



EducaCiência FastCode

Fala Galera,

- Artigo: 50/2021 Data: Março/2021
- Público-alvo: Desenvolvedores – Iniciantes e Intermediários
- Tecnologia: Java + C#
- Tema: Artigo 50 - API SpringBoot_ConsumeCSharp - Post
- Link: <https://github.com/perucello/DevFP>

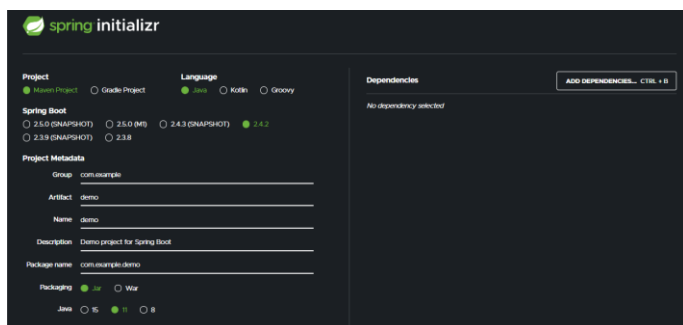
Neste artigo, criaremos uma API em Java com framework SpringBoot e criaremos um ConsoleApp em C# onde consumiremos os dados dos métodos HTTP da nossa API Java.

- ✓ **Banco de Dados => H2**
- ✓ **API => Java SpringBoot 2.4.2 | Repository JPA**
- ✓ **ConsoleApp => C# .NET Framework 4.7.2**

Lembrando que os fins são didáticos !

Para este ambiente , utilizaremos do Banco de Dados H2 , porém, virtual e assim já deixaremos um script de inserção dos dados no próprio código para que ao iniciarmos ele já esteja válido para manipularmos os dados de maneira objetiva.

Criaremos nosso projeto utilizando do link - <https://start.spring.io/> e abriremos na IDE Spring Tool Suíte 3 “STS”.





Com nosso Projeto já aberto no STS , iremos preparar nosso arquivo “pom”, ou seja, nossas dependências que iremos trabalhar no projeto.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.4.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.project.jpa</groupId>
    <artifactId>JavaJPA</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>JavaJPA</name>
    <description>Spring Boot</description>
    <properties>
        <java.version>1.8</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>javax.validation</groupId>
            <artifactId>validation-api</artifactId>
            <version>1.1.0.Final</version>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```



Feito isso, vamos atualizar nosso Maven – para isso, basta executar “Maven Install”.

Vamos preparar agora nossos “pacotes” como demonstrados abaixo:

```
✓ JPA_Project_H2 [boot] [devtools] [JPA_Project_H2 main]
  ✓ src/main/java
    > com.project.jpa.JavaJPA
      > com.project.jpa.JavaJPA.controller
      > com.project.jpa.JavaJPA.model
      > com.project.jpa.JavaJPA.repository
    > src/main/resources
    > src/test/java
    > JRE System Library [JavaSE-1.8]
    > Maven Dependencies
    > bin
    > src
      > target
      mvnw
      mvnw.cmd
      > pom.xml
```

Feito isso, vamos criar nossas classes sendo:

```
✓ JPA_Project_H2 [boot] [devtools]
  ✓ src/main/java
    ✓ com.project.jpa.JavaJPA
      > JavaJpaApplication.java
    ✓ com.project.jpa.JavaJPA.controller
      > ClientesController.java
    ✓ com.project.jpa.JavaJPA.model
      > Cliente.java
    ✓ com.project.jpa.JavaJPA.repository
      > Clientes.java
  > src/main/resources
  > src/test/java
  > JRE System Library [JavaSE-1.8]
  > Maven Dependencies
  > bin
  > src
    > target
    mvnw
    mvnw.cmd
    pom.xml
```



Vamos detalhar:

- a) **Controller** – neste pacote teremos nossa classe “CientesController” onde será escrito nosso código que manipularemos os métodos CRUD e nosso endpoint;
- b) **Model** – neste pacote teremos nossa classe “Cliente” onde criaremos a estrutura da nossa tabela Cliente no Banco de Dados
- c) **Repository** – teremos nossa “interface” que se estenderá da classe “Cliente” e receberá nosso Repository JPA.

Primeiramente, vamos criar a estrutura da nossa tabela, como a seguir:

```
package com.project.jpa.JavaJPA.model;
```

```
import javax.persistence.Entity;  
import javax.persistence.GeneratedValue;  
import javax.persistence.GenerationType;  
import javax.persistence.Id;  
import javax.validation.constraints.NotNull;
```

```
@Entity  
public class Cliente {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String nome;  
  
    @NotNull  
    private String email;  
  
    public Cliente() {  
        super();  
    }  
  
    public Cliente(Long id, String nome, String email) {  
        super();  
        this.id = id;  
        this.nome = nome;  
        this.email = email;  
    }  
  
    public Long getId() {  
        return id;  
    }  
  
    public void setId(Long id) {  
        this.id = id;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {
```



```

        this.email = email;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Cliente other = (Cliente) obj;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }
}

```

Agora, podemos estender nossa interface, para isso , abra o pacote Repositório e clique em Clientes.java e insira o seguinte código:

```

package com.project.jpa.JavaJPA.repository;

import org.springframework.data.jpa.repository.JpaRepository;




import com.project.jpa.JavaJPA.model.Cliente;

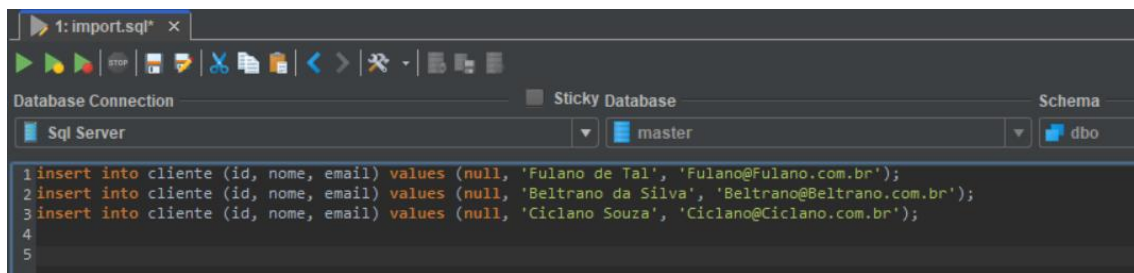
public interface Clientes extends JpaRepository<Cliente, Long> {

}

```

Feito isso, temos a estrutura do nosso banco de dados (tabela) pronta e agora , para que ao iniciarmos nossa aplicação do banco de dados relacional “H2” , vamos deixar um código sql em “resources” , para isso crie um arquivo chamado import.sql com os seguintes comandos descritos abaixo e “cole-o” em src/main/resources:

 src/main/resources
 application.properties
 import.sql



Feito isso, ao iniciarmos nossa API, estes dados já serão inseridos em nosso Banco de Dados H2.

Com a nossa estrutura do Banco de Dados criado, (entidade e repositório) agora, podemos focar em nosso CRUD.

Para este trabalho, abra o Script **ClientesController** que está no pacote Controller e insira os seguintes códigos:

```
package com.project.jpa.JavaJPA.controller;

import java.util.List;
import java.util.Optional;

import javax.validation.Valid;

import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.project.jpa.JavaJPA.model.Cliente;
import com.project.jpa.JavaJPA.repository.Clientes;

@RestController
@RequestMapping("/api/JPA/clientes")
public class ClientesController {

    @Autowired
    private Clientes clientes;

    @GetMapping
    public List<Cliente> listar(){
        System.out.println("Quantidade de Registros de Clientes : " + clientes.count());
        return clientes.findAll();
    }

    @PostMapping("/add")
    public Cliente adicionar(@Valid @RequestBody Cliente cliente) {
        return clientes.save(cliente);
    }
}
```



```
@GetMapping("/{id}")
public ResponseEntity<Optional<Cliente>> buscar(@PathVariable Long id){
    Optional<Cliente> cliente = clientes.findById(id);
    if (clientes == null) {
        return ResponseEntity.notFound().build();
    }
    return ResponseEntity.ok(cliente);
}

@PutMapping("/{id}")
public ResponseEntity<Object> atualizar(@PathVariable Long id, @Valid @RequestBody
Cliente cliente)
{
    Object atualizar = clientes.findById(id);
    if (atualizar == null) {
        return ResponseEntity.notFound().build();
    }
    BeanUtils.copyProperties(cliente, atualizar, "id");
    atualizar = clientes.save(cliente);
    return ResponseEntity.ok(atualizar);
}

@DeleteMapping("delete/{id}")
public ResponseEntity<Void> deletar(@PathVariable Long id){
    Optional<Cliente> cliente = clientes.findById(id);
    if(cliente != null) {
        clientes.deleteById(id);
    }
    return ResponseEntity.noContent().build();
}

}
```

Com isso , temos nosso CRUD pronto onde os métodos que manipularemos são os seguintes:

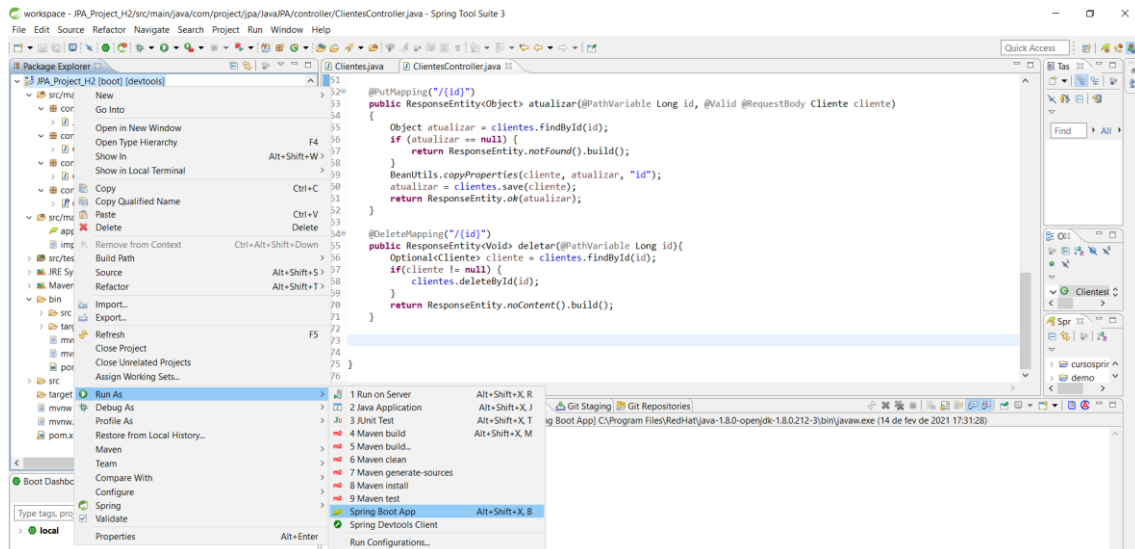
- ⇒ **Get** – este método retornará todos os dados contidos na nossa tabela (Select)
- ⇒ **Get id** – este método retornará os dados de acordo com parâmetro passado , no nosso caso “id” , uma busca por “id” (Select)
- ⇒ **Post** – este método irá realizar a inserção dos dados em nossa tabela, ou seja, nosso Insert.
- ⇒ **Put** – este método irá realizar a atualização de um determinado registro em nossa tabela, ou seja, nosso Update.
- ⇒ **Delete** – este método irá deletar determinado registro por id, ou seja, nosso Delete.

Agora, basta iniciarmos nossa “Aplicação”





⇒ Run As \ Spring Boot App



Nossa aplicação se iniciou como esperado.

Agora, iniciaremos a criação do nosso ConsoleApp em C# , para este artigo, utilizei do Visual Studio Professional 2019.

Sobre Microsoft Visual Studio

?

Visual Studio

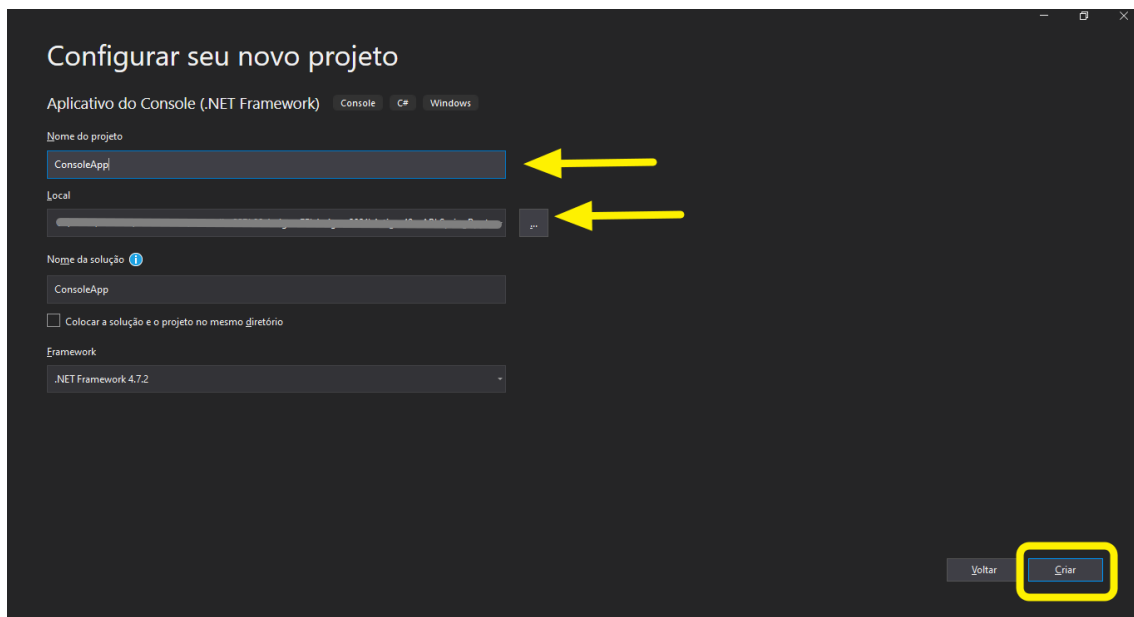
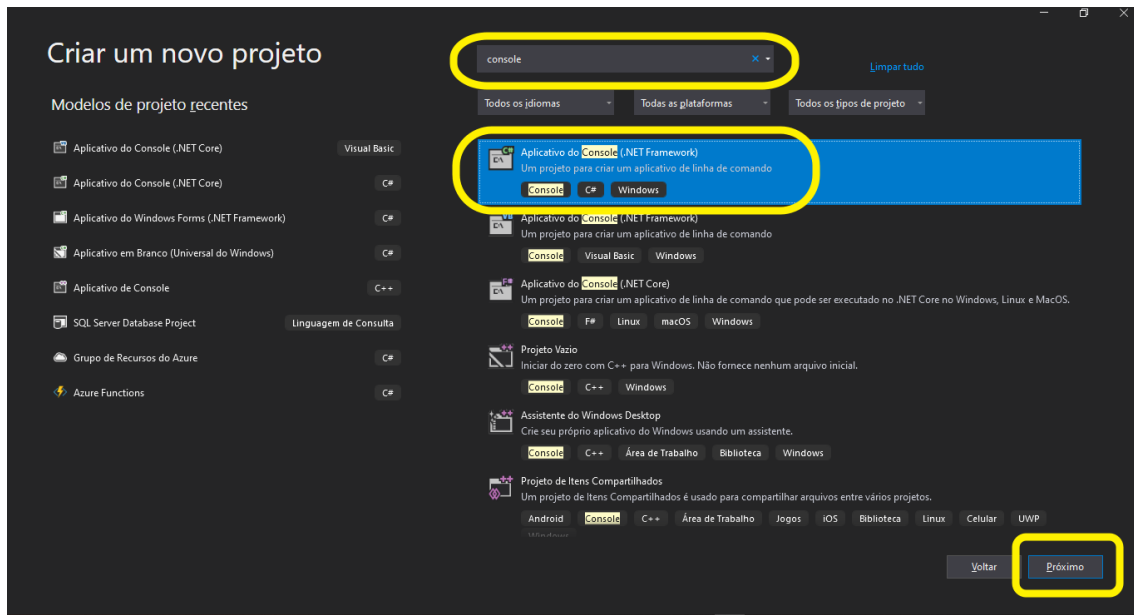
[Status da licença](#)
[Termos da licença](#)

Microsoft Visual Studio Professional 2019
Versão 16.6.4
© 2020 Microsoft Corporation.
Todos os direitos reservados.

Microsoft .NET Framework
Versão 4.8.04084
© 2020 Microsoft Corporation.
Todos os direitos reservados.

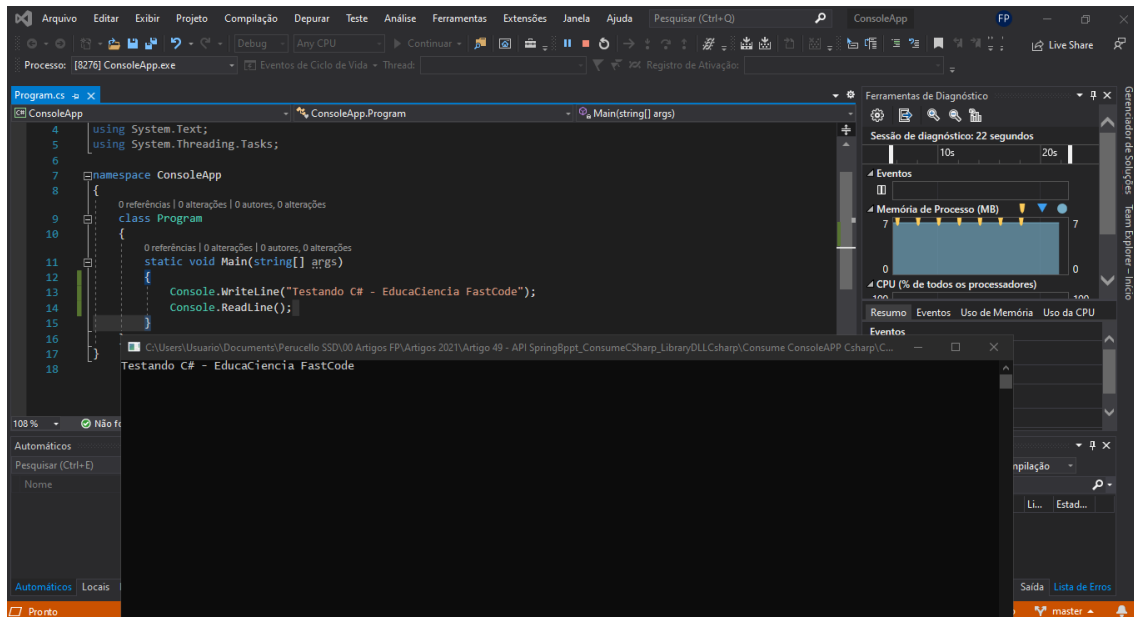


Vamos abrir o Visual Studio e criar nossa aplicação, para isso siga os passos:





Com sua aplicação criada, vamos fazer um teste para sabermos se está tudo correto, para isso insira os seguintes comandos e execute:



Bom, sabemos que nossa aplicação C# está ok, agora vamos dar seguimento à nossa proposta, iremos agora, em **Program.cs** criar nossos métodos que acessarão nossa API Java e testarmos os métodos HTTP que criamos na API SpringBoot, porém, executando pelo C# - ConsoleApp.

Para isso, insira o seguinte código na classe **Program.cs**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net; //importar biblioteca Client
using System.IO;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {

        }

        //Post API Java SpringBoot - WebRequest
        public static void PostAPI_Java()
        {
            string endpointPost = "http://127.0.0.1:8080/api/JPA/clientes/add/";
            WebRequest request = WebRequest.Create(endpointPost);
            request.Method = "POST";
            request.ContentType = "application/json; charset=UTF-8";
            string json =
                "{" +
                "\"nome\":\"EducaCienciaFastCode\"," +
                "\"email\":\"EducaCiencia@FastCode.com.br\"" +
            }
```





```
    }";

    var byteArray = Encoding.UTF8.GetBytes(json);
    request.ContentLength = byteArray.Length;

    Stream stream = request.GetRequestStream();
    stream.Write(byteArray, 0, byteArray.Length);
    stream.Close();

    var response = (HttpWebResponse)request.GetResponse();
    if (((HttpWebResponse)response).StatusCode == HttpStatusCode.OK)
    {
        Console.WriteLine("Post => ok");
        Console.WriteLine(json);
    }
    else
    {
        Console.WriteLine("fail Post");
    }
}

}
```

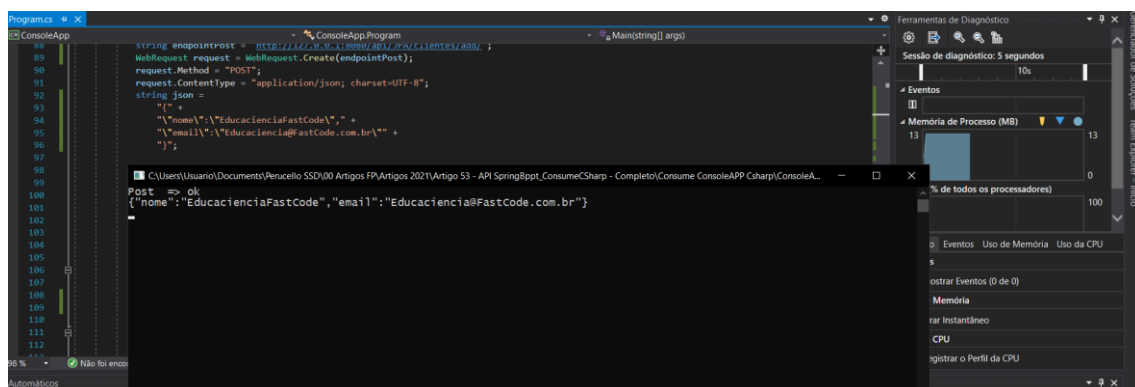
Feito isso, para que possamos testar, precisamos executar nossos métodos, sendo assim vamos agora fazer por parte.

No nosso método main, chamaremos a função que criamos em C# para que seja invocado e retorne os dados da nossa API em Java que já iniciamos anteriormente, para isso faça:

Testando Método Post

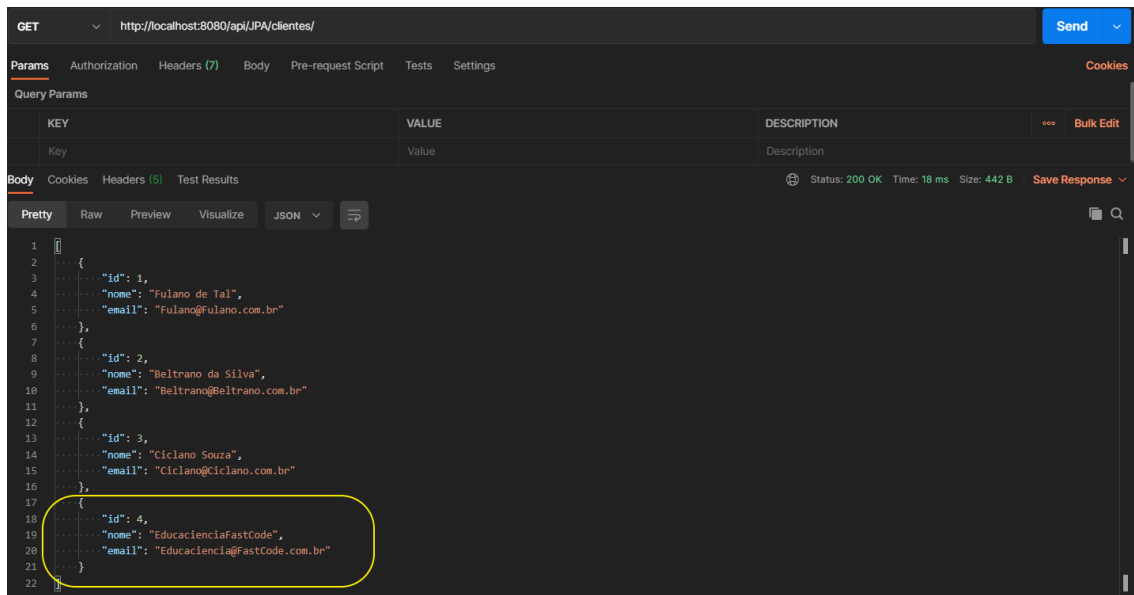
Método Main – ConsoleApp C#

```
static void Main(string[] args)
{
    //PostAPI_Java
    PostAPI_Java();
    Console.ReadKey();
}
```





Postman



Nossa aplicação C# funcionou corretamente retornando os dados da API criada em Java, consideramos que nossa aplicação funcionou de acordo com a nossa proposta, sendo assim finalizamos este artigo, onde os códigos estão disponíveis no GitHub para consumo.

Até mais !
Espero ter ajudado !

