

DESAFIO - CRUD de Contatos

Instruções

- *Objetivo:*

Criar uma API RESTful capaz de gerenciar contatos e hospedá-la em nuvem.

- *Tecnologia:*

Java

- *Entregáveis:*

Link do repositório

Link da API rodando em um servidor da nuvem

Importante

Deve-se utilizar qualquer serviço **gratuito** para hospedar a aplicação (API), base de dados e o código.

(Render.com; AWS; Azure; GitHub; BitBucket; etc.)

Descrição

1. Desenvolver uma API no modelo **CRUD**, tendo endpoints:

- Listar todos os contatos
- Cadastrar um novo contato
- Alterar os dados de um contato
- Excluir um contato

2. Utilizar os métodos GET, POST, PUT, PATCH e um DELETE

3. O contato precisa ter os seguintes campos:

- Nome
- E-mail
- Telefone
- Data de Nascimento
- Lista de Endereços (cada endereço tem uma Rua, Número e CEP)

Dicas

- Documente o projeto em arquivos **markdown** (README.md) com instruções necessárias para que qualquer pessoa consiga rodar sua aplicação.
- Use boas práticas de programação.
- Documentação Swagger é um diferencial

Pontos que serão observados:

- Alcance dos objetivos propostos;
- Entrega do repositório e hospedagem da aplicação;
- Organização, semântica, estrutura, legibilidade e qualidade do código;
- As instruções para execução do projeto;
- Histórico de commits do *Git*.

Diário de Desenvolvimento

Dia 0 (28/01) – Obsoleto

Levantamento Superficial das funcionalidades desejadas no sistema

PRIORIDADE ALTA - INEGOCIÁVEIS:

- Possibilitar o cadastro de inúmeras entidades com valores não nulos para os seus campos:
 - **ID** – Identificador único gerado automaticamente no cadastro de um novo recurso;
 - **Nome** – Input do usuário;
 - **Email** – Input do usuário (único por toda a tabela);
 - **Telefone** – Input do usuário (único por toda a tabela);
 - **Data de Nascimento** – Input do usuário;
 - **Lista de Endereços**
 - Cada contato pode ter um ou mais endereços associados, que serão armazenados em uma tabela filha da tabela principal de contatos:
 - Nome da Rua
 - Número do endereço
 - Código Postal (CEP)
 - Identificador (chave-estrangeira) que permita relacionar os endereços com algum contato da tabela principal.
- Desenvolver funções para execução do CRUD utilizando a API REST
 - **GET**
 - para recuperar dados e portanto listar todos os contatos cadastrados;
 - **POST**
 - para enviar dados novos para a API e portanto cadastrar um novo contato;
 - **PUT**
 - para modificar **integralmente** um recurso já cadastrado e portanto, permitir atualizar um contato já existente;
 - **PATCH**
 - para modificar apenas partes específicas do recurso já cadastrado e portanto, permitir atualizar um contato já existente de forma mais granular;
 - **DELETE**
 - para remover um recurso do servidor e portanto permitir excluir um contato cadastrado no sistema.
- Garantir que a exclusão de um contato da tabela principal exclua corretamente todos os endereços vinculados à ele na tabela de endereços.
- Implementar no escopo ORM – Mapeamento de Objetos para Banco de Dados Relacional
- Documentar corretamente as funções de maneira eficiente e clara

PRIORIDADE MÉDIA - ESSENCIAIS:

- *Validações extras nos campos de cadastro – tanto para a criação quanto para a atualização:*
 - *Nome –*
 - *Garantir que as primeiras letras sejam maiúsculas (com exceção de conectivos como ‘da’, ‘das’, ‘do’, ‘dos’ e ‘de’) e que somente letras e caracteres especiais (como apóstrofe) possam ser utilizados;*
 - *Email –*
 - *Garantir que haja um domínio de email no campo bem como um domínio de nível superior válido (.com; .br; etc.) e que dois contatos não possuam o mesmo email*
 - *Telefone –*
 - *Garantir o armazenamento do código do país, código de área e número, aceitando diferentes formatos (como +55 (11) 98500-1256 ou +1 123 456 7890) e que dois contatos não possuam o mesmo telefone*
 - *Data de Nascimento –*
 - *Garantir que tenha apenas número e barras, seguindo o formato DD/MM/AAAA*
 - *Endereços –*
 - *Garantir que qualquer nome de rua possa ser colocado*
 - *Garantir que o número do endereço possua apenas números inteiros*
 - *Garantir que o CEP possua apenas números e hífen.*
- *Possibilitar pesquisas amplas usando o método GET (por exemplo, pesquisar todos os nomes que começam com “Jo-” ou números de telefone com prefixo de país +55, etc.)*
- *Versionamento da API;*
- *Ferramentas de Autenticação e Segurança para evitar problemas como modificações indesejadas e/ou SQL Injection;*
- *Tratamento de Erros de Requisição;*
- *Testes unitários.*

PRIORIDADE BAIXA - BOAS FUNCIONALIDADES

- *Interface gráfica intuitiva em framework front-end para pesquisa, cadastro e manipulação dos contatos armazenados no sistema.*

Dia 1 (29/01) –

Após enfrentar desafios no desenvolvimento da API, incluindo a manutenção do código, a estruturação do projeto e problemas de conexão com os serviços de hospedagem escolhidos, a melhor opção para solucionar os problemas tanto a curto quanto a longo prazo foi **tornar obsoleta a abordagem anterior e recomeçar o projeto** com uma outra perspectiva de desenvolvimento e planejamento.

Esta decisão é permitida graças ao estado ainda rudimentar da API anterior que contava com poucas chamadas HTTP já desenvolvidas e apenas com mock-ups para teste

Com base nisso, os novos pontos levantados para o desenvolvimento deste sistema são:

- Tecnologias que vão ser utilizadas:
 - Java 17;
 - Spring Boot 3.4.2;
 - Maven 3.9.9
 - MySQL (para teste local);
 - Postgres (para hospedagem no Render)
- Casos de Uso:
 - Adicionar um novo contato (incluindo Nome, Email, Telefone, Data de Nascimento e Endereço);
 - Adicionar um novo endereço à um contato existente (Incluindo Nome da Rua, Número do Logradouro e Código Postal - CEP);
 - Excluir um contato;
 - Excluir um endereço atrelado à um contato existente;
 - Modificar inteira ou parcialmente os dados de um contato;
 - Modificar inteira ou parcialmente os dados de um endereço;
 - Exibir todos os contatos;
 - Exibir todos os endereços;
 - Permitir pesquisas de contato;
 - Permitir pesquisas de endereço;.
- Funcionalidades extras desejadas:
 - Formatar os nomes para que, independente da entrada do usuário, eles sejam registrados com a primeira letra Maiúscula;
 - Garantir que seja cadastrado um email válido (exemplo@dominio.com);
 - Garantir que o CEP cadastrado seja válido e formatá-lo para **01234-567**
 - Garantir que o telefone cadastrado seja válido e formatá-lo para **(01) 12345-6789**
- Hospedagem
 - *Render.com*

Dia 02 (30/01) –

A primeira etapa do desenvolvimento agora é criar o modelo conceitual dos dados que a API está tratando para facilitar o entendimento das entidades que serão criadas e manejadas, bem como dos seus relacionamentos.



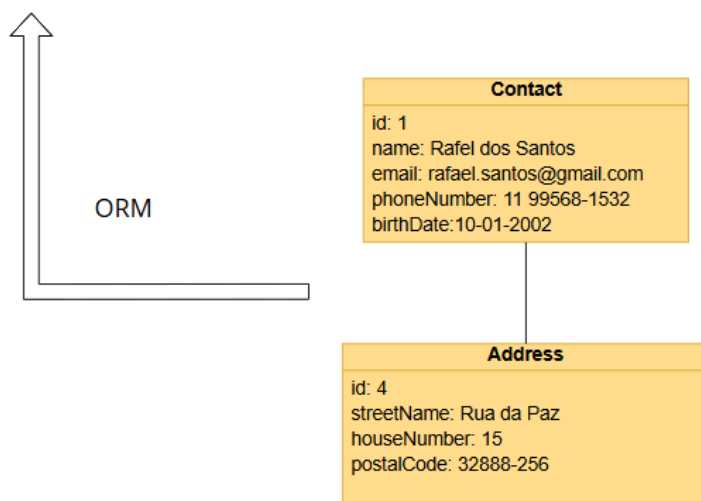
**Diagrama de Relação de Entidades entre o contato (contact) e o endereço (address) no modelo One to Many (Um para Muitos)*

tabela_contatos

id	nome	email	telefone	data_nascimento
1	Rafael dos Santos	rafael.santos@gmail.com	(11) 99568-1532	2002-01-10
2	Maria Aguiar	maria.aguiar@gmail.com	(11) 98775-9696	2002-14-27

tabela_enderecos

id	nome_rua	numero_casa	codigo_CEP	id_contato
1	Avenida das Flores	158	02456-010	1
2	Rua da Cidade	250	01414-025	2
3	Avenida dos Trabalhadores	680	20580-365	2
4	Rua da Paz	15	32888-256	1



**ORM das entidades e suas tabelas relacionais*



Contraparte JSON

```
{
  "id": 1,
  "nome": "Rafael dos Santos",
  "email": "rafael.santos@gmail.com",
  "telefone": "11 99568-1532",
  "data_nascimento": "10-01-2002",
  "enderecos": [
    {
      "id": 4,
      "nome_rua": "Rua da Paz",
      "numero_casa": 15,
      "codigo_CEP": "32888-256"
    }
  ]
},
```

*Representação do relacionamento entre as entidades em JSON

Logo em seguida, é necessário criar as entidades em classes específicas com todos os atributos pertinentes à ambas, incluindo um atributo que referencia o contato associado a um endereço.

Posteriormente, foi necessário começar o mapeamento ORM destas classes para criar, a partir delas, as tabelas e as colunas nas quais serão armazenadas os dados persistentes dos contatos e de seus endereços.

Com o ORM orquestrado pelo JPA pronto e uma conexão com um banco de dados MySQL local pelo MySQL Workbench 8.0, foi possível criar as primeiras requisições HTTP da API – Dois verbos GET, sendo um deles responsável por retornar todos os contatos e os endereços associados à eles e outro responsável por retornar apenas os endereços armazenados no banco de dados.

Para testar se as requisições funcionam corretamente, foi feito um *insert* arbitrário com dados idênticos aos do modelo conceitual do ORM e das tabelas, que permitiram verificar se todas as colunas e tabelas foram criadas corretamente e, principalmente, se as relações entre a tabela *contatos* e tabela *endereços* não só estavam definidas da forma certa como se elas seriam exibidas de forma correta no JSON resposta.











Com o objetivo de garantir essa exibição, foi definido explicitamente os relacionamentos OneToMany e ManyToOne nas entidades junto à maneira como modificações afetam uma à outra (usando a estratégia Cascade; isso tudo somado à normalização do JSON para evitar ciclos infinitos de iterações em virtude da natureza bidericonal da tabela *parent* (tabela *contatos*) com a tabela *child* (tabela *enderecos*), que cumpriram seus respectivos objetivos.

Dia 03 (31/01) –

Com as primeiras requisições do tipo GET feitas para ambas as entidades do sistema (Contato e Endereço), é possível partir para os outros verbos do dicionário HTTP que nos permitem realizar adições, correções e até exclusões de entidades.

Por mais que as verificações não estejam apropriadamente codificadas, criar as requisições brutas para testar o seus funcionamentos nos dados de testes utilizados é essencial para garantir a ausência de problemas na próxima etapa do desenvolvimento e design dessa API.

Para que eu possa acompanhar o meu progresso, esses são as requisições de extrema prioridade para serem implementadas:

- GET - Exibir todos os contatos;  (feita no **Dia 02** de Desenvolvimento)
- GET - Exibir todos os endereços;  (feita no **Dia 02** de Desenvolvimento)
- POST -Adicionar um novo contato; 
- PATCH - Adicionar um novo endereço à um contato existente; 
- DELETE - Excluir um contato; 
- DELETE - Excluir um endereço atrelado à um contato existente; 
- PATCH/PUT - Modificar inteira ou parcialmente os dados de um contato; 
- PATCH/PUT - Modificar inteira ou parcialmente os dados de um endereço; 
- GET - Permitir pesquisas de contato  (Somente por ID no momento)
- GET - Permitir pesquisas de endereço  (Somente por ID no momento)

Percebi durante o desenvolvimento que eu também não estava seguindo a convenção linguística padrão dos *commits*, utilizando mensagens escritas com verbos no Gerúndio ao invés de estarem conjugadas no Imperativo – todos os commits anteriores tem suas mensagens mantidas com o tempo verbal incorreta enquanto as posteriores ao dia 31/01 já seguiram a norma convencional.

Com as requisições das APIs devidamente definidas e funcionais dentro do escopo desejado, já é possível pensar em soluções para a validação dos dados passados tanto na criação quanto na atualização ou edição de um elemento e também em formas de adequar o código para as normas mais recentes do Spring Boot (como por exemplo, substituir o uso do **@Autowired** por *constructors*).

Dia 04 (1/02) –

A implementação do método PATCH foi um problema encontrado no desenvolvimento de ontem em virtude de um problema relacionado ao tratamento de propriedades nulas no sistema – no corpo da requisição PATCH, as propriedades omitidas são consideradas como nulas e, portanto, o método **.save** do repository sobrescreve as propriedades originais com esses valores nulos.

Para isso, criei um metodo auxiliar que cria um novo objeto “contato” mesclando as informações originais junto com as informações que o usuário quer atualizar.

Pesquisando, me deparei com uma forma de criar este objeto por meio de um conceito chamado Reflexão. No entanto, não é algo que eu esteja familiarizado e ele é melhor executado quando o projeto conta DTOs (Data Transfer Objects) – algo faltante no meu projeto.

Portanto, para que pudesse ainda sim realizar a função path da forma correta, decidi criar um algoritmo que mescla as informações no objeto contato que está sendo enviado para a API armazenar, comparando campo a campo entre o contato original e o contato que está sendo modificado e fazendo as substituições adequadas – aquilo que for nulo no objeto contato que está sendo enviado para API é preenchido com as informações originais enquanto o que estiver sendo modificado é mantido.

Para os campos do contato isso funciona bem, no entanto, para os endereços, são necessário alguns tratamentos –

- O primeiro deles é que todos os endereços não mencionados no corpo da requisição devem ser “colocados” na lista de endereços associados que está sendo enviada para a API;
- O segundo é que, endereços que forem colocados no corpo da requisição sem o ID vão ser cadastrados como novos endereços.
- Por fim, todos os endereços que tiverem o ID referenciado devem ser atualizados com as novas informações e mantendo aquilo que foi omitido.

Dia 06 (3/02) –

O dia de ontem (Dia 05 - 2/02) eu ainda tive problemas com a forma de fazer o método PATCH funcionar – a abordagem que eu tinha pensado no Dia 04 me pareceu a mais adequada, no entanto, ela foi implementada de maneira extremamente complexa que trouxe diversos problemas.

Pesquisando, encontrei uma classe personalizada extensora do BeansUtilsBeans que consegue realizar a transferência das propriedades do contato original que está sendo editado para as propriedades nulas do RequestBody – no entanto, para o relacionamento de objetos no modelo OneToMany, ela não possui boa performance e traz diversos erros.

Logo, a melhor solução é criar um algoritmo que seja capaz de realizar a seguinte lógica

- **Percorrer e comparar listas de endereços:** Primeiro, percorrer as duas listas de endereços: a do contato original e aquela recebida. Sempre que encontrar um **id** igual, fazer uma comparação e atualização. Se algum campo no **requestBody** enviado estiver nulo, ele será atualizado com o valor original correspondente.
- **Adicionar contatos não modificados:** Em seguida, adicionar na lista de contatos a ser enviada todos os endereços que não foram modificados durante o processo.
- **Salvar novos contatos:** Por fim, deixar que a função de **save** do Repository uide da criação de novos contatos quando o **ID** não for especificado.

A lógica para realizar essas soluções foi implementada, primeiramente comparando o corpo da requisição com o contato já existente na database e mesclando as informações necessárias (aquilo que foi passado como nulo é sobrescrito pelo valor da propriedade original enquanto o que é válido é mantido).

Na parte dos endereços associados, o objetivo foi, por meio de funções, criar uma nova lista que incluísse os novos endereços passados no corpo de requisição (com ID faltando), incluísse os endereços que existem no contato original, mas foram omitidos no corpo da requisição (que não devem ser modificados) e os endereços modificados no corpo da requisição, mantendo a integridade de suas informações - aquilo que foi passado como nulo é sobrescrito pelo valor da propriedade original enquanto o que é válido é mantido.

Por fim, para evitar que qualquer duplicata nesta lista ocorresse, foi realizada uma verificação para manter somente os valores distintos entre si na lista antes de definí-la como o corpo da requisição e enviá-la para o database.

Com todas as requisições feitas, foi hora de começar a tratar as colunas e seus dados – definir as expressões regulares que cada campo deve seguir, modificar certas partes do código para adequá-las às boas práticas do Spring Boot e criar a documentação Swagger, incluindo exemplos de requisições e respostas bem como o formato das propriedades das entidades armazenadas no banco de dados.

Dia 07 (4/02) –

Por fim, foi realizada a hospedagem do serviço no Render.com, utilizando Docker para criar uma imagem de uma Virtual Machine com o Java instalado e também o PostgreSQL para armazenamento dos dados.

A conexão foi feita com variáveis de ambiente e utilizando o endereço externo do database.

Para complementar, foi realizada a Documentação do projeto no README.md para complementar a documentação feita no Swagger