Application Errors

▶ *Changelog*

Applications fail, servers fail. Sooner or later you will see an exception in production. Even if your code is 100% correct, you will still see exceptions from time to time. Why? Because everything else involved will fail. Here are some situations where perfectly fine code can lead to server errors:

- the client terminated the request early and the application was still reading from the incoming data
- the database server was overloaded and could not handle the query
- a filesystem is full
- a harddrive crashed
- a backend server overloaded
- a programming error in a library you are using
- network connection of the server to another system failed

And that's just a small sample of issues you could be facing. So how do we deal with that sort of problem? By default if your application runs in production mode, Flask will display a very simple page for you and log the exception to the **`logger`**.

But there is more you can do, and we will cover some better setups to deal with errors.

Error Logging Tools

Sending error mails, even if just for critical ones, can become overwhelming if enough users are hitting the error and log files are typically never looked at. This is why we recommend using Sentry for dealing with application errors. It's available as an Open Source project on GitHub and is also available as a hosted version which you can try for free. Sentry aggregates duplicate errors, captures the full stack trace and local variables for debugging, and sends you mails based on new errors or frequency thresholds.

To use Sentry you need to install the *sentry-sdk* client with extra *flask* dependencies:

```
$ pip install sentry-sdk[flask]
```

And then add this to your Flask app:

```
import sentry_sdk
from sentry_sdk.integrations.flask import FlaskIntegration

sentry_sdk.init('YOUR_DSN_HERE',integrations=[FlaskIntegration()])
```

The *YOUR_DSN_HERE* value needs to be replaced with the DSN value you get from your Sentry installation.

After installation, failures leading to an Internal Server Error are automatically reported to Sentry and from there you can receive error notifications.

Follow-up reads:

- Sentry also supports catching errors from your worker queue (RQ, Celery) in a similar fashion. See the Python SDK docs for more information.
- Getting started with Sentry
- Flask-specific documentation.

Error handlers

You might want to show custom error pages to the user when an error occurs. This can be done by registering error handlers.

An error handler is a normal view function that returns a response, but instead of being registered for a route, it is registered for an exception or HTTP status code that would be raised while trying to handle a request.

Registering

Register handlers by decorating a function with **`errorhandler()`**. Or use **`register_error_handler()`** to register the function later. Remember to set the error code when returning the response.

```
@app.errorhandler(werkzeug.exceptions.BadRequest)
def handle_bad_request(e):
    return 'bad request!', 400

# or, without the decorator
app.register_error_handler(400, handle_bad_request)
```

`werkzeug.exceptions.HTTPException` subclasses like **`BadRequest`** and their HTTP codes are interchangeable when registering handlers. (`BadRequest.code == 400`)

Non-standard HTTP codes cannot be registered by code because they are not known by Werkzeug. Instead, define a subclass of **`HTTPException`** with the appropriate code and register and raise that exception class.

```
class InsufficientStorage(werkzeug.exceptions.HTTPException):
    code = 507
    description = 'Not enough storage space.'

app.register_error_handler(InsufficientStorage, handle_507)

raise InsufficientStorage()
```

Handlers can be registered for any exception class, not just **`HTTPException`** subclasses or HTTP status codes. Handlers can be registered for a specific class, or for all subclasses of a parent class.

Handling

When an exception is caught by Flask while handling a request, it is first looked up by code. If no handler is registered for the code, it is looked up by its class hierarchy; the most specific handler is chosen. If no handler is registered, **`HTTPException`** subclasses show a generic message about their code, while other exceptions are converted to a generic 500 Internal Server Error.

For example, if an instance of **`ConnectionRefusedError`** is raised, and a handler is registered for **`ConnectionError`** and **`ConnectionRefusedError`**, the more specific **`ConnectionRefusedError`** handler is called with the exception instance to generate the response.

Handlers registered on the blueprint take precedence over those registered globally on the application, assuming a blueprint is handling the request that raises the exception. However, the blueprint cannot handle 404 routing errors because the 404 occurs at the routing level before the blueprint can be determined.

Generic Exception Handlers

It is possible to register error handlers for very generic base classes such as `HTTPException` or even `Exception`. However, be aware that these will catch more than you might expect.

An error handler for `HTTPException` might be useful for turning the default HTML errors pages into JSON, for example. However, this handler will trigger for things you don't cause directly, such as 404 and 405 errors during routing. Be sure to craft your handler carefully so you don't lose information about the HTTP error.

```
from flask import json
from werkzeug.exceptions import HTTPException
```

```
@app.errorhandler(HTTPException)
def handle_exception(e):
    """Return JSON instead of HTML for HTTP errors."""
    # start with the correct headers and status code from the error
    response = e.get_response()
    # replace the body with JSON
    response.data = json.dumps({
        "code": e.code,
        "name": e.name,
        "description": e.description,
    })
    response.content_type = "application/json"
    return response
```

An error handler for `Exception` might seem useful for changing how all errors, even unhandled ones, are presented to the user. However, this is similar to doing `except Exception:` in Python, it will capture *all* otherwise unhandled errors, including all HTTP status codes. In most cases it will be safer to register handlers for more specific exceptions. Since `HTTPException` instances are valid WSGI responses, you could also pass them through directly.

```
from werkzeug.exceptions import HTTPException

@app.errorhandler(Exception)
def handle_exception(e):
    # pass through HTTP errors
    if isinstance(e, HTTPException):
        return e

    # now you're handling non-HTTP exceptions only
    return render_template("500_generic.html", e=e), 500
```

Error handlers still respect the exception class hierarchy. If you register handlers for both `HTTPException` and `Exception`, the `Exception` handler will not handle `HTTPException` subclasses because it the `HTTPException` handler is more specific.

Unhandled Exceptions

When there is no error handler registered for an exception, a 500 Internal Server Error will be returned instead. See **`flask.Flask.handle_exception()`** for information about this behavior.

If there is an error handler registered for `InternalServerError`, this will be invoked. As of Flask 1.1.0, this error handler will always be passed an instance of

`InternalServerError`, not the original unhandled error. The original error is available as `e.original_error`. Until Werkzeug 1.0.0, this attribute will only exist during unhandled errors, use `getattr` to get access it for compatibility.

```
@app.errorhandler(InternalServerError)
def handle_500(e):
    original = getattr(e, "original_exception", None)

    if original is None:
        # direct 500 error, such as abort(500)
        return render_template("500.html"), 500

    # wrapped unhandled error
    return render_template("500_unhandled.html", e=original), 500
```

Logging

See Logging for information on how to log exceptions, such as by emailing them to admins.

Debugging Application Errors

For production applications, configure your application with logging and notifications as described in Application Errors. This section provides pointers when debugging deployment configuration and digging deeper with a full-featured Python debugger.

When in Doubt, Run Manually

Having problems getting your application configured for production? If you have shell access to your host, verify that you can run your application manually from the shell in the deployment environment. Be sure to run under the same user account as the configured deployment to troubleshoot permission issues. You can use Flask's builtin development server with *debug=True* on your production host, which is helpful in catching configuration issues, but **be sure to do this temporarily in a controlled environment.** Do not run in production with *debug=True*.

Working with Debuggers

To dig deeper, possibly to trace code execution, Flask provides a debugger out of the box (see Debug Mode). If you would like to use another Python debugger, note that debuggers interfere with each other. You have to set some options in order to use your favorite debugger:

- `debug` - whether to enable debug mode and catch exceptions
- `use_debugger` - whether to use the internal Flask debugger
- `use_reloader` - whether to reload and fork the process if modules were changed

`debug` must be True (i.e., exceptions must be caught) in order for the other two options to have any value.

If you're using Aptana/Eclipse for debugging you'll need to set both `use_debugger` and `use_reloader` to False.

A possible useful pattern for configuration is to set the following in your config.yaml (change the block as appropriate for your application, of course):

```
FLASK:
    DEBUG: True
    DEBUG_WITH_APTANA: True
```

Then in your application's entry-point (main.py), you could have something like:

```
if __name__ == "__main__":
    # To allow aptana to receive errors, set use_debugger=False
    app = create_app(config="config.yaml")

    use_debugger = app.debug and not(app.config.get('DEBUG_WITH_APTANA'
    app.run(use_debugger=use_debugger, debug=app.debug,
            use_reloader=use_debugger, host='0.0.0.0')
```