

Relatório Técnico: Web Scraper Portal da Transparência

Arquitetura da Solução

Stack Principal

Backend:

- **Django:** Framework web robusto escolhido pela sua maturidade, facilidade de configuração de APIs e amplo ecossistema.
- **Django REST Framework:** Extensão para construção de APIs com serialização automática e validação de dados.
- **Celery:** Sistema de filas de tarefas distribuídas para processamento assíncrono.

Web Scraping:

- **Playwright:** Ferramenta de automação de navegador escolhida por sua capacidade de lidar com JavaScript moderno e detecção anti-bot avançada.

Infraestrutura:

- **Redis:** Broker de mensagens para Celery, oferecendo alta performance e simplicidade de configuração.
- **Poetry:** Gerenciamento de dependências para python.
- **Docker/Docker Compose:** Containerização para padronização de ambiente e facilidade de deployment.
- **Railway:** Plataforma de deployment em nuvem para ambiente de produção.

Decisões Arquiteturais

A escolha arquitetural principal foi a adoção do Celery.

Ela foi motivada pela natureza lenta do web scraping. Operações que podem levar 90-180 segundos não podem ser executadas de forma síncrona em uma API sem comprometer a estabilidade do sistema.

Com esse setup, configuramos um worker que utiliza o prefork pool, modelo no qual cada processo filho (slot de concorrência) é executado de forma independente. O número de slots é, por padrão, igual ao número de núcleos de CPU disponíveis, permitindo o processamento paralelo de múltiplas tarefas.

Isso nos permite ter um equilíbrio entre a estabilidade e a disponibilidade do sistema, o que é especialmente importante num ambiente de produção limitado como o free trial do Railway que estamos usando para produção (512 MB de RAM, 1 GB de disco e 2 vCPU).

Além disso, com o Celery, é fácil montar uma fila de tarefas, o que garante que toda solicitação será eventualmente tratada, mesmo que os recursos disponíveis não permitam realizá-la no momento.

Para o Celery funcionar, ele precisa de um message broker para enfileirar as tarefas e, opcionalmente, um result backend para armazenar os resultados dessas tarefas. O Redis é usado para ambas as funções por sua velocidade e eficiência, agindo como um intermediário super rápido onde o Celery deposita as tarefas a serem executadas e de onde os "workers" do Celery as pegam para processar, além de poder guardar o que aconteceu com cada tarefa após sua conclusão.

Desafios Técnicos Enfrentados

1. Evasão de Detecção Anti-Bot

O Portal da Transparência implementa mecanismos de detecção de automação que podem bloquear requests automatizados. Tive problemas com o bloqueio imediato no acesso e com captchas. Então foram necessárias algumas soluções para “humanizar” o processo.

Isso não era um problema no modo headed, mas com o modo headless o bloqueio foi imediato.

Solução Implementada:

Randomização de User-Agents

```
user_agents = [
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/119.0.0.0 Safari/537.36",
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/120.0.0.0 Safari/537.36",
    "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:109.0) Gecko/20100101 Firefox/121.0",
]
```

Simula acessos de diferentes navegadores e dispositivos, tornando o tráfego mais parecido com o de usuários reais

Script para mascarar automação

```
stealth_script = """
    Object.defineProperty(navigator, 'webdriver', { get: () => undefined });
    Object.defineProperty(navigator, 'plugins', { get: () => [1, 2, 3] });
    Object.defineProperty(navigator, 'languages', { get: () => ['pt-BR',
'pt', 'en-US', 'en'] });
    window.chrome = { runtime: {} };
    Object.defineProperty(navigator, 'permissions', {
        get: () => ({ query: () => Promise.resolve({ state: 'granted' }) }),
    });

```

```
"""
```

Disfarça características que denunciam a presença de um bot (como navigator.webdriver ou ausência de plugins), fazendo o navegador controlado parecer mais com um navegador usado por humanos.

Delays para simular comportamento humano

```
time.sleep(5)
for char in identifier:
    search_box.type(char, delay=random.uniform(50, 150))
```

2. Paginação Dinâmica e Carregamento Assíncrono

O portal utiliza paginação JavaScript dinâmica que requer navegação através de múltiplas páginas de resultados.

Solução Implementada:

Loop de paginação com detecção de estado

```
while True
...
    next_button = detail_page.locator(
        "#tabelaDetalheValoresSacados_paginate .paginate_button.next button"
    )

    if not next_button.count():
        break

    next_button_parent = detail_page.locator(
        "#tabelaDetalheValoresSacados_paginate .paginate_button.next"
    )
    is_disabled = (
        "disabled"
        in next_button_parent.get_attribute("class")
    )

    if is_disabled:
        break

    next_button.click()
    detail_page.wait_for_timeout(
        random.uniform(1000, 2000)
    )
...

```

Plataforma de Hiperautomação: ActivePieces

Na verdade, o ActivePieces foi a última solução que eu tentei, porque já era familiar com os outros produtos sugeridos (Zapier e Make).

Primeiro, com o Make, descobri que a integração dele com os produtos do Google é bem chata de se fazer, necessita de várias configurações no GCP para fazer funcionar, então deixei ele de lado.

Com o Zapier, achei o funcionamento do webhook dele muito limitado e com algumas características estranhas como a formatação automática do output da API, separando ele em vários campos diferentes que eram difíceis de juntar.

Finalmente, o ActivePieces não teve nenhum desses problemas. Integração fácil, funcionalidades claras e simples. O único empecilho foi ter que tratar o output do formulário utilizado pelo usuário com um script em javascript. Nele, os campos vazios continuam aparecendo, o que não funciona com a API, mas foi uma solução bem simples.

No final, o workflow de hiperautomação ficou dessa maneira:

