PROGRAMMAZIONE ORIENTATA AI MICROSERVIZI

ASP.NET Core e Web API

Tommaso Nanu
tommaso.nanu@alad.cloud

UNIVERSITÀ DI PARMA
DIPARTIMENTO DI SCIENZE MATEMATICHE, FISICHE E INFORMATICHE
Corso di Laurea in Informatica

22 novembre 2023

Presentazione di Jonathan Ambri
jonathan.ambri@alad.cloud

ASP.NET Core

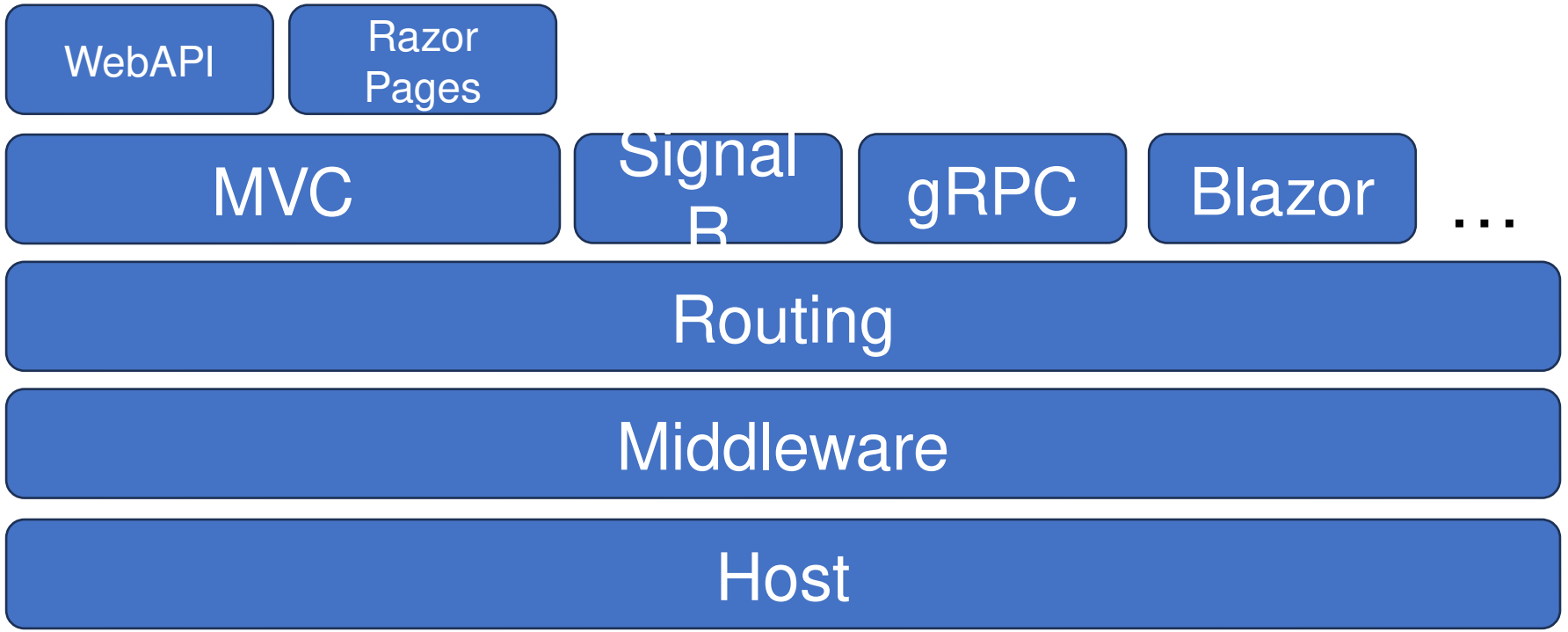
22 novembre 2023

ASP.NET Core

- **ASP.NET Core** è un framework *open source* e *multipiattaforma (cross-platform)* di *Microsoft* dedicato allo sviluppo e all'esecuzione di varie tipologie di *applicazioni web*, all'interno della piattaforma **.NET**.
- ASP.NET Core è più leggero dei suoi predecessori, in quanto modulare: è composto da diversi componenti, utilizzabili in maniera indipendente uno dall'altro, e sfrutta diverse librerie distribuite mediante *pacchetti NuGet*.
- https://github.com/dotnet/aspnetcore

Architettura di ASP.NET Core

- ASP.NET Core è composto da diverse componenti



Architettura di ASP.NET Core

- Lo strato più basso è il componente **Host** che si occupa di eseguire il *Web Server* e quindi di avviare ogni applicazione ASP.NET Core e tutti i servizi che essa utilizzerà durante il suo ciclo di vita: *Dependency Injection* (*DI*) e *Logging.*
  - L'Host permette di ricevere le richieste HTTP.

- Lo strato successivo è quello dei **Middleware**: sono componenti posizionati in un certo *ordine* in modo da formare una *pipeline di elaborazione;* ogni richiesta HTTP dovrà attraversare questa pipeline.
  - Ogni middleware può eseguire azioni prima di passare la richiesta al componente successivo, sia dopo avere ricevuto da questo la risposta.
  - È possibile modificare l'ordine di esecuzione, implementare e aggiungere nuovi middleware custom.

Architettura di ASP.NET Core

- Per ogni *richiesta HTTP*, ASP.NET Core crea 3 oggetti:
  - **HttpRequest**: rappresenta la richiesta;
  - **HttpResponse**: la risposta restituita al client;
  - **HttpContext**: contiene tutte le informazioni sul *contesto* della richiesta e fornisce l'accesso alle funzionalità di ASP.NET Core.
- Queste 3 classi sono definite nel namespace *Microsoft.AspNetCore.Http* del pacchetto *Microsoft.AspNetCore.Http.Abstractions.*

- Lo strato successivo è quello di **Routing** che si occupa di ispezionare ogni richiesta ricevuta e indirizzarla all'endpoint appropriato per l'URL specificato nella richiesta (funzionalità di *Endpoint Routing*): quindi mappa l'URL verso un endpoint.

Architettura di ASP.NET Core

- Sopra lo strato di Routing abbiamo quello degli **Endpoint**, che sono le componenti che si occupano di soddisfare ed elaborare una richiesta. Il compito principale di uno sviluppatore è quello di implementare questi endpoint.
- Un endpoint contiene quindi i metadati relativi la richiesta HTTP e il gestore per elaborarla.
- Un endpoint può essere un metodo (*action*) di un *controller MVC*, oppure servizi *gRPC* o componenti di *Blazor.*
- Le *WebApi* e le *Razor Pages* sono una specializzazione dell'endpoint *MVC*.

Ambiente di sviluppo

- Per utilizzare il framework ASP.NET Core è sufficiente installare il **.NET SDK** (*Software Development Kit*): attualmente siamo alla versione **8**.
- L'SDK comprende oltre alle librerie, strumenti vari e al compilatore, anche il *runtime* di ASP.NET Core, cioè l'ambiente di esecuzione delle applicazioni web.
- L'SDK fornisce anche il *.NET CLI* (*Command Line Interface*):
  - > *dotnet --version*: restituisce il numero di versione installata dell'SDK
  - > *dotnet --info*: restituisce informazioni sull'SDK installato e sui runtime .NET
  - Il comando **dotnet** (detto *driver*) è seguito da un *sottocomando* (*verbo*) e da eventuali argomenti o opzioni.
  - > *dotnet --help* oppure ***-h**:* guida alla riga di comando
  - > *dotnet **new** --help* oppure *-h*: guida al sottocomando *new,* che serve a creare un nuovo progetto .NET

Template di un progetto

Un *template* è un progetto con il codice base per compilare ed eseguire un'applicazione.

> *dotnet* ***new list***: restituisce la lista di template forniti dal .NET SDK

L'opzione ***--tag*** permette di filtrare i tipi di template:

> *dotnet* ***new list --tag Web***: restituisce la lista di template per lo sviluppo web

I principali template per lo sviluppo web sono:

- ***web***: per generare un'applicazione ASP.NET vuota
- ***razor*** o ***webapp***: per generare un'applicazione basata su *Razor Pages*
- ***mvc***: per generare un'applicaione *MVC*
- ***webapi***: per generare un'applicazione *Web API*

Creazione di un progetto Razor Pages

- *> mkdir Esempio/src*
- *> cd ./Esempio/src* (la cartella *src* può contenere più progetti)
- *> dotnet* ***new webapp*** *-n EsempioWebApp*:

  creazione di un progetto usando il template *webapp*, in cui l'opzione *-n* permette di assegnare il nome all'applicazione.

  Il comando genera una cartella con lo stesso nome dell'applicazione (per assegnare un nome diverso alla cartella, usare l'opzione *-o*) contenente il codice e le risorse per un'applicazione Razor Pages,

  Dopo esegue automaticamente il *restore* di tutte le dipendenze e dei pacchetti NuGet utilizzati.
- *> cd ./EsempioWebApp*

Creazione di un progetto Razor Pages

- Ora compiliamo il progetto con il seguente comando:

  > *dotnet* ***build****: v*engono prodotti gli assembly in formato *.dll*

  Possiamo fornire una configurazione di compilazione usando l'opzione *--configuration* o *–c.*

  Le configurazioni possibili sono:

  - *Debug*: utilizzata durante lo sviluppo e permette di produrre i simboli di debug all'interno dei file con estensione *.pdb* per ogni .dll, in modo da avere una correlazione tra i sorgenti e gli assembly. Sarà così possibile risalire alla riga di codice che ha causato un errore.
  - *Release*: utilizzata quando si deve pubblicare in produzione; il compilatore apporta delle ottimizzazioni e non vengono emessi i simboli di debug.

Creazione di un progetto Razor Pages

- Una volta che il progetto è compilato, lo possiamo eseguire con il comando
  - *dotnet* ***run***: esegue anche la compilazione (la *build* eseguita prima) e poi lancia l'esecuzione del *web server* che ospita l'applicazione.

    In output è mostrato:
    - Gli *URL* ai quali è raggiungibile l'applicazione in esecuzione (uno HTTP e uno HTTPS);
    - Le istruzioni su come stoppare l'esecuzione (Press Ctrl+C to shut down);
    - L'ambiente di hosting (*Development* nel nostro caso);
    - Il path in cui si trova il contenuto dell'applicazione.
- In *EsempioWebApp/Properties/**launchSettings.json*** sono definite le impostazioni per l'avvio e in particolare i *profili di esecuzione* che sono letti quando l'applicazione viene eseguita.
  - Per specificare il profilo da utilizzare si usa il comando:

    > *dotnet run* ***--launch-profile*** "*NomeProfile*" (funziona solo per i profili Kestrel)

Struttura del progetto Razor Pages

- Ogni applicazione ASP.NET Core è un'applicazione *Console.*

Inoltre è **self-hosted** in quanto comprende al suo interno tutto il necessario per eseguirla, in particolare il *web server*.

Il web server predefinito è **Kestrel** che è cross-platform.

Struttura del progetto Razor Pages

- Il **progetto** dell'applicazione è descritto da un file con estensione **.csproj** (EsempioWebApp.csproj). Il progetto, dal punto di vista logico, fa parte di una **soluzione**, che è un file con estensione **.sln**: il concetto di soluzione serve a raggruppare diversi progetti.
- Il comando > *dotnet new webapp -n EsempioWebApp* non crea automaticamente il file di soluzione; è possibile ottenerlo mediante un altro apposito template:

  > *cd ../..* (spostiamoci nella cartella *Esempio*)

  > *dotnet **new sln** -n Esempio*
- Per aggiungere quindi il file di progetto alla soluzione si esegue il comando

  > *dotnet **sln add** ./src/EsempioWebApp\EsempioWebApp.csproj*

Struttura del progetto Razor Pages

- Nella cartella del progetto *EsempioWebApp* sono presenti 3 sotto cartelle:
  - Pages: contiene le *view* (pagine) *cshtml*;
  - *Properties*: contiene il *launchSettings.json*;
  - *wwwroot*: contiene i *file statici*, accessibili direttamente dal browser: file CSS, JavaScript e HTML.
- Sempre nella cartella del progetto sono presenti 2 file json: *appsettings.json* e *appsettings.Development.json* che contengono impostazioni per l'esecuzione dell'applicazione, come *stringhe di connessione al DB*, configurazioni specifiche per l'installazione, password, etc.

- L'unico sorgente C# presente è *Program.cs* che è l'*entry point* dell'applicazione e contiene il codice necessario per configurare e avviare l'applicazione.

File di progetto

Il file di progetto è quello con estensione *.csproj* ed è un *xml:* contiene le proprietà necessarie alla compilazione dell'applicazione ed eventuali riferimenti a pacchetti esterni.

```
<Project Sdk="Microsoft.NET.Sdk.Web"> -> tipologia di progetto
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>  -> versione di .NET
    <Nullable>enable</Nullable> -> per abilitare il Nullable context (disable per disabilitarlo)
    <ImplicitUsings>enable</ImplicitUsings> -> per abilitare le Implicit Global Usings
  </PropertyGroup>
</Project>
```

File di progetto

Il *.csproj* contiene anche i *riferimenti* ai *pacchetti NuGet* usati dal progetto.

Per aggiungere, per esempio, il pacchetto NuGet
AutoMapper.Extensions.Microsoft.DependencyInjection
al nostro progetto, possiamo usare il comando:

*> dotnet **add** EsempioWebApp.csproj **package** AutoMapper.Extensions.Microsoft.DependencyInjection*

Con l'opzione *-s* è possibile specificare le origini dei pacchetti NuGet da usare durante il ripristino, per esempio
-s https://api.nuget.org/v3/index.json

Nel .csproj viene aggiunto il seguente nodo:

```
<ItemGroup>
   <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="12.0.1" />
</ItemGroup>
```

Su Windows, I pacchetti NuGet sono salvati in C:\Users\NomeUtente\.nuget\packages

File di progetto

Se vogliamo creare una libreria, contenente classi utilizzate da progetti differenti, possiamo creare un progetto *Class Library* usando il template *classlib*:

> *cd ..* (spostiamoci nella cartella *src*)

> *dotnet new* ***classlib*** *-n EsempioClassLib*

> *cd ..* (spostiamoci nella cartella *Esempio per aggiungere il nuovo progetto alla soluzione*)

> *dotnet sln add ./src/EsempioClassLib/EsempioClassLib.csproj*

Ora possiamo aggiungere all'applicazione web *EsempioWebApp* il *riferimento* al nuovo progetto *EsempioClassLib*, tramite il comando:

> *cd ./src/EsempioWebApp*

> *dotnet* ***add*** *EsempioWebApp.csproj* ***reference*** *../EsempioClassLib/EsempioClassLib.csproj*

Nel csproj di *EsempioWebApp* viene aggiunto il seguente nodo:

```
<ItemGroup>
  <ProjectReference Include="..\EsempioClassLib\EsempioClassLib.csproj" />
</ItemGroup>
```

File di progetto

Il csproj completo del progetto *EsempioWebApp* risulta poi essere il seguente:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="AutoMapper.Extensions.Microsoft.DependencyInjection" Version="12.0.1" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="..\EsempioClassLib\EsempioClassLib.csproj" />
  </ItemGroup>

</Project>
```

Program.cs

- A partire da .NET 6, i template generano le applicazioni usando le *Top Level Statement* (introdotte in C# 9) e quindi le istruzioni contenute nel *Program.cs* sono eseguite direttamente una dopo l'altra come se fossero contenute all'interno di un metodo *Main* di una classe *Program.*
- *Program.cs* si occupa di configurare e avviare un'istanza dell'interfaccia **IHost** (definita nel namespace *Microsoft.Extensions.Hosting* del pacchetto *Microsoft.Extensions.Hosting.Abstractions*), ovvero l'Host dell'applicazione, che incapsula tutte le risorse necessarie:
  - Avvia il *web server*;
  - Definisce i *servizi* e le loro configurazioni;
  - Definisce i componenti *middleware* (ovvero la *pipeline di elaborazione*).

Program.cs

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddRazorPages();
var app = builder.Build();

// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    // The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
app.MapRazorPages();
app.Run();
```

Program.cs

- *WebApplication.CreateBuilder(args)* crea un'istanza della classe **WebApplicationBuilder** (namespace *Microsoft.AspNetCore.Builder* del pacchetto *Microsoft.AspNetCore*), che è l'oggetto tramite cui si aggiungono e configurano i *servizi* da rendere disponibili all'applicazione, tramite un meccanismo chiamato **Dependency Injection** (**DI**).
- Per *servizio* intendiamo una qualsiasi classe che fornisce funzionalità all'applicazione.
- La property *builder.Services* è di tipo **IServiceCollection** (namespace *Microsoft.Extensions.DependencyInjection* del pacchetto *Microsoft.Extensions.DependencyInjection.Abstractions*) e rappresenta il c*ontenitore dei servizi* dell'applicazione ed è utilizzato per aggiungere (registrare) servizi custom oppure del framework.
- *builder.Services.AddRazorPages()* permette appunto di aggiunge i servizi per utilizzare le Razor Pages.

Program.cs

- Terminata la registrazione e configurazione dei servizi, invochiamo il metodo *builder.Build()* che si occupa di costruire e ritornare un'istanza di **WebApplication**, che implementa l'interfaccia *IHost*.
- Ora è possibile utilizzare questo oggetto *WebApplication* per configurare la *pipeline di elaborazione,* aggiungendo i vari *middleware*, e gli *endpoint*: ovvero si configura la gestione delle richieste.
- L'ultima istruzione *app.Run()* permette di avviare l'applicazione mettendo l'Host in ascolto delle richieste HTTP.
  - Possiamo passare in input al metodo *Run* l'URL su cui il server deve mettersi in ascolto (usato solo se l'indirizzo non è stato specificato nella proprietà *applicationUrl* del file di configurazione *launchSettings.json).*

Dependency Injection (DI)

- Il pattern *Dependency Injection* (*DI*) permette di rispettare uno dei principi *SOLID*, noto come *Dependency Inversion*:

  «High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend upon details. Details should depend upon abstractions.»

  La DI permette di ottenere l'*Inversion of control* tra le classi e le loro relative *dipendenze*.
- ASP.NET Core supporta nativamente la DI e ha un solo punto di configurazione.
- Caratteristiche della DI:
  - Invece di usare un'implementazione concreta di una dipendenza, viene utilizzata un'*interfaccia* o *classe astratta*;
  - Le dipendenze (ovvero i servizi) vengono *registrate* nel *contenitore dei servizi* (il *builder.Services*): In ASP.NET Core la registrazione dei servizi avviene nel *Program.cs*.
  - Le dipendenze, sono poi iniettate all'interno del costruttore delle classi che le utilizzano: è il framework che si occupa di istanziare e iniettare queste dipendenze.
- I *contenitori dei servizi*, in cui vengono creati e mantenuti i servizi registrati, vengono detti *IoC Container* (*IoC* è l'acronimo di *Inversion of Control*) .

Service Lifetime

I servizi dell'applicazione possono essere registrati con diversi *lifetime* usando i seguenti metodi di estensione per l'interfaccia *IServiceCollection* (definiti nella classe statica *ServiceCollectionServiceExtensions*):

- **AddSingleton**: Crea una sola istanza del servizio quando viene richiesto per la prima volta e riutilizza la stessa istanza in tutte le posizioni in cui è necessario il servizio: quindi viene creata una singola istanza e viene condivisa in tutte le richieste.
  - Dobbiamo essere consapevoli dei possibili problemi di concorrenza e threading;
  - Possibili memory leaks accumulati nel tempo;
  - Comunque efficienti in quanto creati una sola volta e riutilizzati ovunque.
  - Utili quando bisogna mantenere uno *stato* a livello di applicazione, come per esempio: parametri di configurazione, Logging Service, caching, etc.

Service Lifetime

- **AddScoped**: per ogni richiesta HTTP, otteniamo una nuova istanza del servizio: una richiesta HTTP rappresenta uno *scope*. Tuttavia, all'interno della stessa richiesta HTTP, se il servizio è richiesto in più posizioni, allora viene fornita la stessa istanza per l'intero scope di tale richiesta HTTP. Ogni nuova richiesta HTTP otterrà però una nuova istanza del servizio.
  - È l'opzione migliore quando si desidera mantenere lo stato all'interno di una stessa richiesta HTTP.
- **AddTransient**: viene fornita una nuova istanza ogni volta che viene richiesta un'istanza del servizio, indipendentemente dal fatto che rientri nello stesso scope della richiesta HTTP o tra richieste HTTP diverse.
  - Da utilizzare nel caso di servizi leggeri e con stato minimo o nullo;
  - È l'opzione più sicura in caso di multithreading.

Middleware

- I Middleware sono i componenti che formano la pipeline di elaborazione e servono per gestire le richieste in arrivo e restituire le risposte al client.
- Per aggiungere un middleware alla pipeline si utilizzano dei metodi di estensione per l'interfaccia **IApplicationBuilder** (*WebApplication* implementa anche questa interfaccia), come per esempio il metodo *app.UseHttpsRedirection()* fornito dal framework.
- Possiamo considerare i middleware come una sequenza di **delegate**.
- Possiamo aggiungere anche dei middleware custom alla pipeline attraverso 3 diversi metodi:
  - **Run**: aggiunge un middleware che termina la pipeline restituendo una risposta;
  - **Map**: aggiunge un middleware che crea una diramazione nella pipeline mappandola mediante l'URL;
  - **Use**: aggiunge un middleware che può decidere se passare la richiesta al middleware successivo e che può modificare la richiesta e la risposta prima e dopo di esso.

Middleware

- [C#] Un *delegate* definisce una firma di un metodo, cioè i suoi parametri e il tipo di ritorno, che un'implementazione deve rispettare. Il delegate utilizzato dalla pipeline di elaborazione è questo:

  public delegate Task RequestDelegate(HttpContext context);

- Il *WebApplication* implementa le interfacce *IHost*, *IApplicationBuilder*, *IEndpointRouteBuilder* e *IAsyncDisposable*, e può quindi utilizzare tutti i metodi di estensione definiti per queste interfacce.

- Il metodo *UseExceptionHandler()* è il primo middleware della pipeline e si occupa di gestire eventuali *eccezioni*: essendo il primo, può gestire le eccezioni che si verificano a qualunque livello della pipeline.

Middleware

- Il metodo UseHsts() forza l'utilizzo di *SSL*, in modo che sia possibile utilizzare solo il protocollo HTTPS: aggiunge l'header di risposta *HTTP Strict-Transport-Security.*
- Il metodo *UseHttpsRedirection()* effettua il redirect delle richieste HTTP verso HTTPS.
- Il metodo *UseStaticFiles()* aggiunge il middleware per la gestione delle richieste dei file statici presenti nella cartella speciale *wwwroot*.
- Il metodo *UseRouting()* esegue il routing verso il corretto endpoint, ovvero determina quale pagina restituire interpretandolo dall'URL della richiesta.
- Il metodo *UseAuthorization()* aggiunge il middleware di autorizzazione che determina se consentire o meno l'accesso a una risorsa (per esempio a una pagina). Questo middleware deve trovarsi tra quello di routing e quello dell'endpoint.
- Il metodo *MapRazorPages()* aggiunge l'endpoint Razor Pages che restituisce la pagina HTML (.cshtml) richiesta.

Web Api

22 novembre 2023

Web API

- Un *API* (*Application Programming Interface*) è un'interfaccia esposta tramite un *endpoint* (pubblico o privato) utilizzabile per ottenere dati o chiamare servizi.
- *Web API*: API che funziona sul web, sfruttando il protocollo *HTTP*; possono essere implementate e consumate (utilizzate) con qualsiasi tecnologia e linguaggio.
- Web API: *REST* oppure *SOAP*

REST (REpresentational State Transfer)

- I servizi (web service) basati su REST sono chiamati **RESTful**.
- REST è un *pattern architetturale*: non definisce tecnologie o protocolli ma indica il modo di esporre gli endpoint.
- Un servizio RESTful è quindi un servizio web che definisce una o più API per mezzo di un insieme di *URL* e di operazioni consentite, specificate tramite i metodi (o verbi HTTP) *GET, POST, PUT o DELETE*, e che può restituire dati in genere in formato *JSON* (oppure *XML*), all'interno del corpo della risposta.
  - Viene usato il formato JSON per la rappresentazione dei dati perché semplice, leggero e utilizzabile da qualunque client.

SOAP (Simple Object Access Protocol)

- Questi web service sono noti come **WS-***.
- SOAP è un *protocollo* per la comunicazione tra servizi.
- SOAP sfrutta il protocollo *HTTP*, utilizzandolo come semplice protocollo di trasporto (potrebbe utilizzare anche altri protocolli).
- Prevede lo standard *WSDL* (*Web Services Description Language*) per definire l'interfaccia di un servizio.
  - Un servizio SOAP è auto-descrittivo (molto rigoroso).
- Viene usato il formato *XML* per la rappresentazione dei dati.
- Quando si invia una richiesta a un'API SOAP, è necessario racchiudere la richiesta HTTP in una *busta SOAP.*
- Utilizza gli *XML Schema Definition* (*XSD*) per definire qual è la struttura corretta di un XML.

HTTP Request

- I metodi di una richiesta HTTP sono:
  - **GET**: lettura di uno o più oggetti
  - **POST**: creazione di un oggetto
  - **PUT**: aggiornamento di un oggetto esistente
  - **PATCH**: aggiornamento di una parte di un oggetto
  - **DELETE**: eliminazione di un oggetto
- La richiesta può contenere informazioni aggiuntive da inviare al server, all'interno delle intestazioni (*header*), per esempio per autenticarsi, e altri parametri/dati che costituiscono il corpo della richiesta (body)

HTTP Response

- La risposta HTTP ottenuta contiene un *status code,* per indicare se la richiesta è andata a buon fine oppure per restituire un particolare tipo di errore.
  - 200 OK: La richiesta è andata a buon fine.
  - 201 CREATED: La richiesta è andata a buon fine e ha come risultato la creazione di una nuova risorsa.
  - 301 MOVED PERMANENTLY: La risorsa non è più disponibile all'URL utilizzato; può contenere in risposta il nuovo indirizzo da utilizzare.
  - 400 BAD REQUEST: La richiesta non è andata a buon fine perché non è stata correttamente eseguita; per esempio perché mancano dei parametri oppure non sono nel formato corretto.
  - 401 UNAUTHORIZED: La richiesta è valida ma non è autorizzata: il client non ha inviato le credenziali corrette (per esempio un token di autorizzazione).
  - 403 FORBIDDEN: Il client non ha i permessi necessari per ottenere la risposta.
  - 404 NOT FOUND: La risorsa richiesta non è stata trovata.
  - 405 METHOD NOT ALLOWED: Il server conosce la risorsa richiesta, ma la risorsa non supporta il verbo HTTP usato nella richiesta.
  - 415 UNSUPPORTED MEDIA TYPE: Si verifica tipicamente quando il payload della richiesta oppure l'intestazione *Content-Type* non è nel formato corretto.
  - 500 INTERNAL SERVER ERROR: Si è verificato un errore sul server e la richiesta non può essere elaborata.
  - 503 SERVICE UNAVAILABLE: La richiesta non può essere elaborata, perché il server è occupato/sovraccarico.

Web Api RESTful in ASP.NET Core

22 novembre 2023

ASP.NET Core Web API

- Ricapitolando, un servizio RESTful si occupa di ricevere richieste HTTP dal client, elaborarle e restituire una risposta HTTP, contenete dati in formato JSON.
- I servizi REST in ASP.NET Core posso essere implementati basandosi sui *Controller* di *MVC* oppure nel formato *Minimal APIs*.

- Creiamo un progetto Web API, sfruttando l'architettura basata sui controller, partendo dal template ***webapi***:

  *> dotnet **new webapi** -n EsempioWebApi*

  Si consigli però di utilizzare il template **ASP.NET Core Web Api** fornito da *Visual Studio*.

Program.cs – Entry point

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment()) {
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection(); // -> Aggiunge il middleware per reindirizzare le richieste HTTP verso HTTPS
app.UseAuthorization(); // -> Aggiunge il middleware per le funzionalità di autorizzazione
app.MapControllers();
app.Run(); // -> Esegue l'host e mette il server in asolto
```

Program.cs – Entry point

- *builder.Services.**AddControllers**();*

Aggiunge (registra) i servizi per i *controller* nello specifico *Microsoft.Extensions.DependencyInjection.IServiceCollection* permettendo di utilizzare le classi **ControllerBase** (namespace *Microsoft.AspNetCore.Mvc* del pacchetto *Microsoft.AspNetCore.Mvc.Core*)**.**

- *builder.Services.**AddEndpointsApiExplorer**();*

Aggiunge il supporto alle *Minimal APIs*, anche se noi utilizzeremo i controller.

- app.**MapControllers**();

Aggiunge gli endpoint per le action dei controller: permettono di utilizzare le funzionalità di routing necessarie a inoltrare le richieste alle action.

Program.cs – Entry point

- *builder.Services.**AddSwaggerGen**();*

Aggiunge il supporto alle specifiche **OpenAPI** (**Swagger**), usando il pacchetto *Swashbuckle.AspNetCore.*

- Se siamo in ambiente di sviluppo, le istruzioni *app.**UseSwagger**()* e *app.**UseSwaggerUI**()* aggiungono il middleware per testare le API mediante un view autogenerata.

- L'ambiente di esecuzione dell'applicazione deriva dal valore della variabile d'ambiente *ASPNETCORE_ENVIRONMENT*, definita nel *launchSettings.json*.

Routing

- Ogni richiesta ricevuta dall'applicazione deve essere instradata verso il corrispondente endpoint: l'URL deve essere mappato in qualche modo su una action. Il mapping viene configurato per mezzo di appositi *template testuali* (*template di route*), che rappresentano le diverse possibili *route*.
- Sono possibili due diverse strategie di routing:
  - quella *convenzionale,* che utilizza appositi metodi di configurazione (*UseRouting()* e *MapControllerRoute()*) invocati nel Program.cs;
  - oppure quella *dichiarativa*, che invece funziona applicando appositi attributi sui controller (attributo *Route*) e sulle action.
- Le applicazioni MVC utilizzano entrambe le strategie, mentre le WebAPI sfruttano quella con gli attributi.

Template di route

- In un template di route, gli elementi racchiusi tra ***{}*** definiscono *segmenti variabili di route*, che vengono valorizzati se il particolare template trova corrispondenza.
- Un segmento variabile è *opzionale* se ha il suffisso ***?***, e quindi si avrà corrispondenza anche se nell'URL non viene specificato. Per esempio:

  *nomeAction/{parametro**?**}*
- ***[controller]*** e ***[action]*** sono due *token* speciali che saranno sostituiti con i nomi reali del controller e della action.
- Possiamo anche definire dei *vincoli* sui segmenti (parametri) che compongono un template di route, per esempio:
  - *{parametro**:int**}*
  - *{parametro**:alpha**}*
  - *{parametro**:bool**}*

Controller

- Nei progetti WebApi i controller ereditano dalla classe astratta **ControllerBase**, mentre nei progetti *MVC* i controller, che lavorano con le *view*, ereditano dalla classe astratta **Controller**, che a sua volta eredita da *ControllerBase*).

Esempio del Controller generato dal template:

```
namespace EsempioWebApi.Controllers {
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase {
        //…
     }
}
```

Controller

- L'attributo **ApiController** indica che il controller espone servizi API REST e rende obbligatoria la *gestione del routing* tramite la strategia dichiarativa, ovvero tramite l'altro attributo **Route**, che permette di specificare il *pattern di route principale* per il controller, ovvero permette di definire il template di route da applicare a tutte le action del controller.
- Nell'esempio, *[Route("[controller]")]* indica che il *pattern di route* principale del controller è ottenuto dal nome del controller, eliminando il suffisso *Controller. Quindi sarà: /WeatherForecast*
- Per avere il pattern *webapi/WeatherForecast* dobbiamo usare *[Route("webapi/[controller]")]*

Controller

- Il *ControllerBase* fornisce una serie di property per la gestione della richiesta HTTP:
  - **HttpContext**: il contesto HTTP della richiesta;
  - **ModelState**: lo stato di validazione della richiesta e del model (validazione del *model-binding);*
  - **Request**: l'oggetto *HttpRequest* che rappresenta la richiesta;
  - **Response**: l'oggetto *HttpResponse* che rappresenta la risposta;
  - **RouteData**: dati di routing dell'attuale action, estratti dall'URL;
  - **User**: l'utente associato alla richiesta attuale; ritorna un oggetto *ClaimsPrincipal* (namespace e pacchetto *System.Security.Claims*).

  Tutte queste property sono ricavate dalla property **ControllerContext.**
- In caso di errori di validazione sul model, ovvero quando *ModelState.**IsValid*** ritorna *false*, l'attributo **ApiController** provoca la restituzione (in automatico) di una risposta HTTP con status *400* (*BAD REQUEST*), equivalente a restituire ***BadRequest****(ModelState)*, dove *BadRequest* è un metodo public del *ControllerBase.*

Action

- Ogni action di un controller deve aver specificato il verbo HTTP che supporta, mediante uno specifico attributo:
  - **HttpGet** -> corrisponde a ***AcceptVerbs("get")***
  - **HttpPost**
  - **HttpPut**
  - **HttpDelete**
  - **HttpPatch**
  - **HttpHead**
  - **AcceptVerbs**: specifica che una stessa action può rispondere a più verbi; i verbi accettati sono ***"get"***, ***"post"***, ***"put"***, ***"options"***, ***"patch"***, ***"delete"*** e ***"head".***
- Questi attributi possono contenere anche dei parametri di *routing*, che saranno inviati alla action tramite l'URL usato per invocarla, per esempio: *[HttpGet("{id}")]*
- Ogni combinazione di verbo HTTP e di URL deve essere univoca, cioè ogni pattern di route deve essere mappato a una sola action del controller.

Action

- Le action posso ritornare differenti tipi di ritorno: tipi semplici o complessi. Gli oggetti o collezioni di oggetti, vengono automaticamente serializzati in formato *JSON*.
- Una action può però restituire differenti tipi di risultato in base allo status code HTTP restituito. Per questi casi è possibile usare la classe **ActionResult** oppure l'interfaccia **IActionResult**.
- Alcuni tipi che implementano *IActionResult* e sono utili per ritornare uno status code specifico sono:
  - ***OkResult***
  - ***BadRequestResult***
  - ***NotFoundREsult***
- Il *ControllerBase* espone dei metodi helper per ritornare questi oggetti, come per esempio il metodo ***Ok()*** che restituisce un oggeto *OkResult*.
- Un altro possibile tipo di ritorno è **ActionResult<T>** in cui il parametro di tipo T è la classe dell'oggetto che verrà restituito dalla action.

Action asincrone

- Possiamo implementare le action in modalità *asincrona* sfruttando il pattern async/await e ritornando il tipo **Task<ActionResult>** o **Task<IActionResult>**.

- Le action asincrone non sono più efficienti in termini di tempo di esecuzione, ma permettono più chiamate contemporanee da parte del client.

- I thread di esecuzione di ASP.NET Core possono gestire altre richieste mentre attendono il completamento di richieste precedenti.

Model Binding

- Le action possono ricevere e restituire dati che rappresentano oggetti complessi (che nel nostro progetto chiameremo *DTO (Data Transfer Object*) e in questo caso vengono serializzati in formato *JSON* utilizzando la classe a cui si riferisce l'oggetto. Questo processo prende il nome di **Model Binding**.
- Se la classe che deve essere serializzata contiene delle property che non devono essere inserite nel JSON, possiamo applicare a tali property l'attributo **JsonIgnore**.
- Tramite specifici attributi, è possibile indicare esplicitamente la provenienza dei parametri di una action:
  - **FromBody**: lettura del parametro dal *corpo* della richiesta HTTP;
  - **FromForm**: lettura del parametro dai dati *Form* contenuti nella richiesta HTTP;
  - **FromHeader**: lettura del parametro dagli *Header* della richiesta HTTP;
  - **FromQuery**: lettura del parametro dalla *Query String*;
  - **FromRoute**: lettura del parametro dai *dati di Routing* nell'URL;
  - **FromServices**: il parametro (servizio) viene iniettato via *Dependency Injection*;

Dependency Injection e Controller

- I controller sfruttano la DI per ottenere le istanze dei servizi da utilizzare all'interno delle loro action.
- Un controller viene istanziato ogni volta che una sua action viene invocata da una richiesta HTTP. Quindi gli eventuali servizi passati in input al suo costruttore vengono iniettati nel controller via DI (**constructor injection**) a ogni richiesta. In questo modo tutte le sue action possono utilizzare questi servizi che sono stati iniettati via DI (e mantenuti, per esempio, in un campo *readonly* del Controller).
- In alternativa, possiamo iniettare un servizio ad una action direttamente nel momento in cui viene invocata la specifica action, utilizzando l'attributo **[FromServices]** sul parametro che deve essere iniettato via DI (**action method injection**). In questo modo rendiamo disponibile un servizio solo ad una action e non all'intero controller.
- È possibile ottenere un'istanza di un servizio (registrato) anche manualmente dal container dei servizi, tramite l'*HttpContext*:

  *HttpContext.**RequestServices**.**GetService**(typeof(NomeInterfacciaServizio))*

Dependency Injection e Controller

- L'*HttpContext* viene iniettato automaticamente via DI in ogni classe controller, ed è per questo accessibile mediante l'omonima property *HttpContext*.
- Per registrare un *HttpContext* (utile quando non siamo all'interno di un controller e dobbiamo accedere *all'HttpContext*) possiamo usare questo metodo di estensione (registra un'istanza dell'interfaccia *IHttpContextAccessor*):

  builder.Services.**AddHttpContextAccessor**();

  (namespace *Microsoft.Extensions.DependencyInjection* pacchetto *Microsoft.AspNetCore.Http*)

Servizi di configurazione

- Per leggere delle *opzioni di configurazioni* che dipendono dall'installazione specificata dell'applicazione, è utile utilizzare i *servizi di configurazione*.
- Nel file *appsettings.json* (*appsettings.Development.json*) possiamo definire delle configurazioni/dei dati, che possono essere automaticamente lette e deserializzate in maniera fortemente tipizzata, ovvero trasformandoli in istanze di specifiche classi definite da noi.
- Nel *Program.cs* dobbiamo aggiungere questa istruzione (nella fase in cui vengono registrati i servizi):

  *builder.**Services**.**Configure**<NomeClasse>(builder.**Configuration**)*

  oppure

  *builder.**Services**.**Configure**<NomeClasse>(builder.**Configuration**.**GetSection**("NomeSection"))*

  Questo metodo leggerà automaticamente l'*appsettings.json* creando l'istanza della classe *NomeClasse* e registrandola come servizio.
- Ora per poter accedere e usare questa istanza, dobbiamo aggiungere (per esempio) un nuovo parametro al costruttore di un controller, in modo che l'istanza creata in precedenza sia iniettata via DI, usando il tipo ***IOptions***<*NomeClasse*> (namespace e pacchetto *Microsoft.Extensions.Options*).

Lettura configurazioni tramite IConfiguration

- Per leggere una *connection string* da Program.cs abbiamo diverse possibilità:
  - *builder.Configuration[**"ConnectionStrings:**NomeDbContext**"**];*
  - *builder.Configuration.**GetConnectionString**("NomeDbContext");*
  - *builder.Configuration.**GetSection****("ConnectionStrings")***["NomeDbContext"];*

- Per leggere una configurazione in maniera fortemente tipizzata:

  ***IConfigurationSection** confSection = builder.Configuration.GetSection("NomeSection");*

  *Nomeclasse options = confSection.**Get**<Nomeclasse>() ?? new();*

Automapper

- **Automapper** è una libreria per eseguire il mapping (la conversione) tra istanze di classi diverse: utile per convertire istanze di *DTO* in *Model* e viceversa.

- Aggiungere il riferimento al pacchetto *AutoMapper.Extensions.Microsoft.DependencyInjection*.

- Aggiungiamo un nuovo file *.cs* in cui definiamo le mappature:

```
[DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicConstructors)]
class NomeProfile : Profile {
    public NomeProfile() {
        CreateMap<NomeModel, NomeDto>();
        CreateMap<NomeDto, NomeModel >().AfterMap((src, dest) => dest.Validate());
        CreateMap<Nome2Dto, Nome2Model>()
            .ForMember(dest => dest.CodeType, opt => opt.MapFrom(src => src.BuyerIdType));
    }
}
```

Automapper

- In questo file .cs definiamo anche una classe internal, chiamata per esempio *AssemblyMarker*, che usiamo come maker per individuare l'assembly che contiene le mappature.
- Per registrare le mappature definite in precedenza usiamo questa istruzione nel *Program.cs: builder.Services.**AddAutoMapper**(typeof(AssemblyMarker));*
- Aggiungere ora un parametro di tipo ***IMapper*** (namespace *AutoMapper*) nel costruttore della classe che dovrà utilizzare le mappature: questo parametro sarà iniettato via DI.
- Ora possiamo utilizzare le mappature nel seguente modo (supponiamo che l'oggetto _map sia di tipo *IMapper*):

  *NomeDto dto = _map.**Map<**NomeDto**>(**model**);*

  oppure

  NomeModel model = new(){ NomeCampoSoloDelMoldel = 5 };

  *_map.**Map(**dto**,** model**);* (il primo parametro è la sorgente, il secondo la destinazione)

Controlli sulla Serializzazione

- Nel namespace ***System.Text.Json.Serialization*** sono definite le seguenti interfacce:
  - ***IJsonOnDeserializing***
  - ***IJsonOnDeserialized***
  - ***IJsonOnSerializing***
  - ***IJsonOnSerialized***

  Sono utili per eseguire delle operazioni, per esempio dei controlli (tramite *Reflection*), prima e/o dopo della *deserializzazione* e prima e/o dopo della *serializzazione* di una classe.

Logging Provider

- Un *Logging Provider* è un oggetto che permette di memorizzare messaggi *log* e utilizzarli in fase di debug dell'applicazione.
- ***ILogger*** (namespace *Microsoft.Extensions.Logging*) è l'interfaccia implementata dai Logging Provider.
- Gli oggetti *ILogger* non vengono mai istanziati manualmente ma è il framework che si occupa di creare e aggiungere un insieme di Loggin Provider predefiniti al contenitore dei servizi quando viene invocato il metodo *WebApplication.CreateBuilder().*
- Esempio di utilizzo (_logger è un'istanza di ILogger):

  *_logger.**LogInformation**("log di prova");*

  *_logger.**Log**(**LogLevel**.**Information**, "log di prova");*
- Le configurazioni per il logging sono definite nella sezione *Loggin* dell'*appsettings.json*