Dell IT Academy

Recursos

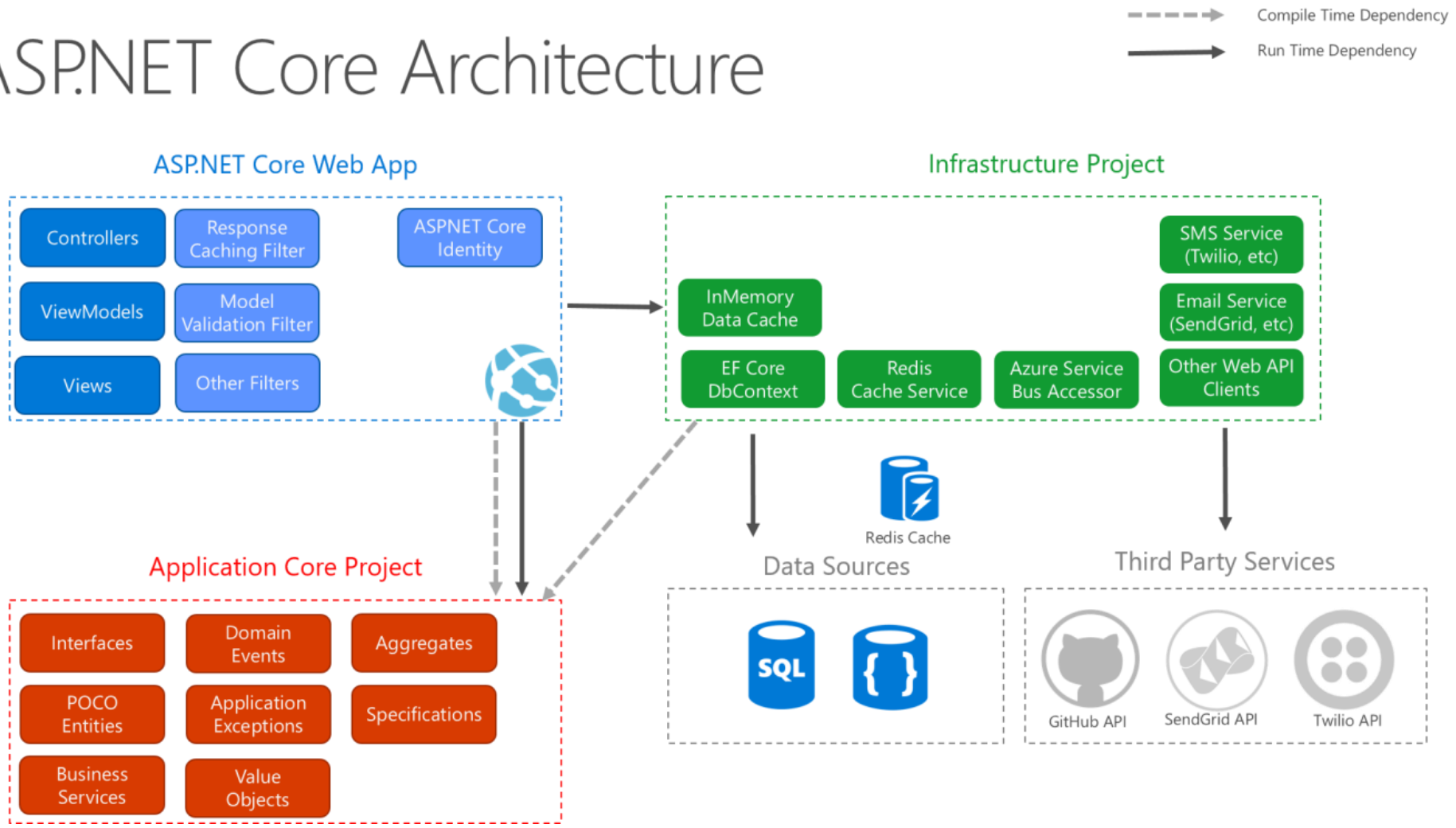
- Documentação
  - https://docs.microsoft.com/en-us/aspnet/core/
- Fontes
  - https://github.com/aspnet/home
- Exemplos
  - https://www.todobackend.com/
  - https://github.com/kkagill/ContosoUniversity-Backend

INTRODUÇÃO AO ASP.NET CORE

O que é?

- ASP.NET Core Web API é um dos frameworks .NET para o desenvolvimento de Web Services no estilo REST

ASP.NET Core Architecture



ASP.NET Core

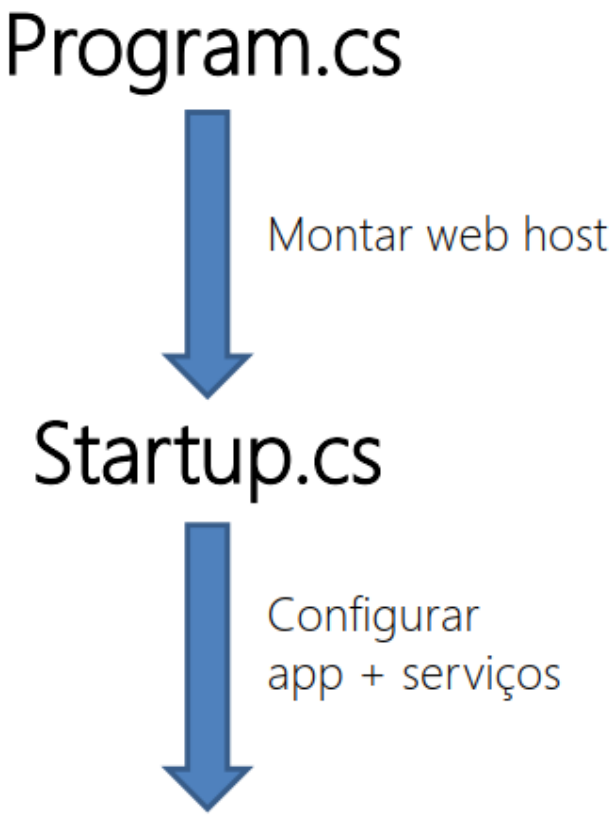
- Características gerais:
  - Framework unificado para IU Web e Serviços Web
  - Implementação de padrões de Injeção de Dependências
  - Implementação de padrões Model-View-Controller MVC
  - Pipeline configurável para processamento de requisições (Middleware)
  - Múltiplas opções de *hosts* (Kestrel, IIS, Nginx, Apache, Docker, processo do sistema operacional)
  - Suporte a configurações de ambientes
  - Multiplataforma (Windows, Linux, Mac)
  - Distribuído via NuGet

ASP.NET Core

- Passos essenciais:
  - Configuração do processo de *bootstrap* (em especial do *host*)
  - Configuração do sistema de injeção de dependências
  - Configuração do *pipeline* de *middleware*

ASP.NET Core - Bootstrap

- Aplicação de linha de comando
  - Ponto de início é *Program.Main()*
  - Configurar *WebHost*
- Configuração via código
  - Arquivo *Program.cs*
  - Arquivo *Startup.cs*
    - Método *ConfigureServices* para configurar injeção de dependências e outros serviços
    - Método *Configure* para configurar middleware
  - Arquivos JSON/XML
- Ver https://docs.microsoft.com/en-us/aspnet/core/fundamentals/startup



ASP.NET Core - Bootstrap

```
public class Startup
{
    // Use this method to add services to the container.
    public void ConfigureServices(IServiceCollection services)
    {
        ...
    }

    // Use this method to configure the HTTP request pipeline.
    public void Configure(IApplicationBuilder app)
    {
        ...
    }
}
```

ASP.NET Core - Servidores

- Servidores suportados
  - IIS – Internet Information Services https://www.iis.net/
  - Servidores integrados
    - Kestrel (multiplataforma, opção padrão dos templates VS)
    - HTTP.sys (Windows)
    - etc
  - Servidores de terceiros
    - Apache http://httpd.apache.org/
    - Nginx http://nginx.org/
    - etc
- Ver https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/

ASP.NET Core – Injeção Dependências

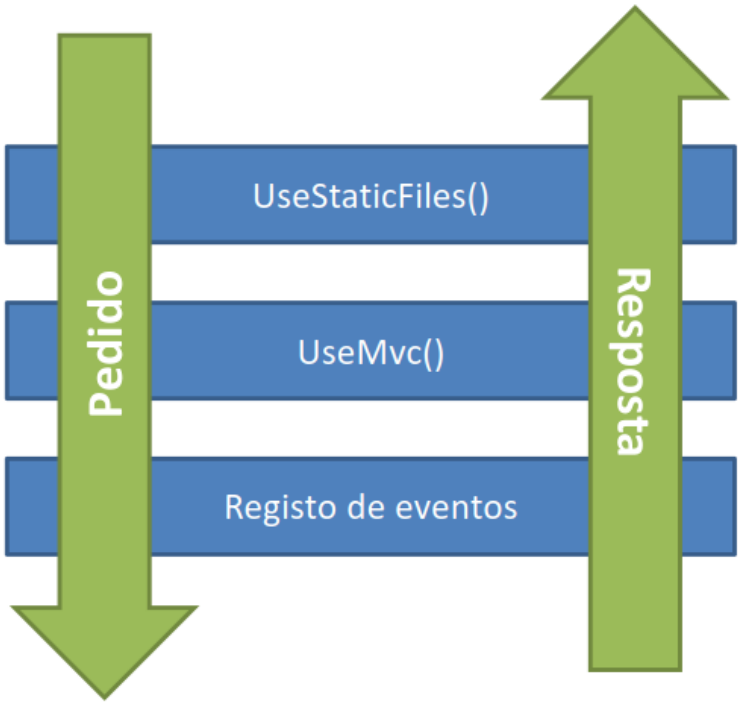
- Um serviço é um componente que oferece uma funcionalidade
  - ASP.NET Core MVC, Entity Framework Core, Identity Framework, Logging, etc
- Serviços se tornam disponíveis via Injeção de Dependências
- ASP.NET Core possui um contêiner gerenciador de Inversão de Controle
  - Suporta injeção de dependência via construtor e em ações de controladores
  - Suporta gerenciamento de escopo
    - Transient, Scoped, Singleton
- Ver https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

ASP.NET Core – Injeção Dependências

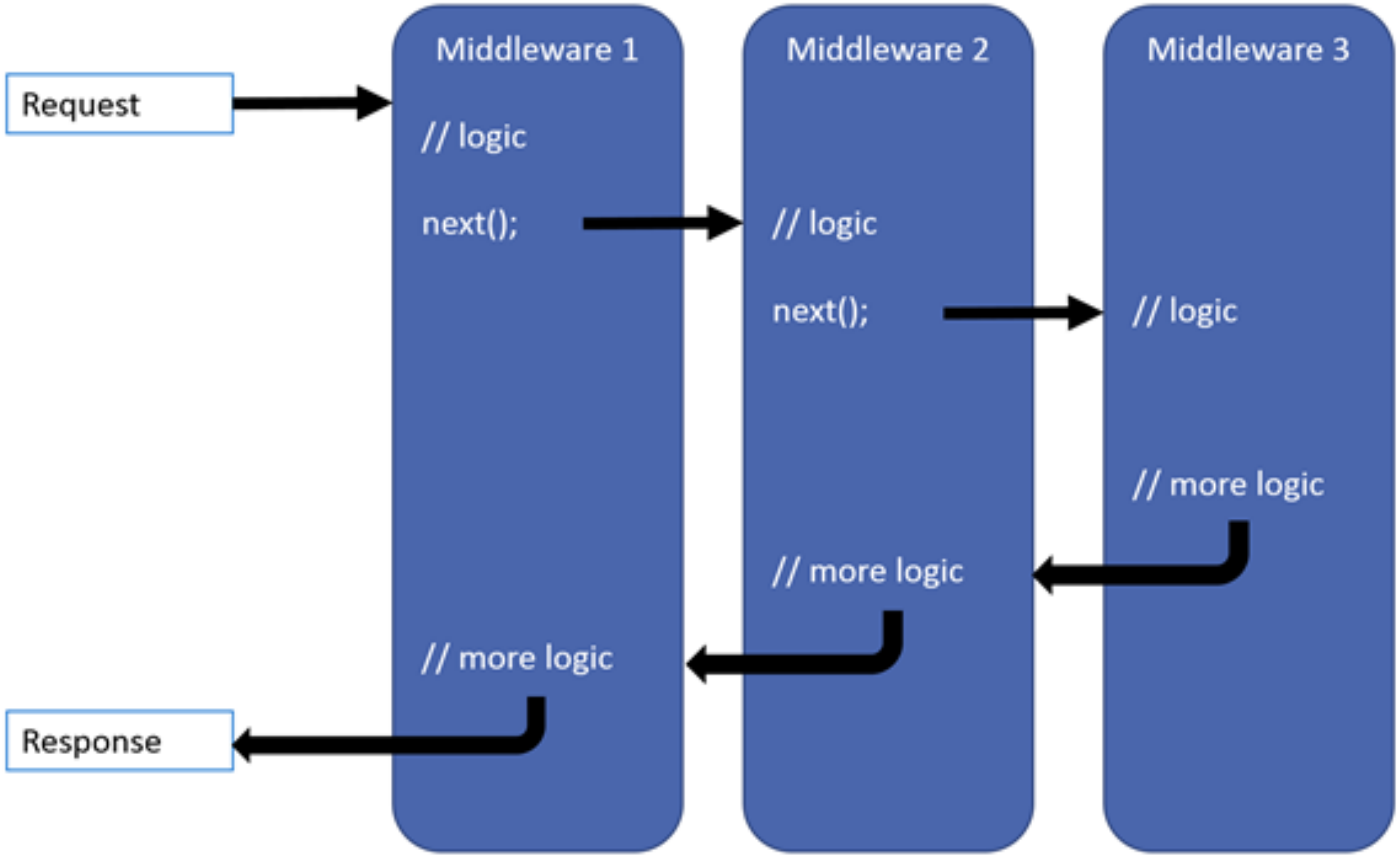
```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<ApplicationDbContext>(options =>
        options.UseInMemoryDatabase()
    );

    // Add framework services.
    services.AddMvc();

    // Register application services.
    services.AddScoped<ICharacterRepository, CharacterRepository>();
    services.AddTransient<IOperationTransient, Operation>();
    services.AddScoped<IOperationScoped, Operation>();
    services.AddSingleton<IOperationSingleton, Operation>();
    services.AddSingleton<IOperationSingletonInstance>(new
Operation(Guid.Empty));
    services.AddTransient<OperationService, OperationService>();
}
```

ASP.NET Core - Middleware

- ASP.NET Core utiliza *pipeline* para o processamento de requisições
  - Arquivos estáticos, roteamento, autenticação, CORS, caching, sessão, etc
- *Middleware* lê e escreve diretamente no *pipeline*
- Ver https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/



ASP.NET Core - Middleware



ASP.NET Core - Middleware

```
public void Configure(IApplicationBuilder app)
{
    app.UseExceptionHandler("/Home/Error");

    app.UseStaticFiles();

    app.UseAuthentication();

    app.UseMvcWithDefaultRoute();
}
```

CONTROLADOR

Controlador e Ações

- Um controlador (uma classe) é utilizado para definir um grupo de ações
- Uma ação (um método) é utilizado para tratar uma requisição
  - Todos métodos públicos definem uma ação por padrão
- Requisições são mapeadas para controladores e ações via roteamento
- IMPORTANTE: controlador é uma abstração de nível de camada de interface, evitar ao máximo implementar lógica de negócio em um método de ação

Regras de Roteamento

- O Web Service necessita mapear as requisições HTTP ao código que irá trata-las
  - URI de requisição
  - Método (verbo) do HTTP
- Regras de roteamento utilizam URIs diretamente
  - Sem a necessidade de arquivos com extensões específicas (.aspx, .asmx, .svc)
- ASP.NET Core MVC possui um *middleware* de gerenciamento de rotas
- Ver:
  - https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing
  - https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing

Regras de Roteamento

- Regras criadas utilizando tabela de roteamento

```
app.UseMvc(routes =>
{
    routes.MapRoute("default",
     "{controller=Home}/{action=Index}/{id?}");
});
```

- Regras criadas através de atributos

```
[Route("customers/{customerId}/orders")]
public IEnumerable<Order> GetOrdersByCustomer(int
customerId) { ... }
```

Roteamento - Métodos de Ação

- Ações são métodos públicos do *controller* que respondem a requisições do HTTP
- Ações são mapeadas com base em:
  - O método HTTP utilizado na requisição
    - Ex.: GET, PUT, POST, DELETE, etc
  - Regras de mapeamento de ação no roteamento

Roteamento - Métodos de Ação

```
public class ProductsApiController : ControllerBase
{
    [HttpGet("/products/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id) { ... }
}
```

```
[Route("products")]
public class ProductsApiController : ControllerBase
{
    [HttpGet]
    public IActionResult ListProducts() { ... }

    [HttpGet("{id}")]
    public ActionResult GetProduct(int id) { ... }
}
```

Roteamento - Métodos de Ação

- Rotas suportam *tokens de substituição*:
  - [action], [area], [controller]

```
[Route("[controller]/[action]")]
public class ProductsController : ControllerBase
{
    [HttpGet] // Matches '/Products/List'
    public IActionResult List() {
        // ...
    }

    [HttpGet("{id}")] // Matches '/Products/Edit/{id}'
    public IActionResult Edit(int id) {
        // ...
    }
}
```

Roteamento - Métodos de Ação

- É possível atribuir restrições sobre as regras de mapeamento dos parâmetros
- Formato geral *{parâmetro:restrição}*

```
[HttpPost("product/{id:int}")]
public IActionResult ShowProduct(int id)
{
    // ...
}
```

- Ver https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing#route-template-reference

AÇÕES

Parâmetros

- ASP.NET Core mapeia dados de uma requisição HTTP em parâmetros de métodos de ação
- É um processo conhecido como *model binding*
- Busca por valores segue uma ordem:
  - Valores de formulário (corpo da requisição POST)
  - Valores em templates de roteameto
  - Valores de *query string* (requisição GET)
- Ação deve verificar status explicitamente:
  - Propriedade *ModelState.IsValid*
- Ver https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding

Parâmetros

- Diversos atributos controlam a vinculação de dados a partir da requisição HTTP:
  - [BindRequired] – resulta em status de erro se a vinculação não ocorrer
  - [BindNever] – vinculação não deve ocorrer
  - [FromHeader], [FromQuery], [FromRoute], [FromForm], [FromBody] – indicam a fonte específica da vinculação
  - [FromServices] – indica vinculação via o serviço de injeção de dependências

Parâmetros

```
[HttpPost]
public IActionResult Create([FromBody] TodoItem item)
{
    if (item == null)
    {
        return BadRequest();
    }

    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id },
item);
}
```

Resultados

- ASP.NET Core converte o valor de retorno de métodos de ação em mensagem de retorno do HTTP
- Tipos de retorno:
  - tipo específico
  - *IActionResult*
  - *ActionResult<T>*
- Ver https://docs.microsoft.com/en-us/aspnet/core/web-api/action-return-types

Resultados

```
[HttpGet]
public IEnumerable<Product> Get()
{
    return _repository.GetProducts();
}
```

Resultados

- *IActionResult* é uma interface que representa um *factory* de objetos *ActionResult* o qual representa diferentes respostas do HTTP
  - *BadRequestResult*, *NotFoundResult*, *OkObjectResult*, etc
- Vantagens:
  - Separação do código do controlador do código de criação das respostas do HTTP
  - Encapsula detalhes de baixo nível da criação de mensagens de resposta
  - Facilita o teste unitário dos controladores

Resultados

```
[HttpGet("{id}")]
[ProducesResponseType(200, Type = typeof(Product))]
[ProducesResponseType(404)]
public IActionResult GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return Ok(product);
}
```

Ações Assíncronas

- Utilizam uma combinação das palavras reservadas *async*, *await* e o tipo *Task*

```
[HttpPost]
[ProducesResponseType(201, Type = typeof(Product))]
[ProducesResponseType(400)]
public async Task<IActionResult> CreateAsync([FromBody]
Product product)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    await _repository.AddProductAsync(product);

    return CreatedAtAction(nameof(GetById), new { id =
product.Id }, product);
}
```

Resultados

- *ActionResult<T>* está disponível somente a partir da versão 2.1
- Vantagens:
  - Metadados de descrição de resultado podem ser excluídos
  - Conversão implícita para o tipo de retorno correto

Resultados

```
[HttpGet("{id}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public ActionResult<Product> GetById(int id)
{
    if (!_repository.TryGetProduct(id, out var product))
    {
        return NotFound();
    }

    return product;
}
```

SERIALIZAÇÃO

Serialização

- ASP.NET Core MVC tem suporte padrão para os seguinte tipos de dados na serialização:
  - JSON – classe *JsonResult* e factory *Json*
  - XML – deve ser configurado explicitamente como *middleware*
  - texto – classe *ContentResult* e factory *Content*
- Tipos adicionais podem ser configurados através de formatadores customizados
- Ver:
  - https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/formatting
  - https://docs.microsoft.com/en-us/aspnet/core/web-api/advanced/custom-formatters

Serialização

```
// GET: api/authors
[HttpGet]
public JsonResult Get()
{
    return Json(_authorRepository.List());
}
```

```
// GET api/authors/about
[HttpGet("About")]
public ContentResult About()
{
    return Content("An API listing authors of docs.asp.net.");
}
```

Serialização

- ASP.NET Core MVC suporta o processo de negociação de conteúdo do protocolo HTTP
  - Cabeçalho *Accept* define o tipo de conteúdo solicitado
  - Se não for solicitado, padrão do framework é JSON
- Para forçar um tipo específico de formatador na serialização utiliza-se o atributo *[Produces]*
  - Aplicado ao controlador, ação ou escopo global

```
[Produces("application/json")]
public class AuthorsController/
```