

ASP.NET Core 3.0

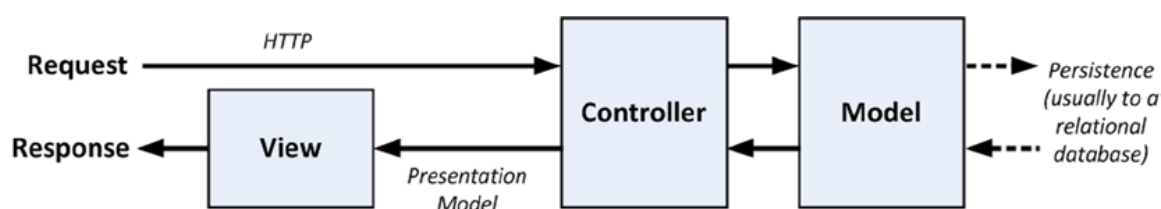
Pattern-ul Model-View-Controller (MVC) separa o aplicatie in trei componente principale:

- ✓ **M: model (Model).**
- ✓ **V: vizualizare (View User Interface).**
- ✓ **C: controler (Controller).**

Observatie

Ceea ce am definit mai sus constituie componentele pattern-ului MVC pentru UI – *User Interface*. Pattern-ul MVC nu spune nimic despre modul cum accesam datele, cum interactioneaza serviciile, etc.

Interactiunea intr-o aplicatie ASP.NET (Core) MVC poate fi reprezentata astfel:



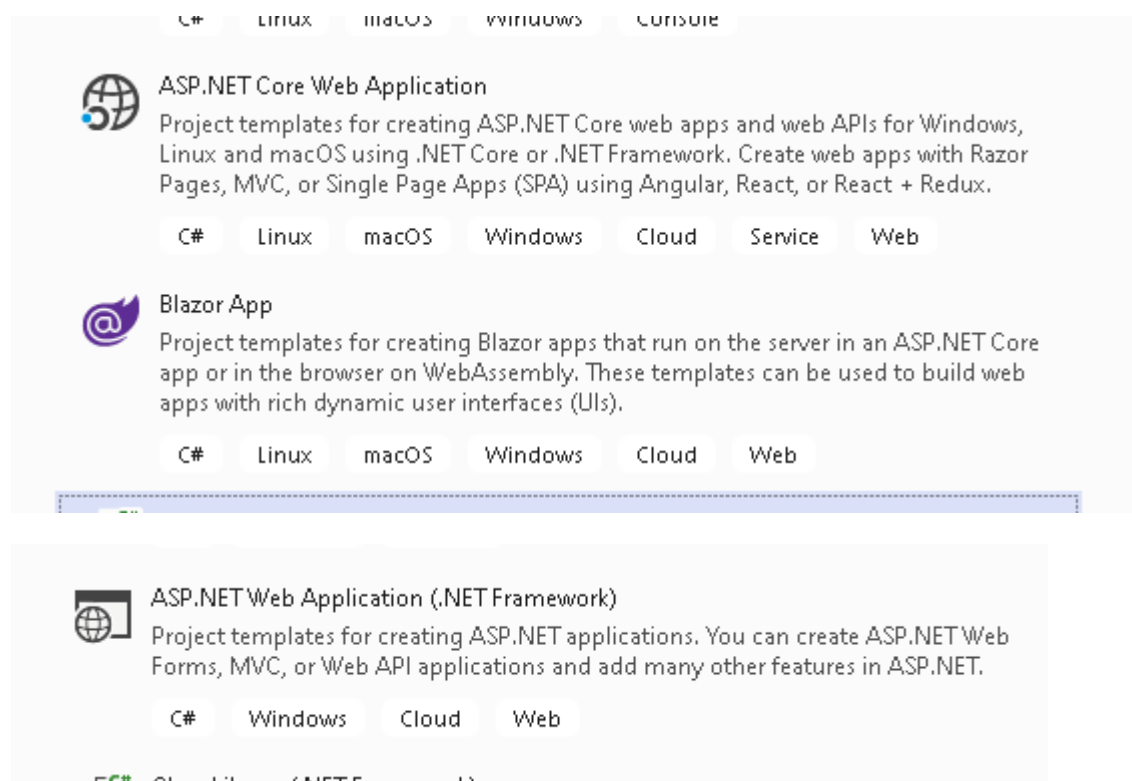
MVC aplicat la framework-ul Web

In ASP.NET MVC, MVC este translatat astfel:

- ✓ **Model.** *Model* contine clase ce reprezinta domeniul aplicatiei. Aceste obiecte incapsuleaza adesea date memorate intr-o baza de date precum si cod folosit pentru a procesa datele si a executa actiuni specifice logicii aplicatiei. Cu ASP.NET MVC, acesta este vazut mai ales ca un *Data Acces Layer – DAL* – de un anumit tip, utilizand de exemplu Entity Framework sau NHibernate combinat cu cod specific logicii aplicatiei.
- ✓ **View.** *View* defineste cum va arata interfata aplicatiei. *View* este un template pentru a genera in mod dinamic HTML.
- ✓ **Controller.** *Controller* este o clasa speciala ce gestioneaza relatiile dintre *View* si *Model*. *Controller*-ul raspunde la actiunile utilizatorului, comunica cu modelul si decide ce vizualizare va afisa (daca exista una). In ASP.NET MVC, numele acestei clase contine sufixul *Controller*.

Proiecte pentru Asp.Net Core in Visual Studio 2019

Visual Studio 2019 propune urmatoarele sabloane pentru proiecte ASP.NET Core:




Pentru un proiect **ASP.NET Core Web Application** se poate alege din urmatoarele tipuri de aplicatii:


Create a new ASP.NET Core web application

.NET Core


ASP.NET Core 3.1

 **Empty**


An empty project template for creating an ASP.NET Core application. This template does not have any content in it.

 **API**


A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

 **Web Application**


A project template for creating an ASP.NET Core application with example ASP.NET Razor Pages content.

 **Web Application (Model-View-Controller)**

A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.

 **Angular**

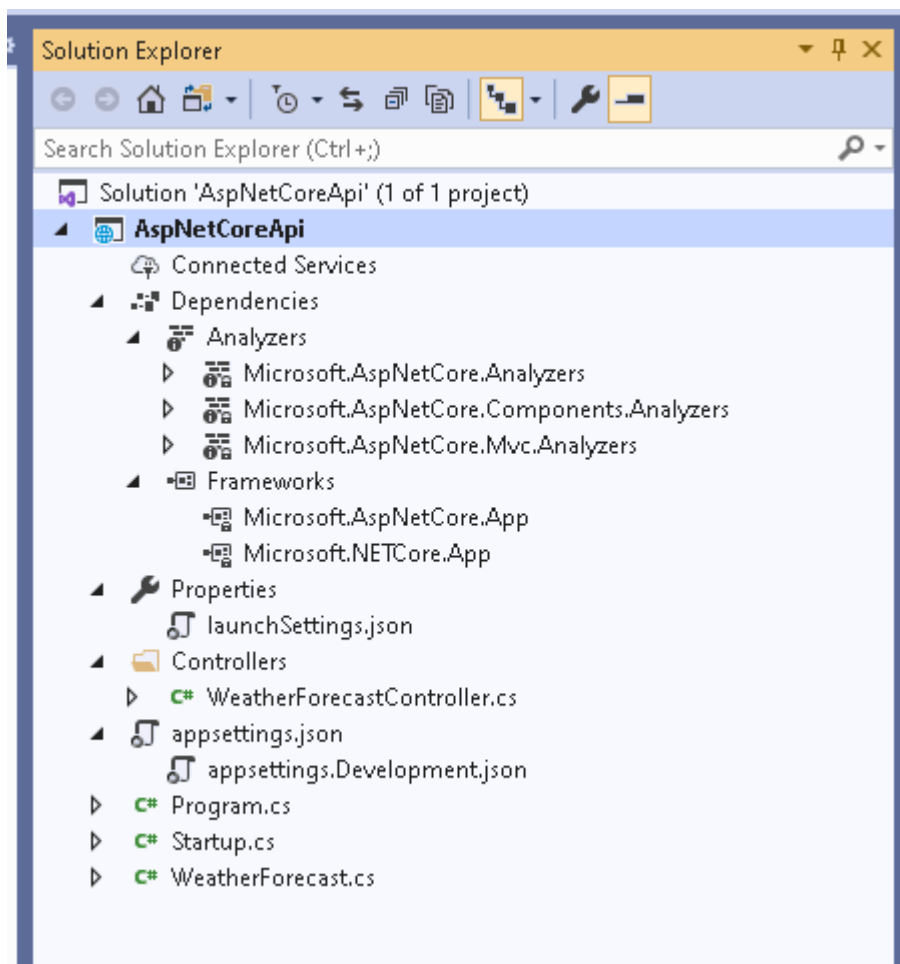
A project template for creating an ASP.NET Core application with Angular

 **React.js**

A project template for creating an ASP.NET Core application with React.js

[Get additional project templates](#)

Pentru un proiect App ASP.NET Core - **API** – VS 2019 genereaza urmatoarea structura:



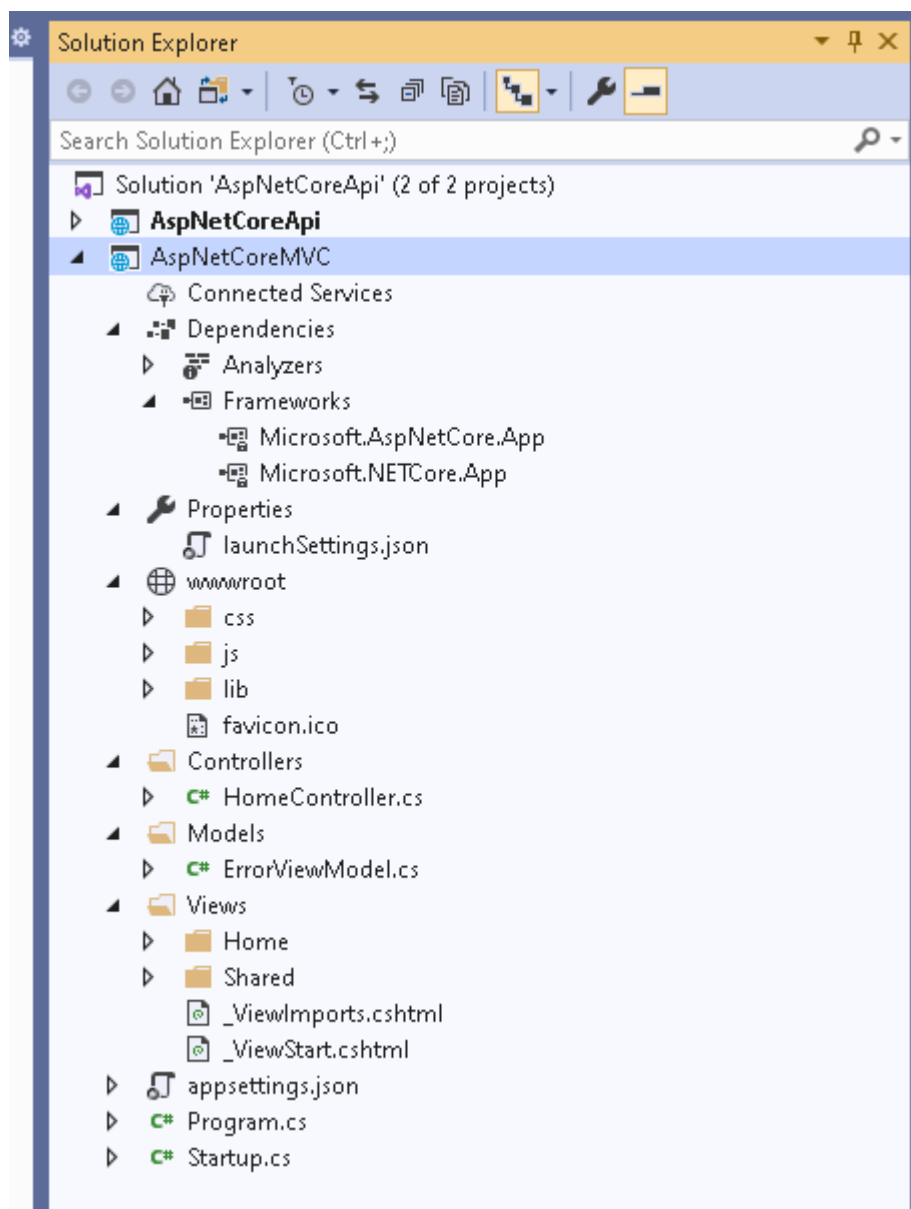
Observam aici:

- Nodul Frameworks din Dependencies ce contine **Microsoft.AspNetCore.App** si **Microsoft.NetCore.App**.
- Doua fisiere cu extensia json: **launchsettings.json** si **appsettings.json**, fisiere ce contin informatii ce vor fi folosite la configurarea aplicatiei si la lansarea acesteia.
- **Program.cs** – fisier ce contine codul pentru lansarea aplicatiei, entry point.
- **Startup.cs** – fisier, apelat de framework. Contine doua metode importante: **ConfigureServices** si **Configure**. Metoda **ConfigureServices** realizeaza configurarea serviciilor, iar metoda **Configure** este responsabila pentru definirea pipeline ce trateaza cererile, o serie de componente middleware.

Observatie

In structura proiectului de mai sus nu sunt create folderele tipice pentru MVC.

Daca selectam tip aplicatie **Web Application (Model-View-Controller)** atunci VS genereaza:



Fata de primul sablon API aici apar in plus folderele:

- wwwroot
- Controllers
- Models
- Views

Observatie

Metodele *ConfigureServices* si *Configure* contin mai mult cod specific pentru MVC.

Observatie

Proiectul API poate folosi MVC. Va trebui sa scriem cod in *Startup.cs*, sa cream structura de directoare specifica MVC,etc.

Metodele *ConfigureServices* si *Configure* din cele doua proiecte.

Startup.cs – Net Core Web (MVC)	Startup.cs – Net Core API
<pre> public void ConfigureServices (IServiceCollection services) { services.AddControllersWithViews(); } </pre>	<pre> public void ConfigureServices (IServiceCollection services) { services.AddControllers(); } </pre>
<pre> public void Configure(IApplicationBuilder app, IWebHostEnvironment env) { if (env.IsDevelopment()){ app.UseDeveloperExceptionPage(); } else { app.UseExceptionHandler("/Home/Error"); app.UseHsts(); } app.UseHttpsRedirection(); app.UseStaticFiles(); app.UseRouting(); app.UseAuthorization(); app.UseEndpoints(endpoints => { endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}"); }); } </pre>	<pre> public void Configure(IApplicationBuilder app, IWebHostEnvironment env) { if (env.IsDevelopment()) { app.UseDeveloperExceptionPage(); } app.UseHttpsRedirection(); app.UseRouting(); app.UseAuthorization(); app.UseEndpoints(endpoints => { endpoints.MapControllers(); }); } </pre>

Observatie

1. Din tabelul de mai sus observam ca VS insereaza acelasi cod pentru *ConfigureServices*. Metoda *Configure* contine un middleware in plus in MVC fata de API. Vedeti ce e in bold in coloana din stanga.
2. Implementare MVC in proiectul Net Core API nu este dificila.
3. Codul din Program.cs este identic in cele doua proiecte.

ASP.NET Core – Fundamente

Ne vom indrepta atentia asupra urmatoarelor notiuni necesare pentru a intelege cum trebuie sa dezvoltam o aplicatie web:

- Startup class
- Dependency injection (services)
- Middleware
- Host
- Configuration

- Environments
- Logging
- Routing
- Error handling
- Make HTTP request
- Content root
- Web root

In cele ce urmeaza vom descrie (pe scurt) fiecare notiune in parte precum si rolul acesteia in cadrul unei aplicatii web.

Clasa Startup

In cadrul acestei clase:

- Se configureaza serviciile cerute de aplicatie.
- Se defineste pipeline ce va trata cererile.

Serviciile sunt componente ce sunt utilizate de app. De exemplu, componenta pentru jurnalizare (logging) este un serviciu.

Codul necesar este adaugat in metoda *ConfigureServices* din aceasta clasa.

Pipeline pentru manipularea cererilor este compus din o serie de componente *middleware*. Fiecare *middleware* executa operatii asincrone pe un **HttpContext** si atunci fie invoca urmatoarea componenta *middleware*, fie termina executia cererii. Codul pentru configurare pipeline tratare cerere se scrie in metoda *Configure* din aceasta clasa.

Observatie

1. Ordinea componentelor *middleware* din cadrul metodei *Configure* are importanta. Componenta *middleware* cea mai generala va fi plasata ultima in cadrul metodei *Configure*.
2. Prima componenta *middleware* ce poate trata o cerere va fi luata in considerare si e posibil ca tratarea cererii sa se opreasca aici.

Exemplu configurare servicii si pipeline

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc()
            .SetCompatibilityVersion(
                CompatibilityVersion.Version_3_0);
        services.AddDbContext<MovieContext>(options =>
```

```

        options.UseSqlServer(Configuration.GetConnectionString(
            "MovieDb"));
    }

    public void Configure(IApplicationBuilder app)
    {
        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseMvc();
    }
}

```

In exemplul de mai sus se configureaza serviciile pentru MVC si Entity Framework.

Metoda **AddMvc()** configureaza serviciile **MVC** pentru app.

Configurarea pipeline pentru tratare cereri este data de urmatorul cod (metoda Configure):

```

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseMvc();

```

Sunt trei componente middleware.

- *UseHttpsRedirection()* : redirectare cereri HTTP catre HTTPS.
- *UseStaticFiles()*: serverul poate trata cereri pentru fisiere.
- *UseMvc()*: componenta middleware pentru MVC.

In .NET Core 2.x inregistrarea serviciilor pentru MVC putea fi facuta in doua moduri:

- *services.AddMvc()*
- *services.AddMvcCore()*

In ASP.NET Core 3.x, pe langa aceste doua modalitati s-au mai adaugat urmatoarele:

- *services.AddControllers()*
- *services.AddControllersWithViews()*
- *services.AddRazorPages()*

Observatie

Inregistrarea serviciului *AddMvc()* este echivalenta cu inregistrarea si a celorlalte servicii enumerate mai sus. Mai multe informatii gasiti la adresa: <https://www.strathweb.com/2020/02/asp-net-core-mvc-3-x-addmvc-addmvccore-addcontrollers-and-other-bootstrapping-approaches/> sau in textul ce urmeaza (copie de la adresa indicata).

ASP.NET Core MVC 3.x – AddMvc(), AddMvcCore(), AddControllers() and other bootstrapping approaches

Sursa: <https://www.strathweb.com/2020/02/asp-net-core-mvc-3-x-addmvc-addmvccore-addcontrollers-and-other-bootstrapping-approaches/>

AddMvcCore()

AddMvcCore() registers all the core services required for the MVC application to work at all. We do not need to list them all, but pretty much everything related to the controller invocation pipeline

gets activated there. These are low(er) level services, that only get customized when you are doing something quite complex or unusual (i.e. building a CMS). Some examples of them are: the controller activation services, the MVC options pipeline, application model provider infrastructure, action constraints, filter pipeline, model binder infrastructure, action result executors and a few more.

At the same time, the initialized framework configuration is completely “bare bones”. It is functional from the perspective of being able to handle an incoming HTTP call, but it is missing several core features. For example, the model validation via data annotations is not activated, same with authorization.

In this set-up, you are in control of (or, if you will, you are responsible for) what is plugged in and used at runtime. In other words, if you need anything beyond the most basic framework feature, you have to add it manually. In fact, in .NET Core 2.x and earlier, not even JSON support was there; this has now changed and the *System.Text.Json* formatter is actually already included in the call to *AddMvcCore()*.

For example:

```
// pick what you need
services.AddMvcCore()
    .AddDataAnnotations() // for model validation
    .AddApiExplorer();    // for Swagger
```

This should be the default choice for you if you really like to bootstrap the minimal amount of things at runtime and only activate the individual features you really use.

AddControllers()

AddControllers() was introduced in ASP.NET Core 3.0 as a mechanism that would simplify the manual setup needed together with calling the lightweight *AddMvcCore()*.

What you get with *AddControllers()* is:

- everything that *AddMvcCore()* does
- authorization services – needed for authorization policies, filters and other authorization components to work
- API explorer – required if you want to build dynamic API documentation, generate Swagger/OpenAPI files
- data annotations – needed for model validation with attributes and *IValidateableObject* to work
- formatter mappings – needed for content negotiation to work
- CORS

In other words, what you can expect from *AddControllers()* is that it would give you the most comfortable setup for API development. None of the view services are registered here so you don't “drag” any of the Razor related baggage with you. What's worth noting is that the name itself – *AddControllers()* – sort of blurs the line between the ASP.NET Core and the MVC framework, as it doesn't really tell you at first glance that you are activating the MVC framework.

This should be the default choice for you if you are developing an API and want to quickly and reliably bootstrap the framework.

```
// ready for API development
services.AddControllers();
```

AddControllersWithViews()

AddControllersWithViews() is the one you should pick if you are building a “classic” MVC site, just like we have been doing it for years – with controllers and Razor views. It will end up activating:

- everything that *AddControllers()* does
- views functionality – explicitly registers the Razor view engine
- cache tag helper

This should be the default choice for you if you do not need the new Razor pages functionality – you are either building the MVC website exactly how it was built in old desktop framework MVC and in earlier versions of ASP.NET Core MVC or if you are migrating an older site.

```
// ready for "classic" MVC website development
// and at the same time ready for API development
services.AddControllersWithViews();
```

AddRazorPages()

AddRazorPages() is intended to serve as a bootstrapping helper for working with the new [Razor Pages](#) feature. Under the hood, it ends up activating the following:

- all the core Razor pages features
- everything that *AddMvcCore()* does – this is a bit surprising at first glance
- authorization services – needed for authorization policies, filters and other authorization components to work
- data annotations – needed for model validation to work
- cache tag helper

The fact that it ends up activating *AddMvcCore()* is an internal implementation detail, since the Razor Pages pipeline is relying on a lot of the core MVC infrastructure under the hood. As a side effect, it means that when calling *AddRazorPages()* you are sort of ready to do API endpoints too. This may change in the future, and therefore I wouldn't take strategic decisions based on that. In other words, even though we could now say:

```
// ready for Razor Pages development
// and at the same time quite ready for API development
services.AddRazorPages();
```

if you ever need to host an API and Razor Pages in same project, I'd rather recommend to make these activations explicit, so that you don't get surprised in the future when something changes internally:

```
// ready for Razor Pages development
// ready for API development
services.AddRazorPages().AddControllers();
```

Of course the *AddRazorPages()* should be your default choice if you plan to work with Razor Pages.

AddMvc()

Finally, we have *AddMvc()*, which simply registers the entire kitchen sink of all the features. It gives you:

- everything that *AddControllersWithViews()* does
- everything that *AddRazorPages()* does

```
// ready for everything
services.AddMvc()
```

While I'd imagine you know what you are trying to build – if you ever have any doubts in which direction your project will evolve, or if you are afraid that some MVC feature would be missing (or in fact, if you already ran into a missing feature), calling *AddMvc()* would be the safest bet to resolve any of those worries or issues.

Dependency injection (services)

ASP.NET Core are un framework preconstruit pentru dependency injection (DI), folosit pentru configurarea serviciilor disponibile claselor unei app.

O metoda de a obtine o instanta a unui serviciu intr-o clasa este aceea de a injecta obiectul prin constructorul clasei, adica constructorul unei clase are ca parametru tipul cerut. Parametrul poate fi tipul serviciului sau o interfata. Sistemul DI va furniza serviciul la runtime.

Exemplu. O clasa ce foloseste DI pentru a obtine un obiect *RazorPagesMovieContext*:

```
public class IndexModel : PageModel
{
    private readonly RazorPagesMovieContext _context;
    public IndexModel(RazorPagesMovieContext context)
    {
        _context = context;
    }
    // ...
    public async Task OnGetAsync()
    {
        Movies = await _context.Movies.ToListAsync();
    }
}
```

DI rezolva problema instantierii claselor astfel:

- Utilizarea unei interfete sau a unei clase de baza pentru a abstractiza DI.
- Inregistrarea dependentei intr-un container de servicii. Vezi mai jos Service lifetimes.
- *Injectia* serviciului in constructorul clasei unde va fi utilizat. Framework-ul este responsabil de crearea instantei clasei si eliminarea acestei instante atunci cand nu mai e necesara.

Service lifetimes

Serviciile ASP.NET Core pot fi configurate cu urmatoarele "lifetimes":

Transient

Transient lifetime services ([AddTransient](#)) are created ***each time they're requested*** from the service container. This lifetime works best for lightweight, stateless services.

[request != requested]

Scoped

Scoped lifetime services ([AddScoped](#)) are created ***once per client request*** (connection).

Warning

When using a scoped service in a middleware, inject the service into the ***Invoke*** or ***InvokeAsync*** method. Don't inject via [constructor injection](#) because it forces the service to behave like a singleton. For more information, see [Write custom ASP.NET Core middleware](#).

Singleton

Singleton lifetime services ([AddSingleton](#)) are created the ***first time they're requested*** (or when `Startup.ConfigureServices` is run and an instance is specified with the service registration). Every subsequent request uses the same instance. If the app requires singleton behavior, allowing the service container to manage the service's lifetime is recommended. Don't implement the singleton design pattern and provide user code to manage the object's lifetime in the class.

Warning

It's dangerous to resolve a scoped service from a singleton. It may cause the service to have incorrect state when processing subsequent requests.

Service registration methods

Suppose:

```
public interface IMyDep { ...}  
public class MyDep: IMyDep { ... }
```

Service registration extension methods offer overloads that are useful in specific scenarios.

Method	Automatic object disposal	Multiple implementations	Pass args
Add{LIFETIME}<{SERVICE}, {IMPLEMENTATION}>() Example: services.AddSingleton<IMyDep, MyDep>();	Yes	Yes	No
Add{LIFETIME}<{SERVICE}>(sp => new {IMPLEMENTATION}) Examples: services.AddSingleton<IMyDep>(sp => new MyDep()); services.AddSingleton<IMyDep>(sp => new MyDep("A string!"));	Yes	Yes	Yes
Add{LIFETIME}<{IMPLEMENTATION}>() Example: services.AddSingleton<MyDep>();	Yes	No	No
AddSingleton<{SERVICE}>(new {IMPLEMENTATION}) Examples: services.AddSingleton<IMyDep>(new MyDep()); services.AddSingleton<IMyDep>(new MyDep("A string!"));	No	Yes	Yes
AddSingleton(new {IMPLEMENTATION}) Examples: services.AddSingleton(new MyDep()); services.AddSingleton(new MyDep("A string!"));	No	No	Yes

Pentru interfata *IMyDep* si clasa *MyDep* ce implementeaza aceasta interfata adaugam una din urmatoarele variante in *ConfigureServices()*:

- AddTransient<IMyDep, MyDep>();
- AddScoped<IMyDep, MyDep>();
- AddSingleton<IMyDep, MyDep>();

Middleware

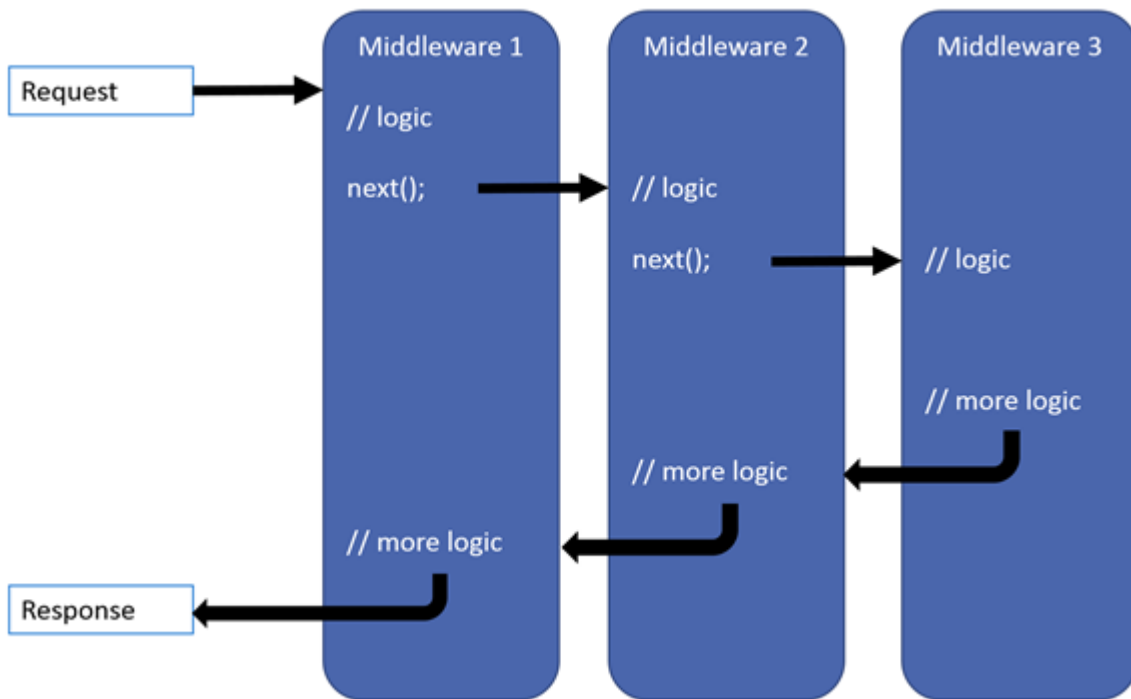
Pipeline pentru tratarea cererilor este compus din o serie de componente middleware. Fiecare componenta executa operatii asincrone pe un HttpContext si apoi fie invoca urmatoarea componenta fie termina tratarea cererii.

Prin conventie o componenta middlewarw este adaugata la pipiline prin invocarea unei metode extinse de tip Use..., metoda apelata in *Startup.Configure()*.

Fiecare componenta poate executa pre si post procesare a cererii in pipeline. Metodele (delegates) din cadrul fiecarui middleware manipuleaza fiecare cerere HTTP. Aceste metode sunt configurate folosind metodele extinse **Run**, **Map** si **Use**.

Un delegate individual pentru o cerere poate fi specificat si in-line ca o metoda anonima (se numeste in-line middleware) sau poate fi definit intr-o clasa reutilizabila. Aceste clase reutilizabile si metodele in-line anonime sunt middleware, cunoscute si sub numele de componente middleware.

Fiecare componenta middleware in pipeline-ul cererii este responsabila pentru invocarea urmatoarei componente din pipeline sau de terminarea tratarii cererii.



Exemplu

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (context, next) =>
    {
        // Do work that doesn't write to the Response.

        await next.Invoke();

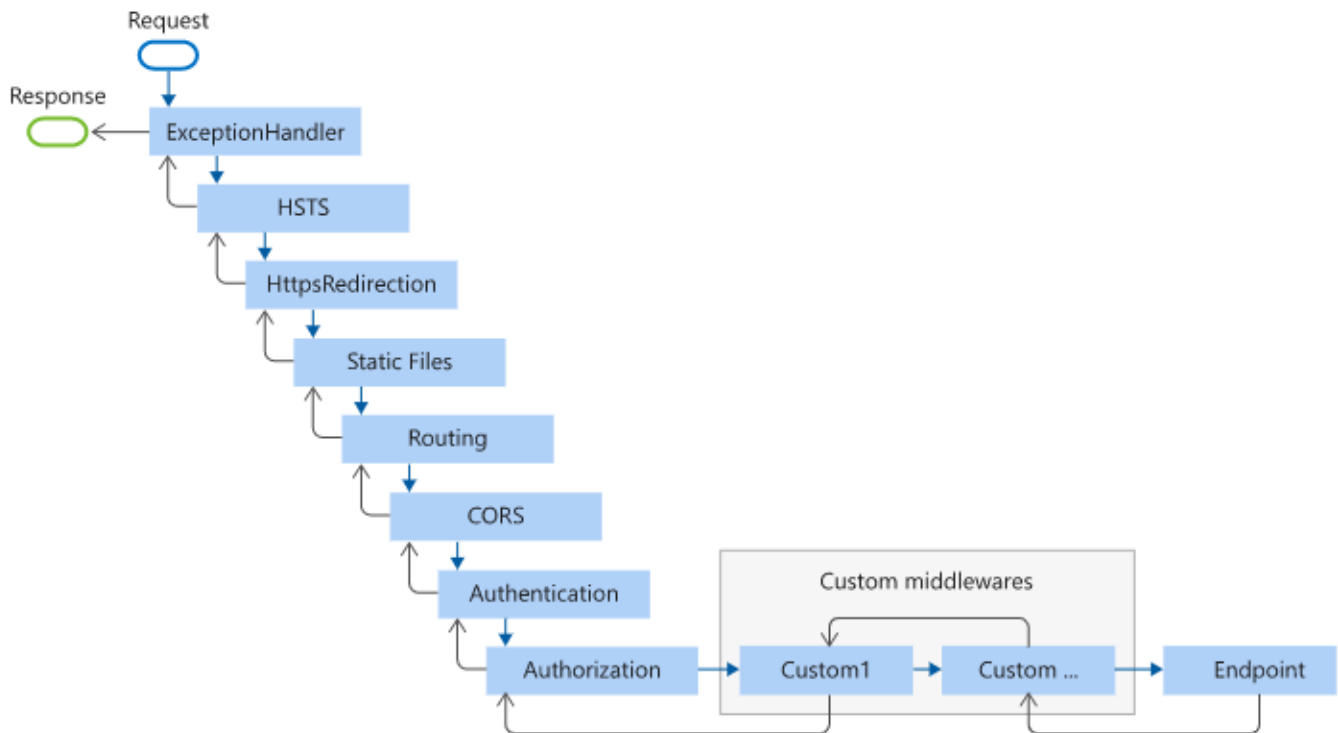
        // Do logging or other work that doesn't write to the Response.
    });

    app.Run(async context =>
    {
        await context.Response.WriteAsync("Hello from 2nd delegate.");
    });
}
```

Delegates multipli pot fi inlantuiti folosind metoda Use. Parametrul **next** reprezinta urmatorul delegate in pipeline. Daca nu se apeleaza **next.Invoke()** atunci tratarea cererii se considera terminata si urmatorii delegates din pipeline nu mai sunt apelati. De exemplu, middleware pentru procesarea cererilor privitoare la fisiere statice, **UseStaticFiles()**, termina procesarea cererii, este middleware terminal. Delegate-ul **Run** nu are parametrul **next**, deci e delegate terminal.

Ordine Middleware

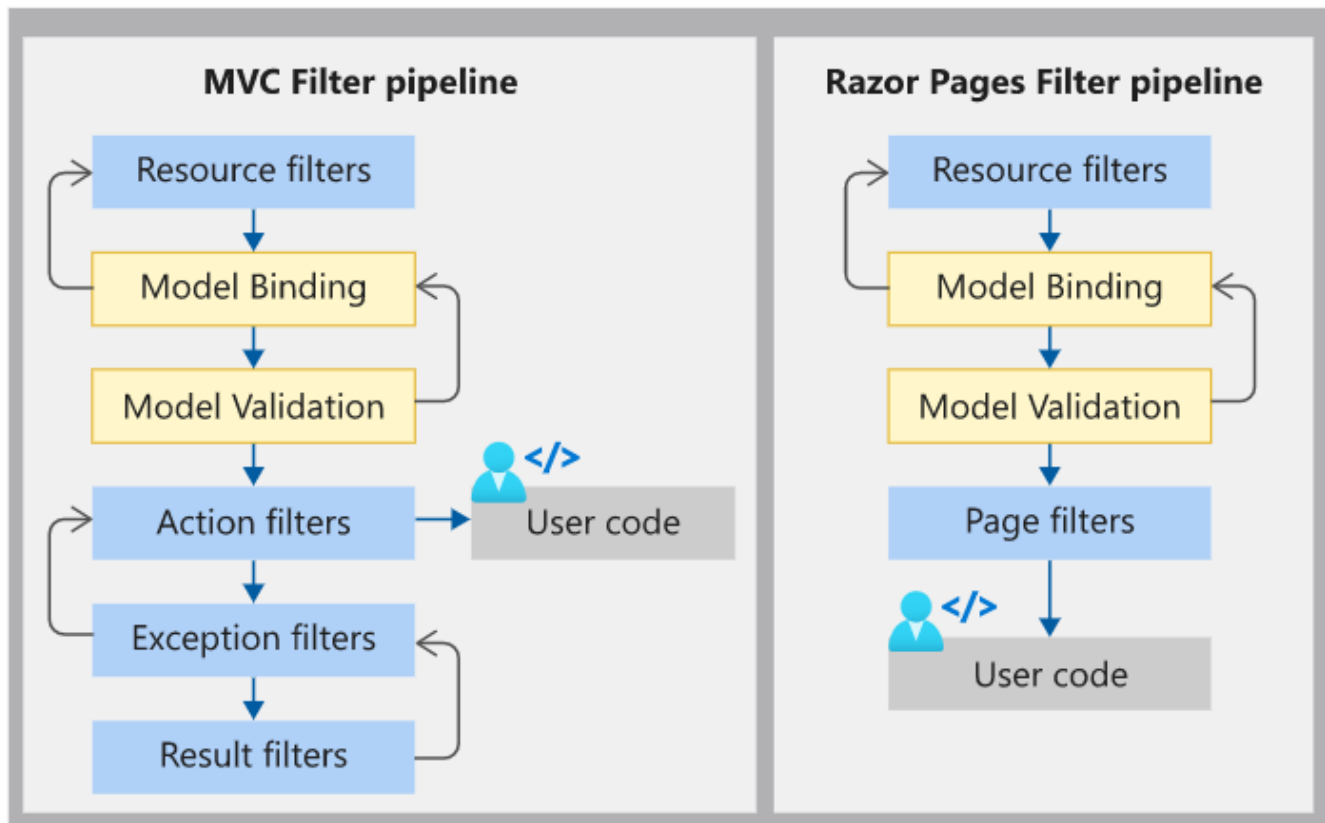
Diagrama ce urmeaza arata pipeline-ul complet ce proceseaza o cerere pentru app ASP.NET Core MVC si Razor Pages. Observam ordonarea middleware si locul unde sunt adaugate middleware personalizate. Se pot reordona middleware existente (ordinea in care apar in Configure()).



Endpoint middleware din diagrama de mai sus executa filtrul pipeline pentru o app de tip MVC sau Razor Pages. Vedeti diagrama de mai jos.

MVC Endpoint

(called by the Endpoint Middleware)



Scenariu comun – varianta 1	Scenarii comune – varianta 2
<pre> public void Configure(IApplicationBuilder app, IWebHostEnvironment env) { if (env.IsDevelopment()) { app.UseDeveloperExceptionPage(); app.UseDatabaseErrorPage(); } else { app.UseExceptionHandler("/Error"); app.UseHsts(); } app.UseHttpsRedirection(); app.UseStaticFiles(); // app.UseCookiePolicy(); app.UseRouting(); // app.UseRequestLocalization(); // app.UseCors(); app.UseAuthentication(); app.UseAuthorization(); // app.UseSession(); app.UseEndpoints(endpoints => </pre>	<pre> public void Configure(IApplicationBuilder app, IWebHostEnvironment env) { if (env.IsDevelopment()) { app.UseDeveloperExceptionPage(); app.UseDatabaseErrorPage(); } else { app.UseExceptionHandler("/Error"); app.UseHsts(); } app.UseHttpsRedirection(); app.UseStaticFiles(); app.UseCookiePolicy(); app.UseRouting(); app.UseAuthentication(); app.UseAuthorization(); app.UseSession(); app.UseEndpoints(endpoints => { endpoints.MapRazorPages(); }); } </pre>

<pre> { endpoints.MapRazorPages(); endpoints.MapControllerRoute(name: "default", pattern: "{controller=Home}/{action=Index}/{id?}"); }); } </pre>	
--	--

Descrierea completa a acestor scenarii o gasiti la adresa: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1>

Host

.NET Generic Host

Un host este un obiect ce incapsuleaza resursele unei app, cum ar fi:

- Dependency injection (DI)
- Logging
- Configuration
- Implementari IHostedService

Cand un host este lansat in executie, se apeleaza *IHostService.StartAsync* pe fiecare implementare a lui *IHostService* pe care o gaseste in containerul DI. Intr-o app web una din implementarile lui *IHostService* este un serviciu web ce lanseaza in executie o implementare a unui server HTTP.

Setare host

Host-ul. in mod obisnuit, este configurat, construit si rulat de cod din clasa *Program*. Se apeleaza metoda *CreateHostBuilder* si apoi metodele extinse *Build* si *Run*.

```

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
}

```

ASP.NET Core Web Host

App ASP.NET Core configureaza si lanseaza un *host*. Host-ul este responsabil pentru lansarea aplicatiei si managementul timpului e viata. Minimum, host-ul configureaza un server si un pipeline pentru procesarea cererii. Host-ul poate seta jurnalizarea, DI si configurarea.

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
        WebHost.CreateDefaultBuilder(args)
            .UseStartup<Startup>();
}
```

Mai multe informatii gasiti la adresa: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/web-host?view=aspnetcore-3.1>

Implementare server Web in ASP.NET Core

O app ASP.NET Core ruleaza intr-un server HTTP **in-process**. Serverul "asculta" pentru cereri HTTP si le transmite la app ca o multime de "request features" compuse intr-un *HttpContext*.

Observatie

ASP.NET Core defines a number of HTTP feature interfaces in `Microsoft.AspNetCore.Http.Features` which are used by servers to identify the features they support.

Observatie

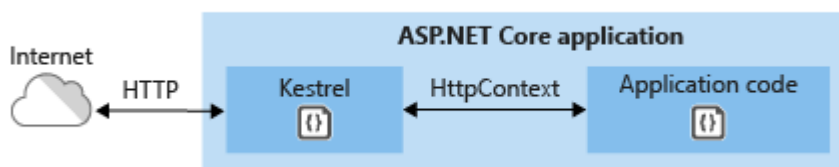
In-process means the component runs in the same **process** space as the one using it. **Out-process** means the component runs in a different **process** space compared to the one using it. The two **processes** may be running on the same machine also.

Kestrel

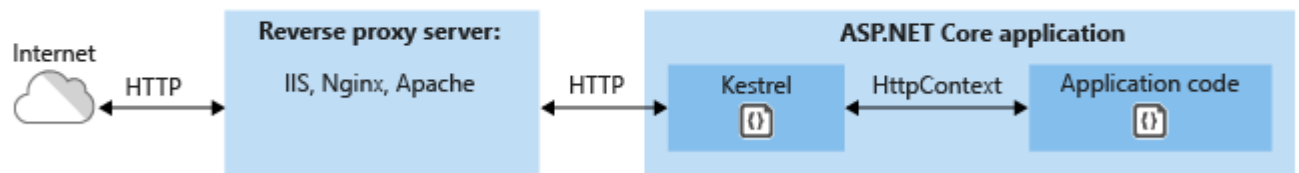
Kestrel este serverul web implicit specificat de template-ul unei app ASP.NET Core.

Utilizare Kestrel:

- Ca un server ce proceseaza cereri, in mod direct, cereri venind din retea, inclusiv Internet.



With a *reverse proxy server*, such as [Internet Information Services \(IIS\)](#), [Nginx](#), or [Apache](#). A reverse proxy server receives HTTP requests from the Internet and forwards them to Kestrel.



Configuration in ASP.NET Core

Provider-i pentru configurare

- Fisiere, ex. appsettings.json.
- variabile de mediu.
- Argumente linia de comanda.
- Custom.

Logging in .NET Core si ASP.NET Core

Un provider pentru log-uri afiseaza sau memoreaza aceste log-uri. Ex provider Console. Se pot folosi mai multi provideri pentru log-uri.

Add providers

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        })
        .ConfigureWebHostDefaults(webBuilder =>
        {
            webBuilder.UseStartup<Startup>();
        });
```

Template implicit pentru proiect de tip ASP.NET Core apeleaza CallDefaultBuilder ce adauga urmatoorii provideri pentru log-uri:

- Console
- Debug
- EventSource
- EventLog (only when running on Windows)

Pentru metodele si metodele extinse folosite in log-uri vedeti interfata ILogger.

Rutarea (Routing) in ASP.NET Core

Rutarea este responsabila pentru identificarea corecta a endpoint-urilor ce vor servici o cerere HTTP. Endpoint-urile reprezinta cod din cadrul app ce trateaza o cerere. Endpoint-urile sunt definite in app si configurate cand se lanseaza in executie app. Endpoint are acces la URL cererii si poate obtine informatii din acesta. Cu ajutorul acestor informatii, codul din rutare poate genera URL ce se potriveste cu endpoint-ul.

App poate configura rutarea folosind:

- Controllers
- Razor Pages
- SignalR
- gRPC Services
- Endpoint-enabled *middleware* such as [Health Checks](#).
- Delegates and lambdas registered with routing.

Observatie

Pentru Health Checks consultati "Health checks in ASP.NET Core" de la adresa:

<https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-3.1>

Rutare – principii de baza

Rutarea este inregistrata in middleware din pipeline, metoda Startup.Configure().

Exemplu (am vazut mai mult cod la middleware si de asemenea unde e plasat acest middleware in pipeline).

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapGet("/", async context =>
        {
            await context.Response.WriteAsync("Hello World!");
        });
    });
}
```

Rutarea foloseste o pereche de middleware, inregistrate de UseRouting si UseEndpoints:

- **UseRouting** adauga rute ce se potrivesc cu middleware din pipeline. Acest middleware “se uita” la o multime de endpoint-uri definite in app, si selecteaza cea mai buna potrivire bazata pe cerere.
- **UseEndpoints** adauga cod pentru executia endpointului la middleware din pipeline. El ruleaza delegate-ul asociat cu endpoint-ul selectat.

Codul anterior include o singura ruta si codul aferent endpoint-ului folosind metoda MapGet:

- Cand o cerere HTTP GET este trimisa la URL /:
 - Delegate-ul asociat cererii se executa.
 - “Hello World!” este scris ca raspuns HTTP. Implicit, radacina URL / este <https://localhost:5001/>.
- Daca metoda ceruta nu este GET sau radacina URL nu este /, atunci nici o ruta nu se potriveste si se genereaza si returneaza un raspuns HTTP 404.

Endpoint

Metoda MapGet este folosita pentru a defini un endpoint. Un endpoint (in acest caz) e ceva ce poate fi:

- Selectat, prin potrivirea cu URL si metoda HTTP.
- Executat, prin rularea delegate-ului.

Endpoint-urile ce pot fi identificate si executate de app sunt configurate in UseEndpoints. De exemplu, MapGet, MapPost si metode similare, conecteaza delegates pentru cereri la sistemul de rutare. Metode aditionale pot fi utilizate pentru a conecta trasaturi ASP.NET Core framework la sistemul de rutare:

- *MapRazorPages* for Razor Pages
- *MapControllers* for controllers
- *MapHub<THub>* for SignalR
- *MapGrpcService<TService>* for gRPC

Rutare – Concepte

Endpoint - definitie

Un ASP.NET Core endpoint este:

- Executabil: Are un [RequestDelegate](#).
- Extensibil: Are o colectie de [Metadata](#).
- Selectabil: Optional, are informatii despre rutare.
- Enumerabil: Colectia de endpoint-uri poate fi listata folosind *EndpointDataSource* din DI.

Reguli pentru endpoint:

- Endpoint este totdeauna null inainte de apel UseRouting.
- Daca a fost gasita o potrivire (cand se ruleaza UseRouting), endpoint-ul este diferit de null intre UseRouting si UseEndpoints.
- Middleware UseEndpoints este terminal cand s-a gasit o “potrivire”.
- Middleware de dupa UseEndpoints se vor executa numai daca nu a fost determinat un alt endpoint inainte.

Intre *UseRouting()* si *UseEndpoints()* se pot intercala middleware care pot schimba anumite informatii in cerere, de exemplu date senzitive si necesita autorizare, scriere in log-uri, etc.

URL matching

Determinare URL:

- Este procesul prin care rutarea determina pentru cererea data un endpoint ce o va trata. E posibil sa nu existe un asemenea endpoint.
- Este bazata pe datele din URL si antet cerere.
- Poate fi extinsa pentru a considera orice data in cerere.

Route template

Token intre {} definesc parametrii rutei. Parametrii rutei trebuie sa fie separati prin / (cel mai des folosit). Parametrii rutei trebuie sa aiba un nume si pot avea atasati attribute. Cautarile pe text sunt *case insensitive*.

Exemplu corect de ruta cu segmente:

`{controller=name}/{action=Index}/{id=0}`

Aceasta ruta este compusa din trei segmente.

Exemplu incorect de ruta:

`{controller=name}{action=Index}`

Nu se face diferentierea intre segmente.

Asterisk * sau dublu asterisk **:

- Poate fi utilizat ca prefix la un parametru al rutei pentru a putea specifica orice in ruta.
- Se numesc parametri "catch-all". De exemplu , `blog/{**name}`:
 - Se potriveste cu orice URI ce incepe cu `/blog` urmat de orice valoare.
 - Valoarea de dupa `/blog` este atribuita valorii rutei `name`.

Parametrii "catch-all" se potrivesc cu stringul vid.

Exemplu

Template **`files/{filename}.{ext?}`** se potriveste cu urmatoarele:

- `/files/myFile.txt`
- `/files/myFile`

Rute cu constrangeri

constraint	Example	Example Matches	Notes
int	{id:int}	123456789, -123456789	Matches any integer
bool	{active:bool}	true, FALSE	Matches true or false. Case-insensitive
datetime	{dob:datetime}	2016-12-31, 2016-12-31 7:32pm	Matches a valid <code>DateTime</code> value in the invariant culture. See preceding warning.
decimal	{price:decimal}	49.99, -1,000.01	Matches a valid <code>decimal</code> value in the invariant culture. See preceding warning.
double	{weight:double}	1.234, -1,001.01e8	Matches a valid <code>double</code> value in the invariant culture. See preceding warning.
float	{weight:float}	1.234, -1,001.01e8	Matches a valid <code>float</code> value in the invariant culture. See preceding warning.
guid	{id:guid}	CD2C1638-1638-72D5-1638-DEADBEEF1638	Matches a valid <code>Guid</code> value
long	{ticks:long}	123456789, -123456789	Matches a valid <code>long</code> value
minlength(value)	{username:minlength(4)}	Rick	String must be at least 4 characters
maxlength(value)	{filename:maxlength(8)}	MyFile	String must be no more than 8 characters
length(length)	{filename:length(12)}	somefile.txt	String must be exactly 12 characters long
length(min,max)	{filename:length(8,16)}	somefile.txt	String must be at least 8 and no more than 16 characters long
min(value)	{age:min(18)}	19	Integer value must be at least 18
max(value)	{age:max(120)}	91	Integer value must be no more than 120
range(min,max)	{age:range(18,120)}	91	Integer value must be at least 18 but no more than 120
alpha	{name:alpha}	Rick	String must consist of one or more alphabetical characters, a-z and case-insensitive.

constraint	Example	Example Matches	Notes
regex(expression)	{ssn:regex(^\\d{{3}}-\\d{{2}}-\\d{{4}}\$)}	123-45-6789	String must match the regular expression. See tips about defining a regular expression.
required	{name:required}	Rick	Used to enforce that a non-parameter value is present during URL generation

Exemplu. Mai multe constrangeri pe acelasi segment.

```
[Route("users/{id:int:min(1)}")]
public User GetById(int id) { }
```

Constrangerea este: id este un intreg cu valoarea minima 1.

Conventii in definire rute

Exemplu

```
endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Ruta are trei segmente.

- Primul segment {controller=Home} mapeaza la numele controller-ului.
- Segmentul doi {action=Index} mapeaza la numele actiunii.
- Segmentultrei este optional {id?}, poate fi orice tip .NET.

Ce se potriveste cu aceasta ruta?

- /Products/List mapeaza la controller *Products* si actiunea *List* definita in acest controller.
- /Products/List/17 asemanator ca mai sus dar parametrul *id* are valoarea 17.

Attribute routing for REST APIs

REST APIs ar trebui sa utilizeze attribute pentru a indica ruta. Attributele se refera la verbele HTTP.

In exemplul ce urmeaza se foloseste atributul *Route* aplicat actiunilor din controller.

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```



```

[Route("Home/About")]
[Route("Home/About/{id?}")]
public IActionResult About(int? id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
}

```

Reserved routing names

Urmatoarele cuvinte cheie sunt rezervate pentru numele parametrilor din ruta atunci cand folosim Controllers si Razor Pages:

- action
- area
- controller
- handler
- page

HTTP verb templates

ASP.NET Core are urmatoarele templates-uri pentru verbele HTTP:

- [\[HttpGet\]](#)
- [\[HttpPost\]](#)
- [\[HttpPut\]](#)
- [\[HttpDelete\]](#)
- [\[HttpHead\]](#)
- [\[HttpPatch\]](#)

Exemple de attribute disponibile.

Attribute	Notes
[Route]	Specifies URL pattern for a controller or action.
[Bind]	Specifies prefix and properties to include for model binding.
[HttpGet]	Identifies an action that supports the HTTP GET action verb.
[Consumes]	Specifies data types that an action accepts.
[Produces]	Specifies data types that an action returns.

Exemplu cu attribute aplicate la nivel de actiune.

```

[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<Pet> Create(Pet pet)
{
    pet.Id = _petsInMemoryStore.Any() ?
        _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
    _petsInMemoryStore.Add(pet);
}

```

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

Rute conventionale dedicate. Exemplu de definire.

```
app.UseEndpoints(endpoints =>  
{  
    endpoints.MapControllerRoute(name: "blog",  
        pattern: "blog/{*article}",  
        defaults: new { controller = "Blog", action = "Article" });  
    endpoints.MapControllerRoute(name: "default",  
        pattern: "{controller=Home}/{action=Index}/{id?}");  
});
```

În exemplul de mai sus se folosesc conventii pentru a defini ruta.