7 Tratamento de erros

Resumo: *Precisamos dar uma pausa na codificação de novos recursos do aplicativo e discutir algumas estratégias para lidar com bugs, que invariavelmente aparecem em todos os projetos de software. Para ajudar a ilustrar este tópico, vamos inserir um bug intencionalmente no código!*

Tratamento de erros em um aplicativo Flask

O que acontece quando ocorre um erro em um aplicativo `Flask`? A melhor maneira de descobrir é experimentá-lo em primeira mão. Vá em frente e inicie o aplicativo, e certifique-se de ter pelo menos dois usuários registrados. Faça login como um dos usuários, abra a página de perfil e clique no *link* "Editar". No editor de perfil, tente alterar o nome de usuário para o nome de usuário de outro usuário que já esteja registrado, e bum! Isso vai trazer uma página assustadora de **Erro Interno do Servidor**.

Se você olhar na sessão de terminal onde o aplicativo está sendo executado, verá um **rastreamento de pilha do erro**. Rastreamentos de pilha são extremamente úteis na depuração de erros, porque mostram a sequência de chamadas naquela pilha, até a linha que produziu o erro:

```
([2025-04-01 23:59:42,300] ERROR in app: Exception on /edit_profile [POST]
Traceback (most recent call last):
  File "venv/lib/python3.11/site-packages/sqlalchemy/engine/base.py", line ↩
      1963, in _exec_single_context
    self.dialect.do_execute(
  File "venv/lib/python3.11/site-packages/sqlalchemy/engine/default.py", line ↩
      918, in do_execute
    cursor.execute(statement, parameters)
sqlite3.IntegrityError: UNIQUE constraint failed: user.username
```

O rastreamento de pilha ajuda a determinar qual é o **bug**. O aplicativo permite que um usuário altere o nome de usuário, mas não valida se o novo nome de usuário escolhido não colide com outro usuário já no sistema. O erro vem do `SQLAlchemy`, que tenta gravar o novo nome de usuário no banco de dados, mas o banco de dados o rejeita porque a coluna `username` é definida com a opção `unique = True`.

É importante observar que a página de erro apresentada ao usuário não fornece muitas informações sobre o erro, e isso é bom. Definitivamente, não quero que os usuários saibam que a falha foi

causada por um erro de banco de dados, ou qual banco de dados estou usando, ou quais são alguns dos nomes de tabelas e campos no meu banco de dados. Todas essas informações devem ser mantidas internamente.

Mas há algumas coisas que estão longe do ideal. Temos uma página de erro que é muito feia e não corresponde ao layout do aplicativo. Também temos rastreamentos de pilha de aplicativos importantes sendo despejados em um terminal que precisamos observar constantemente para ter certeza de que não perco nenhum erro. E, claro, temos um *bug* para corrigir. Vamos abordar todas essas questões, mas primeiro vamos falar sobre o modo de depuração do `Flask`.

Modo Debug

A maneira como os erros são tratados acima representa a forma mais comum quando o nosso sistema já está sendo executado em um servidor de produção. Se houver um erro, o usuário obtém uma página de erro vaga (embora precisemos tornar essa página de erro mais agradável), e os detalhes importantes do erro estão na saída do processo do servidor ou em um arquivo de log.

Porém, quando estamos desenvolvendo o nosso aplicativo, o ideal é que habilitemos o modo de depuração, um modo no qual o `Flask` gera um depurador realmente bom, diretamente no navegador web. Para ativar o modo de depuração, pare o aplicativo e defina a seguinte variável de ambiente:

```
(venv) $ export FLASK_DEBUG=1
```

caso o sistema operacional que esteja utilizando seja Linux ou MacIOs. Se for Windows, execute:

```
(venv) $ $ENV: FLASK_DEBUG=1
```

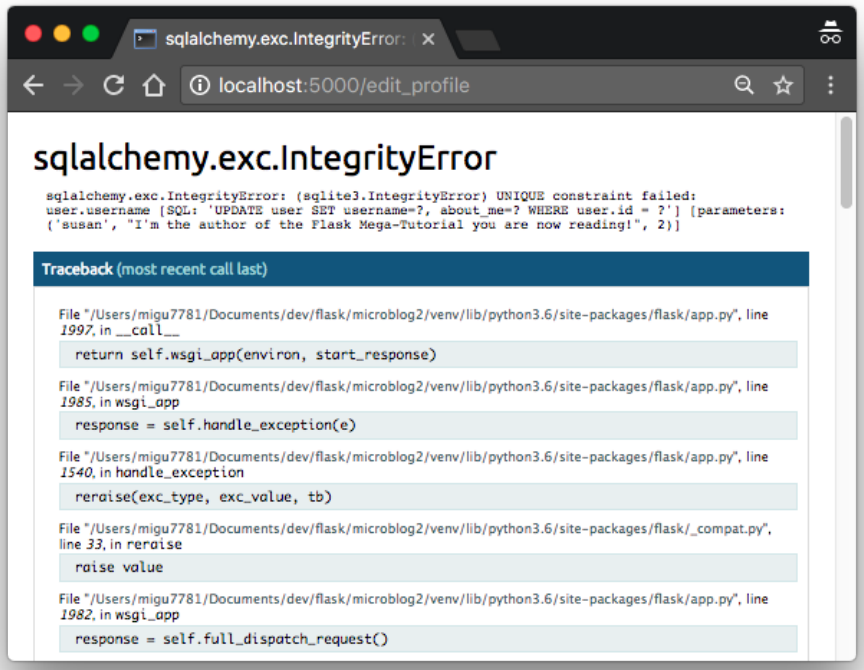
Após definir `FLASK_DEBUG`, reinicie o servidor. A saída no seu terminal será um pouco diferente do que você está acostumado a ver:

```
(venv) $ flask run
 * Serving Flask app 'app.py' (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 118-204-854
```

Agora faça o aplicativo travar mais uma vez para ver o depurador interativo no seu navegador:

O depurador permite que você expanda cada quadro de pilha e veja o código-fonte correspondente. Você também pode abrir um `prompt Python` em qualquer um dos quadros e executar quaisquer expressões `Python` válidas, por exemplo, para verificar os valores das variáveis.

É extremamente importante que você nunca execute um aplicativo Flask no modo de depuração em um servidor de produção. O depurador permite que o usuário execute remotamente o código no servidor, então pode ser um presente inesperado para um usuário malicioso que queira se infiltrar em seu aplicativo ou servidor. Como uma medida de segurança adicional, o depurador em execução no navegador inicia bloqueado e, no primeiro uso, ele solicitará um PIN, que você pode ver na saída do comando `flask run`.



Já que estamos no tópico do modo de depuração, é importante mencionar o segundo recurso que está habilitado com o modo de depuração, que é o recarregador. Este é um recurso de desenvolvimento muito útil que reinicia automaticamente o aplicativo quando um arquivo de origem é modificado. Se você executar `flask run` no modo de depuração, poderá trabalhar no seu aplicativo e, sempre que salvar um arquivo, o aplicativo será reiniciado para pegar o novo código.

Páginas de erro personalizadas

O `Flask` fornece um mecanismo para um aplicativo instalar suas próprias páginas de erro, para que seus usuários não precisem ver as páginas padrão, simples e chatas. Como exemplo, vamos definir páginas de erro personalizadas para os erros `HTTP 404` e `500`, os dois mais comuns. Definir páginas para outros erros funciona da mesma forma.

Para declarar um manipulador de erro personalizado, o decorador `@errorhandler` é usado. Vamos colocar nossos manipuladores de erro em um novo módulo `app/errors.py`.

```
#   app/errors.py

from flask import render_template
from app import app, db

@app.errorhandler(404)
def not_found_error(error):
```

```
    return render_template('404.html'), 404

@app.errorhandler(500)
def internal_error(error):
    db.session.rollback()
    return render_template('500.html'), 500
```

As funções de erro funcionam de forma muito semelhante às funções de exibição. Para esses dois erros, estou retornando o conteúdo de seus respectivos modelos. Observe que ambas as funções retornam um segundo valor após o modelo, que é o número do código de erro. Para todas as funções de exibição que criamos até agora, não precisamos adicionar um segundo valor de retorno porque o padrão de 200 (o código de status para uma resposta bem-sucedida) é o que queríamos. Neste caso, essas são páginas de erro, então queremos que o código de status da resposta reflita isso.

O manipulador de erros para os erros 500 poderia ser invocado após um erro de banco de dados, o que realmente foi o caso com o nome de usuário duplicado acima. Para garantir que nenhuma sessão de banco de dados com falha interfira em nenhum acesso ao banco de dados acionado pelo modelo, emitimos uma reversão de sessão. Isso redefine a sessão para um estado limpo.

Aqui está o modelo para o erro 404:

```
<!-- app/templates/404.html -->


{% extends "base.html" %}

{% block content %}
    <h1>File Not Found</h1>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

E aqui está o erro 500:

```
<!-- app/templates/500.html -->


{% extends "base.html" %}

{% block content %}
    <h1>An unexpected error has occurred</h1>
    <p>The administrator has been notified. Sorry for the inconvenience!</p>
    <p><a href="{{ url_for('index') }}">Back</a></p>
{% endblock %}
```

Ambos os modelos herdam do modelo `base.html`, para que a página de erro tenha a mesma aparência e comportamento das páginas normais do aplicativo.

Para registrar esses manipuladores de erro com o `Flask`, precisamos importar o novo módulo `app/errors.py` após a criação da instância do aplicativo:

```
#   app/__init__.py
```

```
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from flask_login import LoginManager
from config import Config

app = Flask(__name__)
app.config.from_object(Config)
db = SQLAlchemy(app)
migrate = Migrate(app, db)
login = LoginManager(app)
login.login_view = 'login'

from app import routes, models, erros
```

Se definirmos `FLASK_DEBUG=0` na sessão de terminal (ou excluir a variável `FLASK_DEBUG`) e, em seguida, acionar o *bug* de nome de usuário duplicado mais uma vez, verá uma página de erro um pouco mais amigável.