db, masking=mask )
db.create_all()
return app, db, error_monitor
```