ETEC PROFª ILZA NASCIMENTO PINTUS

3º DESENVOLVIMENTO DE SISTEMAS

TURMA B

VICTÓRIA ANTÔNIA BRITO BARBOSA

ATIVIDADE 2 – REQUISIÇÕES SÍNCRONAS ASSÍNCRONAS:

PESQUISA E EXEMPLOS SOBRE REQUISIÇÕES SÍNCRONAS E ASSÍNCRONAS

SÃO JOSÉ DOS CAMPOS

2024

ATIVIDADE 2

Realize uma pesquisa e exponha uma análise comparativa entre requisições síncronas e assíncronas. Apresente dois exemplos elucidativos para cada tipo e procure distinguir dois algoritmos, um para cada abordagem, destacando a diferença fundamental e expondo suas observações.

REQUISIÇÕES SÍNCRONICAS

RESUMO: Requisições síncronas são aquelas em que o cliente aguarda a conclusão da operação no servidor antes de continuar sua própria execução. Em outras palavras, o cliente envia uma solicitação ao servidor e pausa sua execução até que o servidor retorne uma resposta. Durante esse tempo de espera, o cliente fica bloqueado, o que significa que ele não pode prosseguir para outras tarefas ou operações.

Esse tipo de comunicação é crucial quando a ordem das operações é importante ou quando o resultado imediato é necessário para o funcionamento correto do programa.

Sendo assim, as requisições síncronas são uma forma de garantir a sincronia e a ordem correta das operações entre o cliente e o servidor, proporcionando uma experiência mais confiável e previsível para o usuário final.

EXEMPLOS:

- ✓ Requisição síncrona em uma aplicação de E-commerce:

Supondo que em um site de comércio eletrônico, quando um cliente adiciona um item ao carrinho e clica no botão "Comprar", o navegador envia uma requisição síncrona para o servidor para verificar o estoque do produto e calcular o preço total da compra, incluindo impostos e custos de envio. Durante esse período, o cliente aguarda a resposta do servidor e não pode realizar outras ações, como adicionar mais itens ao carrinho. Somente após receber a resposta do servidor, o navegador atualiza a página com as informações relevantes, como o resumo do pedido e as opções de pagamento.

- ✓ Requisição síncrona em um sistema de vendas em uma loja física:

Vamos supor que um cliente em uma loja física esteja fazendo uma compra com cartão de crédito. Quando o caixa insere o cartão na máquina, uma requisição síncrona é enviada para o servidor do banco para verificar se há fundos suficientes na conta do cliente. Durante esse tempo, a transação fica pendente, e o caixa não pode prosseguir com a transação até receber uma resposta do banco. Somente após a confirmação da transação pelo servidor do banco, a compra é concluída e o cliente pode continuar.

EXEMPLO COM CÓDIGO:

Abaixo temos o exemplo de código Python que faz uma requisição SÍNCRONA usando a biblioteca 'requests'. Esta biblioteca é muito popular para fazer requisições HTTP em Python devido à sua simplicidade e facilidade de uso. Este exemplo fará uma requisição GET para o site https://jsonplaceholder.typicode.com/posts/1 para obter um post específico:

```python
import requests

def get_post():
    try:
        response = requests.get('https://jsonplaceholder.typicode.com/posts/1')
        if response.status_code == 200:
            return response.json()
        else:
            print(f'Erro na requisição: {response.status_code}')
    except requests.exceptions.RequestException as e:
        print(f'Erro ao fazer a requisição: {e}')

post_data = get_post()
print(post_data)
```

Explicação passo a passo do que acontece no meu código:

1. Importamos a biblioteca 'requests' para fazer a requisição HTTP.
2. Definimos uma função 'get_post()' que irá fazer a requisição e retornar os dados do post.

3. Dentro da função ‘get_post()’, usamos um bloco ‘try-except’ para capturar possíveis erros durante a requisição.
4. Fazemos a requisição GET para https://jsonplaceholder.typicode.com/posts/1 usando ‘requests.get()’.
5. Verificamos o status da resposta usando ‘response.status_code’. Se for 200, isso significa que a requisição foi bem-sucedida.
6. Se a resposta for bem-sucedida, retornamos os dados do post em formato JSON usando ‘response.json()’.
7. Se houver algum erro na requisição, como falha de conexão ou URL inválida, o bloco ‘except’ capturará a exceção e imprimirá uma mensagem de erro.
8. Finalmente, chamamos a função ‘get_post()’ e armazenamos os dados do post em uma variável ‘post_data’.
9. Imprimimos os dados do post.

Vemos que este código bloqueará a execução até que a requisição seja completada, o que significa que o programa não fará mais nada enquanto aguarda a resposta da requisição.

O exemplo de código acima é síncrono porque ele executa de forma sequencial e bloqueia a execução até que a requisição seja completada.

- ✓ Bloqueio da Execução: Após fazer a chamada para ‘requests.get()’, o código espera até que a resposta seja recebida antes de continuar. Durante esse tempo de espera, nada mais é executado.
- ✓ Execução Sequencial: O código segue uma linha de execução sequencial. Primeiro, faz a requisição, verifica o status da resposta e, em seguida, retorna os dados do post ou imprime uma mensagem de erro, dependendo do resultado da requisição.
- ✓ Retorno Direto dos Dados: O método ‘requests.get()’ retorna diretamente os dados do post ou uma mensagem de erro, dependendo do resultado da requisição. Não há uso de corrotinas ou eventos assíncronos para lidar com múltiplas operações simultaneamente.

Essas características tornam o exemplo de código síncrono porque ele espera cada operação ser concluída antes de prosseguir para a próxima, e não há paralelismo na execução.

REQUISIÇÕES ASSÍNCRONICAS

RESUMO: Ao contrário das requisições síncronas, onde o programa fica bloqueado enquanto aguarda a resposta do servidor, as requisições assíncronas permitem que o programa continue executando outras tarefas, aproveitando melhor o tempo de espera pela resposta. Isso não apenas melhora a eficiência no uso dos recursos do sistema, mas também garante uma experiência mais fluida para o usuário, especialmente em ambientes com alto volume de requisições. Em suma, as requisições assíncronas são essenciais para sistemas distribuídos e aplicações web que buscam alta escalabilidade e responsividade, permitindo que o sistema mantenha sua capacidade de resposta enquanto lida com um grande número de solicitações simultâneas.

EXEMPLOS:

- ✓ Recebimento de Notificações em Redes Sociais:

Quando você está usando uma rede social como o Facebook ou o Twitter, as notificações sobre novas interações, como curtidas, comentários ou mensagens, são geralmente entregues de forma assíncrona. O aplicativo ou site continua funcionando normalmente enquanto aguarda essas notificações. Assim que chegam novas interações, elas são entregues ao usuário, sem interromper a experiência de uso.

- ✓ Atualização de Feed em Aplicativos de Notícias:

Em aplicativos de notícias, como o Feedly ou o Flipboard, a atualização do feed de notícias é realizada de forma assíncrona. Enquanto você navega pelas notícias existentes ou lê artigos, o aplicativo continua a buscar novas notícias nos servidores. Quando novos artigos estão disponíveis, eles são baixados e exibidos para o usuário, sem bloquear a interação atual. Isso permite que o usuário continue navegando de forma fluida pelo aplicativo, sem interrupções.

EXEMPLO COM CÓDIGO:

Abaixo temos um exemplo de código Python usando ‘asyncio’ e ‘aiohttp’ para fazer requisições assíncronas. Este exemplo fará várias requisições GET assíncronas para diferentes URLs e imprimirá os resultados:

```python
import asyncio
import aiohttp

async def fetch_data(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.json()

async def main():
    urls = [
        'https://jsonplaceholder.typicode.com/posts/1',
        'https://jsonplaceholder.typicode.com/posts/2',
        'https://jsonplaceholder.typicode.com/posts/3'
    ]

    tasks = [fetch_data(url) for url in urls]
    results = await asyncio.gather(*tasks)

    for result in results:
        print(result)

asyncio.run(main())
```

Explicação passo a passo do que acontece no meu código:

1. Importamos as bibliotecas ‘asyncio’ e ‘aiohttp’ para lidar com operações assíncronas e fazer requisições HTTP assíncronas, respectivamente.
2. Definimos uma função assíncrona ‘fetch_data()’ que recebe uma URL como argumento, faz uma requisição GET assíncrona para essa URL usando ‘aiohttp.ClientSession(‘`, e retorna os dados do JSON da resposta.
3. Definimos outra função assíncrona chamada ‘main()’ que é o ponto de entrada do nosso programa assíncrono.
4. Criamos uma lista de URLs que queremos buscar.

5. Criamos uma lista de tarefas assíncronas (‘tasks’) usando a compreensão de lista para chamar a função ‘fetch_data()’ para cada URL na lista de URLs.
6. Usamos ‘asyncio.gather()’ para aguardar todas as tarefas assíncronas serem concluídas. Isso permite que todas as requisições sejam feitas de forma assíncrona e, em seguida, esperamos até que todas sejam concluídas antes de prosseguir.
7. Finalmente, iteramos sobre os resultados e imprimimos cada um deles.

Este exemplo ilustra como fazer várias requisições HTTP de forma assíncrona em paralelo usando ‘asyncio’ e ‘aiohttp’. Isso pode melhorar significativamente o desempenho de aplicativos que fazem várias chamadas de API, pois permite que várias chamadas sejam feitas simultaneamente em vez de uma por uma.

Este código é assíncrono porque faz uso da biblioteca ‘asyncio’ e ‘aiohttp’ para realizar operações de rede de forma assíncrona. Vou explicar algumas características que tornam este código assíncrono:

- ✓ Uso de Corrotinas Assíncronas: As funções ‘fetch_data()’ e ‘main()’ são definidas como corrotinas assíncronas usando a palavra-chave ‘async’. Isso permite que elas sejam executadas de forma assíncrona, ou seja, sem bloquear o fluxo de execução principal do programa.
- ✓ Operações Assíncronas de I/O: Dentro da função ‘fetch_data()’, as operações de I/O, como a realização da requisição HTTP utilizando ‘session.get(url)’, são realizadas de forma assíncrona. Isso significa que o programa pode continuar executando outras tarefas enquanto aguarda a resposta do servidor, em vez de ficar bloqueado até que a resposta seja recebida.
- ✓ Utilização de ‘asyncio.gather()’: A função ‘main()’ utiliza ‘asyncio.gather()’ para aguardar a conclusão de várias corrotinas assíncronas ao mesmo tempo. Isso permite que as requisições sejam feitas em paralelo, melhorando a eficiência e o desempenho do programa.

- ✓ Modelo de Execução Não-Bloqueante: Ao usar ‘asyncio’ em conjunto com ‘aiohttp’, o código adota um modelo de execução não-bloqueante, onde as operações de I/O são realizadas de forma eficiente e escalável, sem bloquear o processo principal do programa.

Portanto, o código é assíncrono porque faz uso de técnicas e bibliotecas específicas para permitir que as operações de rede sejam realizadas de forma concorrente e não-bloqueante, melhorando assim a eficiência e a responsividade do programa. Isso é particularmente útil em situações onde é necessário fazer múltiplas requisições HTTP de forma eficiente, como ao lidar com APIs ou realizar scraping de dados da web.