

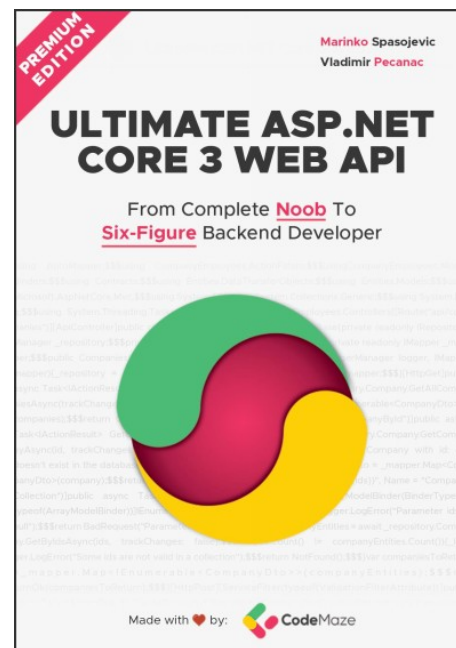
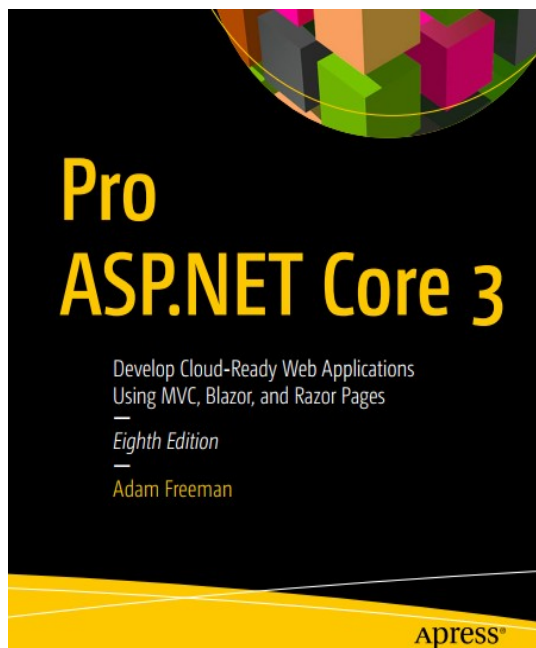
ASP.NET Core WEB API

Entendimento básico

por Jeann Andrade
Desenvolvedor de Software



Referência



Roteiro

- Revisão sobre HTTP e REST
- Entendendo o ASP.NET Core
- Ciclo de vida de uma requisição
- CORS (Cross-Origin Resource Sharing)
- Configuração do IIS
- Acompanhar o código do sistema CompanyEmployee



Quando surgiu HTTP e REST?

- WWW, HTML e HTTP nascem em meados de 1992 (Tim Berners-Lee)
- Em 2000 o termo REST surge na tese de doutorado de Roy Fielding, que também é Co-autor do HTTP.
- Em 2006 o REST começa a se difundir em aplicações.



Requisição: SOAP x REST

Requisição em SOAP

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">  
  <soap:Body>  
    <listarClientes xmlns="http://brejaonline.com.br" />  
  </soap:Body>  
</soap:Envelope>
```

Requisição em REST

<http://localhost:8080/cevejarla/clientes>



Princípios básicos de REST (REpresentational State Transfer)

- Uso adequado dos métodos HTTP;
- Uso adequado de URL's;
- Uso de códigos de status padronizados para representação de sucessos ou falhas;
- Uso adequado de cabeçalhos HTTP;
- Interligações entre vários recursos diferentes.



Fundações do REST

- O “marco zero” de REST é o recurso. Em REST, tudo é definido em termos de recursos, sendo estes os conjuntos de dados que são trafegados pelo protocolo.
- Os recursos são representados por URI's.
 - <http://localhost:8080/cevejaria/clientes> → Todos os clientes
 - <http://localhost:8080/cevejaria/clientes/1> → Cliente com id 1
- Todo serviço REST é baseado nos chamados recursos, que são entidades bem definidas em sistemas, que possuem identificadores e endereços (URL's) próprios.



Fundamentos do HTTP

- Formato básico de requisições:

<método> <URL> HTTP/<versão>

<Cabeçalhos - Sempre vários, um em cada linha>

<corpo da requisição>

Exemplo:

GET /cervejaria/clientes HTTP/1.1

Host: localhost:8080

Accept: text/html

- Formato básico de respostas:

HTTP/<versão> <código de status> <descrição do código>

<cabeçalhos>

<resposta>

Exemplo:

HTTP/1.1 200 OK

Content-Type: text/xml

Content-Length: 245



Métodos HTTP

- Idempotência - se a mesma requisição, realizada múltiplas vezes, provoca alterações no lado do servidor como se fosse uma única, então esta é considerada idempotente.
- Segurança - Quanto à segurança, os métodos são assim considerados se não provocarem quaisquer alterações nos dados contidos.

	Idempotente	Seguro
GET	X	X
POST		
PUT	X	
DELETE	X	
HEAD	X	X
OPTIONS	X	X
PATCH	NÃO	NÃO



Cabeçalhos

- Os cabeçalhos, em HTTP, são utilizados para trafegar todo o tipo de meta informação a respeito das requisições. Exemplo:

GET / HTTP/1.1

Host: www.casadocodigo.com.br

User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:19.0) Gecko/20100101 Fire

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: pt-BR,pt;q=0.8,en-US;q=0.5,en;q=0.3

Accept-Encoding: gzip, deflate

Connection: keep-alive



Cabeçalhos

- Host - mostra qual foi o DNS utilizado para chegar a este servidor
- User-Agent - fornece informações sobre o meio utilizado para acessar este endereço
- Accept - realiza negociação com o servidor a respeito do conteúdo aceito
- Accept-Language - negocia com o servidor qual o idioma a ser utilizado na resposta
- Accept-Encoding - negocia com o servidor qual a codificação a ser utilizada na resposta
- Connection - ajusta o tipo de conexão com o servidor (persistente ou não).
- Tecnicamente falando, os cabeçalhos são utilizados para tráfego de meta dados ou seja, informações a respeito da informação “de verdade”.



Media Types

- Ao realizar uma requisição para o Media Type é utilizado, indicando para o navegador qual é o tipo da informação que está sendo trafegada. Isto é utilizado para que o cliente saiba como trabalhar com o resultado, dessa maneira, os Media Types são formas padronizadas de descrever uma determinada informação.
- São divididos em tipos e subtipos, e acrescentados de parâmetros (se houverem). São compostos com o seguinte formato: tipo/subtipo. Se houverem parâmetros, o ';' será utilizado para delimitar a área dos parâmetros. Portanto, um exemplo de Media Type seria `text/xml; charset="utf-8"` (para descrever um XML cuja codificação seja UTF-8).
- Os mais comuns são: `application`, `audio`, `image`, `text`, `video` e `vnd`
- Em serviços REST, vários tipos diferentes de Media Types são utilizados. As maneiras mais comuns de representar dados estruturados, em serviços REST, são via XML e JSON, que são representados pelos Media Types `application/xml` e `application/json`, respectivamente.
- Os Media Types são negociados a partir dos cabeçalhos `Accept` e `Content-Type`. O primeiro é utilizado em requisições, e o segundo, em respostas.



Códigos de Status

- Toda requisição que é enviada para o servidor retorna um código de status. Esses códigos são divididos em cinco famílias: 1xx, 2xx, 3xx, 4xx e 5xx, sendo:
- 1xx - Informacionais
- 2xx - Códigos de sucesso
- 3xx - Códigos de redirecionamento
- 4xx - Erros causados pelo cliente
- 5xx - Erros originados no servidor



Códigos de Status: 2xx

- 200 – OK → Indica que a operação indicada teve sucesso.
- 201 – Created → Indica que o recurso desejado foi criado com sucesso. Deve retornar um cabeçalho Location, que deve conter a URL onde o recurso recém-criado está disponível.
- 202 – Accepted → Indica que a solicitação foi recebida e será processada em outro momento. É tipicamente utilizada em requisições assíncronas, que não serão processadas em tempo real. Por esse motivo, pode retornar um cabeçalho Location, que trará uma URL onde o cliente pode consultar se o recurso já está disponível ou não.
- 204 - No Content → Usualmente enviado em resposta a uma requisição PUT, POST ou DELETE, onde o servidor pode recusar-se a enviar conteúdo.
- 206 - Partial Content → Utilizado em requisições GET parciais, ou seja, que demandam apenas parte do conteúdo armazenado no servidor (caso muito utilizado em servidores de download)



Códigos de Status: 3xx

- 301 - Moved Permanently → Significa que o recurso solicitado foi realocado permanentemente. Uma resposta com o código 301 deve conter um cabeçalho Location com a URL completa (ou seja, com descrição de protocolo e servidor) de onde o recurso está atualmente.
- 303 - See Other → É utilizado quando a requisição foi processada, mas o servidor não deseja enviar o resultado do processamento. Ao invés disso, o servidor envia a resposta com este código de status e o cabeçalho Location, informando onde a resposta do processamento está.
- 304 - Not Modified → É utilizado, principalmente, em requisições GET condicionais - quando o cliente deseja ver a resposta apenas se ela tiver sido alterada em relação a uma requisição anterior.
- 307 - Temporary Redirect → Similar ao 301, mas indica que o redirecionamento é temporário, não permanente.



Códigos de Status: 4xx

- 400 - Bad Request → É uma resposta genérica para qualquer tipo de erro de processamento cuja responsabilidade é do cliente do serviço.
- 401 – Unauthorized → Utilizado quando o cliente está tentando realizar uma operação sem ter fornecido dados de autenticação (ou a autenticação fornecida for inválida).
- 403 – Forbidden → Utilizado quando o cliente está tentando realizar uma operação sem ter a devida autorização.
- 404 - Not Found → Utilizado quando o recurso solicitado não existe.
- 405 - Method Not Allowed → Utilizado quando o método HTTP utilizado não é suportado pela URL. Deve incluir um cabeçalho Allow na resposta, contendo a listagem dos métodos suportados (separados por “,”).
- 409 – Conflict → Utilizado quando há conflitos entre dois recursos. Comumente utilizado em resposta a criações de conteúdos que tenham restrições de dados únicos - por exemplo, criação de um usuário no sistema utilizando um login já existente. Se for causado pela existência de outro recurso (como no caso citado), a resposta deve conter um cabeçalho Location, explicitando a localização do recurso que é a fonte do conflito.
- 410 – Gone → Semelhante ao 404, mas indica que um recurso já existiu neste local.
- 412 - Precondition failed → Comumente utilizado em resposta a requisições GET condicionais.
- 415 - Unsupported Media Type → Utilizado em resposta a clientes que solicitam um tipo de dados que não é suportado - por exemplo, solicitar JSON quando o único formato de dados suportado é XML.



Códigos de Status: 5xx

- 500 - Internal Server Error → É uma resposta de erro genérica, utilizada quando nenhuma outra se aplica.
- 503 - Service Unavailable → Indica que o servidor está atendendo requisições, mas o serviço em questão não está funcionando corretamente. Pode incluir um cabeçalho Retry-After, dizendo ao cliente quando ele deveria tentar submeter a requisição novamente.



Semânticas de recursos

- Assumindo que uma cerveja é um recurso. Assim, as regras de REST dizem que as cervejas devem ter uma URL própria e que esta URL deve ser significativa. Desta forma, uma boa URL para cervejas pode ser /cervejas.
- De acordo com o modelo REST, esta URL realizará interação com todas as cervejas do sistema. Para tratar de cervejas específicas, são usados identificadores. Estes identificadores podem ter qualquer formato. Desta forma, deve ser possível buscar uma cerveja através do código de barras, com a URL /cervejas/123456.



Interação por métodos

- Para interagir com as URL's, os métodos HTTP são utilizados. A regra de ouro para esta interação é que URL's são substantivos, e métodos HTTP são verbos. Isto quer dizer que os métodos HTTP são os responsáveis por provocar alterações nos recursos identificados pelas URL's.
- Estas modificações são padronizadas, de maneira que:
 - GET - recupera os dados identificados pela URL
 - POST - cria um novo recurso
 - PUT - atualiza um recurso
 - DELETE - apaga um recurso



Interação por métodos: Tarefas

- Criar serviços REST que executem tarefas de negócio (ex: enviar um e-mail, validar um CPF e outras tarefas que não necessariamente envolvam interação com um banco de dados) é tarefa mais complexa. Exige certo grau de domínio da teoria sobre REST, e muitas vezes não existem respostas 100% assertivas em relação à correção da solução.
- Para obter resultados satisfatórios, deve-se sempre ter o pensamento voltado para a orientação a recursos.
- Tomemos como exemplo o caso do envio de e-mails: a URL deve ser modelada em formato de substantivo, ou seja, apenas /email. Para enviar o e-mail, o cliente pode usar o método POST - ou seja, como se estivesse “criando” um e-mail.
- Existem, também, casos em que as soluções podem não parecer triviais à primeira vista. Um caso clássico é o de validações (de CPF, por exemplo). A URL pode ser claramente orientada a substantivos (/validacao/CPF, por exemplo), mas e quanto aos métodos? A solução para este caso é pensar da seguinte maneira: “eu preciso obter uma validação? Ou criar uma, ou apagar, ou atualizar?”. Desta maneira, a resposta natural para a questão parece convergir para o método GET.



Representações distintas

- Um dos pilares de REST é o uso de media types para alterar as representações de um mesmo conteúdo, sob perspectivas distintas. Esta ótica fica evidente quando se responde à pergunta: “que lado do recurso estou procurando ou quero enxergar?”

```
GET /cervejas/1 HTTP/1.1  
Host: localhost:8080  
Accept: text/xml
```



```
<?xml version="1.0" encoding="utf-8" ?>  
<cerveja>  
  <id>1</id>  
  <nome>Erdinger Weissbier</nome>  
  <data-embalagem>2021-10-24</data-embalagem>  
</cerveja>
```

```
GET /cervejas/1 HTTP/1.1  
Host: localhost:8080  
Accept: image/*
```



Uso correto de status codes

- Algo que REST também prevê é o uso correto dos status codes HTTP. Na prática, isso significa conhecê-los e aplicar de acordo com a situação. Por exemplo, o código 200 (OK) é utilizado em grande parte das situações (o que pode acabar “viciando” o desenvolvedor), mas a criação de recursos deve retornar, quase sempre, o código 201 (Created) e o cabeçalho Location, indicando a localização do recurso criado.
- Ainda, se dois clientes realizarem criação de recursos de forma que estes forneçam a chave utilizada para referenciar o recurso (ou seja, existe uma restrição de que a chave deve ser única por recurso), e estes dois clientes fornecerem a mesma chave na criação do recurso, o código de status 409 (Conflict) deve ser utilizado.



HATEOAS

- Hypermedia As The Engine Of Application State (HATEOAS) utiliza links como referência a ações que podem ser tomadas a partir da entidade atual.
- Estes links, em HATEOAS, têm dois segmentos: links estruturais ou transicionais.
- Os links estruturais, como diz o nome, são referentes à estrutura do próprio Conteúdo.
- Já os links transicionais são relativos a ações, ou seja, a ações que o cliente pode efetuar utilizando aquele recurso.

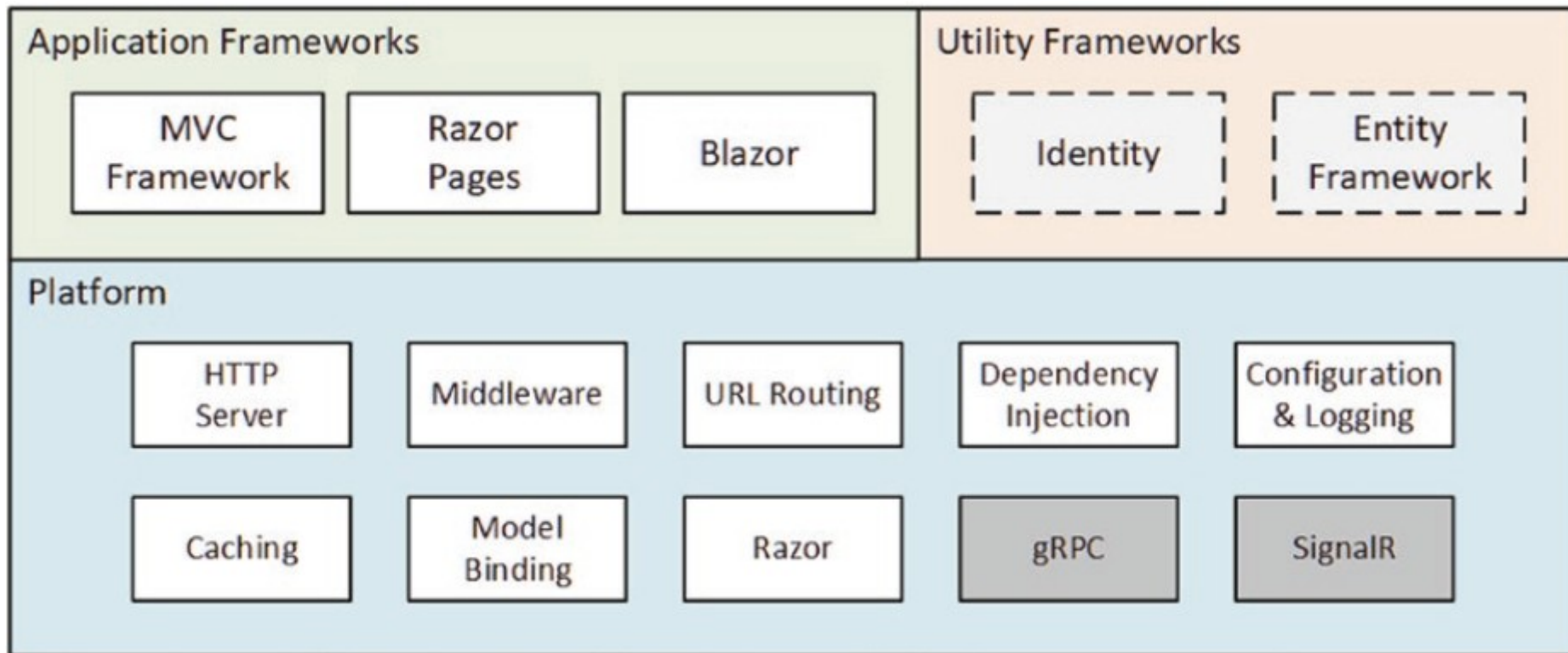
```
<compra>
  <item>
    <cerveja id="1">Stella Artois</cerveja>
    <quantidade>1</quantidade>
  </item>
</compra>
```



```
<compra id="123">
  <item>
    <cerveja id="1">Stella Artois</cerveja>
    <quantidade>1</quantidade>
  </item>
  <link rel="pagamento" href="/pagamento/123" />
</compra>
```



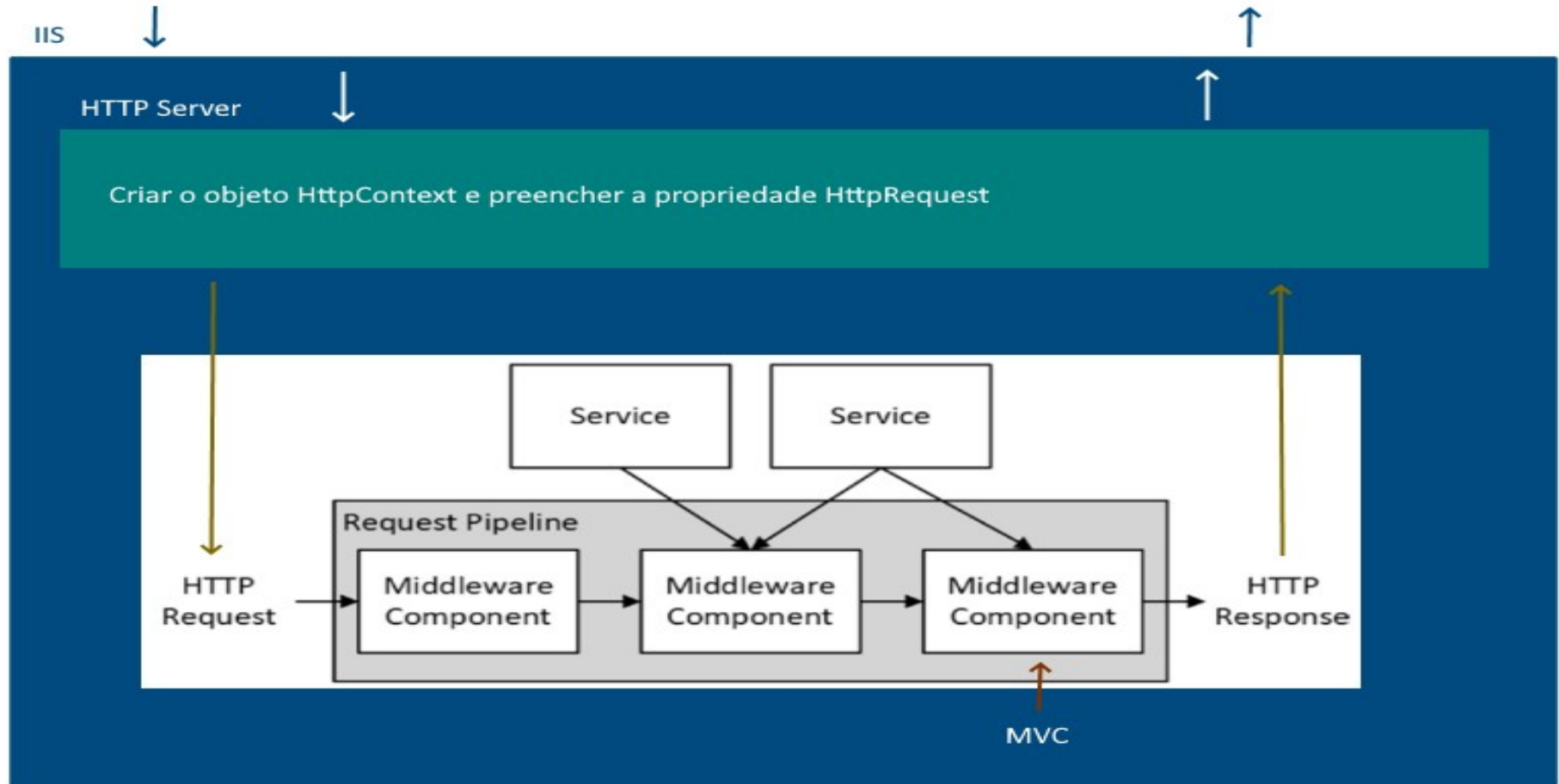
Entendendo o ASP.NET Core



Estrutura do ASP.NET Core



Ciclo de vida e uma requisição



Configuração de Services e Middlewares

```
public class Startup
{
    // References
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    // 1 reference
    public IConfiguration Configuration { get; }

    // This method gets called by the runtime. Use this method to add services to the container.
    // References
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers();
    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    // References
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

Services

Middlewares



Principais membros do HttpContext

Name	Description
Connection	This property returns a <code>ConnectionInfo</code> object that provides information about the network connection underlying the HTTP request, including details of local and remote IP addresses and ports.
Request	This property returns an <code>HttpRequest</code> object that describes the HTTP request being processed.
RequestServices	This property provides access to the services available for the request, as described in Chapter 14.
Response	This property returns an <code>HttpResponse</code> object that is used to create a response to the HTTP request.
Session	This property returns the session data associated with the request. The session data feature is described in Chapter 16.
User	This property returns details of the user associated with the request, as described in Chapters 37 and 38.
Features	This property provides access to request features, which allow access to the low-level aspects of request handling. See Chapter 16 for an example of using a request feature.



Principais membros do HttpRequest

Name	Description
Body	This property returns a stream that can be used to read the request body.
ContentLength	This property returns the value of the Content-Length header.
ContentType	This property returns the value of the Content-Type header.
Cookies	This property returns the request cookies.
Form	This property returns a representation of the request body as a form.
Headers	This property returns the request headers.
IsHttps	This property returns true if the request was made using HTTPS.
Method	This property returns the HTTP verb used for the request.
Path	This property returns the path section of the request URL.
Query	This property returns the query string section of the request URL as key/value pairs.

Principais membros do HttpResponseMessage

Name	Description
ContentLength	This property sets the value of the Content-Length header.
ContentType	This property sets the value of the Content-Type header.
Cookies	This property allows cookies to be associated with the request.
HasStarted	This property returns true if ASP.NET Core has started to send the response headers to the client, after which it is not possible to make changes.
Headers	This property allows the response headers to be set.
StatusCode	This property sets the status code for the response.
WriteAsync(data)	This asynchronous method writes a data string to the response body.
Redirect(url)	This method sends a redirection response.

Body



- CORS (Cross-Origin Resource Sharing)
- É um mecanismo para conceder ou restringir direitos de acesso para aplicações de diferentes domínios.

```
public static void ConfigureCors(this IServiceCollection services) =>
    services.AddCors(options =>
    {
        options.AddPolicy("CorsPolicy", builder =>
            builder.AllowAnyOrigin()
                .AllowAnyMethod()
                .AllowAnyHeader());
    });
```



Configuração do IIS

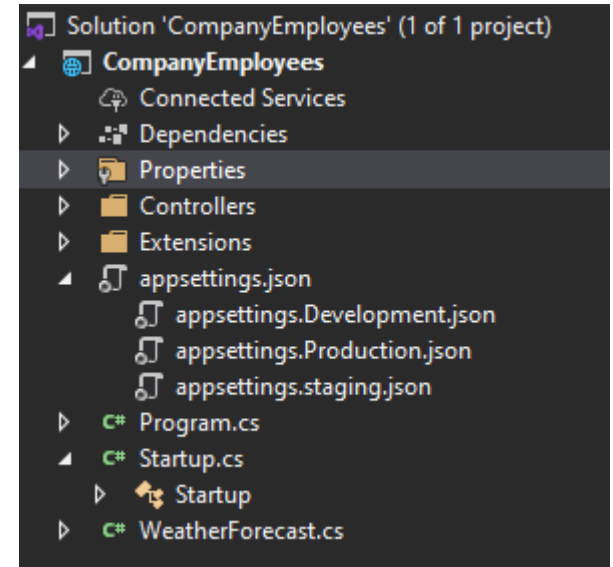
- Aplicações ASP.NET Core são auto-hospedados por padrão. Se quisermos hospedar a aplicação no IIS precisamos configuração a devida integração.

```
public static void ConfigureIISIntegration(this IServiceCollection services) =>  
    services.Configure<IISSOptions>(options =>  
        { });
```



Configuração de ambiente

- Para configurar em qual ambiente a aplicação roda, precisamos definir a variável de ambiente
`ASPNETCORE_ENVIRONMENT`
- No windows, comando **set**
`ASPNETCORE_ENVIRONMENT = Production`



Configuração de Log

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace CompanyEmployees.Contracts
{
    2 references
    public interface ILoggerManager
    {
        1 reference
        void LogInfo(string message);
        1 reference
        void LogWarn(string message);
        1 reference
        void LogDebug(string message);
        1 reference
        void LogError(string message);
    }
}
```

```
using CompanyEmployees.Contracts;
using NLog;

namespace CompanyEmployees.LoggerService
{
    2 references
    public class LogManager : ILoggerManager
    {
        private static ILogger logger = LogManager.GetCurrentClassLogger();
        0 references
        public LogManager()
        { }
        1 reference
        public void LogDebug(string message)
        {
            logger.Debug(message);
        }
        1 reference
        public void LogError(string message)
        {
            logger.Error(message);
        }
        1 reference
        public void LogInfo(string message)
        {
            logger.Info(message);
        }
        1 reference
        public void LogWarn(string message)
        {
            logger.Warn(message);
        }
    }
}
```

Regra básica dos Controllers

- Controllers devem ser responsáveis por receber requisições, validar o modelo de entrada e retornar uma resposta para o front-end ou algum cliente HTTP. Manter a lógica do negócio longe da controller é uma boa maneira de manter a controller leve, com leitura e manutenção mais fácil.
- Tem o formato padrão abaixo:

```
namespace CompanyEmployees.Controllers {  
    [Route("api/[controller]")]  
    [ApiController]  
    public class CompaniesController : ControllerBase  
    {  
    }  
}
```



Rotas em Web API

- Rotas em Web API direcionam requisições HTTP para métodos de ação dentro das controllers.

Navegador

GET /cervejaria/clientes HTTP/1.1

Host: localhost:8080

Accept: text/html



ASP.NET Core Platform

HttpContext e HttpContext.Request preenchido com dados vindo na requisição



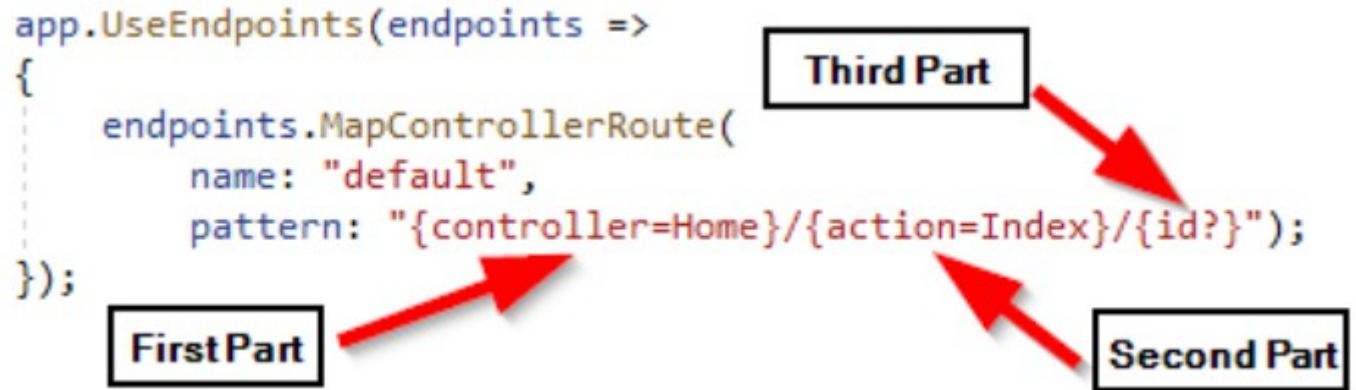
ASP.NET Core MVC

Análise do request. No primeiro casamento com alguma rota registrada, acionamento da Action na Controller correspondente.



Rotas em Web API

- Existem duas maneiras de implementar rotas:
 - Rotas baseadas em convenção
 - Rotas baseadas em atributos
- Rotas baseadas em convenção usam o caminho da URL para encontrar a action. A primeira parte do caminho identifica a Controller, a segunda a action e a terceira é um parâmetro opcional.



Rotas em Web API

- Rotas baseadas em atributos mapeiam as rotas diretamente para a action dentro da controller.
- Enquanto trabalhando com projetos Web API, o time do ASP.NET Core sugere que não se use rotas baseadas em convenção, mas rotas baseadas em atributo.
- O nome do recurso e a URI deve ser sempre um nome e não uma ação.

```
[Route("api/[controller]")]
[ApiController]
public class CompaniesController : ControllerBase
{
    [HttpGet]
    public async Task<IActionResult> GetCompanies()
    {
        var companies = await _repository.Company.GetAllCompaniesAsync(trackChanges: false);
        var companiesDto = _mapper.Map<IEnumerable<CompanyDto>>(companies);

        return Ok(companiesDto);
    }
}
```

URI:
get api/Companies

~~get api/getCompanies~~



Rotas em Web API

- Rotas entre recursos dependentes (entidades dependentes) seguem uma convenção um pouco diferente:
- `/api/principalResource/{principalId}/dependentResource`
 - `/api/companies/{companyId}/employees`



Classes DTO vs. Classe do Modelo de Entidades

- DTOs são objetos usados para transportar dados entre as aplicações cliente e servidor.
- Não é uma boa prática usar as entidades diretamente nas respostas da API. A papel dos DTOs são fornecer a resposta numa estrutura e formato que faça sentido para a aplicação, de acordo com o design planejado para a API.
- `CompanyEmployees\src\Entities\DataTransferObjects`



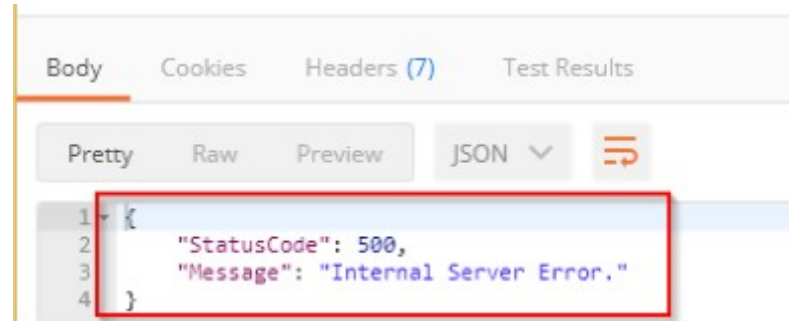
AutoMapper no ASP.NET Core

- Basicamente é uma ferramenta para transformar um objeto de um tipo em um objeto de outro tipo, baseado em convenções.
- Install-Package AutoMapper.Extensions.Microsoft.DependencyInjection
- services.AddAutoMapper(typeof(Startup));
- Documentação em
 - <https://docs.automapper.org/en/latest/Getting-started.html>
- CompanyEmployees\src\WebAPI\Mappings



Tratamento de erro global

- Será criado um middleware para tratamento global de erros, limpando as controllers de blocos try-catch.
- CompanyEmployees\src\WebAPI\Extensions\ExceptionMiddlewareExtensions.cs



The screenshot shows the 'Body' tab of a web browser's developer tools. The response is a JSON object indicating an internal server error. The JSON is formatted in 'Pretty' mode and is highlighted with a red rectangle.

```
1 {  
2   "StatusCode": 500,  
3   "Message": "Internal Server Error."  
4 }
```



Relação Pai/filho em Web API

```
[Route("api/companies/{companyId}/employees")]
```

```
[ApiController]
```

```
public class EmployeesController : ControllerBase
```

```
[HttpGet]
```

```
public async Task<IActionResult> GetEmployeesForCompany(Guid companyId)
```

```
[HttpGet("{id}")]
```

```
public IActionResult GetEmployeeForCompany(Guid companyId, Guid id)
```



Habilitando negociação de conteúdo

- Negociação de conteúdo é quando o cliente solicita o dado num determinado formato e o servidor o atende, quando possível. Por padrão as respostas são em JSON.

```
services.AddControllers(config => {  
    config.RespectBrowserAcceptHeader = true;  
}).AddXmlDataContractSerializerFormatters();
```



Criando recursos

[HttpPost]

```
public IActionResult CreateCompany([FromBody]CompanyForCreationDto company)
....
return CreatedAtRoute("CompanyById", new { id = companyToReturn.Id }, companyToReturn);
```

CreatedAtRoute retornará o código de estado 201 e irá popular o header Location com o endereço para recuperar o recurso.

Também é possível criar o recurso pai e seus filhos no mesmo request

```
public class CompanyForCreationDto {
    public string Name { get; set; }
    public string Address { get; set; }
    public string Country { get; set; }
    public IEnumerable<EmployeeForCreationDto> Employees { get; set; }
}
```



Excluindo e atualizando recurso

[HttpDelete("{id}")]

```
public IActionResult DeleteEmployeeForCompany(Guid companyId, Guid id)
```

[HttpPut("{id}")]

```
public IActionResult UpdateEmployeeForCompany(Guid companyId, Guid id,  
[FromBody] EmployeeForUpdateDto employee)
```

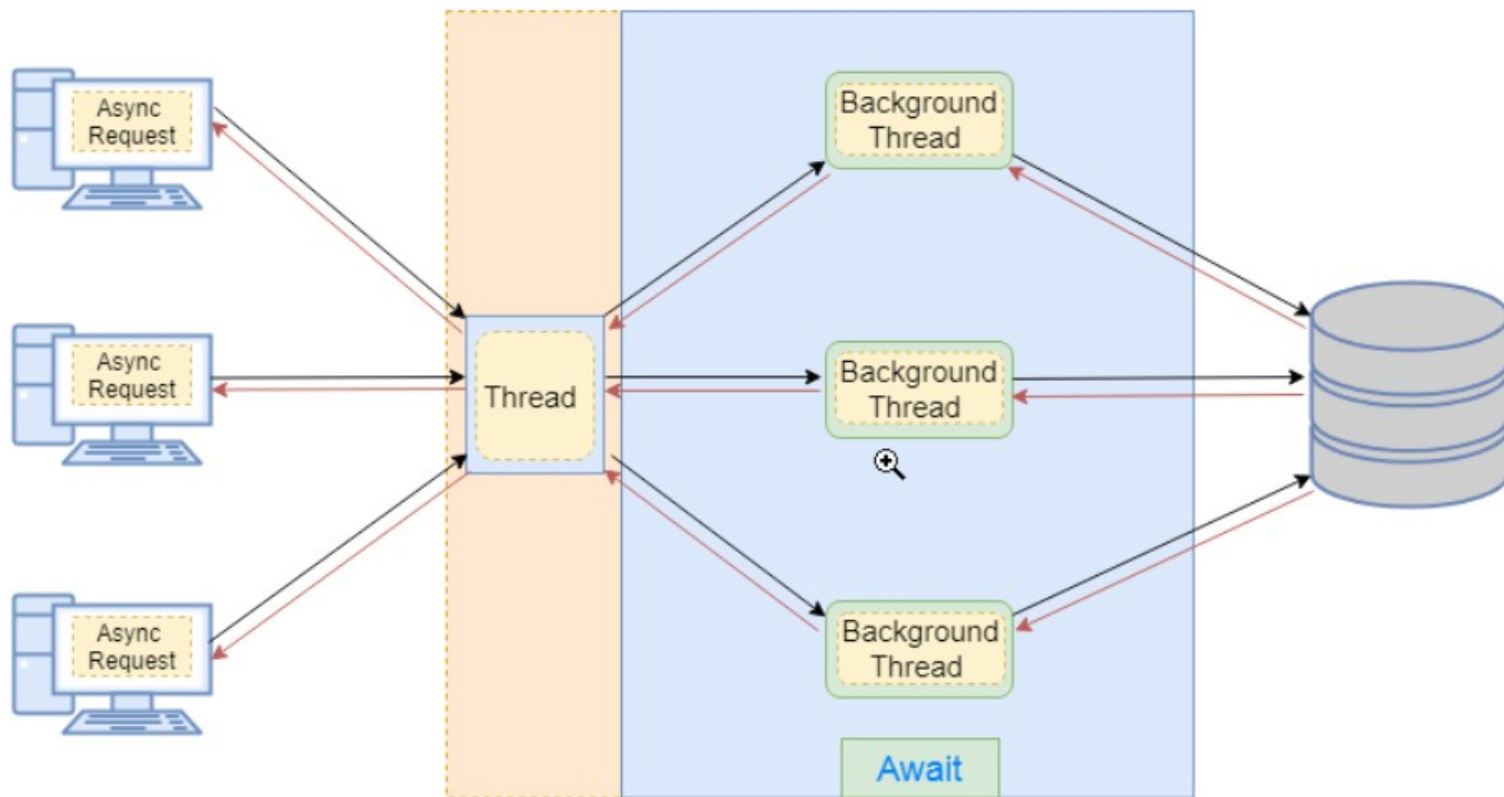


Validação

- É preciso validar as entradas (POST, PUT e PATCH). As saídas não precisam ser validadas (GET)
- O status code na resposta, quando a validação falha, deve ser 422 Unprocessable Entity
 - Para retornar 422 em vez de 400, é preciso evitar o erro BadRequest quando o ModelState está inválido.
 - `services.Configure<ApiBehaviorOptions>(options => {options.SuppressModelStateInvalidFilter = true;});`
- Data Annotation feitas em cima do DTOs.



Código Assíncrono



Código Assíncrono

- Tem **async** e **await** como palavras chaves
 - `async Task<IEnumerable<Company>> GetAllCompaniesAsync()`
- Os métodos assíncronos tem 3 tipos de retorno
 - `Task<Tresult>` para métodos que retornam valor
 - `Tast` para métodos que não retornam valor
 - `Void`, que podem ser usados para um Event Handler



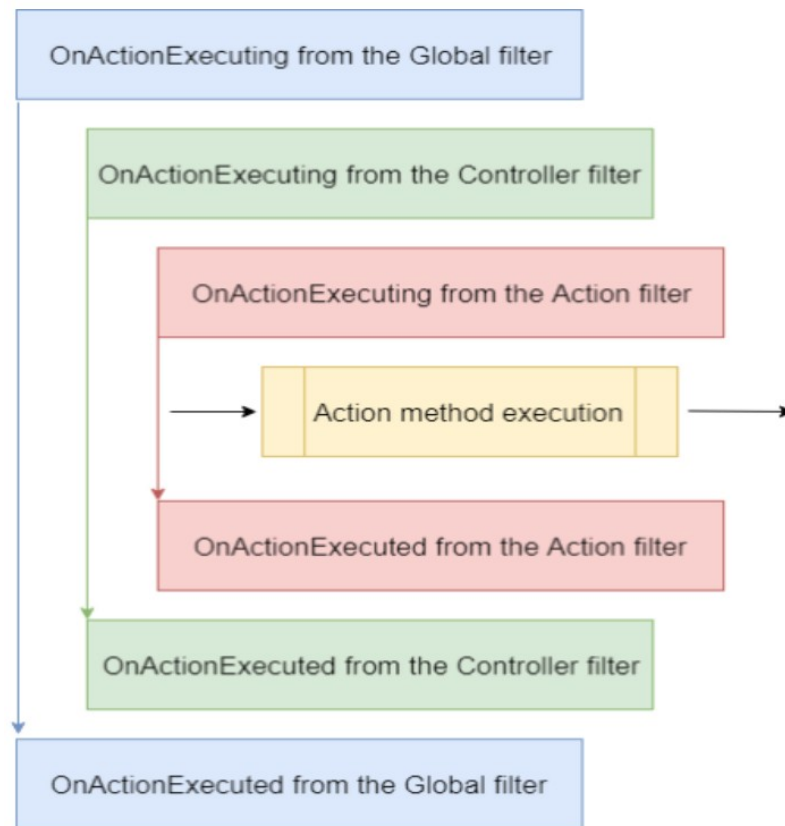
Action Filters

- Existem diferentes tipos de filtro
 - Authorization filters – rodam primeiro para determinar se o usuário está autorizado a fazer o request
 - Resource filters – roda em seguida e são uteis para cache e otimização
 - **Action filters – rodam antes e depois da execução de um método**
 - Exception filters – tratam erros antes do response ser populado
 - Result filters -



Action Filters

- Novos filtros são criados herdando de `IActionFilter` e `IAsyncActionFilter`
- O escopo pode ser global, na controller ou na action
- Segue a ordem de invocação ao lado
- Um uso comum é validação de entrada



Paginação

- Paginação é receber um resultado parcial de uma API.
- <https://localhost:5001/api/companies/companyId/employees?pageNumber=2&pageSize=2>.

[HttpGet]

```
public async Task<ActionResult> GetEmployeesForCompany(Guid  
companyId, [FromQuery]
```

```
EmployeeParameters employeeParameters)
```



Filtro

- Filtro nos ajuda a obter o resultado conforme algum critério pasado.

```
public class EmployeeParameters : RequestParameters
{
    public uint MinAge { get; set; }
    public uint MaxAge { get; set; } = int.MaxValue;
    public bool ValidAgeRange => MaxAge > MinAge;
}
```

