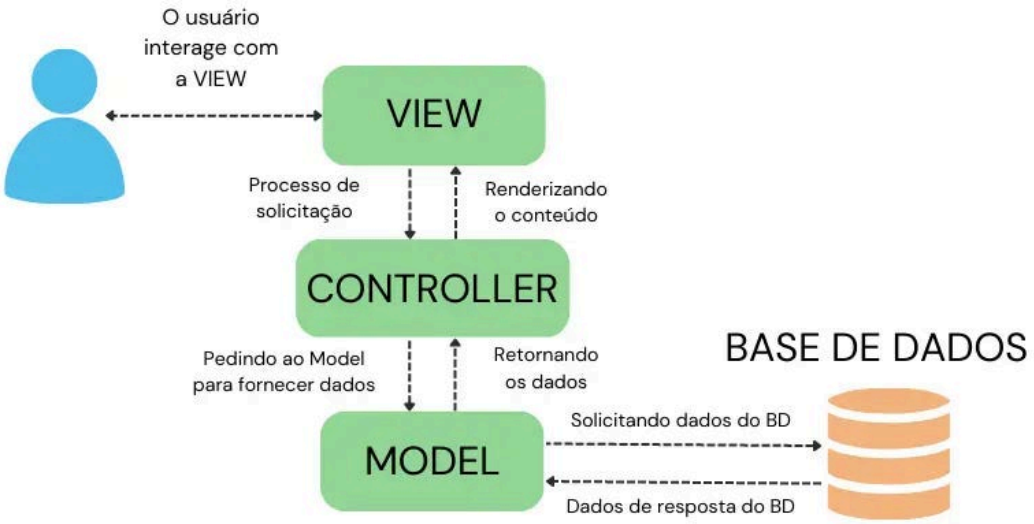
Aluno(a): Juliana Yumi Nishimura
Avaliação Back-End - 24 de Março de 2025
Documentação

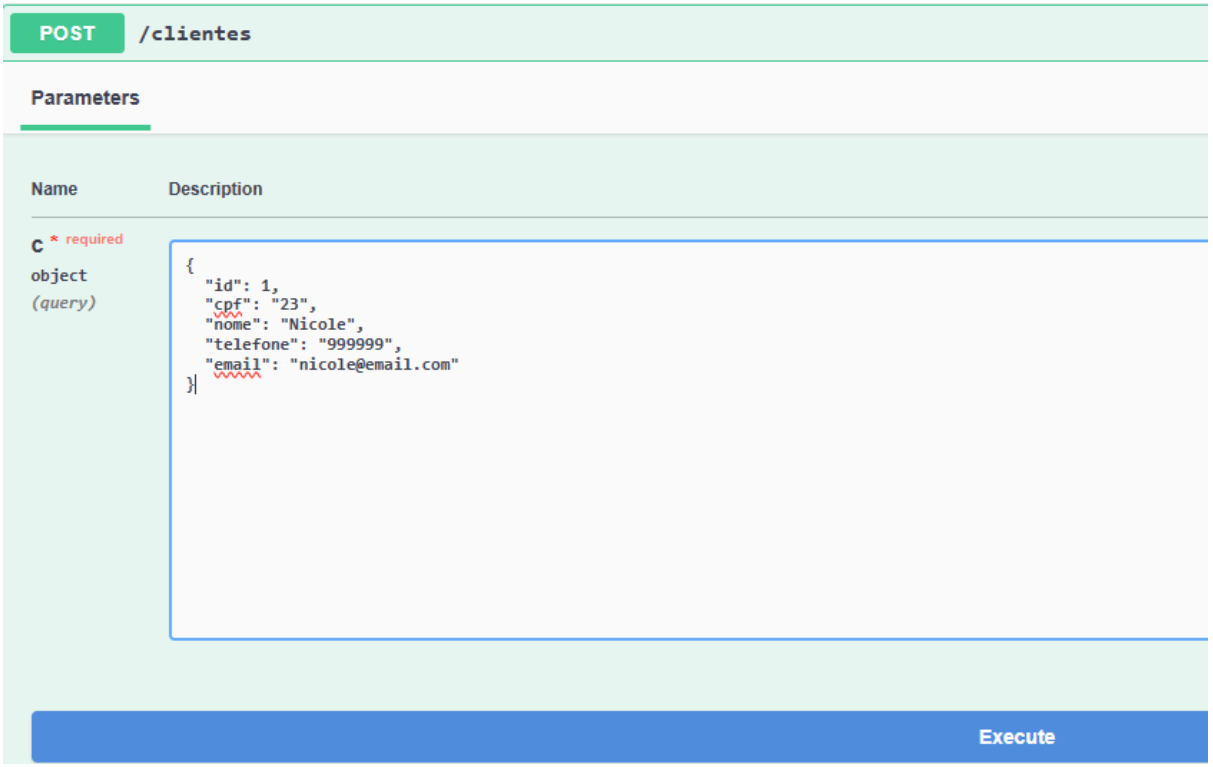
Essa aplicação da biblioteca é um software back-end web de acordo com as seguintes características - Primeiro, é Back-end pois organiza tudo que não é visível para o cliente, processando dados e informações, manipulando-os e verificando a lógica de negócios, normalmente fazendo uma conexão com algum local onde os dados são guardados e também com servidores. O back-end também é responsável pela arquitetura utilizada por trás desses processos, assim como vemos na nossa aplicação biblioteca - onde guardamos informações de livros, clientes, empréstimos e manipulamos esses dados - apenas mostrando o resultado ao cliente. É uma aplicação web pois interage com o usuário, ou seja, de máquina vai para o usuário. Como a aplicação requisita e retorna dados a um usuário, então dizemos que ela é web.

A seguinte arquitetura utilizada foi o MVC, seguindo o diagrama:



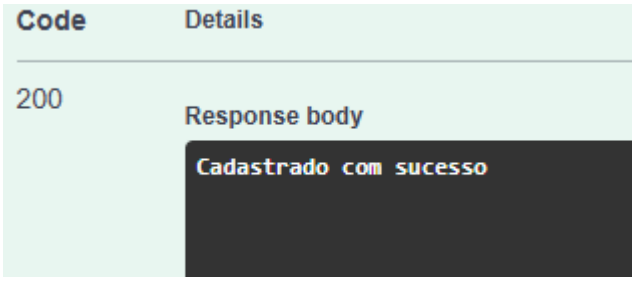
De acordo com o diagrama, temos o controller como a interação entre a view e o model, ou seja, como se fosse uma ponte, solicitando e retornando os dados.
A camada model possui os dados e possibilita sua manipulação, a lógica de negócios, regras do sistemas e interage com o banco de dados, que neste projeto não foi utilizado e sim substituído por uma classe onde há uma lista, utilizando o design pattern de singleton.
E a camada view se responsabiliza pela interação com o usuário, interagindo por meio de respostas em formatos como o de JSON por exemplo, com o usuário solicitando ou recebendo dados.
No projeto, temos então o controller fazendo a ponte entre a view e o model com o banco de dados; a view fornecendo os endpoints para a interação e a model estipulando as regras de sistema dos dados.

No projeto temos três endpoints(endereços-pontos de extremidades) - Livros, Clientes e Empréstimos. Na parte de clientes, temos a possibilidade de adicionar, atualizar, verificar e deletar os clientes. Primeiramente, ao adicionar um cliente utilizando o swagger para interagir, modificamos o json recebido da seguinte forma:

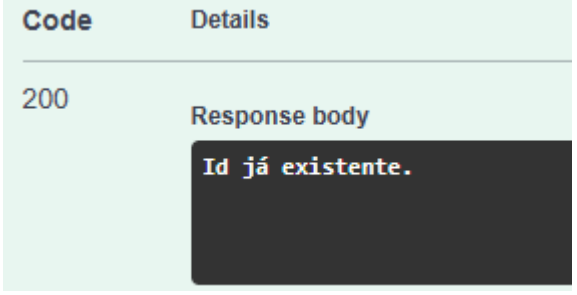


Ao executar podemos ter três respostas diferentes em três situações:

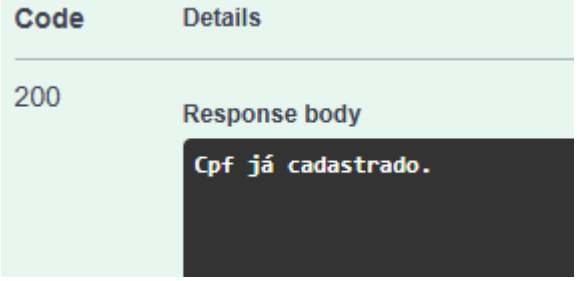
Primeiro, ao ter sucesso em cadastrar:



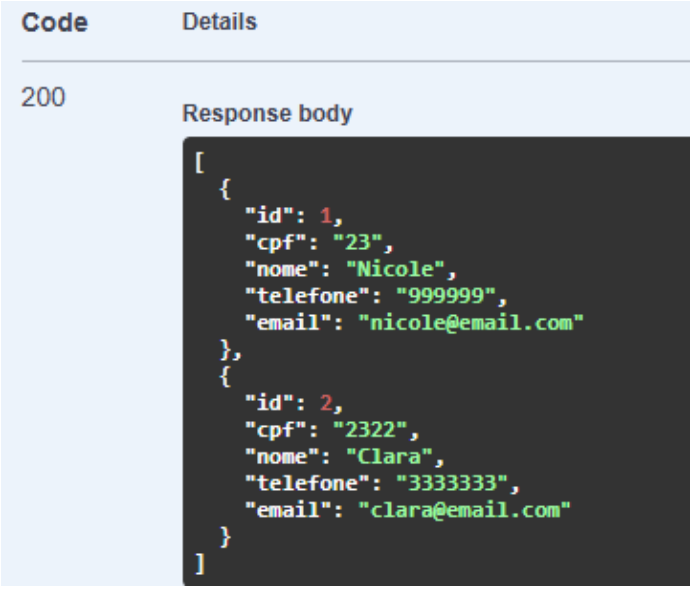
Segundo, ao adicionar um id que já foi cadastrado:



Terceiro, ao adicionar um cpf que já foi cadastrado:

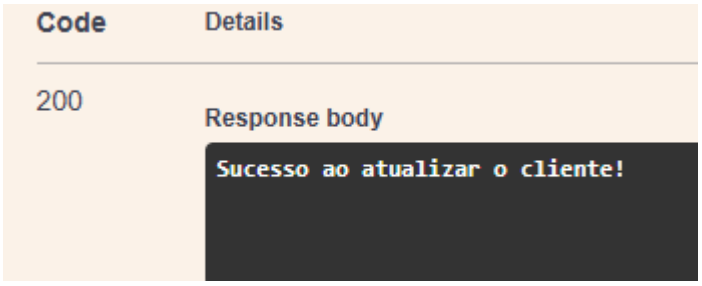


Em relação aos livros e empréstimos - possuímos também duas respostas já apresentadas - a de sucesso e a de caso um id que queiramos adicionar já exista.
Ao utilizar o get, ele busca a lista de clientes existentes pelo controller que comunica com o banco, e retorna da seguinte forma o json

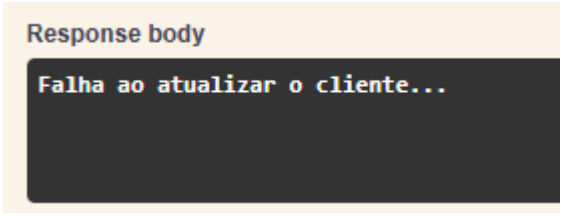


Para atualizar, pedimos o id e o corpo json dos dados:

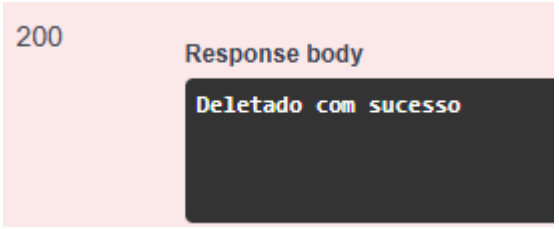
No corpo, é pedido o id também, que podemos apagar. Caso seja inserido dois ids diferentes(um no caminho e outro no json), o utilizado e permanecido será o do caminho. Em caso de sucesso temos a seguinte mensagem:



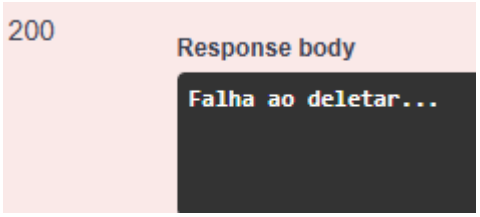
Caso a atualização falhe, temos a seguinte mensagem:



E temos a seguinte situação ao deletar, colocando o id do cliente com a seguinte mensagem de sucesso:



Em caso de falha, temos:

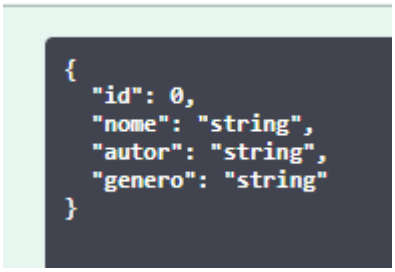


E podemos verificar isso no get novamente:

O nome do id foi atualizado, e o id 2 deletado.

Para os empréstimos e livros, temos a mesma situação mostrada em relação aos clientes, com diferenças no json e uma diferença no empréstimo - a busca por data de fim.

O json do livro é o seguinte:



```
{
  "id": 0,
  "nome": "string",
  "autor": "string",
  "genero": "string"
}
```

E o do empréstimo é:

```
{
  "id": 1,
  "data_inicio": "10-02-1015",
  "data_fim": "15-02-1015",
  "livros_emprestados": [
    {
      "id": 1,
      "nome": "abc",
      "autor": "fulano",
      "genero": "suspense"
    }
  ],
  "cliente": {
    "id": 1,
    "cpf": "23",
    "nome": "Nicole",
    "telefone": "333",
    "email": "nicole@email.com"
  }
}
```

Com isso, a outra diferença também é a busca de empréstimos pela data final.

Temos os seguintes mais famosos padrões de design:

1. Abstract Factory

   Utiliza uma interface para criar uma família desses objetos sem especificar as classes concretas (interfaces de sofá, cadeira e mesa - criando famílias de sofá, cadeira e mesa vintage e outra família desses móveis só que modernos).

2. Adapter

   Converte a interface de uma classe em outra interface que os clientes esperam.

3. Bridge

   Separa a abstração de sua implementação, permitindo que ambos possam ser alterados independentemente.

4. Builder

   Separa a construção de um objeto complexo de sua representação, permitindo a criação de diferentes representações do mesmo tipo de objeto.

5. Chain of Responsibility

   Permite que múltiplos objetos manipulem uma solicitação sem saber qual objeto a manipulará efetivamente.

6. Command

   Encapsula uma solicitação como um objeto, permitindo parametrizar clientes com diferentes solicitações.

7. Composite

   Compõe objetos em estruturas de árvores para representar hierarquias parte-todo, tratando objetos individuais e composições de objetos de maneira uniforme.

8. Decorator

   Anexa responsabilidades adicionais a um objeto dinamicamente, sem alterar sua estrutura.

9. Facade

   Fornece uma interface simplificada para um subsistema complexo, ocultando os detalhes internos.

10. Factory

    Define uma interface para o objeto, mas as subclasses decidem qual instanciar (exemplo - uma classe abstrata pizzaria que dá origem a pizzarias de tipos de pizzas diferentes, e essas pizzarias que instanciar uma pizza específica).

11. Flyweight

    Usa compartilhamento para suportar grandes quantidades de objetos de forma eficiente em termos de memória.

12. Interator

    Fornece uma maneira sequencial de acessar os elementos de um objeto agregado, sem expor sua representação interna.

13. Mediator

Define um objeto que encapsula como um conjunto de objetos interage, promovendo o desacoplamento entre eles.

14. Memento
    Captura e externaliza o estado interno de um objeto sem violar o encapsulamento, permitindo que o objeto seja restaurado a esse estado mais tarde.

15. Observer
    Define uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados.

16. Prototype
    Cria novos objetos copiando um objeto existente, em vez de criar um novo a partir do zero.

17. Proxy
    Fornece um objeto substituto que controla o acesso a outro objeto.

18. Singleton
    Garante que a classe possuirá apenas uma instância que poderá ser acessada.

19. State
    Permite que um objeto altere seu comportamento quando seu estado interno muda, parecendo que o objeto mudou de classe.

20. Strategy
    Define uma família de algoritmos, encapsula cada um e os torna intercambiáveis.

21. Template method
    Define o esqueleto de um algoritmo em um método, deixando alguns passos para que as subclasses implementem.

22. Visitor
    Permite adicionar novas operações a objetos de uma estrutura sem modificar as classes dos objetos.

Os seguintes padrões poderiam ser utilizados para ajudar no projeto seriam:
Inicialmente, é necessário utilizar o singleton para uma única instância no caso de utilizar apenas listas e não um banco de dados. Para um sistema onde temos

clientes, livros e empréstimos, seria interessante utilizar o decorator para diferentes tipos de livros - então uma interface geral livro, e decoradores de tipo, gênero, etc.
No caso de outros produtos que não são livros, mas são parecidos - como revistas, jornais e etc., seria interessante utilizar O template method que tem uma interface geral que pode dar origem a diferentes objetos que são “parecidos” entre si, como os citados. Usaria também decoradores para diferentes tipos de usuários, pelas mesmas motivações.