Plano de Ação e Relatório de Projeto: CyberLogTest

1. Visão Geral

Este documento serve como um relatório técnico completo e um guia de aprendizagem baseado na jornada de desenvolvimento e estabilização do projeto **CyberLogTest**. O objetivo é documentar os desafios enfrentados, as causas-raiz dos problemas e as soluções definitivas implementadas, fornecendo um recurso valioso para a formação de novos desenvolvedores, especialmente juniores e estagiários.

O projeto evoluiu de uma aplicação local com instabilidades para um ambiente de desenvolvimento profissional, containerizado e robusto, alinhado com as melhores práticas da indústria.

2. A Jornada de Depuração: Desafios e Soluções

O processo foi marcado por uma série de desafios de infraestrutura e ambiente que precisaram ser superados. Cada erro ensinou uma lição valiosa sobre o desenvolvimento moderno com Docker e WSL no Windows.

Erro 1: Falha na Inicialização da Aplicação (`TypeError: Reflect.getMetadata`)

- **Sintoma:** A aplicação NestJS falhava imediatamente ao iniciar, antes mesmo de tentar conectar-se à base de dados.
- **Causa-Raiz:** O problema manifestou-se de duas formas. Primeiro, uma mistura inconsistente de sistemas de módulos (`require` vs. `import`) causava falhas na resolução de dependências. Após padronizar para CommonJS (`require`), o erro persistiu, revelando a verdadeira causa: a versão do Node.js (v20) que estávamos a usar não suportava a sintaxe de **decorators** (`@Module`, `@Controller`, etc.) nativamente.
- **Solução Definitiva:** Em vez de adicionar camadas de complexidade com "tradutores" como o Babel (que se provou instável no nosso ambiente), a solução mais limpa foi atacar a causa-raiz. Atualizámos a imagem base no `Dockerfile` para **`node:22-alpine`**. O Node.js v22 possui suporte nativo e estável para decorators, eliminando o `SyntaxError` original.

- **Lição Aprendida:** Garantir a compatibilidade entre a versão do *runtime* (Node.js) e as funcionalidades da linguagem/framework é crucial. Muitas vezes, atualizar o ambiente é uma solução mais robusta do que adicionar ferramentas de transpilação.

Erro 2: Instabilidade Crónica do Ambiente Docker e WSL

Este foi o desafio mais longo e complexo, com múltiplos sintomas que pareciam não ter relação, mas que partiam da mesma causa-raiz: uma comunicação instável entre o Windows, o Subsistema Windows para Linux (WSL) e o Docker Desktop.

- **Sintoma A:** `Could not connect to WSL` / `Docker daemon not running`
    - **O que significava:** O VS Code não conseguia comunicar com o serviço do Docker.
    - **Causa-Raiz:** O serviço do WSL na máquina Windows estava a entrar num estado "travado" ou inconsistente, impedindo o Docker de iniciar ou responder.
    - **Solução:** Após múltiplas tentativas de reparo e reinstalação, a solução mais fiável foi o procedimento de "reset de emergência": executar `wsl --shutdown` num terminal PowerShell (como Administrador) para forçar o reinício completo do subsistema.
- **Sintoma B:** `Exit code 137` **(Falta de Memória)**
    - **O que significava:** O processo de construção do Dev Container era subitamente encerrado.
    - **Causa-Raiz:** O "Exit code 137" em ambientes Linux é um sinal clássico do "Out of Memory (OOM) Killer". O sistema operativo estava a encerrar o processo para se proteger, pois a memória RAM alocada para o WSL/Docker tinha sido excedida.
    - **Solução:** Criar um ficheiro `.wslconfig` na pasta de utilizador do Windows (`C:\Users\seu_nome`) e definir um limite de memória superior (ex: `memory=8GB`).
    - **Lição Aprendida:** O ambiente de desenvolvimento moderno, especialmente com contêineres e servidores de IDE a correrem em simultâneo, consome uma quantidade significativa de RAM. É essencial configurar os limites de recursos adequadamente.
- **Sintoma C:** `EADDRINUSE` **(Conflito de Portas)**
    - **O que significava:** A aplicação não conseguia iniciar porque a porta (3000, depois 8080) já estava em uso.

    - **Causa-Raiz:** Uma "briga" entre duas ferramentas. A extensão Dev Containers (via `"forwardPorts"` no `devcontainer.json`) e o Docker Compose (via `"ports"` no `docker-compose.yml`) estavam ambos a tentar controlar o mapeamento da mesma porta.
    - **Solução:** Centralizar a responsabilidade. Removemos a configuração `forwardPorts` do `devcontainer.json`, deixando o `docker-compose.yml` como a única fonte da verdade para a infraestrutura de rede, o que eliminou o conflito.
- **Sintoma D: Falha no Hot-Reload**
    - **O que significava:** Alterações guardadas nos ficheiros no Windows não reiniciavam a aplicação dentro do contêiner.
    - **Causa-Raiz:** A "ponte" que espelha os ficheiros entre o sistema de ficheiros do Windows (NTFS) e o do Linux (ext4) é notoriamente lenta e ineficiente. As notificações de alteração de ficheiro perdem-se no caminho.
    - **Solução Definitiva:** Adotar o fluxo de trabalho recomendado pela Microsoft. **Movemos o código-fonte para dentro do sistema de ficheiros do WSL** e passámos a usar o VS Code com a extensão **Remote - WSL**. Isto tornou a comunicação de ficheiros nativa (Linux-para-Linux), resultando num hot-reload instantâneo e com um desempenho muito superior.
    - **Lição Aprendida:** Para desenvolvimento com Docker no Windows, o desempenho e a fiabilidade são drasticamente superiores quando o código reside no sistema de ficheiros do WSL.

Erro 3: Falha no Git Hook (`cannot execute binary file`)

- **Sintoma:** O comando `git push` era consistentemente bloqueado por um erro de permissão no script do Husky.
- **Causa-Raiz:** Incompatibilidade fundamental entre o Git para Windows (que não lida bem com permissões de execução do tipo Linux) e a expectativa do Husky de que os seus scripts sejam executáveis.
- **Solução:** Abandonar a "magia" dos hooks automáticos em favor de um fluxo explícito. Criámos um script no `package.json` (`"push:test": "npm test && git push origin develop --no-verify"`) que garante a execução dos testes antes do push de uma forma robusta e independente do ambiente.
- **Lição Aprendida:** Quando uma ferramenta de automação "transparente" (como hooks) se torna um obstáculo, um processo manual e explícito (como um script `npm`) é, muitas vezes, a solução mais fiável.