EBook Gratuito

APPRENDIMENTO

asp.net-core

Free unaffiliated eBook created from
Stack Overflow contributors.

#asp.net-core

Sommario

Di

Capitolo 1: Iniziare con asp.net-core

Osservazioni

.NET Core è una piattaforma di sviluppo generica gestita da Microsoft e dalla comunità .NET su GitHub. È multipiattaforma, supporta Windows, macOS e Linux e può essere utilizzata in scenari dispositivo, cloud e embedded / IoT.

Le seguenti caratteristiche definiscono meglio .NET Core:

- Implementazione flessibile: può essere inclusa nella tua app o installata side-by-side per utente o macchina.
- Multipiattaforma: funziona su Windows, macOS e Linux; può essere portato su altri sistemi operativi. I sistemi operativi (OS) supportati, le CPU e gli scenari applicativi cresceranno nel tempo, forniti da Microsoft, altre società e singoli individui.
- Strumenti da riga di comando: tutti gli scenari di prodotto possono essere esercitati sulla riga di comando.
- Compatibile: .NET Core è compatibile con .NET Framework, Xamarin e Mono, tramite la libreria standard .NET.
- Open source: la piattaforma .NET Core è open source, utilizzando le licenze MIT e Apache 2. La documentazione è concessa in licenza CC-BY. .NET Core è un progetto .NET Foundation.
- Supportato da Microsoft: .NET Core è supportato da Microsoft, per .NET Core Support

Versioni

Versione	Note di rilascio	Data di rilascio
RC1 *	1.0.0-rc1	2015/11/18
RC2 *	1.0.0-RC2	2016/05/16
1.0.0	1.0.0	2016/06/27
1.0.1	1.0.1	2016/09/13
1.1	1.1	2016/11/16

Examples

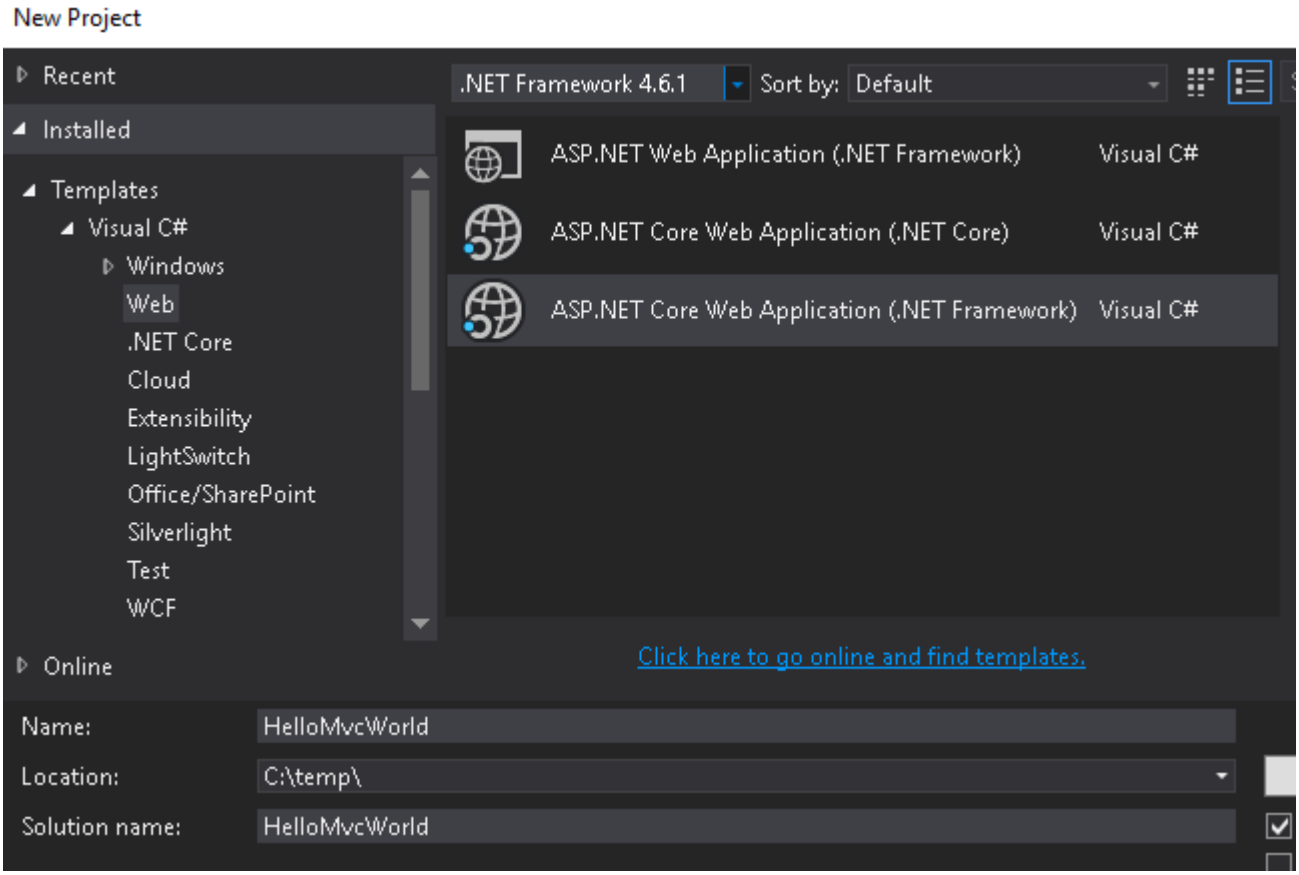
Installazione e configurazione

Installazione di Visual Studio

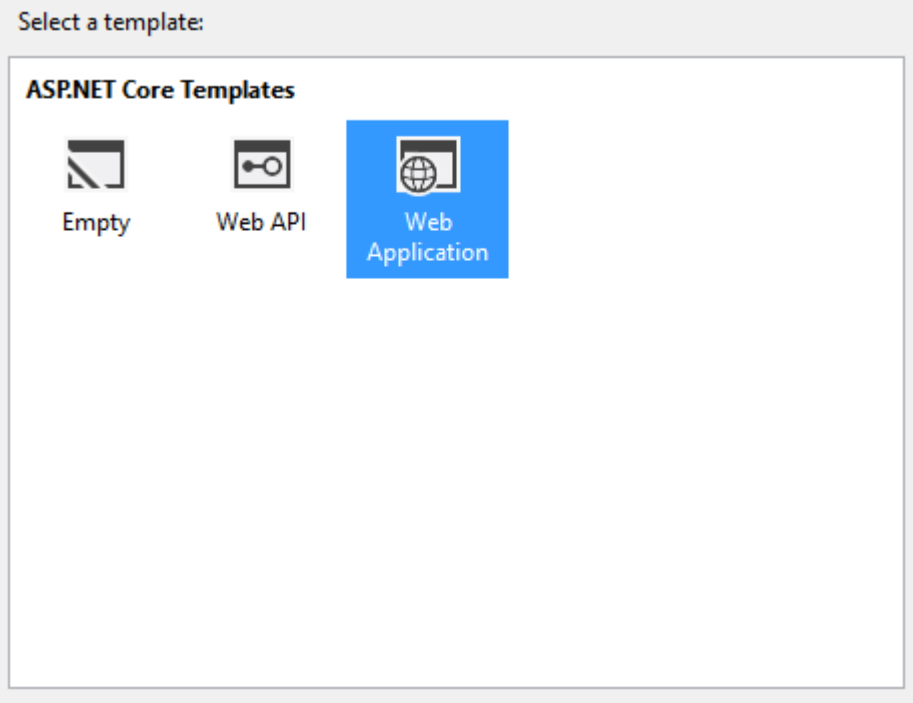
Se non hai installato Visual Studio, puoi scaricare qui la versione gratuita di Visual Studio Community . Se l'hai già installato, puoi procedere al passaggio successivo.

Creazione di un'applicazione ASP.NET Core MVC.

1. **Apri Visual Studio.**
2. **Seleziona File> Nuovo progetto.**
3. **Seleziona Web nella lingua desiderata** nella sezione Modelli a sinistra.
4. **Scegli un tipo di progetto preferito** all'interno della finestra di dialogo.
5. **Opzionale: scegli un .NET Framework che desideri utilizzare come target**
6. **Assegna un nome al tuo progetto** e indica se vuoi creare una soluzione per il progetto.
7. **Fare clic su OK** per creare il progetto.



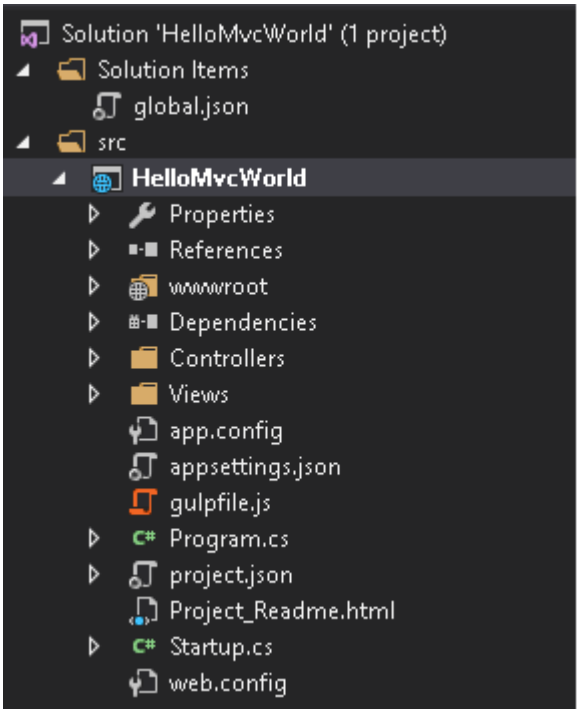
Ti verrà presentata un'altra finestra di dialogo per selezionare il modello che desideri utilizzare per il progetto:



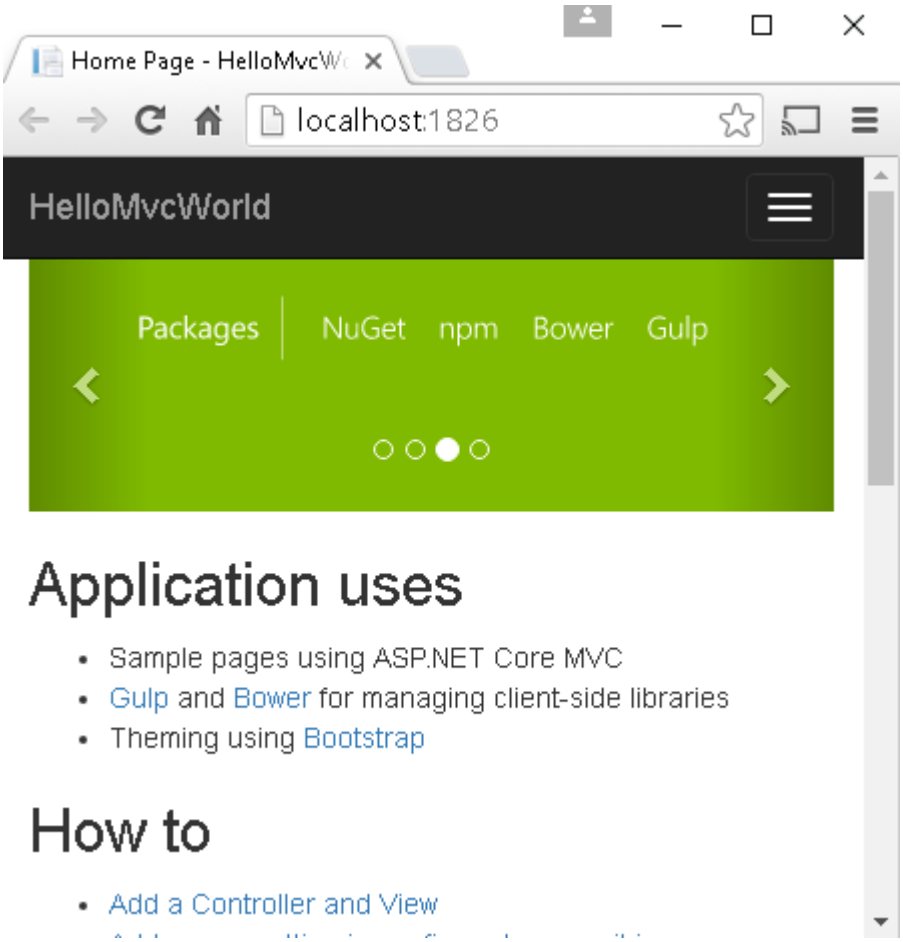
Ciascuna descrizione è autoesplicativa. Per questo primo progetto, **selezionare Applicazione Web** , che conterrà tutte le configurazioni predefinite, l'autenticazione e alcuni contenuti esistenti.

Poiché questa è un'applicazione di introduzione e non richiede alcuna protezione o autenticazione, è possibile **modificare l'opzione di autenticazione** su **Nessuna autenticazione** sul lato destro della finestra di dialogo e **fare clic su OK per creare il progetto** .

Dovresti quindi vedere il nuovo progetto all'interno di Solution Explorer:



Premere il tasto F5 per eseguire l'applicazione e iniziare una sessione di debug, che avvierà l'applicazione all'interno del browser predefinito:



Ora puoi vedere che il tuo progetto è attivo e funzionante a livello locale ed è pronto come punto di partenza per costruire la tua applicazione.

Crea un nuovo progetto dalla riga di comando

È possibile creare un nuovo progetto ASP.NET Core interamente dalla riga di comando utilizzando il comando `dotnet` .

```
dotnet new web
dotnet restore
dotnet run
```

`dotnet new web` scaffolds un nuovo progetto web "vuoto". Il parametro `web` indica allo strumento `dotnet` di utilizzare il modello `ASP.NET Core Empty` . Usa `dotnet new -all` per mostrare tutti i modelli disponibili correntemente installati. Altri modelli chiave includono `console` , `classlib` , `mvc` e `xunit` .

Una volta che il modello è stato impalcato, è possibile ripristinare i pacchetti richiesti per eseguire il progetto (`dotnet restore`), compilare e avviarlo (`dotnet run`).

Una volta che il progetto è in esecuzione, sarà disponibile sulla porta predefinita: http: // localhost: 5000

API Web ASP.NET di base minima con ASP.NET Core MVC

Con ASP.NET Core 1.0, il framework MVC e Web API sono stati uniti in un framework chiamato

ASP.NET Core MVC. Questa è una buona cosa, dal momento che MVC e Web API condividono molte funzionalità, tuttavia ci sono sempre state sottili differenze e duplicazione del codice.

Tuttavia, unire questi due nel framework uno ha reso più difficile distinguere gli uni dagli altri. Ad esempio, `Microsoft.AspNet.WebApi` rappresenta il framework API Web 5.xx, non il nuovo. Ma quando si include `Microsoft.AspNetCore.Mvc` (versione `1.0.0`), si ottiene il pacchetto completo. Questo conterrà *tutte* le funzionalità predefinite offerte dal framework MVC. Come Razor, tag helper e binding di modelli.

Quando vuoi solo creare un'API Web, non abbiamo bisogno di tutte queste funzionalità. Quindi, come possiamo costruire un'API Web minimalista? La risposta è: `Microsoft.AspNetCore.Mvc.Core` . Nel nuovo mondo MVC è suddiviso in più pacchetti e questo pacchetto contiene solo i componenti principali del framework MVC, come routing e autorizzazione.

Per questo esempio, creeremo un'API MVC minima. Compreso un formattatore JSON e CORS. Creare un'applicazione Web ASP.NET Core 1.0 vuota e aggiungere questi pacchetti al progetto.json:

```
"Microsoft.AspNetCore.Mvc.Core": "1.0.0",
"Microsoft.AspNetCore.Mvc.Cors": "1.0.0",
"Microsoft.AspNetCore.Mvc.Formatters.Json": "1.0.0"
```

Ora possiamo registrare MVC usando `AddMvcCore()` nella classe di avvio:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvcCore()
            .AddCors()
            .AddJsonFormatters();
}
```

`AddMvcCore` restituisce un'istanza `IMvcCoreBuilder` che consente ulteriori `IMvcCoreBuilder` . La configurazione del middleware è la stessa del solito:

```
public void Configure(IApplicationBuilder app)
{
    app.UseCors(policy =>
    {
        policy.AllowAnyOrigin();
    });
    app.UseMvc();
}
```

Controller

La "vecchia" API Web viene fornita con la propria classe di base del controller: `ApiController` . Nel nuovo mondo non esiste una cosa del genere, solo la classe `Controller` predefinita. Sfortunatamente, questa è una classe base piuttosto grande ed è legata al binding del modello, alle viste e a JSON.NET.

Fortunatamente, nelle nuove classi di controller del framework non è necessario derivare dal `Controller` per essere rilevato dal meccanismo di routing. Basta accodare il nome con `Controller` è sufficiente. Questo ci consente di costruire la nostra classe base controller. Chiamiamolo `ApiController` , solo per amore dei vecchi tempi:

```
/// <summary>
/// Base class for an API controller.
/// </summary>
[Controller]
public abstract class ApiController
{
    [ActionContext]
    public ActionContext ActionContext { get; set; }

    public HttpContext HttpContext => ActionContext?.HttpContext;

    public HttpRequest Request => ActionContext?.HttpContext?.Request;

    public HttpResponse Response => ActionContext?.HttpContext?.Response;

    public IServiceProvider Resolver => ActionContext?.HttpContext?.RequestServices;
}
```

L'attributo `[Controller]` indica che il tipo o qualsiasi tipo derivato è considerato come un controller dal meccanismo di rilevamento del controller predefinito. L'attributo `[ActionContext]` specifica che la proprietà deve essere impostata con `ActionContext` corrente quando MVC crea il controller. `ActionContext` fornisce informazioni sulla richiesta corrente.

> ASP.NET Core MVC offre anche una classe `ControllerBase` che fornisce una classe di base del controller senza supporto per le viste. È comunque molto più grande del nostro. Usalo se lo trovi conveniente.

Conclusione

Ora è possibile creare un'API Web minima utilizzando il nuovo framework MV.NET di ASP.NET. La struttura modulare del pacchetto ci consente di inserire solo i pacchetti di cui abbiamo bisogno e creare un'applicazione snella e semplice.

Utilizzo del codice di Visual Studio per lo sviluppo dell'applicazione core crossplate aspnet core

Con AspNetCore puoi sviluppare l'applicazione su qualsiasi piattaforma tra cui Mac, Linux, Window e Docker.

Installazione e configurazione

1. Installa Visual Studio Code da qui
2. Aggiungi estensione C #
3. Installa dot net core sdk. Puoi installare da qui

Ora hai tutti gli strumenti disponibili. Per sviluppare l'applicazione. Ora hai bisogno di un'opzione di

ponteggio. Per questo dovresti considerare l'utilizzo di Yeoman. Per installare Yeoman

1. Installa NPM. Per questo è necessario Nodo sulla macchina. Installa da qui
2. Installa Yeoman usando NPM

   npm install -g yo
3. Ora installa il generatore di aspnet

   npm install -g generator-aspnet

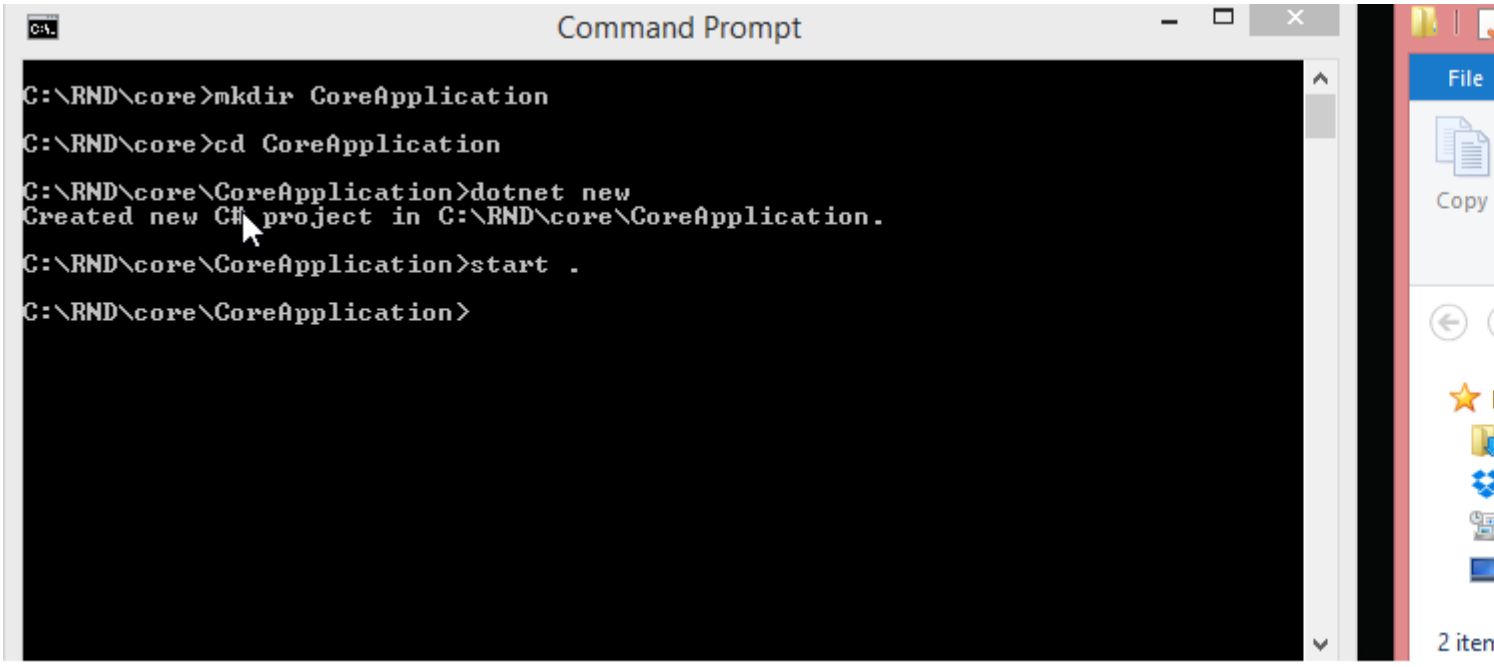
Ora abbiamo tutto il setup sulla tua macchina. Per prima cosa creiamo un nuovo progetto con il comando di base DotNetCore e poi creiamo un nuovo progetto usando Yo.

Nuovo progetto utilizzando la riga di comando

1. Crea una nuova cartella di progetto

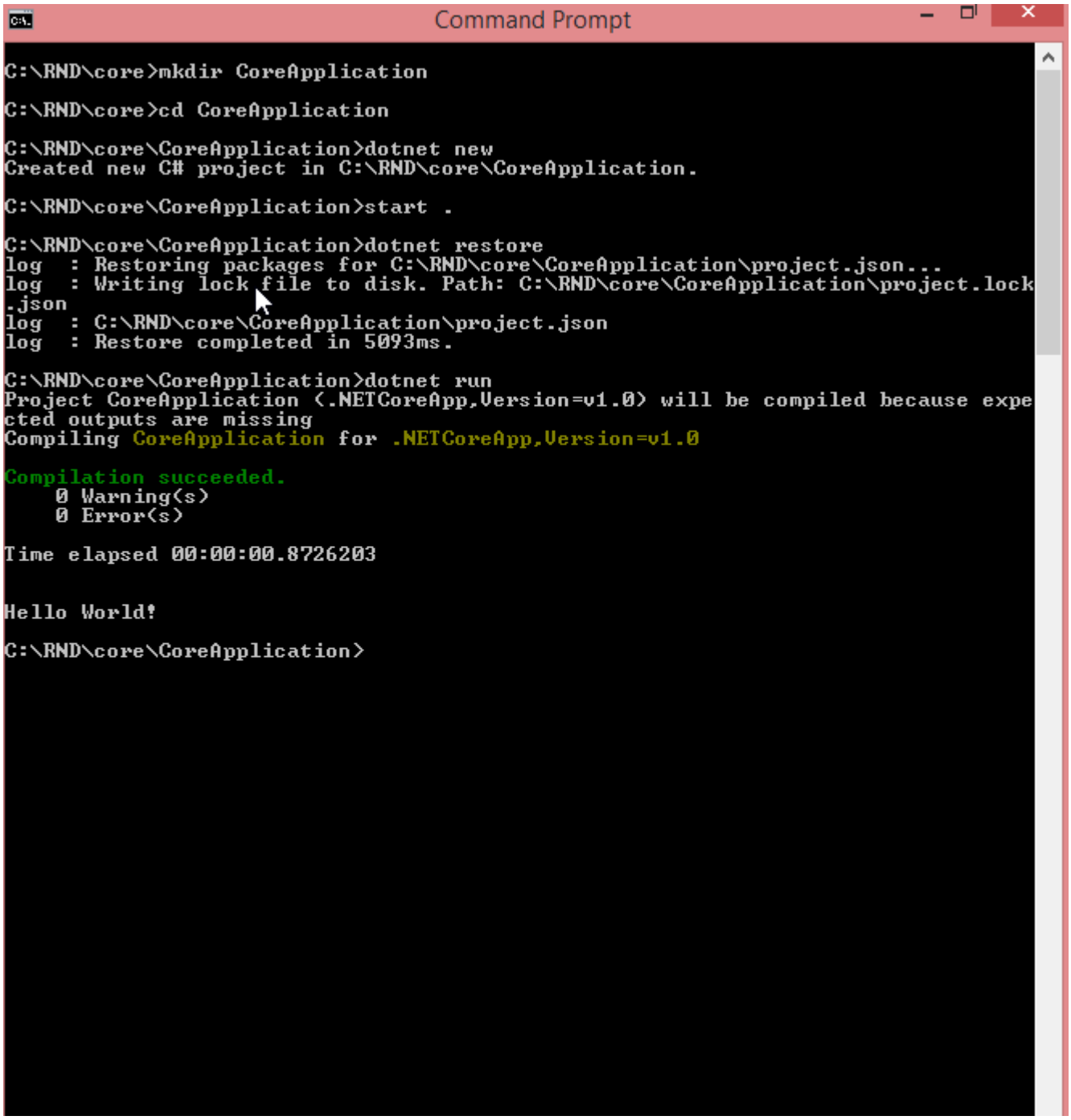
   mkdir CoreApplication cd CoreApplication
2. Scaffold un progetto dotnet molto semplice utilizzando l'opzione della riga di comando predefinita

   dotnet Nuovo



1. Ripristina i pacchetti ed esegui l'applicazione

   dotNet ripristino dotnet run

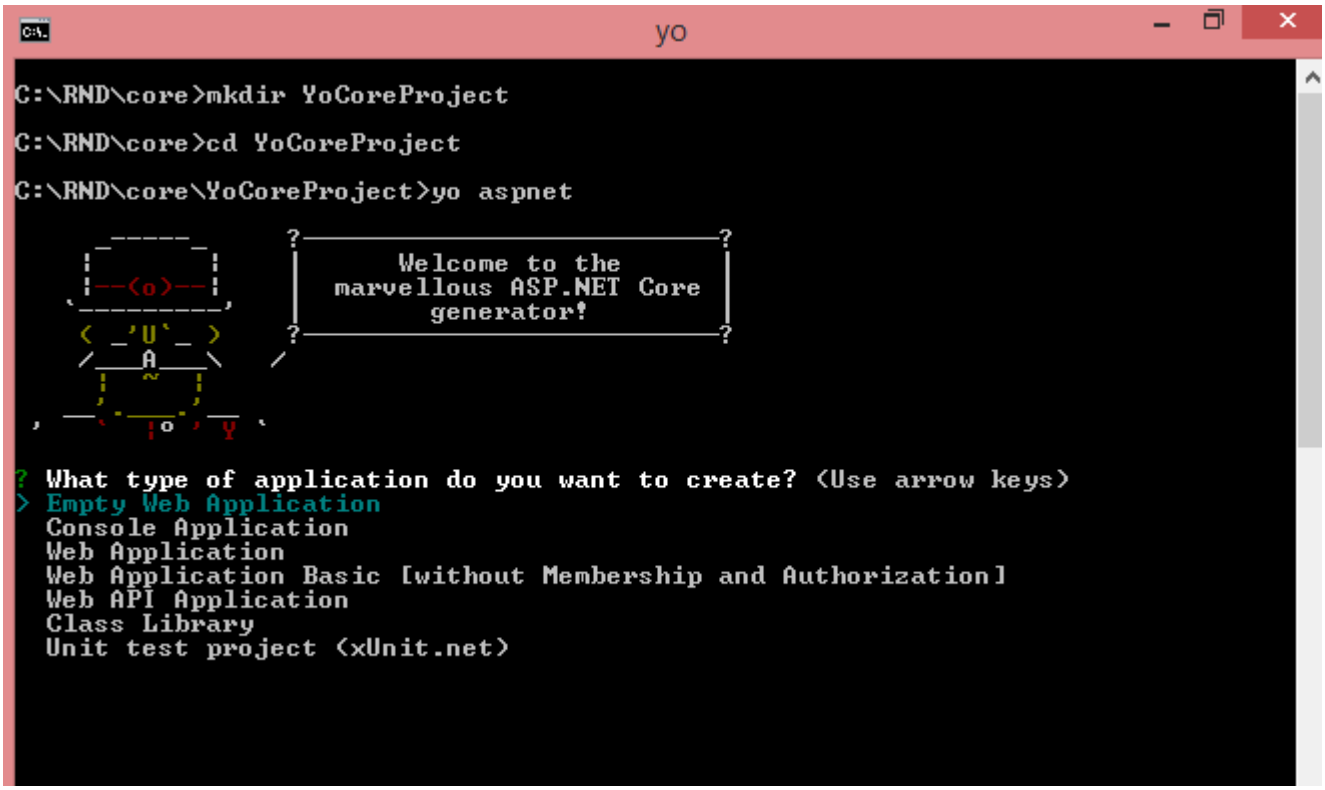


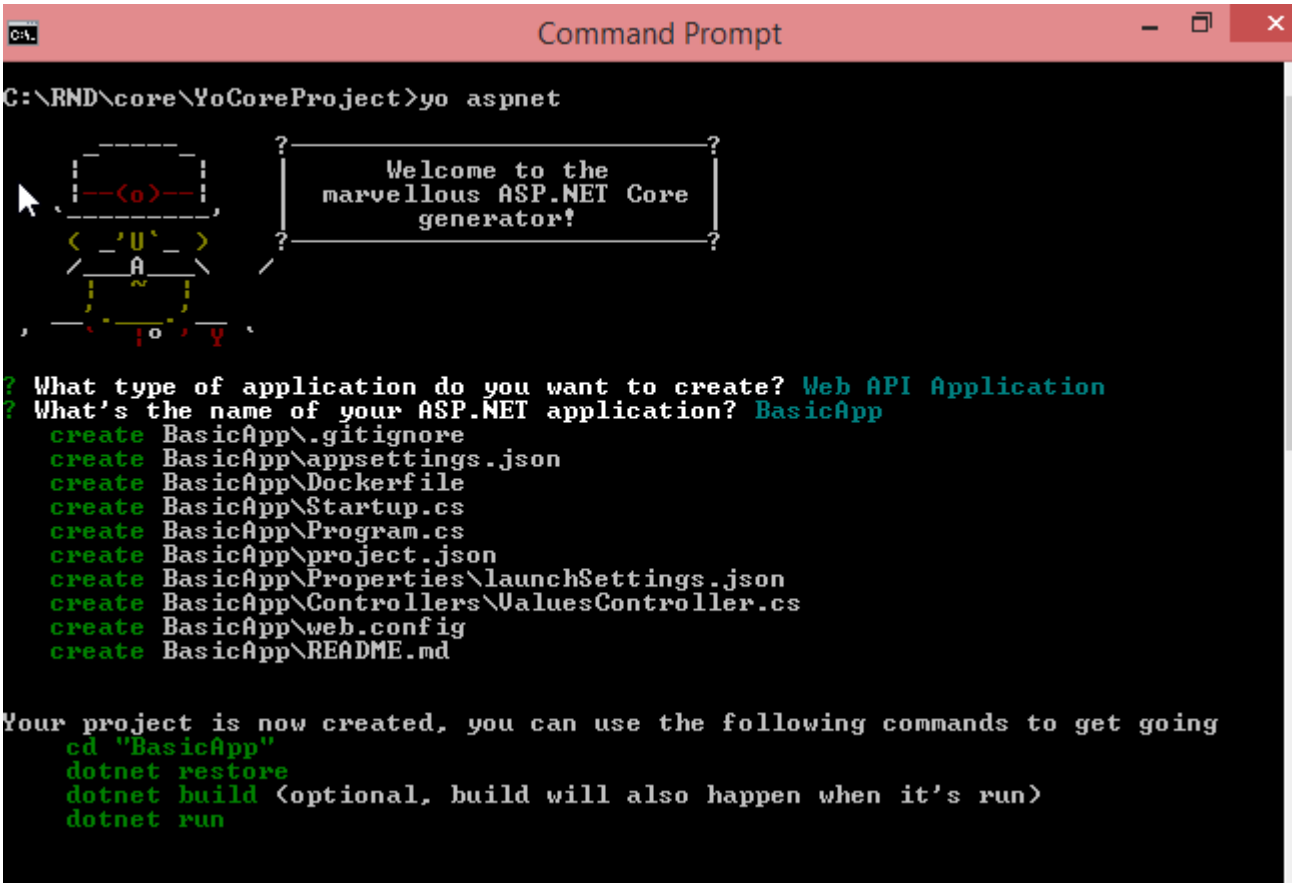
Usa Yeoman come Opzione Scaffolding

Crea la cartella del progetto ed esegui il comando Yo

```
yo aspnet
```

Yeoman chiederà alcuni input come Tipo di Progetto, Nome del Progetto ecc

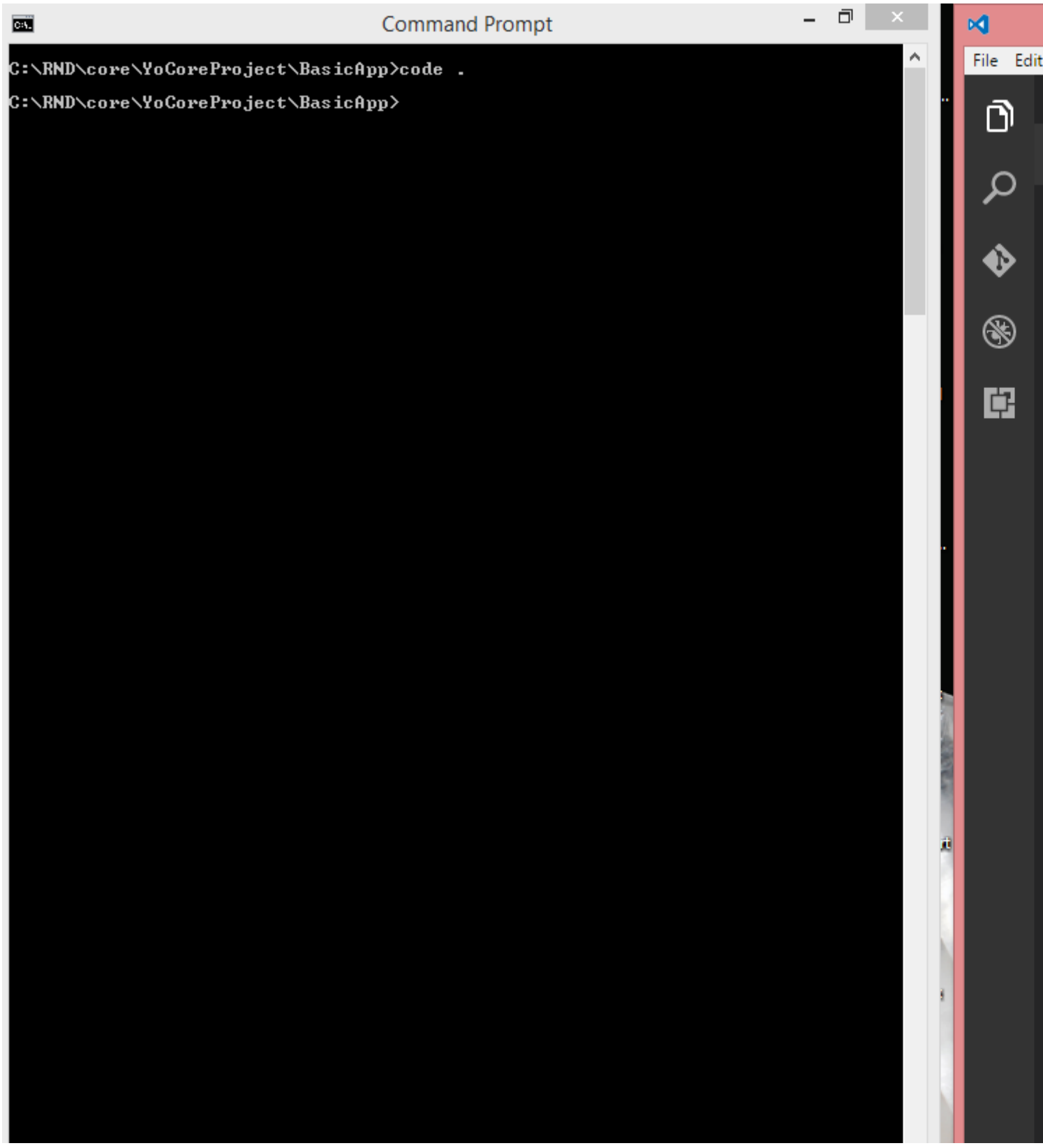




Ora ripristinare i pacchetti eseguendo il comando dotnet restore ed esegui l'applicazione

Utilizzare VS Code per sviluppare l'applicazione

Esegui il codice dello studio visivo come



Ora apri i file ed esegui l'applicazione. Puoi anche cercare l'estensione per il tuo aiuto.

Imposta la variabile di ambiente in ASP.NET Core [Windows]

=> Original Post <=

ASP.NET Core utilizza la variabile di ambiente `ASPNETCORE_ENVIRONMENT` per determinare l'ambiente corrente. Per impostazione predefinita, se si esegue l'applicazione senza impostare questo valore,

verrà automaticamente impostato automaticamente sull'ambiente di `Production` .

```
> dotnet run
Project TestApp (.NETCoreApp,Version=v1.0) was previously compiled. Skipping compilation.

Hosting environment: Production
Content root path: C:\Projects\TestApp
Now listening on: http://localhost:5000
Application started. Press Ctrl+C to shut down.
```

Impostazione della variabile di ambiente in Windows

Alla riga di comando

È possibile impostare facilmente una variabile di ambiente da un prompt dei comandi utilizzando il comando setx.exe incluso in Windows. Puoi usarlo per impostare facilmente una variabile utente:

```
>setx ASPNETCORE_ENVIRONMENT "Development"

SUCCESS: Specified value was saved.
```

Si noti che la variabile di ambiente non è impostata nella finestra aperta corrente. Sarà necessario aprire un nuovo prompt dei comandi per visualizzare l'ambiente aggiornato. È anche possibile impostare le variabili di sistema (anziché solo le variabili utente) se si apre un prompt dei comandi amministrativo e si aggiunge l'opzione / M:

```
>setx ASPNETCORE_ENVIRONMENT "Development" /M

SUCCESS: Specified value was saved.
```

Uso di PowerShell In alternativa, è possibile utilizzare PowerShell per impostare la variabile. In PowerShell, oltre alle normali variabili utente e di sistema, è anche possibile creare una variabile temporanea utilizzando il comando `$Env:` ::

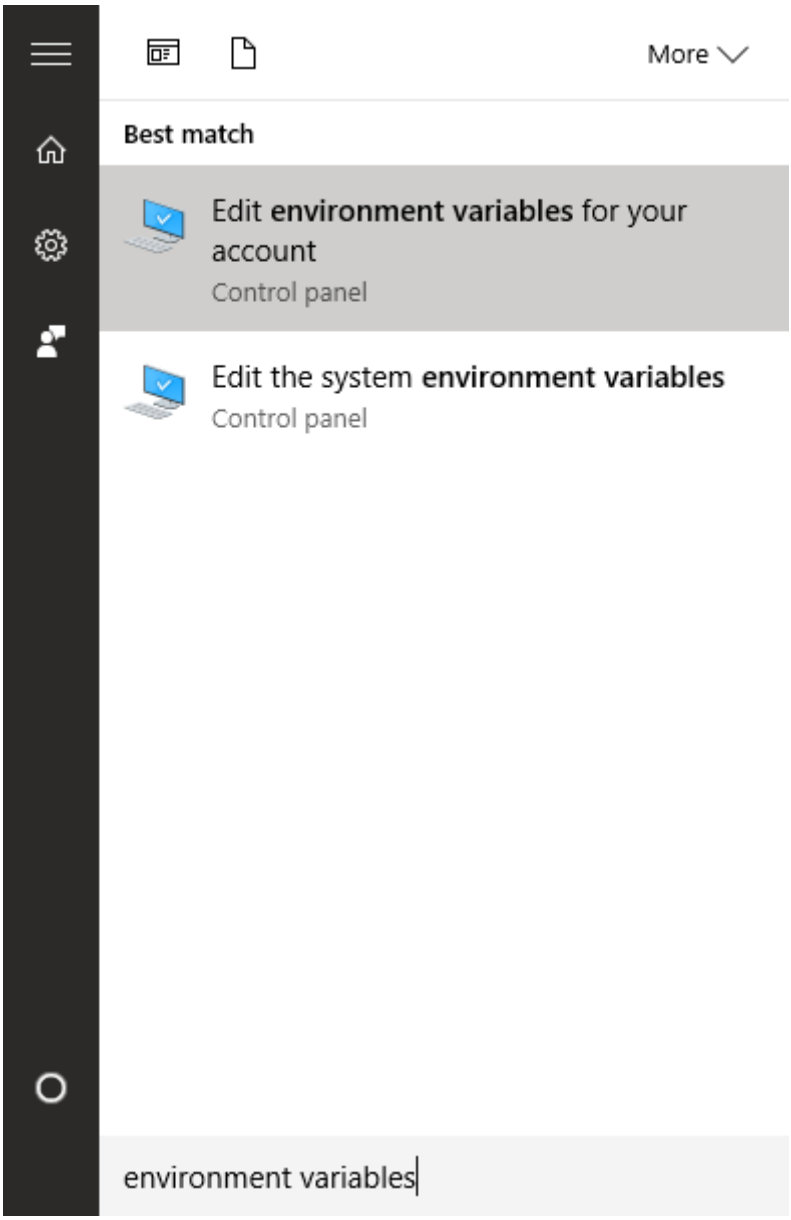
```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

La variabile creata dura solo per la durata della sessione di PowerShell: una volta chiusa la finestra, l'ambiente ritorna al valore predefinito.

In alternativa, è possibile impostare direttamente le variabili di ambiente utente o di sistema. Questo metodo non modifica le variabili di ambiente nella sessione corrente, quindi sarà necessario aprire una nuova finestra di PowerShell per visualizzare le modifiche. Come prima, la modifica delle variabili di sistema (macchina) richiederà l'accesso amministrativo

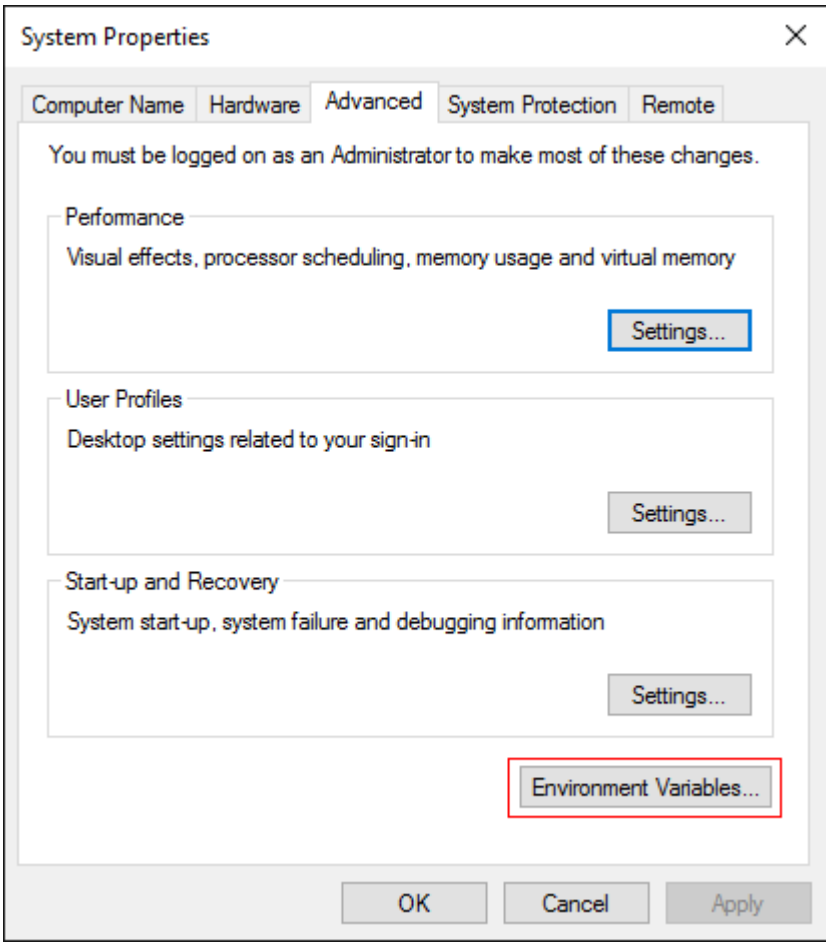
```
[Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Development", "User")
[Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Development", "Machine")
```

Utilizzo del pannello di controllo di Windows Se non sei un fan del prompt dei comandi, puoi facilmente aggiornare le variabili usando il tuo mouse! Fai clic sul pulsante del menu Start di Windows (o premi il tasto Windows), cerca `environment variables` e scegli *Modifica ambiente*

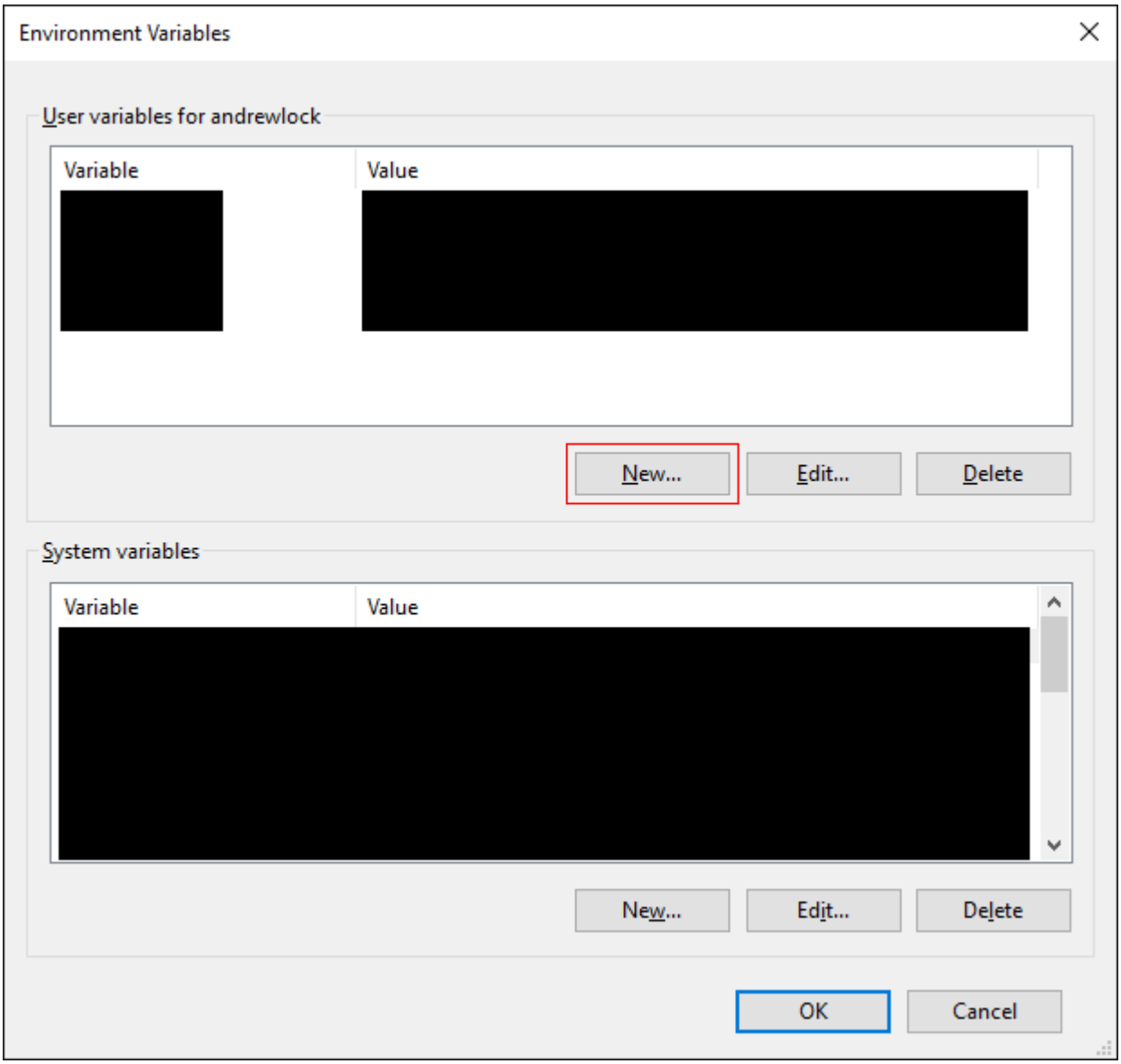


variabili per il tuo account:

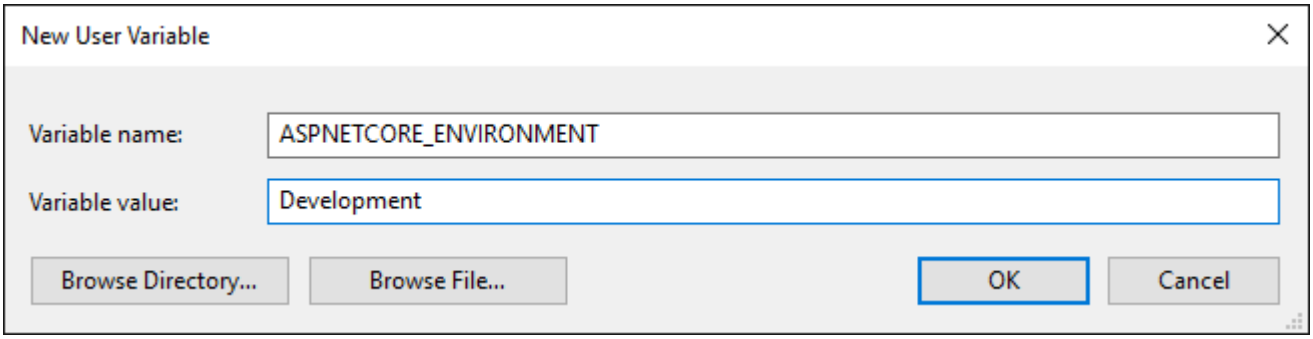
Selezionando questa opzione si aprirà la finestra di dialogo Proprietà del sistema



Fare clic su Variabili d'ambiente per visualizzare l'elenco delle variabili d'ambiente correnti sul proprio sistema.



Supponendo che tu non abbia già una variabile chiamata `ASPNETCORE_ENVIRONMENT` , fai clic sul pulsante Nuovo ... e aggiungi una nuova variabile di ambiente per l'account:



Fare clic su OK per salvare tutte le modifiche. Sarà necessario riaprire qualsiasi finestra di comando per assicurarsi che vengano caricate le nuove variabili di ambiente.

Leggi Iniziare con asp.net-core online: https://riptutorial.com/it/asp-net-core/topic/810/iniziare-con-asp-net-core

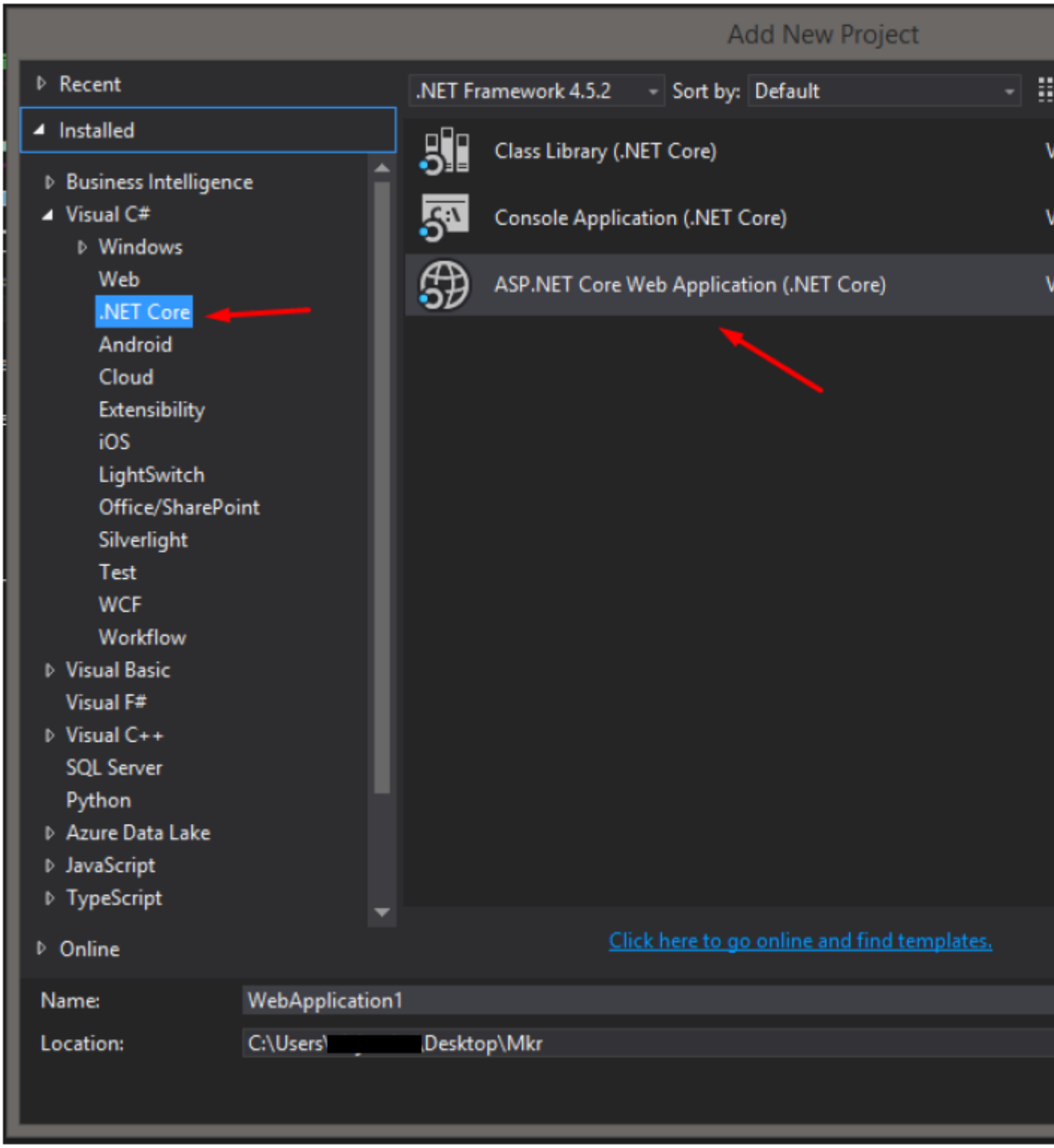
Capitolo 2: Angular2 e .Net Core

Examples

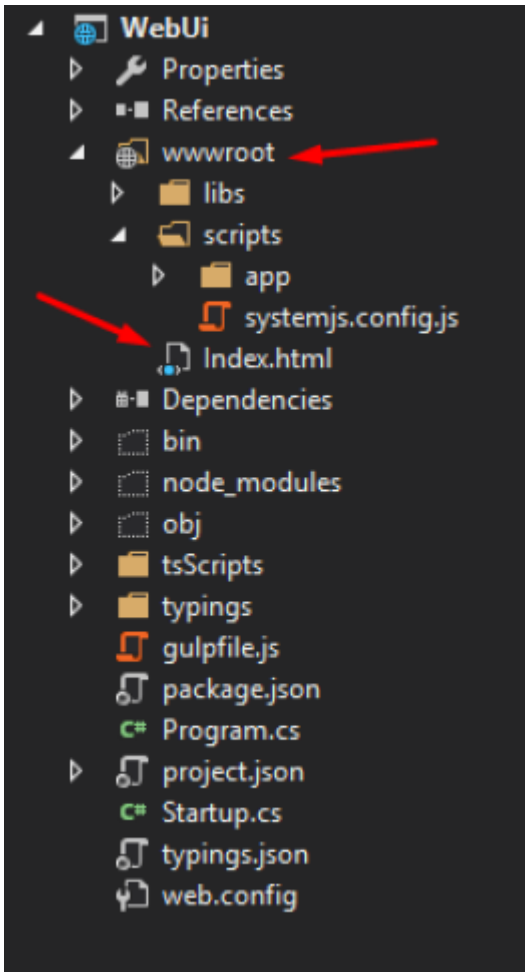
Tutorial rapido per Angular 2 Hello World! App con. Net Core in Visual Studio 2015

passi:

1. Creare un'app Web .Net core vuota:

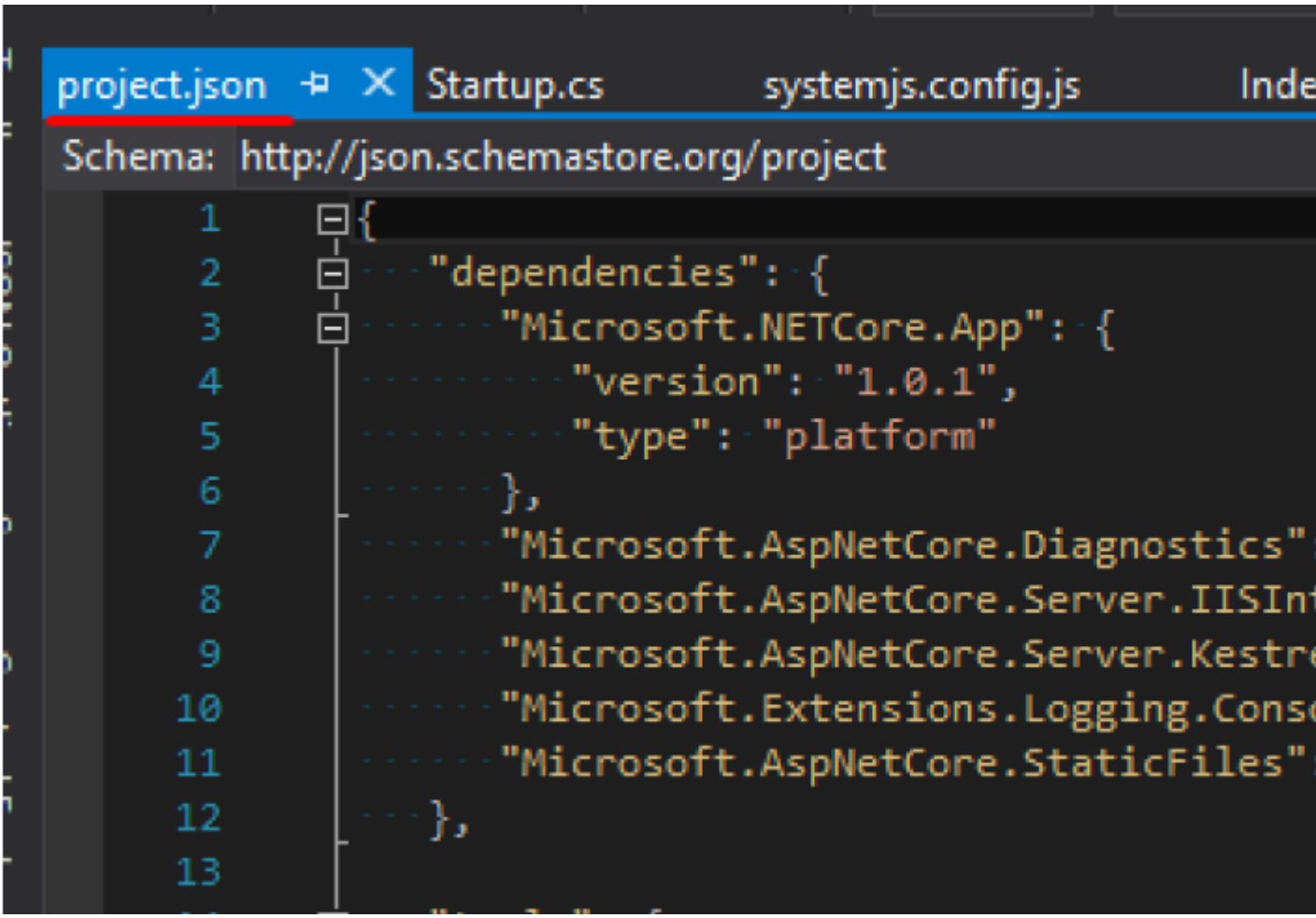


2. Vai a wwwroot e crea una normale pagina html chiamata Index.html:



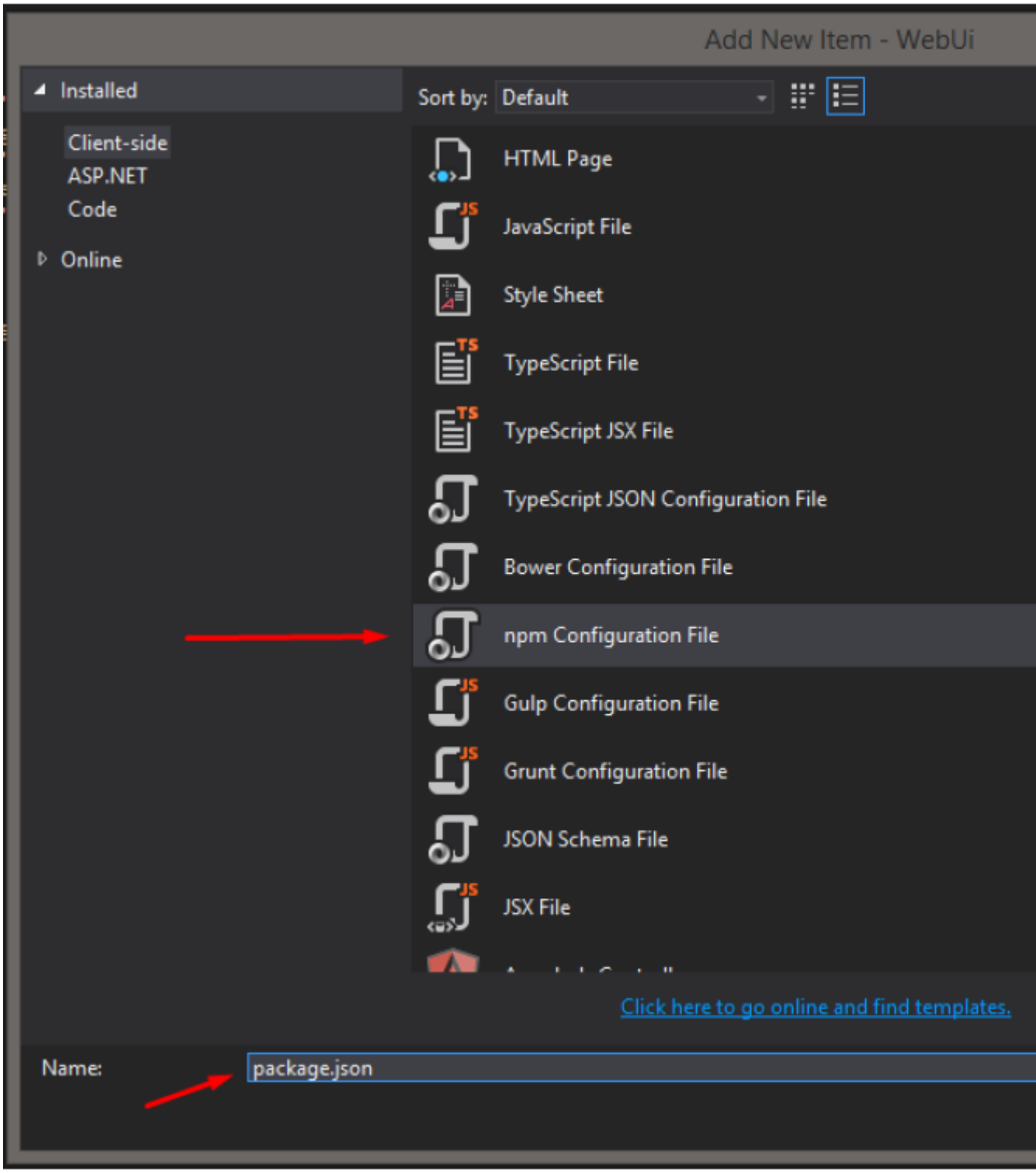
3. Configura Startup.cs per accettare file statici (questo richiederà di aggiungere "Microsoft.AspNetCore.StaticFiles": libreria "1.0.0" nel file "project.json"):

```
// This method gets called by the runtime. Use this method to configure the
-references
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILo
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```



4. Aggiungi file NPN:

    - Fare clic con il pulsante destro del mouse sul progetto WebUi e aggiungere il file di configurazione NPN (package.json):



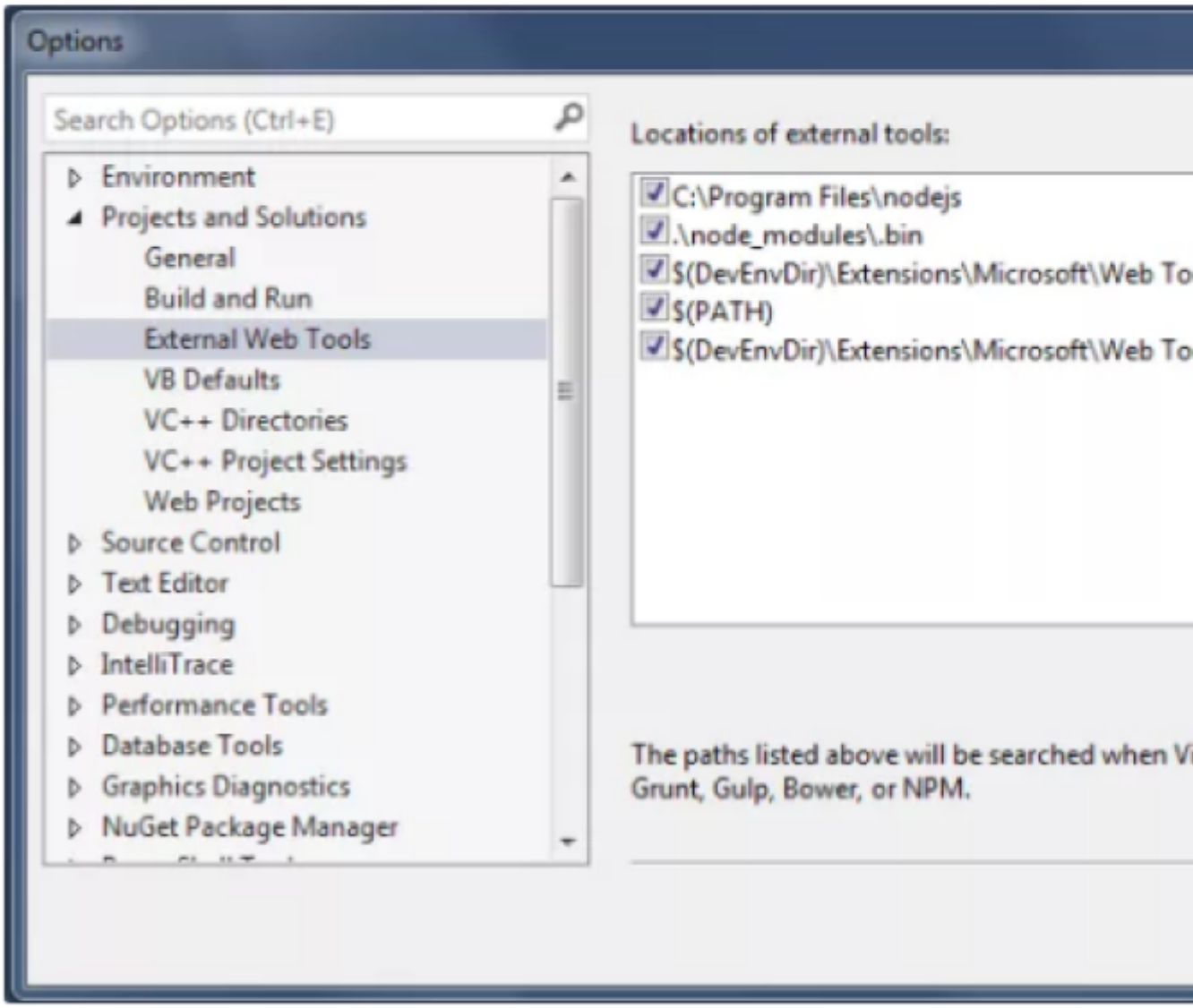
- Verifica le ultime versioni dei pacchetti:

Nota: se Visual Studio non rileva le versioni dei pacchetti (Controlla tutti i pacchetti, perché alcuni di essi mostrano la versione, e alcuni altri no), potrebbe essere perché la versione del Nodo in arrivo in Visual Studio non funziona correttamente , quindi sarà probabilmente necessario installare node js esternamente e quindi collegare l'installazione con Visual Studio.

io. Scarica e installa il nodo js: https://nodejs.org/es/download/

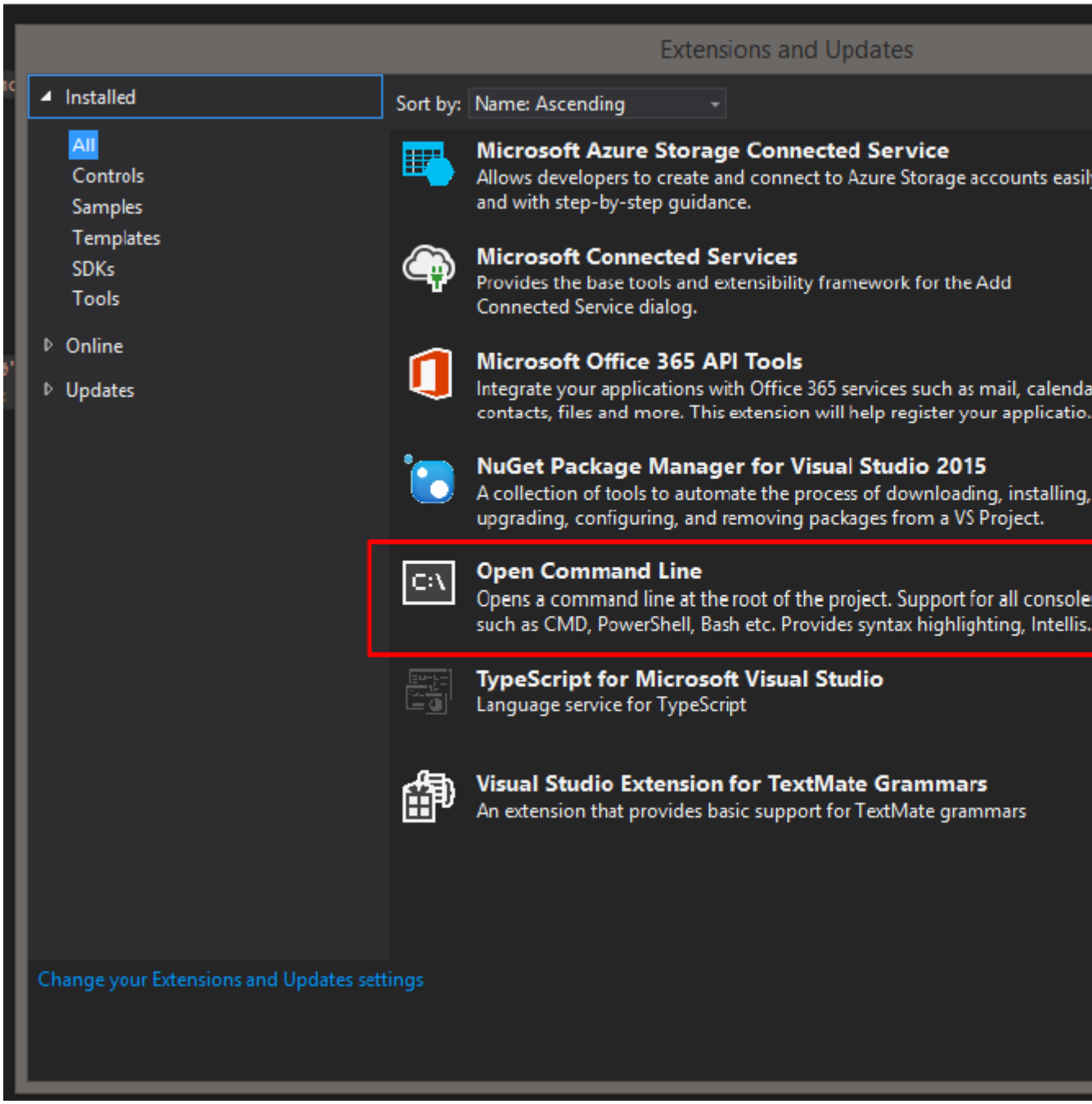
ii. Collega l'installazione con lo studio visivo: https://ryanhayes.net/synchronize-node-js-install-version-with-visual-studio-2015/ :

1. First, **find the Node.js installation you already have** and u
   By default, Node.js 0.12.7 installs to "C:\Program Files\nod
2. Once you've got that all copied out to your clipboard, got t
   Studio 2015.
3. In this dialog, go to **Projects and Solutions > External Web**
   that manages all of the 3rd party tools used within VS. Th
   pointed to.
4. **Add an entry at the top to the path to the node.js directory**
   use that version instead.



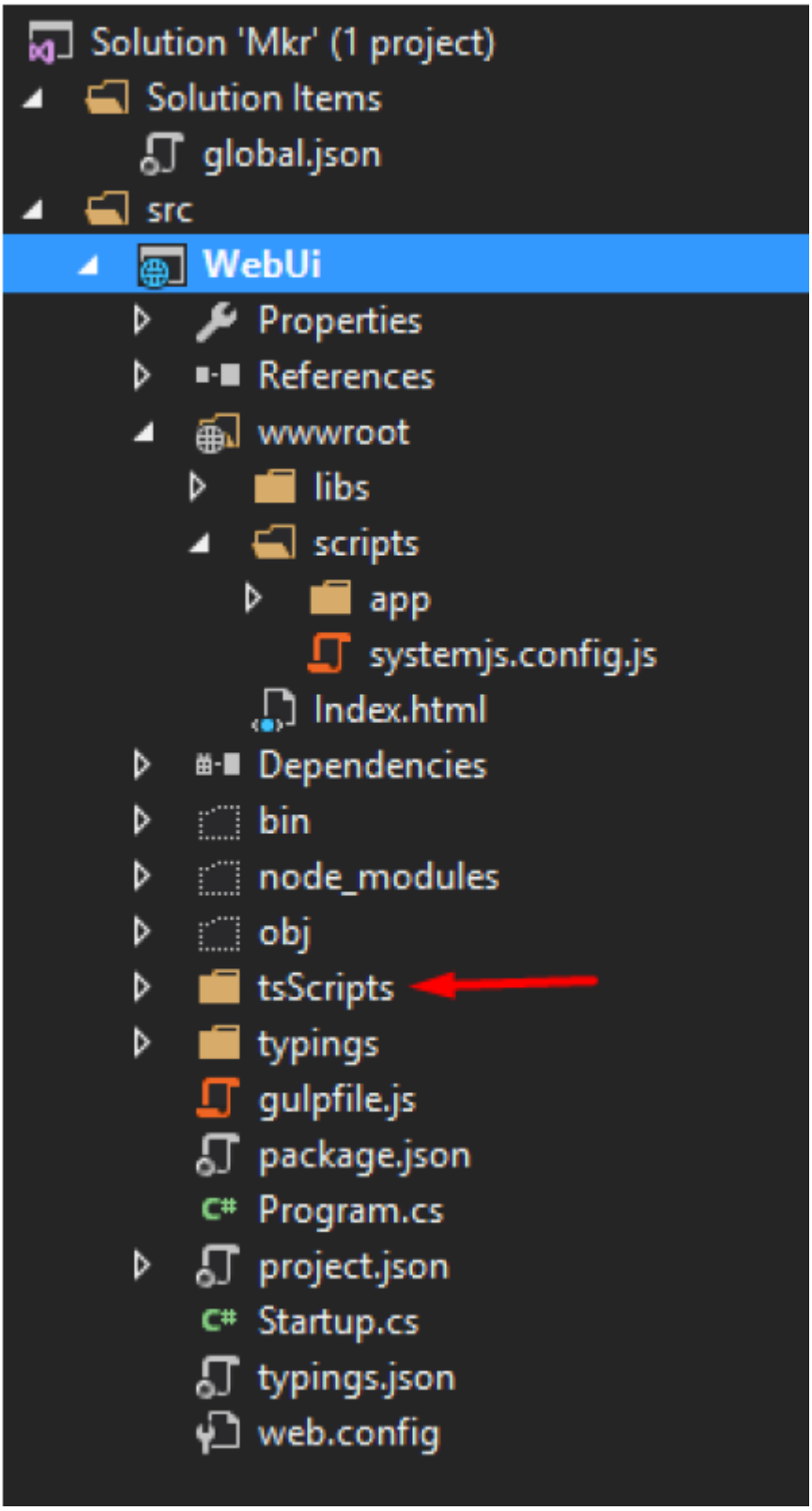
iii. (Facoltativo) dopo il salvataggio di package.json installerà le dipendenze nel progetto, in caso contrario, esegui "npm install" utilizzando un prompt dei comandi dalla stessa posizione del file package.json.

Nota: consigliato per installare "Apri riga di comando", un'estensione che può essere aggiunta a Visual Studio:

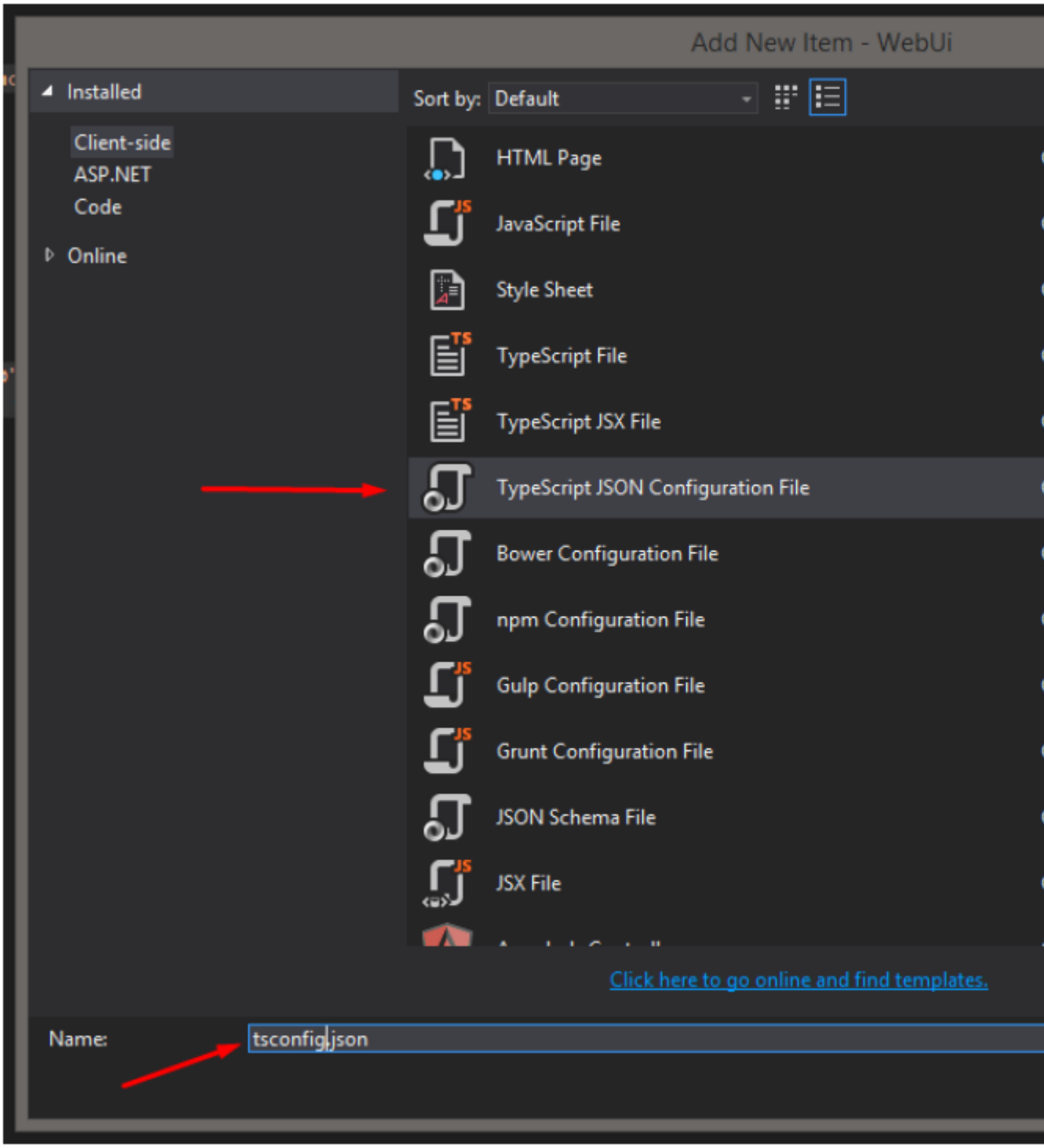


5. Aggiungi dattiloscritto:

   - Crea una cartella TsScript all'interno del progetto WebUi, solo per l'organizzazione (i TypeScripts non andranno al browser, saranno transpiled in un normale file JS, e questo file JS sarà quello che va al foder wwwroot usando gulp, questo sarà spiegato in seguito):

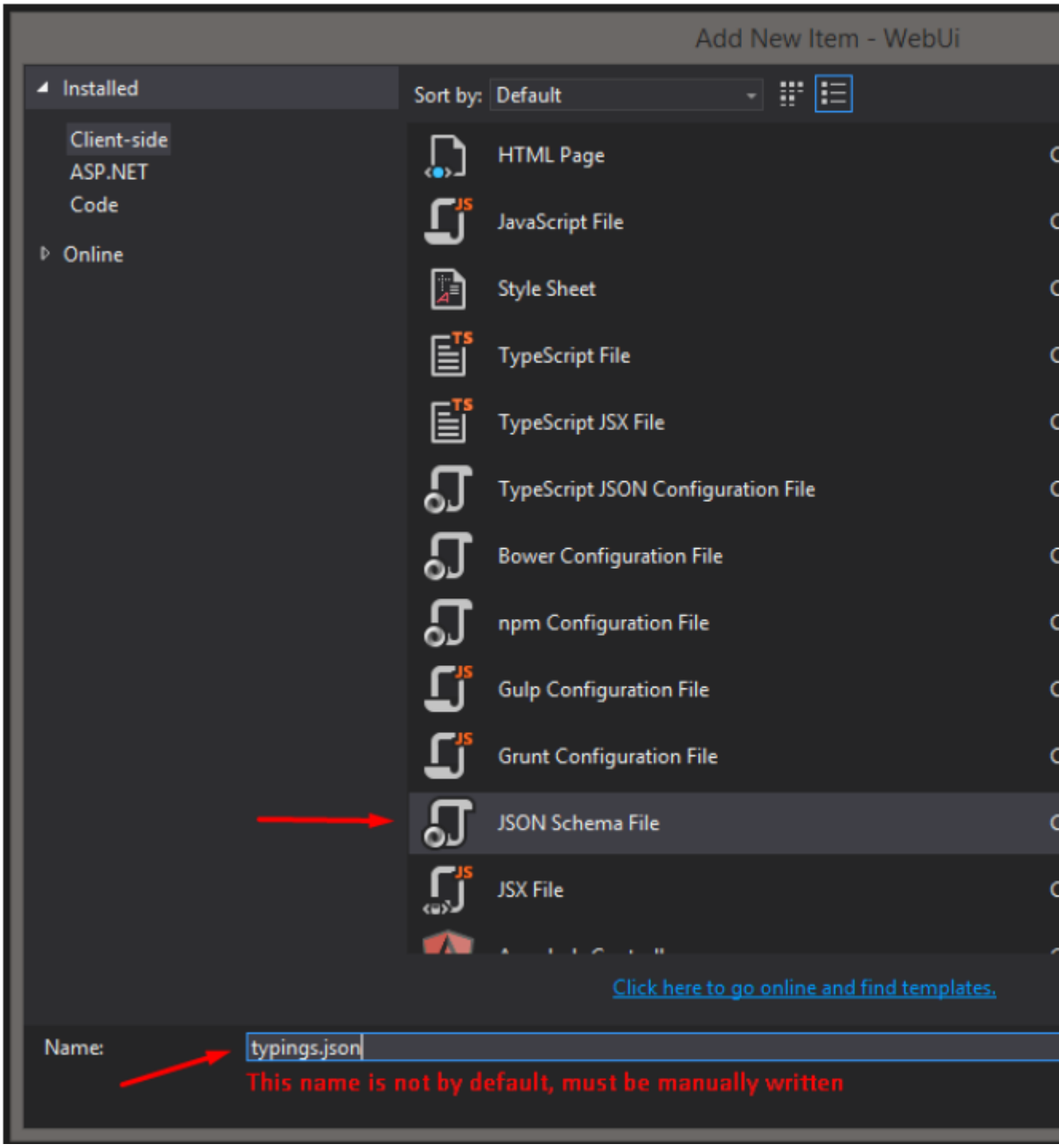


- All'interno di questa cartella aggiungere "TypeScript JSON Configuration File" (tsconfig.json):

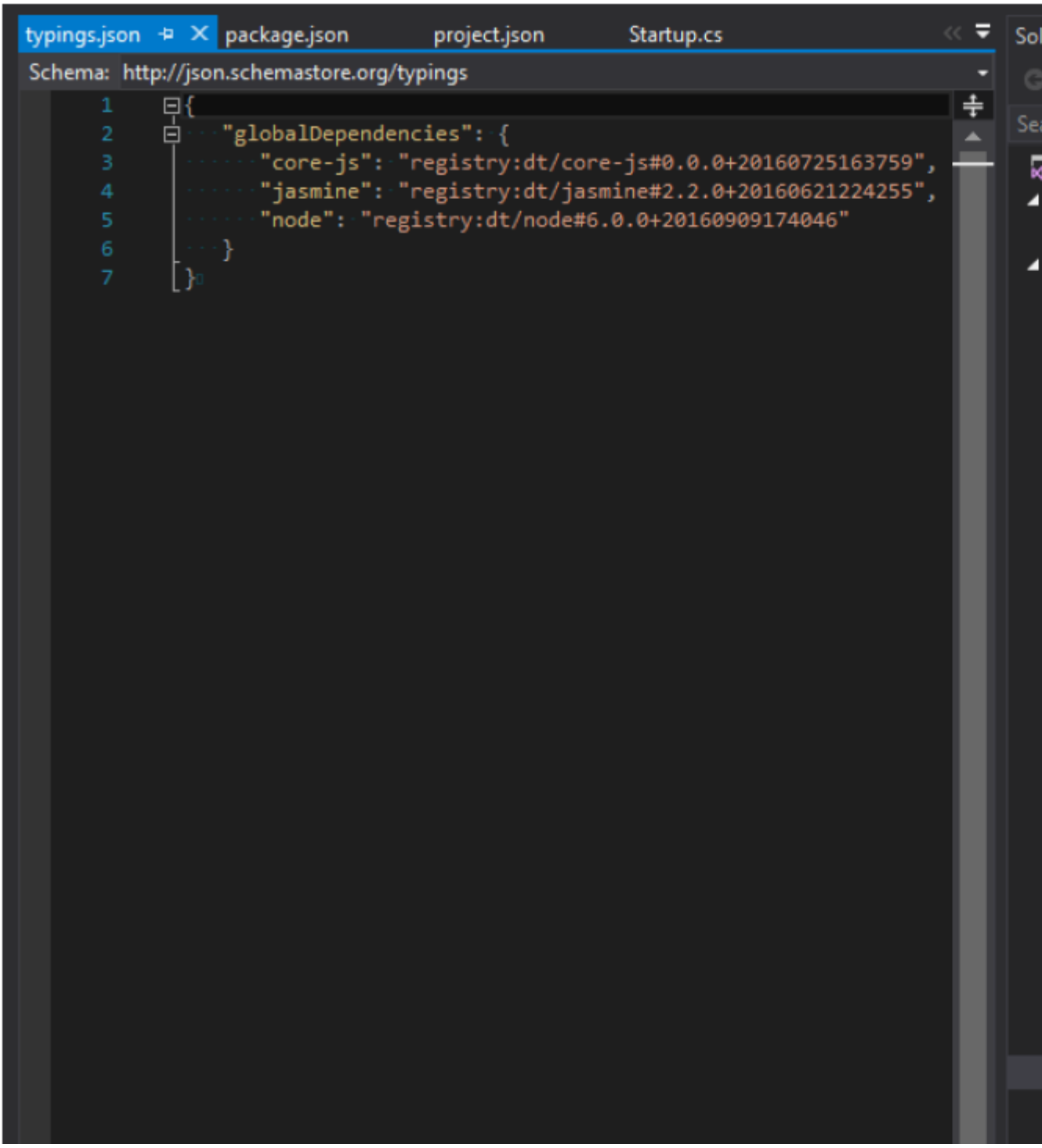


E aggiungi il prossimo codice:

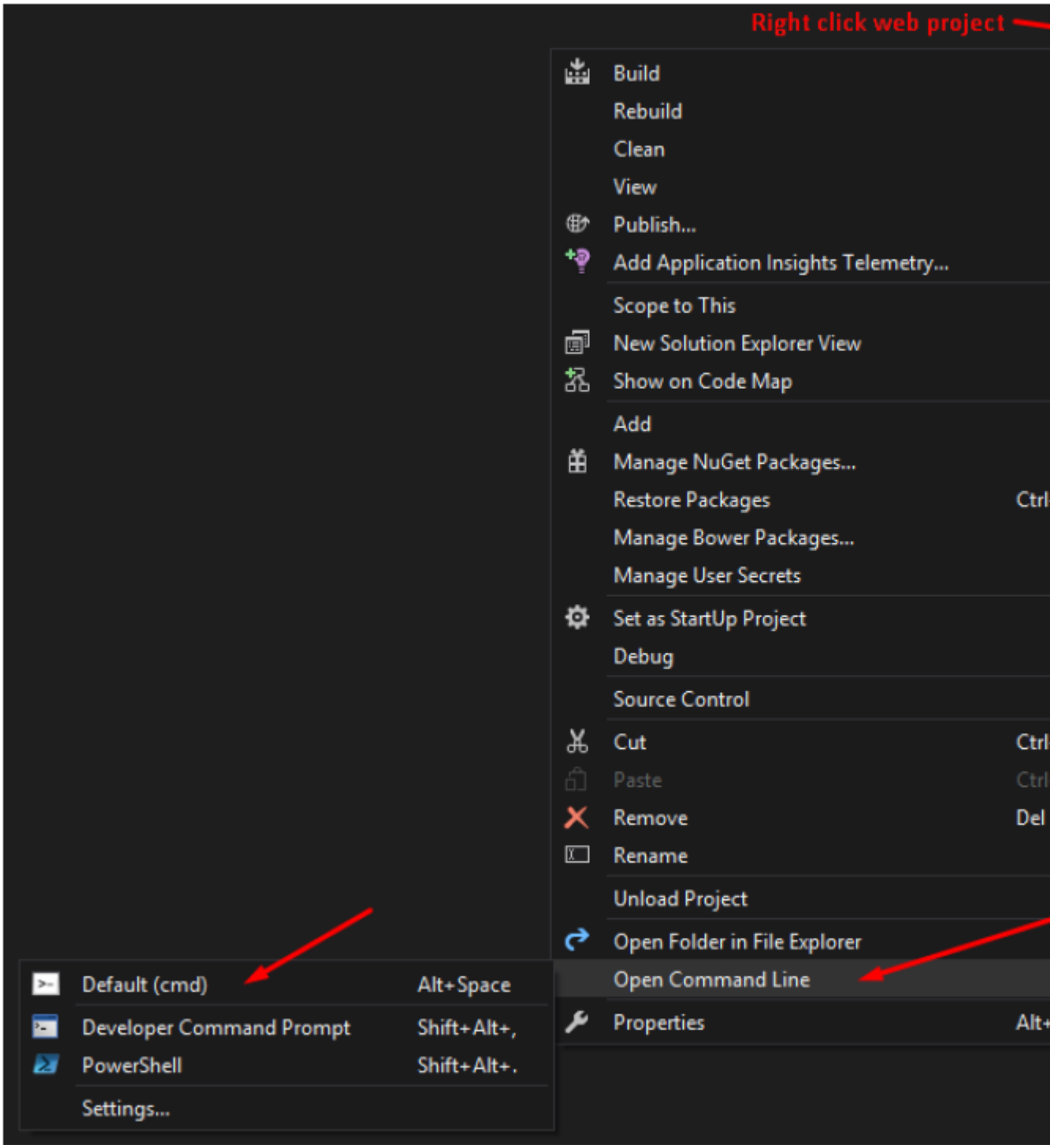
- Nella radice del progetto WebUi, aggiungi un nuovo file chiamato typings.json:

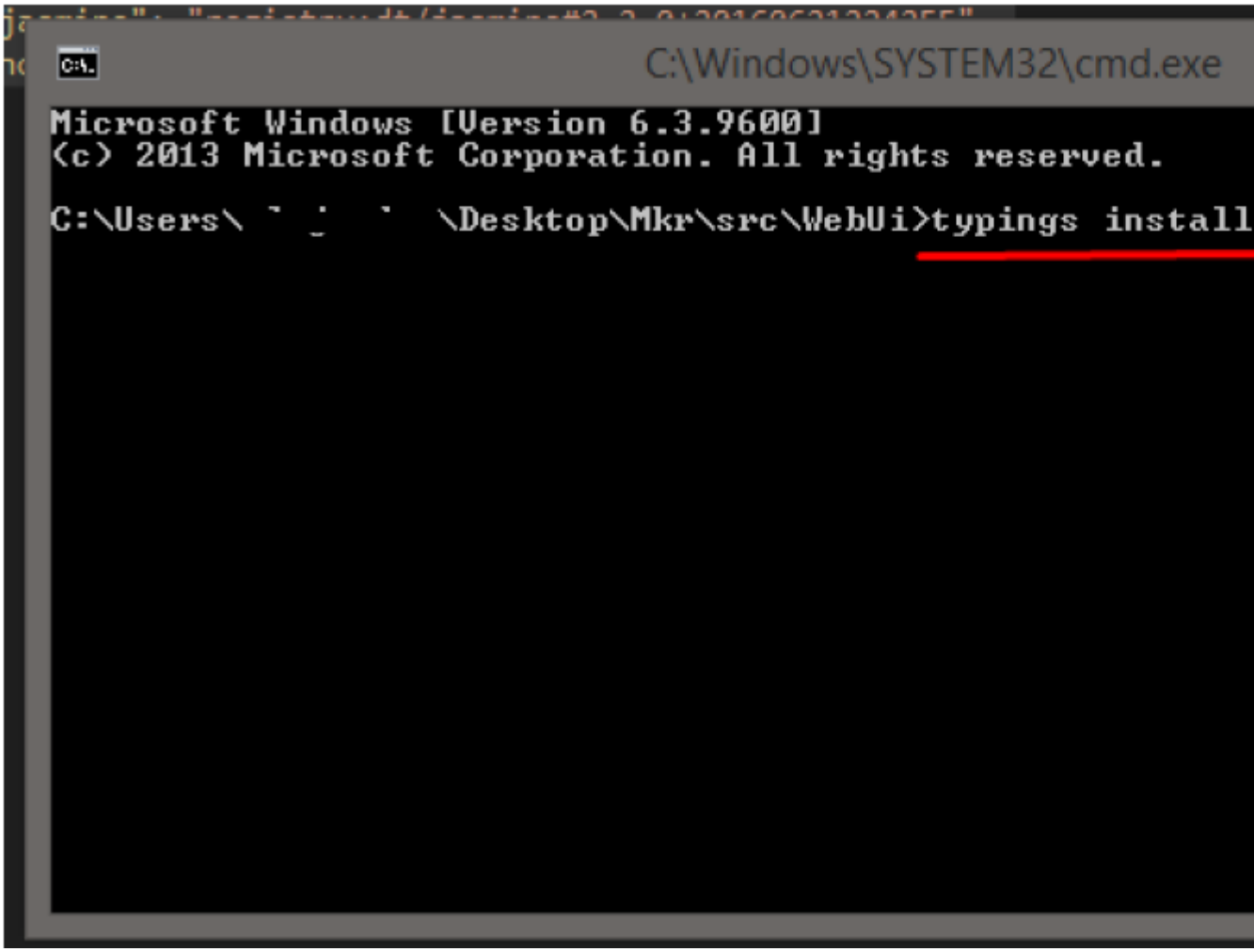


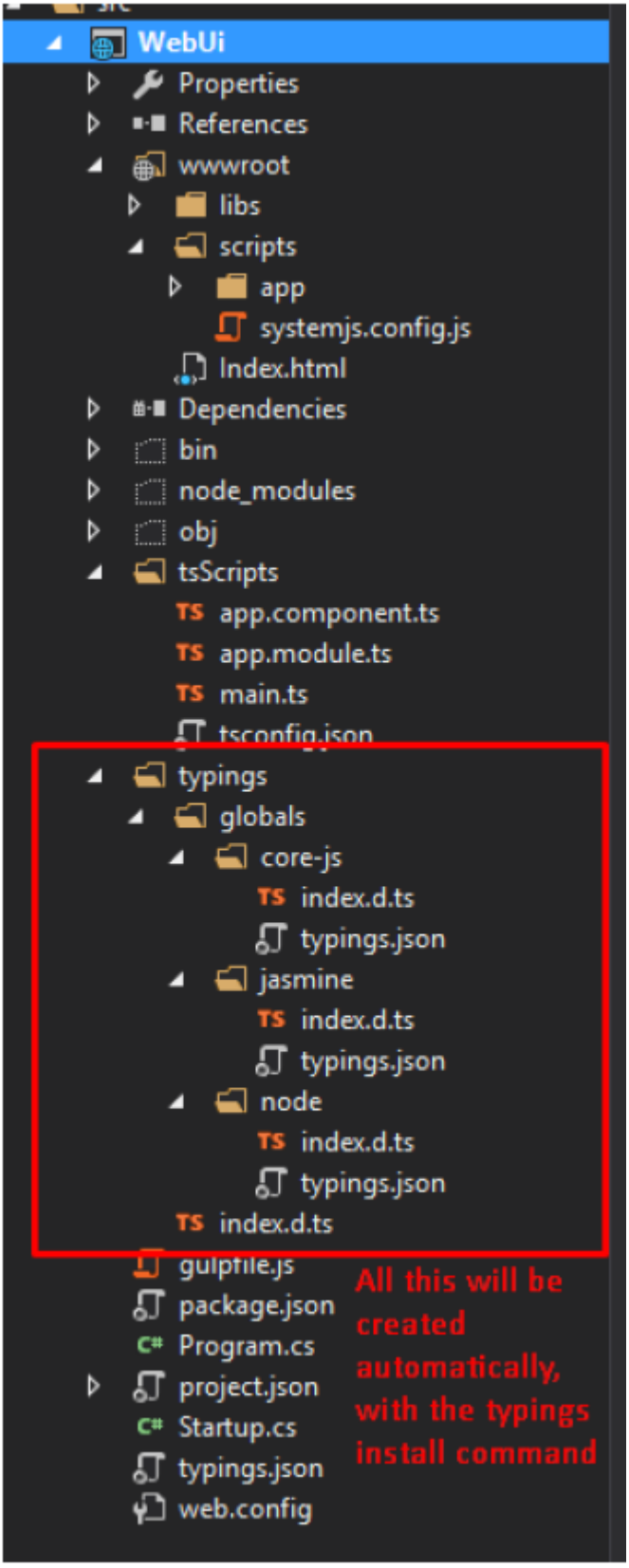
E aggiungi il prossimo codice:



- Nel progetto Web root apri una riga di comando ed esegui "typings install", questo creerà una cartella typings (Questo richiede "Open Command Line" spiegato come passaggio facoltativo nella nota all'interno del Passaggio 4, numero iii).

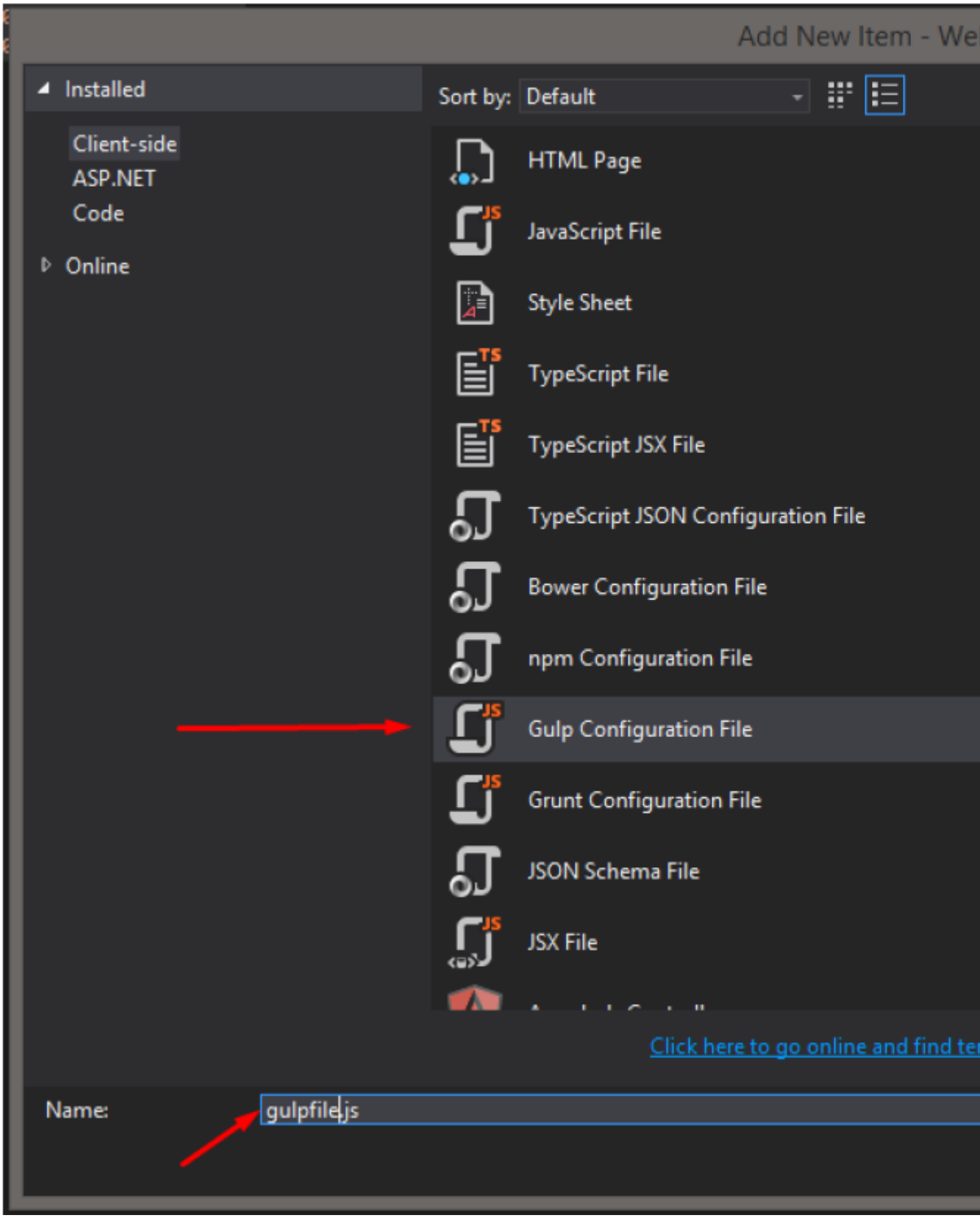

Right click web project
Build
Rebuild
Clean
View
Publish...
Add Application Insights Telemetry...
Scope to This
New Solution Explorer View
Show on Code Map
Add
Manage NuGet Packages...
Restore Packages
Manage Bower Packages...
Manage User Secrets
Set as StartUp Project
Debug
Source Control
Cut
Paste
Remove
Rename
Unload Project
Open Folder in File Explorer
Open Command Line
Properties
Default (cmd) Alt+Space
Developer Command Prompt Shift+Alt+,
PowerShell Shift+Alt+.
Settings...


C:\Windows\SYSTEM32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.
C:\Users\ ` _ ` \Desktop\Mkr\src\WebUi>typings install

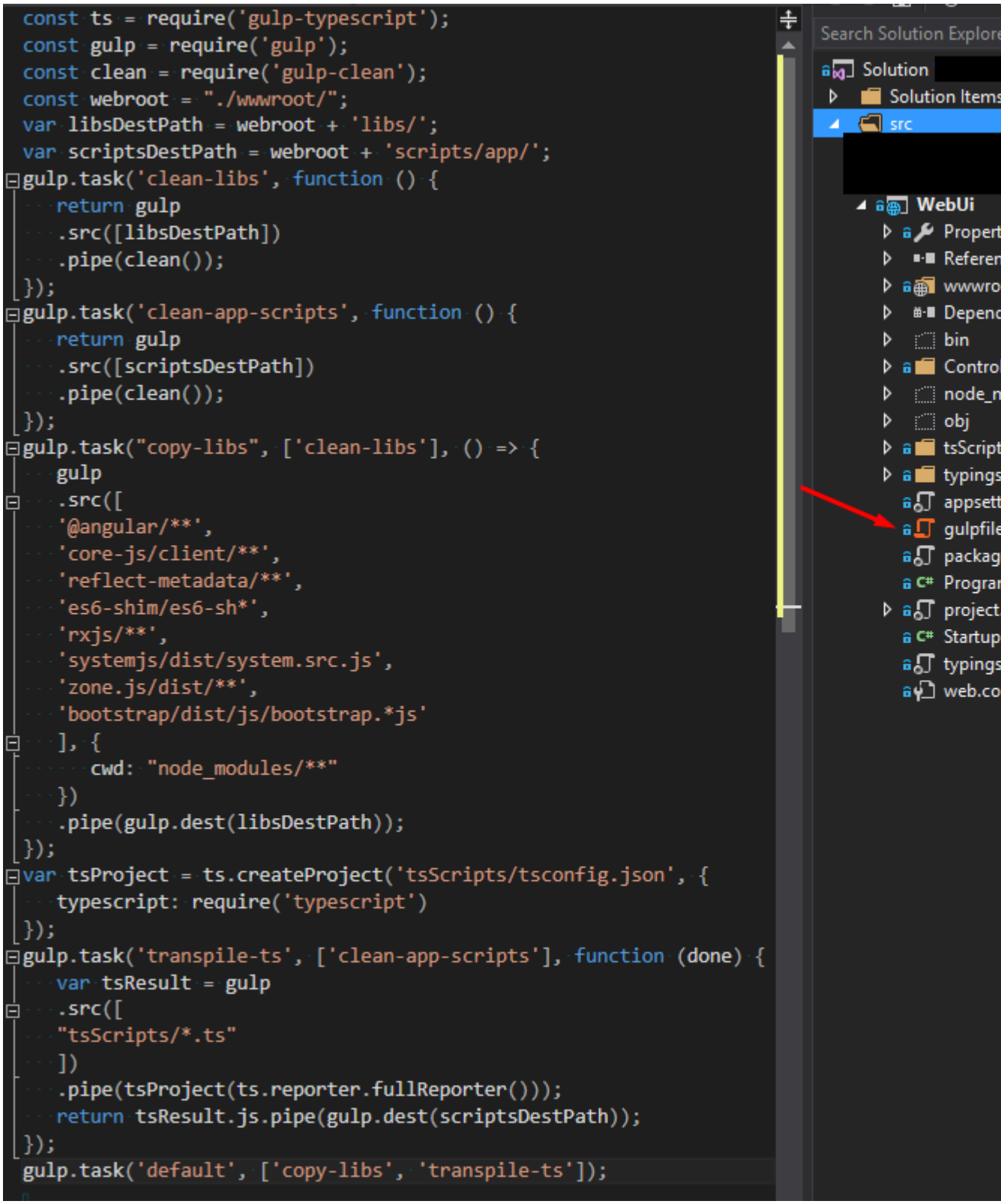


6. Aggiungi gulp per spostare i file:

    - Aggiungi "Gulp Configuration File" (gulpfile.js) nella radice del progetto web:

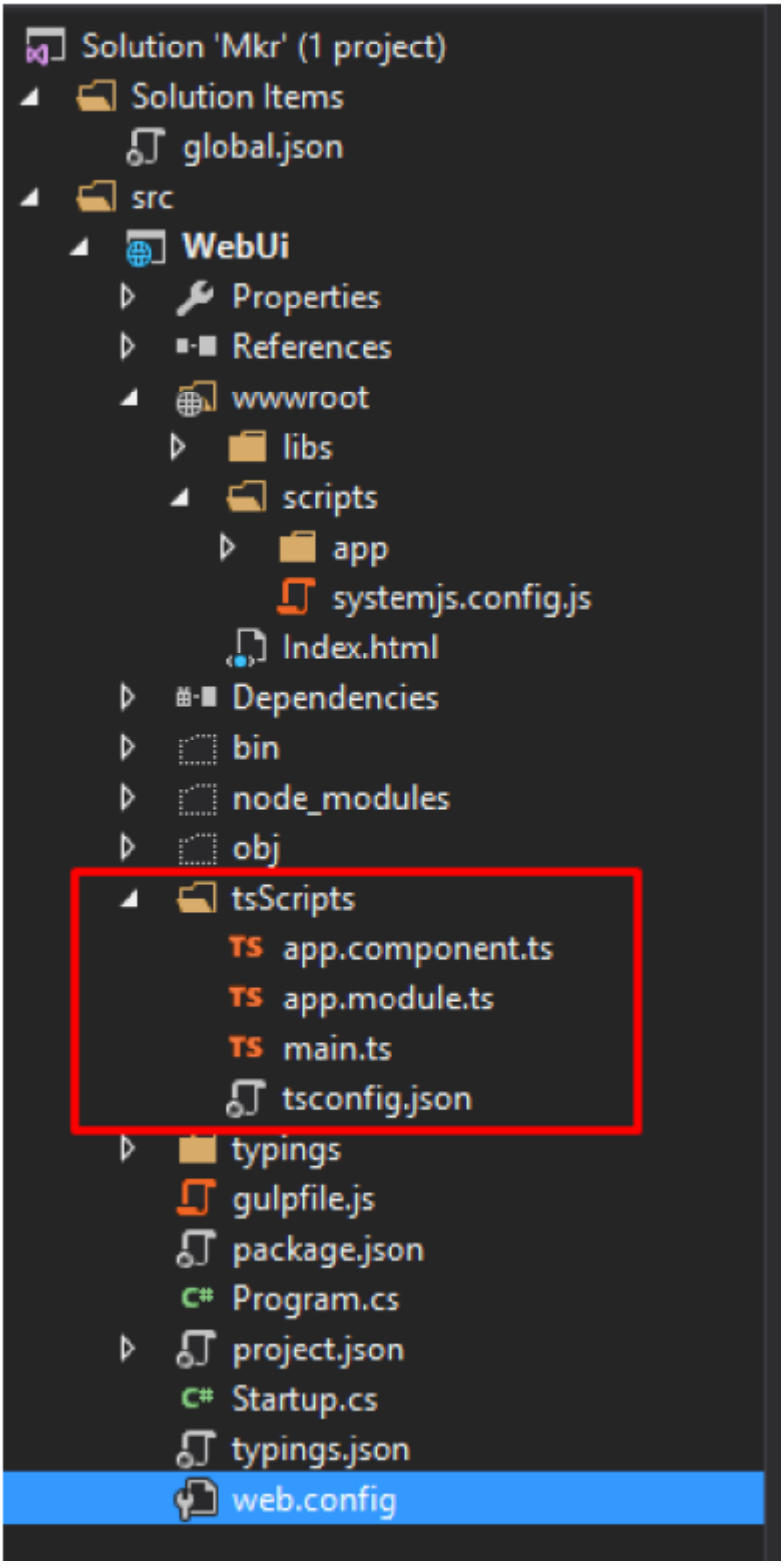


- Aggiungi codice:



```
const ts = require('gulp-typescript');
const gulp = require('gulp');
const clean = require('gulp-clean');
const webroot = "./wwwroot/";
var libsDestPath = webroot + 'libs/';
var scriptsDestPath = webroot + 'scripts/app/';
gulp.task('clean-libs', function () {
    return gulp
    .src([libsDestPath])
    .pipe(clean());
});
gulp.task('clean-app-scripts', function () {
    return gulp
    .src([scriptsDestPath])
    .pipe(clean());
});
gulp.task("copy-libs", ['clean-libs'], () => {
    gulp
    .src([
    '@angular/**',
    'core-js/client/**',
    'reflect-metadata/**',
    'es6-shim/es6-sh*',
    'rxjs/**',
    'systemjs/dist/system.src.js',
    'zone.js/dist/**',
    'bootstrap/dist/js/bootstrap.*js'
    ], {
        cwd: "node_modules/**"
    })
    .pipe(gulp.dest(libsDestPath));
});
var tsProject = ts.createProject('tsScripts/tsconfig.json', {
    typescript: require('typescript')
});
gulp.task('transpile-ts', ['clean-app-scripts'], function (done) {
    var tsResult = gulp
    .src([
    "tsScripts/*.ts"
    ])
    .pipe(tsProject(ts.reporter.fullReporter()));
    return tsResult.js.pipe(gulp.dest(scriptsDestPath));
});
gulp.task('default', ['copy-libs', 'transpile-ts']);
```


7. Aggiungi 2 file bootstrap angolari nella cartella "tsScripts":



app.component.ts

```
import { Component } from '@angular/core';

@Component({
    selector: 'my-app',
    template: `<h1>Hello {{name}}</h1>`
})
export class AppComponent { name = 'Angular'; }
```

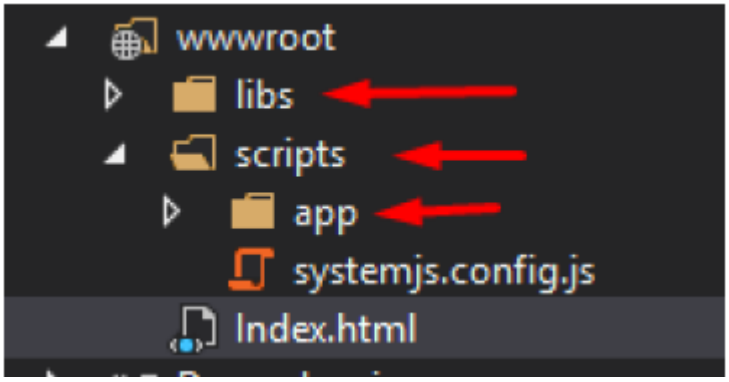
app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

@NgModule({
    imports: [BrowserModule],
    declarations: [AppComponent],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

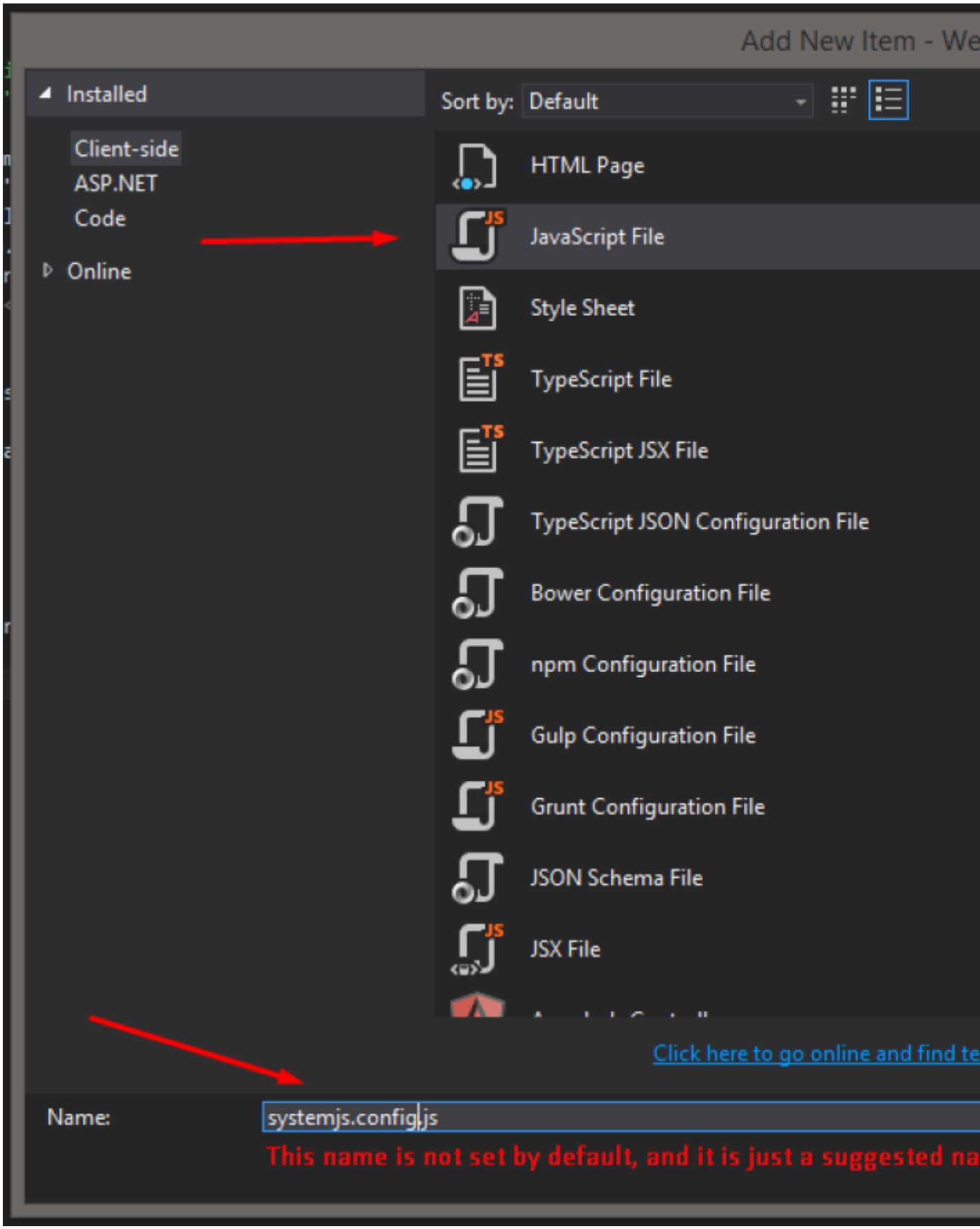
main.ts

```
import { platformBrowserDynamic } from '@angular/platform-br
import { AppModule }              from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

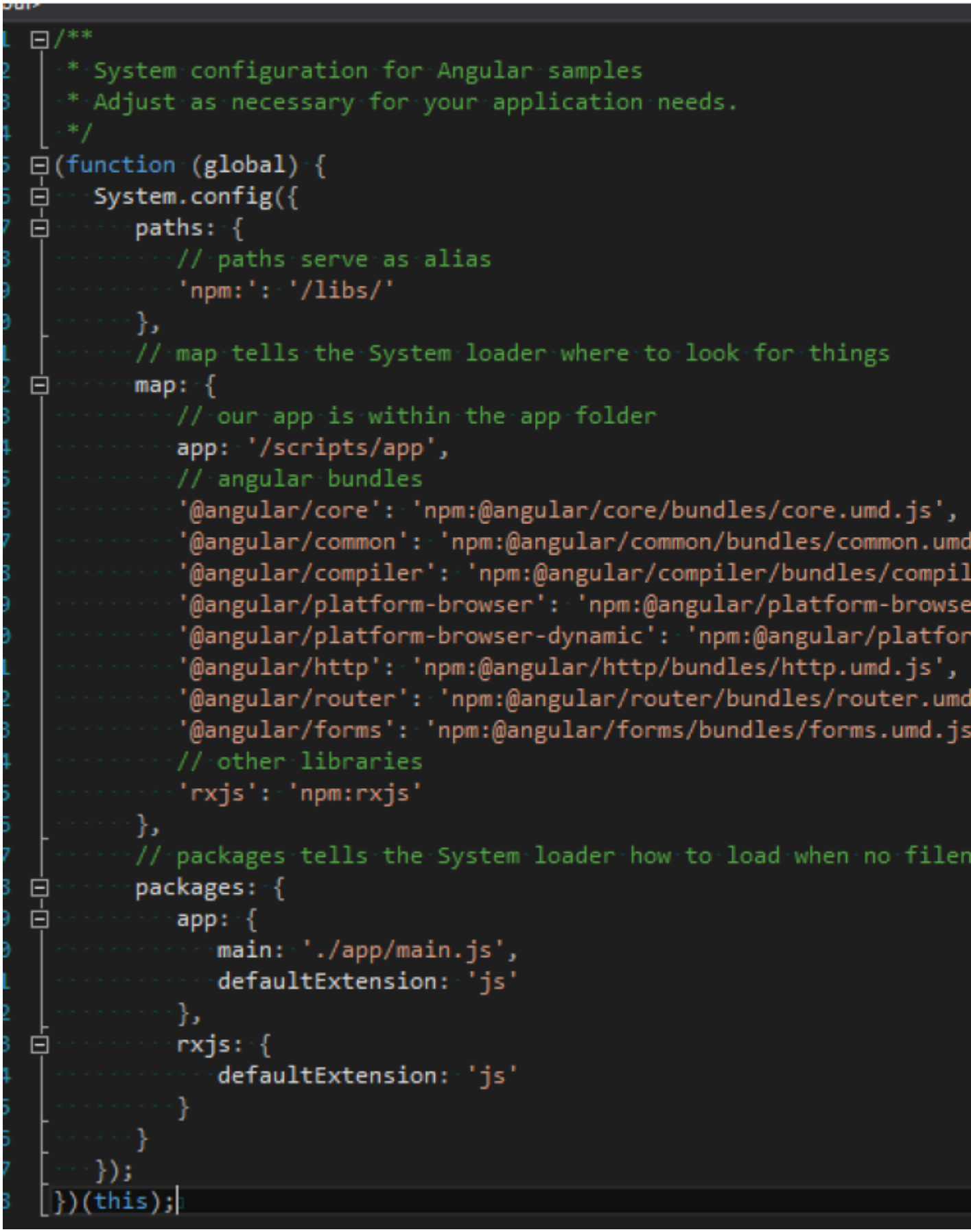
8. All'interno di wwwroot, crea la seguente struttura di file:



9. All'interno della cartella degli script (ma all'esterno dell'app), aggiungi systemjs.config.js:



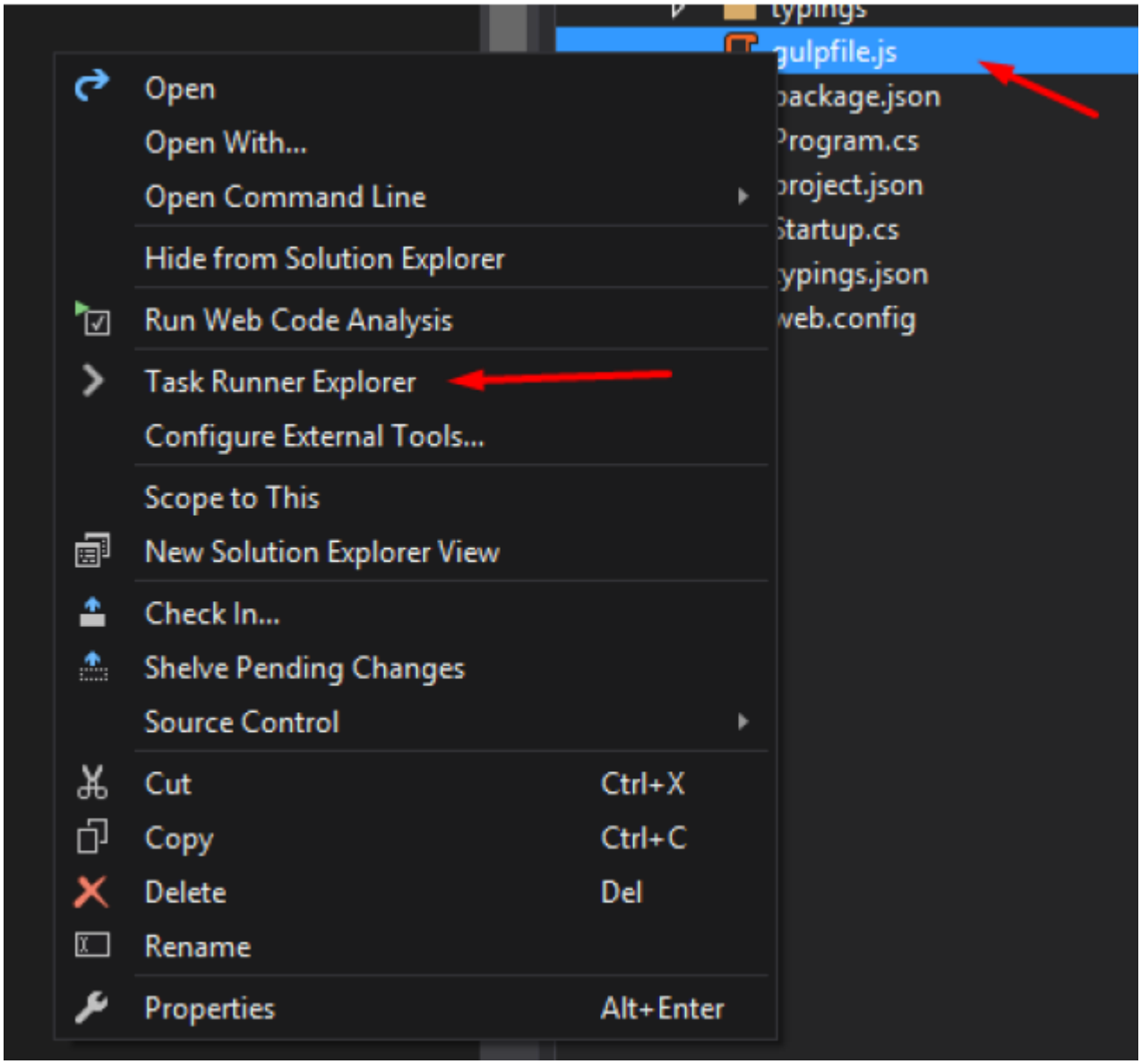
E aggiungi il prossimo codice:



```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
    System.config({
        paths: {
            // paths serve as alias
            'npm:': '/libs/'
        },
        // map tells the System loader where to look for things
        map: {
            // our app is within the app folder
            app: '/scripts/app',
            // angular bundles
            '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
            '@angular/common': 'npm:@angular/common/bundles/common.umd
            '@angular/compiler': 'npm:@angular/compiler/bundles/compil
            '@angular/platform-browser': 'npm:@angular/platform-browse
            '@angular/platform-browser-dynamic': 'npm:@angular/platfor
            '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
            '@angular/router': 'npm:@angular/router/bundles/router.umd
            '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js
            // other libraries
            'rxjs': 'npm:rxjs'
        },
        // packages tells the System loader how to load when no filen
        packages: {
            app: {
                main: './app/main.js',
                defaultExtension: 'js'
            },
            rxjs: {
                defaultExtension: 'js'
            }
        }
    });
})(this);
```

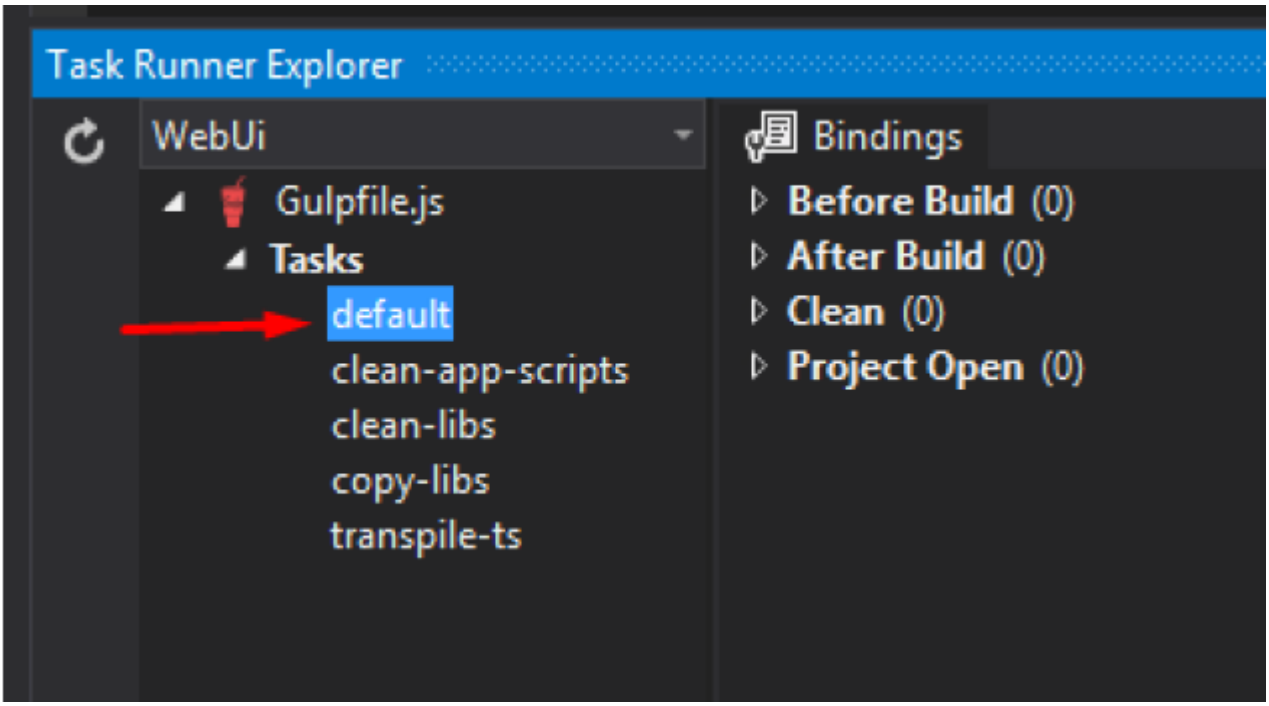
10. Esegui Gulp Task per generare gli script in wwwroot.

    - Fai clic con il tasto destro su gulpfile.js
    - Esegui Runner Explorer

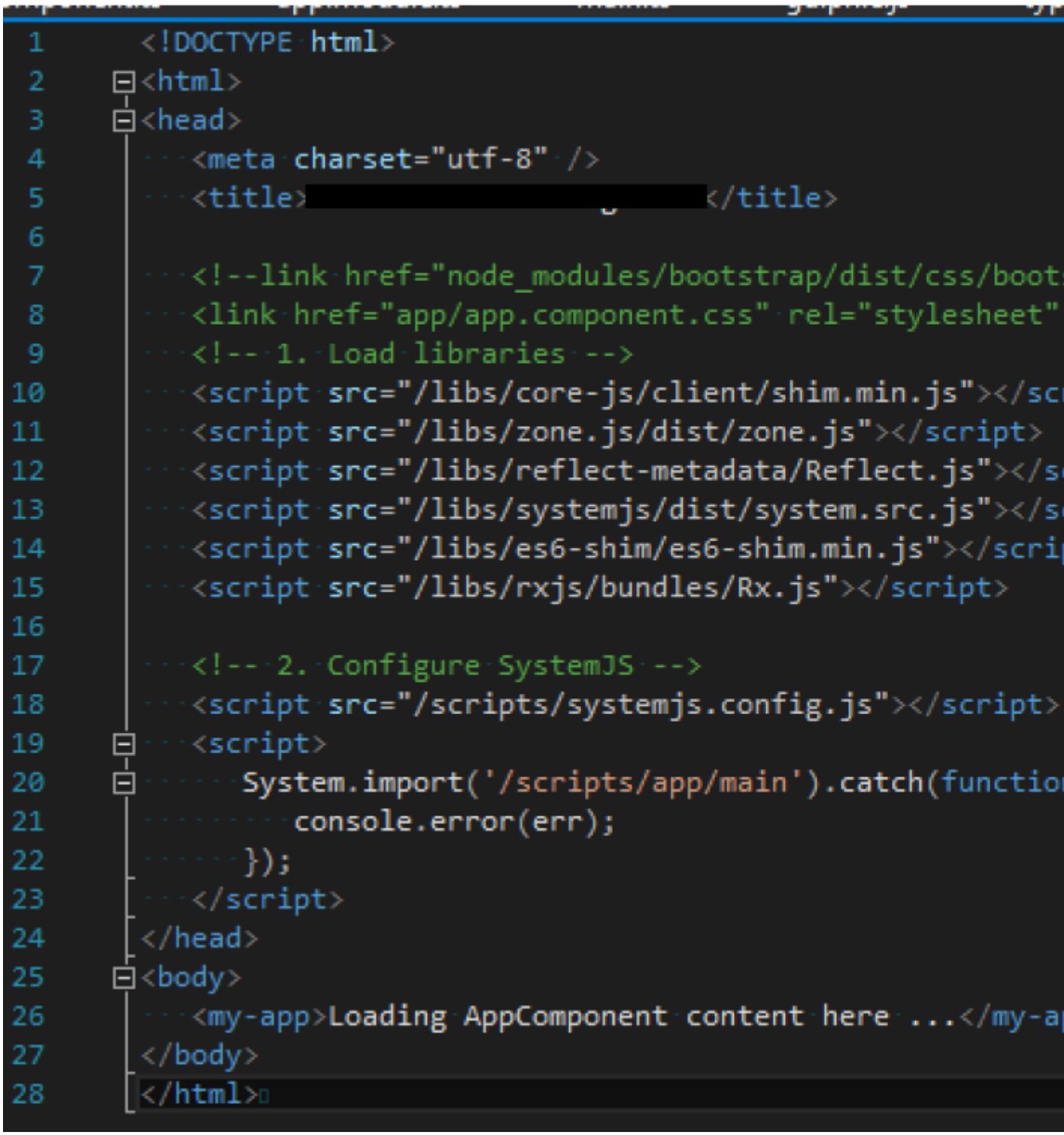


io. Se le attività non sono state caricate ("Impossibile caricare. Vedere la finestra Output") Vai alla finestra di output e dare un'occhiata agli errori, la maggior parte delle volte sono errori di sintassi nel file gulp.

- Fare clic con il tasto destro del mouse su "predefinito" e "Esegui" (ci vorrà un po ', e i messaggi di conferma non sono molto precisi, lo mostra finito ma il processo è ancora in esecuzione, tieni questo in mente):



11. Modifica Index.html come:



12. Adesso corri e divertiti.

    Gli appunti:

    - Nel caso in cui ci siano errori di compilazione con dattiloscritto, ad esempio "Progetto virtuale TypeScript", è un indicatore che la versione di TypeScript per Visual Studio non viene aggiornata in base alla versione selezionata in "package.json", se ciò accade installare : https://www.microsoft.com/en-us/download/details.aspx?id=48593

Riferimenti:

- Il corso "Angular 2: Getting Started" di Deborah Kurata in Pluralsight:

  https://www.pluralsight.com/courses/angular-2-getting-started-update

- Angular 2 Documentazione ufficiale:

  https://angular.io/

- Articoli di Mithun Pattankar:

  http://www.mithunvp.com/angular-2-in-asp-net-5-typescript-visual-studio-2015/

  http://www.mithunvp.com/using-angular-2-asp-net-mvc-5-visual-studio/

Errori previsti durante la generazione di componenti Angular 2 nel progetto .NET Core (versione 0.8.3)

Quando si generano nuovi componenti Angular 2 in un progetto .NET Core, è possibile che si verifichino i seguenti errori (dalla versione 0.8.3):

```
Error locating module for declaration
        SilentError: No module files found
```

O

```
No app module found. Please add your new Class to your component.
        Identical ClientApp/app/app.module.ts
```

[SOLUZIONE]

1. Rinominare app.module.client.ts in app.client.module.ts
2. Apri app.client.module.ts: aggiungi la dichiarazione con 3 punti "..." e avvolgi la dichiarazione tra parentesi.

   Ad esempio: `[...sharedConfig.declarations, <MyComponent>]`

3. Apri boot-client.ts: aggiorna l'importazione per utilizzare il nuovo riferimento a app.client.module.

   Ad esempio: `import { AppModule } from './app/app.client.module';`

4. Ora prova a generare il nuovo componente: `ng g component my-component`

[SPIEGAZIONE]

La CLI angolare cerca un file denominato app.module.ts nel progetto e tenta di trovare riferimenti per la proprietà dichiarazioni per importare il componente. Questo dovrebbe essere un array (come sharedConfig.declarations), ma le modifiche non vengono applicate

[Fonti]

- https://github.com/angular/angular-cli/issues/2962
- https://www.udemy.com/aspnet-core-angular/learn/v4/t/lecture/6848548 (sezione 3.33 collaboratore Bryan Garzon)

Leggi Angular2 e .Net Core online: https://riptutorial.com/it/asp-net-core/topic/9352/angular2-e--net-core

Capitolo 3: ASP.NET Core: registra sia la richiesta che la risposta utilizzando il middleware

introduzione

Da qualche tempo ho cercato il modo migliore per registrare richieste e risposte in un core ASP.Net. Stavo sviluppando servizi e uno dei requisiti era quello di registrare la richiesta con la sua risposta in un record del database. Tanti argomenti là fuori, ma nessuno ha funzionato per me. è solo per richiesta, solo risposta o semplicemente non ha funzionato. Quando finalmente sono riuscito a farlo, mi sono evoluto durante il mio progetto per migliorare la gestione degli errori e la registrazione delle eccezioni, quindi ho pensato di condividere.

Osservazioni

alcuni degli argomenti che mi sono stati utili:

- http://www.sulhome.com/blog/10/log-asp-net-core-request-and-response-using-middleware
- http://dotnetliberty.com/index.php/2016/01/07/logging-asp-net-5-requests-using-middleware/
- Come registrare il corpo di risposta HTTP in ASP.NET Core 1.0

Examples

Logger Middleware

```
using Microsoft.AspNetCore.Http;
using System;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNet.Http.Internal;
using Microsoft.AspNetCore.Http.Internal;

  public class LoggerMiddleware
    {
        private readonly RequestDelegate _next;

        public LoggerMiddleware(RequestDelegate next)
        {
            _next = next;
        }

        public async Task Invoke(HttpContext context)
        {
            using (MemoryStream requestBodyStream = new MemoryStream())
            {
```

```
                using (MemoryStream responseBodyStream = new MemoryStream())
                {
                    Stream originalRequestBody = context.Request.Body;
                    context.Request.EnableRewind();
                    Stream originalResponseBody = context.Response.Body;

                    try
                    {
                        await context.Request.Body.CopyToAsync(requestBodyStream);
                        requestBodyStream.Seek(0, SeekOrigin.Begin);

                        string requestBodyText = new
StreamReader(requestBodyStream).ReadToEnd();

                        requestBodyStream.Seek(0, SeekOrigin.Begin);
                        context.Request.Body = requestBodyStream;

                        string responseBody = "";


                        context.Response.Body = responseBodyStream;

                        Stopwatch watch = Stopwatch.StartNew();
                        await _next(context);
                        watch.Stop();

                        responseBodyStream.Seek(0, SeekOrigin.Begin);
                        responseBody = new StreamReader(responseBodyStream).ReadToEnd();
                        AuditLogger.LogToAudit(context.Request.Host.Host,
                            context.Request.Path, context.Request.QueryString.ToString(),
context.Connection.RemoteIpAddress.MapToIPv4().ToString(),
                            string.Join(",", context.Request.Headers.Select(he => he.Key +
":[" + he.Value + "]").ToList()),
                            requestBodyText, responseBody, DateTime.Now,
watch.ElapsedMilliseconds);

                        responseBodyStream.Seek(0, SeekOrigin.Begin);

                        await responseBodyStream.CopyToAsync(originalResponseBody);
                    }
                    catch (Exception ex)
                    {
                        ExceptionLogger.LogToDatabse(ex);
                        byte[] data = System.Text.Encoding.UTF8.GetBytes("Unhandled Error
occured, the error has been logged and the persons concerned are notified!! Please, try again
in a while.");
                        originalResponseBody.Write(data, 0, data.Length);
                    }
                    finally
                    {
                        context.Request.Body = originalRequestBody;
                        context.Response.Body = originalResponseBody;
                    }
                }
            }
        }
    }
```

Leggi ASP.NET Core: registra sia la richiesta che la risposta utilizzando il middleware online: https://riptutorial.com/it/asp-net-core/topic/9510/asp-net-core--registra-sia-la-richiesta-che-la-

risposta-utilizzando-il-middleware

Capitolo 4: Autorizzazione

Examples

Autorizzazione semplice

L'autorizzazione in asp.net core è semplicemente `AuthorizeAttribute`

```
[Authorize]
public class SomeController : Controller
{
    public IActionResult Get()
    {
    }

    public IActionResult Post()
    {
    }
}
```

Ciò consentirà solo a un utente connesso di accedere a queste azioni.

o utilizzare quanto segue per limitare solo una singola azione

```
public class SomeController : Controller
{
    public IActionResult Get()
    {
    }

    [Authorize]
    public IActionResult Post()
    {
    }
}
```

Se si desidera consentire a tutti gli utenti di accedere a una delle azioni, è possibile utilizzare `AllowAnonymousAttribute`

```
[Authorize]
public class SomeController: Controller
{
    public IActionResult Get()
    {
    }

    [AllowAnonymous]
    public IActionResult Post()
    {
    }
}
```

Ora è possibile accedere a `Post` con qualsiasi utente. `AllowAnonymous` sempre la priorità di

autorizzare, quindi se un controller è impostato su `AllowAnonymous` tutte le sue azioni sono pubbliche, indipendentemente dal fatto che abbiano un `AuthorizeAttribute` o meno.

C'è un'opzione per impostare tutti i controller per richiedere richieste autorizzate -

```
services.AddMvc(config =>
{
    var policy = new AuthorizationPolicyBuilder()
        .RequireAuthenticatedUser()
        .Build();
    config.Filters.Add(new AuthorizeFilter(policy));
})
```

Questo viene fatto aggiungendo un criterio di autorizzazione predefinito a ciascun controller: qualsiasi Attributo `Authorize` / `AllowAnonymous` su un controller / azione sovrascriverà queste impostazioni.

Leggi Autorizzazione online: https://riptutorial.com/it/asp-net-core/topic/6914/autorizzazione

Capitolo 5: Bundling e Minification

Examples

Grunt e Gulp

Nelle app ASP.NET Core, si raggruppano e si minimizzano le risorse lato client durante la fase di progettazione utilizzando strumenti di terze parti, come Gulp e Grunt . Utilizzando il raggruppamento e la minimizzazione in fase di progettazione, i file minificati vengono creati prima della distribuzione dell'applicazione. Raggruppare e minimizzare prima della distribuzione offre il vantaggio di un ridotto carico del server. Tuttavia, è importante riconoscere che il raggruppamento e la minificazione in fase di progettazione aumentano la complessità di creazione e funzionano solo con file statici.

Questo viene fatto in ASP.NET Core configurando Gulp tramite un file `gulpfile.js` all'interno del tuo progetto:

```
// Defining dependencies
var gulp = require("gulp"),
    rimraf = require("rimraf"),
    concat = require("gulp-concat"),
    cssmin = require("gulp-cssmin"),
    uglify = require("gulp-uglify");

// Define web root
var webroot = "./wwwroot/"

// Defining paths
var paths = {
    js: webroot + "js/**/*.js",
    minJs: webroot + "js/**/*.min.js",
    css: webroot + "css/**/*.css",
    minCss: webroot + "css/**/*.min.css",
    concatJsDest: webroot + "js/site.min.js",
    concatCssDest: webroot + "css/site.min.css"
};

// Bundling (via concat()) and minifying (via uglify()) Javascript
gulp.task("min:js", function () {
    return gulp.src([paths.js, "!" + paths.minJs], { base: "." })
        .pipe(concat(paths.concatJsDest))
        .pipe(uglify())
        .pipe(gulp.dest("."));
});

// Bundling (via concat()) and minifying (via cssmin()) Javascript
gulp.task("min:css", function () {
    return gulp.src([paths.css, "!" + paths.minCss])
        .pipe(concat(paths.concatCssDest))
        .pipe(cssmin())
        .pipe(gulp.dest("."));
});
```

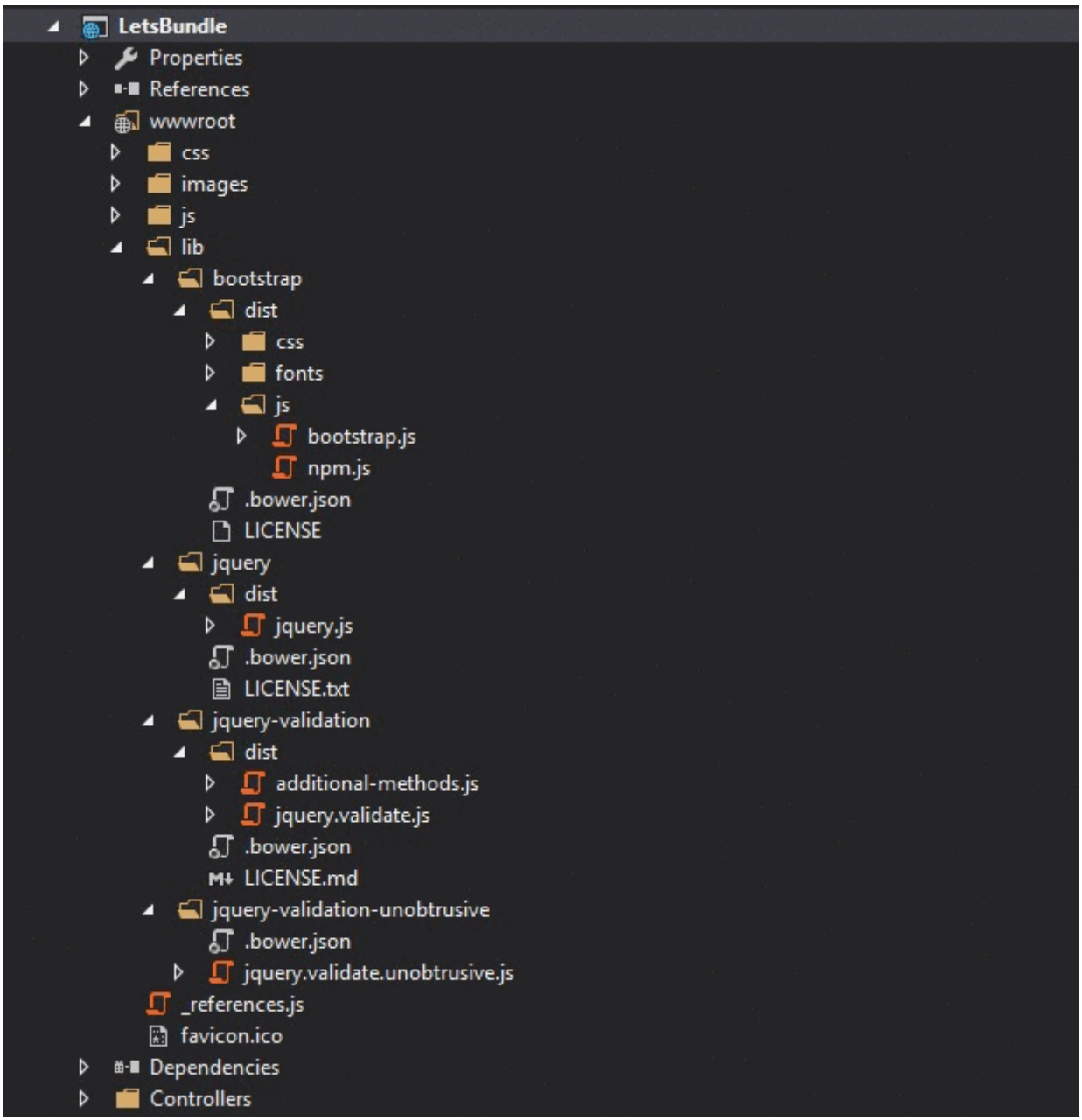
Questo approccio raggruppa e ridimensiona correttamente i file Javascript e CSS esistenti rispettivamente in base alle directory e agli schemi di globbing utilizzati.

Bundler e Minifier Extension

Visual Studio include anche un'estensione Bundler e Minifier disponibile che è in grado di gestire questo processo per te. L'estensione consente di selezionare e raggruppare facilmente i file necessari senza scrivere una riga di codice.

Costruire i tuoi pacchi

Dopo aver installato l'estensione, si **selezionano tutti i file specifici che si desidera includere in un pacchetto e si utilizza l'opzione Bundle and Minify Files dall'estensione:**



Questo ti richiederà di nominare il tuo gruppo e scegliere una posizione in cui salvarlo. Noterai quindi un nuovo file all'interno del tuo progetto chiamato `bundleconfig.json` che assomiglia al

seguente:

```
[
  {
    "outputFileName": "wwwroot/app/bundle.js",
    "inputFiles": [
      "wwwroot/lib/jquery/dist/jquery.js",
      "wwwroot/lib/bootstrap/dist/js/bootstrap.js",
      "wwwroot/lib/jquery-validation/dist/jquery.validate.js",
      "wwwroot/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"
    ]
  }
]
```

> **NOTA:** l'ordine in cui i file sono selezionati determinerà l'ordine in cui appaiono all'interno del gruppo, quindi se si hanno delle dipendenze, assicurarsi di tenerne conto.

Minifying Your Bundles

Ora il passaggio precedente sarà semplicemente raggruppare i file, se si desidera ridurre il pacchetto, è necessario indicarlo all'interno del file bundleconfig.json. **Semplicemente aggiungi un blocco `minify` come il seguente al tuo pacchetto esistente per indicare che lo vuoi minificato:**

```
[
  {
    "outputFileName": "wwwroot/app/bundle.js",
    "inputFiles": [
      "wwwroot/lib/jquery/dist/jquery.js",
      "wwwroot/lib/bootstrap/dist/js/bootstrap.js",
      "wwwroot/lib/jquery-validation/dist/jquery.validate.js",
      "wwwroot/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"
    ],
    "minify": {
      "enabled": true
    }
  }
]
```

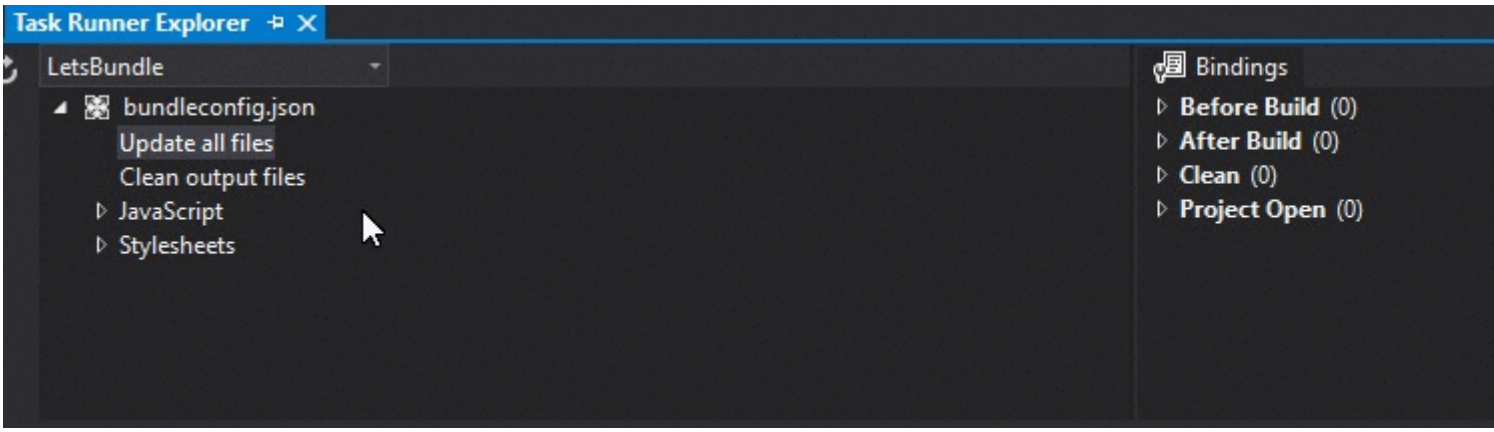
Automatizza i tuoi pacchetti

Infine, se si desidera automatizzare questo processo, è possibile pianificare un'attività da eseguire ogni volta che l'applicazione viene creata per garantire che i propri pacchetti riflettano eventuali modifiche all'interno dell'applicazione.

Per fare questo, dovrai fare quanto segue:

- **Aprire il Task Runner Explorer** (tramite Strumenti> Task Runner Explorer).
- **Fare clic con il tasto destro del mouse sull'opzione Aggiorna tutti i file** sotto `bundleconfig.json` .

- **Seleziona l'associazione preferita** dal menu di scelta rapida Bindings.



Dopo averlo fatto, i tuoi bundle dovrebbero essere aggiornati automaticamente nel passaggio preferito selezionato.

Il comando dotnet bundle

La versione di ASP.NET Core RTM ha introdotto `BundlerMinifier.Core` , un nuovo strumento di raggruppamento e minificazione che può essere facilmente integrato nelle applicazioni esistenti di ASP.NET Core e non richiede estensioni esterne o file di script.

Utilizzando BundlerMinifier.Core

Per utilizzare questo strumento, è **sufficiente aggiungere un riferimento a `BundlerMinifier.Core` nella sezione `tools` del file `project.json` esistente** come mostrato di seguito:

```
"tools": {
  "BundlerMinifier.Core": "2.0.238",
  "Microsoft.AspNetCore.Razor.Tools": "1.0.0-preview2-final",
  "Microsoft.AspNetCore.Server.IISIntegration.Tools": "1.0.0-preview2-final"
}
```

Configurazione dei pacchetti

Dopo aver aggiunto lo strumento, dovrai **aggiungere un file `bundleconfig.json` nel progetto** che verrà utilizzato per configurare i file che desideri includere nei tuoi bundle. Di seguito è riportata una configurazione minima:

```
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  },
  {
    "outputFileName": "wwwroot/js/site.min.js",
    "inputFiles": [
```

```
      "wwwroot/js/site.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    },
    "sourceMap": false
  },
  {
    "outputFileName": "wwwroot/js/semantic.validation.min.js",
    "inputFiles": [
      "wwwroot/js/semantic.validation.js"
    ],
    "minify": {
      "enabled": true,
      "renameLocals": true
    }
  }
]
```

Creazione / Aggiornamento di pacchetti

Dopo aver configurato i bundle, puoi raggruppare e ridimensionare i file esistenti tramite il seguente comando:

```
dotnet bundle
```

Bundling automatico

Il processo di raggruppamento e minificazione può essere automatizzato come parte del processo di compilazione aggiungendo il comando `dotnet bundle` nella sezione di precompilazione del file `project.json` esistente:

```
"scripts": {
  "precompile": [
    "dotnet bundle"
  ]
}
```

Comandi disponibili

Puoi vedere una lista di tutti i comandi disponibili e le loro descrizioni qui sotto:

- **dotnet bundle** - Esegue il comando bundle utilizzando il file `bundleconfig.json` per raggruppare e `bundleconfig.json` file specificati.
- **dotnet bundle clean** - Cancella tutti i file di output esistenti dal disco.
- **dotnet bundle watch** : crea osservatori che eseguiranno automaticamente il `dotnet bundle` ogni volta che un file di input esistente dalla configurazione `bundleconfig.json` raggruppa i file.
- **help bundle dotnet** : visualizza tutte le opzioni di guida disponibili e le istruzioni per l'utilizzo

dell'interfaccia della riga di comando.

Leggi Bundling e Minification online: https://riptutorial.com/it/asp-net-core/topic/4051/bundling-e-minification

Capitolo 6: caching

introduzione

Il caching aiuta a migliorare le prestazioni di un'applicazione mantenendo una copia dei dati facilmente accessibile. *Aspnet Core è* dotato di due astrazioni di caching facili da usare e da testare.

Memory Cache memorizzerà i dati nella memoria cache del server locale.

Distributed Cache conserverà la cache dei dati in una posizione centralizzata accessibile dai server nel cluster. Viene fornito con tre implementazioni fuori dalla scatola: In memoria (per unità di test e dev locale), Redis e Sql Server.

Examples

Utilizzo della cache InMemory nell'applicazione ASP.NET Core

Per utilizzare una cache di memoria nell'applicazione ASP.NET, aggiungere le seguenti dipendenze al file `project.json` :

```
  "Microsoft.Extensions.Caching.Memory": "1.0.0-rc2-final",
```

aggiungere il servizio cache (da Microsoft.Extensions.Caching.Memory) al metodo ConfigureServices nella classe Startup

```
services.AddMemoryCache();
```

Per aggiungere elementi alla cache nella nostra applicazione, utilizzeremo `IMemoryCache` che può essere iniettato in qualsiasi classe (ad esempio Controller) come mostrato di seguito.

```
private IMemoryCache _memoryCache;
public HomeController(IMemoryCache memoryCache)
{
    _memoryCache = memoryCache;
}
```

Get restituirà il valore se esiste, ma restituisce altrimenti `null` .

```
    // try to get the cached item; null if not found
    // greeting = _memoryCache.Get(cacheKey) as string;

    // alternately, TryGet returns true if the cache entry was found
    if(!_memoryCache.TryGetValue(cacheKey, out greeting))
```

Usa il metodo **Set** per scrivere nella cache. Set accetta la chiave da utilizzare per cercare il valore, il valore da memorizzare nella cache e un set di `MemoryCacheEntryOptions` . `MemoryCacheEntryOptions`

consente di specificare la scadenza della cache assoluta o scorrevole basata sul tempo, la priorità di memorizzazione nella cache, i callback e le dipendenze. Uno dei seguenti esempi

```
_memoryCache.Set(cacheKey, greeting,
                new MemoryCacheEntryOptions()
                .SetAbsoluteExpiration(TimeSpan.FromMinutes(1)));
```

Caching distribuito

Per sfruttare la cache distribuita, è necessario fare riferimento a una delle implementazioni disponibili:

- Redis
- Server SQL

Ad esempio, registrerai l'implementazione di Redis come segue:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDistributedRedisCache(options =>
    {
        options.Configuration = "ServerAdress";
        options.InstanceName = "InstanceName";
    });
}
```

Richiedere la dipendenza `IDistributedCache` dove ne hai bisogno:

```
public class BooksController {
    private IDistributedCache distributedCache;

    public BooksController(IDistributedCache distributedCache) {
        this.distributedCache = distributedCache;
    }

    [HttpGet]
    public async Task<Books[]> GetAllBooks() {
        var serialized = this.distributedCache.GetStringAsync($"allbooks");
        Books[] books = null;
        if (string.IsNullOrEmpty(serialized)) {
            books = await Books.FetchAllAsync();
            this.distributedCache.SetStringAsync($"allbooks",
JsonConvert.SerializeObject(books));
        } else {
            books = JsonConvert.DeserializeObject<Books[]>(serialized);
        }
        return books;
    }
}
```

Leggi caching online: https://riptutorial.com/it/asp-net-core/topic/8090/caching

Capitolo 7: Configurazione

introduzione

Asp.net core fornisce astrazioni di configurazione. Permettono di caricare le impostazioni di configurazione da varie fonti e creare un modello di configurazione finale che può essere quindi utilizzato dall'applicazione.

Sintassi

- `IConfiguration`
- `string this[string key] { get; set; }`
- `IEnumerable<IConfigurationSection> GetChildren();`
- `IConfigurationSection GetSection(string key);`

Examples

Accesso alla configurazione mediante Iniezione delle dipendenze

L'approccio consigliato sarebbe evitare di farlo e piuttosto utilizzare `IOptions<TOptions>` e `IServiceCollection.Configure<TOptions>` .

Detto questo, è ancora abbastanza semplice rendere `IConfigurationRoot` disponibile per `IConfigurationRoot` le applicazioni.

Nel costruttore Startup.cs dovresti avere il seguente codice per costruire la configurazione,

```
Configuration = builder.Build();
```

Qui `Configuration` è un'istanza di `IConfigurationRoot` , e aggiungi questa istanza come Singleton alla raccolta di servizi nel metodo ConfigureServices, Startup.cs,

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddSingleton<IConfigurationRoot>(provider => Configuration);
```

Ad esempio, ora è possibile accedere alla configurazione in un controller / servizio

```
 public MyController(IConfigurationRoot config){
     var setting1= config.GetValue<string>("Setting1")
 }
```

Iniziare

In questo esempio descriveremo cosa succede quando impalcature un nuovo progetto.

Per prima cosa, verranno aggiunte le seguenti dipendenze al progetto (attualmente `project.json` file):

```
"Microsoft.Extensions.Configuration.EnvironmentVariables": "1.0.0",
"Microsoft.Extensions.Configuration.Json": "1.0.0",
```

Creerà anche un costruttore nel file `Startup.cs` che sarà responsabile della creazione della configurazione utilizzando l'API fluente di `ConfigurationBuilder` :

1. Prima crea un nuovo `ConfigurationBuilder` .
2. Quindi imposta un percorso di base che verrà utilizzato per calcolare il percorso assoluto di ulteriori file
3. Aggiunge un `appsettings.json` opzionale.json al builder di configurazione e monitora le sue modifiche
4. Aggiunge un file di configurazione `appsettings.environementName.json` relativo all'ambiente opzionale
5. Quindi aggiunge le variabili di ambiente.

```
public Startup(IHostingEnvironment env)
{
    var builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();

    Configuration = builder.Build();
}
```

Se una stessa impostazione è impostata su più fonti, l'ultima sorgente aggiunta vincerà e il suo valore sarà selezionato.

La configurazione può quindi essere utilizzata utilizzando la proprietà dell'indicizzatore. I due punti `:` carattere serve un delimitatore di percorso.

```
Configuration["AzureLogger:ConnectionString"]
```

Questo cercherà un valore di configurazione `ConnectionString` in una sezione di `AzureLogger` .

Lavora con le variabili d'ambiente

È possibile creare la configurazione di origine dalle variabili di ambiente chiamando `.AddEnvironmentVariables()` su `ConfigurationBuilder` .

Caricherà le variabili di ambiente prefissate con `APPSETTING_` Quindi userà due punti `:` come separatore del percorso chiave.

Ciò significa che: seguendo le impostazioni dell'ambiente:

```
APPSETTING_Security:Authentication:UserName = a_user_name
```

```
APPSETTING_Security:Authentication:Password = a_user_password
```

Sarà l'equivalente di questo json:

```
{
    "Security" : {
       "Authentication" : {
           "UserName" : "a_user_name",
           "Password" : "a_user_password"
        }
    }
}
```

** Si noti che il servizio di Azure trasmetterà le impostazioni come variabili di ambiente. Il prefisso verrà impostato per te in modo trasparente. Quindi per fare lo stesso in Azure è sufficiente impostare due Impostazioni applicazione nel pannello AppSettings:

```
Security:Authentication:UserName         a_user_name
Security:Authentication:Password         a_user_password
```

Modello di opzione e configurazione

Quando si ha a che fare con grandi insiemi di valori di configurazione, potrebbe essere piuttosto difficile caricarli uno per uno.

Opzione modello che viene fornito con asp.net offre un modo conveniente per mappare una `section` ad un dotnet `poco` : per esempio, si potrebbe idratare `StorageOptions` direttamente da una sezione di configurazione b aggiungendo `Microsoft.Extensions.Options.ConfigurationExtensions` pacchetto e chiamando il `Configure<TOptions>(IConfiguration config)` metodo di estensione.

```
services.Configure<StorageOptions>(Configuration.GetSection("Storage"));
```

Nella fonte di configurazione della memoria

È anche possibile creare la configurazione da un oggetto in memoria come un `Dictionary<string,string>`

```
.AddInMemoryCollection(new Dictionary<string, string>
{
    ["akey"] = "a value"
})
```

Questo può rivelarsi utile in scenari di test di integrazione / unità.

Leggi Configurazione online: https://riptutorial.com/it/asp-net-core/topic/8660/configurazione

Capitolo 8: Configurazione di più ambienti

Examples

Avere appettazioni per ambiente

Per ogni ambiente è necessario creare un'app separata. {EnvironmentName} .json files:

- appsettings.Development.json
- appsettings.Staging.json
- appsettings.Production.json

Quindi aprire il file project.json e includerli in "include" nella sezione "publishOptions". Elenca tutti i file e le cartelle che verranno inclusi durante la pubblicazione:

```
"publishOptions": {
  "include": [
    "appsettings.Development.json",
    "appsettings.Staging.json",
    "appsettings.Production.json"
    ...
  ]
}
```

Il passo finale. Nella tua classe di avvio aggiungi:

```
.AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true);
```

nel costruttore in cui si impostano le fonti di configurazione:

```
var builder = new ConfigurationBuilder()
            .SetBasePath(env.ContentRootPath)
            .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
            .AddEnvironmentVariables();
```

Ottieni / Controlla il nome dell'ambiente dal codice

Tutto ciò di cui hai bisogno è una variabile di tipo `IHostingEnvironment` :

- ottieni il nome dell'ambiente:

  ```
  env.EnvironmentName
  ```

- per gli ambienti di `Development` , gestione `Staging` e `Production` predefiniti, il modo migliore è utilizzare i metodi di estensione dalla classe HostingEnvironmentExtensions

  ```
  env.IsDevelopment()
  ```

```
env.IsStaging()
env.IsProduction()
```

- ignorare correttamente case (un altro metodo di estensione da HostingEnvironmentExtensions :

```
env.IsEnvironment("environmentname")
```

- variante case sensitive:

```
env.EnvironmentName == "Development"
```

Configurazione di più ambienti

Questo esempio mostra come configurare più ambienti con diversa configurazione di Dependency Injection e middleware separati in una classe di `Startup` .

Oltre `public void Configure(IApplicationBuilder app)` e `public void ConfigureServices(IServiceCollection services)` , è possibile utilizzare i servizi `Configure{EnvironmentName}` e `Configure{EnvironmentName}Services` per avere una configurazione dipendente dall'ambiente.

L'uso di questo modello evita di mettere troppo `if/else` logica con un singolo metodo / classe di `Startup` e tenerlo pulito e separato.

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services) { }
    public void ConfigureStaggingServices(IServiceCollection services) { }
    public void ConfigureProductionServices(IServiceCollection services) { }

    public void Configure(IApplicationBuilder app) { }
    public void ConfigureStagging(IApplicationBuilder app) { }
    public void ConfigureProduction(IApplicationBuilder app) { }
}
```

Quando i servizi `Configure{Environmentname}` o `Configure{Environmentname}Services` non vengono trovati, rientrano rispettivamente in `Configure` o `ConfigureServices` .

La stessa semantica si applica anche alla classe di `Startup` . `StartupProduction` verrà utilizzato quando la variabile `ASPNETCORE_ENVIRONMENT` è impostata su `Production` e ricadrà `Startup` quando si tratta di `Stagging` o di `Development`

Un esempio completo:

```
public class Startup
{
    public Startup(IHostingEnvironment hostEnv)
    {
        // Set up configuration sources.
```

```
        var builder = new ConfigurationBuilder()
            .SetBasePath(hostEnv.ContentRootPath)
            .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true)
            .AddJsonFile($"appsettings.{hostEnv.EnvironmentName}.json", optional: true,
reloadOnChange: true);

        if (hostEnv.IsDevelopment())
        {
            // This will push telemetry data through Application Insights pipeline faster,
allowing you to view results immediately.
            builder.AddApplicationInsightsSettings(developerMode: true);
        }

        builder.AddEnvironmentVariables();
        Configuration = builder.Build();
    }

    public IConfigurationRoot Configuration { get; set; }

    // This method gets called by the runtime. Use this method to add services to the
container
    public static void RegisterCommonServices(IServiceCollection services)
    {
        services.AddScoped<ICommonService, CommonService>();
        services.AddScoped<ICommonRepository, CommonRepository>();
    }

    public void ConfigureServices(IServiceCollection services)
    {
        RegisterCommonServices(services);

        services.AddOptions();
        services.AddMvc();
    }

    public void ConfigureDevelopment(IApplicationBuilder app, IHostingEnvironment env,
ILoggerFactory loggerFactory)
    {
        loggerFactory.AddConsole(Configuration.GetSection("Logging"));
        loggerFactory.AddDebug();

        app.UseBrowserLink();
        app.UseDeveloperExceptionPage();

        app.UseApplicationInsightsRequestTelemetry();
        app.UseApplicationInsightsExceptionTelemetry();
        app.UseStaticFiles();
        app.UseMvc();
    }

    // No console Logger and debugging tools in this configuration
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
    {
        loggerFactory.AddDebug();

        app.UseApplicationInsightsRequestTelemetry();
        app.UseApplicationInsightsExceptionTelemetry();
        app.UseStaticFiles();
        app.UseMvc();
    }
```

```
}
```

Rendi visibile il contenuto specifico per l'ambiente

Potrebbe essere necessario rendere visibile del contenuto, che è specifico solo per alcuni ambienti. Per raggiungere questo obiettivo puoi utilizzare l' helper del tag Environment:

```
<environment names="Development">
    <h1>This is heading for development environment</h1>
</environment>
<environment names="Staging,Production">
    <h1>This is heading for Staging or production environment</h1>
</environment>
```

L'helper del tag Environment eseguirà il rendering solo dei suoi contenuti se l'ambiente corrente corrisponde a uno degli ambienti specificati usando l'attributo `names` .

Imposta la variabile di ambiente dalla riga di comando

Per impostare l'ambiente per `Development`

```
SET ASPNETCORE_ENVIRONMENT=Development
```

Ora l'esecuzione di un'applicazione Asp.Net Core sarà nell'ambiente definito.

Nota

1. Non dovrebbe esserci spazio prima e dopo il segno di uguaglianza `=` .
2. Il prompt dei comandi non dovrebbe essere chiuso prima di eseguire l'applicazione perché le impostazioni non sono persistenti.

Imposta la variabile di ambiente da PowerShell

Quando si utilizza PowerShell, è possibile utilizzare setx.exe per impostare le variabili di ambiente in modo permanente.

1. Avvia PowerShell
2. Digitare uno dei seguenti:

   setx ASPNETCORE_ENVIRONMENT "sviluppo"

   setx ASPNETCORE_ENVIRONMENT "staging"

3. Riavvia PowerShell

Utilizzo di ASPNETCORE_ENVIRONMENT da web.config

Se non si desidera utilizzare ASPNETCORE_ENVIRONMENT dalle variabili di ambiente e utilizzarlo da web.config dell'applicazione, quindi modificare web.config in questo modo-

```
<aspNetCore processPath=".\WebApplication.exe" arguments="" stdoutLogEnabled="false"
stdoutLogFile=".\logs\stdout" forwardWindowsAuthToken="false">
  <environmentVariables>
    <environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
  </environmentVariables>
</aspNetCore>
```

Leggi Configurazione di più ambienti online: https://riptutorial.com/it/asp-net-core/topic/2292/configurazione-di-piu-ambienti

Capitolo 9: Gestione degli errori

Examples

Reindirizza alla pagina di errore personalizzata

ASP.NET Core fornisce il middleware delle pagine di codici di stato , che supporta diversi metodi di estensione, ma siamo interessanti in `UseStatusCodePages` e `UseStatusCodePagesWithRedirects` :

- UseStatusCodePages aggiunge un middleware StatusCodePages con le opzioni fornite che controlla le risposte con codici di stato tra 400 e 599 che non hanno un corpo. Esempio di utilizzo per il reindirizzamento:

```
app.UseStatusCodePages(async context => {
  //context.HttpContext.Response.StatusCode contains the status code

  // your redirect logic

});
```

- UseStatusCodePagesWithRedirects aggiunge un middleware StatusCodePages alla pipeline. Specifica che le risposte devono essere gestite reindirizzando il modello di URL di posizione specificato. Questo può includere un segnaposto '{0}' per il codice di stato. Gli URL che iniziano con "~" includeranno PathBase, dove verrà utilizzato qualsiasi altro URL così com'è. Ad esempio il seguente reindirizzamento a ~ / errors / <codice_errore> (ad esempio ~ / errors / 403 per errore 403):

```
app.UseStatusCodePagesWithRedirects("~/errors/{0}");
```

Gestione globale delle eccezioni in ASP.NET Core

UseExceptionHandler può essere utilizzato per gestire le eccezioni a livello globale. È possibile ottenere tutti i dettagli dell'oggetto eccezione come Stack Trace, Inner exception e altri. E poi puoi mostrarli sullo schermo. Puoi facilmente implementare in questo modo.

```
app.UseExceptionHandler(
 options => {
    options.Run(
    async context =>
    {
      context.Response.StatusCode = (int)HttpStatusCode.InternalServerError;
      context.Response.ContentType = "text/html";
      var ex = context.Features.Get<IExceptionHandlerFeature>();
      if (ex != null)
      {
        var err = $"<h1>Error: {ex.Error.Message}</h1>{ex.Error.StackTrace }";
        await context.Response.WriteAsync(err).ConfigureAwait(false);
      }
    });
```

```
  }
);
```

È necessario inserire questo all'interno di configure () del file startup.cs.

Leggi Gestione degli errori online: https://riptutorial.com/it/asp-net-core/topic/6581/gestione-degli-errori

Capitolo 10: Iniezione di dipendenza

introduzione

Il core Aspnet è costruito con Dependency Injection come uno dei suoi concetti chiave. Introduce un'astrazione di contenitore conforme in modo che sia possibile sostituire quello incorporato con un contenitore di terze parti di propria scelta.

Sintassi

- `IServiceCollection.Add(ServiceDescriptor item);`
- `IServiceCollection.AddScoped(Type serviceType);`
- `IServiceCollection.AddScoped(Type serviceType, Type implementationType);`
- `IServiceCollection.AddScoped(Type serviceType, Func<IServiceProvider, object> implementationFactory);`
- `IServiceCollection.AddScoped<TService>()`
- `IServiceCollection.AddScoped<TService>(Func<IServiceProvider, TService> implementationFactory)`
- `IServiceCollection.AddScoped<TService, TImplementation>()`
- `IServiceCollection.AddScoped<TService, TImplementation>(Func<IServiceProvider, TImplementation> implementationFactory)`
- `IServiceCollection.AddSingleton(Type serviceType);`
- `IServiceCollection.AddSingleton(Type serviceType, Func<IServiceProvider, object> implementationFactory);`
- `IServiceCollection.AddSingleton(Type serviceType, Type implementationType);`
- `IServiceCollection.AddSingleton(Type serviceType, object implementationInstance);`
- `IServiceCollection.AddSingleton<TService>()`
- `IServiceCollection.AddSingleton<TService>(Func<IServiceProvider, TService> implementationFactory)`
- `IServiceCollection.AddSingleton<TService>(TService implementationInstance)`
- `IServiceCollection.AddSingleton<TService, TImplementation>()`
- `IServiceCollection.AddSingleton<TService, TImplementation>(Func<IServiceProvider, TImplementation> implementationFactory)`
- `IServiceCollection.AddTransient(Type serviceType);`
- `IServiceCollection.AddTransient(Type serviceType, Func<IServiceProvider, object> implementationFactory);`
- `IServiceCollection.AddTransient(Type serviceType, Type implementationType);`
- `IServiceCollection.AddTransient<TService>()`
- `IServiceCollection.AddTransient<TService>(Func<IServiceProvider, TService> implementationFactory)`
- `IServiceCollection.AddTransient<TService, TImplementation>()`
- `IServiceCollection.AddTransient<TService, TImplementation>(Func<IServiceProvider, TImplementation> implementationFactory)`
- `IServiceProvider.GetService(Type serviceType)`
- `IServiceProvider.GetService<T>()`
- `IServiceProvider.GetServices(Type serviceType)`
- `IServiceProvider.GetServices<T>()`

Osservazioni

Per utilizzare varianti generiche dei metodi `IServiceProvider` devi includere il seguente spazio dei nomi:

```
using Microsoft.Extensions.DependencyInjection;
```

Examples

Registrati e risolvi manualmente

Il modo migliore di descrivere le dipendenze è usando l'iniezione del costruttore che segue il principio delle dipendenze esplicite :

ITestService.cs

```
public interface ITestService
{
    int GenerateRandom();
}
```

TestService.cs

```
public class TestService : ITestService
{
    public int GenerateRandom()
    {
        return 4;
    }
}
```

Startup.cs (ConfigureServices)

```
public void ConfigureServices(IServiceCollection services)
{
    // ...

    services.AddTransient<ITestService, TestService>();
}
```

HomeController.cs

```
using Microsoft.Extensions.DependencyInjection;

namespace Core.Controllers
{
    public class HomeController : Controller
    {
        public HomeController(ITestService service)
        {
            int rnd = service.GenerateRandom();
        }
    }
}
```

Registrare dipendenze

Il contenitore incorporato include una serie di funzionalità integrate:

Controllo a vita

```
public void ConfigureServices(IServiceCollection services)
    {
        // ...

        services.AddTransient<ITestService, TestService>();
        // or
        services.AddScoped<ITestService, TestService>();
        // or
        services.AddSingleton<ITestService, TestService>();
        // or
        services.AddSingleton<ITestService>(new TestService());
    }
```

- **AddTransient** : creato ogni volta che viene risolto
- **AddScoped** : creato una volta per richiesta
- **AddSingleton** : creato in modo semplice una volta per applicazione
- **AddSingleton (istanza)** : fornisce un'istanza creata in precedenza per ogni applicazione

Dipendenze enumerabili

È anche possibile registrare dipendenze enumerabili:

```
 services.TryAddEnumerable(ServiceDescriptor.Transient<ITestService, TestServiceImpl1>());
 services.TryAddEnumerable(ServiceDescriptor.Transient<ITestService, TestServiceImpl2>());
```

Puoi quindi consumarli come segue:

```
public class HomeController : Controller
{
    public HomeController(IEnumerable<ITestService> services)
    {
        // do something with services.
    }
}
```

Dipendenze generiche

Puoi anche registrare le dipendenze generiche:

```
services.Add(ServiceDescriptor.Singleton(typeof(IKeyValueStore<>), typeof(KeyValueStore<>)));
```

E poi consumarlo come segue:

```
public class HomeController : Controller
{
    public HomeController(IKeyValueStore<UserSettings> userSettings)
    {
        // do something with services.
    }
}
```

Recupera dipendenze su un controller

Una volta registrata, una dipendenza può essere recuperata aggiungendo parametri al costruttore di Controller.

```
// ...
using System;
using Microsoft.Extensions.DependencyInjection;

namespace Core.Controllers
{
    public class HomeController : Controller
    {
        public HomeController(ITestService service)
        {
            int rnd = service.GenerateRandom();
        }
    }
}
```

Iniezione di una dipendenza in un'azione del controllore

Una funzione incorporata meno conosciuta è l'iniezione di Controller Action che utilizza `FromServicesAttribute` .

```
[HttpGet]
public async Task<IActionResult> GetAllAsync([FromServices]IProductService products)
{
     return Ok(await products.GetAllAsync());
}
```

Una nota importante è che i `[FromServices]` **non possono** essere utilizzati come meccanismo generale "Property Injection" o "Method injection"! Può essere utilizzato solo sui parametri del metodo di un'azione del controllore o del costruttore del controllore (nel costruttore è tuttavia obsoleto, poiché il sistema DI di base di ASP.NET utilizza già l'iniezione del costruttore e non sono richiesti marker aggiuntivi).

Non può essere utilizzato ovunque al di fuori di un controller, azione del controller . Inoltre è molto specifico per ASP.NET Core MVC e risiede nell'assembly `Microsoft.AspNetCore.Mvc.Core` .

Preventivo originale dal problema GitHub di ASP.NET Core MVC (Limite [FromServices] da applicare solo ai parametri) relativo a questo attributo:

> @rynowak:

> > @Eilon:
> >
> > Il problema con le proprietà è che sembra a molte persone che possa essere applicato a qualsiasi proprietà di qualsiasi oggetto.
>
> D'accordo, abbiamo avuto un numero di problemi pubblicati dagli utenti con confusione su come questa funzione dovrebbe essere utilizzata. C'è stata una grande quantità di feedback sia del tipo "[FromServices] è strano e non mi piace" e "[FromServices] mi ha confuso". Sembra una trappola, e qualcosa che il team avrebbe comunque risposto alle domande tra anni.
>
> Riteniamo che lo scenario più prezioso per [FromServices] sia sul parametro method su un'azione per un servizio di cui hai bisogno solo in quella posizione.
>
> / cc @ danroth27 - modifiche di documenti
>
> Per chiunque sia innamorato dell'attuale [FromServices], consiglio vivamente di cercare in un sistema DI che possa fare l'iniezione di proprietà (Autofac, ad esempio).

Gli appunti:

- **Qualsiasi** servizio registrato con il sistema .NET Core Dependency Injection può essere iniettato all'interno dell'azione di un controller utilizzando l'attributo `[FromServices]` .
- Il caso più rilevante è quando hai bisogno di un servizio solo in un singolo metodo di azione e non vuoi ingombrare il costruttore del controllore con un'altra dipendenza, che sarà usata una sola volta.
- Non può essere utilizzato al di fuori di ASP.NET Core MVC (ovvero puro .NET Framework o applicazioni della console .NET Core), poiché risiede nell'assembly `Microsoft.AspNetCore.Mvc.Core` .
- Per la proprietà o il metodo di iniezione è necessario utilizzare uno dei contenitori IoC di terze parti disponibili (Autofac, Unity, ecc.).

Il modello di opzioni / opzioni di iniezione nei servizi

Con ASP.NET Core, il team Microsoft ha anche introdotto il modello Options, che consente di disporre di potenti opzioni tipizzate e una volta configurata la possibilità di iniettare le opzioni nei servizi.

Per prima cosa iniziamo con una classe tipizzata forte, che manterrà la nostra configurazione.

```
public class MySettings
{
    public string Value1 { get; set; }
    public string Value2 { get; set; }
}
```

E una voce nel `appsettings.json` .

```
{
  "mysettings" : {
    "value1": "Hello",
    "value2": "World"
  }
}
```

Successivamente inizializziamo nella classe Startup. Ci sono due modi per farlo

1. `appsettings.json` direttamente dalla sezione `appsettings.json` "mysettings"

   ```
   services.Configure<MySettings>(Configuration.GetSection("mysettings"));
   ```

2. Fallo manualmente

   ```
   services.Configure<MySettings>(new MySettings
   {
       Value1 = "Hello",
       Value2 = Configuration["mysettings:value2"]
   });
   ```

   Ogni livello gerarchico di `appsettings.json` è separato da un `:` Poiché `value2` è una proprietà dell'oggetto `mysettings` , accediamo tramite `mysettings:value2` .

Infine possiamo iniettare le opzioni nei nostri servizi, usando l' `IOptions<T>`

```
public class MyService : IMyService
{
    private readonly MySettings settings;

    public MyService(IOptions<MySettings> mysettings)
    {
        this.settings = mySettings.Value;
    }
}
```

Osservazioni

Se le `IOptions<T>` non sono configurate durante l'avvio, l'iniezione di `IOptions<T>` inietterà l'istanza predefinita della classe `T`

Utilizzo dei servizi di ambito durante l'avvio dell'applicazione / Seeding del database

La risoluzione dei servizi con ambito durante l'avvio dell'applicazione può essere difficile, poiché non vi è alcuna richiesta e quindi nessun servizio con ambito.

La risoluzione di un servizio con ambito durante l'avvio dell'applicazione tramite `app.ApplicationServices.GetService<AppDbContext>()` può causare problemi, poiché verrà creata nell'ambito del contenitore globale, rendendola effettivamente un singleton con il ciclo di vita

dell'applicazione, che può portare alle eccezioni come `Cannot access a disposed object in ASP.NET Core when injecting DbContext` .

Il seguente schema risolve il problema creando innanzitutto un nuovo ambito e quindi risolvendo i servizi di ambito da esso, quindi, una volta terminato il lavoro, lo smaltimento del contenitore con ambito.

```
public Configure(IApplicationBuilder app)
{
    // serviceProvider is app.ApplicationServices from Configure(IApplicationBuilder app)
method
    using (var serviceScope =
app.ApplicationServices.GetRequiredService<IServiceScopeFactory>().CreateScope())
    {
        var db = serviceScope.ServiceProvider.GetService<AppDbContext>();

        if (await db.Database.EnsureCreatedAsync())
        {
            await SeedDatabase(db);
        }
    }
}
```

Questo è un modo semi-ufficiale del core team di Entity Framework per seminare i dati durante l'avvio dell'applicazione e si riflette nell'applicazione di esempio MusicStore .

Risolvi controller, ViewComponents e TagHelpers tramite Iniezione dipendenza

Per impostazione predefinita, Controller, ViewComponents e TagHelpers non sono registrati e risolti tramite il contenitore di dipendenze dell'iniezione. Ciò si traduce nell'incapacità di fare l'iniezione di proprietà quando si utilizza un contenitore di Inversion of Control (IoC) di terze parti come AutoFac.

Per fare in modo che ASP.NET Core MVC risolva questi tipi anche tramite IoC, è necessario aggiungere le seguenti registrazioni in `Startup.cs` (prelevate dall'esempio ufficiale di ControllersFromService su GitHub)

```
public void ConfigureServices(IServiceCollection services)
{
    var builder = services
        .AddMvc()
        .ConfigureApplicationPartManager(manager => manager.ApplicationParts.Clear())
        .AddApplicationPart(typeof(TimeScheduleController).GetTypeInfo().Assembly)
        .ConfigureApplicationPartManager(manager =>
        {
            manager.ApplicationParts.Add(new TypesPart(
              typeof(AnotherController),
              typeof(ComponentFromServicesViewComponent),
              typeof(InServicesTagHelper)));

            manager.FeatureProviders.Add(new AssemblyMetadataReferenceFeatureProvider());
        })
        .AddControllersAsServices()
```

```
        .AddViewComponentsAsServices()
        .AddTagHelpersAsServices();

    services.AddTransient<QueryValueService>();
    services.AddTransient<ValueService>();
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
}
```

Esempio di Plain Dependency Injection (senza Startup.cs)

Questo ti mostra come usare il pacchetto Microsoft.Extensions.DependencyInjection nuget senza l'uso di `WebHostBuilder` da kestrel (ad es. Quando vuoi costruire qualcos'altro di una webApp):

```
internal class Program
{
    public static void Main(string[] args)
    {
        var services = new ServiceCollection(); //Creates the service registry
        services.AddTransient<IMyInterface, MyClass>(); //Add registration of IMyInterface
(should create an new instance of MyClass every time)
        var serviceProvider = services.BuildServiceProvider(); //Build dependencies into an
IOC container
        var implementation = serviceProvider.GetService<IMyInterface>(); //Gets a dependency

        //serviceProvider.GetService<ServiceDependingOnIMyInterface>(); //Would throw an error
since ServiceDependingOnIMyInterface is not registered
        var manualyInstaniate = new ServiceDependingOnIMyInterface(implementation);

        services.AddTransient<ServiceDependingOnIMyInterface>();
        var spWithService = services.BuildServiceProvider(); //Generaly its bad practise to
rebuild the container because its heavey and promotes use of anti-pattern.
        spWithService.GetService<ServiceDependingOnIMyInterface>(); //only now i can resolve
    }
}

interface IMyInterface
{
}

class MyClass : IMyInterface
{
}

class ServiceDependingOnIMyInterface
{
    private readonly IMyInterface _dependency;

    public ServiceDependingOnIMyInterface(IMyInterface dependency)
    {
        _dependency = dependency;
    }
}
```

Funzionamento interno di Microsoft.Extensions.DependencyInjection

IServiceCollection

Per iniziare a costruire un container IOC con il pacchetto DI nuget di Microsoft, inizierai con la creazione di un servizio `IServiceCollection` . Puoi utilizzare la raccolta già fornita: `ServiceCollection` :

```
var services = new ServiceCollection();
```

Questo `IServiceCollection` non è altro che un'implementazione di: `IList<ServiceDescriptor>, ICollection<ServiceDescriptor>, IEnumerable<ServiceDescriptor>, IEnumerable`

Tutti i seguenti metodi sono solo metodi di estensione per aggiungere istanze `ServiceDescriptor` all'elenco:

```
services.AddTransient<Class>(); //add registration that is always recreated
services.AddSingleton<Class>(); // add registration that is only created once and then re-used
services.AddTransient<Abstract, Implementation>(); //specify implementation for interface
services.AddTransient<Interface>(serviceProvider=> new
Class(serviceProvider.GetService<IDependency>())); //specify your own resolve function/
factory method.
services.AddMvc(); //extension method by the MVC nuget package, to add a whole bunch of
registrations.
// etc..

//when not using an extension method:
services.Add(new ServiceDescriptor(typeof(Interface), typeof(Class)));
```

IServiceProvider

Il provider di servizi è quello che "Compila" tutte le registrazioni in modo che possano essere utilizzate rapidamente, ciò può essere fatto con `services.BuildServiceProvider()` che è fondamentalmente un mehtod di estensione per:

```
var provider = new ServiceProvider( services, false); //false is if it should validate scopes
```

Dietro le quinte ogni `ServiceDescriptor` in `IServiceCollection` viene compilato con un metodo factory `Func<ServiceProvider, object>` dove object è il tipo restituito ed è: l'istanza creata del tipo Implementation, Singleton o il proprio metodo factory definito.

Queste registrazioni vengono aggiunte al `ServiceTable` che è fondamentalmente un `ConcurrentDictionary` con la chiave come `ServiceType` e il valore del metodo Factory definito sopra.

Risultato

Ora disponiamo di un `ConcurrentDictionary<Type, Func<ServiceProvider, object>>` che possiamo usare contemporaneamente per chiedere di creare servizi per noi. Per mostrare un esempio di base di come questo avrebbe potuto sembrare.

```
var serviceProvider = new ConcurrentDictionary<Type, Func<ServiceProvider, object>>();
var factoryMethod = serviceProvider[typeof(MyService)];
var myServiceInstance = factoryMethod(serviceProvider)
```

Non è così che funziona!

Questo `ConcurrentDictionary` è una proprietà del `ServiceTable` che è una proprietà del `ServiceProvider`

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/asp-net-core/topic/1949/iniezione-di-dipendenza

Capitolo 11: Iniezione di servizi in viste

Sintassi

- `@inject<NameOfService><Identifier>`
- `@<Identifier>.Foo()`
- @inject <tipo> <nome>

Examples

La direttiva @inject

ASP.NET Core introduce il concetto di dipendenza da iniezione in Views tramite la direttiva `@inject` tramite la seguente sintassi:

```
@inject <type> <name>
```

Esempio di utilizzo

L'aggiunta di questa direttiva nella vista genererà fondamentalmente una proprietà del tipo specificato utilizzando il nome specificato all'interno della vista utilizzando l'iniezione di dipendenza appropriata come illustrato nell'esempio seguente:

```
@inject YourWidgetServiceClass WidgetService

<!-- This would call the service, which is already populated and output the results -->
There are <b>@WidgetService.GetWidgetCount()</b> Widgets here.
```

Configurazione richiesta

I servizi che utilizzano l'iniezione delle dipendenze devono ancora essere registrati all'interno del metodo `ConfigureServices()` del file `Startup.cs` e con scope di conseguenza:

```
public void ConfigureServices(IServiceCollection services)
{
     // Other stuff omitted for brevity

     services.AddTransient<IWidgetService, WidgetService>();
}
```

Leggi Iniezione di servizi in viste online: https://riptutorial.com/it/asp-net-core/topic/4284/iniezione-di-servizi-in-viste

Capitolo 12: Invio di email in app .Net core tramite MailKit

introduzione

Attualmente .Net Core non include il supporto per l'invio di e-mail come `System.Net.Mail` da .Net. Il progetto MailKit (che è disponibile su nuget) è una bella libreria per questo scopo.

Examples

Installare il pacchetto nuget

```
Install-Package MailKit
```

Semplice implementazione per l'invio di e-mail

```
using MailKit.Net.Smtp;
using MimeKit;
using MimeKit.Text;
using System.Threading.Tasks;

namespace Project.Services
{
    /// Using a static class to store sensitive credentials
    /// for simplicity. Ideally these should be stored in
    /// configuration files
    public static class Constants
    {
        public static string SenderName => "<sender_name>";
        public static string SenderEmail => "<sender_email>";
        public static string EmailPassword => "email_password";
        public static string SmtpHost => "<smtp_host>";
        public static int SmtpPort => "smtp_port";
    }
    public class EmailService : IEmailSender
    {
        public Task SendEmailAsync(string recipientEmail, string subject, string message)
        {
            MimeMessage mimeMessage = new MimeMessage();
            mimeMessage.From.Add(new MailboxAddress(Constants.SenderName,
Constants.SenderEmail));
            mimeMessage.To.Add(new MailboxAddress("", recipientEmail));
            mimeMessage.Subject = subject;

            mimeMessage.Body = new TextPart(TextFormat.Html)
            {
                Text = message,
            };

            using (var client = new SmtpClient())
            {
```

```
                client.ServerCertificateValidationCallback = (s, c, h, e) => true;

                client.Connect(Constants.SmtpHost, Constants.SmtpPort, false);

                client.AuthenticationMechanisms.Remove("XOAUTH2");

                // Note: only needed if the SMTP server requires authentication
                client.Authenticate(Constants.SenderEmail, Constants.EmailPassword);

                client.Send(mimeMessage);

                client.Disconnect(true);
                return Task.FromResult(0);
            }
        }
    }

}
```

Leggi Invio di email in app .Net core tramite MailKit online: https://riptutorial.com/it/asp-net-core/topic/8831/invio-di-email-in-app--net-core-tramite-mailkit

Capitolo 13: Lavorare con JavascriptServices

introduzione

Secondo la documentazione ufficiale:

`JavaScriptServices` è un insieme di tecnologie per gli sviluppatori di ASP.NET Core. Fornisce un'infrastruttura che ti sarà utile se utilizzi Angular 2 / React / Knockout / ecc. Sul client o se sviluppi le risorse lato client con Webpack o desideri eseguire JavaScript sul server in fase di runtime.

Examples

Abilitazione di webpack-dev-middleware per il progetto asp.net-core

Supponiamo che tu usi `Webpack` per il bundle front-end. Puoi aggiungere `webpack-dev-middleware` per servire le tue statistiche attraverso server piccoli e veloci. Ti consente di ricaricare automaticamente le tue risorse quando il contenuto è cambiato, di servire le statiche in memoria senza scrivere continuamente versioni intermedie su disco.

Prerequisiti

NuGet

Pacchetto di installazione Microsoft.AspNetCore.SpaServices

npm

npm install --save-dev aspnet-webpack, webpack-dev-middleware, webpack-dev-server

Configurazione

Estendi il metodo `Configure` nella tua classe di `Startup`

```
if (env.IsDevelopment())
{
     app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions()
     {
         ConfigFile = "webpack.config.js" //this is defualt value
     });
}
```

Aggiungi sostituzione modulo caldo (HMR)

La sostituzione del modulo caldo consente di aggiungere, modificare o eliminare il modulo dell'app quando l'applicazione è in esecuzione. Il ricaricamento della pagina non è necessario in questo caso.

Prerequisiti

Oltre ai pacchetti `webpack-dev-middleware` :

```
npm install --save-dev webpack-hot-middleware
```

Configurazione

Basta aggiornare la configurazione di `UseWebpackDevMiddleware` con nuove opzioni:

```
app.UseWebpackDevMiddleware(new WebpackDevMiddlewareOptions()
{
    ConfigFile = "webpack.config.js", //this is defualt value
    HotModuleReplacement = true,
    ReactHotModuleReplacement = true, //for React only
});
```

È inoltre necessario accettare i moduli hot nel codice dell'app.

HMR è supportato per Angular 2, React, Knockout e Vue.

Generazione di una singola applicazione di esempio con il nucleo di asp.net

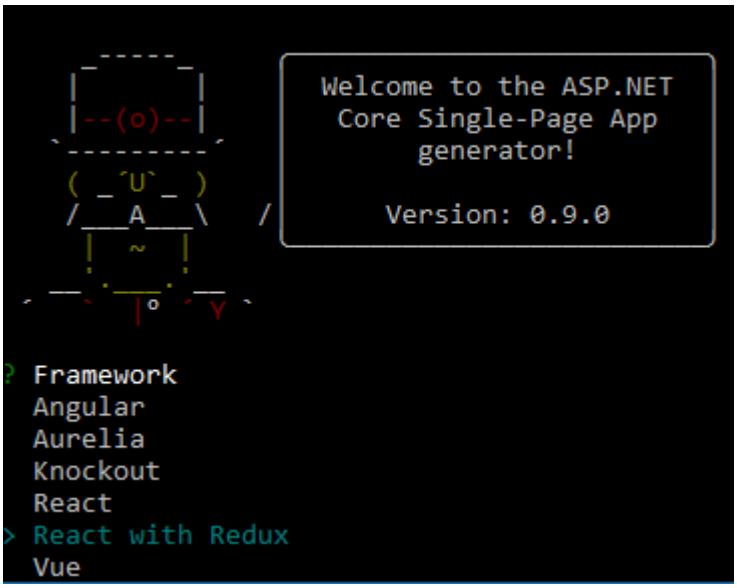
È possibile utilizzare il generatore `aspnetcore-spa` per `Yeoman` per creare un'applicazione singola pagina nuova di zecca con asp.net core.

Ciò consente di scegliere uno dei framework front-end più diffusi e di generare progetti con webpack, server di sviluppo, sostituzione di moduli caldi e funzionalità di rendering lato server.

Corri

```
npm install -g yo generator-aspnetcore-spa
cd newproject
yo aspnetcore-spa
```

e scegli la tua struttura preferita



Leggi Lavorare con JavascriptServices online: https://riptutorial.com/it/asp-net-core/topic/9621/lavorare-con-javascriptservices

Capitolo 14: Limitazione di velocità

Osservazioni

AspNetCoreRateLimit è una soluzione di limitazione della velocità di base di ASP.NET open source progettata per controllare la velocità delle richieste che i client possono eseguire su un'API Web o un'app MVC in base all'indirizzo IP o all'ID client.

Examples

Limitazione della velocità in base all'IP del cliente

Con il middleware IpRateLimit è possibile impostare più limiti per diversi scenari come consentire a un intervallo IP o IP di effettuare un numero massimo di chiamate in un intervallo di tempo come al secondo, 15 minuti, ecc. È possibile definire questi limiti per indirizzare tutte le richieste fatte ad un API o puoi limitare i limiti a ciascun percorso URL o verbo e percorso HTTP.

Impostare

Installa NuGet :

```
Install-Package AspNetCoreRateLimit
```

Codice Startup.cs :

```
public void ConfigureServices(IServiceCollection services)
{
    // needed to load configuration from appsettings.json
    services.AddOptions();

    // needed to store rate limit counters and ip rules
    services.AddMemoryCache();

    //load general configuration from appsettings.json
    services.Configure<IpRateLimitOptions>(Configuration.GetSection("IpRateLimiting"));

    //load ip rules from appsettings.json
    services.Configure<IpRateLimitPolicies>(Configuration.GetSection("IpRateLimitPolicies"));

    // inject counter and rules stores
    services.AddSingleton<IIpPolicyStore, MemoryCacheIpPolicyStore>();
    services.AddSingleton<IRateLimitCounterStore, MemoryCacheRateLimitCounterStore>();

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

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
```

```
    loggerFactory.AddDebug();

    app.UseIpRateLimiting();

    app.UseMvc();
}
```

È necessario registrare il middleware prima di qualsiasi altro componente tranne loggerFactory.

se si carica il bilancio dell'app, sarà necessario utilizzare `IDistributedCache` con Redis o SQLServer in modo che tutte le istanze del gheppio abbiano lo stesso limite di frequenza. Invece dei negozi in memoria dovresti iniettare i negozi distribuiti in questo modo:

```
    // inject counter and rules distributed cache stores
    services.AddSingleton<IIpPolicyStore, DistributedCacheIpPolicyStore>();
    services.AddSingleton<IRateLimitCounterStore,DistributedCacheRateLimitCounterStore>();
```

Configurazione e regole generali appsettings.json :

```
  "IpRateLimiting": {
    "EnableEndpointRateLimiting": false,
    "StackBlockedRequests": false,
    "RealIpHeader": "X-Real-IP",
    "ClientIdHeader": "X-ClientId",
    "HttpStatusCode": 429,
    "IpWhitelist": [ "127.0.0.1", "::1/10", "192.168.0.0/24" ],
    "EndpointWhitelist": [ "get:/api/license", "*:/api/status" ],
    "ClientWhitelist": [ "dev-id-1", "dev-id-2" ],
    "GeneralRules": [
      {
        "Endpoint": "*",
        "Period": "1s",
        "Limit": 2
      },
      {
        "Endpoint": "*",
        "Period": "15m",
        "Limit": 100
      },
      {
        "Endpoint": "*",
        "Period": "12h",
        "Limit": 1000
      },
      {
        "Endpoint": "*",
        "Period": "7d",
        "Limit": 10000
      }
    ]
  }
```

Se `EnableEndpointRateLimiting` è impostato su `false` i limiti verranno applicati globalmente e verranno applicate solo le regole che hanno come endpoint `*` . Ad esempio, se imposti un limite di 5 chiamate al secondo, qualsiasi chiamata HTTP verso qualsiasi endpoint verrà conteggiata a tale limite.

Se `EnableEndpointRateLimiting` è impostato su `true` i limiti si applicano a ciascun endpoint come in `{HTTP_Verb}{PATH}` . Ad esempio, se si imposta un limite di 5 chiamate al secondo per `*:/api/values` un client può chiamare `GET /api/values` 5 volte al secondo, ma anche 5 volte `PUT /api/values` .

Se `StackBlockedRequests` è impostato su `false` chiamate rifiutate non vengono aggiunte al contatore del throttle. Se un cliente effettua 3 richieste al secondo e hai impostato un limite di una chiamata al secondo, altri limiti come i contatori al minuto o al giorno registreranno solo la prima chiamata, quella che non è stata bloccata. Se vuoi che le richieste respinte `StackBlockedRequests` conteggiate rispetto agli altri limiti, dovrai impostare `StackBlockedRequests` su `true` .

`RealIpHeader` viene utilizzato per estrarre l'IP del client quando il server Kestrel si trova dietro un proxy inverso, se il proxy utilizza un'intestazione diversa, quindi `X-Real-IP` utilizza questa opzione per configurarlo.

`ClientIdHeader` viene utilizzato per estrarre l'ID client per l'elenco bianco, se un ID client è presente in questa intestazione e corrisponde a un valore specificato in ClientWhitelist, quindi non vengono applicati limiti di velocità.

Sovrascrivi le regole generali per specifici IP appsettings.json :

```
"IpRateLimitPolicies": {
   "IpRules": [
     {
       "Ip": "84.247.85.224",
       "Rules": [
         {
           "Endpoint": "*",
           "Period": "1s",
           "Limit": 10
         },
         {
           "Endpoint": "*",
           "Period": "15m",
           "Limit": 200
         }
       ]
     },
     {
       "Ip": "192.168.3.22/25",
       "Rules": [
         {
           "Endpoint": "*",
           "Period": "1s",
           "Limit": 5
         },
         {
           "Endpoint": "*",
           "Period": "15m",
           "Limit": 150
         },
         {
           "Endpoint": "*",
           "Period": "12h",
           "Limit": 500
         }
       ]
```

```
      }
    ]
  }
```

Il campo IP supporta i valori IP v4 e v6 e intervalli come "192.168.0.0/24", "fe80 :: / 10" o "192.168.0.0-192.168.0.255".

Definizione delle regole del limite di velocità

Una regola è composta da un endpoint, un periodo e un limite.

Il formato `{HTTP_Verb}:{PATH}` è `{HTTP_Verb}:{PATH}` , puoi indirizzare qualunque verbo HTTP usando il simbolo asterix.

Il formato del periodo è `{INT}{PERIOD_TYPE}` , puoi utilizzare uno dei seguenti tipi di periodo: `s, m, h, d` .

Il formato limite è `{LONG}` .

Esempi :

Limita il numero di tutti i punti finali a 2 chiamate al secondo:

```
{
 "Endpoint": "*",
 "Period": "1s",
 "Limit": 2
}
```

Se, dallo stesso IP, nello stesso secondo, effettuerai 3 chiamate GET su API / valori, l'ultima chiamata verrà bloccata. Ma se nello stesso secondo si chiama PUT api / valori, la richiesta passerà perché è un diverso endpoint. Quando la limitazione della velocità dell'endpoint è abilitata, ogni chiamata viene limitata in base a `{HTTP_Verb}{PATH}` .

Limite di velocità chiamate con qualsiasi Verbo HTTP a `/api/values` a 5 chiamate per 15 minuti:

```
{
 "Endpoint": "*:/api/values",
 "Period": "15m",
 "Limit": 5
}
```

Limite di velocità GET chiamata a `/api/values` a 5 chiamate all'ora:

```
{
 "Endpoint": "get:/api/values",
 "Period": "1h",
 "Limit": 5
}
```

Se, dallo stesso IP, in un'ora, effettuerai 6 GET chiamate su API / valori, l'ultima chiamata verrà

bloccata. Ma se nella stessa ora chiami GET api / values / 1, la richiesta verrà eseguita perché si tratta di un diverso endpoint.

Comportamento

Quando un client effettua una chiamata HTTP, IpRateLimitMiddleware procede come segue:

- estrae IP, ID client, verbo HTTP e URL dall'oggetto richiesta, se vuoi implementare la tua logica di estrazione puoi sovrascrivere `IpRateLimitMiddleware.SetIdentity`
- cerca l'IP, l'ID del cliente e l'URL nelle liste bianche, in caso di corrispondenza, non viene eseguita alcuna azione
- cerca nelle regole IP per una corrispondenza, tutte le regole che si applicano sono raggruppate per periodo, per ogni periodo viene utilizzata la regola più restrittiva
- cerca nelle regole generali per una corrispondenza, se una regola generale che corrisponde ha un periodo definito che non è presente nelle regole IP, viene utilizzata anche questa regola generale
- per ogni regola corrispondente il contatore del limite di velocità viene incrementato, se il valore del contatore è maggiore del limite della regola, allora la richiesta viene bloccata

Se la richiesta viene bloccata, il client riceve una risposta testuale come questa:

```
Status Code: 429
Retry-After: 58
Content: API calls quota exceeded! maximum admitted 2 per 1m.
```

È possibile personalizzare la risposta modificando queste opzioni `HttpStatusCode` e `QuotaExceededMessage` , se si desidera implementare la propria risposta è possibile sovrascrivere `IpRateLimitMiddleware.ReturnQuotaExceededResponse` . Il valore dell'intestazione `Retry-After` è espresso in secondi.

Se la richiesta non ottiene un tasso limitato, il periodo più lungo definito nelle regole di corrispondenza viene utilizzato per comporre le intestazioni X-Rate-Limit, queste intestazioni vengono iniettate nella risposta:

```
X-Rate-Limit-Limit: the rate limit period (eg. 1m, 12h, 1d)
X-Rate-Limit-Remaining: number of request remaining
X-Rate-Limit-Reset: UTC date time when the limits resets
```

Per impostazione predefinita, la richiesta bloccata viene registrata utilizzando `Microsoft.Extensions.Logging.ILogger` , se si desidera implementare la propria registrazione, è possibile sovrascrivere `IpRateLimitMiddleware.LogBlockedRequest` . Il logger predefinito emette le seguenti informazioni quando una richiesta ottiene un limite tariffario:

```
info: AspNetCoreRateLimit.IpRateLimitMiddleware[0]
      Request get:/api/values from IP 84.247.85.224 has been blocked, quota 2/1m exceeded by
3. Blocked by rule *:/api/value, TraceIdentifier 0HKTLISQQVV9D.
```

Aggiorna i limiti di velocità in fase di esecuzione

All'avvio dell'applicazione, le regole del limite di velocità IP definite in `appsettings.json` vengono caricate nella cache da `MemoryCacheClientPolicyStore` o `DistributedCacheIpPolicyStore` seconda del tipo di provider di cache che si sta utilizzando. È possibile accedere all'archivio dei criteri IP all'interno di un controller e modificare le regole IP in questo modo:

```
public class IpRateLimitController : Controller
{
    private readonly IpRateLimitOptions _options;
    private readonly IIpPolicyStore _ipPolicyStore;

    public IpRateLimitController(IOptions<IpRateLimitOptions> optionsAccessor, IIpPolicyStore
ipPolicyStore)
    {
        _options = optionsAccessor.Value;
        _ipPolicyStore = ipPolicyStore;
    }

    [HttpGet]
    public IpRateLimitPolicies Get()
    {
        return _ipPolicyStore.Get(_options.IpPolicyPrefix);
    }

    [HttpPost]
    public void Post()
    {
        var pol = _ipPolicyStore.Get(_options.IpPolicyPrefix);

        pol.IpRules.Add(new IpRateLimitPolicy
        {
            Ip = "8.8.4.4",
            Rules = new List<RateLimitRule>(new RateLimitRule[] {
                new RateLimitRule {
                    Endpoint = "*:/api/testupdate",
                    Limit = 100,
                    Period = "1d" }
            })
        });

        _ipPolicyStore.Set(_options.IpPolicyPrefix, pol);
    }
}
```

In questo modo è possibile memorizzare i limiti di velocità IP in un database e inserirli nella cache dopo l'avvio di ogni app.

Limitazione della velocità in base all'ID cliente

Con il middleware ClientRateLimit puoi impostare più limiti per diversi scenari come consentire al Cliente di effettuare un numero massimo di chiamate in un intervallo di tempo come al secondo, 15 minuti, ecc. Puoi definire questi limiti per soddisfare tutte le richieste fatte a un'API o tu può raggiungere i limiti di ogni percorso URL o verbo e percorso HTTP.

Impostare

Installa NuGet :

```
Install-Package AspNetCoreRateLimit
```

Codice Startup.cs :

```
public void ConfigureServices(IServiceCollection services)
{
    // needed to load configuration from appsettings.json
    services.AddOptions();

    // needed to store rate limit counters and ip rules
    services.AddMemoryCache();

    //load general configuration from appsettings.json

services.Configure<ClientRateLimitOptions>(Configuration.GetSection("ClientRateLimiting"));

    //load client rules from appsettings.json

services.Configure<ClientRateLimitPolicies>(Configuration.GetSection("ClientRateLimitPolicies"));


    // inject counter and rules stores
    services.AddSingleton<IClientPolicyStore, MemoryCacheClientPolicyStore>();
    services.AddSingleton<IRateLimitCounterStore, MemoryCacheRateLimitCounterStore>();

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

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    app.UseClientRateLimiting();

    app.UseMvc();
}
```

È necessario registrare il middleware prima di qualsiasi altro componente tranne loggerFactory.

se si carica il bilancio dell'app, sarà necessario utilizzare `IDistributedCache` con Redis o SQLServer in modo che tutte le istanze del gheppio abbiano lo stesso limite di frequenza. Invece dei negozi in memoria dovresti iniettare i negozi distribuiti in questo modo:

```
    // inject counter and rules distributed cache stores
    services.AddSingleton<IClientPolicyStore, DistributedCacheClientPolicyStore>();
    services.AddSingleton<IRateLimitCounterStore,DistributedCacheRateLimitCounterStore>();
```

Configurazione e regole generali appsettings.json :

```
  "ClientRateLimiting": {
    "EnableEndpointRateLimiting": false,
    "StackBlockedRequests": false,
```

```
    "ClientIdHeader": "X-ClientId",
    "HttpStatusCode": 429,
    "EndpointWhitelist": [ "get:/api/license", "*:/api/status" ],
    "ClientWhitelist": [ "dev-id-1", "dev-id-2" ],
    "GeneralRules": [
      {
        "Endpoint": "*",
        "Period": "1s",
        "Limit": 2
      },
      {
        "Endpoint": "*",
        "Period": "15m",
        "Limit": 100
      },
      {
        "Endpoint": "*",
        "Period": "12h",
        "Limit": 1000
      },
      {
        "Endpoint": "*",
        "Period": "7d",
        "Limit": 10000
      }
    ]
  }
```

Se `EnableEndpointRateLimiting` è impostato su `false` i limiti verranno applicati globalmente e verranno applicate solo le regole che hanno come endpoint `*` . Ad esempio, se imposti un limite di 5 chiamate al secondo, qualsiasi chiamata HTTP verso qualsiasi endpoint verrà conteggiata a tale limite.

Se `EnableEndpointRateLimiting` è impostato su `true` i limiti si applicano a ciascun endpoint come in `{HTTP_Verb}{PATH}` . Ad esempio, se si imposta un limite di 5 chiamate al secondo per `*:/api/values` un client può chiamare `GET /api/values` 5 volte al secondo, ma anche 5 volte `PUT /api/values` .

Se `StackBlockedRequests` è impostato su `false` chiamate rifiutate non vengono aggiunte al contatore del throttle. Se un cliente effettua 3 richieste al secondo e hai impostato un limite di una chiamata al secondo, altri limiti come i contatori al minuto o al giorno registreranno solo la prima chiamata, quella che non è stata bloccata. Se vuoi che le richieste respinte `StackBlockedRequests` conteggiate rispetto agli altri limiti, dovrai impostare `StackBlockedRequests` su `true` .

`ClientIdHeader` viene utilizzato per estrarre l'id del client, se un ID client è presente in questa intestazione e corrisponde a un valore specificato in ClientWhitelist, quindi non vengono applicati limiti di velocità.

Sostituisci le regole generali per client specifici appsettings.json :

```
"ClientRateLimitPolicies": {
    "ClientRules": [
      {
        "ClientId": "client-id-1",
        "Rules": [
          {
```

```
            "Endpoint": "*",
            "Period": "1s",
            "Limit": 10
          },
          {
            "Endpoint": "*",
            "Period": "15m",
            "Limit": 200
          }
        ]
      },
      {
        "Client": "client-id-2",
        "Rules": [
          {
            "Endpoint": "*",
            "Period": "1s",
            "Limit": 5
          },
          {
            "Endpoint": "*",
            "Period": "15m",
            "Limit": 150
          },
          {
            "Endpoint": "*",
            "Period": "12h",
            "Limit": 500
          }
        ]
      }
    ]
  }
```

Definizione delle regole del limite di velocità

Una regola è composta da un endpoint, un periodo e un limite.

Il formato `{HTTP_Verb}:{PATH}` è `{HTTP_Verb}:{PATH}` , puoi indirizzare qualunque verbo HTTP usando il simbolo asterix.

Il formato del periodo è `{INT}{PERIOD_TYPE}` , puoi utilizzare uno dei seguenti tipi di periodo: `s, m, h, d` .

Il formato limite è `{LONG}` .

Esempi :

Limita il numero di tutti i punti finali a 2 chiamate al secondo:

```
{
 "Endpoint": "*",
 "Period": "1s",
 "Limit": 2
}
```

Se nello stesso secondo, un client effettua 3 chiamate GET su API / valori, l'ultima chiamata verrà bloccata. Ma se nello stesso secondo chiama anche PIP api / valori, la richiesta passerà perché è un diverso endpoint. Quando la limitazione della velocità dell'endpoint è abilitata, ogni chiamata viene limitata in base a `{HTTP_Verb}{PATH}` .

Limite di velocità chiamate con qualsiasi Verbo HTTP a `/api/values` a 5 chiamate per 15 minuti:

```
{
 "Endpoint": "*:/api/values",
 "Period": "15m",
 "Limit": 5
}
```

Limite di velocità GET chiamata a `/api/values` a 5 chiamate all'ora:

```
{
 "Endpoint": "get:/api/values",
 "Period": "1h",
 "Limit": 5
}
```

Se in un'ora un client effettua 6 chiamate GET su API / valori, l'ultima chiamata verrà bloccata. Ma se nella stessa ora chiama GET api / values / 1, la richiesta passerà perché è un diverso endpoint.

Comportamento

Quando un client effettua una chiamata HTTP, ClientRateLimitMiddleware effettua quanto segue:

- estrae l'ID del client, il verbo HTTP e l'URL dall'oggetto della richiesta, se vuoi implementare la tua logica di estrazione puoi sovrascrivere il `ClientRateLimitMiddleware.SetIdentity`
- cerca l'id e l'URL del cliente negli elenchi bianchi, in caso di corrispondenza, non viene eseguita alcuna azione
- cerca nelle regole del cliente una corrispondenza, tutte le regole che si applicano sono raggruppate per periodo, per ogni periodo viene utilizzata la regola più restrittiva
- cerca nelle regole generali per una corrispondenza, se una regola generale che corrisponde ha un periodo definito che non è presente nelle regole del client, allora viene utilizzata anche questa regola generale
- per ogni regola corrispondente il contatore del limite di velocità viene incrementato, se il valore del contatore è maggiore del limite della regola, allora la richiesta viene bloccata

Se la richiesta viene bloccata, il client riceve una risposta testuale come questa:

```
Status Code: 429
Retry-After: 58
Content: API calls quota exceeded! maximum admitted 2 per 1m.
```

È possibile personalizzare la risposta modificando queste opzioni `HttpStatusCode` e `QuotaExceededMessage` , se si desidera implementare la propria risposta è possibile sovrascrivere `ClientRateLimitMiddleware.ReturnQuotaExceededResponse` . Il valore dell'intestazione `Retry-After` è espresso in secondi.

Se la richiesta non ottiene un tasso limitato, il periodo più lungo definito nelle regole di corrispondenza viene utilizzato per comporre le intestazioni X-Rate-Limit, queste intestazioni vengono iniettate nella risposta:

```
X-Rate-Limit-Limit: the rate limit period (eg. 1m, 12h, 1d)
X-Rate-Limit-Remaining: number of request remaining
X-Rate-Limit-Reset: UTC date time when the limits resets
```

Per impostazione predefinita, la richiesta bloccata viene registrata utilizzando `Microsoft.Extensions.Logging.ILogger` , se si desidera implementare la propria registrazione, è possibile ignorare `ClientRateLimitMiddleware.LogBlockedRequest` . Il logger predefinito emette le seguenti informazioni quando una richiesta ottiene un limite tariffario:

```
info: AspNetCoreRateLimit.ClientRateLimitMiddleware[0]
      Request get:/api/values from ClientId client-id-1 has been blocked, quota 2/1m exceeded
by 3. Blocked by rule *:/api/value, TraceIdentifier 0HKTLISQQVV9D.
```

Aggiorna i limiti di velocità in fase di esecuzione

All'avvio dell'applicazione, le regole del limite di frequenza client definite in `appsettings.json` vengono caricate nella cache da `MemoryCacheClientPolicyStore` o `DistributedCacheClientPolicyStore` seconda del tipo di provider di cache che si sta utilizzando. È possibile accedere all'archivio delle politiche client all'interno di un controller e modificare le regole in questo modo:

```
public class ClientRateLimitController : Controller
{
    private readonly ClientRateLimitOptions _options;
    private readonly IClientPolicyStore _clientPolicyStore;

    public ClientRateLimitController(IOptions<ClientRateLimitOptions> optionsAccessor,
IClientPolicyStore clientPolicyStore)
    {
        _options = optionsAccessor.Value;
        _clientPolicyStore = clientPolicyStore;
    }

    [HttpGet]
    public ClientRateLimitPolicy Get()
    {
        return _clientPolicyStore.Get($"{_options.ClientPolicyPrefix}_cl-key-1");
    }

    [HttpPost]
    public void Post()
    {
        var id = $"{_options.ClientPolicyPrefix}_cl-key-1";
        var clPolicy = _clientPolicyStore.Get(id);
        clPolicy.Rules.Add(new RateLimitRule
        {
            Endpoint = "*/api/testpolicyupdate",
            Period = "1h",
            Limit = 100
        });
        _clientPolicyStore.Set(id, clPolicy);
```

```
    }
}
```

In questo modo è possibile memorizzare i limiti di velocità del client in un database e inserirli nella cache dopo l'avvio di ogni app.

Leggi Limitazione di velocità online: https://riptutorial.com/it/asp-net-core/topic/5240/limitazione-di-velocita

Capitolo 15: Localizzazione

Examples

Localizzazione tramite risorse di linguaggio JSON

In ASP.NET Core ci sono diversi modi in cui possiamo localizzare / globalizzare la nostra app. È importante scegliere un modo adatto alle tue esigenze. In questo esempio vedremo come possiamo creare un'app di ASP.NET multilingue che legge stringhe specifiche della lingua da file `.json` e li memorizza in memoria per fornire la localizzazione in tutte le sezioni dell'app e mantenere alte prestazioni.

Il modo in cui lo facciamo è utilizzando il pacchetto `Microsoft.EntityFrameworkCore.InMemory` .

Gli appunti:

1. Lo spazio dei nomi per questo progetto è `DigitalShop` che puoi modificare lo spazio dei nomi del tuo progetto
2. Prendi in considerazione la possibilità di creare un nuovo progetto in modo da non incorrere in strani errori
3. In nessun modo questo esempio mostra le migliori pratiche, quindi se pensi che possa essere migliorato, ti preghiamo gentilmente di modificarlo

Per iniziare, aggiungiamo i seguenti pacchetti alla sezione delle `dependencies` **esistenti** nel file `project.json` :

```
"Microsoft.EntityFrameworkCore": "1.0.0",
"Microsoft.EntityFrameworkCore.SqlServer": "1.0.0",
"Microsoft.EntityFrameworkCore.InMemory": "1.0.0"
```

Ora cerchiamo di sostituire lo `Startup.cs` file con: (`using` le dichiarazioni vengono rimossi in quanto possono essere facilmente aggiunti in seguito)

Startup.cs

```
namespace DigitalShop
{
    public class Startup
    {
        public static string UiCulture;
        public static string CultureDirection;
        public static IStringLocalizer _e; // This is how we access language strings

        public static IConfiguration LocalConfig;

        public Startup(IHostingEnvironment env)
        {
            var builder = new ConfigurationBuilder()
                .SetBasePath(env.ContentRootPath)
                .AddJsonFile("appsettings.json", optional: false, reloadOnChange: true) //
```

```
this is where we store apps configuration including language
                .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
                .AddEnvironmentVariables();

            Configuration = builder.Build();
            LocalConfig = Configuration;
        }

        public IConfigurationRoot Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the
container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddMvc().AddViewLocalization().AddDataAnnotationsLocalization();

            // IoC Container
            // Add application services.
            services.AddTransient<EFStringLocalizerFactory>();
            services.AddSingleton<IConfiguration>(Configuration);
        }

        // This method gets called by the runtime. Use this method to configure the HTTP
request pipeline.
        public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory, EFStringLocalizerFactory localizerFactory)
        {
            _e = localizerFactory.Create(null);

            // a list of all available languages
            var supportedCultures = new List<CultureInfo>
            {
                new CultureInfo("en-US"),
                new CultureInfo("fa-IR")
            };

            var requestLocalizationOptions = new RequestLocalizationOptions
            {
                SupportedCultures = supportedCultures,
                SupportedUICultures = supportedCultures,
            };
            requestLocalizationOptions.RequestCultureProviders.Insert(0, new
JsonRequestCultureProvider());
            app.UseRequestLocalization(requestLocalizationOptions);

            app.UseStaticFiles();

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

    public class JsonRequestCultureProvider : RequestCultureProvider
    {
        public override Task<ProviderCultureResult> DetermineProviderCultureResult(HttpContext
httpContext)
        {
```

```
            if (httpContext == null)
            {
                throw new ArgumentNullException(nameof(httpContext));
            }

            var config = Startup.LocalConfig;

            string culture = config["AppOptions:Culture"];
            string uiCulture = config["AppOptions:UICulture"];
            string culturedirection = config["AppOptions:CultureDirection"];

            culture = culture ?? "fa-IR"; // Use the value defined in config files or the
default value
            uiCulture = uiCulture ?? culture;

            Startup.UiCulture = uiCulture;

            culturedirection = culturedirection ?? "rlt"; // rtl is set to be the default
value in case culturedirection is null
            Startup.CultureDirection = culturedirection;

            return Task.FromResult(new ProviderCultureResult(culture, uiCulture));
        }
    }
}
```

Nel codice precedente, per prima cosa aggiungiamo tre variabili di campo `public static` che verranno successivamente inizializzate utilizzando i valori letti dal file delle impostazioni.

Nel costruttore per la classe `Startup` aggiungiamo un file di impostazioni json alla variabile `builder` . Il primo file è necessario affinché l'app funzioni, quindi vai avanti e crea `appsettings.json` nella root del progetto, se non esiste già. Utilizzando Visual Studio 2015, questo file viene creato automaticamente, quindi basta cambiarne il contenuto in: (Puoi omettere la sezione `Logging` se non la usi)

appsettings.json

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "AppOptions": {
    "Culture": "en-US", // fa-IR for Persian
    "UICulture": "en-US", // same as above
    "CultureDirection": "ltr" // rtl for Persian/Arabic/Hebrew
  }
}
```

Andando avanti, crea tre cartelle nella root del tuo progetto:

`Models` , `Services` e `Languages` . Nella cartella `Models` crea un'altra cartella denominata `Localization` .

Nella cartella `Services` creiamo un nuovo file .cs denominato `EFLocalization` . Il contenuto sarebbe: (Anche in questo caso l' `using` dichiarazioni non è incluso)

EFLocalization.cs

```
namespace DigitalShop.Services
{
    public class EFStringLocalizerFactory : IStringLocalizerFactory
    {
        private readonly LocalizationDbContext _db;

        public EFStringLocalizerFactory()
        {
            _db = new LocalizationDbContext();
            // Here we define all available languages to the app
            // available languages are those that have a json and cs file in
            // the Languages folder
            _db.AddRange(
                new Culture
                {
                    Name = "en-US",
                    Resources = en_US.GetList()
                },
                new Culture
                {
                    Name = "fa-IR",
                    Resources = fa_IR.GetList()
                }
            );
            _db.SaveChanges();
        }

        public IStringLocalizer Create(Type resourceSource)
        {
            return new EFStringLocalizer(_db);
        }

        public IStringLocalizer Create(string baseName, string location)
        {
            return new EFStringLocalizer(_db);
        }
    }

    public class EFStringLocalizer : IStringLocalizer
    {
        private readonly LocalizationDbContext _db;

        public EFStringLocalizer(LocalizationDbContext db)
        {
            _db = db;
        }

        public LocalizedString this[string name]
        {
            get
            {
                var value = GetString(name);
                return new LocalizedString(name, value ?? name, resourceNotFound: value ==
null);
            }
```

```
        }

        public LocalizedString this[string name, params object[] arguments]
        {
            get
            {
                var format = GetString(name);
                var value = string.Format(format ?? name, arguments);
                return new LocalizedString(name, value, resourceNotFound: format == null);
            }
        }

        public IStringLocalizer WithCulture(CultureInfo culture)
        {
            CultureInfo.DefaultThreadCurrentCulture = culture;
            return new EFStringLocalizer(_db);
        }

        public IEnumerable<LocalizedString> GetAllStrings(bool includeAncestorCultures)
        {
            return _db.Resources
                .Include(r => r.Culture)
                .Where(r => r.Culture.Name == CultureInfo.CurrentCulture.Name)
                .Select(r => new LocalizedString(r.Key, r.Value, true));
        }

        private string GetString(string name)
        {
            return _db.Resources
                .Include(r => r.Culture)
                .Where(r => r.Culture.Name == CultureInfo.CurrentCulture.Name)
                .FirstOrDefault(r => r.Key == name)?.Value;
        }
    }

    public class EFStringLocalizer<T> : IStringLocalizer<T>
    {
        private readonly LocalizationDbContext _db;

        public EFStringLocalizer(LocalizationDbContext db)
        {
            _db = db;
        }

        public LocalizedString this[string name]
        {
            get
            {
                var value = GetString(name);
                return new LocalizedString(name, value ?? name, resourceNotFound: value ==
null);
            }
        }

        public LocalizedString this[string name, params object[] arguments]
        {
            get
            {
                var format = GetString(name);
                var value = string.Format(format ?? name, arguments);
                return new LocalizedString(name, value, resourceNotFound: format == null);
```

```
            }
        }

        public IStringLocalizer WithCulture(CultureInfo culture)
        {
            CultureInfo.DefaultThreadCurrentCulture = culture;
            return new EFStringLocalizer(_db);
        }

        public IEnumerable<LocalizedString> GetAllStrings(bool includeAncestorCultures)
        {
            return _db.Resources
                .Include(r => r.Culture)
                .Where(r => r.Culture.Name == CultureInfo.CurrentCulture.Name)
                .Select(r => new LocalizedString(r.Key, r.Value, true));
        }

        private string GetString(string name)
        {
            return _db.Resources
                .Include(r => r.Culture)
                .Where(r => r.Culture.Name == CultureInfo.CurrentCulture.Name)
                .FirstOrDefault(r => r.Key == name)?.Value;
        }
    }
}
```

Nel file sopra implementiamo l'interfaccia `IStringLocalizerFactory` da Entity Framework Core per creare un servizio di localizzazione personalizzato. La parte importante è il costruttore di `EFStringLocalizerFactory` cui creiamo un elenco di tutte le lingue disponibili e lo aggiungiamo al contesto del database. Ciascuno di questi file di lingua agisce come un database separato.

Ora aggiungi ognuno dei seguenti file alla cartella `Models/Localization` :

Culture.cs

```
namespace DigitalShop.Models.Localization
{
    public class Culture
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public virtual List<Resource> Resources { get; set; }
    }
}
```

Resource.cs

```
namespace DigitalShop.Models.Localization
{
    public class Resource
    {
        public int Id { get; set; }
        public string Key { get; set; }
        public string Value { get; set; }
        public virtual Culture Culture { get; set; }
    }
```

```
}
```

LocalizationDbContext.cs

```
namespace DigitalShop.Models.Localization
{
    public class LocalizationDbContext : DbContext
    {
        public DbSet<Culture> Cultures { get; set; }
        public DbSet<Resource> Resources { get; set; }

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseInMemoryDatabase();
        }
    }
}
```

I file di cui sopra sono solo modelli che verranno popolati con risorse linguistiche, culture e c'è anche un tipico `DBContext` utilizzato da EF Core.

L'ultima cosa di cui abbiamo bisogno per fare tutto questo è creare i file delle risorse linguistiche. I file JSON utilizzati per memorizzare una coppia chiave-valore per lingue diverse disponibili nella tua app.

In questo esempio la nostra app ha solo due lingue disponibili. Inglese e persiano. Per ognuna delle lingue abbiamo bisogno di due file. Un file JSON contenente coppie chiave-valore e un file `.cs` che contiene una classe con lo stesso nome del file JSON. Quella classe ha un metodo, `GetList` che deserializza il file JSON e lo restituisce. Questo metodo è chiamato nel costruttore di `EFStringLocalizerFactory` che abbiamo creato in precedenza.

Quindi, crea questi quattro file nella cartella `Languages` :

en-US.cs

```
namespace DigitalShop.Languages
{
    public static class en_US
    {
        public static List<Resource> GetList()
        {
            var jsonSerializerSettings = new JsonSerializerSettings();
            jsonSerializerSettings.MissingMemberHandling = MissingMemberHandling.Ignore;
            return
JsonConvert.DeserializeObject<List<Resource>>(File.ReadAllText("Languages/en-US.json"),
jsonSerializerSettings);
        }
    }
}
```

en-US.json

```
[
  {
```

```
    "Key": "Welcome",
    "Value": "Welcome"
  },
  {
    "Key": "Hello",
    "Value": "Hello"
  },
]
```

Fa IR.cs

```
public static class fa_IR
{
    public static List<Resource> GetList()
    {
        var jsonSerializerSettings = new JsonSerializerSettings();
        jsonSerializerSettings.MissingMemberHandling = MissingMemberHandling.Ignore;
        return JsonConvert.DeserializeObject<List<Resource>>(File.ReadAllText("Languages/fa-
IR.json", Encoding.UTF8), jsonSerializerSettings);
    }
}
```

Fa-IR.json

```
[
  {
    "Key": "Welcome",
    "Value": "خوش آمدی"
 },
 {
  "Key": "Hello",
    "Value": "سلام"
 },
]
```

Abbiamo finito. Ora per accedere alle stringhe della lingua (coppie chiave-valore) in qualsiasi punto del codice (`.cs` o `.cshtml`) puoi fare quanto segue:

in un file `.cs` (essere Controller o meno, non importa):

```
// Returns "Welcome" for en-US and "خوش آمدی" for fa-IR
var welcome = Startup._e["Welcome"];
```

in un file di visualizzazione Razor (`.cshtml`):

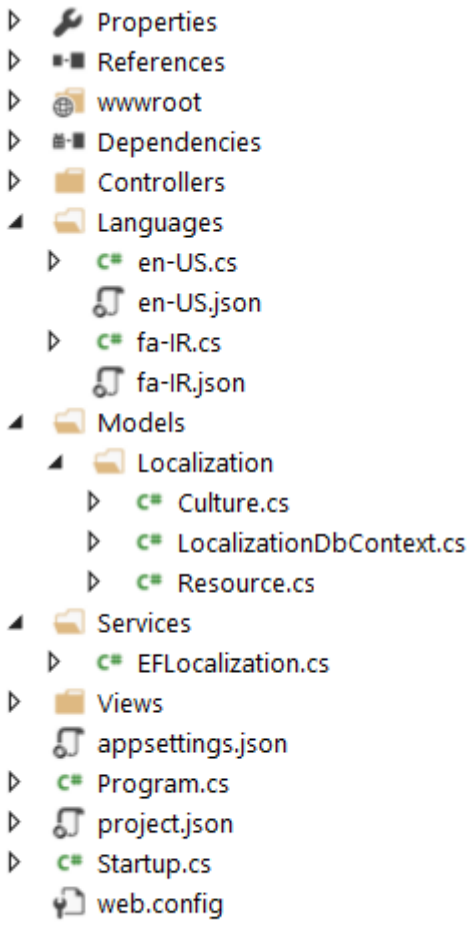
```
<h1>@Startup._e["Welcome"]</h1>
```

Poche cose da tenere a mente:

- Se si tenta di accedere a una `Key` che non esiste nel file JSON o caricata, si otterrà la chiave letterale (nell'esempio sopra, tentando di accedere a `Startup._e["How are you"]` restituirà `How are you` non importa le impostazioni della lingua perché non esiste
- Se si modifica un valore stringa in una lingua `.json` file, sarà necessario **riavviare**

l'applicazione. Altrimenti mostrerà solo il valore predefinito (nome della chiave). **Questo è particolarmente importante quando esegui la tua app senza eseguire il debug.**
- Il `appsettings.json` può essere utilizzato per memorizzare tutti i tipi di impostazioni di cui l'app potrebbe aver bisogno
- Il riavvio dell'app **non** è **necessario** se si desidera semplicemente modificare le **impostazioni di lingua / cultura dal file** **`appsettings.json`** . Ciò significa che è possibile avere un'opzione nell'interfaccia delle app per consentire agli utenti di cambiare lingua / cultura in fase di runtime.

Ecco la struttura finale del progetto:



Imposta la cultura della richiesta tramite il percorso dell'URL

Per impostazione predefinita, il middleware di richiesta localizzazione integrato supporta solo l'impostazione della cultura tramite query, cookie o intestazione `Accept-Language` . Questo esempio mostra come creare un middleware che consente di impostare la cultura come parte del percorso come in `/api/en-US/products` .

Questo middleware di esempio presuppone che le impostazioni internazionali si trovino nel secondo segmento del percorso.

```
public class UrlRequestCultureProvider : RequestCultureProvider
{
```

```
    private static readonly Regex LocalePattern = new Regex(@"^[a-z]{2}(-[a-z]{2,4})?$",
                                                            RegexOptions.IgnoreCase);

    public override Task<ProviderCultureResult> DetermineProviderCultureResult(HttpContext
httpContext)
    {
        if (httpContext == null)
        {
            throw new ArgumentNullException(nameof(httpContext));
        }

        var url = httpContext.Request.Path;

        // Right now it's not possible to use httpContext.GetRouteData()
        // since it uses IRoutingFeature placed in httpContext.Features when
        // Routing Middleware registers. It's not set when the Localization Middleware
        // is called, so this example simply assumes the locale will always
        // be located in the second segment of a path, like in /api/en-US/products
        var parts = httpContext.Request.Path.Value.Split('/');
        if (parts.Length < 3)
        {
            return Task.FromResult<ProviderCultureResult>(null);
        }

        if (!LocalePattern.IsMatch(parts[2]))
        {
            return Task.FromResult<ProviderCultureResult>(null);
        }

        var culture = parts[2];
        return Task.FromResult(new ProviderCultureResult(culture));
    }
}
```

Registrazione del middleware

```
var localizationOptions = new RequestLocalizationOptions
{
    SupportedCultures = new List<CultureInfo>
    {
        new CultureInfo("de-DE"),
        new CultureInfo("en-US"),
        new CultureInfo("en-GB")
    },
    SupportedUICultures = new List<CultureInfo>
    {
        new CultureInfo("de-DE"),
        new CultureInfo("en-US"),
        new CultureInfo("en-GB")
    },
    DefaultRequestCulture = new RequestCulture("en-US")
};

// Adding our UrlRequestCultureProvider as first object in the list
localizationOptions.RequestCultureProviders.Insert(0, new UrlRequestCultureProvider
{
    Options = localizationOptions
});
```

```
app.UseRequestLocalization(localizationOptions);
```

Vincoli di percorso personalizzati

L'aggiunta e la creazione di vincoli di instradamento personalizzati sono mostrati nell'esempio Vincoli percorso . L'uso dei vincoli semplifica l'utilizzo dei vincoli del percorso personalizzato.

Registrazione del percorso

Esempio di registrazione dei percorsi senza l'utilizzo di vincoli personalizzati

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "api/{culture::regex(^[a-z]{{2}}-[A-Za-z]{{4}}$)}}/{controller}/{id?}");
    routes.MapRoute(
        name: "default",
        template: "api/{controller}/{id?}");
});
```

Leggi Localizzazione online: https://riptutorial.com/it/asp-net-core/topic/2869/localizzazione

Capitolo 16: middleware

Osservazioni

Il middleware è un componente software che determinerà come elaborare la richiesta e decidere se passarlo al componente successivo nella pipeline dell'applicazione. Ogni middleware ha un ruolo e azioni specifici diversi da preformare nella richiesta.

Examples

Utilizzo del middleware ExceptionHandler per inviare al client un errore JSON personalizzato

Definisci la tua classe che rappresenterà il tuo errore personalizzato.

```
public class ErrorDto
{
   public int Code { get; set; }
   public string Message { get; set; }

   // other fields

   public override string ToString()
   {
      return JsonConvert.SerializeObject(this);
   }
}
```

Quindi inserire il prossimo middleware ExceptionHandler nel metodo Configure. Fai attenzione che l'ordine del middleware è importante.

```
app.UseExceptionHandler(errorApp =>
    {
        errorApp.Run(async context =>
        {
            context.Response.StatusCode = 500; // or another Status
            context.Response.ContentType = "application/json";

            var error = context.Features.Get<IExceptionHandlerFeature>();
            if (error != null)
            {
                var ex = error.Error;

                await context.Response.WriteAsync(new ErrorDto()
                {
                    Code = <your custom code based on Exception Type>,
                    Message = ex.Message // or your custom message

                    ... // other custom data
                }.ToString(), Encoding.UTF8);
            }
        });
```

```
    });
```

Middleware per impostare la risposta ContentType

L'idea è di usare `HttpContext.Response.OnStarting` callback, poiché questo è l'ultimo evento che viene generato prima che le intestazioni vengano inviate. Aggiungi quanto segue al tuo metodo di `Invoke` middleware.

```
public async Task Invoke(HttpContext context)
{
    context.Response.OnStarting((state) =>
    {
        if (context.Response.StatusCode == (int)HttpStatusCode.OK)
        {
           if (context.Request.Path.Value.EndsWith(".map"))
           {
             context.Response.ContentType = "application/json";
           }
        }
        return Task.FromResult(0);
    }, null);

    await nextMiddleware.Invoke(context);
}
```

Passa i dati attraverso la catena del middleware

Dalla documentazione :

> La raccolta ***HttpContext.Items*** è la posizione migliore in cui archiviare i dati necessari solo durante l'elaborazione di una determinata richiesta. I suoi contenuti vengono scartati dopo ogni richiesta. Si utilizza al meglio come mezzo di comunicazione tra componenti o middleware che operano in momenti diversi durante una richiesta e non hanno alcuna relazione diretta tra loro per il passaggio di parametri o valori di ritorno.

`HttpContext.Items` è una semplice raccolta di dizionari di tipo `IDictionary<object, object>` . Questa collezione è

- disponibile dall'inizio di `HttpRequest`
- e viene scartato alla fine di ogni richiesta.

È possibile accedervi semplicemente assegnando un valore a una voce con chiave o richiedendo il valore per una determinata chiave.

Ad esempio, alcuni semplici middleware potrebbero aggiungere qualcosa alla raccolta Articoli:

```
app.Use(async (context, next) =>
{
    // perform some verification
    context.Items["isVerified"] = true;
    await next.Invoke();
});
```

e più avanti nella pipeline, un altro pezzo di middleware potrebbe accedervi:

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Verified request? " + context.Items["isVerified"]);
});
```

Esegui, Mappa, Usa

Correre

Termina la catena. Nessun altro metodo middleware verrà eseguito dopo questo. Dovrebbe essere posizionato alla fine di qualsiasi oleodotto.

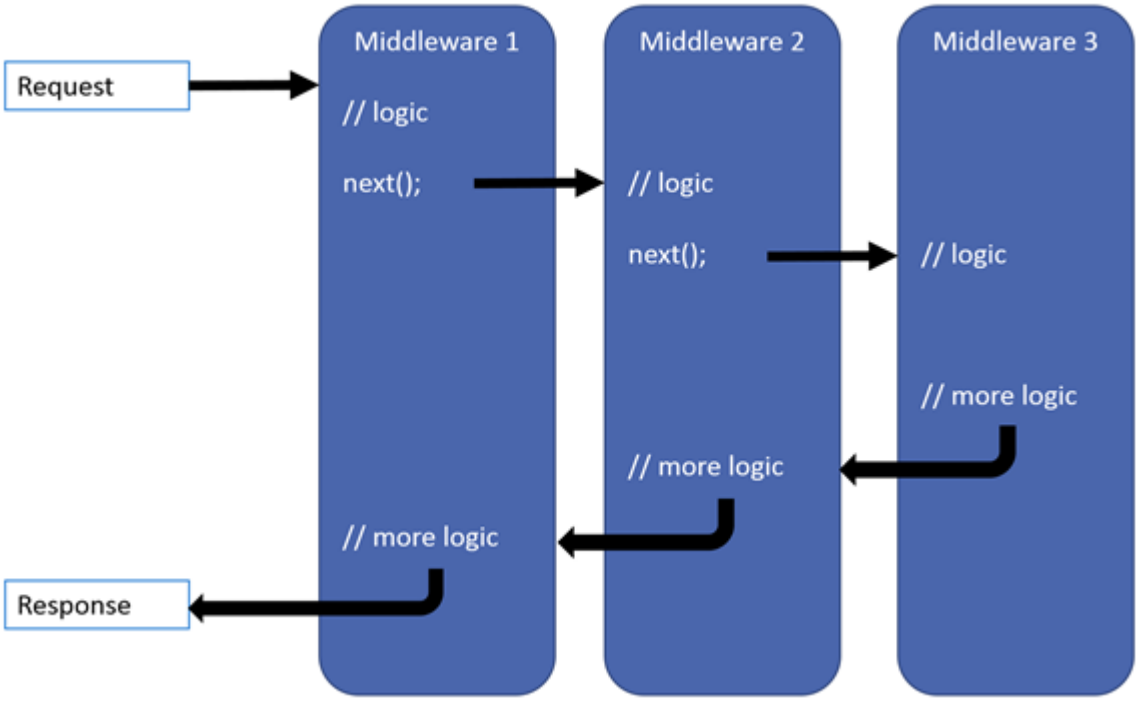
```
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from " + _environment);
});
```

Uso

Esegue un'azione prima e dopo il prossimo delegato.

```
app.Use(async (context, next) =>
{
    //action before next delegate
    await next.Invoke(); //call next middleware
    //action after called middleware
});
```

Ilustration di come funziona:



MapWhen

Abilita la pipeline di ramificazione. Esegue il middleware specificato se la condizione è soddisfatta.

```
private static void HandleBranch(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Condition is fulfilled");
    });
}

public void ConfigureMapWhen(IApplicationBuilder app)
{
    app.MapWhen(context => {
        return context.Request.Query.ContainsKey("somekey");
    }, HandleBranch);
}
```

Carta geografica

Simile a MapWhen. Esegue il middleware se il percorso richiesto dall'utente equivale al percorso fornito nel parametro.

```
private static void HandleMapTest(IApplicationBuilder app)
{
    app.Run(async context =>
    {
        await context.Response.WriteAsync("Map Test Successful");
    });
}

public void ConfigureMapping(IApplicationBuilder app)
{
    app.Map("/maptest", HandleMapTest);

}
```

Basato su ASP.net Core Docs

Leggi middleware online: https://riptutorial.com/it/asp-net-core/topic/1479/middleware

Capitolo 17: Modelli

Examples

Validazione del modello con Validation Attrributes

Gli attributi di convalida possono essere utilizzati per configurare facilmente la convalida del modello.

```
public class MyModel
{
    public int id { get; set; }

    //sets the FirstName to be required, and no longer than 100 characters
    [Required]
    [StringLength(100)]
    public string FirstName { get; set; }
}
```

Gli attributi incorporati sono:

- `[CreditCard]` : convalida che la proprietà abbia un formato di carta di credito.
- `[Compare]` : convalida due proprietà in una corrispondenza del modello.
- `[EmailAddress]` : convalida che la proprietà ha un formato email.
- `[Phone]` : convalida che la proprietà ha un formato telefonico.
- `[Range]` : convalida il valore della proprietà rientra nell'intervallo specificato.
- `[RegularExpression]` : convalida che i dati corrispondano all'espressione regolare specificata.
- `[Required]` : rende necessaria una proprietà.
- `[StringLength]` : convalida che una proprietà stringa abbia al massimo la lunghezza massima specificata.
- `[Url]` : convalida che la proprietà ha un formato URL.

Convalida del modello con attributo personalizzato

Se gli attributi incorporati non sono sufficienti per convalidare i dati del modello, è possibile inserire la logica di convalida in una classe derivata da ValidationAttribute. In questo esempio, solo i numeri dispari sono valori validi per un membro del modello.

Attributo di convalida personalizzato

```
public class OddNumberAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
    {
        try
        {
            var number = (int) value;
            if (number % 2 == 1)
```

```
                return ValidationResult.Success;
            else
                return new ValidationResult("Only odd numbers are valid.");
        }
        catch (Exception)
        {
            return new ValidationResult("Not a number.");
        }
    }
}
```

Classe di modello

```
public class MyModel
{
    [OddNumber]
    public int Number { get; set; }
}
```

Leggi Modelli online: https://riptutorial.com/it/asp-net-core/topic/4625/modelli

Capitolo 18: project.json

introduzione

Project json è una struttura di file di configurazione del progetto, utilizzata temporaneamente dai progetti asp.net-core, prima che Microsoft tornasse ai file csproj a favore di msbuild.

Examples

Esempio di progetto Libreria semplice

Una libreria basata su NETStandard 1.6 sarebbe simile a questa:

```
{
  "version": "1.0.0",
  "dependencies": {
    "NETStandard.Library": "1.6.1", //nuget dependency
   },
  "frameworks": { //frameworks the library is build for
    "netstandard1.6": {}
  },
  "buildOptions": {
    "debugType": "portable"
  }
}
```

File json completo:

Tratto dalla pagina github di Microsoft con la documentazione ufficiale

```
{
"name": String, //The name of the project, used for the assembly name as well as the name of
the package. The top level folder name is used if this property is not specified.
"version": String, //The Semver version of the project, also used for the NuGet package.
"description": String, //A longer description of the project. Used in the assembly properties.
"copyright": String, //The copyright information for the project. Used in the assembly
properties.
"title": String, //The friendly name of the project, can contain spaces and special characters
not allowed when using the `name` property. Used in the assembly properties.
"entryPoint": String, //The entrypoint method for the project. `Main` by default.
"testRunner": String, //The name of the test runner, such as NUnit or xUnit, to use with this
project. Setting this also marks the project as a test project.
"authors": String[], // An array of strings with the names of the authors of the project.
"language": String, //The (human) language of the project. Corresponds to the "neutral-
language" compiler argument.
"embedInteropTypes": Boolean, //`true` to embed COM interop types in the assembly; otherwise,
`false`.
"preprocess": String or String[], //Specifies which files are included in preprocessing.
"shared": String or String[], //Specifies which files are shared, this is used for library
export.
"dependencies": Object { //project and nuget dependencies
    version: String, //Specifies the version or version range of the dependency. Use the \*
```

```
wildcard to specify a floating dependency version.
    type: String, //type of dependency: build
    target: String, //Restricts the dependency to match only a `project` or a `package`.
    include: String,
    exclude: String,
    suppressParent: String
},
"tools": Object, //An object that defines package dependencies that are used as tools for the
current project, not as references. Packages defined here are available in scripts that run
during the build process, but they are not accessible to the code in the project itself. Tools
can for example include code generators or post-build tools that perform tasks related to
packing.
"scripts": Object, // commandline scripts: precompile, postcompile, prepublish & postpublish
"buildOptions": Object {
    "define": String[], //A list of defines such as "DEBUG" or "TRACE" that can be used in
conditional compilation in the code.
    "nowarn": String[], //A list of warnings to ignore.
    "additionalArguments": String[], //A list of extra arguments that will be passed to the
compiler.
    "warningsAsErrors": Boolean,
    "allowUnsafe": Boolean,
    "emitEntryPoint": Boolean,
    "optimize": Boolean,
    "platform": String,
    "languageVersion": String,
    "keyFile": String,
    "delaySign": Boolean,
    "publicSign": Boolean,
    "debugType": String,
    "xmlDoc": Boolean,
    "preserveCompilationContext": Boolean,
    "outputName": String,
    "compilerName": String,
    "compile": Object {
        "include": String or String[],
        "exclude": String or String[],
        "includeFiles": String or String[],
        "excludeFiles": String or String[],
        "builtIns": Object,
        "mappings": Object
    },
    "embed": Object {
        "include": String or String[],
        "exclude": String or String[],
        "includeFiles": String or String[],
        "excludeFiles": String or String[],
        "builtIns": Object,
        "mappings": Object
    },
    "copyToOutput": Object {
        "include": String or String[],
        "exclude": String or String[],
        "includeFiles": String or String[],
        "excludeFiles": String or String[],
        "builtIns": Object,
        "mappings": Object
    }
},
"publishOptions": Object {
    "include": String or String[],
    "exclude": String or String[],
```

```
    "includeFiles": String or String[],
    "excludeFiles": String or String[],
    "builtIns": Object,
    "mappings": Object
},
"runtimeOptions": Object {
    "configProperties": Object {
        "System.GC.Server": Boolean,
        "System.GC.Concurrent": Boolean,
        "System.GC.RetainVM": Boolean,
        "System.Threading.ThreadPool.MinThreads": Integer,
        "System.Threading.ThreadPool.MaxThreads": Integer
    },
    "framework": Object {
        "name": String,
        "version": String,
    },
    "applyPatches": Boolean
},
"packOptions": Object {
    "summary": String,
    "tags": String[],
    "owners": String[],
    "releaseNotes": String,
    "iconUrl": String,
    "projectUrl": String,
    "licenseUrl": String,
    "requireLicenseAcceptance": Boolean,
    "repository": Object {
        "type": String,
        "url": String
    },
    "files": Object {
        "include": String or String[],
        "exclude": String or String[],
        "includeFiles": String or String[],
        "excludeFiles": String or String[],
        "builtIns": Object,
        "mappings": Object
    }
},
"analyzerOptions": Object {
    "languageId": String
},
"configurations": Object,
"frameworks": Object {
    "dependencies": Object {
        version: String,
        type: String,
        target: String,
        include: String,
        exclude: String,
        suppressParent: String
    },
    "frameworkAssemblies": Object,
    "wrappedProject": String,
    "bin": Object {
        assembly: String
    }
},
"runtimes": Object,
```

```
"userSecretsId": String
}
```

Semplice progetto di avvio

Un semplice esempio di configurazione del progetto per un'app Console .NetCore 1.1

```
{
  "version": "1.0.0",
  "buildOptions": {
    "emitEntryPoint": true // make sure entry point is emitted.
  },
  "dependencies": {
  },
  "tools": {
  },
  "frameworks": {
    "netcoreapp1.1": { // run as console app
      "dependencies": {
        "Microsoft.NETCore.App": {
          "type": "platform",
          "version": "1.1.0"
        }
      },
      "imports": "dnxcore50"
    }
  },
}
```

Leggi project.json online: https://riptutorial.com/it/asp-net-core/topic/9364/project-json

Capitolo 19: Pubblicazione e distribuzione

Examples

Gheppio. Configurazione dell'indirizzo di ascolto

Usando Kestrel puoi specificare la porta usando i seguenti approcci:

1. Definizione della variabile di ambiente `ASPNETCORE_URLS` .

   finestre

   ```
   SET ASPNETCORE_URLS=https://0.0.0.0:5001
   ```

   OS X

   ```
   export ASPNETCORE_URLS=https://0.0.0.0:5001
   ```

2. Tramite la riga di comando passando il parametro `--server.urls`

   ```
   dotnet run --server.urls=http://0.0.0.0:5001
   ```

3. Utilizzando il metodo `UseUrls()`

   ```
   var builder = new WebHostBuilder()
                 .UseKestrel()
                 .UseUrls("http://0.0.0.0:5001")
   ```

4. Definizione dell'impostazione di `server.urls` di configurazione.

Il prossimo esempio usa il file hosting.json per esempio.

```
Add `hosting.json` with the following content to you project:

    {
       "server.urls": "http://<ip address>:<port>"
    }
```

Esempi di valori possibili:

- ascolta 5000 su qualsiasi indirizzo IP4 e IP6 da qualsiasi interfaccia:

  ```
  "server.urls": "http://*:5000"
  ```

  o

  ```
  "server.urls": "http://::5000;http://0.0.0.0:5000"
  ```

- ascolta 5000 su ogni indirizzo IP4:

  ```
  "server.urls": "http://0.0.0.0:5000"
  ```

  Uno dovrebbe essere attentamente e non utilizzare `http://*:5000;http://::5000` , `http://::5000;http://*:5000` , `http://*:5000;http://0.0.0.0:5000` o `http://*:5000;http://0.0.0.0:5000` perché richiederà la registrazione dell'indirizzo IP6 :: o dell'indirizzo IP4 0.0.0.0 due volte

Aggiungi file a `publishOptions` in `project.json`

```
"publishOptions": {
"include": [
    "hosting.json",
    ...
  ]
}
```

e nel punto di ingresso per la chiamata dell'applicazione `.UseConfiguration(config)` durante la creazione di WebHostBuilder:

```
public static void Main(string[] args)
{
    var config = new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("hosting.json", optional: true)
        .Build();

    var host = new WebHostBuilder()
        .UseConfiguration(config)
        .UseKestrel()
        .UseContentRoot(Directory.GetCurrentDirectory())
        .UseIISIntegration()
        .UseStartup<Startup>()
        .Build();

    host.Run();
}
```

Leggi Pubblicazione e distribuzione online: https://riptutorial.com/it/asp-net-core/topic/2262/pubblicazione-e-distribuzione

Capitolo 20: Registrazione

Examples

Utilizzando NLog Logger

NLog.Extensions.Logging è il provider NLog ufficiale per Microsoft in. NET Core e ASP.NET Core. Qui e qui sono rispettivamente le istruzioni e l'esempio.

Aggiungi logger al controller

Invece di richiedere un ILoggerFactory e creare un'istanza di ILogger in modo esplicito, è possibile richiedere un ILogger (dove T è la classe che richiede il logger).

```
public class TodoController : Controller
{
    private readonly ILogger _logger;

    public TodoController(ILogger<TodoController> logger)
    {
     _logger = logger;
    }
}
```

Utilizzo di Serilog nell'applicazione core 1.0 di ASP.NET

1) In project.json, aggiungi dipendenze in basso-

```
"Serilog": "2.2.0",
"Serilog.Extensions.Logging": "1.2.0",
"Serilog.Sinks.RollingFile": "2.0.0",
"Serilog.Sinks.File": "3.0.0"
```

2) In Startup.cs, aggiungi le linee sottostanti in costruttore

```
Log.Logger = new LoggerConfiguration()
    .MinimumLevel.Debug()
    .WriteTo.RollingFile(Path.Combine(env.ContentRootPath, "Serilog-{Date}.txt"))
    .CreateLogger();
```

3) In Configure method of Startup class-

```
loggerFactory.AddSerilog();
```

4) In Controller, crea un'istanza di ILogger come questa-

```
public class HomeController : Controller
{
```

```
    ILogger<HomeController> _logger = null;
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }
```

5) Registrazione dei campioni sotto-

```
try
{
    throw new Exception("Serilog Testing");
}
catch (System.Exception ex)
{
    this._logger.LogError(ex.Message);
}
```

Leggi Registrazione online: https://riptutorial.com/it/asp-net-core/topic/1946/registrazione

Capitolo 21: Richieste di origine incrociata (CORS)

Osservazioni

La sicurezza del browser impedisce a una pagina web di inoltrare richieste AJAX a un altro dominio. Questa restrizione è denominata criterio di origine identica e impedisce a un sito dannoso di leggere i dati riservati da un altro sito. Tuttavia, a volte potresti voler consentire ad altri siti di effettuare richieste di origine incrociata alla tua app web.

Cross Origin Resource Sharing (CORS) è uno standard W3C che consente a un server di attenuare la politica dell'origine stessa. Utilizzando CORS, un server può consentire esplicitamente alcune richieste di origine incrociata mentre respinge le altre. CORS è più sicuro e più flessibile rispetto alle tecniche precedenti come JSONP.

Examples

Abilita CORS per tutte le richieste

Utilizzare il metodo di estensione `UseCors()` su `IApplicationBuilder` nel metodo `Configure` per applicare la politica CORS a tutte le richieste.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();
    services.AddCors();
}

public void Configure(IApplicationBuilder app)
{
    // Other middleware..

    app.UseCors(builder =>
    {
        builder.AllowAnyOrigin()
               .AllowAnyHeader()
               .AllowAnyMethod();
    });

    // Other middleware..

    app.UseMvc();
}
```

Abilita la politica CORS per controller specifici

Per abilitare una determinata politica CORS per controller specifici, è necessario creare la politica nell'estensione `AddCors` all'interno del metodo `ConfigureServices` :

```
services.AddCors(cors => cors.AddPolicy("AllowAll", policy =>
{
    policy.AllowAnyOrigin()
            .AllowAnyMethod()
            .AllowAnyHeader();
}));
```

Ciò consente di applicare la politica a un controller:

```
[EnableCors("AllowAll")]
public class HomeController : Controller
{
   // ...
}
```

Politiche CORS più sofisticate

Il builder di criteri consente di creare policy sofisticate.

```
app.UseCors(builder =>
{
    builder.WithOrigins("http://localhost:5000", "http://myproductionapp.com")
           .WithMethods("GET", "POST", "HEAD")
           .WithHeaders("accept", "content-type", "origin")
           .SetPreflightMaxAge(TimeSpan.FromDays(7));
});
```

Questo criterio consente solo le origini `http://localhost:5000` e `http://myproductionapp.com` con solo i metodi `GET` , `POST` e `HEAD` e accetta solo le intestazioni HTTP di `content-type accept` , `content-type` e di `origin` . Il metodo `SetPreflightMaxAge` fa in modo che i browser memorizzino nella cache il risultato della richiesta di preflight (`OPTIONS`) da memorizzare nella cache per il periodo di tempo specificato.

Abilita la politica CORS per tutti i controller

Per abilitare un criterio CORS su tutti i controller MVC, è necessario creare il criterio nell'estensione AddCors all'interno del metodo ConfigureServices e quindi impostare il criterio su `CorsAuthorizationFilterFactory`

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Cors.Internal;
...
public void ConfigureServices(IServiceCollection services) {
    // Add AllowAll policy just like in single controller example.
    services.AddCors(options => {
        options.AddPolicy("AllowAll",
           builder => {
                builder.AllowAnyOrigin()
                       .AllowAnyMethod()
                       .AllowAnyHeader();
        });
    });

    // Add framework services.
```

```
    services.AddMvc();

    services.Configure<MvcOptions>(options => {
        options.Filters.Add(new CorsAuthorizationFilterFactory("AllowAll"));
    });
}

public void Configure(IApplicationBuilder app) {
    app.useMvc();
    // For content not managed within MVC. You may want to set the Cors middleware
    // to use the same policy.
    app.UseCors("AllowAll");
}
```

Questo criterio CORS può essere sovrascritto su un controller o su una base azione, ma può impostare l'impostazione predefinita per l'intera applicazione.

Leggi Richieste di origine incrociata (CORS) online: https://riptutorial.com/it/asp-net-core/topic/2556/richieste-di-origine-incrociata--cors-

Capitolo 22: Routing

Examples

Routing di base

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

Questo corrisponderà alle richieste di `/Home/Index` , `/Home/Index/123` e `/`

Vincoli di instradamento

È possibile creare un vincolo di routing personalizzato che può essere utilizzato all'interno dei percorsi per vincolare un parametro a valori o pattern specifici.

Questo vincolo corrisponderà a un tipico modello di cultura / locale, come en-US, de-DE, zh-CHT, zh-Hant.

```
public class LocaleConstraint : IRouteConstraint
{
    private static readonly Regex LocalePattern = new Regex(@"^[a-z]{2}(-[a-z]{2,4})?$",
                                    RegexOptions.Compiled | RegexOptions.IgnoreCase);

    public bool Match(HttpContext httpContext, IRouter route, string routeKey,
                        RouteValueDictionary values, RouteDirection routeDirection)
    {
        if (!values.ContainsKey(routeKey))
            return false;

        string locale = values[routeKey] as string;
        if (string.IsNullOrWhiteSpace(locale))
            return false;

        return LocalePattern.IsMatch(locale);
    }
}
```

Successivamente, il vincolo deve essere registrato prima di poter essere utilizzato nei percorsi.

```
services.Configure<RouteOptions>(options =>
{
    options.ConstraintMap.Add("locale", typeof(LocaleConstraint));
});
```

Ora può essere utilizzato all'interno dei percorsi.

Usandolo sui controller

```
[Route("api/{culture:locale}/[controller]")]
public class ProductController : Controller { }
```

Usandolo su Azioni

```
[HttpGet("api/{culture:locale}/[controller]/{productId}")]
public Task<IActionResult> GetProductAsync(string productId) { }
```

Usandolo in percorsi predefiniti

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "api/{culture:locale}/{controller}/{id?}");
    routes.MapRoute(
        name: "default",
        template: "api/{controller}/{id?}");
});
```

Leggi Routing online: https://riptutorial.com/it/asp-net-core/topic/2863/routing

Capitolo 23: Sessioni in ASP.NET Core 1.0

introduzione

Utilizzo delle sessioni in ASP.NET Core 1.0

Examples

Esempio base di gestione della sessione

1) Innanzitutto, aggiungi dipendenza in project.json - `"Microsoft.AspNetCore.Session": "1.1.0",`

2) In `startup.cs` e aggiungi le `AddSession()` e `AddDistributedMemoryCache()` ai `ConfigureServices` come questo-

```
services.AddDistributedMemoryCache(); //This way ASP.NET Core will use a Memory Cache to store
session variables
services.AddSession(options =>
        {
            options.IdleTimeout = TimeSpan.FromDays(1); // It depends on user requirements.
            options.CookieName = ".My.Session"; // Give a cookie name for session which will
be visible in request payloads.
        });
```

3) Aggiungi la chiamata `UseSession()` in Configure method of startup come this-

```
app.UseSession(); //make sure add this line before UseMvc()
```

4) In Controller, l'oggetto Session può essere usato in questo modo-

```
using Microsoft.AspNetCore.Http;

public class HomeController : Controller
{
    public IActionResult Index()
    {
        HttpContext.Session.SetString("SessionVariable1", "Testing123");
        return View();
    }

    public IActionResult About()
    {
        ViewBag.Message = HttpContext.Session.GetString("SessionVariable1");

        return View();
    }
}
```

5. Se si utilizza la politica cors, a volte può dare errori, dopo l'abilitazione sessione relativa alle intestazioni sull'abilitazione dell'intestazione di *AllowCredentials* e

sull'utilizzo
L' intestazione *WithOrigins* invece di *AllowAllOrigins* .

Leggi Sessioni in ASP.NET Core 1.0 online: https://riptutorial.com/it/asp-net-core/topic/8067/sessioni-in-asp-net-core-1-0

Capitolo 24: Tag Helpers

Parametri

Nome	Informazioni
asp-action	Il nome del metodo di azione a cui il modulo deve essere pubblicato
asp-controllore	Il nome del controller in cui esiste il metodo di azione specificato in asp-action
asp-Route-*	Valori di instradamento personalizzati che si desidera aggiungere come querystring al valore dell'attributo action modulo. Sostituisci 8 con il nome querystring desiderato

Examples

Form Tag Helper - Esempio di base

```
<form asp-action="create" asp-controller="Home">
     <!--Your form elements goes here-->
</form>
```

Form Tag Helper - Con attributi di percorso personalizzati

```
<form asp-action="create"
      asp-controller="Home"
      asp-route-returnurl="dashboard"
      asp-route-from="google">
     <!--Your form elements goes here-->
</form>
```

Questo genererà il markup sottostante

```
<form action="/Home/create?returnurl=dashboard&amp;from=google" method="post">
    <!--Your form elements goes here-->
</form>
```

Input Tag Helper

Supponendo che la tua vista sia fortemente battuta su un modello di visualizzazione come

```
public class CreateProduct
{
   public string Name { set; get; }
}
```

E stai passando un oggetto di questo alla vista dal tuo metodo di azione.

```
@model CreateProduct
<form asp-action="create" asp-controller="Home" >

    <input type="text" asp-for="Name"/>
    <input type="submit"/>

</form>
```

Questo genererà il markup sottostante.

```
<form action="/Home/create" method="post">

    <input type="text" id="Name" name="Name" value="" />
    <input type="submit"/>
    <input name="__RequestVerificationToken" type="hidden" value="ThisWillBeAUniqueToken" />

</form>
```

Se si desidera eseguire il rendering del campo di input con un valore predefinito, è possibile impostare il valore della proprietà Nome del modello di vista nel metodo di azione.

```
public IActionResult Create()
{
  var vm = new CreateProduct { Name="IPhone"};
  return View(vm);
}
```

Invio del modulo e associazione del modello

Il collegamento del modello funzionerà correttamente se si utilizza `CreateProduct` come parametro del metodo di azione HttpPost / un parametro denominato `name`

Seleziona Tag Helper

Supponendo che la tua vista sia fortemente digitata su un modello di visualizzazione come questo

```
public class CreateProduct
{
   public IEnumerable<SelectListItem> Categories { set; get; }
   public int SelectedCategory { set; get; }
}
```

E nel tuo metodo di azione GET, stai creando un oggetto di questo modello di vista, impostando la proprietà Categorie e inviando alla vista

```
public IActionResult Create()
{
    var vm = new CreateProduct();
    vm.Categories = new List<SelectListItem>
    {
        new SelectListItem {Text = "Books", Value = "1"},
```

```
        new SelectListItem {Text = "Furniture", Value = "2"}
    };
    return View(vm);
}
```

e secondo te

```
@model CreateProduct
```

```
<form asp-action="create" asp-controller="Home">
    <select asp-for="SelectedCategory" asp-items="@Model.Categories">
        <option>Select one</option>
    </select>
    <input type="submit"/>
</form>
```

Questo renderà il markup sottostante (*incluse solo le parti rilevanti del modulo / campi*)

```
<form action="/Home/create" method="post">
    <select data-val="true" id="SelectedCategory" name="SelectedCategory">
        <option>Select one</option>
        <option value="1">Shyju</option>
        <option value="2">Sean</option>
    </select>
    <input type="submit"/>
</form>
```

Ottenere il valore di dropdown selezionato nell'invio del modulo

È possibile utilizzare lo stesso modello di visualizzazione del parametro del metodo di azione HttpPost

```
[HttpPost]
public ActionResult Create(CreateProduct model)
{
  //check model.SelectedCategory value
  / /to do : return something
}
```

Imposta un'opzione come quella selezionata

Se si desidera impostare un'opzione come opzione selezionata, è possibile semplicemente impostare il valore della proprietà `SelectedCategory` .

```
public IActionResult Create()
{
    var vm = new CreateProduct();
    vm.Categories = new List<SelectListItem>
    {
        new SelectListItem {Text = "Books", Value = "1"},
        new SelectListItem {Text = "Furniture", Value = "2"},
        new SelectListItem {Text = "Music", Value = "3"}
    };
    vm.SelectedCategory = 2;
```

```
    return View(vm);
}
```

Rendering di un menu a discesa / selezione multi selezione

Se vuoi eseguire il rendering di un menu a discesa a selezione multipla, puoi semplicemente modificare la proprietà del modello di vista che utilizzi `asp-for` attributo `asp-for` nella tua vista su un tipo di matrice.

```
public class CreateProduct
{
   public IEnumerable<SelectListItem> Categories { set; get; }
   public int[] SelectedCategories { set; get; }
}
```

Nella vista

```
@model CreateProduct
```

```
<form asp-action="create" asp-controller="Home" >
    <select asp-for="SelectedCategories" asp-items="@Model.Categories">
        <option>Select one</option>
    </select>
    <input type="submit"/>
</form>
```

Questo genererà l'elemento SELECT con `multiple` attributi

```
<form action="/Home/create" method="post">
     <select id="SelectedCategories" multiple="multiple" name="SelectedCategories">
        <option>Select one</option>
        <option value="1">Shyju</option>
       <option value="2">Sean</option>
     </select>
    <input type="submit"/>
</form>
```

Helper tag personalizzato

È possibile creare i propri helper di tag implementando `ITagHelper` o derivando dalla classe di convenienza `TagHelper` .

- La convenzione di default è di indirizzare un tag html che corrisponda al nome dell'helper senza il suffisso TagHelper opzionale. Ad esempio `WidgetTagHelper` avrà come target un tag `<widget>` .
- L'attributo `[HtmlTargetElement]` può essere utilizzato per controllare ulteriormente il tag che viene scelto come target
- A qualsiasi proprietà pubblica della classe può essere assegnato un valore come attributo nel markup del rasoio. Ad esempio una `public string Title {get; set;}` pubblica di proprietà `public string Title {get; set;}` può avere un valore come `<widget title="my title">`
- Per impostazione predefinita, tag helper traduce i nomi di classe C # Pascal e le proprietà

per gli helper dei tag in caso di kebab inferiore. Ad esempio, se ometti di utilizzare `[HtmlTargetElement]` e il nome della classe è `WidgetBoxTagHelper` , in Razor scriverai `<widget-box></widget-box>` .

- `Process` e `ProcessAsync` contengono la logica di rendering. Entrambi ricevono un parametro di **contesto** con informazioni sul tag corrente da sottoporre a rendering e un parametro di **output** utilizzato per personalizzare il risultato del rendering.

Qualsiasi assembly che contiene helper di tag personalizzati deve essere aggiunto al file **_ViewImports.cshtml** (si noti che è stato registrato l'assembly, non lo spazio dei nomi):

```
@addTagHelper *, MyAssembly
```

Sample Tag Helper Tag personalizzati

L'esempio seguente crea un helper di tag widget personalizzato che targetizzerà il markup del rasoio come:

```
<widget-box title="My Title">This is my content: @ViewData["Message"]</widget-box>
```

Che sarà reso come:

```
<div class="widget-box">
    <div class="widget-header">My Title</div>
    <div class="widget-body">This is my content: some message</div>
</div>
```

Il codice necessario per creare un tale tag helper è il seguente:

```
[HtmlTargetElement("widget-box")]
public class WidgetTagHelper : TagHelper
{
    public string Title { get; set; }

    public override async Task ProcessAsync(TagHelperContext context, TagHelperOutput output)
    {
        var outerTag = new TagBuilder("div");
        outerTag.Attributes.Add("class", output.TagName);
        output.MergeAttributes(outerTag);
        output.TagName = outerTag.TagName;

        //Create the header
        var header = new TagBuilder("div");
        header.Attributes.Add("class", "widget-header");
        header.InnerHtml.Append(this.Title);
        output.PreContent.SetHtmlContent(header);

        //Create the body and replace original tag helper content
        var body = new TagBuilder("div");
        body.Attributes.Add("class", "widget-body");
        var originalContents = await output.GetChildContentAsync();
        body.InnerHtml.Append(originalContents.GetContent());
        output.Content.SetHtmlContent(body);
```

```
    }
}
```

Etichetta Tag Helper

Label Tag Helper può essere utilizzato per rendere l' `label` per una proprietà del modello. Sostituisce il metodo `Html.LabelFor` nelle versioni precedenti di MVC.

Diciamo che hai un modello:

```
public class FormViewModel
{
    public string Name { get; set; }
}
```

Nella vista puoi usare l'elemento HTML `label` e l'helper dei tag `asp-for` :

```
<form>
    <label asp-for="Name"></label>
    <input asp-for="Name" type="text" />
</form>
```

Questo è equivalente al seguente codice nelle versioni precedenti di MVC:

```
<form>
    @Html.LabelFor(x => x.Name)
    @Html.TextBoxFor(x => x.Name)
</form>
```

Entrambi i frammenti di codice sopra renderizzano lo stesso HTML:

```
<form>
    <label for="Name">Name</label>
    <input name="Name" id="Name" type="text" value="">
</form>
```

Anchor tag helper

L'helper del tag di ancoraggio viene utilizzato per generare gli attributi href per il collegamento a una particolare azione del controller o una route MVC. Esempio di base

```
<a asp-controller="Products" asp-action="Index">Login</a>
```

A volte, è necessario specificare parametri aggiuntivi per l'azione del controller a cui si sta vincolando. Possiamo specificare i valori per questi parametri aggiungendo attributi con il prefisso asp-route.

```
<a asp-controller="Products" asp-action="Details" asp-route-id="@Model.ProductId">
   View Details
</a>
```

Leggi Tag Helpers online: https://riptutorial.com/it/asp-net-core/topic/2665/tag-helpers

Capitolo 25: Visualizza componenti

Examples

Crea un componente di vista

I componenti della vista incapsulano parti riutilizzabili di logica e viste. Sono definiti da:

- Una classe ViewComponent contenente la logica per prelevare e preparare i dati per la vista e decidere quale vista visualizzare.
- Una o più viste

Dal momento che contengono la logica, sono più flessibili delle viste parziali, pur continuando a promuovere una buona separazione delle preoccupazioni.

Un semplice componente di visualizzazione personalizzata è definito come:

```
public class MyCustomViewComponent : ViewComponent
{
    public async Task<IViewComponentResult> InvokeAsync(string param1, int param2)
    {
        //some business logic

        //renders ~/Views/Shared/Components/MyCustom/Default.cshtml
        return View(new MyCustomModel{ ... });
    }
}

@*View file located in ~/Views/Shared/Components/MyCustom/Default.cshtml*@
@model WebApplication1.Models.MyCustomModel
<p>Hello @Model.UserName!</p>
```

Possono essere richiamati da qualsiasi vista (o persino da un controller restituendo `ViewComponentResult`)

```
@await Component.InvokeAsync("MyCustom", new {param1 = "foo", param2 = 42})
```

Accedi Visualizza componente

Il modello di progetto predefinito crea una vista parziale **_LoginPartial.cshtml** che contiene un po 'di logica per scoprire se l'utente è connesso o meno e scoprire il suo nome utente.

Dal momento che un componente di visualizzazione potrebbe essere più adatto (dato che la logica è coinvolta e anche 2 servizi iniettati) l'esempio seguente mostra come convertire LoginPartial in un componente di visualizzazione.

Visualizza la classe del componente

```
public class LoginViewComponent : ViewComponent
```

```
{
    private readonly SignInManager<ApplicationUser> signInManager;
    private readonly UserManager<ApplicationUser> userManager;

    public LoginViewComponent(SignInManager<ApplicationUser> signInManager,
UserManager<ApplicationUser> userManager)
    {
        this.signInManager = signInManager;
        this.userManager = userManager;
    }

    public async Task<IViewComponentResult> InvokeAsync()
    {
        if (signInManager.IsSignedIn(this.User as ClaimsPrincipal))
        {
            return View("SignedIn", await userManager.GetUserAsync(this.User as
ClaimsPrincipal));
        }
        return View("SignedOut");
    }
}
```

SignedIn view (in ~ / Views / Shared / Components / Login / SignedIn.cshtml)

```
@model WebApplication1.Models.ApplicationUser

<form asp-area="" asp-controller="Account" asp-action="LogOff" method="post" id="logoutForm"
class="navbar-right">
    <ul class="nav navbar-nav navbar-right">
        <li>
            <a asp-area="" asp-controller="Manage" asp-action="Index" title="Manage">Hello
@Model.UserName!</a>
        </li>
        <li>
            <button type="submit" class="btn btn-link navbar-btn navbar-link">Log off</button>
        </li>
    </ul>
</form>
```

SignedOut view (in ~ / Views / Shared / Components / Login / SignedOut.cshtml)

```
<ul class="nav navbar-nav navbar-right">
    <li><a asp-area="" asp-controller="Account" asp-action="Register">Register</a></li>
    <li><a asp-area="" asp-controller="Account" asp-action="Login">Log in</a></li>
</ul>
```

Invocazione da **_Layout.cshtml**

```
@await Component.InvokeAsync("Login")
```

Return from Controller Action

Quando si eredita dalla classe `Controller` base fornita dal framework, è possibile utilizzare il metodo di convenienza `ViewComponent()` per restituire un componente di visualizzazione dall'azione:

```
public IActionResult GetMyComponent()
{
    return ViewComponent("Login", new { param1 = "foo", param2 = 42 });
}
```

Se si utilizza una classe POCO come controller, è possibile creare manualmente un'istanza della classe `ViewComponentResult` . Questo sarebbe equivalente al codice sopra:

```
public IActionResult GetMyComponent()
{
    return new ViewComponentResult
    {
        ViewComponentName = "Login",
        Arguments = new { param1 = "foo", param2 = 42 }
    };
}
```

Leggi Visualizza componenti online: https://riptutorial.com/it/asp-net-core/topic/3248/visualizza-componenti

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con asp.net-core	Alex Logan, Alexan, Ashish Rajput, Ashley Medway, Bogdan Stefanjuk, BrunoLM, ChadT, Community, gbellmann, Henk Mollema, Nate Barbettini, Rion Williams, Shog9, Shyju, Svek, Tseng, VSG24, Zach Becknell
2	Angular2 e .Net Core	Alejandro Tobón, Sentient Entities
3	ASP.NET Core: registra sia la richiesta che la risposta utilizzando il middleware	Gubr
4	Autorizzazione	gilmishal, RamenChef
5	Bundling e Minification	Rion Williams, Zach Becknell
6	caching	Cyprien Autexier, Sanket
7	Configurazione	Cyprien Autexier, Jayantha Lal Sirisena
8	Configurazione di più ambienti	dotnetom, Johnny, Robert Paulsen, Sanket, Set, Tseng
9	Gestione degli errori	Sanket, Set
10	Iniezione di dipendenza	Alexan, BrunoLM, Cyprien Autexier, Dan Soper, Darren Evans, gilmishal, Gurgen Hakobyan, Jayantha Lal Sirisena, Joel Harkes, maztt, Tseng, Zach Becknell
11	Iniezione di servizi in viste	Alex Logan, Rion Williams
12	Invio di email in app .Net core tramite MailKit	Ankit
13	Lavorare con JavascriptServices	hmnzr
14	Limitazione di velocità	Stefan P.

15	Localizzazione	Tseng, VSG24, Zach Becknell
16	middleware	Ali, Piotrek, Set, VSG24, Zach Becknell
17	Modelli	Alex Logan, Ralf Bönning
18	project.json	Joel Harkes
19	Pubblicazione e distribuzione	Set
20	Registrazione	Dmitry, Sanket, Set, Tseng
21	Richieste di origine incrociata (CORS)	Henk Mollema, Sanket, Saqib Rokadia, Tseng
22	Routing	ChadT, Tseng
23	Sessioni in ASP.NET Core 1.0	ravindra, Sanket
24	Tag Helpers	Ali, Daniel J.G., dotnetom, Shyju, tmg, Zach Becknell
25	Visualizza componenti	Daniel J.G.