Construindo e Refatorando uma API com Spring Boot (2.x.x → 3.x.x) e Consumindo com C# .NET

Desenvolvimento de uma API REST em Spring Boot e Consumo via C# ConsoleApp

Este artigo apresenta um **procedimento técnico detalhado** para criar uma **API RESTful usando Spring Boot (Java)** e consumi-la via **ConsoleApp em C#**.

O objetivo é demonstrar como integrar sistemas de tecnologias diferentes, utilizando **Spring Boot para back-end** e **.NET Framework para consumo de serviços**.

O conteúdo foi elaborado para a comunidade **EducaCiência FastCode**, atendendo a desenvolvedores de **nível iniciante e intermediário** interessados em aprender **boas práticas na construção e consumo de APIs REST**.

1. Criando a API REST com Spring Boot

A API será desenvolvida com **Spring Boot**, utilizando **JPA para persistência de dados** no banco **H2**.

1.1 Configuração do Projeto

Acesse Spring Initializr e configure o projeto:

- **Group:** com.project.jpa
- **Artifact:** JavaJPA
- **Dependencies:** Spring Web, Spring Data JPA, H2 Database, Validation API

Baixe o projeto e abra na **Spring Tool Suite (STS)**.

1.2 Configuração do Maven (pom.xml)

Inclua as dependências no arquivo pom.xml:

```xml
<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>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>1.1.0.Final</version>
    </dependency>
</dependencies>
```

Atualize o Maven com **Maven Install**.

1.3 Criando o Modelo de Dados (Cliente.java)

Crie a classe **Cliente** no pacote com.project.jpa.JavaJPA.model:

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

import javax.persistence.*;
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() {}

    public Cliente(Long id, String nome, String email) {
        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; }
}
```

1.4 Criando o Repositório (Clientes.java)

Crie a interface Clientes no pacote com.project.jpa.JavaJPA.repository:

```
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> {
}
```

1.5 Criando o Controlador REST (ClientesController.java)

Adicione todos os métodos HTTP na API

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

import java.util.List;
import java.util.Optional;
import javax.validation.Valid;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

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() {
        return clientes.findAll();
    }

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

    @PostMapping("/add")
    public Cliente adicionar(@Valid @RequestBody Cliente cliente) {
```

```
        return clientes.save(cliente);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Cliente> atualizar(@PathVariable Long id, @Valid @RequestBody
Cliente cliente) {
        if (!clientes.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        cliente.setId(id);
        return ResponseEntity.ok(clientes.save(cliente));
    }

    @DeleteMapping("/delete/{id}")
    public ResponseEntity<Void> deletar(@PathVariable Long id) {
        if (!clientes.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        clientes.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

2. Criando o ConsoleApp em C# para Consumir a API

2.1 Criando o Projeto

1. **Abra o Visual Studio**.
2. **Crie um novo projeto** do tipo **Console Application**.
3. Nomeie como ConsoleApp.

2.2 Implementação do Código

Crie os métodos para consumir a API.

```
using System;
using System.Net;
using System.IO;
using System.Text;

namespace ConsoleApp
{
    class Program
    {
        static void Main(string[] args)
        {
            GetAPI();
            GetByIdAPI();
            PostAPI();
            PutAPI();
            DeleteAPI();
        }

        public static void GetAPI()
        {
            string endpoint = "http://localhost:8080/api/JPA/clientes/";
```

```
        WebClient api = new WebClient();
        string content = api.DownloadString(endpoint);
        Console.WriteLine("GET: " + content);
    }

    public static void GetByIdAPI()
    {
        string endpoint = "http://localhost:8080/api/JPA/clientes/1";
        WebClient api = new WebClient();
        string content = api.DownloadString(endpoint);
        Console.WriteLine("GET by ID: " + content);
    }

    public static void PostAPI()
    {
        string endpoint = "http://localhost:8080/api/JPA/clientes/add/";
        WebRequest request = WebRequest.Create(endpoint);
        request.Method = "POST";
        request.ContentType = "application/json";

        string json = "{\"nome\":\"João\",\"email\":\"joao@email.com\"}";
        byte[] data = Encoding.UTF8.GetBytes(json);
        request.GetRequestStream().Write(data, 0, data.Length);

        WebResponse response = request.GetResponse();
        Console.WriteLine("POST OK");
    }

    public static void PutAPI()
    {
        string endpoint = "http://localhost:8080/api/JPA/clientes/1";
        WebRequest request = WebRequest.Create(endpoint);
        request.Method = "PUT";
        request.ContentType = "application/json";

        string json = "{\"id\":1,\"nome\":\"Carlos\",\"email\":\"carlos@email.com\"}";
        byte[] data = Encoding.UTF8.GetBytes(json);
        request.GetRequestStream().Write(data, 0, data.Length);

        WebResponse response = request.GetResponse();
        Console.WriteLine("PUT OK");
    }

    public static void DeleteAPI()
    {
        string endpoint = "http://localhost:8080/api/JPA/clientes/delete/1";
        WebRequest request = WebRequest.Create(endpoint);
        request.Method = "DELETE";

        WebResponse response = request.GetResponse();
        Console.WriteLine("DELETE OK");
    }
  }
}
```

Este artigo demonstrou como criar uma **API RESTful** em **Spring Boot** e consumi-la em um **ConsoleApp em C#**. A abordagem permite integração entre aplicações de linguagens diferentes, fornecendo soluções eficientes para sistemas distribuídos.

Refatoração da API para Spring Boot 3.x.x

Objetivo da Refatoração

A atualização da API para **Spring Boot 3.x.x** garante **melhor desempenho, suporte a Jakarta EE 9+ e compatibilidade com as versões mais recentes do Java**.
Além disso, essa refatoração:

- **Atualiza dependências** para usar Jakarta EE 9+ (substituindo pacotes javax por jakarta).
- **Melhora a segurança e as boas práticas** com a nova abordagem de APIs REST.
- **Garante suporte a Java 17**, que é a versão mínima recomendada pelo Spring Boot 3.

1. Atualizando o Projeto para Spring Boot 3.x.x

Acesse Spring Initializr e configure o projeto:

- **Group:** com.project.jpa
- **Artifact:** JavaJPA
- **Spring Boot Version: 3.x.x**
- **Dependencies:** Spring Web, Spring Data JPA, H2 Database, Validation, Lombok

1.1 Atualizando o pom.xml

Substituímos as dependências antigas por versões compatíveis com Spring Boot 3:

```
<properties>
    <java.version>17</java.version>
</properties>

<dependencies>
    <!-- Spring Boot Web -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Spring Boot Data JPA -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
```

```
    <!-- H2 Database -->
    <dependency>
        <groupId>com.h2database</groupId>
        <artifactId>h2</artifactId>
        <scope>runtime</scope>
    </dependency>

    <!-- Jakarta Validation -->
    <dependency>
        <groupId>jakarta.validation</groupId>
        <artifactId>jakarta.validation-api</artifactId>
    </dependency>

    <!-- Lombok -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Mudanças

- **Java 17:** Agora é o mínimo exigido pelo Spring Boot 3.
- **Pacotes javax foram substituídos por jakarta** (exemplo: javax.validation → jakarta.validation).
- **Lombok** foi adicionado para reduzir código boilerplate.

2. Refatorando a Classe Modelo (Cliente.java)

Antes, a classe continha **getters, setters e construtores manuais**. Agora, com **Lombok**, podemos simplificá-la.

Código Refatorado

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

import jakarta.persistence.*;
import jakarta.validation.constraints.NotNull;
import lombok.*;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nome;

    @NotNull
    private String email;
}
```

Mudanças

- **@Getter e @Setter**: Geram automaticamente os métodos de acesso.
- **@NoArgsConstructor e @AllArgsConstructor**: Criam construtores sem e com argumentos.
- **@EqualsAndHashCode**: Gera automaticamente métodos equals() e hashCode().

3. Refatorando o Repositório (Clientes.java)

O repositório permanece praticamente inalterado:

```
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> {
}
```

Explicação

- **Nenhuma mudança foi necessária**, pois JpaRepository já está atualizado para o Spring Boot 3.

4. Refatorando o Controlador (ClientesController.java)

Antes, o código usava Optional<Cliente> diretamente nas respostas. Agora, usamos a nova **API de ResponseEntity** corretamente.

Código Refatorado

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

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

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

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

import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;

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

    private final Clientes clientes;

    @GetMapping
```

```
    public ResponseEntity<List<Cliente>> listar() {
        return ResponseEntity.ok(clientes.findAll());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Cliente> buscar(@PathVariable Long id) {
        return clientes.findById(id)
                .map(ResponseEntity::ok)
                .orElseGet(() -> ResponseEntity.notFound().build());
    }

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

    @PutMapping("/{id}")
    public ResponseEntity<Cliente> atualizar(@PathVariable Long id, @Valid @RequestBody
Cliente cliente) {
        if (!clientes.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        cliente.setId(id);
        return ResponseEntity.ok(clientes.save(cliente));
    }

    @DeleteMapping("/delete/{id}")
    public ResponseEntity<Void> deletar(@PathVariable Long id) {
        if (!clientes.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        clientes.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

Mudanças

- **@RequiredArgsConstructor**: Injeta automaticamente o repositório via construtor.
- **ResponseEntity.ok()**: Retorna respostas HTTP padronizadas.
- **orElseGet(() -> ResponseEntity.notFound().build())**: Simplifica a lógica do GET by ID.

5. Testando a API

Execute a aplicação com:

```
mvn spring-boot:run
```

Os endpoints podem ser testados no **Postman** ou via **cURL**.

Exemplo de Teste GET:

```
curl -X GET http://localhost:8080/api/JPA/clientes/
```

6. Refatorando o ConsoleApp em C#

O código do ConsoleApp permanece quase inalterado, mas **modernizamos a chamada de requisições**.

Código Refatorado (Program.cs)

```
using System;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp
{
    class Program
    {
        private static readonly HttpClient client = new HttpClient();

        static async Task Main(string[] args)
        {
            await GetAPI();
            await GetByIdAPI(1);
            await PostAPI();
            await PutAPI(1);
            await DeleteAPI(1);
        }

        private static async Task GetAPI()
        {
            var response = await client.GetStringAsync("http://localhost:8080/api/JPA/clientes/");
            Console.WriteLine("GET: " + response);
        }

        private static async Task GetByIdAPI(int id)
        {
            var response = await
client.GetStringAsync($"http://localhost:8080/api/JPA/clientes/{id}");
            Console.WriteLine("GET by ID: " + response);
        }

        private static async Task PostAPI()
        {
            var json = "{\"nome\":\"João\",\"email\":\"joao@email.com\"}";
            var content = new StringContent(json, Encoding.UTF8, "application/json");
            var response = await client.PostAsync("http://localhost:8080/api/JPA/clientes/add/",
content);
            Console.WriteLine("POST OK: " + response.StatusCode);
        }

        private static async Task PutAPI(int id)
        {
            var json = "{\"id\":1,\"nome\":\"Carlos\",\"email\":\"carlos@email.com\"}";
            var content = new StringContent(json, Encoding.UTF8, "application/json");
            var response = await client.PutAsync($"http://localhost:8080/api/JPA/clientes/{id}",
content);
            Console.WriteLine("PUT OK: " + response.StatusCode);
        }

        private static async Task DeleteAPI(int id)
```

```
        {
            var response = await
client.DeleteAsync($"http://localhost:8080/api/JPA/clientes/delete/{id}");
            Console.WriteLine("DELETE OK: " + response.StatusCode);
        }
    }
}
```

A atualização para **Spring Boot 3.x.x** melhorou a API, tornando-a mais moderna e eficiente.
Essa refatoração:

✓ **Tornou o código mais limpo e padronizado** com Lombok.
✓ **Utilizou boas práticas modernas de REST API.**
✓ **Atualizou as chamadas HTTP no C# para HttpClient assíncrono.**

Isso garante maior **performance, segurança e compatibilidade** para aplicações **Java + .NET**.

Abraços

EducaCiência FastCode para a comunidade