API de Gerenciamento de Tarefas - TASKMASTER

Sua equipe foi contratada para desenvolver o backend de uma nova e moderna plataforma de gestão de tarefas, a TASKMASTER. O cliente precisa de uma API RESTful robusta, escalável e de fácil manutenção que servirá de base para seus futuros aplicativos web e mobile.

O sucesso do projeto depende da qualidade da API. Ela deve ser intuitiva para outros desenvolvedores, segura e eficiente. Seu desafio agora é projetar e implementar a API, aplicando as melhores práticas de desenvolvimento de software para garantir um MVP de alta qualidade que atenda a todos os requisitos funcionais e técnicos definidos pelo cliente e pela sua equipe, de forma que ela possa servir de base para futuros desenvolvimentos.

Funcionalidades do Usuário (CRUD e Features)

Estas histórias focam nas funcionalidades que o usuário final da API irá consumir diretamente.

Épico 1: Gerenciamento de Tarefas

Este épico engloba as funcionalidades centrais de criação e manipulação de tarefas.

História 1.1: Criar uma nova tarefa

- **Como um** usuário da API, **eu quero** criar uma nova tarefa fornecendo seus detalhes (título, descrição, categoria e data limite), **para que** eu possa registrar uma nova responsabilidade na minha lista.
- **Contexto:** O usuário precisa adicionar um novo item à sua lista de tarefas. Ele fornecerá os dados necessários através de uma requisição para a API.
- **Critérios de Aceitação:**
    1. **Dado** que eu envie uma requisição POST para /tasks com um corpo JSON contendo dados válidos para uma nova tarefa. **Quando** a API processar a requisição. **Então** a API deve retornar o status 201 Created. **E** o corpo da resposta deve conter os dados da tarefa recém-criada, incluindo um ID único gerado pelo sistema.
    2. **Dado** que eu envie uma requisição POST para /tasks sem um campo obrigatório (ex: titulo). **Quando** a API processar a requisição. **Então** a API deve retornar o status 400 Bad Request. **E** o corpo da resposta deve conter uma mensagem de erro clara indicando qual campo está faltando.
    3. **Dado** que eu envie uma requisição POST para /tasks com um tipo de dado inválido (ex: prioridade como texto em vez de número). **Quando** a API processar a requisição. **Então** a API deve retornar o status 400 Bad Request com uma mensagem de erro apropriada.

História 1.2: Atualizar uma tarefa existente

- **Como um** usuário da API, **eu quero** atualizar os detalhes de uma tarefa existente, **para que** eu possa corrigir informações ou alterar suas propriedades.
- **Contexto:** Uma tarefa já existe no sistema e o usuário precisa modificar um ou mais de seus atributos, como o título, a descrição ou a data limite.
- **Critérios de Aceitação:**

1. **Dado** que uma tarefa com um ID específico existe. **E** eu envie uma requisição PUT para /tasks/{id} com um corpo JSON contendo os dados atualizados. **Quando** a API processar a requisição. **Então** a API deve retornar o status 200 OK. **E** o corpo da resposta deve conter a representação completa da tarefa com os dados atualizados.
2. **Dado** que eu tente atualizar uma tarefa com um ID que não existe. **Quando** eu enviar uma requisição PUT para /tasks/{id_inexistente}. **Então** a API deve retornar o status 404 Not Found.

História 1.3: Excluir uma tarefa

- **Como um** usuário da API, **eu quero** excluir uma tarefa específica, **para que** eu possa remover itens que não são mais relevantes da minha lista.
- **Contexto:** O usuário decidiu que uma tarefa não é mais necessária e deseja removê-la permanentemente do sistema.
- **Critérios de Aceitação:**
    1. **Dado** que uma tarefa com um ID específico existe. **Quando** eu enviar uma requisição DELETE para /tasks/{id}. **Então** a API deve retornar o status 204 No Content.
    2. **Dado** que eu tente excluir uma tarefa com um ID que não existe. **Quando** eu enviar uma requisição DELETE para /tasks/{id_inexistente}. **Então** a API deve retornar o status 404 Not Found.

Épico 2: Consulta e Organização de Tarefas

Este épico cobre as formas como os usuários podem visualizar e organizar suas listas de tarefas.

História 2.1: Listar tarefas com paginação e ordenação

- **Como um** usuário da API, **eu quero** listar minhas tarefas de forma paginada e poder ordená-las por diferentes critérios, **para que** eu possa navegar por uma grande quantidade de tarefas de forma eficiente e priorizar o que ver primeiro.
- **Contexto:** O usuário possui muitas tarefas e precisa de uma maneira organizada e performática para visualizar sua lista, sem carregar todos os dados de uma só vez.
- **Critérios de Aceitação:**
    1. **Dado** que eu envie uma requisição GET para /tasks. **Quando** a API processar a requisição. **Então** a API deve retornar o status 200 OK com a primeira página de tarefas, usando um tamanho de página padrão.
    2. **Dado** que eu envie uma requisição GET para /tasks com parâmetros de paginação (ex: ?page=1&size=5). **Quando** a API processar a requisição. **Então** a API deve retornar a página e o tamanho solicitados.
    3. **Dado** que eu envie uma requisição GET para /tasks com parâmetros de ordenação (ex: ?sort=dataLimite,asc). **Quando** a API processar a requisição. **Então** a lista de tarefas retornada deve estar ordenada de acordo com o critério especificado.

4. **E** em todos os casos, a resposta deve incluir metadados de paginação (número da página, total de elementos, total de páginas).

História 2.2: Filtrar tarefas por categoria

- **Como um** usuário da API, **eu quero** filtrar minha lista de tarefas por categoria, **para que** eu possa focar em um contexto específico (ex: "Trabalho", "Estudo").
- **Contexto:** O usuário deseja visualizar apenas um subconjunto de suas tarefas que pertencem a uma categoria de interesse.
- **Critérios de Aceitação:**
    1. **Dado** que existem tarefas de diferentes categorias. **Quando** eu enviar uma requisição GET para /tasks com o parâmetro de filtro (ex: ?categoria=Trabalho). **Então** a API deve retornar o status 200 OK. **E** o corpo da resposta deve conter apenas as tarefas que pertencem à categoria "Trabalho".
    2. **Dado** que eu filtre por uma categoria que não possui nenhuma tarefa. **Quando** eu enviar a requisição GET. **Então** a API deve retornar 200 OK com uma lista vazia.

Diretrizes Técnicas para o Desenvolvimento da API de Tarefas

Para garantir que nossa API seja robusta, organizada e siga as melhores práticas do mercado, vamos seguir as seguintes diretrizes técnicas durante o desenvolvimento.

1. Arquitetura em Camadas (Controller-Service-Repository)

Adote uma arquitetura de camadas para separar claramente as responsabilidades do código.

- **Controller (@RestController):** Sua única responsabilidade é gerenciar o tráfego HTTP. Deve receber as requisições, chamar o método apropriado na camada de serviço e retornar uma ResponseEntity com o status e o corpo adequados. Mantenha os controllers "magros" (sem lógica de negócio).
- **Service (@Service):** Esta é a camada central da aplicação, onde toda a lógica de negócio deve residir. Ela orquestra as operações, executa validações de regras de negócio e chama a camada de repositório para persistir os dados.
- **Repository (@Repository):** Responsável exclusivamente pela comunicação com o banco de dados. Deve conter apenas as operações de acesso a dados (CRUD, consultas específicas, etc.), sem nenhuma regra de negócio.

2. Transferência de Dados com DTOs (Data Transfer Objects)

Utilize DTOs para trafegar dados entre as camadas e, principalmente, como contrato de entrada e saída da sua API. Isso evita expor a estrutura interna do seu modelo de domínio (@Entity) e permite customizar os dados que são enviados ou recebidos.

3. Injeção de Dependência via Construtor

Prefira a injeção de dependência via construtor em vez de usar @Autowired em atributos. Essa abordagem torna as dependências da classe explícitas, facilita a criação de testes unitários e garante que os objetos sejam criados em um estado válido.

4. Validação de Dados de Entrada

Implemente validações para garantir a integridade dos dados recebidos pela API.

- **Validação de Sintaxe:** Use as anotações do Jakarta Bean Validation (ex: @NotNull, @Size, @Email) diretamente nos seus DTOs para validar o formato e a presença dos dados.
- **Validação de Negócio:** Regras mais complexas (ex: "um usuário não pode criar tarefas com datas no passado") devem ser implementadas na camada de serviço (@Service).

5. Tratamento Centralizado de Exceções

Crie um handler global de exceções utilizando a anotação @RestControllerAdvice. Centralize o tratamento de erros em um único lugar, evitando blocos try-catch repetitivos nos controllers e garantindo que a API retorne respostas de erro padronizadas e significativas para o cliente.

6. Documentação da API com Swagger/OpenAPI

Documente todos os endpoints da API utilizando anotações do Springdoc/Swagger. Uma boa documentação é crucial para que outros desenvolvedores (ou você mesmo no futuro) possam entender e consumir a API de forma fácil e rápida.

7. Extra: Testes Automatizados

Como implementação extra, garanta a qualidade e a confiabilidade da API com uma suíte de testes completa, dividida em duas frentes:

- **Testes Unitários:** Foque em testar a camada de serviço (@Service) de forma isolada. Utilize mocks (com Mockito, por exemplo) para simular o comportamento da camada de repositório e validar a lógica de negócio.
- **Testes Funcionais/De Integração:** Teste os endpoints da API de ponta a ponta, desde a requisição HTTP no controller até a interação com um banco de dados de teste (como o H2 em memória). Utilize ferramentas como o MockMvc para simular as requisições.