Error Handling & Debugging with Flask

Topics covered

- Flask debugger, common error types, production safety
- best practices for logging and debugging
- **Flask project** shows the following:
  - how to enable/disable debugging safely
  - how to log errors
  - how to create **custom error pages** (400, 403, 404, 500)
  - how to write a global exception handler
  - how to raise and simulate errors and test them
- example curl/browser steps to test

Important security note:

- **Never enable the interactive debugger (Werkzeug) or debug=True in production.** It gives remote users an interactive Python shell if exploited.
- Use logging + error reporting (Sentry, etc.) in production.

1 — Error Handling & Debugging with Flask

Flask debugger

- Flask uses the **Werkzeug debugger** when the app is run with debug=True or FLASK_ENV=development. It shows interactive tracebacks in the browser when an unhandled exception occurs.
- The debugger is great for local development but **dangerous in production** because it can allow remote code execution.

Common HTTP error types

- **400 Bad Request** — client sent malformed request (e.g., missing JSON).
- **401 Unauthorized** — authentication required.
- **403 Forbidden** — authenticated but not allowed.
- **404 Not Found** — requested resource doesn't exist.
- **405 Method Not Allowed** — wrong HTTP method used.
- **500 Internal Server Error** — unexpected server error (exception on server side).

Handling errors in Flask

- Use abort(status_code) to return an HTTP error.
- Use @app.errorhandler(code) or @app.errorhandler(ExceptionClass) to register custom error handlers.
- For generic exceptions, handle Exception or subclass to log and show a user-friendly message.
- Use app.logger or standard logging to persist errors to console/files.

Custom error pages

- Create HTML templates like 404.html, 500.html.
- Use error handlers to render those templates and appropriate status codes.

Debugging best practices

- Reproduce locally with debug=True (but not in prod).
- Add rich logs (timestamp, level, message, traceback).
- Use Flask's app.test_client() or pytest to write automated tests for error scenarios.
- Use remote error monitoring (Sentry, Rollbar) in production.
- Fail fast: validate inputs and raise explicit errors with helpful messages.

2 — Project: Files & Folder Structure

```
flask_error_debug_demo/
|
 ├── app.py
 ├── requirements.txt
 ├── logs/
|   └── app.log          # created at runtime by logger
└── templates/
    ├── base.html
    ├── index.html
    ├── cause_error.html
    ├── 400.html
    ├── 403.html
    ├── 404.html
   └── 500.html
```

- app.py — main Flask program (below).
- templates/ — HTML templates for UI and custom error pages.
- logs/app.log — created by program; stores rotating logs.

3 — Full program: app.py

Save this as app.py in the project root.

app.py

```
            import logging
import traceback
from logging.handlers import RotatingFileHandler
from flask import Flask, render_template, request, jsonify, abort

app = Flask(__name__)

# -----------------------
# Logging setup
# -----------------------
LOG_FILE = "logs/app.log"
# Ensure logs directory exists (simple)
import os
os.makedirs(os.path.dirname(LOG_FILE), exist_ok=True)

formatter = logging.Formatter(
    "%(asctime)s | %(levelname)s | %(module)s:%(lineno)d | %(message)s"
)

file_handler = RotatingFileHandler(LOG_FILE, maxBytes=5*1024*1024, backupCount=3)
file_handler.setLevel(logging.INFO)
file_handler.setFormatter(formatter)

stream_handler = logging.StreamHandler()
```

```
stream_handler.setLevel(logging.DEBUG)
stream_handler.setFormatter(formatter)

app.logger.setLevel(logging.DEBUG)  # set app logger level
app.logger.addHandler(file_handler)
app.logger.addHandler(stream_handler)

# -----------------------
# Simple routes
# -----------------------
@app.route("/")
def index():
    return render_template("index.html")

@app.route("/cause-error")
def cause_error_page():
    # page with a button to simulate an error
    return render_template("cause_error.html")

@app.route("/api/divide", methods=["GET"])
def api_divide():
    """
    Example endpoint to demonstrate error handling.
    Query params: a, b
    Example: /api/divide?a=10&b=0  -> triggers ZeroDivisionError
    """
    a = request.args.get("a")
    b = request.args.get("b")
    # Input validation
    try:
        if a is None or b is None:
            # Missing params -> 400
            abort(400, description="Query params 'a' and 'b' are required (e.g.
/api/divide?a=10&b=2)")
        a = float(a)
        b = float(b)
    except ValueError:
        abort(400, description="Query params must be numbers.")
    # This can raise ZeroDivisionError
    result = a / b
    return jsonify({"a": a, "b": b, "result": result})

# -----------------------
# Custom error handlers
# -----------------------
@app.errorhandler(400)
def bad_request(e):
    # e may be HTTPException with description
```

```
    app.logger.warning("400 Bad Request: %s", getattr(e, "description", str(e)))
    # If client expects JSON, return JSON
    if request.accept_mimetypes.accept_json and not request.accept_mimetypes.accept_html:
        return jsonify({"error": "Bad Request", "message": getattr(e, "description", str(e))}), 400
    # Otherwise render an HTML page
    return render_template("400.html", message=getattr(e, "description", None)), 400

@app.errorhandler(403)
def forbidden(e):
    app.logger.warning("403 Forbidden: %s", getattr(e, "description", str(e)))
    if request.accept_mimetypes.accept_json and not request.accept_mimetypes.accept_html:
        return jsonify({"error": "Forbidden", "message": getattr(e, "description", str(e))}), 403
    return render_template("403.html", message=getattr(e, "description", None)), 403

@app.errorhandler(404)
def not_found(e):
    app.logger.info("404 Not Found: %s %s", request.method, request.path)
    if request.accept_mimetypes.accept_json and not request.accept_mimetypes.accept_html:
        return jsonify({"error": "Not Found", "message": "The requested URL was not found on the
server."}), 404
    return render_template("404.html", path=request.path), 404

@app.errorhandler(500)
def internal_error(e):
    # Log exception traceback
    tb = traceback.format_exc()
    app.logger.error("500 Internal Server Error: %s\nTraceback:\n%s", e, tb)
    # For API clients prefer JSON, for browsers show friendly HTML
    if request.accept_mimetypes.accept_json and not request.accept_mimetypes.accept_html:
        return jsonify({"error": "Internal Server Error", "message": "An unexpected error occurred."}),
500
    return render_template("500.html"), 500

# Generic catch-all for unhandled exceptions (optional)
@app.errorhandler(Exception)
def handle_exception(e):
    """
    This catches unhandled exceptions and returns a 500.
    Note: HTTPExceptions like 404 are subclasses of Exception, so those are handled above first.
    """
    # If it's an HTTPException, let the specific handler run
    from werkzeug.exceptions import HTTPException
    if isinstance(e, HTTPException):
        return e

    # Log full traceback
    app.logger.exception("Unhandled exception: %s", e)
```

```
    # Return 500 to clients
    if request.accept_mimetypes.accept_json and not request.accept_mimetypes.accept_html:
        return jsonify({"error": "Internal Server Error", "message": "An unexpected error occurred."}),
500
    return render_template("500.html"), 500

# -----------------------
# Demo: raise custom error
# -----------------------
class InvalidUsage(Exception):
    """Custom exception we can raise with a status code."""
    def __init__(self, message, status_code=400, payload=None):
        super().__init__(message)
        self.message = message
        self.status_code = status_code
        self.payload = payload

@app.errorhandler(InvalidUsage)
def handle_invalid_usage(error):
    app.logger.info("InvalidUsage: %s", error.message)
    response = {"error": "InvalidUsage", "message": error.message}
    if error.payload:
        response.update(error.payload)
    return jsonify(response), error.status_code

@app.route("/api/custom-error")
def api_custom_error():
    # Example: raise a custom error that becomes JSON for API clients
    raise InvalidUsage("This is a custom error example", status_code=422, payload={"hint": "use
different data"})

# -----------------------
# Run the app
# -----------------------
if __name__ == "__main__":
    # NOTE: debug=True enables the Werkzeug interactive debugger (do not use in production)
    app.run(debug=True)
```

4 — Templates

Create templates/ directory and the following files.

templates/base.html

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Flask Error & Debug Demo</title>
  <style>
```

```
    body { font-family: Arial, sans-serif; max-width: 900px; margin: 24px auto; }
    header { margin-bottom: 16px; }
    .card { border:1px solid #ddd; padding:12px; border-radius:6px; margin-bottom:12px; }
    .muted { color:#666; font-size:0.9em; }
  </style>
</head>
<body>
  <header>
    <h1>Flask Error & Debug Demo</h1>
    <p class="muted">Use this app to learn how to debug and display user-friendly errors.</p>
    <nav>
      <a href="{{ url_for('index') }}">Home</a> |
      <a href="{{ url_for('cause_error_page') }}">Cause Error</a> |
      <a href="{{ url_for('api_divide') }}?a=10&b=2">API: divide</a> |
      <a href="{{ url_for('api_custom_error') }}">API: custom error</a>
    </nav>
    <hr>
  </header>

  {% block content %}{% endblock %}
</body>
</html>
```

templates/index.html

```
{% extends "base.html" %}
{% block content %}
<div class="card">
  <h2>Welcome</h2>
  <p>Try the endpoints listed in the nav. Use <code>/api/divide?a=10&b=2</code> and try dividing by zero to see an error.</p>
</div>
{% endblock %}
```

templates/cause_error.html

```
{% extends "base.html" %}
{% block content %}
<div class="card">
  <h2>Cause an Error</h2>
  <p>Click the button below to call <code>/api/divide</code> with <code>b=0</code> and trigger a <code>ZeroDivisionError</code>.</p>
  <p><a href="{{ url_for('api_divide') }}?a=10&b=0">Cause divide-by-zero error (API)</a></p>
</div>
{% endblock %}
```

templates/400.html

```
{% extends "base.html" %}
{% block content %}
<div class="card">
  <h2>400 — Bad Request</h2>
  {% if message %}
    <p>{{ message }}</p>
```

```
  {% else %}
    <p>The server could not understand your request.</p>
  {% endif %}
</div>
{% endblock %}
```

templates/403.html

```
{% extends "base.html" %}
{% block content %}
<div class="card">
  <h2>403 — Forbidden</h2>
  <p>You do not have permission to access this resource.</p>
</div>
{% endblock %}
```

templates/404.html

```
{% extends "base.html" %}
{% block content %}
<div class="card">
  <h2>404 — Not Found</h2>
  <p>The URL <code>{{ path }}</code> was not found on this server.</p>
</div>
{% endblock %}
```

templates/500.html

```
{% extends "base.html" %}
{% block content %}
<div class="card">
  <h2>500 — Server Error</h2>
  <p>Sorry — something went wrong on our side. The incident has been logged.</p>
  <p class="muted">If you are the developer, check the logs (logs/app.log) or run the app with debug=True locally.</p>
</div>
{% endblock %}
```

5 — How to run & test (step-by-step)

1. Create virtualenv (recommended) and install Flask:

```
python -m venv venv
# activate
# mac / linux:
source venv/bin/activate
# windows:
venv\Scripts\activate
```

pip install Flask

2. Create folder structure and files above (templates, logs directory will be created automatically).
3. Run:

python app.py

You will see Running on http://127.0.0.1:5000/ and logs in console.

4. Test endpoints in browser or with curl:
   - Normal call:
     - http://127.0.0.1:5000/api/divide?a=10&b=2 → returns JSON {"a":10,"b":2,"result":5}
   - Cause divide-by-zero error (demonstrates debug page when debug=True):
     - http://127.0.0.1:5000/api/divide?a=10&b=0
       - With debug=True you'll see Werkzeug interactive traceback (local only).
       - With debug=False you'll see the custom 500 page and the stacktrace will be written to logs/app.log.
   - Custom API error:
     - http://127.0.0.1:5000/api/custom-error → returns JSON {"error":"InvalidUsage","message":"..."} with status 422.
   - Non-existent URL:
     - http://127.0.0.1:5000/does-not-exist → custom 404 page.

Examples using curl:

```
# good call
curl "http://127.0.0.1:5000/api/divide?a=10&b=2"

# bad input -> 400
curl -i "http://127.0.0.1:5000/api/divide?a=ten&b=2"

# divide by zero -> 500
curl -i "http://127.0.0.1:5000/api/divide?a=10&b=0"

# custom error -> 422
curl -i "http://127.0.0.1:5000/api/custom-error"
```

5. Inspect logs: open logs/app.log to see error tracebacks, timestamps and request info.

6 — Examples & exercises

1. **Exercise:** Modify api_divide to return a friendly JSON error when dividing by zero instead of a 500. (Hint: catch ZeroDivisionError and abort(400) or return JSON with message.)
2. **Exercise:** Add an endpoint that accepts JSON body and validates required keys; use abort(400, description="...") for missing fields.
3. **Exercise:** Replace RotatingFileHandler with SMTPHandler to email critical errors (requires SMTP server).
4. **Homework:** Integrate Sentry (or equivalent) for production error capture; ensure PII is not sent.

7 — Quick checklist:

- How to **enable/disable** Flask debug mode (and why it's dangerous in production).
- How to use abort() to trigger HTTP errors.
- How to implement @app.errorhandler() for 400/403/404/500 and custom exceptions.
- How to log exceptions with app.logger.exception() and save them to files.
- How to produce **user-friendly** error pages and **API-friendly** JSON errors based on Accept headers.
- How to test error flows with curl/Postman and check logs/app.log.

Student activity:

- produce a **short slide deck** summarizing these concepts (one-slide per concept),
- provide **unit tests** (pytest) that assert the handlers return correct status codes and messages,
- or give a **smaller demo** that only focuses on logging configuration.