ChatGPT

Sistema Web Full-Stack de Gestão de Notas Acadêmicas

Você é um engenheiro de software sênior. Gere código **completo e executável** para um sistema web full-stack (frontend + backend + banco de dados), conteinerizado com Docker, com as seguintes especificações:

Padrões e Stack Tecnológico (pode ajustar se necessário)

- **Frontend:** React com Next.js 14 (utilizando App Router), TypeScript e Tailwind CSS para o design responsivo e moderno.
- **Backend:** Node.js 20 com NestJS (preferencialmente; ou Express com Zod para validações), em TypeScript.
- **Banco de Dados:** PostgreSQL 16 (relacional), acessado via **Prisma ORM**.
- **Containers:** Docker + docker-compose definindo serviços separados: web (frontend), api (backend) e db (PostgreSQL).
- **Testes:** Implementar testes unitários e de integração no backend (Jest) e testes end-to-end no frontend (Playwright).
- **Qualidade de código:** Configurar ESLint e Prettier (com padrões adequados de estilo), e hooks Git via Husky + lint-staged para garantir formatação e lint nos commits.
- **Autenticação (opcional):** Se possível, proteger as áreas de importação e relatórios com login (e-mail/senha). Uma implementação simples (por ex. JWT no backend com estratégia local do Passport, e frontend com página de login) é suficiente.

Funcionalidades Obrigatórias

Upload & Validação do XML

- Implementar uma **tela de upload** de arquivo XML (frontend) com suporte a *drag-and-drop*. Deve aceitar apenas arquivos XML válidos (verificar extensão/mimetype) e de tamanho limitado (definir um tamanho máximo, por ex. alguns MB), exibindo feedback ao usuário (barra de progresso, estados de carregando, sucesso ou erro via mensagens/toasts).
- **Validação XSD:** O backend deve validar o XML importado contra um **XSD** que define a estrutura esperada (alunos, cursos, turmas e notas). Se o XML não for válido ou bem-formado, retornar erros claros e amigáveis ao usuário (por exemplo, indicando linha/coluna ou campo obrigatório faltando). Forneça o arquivo XSD no código (por exemplo, em `assets/schema.xsd`) e utilize uma biblioteca de parser XML no Node.js para fazer a validação (como `libxmljs2` ou equivalente).

Exemplo de XSD (estrutura esperada do XML de entrada):

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

```
<xs:element name="Importacao">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Cursos">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Curso" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Codigo" type="xs:string"/>
                  <xs:element name="Nome" type="xs:string"/>
                  <xs:element name="CargaHoraria" type="xs:int"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Turmas">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Turma" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="Id" type="xs:string"/>
                  <xs:element name="CursoCodigo" type="xs:string"/>
                  <xs:element name="Semestre" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Alunos">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Aluno" maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="RA" type="xs:string"/>
                  <xs:element name="Nome" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Notas">
```

```
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Nota" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="RA" type="xs:string"/>
                    <xs:element name="TurmaId" type="xs:string"/>
                    <xs:element name="Valor" type="xs:decimal"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Obs: O XSD acima define que o XML de entrada possui quatro seções principais: **Cursos** (cada curso com código, nome e carga horária), **Turmas** (cada turma com ID único, código do curso associado e semestre), **Alunos** (cada aluno com RA – registro acadêmico – e nome) e **Notas** (cada nota vinculando um RA de aluno a um ID de turma e um valor de nota). O sistema deve usar essa estrutura para importar os dados.

Parse & Persistência dos Dados

- Após upload e validação bem-sucedida, o backend deve **parsear o XML** e persistir os dados no banco PostgreSQL através do Prisma, dentro de uma **transação** (para garantir que todos os dados relacionados sejam inseridos conjuntamente).
- **Mapeamento para modelos relacionais:** Defina modelos Prisma para **Student (Aluno)**, **Course (Curso)**, **Class (Turma)**, **Enrollment/Grade (Nota)**, e **ImportLog** (log de importação, ver seção de auditoria). Estabeleça os relacionamentos apropriados (por exemplo: Student possui muitas Enrollment, Course possui muitas Class, Student many-to-many Course via Enrollment, etc). Considere que:
- Cada **Aluno** deve ser único por `student_id` (RA). Evite inserir duplicatas se um mesmo aluno aparecer em múltiplas importações (use o RA como chave natural ou defina uma chave única).
- Cada **Curso** (aqui no sentido de disciplina/assunto) deve ser único por `course_code`. Evite duplicar registros de curso com o mesmo código.
- Cada **Turma** (oferta de curso em um semestre) deve ser única por seu `Id`. Vincule cada Turma a um Curso existente (foreign key via `CursoCodigo`).
- Cada **Nota/Enrollment** associa um Aluno a uma Turma específica, com um valor de nota. Evite duplicatas de combinação aluno+turma (um aluno não deve ter duas notas para a mesma turma).
- **Regras de negócio na importação:** Se o XML contém dados já existentes (mesmo RA ou mesmo código de curso), decida como tratar: pode **ignorar** inserção duplicada ou **atualizar** os dados existentes. Uma abordagem segura é não duplicar e talvez registrar que foram ignorados no log.
- Implemente validações adicionais no backend conforme necessário (por exemplo, garantir que valores de nota estejam entre 0 e 10, que referências de IDs realmente existam – e.g., cada

`TurmaId` em Notas corresponda a um Turma carregada, etc., retornando erro se inconsistência). Use o Prisma (ou Zod, caso Express) para ajudar em validações de esquema de dados.

Cálculos de Métricas (Regras de Negócio)

Após persistir os dados, o sistema deve calcular várias métricas acadêmicas para cada aluno e para conjuntos agregados: - **Média por disciplina (por aluno):** Calcular a média de cada aluno em cada disciplina cursada. *Nota:* Se o aluno possui apenas uma nota final por disciplina (como definido no XML), essa própria nota é a média da disciplina. (Caso futuramente houvesse múltiplas avaliações por disciplina, calcularia-se a média aritmética, mas aqui podemos assumir uma nota final por disciplina).
- **Média Geral (GPA) por aluno:** Calcular o **índice de desempenho global** de cada aluno considerando todas as disciplinas. Essa média geral deve ser **ponderada pela carga horária** de cada disciplina (GPA ponderado). Ou seja, para cada aluno: $$ GPA = \frac{\sum (nota_disciplina \times carga_horaria_disciplina)} {\sum (carga_horaria_disciplina)} $$. Utilize as cargas horárias fornecidas no cadastro de cursos para esse cálculo.
- **Situação em cada disciplina:** Determinar se o aluno **aprovou ou reprovou** em cada disciplina cursada, com base na nota final daquela disciplina. Critério: nota < 6,0 é **Reprovado**, nota ≥ 6,0 é **Aprovado**.
- **Situação Geral do Aluno:** Determinar se o aluno está **Aprovado** ou **Reprovado** no curso como um todo. Critério: o aluno é considerado **aprovado** globalmente se **sua média geral ≥ 6,0 e não possui reprovações críticas**. Interprete "reprovações críticas" como qualquer disciplina reprovada (ou seja, o aluno não pode ter nenhuma disciplina com nota < 6). Se a média geral for abaixo de 6,0 **ou** se o aluno tiver pelo menos uma disciplina reprovada, então sua situação geral é **Reprovado**.
- **Estatísticas agregadas:** Calcular também métricas por turma e por curso: - *Por Turma (classe):* aqui "turma" refere-se à classe/oferta específica de uma disciplina num semestre (conforme lista de Turmas do XML). Para cada Turma, calcule a **média da turma** (média das notas de todos os alunos naquela turma), o **desvio padrão** das notas naquela turma, e a **percentual de aprovação** naquela turma (% de alunos com nota ≥ 6 naquela classe).
- *Por Curso (disciplina):* para cada disciplina (curso no sentido de matéria), calcule a **média geral** de notas de todos os alunos que cursaram aquela disciplina (possivelmente em diferentes turmas/semestres), bem como o **% de aprovação** agregado na disciplina (considerando todos os alunos que cursaram). O desvio padrão agregado por disciplina também pode ser calculado.
- *Por Programa: (Se houver múltiplos cursos de graduação no sistema)*, poderia-se calcular a média geral por curso de graduação e coorte, mas isso é opcional e não explicitamente requerido. (No contexto atual, assumiremos foco nas disciplinas e turmas conforme acima.)

Os cálculos devem ser realizados no backend após a importação, armazenando os resultados em estruturas adequadas para exibir nos relatórios e para exportação. Opcionalmente, pode-se guardar alguns resultados no banco (por exemplo, média geral e situação de cada aluno, para facilitar consultas), ou calcular sob demanda.

Relatórios e Dashboard (Frontend)

Implemente uma interface frontend (React/Next.js) para visualizar os resultados processados, incluindo: - **Dashboard de indicadores (KPIs):** Uma página inicial de relatórios apresentando métricas gerais, como: - Número total de alunos importados.
- Média geral do curso (média geral de GPA dos alunos ou outra métrica global relevante).
- Percentual geral de aprovação (porcentagem de notas ≥ 6 no total, ou de alunos aprovados vs reprovados

globalmente).
- (Esses indicadores podem ser ajustados conforme interpretado, mas devem dar uma visão geral.) - **Tabela de resultados por aluno:** Exibir uma tabela listando cada aluno com suas informações e resultados, incluindo colunas como: RA, Nome, Média Geral (GPA) do aluno, Situação Geral (Aprovado/Reprovado), e possibilitar expandir ou navegar para detalhes do aluno. Nos detalhes ou expandido, mostrar as disciplinas cursadas por aquele aluno com suas respectivas notas, médias (se houver subdivisão), e situação em cada disciplina.
- **Tabela de resultados agregados por turma:** Exibir uma tabela listando cada Turma (por exemplo, identificada por código/ID da turma e semestre) com colunas: Disciplina (nome da matéria), Semestre, Média da Turma, Desvio Padrão, % Aprovados naquela turma, e número de alunos naquela turma.
- **Tabela de resultados agregados por disciplina (curso):** Outra tabela listando cada disciplina (curso/matéria) com colunas: Código da disciplina, Nome da disciplina, Média geral de todas as turmas/anos, Desvio Padrão geral, % de Aprovados total na disciplina, e total de alunos que cursaram.
- **Gráficos:** Incluir visualizações gráficas para facilitar a análise: - **Gráfico de barras por disciplina:** Mostrar, por exemplo, a média de cada disciplina (e/ou comparar distribuição de notas). Poderia ser um gráfico de barras onde cada barra é uma disciplina mostrando a média da turma ou média geral da disciplina. Alternativamente, poderiam ser barras agrupadas por turma para cada disciplina se detalhar por semestre.
- **Gráfico de pizza (pie chart) de aprovação/reprovação:** Mostrar a proporção geral de notas aprovadas vs reprovadas (ou alunos aprovados vs reprovados). Esse gráfico de pizza daria uma visão rápida do percentual de sucesso.
- **Filtros de relatório:** Permitir filtrar os dados exibidos por **curso (disciplina)**, **turma (classe específica)** e **semestre**, bem como buscar por nome ou RA de aluno: - Por exemplo, um conjunto de dropdowns: selecionar um Curso (disciplina) para ver dados apenas daquela matéria; selecionar uma Turma específica (talvez filtrada após escolher a disciplina ou listando todas) para ver dados somente daquela oferta; selecionar um Semestre para filtrar turmas/disciplinas oferecidas naquele período; e um campo de busca textual para filtrar a tabela de alunos por nome ou RA. A interface deve combinar os filtros conforme aplicados.
- **Implementação:** Utilize componentes React funcionais com useState/useEffect ou Next.js App Router data fetching (por exemplo, React Server Components ou API routes chamadas do client) para buscar os dados calculados do backend. Para os gráficos, pode-se usar uma biblioteca como **Chart.js** (via `react-chartjs-2`) ou **Recharts** para construir as visualizações. Os componentes devem ser responsivos e acessíveis (uso de tags semânticas, atributos ARIA nos gráficos se necessário, contraste adequado etc.).
- **UX:** Fornecer feedback claro ao usuário nas operações (ex: após importação, exibir um toast "Importação realizada com sucesso" ou em caso de erro "Falha na importação: [mensagem]"). As páginas de relatório devem indicar quando estão carregando dados ou se não há dados disponíveis.

Exportação de Dados Processados

O sistema deve permitir exportar os resultados consolidados após a importação e cálculo, nos seguintes formatos: **XML consolidado, CSV, JSON e PDF**. - **XML consolidado:** Gerar um novo arquivo XML contendo todas as informações originais dos alunos e resultados calculados adicionais. Esse XML de saída deve incluir os mesmos dados de entrada mais campos extra, como por exemplo: - Em cada aluno, acrescentar elementos de resultado: média geral e situação geral.
- Em cada disciplina cursada pelo aluno, indicar a situação (aprovado/reprovado) daquela disciplina.
- (Opcionalmente, poderia incluir estatísticas agregadas, mas não é obrigatório; foco nas informações por aluno).

Exemplo de estrutura XML de saída para um aluno:

```
<Aluno RA="123456" Nome="João da Silva">
  <Disciplinas>
    <Disciplina Codigo="MAT101" Nome="Matemática Básica" CargaHoraria="60"
Semestre="2023-1">
      <Nota>5.0</Nota>
      <Status>Reprovado</Status>
    </Disciplina>
    <Disciplina Codigo="PROG101" Nome="Introdução à Programação"
CargaHoraria="80" Semestre="2023-1">
      <Nota>7.5</Nota>
      <Status>Aprovado</Status>
    </Disciplina>
  </Disciplinas>
  <MediaGeral>6.5</MediaGeral>
  <SituacaoGeral>Reprovado</SituacaoGeral>
</Aluno>
```

(No exemplo acima, o aluno João tem duas disciplinas, uma aprovada e outra reprovada; sua média geral calculada é 6,5 e a situação geral é Reprovado por ter reprovação em Matemática.)
O backend deve fornecer um endpoint para baixar esse XML consolidado.
- **CSV:** Permitir exportar os dados principais em formato CSV. Pode haver mais de um CSV conforme necessidade: - Por exemplo, um CSV contendo a lista de alunos com suas médias e situação geral.
- Outro CSV contendo todas as notas (cada linha: aluno, disciplina, nota, status).
- Ou um único CSV consolidado com colunas suficientes (pode tornar-se amplo).
Implemente pelo menos uma exportação CSV coerente (por exemplo, todas as disciplinas de todos os alunos, com colunas: RA, Nome, Código da Disciplina, Nome da Disciplina, Nota, Status Disciplina, Média Geral do Aluno, Situação Geral do Aluno).
- **JSON:** Similar ao XML consolidado, disponibilize um endpoint que retorna todos os dados consolidados em **JSON**. Pode ser estruturado, por exemplo, com um array de alunos, onde cada aluno contém seus dados pessoais e um array de disciplinas com notas e status, além da média geral e situação.
- **PDF:** Gerar um relatório completo em PDF contendo todas as informações processadas, de forma formatada e apresentável: - Inclua uma **capa** com título, data da geração, e talvez logo fictício da instituição.
- Inclua um **sumário** ou visão geral com indicadores gerais (número de alunos, etc).
- Em seguida, uma seção **por aluno** listando para cada aluno suas disciplinas, notas, média e situação (pode ser uma tabela por aluno ou uma tabela consolidada quebrando por aluno – escolha um formato que fique legível, eventualmente quebrando página por aluno se necessário).
- Uma seção **por disciplina** mostrando as médias gerais e estatísticas de cada disciplina/turma.
- **Anexos de logs:** por fim, pode anexar páginas com os logs de importação (detalhes de data, arquivo, e quaisquer erros ou alertas durante a importação).
- O PDF deve ter paginação, cabeçalho/rodapé básicos (por exemplo, número de página, data, nome do relatório).
- Para gerar PDF no backend Node.js, pode-se usar uma biblioteca como **PDFKit** (para montar PDF via código) ou gerar um HTML e usar um headless browser (ex: Puppeteer) para renderizar em PDF. De

preferência, implemente com PDFKit ou similar para não depender de ambiente gráfico.
- Disponibilize o PDF via endpoint para download; opcionalmente, também permitir download pelo frontend (por exemplo, um botão "Exportar PDF" que chama o endpoint e baixa o arquivo).

Histórico de Importações & Auditoria

- Manter um registro de cada importação realizada em uma tabela **Imports** (ou ImportLog). Cada registro deve incluir os campos:
- `id` da importação (identificador único, pode ser autoincremental),
- `usuario` responsável (se houver autenticação; caso contrário, pode ser null ou “anon”),
- data/hora da importação,
- hash ou nome do arquivo importado (para referência/garantia de unicidade),
- número de registros importados (pode ser número de alunos ou total de linhas, defina claramente, ex: total de alunos importados),
- status (sucesso ou falha),
- mensagem de erro se falhou (ou resumo dos erros de validação),
- tempo de processamento (duração da importação).
- Fornecer uma **interface/admin ou endpoints** para consultar esse histórico. Por exemplo, uma página no frontend que lista as importações realizadas (mostrando colunas como data, usuário, arquivo, nº de alunos, status) e permite clicar em uma importação para ver detalhes.
- **Download de arquivos originais e processados:** Para cada importação, guardar o arquivo XML original enviado (ou um caminho/referência para ele) e o arquivo consolidado gerado (XML/CSV/JSON/PDF). Disponibilizar endpoints no backend para baixar tanto o arquivo original enviado quanto os arquivos consolidados gerados daquela importação. Exemplo: `GET /api/imports/{id}/original` e `GET /api/imports/{id}/consolidated/xml` (e análogos para csv, json, pdf). Na interface, na tela de histórico de importações ou resultados, ter links/botões para baixar esses arquivos.
- **Log de aplicação:** Implementar logging estruturado no backend. Cada ação importante (início/fim de import, erros de validação, cálculos, geração de relatório) deve ser registrada em um log **JSON** (uma linha por evento) no console ou em arquivo. Por exemplo, usar a biblioteca **Winston** configurada para formato JSON, incluindo campos como timestamp, nível (info/erro), mensagem e detalhes (ex: `{"level":"error","time":"...","msg":"Validação XSD falhou","errors":[...]} }`). Isso auxiliará na auditoria e depuração.

UX e Internacionalização (i18n)

- Toda a interface deve estar em **pt-BR** (textos, rótulos, mensagens de erro). Centralize as strings de interface para fácil manutenção; por exemplo, utilizar um arquivo de tradução JSON ou um objeto em que todas as strings fiquem definidas, caso se pretenda futuramente traduzir para inglês.
- Seguir boas práticas de UX: fornecer **feedback claro** em todas as interações (carregando, sucesso, erro). Por exemplo, durante upload exibir um indicador de carregamento, após conclusão mostrar mensagem de sucesso ou erros em destaque. Nos relatórios, se um filtro resulta em nenhum dado, exibir mensagem "Nenhum registro encontrado".
- A interface deve ser **acessível**: incluir labels em inputs, textos alternativos em gráficos se necessário, permitir navegação por teclado nos elementos principais, cuidar com contraste de cores usando Tailwind classes para temas acessíveis.

- Layout limpo e intuitivo: usar Tailwind para criar um design agradável, talvez com um menu ou header de navegação para as diferentes áreas (Importar, Relatórios, Histórico). Usar componentes reutilizáveis para cartões de KPI, tabelas (paginadas ou com scroll), etc., garantindo consistência visual.

Autenticação (Opcional)

- Se implementada, criar uma página de **Login** no frontend onde o usuário insere e-mail e senha para acessar o sistema. No backend, configurar autenticação básica: por exemplo, criar um módulo de Auth no NestJS com estratégia local (verificando e-mail/senha contra registros de usuários em uma tabela Users) e emitindo um JWT em caso de sucesso. Proteger as rotas da API relacionadas à importação de dados e acesso a relatórios, de modo que apenas usuários autenticados possam usá-las (por exemplo, usando Guards no NestJS ou middleware no Express). No frontend, proteger as páginas (utilizando perhaps middleware de auth em Next.js 14 ou verificações de token) para redirecionar ao login se não autenticado.
- **Obs:** Caso não implementado, pode-se supor que o sistema é single-user ou rodando em ambiente seguro, e simplesmente esconder/omitir o login. Dê preferência à implementação se o tempo permitir, mas não é o foco principal comparado às funcionalidades acima.

Considerações Finais

Desenvolva o código organizadamente, separando em módulos e camadas conforme o stack exige: - No **frontend**, utilize a estrutura do Next.js App Router (`app/` directory) para páginas. Por exemplo, `app/import` para página de upload, `app/dashboard` para dashboard de KPIs, etc. Utilize componentes para separar a lógica de upload, tabela de alunos, gráficos, etc. A comunicação com o backend pode ser feita via fetch API (ou Axios) chamando os endpoints do NestJS. - No **backend**, estruture a aplicação NestJS em módulos (por exemplo: `ImportModule`, `StudentModule`, `ReportModule`, etc.), com controllers, services e repositories (ou usar Prisma Client diretamente no service). Implemente os endpoints REST necessários: upload XML, obter dados processados (para dashboard, tabelas, gráficos), export endpoints (XML/CSV/JSON/PDF download), histórico, etc. Utilize DTOs e Pipes (ou Zod schemas) para validar dados das requisições quando aplicável (embora grande parte dos dados venha do XML já validado). - Configure o **Prisma** adequadamente: além dos modelos, crie seeds se necessário para testar (opcional). Use migrations para gerar as tabelas. - Prepare os **Dockerfile** tanto do frontend (baseando em node:20-alpine, construir o Next.js para produção) quanto do backend (node:20-alpine, instalar deps e start NestJS). Use o **docker-compose.yml** para orquestrar os 3 serviços (talvez adicionar um volume para persistir dados do Postgres). - Inclua no repositório/documentação instruções de como rodar: por exemplo, comandos `docker-compose up --build` para levantar tudo. Os testes (Jest/Playwright) devem poder ser executados separadamente (por exemplo, via `docker-compose run` ou local). - Assegure que o sistema gerado seja **executável end-to-end**: após `docker-compose up`, deve ser possível acessar a aplicação web, realizar um upload de arquivo XML de exemplo e visualizar os relatórios gerados, bem como baixar os arquivos exportados.

Por favor, entregue toda a base de código completa seguindo os requisitos acima.
