Lições Aprendidas, Erros e Soluções no Projeto Nomad-Test

Este documento detalha o processo de desenvolvimento e depuração do projeto Nomad-Test, com foco nas lições aprendidas, nos erros encontrados, suas causas-raiz e as soluções aplicadas.

1. Premissas do Desafio

O desafio inicial era desenvolver uma API em Node.js com o framework NestJS para processar logs de jogos de tiro em primeira pessoa, gerando relatórios de partidas, rankings de jogadores e identificando o MVP (Most Valuable Player). As premissas incluíam:

- **Funcionalidades:** Ranking por partida (frags, mortes), arma preferida do vencedor, awards (Killing Spree, Flawless Victory), ranking global, MVP (top 5 por saldo de frags), Friendly Fire.
- **Tecnologias:** Node.js, NestJS, TypeORM, PostgreSQL, Jest.
- **Boas Práticas:** Orientação a Objetos (SOLID, Use Cases, Services, Repositories), Test-Driven Development (TDD), Commits atômicos e progressivos.
- **Ambiente:** Upload de arquivo de log via API, persistência em banco de dados.

2. Inventário Lógico de Recursos e Pilares Cruciais

Durante o desenvolvimento e a depuração, diversos recursos e ferramentas foram utilizados, cada um desempenhando um papel crucial:

- **Powershell (Terminal):**
  - **Papel:** Ambiente de linha de comando para executar comandos npm (instalação de dependências, iniciar a aplicação, rodar testes) e jest (diretamente ou via npx).
  - **Crucialidade:** Interface primária para interação com o projeto Node.js.
- **VS Code (Editor de Código):**
  - **Papel:** Ambiente de Desenvolvimento Integrado (IDE) para escrever, editar e organizar o código-fonte (JavaScript/TypeScript).
  - **Crucialidade:** Ferramentas de autocompletar, realce de sintaxe, depuração visual (breakpoints) e organização de arquivos foram essenciais para a produtividade e a identificação de erros.
- **Postman (Cliente API):**
  - **Papel:** Ferramenta para testar os endpoints da API (POST para upload de logs, GET para rankings e MVP). Permite enviar requisições formatadas e inspecionar as respostas JSON.
  - **Crucialidade:** Essencial para a validação funcional das rotas e para observar

o formato dos dados de entrada (logs) e saída (respostas da API).

- **Supabase (PostgreSQL Database):**
  - **Papel:** Banco de dados relacional (PostgreSQL) para persistir os relatórios das partidas. O frontend do Supabase foi usado para inspecionar e manipular diretamente a tabela matches.
  - **Crucialidade:** A capacidade de visualizar o estado dos dados salvos no banco foi **absolutamente crítica** para depurar problemas de processamento de log e agregação de dados. O comando TRUNCATE TABLE matches RESTART IDENTITY; via Supabase ou terminal foi vital para garantir um estado limpo para cada teste de correção.
- **Node.js:**
  - **Papel:** Ambiente de execução JavaScript para o backend da aplicação.
  - **Crucialidade:** A base sobre a qual toda a aplicação NestJS foi construída.
- **NestJS:**
  - **Papel:** Framework Node.js para construção de aplicações backend escaláveis e manuteníveis. Fornece estrutura modular (Controllers, Services, Modules), injeção de dependência e um ecossistema robusto.
  - **Crucialidade:** Garantiu a aplicação de princípios de engenharia de software (SOLID) e a organização do código em camadas claras.
- **TypeORM:**
  - **Papel:** Object-Relational Mapper (ORM) para interagir com o banco de dados PostgreSQL. Mapeia classes JavaScript (Entities) para tabelas do banco e vice-versa.
  - **Crucialidade:** Simplificou as operações de banco de dados (salvar, buscar). A opção synchronize: true foi útil em desenvolvimento para recriar o schema automaticamente.
- **Jest:**
  - **Papel:** Framework de testes JavaScript para testes unitários e de integração (E2E).
  - **Crucialidade:** Essencial para validar a lógica de negócio e as rotas da API. A configuração correta do Jest foi um desafio em si, mas sua funcionalidade é indispensável.
- **Gemini (Eu):**
  - **Papel:** O assistente de IA que forneceu o código, as explicações e as orientações para depuração.
  - **Crucialidade:** Atuou como um parceiro de pair programming e um depurador, embora com falhas iniciais de assertividade e compreensão.

3. Lições Aprendidas, Erros e Causas-Raiz

O processo foi marcado por diversos erros, cada um ensinando lições valiosas.

Erro 1: Cannot find module '../src/log/data/log.repository' (Caminhos de Módulo)

- **Sintoma:** Testes Jest falhavam com erro de módulo não encontrado.
- **Causa Raiz:** Caminhos de importação e mockagem incorretos no arquivo de teste (log.service.spec.js). O caminho relativo ../src/log/ era redundante, e jest.mock() exige a string do caminho do módulo.
- **Solução:** Ajustar o caminho para ./data/log.repository em require() e jest.mock().
- **Lição Aprendida:** A precisão nos caminhos relativos em Node.js e a sintaxe exata das ferramentas (como jest.mock()) são cruciais.

Erro 2: "No tests found" (Configuração Jest)

- **Sintoma:** Jest não encontrava arquivos de teste, mesmo com eles presentes.
- **Causa Raiz:** Configuração incorreta do Jest no package.json (testRegex estava procurando apenas por .ts arquivos, enquanto os testes eram .js).
- **Solução:** Alterar testRegex para testMatch e configurar para **/?(*.)+(spec|test).[tj]s?(x).
- **Lição Aprendida:** A importância de uma configuração Jest correta e abrangente para garantir que todos os testes sejam descobertos e executados.

Erro 3: QueryFailedError: relation "matches" does not exist (Schema do Banco de Dados)

- **Sintoma:** Erro ao tentar inserir dados no banco de dados, indicando que a tabela matches não existia.
- **Causa Raiz:** A tabela matches foi deletada manualmente no Supabase, mas a aplicação NestJS não foi reiniciada. O TypeORM com synchronize: true só recria o schema na inicialização da aplicação.
- **Solução:** Sempre reiniciar a aplicação NestJS após alterações no schema ou deleção manual de tabelas para permitir que o TypeORM recrie a estrutura.
- **Lição Aprendida:** Entender o ciclo de vida do ORM e suas opções de sincronização de schema, especialmente em ambientes de desenvolvimento. Em produção, migrações são essenciais.

Erro 4: total_kills: 0 e ranking: [] no retorno do upload (Parsing de Log)

- **Sintoma:** A rota POST /logs/upload indicava sucesso, mas os relatórios das partidas salvas no banco de dados (e retornados no JSON de sucesso) tinham total_kills: 0 e ranking: [] para muitas partidas.
- **Causa Raiz:** As expressões regulares (killRegex, worldKillRegex) em _generateReportForMatch não eram flexíveis o suficiente para o formato exato das linhas do log (variações de espaços, anos de 2 dígitos, caracteres especiais

em nomes). Isso impedia a extração correta dos dados de kills e deaths.
- **Solução:** Refinar as expressões regulares para serem ultra-robustas, utilizando \s* para zero ou mais espaços, .+? para captura não-gananciosa, e ajustando para o formato de ano \d{2,4}.
- **Lição Aprendida:** A depuração minuciosa do match() da regex com console.logs é fundamental. Pequenas variações no formato do input podem quebrar regexes aparentemente corretas.

Erro 5: ReferenceError: killsByBeca is not defined (Erro de Digitação)

- **Sintoma:** Erro de referência durante o processamento do log.
- **Causa Raiz:** Um erro de digitação simples (killsByBeca em vez de killsByWeapon).
- **Solução:** Correção do erro de digitação.
- **Lição Aprendida:** A importância de revisões de código (code review) e de testes automatizados para capturar erros triviais, mas críticos.

Erro 6: MVP ainda retornando "saldo negativo/zero" (Agregação de Dados)

- **Sintoma:** Mesmo após o upload correto (com total_kills preenchido), a rota /matches/mvp ainda não retornava o MVP.
- **Causa Raiz:** A lógica de agregação em getMVReport() não estava acessando os dados de kills e deaths do report de cada partida de forma consistente. Embora o report estivesse correto, a forma como playerStats era populado não estava usando as chaves normalizadas corretamente para somar os valores.
- **Solução:** Refatorar a lógica de agregação em getMVReport() para garantir que os frags sejam somados a partir do report.ranking e as mortes do report.deaths, ambos com os nomes dos jogadores normalizados.
- **Lição Aprendida:** A complexidade na agregação de dados pode esconder falhas sutis. É crucial garantir que os dados sejam acessados e somados usando as chaves corretas e normalizadas em todas as etapas do pipeline de processamento e agregação.

Erro 7: TypeError: Cannot read properties of undefined (reading 'bind') (Contexto this em Testes)

- **Sintoma:** Testes unitários falhavam com TypeError ao tentar chamar métodos auxiliares do LogService.
- **Causa Raiz:** Incompatibilidade na forma como o Jest lida com a vinculação do this para métodos de classe (especialmente quando definidos como arrow functions como propriedades de classe) em conjunto com a linha Object.assign(logService, LogService.prototype); no beforeEach do teste.
- **Solução:** Remover a linha Object.assign(logService, LogService.prototype); do log.service.spec.js. A definição dos métodos auxiliares como arrow functions no

log.service.js principal já garante o this correto.
- **Lição Aprendida:** Entender as nuances do contexto this em JavaScript e como as ferramentas de teste (Jest) e ambientes de execução (Node.js/NestJS) podem interpretá-lo de forma diferente. Simplificar a vinculação quando a sintaxe moderna já a oferece.

4. Próximos Passos e Melhorias Contínuas

- **Implementação de TDD:** Para futuras funcionalidades e refatorações, a aplicação rigorosa do TDD (escrever teste que falha, escrever código para passar, refatorar) é essencial para garantir a qualidade desde o início.
- **Dockerização:** Contêinerizar a aplicação e o banco de dados com Docker Compose para garantir portabilidade e consistência em diferentes ambientes.
- **CI/CD:** Implementar um pipeline de CI/CD para automatizar a construção, teste e deployment, garantindo a conformidade e a qualidade contínua.
- **Documentação Contínua:** Manter o README.md atualizado e considerar a integração de Swagger/OpenAPI para documentação da API.
- **Tratamento de Erros Mais Robusto:** Implementar tratamento de erros mais sofisticado (ex: logging de erros com ferramentas como Sentry, retries com backoff exponencial para chamadas externas).
- **Otimização de Performance:** Para grandes volumes de dados, explorar indexação de banco de dados, paginação e estratégias de agregação em tempo real.