error-tracker Documentation

Release 1.0

Sonu Kumar

Jan 29, 2020

Contents

Error Tracker is a python app plugins for Flask and Django, that provides many of the essentials features of system exceptions tracking.

CHAPTER 1

Features

- Mask all the variables, including dict keys, HTTP request body which contain *password* and *secret* in their name.
- Recorded exceptions will be visible to the configured path
- Send notification on failures
- Record exceptions with frame data, including local and global variables
- Raise bugs or update ticket in Bug tracking systems.
- Provide customization for notification, context building, ticketing systems and more

Exception Listing

Errors Seen

Host	Method	Path	Exception	Last seen	Occurrences	Action
http://127.0.0.1:5000/	GET	/go	IndentationError	2019-12-01 12:38:24.507043	1	Delete
http://127.0.0.1:5000/	GET	/go	RuntimeError	2019-12-01 12:38:11.642151	2	Delete
http://127.0.0.1:5000/	GET	/go	OSError	2019-12-01 12:37:45.876906	1	Delete
http://127.0.0.1:5000/	GET	/go	EOFError	2019-12-01 12:14:12.010774	1	Delete
http://127.0.0.1:5000/	GET	/go	KeyError	2019-11-30 10:46:08.704501	1	Delete
http://127.0.0.1:5000/	GET	/go	EnvironmentError	2019-11-30 10:46:04.704289	1	Delete
http://127.0.0.1:5000/	GET	/go	TypeError	2019-11-30 10:45:27.187173	1	Delete
http://127.0.0.1:5000/	GET	/dev/error/	AttributeError	2019-11-29 12:09:37.567483	1	Delete
http://127.0.0.1:5000/	GET	/go	ChildProcessError	2019-11-29 12:09:15.298448	1	Delete

Detailed Exception

```
File "/Users/sonu/Desktop/ps/venv/py3.7/lib/python3.7/site-packages/flask/app.py", line 1820, in handle_user_exception
e = EOFError()
exc_value = EOFError()
handler = None
tb = <traceback object at 0x10f67cf00>

File "/Users/sonu/Desktop/ps/venv/py3.7/lib/python3.7/site-packages/flask/_compat.py", line 39, in reraise
tb = <traceback object at 0x10f67cf00>
value = EOFError()

File "/Users/sonu/Desktop/ps/venv/py3.7/lib/python3.7/site-packages/flask/app.py", line 1951, in full_dispatch_request
rv = None

File "/Users/sonu/Desktop/ps/venv/py3.7/lib/python3.7/site-packages/flask/app.py", line 1935, in dispatch_request
req = <Request 'http://127.0.0.1:5000/go' [GET]>
rule = <Rule '/go' (HEAD, OPTIONS, GET) -> die>

File "/Users/sonu/Desktop/ps/error-tracker/examples/flask-sample/app.py", line 55, in die
exceptions = [<class 'KeyError'>, <class 'ArithmeticError'>, <class 'BaseException'>, <class 'IndentationError'>, <class 'IndexError'>, <class 'MemoryError'>, <class 'NameError'>,
<class 'NotImplementedError'>, <class 'ImportError'>, <class 'FloatingPointError'>, <class 'EOFError'>, <class 'OSError'>, <class 'AssertionError'>, <class 'AttributeError'>, <class
'GeneratorExit'>, <class 'Exception'>, <class 'OSError'>, <class 'ImportError'>, <class 'NotImplementedError'>, <class 'RuntimeError'>]
foo = {'password' : '**********', 'secret' : '**********'}
password = '**********'
```

1.1 Quick start

CHAPTER 2

Installation

To install Error Tracker, open an interactive shell and run:

```
pip install error-tracker
```

Error Tracker can be used with

- Standalone Python application
- Flask Application
- Django Application

Using **Error Tracker** as simple as plugging any other module.

CHAPTER 3

Recording exception/error

An error/exception can be recorded using decorator or function call.

- To record the error using decorator, decorate a function with `track_exception` or `auto_track_exception`
- Where as to record error using function call use `capture_exception` function.
- Exception detail can be written to a file, console or logger etc call method `print_exception`

All the data will be stored in the configured data store and these data will be available at configure URL path.

CHAPTER 4

Flask App setup

An instance of `AppErrorTracker` 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.

Note: Exception listing page is disabled by default. You need to enable that using view_permission parameter. view_permission function/callable class must return True/False based on the current request detail. This method would be called as view_permission(request).

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 error_tracker import AppErrorTracker, NotificationMixin
from flask_sqlalchemy import SQLAlchemy
...
app = Flask(__name__)
app.config.from_object(settings)
db = SQLAlchemy(app)
class Notifier(Mail, NotificationMixin):
    def notify(self, request, exception,
               email_subject=None,
               email_body=None,
               from_email=None,
```

(continues on next page)

(continued from previous page)

```
                recipient_list=None):
        message = Message(email_subject, recipient_list, email_body, sender=from_
↪email)
        self.send(message)
mailer = Notifier(app=app)

# enable for all users
class ViewPermission(ViewPermissionMixin):
    def __call__(self, request):
        return True

error_tracker = AppErrorTracker(app=app, db=db, notifier=mailer, view_
↪permission=ViewPermission())


....

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

# Record error using decorator
@app.errorhandler(500)
@error_tracker.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.

CHAPTER 5

Django App Setup

We need to update settings.py file as

- Add app `error_tracker.DjangoErrorTracker` to installed apps list
- Add Middleware `error_tracker.django.middleware.ExceptionTrackerMiddleWare` for exception tracking[1].
- Other configs related to notification
- Add URLs to the list of URL patterns

Note: Exception listing page is only enable for super users by default. You can enable for others by providing a custom implementation of ViewPermissionMixin. This class must return True/False based on the current request, False means not authorized, True means authorized.

```
...
APP_ERROR_RECIPIENT_EMAIL = ('example@example.com',)
APP_ERROR_SUBJECT_PREFIX = "Server Error"
APP_ERROR_EMAIL_SENDER = 'user@example.com'
# optional setting otherwise it's enabled for super users only
APP_ERROR_VIEW_PERMISSION = 'permission.ErrorViewPermission'

INSTALLED_APPS = [
    ...
    'error_tracker.DjangoErrorTracker'
]
MIDDLEWARE = [
    ...
    'error_tracker.django.middleware.ExceptionTrackerMiddleWare'
]
```

We need to add URLs to the urls.py so that we can browse the default pages provided by Error Tracker

[1] This should be added at the end so that it can process exception 1st in the middleware call stack.

```
from error_tracker.django import urls

urlpatterns = [
    ...
    url("dev/", include(urls)),
]
```

CHAPTER 6

Using With Python App (NO WEB SERVER)

Choose either of the preferred framework, flask or Django and configure the app as per their specifications. For example, if we want to use Flask then do

- **Flask App**
  - Create Flask App instance
  - Create Error Tracker app instance
  - DO NOT call run method of Flask app instance
  - To track exception call `capture_exception` method
- **Django App**
  - Create Django App with settings and all configuration
  - Set environment variable **DJANGO_SETTINGS_MODULE**
  - call `django.setup()`
  - `from error_tracker import error_tracker`
  - To track exception do `capture_exception(None, exception)`

6.1 Flask App Usage

6.1.1 Lazy initialization

Use error_tracker.init_app method to configure

```
error_tracker = AppErrorTracker()
...
error_tracker.init_app(app=app, db=db, notifier=notifier)
```

6.1.2 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"
```

6.1.3 Manual Exception Tracking

Error can be tracked programmatically using AppErrorTracker's capture_exception method. ErrorTracker provides many ways to capture error.

Capture Error using *capture_exception* method *capture_exception* takes another parameter for *additional_context* (dictionary of key value pairs). This parameter can be used to provide additional details about the failure.

```
error_tracker = AppErrorTracker(...)
...
try
    ...
catch Exception as e:
    error_tracker.capture_exception()
```

A simple Message can be captured using *capture_message* method.

```
try
    ...
catch Exception as e:
    error_tracker.capture_message("Something went wrong!")
```

Decorator based exception recording, record exception as it occurs in a method call.

Note: Exception will be re-raised so it must be caught in the caller or ignored. Raised exception can be ignored by passing *silent=True*. Also more context detail can be provided using *additional_context* parameter.

```
@error_tracker.auto_track_exception
def fun():
    pass
```

So far, you have seen only uses where context is provided upfront using default context builder or some other means. Sometimes, we need to put context based on the current code path, like add user_id and email in login flow. Error-Tracker comes with context manager that can be used for such use cases.

```
from error_tracker import flask_scope

with flask_scope() as scope:
    scope.set_extra("user_id", 1234)
    scope.set_extra("email", "example@example.com" )
```

Now error_tracker will automatically capture exception as it will occur. This data will be stored in request_data detail as

```
{
   ...
    "context" : {
        "id" : 1234,
        "email" :  "example@example.com"
    }
}
```

6.2 Django App Settings

Error Tracker fits nicely with Django framework, error tracker can be configured in different ways. Multiple settings are available, these settings can be configured using settings file.

6.2.1 Setting details

- Home page list size, display 10 exceptions per page

```
EXCEPTION_APP_DEFAULT_LIST_SIZE = 10
```

- What all sensitive data should be masked

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

Note: This means any variables whose name have either password or secret would be masked

- Sensitive data masking value

```
APP_ERROR_MASK_WITH = '*************'
```

- Exception email subject prefix

```
APP_ERROR_SUBJECT_PREFIX = get('APP_ERROR_SUBJECT_PREFIX', '')
```

- **Email sender's email id**

```
APP_ERROR_EMAIL_SENDER = "server@example.com"
```

- Whom email should be sent in the case of failure

```
APP_ERROR_RECIPIENT_EMAIL = ('dev-group1@example.com', 'dev@example.com')
```

- By default only 500 errors are tracked but HTTP 404, 401 etc can be tracked as well

```
TRACK_ALL_EXCEPTIONS = True
```

Note: Below configurations are required path to some class.

- Custom Masking Module

```
APP_ERROR_MASKING_MODULE = "path to Masking class"
```

- Ticketing/Bugging module

```
APP_ERROR_TICKETING_MODULE = "path to Ticketing class"
```

Note: Class must not have any constructor arguments

- Notifier module

```
APP_ERROR_NOTIFICATION_MODULE = "path to Notification class"
```

Note: Class must not have any constructor arguments

- Context Builder module

```
APP_ERROR_CONTEXT_BUILDER_MODULE = "path to ContextBuilder class"
```

Note: Class must not have any constructor arguments

- **Custom Model used for exceptions storage**

```
APP_ERROR_DB_MODEL = "path to Model class"
```

Note: Class must implements all abstract methods

- **Exception Listing page permission** By default exception listing is enabled for only admin users.

```
APP_ERROR_VIEW_PERMISSION = 'permission.ErrorViewPermission'
```

Note: Class must not have any constructor parameters and should implement __call__ method.

6.2.2 Manual Exception Tracking

Error can be tracked programmatically using *ErrorTracker*'s utility methods available in error_tracker module. For tracking exception call *capture_exception* method.

```
from error_tracker import capture_exception

...
try
    ...
catch Exception as e:
    capture_exception(request=request, exception=e)
```

A message can be captured using *capture_message* method.

```
from error_tracker import capture_message

try
    ...
catch Exception as e:
    capture_message("Something went wrong", request=request, exception=e)
```

Decorator based exception recording, record exception as it occurs in a method call.

Note: Exception will be re-raised so it must be caught in the caller or ignored. Re-raising of exception can be disabled using *silent=True* parameter

```
from error_tracker import track_exception

@track_exception
def do_something():
    ...
```

So far, you have seen only uses where context is provided upfront using default context builder or some other means. Sometimes, we need to put context based on the current code path, like add user_id and email in login flow. ErrorTracker comes with context manager that can be used for such use cases.

```
from error_tracker import configure_scope

with configure_scope(request=request) as scope:
```

(continues on next page)

(continued from previous page)

```
    scope.set_extra("user_id", 1234)
    scope.set_extra("email", "example@example.com"
```

In this case whenever exception would be raised, it will capture the exception automatically and these context details would be stored as well.

```
{
   ...
    "context" : {
        "id" : 1234,
        "email" :  "example@example.com"
    }
}
```

6.3 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 error_tracker import NotificationMixin
 class Notifier(NotificationMixin):
     def notify(self, request, exception,
                email_subject=None,
                email_body=None,
                from_email=None,
                recipient_list=None):
         # add logic here
```

6.3.1 Flask App Usage

```
error_tracker = AppErrorTracker(app=app, db=db, notifier=Notifier())
```

6.3.2 Django App Usage

settings.py

```
APP_ERROR_NOTIFICATION_MODULE = "path to Notifier class"
```

6.4 Ticketing

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

Using TicketingMixin class

implement raise_ticket method of TicketingMixin interface

```
from error_tracker import TicketingMixin
class Ticketing(TicketingMixin):
    def raise_ticket(self, exception, request=None):
        # Put your logic here
```

6.4.1 Flask App Usage

```
app = Flask(__name__)
db = SQLAlchemy(app)
error_tracker = AppErrorTracker(app=app, db=db, ticketing=Ticketing() )
db.create_all()
```

6.4.2 Django App Usage

settings.py

```
APP_ERROR_TICKETING_MODULE = "path to Ticketing class"
```

6.5 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 = ()`
- 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*, *QueryDict* standard dict or any object whose super class is dict.

Custom masking rule using MaskingMixin

Note: implement __call__ method of MaskingMixin

```
from error_tracker import MaskingMixin
class MyMaskingRule(MaskingMixin):
    def __call__(self, key):
        # Put any logic
```

(continues on next page)

(continued from previous page)

```
        # Do not mask return False, None
        # To mask return True, Value
```

6.5.1 Flask App Usage

```
error_tracker = AppErrorTracker(app=app, db=db,
                                masking=MyMaskingRule("#########", ('pass', 'card')␣
↪) )
```

6.5.2 Django App Usage

settings.py

```
APP_ERROR_MASKING_MODULE="path to MyMaskingRule"
APP_ERROR_MASKED_KEY_HAS = ('pass', 'card')
APP_ERROR_MASKED_WITH = "############"
```

6.6 Custom Context Builder

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. Error Tracker comes with two type of context builders DefaultFlaskContextBuilder and DefaultDjangoContextBuilder for Flask and Django respectively. We can either reuse the same context builder or customize them as per our need.

Using ContextBuilderMixin

Note: Implement get_context method of ContextBuilderMixin, default context builders capture *request body*, *headers* and URL parameters.

```
from error_tracker import ContextBuilderMixin
class ContextBuilder(ContextBuilderMixin):
     def get_context(self, request, masking=None):
          # return context dictionary
```

6.6.1 Flask App Usage

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

```
error_tracker = AppErrorTracker(app=app, db=db,
                                context_builder=ContextBuilder())
```

6.6.2 Django App Usage

Add path of the custom builder class to the settings file, this class should not take any arguments for constructions.

settings.py

```
APP_ERROR_CONTEXT_BUILDER_MODULE = "path to ContextBuilder class"
```

6.7 Using Mongo or other data store

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

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

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

    @classmethod
    def create_or_update_entity(cls, rhash, host, path, method, request_data,␣
↪exception_name, traceback):
        ...

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

    @classmethod
    def get_entity(cls, rhash):
        ...
```

6.7.1 Flask App Usage

Create app with the specific model

```
error_tracker = AppErrorTracker(app=app, model=CustomModel)
```

6.7.2 Django App Usage

Add path to the model in settings file as

```
APP_ERROR_DB_MODEL = core.CustomModel
```