Actualización sobre hilos, procesos, red y seguridad en Python, utilizando las librerías y prácticas actuales

Últimas Novedades sobre Hilos, Procesos, Red y Seguridad en Python

Hilos y Procesos

Hilos con concurrent.futures

El módulo concurrent.futures proporciona una interfaz de alto nivel para ejecutar tareas de forma asíncrona usando hilos o procesos.

```
from concurrent.futures import ThreadPoolExecutor

def tarea(n):
    print(f"Tarea {n} en ejecución")
    return n * n

with ThreadPoolExecutor(max_workers=5) as executor:
    futuros = [executor.submit(tarea, i) for i in range(5)]
    resultados = [f.result() for f in futuros]

print(f"Resultados: {resultados}")
```

Procesos con concurrent.futures

El mismo módulo también puede ser usado para manejar procesos en paralelo.

```
from concurrent.futures import ProcessPoolExecutor

def tarea_proceso(n):
    print(f"Tarea {n} en ejecución en un proceso separado")
    return n * n

with ProcessPoolExecutor(max_workers=5) as executor:
     futuros = [executor.submit(tarea_proceso, i) for i in
range(5)]
    resultados = [f.result() for f in futuros]

print(f"Resultados: {resultados}")
```

Red

Comunicación HTTP con HTTPX

HTTPX es una biblioteca HTTP moderna para Python que permite realizar peticiones HTTP/1.1 y HTTP/2 de forma asíncrona.

```
import httpx

async def fetch(url):
    async with httpx.AsyncClient() as client:
        response = await client.get(url)
        return response.text

import asyncio
url = "https://www.example.com"
contenido = asyncio.run(fetch(url))
print(contenido)
```

Websockets con websockets

La biblioteca websockets permite la creación de clientes y

servidores WebSocket, facilitando la comunicación en tiempo real.

```
import asyncio
import websockets

async def echo(websocket, path):
    async for message in websocket:
        await websocket.send(message)

start_server = websockets.serve(echo, "localhost", 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Seguridad

Autenticación con Authlib

Authlib es una biblioteca que facilita la implementación de autenticación OAuth y OpenID Connect en aplicaciones Python.

```
from authlib.integrations.flask_client import OAuth
from flask import Flask, redirect, url_for

app = Flask(__name__)
app.secret_key = 'secret'
oauth = OAuth(app)
oauth.register('example',   client_id='EXAMPLE_CLIENT_ID',
client_secret='EXAMPLE_CLIENT_SECRET',
authorize_url='https://example.com/authorize',
access_token_url='https://example.com/token',
client_kwargs={'scope': 'openid profile email'})

@app.route('/')
def homepage():
    return 'Welcome to the secure app'

@app.route('/login')
```

```
def login():
    redirect_uri = url_for('auth', _external=True)
    return oauth.example.authorize_redirect(redirect_uri)

@app.route('/auth')
def auth():
    token = oauth.example.authorize_access_token()
    user = oauth.example.parse_id_token(token)
    return f'Hello, {user["name"]}!'

if __name__ == '__main__':
    app.run()
```

Seguridad en API con FastAPI

FastAPI es un framework moderno y de alto rendimiento para construir APIs con Python, que incluye herramientas para gestionar la autenticación y la autorización.

```
from fastapi import FastAPI, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm

app = FastAPI()

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="token")

@app.post("/token")
async def token(form_data: OAuth2PasswordRequestForm = Depends()):
    if form_data.username == "user" and form_data.password == "password":
        return {"access_token": form_data.username, "token_type": "bearer"}
    raise HTTPException(status_code=400, detail="Incorrect username or password")

@app.get("/users/me")
async def read_users_me(token: str = Depends(oauth2_scheme)):
    if token != "user":
        raise HTTPException(status_code=401, detail="Invalid
```

```
token")
    return {"username": token}

if __name__ == '__main__':
    import uvicorn
    uvicorn.run(app, host="127.0.0.1", port=8000)
```