

Tópicos relacionados ao
desenvolvimento de APIs Web
utilizando a plataforma Microsoft
ASP.NET.

ASP.NET Web API

Introdução

Israel Aece
<http://www.israelaece.com>

Conteúdo

Prefácio	3
Introdução	5
HTTP, REST e o ASP.NET	7
Estrutura da API.....	12
Roteamento.....	32
Hosting.....	42
Consumo.....	46
Formatadores	53
Segurança	60
Testes e Tracing	67
Arquitetura e Estensibilidade	77

Prefácio

A multiplicação de dispositivos representa um desafio e uma oportunidade para os desenvolvedores de software. Nossos sistemas são executados em computadores, telefones, tablets, aparelhos de TV. As possibilidades são muitas e as dificuldades são ainda maiores. Arquiteturas convencionais de software não atendem mais, de forma satisfatória, essa pluralidade de plataformas, tanto de hardware quanto operacionais, com que temos de lidar. Há bem pouco tempo, falava-se em sistemas Web como uma alternativa para reduzir os obstáculos de distribuição. Hoje, esse argumento parece não ser mais suficiente.

Nesse mesmo cenário, temos empresas que operam dezenas – algumas falam em centenas ou, até mesmo, milhares – de sistemas. Tudo isso com "algum nível" de integração. Entretanto, a competitividade tem exigido dessas empresas e, em consequência, de seus sistemas, grande flexibilidade e agilidade. Regras de negócio são incluídas, revisadas e modificadas frequentemente. Aquisições e fusões trazem ainda mais sistemas – com sobreposições aos existentes, inclusive – que também precisam ser mantidos e integrados.

Tanto a pluralidade de plataformas, como a flexibilidade e multiplicidade inevitável de sistemas, tem levado ao desenvolvimento de arquiteturas que separam, de forma consistente, as interfaces com usuário da exposição efetiva das funcionalidades das aplicações. De fato, é crescente a adoção de arquiteturas orientadas a serviço (SaaS). Ou seja, cada vez mais o "core" das aplicações pode ser acessado remotamente através de interfaces programáveis (APIs) permitindo, dessa forma, o desenvolvimento de aplicações leves e ajustadas para diversas plataformas, bem como a adoção de práticas efetivas de integração.

Arquiteturas baseadas em serviços também são comuns em desenvolvimento de aplicações B2C. Empresas como Facebook, Twitter, Google e Microsoft já as adotam há muito tempo. Hoje, podemos escrever facilmente aplicações que consomem e se integram a essas plataformas graças a essa característica.

Sensível a necessidade crescente de arquiteturas orientadas a serviço, vimos o surgimento de diversos frameworks. Dentre eles, está o Microsoft ASP.net Web API. Trata-se de uma tecnologia leve para o desenvolvimento de serviços baseados em HTTP, desenvolvida pela Microsoft, que destaca-se pela elegância, objetividade e simplicidade. Com ASP.net Web API, podemos desenvolver serviços HTTP plenamente adaptados ao modelo REST, com suporte a diversas formatações (XML e JSON são suportados nativamente), que são fáceis de testar, consumir e manter. Além de tudo isso, há ainda um amplo suporte de ferramentas – de desenvolvimento, distribuição, hospedagem e manutenção - providas pela própria Microsoft ou terceiros.

Este livro é uma introdução séria ao desenvolvimento de serviços usando Microsoft ASP.net Web API. Ele foi escrito por uma das maiores autoridades nesse assunto e um dos profissionais de TI mais respeitados no Brasil. Há muitos anos, Israel Aece vem compartilhando conhecimentos com a comunidade desenvolvedora. Além disso, ele

tem ensinado, inspirado e influenciado a forma como eu penso o desenvolvimento de serviços Web. Por isso, sinto-me imensamente honrado pela oportunidade de recomendar esse livro. Aqui, você encontrará uma orientação valiosa para adoção e uso de Web API. É uma leitura agradável mesmo para desenvolvedores que já dominem o framework. Aliás, é um daqueles livros que gostaria de ter tido a oportunidade de ler há mais tempo.

Elemar Junior

<http://elemarjr.net>

Introdução

Com a popularização da *internet* temos cada vez mais informações sendo disponibilizadas em tempo real, acessos à recursos que antes eram apenas possíveis em algum ponto específico e com uma série de limitações. A evolução tornou o acesso a estas informações muito mais dinâmica, conseguindo acessá-la a qualquer momento, em qualquer lugar e, principalmente, em qualquer dispositivo.

Com isso em mente algumas opções foram sendo criadas para permitir com que as aplicações pudessem disponibilizar tais informações e outras pudessem consumir. Mas em pouco tempo foram criadas diversas tecnologias e padrões para isso, e opções que tínhamos até pouco tempo atrás, hoje já são obsoletas.

A proposta deste livro é introduzir a tecnologia que a Microsoft incorporou no ASP.NET para a construção e o consumo de APIs Web. A ideia é formar o conhecimento desde a introdução à estrutura, passando pela arquitetura e alguns detalhes internos sobre o funcionamento, que permitirá extrair grande parte do potencial que esta tecnologia fornece.

Como pré-requisitos deste livro, precisaremos de conhecimento em alguma linguagem .NET (os exemplos serão baseados em C#) e, opcionalmente, alguma familiaridade com projetos Web no *Visual Studio*, para conseguirmos utilizar a IDE a nosso favor em alguns pontos importantes durante o desenvolvimento.

Capítulo 1 – HTTP, REST e o ASP.NET: Para basear todas as funcionalidades expostas pela tecnologia, precisamos ter um conhecimento básico em relação ao que motivou tudo isso, contando um pouco da história e evolução, passando pela estrutura do protocolo HTTP e a relação que tudo isso tem com o ASP.NET.

Capítulo 2 – Estrutura da API: Entenderemos aqui a *template* de projeto que o *Visual Studio* fornece para a construção das APIs, bem como sua estrutura e como ela se relaciona ao protocolo.

Capítulo 3 – Roteamento: Como o próprio nome diz, o capítulo irá abordar a configuração necessária para que a requisição seja direcionada corretamente para o destino solicitado, preenchendo e validando os parâmetros que são por ele solicitado.

Capítulo 4 – Hosting: Um capítulo de extrema relevância para a API. É o *hosting* que dá vida à API, disponibilizando para o consumo por parte dos clientes, e a sua escolha interfere diretamente em escalabilidade, distribuição e gerenciamento. Existem diversas formas de se expor as APIs, e aqui vamos abordar as principais delas.

Capítulo 5 – Consumo: Como a proposta é ter uma API sendo consumido por qualquer cliente, podem haver os mais diversos meios (bibliotecas) de consumir estas APIs. Este capítulo tem a finalidade de exibir algumas opções que temos para este consumo, incluindo as opções que a Microsoft criou para que seja possível efetuar o consumo por aplicações .NET.

Capítulo 6 – Formatadores: Os formatadores desempenham um papel importante na API. São eles os responsáveis por avaliar a requisição, extrair o seu conteúdo, e quando a resposta é devolvida ao cliente, ele entra em ação novamente para formatar o conteúdo no formato em que o cliente possa entender. Aqui vamos explorar os formatadores padrões que já estão embuidos, bem como a criação de um novo.

Capítulo 7 – Segurança: Como a grande maioria das aplicações, temos também que nos preocupar com a segurança das APIs. E quando falamos de aplicações distribuídas, além da autenticação e autorização, é necessário nos preocuparmos com a segurança das mensagens que são trocadas entre o cliente e o serviço. Este capítulo irá abordar algumas opções que temos disponíveis para tornar as APIs mais seguras.

Capítulo 8 – Testes e Tracing: Para toda e qualquer aplicação, temos a necessidade de escrever testes para garantir que a mesma se comporte conforme o esperado. Isso não é diferentes com APIs Web. Aqui iremos abordar os recursos, incluindo a própria IDE, para a escrita, gerenciamento e execução dos testes.

Capítulo 9 – Extensibilidade e Arquitetura: Mesmo que já tenhamos tudo o que precisamos para criar e consumir uma API no ASP.NET Web API, a customização de algum ponto sempre acaba sendo necessária, pois podemos criar mecanismos reutilizáveis, “externalizando-os” do processo de negócio em si. O ASP.NET Web API foi concebido com a extensibilidade em mente, e justamente por isso que existe um capítulo exclusivo para abordar esse assunto.

Por fim, este livro foi escrito com utilizando o *Visual Studio 2012 com Update 3, ASP.NET and Web Tools 2012.2, .NET Framework 4.5 e Visual Studio Express 2013 Preview for Web* para criar e executar os exemplos contidos aqui. Qualquer dúvida, crítica ou sugestão, entre em contato através do site <http://www.israelaece.com> ou enviando um e-mail para ia@israelaece.com.

HTTP, REST e o ASP.NET

Os sistemas que são construídos atualmente produzem toneladas e toneladas de informações, que são armazenadas em uma base de dados, para mais tarde, estas mesmas informações serão consumidas por estas e outras aplicações.

Não existe muitos problemas enquanto há apenas uma única aplicação consumidora destas informações, afinal ela pode consumir diretamente a base de dados para ler e gravar as informações desejadas. Se estamos dentro de um ambiente controlado, como por exemplo, aplicações construídas dentro de uma mesma empresa, podemos utilizar a própria base de dados para “integrar” as mesmas.

Enquanto estamos consumindo informações locais, estamos diretamente conectados a nossa infraestrutura, sem qualquer impecílio mais crítico para acessar e consumir as informações. O dinamismo do mercado faz com que hoje as empresas operem em tempo real, onde as informações que precisam ser consumidas ou disponibilizadas estão além do ambiente interno, ou seja, a necessidade de uma integração real com parceiros de negócios, órgãos governamentais, etc.

Utilizar a base de dados como meio de integração não é mais uma opção. Estamos agora lidando com empresas que estão fora da nossa rede, com aplicações construídas com tecnologias diferentes daquelas que adotamos internamente, padrões diferentes, etc.

Tudo isso motivou algumas grandes empresas de tecnologia do mundo a trabalharem na construção de forma de integração, criando um padrão predefinido para troca de informações entre tais empresas. Em pouco tempo surgiu o conceito de *Web Services*, onde a ideia é permitir com que sistemas se conectem a fim de trocar informações entre eles, sem a necessidade da intervenção humana.

Para dar suporte a tudo isso, o XML foi utilizado como forma de expressar essa comunicação entre as partes. O XML é uma linguagem de marcação, e que apesar de ser texto puro, fornece uma gama de outros recursos agregados que o torna bastante poderoso, como por exemplo, a possibilidade de extensível, separar definição de conteúdo, opções para validação de estrutura, simplicidade na leitura (inclusive por humanos).

Como a *internet* estava cada vez mais difundida, utilizá-la como forma de integração foi uma das principais opções, pois tratavam-se de tecnologias de padrão aberto, como é o caso do HTTP e HTML. O HTTP dá vida aos *Web Services*, pois é ele que o torna acessível, para que as aplicações interessadas possam acessá-los e trocar as informações. Além disso, o fato de se apoiar em tecnologias de padrão aberto, torna o consumo muito mais simplificado pelas mais diversas linguagens e tecnologias, desde uma aplicação de linha de comando, passando por um navegador e até mesmo um dispositivo móvel.

Sendo o HTTP o responsável pela infraestrutura de comunicação e o XML a linguagem que descreverá a comunicação, ainda era necessário um formato para formalizar a interação entre as partes (produtora e consumidora). Eis que surge o SOAP (*Simple*

Object Access Protocol). A finalidade do SOAP foi padronizar o conteúdo que trafega entre as partes sob o protocolo HTTP. O SOAP é baseado em XML e, consequentemente, é extensível, e em pouco tempo ele ficou popular e foi utilizado em larga escala. Cada empresa utilizou as especificações criadas e regidas por órgãos independentes, criando um ferramental para que seja possível a criação e consumo destes tipos de serviços. A Microsoft fez a sua parte criando os *ASP.NET Web Services* (ASMX), que seguia as especificações do W3C para a criação e consumo de serviços dentro da plataforma .NET, e isso contribuiu ainda mais para a popularização destes tipos de serviços.

O grande uso destes tipos de serviços motivou a criação de algumas outras tecnologias para incrementar os *Web Services*, incluindo algumas características referentes a segurança, entrega de mensagens, transações, etc., e então uma série de especificações (WS-*) foram criadas (graças a extensibilidade que o SOAP possibilita), a fim de padronizar cada um destes novos recursos. A Microsoft correu para adequar os *ASP.NET Web Services* para suportar estas novas funcionalidades, e aí surgiu o WS-E (*Web Services Enhancements*).

Dentro do universo Microsoft há várias opções para comunicação distribuída, onde cada uma tinha um objetivo diferente, uma implementação única e uma API nada comum. Com o intuito de facilitar a comunicação distribuída dentro da plataforma .NET ela construiu o WCF – *Windows Communication Foundation*. Ele é um dos pilares do *.NET Framework*, sendo o framework de comunicação para toda a plataforma. O WCF unifica todas as tecnologias de comunicação distribuídas que a Microsoft tinha até então. A proposta com ele é tornar a construção e consumo de serviços algo simples, onde o foco está apenas nas regras de negócios, e detalhes com a infraestrutura, comunicação, protocolo, etc., seriam poucas configurações a serem realizadas.

O WCF foi construído com o protocolo (HTTP, TCP, MQ) sendo apenas uma forma de comunicação, onde o SOAP é o padrão que define todo o tráfego das mensagens, independentemente do protocolo que seja utilizado para a comunicação.

Como vimos até então, o SOAP se apresenta como a solução ideal para a integração entre os mais variados sistemas, de qualquer tecnologia, justamente por trabalhar com padrões abertos e gerenciados por órgãos independentes. Só que o SOAP foi ganhando uma dimensão em funcionalidades, e grande parte das plataformas e linguagens não conseguiram acompanhar a sua evolução, tornando cada vez mais complicado a sua adoção, pois a cada nova funcionalidade adicionada, uma série de novos recursos precisam serm, também, adicionados nas plataformas/linguagens para suportar isso.

Motivada pela complexidade que o SOAP ganhou ao longo do tempo, e também pela dificuldade na evolução por parte dos produtos e consumidores destes tipos de serviços, uma nova alternativa foi ganhando cada vez mais espaço: o REST (*Representational State Transfer*). O estilo REST vai contra tudo o que prega o SOAP, ou seja, enquanto o SOAP entende que o HTTP é uma forma de tráfego da informação, o REST abraça o HTTP, utilizando integralmente todos os seus recursos.

Orientado à recursos, o estilo REST define o acesso à elementos como se eles fossem um conjunto predefinido de informações quais queremos interagir (acessível através de sua URI), extraíndo e/ou criando estes recursos, diferindo do estilo baseado em RPC (*Remote Procedure Call*) que usávamos até então, que define que o acesso é à alguma funcionalidade.

Novamente, correndo atrás do que o mercado estava utilizando como principal meio de comunicação e integração, a Microsoft começou a trabalhar em uma nova versão do WCF, já que era o pilar de comunicação da plataforma, para agregar a possibilidade de construir serviços baseados em REST. Passamos então a ter a possibilidade de criar este tipo de serviço no WCF, mas como o mesmo foi construído abstraíndo a figura do protocolo (HTTP e outros), ficou complexo demais criar serviços REST sobre ele, já que era necessário entender toda a infraestrutura do WCF para criar um serviço deste tipo, pois para interagir com os elementos do HTTP, tínhamos que lidar com objetos do *framework* que passava a expor o acesso aos recursos do HTTP (URIs, *headers*, requisição, resposta, etc.).

Como a criação de serviços REST no WCF acabou ficando mais complexo do que deveria, a Microsoft começou a investir na criação de uma nova forma de construir e consumir serviços REST na plataforma .NET. Inicialmente ele seria uma espécie de agregado ao WCF, mas pela complexidade de manter o modelo de classes que já existia, a Microsoft decidiu criar a vincular isso ao ASP.NET, tornando-o uma plataforma completa para a criação de qualquer funcionalidade para ser exposta via Web, e assim surgiu o ASP.NET Web API. Como toda e qualquer tecnologia, o ASP.NET Web API abstrai vários elementos que são necessários para o funcionamento destes tipos de serviços. Mas isso não é motivo para não entender quais são e para o que eles servem.

Web API nada mais é que uma *interface* que um sistema expõe através do HTTP para ser acessado pelos mais variados clientes, utilizando as características do próprio protocolo para interagir com o mesmo. E como estamos agora fundamentados no HTTP, é necessário conhecermos alguns elementos que passam a ser essências para a construção destes serviços. Sendo assim, quando estamos lidando com a Web, toda e qualquer requisição e resposta possuem algumas características específicas que envolvem todo o processo, a saber:

- **Recursos:** recursos podem ser qualquer coisa que esteja disponível e desejamos acessar. Podemos entender o recurso como uma página HTML, produtos de uma loja, um video, uma imagem e também uma funcionalidade que pode ser acessível através de outras aplicações.
- **URIs:** Todo e qualquer recurso para ser acessível, precisa ter uma URI que determina o endereço onde ele está localizado. A URI segue a seguinte estrutura: `http://Servidor/MusicStore/Artistas?max+pezzali`.
- **Representação:** Basicamente é uma fotografia de um recurso em um determinado momento.
- **Media Type:** O recurso retornado ao cliente é uma fotografia em um determinado momento, seguindo um determinado formato. O *Media Type* é

responsável por determinado o formato em que a requisição é enviado e como a resposta é devolvida. Alguns *media types*: “*image/png*”, “*text/plain*”, etc.

Se depurarmos uma requisição para qualquer tipo de API via HTTP, podemos enxegar cada um dos elementos elencados acima. Novamente, apesar de não ser necessário entender os “bastidores” da comunicação, o entendimento básico nos dará conhecimento para depurar e encontrar mais facilmente eventuais problemas, entendimento para tirar proveito de alguns recursos que o próprio HTTP nos fornece.

O HTTP é baseado no envio e recebimento de mensagens. Tanto a requisição quanto a resposta HTTP possui a mesma estrutura. Temos o cabeçalho da mensagem, uma linha em branco, e o corpo da mensagem. O cabeçalho é composto por informações inerentes ao endereço do serviço e um conjunto de *headers*, que nada mais é que um dicionário contendo uma série de informações contextuais à requisição/resposta, que guiará o serviço ou o cliente a como tratar a mensagem.

```
POST http://localhost:1062/api/clientes HTTP/1.1
User-Agent: Fiddler
Content-Type: application/json
Host: localhost:1062
Content-Length: 48

{"Nome": "Israel", "Email": "ia@israelaece.com"}
```

Na primeira linha vemos a forma (verbo) e para onde (URI) a requisição está sendo encaminhada. Vale lembrar que o verbo tem um significado importante no HTTP, pois ele determina o formato da mensagem e como ela será processada. Vemos também o *Content-Type*, que determina o formato do conteúdo da mensagem, e que neste caso, está serializado em JSON. Depois da linha em branco temos o conteúdo da mensagem. Se olharmos o todo, vemos que o cliente está interessado em postar (adicionar) um novo cliente chamado “*Israel*” na coleção de clientes.

Depois de processado pelo cliente, e partindo do princípio que tudo deu certo, uma mensagem de resposta será retornada, e lá teremos informações referente ao processamento da requisição (referente ao sucesso ou a falha):

```
HTTP/1.1 201 Created
Content-Type: application/json; charset=utf-8
Location: http://localhost:1062/api/clientes/12
Content-Length: 58

{"Id": 12, "Nome": "Israel", "Email": "ia@israelaece.com"}
```

Estruturalmente a mensagem de resposta é igual a da requisição, com cabeçalho, linha em branco e o corpo. A resposta contém também a coleção de *headers*, incluindo o *Content-Type* do corpo, pois a entidade postada está sendo retornada ao cliente informando, através do código de *status 201 (Created)*, que o cliente foi criado com sucesso e o *Id* que foi gerado e atribuído à ele.

Claro que o HTTP fornece muito mais do que isso que vimos nestes exemplos de requisição e resposta. No decorrer dos próximos capítulos iremos abordarmos com mais detalhes cada um desses elementos e diversos outros recursos que sob demanda iremos explorando e, conseqüentemente, agregando e exibindo novas funcionalidades que o HTTP fornece.

Estrutura da API

Apesar de serviços REST utilizar completamente o HTTP, é importante que tenhamos suporte para a construção e consumo destes tipos de serviços. Precisamos entender como estruturar, configurar e distribuir estes tipos de serviços.

Para facilitar tudo isso, a Microsoft preparou o ASP.NET para suportar o desenvolvimento de serviços REST. A finalidade deste capítulo é introduzir a *template* de projeto que temos, a API, a configuração mínima para a construção e exposição do mesmo.

A Template de Projeto

A construção de Web API está debaixo de um projeto ASP.NET MVC 4, e logo na sequência da escolha deste projeto, você deve escolher qual a *template* de projeto. Neste caso, temos que recorrer a opção chamada Web API, conforme vemos nas imagens abaixo. O projeto já está pré-configurado com o que precisamos para criar um serviço REST e expor para que seja consumido, sem a necessidade de realizar muitas configurações.

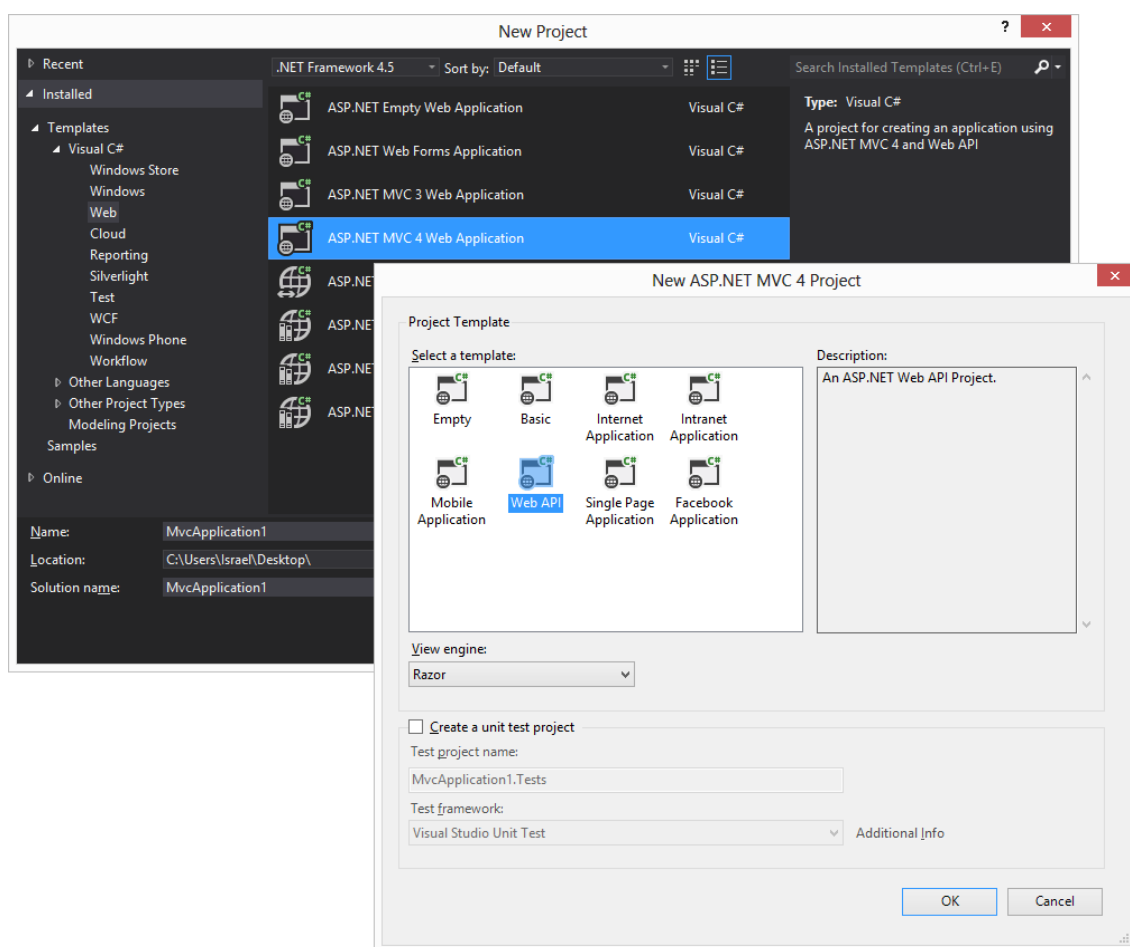


Figura 1 - Escolha da Template do Projeto

O projeto já está configurado com as referências (DLLs) necessárias que contêm os tipos e membros que utilizaremos na construção das Web APIs, sendo a principal delas o *assembly System.Web.Http.dll*. Dentro deste *assembly* temos vários *namespaces* com todos os elementos necessários que iremos utilizar para a construção de Web APIs. Além disso, já temos algumas configurações definidas para que seja possível criar e executar uma API criada, sem a necessidade de conhecer detalhes mais profundos em um primeiro momento.

Analizando os itens do *Solution Explorer*, vemos que além dos arquivos tradicionais de um projeto ASP.NET (como o *Global.asax* e o *Web.config*), vemos ali um arquivo chamado *WebApiConfig.cs*, que é uma classe que contém a configuração padrão do roteamento (que será abordado mais adiante) para o funcionamento dos serviços REST. Além disso, temos um serviço já criado como exemplo, que está contido na classe/arquivo *ValuesController.cs*.

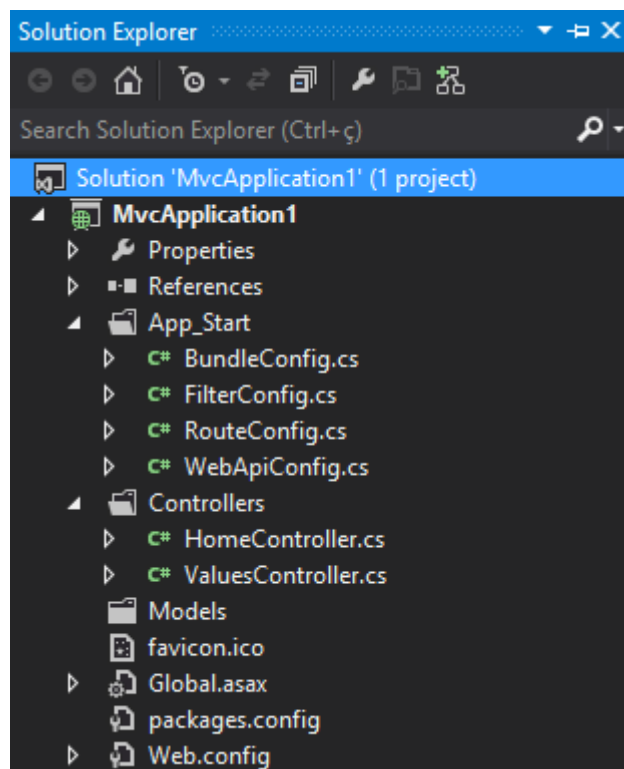


Figura 2 - Elementos Padrão de um Projeto Web API

A classe ApiController

A construção de Web APIs utilizando o ASP.NET segue uma certa simetria em relação a construção de um site baseado no padrão MVC. Para a construção de *views* o MVC exige que se tenha um controlador (*controller*) para receber, processar e retornar as requisições que são realizadas para o site.

De forma parecida trabalha a Web API. Todas as Web APIs construídas no ASP.NET devem herdar de uma classe abstrata chamada *ApiController*. Esta classe fornece toda a infraestrutura necessária para o desenvolvimento destes tipos de serviços, e entre as

suas tarefas, temos: fazer a escolha do método a ser executado, conversão das mensagens em parâmetros, aplicação de eventuais filtros (de vários níveis), etc. Cada requisição, por padrão, terá como alvo um método dentro desta classe, que será responsável por processar a mesma e retornar o resultado.

A criação de uma Web API pode ser realizada de forma manual herdando da classe *ApiController*, ou se preferir, pode ser utilizado o assistente que o próprio *Visual Studio* disponibiliza, onde já existe algumas opções predefinidas para que a classe já seja criada com a estrutura básica para alguns cenários.

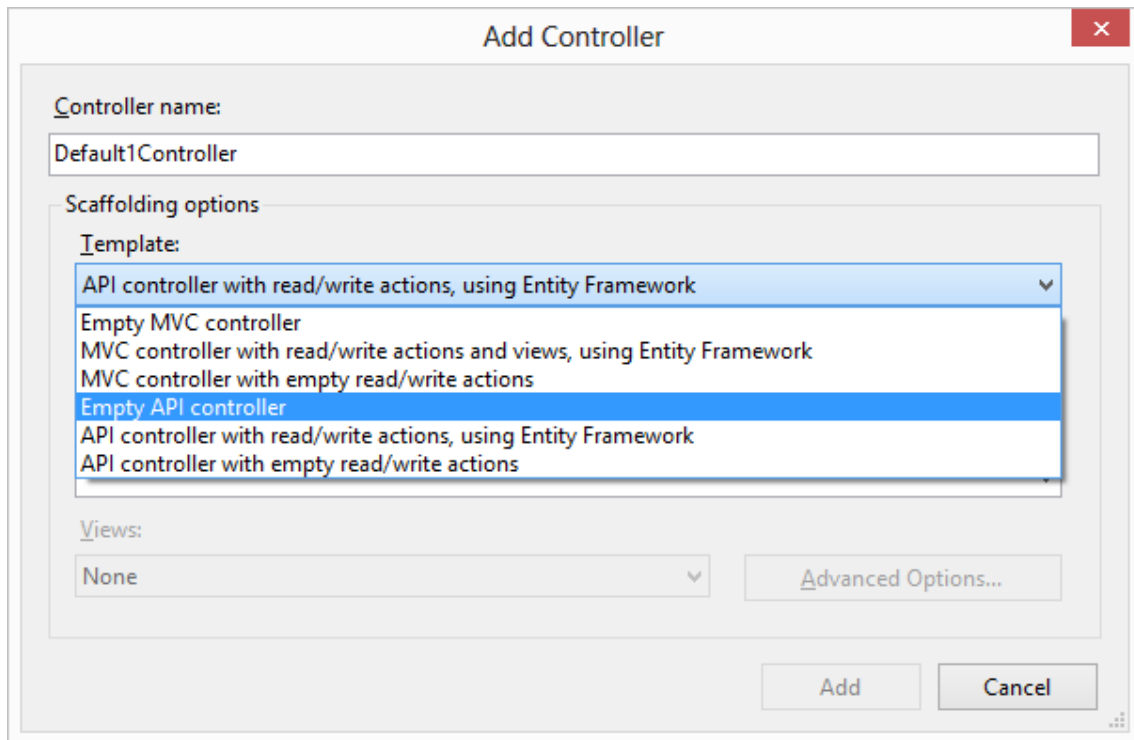


Figura 3 -Opções para a criação do Controller

Independentemente da forma que você utilize para a criação do controlador, sempre teremos uma classe que servirá como a base para a API, e o código abaixo ilustra a herança da classe *ApiController*:

```
using System.Web.Http;

namespace MusicStore.Controllers
{
    public class ArtistasController : ApiController
    {
    }
}
```

Uma consideração importante é com relação ao sufixo *Controller* que o nome da classe que representará a API deverá ter. Apesar dele ser transparente para o cliente, isso é utilizado pelo ASP.NET para encontrar o *controller* durante a requisição. Podemos notar também que a classe não possui nenhum método. Para que ela comece a ter sentido,

precisamos criar os métodos que atenderão as requisições que serão feitas para este serviço. Da mesma forma que fazemos no MVC, aqui criaremos as ações que retornarão dados aos clientes, formatados em algum padrão e, opcionalmente, os clientes poderão parametrizar a requisição, caso o serviço permita isso.

A classe pode ter quantos métodos forem necessários, apenas temos que ter um pouco do bom senso aqui para não darmos mais funcionalidade do que deveria para a mesma, considerando a granularidade. A criação dos métodos possuem algumas convenções que se seguidas corretamente, não haverá maiores configurações a serem realizadas para que ele já esteja acessível ao rodar o serviço.

Mas como que o ASP.NET escolhe qual dos métodos acessar? O HTTP possui o que chamamos de verbos (algumas vezes chamados de métodos), e os mais comuns são: GET, POST, PUT e DELETE, e cada um deles indica uma determinada ação a ser executada em um recurso específico.

- **GET:** Está requisitando ao serviço um determinado recurso, apenas isso. Este verbo deve apenas extrair a informação, não alterando-a.
- **POST:** Indica ao serviço que a ele deve acatar o recurso que está sendo postado para o mesmo, e que muito vezes, o adicionamos em algum repositório.
- **PUT:** Indica que ao serviço que o recurso que está sendo colocado deve ser alterado se ele já existir, ou ainda, pode ser adicionado caso ele ainda não exista.
- **DELETE:** Indica que o serviço deve excluir o recurso.

Mas o que isso tem a ver com o Web API? O ASP.NET já mapeia todos estes conhecidos verbos do HTTP para métodos que estejam criados o interior do *controller*. Para que isso aconteça, precisamos definir os métodos com o nome de cada verbo acima descrito, e com isso, automaticamente, quando o ASP.NET recepcionar a requisição, esse será o primeiro critério de busca aos métodos.

```
public class ArtistasController : ApiController
{
    public Artista Get(int id)
    {
    }

    public void Post(Artista novoArtista)
    {
    }
}
```

É claro que não estamos condicionados a trabalhar desta forma. Se você quer criar uma API em português, talvez utilizar a configuração padrão não seja a melhor opção pela coerência. Podemos nomear os métodos da forma que desejarmos, mas isso nos obrigará a realizar algumas configurações extras, para direcionar o ASP.NET a como encontrar o método dentro da classe, pois agora, difere daquilo que foi previamente configurado.

Para realizar essa configuração, vamos recorrer à alguns atributos que já existem dentro do ASP.NET e que foram construídos para cada um dos verbos do HTTP: *HttpGetAttribute*, *HttpPutAttribute*, *HttpPostAttribute*, *HttpDeleteAttribute*, etc. Quando um destes atributos é colocado em um método, ele permitirá que ele seja acessível através daquele verbo. Utilizando o mesmo exemplo anterior, se alterarmos o nome dos métodos apenas, eles deixarão de estar acessíveis aos clientes.

```
public class ArtistasController : ApiController
{
    [HttpGet]
    public Artista Recuperar(int id)
    {
    }

    [HttpPost]
    public void Adicionar(Artista novoArtista)
    {
    }
}
```

E ainda, como alternativa, podemos recorrer ao atributo *ActionNameAttribute* para alterar o nome que será publicado em relação aquele que definido no método, dando a chance de utilizar uma convenção de nomenclatura para escrita e outra para publicação.

```
public class ArtistasController : ApiController
{
    [ActionName("Recuperar")]
    public Artista Get(int id)
    {
    }
}
```

Como comentado acima, existe um atributo para cada verbo. Para uma maior flexibilidade, temos também o atributo *AcceptVerbsAttribute*, que nos permite informar em seu construtor quais os verbos que podem chegar até o método em questão.

```
public class ArtistasController : ApiController
{
    [AcceptVerbs("POST", "PUT")]
    public void Adicionar(Artista novoArtista)
    {
    }
}
```

E se houver um método público que se encaixe com as regras dos verbos que foram comentadas acima e não queremos que ele esteja disponível publicamente, podemos proibir o acesso decorando o método com o atributo *NonActionAttribute*.

Parametrização dos Métodos

Todos os métodos podem receber informações como parâmetros, e como saída, podemos retornar alguma informação que caracteriza o sucesso ou a falha referente a

execução do mesmo. Quando falamos de métodos que são disponibilizados para acesso remoto, isso não é diferente.

Os métodos podem necessitar alguns parâmetros para executar a tarefa que está sendo solicitada. Os parâmetros podem ter tipos mais simples (como inteiro, *string*, etc.) até objetos mais complexos (*Usuario*, *Pedido*, *Produto*, etc.). A utilização de objetos complexo nos permite descrever o nosso negócio, tornando-o bem mais intuitivo que criar um método contendo uma infinidade de parâmetros.

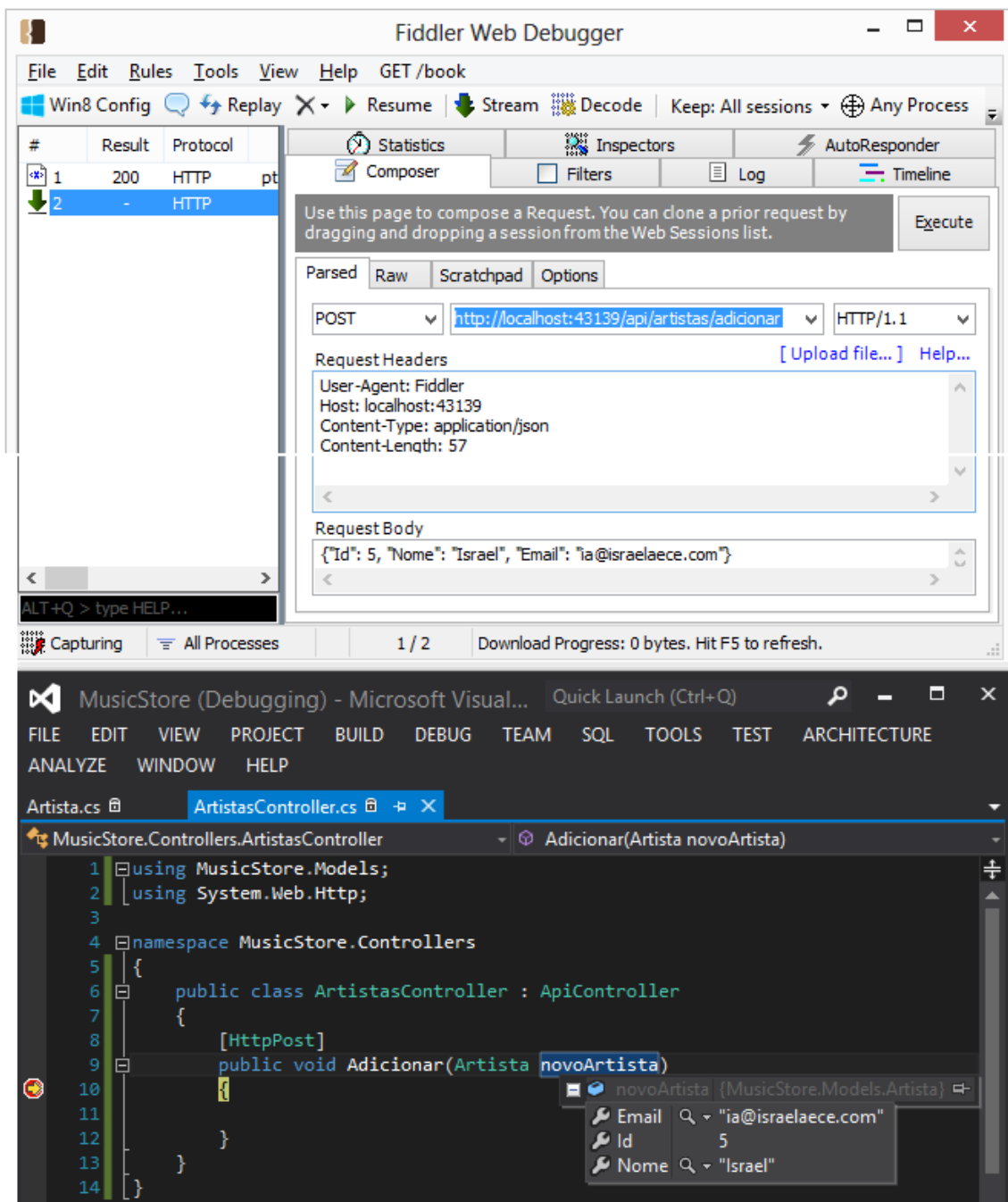


Figura 4 - Postando um recurso de um objeto complexo.

Apesar de estarmos habituados a declarar o método, seus parâmetros e resultado da forma tradicional quando estamos construindo uma Web API, eles são trafegados entre o cliente e o serviço utilizando o modelo do HTTP, ou seja, eles poderão ser carregados através de *querystrings*, *headers* ou do corpo das mensagens.

De forma semelhante ao que ocorre no ASP.NET MVC, o ASP.NET Web API é capaz de mapear as *querystrings* (que são tipos simples (textos, números, etc.)) para os parâmetros dos métodos do *controller* de forma automática. Já os objetos complexos viajam entre o cliente e o serviço (e vice-versa) no corpo da mensagem. Toda a mágica da transformação da mensagem em um objeto customizado é realizada pelos formatadores de conteúdo, qual terá um capítulo específico para abordar este assunto.

Quando estamos acessando os métodos através do verbo GET, as *querystrings* são mapeadas para os parâmetros destes métodos; já quando realizamos o POST, o corpo da mensagem é transformado no objeto complexo, conforme vimos na imagem acima. Existe algumas opções mais rebuscadas se quisermos customizar como estes parâmetros são lidos das mensagens, e para isso, podemos recorrer aos atributos *FromBodyAttribute* ou ao *FromUriAttribute*. Estes atributos nos permite direcionar o ASP.NET a buscar os valores para abastecer os parâmetros em locais diferentes dos padrões.

Podemos abastecer os parâmetros baseados nas *querystrings* ou no corpo da mensagem. O corpo da mensagem pode estar em diversos formatos (abordaremos com mais detalhes adiante), e um deles é a postagem sendo realizada através de um formulário HTML.

```
<form action="http://localhost:43139/api/Artistas/Adicionar" method="post">
  <input type="text" id="Id" name="Id" />
  <input type="text" id="Nome" name="Nome" />
  <input type="text" id="Email" name="Email" />
  <input type="submit" name="Enviar" value="Enviar" />
</form>
```

Ao preencher os campos e postar o formulário, podemos capturar a requisição e analisar o que está sendo enviado ao serviço mencionado. O que chama atenção na requisição abaixo é o *header Content-Type* definido como *application/x-www-form-urlencoded*, que corresponde ao valor padrão para formulários HTML. No corpo da mensagem temos os campos do formulário separados pelo caracter &. Já os espaços são substituídos pelo caracter +. E, para finalizar, o nome do controle é definido como chave, enquanto o conteúdo do controle é definido como valor no dicionário.

```
POST http://localhost:43139/api/artistas/adicionar HTTP/1.1
User-Agent: Fiddler
Host: localhost:43139
Content-Type: application/x-www-form-urlencoded
Content-Length: 41

Id=12&Nome=Israel&Email=ia@israelaece.com
```

Ao postar um formulário para um método que possui um objeto, o ASP.NET Web API já é capaz de extrair as informações do corpo da mensagem (a partir do dicionário de valores) e abastecer cada propriedade deste objeto. Se desejarmos, podemos ler individualmente as informações que estão sendo encaminhadas, não acionando o formatador de conteúdo, utilizando um outro tipo de codificação quando estamos postando o formulário.

No exemplo acima, o formulário foi postado utilizando o formato *application/x-www-form-urlencoded*, que é mais ou menos os valores do formulário codificados como se fossem itens de uma *querystring*. Existe um outro formato que é o *multipart/form-data*, que possui uma codificação mais sofisticada. Em geral, ele é utilizado em conjunto com o elemento do tipo *file*, que é quando queremos fazer *upload* de arquivos para o servidor.

Quando postamos um arquivo, automaticamente o tipo é definido como sendo *multipart/form-data* e na sequência, vemos o arquivo anexado.

```
POST http://localhost:43139/api/artistas/AlterarFoto HTTP/1.1
Content-Type: multipart/form-data; boundary=-----acebdf13572468
User-Agent: Fiddler
Host: localhost:43139
Content-Length: 1168

-----acebdf13572468
Content-Disposition: form-data; name="fieldNameHere"; filename="MaxPezzali.png"
Content-Type: image/png

---- REMOVIDO POR QUESTÕES DE ESPAÇO ----
```

Mas para receber este tipo de requisição temos que preparar o serviço. Note que ele não recebe nenhum parâmetro; ele é extraído do corpo da mensagem ao executar o método *ReadAsMultipartAsync*, que assincronamente lê e “materializa” os arquivos, salvando automaticamente no caminho informado no *provider*. Se desejarmos, podemos iterar através da propriedade *Contents*, acessando individualmente cada um dos arquivos que foram postados.

```
[HttpPost]
public async Task<HttpResponseMessage> AlterarFoto()
{
    var provider =
        new MultipartFormDataStreamProvider(
            HttpContext.Current.Server.MapPath("~/Uploads"));

    return await Request
        .Content
        .ReadAsMultipartAsync(provider)
        .ContinueWith<HttpResponseMessage>(t =>
        {
            if (t.IsFaulted || t.IsCanceled)
                return Request.CreateErrorResponse(
                    HttpStatusCode.InternalServerError, t.Exception);

            return Request.CreateResponse(HttpStatusCode.OK);
        });
}
```

Apesar das técnicas acima serem interessantes, utilizar uma delas pode ser um problema ao trabalhar com serviços REST, devido ao fato de que em algumas situações haver a necessidade de ter o controle total das mensagens HTTP.

Com o intuito de facilitar e dar mais controle para ao desenvolvedor, a Microsoft incluiu nesta API classes que representam a mensagem de requisição (*HttpRequestMessage*) e de resposta (*HttpResponseMessage*). Cada uma dessas classes trazem várias propriedades, onde cada uma delas expõe características do protocolo HTTP, tais como: *Content*, *Headers*, *Method*, *Uri*, *StatusCode*, etc.

O serviço passará a utilizar essas classes em seus métodos, ou seja, receberá um parâmetro do tipo *HttpRequestMessage*, que possui todas as informações necessárias solicitadas pelo cliente, enquanto o retorno será do tipo *HttpResponseMessage*, que será onde colocaremos todas as informações de resposta para o mesmo cliente, sendo essas informações o resultado em si, o código de status do HTTP, eventuais *headers*, etc.

```
public class ArtistasController : ApiController
{
    [HttpGet]
    public HttpResponseMessage Ping(HttpRequestMessage info)
    {
        return new HttpResponseMessage(HttpStatusCode.OK)
        {
            Content = new StringContent("ping...")
        };
    }
}
```

O corpo da mensagem de retorno também pode ser customizado, onde podemos retornar uma simples *string* ou até mesmo um objeto complexo. Isso tudo será abordado com maiores detalhes nos capítulos seguintes.

Métodos Assíncronos

A Microsoft incorporou diretamente no C# e no VB.NET o suporte para programação assíncrona. A ideia é facilitar a programação assíncrona, que não era nada trivial até o momento, tornando a escrita de um código assíncrono muito próximo a escrita de um código síncrono, e nos bastidores, o compilador faz grande parte do trabalho. Grande parte das funcionalidades do *.NET Framework* que já possuem suporte nativo ao consumo em formato assíncrono, foram readaptados para que assim, os desenvolvedores possam fazer uso dos novos recursos oferecidos pela linguagem para consumi-los.

Com o ASP.NET Web API também podemos fazer com que os métodos expostos pela API sejam processados assincronamente, usufruindo de todos os benefícios de um método ser executado de forma assíncrona.

Vamos supor que o nosso serviço de *Artistas* deve recorrer à um segundo serviço para extrair as notícias referentes à um determinado artista. No interior do método que retorna o artista, faremos a chamada para o serviço de *Notícias* e esperamos pelo

resultado, que ao voltar, efetuamos o *parser* e, finalmente, convertemos para o formato esperado e retornamos ao cliente.

Ao executar este tipo de serviço, a requisição será bloqueada pelo *runtime* até que o resultado seja devolvido para o serviço. Isso prejudica, e muito, a escalabilidade do serviço. O fato da *thread* ficar bloqueada enquanto espera pelas notícias, ela poderia estar atendendo à outras requisições, que talvez não exijam recursos de terceiros (*I/O bound*). O fato de disponibilizar a *thread* para que ela possa atender à outras requisições, farão com que elas não esperem por um tempo indeterminado, pois como dependemos do resultado de um terceiro, poderíamos arranjar muito trabalho para esta *thread*, até que ela precise retomar o trabalho da requisição anterior.

Para implementar o *controller* da API de forma assíncrona, exigirá algumas mudanças, mas nada que faça com que seja necessário escrever e/ou gerenciar uma porção de código para garantir o assincronismo (*IAsyncResult* por exemplo). Com isso, o primeiro detalhe a notar na escrita da ação assíncrona, é a exigência da *keyword async*, que faz do C#.

```
public class ArtistasController : ApiController
{
    [HttpGet]
    public async Task<Artista> Recuperar(int id)
    {
        var artista = this.repositorio.Buscar(id);

        using (var client = new HttpClient())
            artista.Noticias =
                await
                    (await client.GetAsync(ServicoDeNoticias))
                    .Content
                    .ReadAsStringAsync<IEnumerable<Noticia>>();

        return artista;
    }
}
```

Tratamento de Erros

Durante a execução, uma porção de exceções podem acontecer, sejam elas referentes à infraestrutura ou até mesmo à alguma regra de negócio, e o não tratamento correto delas, fará com as mesmas não sejam propagadas corretamente ao cliente que consome a API. A maior preocupação aqui é mapear o problema ocorrido para algum código HTTP correspondente.

Isso se faz necessário porque exceções são características de plataforma, e precisamos de alguma forma expressar isso através de algum elemento do HTTP, para que cada um dos – mais variados – clientes possam interpretar de uma forma específica. Por padrão, todos os erros que ocorrem no serviço e não tratados, retornando ao cliente o código de *status* 500, que indica um erro interno do serviço (*Internal Server Error*).

O ASP.NET Web API possui uma exceção chamada *HttpResponseException*, que quando é instanciada definimos em seu construtor o código de status do HTTP indicando o erro

que ocorreu. Para exemplificar o uso deste tipo, podemos disparar o erro 404 (*Not Found*) se o cliente está solicitando um artista que não existe.

```
public class ArtistasController : ApiController
{
    [HttpGet]
    public Artista Recuperar(int id)
    {
        var artista = this.repositorio.Buscar(id);

        if (artista == null)
            throw new HttpResponseException(
                new HttpResponseMessage(HttpStatusCode.NotFound)
                {
                    Content = new StringContent("Artista não encontrado"),
                    ReasonPhrase = "Id Inválido"
                });

        return artista;
    }
}
```

Ainda como opção temos a classe *HttpError* para expressar o problema que ocorreu dentro do método. A principal vantagem de utilizar esta classe em relação aquele que vimos acima, é que o conteúdo que identifica o erro é serializado no corpo da mensagem, seguindo as regras de negociação de conteúdo.

```
public class ArtistasController : ApiController
{
    [HttpGet]
    public HttpResponseMessage Recuperar(int id)
    {
        var artista = this.repositorio.Buscar(id);

        if (artista == null)
            return Request.CreateErrorResponse(
                HttpStatusCode.NotFound,
                new HttpError("Artista não encontrado"));
        else
            return Request.CreateResponse(HttpStatusCode.OK, artista);
    }
}
```

Tratar as exceções *in-place* (como acima) pode não ser uma saída elegante, devido a redundância de código. Para facilitar, podemos centralizar o tratamento em nível de aplicação, o que permitirá com que qualquer exceção não tratada no interior da ação, será capturada por este tratador, que por sua vez analisará o erro ocorrido, podendo efetuar algum tipo de *logging* e, finalmente, encaminhar o problema ao cliente. É neste momento que podemos efetuar alguma espécie de tradução, para tornar a resposta coerente ao que determina os códigos do HTTP, como por exemplo: se algum erro relacionado à autorização, devemos definir como resposta 403 (*Forbidden*); já se algum informação está faltante (assim como vimos no exemplo acima), devemos retornar o *status* 400 (*Bad Request*); já se o registro procurado não foi encontrado, ele deverá receber o código 404 (*Not Found*), e assim por diante.

Para isso vamos recorrer a criação de um filtro customizado para centralizar a tradução de algum problema que acontecer. Só que neste caso, temos uma classe abstrata chamada de *ExceptionHandlerAttribute*, que já fornece parte da infraestrutura necessária para o tratamento de erros que ocorrem, e é equivalente ao atributo *HandleErrorAttribute* que temos no ASP.NET MVC. Tudo o que precisamos fazer aqui é sobrescrever o método *OnException* e definir toda a regra de tradução necessária. Abaixo um exemplo simples de como proceder para realizar esta customização:

```
public class ExceptionTranslatorAttribute : ExceptionFilterAttribute
{
    public override void OnException(HttpContext ctx)
    {
        var errorDetails = new ErrorDetails();
        var statusCode = HttpStatusCode.InternalServerError;

        if (ctx.Exception is HttpException)
        {
            var httpEx = (HttpException)ctx.Exception;

            errorDetails.Message = httpEx.Message;
            statusCode = (HttpStatusCode)httpEx.GetHttpCode();
        }
        else
        {
            errorDetails.Message = "*** Internal Server Error ***";
        }

        ctx.Result =
            new HttpResponseMessage<ErrorDetails>(errorDetails, statusCode);
    }
}
```

Não vamos se aprofundar muito aqui, mas a criação de filtros é um ponto de extensibilidade, e que teremos um capítulo específico para abordar este assunto. O que precisamos nos atentar aqui é existem vários escopos para registros um filtro deste tipo, podendo ele ser aplicado em nível de ação, de *controller* ou globalmente.

O código abaixo registra o tradutor de exceções em nível global, recorrendo ao arquivo *Global.asax* para isso, que será utilizado por todo e qualquer API que estiver abaixo deste projeto. Maiores detalhes sobre a configuração de serviço, serão abordados em breve, ainda neste capítulo.

```
GlobalConfiguration
    .Configuration
    .Filters
    .Add(new ExceptionTranslatorAttribute());
```

Validações

Como vimos anteriormente, o ASP.NET Web API é capaz de construir um objeto complexo baseado no corpo da requisição. Mas como acontece em qualquer aplicação, pode ser que o objeto esteja com um valores inválidos, o que impedirá do mesmo ser processado corretamente.

Mesma se tratando de serviços onde o foco é a integração entre aplicações, é necessário que se faça a validação, para garantir que a requisição tenha as informações corretas para ser processada.

Uma das opções que temos para isso, é fazer como já realizamos no ASP.NET MVC: recorrer aos *Data Annotations* do *.NET Framework* para validar se o objeto encontra-se em um estado válido. Para isso, basta decorarmos as propriedades com os mais variados atributos que existem debaixo do *namespace* (e *assembly*) *System.ComponentModel.DataAnnotations*.

Para iniciar devemos referenciar em nosso projeto o assembly que contém os tipos para a validação: *System.ComponentModel.DataAnnotations.dll*. A partir do momento em que referenciamos este *assembly* na aplicação, podemos aplicar em nossos objetos os atributos que determinam as regras que cada propriedade deverá respeitar. No exemplo abaixo ilustra estamos informando ao ASP.NET que a propriedade *Nome* deve ser preenchida e a propriedade e-mail deve representar um endereço de e-mail em formato válido.

```
public class Artista
{
    public int Id { get; set; }

    [Required]
    public string Nome { get; set; }

    [EmailAddress]
    public string Email { get; set; }
}
```

Só que os atributos não funcionam por si só. O ASP.NET Web API já está preparado para realizar a validação do objeto que está sendo postado, avaliando se cada propriedade do mesmo está seguindo os atributos aplicados à ela.

Como o ASP.NET já realiza toda a validação, o que nos resta no interior do método é verificar se o objeto está válido. A classe *ApiController* fornece uma propriedade chamada *ModelState*. Essa propriedade dá ao *controller* a possibilidade de verificar se o modelo que está sendo postado está ou não valido através da propriedade booleana *IsValid*, e além disso, nos permite acessar as propriedades problemáticas, ou melhor, aquelas que não passaram na validação, que está baseada nos atributos que foram aplicados.

Abaixo estamos avaliando essa propriedade, e se houver alguma informação errada, retornamos ao cliente o código 400 (*Bad Request*), indicando que há problemas na requisição que foi encaminhada para o serviço. O que é importante notar é que o método que recebe e trata a requisição não possui regras de validação para o objeto. Isso acabou sendo terceirizado para os *Data Annotations* do *.NET*, que acabam realizando toda a validação em estágios anteriores ao processamento da requisição, e com isso o método fica apenas com a responsabilidade de processar se a requisição foi encaminhada da forma correta.


```

public class ArtistasController : ApiController
{
    [HttpPost]
    public HttpResponseMessage Adicionar(Artista artista)
    {
        if (ModelState.IsValid)
        {
            //...
            return new HttpResponseMessage(HttpStatusCode.Created);
        }

        return new HttpResponseMessage(HttpStatusCode.BadRequest);
    }
}

```

Configuração

Ao trabalhar com algum tipo de projeto .NET, estamos acostumados a lidar com a configuração e extensibilidade do mesmo utilizando os arquivos de configuração (*App.config* ou *Web.config*). O ASP.NET Web API trouxe toda a configuração para ser realizada através do modelo imperativo (via código) ao invés do modelo declarativo (via Xml).

Para centralizar toda a configuração das APIs, temos a disposição um objeto global que possui uma variedade de métodos para registrarmos todos os elementos que podemos customizar. Toda essa configuração é realizada a partir do método *Application_Start* do arquivo *Global.asax*. Como a *template* do projeto segue a mesma linha do ASP.NET MVC, a configuração das APIs são realizadas em uma classe estática chamada *WebApiConfig*, que recebe como parâmetro um objeto do tipo *HttpConfiguration*, acessível através da propriedade estática *Configuration* da classe *GlobalConfiguration*.

```

public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        WebApiConfig.Register(GlobalConfiguration.Configuration);
    }
}

public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}

```

Inicialmente o configurador das APIs possui apenas um código que registra a rota padrão para que as requisições sejam encaminhadas corretamente para os métodos. Não vamos nos estender aqui, pois há um capítulo dedicado a este assunto. A classe *HttpConfiguration* possui uma série de propriedades que serão exploradas ao longo dos próximos capítulos.

Elementos do HTTP

Além das informações que são enviadas e recebidas através do corpo da mensagem, podemos recorrer à alguns elementos inerentes ao HTTP, para incluir informações que fazem parte da requisição, como por exemplo, ID para controle da segurança, *tags* para *caching*, etc.

Para essas informações, podemos recorrer aos famosos elementos do HTTP que são a coleção de *headers*, de *querystrings* e *cookies*. Como falamos anteriormente, se quisermos ter o controle da mensagem, o método/ação deve lidar diretamente com as classes *HttpRequestMessage* e *HttpResponseMessage*. Esses objetos expõem propriedades para acesso à estes recursos, e métodos de extensão facilitam o acesso a estas informações.

O *cookie* nada mais é que um *header* dentro da requisição, que quando o serviço adiciona um *cookie*, chega uma “instrução” (*header*) ao cliente chamado *Set-Cookie*, que faz com que o mesmo seja salvo e adicionado nas futuras requisições. No ASP.NET Web API temos uma classe chamada *CookieHeaderValue*, que representa um *cookie*, que depois de criado, podemos adicionar na coleção de *cookies* da resposta e entregar ao cliente.

```
[HttpGet]
public HttpResponseMessage Ping(HttpRequestMessage request)
{
    var response = new HttpResponseMessage(HttpStatusCode.OK);
    var id = request.Headers.GetCookies("Id").FirstOrDefault();

    if (id == null)
    {
        response.Headers.AddCookies(new CookieHeaderValue[]
        {
            new CookieHeaderValue("Id", Guid.NewGuid().ToString())
            {
                Expires = DateTimeOffset.Now.AddDays(10)
            }
        });
    }

    return response;
}
```

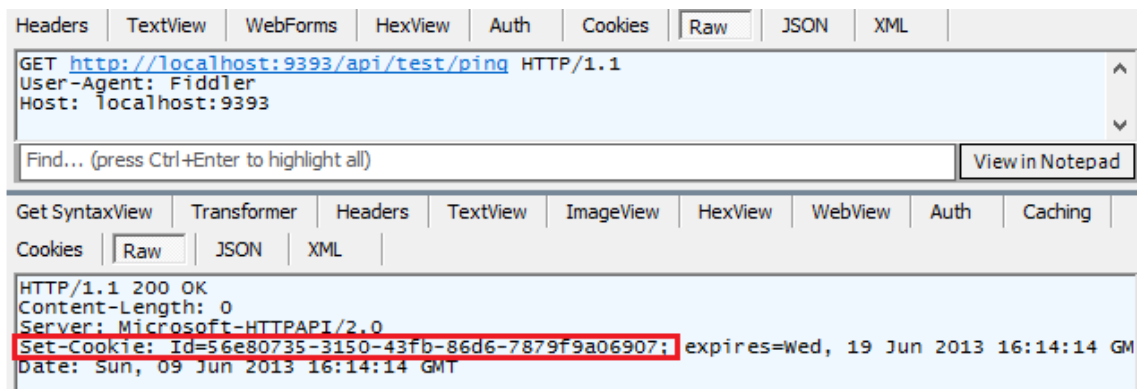


Figura 5 - Cookie sendo levado para o cliente.

Ao realizar a requisição, temos o *header Set-Cookie* sendo devolvido, que competirá ao cliente que está acessando, salvá-lo para enviar em futuras requisições. Se fizermos isso com o *Fiddler*, podemos perceber que quando a ação é executada, o *header Cookie* é interpretado e entregue ao ASP.NET Web API como uma instância da classe *CookieHeaderValue*.

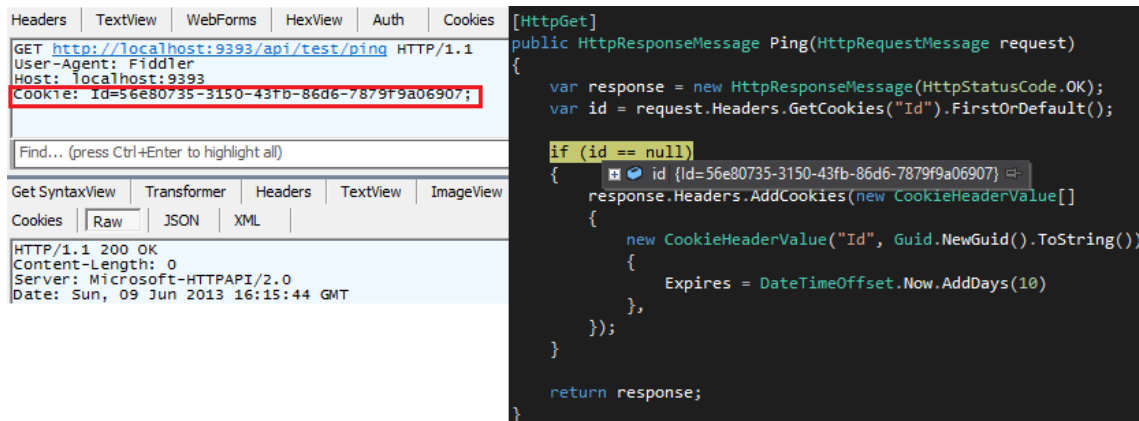


Figura 6 - Cookie sendo retornado para o serviço.

No exemplo acima apenas estamos lidando com um único valor, mas nada impede de termos uma informação mais estruturada dentro do *cookie*, ou seja, podemos armazenar diversos dados. A classe *CookieHeaderValue* possui um construtor que permite informar uma coleção do tipo *NameValueCollection*, que define uma lista de valores nomeados, onde podemos adicionar todos elementos que queremos que sejam levado ao cliente, e que serão separados pelo caracter “&”.

```
[HttpGet]
public HttpResponseMessage Ping(HttpRequestMessage request)
{
    var response = new HttpResponseMessage(HttpStatusCode.OK);
    var info = request.Headers.GetCookies("Info").FirstOrDefault();

    if (info == null)
    {
        var data = new NameValueCollection();
        data["Id"] = Guid.NewGuid().ToString();
        data["Type"] = "Simplex";
        data["Server"] = "SRV01";

        response.Headers.AddCookies(new CookieHeaderValue[]
        {
            new CookieHeaderValue("Info", data)
        });
    }

    return response;
}
```

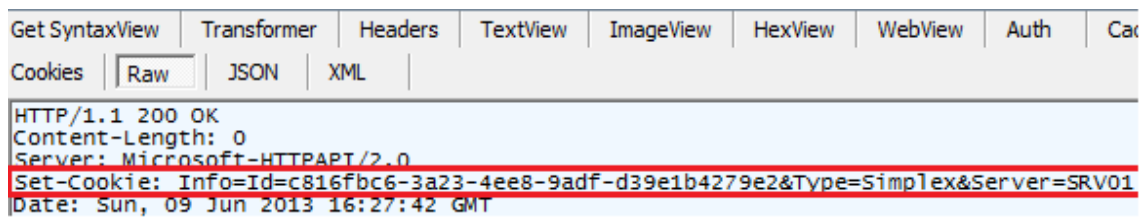


Figura 7 - Cookie com múltiplos valores.

Documentação

Quando estamos falando de serviços baseados em SOAP, temos um documento comando de WSDL (*Web Service Description Language*) baseado em XML, que descreve todas as características (definições) de um determinado serviço. É justamente esse documento que é utilizado por ferramentas como o *svcutil.exe* e a opção "*Add Service Reference*" do *Visual Studio*, para gerar uma classe que representará o *proxy*, que por sua vez, será utilizado pelos clientes para consumir o serviço como se fosse uma classe local, mas durante a execução, a mensagem será enviada ao ponto remoto.

Mas é importante dizer que mesmo serviços baseados em REST, também precisam, de alguma forma, expor alguma espécie de documentação, para descrever as ações que as APIs estão disponibilizando aos consumidores, apontando o caminho (URI) até aquele ponto, método/verbo (HTTP), informações que precisam ser passadas, formatos suportados, etc.

A ideia é apenas ser apenas informativo, ou seja, isso não será utilizado pelo cliente para a criação automática de um *proxy*. Pensando nisso, a Microsoft incluiu no ASP.NET Web API a opção para gerar e customizar as documentações de uma API.

Mas a documentação é sempre exibida, na maioria das vezes, de forma amigável ao consumidor, para que ele possa entender cada uma das ações, suas exigências, para que ele possa construir as requisições da forma correta. Sendo assim, podemos na própria aplicação onde nós temos as APIs, criar um *controller* que retorna uma *view* (HTML), contendo a descrição das APIs que estão sendo hospedadas naquela mesma aplicação.

```
public class DeveloperController : Controller
{
    public ActionResult Apis()
    {
        var explorer = GlobalConfiguration.Configuration.Services.GetApiExplorer();

        return View(explorer.ApiDescriptions);
    }
}
```

Note que estamos recorrendo ao método *GetApiExplorer*, disponibilizado através da configuração global das APIs. Este método retorna um objeto que implementa a *interface IApiExplorer*, que como o próprio nome sugere, define a estrutura que permite obter a descrição das APIs. Nativamente já temos uma implementação chamada *ApiExplorer*, que materializa todas as APIs em instâncias da classe *ApiDescription*, e

uma coleção deste objeto é retornada através da *propriedade* *ApiDescriptions*, e que repassamos para que a *view* possa renderizar isso.

Na *view*, tudo o que precisamos fazer é iterar pelo modelo, e cada elemento dentro deste laço representa uma ação específica que está dentro da API. A classe que representa a ação, possui várias propriedades, fornecendo tudo o que é necessário para que os clientes possam consumir qualquer uma destas ações. Abaixo temos o código que percorre e exibe cada uma delas:

```
@model IEnumerable<System.Web.Http.Description.ApiDescription>
<body>
    @foreach (var descriptor in this.Model)
    {
        <ul>
            <li><b>@descriptor.HttpMethod - @descriptor.RelativePath</b></li>
            <li>Documentation: @descriptor.Documentation</li>

            @if (descriptor.SupportedResponseFormatters.Count > 0)
            {
                <li>Media Types
                <ul>
                    @foreach (var mediaType in
descriptor.SupportedResponseFormatters.Select(
mt => mt.SupportedMediaTypes.First().MediaType))
                    {
                        <li>@mediaType</li>
                    }
                </ul>
            </li>
            }

            @if (descriptor.ParameterDescriptions.Count > 0)
            {
                <li>Parameters
                <ul>
                    @foreach (var parameter in descriptor.ParameterDescriptions)
                    {
                        <li>Name: @parameter.Name</li>
                        <li>Type: @parameter.ParameterDescriptor.ParameterType</li>
                        <li>Source: @parameter.Source</li>
                    }
                </ul>
            </li>
            }
        </ul>
    }
</body>
```

Ao acessar essa *view* no navegador, temos a relação de todas as ações que estão expostas pelas APIs. A visibilidade das ações é controlada a partir do atributo *ApiExplorerSettingsAttribute*, que possui uma propriedade booleana chamada *IgnoreApi*, que quando definida como *True*, omite a extração e, conseqüentemente, a sua visualização.

- **GET - api/Clientes**
- Documentation: Retorna todos os clientes.
- Media Types
 - application/json
 - application/xml
- **GET - api/Clientes/{id}**
- Documentation: Retorna um cliente pelo seu Id.
- Media Types
 - application/json
 - application/xml
- Parameters
 - Name: id
 - Type: System.Int32
 - Source: FromUri
- **POST - api/Clientes**
- Documentation: Inclui um novo cliente.
- Parameters
 - Name: cliente
 - Type: MvcApplication1.Models.Cliente
 - Source: FromBody
- **DELETE - api/Clientes/{id}**
- Documentation: Exclui um cliente existente.
- Parameters
 - Name: id
 - Type: System.Int32
 - Source: FromUri

Figura 8 - Documentação sendo exibida no browser.

É importante notar que na imagem acima, estamos apresentando a propriedade *Documentation*. A mensagem que aparece ali é uma customização que podemos fazer para prover essa informação, extraíndo-a de algum lugar. Para definir a descrição da ação, vamos criar um atributo customizado para que quando decorado no método, ele será extraído por parte da infraestrutura do ASP.NET, alimentando a propriedade *Documentation*. O primeiro passo, consiste na criação de um atributo para definir a mensagem:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
public class ApiDocumentationAttribute : Attribute
{
    public ApiDocumentationAttribute(string message)
    {
        this.Message = message;
    }

    public string Message { get; private set; }
}
```

O próximo passo é decorá-lo em cada uma das ações que quisermos apresentar uma informação/descrição. A classe abaixo representa a nossa API, e o atributo recentemente criado foi decorado em todas as ações, descrevendo suas respectivas funcionalidades:

```
public class ClientesController : ApiController
{
    [ApiDocumentation("Retorna todos os clientes.")]
    public IEnumerable<Cliente> Get()
    {
        //...
    }

    [ApiDocumentation("Retorna um cliente pelo seu Id.")]
    public Cliente Get(int id)
    {
        //...
    }

    [ApiDocumentation("Inclui um novo cliente.")]
    public void Post(Cliente cliente)
```

```

    {
        //...
    }

    [ApiDocumentation("Exclui um cliente existente.")]
    public void Delete(int id)
    {
        //...
    }
}

```

Só que o atributo por si só não funciona. Precisamos de algum elemento para extrair essa customização que fizemos, e para isso, a temos uma segunda *interface*, chamada *IDocumentationProvider*, que fornece dois métodos com o mesmo nome: *GetDocumentation*. A diferença entre eles é o parâmetro que cada um deles recebe. O primeiro recebe um parâmetro do tipo *HttpParameterDescriptor*, o que permitirá descrever, também, cada um dos parâmetros de uma determinada ação. Já o segundo método, recebe um parâmetro do tipo *HttpActionDescriptor*, qual utilizaremos para extrair as informações pertinentes à uma ação específica.

```

public class ApiDocumentationAttributeProvider : IDocumentationProvider
{
    public string GetDocumentation(HttpParameterDescriptor parameterDescriptor)
    {
        return null;
    }

    public string GetDocumentation(HttpActionDescriptor actionDescriptor)
    {
        var attributes =
            actionDescriptor.GetCustomAttributes<ApiDocumentationAttribute>();

        if (attributes.Count > 0)
            return attributes.First().Message;

        return null;
    }
}

```

Aqui extraímos o atributo que criamos, e se ele for encontrado, retornamos o valor definido na propriedade *Message*. A ausência deste atributo, faz com que um valor nulo seja retornado, fazendo com que nenhuma informação extra seja incluída para a ação.

E, finalmente, para incluir o provedor de documentação ao runtime do ASP.NET Web API, recorreremos à configuração das APIs, substituindo qualquer implementação existente para este serviço, para o nosso provedor que extraí a documentação do atributo customizado.

```

GlobalConfiguration.Configuration.Services.Replace(
    typeof(IDocumentationProvider),
    new ApiDocumentationAttributeProvider());

```

Roteamento

No início foi comentado que uma das características dos serviços REST, é que todo recurso é endereçável através de uma URI (*Universal Resource Identifier*). É através da URI que sabemos onde ele está localizado e, conseqüentemente, conseguiremos chegar até ele recurso e executar uma determinada ação.

Cada URI identifica um recurso específico, e ela tem uma sintaxe que é bastante comum para os desenvolvedores: [scheme]:[port]://[host]/[path][?query]. Cada pedaço dela tem uma representação e funcionalidade, que estão detalhados na listagem abaixo:

- **Scheme:** Identifica o protocolo que será utilizado para realizar a requisição à algum recurso, podendo ser: HTTP, HTTPS, FTP, etc.
- **Port:** Opção para definir o número da porta (no destino) que será utilizada para receber a requisição. Se omitido, utilizará a porta padrão do protocolo, onde no HTTP é a porta 80, no HTTPS a porta 443, etc.
- **Host:** O Host define o nome (fornecido pelo DNS) ou o endereço IP do servidor.
- **Path:** Esta parte da URI é qual identifica o recurso em si. É através dele que apontamos qual dos recursos que desejamos acesso dentro daquele servidor.
- **Query:** Trata-se de uma informação opcional que pode ser encaminhada para o serviço, fornecendo informações adicionais para o processamento da requisição. Um exemplo disso é incluir o identificador do recurso que está sendo solicitado.

Até pouco tempo atrás as URIs eram algo indecifrável para o usuário final, sendo apenas um endereço para algum recurso (seja ele qual for) em um ponto da rede. Aos poucos as URIs foram ficando cada vez mais legíveis, ou seja, tendo a mesma funcionalidade mas sendo mais amigáveis, o que facilita o retorno do usuário, ou ainda, a própria URI refletir o que ela representa. Há também benefícios relacionados ao SEO (*Search Engine Optimization*), onde os buscadores consideram as palavras encontradas na URI durante a pesquisa.

Durante o desenvolvimento do ASP.NET MVC, a Microsoft criou dentro dele um mecanismo de roteamento, que baseado em uma tabela de regras, ele é capaz de interpretar a requisição (URI) que está chegando para o mesmo e identificar o local, o serviço, a classe e o método a ser executado. Antes mesmo do lançamento oficial do ASP.NET MVC, a Microsoft fez com que este recurso de roteamento fosse desacoplado e levado para o *core* do ASP.NET, e com isso, pode ser utilizado pelo ASP.NET MVC, Web Forms e também pelo ASP.NET Web API, que é o nosso foco aqui.

Tudo começa pela configuração dessa tabela de regras que está localizada no arquivo *Global.asax*. Como sabemos, ao rodar a aplicação pela primeira vez, o evento *Application_Start* é disparado, e é justamente dentro dele onde o código que faz toda a configuração das rotas é colocada.

O objeto de configuração fornece uma coleção de rotas, e que na *template* de projeto que estamos utilizando, a rota padrão é colocada no arquivo *WebApiConfig*. Notamos que a rota padrão possui o nome de *“DefaultApi”*, e existe um segmento na seção *Path*

da URI chamado de “*api*”. Ele é utilizado para diferenciar e não causar conflito com outras eventuais rotas que já existam naquela aplicação, como por exemplo, aquelas que são criadas pelo projeto ASP.NET MVC. Essa necessidade se deve pelo fato de que podemos hospedar debaixo de um mesmo projeto, *controllers* que renderizam *views* (HTML) bem como *controllers* que retornam dados (Web API).

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{id}",
            defaults: new { id = RouteParameter.Optional }
        );
    }
}
```

Toda rota pode ser composta de literais e de *placeholders*, que são substituídos durante a execução. No exemplo acima, “*api*” trata-se de um literal, enquanto o {*controller*} e {*id*} são os *placeholders*. Quando a requisição é realizada, o ASP.NET tenta mapear os valores extraídos da URL e preencher os *placeholders* colocados na rota. Caso nenhuma rota se enquadre com a requisição que está solicitada, o erro 404 (*Not Found*) é retornado ao cliente.

É importante mencionar que os *placeholders* podem ter valor padrão definido. É isso que define o último parâmetro do método *MapHttpRoute*, chamado *defaults*. Podemos determinar se ele é opcional, e definir um valor padrão se ele for omitido durante a requisição através de um dicionário, onde cada item corresponde ao nome do *placeholder* e seu respectivo valor. Abaixo temos algumas URLs indicando se ela está ou não enquadrada na rota padrão:

URL	Método
api/Artistas/MaxPezzali	ArtistasController.Get(“MaxPezzali”)
api/Artistas/34	ArtistasController.Get(“34”)
Artistas/Eros	Como não há o sufixo “api/”, ele não se enquadrará na rota padrão, rejeitando-a.
api/Artistas	ArtistasController.Get() e o parâmetro <i>id</i> será nulo.

Para o exemplo acima, estamos utilizando o seguinte método:

```
public Artista Get(string id = "") { }
```

Por convenção, se o método for nomeado com o verbo do HTTP, automaticamente o ASP.NET Web API entrega a requisição naquela verbo para ele, sem qualquer trabalho adicional para que isso ocorra.

Durante a execução o processo de roteamento possui alguns estágios de processamento que são executados até que a ação seja de fato executada. O primeiro passo consiste no enquadramento da URI à uma rota configurada na tabela de roteamento. Se encontrada, as informações colocadas na URI são extraídas e adicionadas no dicionário de rotas, que poderão ser utilizadas por toda a requisição, sendo pelo ASP.NET Web API (*runtime*) ou pelo código customizado.

Depois de alguma rota encontrada, começa o processo para identificar qual é a ação, ou melhor, o método que o cliente está solicitando. Só que antes de disso, é necessário identificar a classe (*controller*) onde ele está. Como a rota padrão já vem configurada no projeto, os dois *placeholders* determinam indiretamente a classe (*controller*) e o método (ação), competindo ao ASP.NET realizar o *parser* dos fragmentos da URI, encontrar a classe e o método, e depois disso, abastecer os parâmetros (caso eles existam) e, finalmente, invocar o método, retornando o resultado (sucesso ou falha) ao cliente solicitante.

Apesar do compilador não indicar, toda a classe que será considerada uma API, é necessário que ela não seja abstrata e tenha o seu modificador de acesso definido como público, caso contrário, o ASP.NET Web API não será capaz de instanciá-la. E, finalmente, apesar de termos a necessidade do sufixo *Controller* nas classes, essa palavra não está presente na URI. Isso é apenas uma exigência do ASP.NET Web API, para identificar quais classes dentro do projeto correspondem à uma API.

Depois de encontrada a classe, é o momento de escolher o método a ser executado. Da mesma forma que a classe, o método a ser executado também é escolhido baseado na URI da requisição. Pela afinidade ao HTTP, a busca do método dentro da classe considera os verbos do HTTP (GET, POST, PUT, DELETE, etc.) durante a busca. O algoritmo de pesquisa do método dentro do *controller* segue os passos descritos abaixo:

1. Seleciona todos os métodos do *controller* que se enquadram com o verbo do HTTP da requisição. O método pode ser nomeado ou prefixado apenas com o nome do verbo.
2. Remove aquelas ações com o nome diferente do valor que está no atributo *action* do dicionário de rotas.
3. Tentar realizar o mapeamento dos parâmetros, onde os tipos simples são extraídos da URI, enquanto tipos complexos são extraídos do corpo da mensagem.
 - a. Para cada ação, extrai uma lista de parâmetros de tipos simples, e exceto os parâmetros opcionais, tenta encontrar o valor de cada um deles da URI.

- b. Dessa lista tenta encontrar o respectivo valor no dicionário de rotas ou na *querystring* da URI, independente de *case* ou de ordem dos parâmetros.
 - c. Seleciona aquela ação onde cada um dos parâmetros é encontrado na URI.
4. Ignora as ações que estão decoradas com o atributo *NonActionAttribute*.

Como notamos no algoritmo acima, ele não considera a busca de ações quando há tipos complexos. A ideia é tentar encontrar uma ação através da descrição estática da mensagem, ou seja, sem a necessidade de recorrer a qualquer mecanismo extra. Caso isso não seja suficiente, o ASP.NET Web API recorrerá a um *binder* ou a um formatador, baseado na descrição do parâmetro e que na maioria das vezes, é oriundo a partir do corpo da mensagem.

E o método *MapHttpRoute* ainda fornece um outro parâmetro que nos permite especificar *constraints*. Através delas podemos determinar um validador para um parâmetro específico, ou seja, se conseguimos antecipar qual o formato que a informação deve ser colocada, isso evitará que ela seja rejeitada logo nos primeiros estágios do processamento, retornando o erro 404 (*Not Found*).

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        config.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "api/{controller}/{periodo}",
            defaults: new { periodo = DateTime.Now.ToString("yyyy-MM") },
            constraints: new { periodo = @"[0-9]{4}\-[0-9]{2}" }
        );
    }
}
```

Da mesma forma que fazemos com os valores padrão para cada *placeholder*, criamos um novo dicionário onde a chave é o nome do parâmetro, e o valor é uma expressão regular que determina o formato em que o parâmetro deve ter. O fato de adicionar isso, o próprio ASP.NET Web API se encarrega de avaliar se o valor dos parâmetros se enquadram com a regra determinada na expressão regular. Isso nos dá a chance de separar as validações do método que executa a ação, ficando responsável apenas por lidar com a regra de negócio.

Depurando Rotas

Acima vimos como podemos configurar as rotas no projeto, mas não estamos limitados a incluir apenas uma opção de roteamento. Na medida em que as APIs vão sendo construídas, você está livre para criar rotas específicas para recepcionar as requisições.

A medida em que as rotas vão sendo adicionadas, podemos enfrentar alguma dificuldade no desenvolvimento, onde a requisição não chega ao destino esperado, não

encontrando o *controller* ou a ação, ou até mesmo, em casos mais extremos, se enquadrando em uma rota diferente daquela que estávamos imaginando.

Para facilitar a depuração disso, existe uma ferramenta chamada *ASP.NET Web API Route Debugger*, que pode ser adicionado à aplicação através do *Nuget*. Para adicioná-la ao projeto, basta executar o comando abaixo na *Package Manager Console*, conforme é mostrado abaixo:

```
PM> Install-Package WebApiRouteDebugger
```

Ao rodar este comando, alguns elementos são incluídos no projeto. Basicamente trata-se de uma nova área, que possui a interface gráfica para exibir o resultado da requisição com os parâmetros e resultados da interceptação que foi realizada pelo *debugger*.

Depois de instalar este depurador, basta rodarmos a aplicação e quando chegarmos no navegador, basta acrescentarmos na URL o sufixo */rd*. Isso fará com que uma tela seja exibida para que você digite o endereço completo para a API que deseja invocar.

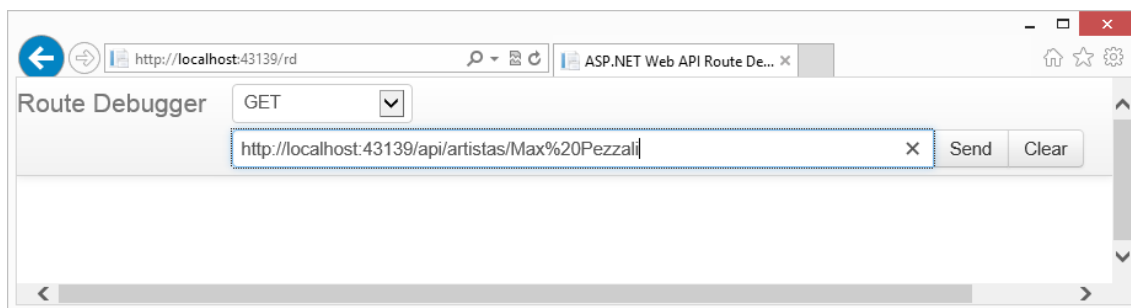


Figura 9 - Route Debugger.

A partir desta tela podemos ir testando as URIs que temos para validar se alguma rota dentro deste projeto consegue ser encontrada e, consequentemente, direcionada para o método que tratará a requisição. Quando pressionamos o botão *Send*, um resumo como o resultado da avaliação é exibido, e nele você conseguirá analisar de forma simples as informações que foram extraídas e facilmente identificará onde está o problema.

Entre as informações, ele exibe o dicionário de dados que foi extraído da URI. Ele elenca as informação (chave e valor), quais você poderá avaliar se o valor informado na URI está, corretamente, abastecendo o parâmetro desejado.

A próxima tabela aponta as rotas que estão configuradas no projeto, incluindo as informações extras, tais como: valores padrão, *constraints*, etc. Com as informações que compõem a configuração da rota sendo colocada aqui, fica fácil determinar o porque os parâmetros entraram ou não, porque eles os parâmetros assumiram o valor padrão ao invés de receber aquele que está na URI, e assim por diante.

Entre as duas últimas tabelas temos a primeira que lista todos os *controllers* que estão adicionados ao projeto, bem como o seu tipo, *namespace* e *assembly*. Já a segunda, e não menos importante, exibe todas as ações que foram localizadas, descrevendo-as complementos, incluindo o verbo HTTP que ela atende, o seu nome e o(s) parâmetro(s). Logo na sequência, ele nos aponta se ação (método) foi encontrado pelo nome (*By Action Name*) ou pelo verbo (*By HTTP Verb*). A imagem abaixo ilustra essa tela com todos os detalhes descritos aqui.

The screenshot shows the ASP.NET Web API Route Debugger interface. The address bar displays `http://localhost:43139/rd`. The route being debugged is `http://localhost:43139/api/artistas/Max%20Pezzali` with the HTTP verb `GET` selected. The status code is `200`.

Route Data

Key	Value
Controller	Artistas
Nome	Max Pezzali

Route selecting

Template	Defaults	Constraints	Data Tokens	Handler
Api/{Controller}/{Nome}				

Controller selecting

Name	Type	Assembly	Version	Culture	Token
Musicas	MusicStore.Controllers.MusicasController	MusicStore	1.0.0.0	Neutral	Null
Artistas	MusicStore.Controllers.ArtistasController	MusicStore	1.0.0.0	Neutral	Null
ODataMetadata	System.Web.Http.OData.ODataMetadataController	System.Web.Http.OData	4.0.0.0	Neutral	31bf3856ad364e35

Action selecting

#	Details			By Action Name		By HTTP Verb	Later Stage	
	Verb	Name	Param	Action	Verb	Verb	Parameter	NonAction
0	GET	Get	Nome:String			True	True	True

Figura 10 - Route Debugger exibindo as informações.

Para a grande maioria das tarefas discutidas neste capítulo, existem objetos responsáveis por executar cada uma delas, e que podemos customizar cada um deles para que seja possível mudarmos algumas regras ou interceptarmos os estágios do processamento e incluir um código para efetuar *logs*, mensurar performance, etc. A extensibilidade é abordada em detalhes em um capítulo mais adiante.

Rotas Declarativas

O modelo de roteamento que vimos até aqui é baseado em uma convenção que é definida através de *templates* e estão centralizadas arquivo *Global.asax*. Como falamos acima, durante a execução os *placeholders* são substituídos pelos parâmetros que são

extraídos, na maioria das vezes, da URI e, conseqüentemente, encaminhada para a ação dentro do *controller*. Por ser uma coleção, podemos predefinir várias rotas para atender todas as URIs que são criadas para as APIs que rodam dentro daquela aplicação.

Apesar da centralização ser um ponto positivo, começa a ficar complicado quando o número de rotas criadas começa a crescer. Haverá um trabalho em olhar para essa tabela e analisar qual das rotas a ação se enquadra. Nem sempre haverá uma rota genérica para atender todas as requisições, já que para expressar melhor a estrutura e hierarquia dos recursos, teremos que definir a rota para cada ação contida no *controller*.

É para este fim que a Microsoft incluiu no ASP.NET Web API uma funcionalidade chamada de *attribute routing*, ou roteamento declarativo. A ideia aqui é configurar o roteamento para uma ação em um local mais próximo dela, para além de permitir a configuração específica, facilitar a manutenção.

A implementação é bastante simples mas muito poderosa. Aqueles atributos que vimos acima que determinam através de que verbo do HTTP a ação estará acessível (*HttpGetAttribute*, *HttpPutAttribute*, *HttpPostAttribute*, *HttpDeleteAttribute*, etc.), passam a ter um *overload* no construtor que aceita uma *string* com a *template* da rota que será atendida por aquela ação (método).

```
public class ArtistasController : ApiController
{
    [HttpGet("artistas/{nomeDoArtista}/albuns")]
    public IEnumerable<Album> RecuperarAlbuns(string nomeDoArtista)
    {
        return new[]
        {
            new Album() { Titulo = "Max 20" },
            new Album() { Titulo = "Terraferma" }
        };
    }
}
```

No exemplo acima vemos que a rota foi configurada *in-place*, ou seja, no próprio local onde a ação foi criada. Da mesma forma que fazemos na definição da rota no arquivo *Global.asax*, aqui também podemos utilizar *placeholders*, para que eles sejam substituídos pelos parâmetros da própria ação que, novamente, são extraídos da requisição.

Mas aplicar os atributos com as rotas não é suficiente. Há a necessidade de instruir o ASP.NET a coletar as informações sobre o roteamento diretamente nas ações. Para isso há um método de extensão que é aplicado à classe *HttpConfiguration* chamado *MapHttpAttributeRoutes* no arquivo *Global.asax*:

```
config.MapHttpAttributeRoutes();
```

O resultado final para a chamada desta ação fica:

```
http://localhost:49170/artistas/MaxPezzali/albums
```

É claro que o controller pode conter diversas outras ações, onde cada uma delas retorna outras informações, neste caso, de um determinado artista. Suponhamos que também teremos uma ação que retorne as notícias do artista. Bastaria adicionar o método à classe, configurar o roteamento apenas alterando a última parte, trocando *albums* por *noticias*.

Há também uma opção para elevar o prefixo das rotas para o nível da classe (*controller*). Para isso, utilizaremos um atributo chamado *RoutePrefixAttribute* que nos permite definir um valor que será combinado com as rotas configuradas ação por ação para compor a rota final de acesso para cada uma delas.

```
[RoutePrefix("artistas")]
public class ArtistasController : ApiController
{
    [HttpGet("{nomeDoArtista}/albums")]
    public IEnumerable<Album> RecuperarAlbums(string nomeDoArtista)
    {
        return new[]
        {
            new Album() { Titulo = "Max 20" },
            new Album() { Titulo = "Terraferma" }
        };
    }

    [HttpGet("{nomeDoArtista}/noticias")]
    public IEnumerable<Noticia> RecuperarNoticias(string nomeDoArtista)
    {
        return new[]
        {
            new Noticia() { Titulo = "Novo Album em 2013" }
        };
    }
}
```

O fato de configurarmos as rotas nas ações, não quer dizer que perderemos certas funcionalidades. Mesmo utilizando esta técnica, teremos a possibilidade de configurar valores padrão para os parâmetros, bem como *constraints* para garantir que o cliente informe os dados seguindo uma regra já estabelecida.

Para ambos os casos, teremos uma sintaxe ligeiramente diferente comparada ao modelo tradicional (centralizado, de forma imperativa). O próprio *placeholder* irá contemplar os valores padrão e as *constraints*. Para casos onde teremos um parâmetro opcional e desejarmos definir um valor padrão, teremos duas formas para representar, onde a primeira delas basta utilizar o recurso do próprio C# para isso, e no *placeholder* sufixar o parâmetro com o caracter "?".

```
[HttpGet("artistas/noticias/{cultura?}")]
public IEnumerable<Noticia> RecuperarNoticias(string cultura = "pt-BR") { }

[HttpGet("artistas/noticias/{cultura=pt-BR}")]
public IEnumerable<Noticia> RecuperarNoticias(string cultura) { }
```

Já a segunda forma, definimos o valor padrão do parâmetro no próprio *placeholder*, que ponto de vista daquele que está acessando a cultura de dentro da ação, não muda em nada. A diferença é que no segundo caso, o valor será encaminhado para o *model binder*, onde você poderá fazer algo diferente se assim desejar.

No caso das *constraints* temos um conjunto de opções e funções que garantem que os parâmetros informados pelo cliente se enquadrem com a exigência da API. A ideia é definir na configuração da rota a *constraint* usando a seguinte sintaxe: *{parametro:constraint}*. Na primeira parte definiremos o nome do parâmetro que está declarado na assinatura do método (ação), e na opção *constraint* iremos escolher uma das várias opções predefinidas que temos a nossa disposição:

Constraint	Descrição	Exemplo
alpha, datetime, bool, guid	Garante com que o parâmetro seja alfanumérico (a-z, A-Z). Há também opções para tipos de dados especiais.	{cultura:alpha}
int, decimal, double, float, long	Opções para números de diferentes tipos.	{pagina:int}
length, maxlength, minlength	Garante que uma determinada string tenha um tamanho específico.	{razaoSocial:maxlength(100)}
max, min, range	Opções para determinar intervalos.	{idade:range(18, 65)}
regex	Assegura que o parâmetro se encaixe em uma expressão regular.	{email:(\b[A-Z0-9._%~]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b)}

Nada impede definirmos mais que uma *constraint* para um mesmo parâmetro. Para isso basta utilizarmos uma vírgula para separar as *constraints* que desejamos aplicar a este parâmetro.

```
[HttpGet("artistas/detalhes/{artistaId:int}")]  
public Artista RecuperarDetalhes(int artistaId) { }
```


Nomenclatura das Rotas

Tanto para as rotas que definimos em nível centralizado quanto estar em nível de método (ação), podemos opcionalmente definir um nome que a identifique. A finalidade deste nome é facilitar a geração dos *links* para recursos específicos que já existem ou que são criados sob demanda, evitando assim a definição em *hard-code* de *links*, o que dificulta a manutenção futura.

Como facilitador para a geração destes links, temos uma classe chamada *UrlHelper*, que utiliza a tabela de roteamento para compor as URLs necessárias para a API. A classe *ApiController* possui uma propriedade chamada *Url*, que retorna a instância da classe *UrlHelper* já configurada para a utilização.

```
[RoutePrefix("artistas")]
public class ArtistasController : ApiController
{
    private static IList<Artista> artistas = new List<Artista>();
    private static int Ids = 0;

    [HttpPost("adicionar")]
    public HttpResponseMessage AdicionarArtista(HttpRequestMessage request)
    {
        var novoArtista = request.Content.ReadAsAsync<Artista>().Result;
        novoArtista.Id = ++Ids;
        artistas.Add(novoArtista);

        var resposta = new HttpResponseMessage(HttpStatusCode.Created);
        resposta.Headers.Location =
            new Uri(Url.Link("Detalhamento", new { artistaId = novoArtista.Id }));

        return resposta;
    }

    [HttpGet("detalhes/{artistaId:int}", RouteName = "Detalhamento")]
    public Artista BuscarDetalhes(int artistaId)
    {
        return artistas.SingleOrDefault(a => a.Id == artistaId);
    }
}
```

Ao incluir o novo artista na coleção, o método cria e retorna um objeto do tipo *HttpResponseMessage*, definindo o *header Location* com o endereço para detalhar a entidade que foi recentemente criada. O método *Link* da classe *UrlHelper* já combina o endereço geral (*host*) da API com o *path* até a ação que é especificada, retornando assim o endereço absoluto para um determinado local. O resultado da resposta é mostrado abaixo:

```
HTTP/1.1 201 Created
Location: http://localhost:49170/artistas/detalhes/1
Date: Mon, 01 Jul 2013 23:32:58 GMT
Content-Length: 0
```

Hosting

Até o momento vimos algumas características do HTTP, a estrutura que uma API deve ter quando utilizamos o ASP.NET Web API, a configuração para rotear as requisições para um tratador (método), etc. Mas para dar vida a tudo isso, precisamos de algum elemento que faça com que toda essa estrutura seja ativada e, consequentemente, passe a tratar as requisições.

O responsável por isso é o *hosting*. Dado todas as configurações necessárias a ele, ele será responsável por construir toda a infraestrutura para receber, manipular, processar e devolver a resposta ao cliente.

Nós temos algumas opções para realizar a hospedagem de APIs dentro do .NET, uma cada uma delas possuem suas vantagens e desvantagens. As opções são: *self-hosting*, *web-hosting* e *cloud-hosting*.

A primeira opção, *self-hosting*, consiste em hospedar as APIs em um processo que não seja o IIS como habitualmente é. Isso permitirá utilizar algum outro processo do Windows, como por exemplo, uma aplicação *Console*, um *Windows Service*, ou até mesmo um projeto de testes, que poderemos utilizar para facilitar a criação de testes unitários.

O primeiro passo para ter acesso a todos os recursos para hospedar APIs escritas em ASP.NET Web API em um processo próprio, é instalar o pacote disponível via *Nuget* chamado *Microsoft ASP.NET Web API Self Host*.

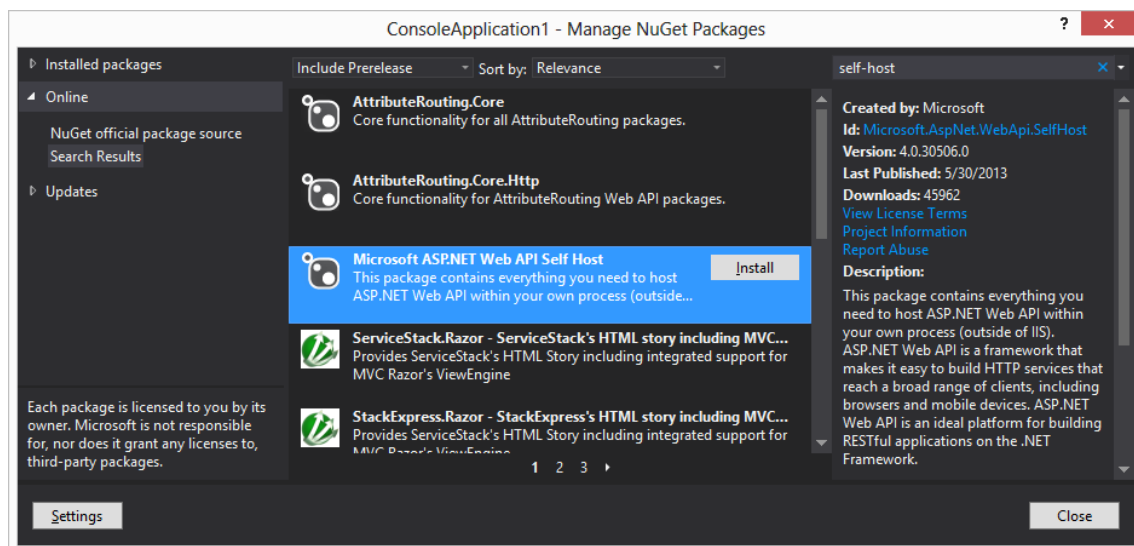


Figura 11 - Pacote para *self-hosting*.

Quando estamos falando em *self-hosting*, somos obrigados a criar toda a estrutura necessária para hospedar a API, e além disso, precisamos gerenciar a “vida” destes objetos, pois eles não são ativados sob demanda, obrigando eles estarem disponíveis no momento em que a requisição chegar, caso contrário, ela será rejeitada.

Ao referenciar o pacote que vimos acima, o principal *assembly* que temos é o *System.Web.Http.SelfHost.dll*. A principal classe a ser utilizada é a *HttpSelfHostServer*, que em seu construtor recebe a instância da classe *HttpSelfHostConfiguration*, que tem a função de receber todas as configurações necessárias, para que em tempo de execução, forneça todos os recursos necessários para a API ser executada.

Antes de qualquer coisa, precisamos criar a classe que representará a API, e isso não muda em nada, independente de como hospedamos a API. Sendo assim, temos que criar uma classe, sufixada com a palavra *Controller* e herdar da classe abstrata *ApiController*, e a implementação dos métodos nada muda em relação ao que já conhecemos nos capítulos anteriores.

A partir das APIs criadas, é o momento de fazermos as configurações e, consequentemente, ceder essas informações para que a infraestrutura consiga fazer o papel dela. Como sabemos, as rotas tem um papel extremamente importante para busca pela API, pelo método e mapeamento dos respectivos parâmetros. Todas as configurações, incluindo a tabela de roteamento, está acessível a partir da classe *HttpSelfHostConfiguration*.

Depois das configurações realizadas, passamos a instância desta classe para o construtor da classe *HttpSelfHostServer*. Internamente esta classe utiliza alguns recursos do *framework* do WCF para extrair as mensagens do HTTP, materializando em classes para que o ASP.NET Web API possa começar a manipular.

```
static void Main(string[] args)
{
    var config = new HttpSelfHostConfiguration("http://localhost:9393");
    config.Routes.MapHttpRoute("Default", "api/{controller}");

    using (var server = new HttpSelfHostServer(config))
    {
        server.OpenAsync().Wait();
        Console.ReadLine();
    }
}
```

O método *OpenAsync*, como já suspeita-se, ele tem o sufixo *Async* por se tratar de um método que é executado de forma assíncrona, retornando um objeto do tipo *Task*, que representa a inicialização e abertura do serviço, que quando concluída, passará a receber as requisições. O método *Wait*, exposto através da classe *Task*, bloqueia a execução até que o host seja aberto.

Um dos *overloads* do construtor da classe *HttpSelfHostServer* recebe como parâmetro uma instância da classe *HttpMessageHandler*, qual podemos utilizar para interceptar a requisição e a resposta, injetando algum código customizado, como por exemplo, efetuar o *log* das mensagens, inspecionar os *headers*, etc. No capítulo sobre extensibilidade será abordado as opções disponíveis que temos para isso.

Como vimos, a opção do *self-hosting* dá a chance de hospedar a API em um projeto que não seja ASP.NET e, consequentemente, não temos a necessidade termos

obrigatoriamente o IIS (*Internet Information Services*), tendo um controle mais refinado sobre a criação e gerenciamento dos componentes da aplicação.

A outra opção que temos para hospedar APIs Web é justamente utilizar o IIS (*web-hosting*), que por sua vez, nos dá uma infinidade de recursos para gerenciamento das APIs, reciclagem de processo, controle de acesso, ativação, segurança, etc. Esse modelo reutiliza a infraestrutura criada para o ASP.NET e também para o ASP.NET MVC para executar os serviços que serão hospedados ali.

Ao contrário do que vimos na outra opção de hospedagem, quando optamos pelo ASP.NET, é que já temos uma *template* de projeto disponível chamada Web API (mais detalhes acima), que já traz algumas coisas já pré-configuradas, e os objetos que utilizamos para realizar as configurações e customizações, já estão mais acessíveis. Caso você já esteja utilizando outra *template* de projeto, podemos recorrer à um outro pacote (via *Nuget*) chamado *Microsoft ASPNET Web API Web Host*.

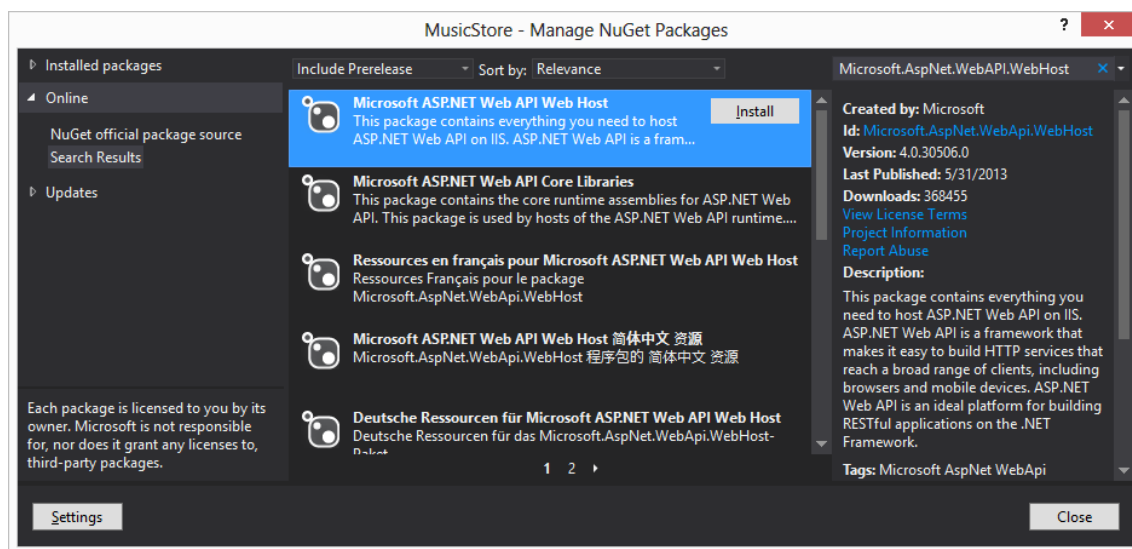


Figura 12 - Pacote para web-hosting.

A configuração já está acessível através do arquivo *Global.asax*, que nos permite interceptar alguns eventos em nível de aplicação, e um deles é o *Application_Start*, onde podemos realizar todas as configurações referente ao roteamento para assim possibilitar os clientes chegarem até eles.

Essa configuração inicial já foi abordada nos capítulos anteriores, mas é claro que o objeto de configuração (tanto no *self-hosting* quanto no *web-hosting*) são utilizados apenas para armazenar a tabela de roteamento. Ele vai muito além disso, e vamos explorar mais algumas funcionalidades que ele expõe nos próximos capítulos.

A terceira e última opção abordada aqui é *cloud-hosting*, onde estaremos utilizando o *Windows Azure* para publicar e rodar as nossas APIs. Hospedar o serviço na nuvem pode facilitar o escalonamento do mesmo sem que haja a necessidade realizar mudanças na infraestrutura, afinal, o próprio mecanismo de gerenciamento e execução possui

algoritmos que detectam o alto volume de acesso e, conseqüentemente, sugerem e/ou ativam novas instâncias (máquinas virtuais) para conseguir atender toda a demanda.

É importante dizer que o hosting nada tem a ver com o meio em que a API ou qualquer tipo de aplicação é distribuída (*deployment*). A distribuição determina como ela é empacotada, disponibilizada e instalada no computador onde ela será executada. Para cada tipo de *hosting* que vimos até aqui, há um mecanismo de distribuição diferente, combinando o Visual Studio para a criação do pacote e tecnologias que existem nos computadores e servidores para a instalação do mesmo.

Consumo

Depois da API construída, hospedada e rodando no servidor, chega o momento de consumirmos isso do lado do cliente. Como o foco é o HTTP, que é bastante popular, a grande maioria das tecnologias cliente dão suporte ao consumo deste tipo de serviços. É aqui que serviços REST diferem bastante do SOAP. O SOAP define uma estrutura para comunicação, que sem um ferramental que ajude, o consumo se torna bastante complexo, uma vez que temos que manualmente formatar as mensagens de envio e interpretar as mensagens de retorno.

Mesmo com poucos facilitadores para consumir serviços por parte de uma tecnologia, o trabalho com serviços HTTP não é muito complexo. Com um objeto que gerencie a conexão, a configuração da requisição e leitura da resposta acabo sendo uma tarefa bem mais simples do que quando comparado ao SOAP.

Podemos ter os mais variados tipos de clientes, podendo ser aplicações *desktops*, de linha de comando, aplicações para internet, dispositivos móveis ou até mesmo outros serviços. Quando o consumo é realizado por aplicações Web, é muito comum utilizarmos código script para isso, onde recorreremos à *Javascript* para solicitar o envio, recuperar a resposta e apresentar o resultado na tela. Neste ambiente temos uma biblioteca chamada *jQuery*, que fornece uma infinidade de facilitadores para o consumo de serviços REST.

Além disso, podemos consumir serviços REST em aplicações que não são Web. Da mesma forma que a Microsoft criou tudo o que vimos até o momento para a construção de serviços, ela também se preocupou em criar recursos para tornar o consumo de serviços REST em aplicações .NET muito simples.

Para consumirmos serviços REST em aplicações .NET, o primeiro passo é referenciar o *assembly System.Net.Http.dll*. Dentro deste *assembly* temos o *namespace System.Net.Http*, que possui tipos para consumir serviços baseados exclusivamente no protocolo HTTP. Os componentes que veremos aqui podem também ser utilizados pelo lado do serviço (como é o caso das classes *HttpRequestMessage* e *HttpResponseMessage*), fornecendo uma certa simetria entre o código do lado do cliente e do lado do serviço, obviamente quando ambos forem .NET.

Só que este *assembly* apenas não é o suficiente. Ao invés de descobrirmos quais os *assemblies* e recursos que precisamos, podemos recorrer ao *Nuget* para que ele registre tudo o que precisamos nas aplicações cliente para consumir serviços HTTP utilizando o *ASP.NET Web API*. Para isso podemos digitar a chave “*Microsoft.AspNet.WebApi.Client*”, e já teremos a *Microsoft ASP.NET Web API Client Libraries* listada nos resultados, e clicando no botão *Install*, elas são configuradas no projeto em questão.

É importante dizer que também temos a versão portátil destas bibliotecas. Com isso, poderemos reutilizar o consumo de APIs REST (HTTP) através de diferentes plataformas, sem a necessidade de reconstruir a cada novo projeto uma ponte de comunicação com

estes tipos de serviços, e ainda, sem a necessidade de recorrer de classes de mais baixo nível para atingir o mesmo objetivo.

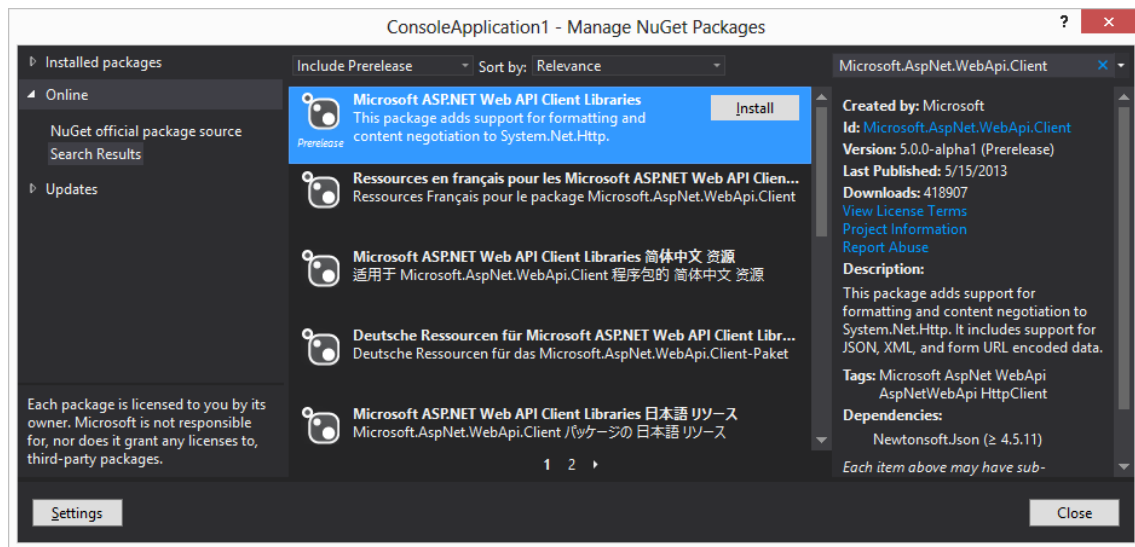


Figura 13 - Pacote com recursos para consumo de APIs.

Do lado do cliente a principal classe é a *HttpClient*. Ela é responsável por gerenciar toda a comunicação com um determinado serviço. Ela atua como sendo uma espécie de sessão, onde você pode realizar diversas configurações e que serão aplicadas para todas as requisições que partirem da instância desta classe.

Essa é a principal classe que é utilizada quando estamos falando de um cliente .NET, fornecendo métodos nomeados com os principais verbos do HTTP, e que trabalham com as classes *HttpRequestMessage* e *HttpResponseMessage*, o que dará ao cliente o controle total da mensagem que é enviada e da resposta que é retornada. Internamente a classe *HttpClient* trabalha com classes que são velhas conhecidas de quem já trabalha com .NET: *HttpWebRequest* e *HttpWebResponse*, mas como todo e qualquer facilitador, a classe *HttpClient* encapsula toda a complexidade destas classes, expondo métodos mais simples para consumo dos serviços.

Apesar da classe *HttpClient* ser comparada com um *proxy* de um serviço WCF, você não está obrigado a somente invocar APIs que estão debaixo de um mesmo servidor, pois isso deve ser definido o endereço (URL) onde está localizado o serviço. Caso quisermos criar um cliente específico para um determinado serviço, podemos herdar da classe *HttpClient*, e fornecer métodos específicos para acessar os recursos que são manipulados pela API. Um exemplo disso seria ter uma classe chamada *FacebookClient*, que internamente recorrerá aos métodos da classe *HttpClient* encapsulando ainda mais a – pouca – complexidade que temos, tornando o consumo um pouco amigável.

Antes de analisarmos algum código, é importante dizer que a classe *HttpClient* expõe os métodos para consumo dos serviços de forma assíncrona. Como o consumo depende de recursos de I/O, os métodos foram desenhados para que ao consumir qualquer recurso remoto, essa tarefa seja executada por uma outra thread. E, como ele já faz uso do novo

mecanismo de programação assíncrona do .NET (*Tasks*), ficar bastante simples gerenciar e coordenar requisições que são realizadas.

```
class Program
{
    private const string Endereco = "http://localhost:43139/api/artistas/MaxPezzali";

    static void Main(string[] args)
    {
        Executar();
        Console.ReadLine();
    }

    private async static void Executar()
    {
        using (var client = new HttpClient())
        {
            using (var request = await client.GetAsync(Endereco))
            {
                request.EnsureSuccessStatusCode();

                await request.Content.ReadAsAsync<JObject>().ContinueWith(t =>
                {
                    Console.WriteLine(t.Result["Nome"]);
                });
            }
        }
    }
}
```

Dentro do método *Executar* criamos a instância da classe *HttpClient*, que através do método *GetAsync*, realiza uma requisição através do verbo GET para o serviço de artistas, que estamos utilizando para os exemplos. Temos que fazer uso das palavras *async* e *await* para que o C# possa criar o mecanismo necessário para que o serviço seja invocada de forma assíncrona.

O método *EnsureSuccessStatusCode* assegura que o serviço foi invocado com sucesso, disparando uma exceção se o código do *status* de retorno caracterizar uma falha. Finalmente chamando o método, também de forma assíncrona, *ReadAsAsync<T>*. Este método realiza a leitura do corpo da mensagem de retorno, tentando converter a mesma no tipo genérico T. No exemplo acima estamos utilizando a classe *JObject*, que é fornecida pelo *framework Json.NET*, que é o mais popular meio para manipular a serialização de objetos em *Json* dentro do .NET. Para instalarmos, basta recorrer ao comando abaixo no *Nuget*:

```
PM> Install-Package Newtonsoft.Json
```

Finalmente, quando finalizamos a leitura do corpo da mensagem, entregamos o objeto *JObject* ao *ContinueWith* para que ele exiba na tela. O objeto *JObject* implementa a *interface IDictionary*, que dado uma string com o nome da propriedade, ele retorna o seu valor.

O problema da técnica acima é que ela é baseada em *strings* e *objects*, sem qualquer tipificação, o que pode gerar erros, e que na maioria das vezes, acontecem somente

durante a execução. Para facilitar isso, o Visual Studio .NET 2012, fornece dois recursos chamados “*Paste JSON as Classes*” e “*Paste XML as Classes*”, que dado um conteúdo Json ou Xml que está na área de transferência, ele consegue construir as classes no projeto, configurando suas propriedades e seus respectivos tipos.

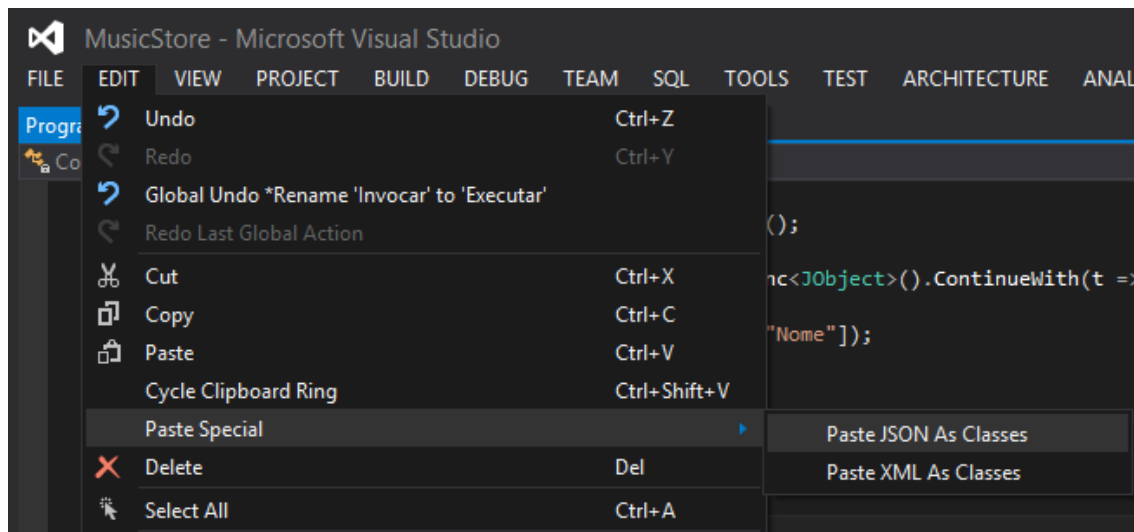


Figura 14 -Opções do Visual Studio.

```
public class Rootobject
{
    public int Id { get; set; }
    public string Nome { get; set; }
    public int AnoDeNascimento { get; set; }
    public int QuantidadeDeAlbuns { get; set; }
}
```

Desta forma, podemos passar a utilizar as classes específicas ao invés de ficar lidando com índices e nome das propriedades, que acaba sendo sujeito a erros. O código abaixo exhibe um trecho do consumo do serviço recorrendo a classe que acabamos de construir:

```
await request.Content.ReadAsAsync<Artista>().ContinueWith(t =>
{
    Console.WriteLine(t.Result.Nome);
});
```

Como pudemos perceber, o método *GetAsync* recebe apenas o endereço para onde a requisição deve ser realizada. Mas e se quisermos customizar a requisição? Por exemplo, incluir *headers* específicos para *caching* e segurança, especificar o formato de resultado que desejamos, etc. Devido a essas necessidades é que devemos recorrer a construção e configuração da classe *HttpRequestMessage*. É através dela que toda a configuração é realizada antes da mesma partir para o serviço, e depois que ela estiver devidamente configurada, recorreremos ao método *SendAsync*, que como parâmetro recebe a classe *HttpRequestMessage* e retorna a classe *HttpResponseMessage*. Apesar de *Send* não refletir um verbo do HTTP, é dentro da mensagem que vamos especificar qual verbo será utilizado para realizar a requisição.

Utilizando o mesmo exemplo acima, vamos customizar a requisição para que ela retorne o artista em formato Xml:

```
using (var client = new HttpClient())
{
    using (var request = new HttpRequestMessage(HttpMethod.Get, Endereco))
    {
        request.Headers.Add("Accept", "application/xml");

        using (var response = await client.SendAsync(request))
            Console.WriteLine(response.Content.ReadAsStringAsync().Result);
    }
}
```

A classe *HttpMethod* traz algumas propriedades estáticas com os principais verbos criados, prontos para serem utilizados. Logo na sequência estamos adicionando um novo *header* chamado *Accept*, que determina o formato que queremos que o retorno seja devolvido. Da mesma forma que podemos utilizar a classe *HttpRequestMessage* para customizar a requisição, podemos inspecionar a classe *HttpResponseMessage* para explorarmos tudo aquilo que o serviço nos devolveu e, conseqüentemente, tomar decisões, realizar *logs*, novas requisições, etc.

Tanto a classe *HttpRequestMessage* quanto a *HttpResponseMessage* implementam a *interface IDisposable*, e por isso, podem ser envolvidas em blocos *using*. Ambas as classes fornecem uma propriedade chamada *Content*, onde temos o corpo das mensagens, e como ele pode ser um *stream*, a implementação dessa *interface* auxilia no descarte imediato quando elas não estão mais sendo utilizadas.

Quando vamos utilizar um verbo diferente do GET, como é o caso do POST, então precisamos definir o corpo da mensagem de envio, onde temos várias opções para os mais diversos conteúdos que enviamos para o servidor. O diagrama abaixo dá uma dimensão disso.

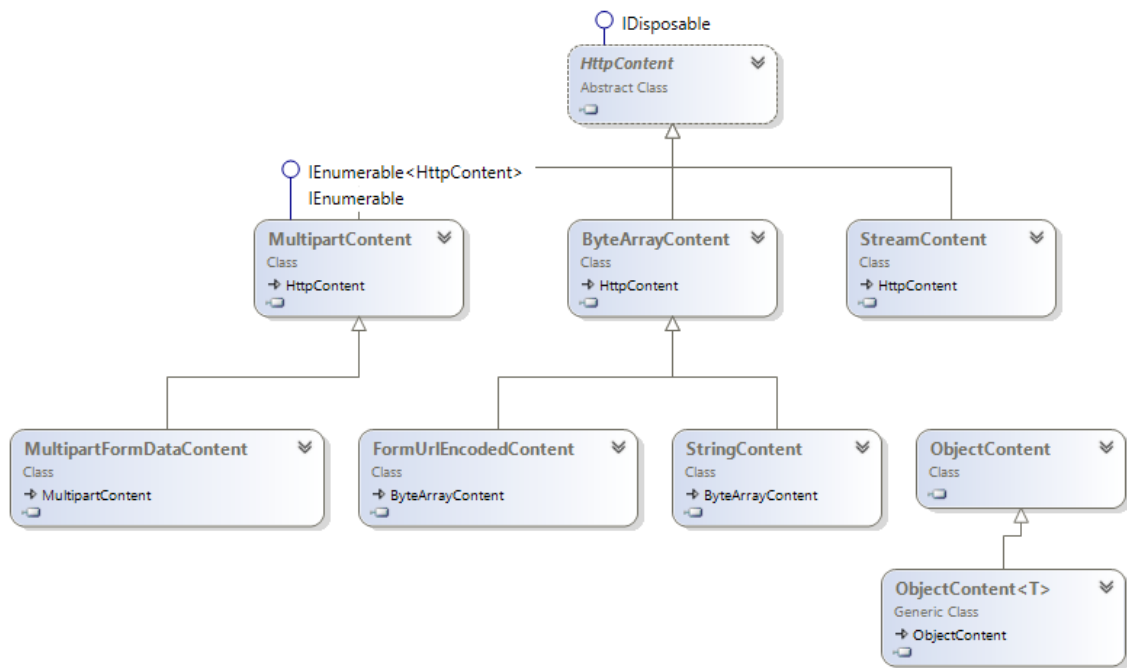


Figura 15 - Hierarquia das classes que representam o conteúdo das mensagens.

Cada uma dessas classes lida exclusivamente com um tipo de conteúdo. Temos como customizar o envio determinando que o corpo deve ser uma espécie de formulário, string, *stream* e objetos.

Independente se estamos falando de requisição ou resposta, a classe *ObjectContent<T>* tem a função de materializar objetos, baseando-se no formato que ele foi serializado. Existem classes que representam esses formatos, quais falaremos delas mais adiante, mas quando estamos consumindo o serviço através desta biblioteca, não precisamos lidar diretamente com elas, pois existem alguns facilitadores que abstraem essa complexidade, e os formatadores apenas farão o trabalho (bidirecional) sobre os *bytes*, o que de fato é o que trafega entre as partes.

```

private const string Endereco = "http://localhost:43139/api/artistas/Adicionar";

private async static void Executar()
{
    using (var client = new HttpClient())
    {
        await client.PostAsync(Endereco, new ObjectContent<Artista>(new Artista()
        {
            Nome = "PaoloMeneguzzi"
        }, new JsonMediaTypeFormatter())).ContinueWith(t =>
        {
            Console.WriteLine(t.Result.StatusCode);
        });
    }
}

```

No exemplo que temos acima, estamos recorrendo ao método *PostAsync*, para que o objeto (classe *Artista*) seja postado. Neste caso, temos que explicitamente mencionar que queremos que o conteúdo seja serializado através de Json. Felizmente, graças à

algumas extensões que já temos no *ASP.NET Web API*, podemos tornar esse código menos verboso:

```
private const string Endereco = "http://localhost:43139/api/artistas/Adicionar";

private async static void Executar()
{
    using (var client = new HttpClient())
    {
        await client.PostAsJsonAsync(Endereco,
            new Artista() { Nome = "PaoloMeneguzzi" }).ContinueWith(t =>
        {
            Console.WriteLine(t.Result.StatusCode);
        });
    }
}
```

Da mesma forma que temos do lado do serviço, a classe *HttpClient* também possui um *overload* do construtor que recebe a instância de uma classe do tipo *HttpMessageHandler*, qual podemos utilizar para interceptar a requisição e a resposta, injetando algum código customizado, como por exemplo, efetuar o *log* das mensagens, inspecionar os *headers*, etc. No capítulo sobre extensibilidade será abordado as opções disponíveis que temos para isso.

Formatadores

Um dos maiores benefícios ao se utilizar uma biblioteca ou um *framework* é a facilidade que ele nos dá para tornar a construção de algo mais simplificada, abstraindo alguns pontos complexos, permitindo com que o utilizador foque diretamente (e na maioria das vezes) na resolução de problemas voltados ao negócio, sem gastar muito tempo com questões inerentes à infraestrutura e/ou similares.

A finalidade do ASP.NET Web API é justamente facilitar a construção de APIs para expor via HTTP. Apesar de uma das principais características de uma API é abraçar o HTTP, a abstração acaba sendo útil em alguns casos, mas em outros não. A Microsoft se preocupou com isso, abstraindo alguns aspectos para tornar a construção e o consumo destas APIs mais fáceis, sem perder o poder de customização e acesso aos recursos expostos pelo protocolo HTTP.

Como vimos nos capítulos anteriores, possuímos os objetos *HttpRequestMessage* e *HttpResponseMessage*, quais podemos definir na assinatura das ações do *controller* e, consequentemente, ter o acesso total à todos os recursos do HTTP. Apesar de que em alguns casos isso possa ser útil, na sua grande maioria, temos que lidar com alguns pontos que a abstração poderia ajudar a nos manter mais focados no negócio do que na infraestrutura.

Um grande exemplo disso tudo é o conteúdo (*payload*) da mensagem. Em geral, o corpo da mensagem representa um objeto que deve ser materializado e entregue ao método que tratará a requisição. A postagem que o cliente faz ao serviço pode ser realizada em diferentes formatos, como por exemplo: Json, Xml, Csv, formulário, etc.

Como já foi dito anteriormente, o uso de um *framework* tem a finalidade de abstrair certos pontos para tornar a programação mais simples. Aqui temos um grande exemplo disso. O ASP.NET Web API fornece alguns recursos intrínsecos que torna a serialização e deserialização transparente ao ponto de vista do serviço/método. Esses recursos são o *Model Binding* e os Formatadores.

As aplicações geralmente trabalham com objetos que descrevem suas características, onde estes objetos são manipulados o tempo todo, já que na grande maioria dos casos, ela acaba também sendo persistido no banco de dados, apresentado na tela, etc. Como esses objetos são parte do core da aplicação, é muito comum criarmos formulários que apresente a instância no mesmo na tela (HTML), para que o usuário seja capaz de editá-lo.

Ao submeter o formulário para o servidor, todas as informações (*querystrings*, *body*, URI, etc.) chegam através de um dicionário, onde cada valor está associado à uma chave. Ao invés de manualmente construirmos a instância da classe baseada no corpo da requisição, o ASP.NET MVC já fez esse árduo trabalho para nós, e o responsável por isso são os model binders. Baseando-se na action para qual estamos postando a requisição, ele captura o tipo do objeto que precisa ser criado, mapeando o dicionário para cada uma de suas propriedades.

Olhando mais de perto, os *model binders* são os responsáveis por construir os objetos, baseando-se em um dicionário que contém as informações que foram extraídas da requisição através de *Value Providers*. O ASP.NET Web API possui algumas implementações embutidas, mas nada impede de criarmos alguma customização tanto para o *value provider* (se quisermos customizar como extrair as informações da requisição), bem com o *model binder* (se quisermos customizar como construir a instância do objeto).

Ao efetuar uma requisição para algum recurso sobre o protocolo HTTP, o servidor identifica o mesmo, faz o processamento, gera o resultado e, finalmente, devolve o resultado para o cliente que fez a solicitação. Por mais que isso não fica explícito, o conteúdo que trafega do cliente para o servidor (requisição) e do servidor para o cliente (resposta), sempre possui um formato específico.

Em grande parte de todos os recursos fornecidos através do protocolo HTTP, uma das necessidades é justamente definir o formato deste conteúdo, que por sua vez, direciona a interpretação pelo navegador, por uma outra aplicação ou até mesmo de uma biblioteca, permitindo efetuar o parser do resultado e, conseqüentemente, materializar o mesmo em algo "palpável"/visível.

Os formatos são representados por uma simples *string*, onde você tem uma primeira parte para descrever qual o tipo de conteúdo, e depois o seu formato. Por exemplo, ao invocar uma página onde o retorno é um conteúdo HTML, o formato será definido como *text/html*; ao solicitar uma imagem, o seu formato será definido como *image/jpeg*. Uma lista contendo todos os formatos de conteúdos disponíveis na *internet*, é gerenciada e mantida por entidade chamada IANA.

Como o ASP.NET Web API tem uma forte afinidade com as características do HTTP, ele permite receber ou gerar conteúdos em formatos popularmente conhecidos pelo mercado, e vamos perceber que os serviços que criamos utilizando essa API nada sabem sobre o formato em que ele chegou ou o formato em que ele será devolvido para o cliente. O ASP.NET Web API unifica o processo de serialização e deserialização dos modelos através de formatadores, que baseado em um *media type* específico, executa o trabalho para materializar a requisição em um objeto de negócio (modelo).

Apesar dos principais formatadores já estarem vinculados à execução, precisamos analisar a estrutura da classe que representa um *media type* para realizar futuras customizações. Para isso, a Microsoft criou uma classe abstrata chamada de *MediaTypeFormatter*, e já existem algumas implementações definidas dentro da API, como por exemplo, as classes *XmlMediaTypeFormatter* e *JsonMediaTypeFormatter*.

Uma pergunta pertinente que aparece é como o ASP.NET Web API escolhe qual dos formatadores utilizar. A escolha se baseia no formato solicitado pelo cliente. O formato pode ser incluído como um item na requisição, através do *header Accept* ou do *Content-Type*. O ASP.NET Web API escolhe o formatador de acordo com o valor que ele encontra em um desses *headers*, e caso o formato definido não for encontrado, o padrão é sempre devolver o conteúdo em formato Json.

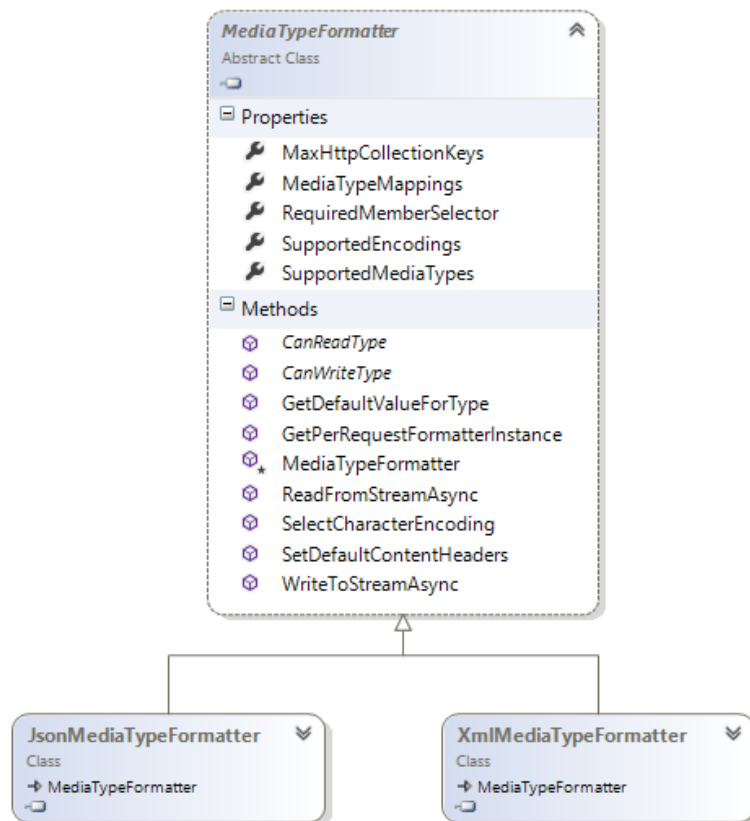


Figura 16 - Hierarquia das classes de formadores.

Ao contrário do que fazemos no cliente, o serviço não trabalha diretamente com estes formadores, pois isso fica sob responsabilidade de objetos internos, que durante a execução fazem todo o trabalho para encontrar para ler ou escrever o conteúdo no formato requisitado. O responsável por fazer esta análise e encontrar o formador adequado é a classe *DefaultContentNegotiator* através do método *Negotiate* (fornecido pela interface *IContentNegotiator*).

Para percebermos a mágica que o ASP.NET Web API faz, vamos utilizar o *Fiddler* para criar e monitorar as requisições. No primeiro exemplo, vamos invocar um método que dado o nome do artista, ele retorna um objeto complexo contendo as características deste artista. Abaixo temos a primeira requisição sendo executada e seu respectivo retorno.

[Requisição]

GET http://localhost:43139/api/artistas?nome=MaxPezzali HTTP/1.1
 User-Agent: Fiddler
 Host: localhost:43139

[Resposta]

HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
 Content-Length: 75

```
{"Id":5,"Nome":"MaxPezzali","AnoDeNascimento":1967,"QuantidadeDeAlbuns":20}
```

Note que não mencionamos nenhum *header* extra na requisição, e o objeto artista foi devolvido serializado em Json. Nós podemos requisitar que o conteúdo seja devolvido em um determinado padrão. Para isso, recorreremos ao atributo *Accept* para informar ao serviço que o aceite o retorno em um determinado formato. Os *logs* abaixo exibem a requisição e a resposta para o formato Xml:

[Requisição]

GET http://localhost:43139/api/artistas?nome=MaxPezzali HTTP/1.1

Host: localhost:43139

Accept: application/xml

[Resposta]

HTTP/1.1 200 OK

Content-Type: application/xml; charset=utf-8

Content-Length: 252

```
<Artista xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.datacontract.org/2004/07/MusicStore.Models"><AnoDeNasci
mento>1967</AnoDeNascimento><Id>0</Id><Nome>MaxPezzali</Nome><Quantidad
eDeAlbuns>20</QuantidadeDeAlbuns></Artista>
```

Percebemos que a o resultado é bem maior quando o resultado é devolvido em Xml, pelo fato da sua estrutura ser bem mais verbosa que o Json. Além disso, ainda há quem defenda que se o serviço não suportar o formato requisitado pelo cliente, ele não deveria acatar a requisição. Para termos este comportamento no ASP.NET Web API, nós temos que criar código customizado para conseguir rejeitar a requisição.

Como vimos acima, por padrão, o ASP.NET Web API faz a busca do formato na coleção de *headers* da requisição, mas isso pode ser customizado. Podemos instruir o ASP.NET Web API localizar o formato na *querystring*, e utilizaremos o objeto de configuração para realizar esse ajuste.

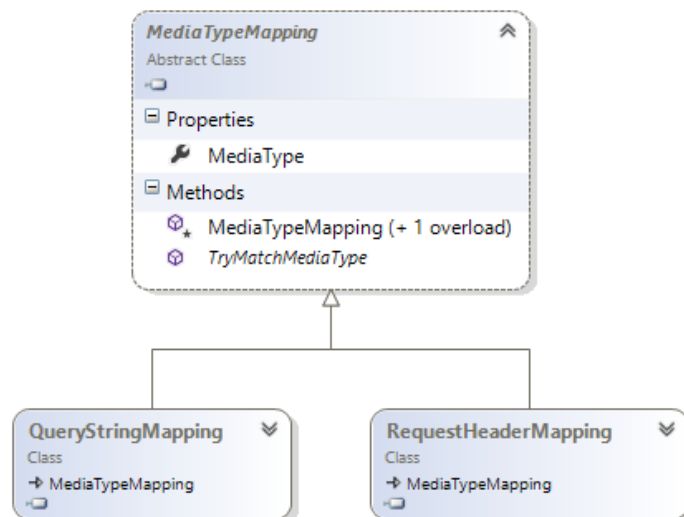


Figura 17 - Hierarquia das classes referente ao mapeamento de media-types.

Existe uma classe chamada *MediaTypeMapping*, e como o nome sugere, ele traz a infraestrutura para extrair o media type de algum elemento da requisição. Como vemos no diagrama acima, temos as classes *QueryStringMapping* e *RequestHeaderMapping*, que buscam o *media type* dentro da coleção de *querystrings* e *headers*, respectivamente.

Note que no código abaixo inserimos o mapeador de *querystring*, indicando qual a chave onde será informado o formato, e se ele for encontrado, o media type será utilizado pelos componentes internos para gerar o resultado no formato que o cliente está solicitando em um local diferente do padrão.

```

public class WebApiApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        WebApiConfig.Register(GlobalConfiguration.Configuration);

        this.ConfigMediaTypeMappings(GlobalConfiguration.Configuration);
    }

    private void ConfigMediaTypeMappings(HttpConfiguration config)
    {
        config.Formatters.JsonFormatter.MediaTypeMappings.Add(
            new QueryStringMapping("formato", "json", "application/json"));

        config.Formatters.JsonFormatter.MediaTypeMappings.Add(
            new QueryStringMapping("formato", "xml", "application/xml"));
    }
}
  
```

Com o código acima em funcionamento, a requisição pode ser realizada da seguinte forma:

`http://localhost:43139/api/artistas?nome=MaxPezzali&formato=json`
`http://localhost:43139/api/artistas?nome=MaxPezzali&formato=xml`

Customização

Apesar do Xml e Json serem os principais formatos que temos atualmente, pode haver situações onde desejamos criar um formato próprio, para que esse possa gerar e/ou receber o conteúdo em um formato específico. Como já percebemos acima, se desejarmos fazer isso, temos que recorrer à implementação da classe abstrata *MediaTypeFormatter*, customizando basicamente dois métodos principais *OnReadFromStream* e *OnWriteToStream*. Vamos analisar cada um desses métodos abaixo:

- **CanReadType:** Dado um objeto do tipo *Type*, este método retorna um valor booleano (onde o padrão é *True*), indicando se aquele tipo é ou não entendido por aquele formatador. O tipo identifica os eventuais parâmetros que existem no método do serviço. Aqui podemos fazer validações, como por exemplo, identificar se o tipo é ou não serializável, se possui um determinado atributo, etc.
- **ReadFromStreamAsync:** Se o método acima retornar *True*, então este método é executado. Como parâmetro ele recebe o tipo do objeto (o mesmo que foi passado para o método acima), e um objeto do tipo *Stream*, que é fonte das informações (estado) do objeto a ser criado. Este método retorna um *object*, que corresponde à um objeto criado dinamicamente e configurado com os valores provenientes do *Stream*.
- **CanWriteType:** Dado um objeto do tipo *Type*, este método retorna um valor booleano (padrão é *True*), indicando se aquele tipo é ou não entendido pelo formatador. O tipo identifica o retorno do método do serviço. Aqui podemos fazer validações, como por exemplo, identificar se o tipo é ou não serializável, se possui um determinado atributo, etc.
- **WriteToStreamAsync:** Se o método acima retornar *True*, então este método é executado. Como parâmetro, ele recebe o tipo do objeto a ser serializado e um *object* que corresponde a instância do objeto a ser gravado. Ainda recebemos um *Stream*, que é o destino do objeto serializado.

Como um possível exemplo, podemos criar um formatador específico para o formato CSV, que é um padrão bem tradicional, onde cada valor é separado pelo caracter ";". Abaixo temos a classe *CsvMediaTypeFormatter*, que herda de *MediaTypeFormatter*. Note a definição do formato *application/csv* sendo adicionado à coleção de *media types* suportados por este formatador customizado, que está acessível através da propriedade *SupportedMediaTypes*.

```
public class CsvMediaTypeFormatter : MediaTypeFormatter
{
    private const char SEPARATOR = ';';

    public CsvMediaTypeFormatter()
    {
        this.SupportedMediaTypes.Add(
            new MediaTypeHeaderValue("application/csv"));
    }
}
```

```
//implementação  
}
```

Como já era esperado, o que precisamos fazer a partir de agora é instalá-lo à execução. E para isso, recorreremos novamente a classe de configuração do ASP.NET Web API, que através da propriedade *Formatter* (que é uma coleção), podemos incluir classes que herdam de *MediaTypeFormatter*. O que fazemos aqui é instanciar e adicionar a classe *CsvMediaTypeFormatter*:

```
config.Formatters.Add(new CsvMediaTypeFormatter());
```

Com isso, ao receber ou retornar uma mensagem com o *header Accept* ou *Content-Type* definido como *application/csv*, a API já é capaz de interpretar e retornar objetos serializados neste formato.

Segurança

Como qualquer outro tipo de aplicação, serviços também devem lidar com segurança. E quando falamos em segurança sempre abordamos dois itens: autenticação e autorização. A autenticação é o processo que consiste em saber quem o usuário é, se ele está corretamente cadastrado e configurado, enquanto a autorização determina se esse usuário possui permissões para acessar o recurso que ele está solicitando. A autenticação, obrigatoriamente, sempre ocorre antes da autorização, pois não há como avaliar/conceder permissões sem antes saber quem ele é.

E como se não bastasse isso, quando estamos lidando com aplicações que encaminham mensagens de um lado para outro, é necessário também nos preocuparmos com a proteção da mesma enquanto ela viaja de um lado ao outro. Neste ponto o protocolo HTTPS ajuda bastante, já que ele é popularmente conhecido e a grande maioria dos hosts e clientes sabem lidar com ele.

Antes de falar sobre as peculiaridades do ASP.NET Web API, precisamos entender alguns conceitos de segurança que existem dentro da plataforma .NET desde a versão 1.0. Duas *interfaces* são utilizadas como base para os mecanismos de autenticação e autorização: *Identity* e *Principal* (namespace *System.Security.Principal*), respectivamente. A *interface Identity* fornece três propriedades autoexplicativas: *Name*, *AuthenticationType* e *IsAuthenticated*. Já a segunda possui dois membros que merecem uma atenção especial. O primeiro deles é a propriedade *Identity* que retorna a instância de uma classe que implemente a *interface Identity*, representando a identidade do usuário; já o segundo membro trata-se de um método chamado *IsInRole* que, dado uma papel, retorna um valor booleano indicando se o usuário corrente possui ou não aquele papel. Como podemos notar, as classes de autenticação e autorização trabalham em conjunto.

Dentro do namespace *System.Threading* existe uma classe chamada *Thread*. Essa classe determina como controlar uma *thread* dentro da aplicação. Essa classe, entre vários membros, possui uma propriedade estática chamada *CurrentPrincipal* que recebe e retorna uma instância de um objeto que implementa a *interface IPrincipal*. É através desta propriedade que devemos definir qual será a *identity* e *principal* que irá representar o contexto de segurança para a *thread* atual.

Há algumas implementações das *interfaces Identity* e *Principal* dentro do .NET Framework, como é o caso das classes *GenericIdentity*, *WindowsIdentity*, *GenericPrincipal* e *WindowsPrincipal*. Essas classes são utilizadas, principalmente, quando queremos implementar no sistema o mecanismo de autorização baseado em papéis, que dado um papel (muitas vezes um departamento), indica se o usuário pode ou não acessar o recurso. Para refinar melhor isso, ao invés de nomearmos por departamento, podemos definir as seções e funcionalidades do sistema que ele poderá acessar/executar.

Em qualquer tipo de projeto ASP.NET, há uma classe chamada *HttpContext*. Como o próprio nome sugere, essa classe expõe uma série de recursos que estarão acessível por

todo o ciclo da requisição dentro do servidor, e um desses recursos é o contexto de segurança do usuário, que através da propriedade *User* podemos ter acesso ao objeto (*IPrincipal*) que define a credencial do usuário corrente. Essa propriedade também estará acessível quando hospedar a API em um modelo de *web-hosting*. Quando utilizarmos o *self-hosting*, temos que recorrer a propriedade *CurrentPrincipal* da classe *Thread*. Se precisarmos hospedar a API em ambos os locais, temos que definir a credencial em ambos os locais, mas no caso do *HttpContext*, temos que nos certificar que a propriedade estática *Current* não seja nula, pois ela será se estiver em modo *self-hosting*.

Há diversas formas de trabalhar com autenticação no ASP.NET Web API, onde a maioria já são bastante conhecidas pelos desenvolvedores e padrões já estabelecidos no mercado. Abaixo temos as principais opções:

- **Basic:** A autenticação *Basic* faz parte da especificação do protocolo HTTP, que define um modelo simples (básico) para transmissão de nome de usuário e senha nas requisições. A sua simplicidade é tão grande quanto a sua insegurança. Por não haver qualquer meio de proteção (*hash*, criptografia, etc.), obriga a sempre trafegar essas requisições recorrendo à segurança do protocolo, e para isso, seremos obrigados a utilizarmos HTTPS.
- **Digest:** Também parte da especificação do protocolo HTTP, é uma modelo mais seguro quando comparado ao *Basic*, pois apenas o *hash* da senha (MD5) é enviado ao serviço.
- **Windows:** Como o próprio nome diz, a autenticação é baseada nas credenciais do usuário que está acessando o recurso, baseando-se em uma conta no *Windows (Active Directory)*. Entretanto isso é apenas útil quando estamos acessando a partir de uma *intranet* onde conseguimos ter um ambiente controlado e homogêneo.
- **Forms:** Desenhado para a *internet* a autenticação baseada em forms está presente no ASP.NET desde a sua versão 1.0, e é baseada em *cookies*.

Até então somente foi falado sobre os tipos de autenticação, o gerenciamento da identidade e dos objetos que temos a disposição e que representam o usuário, mas não de como e quando configurar isso. A escolha dependerá de como e onde quer (re)utilizar esse mecanismo de autenticação. Se quisermos avaliar em qualquer modelo de *hosting*, então a melhor opção é recorrer ao uso de *message handlers*, que trata-se de um pouco de extensibilidade do ASP.NET Web API e que pode rodar para todas as requisições ou para uma determinada rota. Haverá um capítulo para esgotar este assunto.

Para exemplificar vamos nos basear na autenticação *Basic*. A ideia é criar um *handler* para interceptar a requisição e extrair o *header* do HTTP que representa a credencial informada pelo usuário (*WWW-Authenticate*), e a valida em algum repositório de sua escolha, como uma base de dados. A partir daqui é necessário conhecermos como funciona o processo deste modelo, para conseguirmos dialogar com o cliente, para que assim ele consiga coordenar o processo de autenticação do usuário.

- O cliente solicita um recurso (página, serviço, etc.) que está protegido.
- Ao detectar que o cliente não está autenticado, o servidor exige que ele se autentique e informe as credenciais a partir do modelo *Basic*. Isso é informado a partir de um *header* chamado *WWW-Authenticate: Basic*.
- Neste momento, o servidor retorna uma resposta com o código 401 (*Unauthorized*), que instrui o cliente (navegador) a exigir as credenciais de acesso do usuário que está acessando o recurso.
- Uma vez informado, o browser recria a mesma requisição, mas agora envia nos headers da mesma o login e senha codificados em *Base64*, sem qualquer espécie de criptografia. Na resposta, o *header* enviado é o *Authorization: Basic [Username+Password Codificado]*.
- Quando este header acima estiver presente, o servidor (IIS) é capaz de validá-lo no *Windows/Active Directory*, e se for um usuário válido, permitirá o acesso, caso contrário retornará a mesma resposta com código 401, até que ele digite uma credencial válida.

Como dito acima, o login e a senha não são enviados até que sejam efetivamente exigidos. Nesta customização, ao identificarmos que o *header* não está presente na requisição, precisamos configurar a resposta para o cliente com o código 401, que representa acesso não autorizado, e informar na resposta o mesmo *header*, para continuar obrigando o usuário a informar o login e senha.

Para saber se o cliente informou as credenciais, precisamos detectar a presença do header chamado *Authorization*. Se existir, então precisamos decodificá-lo, utilizando o método *FromBase64String* da classe *Convert*, que dado uma *string*, retorna um *array* de *bytes* representando as credenciais separadas por um ":". Depois disso, tudo o que precisamos fazer é separá-los, para que assim poderemos efetuar a validação em algum repositório.

Depois de conhecer um pouco mais sobre o processo que ocorre entre cliente e serviço, vamos implementar isso no ASP.NET Web API utilizando um *message handler*, que é a forma que temos para interceptar as requisições que chegam para a API.

```
public class AuthenticationHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        string username = null;

        if (IsValid(request, out username))
        {
            var principal = new GenericPrincipal(new GenericIdentity(username), null);
            Thread.CurrentPrincipal = principal;

            if (HttpContext.Current != null)
                HttpContext.Current.User = principal;

            return base.SendAsync(request, cancellationToken);
        }
        else
        {
            return Task.Factory.StartNew(() =>
            {

```

```

        var r = new HttpResponseMessage(HttpStatusCode.Unauthorized);
        r.Headers.Add("WWW-Authenticate", "Basic realm=\"AppTeste\"");
        return r;
    });
}
}
}

```

Ao receber a requisição e ela for válida, antes dele encaminhar a mesma adiante, para que ela chegue até a ação que o cliente requisitou, ele cria o objeto que definirá a credencial/identidade do usuário que está acessando o recurso. Ao requisitar pela primeira vez e se estivermos consumindo isso em um navegador, ao receber esse código em conjunto com este *header*, uma janela é aberta para que você informe o *login* e senha, que serão encaminhados ao serviço.

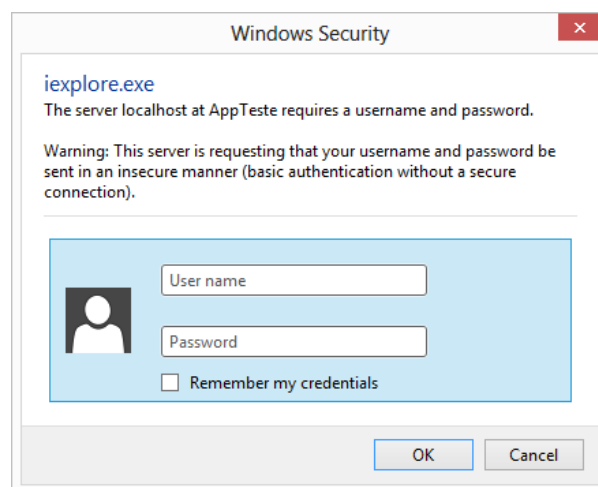


Figura 18 - Solicitação de login e senha pelo browser.

Ao interceptar essa requisição, podemos comprovar tudo o que foi escrito acima. Temos a resposta contendo o código 401, e depois de informado o login e senha, a nova requisição com o *header Authorization* contendo o valor codificado em *Base64*.



Figura 19 - Headers referente à autenticação Basic.

```
private static bool IsValid(HttpRequestMessage request, out string username)
{
    username = null;
    var header = request.Headers.Authorization;

    if (header != null && header.Scheme == "Basic")
    {
        var credentials = header.Parameter;

        if (!string.IsNullOrEmpty(credentials))
        {
            var decodedCredentials =
                Encoding.Default.GetString(Convert.FromBase64String(credentials));

            var separator = decodedCredentials.IndexOf(':');
            var password = decodedCredentials.Substring(separator + 1);

            username = decodedCredentials.Substring(0, separator);

            return username == password; //Validação em algum repositório
        }
    }

    return false;
}
```

A classe *ApiController* fornece uma propriedade chamada *User*, que retorna um objeto do tipo *IPrincipal*. Isso quer dizer que, se precisar extrair as credenciais do usuário dentro do método/ação, podemos recorrer a ela. E, para finalizar, é necessário incluir este *handler* na coleção de *handlers* do serviço, através do arquivo *Global.asax*:

```
config.MessageHandlers.Add(new AuthenticationHandler());
```

Depois da autenticação finalizada, de sabermos quem é o usuário que está acessando o recurso, chega o momento de controlarmos e sabermos e se ele tem as devidas permissões para acesso, que é o processo de autorização.

O principal elemento que controla o acesso é o atributo *AuthorizeAttribute* (*namespace System.Web.Http*), e pode ser aplicado em um *controller* inteiro ou individualmente em cada ação, caso precisemos de um controle mais refinado, ou até mesmo, em nível global, onde ele deve ser executado/assegurado independente de qualquer *controller* ou ação que seja executada dentro daquela aplicação. Este atributo possui duas propriedades: *Roles* e *Users*. Cada uma delas recebe um *string* com o nome dos papéis ou usuários que podem acessar determinado recurso, separados por vírgula.

Pelo fato deste atributo ser um filtro (falaremos mais sobre eles abaixo), ele é executado diretamente pela infraestrutura do ASP.NET Web API, que irá assegurar que o usuário está acessando está dentro daqueles nomes colocados na propriedade *Users* ou que ele esteja contido em algum dos papéis que são colocados na propriedade *Roles*. E como já era de se esperar, ele extrai as informações do usuário da propriedade *CurrentPrincipal* da classe *Thread*, qual definimos durante o processo de autenticação, dentro do método *SendAsync* do *handler* criado acima.

Depois de saber quem o usuário é, podemos extrair as permissões que ele possui, que provavelmente estarão armazenadas em algum repositório também. No exemplo que vimos acima da autenticação, há um segundo parâmetro na classe *GenericPrincipal* que é um *array* de *strings*, que representam os papéis do usuário. Abaixo temos aquele mesmo código ligeiramente alterado para buscar pelas permissões do usuário:

```
var principal =
    new GenericPrincipal(
        new GenericIdentity(username),
        CarregarPermissoes(username));

//....

private static string[] CarregarPermissoes(string username)
{
    if (username == "Israel")
        return new[] { "Admin", "IT" };

    return new[] { "Normal" };
}
```

Como comentado acima, temos três níveis que podemos aplicar o atributo *AuthorizeAttribute*: no método, no *controller* e em nível global. O código abaixo ilustra o uso destas três formas de utilização:

```
//Nível de Método
public class ClientesController : ApiController
{
    [Authorize(Roles = "Admin, IT")]
    public IEnumerable<Cliente> Get()
    {
        //...
    }
}

//Nível de Controller
[Authorize(Roles = "Admin")]
public class ClientesController : ApiController
{
    public IEnumerable<Cliente> Get()
    {
        //...
    }
}

//Nível Global
config.Filters.Add(new AuthorizeAttribute());
```

E se quisermos refinar ainda mais a autorização, podemos levar a validação disso para dentro do método. Basta remover o atributo e recorrer ao método *IsInRole* através da propriedade *User*, que dado o nome do papel, retorna um valor booleano indicando se o usuário atual está ou não contido nele.

```
public IEnumerable<Cliente> Get()
{
    if (User.IsInRole("Admin"))
        //Retornar todos os clientes.
}
```

```
//Retornar apenas os clientes da carteira  
}
```

Somente o fato de utilizar o atributo *AuthorizeAttribute* aplicado em alguns dos níveis que vimos, é o suficiente para que o ASP.NET Web API consiga assegurar que ele somente aquele determinado método/*controller* se ele estiver autenticado, independente dos papéis que ele possua. Se quisermos flexibilizar o acesso à alguns métodos, podemos recorrer ao atributo *AllowAnonymousAttribute* para conceder o acesso à um determinado método, mesmo que o *controller* esteja marcado com o atributo *AuthorizeAttribute*.

Testes e Tracing

É importante testarmos a maioria dos códigos que escrevemos, e quando estamos falando em testes, não estamos necessariamente nos referindo sobre testes de alto nível, onde colocamos o usuário para realizar os testes. Nos referimos a testes automatizados, onde conseguimos escrever códigos para testar códigos, possibilitando a criação de diversos cenários para se certificar de que tudo funcione como esperado.

Como comentamos no decorrer dos capítulos anteriores, o ASP.NET Web API possibilita a construção de serviços de modelo tradicional, ou seja, definir tipos que refletem o nosso negócio (*Cliente*, *Produto*, *Pedido*, etc.), bem como tipos mais simples (inteiro, string, booleano, etc.). Como sabemos, a finalidade é conseguir desenhar um serviço que nada saiba sobre a infraestrutura, como ele é exposto, características, etc.

Ainda temos a possibilidade de receber e/ou retonar objetos que refletem e fazem uso de algumas informações do protocolo HTTP, que é um detalhe muito importante na estrutura REST. Ao utilizar as classes que descrevem a requisição (*HttpRequestMessage*) e a resposta (*HttpResponseMessage*), podemos interagir com detalhes do protocolo.

Não há muito mistério em aplicar testes em cima da classe que representa o serviço quando estamos lidando com tipos customizados. Isso se deve ao fato de que neste modelo, como são simples classes, com métodos que executam tarefas e, eventualmente, retornam algum resultado, isso acaba sendo tratado como sendo uma classe de negócio qualquer.

Mas e quando queremos receber e/ou enviar dados para este serviço, utilizando instâncias das classes *HttpRequestMessage* e *HttpResponseMessage*? Felizmente, assim como no ASP.NET MVC, a Microsoft desenvolveu o ASP.NET Web API com a possibilidade de testá-lo sem estar acoplado à infraestrutura do ASP.NET, o que permite testar a classe do serviço, mesmo que ela receba ou devolva objetos característicos do protocolo HTTP.

Supondo que temos um serviço que possui dois métodos (*Ping* e *PingTipado*), podemos escrever testes e, conseqüentemente, utilizar a IDE do Visual Studio para executá-los, e como isso, nos antecipamos à eventuais problemas que possam acontecer, pois talvez seja possível capturar alguns desses problemas antes mesmo de levar o mesmo ao ambiente de produção.

```
public class ServicoDeExemplo : ApiController
{
    public HttpResponseMessage Ping(HttpRequestMessage request)
    {
        return new HttpResponseMessage()
        {
            StatusCode = HttpStatusCode.OK,
            Content = new StringContent(request.Content.ReadAsStringAsync().Result)
        };
    }

    public HttpResponseMessage PingTipado(HttpRequestMessage request)
    {
```

```

        if (request.Content == null)
            return request.CreateErrorResponse(HttpStatusCode.BadRequest,
                new HttpError("Conteúdo não definido"));

        return new HttpResponseMessage()
        {
            StatusCode = HttpStatusCode.OK,
            Content =
                new ObjectContent<Informacao>(
                    request.Content.ReadAsAsync<Informacao>().Result,
                    new JsonMediaTypeFormatter())
        };
    }
}

```

No primeiro exemplo, estamos testando o método *Ping*, instanciando a classe que representa o serviço, e passando ao método *Ping* a instância da classe *HttpRequestMessage*. Neste momento, poderíamos abastecer informações na coleção de *headers* da requisição, com o intuito de fornecer tudo o que é necessário para o que o método/teste possa executar com sucesso. Depois da requisição realizada, verificamos se o status da resposta corresponde ao status OK. Além disso verificamos também se o conteúdo da resposta está igual à informação que enviamos.

```

[TestMethod]
public void DadoUmaRequisicaoSimplesDeveRetornarStatusComoOK()
{
    var info = "teste";

    var response = new ServicoDeExemplo().Ping(new HttpRequestMessage()
    {
        Content = new StringContent(info)
    });

    Assert.AreEqual(HttpStatusCode.OK, response.StatusCode);
    Assert.AreEqual(info, response.Content.ReadAsStringAsync().Result);
}

```

O próximo passo é construir um teste para o método *PingTipado*. Esse método recebe como parâmetro a instância da classe *HttpRequestMessage*, definindo o conteúdo uma instância da classe *ObjectContent<T>*, onde definimos o tipo genérico *T* como sendo do tipo *Informacao*. A finalidade do teste é assegurar que, se passarmos uma instância nula da classe *Informacao*, uma resposta será retornada definindo o código de *status* como 400 (*Bad Request*).

```

[TestMethod]
public void DadoUmObjetoInfoNuloDeveRetornarComoErro()
{
    var response =
        new ServicoDeExemplo().PingTipado(new HttpRequestMessage());

    Assert.AreEqual(HttpStatusCode.BadRequest, response.StatusCode);
}

```

Finalmente, o próximo passo consiste em criar um teste, também para o método *PingTipado*, mas agora fornecendo a instância do objeto *Informacao* em um estado válido, para que o teste possa suceder se o serviço retornar a mesma instância da classe

Informacao, onde os membros da requisição reflitam o que está sendo retornado como resposta. Abaixo o código que efetua o tal teste:

```
[TestMethod]
public void DadoUmObjetoInfoDeveRetornarEleComInformacoesExtra()
{
    var info = new Informacao() {Codigo = "123", Dado = "Alguma Info" };
    var response =
        new ServicoDeExemplo()
            .PingTipado(new HttpRequestMessage()
                {
                    Content =
                        new ObjectContent<Informacao>(info, new JsonMediaTypeFormatter())
                })
            .Content
            .ReadAsStringAsync().Result;

    Assert.AreEqual(info.Dado, response.Dado);
}
```

Para abstrair ainda o que está sendo testado, a Microsoft criou uma *interface* chamada *IHttpActionResult*, que encapsula o resultado dos retornos de ações. A implementação desta classe será responsável por criar a mensagem de retorno, e a ação dentro do *controller* passa a retornar uma classe que implemente esta *interface*, facilitando os testes unitários. O ASP.NET já está preparado para também entender este tipo resultado, processando o retorno normalmente.

Já temos nativamente cinco implementações: *FormattedContentResult<T>*, *NegotiatedContentResult<T>*, *StatusCodeResult*, *ContinuationResult* e *MessageResult*. Cada uma delas é responsável por receber um determinado tipo de resultado, prepará-lo e repassar para o sistema de testes ou para a *pipeline* ASP.NET um tipo genérico, que nada mais é que a instância da classe *HttpResponseMessage*.

```
public IHttpActionResult Get(string nome)
{
    var artista =
        new Artista() { Id = 12, Nome = nome };

    return new FormattedContentResult<Artista>(
        HttpStatusCode.OK,
        artista,
        new JsonMediaTypeFormatter(),
        new MediaTypeHeaderValue("application/json"),
        this.Request);
}
```

Estamos recorrendo à classe *FormattedContentResult<T>* para retornar a instância da classe *Artista* no formato Json e com status de número 200 (OK). O ASP.NET entenderá o retorno normalmente, e quando estivermos criando os testes unitários para esta ação, independente do tipo de retorno que ela internamente defina (um objeto customizado, uma *string*, etc.), os testes sempre irão lidar a instância da classe *HttpResponseMessage*, e a partir dela, realizar todas as conferências necessárias para determinar se os testes executaram com sucesso ou não.

```

[TestMethod]
public void DeveRetornarRespostaCorreta()
{
    using (var request = new HttpRequestMessage())
    {
        var nome = "Max Pezzali";
        var response =
            new ArtistasController() { Request = request }
                .Get(nome)
                .ExecuteAsync(CancellationToken.None)
                .Result;

        Assert.AreEqual(HttpStatusCode.OK, response.StatusCode);
        Assert.AreEqual(nome, response.Content.ReadAsAsync<Artista>().Result.Nome);
    }
}

```

O que vimos até agora neste capítulo consiste em realizar os testes apenas nas classes que representam os serviços. Como os serviços REST usam o HTTP como parte do processo, muitas vezes somente a execução do *controller* é o suficiente para entender que ele foi executado com sucesso, o que nos obriga a recorrer à recursos do próprio HTTP para complementar a tarefa que está sendo executada e, conseqüentemente, também devemos compor isso em nossos testes.

Felizmente, pelo fato do ASP.NET Web API ser completamente desvinculado da infraestrutura, isso nos permite considerar os objetos que representam o “*proxy*” do cliente e o *hosting* do serviço nos testes, e validar se ao enviar, processar e retornar uma determinada requisição, se ela passa por todas os estágios de processamento dentro do *pipeline* do ASP.NET.

Os objetos *HttpServer* e o *HttpClient* foram construídos totalmente desvinculados de qualquer necessidade de somente executá-los em ambiente real. Com isso, podemos fazer uso destes mesmos objetos em um projeto de testes, onde podemos simular a mesma estrutura de objetos, suas configurações e seus interceptadores, que ao executar os testes, a requisição e resposta percorrerão todo o fluxo que percorreria quando ele for colocado em produção.

Para exemplificar isso, vamos considerar que temos um serviço que possui apenas dois métodos: um onde ele adiciona um objeto *Cliente* em um repositório qualquer, e outro que dado o *Id* deste *Cliente*, ele retorna o respectivo registro. Não vamos nos preocupar neste momento com boas práticas, mas no interior do *controller* podemos visualizar o repositório criado e sendo utilizado pelos dois métodos.

```

public class ClientesController : ApiController
{
    private static RepositorioDeClientes repositorio = new RepositorioDeClientes();

    [HttpGet]
    public Cliente Recuperar(int id)
    {
        return repositorio.RecuperarPorId(id);
    }

    [HttpPost]
    public HttpResponseMessage Adicionar(HttpRequestMessage request)

```

```

    {
        var cliente = request.Content.ReadAsAsync<Cliente>().Result;
        repositorio.Adicionar(cliente);

        var resposta = Request.CreateResponse<Cliente>(HttpStatusCode.Created,
cliente);
        resposta.Headers.Location =
            new Uri(string.Format("http://xpto/Clientes/Recuperar/{0}", cliente.Id));
        return resposta;
    }
}

```

Depois do serviço criado resta hospedarmos e consumirmos o mesmo através do projeto de testes.

```

[TestClass]
public class AcessoAosClientes
{
    private static HttpConfiguration configuracao;
    private static HttpServer servidor;
    private static HttpClient proxy;

    [ClassInitialize]
    public static void Inicializar(TestContext context)
    {
        configuracao = new HttpConfiguration();
        configuracao.Routes.MapHttpRoute(
            name: "DefaultApi",
            routeTemplate: "{controller}/{action}/{id}"
        );

        servidor = new HttpServer(configuracao);
        proxy = new HttpClient(servidor);
    }

    [TestMethod]
    public void DeveSerCapazDeFazerPingComUmNovoRegistro()
    {
        var resultadoDaCriacao =
            proxy.PostAsync(
                "http://xpto/Clientes/Adicionar",
                new StringContent(
                    "{\"Nome\":\"Israel\", \"Cidade\":\"Valinhos\"}",
                    Encoding.Default, "application/json")
            ).Result;

        Assert.AreEqual(HttpStatusCode.Created, resultadoDaCriacao.StatusCode);
        Assert.IsNotNull(resultadoDaCriacao.Headers.Location);

        var resultadoDaBusca =
            proxy.GetAsync(resultadoDaCriacao.Headers.Location).Result;
        var cliente = resultadoDaBusca.Content.ReadAsAsync<Cliente>().Result;

        Assert.AreEqual(1, cliente.Id);
        Assert.AreEqual("Israel", cliente.Nome);
        Assert.AreEqual("Valinhos", cliente.Cidade);
    }

    [ClassCleanup]
    public static void Finalizar()
    {
        proxy.Dispose();
        servidor.Dispose();
    }
}

```

As classes que representam o “*proxy*” e o *hosting* são declarados em nível de classe (teste). É interessante notar a construção destes objetos é realizada durante a inicialização da classe que representa o teste. No construtor do *hosting* (*HttpServer*) recebe como parâmetro as configurações para o serviço; já a classe *HttpClient* recebe como parâmetro o *HttpServer*, para que internamente, quando solicitarmos a requisição para este cliente, ele encaminhe para o serviço. A URI aqui pouco importa, já que o tráfego será realizado diretamente. Isso é possível porque a classe *HttpServer* herda da classe *HttpMessageHandler*.

Dependências

Não há como falarmos de testes unitários sem que se tenha uma API que seja bem construída. As boas práticas pregam que uma classe não deve ter mais responsabilidade do que seu propósito, ou seja, se você tem uma API que expõe as músicas de um determinado álbum, ela (a API) deve coordenar como essa listagem será montada, mas não é responsabilidade dela conhecer detalhes, por exemplo, do banco de dados.

Ao desenhar uma classe, antes de colocar um código dentro dela, é necessário analisar se é ela quem deveria realizar essa atividade. Quanto mais a classe depender de uma abstração ao invés de uma implementação, será muito mais fácil substituir isso durante a escrita dos testes. No exemplo abaixo temos um controller que necessita de um repositório para extrair o álbum de um artista.

```
public interface IRepositoryo
{
    Album BuscarAlbumPor(string artista);
}

public class ArtistasController : ApiController
{
    private readonly IRepositoryo repositorio;

    public Artistas(IRepositoryo repositorio)
    {
        this.repositorio = repositorio;
    }

    [HttpGet]
    public Album RecuperarAlbum(string artista)
    {
        return this.repositorio.BuscarAlbumPor(artista);
    }
}
```

Durante a escrita dos testes unitários, podemos criar e passar à classe *Artistas* uma representação em memória do repositório e, conseqüentemente, avaliar se o método *RecuperarAlbum* está atendendo o necessidade. A questão é como isso será realizado durante a execução da API.

Felizmente o ASP.NET Web API já possui internamente um local onde podemos adicionar todas as dependências do nosso *controller*, que durante a execução, ele será capaz de analisar a necessidade, construir o objeto, e entregá-lo à API para que seja utilizada. Para isso temos a interface *IDependencyResolver* (*namespace*

System.Web.Http.Dependencies), qual podemos utilizar para customizar a criação dos controllers, onde poderemos abastecer manualmente toda a necessidade que cada um possui.

```
public class HardcodeResolver : IDependencyResolver
{
    public IDependencyScope BeginScope()
    {
        return this;
    }

    public object GetService(Type serviceType)
    {
        if (serviceType == typeof(ArtistasController))
            return new ArtistasController(new RepositorioXml("Artistas.xml"));

        return null;
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return new List<object>();
    }

    public void Dispose() { }
}
```

Para que a classe *HardcodeResolver* funcione durante a execução, temos que apontar ao ASP.NET Web API que o objeto que criará a instância da classe que representará a API, resolverá todas as dependências e entregar para atender as requisições é ela. Novamente vamos recorrer ao objeto de configuração, que através da propriedade *DependencyResolver* podemos definir qualquer classe que implemente a *interface IDependencyResolver*.

```
config.DependencyResolver = new HardcodeResolver();
```

Os métodos *BeginScope* e *Dispose* são utilizados para controlar o tempo de vida dos objetos que são criados. Quando o controller ou qualquer objeto que ele seja capaz de resolver e criar é criado, podemos criar um objeto que define um escopo para ele, e após o *runtime* utilizá-lo, ele é devolvido para que seja adequadamente descartado, incluindo suas dependências internas que ela possa utilizar. Isso pode ser útil quando está utilizando algum *container* de inversão de controle (IoC). Se os objetos criados não tiverem a necessidade de gerenciamento de escopo para o descarte de recursos, então podemos retornar o *this*.

Tracing

Tracing é a forma que temos para monitorar a execução da aplicação enquanto ela está rodando. Isso é extremamente útil para diagnosticar problemas que ocorrem em tempo de execução, e que geralmente, por algum motivo específico faz com que a aplicação não se comporte como esperado.

O ASP.NET Web API já traz um mecanismo de captura extremamente simples de se trabalhar e tão poderoso quanto. Tudo acaba sendo realizado através da *interface ITraceWriter* (namespace *System.Web.Http.Tracing*), que dado uma implementação dela, o ASP.NET Web API captura detalhes referentes as mensagens HTTP e submete para que ela armazene no local de sua escolha.

Ele não vem com nenhuma implementação nativa, o que nos obriga a criar uma e acoplarmos à execução. Isso nos permitirá escolher qualquer meio de *logging*, como por exemplo o *log4net*, *ETW*, *Logging Application Block*, *System.Diagnostics*, etc. Esta interface fornece um único método chamado *Trace*, que recebe os seguintes parâmetros:

- **request:** recebe o objeto *HttpRequestMessage* associado com as informações que serão coletadas.
- **category:** uma string que determina a categoria em que as informação serão gravadas, permitindo agrupar informações que está relacionadas em pontos distintos da coleta.
- **level:** um enumerador com as opções (já conhecidas) que definem o nível de severidade da informação.
- **traceAction:** representa um delegate que permite ao chamador definir qualquer ação, que será executada quando o mecanismo de trace decidir coletar alguma informação.

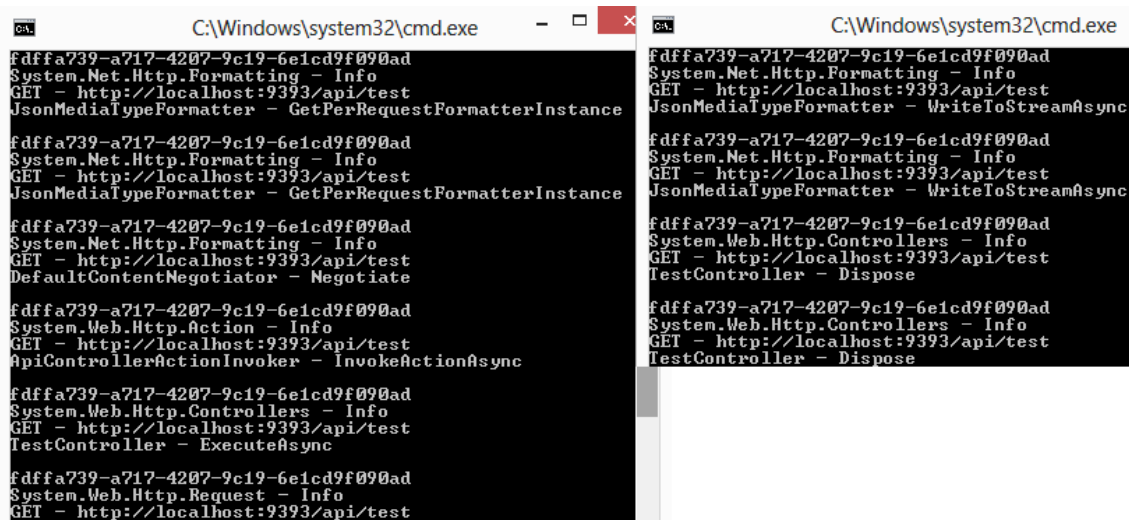
Para termos uma ideia das informações que são coletadas, abaixo temos um *logging* que se exibe as informações em uma aplicação console. A utilização da aplicação console é para mostrar o funcionamento do mecanismo, mas como já falado acima, poderíamos criar várias implementações. Quando estamos lidando com aplicações do mundo real, é necessário recorrermos a alguma biblioteca já existente e que faça grande parte do trabalho para armazenar e, principalmente, forneça uma forma simples para monitorar.

```
public class ConsoleLogging : ITraceWriter
{
    public void Trace(HttpRequestMessage request, string category,
        TraceLevel level, Action<TraceRecord> traceAction)
    {
        var record = new TraceRecord(request, category, level);
        traceAction(record);

        View(record);
    }

    private void View(TraceRecord record)
    {
        Console.WriteLine(record.RequestId);
        Console.WriteLine("{0} - {1}", record.Category, record.Level);
        Console.WriteLine("{0} - {1}", record.Request.Method,
record.Request.RequestUri);
        Console.WriteLine("{0} - {1}", record.Operator, record.Operation);
        Console.WriteLine();
    }
}
```

A classe *TraceRecord* representa um item de rastreamento e é ele que deve ser catalogado para futuras análises. No interior do método *Trace* construímos o objeto *TraceRecord* e antes de passarmos para o *delegate traceAction*, podemos customizar com informações específicas. E no exemplo acima, depois de configurado o *TraceRecord*, exibimos as propriedades deste projeto na console:



```
C:\Windows\system32\cmd.exe
fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Net.Http.Formatting - Info
GET - http://localhost:9393/api/test
JsonMediaTypeFormatter - GetPerRequestFormatterInstance

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Net.Http.Formatting - Info
GET - http://localhost:9393/api/test
JsonMediaTypeFormatter - GetPerRequestFormatterInstance

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Net.Http.Formatting - Info
GET - http://localhost:9393/api/test
DefaultContentNegotiator - Negotiate

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Web.Http.Action - Info
GET - http://localhost:9393/api/test
ApiControllerActionInvoker - InvokeActionAsync

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Web.Http.Controllers - Info
GET - http://localhost:9393/api/test
TestController - ExecuteAsync

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Web.Http.Request - Info
GET - http://localhost:9393/api/test

C:\Windows\system32\cmd.exe
fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Net.Http.Formatting - Info
GET - http://localhost:9393/api/test
JsonMediaTypeFormatter - WriteToStreamAsync

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Net.Http.Formatting - Info
GET - http://localhost:9393/api/test
JsonMediaTypeFormatter - WriteToStreamAsync

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Web.Http.Controllers - Info
GET - http://localhost:9393/api/test
TestController - Dispose

fdffa739-a717-4207-9c19-6e1cd9f090ad
System.Web.Http.Controllers - Info
GET - http://localhost:9393/api/test
TestController - Dispose
```

Figura 20 - Logs sendo exibidos na console.

É claro que a implementação não é suficiente para que tudo isso funcione. Para que ele seja acionado, é necessário acoplarmos à execução, e para isso, recorreremos ao objeto de configuração do ASP.NET Web API. Neste momento, tudo o que precisamos saber para que o *logging* customizado funcione é adicionar o seguinte comando na configuração da API:

```
config.Services.Replace(typeof(ITraceWriter), new ConsoleLogging());
```

E para finalizar, como pudemos perceber, somente informações inerentes aos estágios do processamento da requisição foram logados. E se desejarmos também incluir informações referentes as regras de negócio, ou melhor, incluir informações que são geradas no interior do *controller*? A classe *ApiController* possui uma propriedade chamada *Configuration*, que expõe o objeto de configuração a API e, consequentemente, nos permite acessar o *tracing* para incluir qualquer informação que achemos importante e necessário para quando precisarmos monitorar.

Para facilitar a inserção destas informações customizadas, a Microsoft incluiu uma classe estática com métodos de extensões à *interface ITraceWriter* com métodos nomeados com as severidades

```
using System.Web.Http;
using System.Web.Http.Tracing;

public class TestController : ApiController
{
```

```
public string Get()
{
    this.Configuration
        .Services
        .GetTraceWriter()
        .Info(this.Request, "Ações", "Alguma Info", "xpto");

    return "test";
}
```

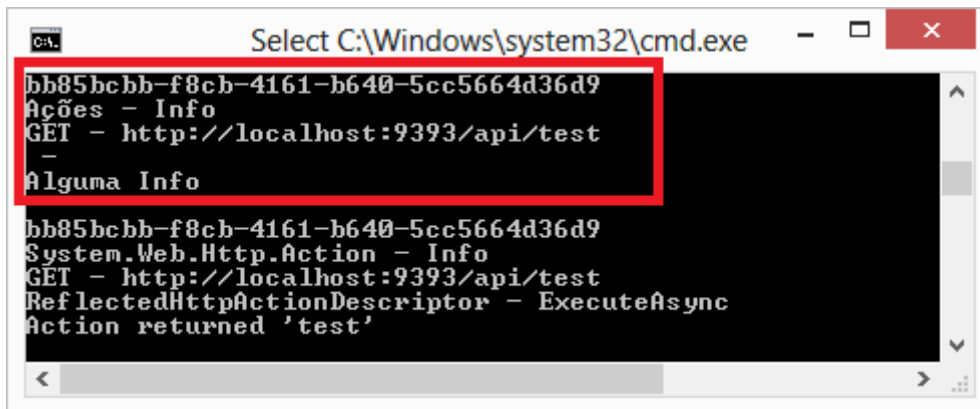


Figura 21 - Informação customizada sendo exibida.

Arquitetura e Extensibilidade

Para que seja possível tirar um maior proveito do que qualquer biblioteca ou *framework* tem a oferecer, é termos o conhecimento mais profundo de sua arquitetura. Apesar de ser opcional no primeiro momento, é de grande importância o conhecimento destes mecanismos, pois podem ser úteis durante alguma depuração que seja necessária ou durante a extensão de algum ponto para uma eventual customização, algo que também será abordado neste capítulo.

O entendimento da arquitetura nos dará uma visão bem detalhada do processamento das mensagens, sabendo o ponto correto para interceptar uma requisição a fim customizar algum elemento, interferir na escolha de alguma ação, aplicação de segurança (autenticação e autorização), implementar uma camada de *caching*, etc. Muito desses pontos já vimos no decorrer dos capítulos anteriores, e a própria Microsoft fez uso deles para implementar alguns elementos que já estão embutidos no ASP.NET Web API.

Antes de falarmos sobre a arquitetura do ASP.NET Web API, precisamos recapitular – de forma resumida – como é a infraestrutura do ASP.NET, e depois disso, veremos a bifurcação onde temos o desvio para o MVC, *Web Forms* e Web API.

Tudo começa com a criação da classe *HttpApplication*, que é o objeto que irá coordenar e gerenciar toda a execução das requisições que chegam para uma aplicação. Dentro deste objeto temos uma coleção de módulos, que são classes que implementam a interface *IHttpModule*, que são como filtros onde podemos examinar e modificar o conteúdo da mensagem que chega e que parte através do *pipeline*.

Depois que a mensagem passar pelos módulos, chega o momento de escolher o *handler* que tratará a requisição. O *handler* é o alvo da requisição, que irá receber a requisição e tratá-la, e devolver a resposta. Os *handlers* implementam a interface *IHttpHandler* ou *IHttpAsyncHandler* (para implementação assíncrona), e existem diversas classes dentro do ASP.NET, onde uma trata a requisição para uma página do *Web Forms*, para uma aplicação MVC, para um serviço ASMX, etc. Depois do *handler* executado, a mensagem de retorno é gerada, passa pelos módulos que interceptam o retorno, e parte para o cliente que solicitou o recurso.

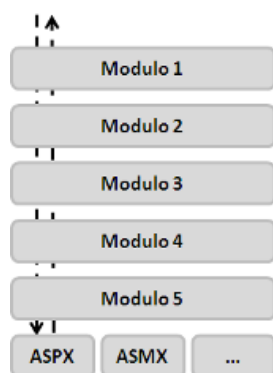


Figura 22 - Caminho percorrido pela requisição nos módulos e handlers.

Independentemente de qual recurso será acessado, o estágio inicial é comum para todos eles. Tudo o que falamos até aqui vale para quando estamos utilizando o *hosting* baseado na *web* (IIS/ASP.NET). No caso de *self-hosting*, onde hospedamos a API em nosso próprio processo, o caminho para a ser igual.

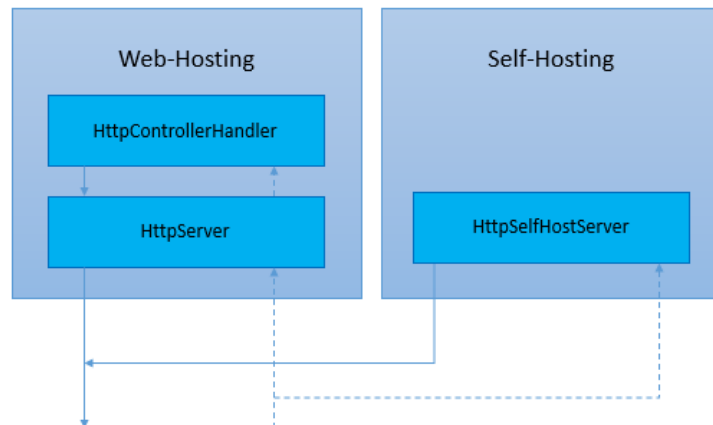


Figura 23 - Comparação entre web e self-hosting.

Como há comentado anteriormente, a classe *HttpSelfHostServer* (que herda da classe *HttpServer*), utilizada quando optamos pelo modelo de *self-hosting*, faz uso de recursos fornecidos pelo WCF, e depois de coletar as informações que chegam até o serviço, ele cria e passa adiante a instância da classe *HttpRequestMessage*.

É importante notar que do lado do *web-hosting* temos a classe *HttpControllerHandler*, que é a implementação da classe *IHttpHandler*, responsável por materializar a requisição (*HttpRequest*) no objeto do tipo *HttpRequestMessage*, enviando-a para a classe *HttpServer*.

Depois que a requisição pela classe *HttpServer*, um novo *pipeline* é iniciado, que possui vários pontos, e o primeiro deles é chamado de *Message Handlers*. Como o próprio nome diz, eles estão logo no primeiro estágio do *pipeline*, ou seja, independentemente de qual sua intenção para com o serviço, elas serão sempre serão executadas, a menos que haja algum critério que você avalie e rejeite a solicitação, o que proibirá o avanço do processamento para os próximos *handlers*.

Basicamente esses *handlers* recebem a instância de uma classe do tipo *HttpRequestMessage*, que traz toda a solicitação do usuário, e retornam a instância da classe *HttpResponseMessage*, contendo a resposta gerada para aquela solicitação. E como já ficou subentendido, podemos ter vários *handlers* adicionados ao *pipeline*, onde cada um deles pode ser responsável por executar uma tarefa distinta, como *logging*, autenticação, autorização, etc. A imagem abaixo ilustra esse fluxo:

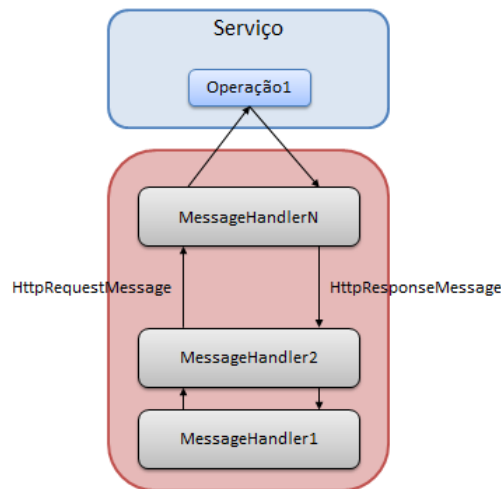


Figura 24 - Estrutura dos message handlers.

Para a criação de um *message handler* customizado, é necessário herdar da classe abstrata *DelegatingHandler*. Essa classe pode receber em seu construtor um objeto do tipo *HttpMessageChannel*. A finalidade deste objeto que é passado no construtor, é com o intuito de cada *handler* seja responsável por executar uma determinada tarefa, e depois passar para o próximo, ou seja, uma implementação do padrão *Decorator*.

```
public class ApiKeyVerification : DelegatingHandler
{
    private const string ApiKeyHeader = "Api-Key";
    private static string[] ValidKeys = new string[] { "18372", "92749" };

    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        if (IsValidKey(request))
            return base.SendAsync(request, cancellationToken);

        return Task.Factory.StartNew(() =>
            new HttpResponseMessage(HttpStatusCode.Unauthorized));
    }

    private static bool IsValidKey(HttpRequestMessage request)
    {
        var header = request.Headers.FirstOrDefault(h => h.Key == ApiKeyHeader);

        return
            header.Value != null &&
            ValidKeys.Contains(header.Value.FirstOrDefault());
    }
}
```

A classe acima intercepta a requisição e procura se existe um *header* chamado *Api-Key*. Se não houver ou se existir e não for uma chamada válida, ele rejeita a requisição retornando o código 401 (*Unauthorized*) ao cliente, que significa que ele não está autorizado a visualizar o conteúdo. É importante ressaltar que se a chave não for válida, a requisição não vai adiante, ou seja, ela já é abortada quando a primeira inconsistência for encontrada.

Podemos acoplar os *message handlers* em dois níveis para serem executados. Eles podem ser globais, que como o próprio nome sugere, serão executadas para todas as requisições que chegam a API, ou serem específicos para uma determinada rota. No primeiro caso, a instância do *message handler* é adicionada à coleção de *handlers*, através da propriedade *MessageHandlers*. Já a segunda opção, recorreremos à um *overload* do método *MapHttpRequest*, onde em seu último parâmetro temos a opção de incluir o *message handler* específico para ela. Abaixo temos o exemplo de como fazemos para utilizar uma ou outra opção:

```
//Global
config.MessageHandlers.Add(new ApiKeyVerification());

//Por rota
config.Routes.MapHttpRequest(
    name: "Default",
    routeTemplate: "api/{controller}",
    defaults: null,
    constraints: null,
    handler:
        HttpClientFactory.CreatePipeline(
            new ApiControllerDispatcher(config),
            new DelegatingHandler[] { new ApiKeyVerification() }));
```

Se encaminharmos a requisição com uma chave inválida, podemos perceber que não somos autorizados a acessar o recurso. A partir do momento que colocamos uma chave que o serviço entende como válida, o resultado é retornado. A imagem abaixo comprova essa análise.

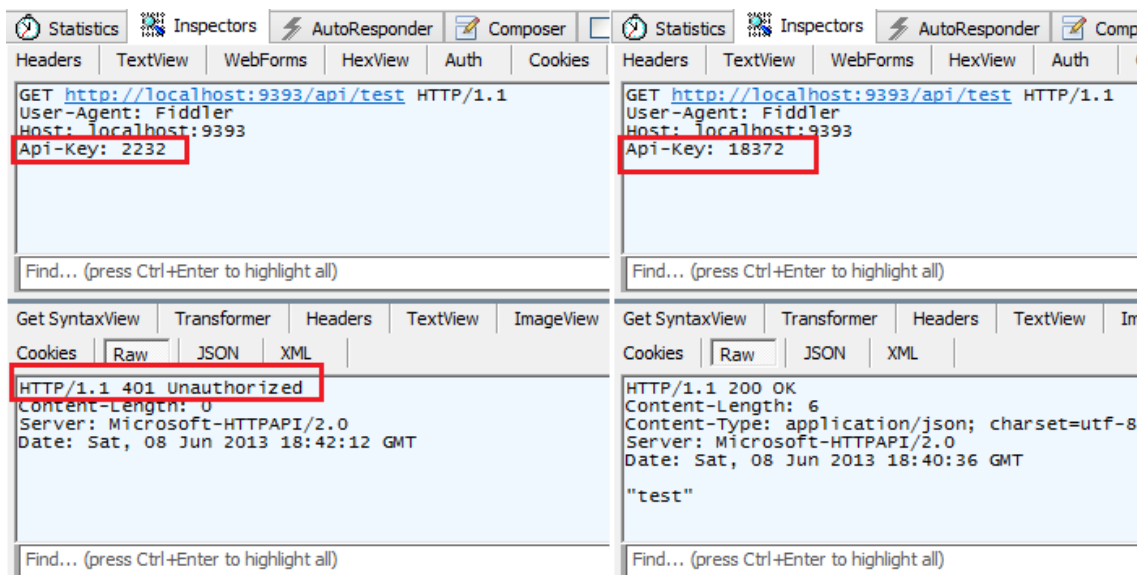


Figura 25 - Headers customizados para controle de acesso.

Existem dois *message handlers* embutidos no ASP.NET Web API que desempenham um papel extremamente importante no *pipeline*. O primeiro deles é *HttpRequestDispatcher*, que avalia se existe um *message handler* específico para a rota que foi encontrada. Se houver, ele deve ser executado.

Caso não seja, a requisição é encaminhada para um outro *message handler* chamado *HttpControllerDispatcher*. Uma vez que passamos por todos os *handlers* configurados, este será responsável por encontrar e ativar o *controller*. No código acima mencionamos a classe *HttpControllerDispatcher* quando configuramos o *message handler* *ApiKeyVerification* em nível de rota.

A procura, escolha e ativação do *controller* são tarefas realizadas por elementos que também são extensíveis. Eles são representados pelas seguintes *interfaces*: *IHttpControllerSelector* e *IHttpControllerActivator*. Depois do *controller* encontrado, é o momento de saber qual ação (método) dentro dele será executada. Da mesma forma, se quisermos customizar, basta recorrer à implementação da *interface* *IHttpActionSelector*.

Acima vimos os *message handlers* no contexto do lado do servidor, mas pela simetria que existe na codificação do servidor comparado ao cliente, podemos recorrer aos mesmos recursos para interceptar e manipular tanto a requisição quanto a resposta do lado do cliente. O construtor da classe *HttpClient* pode receber como parâmetro a instância de uma classe do tipo *HttpMessageHandler*. Ao omitir qualquer inicialização, por padrão, o *handler* padrão é o *HttpClientHandler*, que herda de *HttpMessageHandler*, que é responsável pela comunicação com o HTTP, já em nível de rede.

Se quisermos customizar, incluindo *handlers* para analisar e/ou alterar a saída ou o retorno da requisição no cliente, podemos recorrer ao método de extensão chamado *Create* da classe *HttpClientFactory*, que recebe um *array* contendo todos os *handlers* que serão disparados, quais serão disparados em ordem inversa ao que é inserido na coleção. Este mesmo método faz o trabalho para – também – inserir o *handler* *HttpClientHandler* que se faz necessário em qualquer situação.

```
using (var client =  
    HttpClientFactory.Create(new ValidacaoDeSessao()))  
{  
    //...  
}  
  
public class ValidacaoDeSessao : DelegatingHandler  
{  
    //...  
}
```

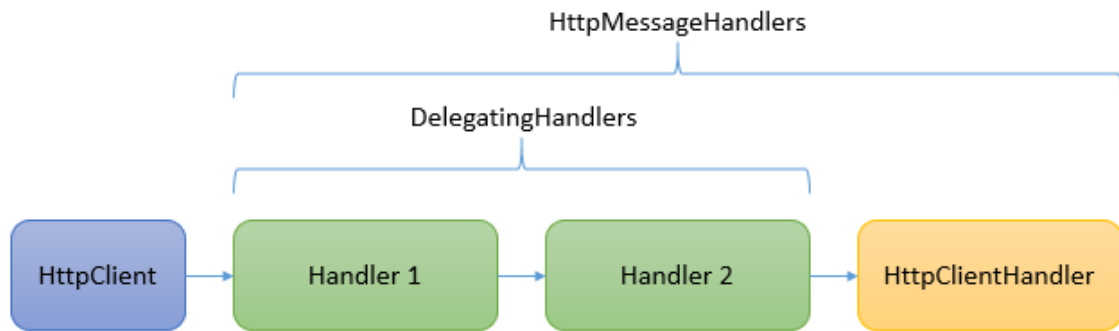


Figura 26 - Estrutura dos message handlers do lado do cliente.

Apesar do *controller* e a ação dentro dele terem sido encontrados, podemos ainda realizar alguma espécie de interceptação para que ainda façamos alguma tarefa antes de executarmos a ação. Eis que surgem os filtros, que servem como uma forma de concentrar alguns elementos de *cross-cutting*, como segurança, tradução de exceções em erros HTTP, etc.

Os filtros podem ser aplicados em ações específicas dentro do *controller*, no *controller* como um todo, ou para todas as ações em todos os *controllers* (global). O benefício que temos na utilização de filtros é a granularidade onde podemos aplicá-los. Talvez interromper a requisição logo nos primeiros estágios (via *handlers*) possa ser mais eficaz, pois muitas vezes você não precisa passar por todo o *pipeline* para tomar essa decisão.

Há um *namespace* chamada *System.Web.Http.Filters*, que possui vários filtros já predefinidos. Todo filtro herda direta ou indiretamente da classe abstrata *FilterAttribute*, que já possui a estrutura padrão para todos os filtros. O diagrama abaixo ilustra essa hierarquia.

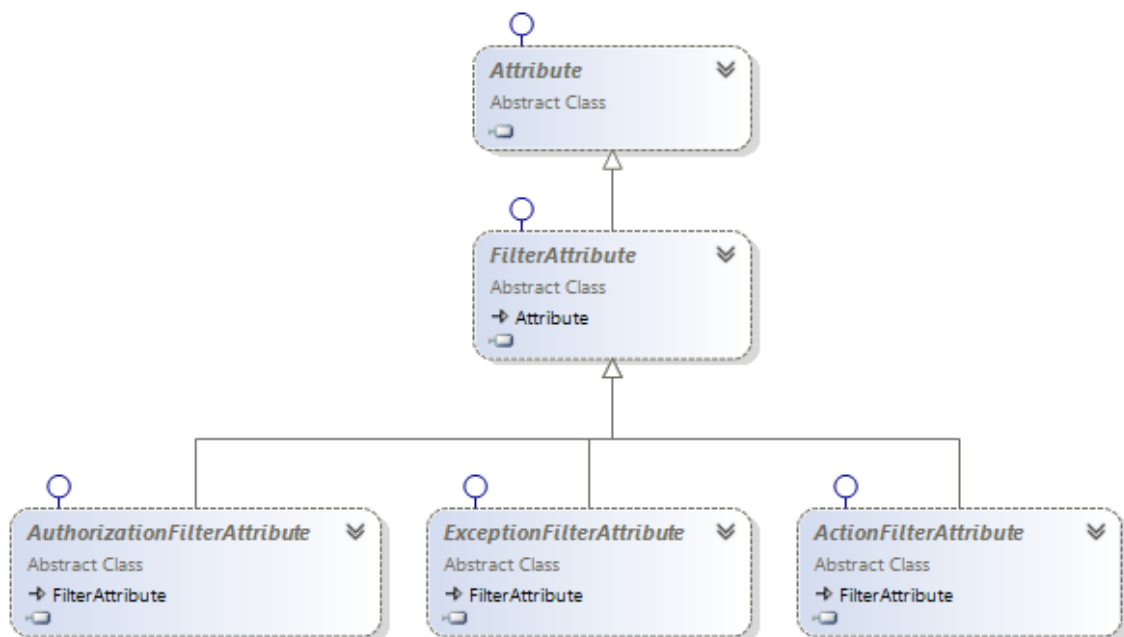


Figura 27 - Hierarquia das classes dos filtros existentes dentro do framework.

Além da classe base para os filtros, já temos algumas outras, também abstratas, que definem a estrutura para que seja criado um filtro para controlarmos a autorização, outro para controlarmos o tratamento de erros e um que nos permite interceptar a execução de alguma ação dentro do *controller*.

Para exemplificar a customização de um filtro, podemos criar um que impossibilite o acesso à uma ação se ela não estiver protegida por HTTPS. A classe *ActionFilterAttribute* fornece dois métodos que podemos sobrescrever na classe derivada: *OnActionExecuting* e *OnActionExecuted*. Como podemos perceber, um deles é disparado antes e o outro depois da ação executada.

```
public class ValidacaoDeHttps : ActionFilterAttribute
{
    public override void OnActionExecuting(HttpContext actionContext)
    {
        var request = actionContext.Request;

        if (request.RequestUri.Scheme != Uri.UriSchemeHttps)
        {
            actionContext.Response =
                request.CreateResponse(
                    HttpStatusCode.Forbidden,
                    new StringContent("É necessário que a requisição seja HTTPS."));
        }
    }
}
```

O fato da classe *ActionFilterAttribute* herdar da classe *Attribute*, podemos aplicar este atributo tanto no *controller* quanto em um ou mais ações, ou seja, podemos ter um refinamento mais apurado, pois temos condições de aplicar isso em certos casos, em outros não. No exemplo abaixo optamos apenas por proteger por HTTPS a ação *Teste2*. Se quisermos que todas as ações dentro deste *controller* sejam protegidas, basta apenas elevarmos o atributo, decorando a classe com o filtro criado.

```
public class TestController : ApiController
{
    [HttpGet]
    public string Teste1()
    {
        return "teste1";
    }

    [HttpGet]
    [ValidacaoDeHttps]
    public string Teste2()
    {
        return "teste2";
    }
}
```

Finalmente, se desejarmos que este atributo seja aplicado em todas as ações de todos os controllers, adicionamos este filtro em nível global, através da configuração da API:

```
config.Filters.Add(new ValidacaoDeHttps());
```

Sobrescrita de Filtros

Ao aplicar o filtro em nível global, evita termos que decorarmos cada (nova) ação ou cada (novo) *controller* com um determinado atributo, evitando assim que, por algum descuido, um determinado código deixe de rodar antes e/ou depois de cada ação. Sendo assim, o nível global nos permite aplicar incondicionalmente para todas as ações, e se quisermos aplicar um filtro para uma ou outra ação, decoramos o atributo diretamente nele.

Só que ainda há uma outra situação, que é quando precisamos aplicar determinados filtros para a grande maioria das ações, mas em poucas delas não queremos que o filtro seja aplicado. Para isso, quando configurarmos um filtro em nível global, podemos sobrescrever uma determinada ação para que os filtros não sejam aplicados nela. Para isso, entra em cena um atributo chamado *OverrideActionFiltersAttribute*, que quando aplicado em uma determinada ação, ele ignora os filtros aplicados em nível global.

```
public class TestController : ApiController
{
    [HttpGet]
    public string Teste1()
    {
        return "teste1";
    }

    [HttpGet]
    [OverrideActionFilters]
    public string Teste2()
    {
        return "teste2";
    }
}
```

Além deste atributo, temos outros três com a mesma finalidade, ou seja, interromper a execução de determinados tipos de filtros que foram aplicados em nível global. Os outros atributos que temos para isso são: *OverrideAuthenticationAttribute*, *OverrideAuthorizationAttribute* e *OverrideExceptionAttribute*.

Configurações

Durante todos os capítulos vimos diversas configurações que são realizadas em nível global. Para todas elas recorremos ao objeto *HttpConfiguration*. Ele fornece diversas propriedades e métodos que nos permite interagir com todos os recursos que são utilizados durante a execução das APIs. Uma das propriedades que vale ressaltar é a *Services* do tipo *ServicesContainer*. Essa classe consiste em armazenar todos os recursos que são utilizados pelo ASP.NET Web API para fornecer ao *runtime* os responsáveis por executar cada tarefa específica, tais como: criação do *controller*, gestor de dependências, gestor de *tracing*, etc.

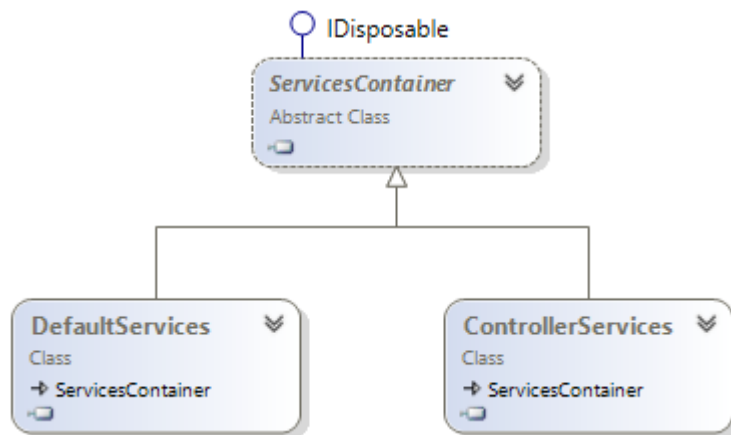


Figura 28 - Classes para a customização das configurações.

A propriedade *Services* da classe *ServicesContainer* é inicializada com a instância da classe *DefaultServices*, que vale para todos e qualquer *controller* dentro da aplicação. Podemos variar certas configurações para cada *controller*, e é justamente para isso que temos a classe *ControllerServices*.

Se quisermos customizar a configuração por *controller*, onde cada um deles possui uma necessidade específica, basta implementarmos a *interface IConfiguration* (namespace *System.Web.Http.Controllers*), onde através do método *Initialize*, realizamos todas as configurações específicas, e durante a execução o ASP.NET irá considerar essas configurações, sobrescrevendo as globais, e para aquelas que não alterarmos, a configuração global será utilizada.

```

public class ConfiguracaoPadrao : Attribute, IConfiguration
{
    public void Initialize(
        HttpControllerSettings controllerSettings,
        HttpControllerDescriptor controllerDescriptor)
    {
        controllerSettings.Formatters.Add(new CsvMediaTypeFormatter());
    }
}

[ConfiguracaoPadrao]
public class TestController : ApiController
{
    //ações
}
  
```