Passo a Passo para Implementação da API em .NET

Criar o Projeto

Primeiro, abra um terminal e execute o seguinte comando para criar um novo projeto **Minimal API**:

```
dotnet new webapi -minimal -n OrderManagementAPI
```

> **Explicação**:
>
> - dotnet new webapi → Cria um novo projeto do tipo Web API.
> - -minimal → Usa o modelo de **Minimal APIs** sem controllers.
> - -n OrderManagementAPI → Define o nome do projeto.

Depois, acesse a pasta do projeto recém-criado e acesse o VS Code tendo essa pasta como base.

```
cd OrderManagementAPI
code .
```

Estrutura Inicial do Projeto

Ao criar o projeto, a estrutura inicial será parecida com esta:

```
📂 OrderManagementAPI
 ├ 📄 Program.cs
 ├ 📄 Order.cs (a ser criado)
 ├ 📄 appsettings.json
 ├ 📂 Properties
 ├ 📂 bin
 ├ 📂 obj
```

Vamos adicionar os arquivos necessários para modelar a aplicação.

Criar o Modelo de Pedido (Order.cs)

No diretório principal do projeto, crie um arquivo chamado **Order.cs** com o seguinte conteúdo:

```
public class Order
{
    public int Id { get; set; }
    public string Customer { get; set; }
    public decimal Amount { get; set; }
}
```

Explicação:

- Essa classe representa um **Pedido**.
- Contém **Id**, **Customer** e **Amount** como propriedades.

Explicação do Erro (CS8618)

O erro **"Non-nullable property 'Customer' must contain a non-null value when exiting constructor"** ocorre porque a propriedade `Customer` da classe `Order` é do tipo `string`, e, em versões recentes do C# (com **nullable reference types ativados**), todas as **propriedades do tipo string são consideradas não anuláveis (non-nullable) por padrão**.

Como `Customer` não foi inicializada explicitamente, o compilador exige que seja atribuído um valor antes da saída do construtor, garantindo que nunca seja `null`.

Soluções para Corrigir o Erro

Aqui estão **três formas** de corrigir esse problema:

Inicializar a Propriedade com um Valor Padrão

Uma maneira simples e eficaz de resolver o erro é definir um valor inicial para `Customer`, garantindo que nunca seja `null`:

```
public string Customer { get; set; } = string.Empty;
```

✅ **Vantagem:** Resolve o erro rapidamente e evita `null`.
❌ **Desvantagem:** Pode mascarar situações onde um valor real deveria ser atribuído.

Tornar a Propriedade Nullable (`string?`)

Se a intenção for permitir valores nulos para `Customer`, basta adicionar um `?` ao tipo `string`, indicando que a propriedade pode ser nula:

```
public string? Customer { get; set; }
```

✅ **Vantagem:** Permite valores nulos quando necessário.
❌ **Desvantagem:** Pode exigir verificações extras para evitar `NullReferenceException`.

Usar o Modificador `required` (C# 11+)

Se você estiver usando **C# 11 ou superior**, pode usar o modificador `required`, que exige que `Customer` seja definido no momento da instanciação do objeto:

```
public required string Customer { get; set; }
```

✅ **Vantagem:** Garante que `Customer` **sempre** será preenchido ao criar um `Order`.
❌ **Desvantagem: Só funciona a partir do C# 11** e exige compatibilidade no projeto.

Melhor Opção para Este Caso

A opção **inicializar com** `string.Empty` é a mais prática e segura para evitar problemas de `null`, especialmente em APIs.

Se a intenção for garantir que `Customer` **sempre tenha um valor**, a opção `required` é a melhor escolha para evitar inicializações erradas.

Se a aplicação permitir que um **pedido seja criado sem a necessidade de um cliente identificado imediatamente**, faz sentido que `Customer` possa ser nulo.

Implementar os Endpoints no Program.cs

Abra o arquivo **Program.cs** e substitua o conteúdo pelo seguinte código:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

List<Order> orders = new()
{
    new Order { Id = 1, Customer = "John", Amount = 150.00m },
    new Order { Id = 2, Customer = "Mary", Amount = 200.50m }
};
```

```
//  Endpoint to get all orders
app.MapGet("/orders", () => orders);

//  Endpoint to get a specific order by ID
app.MapGet("/orders/{id}", (int id) =>
{
    var order = orders.FirstOrDefault(o => o.Id == id);
    return order is not null ? Results.Ok(order) : Results.NotFound("Order not found.");
});

//  Endpoint to add a new order
app.MapPost("/orders", (Order newOrder) =>
{
    newOrder.Id = orders.Count + 1; // Simulating a unique ID
    orders.Add(newOrder);
    return Results.Created($"/orders/{newOrder.Id}", newOrder);
});

//  Endpoint to delete an order
app.MapDelete("/orders/{id}", (int id) =>
{
    var order = orders.FirstOrDefault(o => o.Id == id);
    if (order is null) return Results.NotFound("Order not found.");

    orders.Remove(order);
    return Results.Ok("Order successfully removed.");
});

app.Run();
```

Explicação:

Armazena os pedidos em memória com uma List<Order>.
Define 4 endpoints:

- GET /orders → Lista todos os pedidos.
- GET /orders/{id} → Retorna um pedido específico.
- POST /orders → Adiciona um novo pedido.
- DELETE /orders/{id} → Remove um pedido.

Executar a API

Agora, execute o projeto com o seguinte comando:

```
dotnet run
```

📌 **Se tudo estiver correto, o terminal mostrará algo como:**

Now listening on: http://localhost:5000 Now listening on: https://localhost:5001

Isso significa que sua API está rodando nas portas **5000 (HTTP)** e **5001 (HTTPS)**.

Lista de Endpoints

Base URL:

```
http://localhost:5118
```

Obter todos os pedidos (GET)

```
http://localhost:5118/orders
```

Obter um pedido específico por ID (GET)

```
http://localhost:5118/orders/{id}
http://localhost:5118/orders/1
```

Se o ID não existir, retorna **404 Not Found**.

Criar um novo pedido (POST)

```
http://localhost:5118/orders
```

Adiciona um novo pedido. Deve ser enviado um **JSON** no **corpo da requisição**.

Exemplo de corpo da requisição (enviar via **Postman** ou `curl`):

```
{
   "customer": "Alice",
   "amount": 250.75
}
```

Exemplo de comando `curl` **para testar via terminal**:

```
curl -X POST http://localhost:5118/orders \
    -H "Content-Type: application/json" \
    -d '{"customer": "Alice", "amount": 250.75}'
```

Se tudo der certo, retorna **201 Created** com os detalhes do novo pedido.

Excluir um pedido por ID (DELETE)

```
http://localhost:5118/orders/{id}
http://localhost:5118/orders/2
curl -X DELETE http://localhost:5118/orders/2
```

Remove um pedido existente pelo ID.
Se o ID não existir, retorna **404 Not Found**.

Swagger para Testes

Se quiser testar os endpoints diretamente no **Swagger**, instale o pacote, restaure o projeto e ative o Swagger no `Program.cs`

```
dotnet add package Swashbuckle.AspNetCore --version 6.5.0
dotnet restore
---
// Adiciona suporte ao Swagger
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
---
// Habilita o Swagger na aplicação
app.UseSwagger();
app.UseSwaggerUI();
---
dotnet run
```

Então, acesse:

```
http://localhost:5118/swagger
```

Resumo

Método	Endpoint	Descrição
GET	`/orders`	Retorna todos os pedidos
GET	`/orders/{id}`	Retorna um pedido específico
POST	`/orders`	Cria um novo pedido
DELETE	`/orders/{id}`	Remove um pedido