PROGRAMAÇÃO I

AULA 6

Prof. Alan Matheus Pinheiro Araya

CONVERSA INICIAL

Consumindo *web services* e *web apis*

O objetivo desta aula é abordar como a linguagem C# é prática e simples no consumo e interação com *web services* ou *web APIs*, e como você pode utilizar os recursos da linguagem e mesmo de pacotes auxiliares para ganhar produtividade no seu dia a dia nesse tipo de tarefa.

TEMA 1 – INTRODUÇÃO A *WEB SERVICES* E *WEB APIS*

O C# e a plataforma .NET oferecem um conjunto muito poderoso de funcionalidades, bibliotecas e pacotes para que você possa construir aplicações *web*, *mobile*, *desktop* e mesmo para IoT e IA. No mundo atual, praticamente todas as aplicações de nível empresarial precisaram se comunicar por serviços *web*, chamados ***web services*** ou ***web APIs***.

É importante conceituarmos os dois termos logo no início de nossa aula. **API** é um termo utilizado para definir *application programming interface*, com o qual você pode basicamente expor comportamentos (métodos) para um cliente consumir. As bibliotecas de *threading* do .NET são APIs que disponibilizam o recurso do sistema operacional pela plataforma e linguagem. ***Web services* são tipos de API que operam via rede**, tipicamente utilizando protocolos como *simple object access protocol* (Soap), *representational state transfer* (Rest) ou XML-RPC.

O termo *web services* não é novo e começou a ser utilizado tão logo saíram as primeiras formas de comunicação de serviços *web*, no início baseados em XML e logo evoluindo para o protocolo **Soap**. Já o termo *web API* começou a ser empregado quando as aplicações *web* começaram a implementar o protocolo **Rest** como padrão. Ambos rodam em cima de HTTP (protocolo-padrão da *web*) e basicamente diferem na forma de "serializar", ou seja, de transformar as informações para que elas trafeguem via *web* de uma ponta à outra.

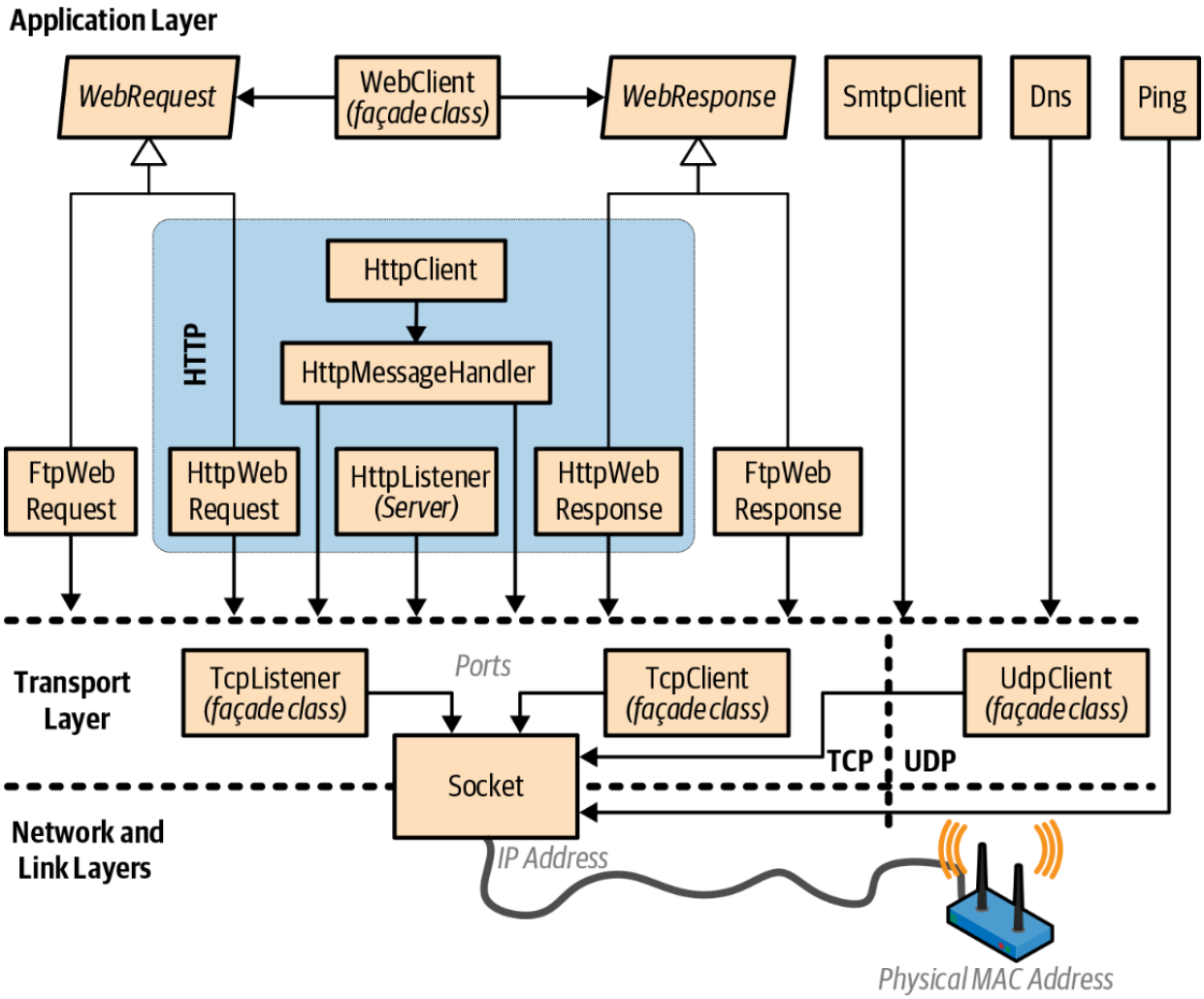
Por isso é comum vermos o termo *web services* associado a plataformas ou sistemas mais antigos, se comunicando pelo protocolo Soap, geralmente baseados em XML. Já o termo *web API* representa aplicações que utilizam verbos HTTP (*get*, *post*, *put*, *patch*, *delete* etc.) para orientar o tipo de "ação" de suas interfaces, quase como método de um código. Geralmente operam em cima

do protocolo Rest, por isso são chamados de *serviços RESTful*, ou seja, que implementam o padrão Rest.

1.1 Arquitetura de rede do .NET

A Figura 1 demonstra os tipos e camadas de comunicação de rede no .NET. A maioria dos tipos reside na camada de transporte ou na camada de aplicação. A camada de transporte define protocolos básicos para enviar e receber *bytes* (TCP e UDP); a camada de aplicação define protocolos de nível superior projetados para as aplicações receberem e enviarem dados via *web* (HTTP), transferência de arquivos (FTP), envio de e-mail (SMTP) e conversão entre nomes de domínio e endereços IP (DNS) (Albahari; Johannsen, 2020, p. 706).

Figura 1 – *Application layer*



Fonte: Albahari; Johannsen, 2020, p. 706.

Os serviços de rede do .NET estão sob o domínio do *namespace* **Sytem.Net.***, e você poderá utilizar o *HttpClient*, por exemplo, para realizar chamadas a serviços *web*, como *web APIs*.

TEMA 2 – TRABALHANDO COM *WEB SERVICES* SOAP

O padrão de protocolo Soap define uma série de metadados junto com sua "declaração". O modo de defini-los e declará-los é o *web services description language* (WSDL). Como, no início da utilização de *web services*, era muito comum os desenvolvedores ficarem inseguros quanto ao tipo do dado trafegado e quanto à sua formatação – uma vez que demoraram muitos anos até que a W3C formatasse as primeiras especificações formais do Soap –, logo que ele começou a se popularizar, também veio junto o WSDL como forma de padronizar os tipos de dados trafegados entre as aplicações. O WSDL é muito "verboso" propositalmente, pois no início os desenvolvedores criaram um tipo de "metalinguagem" em que eles pudessem entender o que as aplicações estavam gerando de dados entre elas.

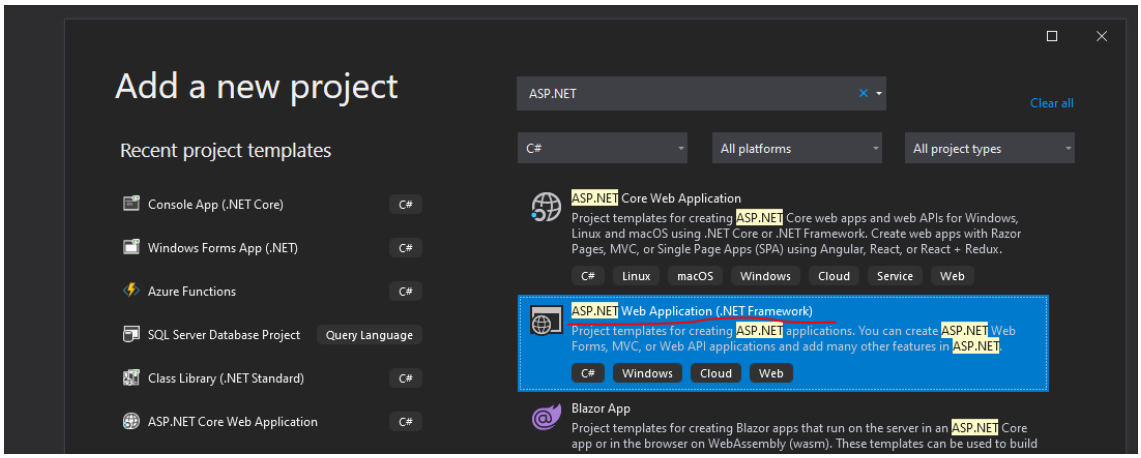
Para trabalhar com *web services* Soap no .NET, vamos precisar recorrer a algumas ferramentas do Visual Studio. Caso você esteja utilizando o VS Code, pode encontrar dificuldades em gerar o *client*, uma vez que as ferramentas que geram o código para serviços Soap estão aos poucos perdendo o suporte. Por isso, vamos apenas apresentar de forma sucinta como criar um *web service* Soap e consumi-lo.

2.1 Criando um *web service* Soap

Primeiramente é necessário adicionar um projeto .NET Framework em nossa *solution*, pois projetos Core ou mesmo .NET 5/6 não têm mais suporte a esse modelo de projeto.

A Figura 2 demonstra como adicionar um projeto .NET Framework. Pode-se utilizar qualquer versão do .NET Framework, da 3.5 à 4.8. Recomendamos trabalhar com a 4.61 em diante.

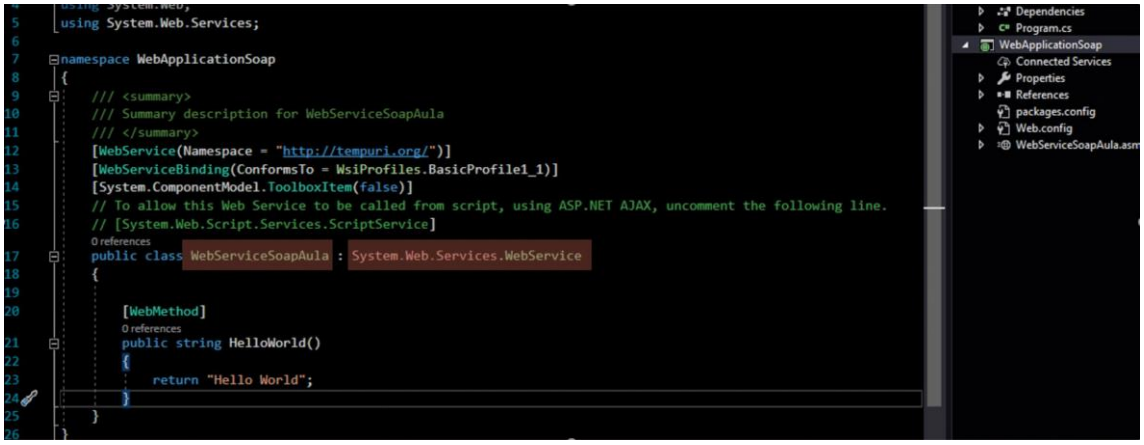
Figura 2 – Como adicionar um projeto .NET Framework



Fonte: Araya, 2021.

Após adicionar o projeto, crie um arquivo com a extensão “.asmx” pela interface (Control + Shift + A). Seu projeto deve então ter um arquivo como este:

Figura 3 – Representação do arquivo



Fonte: Araya, 2021.

Veja que o Visual Studio gerou para nós uma classe herdada de “System.Web.Services.WebService”. Note que o método “HelloWorld” vem por padrão. Observe também **o atributo de notação** em cima do método “HelloWorld”, descrevendo que este é um “**WebMethod**”. Dessa forma, o .NET saberá diferenciar métodos públicos que devem ser expostos à *web* e métodos públicos de sua classe que não serão expostos.

Vamos incluir alguns métodos e analisar seu comportamento:

Figura 4 – Exemplo de método

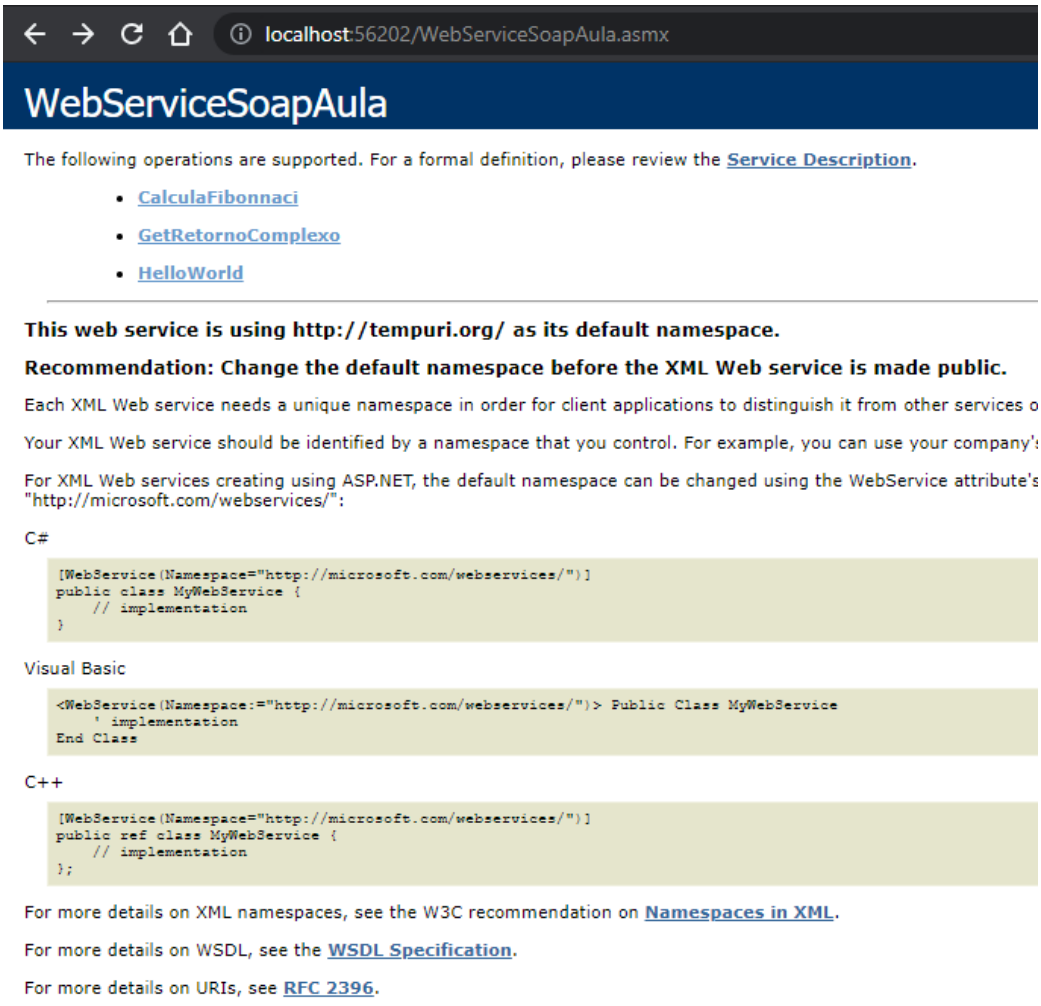
```
[WebMethod]
public List<int> CalculaFibonnaci(int numero)
{

    var sequencia = FibonacciSerie(numero);

    return sequencia.ToList();
}

[WebMethod]
public Autor[] GetRetornoComplexo(string editoras)
{
    var autores = GeraAutores(6, editoras.Split(','));

    return autores.ToArray();
}

public IEnumerable<Autor> GeraAutores(int quantidade, string[] editoras)
{
    List<Autor> autores = new List<Autor>();
    var nomes = new string[] { "ALAN", "JOSE", "MARIA", "LUCAS", "JOÃO", "
MARCELO", "SIMONE", "JOSIANE", "PATRICK", "LAUREN", "GABRIELA", "NICOLE",
"ISABELA", "MATHEUS", "THIAGO", "GABRIEL", "PAULO", "EDUARDO" };
    Random randomizer = new Random();

    for (int i = 0; i < quantidade; i++)
    {
        autores.Add(new Autor()
        {
            Codigo = i,
            Editora = editoras[randomizer.Next(0,editoras.Length)],
            Nome = nomes[randomizer.Next(0, nomes.Length)],
            DataNascimento = new DateTime(randomizer.Next(1960, 2002), ran
domizer.Next(1, 13), randomizer.Next(1, 31))
        });
    }

    return autores;
}
```

Fonte: Araya, 2021.

Adicionamos mais dois “WebMethods” em nosso *web service* e dois novos métodos internos: um para calcular as séries Fibonacci de um determinado número; e outro para gerar uma lista de “Autor” (classe com algumas propriedades que criamos para demonstrar o retorno de um tipo complexo).

Ao executarmos esse projeto, ele irá abrir uma página e apontar para o *localhost* e uma determinada porta. Se acrescentarmos **“/WebServiceSoapAula.asmx**” no endereço, o *browser* irá exibir uma tela como esta:

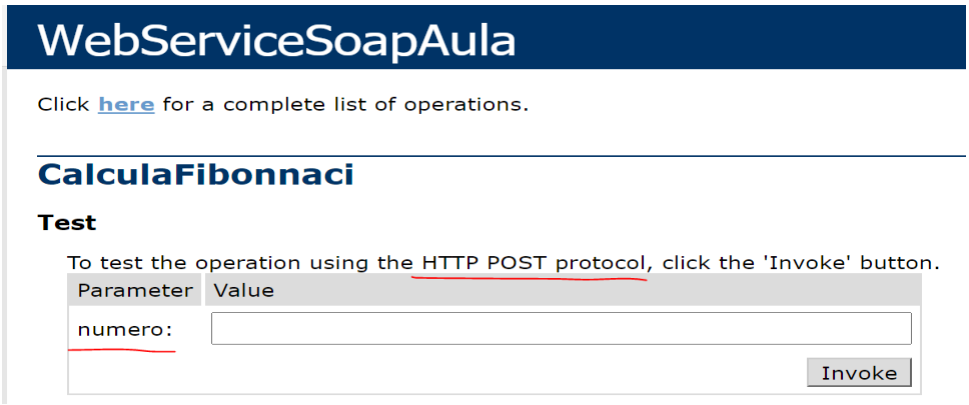
Figura 5 – Tela exibida no *browser*



Fonte: Araya, 2021.

Esta página é gerada automaticamente pelo Asp.Net quando você acessa um *web service* C# via *localhost*. Ela ajuda os testes de seus métodos. Se clicarmos em “CalculaFibonnaci”, veremos o seguinte:

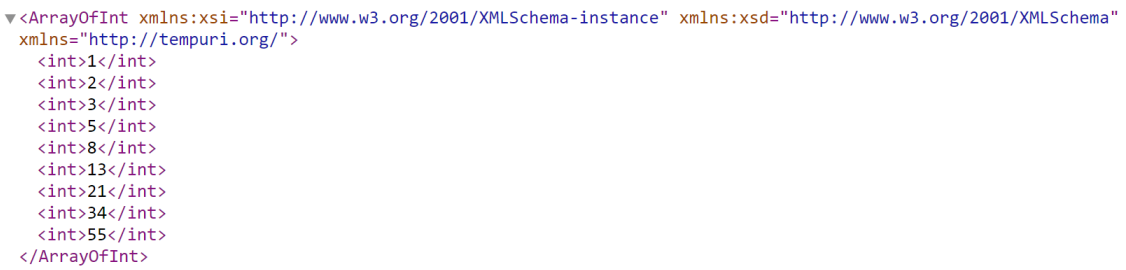
Figura 6 – Tela da opção “CalculaFibonnaci”



Fonte: Araya, 2021.

Observe como foi gerada automaticamente uma “interface de testes” com o parâmetro “numero” de nosso método “CalculaFibonnaci”. Ao executarmos com o valor 10, o retorno será este:

Figura 7 – Retorno com o valor 10



Fonte: Araya, 2021.

Observe como o retorno *de* “*List<int:>*” foi convertido para um “*ArrayOfInt*” no XML. Esse processo é conhecido como “**Serialização**”; falaremos dele mais adiante. Nosso método “*GetRetornoComplexo*” retorna um *array* de autor. Autor é uma classe com algumas propriedades, das quais criamos um método para gerar randomicamente um retorno, a título de exemplo para a aula (todo o código de exemplo desta aula está em sua rota de ensino e no GitHub, nas referências).

Figura 8 – Retorno de “*List<int:>*”

Fonte: Araya, 2021.

Observe o retorno do método “*GetRetornoComplexo*” mostrado. Ele contém propriedades *string, int e Datetime*. Veja como são escritas dentro do XML de retorno.

2.2 Consumindo um *web service* Soap

Para consumir um *web service* Soap, devemos **gerar um “*client*”** com base no **WSDL**. Para visualizar o WSDL de nosso *web service*, basta acrescentar “**?wsdl**” ao final de nossa rota do *browser*:

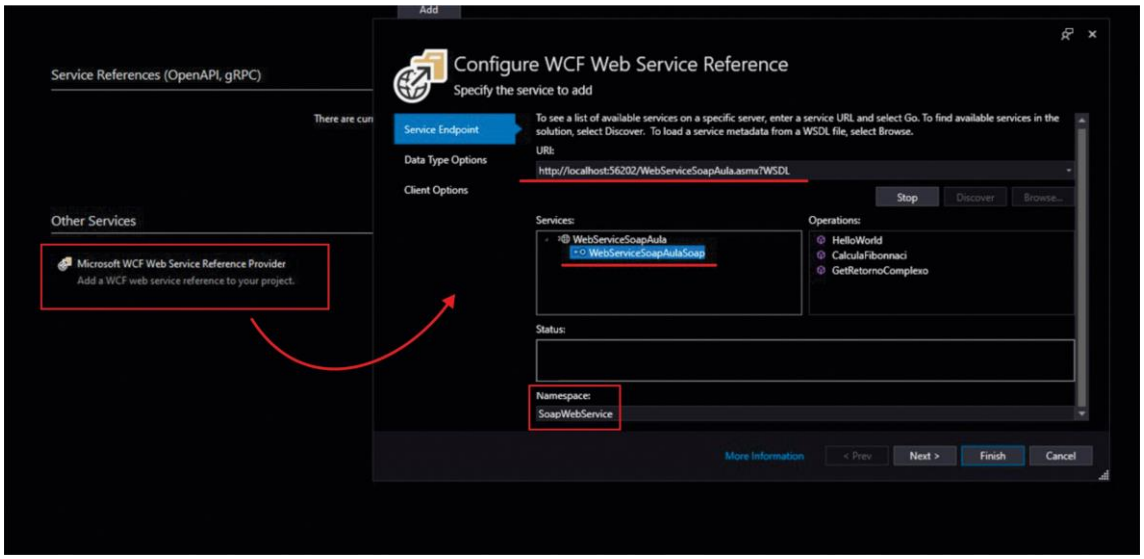
Figura 9 – WSDL do *web service*

Fonte: Araya, 2021.

Observe como o WSDL descreve o tipo do parâmetro, quantas vezes o parâmetro “aparece” no método, o nome do parâmetro etc., e ao mesmo tempo declara a resposta como do tipo “**ArrayOfInt**”, com “*minOccurs=0*” e “*maxOccurs=1*”. Depois temos a definição do que é um “*ArrayOfInt*”, e seu tipo é “um tipo complexo”, com uma sequência de elementos inteiros.

Parar criar o “*client*” e consumir nosso *web service*, vamos utilizar uma ferramenta do Visual Studio. Basta clicar com o botão direito em “*Dependencies*” do nosso projeto ***ConsoleApp*** e então em “***Add Connected Services***”. Uma janela como a Figura 10 abrirá. Nela basta informar a URL do WSDL do *web service* e dar um nome ao “*namespace*” que a ferramenta irá gerar:

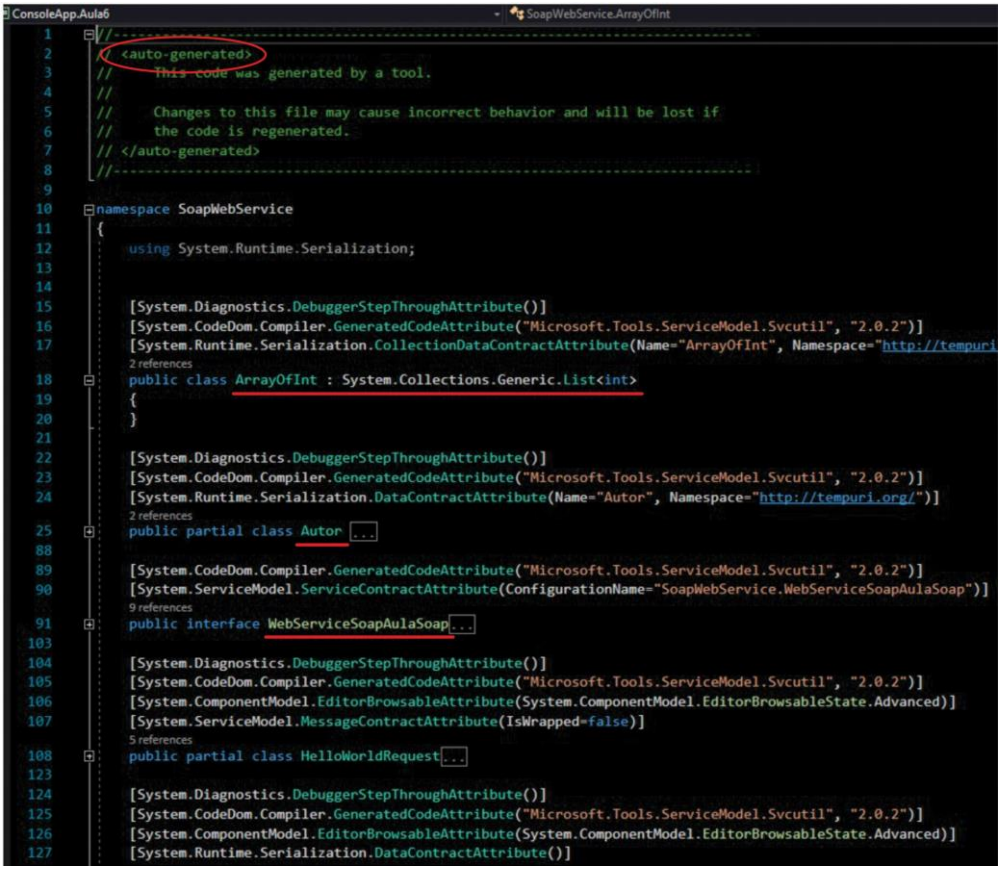
Figura 10 – Informando a URL do WSDL



Fonte: Araya, 2021.

A ferramenta irá "**ler**" o WSDL e gerar uma série de classes automaticamente, que representam os métodos, parâmetros e retornos desse *web service*:

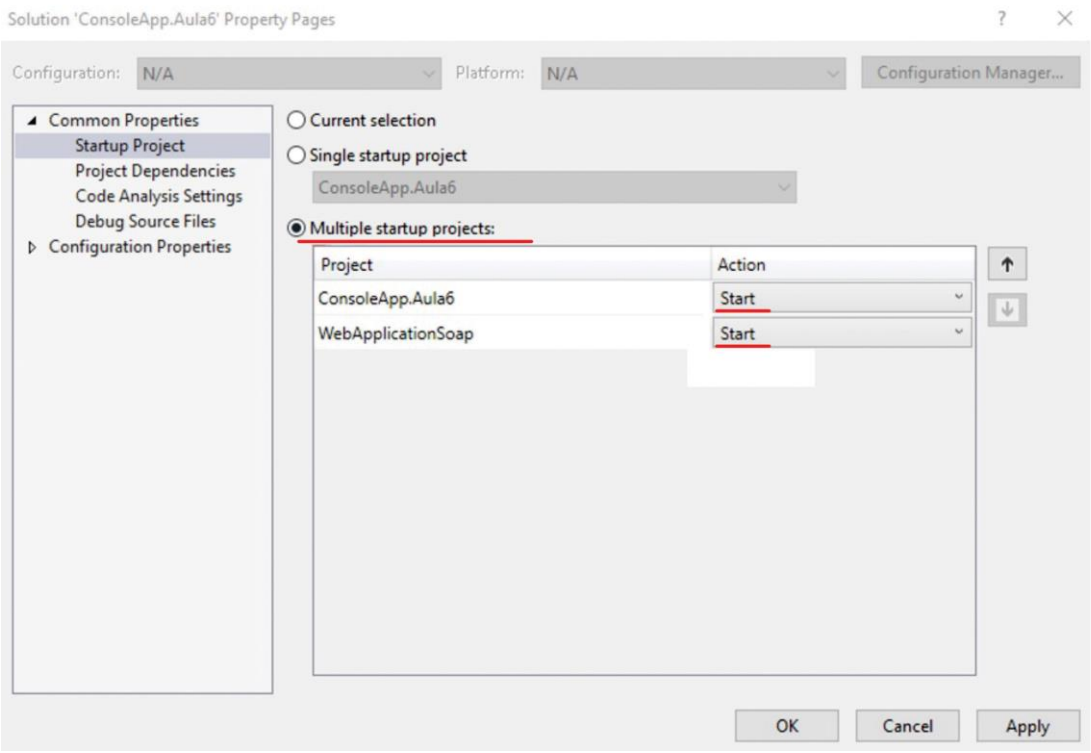
Figura 11 – Leitura do WSDL



Fonte: Araya, 2021.

Para executar a aplicação *console*, vamos precisar configurar o "*start*" de múltiplos projetos no Visual Studio, pois precisaremos manter nosso *web service* e executá-lo ao mesmo tempo que o Console App. Para isso, basta clicar com o botão direito em cima da "*Solution*" e depois na opção "*Set Startup Projects*" e então escolher as opções a seguir:

Figura 12 – Opções que devem ser escolhidas



Fonte: Araya, 2021.

Para consumir o serviço, basta inicializar o *client* gerado pela ferramenta. Observe que o nome da classe, "WebServiceSoapAulaSoapClient", foi gerado pela ferramenta que utilizamos:

Figura 13 – Nome da classe gerado

```
//Incializa o client
var soapClient = new WebServiceSoapAulaSoapClient(WebServiceSoapAulaSoapCl
ient.EndpointConfiguration.WebServiceSoapAulaSoap);

var retornoFibonnaci = await soapClient.CalculaFibonnaciAsync(10);

Console.WriteLine("Resultado Fibonnaci:");
foreach (int numero in retornoFibonnaci.Body.CalculaFibonnaciResult)
{
    Console.Write($"{numero},");
}
//output no console:
//1,2,3,5,8,13,21,34,55,

//-----------------------------------------------------------------

var retornoAutores = await soapClient.GetRetornoComplexoAsync("Intersabere
s, O'Relly, Meaning, B2you");

Console.WriteLine("");
Console.WriteLine("Resultado Autores:");
foreach (var autor in retornoAutores.Body.GetRetornoComplexoResult)
{
    Console.WriteLine($"{autor.Nome} - {autor.Editora} - {autor.DataNascim
ento}");
}
//Output no console:
//Resultado Autores:
//JOSIANE - Meaning - 25 / 06 / 1965 00:00:00
//GABRIEL - O'Relly - 29/04/1973 00:00:00
//JOSIANE - Intersaberes - 30 / 11 / 1982 00:00:00
//EDUARDO - B2you - 09 / 10 / 1962 00:00:00
//MARIA - O'Relly - 07/10/1998 00:00:00
//JOSIANE - Intersaberes - 19 / 03 / 1981 00:00:00
```

Fonte: Araya, 2021.

Podemos notar que gerar um serviço Soap não é uma tarefa muito simples, em partes porque **é um padrão pouco utilizado em novos desenvolvimentos**, sendo um padrão “legado”, ainda muito encontrado em grandes sistemas ou sistemas mais antigos.

TEMA 3 – TRABALHANDO COM *WEB APIS*

Web APIs são serviços *web* assim como os *web services*, porém menos “restritivos” e seguem o protocolo HTTP com seus verbos. Podemos ter *web APIs* que trabalham com troca de mensagens via “*string* pura”, via JSON ou mesmo via XML. O que vai caracterizá-lo como *web API* é sua capacidade de operar utilizando apenas o HTTP e seus verbos, sem uma série de protocolos ou descritores das mensagens e métodos, como o WSDL do Soap.

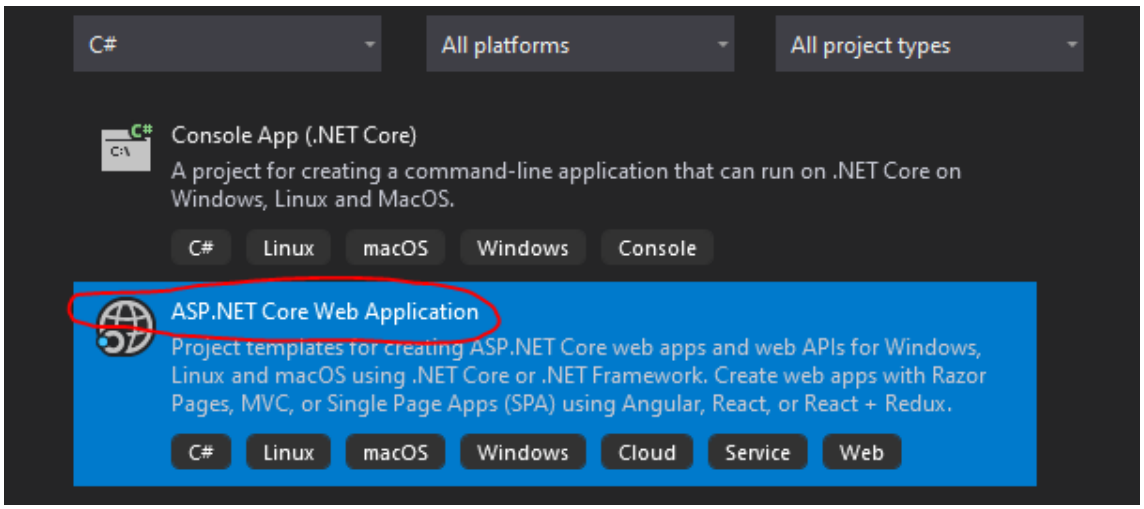
No entanto, isso não significa que *web APIs* não sigam um padrão. Como falamos, existe o **padrão Rest** e também uma iniciativa liderada por grandes *players* de mercado chamada **OpenAPI**, que define e normaliza padrões de *APIs web*, atualmente em sua especificação 3, apresentando como são trafegados dados como mensagens JSON ou em outros formatos, *headers (*cabeçalhos HTTP) e notações de segurança, *cookies* etc.

3.1 Criando um *web API*

Agora vamos criar uma *web API* com os mesmos métodos e retorno para que possamos compreender as diferenças entre esses dois tipos de serviço.

Para trabalharmos com *web APIs*, basta criar um projeto AspNet Core ou AspNet com .NET 5 ou 6, conforme a Figura 14:

Figura 14 – Criando um projeto AspNet Core



Fonte: Araya, 2021.

Toda *web API* em .NET precisa ter uma classe que chamamos de "***Controller***" para cada "recurso" que ela disponibiliza. Essa classe irá agrupar os diversos métodos disponibilizados sobre aquele recurso. Para nosso exemplo, criamos uma classe *controller* chamada "*AulaController*". Ela obrigatoriamente é herdada de "***ControllerBase***":

Figura 15 – Criação da *AulaController*

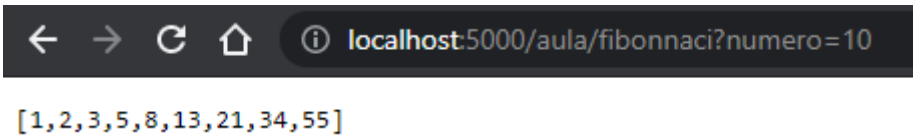
```
[Route("[controller]")]
[ApiController]
public class AulaController : ControllerBase
{

    [HttpGet]
    public string HelloWorld()
    {
        return "Essa é uma Web API. Digite o método que deseja acessar na
url...";
    }

    [HttpGet]
    [Route("fibonnaci")]
    public IEnumerable<int> CalculaFibonnaci(int numero)
    {

        var sequencia = FibonacciSerie(numero);

        return sequencia.ToList();
    }

    [HttpGet]
    [Route("autores")]
    public Autor[] GetRetornoComplexo(string editoras)
    {
        var autores = GeraAutores(6, editoras.Split(','));

        return autores.ToArray();
    }
}
```

Fonte: Araya, 2021.

Observe que o *controller* também tem várias "**anotações**" em cima de seus métodos, as quais chamamos de *atributos*, que auxiliam o compilador a entender comportamentos especiais para eles (Atributos…, 2018). Esses comportamentos, no caso da *web API*, disponibilizam publicamente o acesso a esses métodos via *web*.

A anotação "*HttpGet*" significa que, quando alguém chamar a URL em que nossa API estiver publicada – por exemplo **http://localhost:5000/aula** –, o .NET vai procurar o primeiro método com "*HttpGet*" para responder a essa requisição. Os demais que tiverem o atributo "*Route*" estão dizendo que, para acessá-los, precisamos chamar uma sub-rota, similar a **http://localhost:5000/aula/fibonnaci**. Observe que não usamos o nome do método, e sim a sua "rota". Caso você execute a URL **http://localhost:5000/aula/fibonnaci?numero=10**, obterá em seu *browser* a seguinte resposta:

Figura 16 – Resposta no *browser*

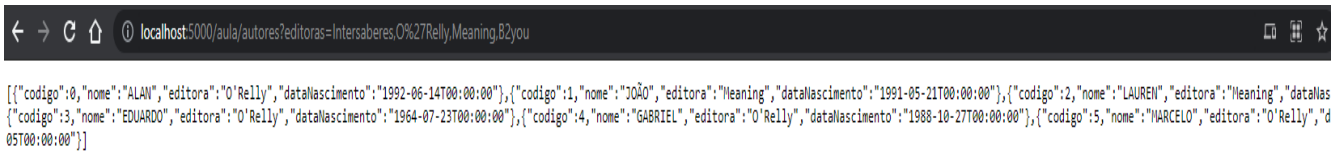


Fonte: Araya, 2021.

Essa resposta está em um formato bem diferente da resposta dada pelo *web service* em Soap. Percebemos que utiliza o formato JSON de saída. Vamos abordar melhor esse formato mais adiante.

Para chamar o método que retorna os autores, basta executarmos http://localhost:5000/aula/autores?editoras=Intersaberes,O%27Relly,Meaning,B2you:

Figura 17 – Chamando o método



Fonte: Araya ,2021.

3.2 Consumindo uma *web API*

Para que possamos consumir esta *web API*, devemos utilizar o "***HttpClient***" (já o utilizamos anteriormente, na aula sobre *threads* e *tasks*). O C# não tem gerador de "*client*" para *web APIs* como tem para Soap. Você precisa utilizar alguns pacotes adicionais para isso (veremos isso adiante). Primeiro, vamos entender como funciona o consumo das *web APIs* de forma direta, usando o *HttpClient*:

Figura 18 – Usando o *HttpClient*

```
var httpClient = new HttpClient();
int numeroFibonnaci = 10;
HttpResponseMessage responseFibonnaci = await httpClient.GetAsync($"http:/
/localhost:5000/aula/fibonnaci?numero={numeroFibonnaci}");

if (responseFibonnaci.IsSuccessStatusCode)
{
    string contentString = await responseFibonnaci.Content.ReadAsStringAsy
nc();
    var sequenciaFibonnaci = System.Text.Json.JsonSerializer.Deserialize
<int[]>(contentString);

    foreach (int numero in sequenciaFibonnaci)
    {
        Console.Write($"{numero},");
    }
}
//output no console:
//1,2,3,5,8,13,21,34,55,
```

Fonte: Araya, 2021.

Observe que o *HttpClient* nos retorna um ***HttpResponseMessage***. Essa classe representa o retorno de uma chamada HTTP. No protocolo HTTP, temos alguns tipos de códigos de retorno, chamados de ***StatusCodes***; o código 200 representa que nossa chamada HTTP o executou com sucesso. Antes de ler o retorno do *response*, precisamos checar seu código de *status*. Depois podemos ler o conteúdo do "*body*" do *HTTP response*, presente na variável *Content*.

Quando executamos um "*ReadAsStringAsync*()", estamos lendo o conteúdo da variável *Content*, que tem uma estrutura de *stream*. Nesse ponto, o conteúdo retornado na variável "sequenciaFibonnaci" seria "[1,2,3,5,8,13,21,34,55]". **Esse conteúdo está "serializado" em *string***. Para que possamos converter esse texto (*string*) em um objeto, precisamos utilizar uma técnica de "desserialização". Para isso, utilizamos o ***System.Text.Json.JsonSerializer***, passando dentro do *type* "<in[]>" (*array* de inteiros) para então obter um ***array* de inteiros** e agora poder iterar sobre eles. O *output* será idêntico ao do método consumido do *web service*.

Em um primeiro momento, pode parecer mais complicado o retorno/conversão do resultado de um *HttpClient*, mas isso acontece também por "trás dos panos" dentro do *Client Soap*. Na classe gerada pelo C#, também existe um processo de serialização, que trataremos no Tema 4.

TEMA 4 – SERIALIZAÇÃO DE DADOS

A serialização é o ato de transformar um objeto na memória ou uma relação de objetos (conjunto de objetos que referenciam um ao outro) e "achatá-los" em um fluxo de *bytes* (*stream*), em XML ou em JSON, de tal forma que possam ser armazenados ou transmitidos via arquivos ou via rede. A desserialização funciona ao contrário: obtendo um fluxo de dados e reconstituindo-o em um objeto na memória (Albahari; Johannsen, 2020, p. 743).

A serialização e a desserialização são normalmente usadas para:

- transmitir objetos pela rede ou entre aplicativos;
- armazenar representações de objetos em um arquivo ou banco de dados.

Existem quatro mecanismos "nativos" de serialização no .NET (alguns podem ser entregues via pacotes à parte):

1. *Data contract serializer*;
2. *Binary serializer*;
3. *XML serializer*;
4. *JSON serializer*.

Quando você estiver serializando para XML, pode escolher entre *XmlSerializer* e "*data contract serializer*". O *XmlSerializer* oferece maior flexibilidade sobre como o XML é estruturado, enquanto o "*data contract serializer*" tem a capacidade única de preservar referências de objetos compartilhados.

Se você estiver serializando para JSON, também tem uma escolha. *JsonSerializer* (dentro do *namespace System.Text.Json*) oferece o melhor desempenho, enquanto o "*data contract serializer*" tem alguns recursos extras por ser um método-base a todos. No entanto, se você realmente precisar de recursos extras, uma escolha melhor provavelmente será a biblioteca *Newtonsoft.Json* (pacote *NuGet* de terceiros).

Se você precisar comunicar-se com serviços da *web* baseados em Soap, o "*data contract serializer*" é a melhor escolha. E se você não se preocupa com o formato dos dados após a serialização, o motor de serialização binária é o mais poderoso e fácil de usar. A saída, no entanto, não é legível por humanos e é menos tolerante às diferenças entre os objetos serializados e às classes que os representam do que os outros serializadores (por exemplo, se houver uma

propriedade faltante na classe quando esta for desserializada, podem ocorrer problemas) (Albahari; Johannsen, 2020, p. 747).

Você deverá lidar com serialização o **tempo todo** quando **transferir e receber dados via rede de *web services* ou *web APIs* (via rede de forma geral)**. Isso não é uma exclusividade do .NET; é simplesmente como a *web* funciona. Precisamos converter um objeto em memória de um servidor para um cliente, transformando-o em uma representação que o cliente do outro lado possa ler e "reconstruir" novamente.

A serialização nunca propaga comportamentos (métodos); apenas estado, ou seja, **dados** (Albahari; Johannsen, 2020, p. 747). Nesta aula vamos apenas explorar a serialização em JSON e suas propriedades, não entrando nos detalhes dos outros tipos, mas você pode aprofundar o tema nas referências desta aula.

4.1 Serialização em JSON

O formato JSON é atualmente um dos mais conhecidos e utilizados para trafegar dados entre aplicações, em especial *web APIs*.

A raiz de um documento JSON é sempre um *array* ou um objeto. Sob essa raiz estão as propriedades, que podem ser objetos, *arrays*, *strings*, números, "verdadeiro", "falso" ou "nulo". O serializador JSON mapeia diretamente nomes de propriedade da classe para nomes de propriedade em JSON.

Para exemplificar um JSON, vamos usar a classe Autor que utilizamos na criação de nossa *web API*:

Figura 19 – Usando a classe Autor

```
public class Autor
{
    public int Codigo { get; set; }
    public string Nome { get; set; }
    public string Editora { get; set; }
    public DateTime DataNascimento { get; set; }
}
```

Fonte: Araya, 2021.

O JSON de um objeto Autor serializado tem a seguinte forma:

Figura 20 – JSON de um objeto Autor serializado

```
{
  "codigo": 2,
  "nome": "LUCAS",
  "editora": "Intersaberes",
  "dataNascimento": "1962-06-25T00:00:00"
}
```

Fonte: Araya, 2021.

Podemos perceber como a estrutura JSON mantém os dados “legíveis” e ao mesmo tempo relativamente enxutos; ou seja, tem muito menos textos de declaração e finalização do que o XML, por exemplo, tornando-o mais leve para transmissão pela rede.

Para serializar um objeto, basta uma simples instrução para o *JsonSerializer*, e dessa forma ele transforma todos os dados de seu objeto e objetos relacionados em uma *string* JSON.

Para deixar o exemplo mais rico, vamos adicionar uma classe de endereço a um Autor:

Figura 21 – Adicionando uma classe de endereço a um Autor

```
public class Autor
{
    public int Codigo { get; set; }
    public string Nome { get; set; }
    public string Editora { get; set; }
    public DateTime DataNascimento { get; set; }
    public Endereco EnderecoAutor { get; set; }
}
public class Endereco
{
    public string Rua { get; set; }
    public int Numero { get; set; }
    public string CEP { get; set; }
    public string Complemento { get; set; }
    public bool PossuiPortaria { get; set; }
}
```

Fonte: Araya, 2021.

Para serializar um objeto, basta utilizarmos o método “Serialize” do *JsonSerializer*. Passando o objeto como parâmetro, ele retornará uma *string*:

Figura 22 – Retornando uma *string*

```
var autor1String = JsonSerializer.Serialize(autor);
```

Fonte: Araya, 2021.

Para desserializar uma *string*, convertendo-a novamente em um objeto, podemos utilizar o método “Deserialize”*:*

Figura 23 – Usando o método “Deserialize”

```
var autorDeseriazado = JsonSerializer.Deserialize<Autor>(autor1String);
```

Fonte: Araya, 2021.

No exemplo a seguir, vamos mostrar uma estrutura JSON serializada com base numa lista de objetos Autor, contendo dois autores, com dados distintos preenchidos. O Autor 2 tem algumas propriedades não “definidas”, como o “Numero” do endereço e o “Complemento”.

Observe como essas propriedades assumiram valores “*default*” de seus Types dentro do JSON:

Figura 24 – Valores “*default*” dentro do JSON

```
[
  {
    "Codigo": 100,
    "Nome": "JON Author",
    "Editora": "Intersaberes",
    "DataNascimento": "1979-06-20T00:00:00",
    "EnderecoAutor": {
      "Rua": "Rua de Teste",
      "Numero": 2021,
      "CEP": "81560-120",
      "Complemento": "AP 20",
      "PossuiPortaria": true
    }
  },
  {
    "Codigo": 200,
    "Nome": "POP Author",
    "Editora": "Intersaberes",
    "DataNascimento": "1999-02-10T00:00:00",
    "EnderecoAutor": {
      "Rua": "Av. dos Estados",
      "Numero": 0,
      "CEP": "81550-110",
      "Complemento": null,
      "PossuiPortaria": false
    }
  }
]
```

Fonte: Araya, 2021.

Podemos notar também como o JSON passou a iniciar com os colchetes, “[ ]”, em vez de chaves, “{ }”, porque serializamos um *array*. Dessa forma,

podemos assumir que, sempre que visualizarmos objetos dentro de “[ ]”, é porque são elementos de um *array*.

Outro detalhe: observe como as propriedades com valores *null*, *true*, *false* e numéricos ficam definidas sem as aspas (“ ”) de uma *string* comum. Mesmo o valor de data foi representado em *string*, mas esses outros, não. Isso faz parte do padrão de notação do JSON.

Note também que o objeto Endereço foi serializado junto ao objeto principal Autor; isso é um comportamento comum do serializador de JSON. Ele sempre irá serializar o objeto e toda sua árvore de dependências, para que ele possa ser lido novamente, preservando completamente seu estado.

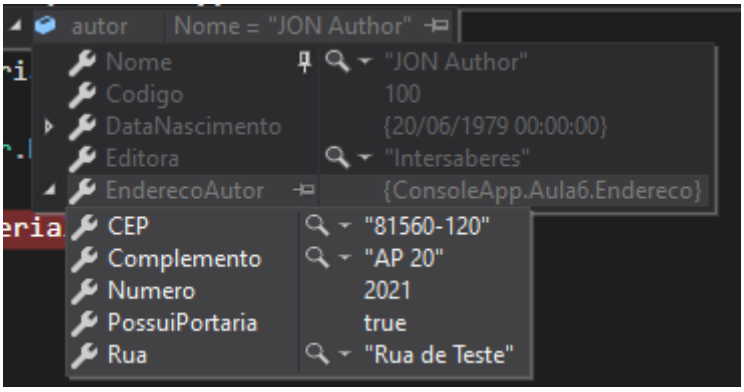
4.2 JSON: tolerância a mudanças

Um dos principais fatores que fazem o formato JSON ser tão popular no mundo *web* e em especial nas *web APIs* é o fato de ser muito flexível quando se trata de mudanças, principalmente no lado da desserialização (conversão da *string* para objeto novamente). Podemos serializar um objeto com quatro campos totalmente fora da ordem original e, mesmo assim, podemos desserializá-lo para um outro objeto com cinco campos ou menos, em qualquer ordem.

Veja o exemplo a seguir, em que modificamos a ordem do JSON de nosso objeto Autor e depois o desserializamos normalmente:

Figura 25 – Desserializando o objeto Autor

```
{
  "DataNascimento": "1979-06-20T00:00:00",
  "EnderecoAutor": {
    "CEP": "81560-120",
    "Complemento": "AP 20",
    "Rua": "Rua de Teste",
    "Numero": 2021,
    "PossuiPortaria": true
  },
  "Codigo": 100,
  "Nome": "JON Author",
  "Editora": "Intersaberes"
}
```



Fonte: Araya, 2021.

TEMA 5 – REFIT

O C# conta atualmente com algumas bibliotecas muito interessantes para trabalhar com *web APIs*, que facilitam a vida de quem precisa consumir esse tipo de serviço. Uma das mais famosas e utilizadas é o Refit.

O Refit é um pacote NuGet, *open-source*, desenvolvido por terceiros, inspirado no pacote Retrofit para Android, que cria uma abstração de serviços *web RESTful* em C# de forma similar ao "*client*" do serviço Soap, conseguindo ser ainda mais simples e leve.

O Refit nos permite usar e chamar métodos de *web APIs* utilizando qualquer verbo HTTP, passando *headers* customizados e mesmo *tokens* de autenticação, como JWT. Para instalá-lo, basta adicionar o pacote NuGet Refit a seu projeto. Em seguida, você deve criar uma interface que represente a *web API* que será consumida. Dentro dessa interface, cada método do C# será equivalente a um método da API, como na Figura 26:

Figura 26 – Usando o Refit

```
interface IWebApiAulaRefit
{
    [Get("/fibonnaci")]
    public Task<List<int>> CalculaFibonnacciAsync(int numero);

    [Get("/autores")]
    public Task<IEnumerable<Autor>> GetAutoresAsync([AliasAs("editoras")]
string editorasSeparadoPorVirgula);
}
```

Fonte: Araya, 2021.

Observe que criamos dois métodos, um para cada método da *web API*. Além disso, observe a "anotação" (o atributo) em cima dos métodos, dizendo que são métodos do verbo "GET" do HTTP. Também podemos observar que os dois

métodos retornam uma Task e têm “Async” no nome. O retorno de Task é obrigatório no Refit, pois trabalha somente com métodos assíncronos. O “Async” no nome é uma questão de convenção, para lembrar o desenvolvedor de realizar o “*await*” desse método. Observe também o uso do atributo “AliasAs” no parâmetro do método “GetAutoresAsync”. Ele é utilizado para que possamos ter um nome do parâmetro no C# diferente do nome do parâmetro que é passado via HTTP para a API.

Para chamar nossa API, precisamos apenas de duas linhas de código, como na Figura 27:

Figura 27 – Duas linhas de código

```
//Cria o client com base na interface
var refitClient = RestService.For<IWebApiAulaRefit>("http://localhost:5000
/aula");

//executa o método "GET fibonnaci" da web api
var reultadoFibonnaci = await refitClient.CalculaFibonnacciAsync(10);

//executa o método "GET autores" da web api
var autores = await refitClient.GetAutoresAsync("Intersaberes, O'Relly, Me
aning, B2you");
```

Fonte: Araya, 2021.

Vamos explicar o que está acontecendo por trás desse código pela Figura 28:

Figura 28 – Como o código funciona

Fonte: Araya, 2021.

Logo, podemos concluir que o Refit executa, por dentro de seu código, algo **muito similar ao que faríamos de forma "manual"**, manipulando o *HttpClient*. É claro que, quando você manipula diretamente o *HttpClient*, você tem um controle preciso sobre tudo que é fornecido de parâmetro para a *Request* e todos os dados disponíveis no retorno da chamada.

Recapitulando: para usar o Refit, basta importar seu pacote NuGet, adicionar uma interface no código (ela pode ter o nome que você desejar) e adicionar métodos nela com os atributos (anotações em cima do método) do Refit. O *site* do repositório do GitHub do Refit tem vários exemplos de tipos diferentes de chamadas, com as mais diversas características. É muito raro o Refit não atender uma chamada para uma *web API*.

O Refit ainda suporta a desserialização para dois tipos especiais:

- ***String***: qualquer método HTTP pode ter o conteúdo de seu retorno desserializado para *string*, porque o "*body*" do HTTP suporta essa funcionalidade. Logo, independente de você trafegar binário, XML, texto livre ou JSON, o Refit poderá "ler o conteúdo" do "*body*" como *string* e retorná-lo sem convertê-lo para um tipo especial;
- **HttpResponseMessage**: é o retorno-padrão do *HttpClient*. Toda chamada HTTP retornará esse objeto. Se você passá-lo dentro do Type esperado em sua interface, receberá como retorno o objeto-padrão retornado da chamada HTTP do *HttpClient*, sem nenhuma desserialização ou manipulação específica pelo Refit. Pode ser muito útil para situações de *debug* (depuração) e de manipulação do Headers de Response (por exemplo).

FINALIZANDO

Nesta aula pudemos apreender como consumir *web services* e *web APIs*, recursos comuns ao dia a dia de qualquer desenvolvedor. Como sabemos, quase toda aplicação *mobile* depende de um serviço *web*. Para realizar *login* (autenticação), para buscar ou salvar dados, é improvável que você tenha um *app* que não se comunique com a *web*.

Os aprendizados desta aula poderão ser úteis sempre que você precisar chamar um serviço *web*, pois são genéricos para qualquer plataforma do .NET.

Você pode usá-los tanto em aplicações *console*, como as que fizemos, usá-los em uma aplicação *web* e, claro, em aplicações Xamarin Mobile.

Não é o objetivo desta aula que você domine a criação e o funcionamento de serviços *web*; isso será abordado futuramente. Nosso intuito é prepará-lo para lidar com a interação desses serviços na sua aplicação.

Entendemos também a diferença básica entre *web APIs* e *web services*, além de entendermos como funcionam aplicações Soap e os passos necessários para consumir esse tipo de serviço, que, por mais que seja pouco utilizado, ainda é encontrado em aplicações grandes e legadas.

Pudemos aprender também sobre os componentes de serialização do .NET, principalmente sobre serialização com JSON, pilar principal para a interação com *web APIs*. A serialização é um processo normal no dia a dia do desenvolvimento, e temos que lidar com ela sempre que trafegarmos algum objeto pela rede ou mesmo salvar seus dados em um arquivo ou banco de dados. Por isso, lembre-se de sempre buscar um serializador que suporte seu formato e conteúdo.

REFERÊNCIAS

ANDERSON, R.; LARKIN, K. Tutorial: criar uma API Web com o ASP.NET Core. **Docs.Microsoft**, [S.l.], 9 set. 2021. Disponível em: <https://docs.microsoft.com/pt-br/aspnet/core/tutorials/first-web-api>. Acesso em: 22 set. 2021.

ALBAHARI, J.; JOHANNSEN, E. **C# 8.0 in a nutshell**. Sebastopol: O'Reilly Media Inc., 2020.

ARAYA, A. Programação da Aula 6. **GitHub**, [S.l.], [20-?]. Disponível em: <https://github.com/alan-araya/programacao.Aula6>. Acesso em: 22 set. 2021.

ATRIBUTOS (C#). **Docs.Microsoft**, [S.l.], 26 abr. 2018. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/csharp/programming-guide/concepts/attributes/>. Acesso em: 22 set. 2021.

GRIFFITHS, I. **Programming C# 8.0**. 2. ed. Sebastopol: O'Reilly Media Inc., 2019.

HTTPCLIENT classe. **Docs.Microsoft**, [S.l], 11 jun. 2018. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/api/system.net.http.httpclient>. Acesso em: 22 set. 2021.

INTRODUCTION. **OpenAPI**, [S.l.], 1º jan. 2021. Disponível em: <https://oai.github.io/Documentation/introduction.html>. Acesso em: 22 set. 2021.

OPÇÕES de configuração de runtime para rede. **Docs.Microsoft**, [S.l.], 27 nov. 2019. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/core/run-time-config/networking>. Acesso em: 22 set. 2021.

USAR a ferramenta WCF Web Service Reference Provider. **Docs.Microsoft**, [S.l.], 29 out. 2019. Disponível em: <https://docs.microsoft.com/pt-br/dotnet/core/additional-tools/wcf-web-service-reference-guide>. Acesso em: 22 set. 2021.