

Detalhamento da Implementação: Etapa 1 - Desbloqueio da UI (ETL)

O objetivo desta etapa é mover o processamento de arquivos para um **Web Worker** e utilizar a API de **Streaming** para garantir que a interface do usuário permaneça responsiva e que o consumo de memória seja otimizado.

1. Criação do Web Worker (`parser.worker.ts`)

O Web Worker será o novo coração do seu motor de ETL. Ele receberá o arquivo e executará todo o processamento de forma assíncrona.

1.1. Lógica de Streaming e Processamento

A chave para a eficiência é usar a API `File.stream()` para ler o arquivo linha por linha, sem carregá-lo totalmente na memória.

TypeScript

```
// parser.worker.ts

// Importe suas funções de parsing e tipagens aqui
// import { parseLine, MarketItem } from './types';

self.onmessage = async (event) => {
    const file: File = event.data.file;
    const totalSize = file.size;
    let processedSize = 0;
    const results: MarketItem[] = [];

    try {
        // 1. Obter o ReadableStream do arquivo
        const stream = file.stream();

        // 2. Decodificar o stream para texto
        const textDecoder = new TextDecoderStream();
        const readableStream = stream.pipeThrough(textDecoder);

        // 3. Criar um leitor para iterar sobre o stream
        const reader = readableStream.getReader();
        let buffer = '';

        while (true) {
            const { done, value } = await reader.read();
            if (done) break;
            const line = value.toString();
            const item = parseLine(line);
            results.push(item);
            processedSize += item.size;
            if (processedSize === totalSize) {
                self.postMessage({ results });
                break;
            }
        }
    } catch (error) {
        self.postMessage({ error });
    }
}
```

```

    // 4. Processar o chunk (pedaço) de dados
    buffer += value;
    const lines = buffer.split('\n');

    // A última linha pode estar incompleta, então a guardamos no
    buffer
    buffer = lines.pop() || '';

    for (const line of lines) {
        // 5. Executar a lógica de parsing (seu antigo fileParser.ts)
        // A função parseLine deve conter a lógica de Regex/Lexer
        const item = parseLine(line);
        if (item) {
            results.push(item);
        }

        // 6. Atualizar o progresso (aproximado)
        processedSize += line.length + 1; // +1 para o '\n'
        const progress = Math.min(100, Math.floor((processedSize /
totalSize) * 100));

        // Enviar progresso para a Main Thread
        self.postMessage({ type: 'progress', progress });
    }
}

// Processar a linha final restante no buffer
if (buffer) {
    const item = parseLine(buffer);
    if (item) results.push(item);
}

// 7. Enviar os resultados finais para a Main Thread
self.postMessage({ type: 'success', data: results });

} catch (error) {
    self.postMessage({ type: 'error', message: error.message });
}
};

```

1.2. Considerações sobre o Web Worker

- **Tipagem:** Como o Web Worker não tem acesso direto ao seu ambiente React, você precisará garantir que as tipagens (MarketItem , parseLine) sejam importadas ou definidas dentro do arquivo do Worker.

- **Criação do Worker:** Devido ao uso de TypeScript e Vite, você pode criar o Worker diretamente no seu componente React usando a sintaxe de importação:

TypeScript

```
const worker = new Worker(new URL('./parser.worker.ts',
import.meta.url), { type: 'module' });
```

2. Refatoração do Módulo de Parsing (`fileParser.ts` ou um novo Hook)

O componente que lida com o upload do arquivo (provavelmente o `fileParser.ts` ou um componente pai) será refatorado para iniciar o Worker e gerenciar a comunicação.

2.1. Novo Hook de Processamento (Exemplo)

É recomendado criar um *hook* customizado (`useFileProcessor.ts`) para encapsular a lógica do Worker.

TypeScript

```
// useFileProcessor.ts

import { useState, useEffect, useCallback } from 'react';
// Importe o Worker usando a sintaxe do Vite
const worker = new Worker(new URL('./parser.worker.ts', import.meta.url), {
type: 'module' });

export const useFileProcessor = () => {
  const [data, setData] = useState<MarketItem[] | null>(null);
  const [progress, setProgress] = useState(0);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    worker.onmessage = (event) => {
      const { type, data, progress, message } = event.data;

      if (type === 'progress') {
        setProgress(progress);
      } else if (type === 'success') {
        setData(data);
        setIsLoading(false);
        setProgress(100);
      } else if (type === 'error') {
        setError(message);
      }
    }
  }, []);
}
```

```

        setError(message);
        setIsLoading(false);
    }
};

return () => {
    // Limpeza: Terminar o worker se o componente for desmontado
    worker.terminate();
};
}, []);
};

const processFile = useCallback((file: File) => {
    setIsLoading(true);
    setProgress(0);
    setError(null);
    setData(null);

    // Enviar o objeto File para o Worker
    // O objeto File é transferível, o que é eficiente
    worker.postMessage({ file });
}, []);

return { data, progress, isLoading, error, processFile };
};

```

2.2. Uso no Componente de Upload

O componente de upload (onde você tinha a lógica síncrona) agora será muito mais limpo:

TypeScript

```

// MarketTable.tsx (ou componente de upload)

import { useFileProcessor } from './useFileProcessor';

const MarketTable = () => {
    const { data, progress, isLoading, error, processFile } =
useFileProcessor();

    const handleFileUpload = (event: React.ChangeEvent<HTMLInputElement>) =>
{
    const file = event.target.files?[0];
    if (file) {
        processFile(file);
    }
};

```

```
return (
  <div>
    <input type="file" onChange={handleFileUpload} disabled=
{isLoading} />
    {isLoading && <p>Processando... {progress}%</p>}
    {error && <p style={{ color: 'red' }}>Erro: {error}</p>}

    /* Renderizar a tabela usando 'data' */
    {data && <Table data={data} />}
  </div>
);
};
```

Esta refatoração garante que o processamento de dados ocorra em segundo plano, liberando a UI e resolvendo o gargalo mais crítico do seu aplicativo. A próxima etapa seria aprimorar a função `parseLine` dentro do Worker, aplicando a lógica de Lexer/Tokenizer.