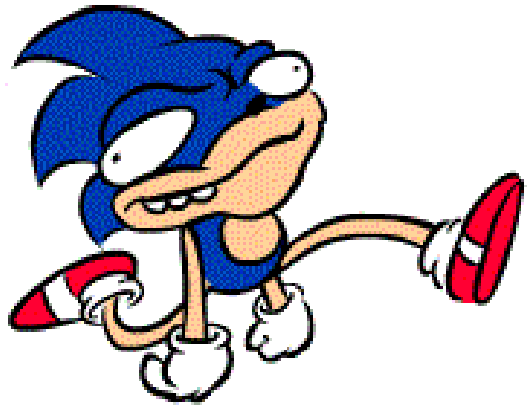
Sanic 18.12 LTS

Novos recursos e websockets

Danilo J. S. Bellini
@danilobellini



2019-03-09

GruPy-ABC @ Fatec Mauá – SP

Isto é uma continuação!

Estes *slides* são uma continuação do apresentado na Flask Conf 2018 em 2018-08-25, utilizando o Sanic 18.12.0 ao invés do 0.7.0 (versão estável que estava no PyPI na época).

Estes slides dizem respeito ao versionamento do projeto, às novidades de sua versão 18.12.0, e aos uso de WebSockets com o framework. Para uma introdução ao Sanic é recomendável ver os já citados slides da Flask Conf.

Todos os *slides* podem ser obtidos em `https://github.com/danilobellini/slides-latex`.

Versões do Sanic

2016-05-25	Primeiro commit (#60f1004)
2016-10-01	0.0.1 (#afeed3c)
2016-10-06	0.0.2 (#74b0cba)
2016-10-14	0.1.0 (#5a7447e)
2016-10-16	0.1.2 (Primeiro release com *tag*)

2016-10-16	0.1.3	2017-01-27	0.3.0	2017-05-08	0.5.4
2016-10-18	0.1.4	2017-02-08	0.3.1	2017-08-02	0.6.0
2016-10-23	0.1.5	2017-02-25	0.4.0	2017-12-05	0.7.0
2016-10-25	0.1.6	2017-02-28	0.4.1	2018-08-17	0.8.0
2016-10-25	0.1.7	2017-04-11	0.5.0	2018-09-06	0.8.1
2016-11-29	0.1.8	2017-04-14	0.5.1	2018-09-13	0.8.2
2016-12-24	0.1.9	2017-04-24	0.5.2	2018-09-13	0.8.3
2017-01-14	0.2.0	2017-05-05	0.5.3	2018-12-27	18.12.0

Novidades da versão 0.8

O que mudou (além de testes/documentação/*bug fixes*/detalhes)?

- Status passou de *Pre-Alpha* para *Beta*
- Auto-reload
- Grupos e aninhamentos de blueprints
- Rotas com UUID
- Novo método `register_listener` como alternativa ao decorator
- Diversas melhorias no tratamento de websockets e streaming
- Várias internalidades e melhor cobertura de especificações (sanitização, cabeçalho HTTP, etc.)

A partir da v0.8.3, o repositório foi movido para o projeto `https://github.com/huge-success`.

Novidades da versão 18.12 LTS

O que mudou (além de testes/documentação/*bug fixes*/detalhes)?

- Novo sistema de versionamento (calendário ao invés de semântico)
- Cancelamento de tarefa no `connection_lost`
- Novo `stream_large_files` no `Sanic.static`
- Mudanças internas relativas à identificação do IP e porta de requisições, e ao tratamento de erros em arquivos de configuração
- Criação de métodos `body_init`, `body_push` e `body_finish` permitindo a customização da classe `Request`
- Logging "principal" em `sanic.root` ao invés do logger raiz, `Handler.log` tornou-se obsoleto

UUID – Identificador único universal

As rotas podem ter agora uuid como tipo de dado:

```
from sanic import Sanic, response

app = Sanic(__name__)

@app.route("/<uid:uuid>")
def root(request, uid):
    return response.json({"user": uid.hex})
```

WebSockets

Veremos agora um exemplo de WebSockets. O próximo exemplo consiste em uma simples sala de bate-papo, em que a primeira mensagem enviada por um usuário é seu nome/apelido, e as mensagens seguintes são os conteúdos.

Uma possível estrutura para uso de websockets é:

```
@app.websocket("/somewhere")
async def websocket_route(request, ws):
    try: # Conexão aberta!
        while True:
            msg = await ws.recv() # Mensagem recebida
            await ws.send(msg) # Envio de mensagem
    finally:
        pass # Conexão encerrada!
```

O objeto ws representa a conexão, podendo ser utilizado dentro das rotinas de tratamento de outras requisições para realizar uma comunicação iniciada pelo servidor.

WebSockets – ws_server.py

```python
import asyncio
from sanic import Sanic

app = Sanic(__name__)
app.static("/", "ws_client.html")
connections = set()

async def broadcast(msg):
    print(msg)
    for ws in connections:
        await ws.send(msg)

@app.websocket("/ws")
async def websocket_route(request, ws):
    name = None
    connections.add(ws)
    try:
        name = await ws.recv()
        await broadcast(f"New user: {name}")
        while True:
            message = await ws.recv()
            await broadcast(f"[{name}]: {message}")
    finally:
        connections.remove(ws)
        if name is not None:
            await broadcast(f"{name} left the chat")
```

WebSockets – ws_client.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF8" />
    <script>
      document.addEventListener("DOMContentLoaded", function(){
        var ws = new WebSocket(location.href.replace(/^http/, "ws")
                                            .replace(/\/?$/, "/ws"));
        var button = document.getElementById("btn");
        var body = document.getElementsByTagName("body")[0];
        button.addEventListener("click", function(){
          ws.send(document.getElementById("txt").value);
        })
        ws.onmessage = function(evt){
          console.log(evt.data);
          body.insertAdjacentHTML("beforeend", `<div>${evt.data}</div>`);
        }
      });
    </script>
  </head>
  <body>
    <input id="txt">
    <button id="btn">Send</button>
  </body>
</html>
```

Auto-reload

Para desenvolvimento, é possível aproveitar do recurso de auto-reload, fazendo com que o código seja re-executado caso modificado, sem interromper esse monitoramento mesmo que o código tenha algum erro. Para isso, basta fazer o código do servidor ser um script, com:

```
from sanic.websocket import WebSocketProtocol
# [... Colocar aqui o conteúdo do ws_server.py ...]
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000,
            protocol=WebSocketProtocol,
            auto_reload=True)
```

Para iniciar o `ws_server.py` diretamente com essa mesma configuração mas sem auto-reload, independente de ter o fragmento acima ou de ser um script:

```
python -m sanic --host 0.0.0.0 --port 8000 ws_server.app
```

UNIX Socket

É possível usar o Sanic de outras formas, por exemplo:

```
#!/usr/bin/env python3
import socket, sys, os
from sanic import Sanic, response

app = Sanic(__name__)
server_socket = "/tmp/sanic.sock"
sock = socket.socket(socket.AF_UNIX,
                     socket.SOCK_STREAM)
sock.bind(server_socket)
app.count = 0

@app.route("/")
async def counter(request):
    app.count += 1
    return response.text(app.count)

def signal_handler(sig, frame):
    print("Exiting")
    os.unlink(server_socket)
    sys.exit(0)

if __name__ == "__main__":
    app.run(sock=sock)
```

Esse é um exemplo simples com um contador global. Para verificar o resultado, pode-se usar:

```
curl -s --unix-socket                                    ...
↪ /tmp/sanic.sock http://localhost/ |                    ...
↪ xargs
```

Embora não seja realmente uma novidade, a documentação do Sanic foi atualizada para explicitar que ele pode ser utilizado nesse tipo de situação.

Download via streaming de arquivo estático

Este código serve um arquivo `sanic.mp4`:

```
from sanic import Sanic, response

app = Sanic(__name__)
app.static("/video", "sanic.mp4", stream_large_files=True)
app.static("/video_not_chunked", "sanic.mp4") # Should not be used!

@app.route("/")
def root(request):
    return response.html("""<html><body><video controls>
        <source src="/video"><a href="/video">Video</a>
    </video></body></html>""")
```

A rota `/video_not_chunked` pode ser utilizada para enviar o arquivo de vídeo inteiro de uma só vez, em contraste com o picotamento da rota `/video`. Para verificar o número de "pacotes"TCP enviados, pode-se utilizar:

```
sudo tcpdump --interface=any port 8000 and '(tcp-syn|tcp-ack)!=0'
```

Referências

- https://youtu.be/of9gIoQpPmA — Origem do sonic.mp4 utilizado no exemplo
- https://github.com/huge-success/sanic
- https://sanic.readthedocs.io
- https://sanicframework.org/

FIM!