Laboratório Prático

Processamento Distribuído com Celery e Redis

Objetivo

Neste laboratório, o aluno irá compreender e aplicar na prática os conceitos de **processamento assíncrono e distribuído** utilizando a ferramenta **Celery** em conjunto com o **Redis** como broker de mensagens.

Ao final da atividade, o aluno será capaz de:

- Entender a arquitetura do Celery (cliente, broker e worker);
- Criar e executar tarefas assíncronas;
- Compreender o papel das filas no processamento distribuído;
- Observar o comportamento assíncrono do sistema na prática;
- Testar concorrência e paralelismo com múltiplos workers.

1. Pré-requisitos

Antes de iniciar, certifique-se de que o ambiente possui:

- Sistema operacional Linux (Ubuntu recomendado);
- Python 3.10 ou superior;
- Redis instalado e em execução;
- Ferramenta `uv` instalada (gerenciador de dependências Python);
- Conhecimentos básicos de terminal e Python.

2. Preparação do Ambiente

2.1 Criar diretório do projeto

No terminal, execute:

```
mkdir celery_lab
cd celery_lab
```

2.2 Criar ambiente virtual

```
uv venv
source .venv/bin/activate
```

2.3 Criar o projeto com nome diferente de “celery”

> ⚠️ Importante: o projeto **não pode se chamar `celery`**, pois isso gera conflito com a biblioteca.

```
uv init celery_demo
cd celery_demo
```

2.4 Instalar dependências

```
uv add celery redis
```

3. Estrutura do Projeto

A estrutura final do projeto será:

```
celery_demo/
│
├── app.py
├── celery_app.py
└── tasks.py
```

4. Configuração do Celery

4.1 Arquivo `celery_app.py`

Este arquivo é responsável por configurar o Celery e conectá-lo ao Redis.

```
from celery import Celery

celery_app = Celery(
    "celery_demo",
```

```
    broker="redis://localhost:6379/0",
    backend="redis://localhost:6379/0"
)

celery_app.autodiscover_tasks(["tasks"])
```

Explicação:

- `broker`: define onde as mensagens serão enfileiradas (Redis);
- `backend`: armazena os resultados das tarefas;
- `autodiscover_tasks`: permite que o Celery encontre automaticamente as tasks definidas.

5. Definição das Tarefas

5.1 Arquivo `tasks.py`

Neste arquivo são definidas as funções que serão executadas de forma assíncrona.

```
import time
from celery_app import celery_app

@celery_app.task
def soma(a, b, task_id):
    print(f"Iniciando task {task_id}")
    time.sleep(5)
    print(f"Finalizando task {task_id}")
    return a + b
```

Explicação:

- `@celery_app.task` transforma a função em uma tarefa Celery;
- `time.sleep(5)` simula um processamento pesado;
- Cada tarefa demora 5 segundos para concluir.

6. Cliente: Envio de Tarefas

6.1 Arquivo `app.py`

Este arquivo representa o cliente que envia as tarefas para o Celery.

```
from tasks import soma

if __name__ == "__main__":
    print("Enviando várias tasks...")

    for i in range(1, 6):
        soma.delay(10, 20, i)
        print(f"Task {i} enviada")

    print("Todas as tasks foram enviadas.")
    print("Cliente FINALIZOU a execução.")
```

Explicação:

- `delay()` envia a tarefa para o broker de forma assíncrona;
- O cliente não espera o resultado;
- O programa finaliza imediatamente após enviar as tarefas.

7. Executando o Redis

Certifique-se de que o Redis está em execução:

```
sudo systemctl start redis
```

Verifique:

```
redis-cli ping
```

Resposta esperada:

```
PONG
```

8. Executando o Worker Celery

Em um terminal separado, dentro do projeto e com o ambiente virtual ativo, execute:

```
uv run celery -A celery_app.celery_app worker --loglevel=info
--concurrency=1
```

Explicação dos parâmetros:

- `-A celery_app.celery_app`: aponta onde está a instância do Celery;
- `worker`: inicia um worker Celery;
- `--loglevel=info`: exibe logs detalhados;
- `--concurrency=1`: executa uma tarefa por vez (processamento sequencial).

9. Executando o Cliente

Em outro terminal, execute:

```
python app.py
```

Observações importantes:

- O cliente termina imediatamente;
- As tarefas continuam sendo executadas pelo worker;
- Cada tarefa demora aproximadamente 5 segundos;
- O total será cerca de 25 segundos para processar todas as tasks.

Isso demonstra **processamento assíncrono**.

10. Experimento 1 — Alterando a Concorrência

Pare o worker e execute novamente:

```
uv run celery -A celery_app.celery_app worker --loglevel=info
--concurrency=4
```

Agora execute novamente o `app.py`.

O que observar:

- Múltiplas tarefas sendo executadas simultaneamente;
- Tempo total de execução reduzido;
- Demonstração de paralelismo dentro do worker.

11. Experimento 2 — Processamento Distribuído

Para simular processamento distribuído:

- Abra **dois terminais**;
- Inicie **dois workers Celery** (mesmo comando);
- Execute o cliente novamente.

As tarefas serão automaticamente distribuídas entre os workers.

12. Conclusão

Este laboratório demonstrou na prática:

- O funcionamento do Celery como sistema de processamento distribuído;
- O uso de filas para desacoplamento entre cliente e executor;
- A execução assíncrona de tarefas;
- A escalabilidade horizontal através de múltiplos workers;
- A importância do broker como intermediário confiável.

O Celery é amplamente utilizado em sistemas reais para processamento em larga escala, sendo uma ferramenta essencial no desenvolvimento de aplicações distribuídas modernas.