



Airports API

Airports API

Fase 2 – Endpoint Airport e o Padrão de Camadas



Introdução

Lembrando a Arquitetura da API

A API terá como raiz o endpoint `/airports`.

A partir da raiz, teremos em torno de cinco controladores, cada um responsável por um serviço REST específico:

Endpoint Airport

Retorna os dados completos de TODOS os aeroportos cadastrados na base de dados. Utiliza a URI `/airports/airport`.

Endpoint CitySearch

Lida com buscas por cidade, utilizando a URI `/airports/city/{cidade}`. O parâmetro `{cidade}` representa o nome da cidade desejada, e a busca é realizada de forma case-insensitive, ou seja, "Brasilia", "brasilia" e "BRASILIA" retornam o mesmo resultado.

Endpoint CountrySearch

Similar ao CitySearch, lida com buscas por país, utilizando a URI `/airports/country/{pais}`. Iremos retornar apenas o nome do aeroporto, código IATA e a cidade.

Endpoint IataCodeSearch

Lida com buscas por código IATA, utilizando a URI `/airports/iata/{iataCode}`. Iremos devolver os dados completo do aeroporto.

Endpoint nearMe (próximos a mim)

Este endpoint buscará o aeroporto mais próximo, utilizando a URI: `/airports/nearme`. O Endpoint irá receber dados da sua latitude e longitude (no corpo da requisição) e irá devolver uma lista de aeroportos ordenada por distância referente ao seu ponto, do mais próximo até o mais distante, limitado a 10 aeroportos.

Agora vamos fazer nosso primeiro endpoint: Endpoint “Airport”.



Padrão de Camadas

O padrão de camadas Java ORM é uma técnica de desenvolvimento que mapeia as classes de uma aplicação para as tabelas de um banco de dados. O ORM (Mapeamento Objeto-Relacional) é responsável por converter os dados entre o objeto Java e os dados da tabela correspondente.

O padrão de camadas Java ORM é utilizado para:

- Separar responsabilidades
- Facilitar testes
- Reutilizar lógica
- Simplificar a manutenção.

Alguns frameworks ORM em Java são:

- Hibernate,
- JPA (Java Persistence API),
- Spring Data JPA (Estamos usando esse! Lembra da preparação do projeto?)
- MyBatis.

O funcionamento do ORM é o seguinte:

O framework ORM cria um mapeamento entre os objetos do aplicativo e as tabelas do banco de dados.

Quando o aplicativo precisa recuperar dados, o ORM executa a consulta ao banco de dados

O ORM converte os resultados da consulta em objetos para serem usados pelo aplicativo

Quando o aplicativo precisa salvar dados, o ORM converte os objetos no formato de dados adequado para o banco de dados.

O padrão DTO (Data Transfer Object) é uma estratégia de design que permite a transferência de dados entre as camadas de uma aplicação Java. Ele é usado para encapsular dados em uma única estrutura, o que facilita a comunicação entre as camadas.

O padrão DTO é útil para:

- Reduzir o tráfego de rede
- Manter o código modular e desacoplado
- Melhorar a eficiência do software
- Suportar a serialização e desserialização de objetos.

Para entender como o padrão DTO funciona, vamos considerar um exemplo de um sistema de gerenciamento de usuários.

Exemplo:

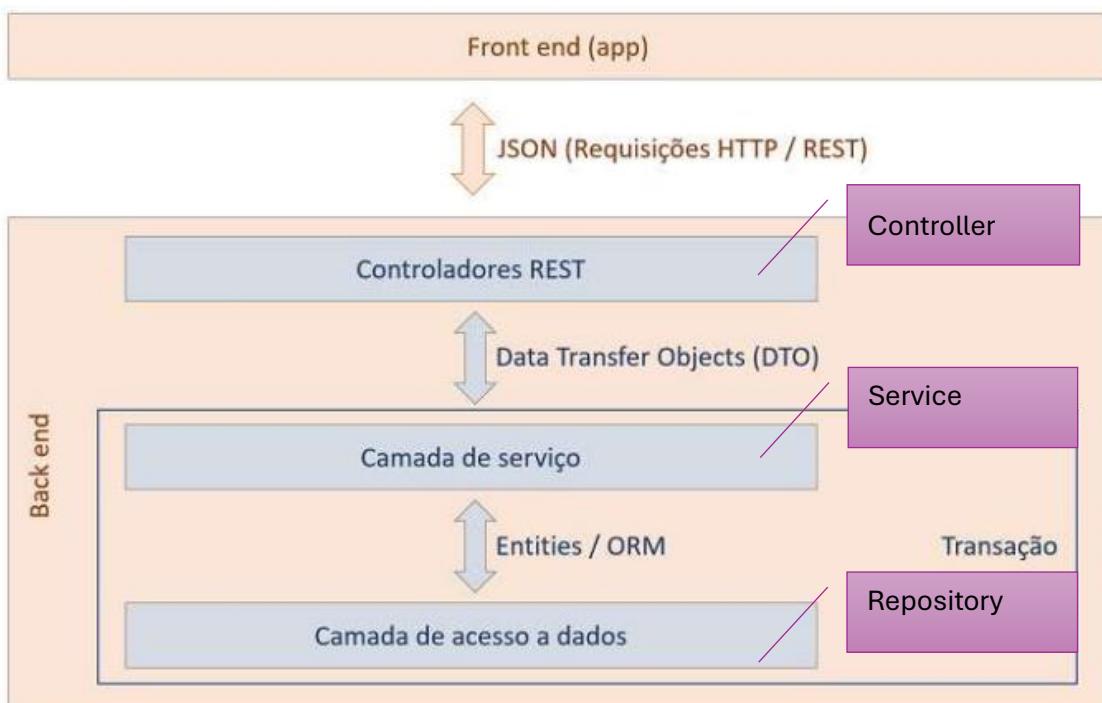
Criamos uma classe Usuario para representar o usuário em sua totalidade. Essa classe é mapeada com ORM (Mapeia Classe com Tabela do Database).

Um usuário possui muitos atributos e não necessariamente precisamos deles a todo momento. Na hora do login, o sistema precisa apenas de id, email e senha. Na hora de um perfil, já é necessária as informações de nome e foto. Quando se trata do formulário de inscrição, ai sim utilizamos todos os atributos.

Suponha que precisaremos apenas do id, nome e email. Então, criamos uma classe UserMinDto com os campos id, nome e email (o Min vem de mínimo – uma notação pra indicar que devolve um conjunto específico de atributos).

O UserMinDto pode ser usado para transferir dados de um formulário de cadastro para um Controller. O Controller pode então chamar o service, que transforma o DTO em uma entidade e o passa para o repository. O repository executa uma consulta no banco de dados e devolve o objeto ORM para o service. O service transforma o objeto ORM novamente em DTO e o devolve para o controller.

Padrão camadas





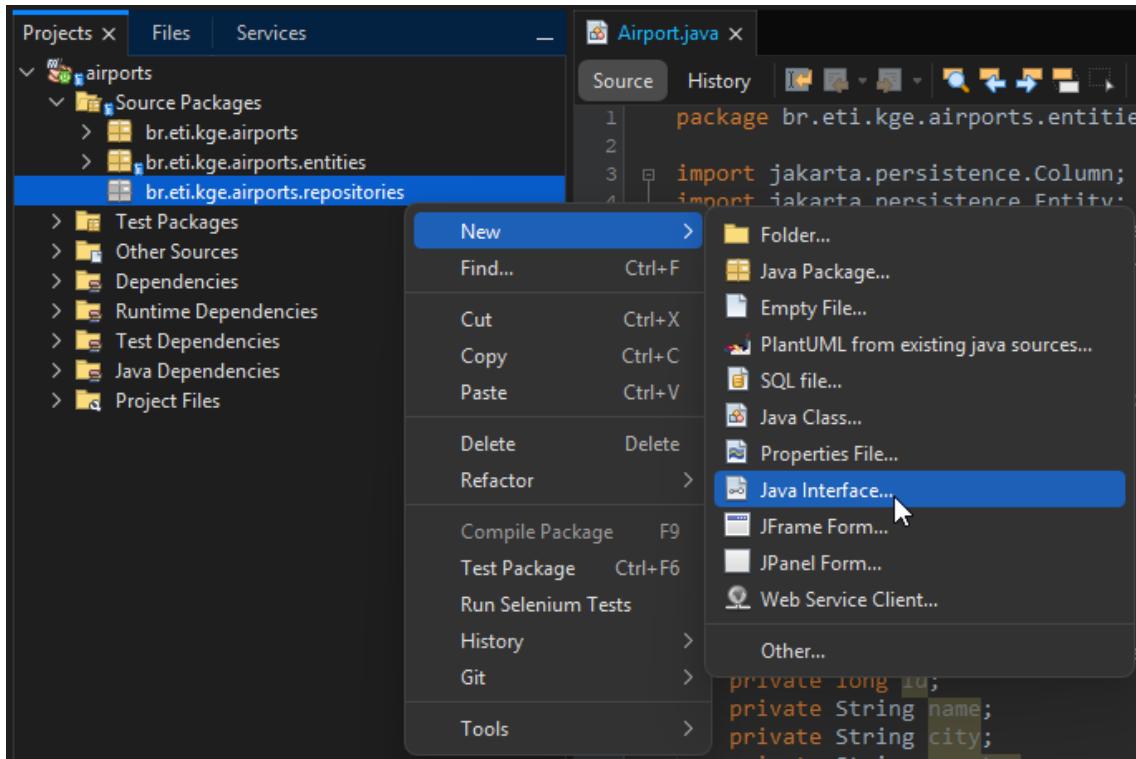
O mais legal de toda essa estrutura é que com o Framework nós só se concentramos nos objetos do escopo. Toda essa manobra de ORM -> DTO -> DTO -> ORM é feita automaticamente!.

Bora codar!

Interface AirportRepository

Crie um package chamado .repositories.

Dentro desse package crie a **interface AirportRepository**.



Caso não apareça a opção “Java Interface”, clique em Other e procure na categoria “Java” o FileType “Java Interface”.

Edito o arquivo para que fique conforme o exemplo:

```
14 |     /**
15 |     *
16 |     * @author KGe
17 |     */
18 |     public interface AirportRepository extends JpaRepository<Airport, Long> {
19 |
20 | }
```

Importe Airport de seu projeto.

Importe JpaRepository de org.springframework.data.jpa.repository.JpaRepository.

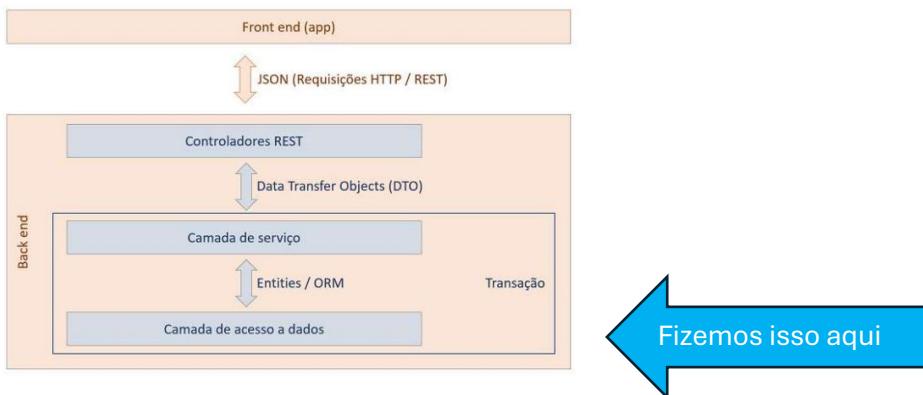
Só essa interface sem nada dentro já vai criar um componente para trazer funcionalidades de CRUD (Create, Read, Update e Delete) na tabela! (Lembra no outro exemplo o trabalho que deu fazer a classe DAO? (Vide exemplo do DukeMarket)).

Por isso a importância do framework!!! Agiliza demais o processo!!!

Esse é o componente da camada de acesso a dados. É um componente de sistema!

Será ser injetado nas classes que vão depender dele.

Verifique se tem erros, corrija-os se necessário, salve e feche o arquivo.



Classe AirportService

Crie um package .service

Dentro desse package, crie a **classe AirportService**. Edite o conteúdo conforme exemplo abaixo:

```

22  @Service
23  public class AirportService {
24
25      @Autowired
26      private AirportRepository airportRepository;
27
28      public List<Airport> findAll() {
29
30          List<Airport> result = airportRepository.findAll();
31          return result;
32
33      }
34
35  }

```

Fique atento novamente as anotações!

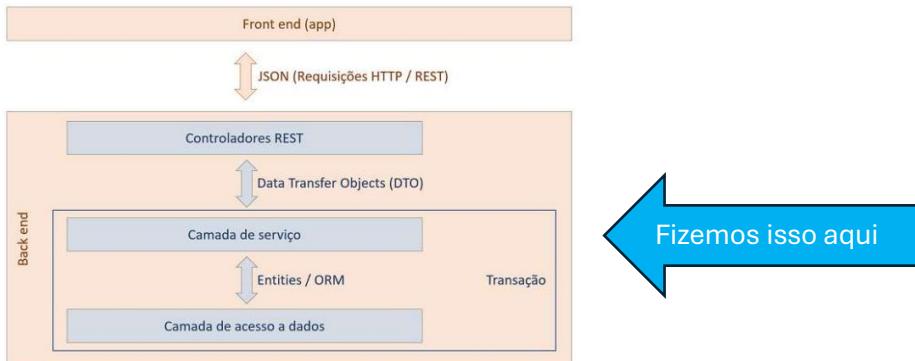
Faça os imports de:

- `.entities.Airport;`
- `.repositories.AirportRepository;`
- `java.util.List;`

- org.springframework.beans.factory.annotation.Autowired;
- org.springframework.stereotype.Service;

@Service indica que uma classe contém lógica de negócios.

@Autowired injeta automaticamente, neste caso, a interface AirportRepository. Seria uma abreviação de “AirportRepository airportRepository = new AirportRepository();”. Ele injeta a dependência dentro da classe.



Exemplificando quem é quem até agora:

Repository: Camada de acesso a dados: faz a consulta no banco.

Service: Implementa a regra de negócio (consulta/atualiza banco usando o repository).

Respeitando a arquitetura, Service devolve *DTO* (Veja o return do método findAll() 😊)

Bora que agora vamos subir na camada e fazer o controller!

Classe AirportController

Crie o package .controllers

Dentro do package criado, crie a classe AirportController

Esse será a porta de entrada do BackEnd.

Edite o conteúdo da classe conforme o exemplo:



```
18  /**
19   * Controller de Airports
20   * @author KGe
21   */
22
23  @RestController
24  public class AirportController {
25
26      @Autowired
27      private AirportService airportService;
28
29      /**
30       * Endpoint /airports/airport
31       * Retorna TODOS os aeroportos da base de dados.
32       * @return
33       */
34      @GetMapping("/airport")
35      public List<Airport> findAll() {
36          List<Airport> result = airportService.findAll();
37          return result;
38      }
39
40  }
```

Importe as dependências de:

- org.springframework.beans.factory.annotation.Autowired;
- org.springframework.web.bind.annotation.RestController;
- .entities.Airport;
- .services.AirportService;
- import java.util.List;
- import org.springframework.beans.factory.annotation.Autowired;
- import org.springframework.web.bind.annotation.GetMapping;
- import org.springframework.web.bind.annotation.RestController;



Agora presta atenção nas notações que foram utilizadas:

- Linha 23 – @RestController
Anuncia ao Spring que essa classe vai ser um Controller – e que haverá Endpoints nela.
- Linha 26 - @AutoWired
Já explicado. Vai injetar a dependência automaticamente.
- Linha 34 - @GetMapping("/airport")
Indica que o método logo abaixo responde ao endpoint /airports/airport.

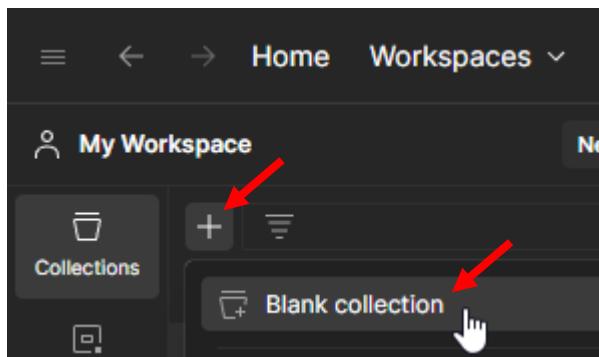
Atenção:



Os números das linhas são referentes ao print do código.
Não importa se no seu código as linhas não baterem com exemplo.

Testes

No postman, no seu Workspace, crie uma Collection



Dê o nome de AirportsAPI

The screenshot shows the Postman interface with the 'Airports API' collection selected. The collection name 'Airports API' is visible at the top. Below it, there's a brief description: 'Make things easier for your teammates with a complete collection description.' On the right side, it shows 'Created by You' and 'Created on 11 Feb 2025, 10:25 AM'. At the bottom, there's a link 'View complete documentation →'.

Nessa sua collection, crie uma request “Airports” apontando para o nosso endpoint que acabamos de criar:



▼ Airports API

This collection is empty
Add a request to start working.

Preencha a URI com o método GET e a URI do seu endpoint e clique em SEND

Essa deve ser a resposta.

Airports API | GET New Request | + | No environment | Save | Share

HTTP Airports API / New Request

GET http://localhost:8080/airports/airport Send

Params Auth Headers (7) Body Scripts Tests Settings Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

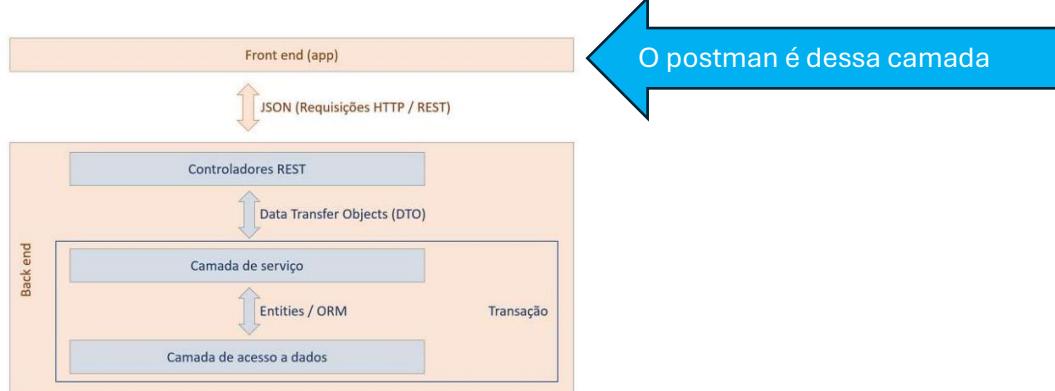
Body Cookies Headers (5) Test Results | 200 OK • 229 ms • 1.93 KB • ⓘ • ⏺ • ⌂

Pretty Raw Preview Visualize JSON

```
1 [  
2 {  
3   "id": 1,  
4   "name": "Guarulhos - Governador André Franco Montoro International Airport",  
5   "city": "Sao Paulo",  
6   "country": "Brazil",  
7   "iataCode": "GRU",  
8   "icaoCode": "SBGR",  
9   "latitude": -23.4355641,  
10  "longitude": -46.47305679,  
11  "altitude": 2459.0,  
12  "offsetFromUTC": -3.0,  
13  "dstCode": "S",  
14  "timezone": "America/Sao_Paulo"  
15 },  
16 {  
17   "id": 2,  
18   "name": "Campo de Marte Airport",  
19   "city": "Sao Paulo",  
20   "country": "Brazil",  
21   "iataCode": "\N",  
22   "icaoCode": "SBNT",  
23   "latitude": -23.50909996,  
24   "longitude": -46.63779831,  
25   "altitude": 1000.0  
]
```



Airports API



Parabéns! Seu primeiro endpoint já está funcionando!!!

Perceba que os aeroportos cadastrados na base de dados estão sendo relacionados no formato JSON.

Controle de Versionamento

Atenção:



Apenas quando terminar a fase e os testes derem Ok!

Abra o prompt powershell na raiz do seu projeto.

Realize as seguintes instruções:

```
git add *  
  
git commit -m "Término da Fase 2  
-Implementado endpoint Airport  
-Add interface AirportRepository  
-Add classe AirportService  
-Add classe AirportController"  
  
git push origin main
```

Bora para a fase 3. 😊