Desafio de Desenvolvimento Júnior - CGR - Sistema de Monitoramento de Rede e Alocação de Recursos

Contexto:
Você faz parte da equipe de desenvolvimento de uma empresa de telecomunicações que gerencia uma vasta infraestrutura de rede. Para otimizar a performance e a utilização dos recursos, precisamos de uma ferramenta que permita monitorar o status dos equipamentos de rede, alocar e desalocar recursos (como portas de switch, endereços IP, etc.) e identificar gargalos ou falhas.

O Desafio:
Seu objetivo é desenvolver um protótipo de um sistema simplificado de monitoramento e alocação de recursos de rede. O foco não é na interface gráfica complexa, mas sim na robustez da lógica de negócio, na integração entre os componentes e na demonstração das suas habilidades em Python, SQL, APIs e algoritmos.

Requisitos Essenciais:

1. **Modelagem de Dados (SQL):**
    - Crie um esquema de banco de dados (SQL) que represente, no mínimo, as seguintes entidades:
        - **Equipamentos de Rede:** (Ex: Switches, Roteadores, Servidores)
            - `id` (PK)
            - `nome`
            - `tipo` (e.g., 'Switch', 'Router', 'Server')
            - `ip_gerenciamento`
            - `status` (e.g., 'Online', 'Offline', 'Manutenção')
            - `localizacao`
        - **Recursos de Rede:** (Ex: Portas, Endereços IP)
            - `id` (PK)
            - `equipamento_id` (FK para Equipamentos de Rede)
            - `tipo_recurso` (e.g., 'Porta Ethernet', 'IP v4', 'IP v6')
            - `valor_recurso` (e.g., 'Eth0/1', '192.168.1.10', '2001:db8::1')
            - `status_alocacao` (e.g., 'Disponível', 'Alocado', 'Reservado')
            - `cliente_associado` (opcional, para qual cliente o recurso está alocado, pode ser nulo)
        - **Eventos/Logs:**
            - `id` (PK)
            - `equipamento_id` (FK)
            - `timestamp`
            - `tipo_evento` (e.g., 'Status Change', 'Resource Allocated', 'Resource Deallocated')
            - `descricao`
    - Forneça os scripts SQL para a criação das tabelas e alguns dados de exemplo (pelo menos 3 equipamentos, 5 recursos por equipamento e 3 logs).
2. **API REST (Python com Flask ou FastAPI):**

- Desenvolva uma API REST utilizando Python (sugestão: Flask ou FastAPI) que exponha os seguintes endpoints:
  - `GET /equipamentos`: Retorna todos os equipamentos de rede cadastrados.
  - `GET /equipamentos/{id}`: Retorna os detalhes de um equipamento específico.
  - `PUT /equipamentos/{id}/status`: Atualiza o status de um equipamento (e.g., de 'Online' para 'Offline'). Esta ação deve gerar um log de evento.
  - `GET /equipamentos/{id}/recursos`: Retorna todos os recursos associados a um equipamento, com seu status de alocação.
  - `POST /recursos/alocar`: Aloca um recurso disponível. A requisição deve conter `equipamento_id`, `tipo_recurso` e, opcionalmente, `cliente_associado`. Deve-se verificar se o recurso está disponível antes de alocar. Em caso de sucesso, o status do recurso deve ser atualizado para 'Alocado' e um log de evento deve ser gerado.
  - `POST /recursos/desalocar`: Desaloca um recurso alocado. A requisição deve conter `recurso_id`. Em caso de sucesso, o status do recurso deve ser atualizado para 'Disponível' e um log de evento deve ser gerado.
  - `GET /logs`: Retorna todos os logs de eventos.

3. **Lógica de Negócio e Algoritmos (Python):**
   - **Alocação Inteligente de Recursos:** Implemente uma função ou endpoint que, dado um `tipo_recurso` e um `equipamento_id` (opcional), retorne o *melhor* recurso disponível para alocação. A lógica de "melhor" pode ser, por exemplo:
     - O recurso com o menor `id` disponível.
     - O recurso que está "disponível" há mais tempo (se você adicionar um timestamp de última atualização de status de recurso).
     - *Seja criativo aqui!* Você pode implementar um algoritmo simples de priorização.
   - **Análise de Gargalos/Falhas (Algoritmo):** Implemente uma função que, dado um `equipamento_id`, verifique se há algum "gargalo" ou "problema" na sua alocação de recursos. Exemplos:
     - Mais de 80% dos recursos de um tipo específico estão alocados.
     - Muitos eventos de "Offline" para um equipamento em um curto período.
     - *Novamente, seja criativo!* Defina uma métrica simples de "gargalo" e use-a. Esta função não precisa ser um endpoint, mas deve ser demonstrável no código.

4. **Resolução de Problemas e Criatividade:**
   - **Simulação de Falhas:** Adicione um endpoint `POST /equipamentos/{id}/simular_falha` que, ao ser chamado, altere aleatoriamente o status de alguns recursos do equipamento para 'Indisponível' ou 'Com Problema' e gere logs correspondentes. Isso permitirá testar a resiliência do seu sistema e a capacidade de identificar problemas.
   - **Documentação e Explicação:** Forneça um `README.md` detalhado explicando:
     - Como rodar o projeto.
     - A arquitetura geral.
     - As decisões de design tomadas (especialmente em relação à lógica de negócio e algoritmos).
     - Como você abordou a simulação de falhas e como seu sistema reagiria a elas.
     - Quais seriam os próximos passos para tornar este protótipo um sistema completo.

Tecnologias Sugeridas:

- **Python:** Linguagem principal para a API e lógica.
- **Banco de Dados:** SQLite (pela simplicidade e por ser arquivo) ou PostgreSQL (se você tiver experiência e quiser um ambiente mais robusto).
- **Framework Web:** Flask ou FastAPI.
- **Gerenciamento de Dependências:** `pipenv` ou `venv` com `requirements.txt`.

Entrega:

O projeto deve ser entregue em um repositório Git (GitHub, GitLab, Bitbucket) público ou privado (com acesso concedido). O repositório deve conter:

- Todos os arquivos de código-fonte.
- Scripts SQL para criação do banco de dados e dados iniciais.
- O arquivo `requirements.txt` (ou similar para `pipenv`) com as dependências do projeto.
- Um `README.md` completo, conforme especificado no requisito 4.

Avaliação:

Serão avaliados os seguintes aspectos:

1. **Qualidade do Código:** Clareza, legibilidade, organização, uso de boas práticas.
2. **Modelagem de Dados:** Adequação do esquema do banco de dados aos requisitos.
3. **Funcionalidade da API:** Correção e completude dos endpoints.
4. **Lógica de Negócio e Algoritmos:** Implementação eficaz e criativa das regras de alocação e detecção de gargalos.
5. **Resolução de Problemas:** A forma como a simulação de falhas foi implementada e a explicação da reação do sistema.
6. **Documentação:** Clareza, detalhe e abrangência do `README.md`.
7. **Comprometimento e Cuidado:** Atenção aos detalhes, organização do projeto e apresentação.

Pontos Bônus (Opcional, se o tempo permitir):

- **Testes Unitários:** Escrever testes para alguns dos endpoints da API ou funções da lógica de negócio.
- **Docker:** Fornecer um `Dockerfile` para rodar a aplicação em um contêiner.
- **Notificações Simples:** Adicionar uma função (mesmo que apenas imprimindo no console) que "notifique" quando um equipamento fica offline ou um recurso é alocado/desalocado.
- **Melhoria na Lógica de Alocação:** Algoritmos mais complexos para alocação de recursos, considerando múltiplos critérios.

Dicas para o Participante:

- Comece pela modelagem do banco de dados.
- Desenvolva a API aos poucos, testando cada endpoint.
- Pense em como as diferentes partes do sistema se comunicarão.

- Não se preocupe em fazer algo "perfeito" de primeira. Foque em entregar uma solução funcional e bem explicada.
- Mostre seu raciocínio no `README.md`!
- by André, boa sorte ae kkk.