eBook Gratuit

APPRENEZ asp.net-core

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#asp.net-core

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: asp-net-core

It is an unofficial and free asp.net-core ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official asp.net-core.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec asp.net-core

Remarques

.NET Core est une plate-forme de développement à usage général gérée par Microsoft et la communauté .NET sur GitHub. Il est multi-plateforme, prend en charge Windows, macOS et Linux et peut être utilisé dans des scénarios de périphériques, de cloud et intégrés / IoT.

Les caractéristiques suivantes définissent mieux .NET Core:

- Déploiement flexible: peut être inclus dans votre application ou installé côte à côte à l'échelle de l'utilisateur ou de la machine.
- Multiplate-forme: s'exécute sur Windows, macOS et Linux; peut être porté sur d'autres systèmes d'exploitation. Les systèmes d'exploitation, les processeurs et les scénarios d'application pris en charge évolueront avec le temps, fournis par Microsoft, d'autres sociétés et des particuliers.
- Outils de ligne de commande: tous les scénarios de produit peuvent être exercés sur la ligne de commande.
- Compatible: .NET Core est compatible avec .NET Framework, Xamarin et Mono, via la bibliothèque standard .NET.
- Open source: la plate-forme .NET Core est open source et utilise les licences MIT et Apache 2. La documentation est sous licence CC-BY. .NET Core est un projet .NET Foundation.
- Pris en charge par Microsoft: .NET Core est pris en charge par Microsoft, par le biais du support .NET Core

Versions

Version	Notes de version	Date de sortie
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

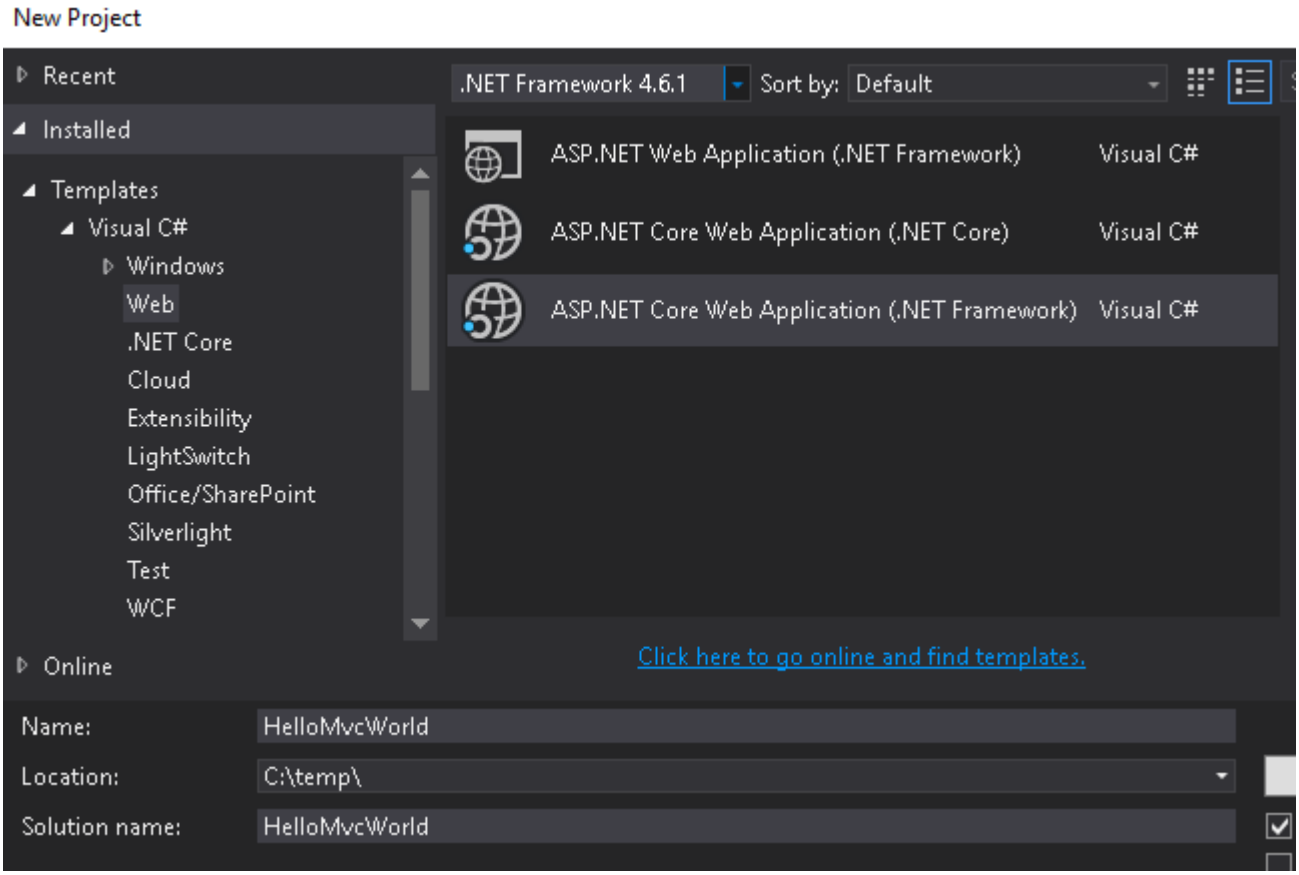
Installation et configuration

Installation de Visual Studio

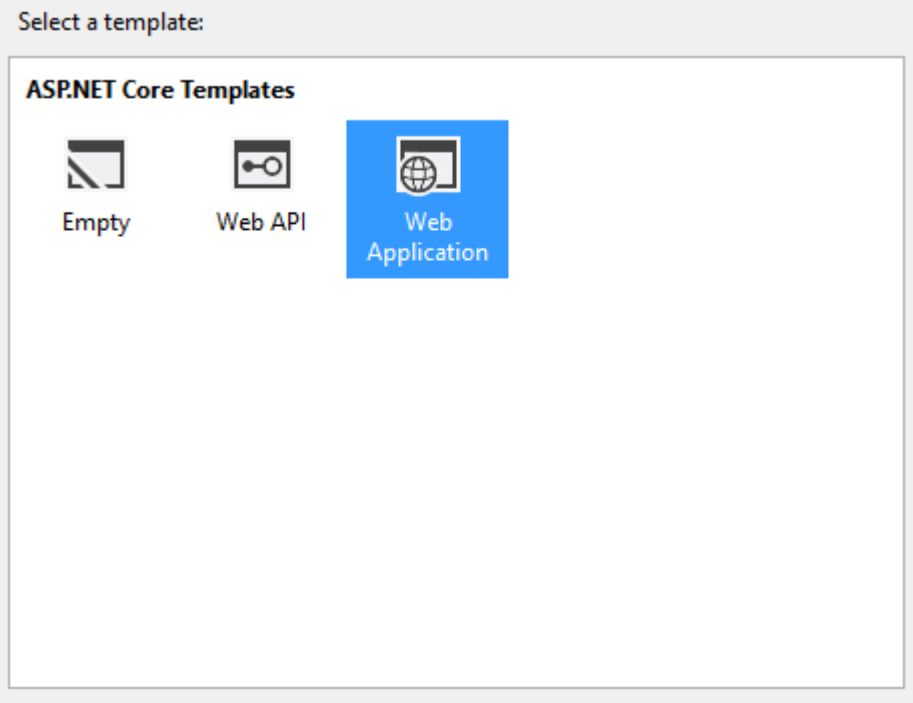
Si Visual Studio n'est pas installé, vous pouvez télécharger gratuitement Visual Studio Community Edition ici . Si vous l'avez déjà installé, vous pouvez passer à l'étape suivante.

Création d'une application ASP.NET Core MVC.

1. **Ouvrez Visual Studio.**
2. **Sélectionnez Fichier> Nouveau projet.**
3. **Sélectionnez Web dans la langue de votre choix** dans la section Modèles à gauche.
4. **Choisissez un type de projet préféré** dans la boîte de dialogue.
5. **Facultatif: Choisissez un framework .NET que vous souhaitez cibler**
6. **Nommez votre projet** et indiquez si vous souhaitez créer une solution pour le projet.
7. **Cliquez sur OK** pour créer le projet.



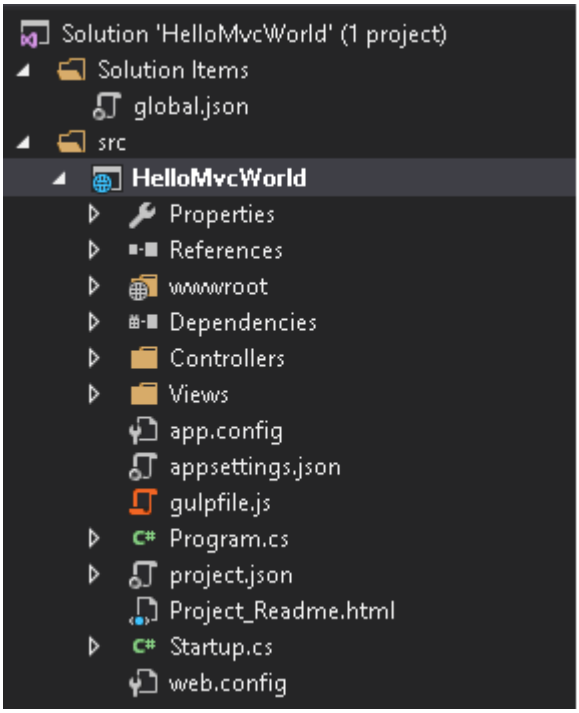
Une autre boîte de dialogue vous sera proposée pour sélectionner le modèle que vous souhaitez utiliser pour le projet:



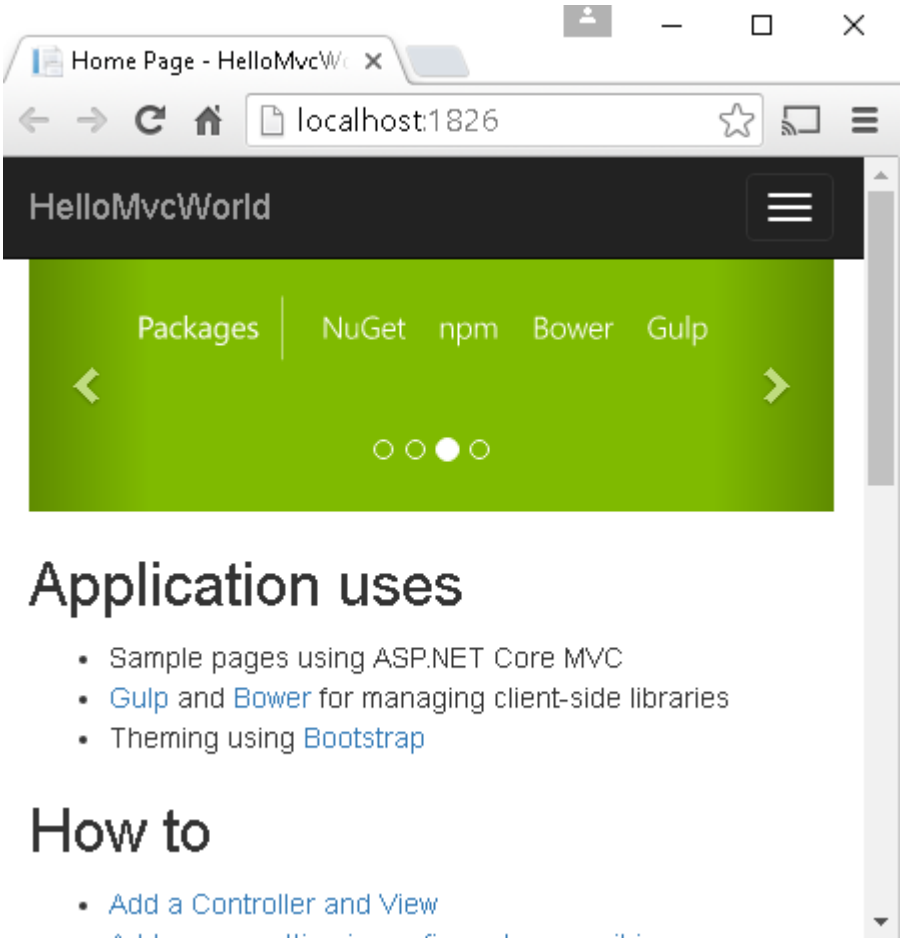
Chacune des descriptions est explicite. Pour ce premier projet, **sélectionnez Application Web** , qui contiendra toutes les configurations par défaut, l'authentification et certains contenus existants.

Comme il s'agit d'une application d'introduction ne nécessitant aucune sécurité ou authentification, vous pouvez **modifier l'option d'authentification** sur **Aucune authentification** du côté droit de la boîte de dialogue et **cliquer sur OK pour créer le projet** .

Vous devriez alors voir le nouveau projet dans l'Explorateur de solutions:



Appuyez sur la touche F5 pour exécuter l'application et lancer une session de débogage, qui lancera l'application dans votre navigateur par défaut:



Vous pouvez maintenant voir que votre projet est opérationnel localement et que vous êtes prêt à créer votre application.

Créer un nouveau projet à partir de la ligne de commande

Il est possible de créer un nouveau projet ASP.NET Core entièrement à partir de la ligne de commande en utilisant la commande `dotnet` .

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

`dotnet new web` un nouveau projet web "vide". Le paramètre `web` indique à l'outil `dotnet` d'utiliser le modèle `ASP.NET Core Empty` . Utilisez `dotnet new -all` pour afficher tous les modèles disponibles actuellement installés. Les autres modèles de clé incluent `console` , `classlib` , `mvc` et `xunit` .

Une fois que le modèle a été échafaudé, vous pouvez restaurer les packages requis pour exécuter le projet (`dotnet restore`), puis le compiler et le démarrer (`dotnet run`).

Une fois le projet en cours, il sera disponible sur le port par défaut: http: // localhost: 5000

API Web ASP.NET Core minimale avec ASP.NET Core MVC

Avec ASP.NET Core 1.0, le framework MVC et Web API ont été fusionnés en un seul framework appelé ASP.NET Core MVC. C'est une bonne chose, puisque MVC et Web API partagent de

nombreuses fonctionnalités, mais il y avait toujours des différences subtiles et la duplication de code.

Cependant, le fait de fusionner ces deux éléments dans le cadre du premier a rendu plus difficile la distinction entre les deux. Par exemple, `Microsoft.AspNet.WebApi` représente le framework Web API 5.xx, pas le nouveau. Mais lorsque vous incluez `Microsoft.AspNetCore.Mvc` (version `1.0.0`), vous obtenez le package complet. Cela contiendra *toutes* les fonctionnalités prêtes à l'emploi proposées par le framework MVC. Comme le rasoir, les aides de balise et la reliure de modèle.

Lorsque vous souhaitez simplement créer une API Web, nous n'avons pas besoin de toutes ces fonctionnalités. Alors, comment pouvons-nous construire une API Web minimaliste? La réponse est: `Microsoft.AspNetCore.Mvc.Core` . Dans le nouveau monde, MVC est divisé en plusieurs packages et ce package ne contient que les composants principaux du framework MVC, tels que le routage et l'autorisation.

Pour cet exemple, nous allons créer une API MVC minimale. Y compris un formateur JSON et CORS. Créez une application Web ASP.NET Core 1.0 vide et ajoutez ces packages à votre project.json:

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

Maintenant, nous pouvons enregistrer MVC en utilisant `AddMvcCore()` dans la classe de démarrage:

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

`AddMvcCore` renvoie une instance `IMvcCoreBuilder` qui permet la création ultérieure. La configuration du middleware est la même que d'habitude:

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

Contrôleurs

L'ancienne API Web est livrée avec sa propre classe de base de contrôleur: `ApiController` . Dans le nouveau monde, il n'ya rien de tel, seulement la classe de `Controller` par défaut. Malheureusement, cette classe de base est relativement volumineuse et liée à la liaison de modèle, aux vues et à JSON.NET.

Heureusement, dans le nouveau framework, les classes de contrôleur n'ont pas à dériver de `Controller` pour être captées par le mécanisme de routage. Il suffit d'ajouter le nom avec `Controller` . Cela nous permet de construire notre propre classe de base de contrôleur. Appelons cela `ApiController` , juste pour le bon vieux temps:

```
/// <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'attribut `[Controller]` indique que le type ou tout type dérivé est considéré comme un contrôleur par le mécanisme de découverte de contrôleur par défaut. L'attribut `[ActionContext]` spécifie que la propriété doit être définie avec le paramètre `ActionContext` actuel lorsque MVC crée le contrôleur. `ActionContext` fournit des informations sur la requête en cours.

> ASP.NET Core MVC offre également une classe `ControllerBase` qui fournit une classe de base de contrôleur sans support de vues. C'est quand même beaucoup plus grand que le nôtre. Utilisez-le si vous le trouvez pratique.

Conclusion

Nous pouvons maintenant construire une API Web minimale en utilisant le nouveau framework ASP.NET Core MVC. La structure modulaire du package nous permet de récupérer les packages dont nous avons besoin et de créer une application simple et simple.

Utilisation du code Visual Studio pour développer une application core aspnet cross plateforme

Avec AspNetCore, vous pouvez développer l'application sur n'importe quelle plate-forme, y compris Mac, Linux, Window et Docker.

Installation et configuration

1. Installer Visual Studio Code d' ici
2. Ajouter l' extion C #
3. Installez dot net core sdk. Vous pouvez installer à partir d' ici

Vous avez maintenant tous les outils disponibles. Développer l'application Maintenant, vous avez

besoin d'une option d'échafaudage. Pour cela, vous devriez envisager d'utiliser Yeoman. Pour installer Yeoman

1. Installez NPM. Pour cela, vous avez besoin d'un nœud sur votre machine. Installer à partir d' ici

2. Installez Yeoman en utilisant NPM

   npm install -g yo

3. Maintenant, installez le générateur d'aspnet

   npm install -g generator-aspnet

Maintenant, nous avons toute la configuration sur votre machine. Commençons par créer un nouveau projet avec la commande de base DotNetCore et ensuite créer un nouveau projet en utilisant Yo.

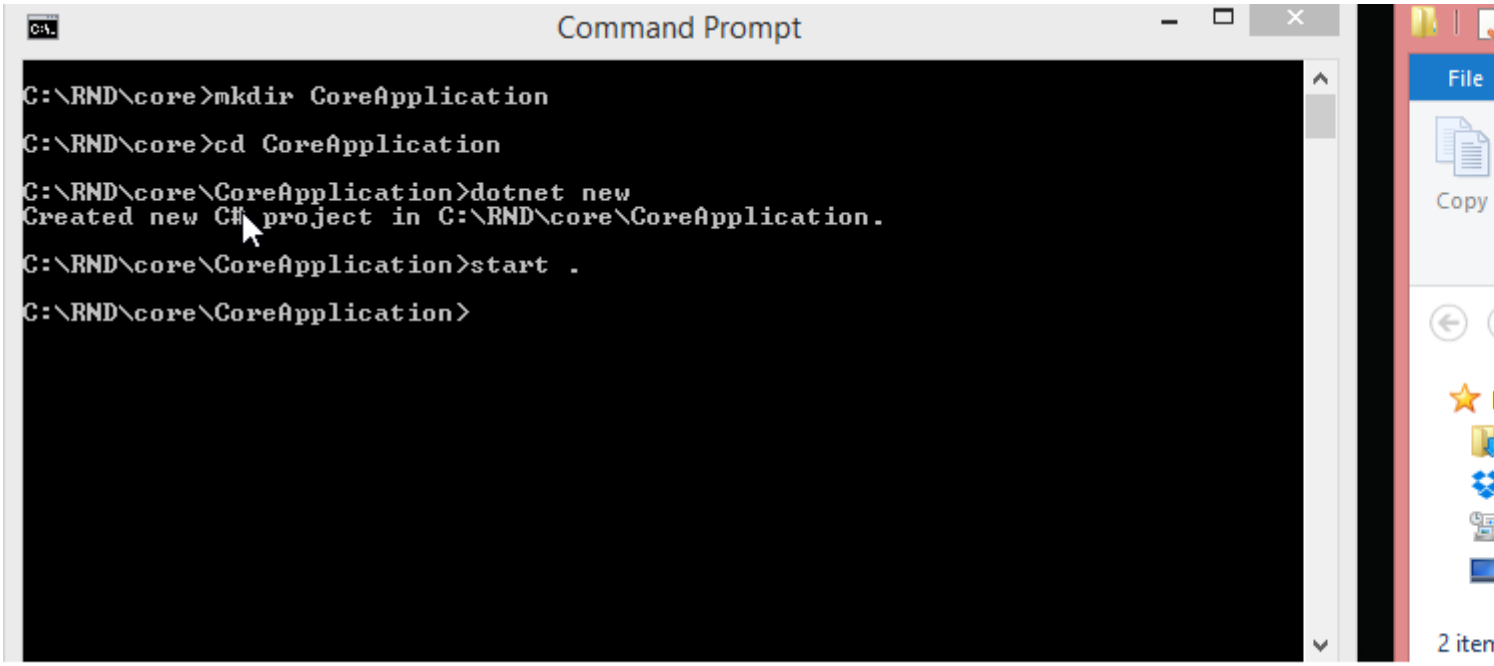
Nouveau projet utilisant la ligne de commande

1. Créer un nouveau dossier de projet

   mkdir CoreApplication cd CoreApplication

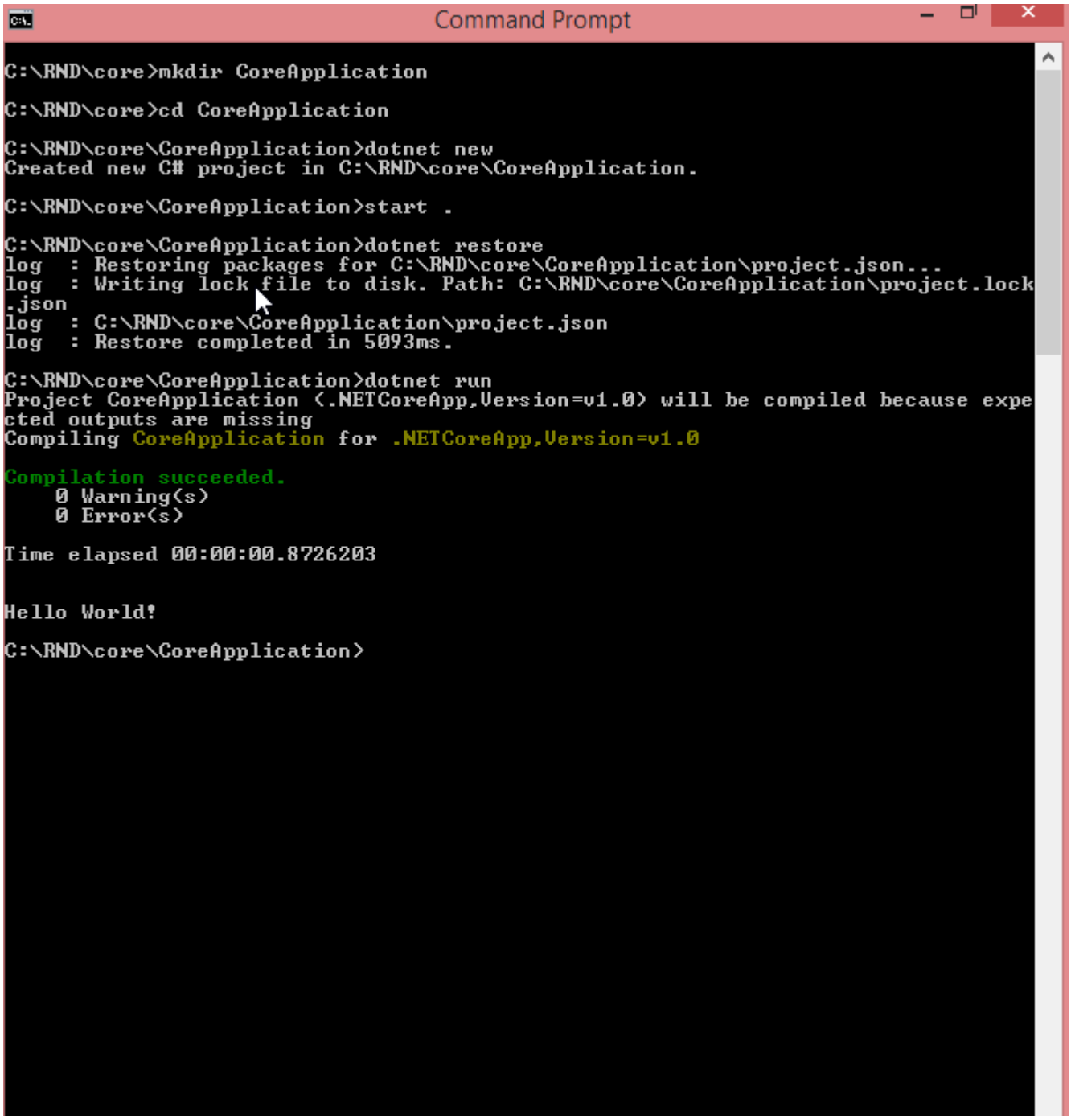
2. Echafaudage d'un projet dotnet très basique en utilisant l'option de ligne de commande par défaut

   dotnet Nouveau



1. Restaurez les packages et exécutez l'application

   dotNet restauration de runnet

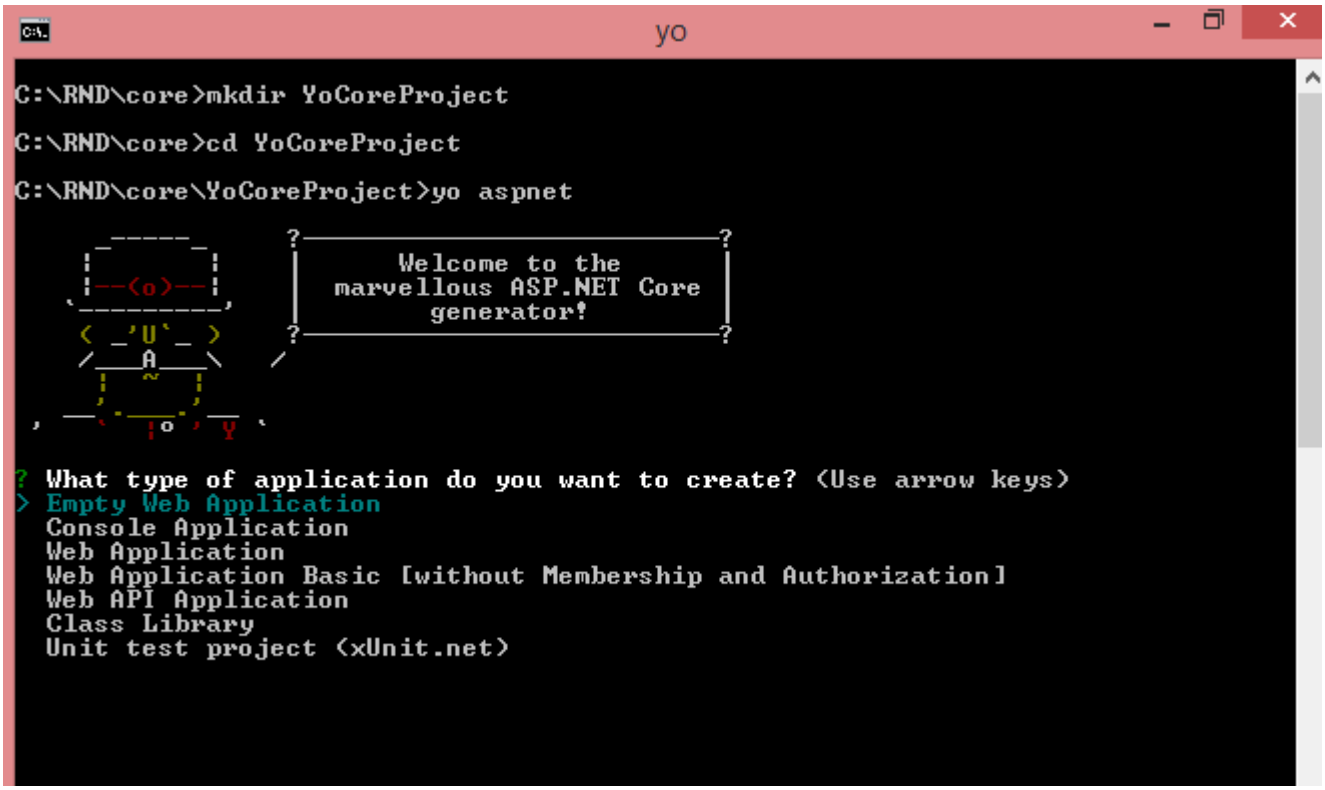


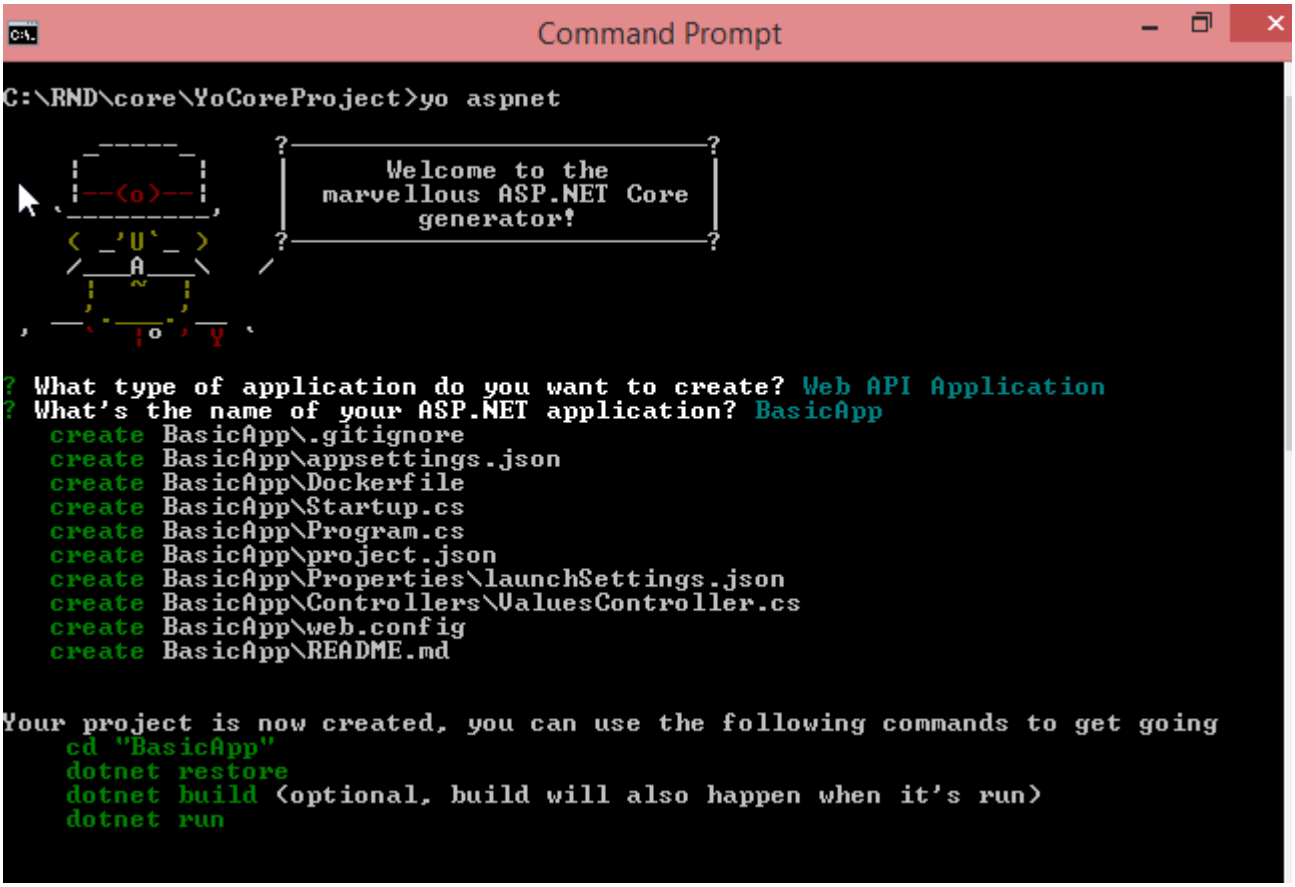
Utilisez Yeoman comme option d'échafaudage

Créer un dossier de projet et exécuter la commande Yo

```
yo aspnet
```

Yeoman demandera des entrées comme Type de projet, Nom du projet, etc.

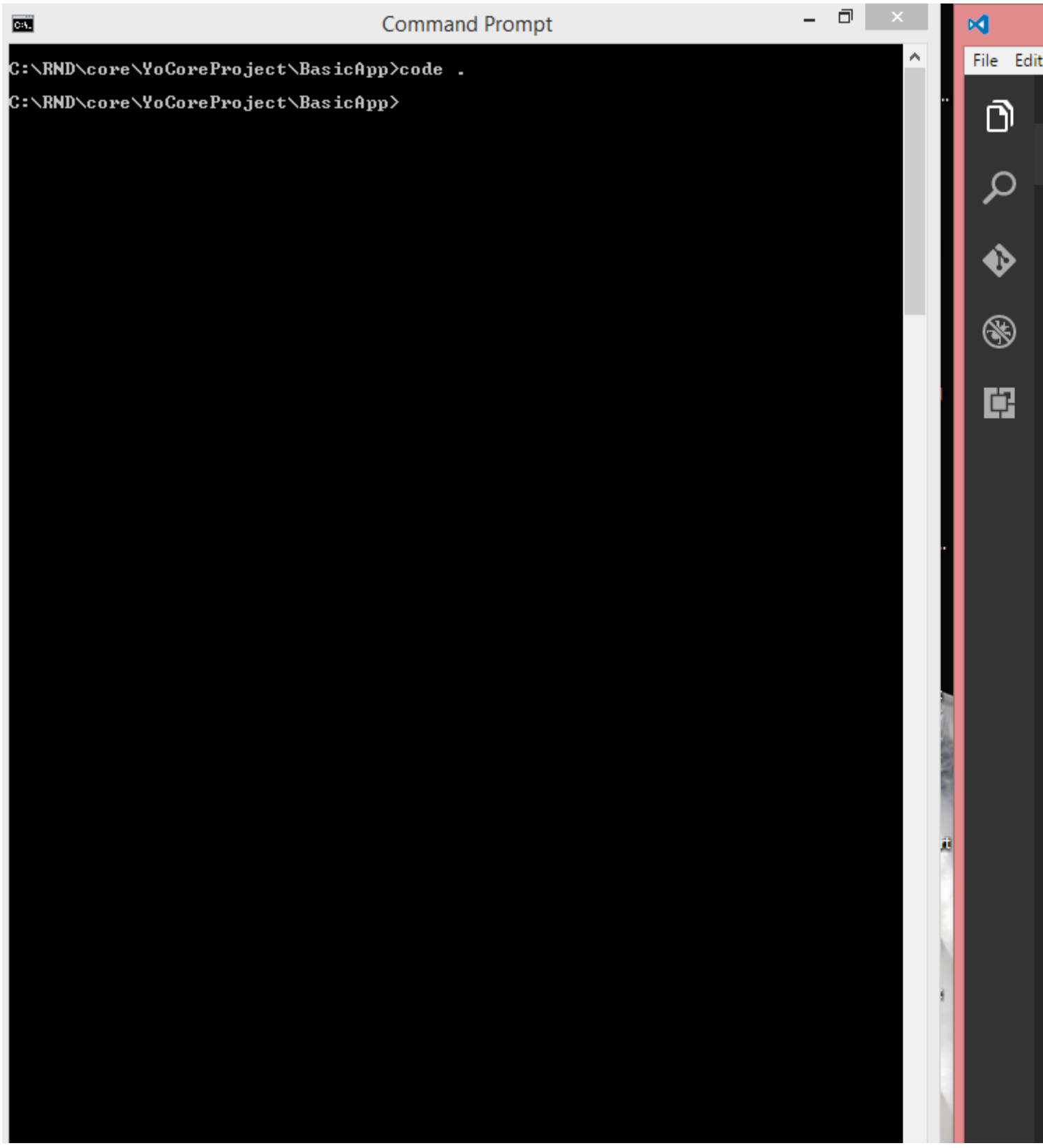




Maintenant, restaurez les paquets en exécutant la commande dotnet restore et exécutez l'application

Utiliser le code VS pour développer l'application

Exécutez le code studio visuel comme



Ouvrez maintenant les fichiers et exécutez l'application. Vous pouvez également rechercher l'extension pour votre aide.

Variable d'environnement d'installation dans ASP.NET Core [Windows]

=> Poste d'origine <=

ASP.NET Core utilise la variable d'environnement `ASPNETCORE_ENVIRONMENT` pour déterminer

l'environnement actuel. Par défaut, si vous exécutez votre application sans définir cette valeur, elle sera automatiquement utilisée par défaut dans l'environnement de `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.
```

Définition de la variable d'environnement dans Windows

Sur la ligne de commande

Vous pouvez facilement définir une variable d'environnement à partir d'une invite de commandes à l'aide de la commande setx.exe incluse dans Windows. Vous pouvez l'utiliser pour définir facilement une variable utilisateur:

```
>setx ASPNETCORE_ENVIRONMENT "Development"

SUCCESS: Specified value was saved.
```

Notez que la variable d'environnement n'est pas définie dans la fenêtre ouverte en cours. Vous devrez ouvrir une nouvelle invite de commande pour voir l'environnement mis à jour. Il est également possible de définir des variables système (plutôt que des variables utilisateur) si vous ouvrez une invite de commande administrative et ajoutez le commutateur / M:

```
>setx ASPNETCORE_ENVIRONMENT "Development" /M

SUCCESS: Specified value was saved.
```

Utilisation de PowerShell Vous pouvez également utiliser PowerShell pour définir la variable. Dans PowerShell, ainsi que les variables utilisateur et système normales, vous pouvez également créer une variable temporaire à l'aide de la commande `$Env:` :

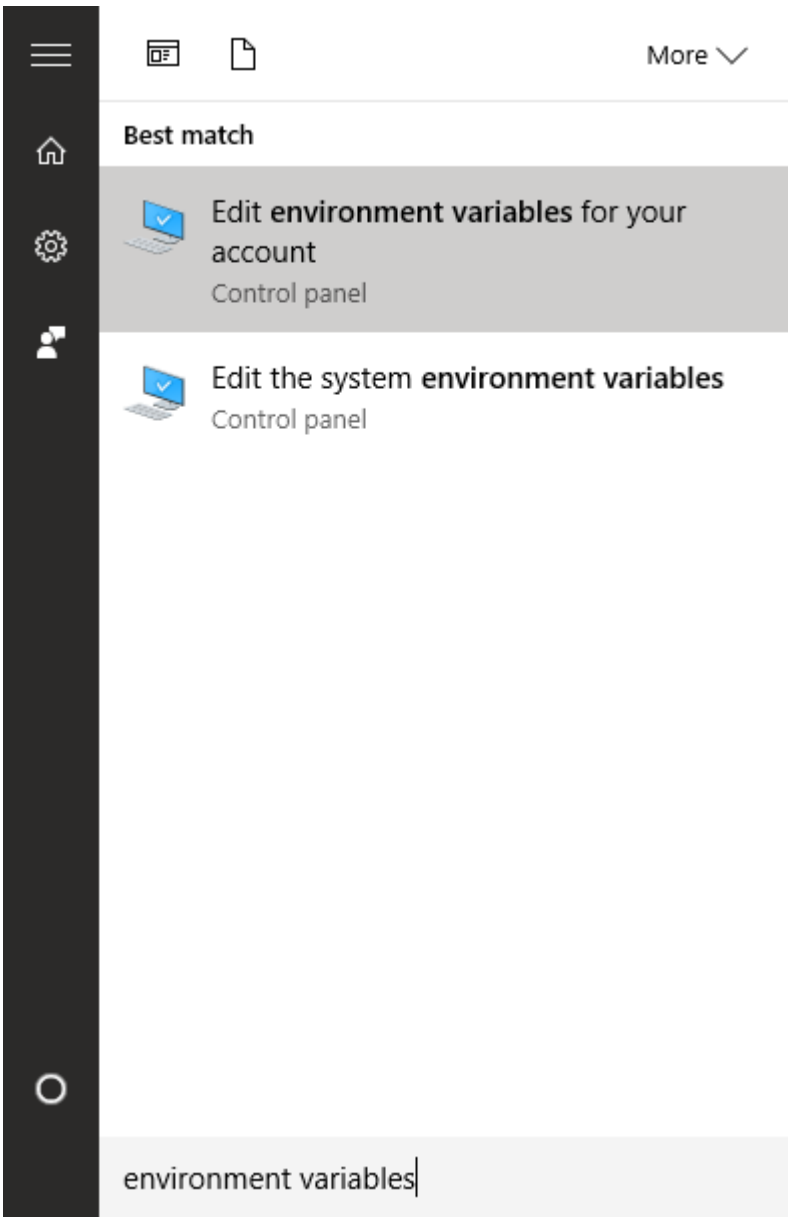
```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

La variable créée ne dure que pendant la durée de votre session PowerShell. Une fois la fenêtre fermée, l'environnement revient à sa valeur par défaut.

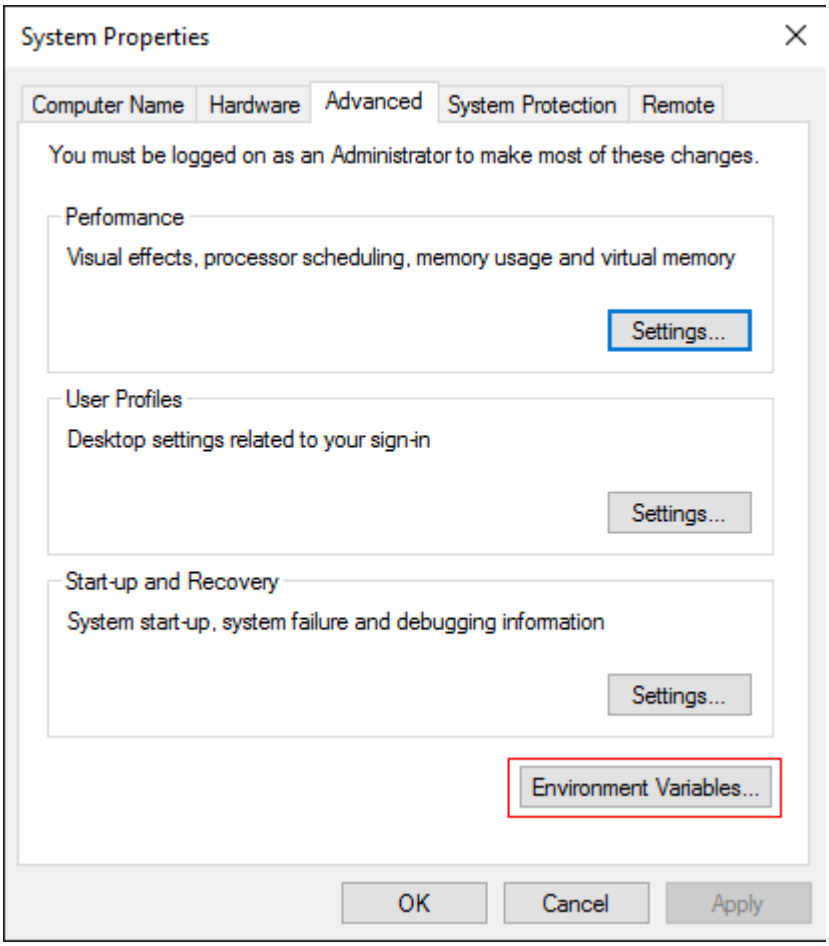
Vous pouvez également définir directement les variables d'environnement utilisateur ou système. Cette méthode ne modifie pas les variables d'environnement dans la session en cours. Vous devrez donc ouvrir une nouvelle fenêtre PowerShell pour voir vos modifications. Comme précédemment, la modification des variables système (machine) nécessitera un accès administratif

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

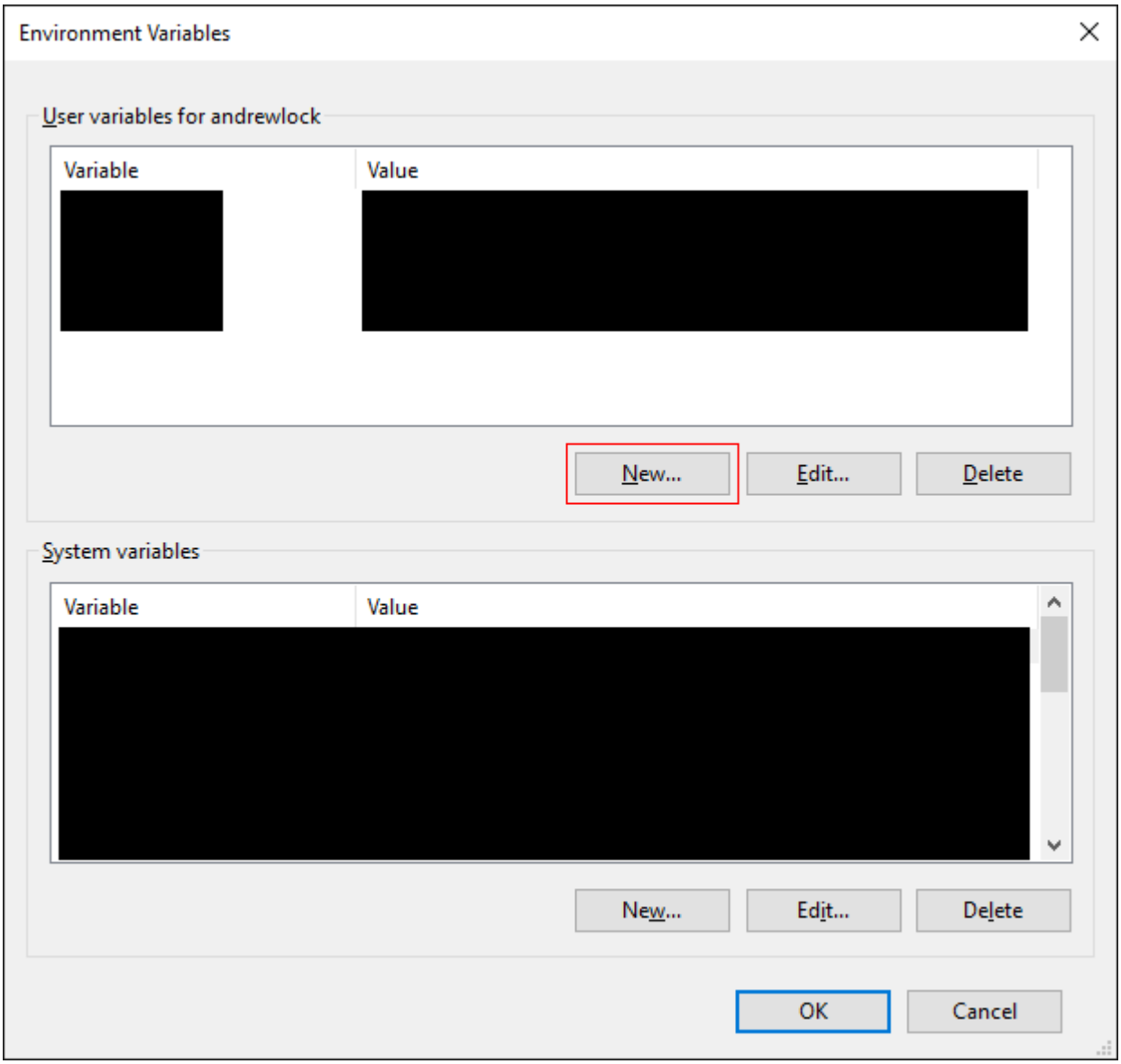
Utilisation du panneau de configuration Windows Si vous n'êtes pas fan de l'invite de commande, vous pouvez facilement mettre à jour vos variables avec la souris! Cliquez sur le bouton Démarrer de Windows (ou appuyez sur la touche Windows), recherchez `environment variables` *environnement variables pour votre compte:*



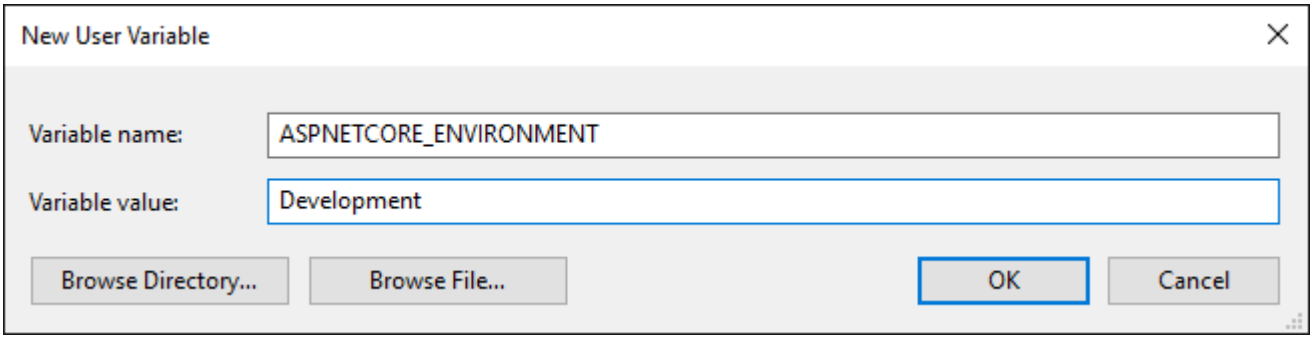
La sélection de cette option ouvre la boîte de dialogue Propriétés système



Cliquez sur Variables d'environnement pour afficher la liste des variables d'environnement actuelles sur votre système.



En supposant que vous ne possédez pas déjà une variable appelée `ASPNETCORE_ENVIRONMENT` , cliquez sur le bouton Nouveau ... et ajoutez une nouvelle variable d'environnement de compte:



Cliquez sur OK pour enregistrer toutes vos modifications. Vous devrez rouvrir toutes les fenêtres de commande pour vous assurer que les nouvelles variables d'environnement sont chargées.

Lire Démarrer avec asp.net-core en ligne: https://riptutorial.com/fr/asp-net-core/topic/810/demarrer-avec-asp-net-core

Chapitre 2: Afficher les composants

Examples

Créer un composant de vue

Les composants de vue encapsulent des éléments de logique et des vues réutilisables. Ils sont définis par:

- Classe ViewComponent contenant la logique d'extraction et de préparation des données pour la vue et de choix de la vue à rendre.
- Une ou plusieurs vues

Comme ils contiennent une logique, ils sont plus flexibles que les vues partielles tout en favorisant une bonne séparation des préoccupations.

Un composant de vue personnalisée simple est défini comme suit:

```
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>
```

Ils peuvent être appelés depuis n'importe quelle vue (ou même un contrôleur en retournant un `ViewComponentResult`)

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

Composant View View

Le modèle de projet par défaut crée une vue partielle **_LoginPartial.cshtml** qui contient un peu de logique pour savoir si l'utilisateur est connecté ou non et trouver son nom d'utilisateur.

Comme un composant de vue peut convenir mieux (car il y a une logique impliquée et même 2 services injectés), l'exemple suivant montre comment convertir le LoginPartial en un composant de vue.

Afficher la classe de composant

```
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>
```

Vue SignedOut (dans ~ / 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>
```

Invocation depuis **_Layout.cshtml**

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

Retour de l'action du contrôleur

Lorsque vous héritez de la classe `Controller` de base fournie par la structure, vous pouvez utiliser la méthode `ViewComponent()` pour renvoyer un composant de vue à partir de l'action:

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

Si vous utilisez une classe POCO en tant que contrôleur, vous pouvez créer manuellement une instance de la classe `ViewComponentResult` . Ce serait équivalent au code ci-dessus:

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

Lire Afficher les composants en ligne: https://riptutorial.com/fr/asp-net-core/topic/3248/afficher-les-composants

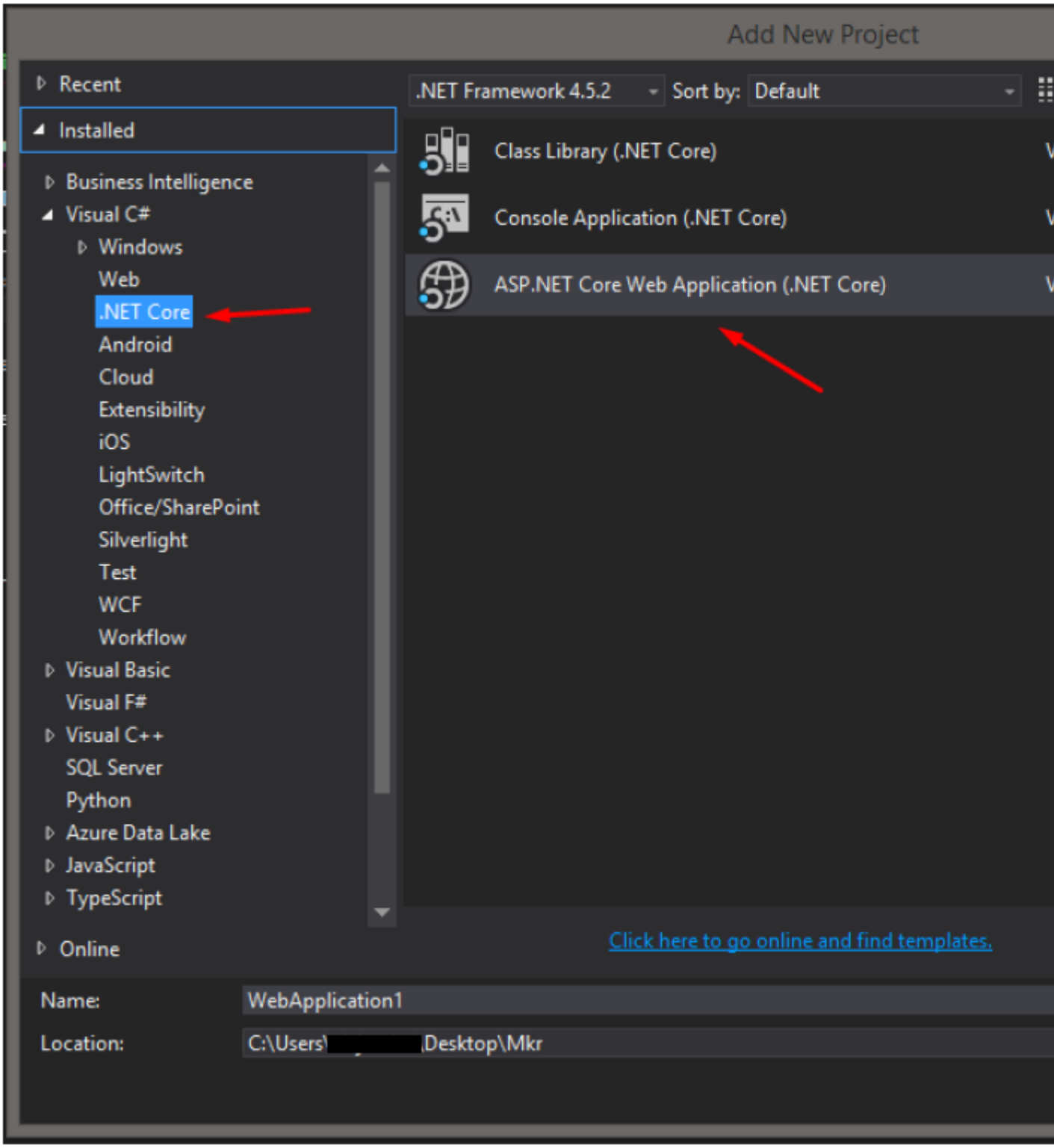
Chapitre 3: Angular2 et .Net Core

Examples

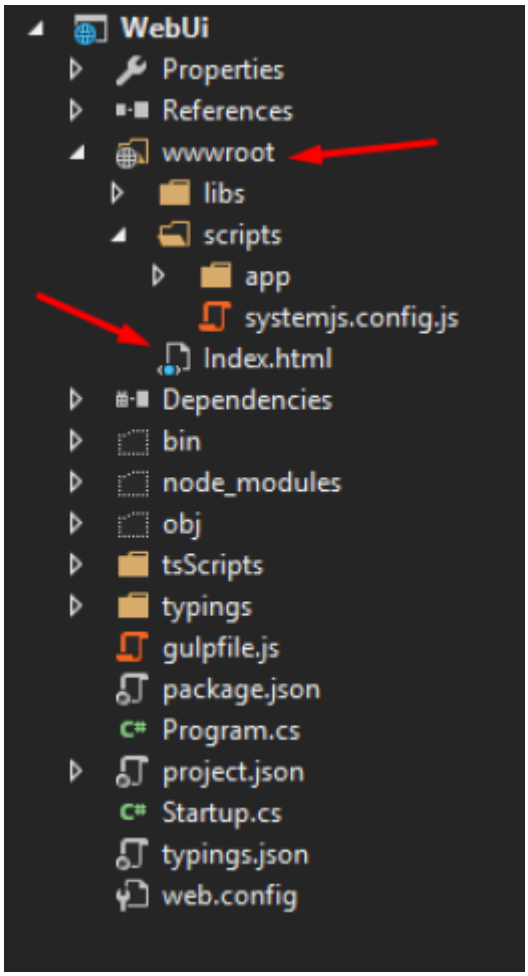
Tutoriel rapide pour un Angular 2 Hello World! App avec .Net Core dans Visual Studio 2015

Pas:

1. Créer une application Web .Net Core vide:

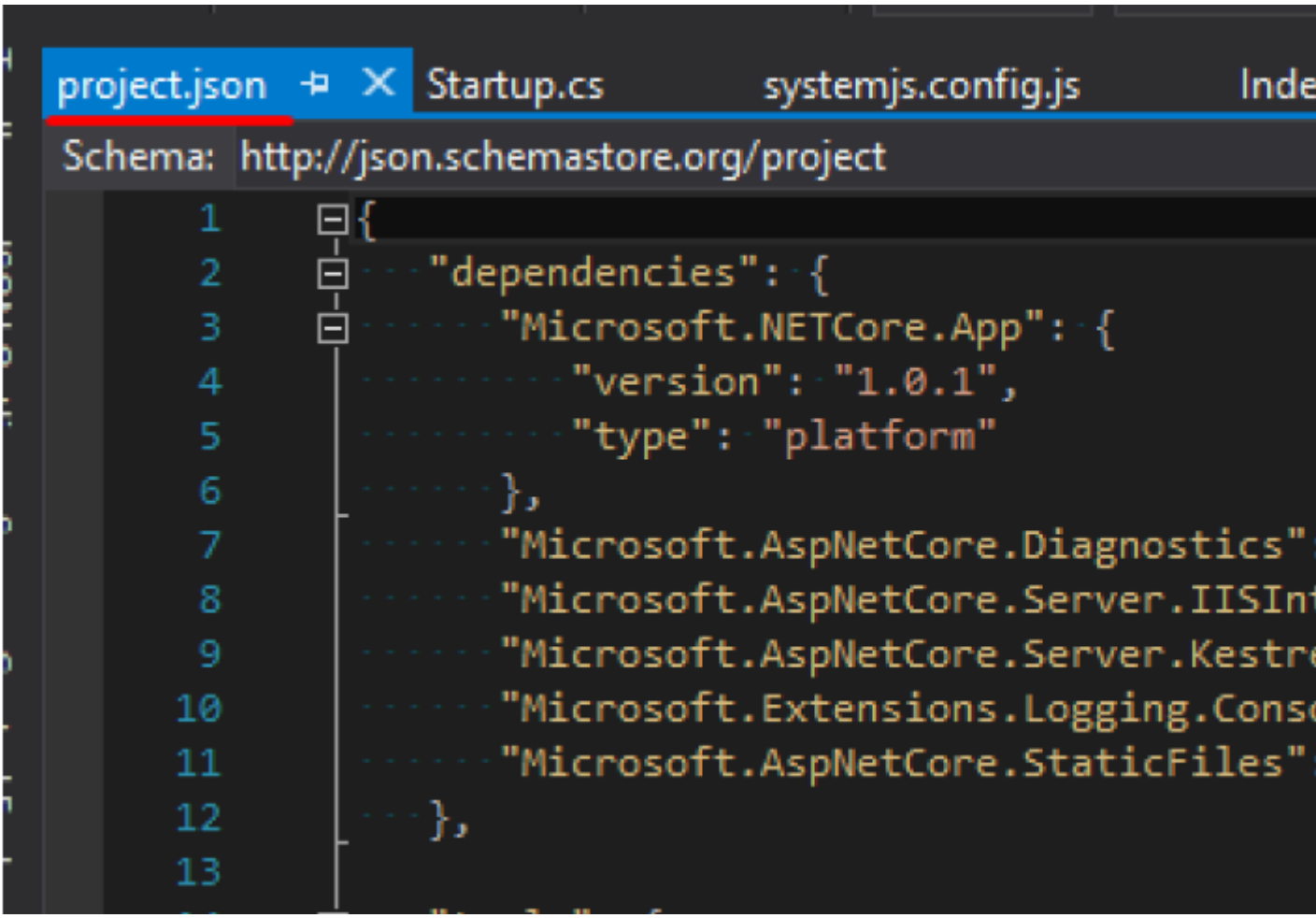


2. Allez sur wwwroot et créez une page HTML normale appelée Index.html:



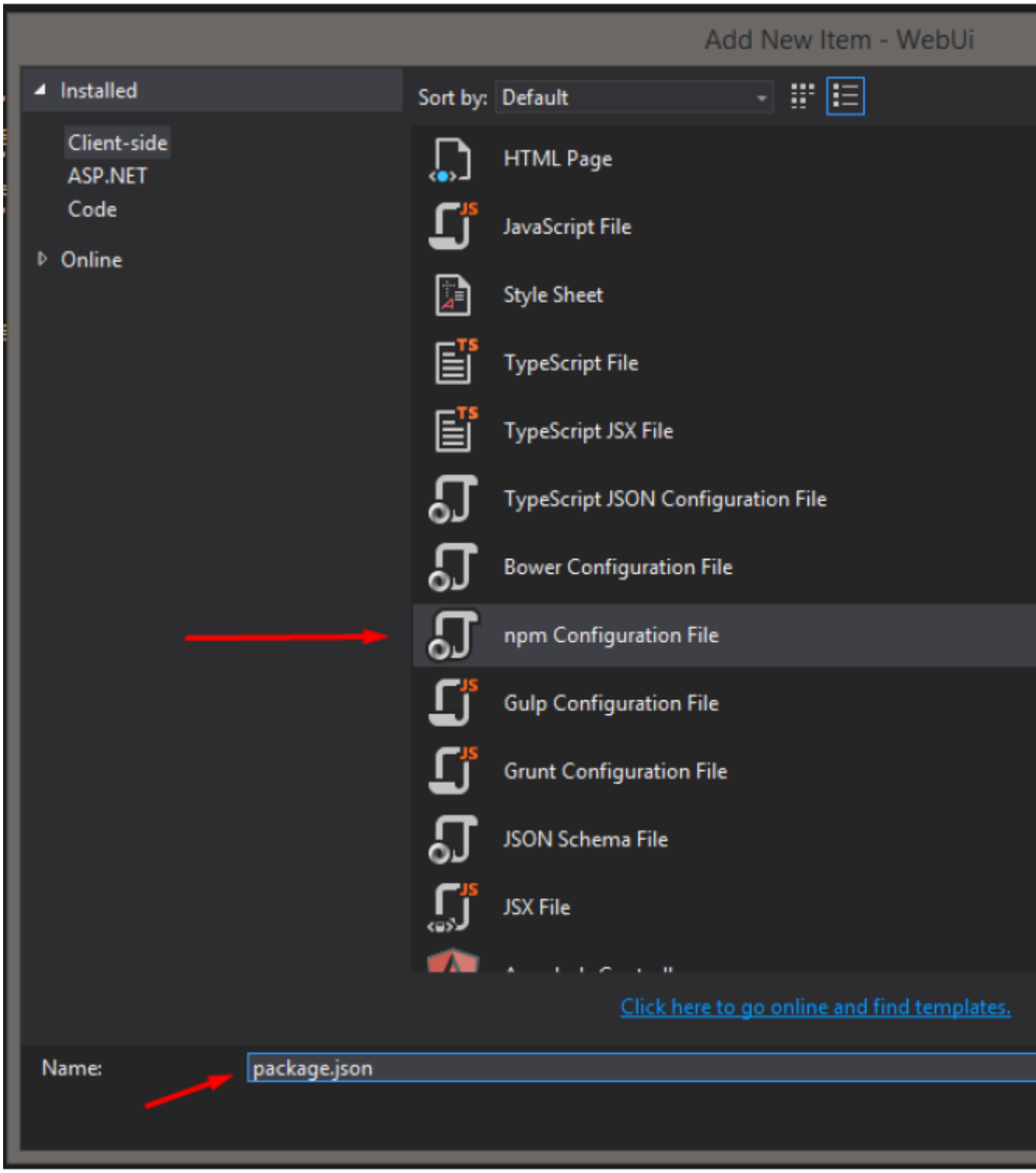
3. Configurez Startup.cs pour accepter les fichiers statiques (ceci nécessitera d'ajouter la bibliothèque "Microsoft.AspNetCore.StaticFiles": "1.0.0" dans le fichier "project.json"):

```
// 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. Ajouter un fichier NPN:

    - Cliquez avec le bouton droit sur le projet WebUi et ajoutez le fichier de configuration NPN (package.json):



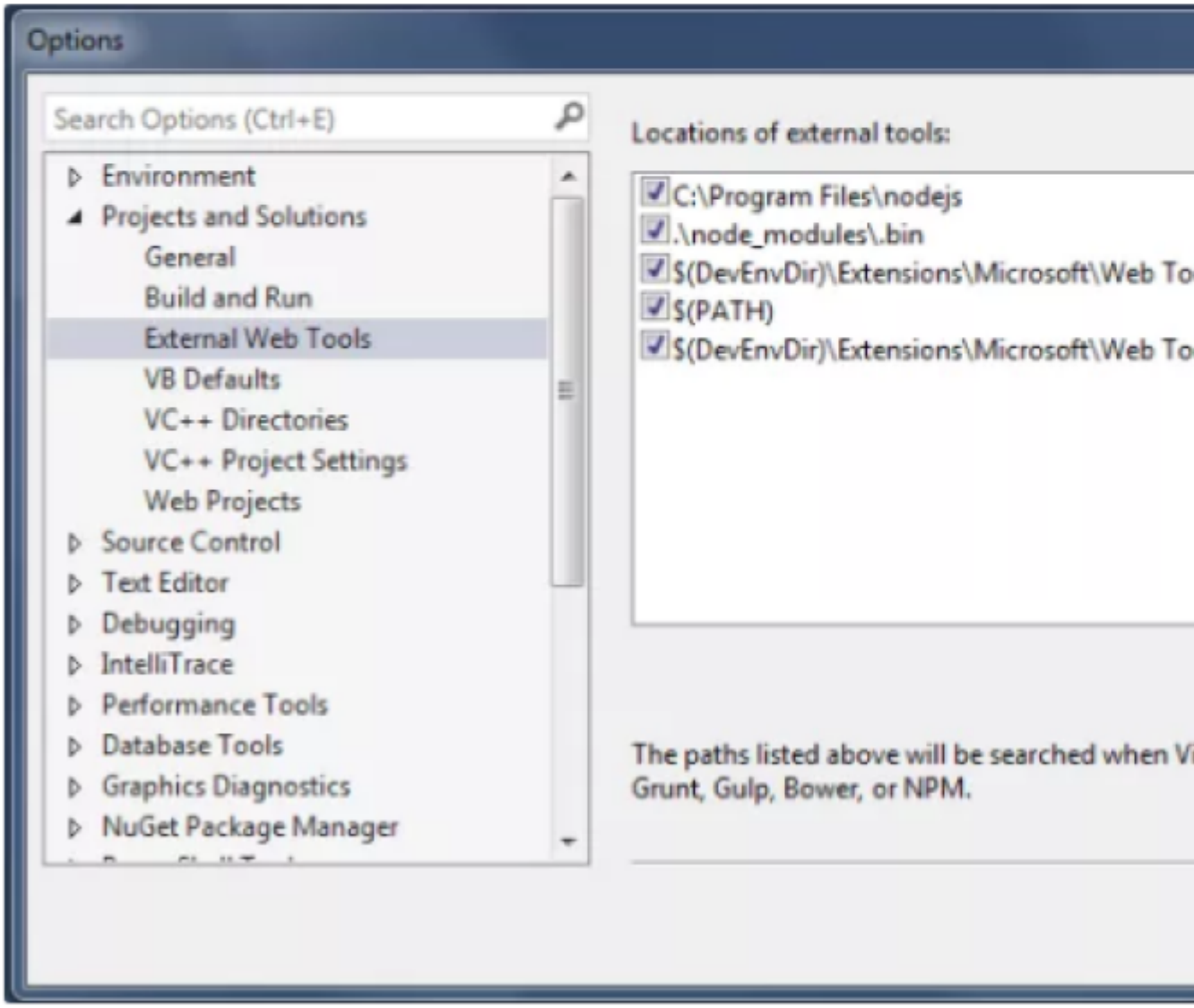
- Vérifiez les dernières versions des packages:

Remarque: Si Visual Studio ne détecte pas les versions des packages (vérifiez tous les packages, car certains d'entre eux affichent la version, et d'autres non), cela peut être dû au fait que la version Node fournie par Visual Studio ne fonctionne pas correctement , il faudra donc probablement installer le noeud js en externe et lier cette installation à Visual Studio.

je. Téléchargez et installez le noeud js: https://nodejs.org/es/download/

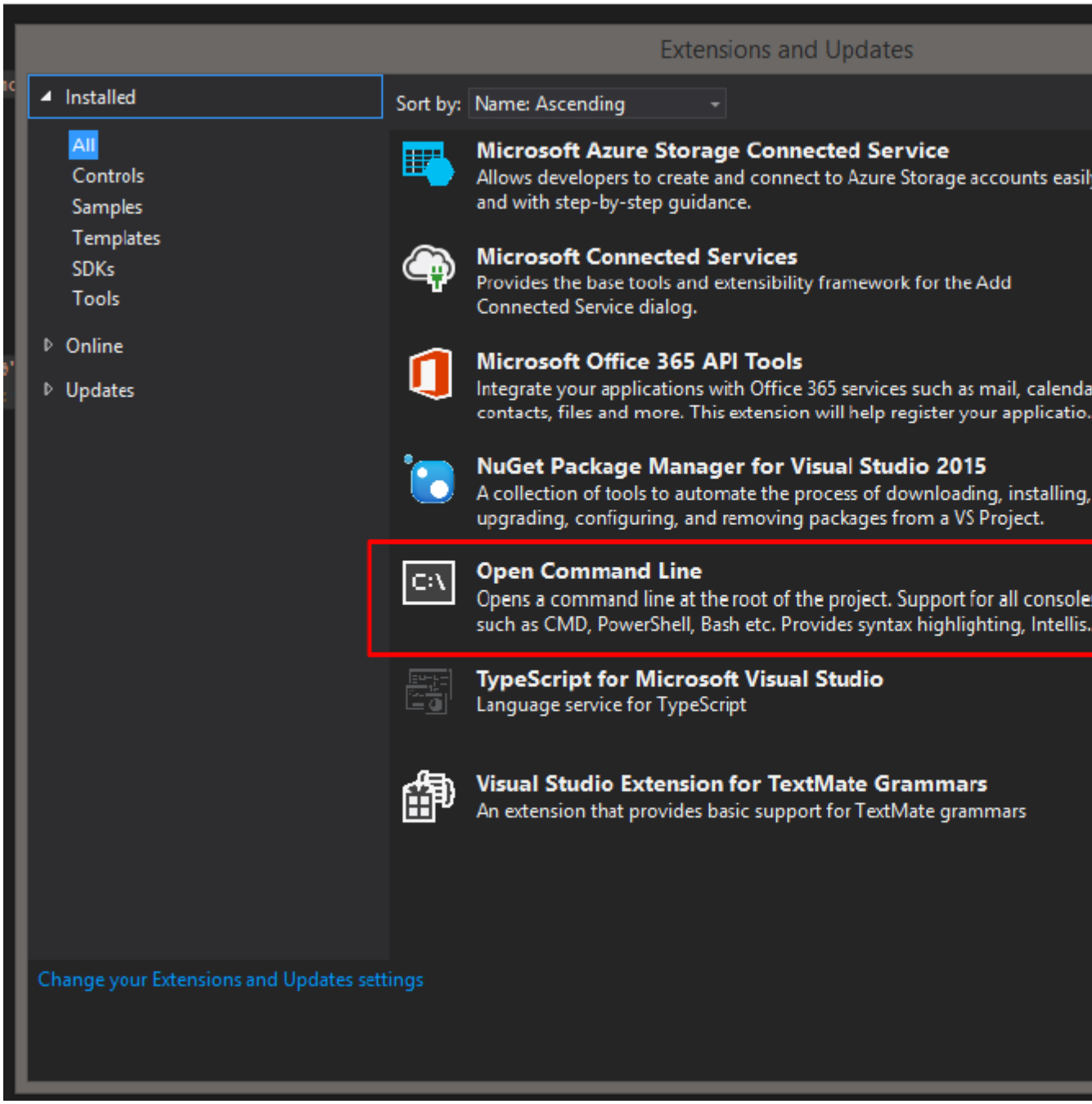
ii. Liez l'installation à Visual Studio: https://ryanhayes.net/synchronize-node-js-install-version-with-visual-studio-2015/ :

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.



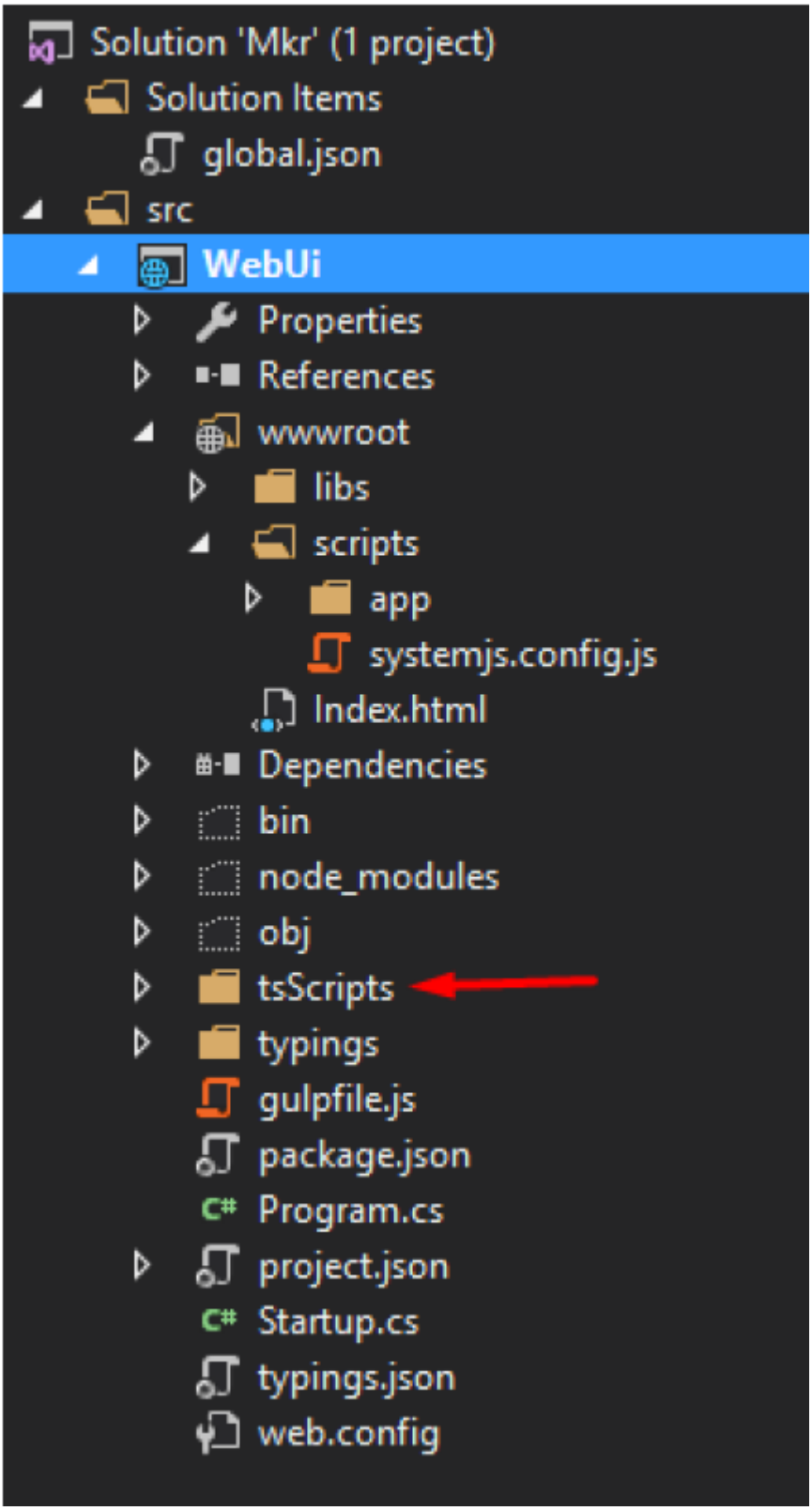
iii. (Facultatif) Après avoir enregistré le package.json, il installera les dépendances dans le projet, sinon, exécutez "npm install" en utilisant une invite de commande du même emplacement que le fichier package.json.

Remarque: Recommandé pour installer "Open Command Line", une extension qui peut être ajoutée à Visual Studio:

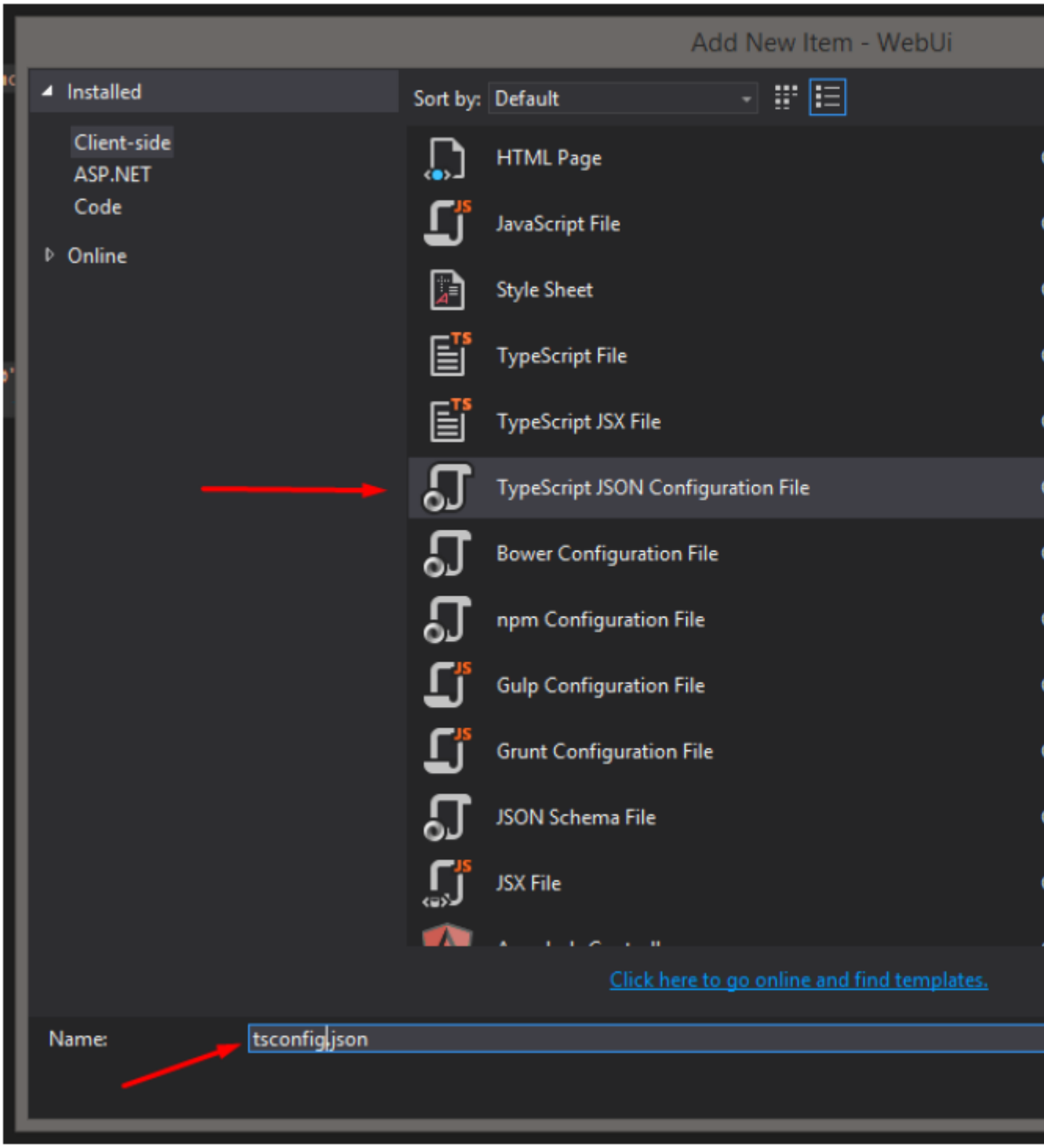


5. Ajouter un texte dactylographié:

   - Créez un dossier TsScript dans le projet WebUi, juste pour l'organisation (les scripts TypeScripts ne seront pas envoyés au navigateur, ils seront transposés dans un fichier JS normal, et ce fichier JS sera celui qui va au générateur wwwroot en utilisant gulp, ceci sera expliqué plus tard):

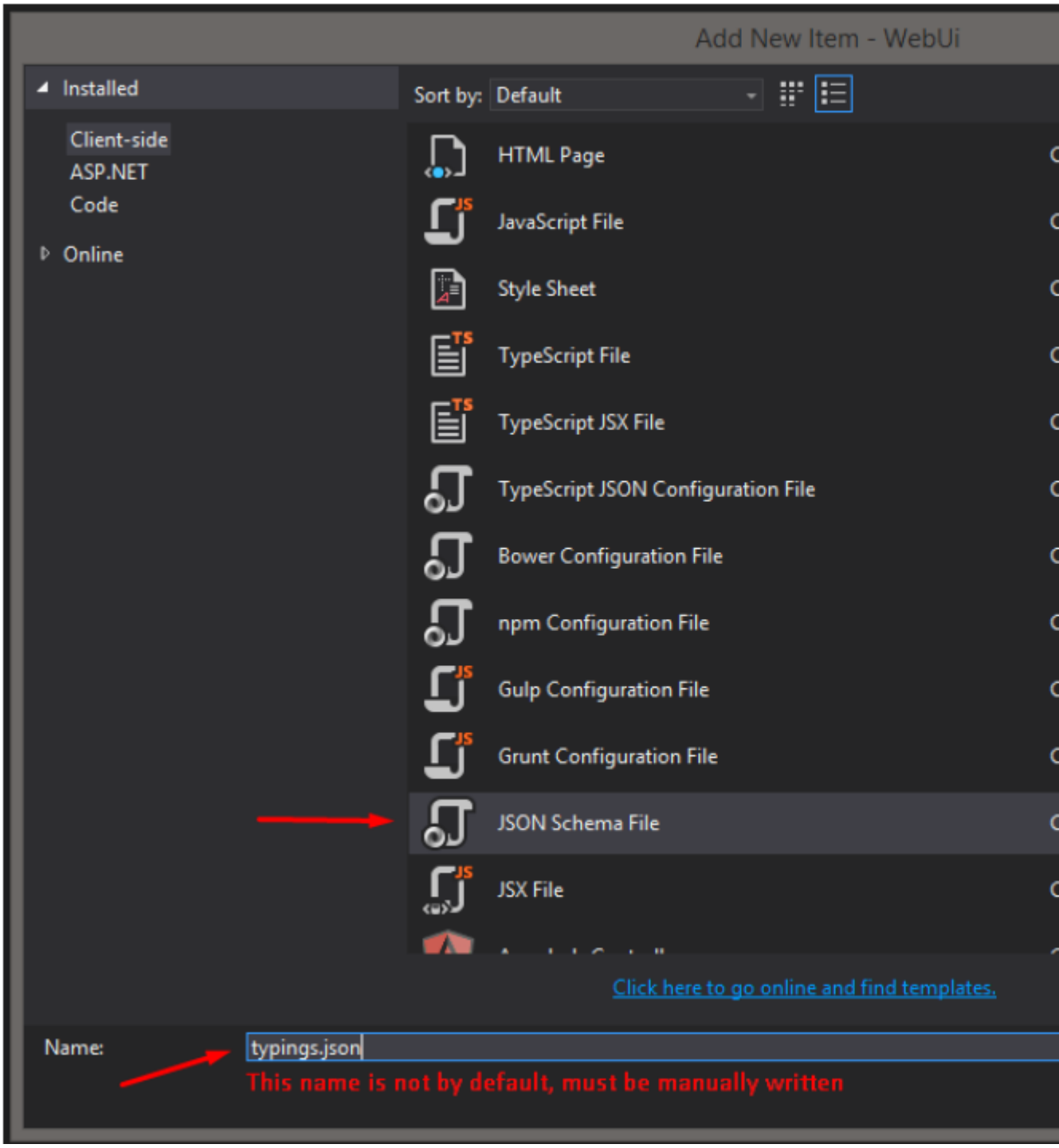


- Dans ce dossier, ajoutez "Fichier de configuration JSON TypeScript" (tsconfig.json):

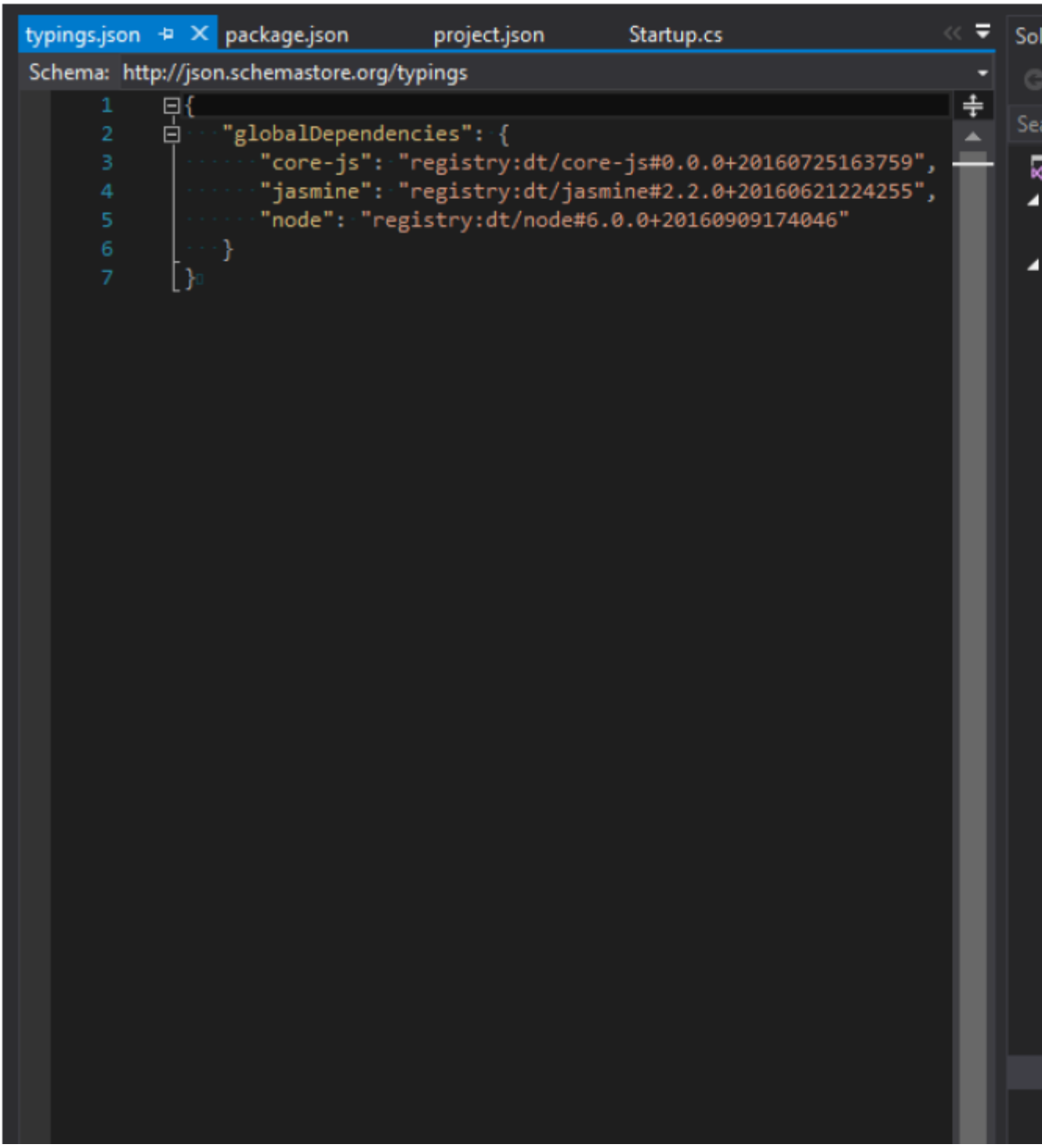


Et ajoutez le code suivant:

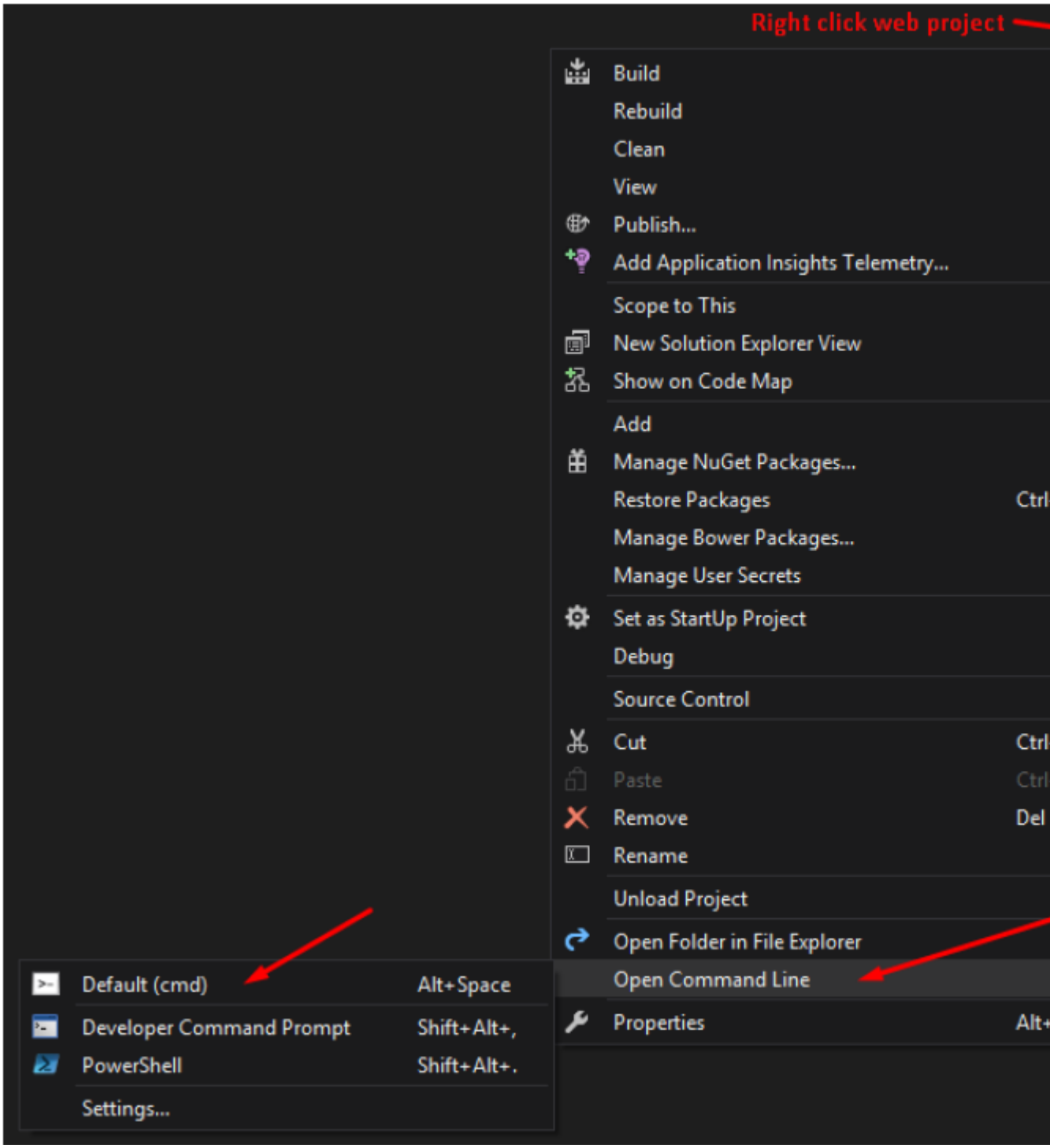
- Dans la racine du projet WebUi, ajoutez un nouveau fichier appelé typings.json:

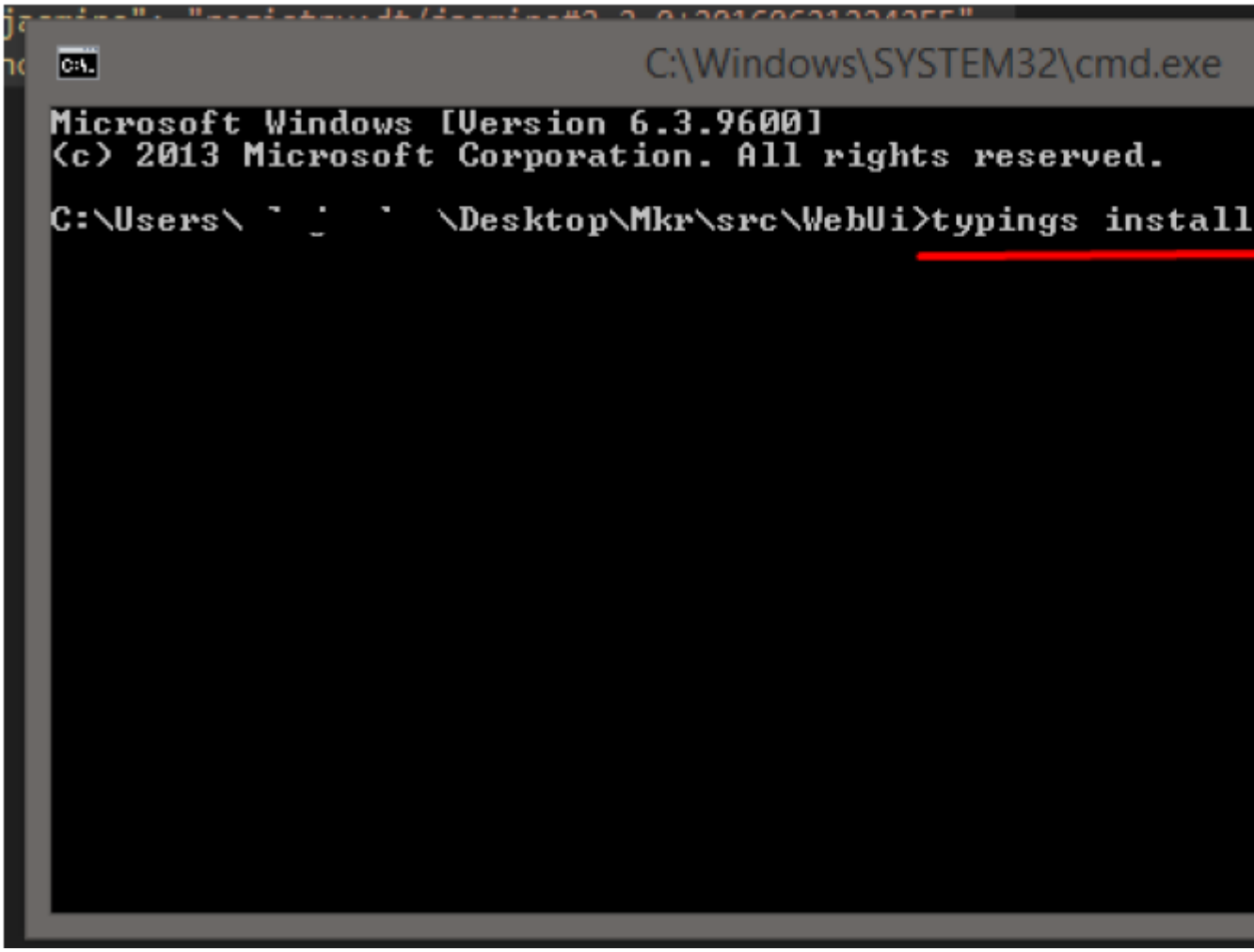


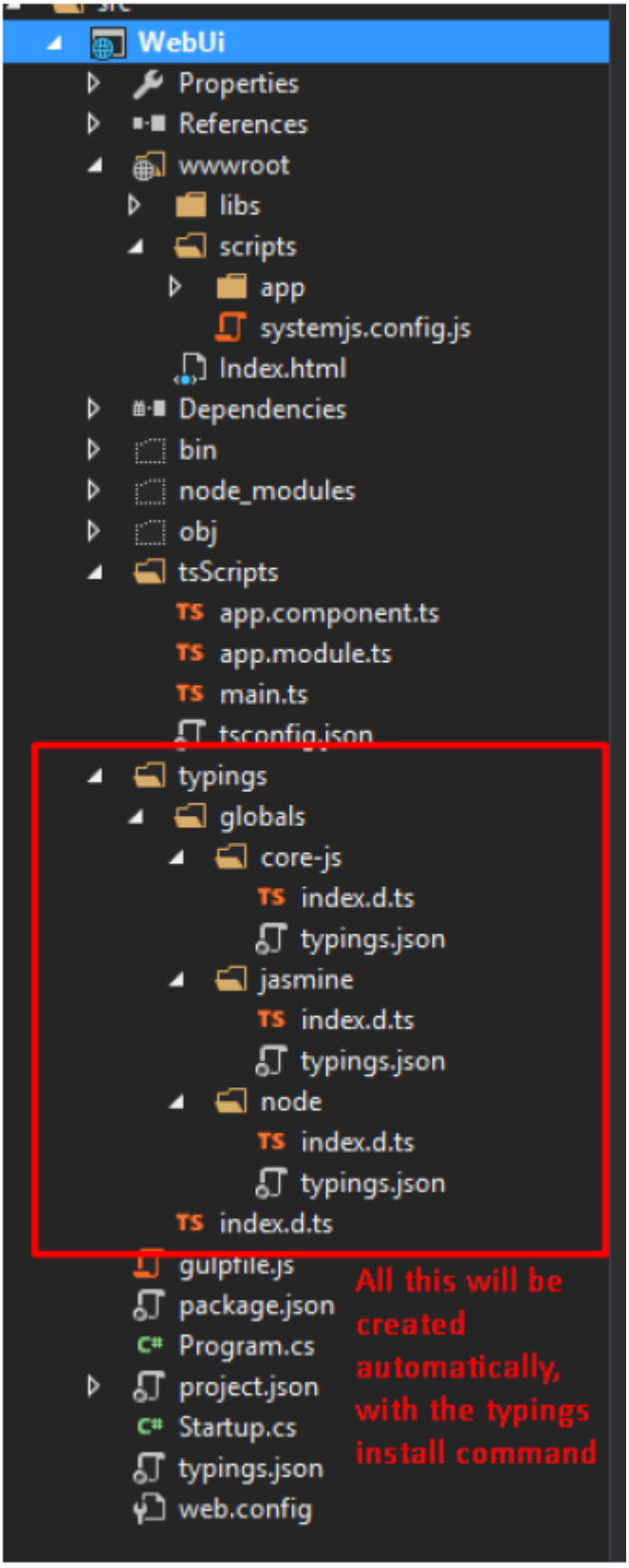
Et ajoutez le code suivant:



- Dans la racine du projet Web, ouvrez une ligne de commande et exécutez «typings install», cela créera un dossier de saisie (Cela nécessite «Ouvrir une ligne de commande» expliquée comme une étape facultative dans la note de l'étape 4, chiffre iii).

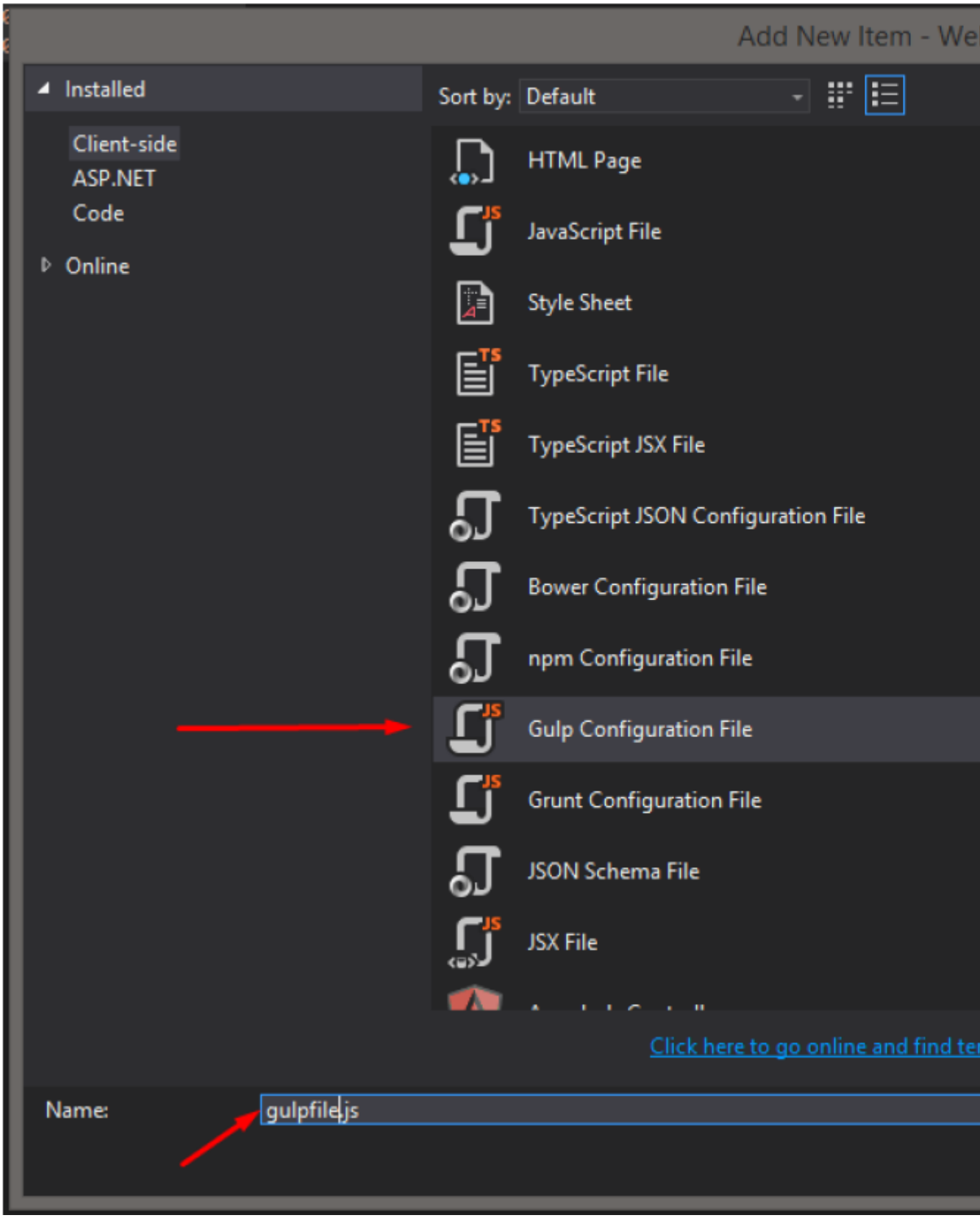

Right click web project
Build
Rebuild
Clean
View
Publish...
Add Application Insights Telemetry...
Scope to This
New Solution Explorer View
Show on Code Map
Add
Manage NuGet Packages...
Restore Packages
Manage Bower Packages...
Manage User Secrets
Set as StartUp Project
Debug
Source Control
Cut
Paste
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

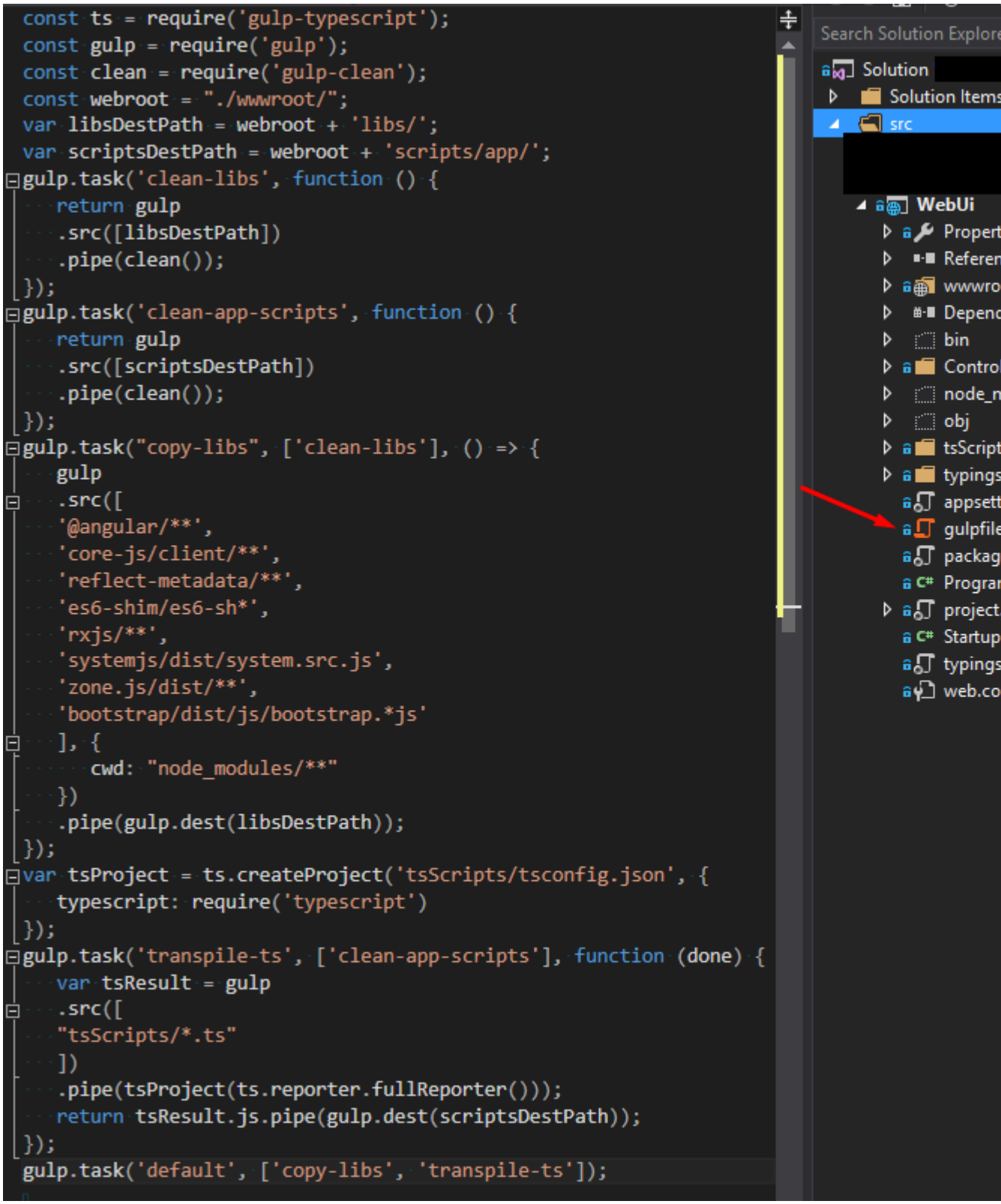


6. Ajoutez gulp pour déplacer les fichiers:

    - Ajoutez "Fichier de configuration Gulp" (gulpfile.js) à la racine du projet Web:

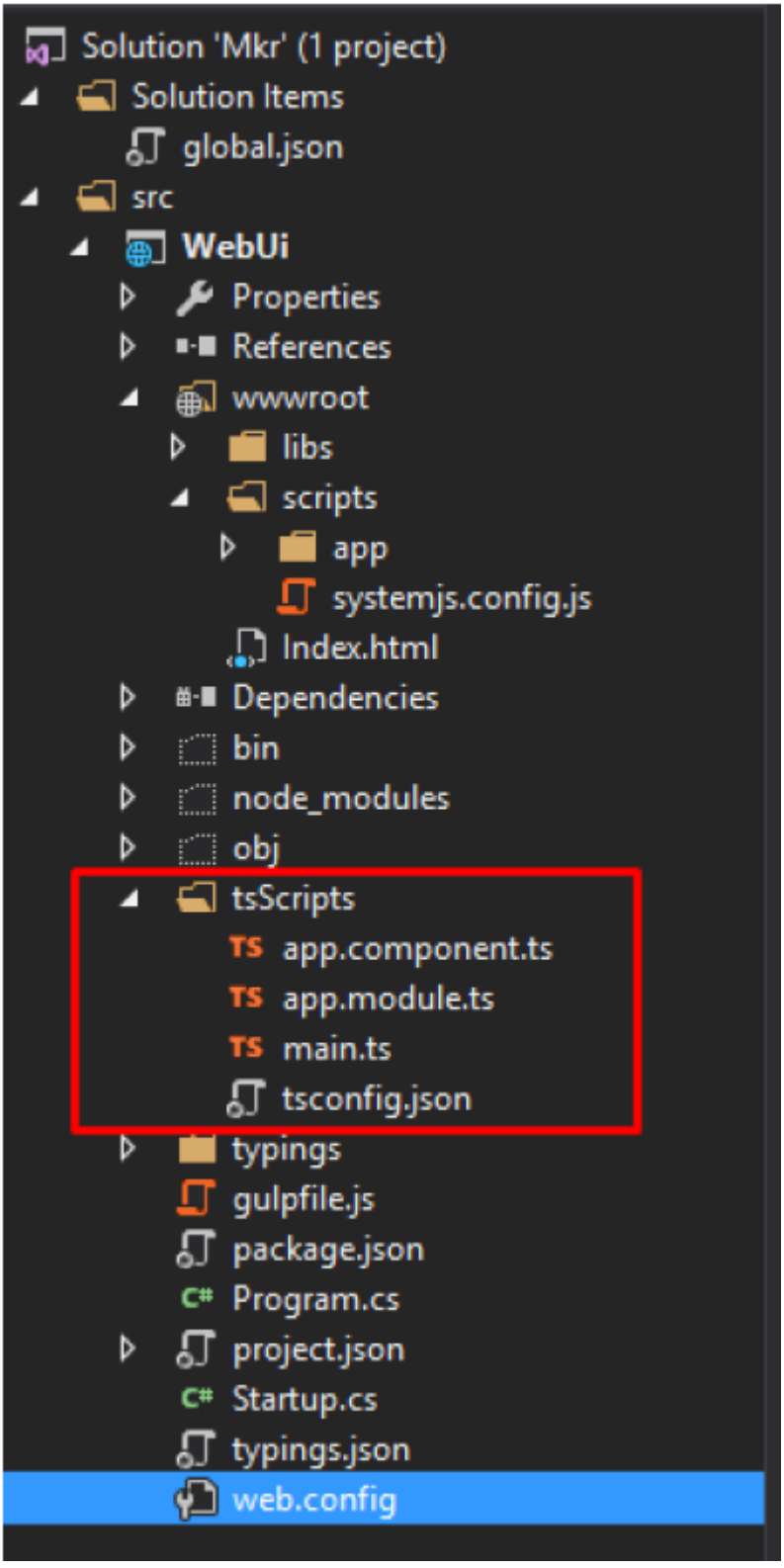


- Ajouter un code:



```
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. Ajoutez les fichiers d'amorçage Angular 2 dans le dossier «tsScripts»:



app.component.ts

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

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

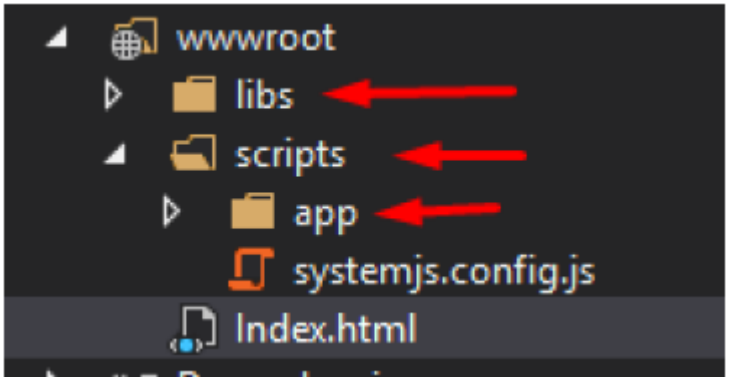
app.module.ts

```
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 { }
```

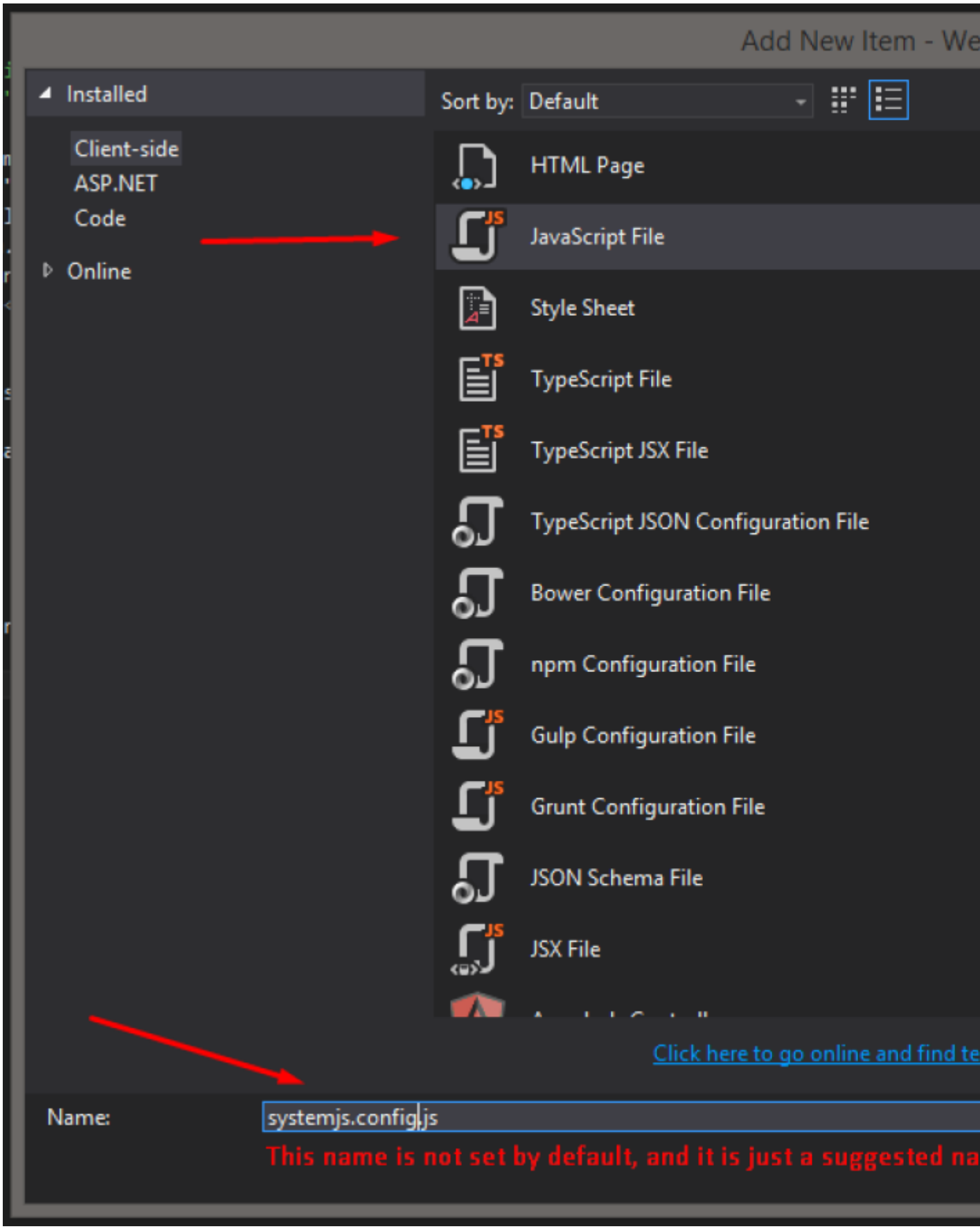
main.ts

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

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

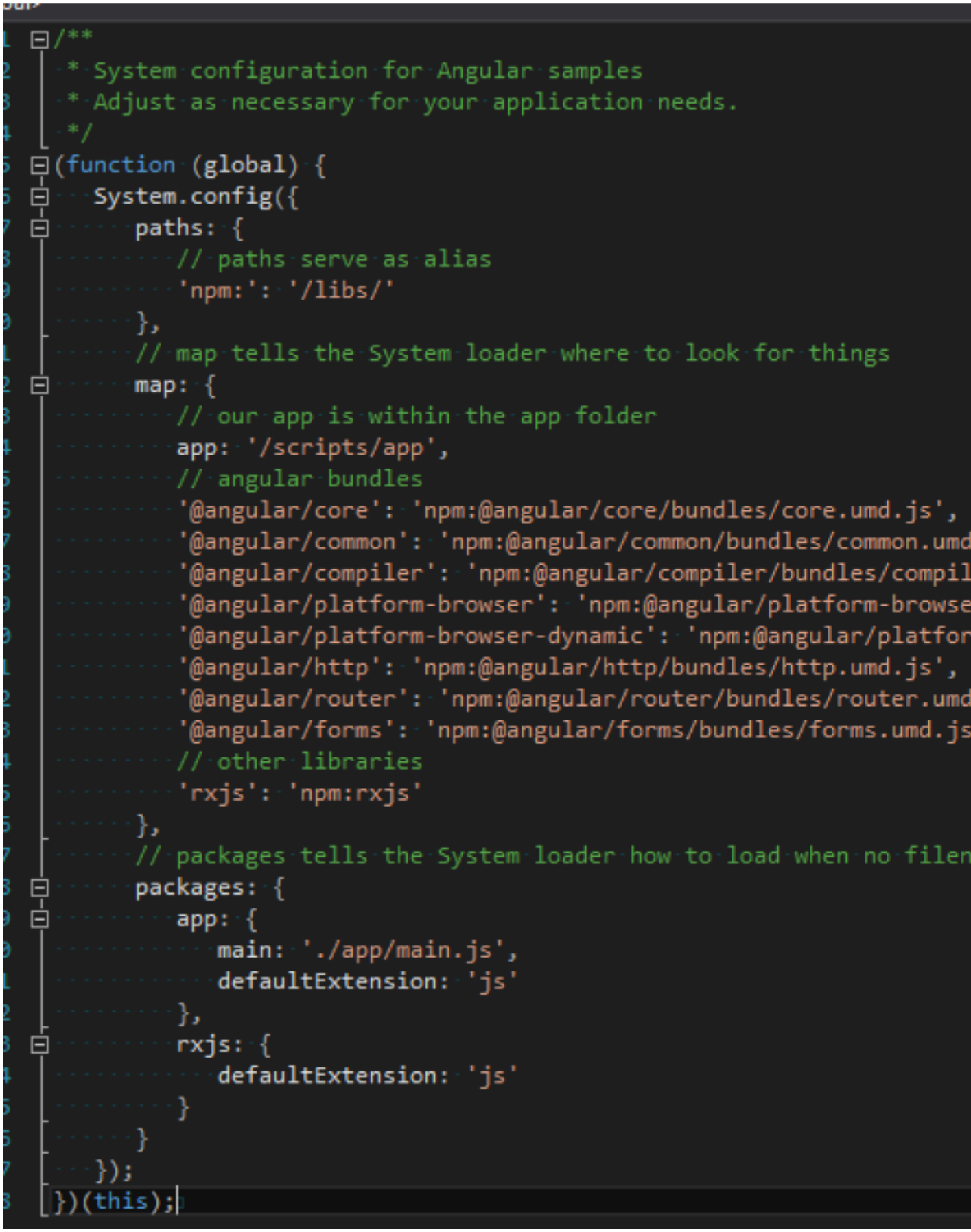
8. Dans wwwroot, créez la structure de fichier suivante:



9. Dans le dossier scripts (mais en dehors de l'application), ajoutez le fichier systemjs.config.js:



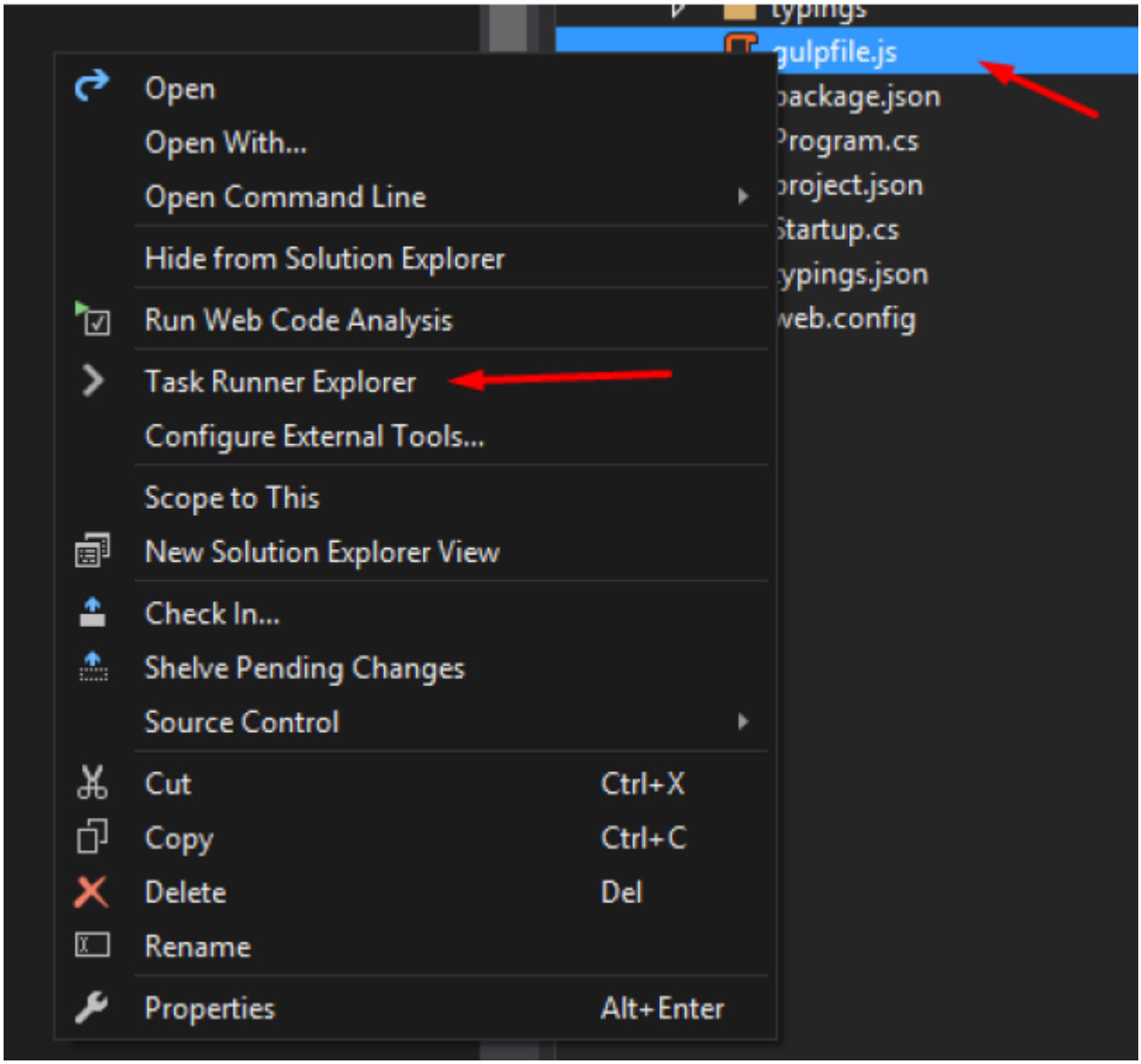
Et ajoutez le code suivant:



```
/**
 * 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);
```

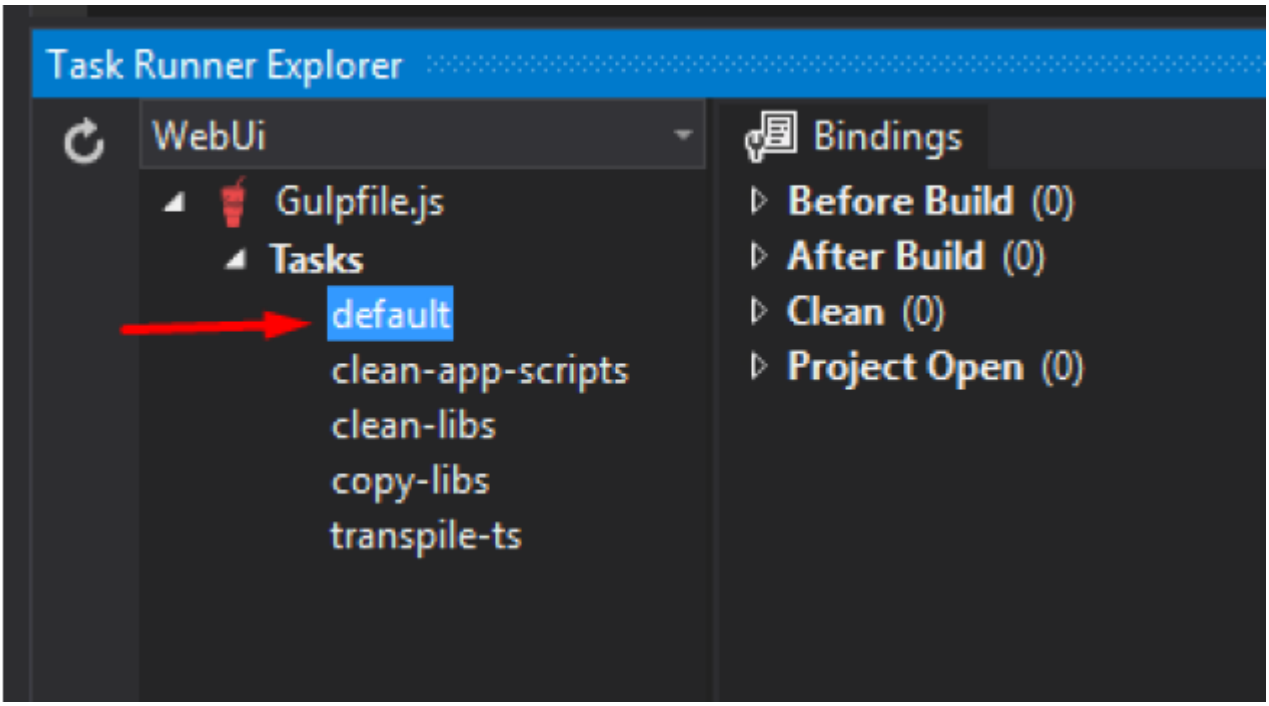
10. Exécutez la tâche Gulp pour générer les scripts dans wwwroot.

    - Faites un clic droit sur gulpfile.js
    - Task Runner Explorer

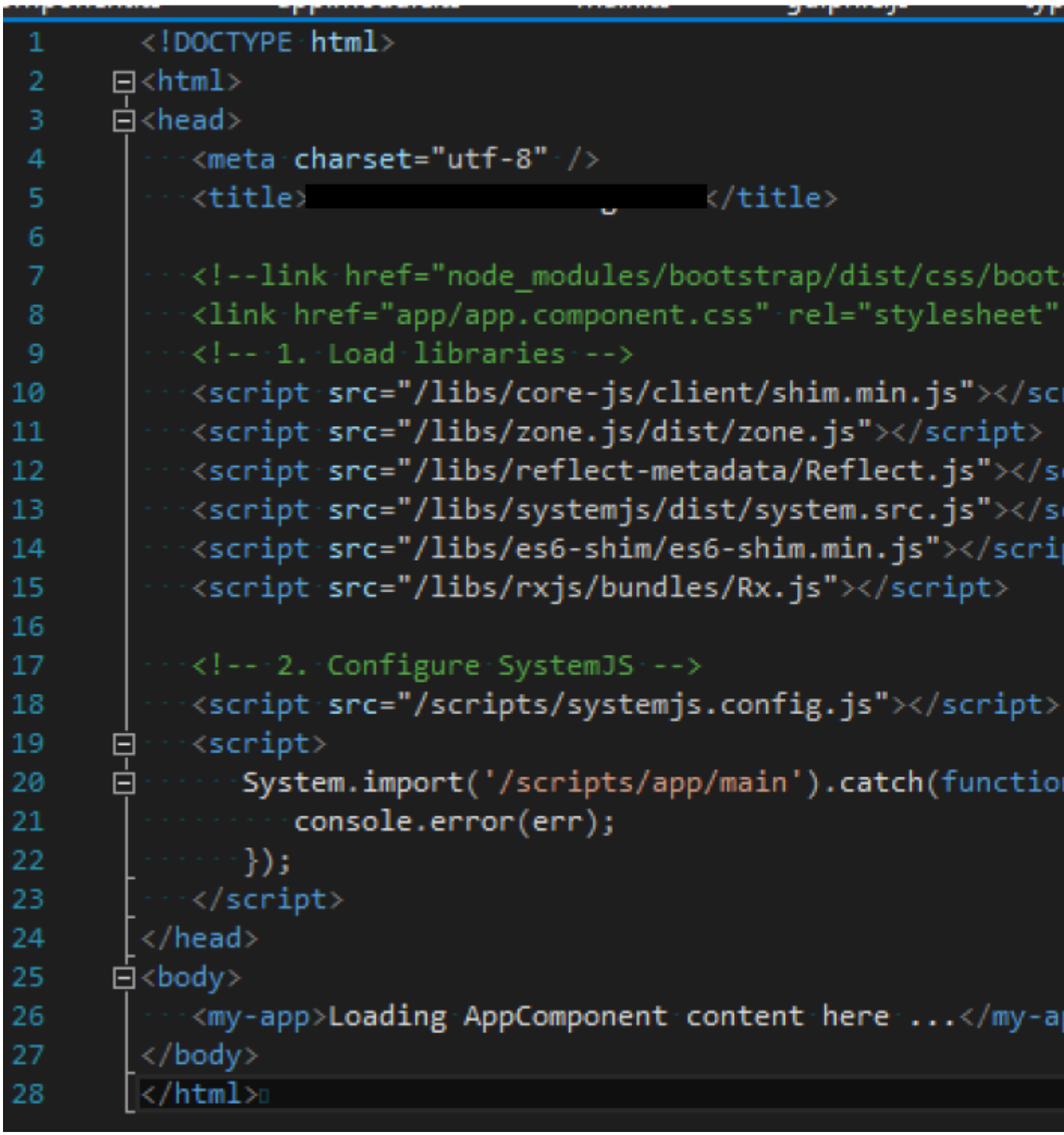


je. Si les tâches ne sont pas chargées ("Echec du chargement. Voir Fenêtre de sortie") Allez dans la fenêtre de sortie et examinez les erreurs, la plupart du temps des erreurs de syntaxe dans le fichier gulp.

- Clic droit sur la tâche "par défaut" et "Exécuter" (cela prendra du temps, et les messages de confirmation ne sont pas très précis, cela montre que c'est fini mais le processus est toujours en cours, gardez cela à l'esprit):



11. Modifier Index.html comme:



12. Maintenant courez et appréciez.

    Remarques:

    - S'il existe des erreurs de compilation avec typecript, par exemple "TypeScript Virtual Project", cela indique que la version de TypeScript pour Visual Studio n'est pas mise à jour en fonction de la version sélectionnée dans le fichier "package.json". : https://www.microsoft.com/en-us/download/details.aspx?id=48593

Les références:

- Le cours "Angular 2: Getting Started" de Deborah Kurata à Pluralsight:

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

- Angular 2 Documentation officielle:

  https://angular.io/

- Articles de 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/

Erreurs attendues lors de la génération de composants Angular 2 dans le projet .NET Core (version 0.8.3)

Lors de la génération de nouveaux composants Angular 2 dans un projet .NET Core, vous pouvez rencontrer les erreurs suivantes (à partir de la version 0.8.3):

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

OU

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

[SOLUTION]

1. Renommez app.module.client.ts à app.client.module.ts
2. Ouvrez app.client.module.ts: placez la déclaration avec 3 points "..." et placez la déclaration entre parenthèses.

   Par exemple: `[...sharedConfig.declarations, <MyComponent>]`

3. Ouvrez boot-client.ts: mettez à jour votre importation pour utiliser la nouvelle référence app.client.module.

   Par exemple: `import { AppModule } from './app/app.client.module';`

4. Maintenant, essayez de générer le nouveau composant: `ng g component my-component`

[EXPLICATION]

CLI angulaire recherche un fichier nommé app.module.ts dans votre projet et tente de trouver une référence à la propriété declarations pour importer le composant. Cela devrait être un tableau (comme le partage SharedConfig.declarations), mais les modifications ne sont pas appliquées

[SOURCES]

- https://github.com/angular/angular-cli/issues/2962
- https://www.udemy.com/aspnet-core-angular/learn/v4/t/lecture/6848548 (Bryan Garzon, collaborateur de la section 3.33)

Lire Angular2 et .Net Core en ligne: https://riptutorial.com/fr/asp-net-core/topic/9352/angular2-et--net-core

Chapitre 4: ASP.NET Core - Journal à la fois de la demande et de la réponse à l'aide de middleware

Introduction

Pendant un certain temps, j'ai cherché le meilleur moyen de consigner les demandes et les réponses dans un noyau ASP.Net. Je développais des services et l'une des exigences était d'enregistrer la demande avec sa réponse dans un enregistrement de la base de données. Tant de sujets là-bas mais aucun n'a fonctionné pour moi. c'est soit pour la demande seulement, la réponse seulement ou tout simplement n'a pas fonctionné. Lorsque j'ai pu enfin le faire, et au cours de mon projet, il a évolué pour mieux gérer les erreurs et enregistrer les exceptions, j'ai donc pensé au partage.

Remarques

certains des sujets qui m'ont aidé:

- 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/
- Comment connecter le corps de réponse HTTP dans 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;
                    }
                }
            }
        }
    }
```

Lire ASP.NET Core - Journal à la fois de la demande et de la réponse à l'aide de middleware en

ligne: https://riptutorial.com/fr/asp-net-core/topic/9510/asp-net-core---journal-a-la-fois-de-la-demande-et-de-la-reponse-a-l-aide-de-middleware

Chapitre 5: Autorisation

Examples

Autorisation simple

L'autorisation dans le noyau asp.net est simplement `AuthorizeAttribute`

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

    public IActionResult Post()
    {
    }
}
```

Cela permettra uniquement à un utilisateur connecté d'accéder à ces actions.

ou utilisez ce qui suit pour limiter une seule action

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

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

Si vous souhaitez autoriser tous les utilisateurs à accéder à l'une des actions, vous pouvez utiliser `AllowAnonymousAttribute`

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

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

Maintenant, n'importe quel utilisateur peut accéder à `Post` . `AllowAnonymous` toujours une priorité à

autoriser, donc si un contrôleur est défini sur `AllowAnonymous` toutes ses actions sont publiques, qu'elles aient ou non un `AuthorizeAttribute` .

Il existe une option permettant de configurer tous les contrôleurs pour exiger des requêtes autorisées -

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

Cela se fait en ajoutant une stratégie d'autorisation par défaut à chaque contrôleur - tout attribut `Authorize` / `AllowAnonymous` sur un contrôleur / action remplacera ces paramètres.

Lire Autorisation en ligne: https://riptutorial.com/fr/asp-net-core/topic/6914/autorisation

Chapitre 6: Configuration

Introduction

Le noyau Asp.net fournit des abstractions de configuration. Ils vous permettent de charger les paramètres de configuration à partir de différentes sources et de créer un modèle de configuration final qui peut ensuite être utilisé par votre application.

Syntaxe

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

Examples

Accès à la configuration en utilisant l'injection de dépendance

L'approche recommandée serait d'éviter de le faire et d'utiliser plutôt `IOptions<TOptions>` et `IServiceCollection.Configure<TOptions>` .

Cela dit, cela reste assez simple pour rendre `IConfigurationRoot` disponible à l'échelle des applications.

Dans le constructeur Startup.cs, vous devez avoir le code suivant pour générer la configuration,

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

Ici, la `Configuration` est une instance de `IConfigurationRoot` , et ajoute cette instance en tant que Singleton à la collection de services dans la méthode ConfigureServices, Startup.cs,

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

Par exemple, vous pouvez maintenant accéder à la configuration dans un contrôleur / service

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

Commencer

Dans cet exemple, nous allons décrire ce qui se passe lorsque vous échafaudez un nouveau projet.

Tout d'abord, les dépendances suivantes seront ajoutées à votre projet (actuellement fichier `project.json`):

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

Il va également créer un constructeur dans votre fichier `Startup.cs` qui sera chargé de construire la configuration à l'aide d'Api `ConfigurationBuilder` fluent:

1. Il commence par créer un nouveau `ConfigurationBuilder` .
2. Il définit ensuite un chemin de base qui sera utilisé pour calculer le chemin absolu des autres fichiers
3. Il ajoute un `appsettings.json` facultatif au constructeur de configuration et surveille ses modifications
4. Il ajoute un fichier de configuration `appsettings.environementName.json` lié à l'environnement facultatif
5. Il ajoute ensuite des variables d'environnement.

```
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();
}
```

Si un même paramètre est défini dans plusieurs sources, la dernière source ajoutée gagnera et sa valeur sera sélectionnée.

La configuration peut ensuite être consommée à l'aide de la propriété de l'indexeur. Les deux points `:` caractère servent un délimiteur de chemin.

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

Cela recherchera une valeur de configuration `ConnectionString` dans une section `AzureLogger` .

Travailler avec des variables d'environnement

Vous pouvez `.AddEnvironmentVariables()` configuration à partir des variables d'environnement en appelant `.AddEnvironmentVariables()` sur `ConfigurationBuilder` .

Il chargera les variables d'environnement préfixées par `APPSETTING_` Il utilisera alors deux points `:` comme séparateur de chemin de clé.

Cela signifie que: les paramètres d'environnement suivants:

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

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

Sera l'équivalent de ce json:

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

** Notez que Azure Service transmettra les paramètres en tant que variables d'environnement. Le préfixe sera défini pour vous de manière transparente. Donc, pour faire la même chose dans Azure, il suffit de définir deux paramètres d'application dans la lame AppSettings:

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

Modèle d'option et configuration

Lorsque vous traitez de grands ensembles de valeurs de configuration, il peut être difficile de les charger en un seul.

Modèle d'option qui vient avec asp.net offre un moyen pratique pour mapper une `section` à une dotnet `poco` : Par exemple, on peut hydrater `StorageOptions` directement à partir d' une section de configuration b ajoutant `Microsoft.Extensions.Options.ConfigurationExtensions` package et en appelant le `Configure<TOptions>(IConfiguration config)` méthode d'extension.

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

Source de configuration en mémoire

Vous pouvez également créer une configuration à partir d'un objet en mémoire tel qu'un `Dictionary<string,string>`

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

Cela peut s'avérer utile dans les scénarios de test d'intégration / d'unité.

Lire Configuration en ligne: https://riptutorial.com/fr/asp-net-core/topic/8660/configuration

Chapitre 7: Configuration de plusieurs environnements

Examples

Avoir des appsettings par environnement

Pour chaque environnement, vous devez créer un fichier appsettings distinct. {EnvironmentName} .json:

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

Ensuite, ouvrez le fichier project.json et incluez-les dans "include" dans la section "publishOptions". Cela répertorie tous les fichiers et dossiers qui seront inclus lorsque vous publiez:

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

La dernière étape Dans votre classe de démarrage, ajoutez:

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

dans constructeur où vous définissez les sources de configuration:

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

Obtenir / Vérifier le nom de l'environnement à partir du code

Tout ce dont vous avez besoin est une variable de type `IHostingEnvironment` :

- obtenir le nom de l'environnement:

  ```
  env.EnvironmentName
  ```

- Prédéfini `Development` , `Staging` en `Production Staging` , `Production` des environnements est la meilleure façon d'utiliser des méthodes d'extension de HostingEnvironmentExtensions classe

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

- ignorez correctement la casse (une autre méthode d'extension de HostingEnvironmentExtensions :

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

- variante sensible à la casse:

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

Configuration de plusieurs environnements

Cet exemple montre comment configurer plusieurs environnements avec une configuration d'injection de dépendances différente et des middlewares distincts dans une classe de `Startup` .

En plus des méthodes `public void Configure(IApplicationBuilder app)` et `public void ConfigureServices(IServiceCollection services)` , on peut utiliser `Configure{EnvironmentName}` et `Configure{EnvironmentName}Services` pour avoir une configuration dépendante de l'environnement.

L'utilisation de ce modèle évite de mettre beaucoup de logique `if/else` une seule méthode / classe de `Startup` et de la garder propre et séparée.

```
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) { }
}
```

Si les services `Configure{Environmentname}` ou `Configure{Environmentname}Services` ne sont pas trouvés, ils `ConfigureServices` respectivement `Configure` ou `ConfigureServices` .

La même sémantique s'applique également à la classe `Startup` . `StartupProduction` sera utilisé lorsque la `ASPNETCORE_ENVIRONMENT` variable est définie sur la `Production` et revenir à `Startup` quand il est `Stagging` ou le `Development`

Un exemple complet:

```
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();
    }
}
```

Afficher le contenu spécifique de l'environnement dans la vue

Il se peut que vous deviez afficher du contenu dans une vue spécifique à un environnement uniquement. Pour atteindre cet objectif, vous pouvez utiliser l' assistant de balise 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'assistant de balise Environment restitue uniquement son contenu si l'environnement actuel correspond à l'un des environnements spécifiés à l'aide de l'attribut `names` .

Définir la variable d'environnement à partir de la ligne de commande

Pour définir l'environnement au `Development`

```
SET ASPNETCORE_ENVIRONMENT=Development
```

L'exécution d'une application Asp.Net Core se fera désormais dans l'environnement défini.

Remarque

1. Il ne devrait y avoir aucun espace avant et après le signe d'égalité `=` .
2. L'invite de commandes ne doit pas être fermée avant d'exécuter l'application car les paramètres ne sont pas conservés.

Définir la variable d'environnement à partir de PowerShell

Lorsque vous utilisez PowerShell, vous pouvez utiliser setx.exe pour définir les variables d'environnement de manière permanente.

1. Démarrer PowerShell
2. Tapez l'une des options suivantes:

   setx ASPNETCORE_ENVIRONMENT "développement"

   setx ASPNETCORE_ENVIRONMENT "mise en scène"

3. Redémarrer PowerShell

Utiliser ASPNETCORE_ENVIRONMENT depuis web.config

Si vous ne souhaitez pas utiliser ASPNETCORE_ENVIRONMENT à partir de variables d'environnement et l'utiliser depuis web.config de votre application, modifiez le fichier web.config comme ceci -

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

Lire Configuration de plusieurs environnements en ligne: https://riptutorial.com/fr/asp-net-core/topic/2292/configuration-de-plusieurs-environnements

Chapitre 8: Demandes d'origine croisée (CORS)

Remarques

La sécurité du navigateur empêche une page Web de faire des requêtes AJAX vers un autre domaine. Cette restriction s'appelle la politique de même origine et empêche un site malveillant de lire des données sensibles d'un autre site. Cependant, il peut arriver que vous souhaitiez laisser d'autres sites créer des requêtes d'origine croisée vers votre application Web.

Cross Origin Resource Sharing (CORS) est un standard W3C qui permet à un serveur d'assouplir la politique de même origine. En utilisant CORS, un serveur peut autoriser explicitement certaines requêtes d'origine croisée tout en en rejetant d'autres. CORS est plus sûr et plus flexible que les techniques antérieures telles que JSONP.

Examples

Activer CORS pour toutes les demandes

Utilisez la méthode d'extension `UseCors()` sur `IApplicationBuilder` dans la méthode `Configure` pour appliquer la stratégie CORS à toutes les demandes.

```
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();
}
```

Activer la stratégie CORS pour des contrôleurs spécifiques

Pour activer une stratégie CORS pour des contrôleurs spécifiques, vous devez créer la stratégie dans l'extension `AddCors` dans la méthode `ConfigureServices` :

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

Cela vous permet d'appliquer la stratégie à un contrôleur:

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

Des politiques CORS plus sophistiquées

Le générateur de règles vous permet de créer des stratégies sophistiquées.

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

Cette stratégie n'autorise que les origines `http://localhost:5000` et `http://myproductionapp.com` avec uniquement les méthodes `GET` , `POST` et `HEAD` et accepte uniquement les `content-type` têtes HTTP `accept` , `content-type` et `origin` . La méthode `SetPreflightMaxAge` oblige les navigateurs à mettre en cache le résultat de la demande de contrôle en amont (`OPTIONS`) à mettre en cache pour la durée spécifiée.

Activer la stratégie CORS pour tous les contrôleurs

Pour activer une stratégie CORS sur tous vos contrôleurs MVC, vous devez créer la stratégie dans l'extension AddCors dans la méthode ConfigureServices, puis définir la stratégie sur `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");
}
```

Cette stratégie CORS peut être remplacée sur une base de contrôleur ou d'action, mais cela peut définir la valeur par défaut pour toute l'application.

Lire Demandes d'origine croisée (CORS) en ligne: https://riptutorial.com/fr/asp-net-core/topic/2556/demandes-d-origine-croisee--cors-

Chapitre 9: Des modèles

Examples

Validation de modèles avec attributions de validation

Les attributs de validation peuvent être utilisés pour configurer facilement la validation du modèle.

```
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; }
}
```

Les attributs intégrés sont:

- `[CreditCard]` : Valide que la propriété a un format de carte de crédit.
- `[Compare]` : valide deux propriétés dans une correspondance de modèle.
- `[EmailAddress]` : valide la propriété a un format de courrier électronique.
- `[Phone]` : Valide que la propriété dispose d'un format téléphonique.
- `[Range]` : valide la valeur de la propriété dans la plage donnée.
- `[RegularExpression]` : Valide que les données correspondent à l'expression régulière spécifiée.
- `[Required]` : Rend une propriété requise.
- `[StringLength]` : `[StringLength]` qu'une propriété de chaîne a au maximum la longueur maximale indiquée.
- `[Url]` : valide la propriété a un format d'URL.

Validation du modèle avec un attribut personnalisé

Si les attributs intégrés ne sont pas suffisants pour valider les données de votre modèle, vous pouvez placer votre logique de validation dans une classe dérivée de ValidationAttribute. Dans cet exemple, seuls les nombres impairs sont des valeurs valides pour un membre du modèle.

Attribut de validation personnalisé

```
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 de modèle

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

Lire Des modèles en ligne: https://riptutorial.com/fr/asp-net-core/topic/4625/des-modeles

Chapitre 10: Enregistrement

Examples

Utiliser NLog Logger

NLog.Extensions.Logging est le fournisseur officiel NLog pour Microsoft dans .NET Core et ASP.NET Core. Ici et ici sont des instructions et des exemples respectivement.

Ajouter un enregistreur au contrôleur

Au lieu de demander un ILoggerFactory et de créer explicitement une instance d'ILogger, vous pouvez demander un ILogger (où T est la classe qui demande le consignateur).

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

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

Utilisation de Serilog dans l'application ASP.NET core 1.0

1) Dans project.json, ajoutez ci-dessous les dépendances-

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

2) Dans Startup.cs, ajoutez les lignes ci-dessous dans constructor

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

3) Dans la méthode Configure de la classe Startup

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

4) Dans Controller, créez une instance de ILogger comme ceci-

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

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

5) Enregistrement des échantillons ci-dessous

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

Lire Enregistrement en ligne: https://riptutorial.com/fr/asp-net-core/topic/1946/enregistrement

Chapitre 11: Envoi d'e-mails dans les applications .Net Core à l'aide de MailKit

Introduction

Actuellement, .Net Core n'inclut pas le support pour envoyer des emails comme `System.Net.Mail` de .Net. Le projet MailKit (disponible sur nuget) est une bibliothèque intéressante à cet effet.

Examples

Installation du paquet nuget

```
Install-Package MailKit
```

Implémentation simple pour l'envoi d'emails

```
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);
            }
        }
    }

}
```

Lire Envoi d'e-mails dans les applications .Net Core à l'aide de MailKit en ligne: https://riptutorial.com/fr/asp-net-core/topic/8831/envoi-d-e-mails-dans-les-applications--net-core-a-l-aide-de-mailkit

Chapitre 12: Injection de dépendance

Introduction

Le noyau Aspnet est construit avec l'injection de dépendance comme l'un de ses concepts clés. Il introduit une abstraction de conteneur conforme afin que vous puissiez remplacer celle intégrée par un conteneur tiers de votre choix.

Syntaxe

- `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>()`

Remarques

Pour utiliser des variantes génériques de méthodes `IServiceProvider` , vous devez inclure l'espace de noms suivant:

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

Examples

Enregistrez et résolvez manuellement

La méthode privilégiée pour décrire les dépendances consiste à utiliser l'injection de constructeur qui suit le principe de dépendances explicites :

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();
        }
    }
}
```

Enregistrer les dépendances

Le conteneur intégré est livré avec un ensemble de fonctionnalités intégrées:

Contrôle à vie

```
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** : créé chaque fois qu'il est résolu
- **AddScoped** : créé une fois par requête
- **AddSingleton** : récemment créé une fois par application
- **AddSingleton (instance)** : fournit une instance précédemment créée par application

Dépendances énumérables

Il est également possible d'enregistrer des dépendances énumérables:

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

Vous pouvez ensuite les consommer comme suit:

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

Dépendances génériques

Vous pouvez également enregistrer des dépendances génériques:

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

Et puis consommez-le comme suit:

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

Récupérer des dépendances sur un contrôleur

Une fois enregistrée, une dépendance peut être extraite en ajoutant des paramètres au constructeur Controller.

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

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

Injection d'une dépendance dans une action de contrôleur

Une fonction intégrée moins connue est l'injection d'action de contrôleur à l'aide de `FromServicesAttribute` .

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

Une note importante est que le `[FromServices]` **ne peut pas** être utilisé comme mécanisme général "Injection de propriété" ou "Injection de méthode"! Il ne peut être utilisé que sur les paramètres de méthode d'une action de contrôleur ou d'un constructeur de contrôleur (dans le constructeur, il est obsolète, car le système ASP.NET Core DI utilise déjà l'injection de constructeur et aucun marqueur supplémentaire n'est requis).

Il ne peut être utilisé nulle part en dehors des contrôleurs, action du contrôleur . Il est également très spécifique à ASP.NET Core MVC et réside dans l'assembly `Microsoft.AspNetCore.Mvc.Core` .

Citation originale du problème ASP.NET Core MVC GitHub (Limiter [FromServices] à appliquer uniquement aux paramètres) concernant cet attribut:

> @rynowak:
>
> > @Eilon:
> >
> > Le problème avec les propriétés est qu'il apparaît à beaucoup de personnes qu'il peut être appliqué à n'importe quelle propriété de n'importe quel objet.
>
> D'accord, nous avons eu un certain nombre de problèmes publiés par les utilisateurs avec la confusion sur la façon dont cette fonctionnalité doit être utilisée. Il y a vraiment eu pas mal de commentaires à la fois: "[FromServices] est bizarre et je ne l'aime pas" et "[FromServices] m'a confondu". On se croirait dans un piège et l'équipe répondrait encore à des questions dans des années.
>
> Nous pensons que le scénario le plus précieux pour [FromServices] est sur le paramètre de méthode d'une action pour un service dont vous avez besoin uniquement à cet endroit.
>
> / cc @ danroth27 - modifications de documents
>
> Pour ceux qui aiment le [FromServices] actuel, je vous recommande fortement d’examiner un système de DI qui peut faire une injection de propriété (Autofac, par exemple).

Remarques:

- **Tous les** services enregistrés avec le système d'injection de dépendance .NET Core peuvent être injectés dans l'action d'un contrôleur à l'aide de l'attribut `[FromServices]` .
- Le cas le plus pertinent est celui où vous avez besoin d'un service uniquement en une seule méthode et que vous ne voulez pas encombrer le constructeur de votre contrôleur avec une autre dépendance, qui ne sera utilisée qu'une seule fois.
- Ne peut pas être utilisé en dehors de ASP.NET Core MVC (c'est-à-dire des applications de console .NET Framework ou .NET Core pures), car il réside dans l'assembly `Microsoft.AspNetCore.Mvc.Core` .
- Pour l'injection de propriété ou de méthode, vous devez utiliser l'un des conteneurs IoC tiers disponibles (Autofac, Unity, etc.).

Le modèle Options / Options d'injection dans les services

Avec ASP.NET Core, l’équipe Microsoft a également introduit le modèle Options, qui permet d’avoir des options typées robustes et une fois configuré la possibilité d’injecter les options dans vos services.

Nous commençons par une classe typée forte, qui conservera notre configuration.

```
public class MySettings
{
```

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

Et une entrée dans le `appsettings.json` .

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

Ensuite, nous l'initialisons dans la classe Startup. Il y a deux façons de faire ça

1. Chargez-le directement depuis la `appsettings.json` "mysettings" de appsettings.json

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

2. Le faire manuellement

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

   Chaque niveau hiérarchique du `appsettings.json` est séparé par un `:` . Puisque `value2` est une propriété de l'objet `mysettings` , on y accède via `mysettings:value2` .

Enfin, nous pouvons injecter les options dans nos services, en utilisant l' `IOptions<T>`

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

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

Remarques

Si `IOptions<T>` n'est pas configuré au démarrage, l'injection d' `IOptions<T>` injectera l'instance par défaut de la classe `T`

Utilisation de services étendus au démarrage de l'application / Amorçage de la base de données

Résoudre les services de portée au démarrage de l'application peut être difficile, car il n'y a pas de demande et donc pas de service de portée.

Résoudre un service de portée au démarrage de l'application via `app.ApplicationServices.GetService<AppDbContext>()` peut entraîner des problèmes, car il sera créé dans la portée du conteneur global, ce qui en fait un singleton avec la durée de vie de l'application, ce qui peut entraîner à des exceptions comme `Cannot access a disposed object in ASP.NET Core when injecting DbContext` .

Le modèle suivant résout le problème en créant d'abord une nouvelle étendue, puis en résolvant les services qui lui sont associés, puis, une fois le travail terminé, en éliminant le conteneur de portée.

```
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);
        }
    }
}
```

Il s'agit d'une manière semi-officielle de l'équipe principale d'Entity Framework d'ensemencer les données lors du démarrage de l'application et est reflétée dans l' exemple d' application MusicStore .

Résoudre les contrôleurs, ViewComponents et TagHelpers via l'injection de dépendances

Par défaut, les contrôleurs, ViewComponents et TagHelpers ne sont pas enregistrés et résolus via le conteneur d'injection de dépendance. Cela se traduit par l'incapacité de faire, c'est-à-dire l'injection de propriété lors de l'utilisation d'un conteneur Inversion of Control (IoC) tiers tel qu'AutoFac.

Pour que les types ASP.NET Core MVC résolvent ces types via IoC, il faut ajouter les enregistrements suivants dans `Startup.cs` (extrait de l'exemple officiel ControllersFromService sur 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>();
}
```

Exemple d'injection de dépendance simple (sans Startup.cs)

Cela vous montre comment utiliser le package nuget Microsoft.Extensions.DependencyInjection sans utiliser `WebHostBuilder` partir de `WebHostBuilder` (par exemple lorsque vous souhaitez créer quelque chose d'autre qu'une application Web):

```
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;
    }
}
```

Fonctionnement interne de Microsoft.Extensions.DependencyInjection

IServiceCollection

Pour commencer à créer un conteneur IOC avec le package DI nuget de Microsoft, commencez par créer un `IServiceCollection` . Vous pouvez utiliser la collection déjà fournie: `ServiceCollection` :

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

Cet `IServiceCollection` n'est rien d'autre qu'une implémentation de: `IList<ServiceDescriptor>, ICollection<ServiceDescriptor>, IEnumerable<ServiceDescriptor>, IEnumerable`

Toutes les méthodes suivantes ne sont que des méthodes d'extension permettant d'ajouter des instances `ServiceDescriptor` à la liste:

```
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

Le serviceprovider est celui qui "Compile" tous les enregistrements afin qu'ils puissent être utilisés rapidement, cela peut être fait avec `services.BuildServiceProvider()` qui est fondamentalement un code d'extension pour:

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

En arrière-plan, chaque `ServiceDescriptor` dans `IServiceCollection` est compilé dans une méthode de fabrique `Func<ServiceProvider, object>` où object est le type de retour et correspond à: l'instance créée du type Implementation, Singleton ou votre propre méthode de fabrique définie.

Ces enregistrements sont ajoutés à la `ServiceTable` qui est essentiellement un `ConcurrentDictionary` la clé étant `ServiceType` et la valeur de la méthode Factory définie ci-dessus.

Résultat

Nous avons maintenant un objet `ConcurrentDictionary<Type, Func<ServiceProvider, object>>` que nous pouvons utiliser simultanément pour demander de créer des services pour nous. Pour montrer un exemple de base de ce à quoi cela aurait pu ressembler.

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

Ce n'est pas comme ça que ça marche!

Ce `ConcurrentDictionary` est une propriété du `ServiceTable` qui est une propriété du `ServiceProvider`

Lire Injection de dépendance en ligne: https://riptutorial.com/fr/asp-net-core/topic/1949/injection-de-dependance

Chapitre 13: Injection de services dans des vues

Syntaxe

- `@inject<NameOfService><Identifier>`
- `@<Identifier>.Foo()`
- @inject <type> <nom>

Examples

La directive @inject

ASP.NET Core introduit le concept d'injection de dépendance dans Views via la directive `@inject` via la syntaxe suivante:

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

Exemple d'utilisation

L'ajout de cette directive dans votre View génère essentiellement une propriété du type donné en utilisant le nom donné dans votre View en utilisant une injection de dépendance appropriée, comme illustré dans l'exemple ci-dessous:

```
@inject YourWidgetServiceClass WidgetService

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

Configuration requise

Les services qui utilisent l'injection de dépendance doivent toujours être enregistrés dans la méthode `ConfigureServices()` du fichier `Startup.cs` et définis en conséquence:

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

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

Lire Injection de services dans des vues en ligne: https://riptutorial.com/fr/asp-net-core/topic/4284/injection-de-services-dans-des-vues

Chapitre 14: La gestion des erreurs

Examples

Rediriger vers une page d'erreur personnalisée

ASP.NET Core fournit le middleware de pages de codes d'état , qui prend en charge plusieurs méthodes d'extension différentes, mais nous sommes intéressants dans `UseStatusCodePages` et `UseStatusCodePagesWithRedirects` :

- UseStatusCodePages ajoute un middleware StatusCodePages avec les options données qui vérifient les réponses avec des codes de statut compris entre 400 et 599 sans corps. Exemple d'utilisation pour redirection:

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

  // your redirect logic

});
```

- UseStatusCodePagesWithRedirects ajoute un middleware StatusCodePages au pipeline. Spécifie que les réponses doivent être gérées en redirigeant avec le modèle d'URL d'emplacement donné. Cela peut inclure un espace réservé '{0}' pour le code d'état. PathBase sera ajouté aux URL commençant par '~', où toute autre URL sera utilisée telle quelle. Par exemple, les éléments suivants seront redirigés vers ~ / errors / <code_erreur> (par exemple ~ / errors / 403 pour erreur 403):

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

Gestion des exceptions globales dans ASP.NET Core

UseExceptionHandler peut être utilisé pour gérer les exceptions globalement. Vous pouvez obtenir tous les détails de l'objet d'exception, comme Stack Trace, Inner exception et autres. Et puis vous pouvez les montrer à l'écran. Vous pouvez facilement mettre en œuvre comme ça.

```
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);
      }
```

```
    });
  }
);
```

Vous devez mettre cela dans configure () du fichier startup.cs.

Lire La gestion des erreurs en ligne: https://riptutorial.com/fr/asp-net-core/topic/6581/la-gestion-des-erreurs

Chapitre 15: Le routage

Examples

Routage de base

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

Cela correspondra aux demandes pour `/Home/Index` , `/Home/Index/123` et `/`

Contraintes de routage

Il est possible de créer une contrainte de routage personnalisée qui peut être utilisée dans des routes pour contraindre un paramètre à des valeurs ou à un modèle spécifiques.

Cette contrainte correspondra à un modèle de culture / locale typique, comme 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);
    }
}
```

Par la suite, la contrainte doit être enregistrée avant de pouvoir être utilisée dans les itinéraires.

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

Maintenant, il peut être utilisé dans les itinéraires.

Utilisation sur les contrôleurs

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

Utilisation sur des actions

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

Utilisation dans les routes par défaut

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

Lire Le routage en ligne: https://riptutorial.com/fr/asp-net-core/topic/2863/le-routage

Chapitre 16: Limitation de débit

Remarques

AspNetCoreRateLimit est une solution de limitation de débit ASP.NET Core open source conçue pour contrôler le taux de requêtes que les clients peuvent adresser à une API Web ou à une application MVC en fonction de l'adresse IP ou de l'ID client.

Examples

Limitation de débit basée sur l'adresse IP du client

Avec le middleware IpRateLimit, vous pouvez définir plusieurs limites pour différents scénarios, comme autoriser une plage IP ou IP à effectuer un nombre maximum d'appels dans un intervalle de temps de 15 minutes, etc. Vous pouvez définir ces limites pour adresser toutes les requêtes API ou vous pouvez définir les limites de chaque chemin d'URL ou verbe et chemin HTTP.

Installer

NuGet installer :

```
Install-Package AspNetCoreRateLimit
```

Code 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();
}
```

Vous devez enregistrer le middleware avant tout autre composant, à l'exception de loggerFactory.

Si vous chargez votre application, vous devez utiliser `IDistributedCache` avec Redis ou SQLServer pour que toutes les instances de `IDistributedCache` aient le même magasin de limites de débit. Au lieu des magasins en mémoire, vous devez injecter les magasins distribués comme ceci:

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

Configuration et règles générales 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
      }
    ]
  }
```

Si `EnableEndpointRateLimiting` est défini sur `false` les limites s'appliquent globalement et seules les règles ayant pour point de terminaison `*` seront appliquées. Par exemple, si vous définissez une limite de 5 appels par seconde, tout appel HTTP à un noeud final sera comptabilisé dans cette limite.

Si `EnableEndpointRateLimiting` est défini sur `true` les limites s'appliquent à chaque noeud final, comme dans `{HTTP_Verb}{PATH}` . Par exemple, si vous définissez une limite de 5 appels par seconde pour `*:/api/values` un client peut appeler `GET /api/values` 5 fois par seconde mais également 5 fois `PUT /api/values` .

Si `StackBlockedRequests` est défini sur `false` appels rejetés ne sont pas ajoutés au compteur d'accélération. Si un client effectue 3 requêtes par seconde et que vous définissez une limite d'un appel par seconde, les autres limites, telles que les compteurs par minute ou par jour, enregistrent uniquement le premier appel, celui qui n'a pas été bloqué. Si vous souhaitez que les demandes rejetées soient comptabilisées dans les autres limites, vous devez définir `StackBlockedRequests` sur `true` .

`RealIpHeader` est utilisé pour extraire l'adresse IP du client lorsque votre serveur Kestrel se trouve derrière un proxy inverse. Si votre proxy utilise un en-tête différent, `X-Real-IP` utilise cette option pour le configurer.

`ClientIdHeader` est utilisé pour extraire l'ID client pour la liste blanche, si un identifiant client est présent dans cet en-tête et correspond à une valeur spécifiée dans ClientWhitelist, aucune limite de taux n'est appliquée.

Remplacez les règles générales pour des adresses IP spécifiques 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
          }
        ]
      }
    ]
  }
```

Le champ IP prend en charge les valeurs IP v4 et v6 et se situe entre "192.168.0.0/24", "fe80 :: / 10" ou "192.168.0.0-192.168.0.255".

Définition des règles de limite de taux

Une règle est composée d'un noeud final, d'une période et d'une limite.

Le format du point de terminaison est `{HTTP_Verb}:{PATH}` , vous pouvez cibler n'importe quel verbe HTTP en utilisant le symbole astérisque.

Le format de période est `{INT}{PERIOD_TYPE}` , vous pouvez utiliser l'un des types de période suivants: `s, m, h, d` .

Le format de limite est `{LONG}` .

Exemples :

Le taux limite tous les points de terminaison à 2 appels par seconde:

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

Si, à partir de la même adresse IP, dans la même seconde, vous effectuerez 3 appels GET vers api / valeurs, le dernier appel sera bloqué. Mais si, dans la même seconde, vous appelez PIT api / values, la requête sera transmise parce que le point de terminaison est différent. Lorsque la limitation de la fréquence d'extrémité est activée, chaque appel est limité par le taux basé sur `{HTTP_Verb}{PATH}` .

Le taux limite les appels avec n'importe quel verbe HTTP à `/api/values` à 5 appels par 15 minutes:

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

Taux limite GET appel à `/api/values` à 5 appels par heure:

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

```
}
```

Si, à partir de la même adresse IP, vous effectuez 6 appels GET vers api / valeurs, le dernier appel sera bloqué. Mais si, dans la même heure, vous appelez GET api / values / 1, la requête est transmise parce que le point de terminaison est différent.

Comportement

Lorsqu'un client effectue un appel HTTP, IpRateLimitMiddleware effectue les opérations suivantes:

- extrait l'adresse IP, l'identifiant du client, le verbe HTTP et l'URL de l'objet de requête, si vous souhaitez implémenter votre propre logique d'extraction, vous pouvez remplacer l' `IpRateLimitMiddleware.SetIdentity`
- recherche l'adresse IP, l'identifiant du client et l'URL dans les listes blanches, s'il y a des correspondances, aucune action n'est prise
- recherche dans les règles IP pour une correspondance, toutes les règles qui s'appliquent sont regroupées par période, pour chaque période la règle la plus restrictive est utilisée
- recherche dans les règles générales une correspondance, si une règle générale qui correspond à une période définie n'est pas présente dans les règles IP, cette règle générale est également utilisée
- pour chaque règle de correspondance, le compteur de limite de débit est incrémenté, si la valeur du compteur est supérieure à la limite de la règle, la requête est bloquée

Si la requête est bloquée, le client reçoit une réponse textuelle comme ceci:

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

Vous pouvez personnaliser la réponse en modifiant ces options `HttpStatusCode` et `QuotaExceededMessage` . Si vous souhaitez implémenter votre propre réponse, vous pouvez remplacer `IpRateLimitMiddleware.ReturnQuotaExceededResponse` . La valeur de l'en `Retry-After` tête `Retry-After` est exprimée en secondes.

Si la demande ne reçoit pas de taux limité, la plus longue période définie dans les règles de correspondance est utilisée pour composer les en-têtes X-Rate-Limit, ces en-têtes sont injectés dans la réponse:

```
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
```

Par défaut, les requêtes bloquées sont consignées à l'aide de `Microsoft.Extensions.Logging.ILogger` . Si vous souhaitez implémenter votre propre journalisation, vous pouvez remplacer `IpRateLimitMiddleware.LogBlockedRequest` . Le consignateur par défaut émet les informations suivantes lorsqu'une demande reçoit un taux limité:

```
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.
```

Mettre à jour les limites de taux lors de l'exécution

Au démarrage de l'application, les règles de limite de débit IP définies dans `appsettings.json` sont chargées dans le cache par `MemoryCacheClientPolicyStore` ou `DistributedCacheIpPolicyStore` selon le type de fournisseur de cache utilisé. Vous pouvez accéder au magasin de règles Ip dans un contrôleur et modifier les règles IP comme suit:

```
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);
    }
}
```

De cette façon, vous pouvez stocker les limites de taux IP dans une base de données et les mettre en cache après le démarrage de chaque application.

Limitation du taux basé sur l'ID du client

Avec le middleware ClientRateLimit, vous pouvez définir plusieurs limites pour différents scénarios, comme autoriser un client à effectuer un nombre maximal d'appels par intervalle de

temps, par seconde, 15 minutes, etc. Vous pouvez définir ces limites pour adresser toutes les requêtes adressées à une API peut couvrir les limites de chaque chemin d'URL ou de verbe et chemin HTTP.

Installer

NuGet installer :

```
Install-Package AspNetCoreRateLimit
```

Code 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();
}
```

Vous devez enregistrer le middleware avant tout autre composant, à l'exception de loggerFactory.

Si vous chargez votre application, vous devez utiliser `IDistributedCache` avec Redis ou SQLServer pour que toutes les instances de `IDistributedCache` aient le même magasin de limites de débit. Au lieu des magasins en mémoire, vous devez injecter les magasins distribués comme ceci:

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

```
    services.AddSingleton<IRateLimitCounterStore,DistributedCacheRateLimitCounterStore>();
```

Configuration et règles générales 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
      }
    ]
  }
```

Si `EnableEndpointRateLimiting` est défini sur `false` les limites s'appliquent globalement et seules les règles ayant pour point de terminaison `*` seront appliquées. Par exemple, si vous définissez une limite de 5 appels par seconde, tout appel HTTP à un noeud final sera comptabilisé dans cette limite.

Si `EnableEndpointRateLimiting` est défini sur `true` les limites s'appliquent à chaque noeud final, comme dans `{HTTP_Verb}{PATH}` . Par exemple, si vous définissez une limite de 5 appels par seconde pour `*:/api/values` un client peut appeler `GET /api/values` 5 fois par seconde mais également 5 fois `PUT /api/values` .

Si `StackBlockedRequests` est défini sur `false` appels rejetés ne sont pas ajoutés au compteur d'accélération. Si un client effectue 3 requêtes par seconde et que vous définissez une limite d'un appel par seconde, les autres limites, telles que les compteurs par minute ou par jour, enregistrent uniquement le premier appel, celui qui n'a pas été bloqué. Si vous souhaitez que les demandes rejetées soient comptabilisées dans les autres limites, vous devez définir `StackBlockedRequests` sur `true` .

`ClientIdHeader` est utilisé pour extraire l'ID du client, si un identifiant de client est présent dans cet en-tête et correspond à une valeur spécifiée dans ClientWhitelist, aucune limite de taux n'est

appliquée.

Remplacez les règles générales pour des clients spécifiques 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
         }
       ]
     }
   ]
 }
```

Définition des règles de limite de taux

Une règle est composée d'un noeud final, d'une période et d'une limite.

Le format du point de terminaison est `{HTTP_Verb}:{PATH}` , vous pouvez cibler n'importe quel verbe HTTP en utilisant le symbole astérisque.

Le format de période est `{INT}{PERIOD_TYPE}` , vous pouvez utiliser l'un des types de période suivants: `s, m, h, d` .

Le format de limite est `{LONG}` .

Exemples :

Le taux limite tous les points de terminaison à 2 appels par seconde:

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

Si, dans la même seconde, un client effectue 3 appels GET vers api / valeurs, le dernier appel sera bloqué. Mais si, dans la même seconde, il appelle PIT api / values, la requête passera par un point final différent. Lorsque la limitation de la fréquence d'extrémité est activée, chaque appel est limité par le taux basé sur `{HTTP_Verb}{PATH}` .

Le taux limite les appels avec n'importe quel verbe HTTP à `/api/values` à 5 appels par 15 minutes:

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

Taux limite GET appel à `/api/values` à 5 appels par heure:

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

Si dans une heure, un client effectue 6 appels GET vers api / valeurs, le dernier appel sera bloqué. Mais si dans la même heure il appelle GET api / values / 1 aussi, la requête passera parce que c'est un point final différent.

Comportement

Lorsqu'un client effectue un appel HTTP, ClientRateLimitMiddleware effectue les opérations suivantes:

- extrait l'ID du client, le verbe HTTP et l'URL de l'objet de requête, si vous souhaitez implémenter votre propre logique d'extraction, vous pouvez remplacer `ClientRateLimitMiddleware.SetIdentity`
- recherche l'identifiant et l'URL du client dans les listes blanches, s'il y a des correspondances, aucune action n'est prise
- recherche dans le client les règles d'une correspondance, toutes les règles qui s'appliquent sont regroupées par période, pour chaque période la règle la plus restrictive est utilisée
- recherche dans les règles générales une correspondance, si une règle générale qui correspond à une période définie est absente des règles du client, cette règle générale est également utilisée
- pour chaque règle de correspondance, le compteur de limite de débit est incrémenté, si la valeur du compteur est supérieure à la limite de la règle, la requête est bloquée

Si la requête est bloquée, le client reçoit une réponse textuelle comme ceci:

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

Vous pouvez personnaliser la réponse en modifiant ces options `HttpStatusCode` et `QuotaExceededMessage` . Si vous souhaitez implémenter votre propre réponse, vous pouvez remplacer `ClientRateLimitMiddleware.ReturnQuotaExceededResponse` . La valeur de l'en `Retry-After` tête `Retry-After` est exprimée en secondes.

Si la demande ne reçoit pas de taux limité, la plus longue période définie dans les règles de correspondance est utilisée pour composer les en-têtes X-Rate-Limit, ces en-têtes sont injectés dans la réponse:

```
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
```

Par défaut, les requêtes bloquées sont consignées à l'aide de `Microsoft.Extensions.Logging.ILogger` . Si vous souhaitez implémenter votre propre journalisation, vous pouvez remplacer `ClientRateLimitMiddleware.LogBlockedRequest` . Le consignateur par défaut émet les informations suivantes lorsqu'une demande reçoit un taux limité:

```
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.
```

Mettre à jour les limites de taux lors de l'exécution

Au démarrage de l'application, les règles de limite de débit du client définies dans `appsettings.json` sont chargées dans le cache par `MemoryCacheClientPolicyStore` ou `DistributedCacheClientPolicyStore` selon le type de fournisseur de cache utilisé. Vous pouvez accéder au magasin de règles client dans un contrôleur et modifier les règles comme suit:

```
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);
    }
}
```

De cette façon, vous pouvez stocker les limites de taux client dans une base de données et les mettre en cache après le démarrage de chaque application.

Lire Limitation de débit en ligne: https://riptutorial.com/fr/asp-net-core/topic/5240/limitation-de-debit

Chapitre 17: Localisation

Examples

Localisation à l'aide de ressources de langage JSON

Dans ASP.NET Core, il existe différentes manières de localiser / globaliser notre application. Il est important de choisir un moyen adapté à vos besoins. Dans cet exemple, vous verrez comment créer une application ASP.NET Core multilingue qui lit des chaînes spécifiques à une langue à partir de fichiers `.json` et les stocke en mémoire pour assurer la localisation dans toutes les sections de l'application et maintenir des performances élevées.

Nous procédons ainsi en utilisant `Microsoft.EntityFrameworkCore.InMemory` package `Microsoft.EntityFrameworkCore.InMemory` .

Remarques:

1. L'espace de noms de ce projet est `DigitalShop` que vous pouvez modifier dans votre propre espace de noms de projets.
2. Envisagez de créer un nouveau projet afin de ne pas rencontrer d'erreurs étranges
3. En aucun cas cet exemple ne montre les meilleures pratiques, donc si vous pensez que cela peut être amélioré, veuillez le modifier

Pour commencer, ajoutons les packages suivants à la section `dependencies` **existante** du fichier `project.json` :

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

Maintenant , nous allons remplacer le `Startup.cs` fichier: (en `using` les états sont supprimés car ils peuvent être facilement ajoutés plus tard)

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));
        }
    }
}
```

Dans le code ci-dessus, nous ajoutons d'abord trois variables de champs `public static` que nous initialiserons ultérieurement à l'aide des valeurs lues dans le fichier de paramètres.

Dans le constructeur de la classe `Startup` , nous ajoutons un fichier de paramètres json à la variable de `builder` . Le premier fichier est requis pour que l'application fonctionne, alors n'hésitez pas à créer `appsettings.json` dans la racine de votre projet si elle n'existe pas déjà. À l'aide de Visual Studio 2015, ce fichier est créé automatiquement. Changez simplement son contenu pour: (vous pouvez omettre la section de `Logging` si vous ne l'utilisez pas)

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
  }
}
```

À l'avenir, créez trois dossiers dans la racine de votre projet:

`Models` , `Services` et `Languages` . Dans le dossier `Models` , créez un autre dossier nommé `Localization`

.

Dans le dossier `Services` , nous créons un nouveau fichier .cs nommé `EFLocalization` . Le contenu serait: (Encore une fois à l' `using` ne sont pas inclus déclarations)

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;
        }
    }
}
```

Dans le fichier ci-dessus, nous implémentons l'interface `IStringLocalizerFactory` depuis Entity Framework Core afin de créer un service de localisation personnalisé. La partie importante est le constructeur de `EFStringLocalizerFactory` où nous faisons une liste de toutes les langues disponibles et l'ajoutons au contexte de la base de données. Chacun de ces fichiers de langue agit comme une base de données distincte.

Ajoutez maintenant chacun des fichiers suivants au dossier `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();
        }
    }
}
```

Les fichiers ci-dessus ne sont que des modèles qui seront remplis avec des ressources linguistiques, des cultures et un `DBContext` typique utilisé par EF Core.

La dernière chose dont nous avons besoin pour faire tout ce travail est de créer les fichiers de ressources linguistiques. Les fichiers JSON utilisés pour stocker une paire clé-valeur pour différentes langues disponibles dans votre application.

Dans cet exemple, notre application ne dispose que de deux langues. Anglais et persan. Pour chacune des langues, nous avons besoin de deux fichiers. Un fichier JSON contenant des paires clé-valeur et un fichier `.cs` contenant une classe portant le même nom que le fichier JSON. Cette classe a une méthode, `GetList` qui désérialise le fichier JSON et le renvoie. Cette méthode est appelée dans le constructeur de `EFStringLocalizerFactory` que nous avons créé précédemment.

Donc, créez ces quatre fichiers dans votre dossier `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": "خوشآمدی"
 },
 {
  "Key": "Hello",
    "Value": "سالم"
 },
]
```

Nous avons tous fini. Maintenant, pour accéder aux chaînes de langue (paires clé-valeur) n'importe où dans votre code (`.cs` ou `.cshtml`), vous pouvez effectuer les opérations suivantes:

dans un fichier `.cs` (soit Controller ou non, peu importe):

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

dans un fichier de vue Razor (`.cshtml`):

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

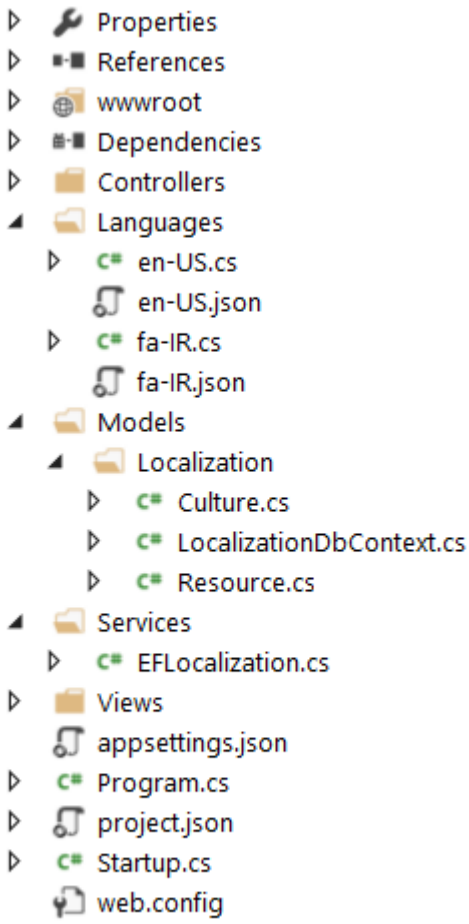
Peu de choses à retenir:

- Si vous essayez d'accéder à une `Key` qui n'existe pas dans le fichier JSON ou chargé, vous obtiendrez juste le littéral clé (dans l'exemple ci - dessus, en essayant d'accéder `Startup._e["How are you"]` - `How are you` `Startup._e["How are you"]` retournerez `How are you` -

`How are you` peu importe les paramètres linguistiques, car il n'existe pas

- Si vous modifiez une valeur de chaîne dans un fichier de langue `.json` , vous devrez **redémarrer** l'application. Sinon, il affichera simplement la valeur par défaut (nom de la clé). **Ceci est particulièrement important lorsque vous exécutez votre application sans déboguer.**
- Le `appsettings.json` peut être utilisé pour stocker toutes sortes de paramètres dont votre application peut avoir besoin
- Le redémarrage de l'application n'est **pas nécessaire** si vous souhaitez simplement modifier les **paramètres de langue / culture du fichier** `appsettings.json` . Cela signifie que vous pouvez avoir une option dans l'interface de vos applications pour permettre aux utilisateurs de modifier la langue / culture au moment de l'exécution.

Voici la structure finale du projet:



Définir la culture de la demande via le chemin de l'URL

Par défaut, le middleware Request Localization intégré prend uniquement en charge la définition de la culture via une requête, un cookie ou un en `Accept-Language` tête `Accept-Language` . Cet exemple montre comment créer un middleware qui permet de définir la culture comme faisant partie du chemin comme dans `/api/en-US/products` .

Cet exemple de middleware suppose que les paramètres régionaux se trouvent dans le deuxième segment du chemin.

```
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));
    }
}
```

Enregistrement du 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);
```

Contraintes d'itinéraire personnalisé

L'ajout et la création de contraintes de route personnalisées sont illustrés dans l'exemple des contraintes de route . L'utilisation de contraintes simplifie l'utilisation des contraintes de route personnalisées.

Enregistrer l'itinéraire

Exemple d'enregistrement des routes sans utiliser de contraintes personnalisées

```
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?}");
});
```

Lire Localisation en ligne: https://riptutorial.com/fr/asp-net-core/topic/2869/localisation

Chapitre 18: Middleware

Remarques

Le middleware est un composant logiciel qui détermine comment traiter la demande et décide de la transmettre au composant suivant du pipeline d'applications. Chaque middleware a un rôle et des actions spécifiques à exécuter sur la demande.

Examples

Utilisation du middleware ExceptionHandler pour envoyer une erreur JSON personnalisée au client

Définissez votre classe qui représentera votre erreur personnalisée.

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

   // other fields

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

Ensuite, mettez le middleware ExceptionHandler suivant à la méthode Configure. Attention, la commande de middleware est 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 pour définir la réponse ContentType

L'idée est d'utiliser le rappel `HttpContext.Response.OnStarting` , car il s'agit du dernier événement déclenché avant l'envoi des en-têtes. Ajoutez ce qui suit à votre méthode `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);
}
```

Passer des données à travers la chaîne de middleware

De la documentation :

> La collection ***HttpContext.Items*** est le meilleur emplacement pour stocker des données uniquement nécessaires lors du traitement d'une requête donnée. Son contenu est supprimé après chaque demande. Il est préférable de l'utiliser comme moyen de communication entre des composants ou des intergiciels fonctionnant à différents moments au cours d'une requête et n'ayant aucune relation directe entre eux pour transmettre des paramètres ou renvoyer des valeurs.

`HttpContext.Items` est une collection de dictionnaires simple de type `IDictionary<object, object>` . Cette collection est

- disponible dès le début d'une `HttpRequest`
- et est rejeté à la fin de chaque demande.

Vous pouvez y accéder simplement en attribuant une valeur à une entrée à clé ou en demandant la valeur d'une clé donnée.

Par exemple, certains Middleware simples pourraient ajouter quelque chose à la collection Items:

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

et plus tard, un autre middleware pourrait y accéder:

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

Exécuter, mapper, utiliser

Courir

Termine la chaîne. Aucune autre méthode de middleware ne fonctionnera après cela. Devrait être placé à la fin de tout pipeline.

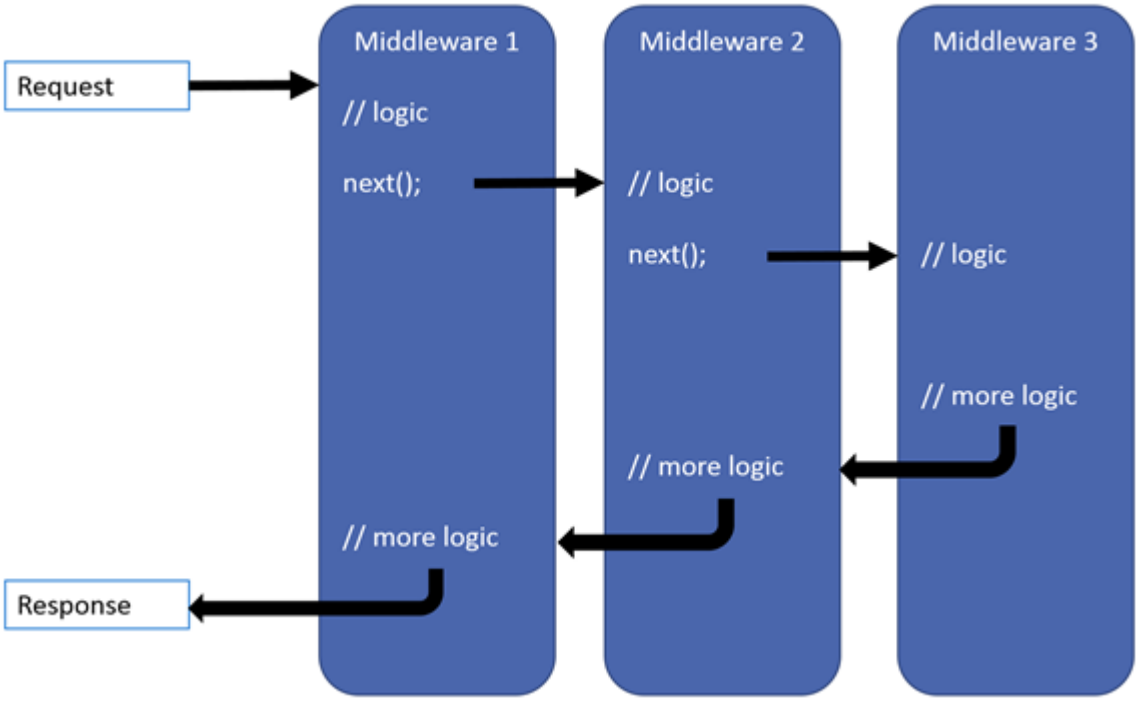
```
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from " + _environment);
});
```

Utilisation

Effectue une action avant et après le prochain délégué.

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

Illustration de son fonctionnement:



MapWhen

Active le pipeline de branchement. Exécute le middleware spécifié si la condition est remplie.

```
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);
}
```

Carte

Similaire à MapWhen. Exécute le middleware si le chemin demandé par l'utilisateur est égal au chemin fourni en paramètre.

```
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);

}
```

Basé sur ASP.net Core Docs

Lire Middleware en ligne: https://riptutorial.com/fr/asp-net-core/topic/1479/middleware

Chapitre 19: Mise en cache

Introduction

La mise en cache permet d'améliorer les performances d'une application en conservant une copie facilement accessible des données. *Aspnet Core* est livré avec deux abstractions de mise en cache faciles à utiliser et à tester.

La mémoire cache stockera les données dans la mémoire cache du serveur local.

Le cache distribué contiendra le cache de données dans un emplacement centralisé accessible par les serveurs du cluster. Il est livré avec trois implémentations originales: en mémoire (pour les tests unitaires et en développement local), Redis et Sql Server.

Examples

Utilisation du cache InMemory dans l'application ASP.NET Core

Pour utiliser un cache en mémoire dans votre application ASP.NET, ajoutez les dépendances suivantes à votre fichier `project.json` :

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

Ajouter le service de cache (à partir de Microsoft.Extensions.Caching.Memory) à la méthode ConfigureServices dans la classe de démarrage

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

Pour ajouter des éléments au cache dans notre application, nous utiliserons `IMemoryCache` qui peut être injecté dans n'importe quelle classe (par exemple Controller), comme indiqué ci-dessous.

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

Get retournera la valeur si elle existe, mais renvoie `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))
```

Utilisez la méthode **Set** pour écrire dans le cache. Set accepte la clé à utiliser pour rechercher la valeur, la valeur à mettre en cache et un ensemble de `MemoryCacheEntryOptions` .

`MemoryCacheEntryOptions` vous permet de spécifier une expiration absolue ou en temps réel du cache, une priorité de mise en cache, des rappels et des dépendances. Un des échantillons ci-dessous-

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

Mise en cache distribuée

Pour exploiter le cache distribué, vous devez référencer l'une des implémentations disponibles:

- Redis
- Serveur SQL

Par exemple, vous allez enregistrer l'implémentation de Redis comme suit:

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

Requiert `IDistributedCache` dépendance `IDistributedCache` là où vous en avez besoin:

```
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;
    }
}
```

Lire Mise en cache en ligne: https://riptutorial.com/fr/asp-net-core/topic/8090/mise-en-cache

Chapitre 20: project.json

Introduction

Project json est une structure de fichier de configuration de projet, utilisée temporairement par les projets asp.net-core, avant que Microsoft ne retourne dans les fichiers csproj en faveur de msbuild.

Examples

Exemple de projet de bibliothèque simple

Une bibliothèque basée sur NETStandard 1.6 ressemblerait à ceci:

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

Fichier json complet:

Tiré de la page github de Microsoft avec la documentation officielle

```
{
"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
}
```

Projet de démarrage simple

Un exemple simple de configuration de projet pour une application 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"
    }
  },
}
```

Lire project.json en ligne: https://riptutorial.com/fr/asp-net-core/topic/9364/project-json

Chapitre 21: Publication et déploiement

Examples

Crécerelle. Configuration de l'adresse d'écoute

En utilisant Kestrel, vous pouvez spécifier le port en utilisant les approches suivantes:

1. Définition de la variable d'environnement `ASPNETCORE_URLS` .

   les fenêtres

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

   OS X

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

2. Via la ligne de commande en passant le paramètre `--server.urls`

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

3. Utiliser la méthode `UseUrls()`

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

4. Définition `server.urls` paramètre `server.urls` dans la source de configuration.

Exemple suivant utilisez le fichier hosting.json par exemple.

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

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

Exemples de valeurs possibles:

- écoutez 5000 sur toutes les adresses IP4 et IP6 depuis n'importe quelle interface:

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

  ou

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

- écoutez 5000 sur chaque adresse IP4:

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

  On devrait être prudent et ne pas utiliser `http://*:5000;http://::5000` , `http://::5000;http://*:5000` , `http://*:5000;http://0.0.0.0:5000` ou `http://*:5000;http://0.0.0.0:5000` car il faudra enregistrer l'adresse IP6 :: ou l'adresse IP4 0.0.0.0 deux fois

Ajouter un fichier à `publishOptions` dans `project.json`

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

et dans le point d'entrée de l'appel d'application `.UseConfiguration(config)` lors de la création de 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();
}
```

Lire Publication et déploiement en ligne: https://riptutorial.com/fr/asp-net-core/topic/2262/publication-et-deploiement

Chapitre 22: Regroupement et Minification

Examples

Grunt et Gulp

Dans les applications ASP.NET Core, vous regroupez et gérez les ressources côté client pendant la conception à l'aide d'outils tiers, tels que Gulp et Grunt . En utilisant le regroupement et la minification au moment du design, les fichiers minifiés sont créés avant le déploiement de l'application. Le regroupement et la minimisation avant le déploiement offrent l'avantage de réduire la charge du serveur. Cependant, il est important de reconnaître que le regroupement et la minification au moment du design augmentent la complexité de la construction et ne fonctionnent qu'avec des fichiers statiques.

Cela se fait dans ASP.NET Core en configurant Gulp via un fichier `gulpfile.js` dans votre projet:

```
// 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("."));
});
```

Cette approche va regrouper et minimiser correctement vos fichiers Javascript et CSS existants

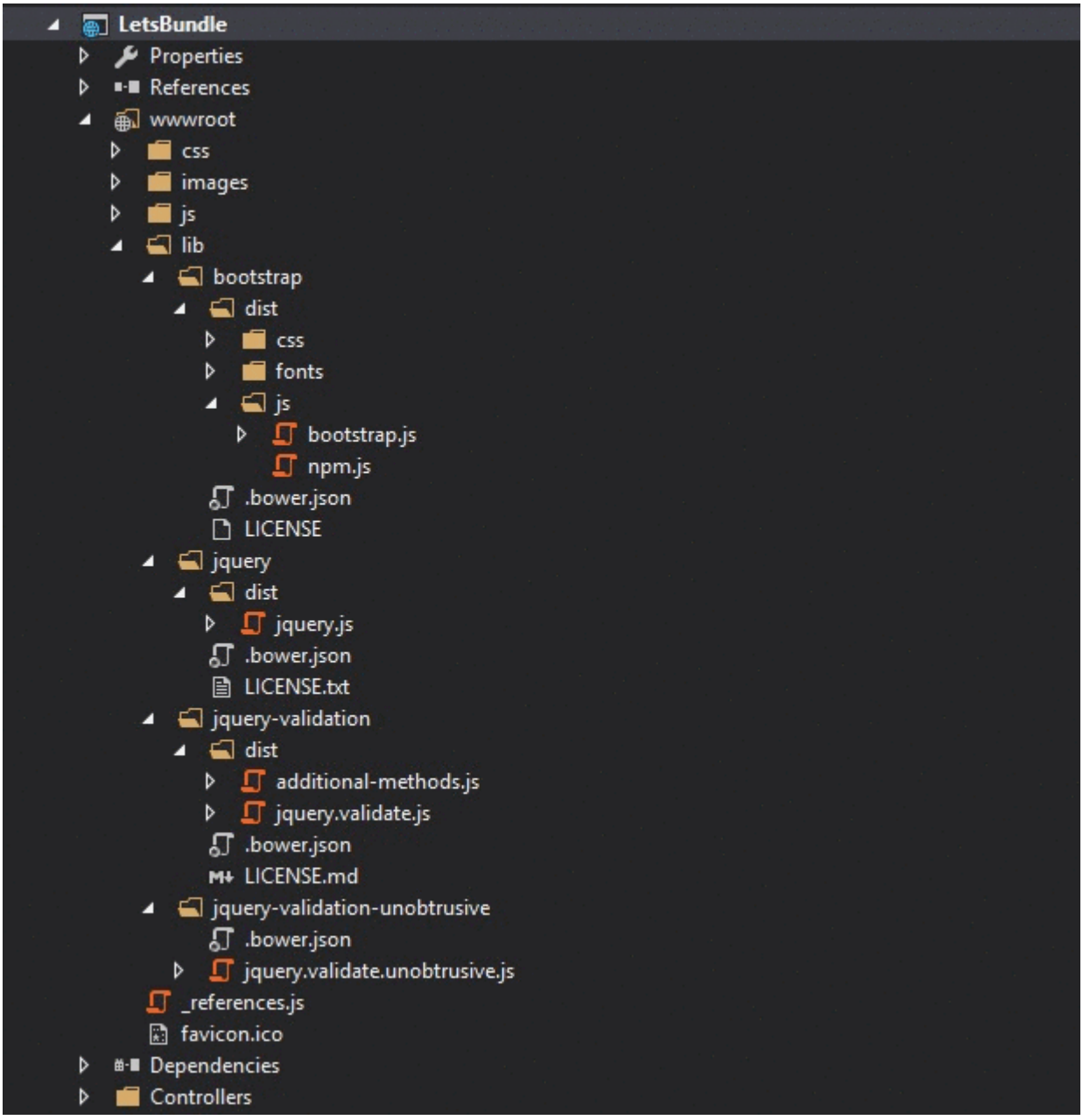
en fonction des répertoires et des modèles de globalisation utilisés.

Extension Bundler et Minifier

Visual Studio propose également une extension Bundler and Minifier disponible capable de gérer ce processus pour vous. L'extension vous permet de sélectionner et de regrouper facilement les fichiers dont vous avez besoin sans écrire de ligne de code.

Construire vos paquets

Après avoir installé l'extension, **sélectionnez tous les fichiers spécifiques à inclure dans un ensemble et utilisez l'option Assembler et réduire les fichiers de l'extension:**



Cela vous invitera à nommer votre paquet et à choisir un emplacement pour le sauvegarder. Vous remarquerez alors un nouveau fichier dans votre projet appelé `bundleconfig.json` qui ressemble à ceci:

```
[
  {
    "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"
    ]
  }
]
```

> **REMARQUE:** L'ordre dans lequel les fichiers sont sélectionnés déterminera l'ordre dans lequel ils apparaissent dans l'ensemble, donc si vous avez des dépendances, assurez-vous d'en tenir compte.

Réduire vos paquets

Maintenant, l'étape précédente regroupera simplement vos fichiers, si vous souhaitez en réduire la taille, vous devez l'indiquer dans le fichier bundleconfig.json. **Ajoutez simplement un bloc de `minify` comme le suivant à votre bundle existant pour indiquer que vous voulez le minifier:**

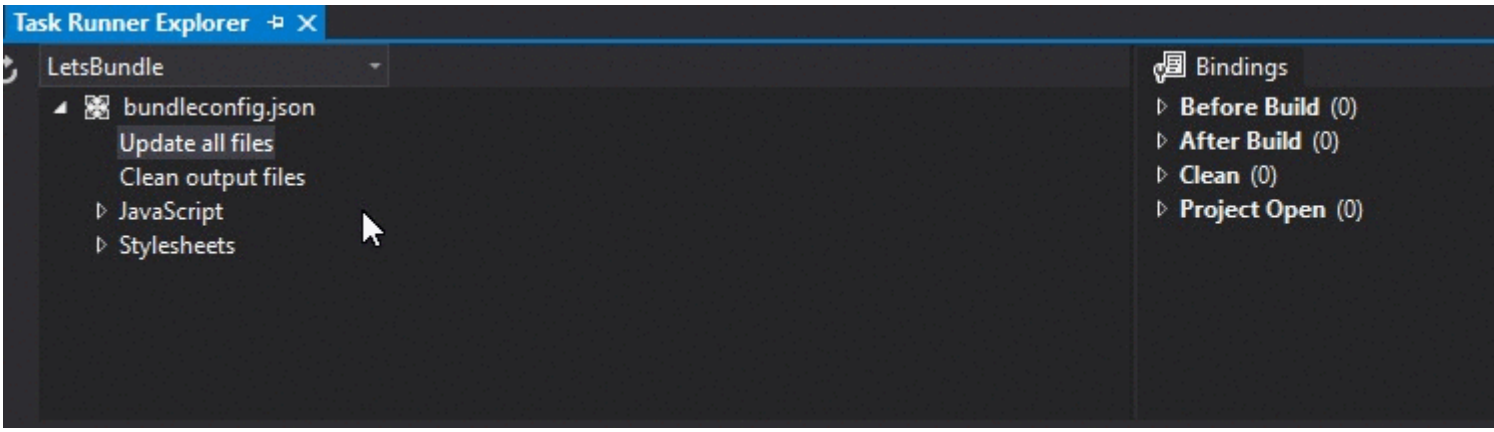
```
[
  {
    "outputFileName": "wwwroot/app/bundle.js",
    "inputFiles": [
      "wwwroot/lib/jquery/dist/jquery.js",
      "wwwroot/lib/bootstrap/dist/js/bootstrap.js",
      "wwwroot/lib/jquery-validation/dist/jquery.validate.js",
      "wwwroot/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"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

Automatisez vos paquets

Enfin, si vous souhaitez automatiser ce processus, vous pouvez planifier l'exécution d'une tâche à chaque fois que votre application est créée pour garantir que vos offres reflètent toutes les modifications de votre application.

Pour ce faire, vous devez procéder comme suit:

- **Ouvrez l'explorateur de tâches** (via Outils> Explorateur de tâches).
- **Cliquez avec le bouton droit sur l'option Mettre à jour tous les fichiers** sous `bundleconfig.json` .
- **Sélectionnez votre liaison préférée** dans le menu contextuel Liaisons.



Après cela, vos ensembles doivent être automatiquement mis à jour à l'étape que vous avez sélectionnée.

La commande dotnet bundle

La version ASP.NET Core RTM a introduit `BundlerMinifier.Core` , un nouvel outil de regroupement et de simplification qui peut être facilement intégré dans les applications ASP.NET Core existantes et ne nécessite aucune extension externe ni fichier de script.

Utiliser BundlerMinifier.Core

Pour utiliser cet outil, **ajoutez simplement une référence à `BundlerMinifier.Core` dans la section des `tools` de votre fichier `project.json` existant,** comme indiqué ci-dessous:

```
"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"
}
```

Configuration de vos bundles

Après avoir ajouté l'outil, vous devrez **ajouter un fichier `bundleconfig.json` dans votre projet** qui sera utilisé pour configurer les fichiers que vous souhaitez inclure dans vos ensembles. Une configuration minimale peut être vue ci-dessous:

```
[
  {
    "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
    }
  }
]
```

Création / mise à jour de bundles

Une fois vos offres configurées, vous pouvez regrouper et minimiser vos fichiers existants à l'aide de la commande suivante:

```
dotnet bundle
```

Groupage automatisé

Le processus de regroupement et de réduction peut être automatisé dans le cadre du processus de génération en ajoutant la commande `dotnet bundle` dans la section `dotnet bundle` de votre fichier `project.json` existant:

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

Commandes disponibles

Vous pouvez voir une liste de toutes les commandes disponibles et leurs descriptions ci-dessous:

- **dotnet bundle** - Exécute la commande bundle à l'aide du fichier `bundleconfig.json` pour regrouper et réduire vos fichiers spécifiés.
- **dotnet bundle clean** - Efface tous les fichiers de sortie existants du disque.
- **dotnet bundle watch** - Crée des observateurs qui exécutent automatiquement `dotnet bundle` chaque fois qu'un fichier d'entrée existant de la configuration `bundleconfig.json` regroupe vos fichiers.
- **dotnet bundle help** - Affiche toutes les options d'aide disponibles et les instructions pour utiliser l'interface de ligne de commande.

Lire Regroupement et Minification en ligne: https://riptutorial.com/fr/asp-net-

core/topic/4051/regroupement-et-minification

Chapitre 23: Sessions dans ASP.NET Core 1.0

Introduction

Utilisation de sessions dans ASP.NET Core 1.0

Examples

Exemple de base de manipulation de session

1) Tout d'abord, ajoutez la dépendance dans project.json - `"Microsoft.AspNetCore.Session": "1.1.0",`

2) Dans `startup.cs` et ajoutez des `AddSession()` et `AddDistributedMemoryCache()` au `ConfigureServices` comme ceci -

```
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) Ajoutez l'appel `UseSession()` dans la méthode Configure du démarrage comme ceci -

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

4) Dans Controller, l'objet Session peut être utilisé comme ceci-

```
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. Si vous utilisez la stratégie cors, il peut parfois y avoir des erreurs, après avoir activé session concernant les en-têtes sur l'activation de l'en - tête *AllowCredentials* et l'utilisation de
   En- tête *WithOrigins* au lieu de *AllowAllOrigins* .

Lire Sessions dans ASP.NET Core 1.0 en ligne: https://riptutorial.com/fr/asp-net-core/topic/8067/sessions-dans-asp-net-core-1-0

Chapitre 24: Tag Helpers

Paramètres

prénom	Info
asp-action	Le nom de la méthode d'action à laquelle le formulaire doit être envoyé
asp-controller	Le nom du contrôleur où la méthode d'action spécifiée dans asp-action existe
asp-route-*	Valeurs de routage personnalisées que vous souhaitez ajouter en tant que chaîne de requête à la valeur de l'attribut d'action de formulaire. Remplacez 8 par le nom de chaîne de requête que vous voulez

Examples

Form Tag Helper - Exemple de base

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

Form Tag Helper - Avec des attributs de route personnalisés

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

Cela générera le balisage ci-dessous

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

Assistant de saisie

En supposant que votre vue est fortement saisie dans un modèle de vue tel que

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

Et vous transmettez un objet de ceci à la vue de votre méthode d'action.

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

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

</form>
```

Cela générera le balisage ci-dessous.

```
<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>
```

Si vous souhaitez que le champ de saisie soit rendu avec une valeur par défaut, vous pouvez définir la valeur de la propriété Name de votre modèle de vue dans la méthode d'action.

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

Soumission de formulaire et liaison de modèle

La liaison de modèle fonctionnera `CreateProduct` si vous utilisez `CreateProduct` comme paramètre de méthode d'action HttpPost / paramètre nommé `name`

Sélectionnez Tag Helper

En supposant que votre vue est fortement typée dans un modèle de vue comme celui-ci

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

Et dans votre méthode d'action GET, vous créez un objet de ce modèle de vue, en définissant la propriété Catégories et en l'envoyant à la vue.

```
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);
}
```

et à votre avis

```
@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>
```

Cela rendra le balisage ci-dessous (*inclus uniquement les parties pertinentes du formulaire / champs*)

```
<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>
```

Obtenir la valeur de liste déroulante sélectionnée dans la soumission du formulaire

Vous pouvez utiliser le même modèle de vue que votre paramètre de méthode d'action HttpPost

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

Définir une option comme celle sélectionnée

Si vous souhaitez définir une option en tant qu'option sélectionnée, vous pouvez simplement définir la valeur de la propriété `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);
}
```

Rendu d'une liste déroulante à sélection multiple / ListBox

Si vous voulez rendre une liste déroulante à sélection multiple, vous pouvez simplement modifier votre propriété de modèle de vue que vous utilisez `asp-for` attribut `asp-for` dans votre vue en un type de tableau.

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

Dans la vue

```
@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>
```

Cela va générer l'élément SELECT avec `multiple` attributs

```
<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>
```

Assistant de tag personnalisé

Vous pouvez créer vos propres aides au tag en implémentant `ITagHelper` ou en dérivant de la classe de commodité `TagHelper` .

- La convention par défaut consiste à cibler une balise html correspondant au nom de l'assistant sans le suffixe facultatif TagHelper. Par exemple, `WidgetTagHelper` ciblera une `<widget>` .
- L'attribut `[HtmlTargetElement]` peut être utilisé pour contrôler davantage la balise ciblée
- Toute propriété publique de la classe peut recevoir une valeur en tant qu'attribut dans le balisage de rasoir. Par exemple une propriété `public string Title {get; set;}` peut recevoir une valeur comme `<widget title="my title">`
- Par défaut, tag helpers convertit les noms et les propriétés de la classe C # en cascade pour les helpers de tag en cas de kebab inférieur. Par exemple, si vous omettez d'utiliser

`[HtmlTargetElement]` et que le nom de la classe est `WidgetBoxTagHelper` , alors dans Razor, vous écrirez `<widget-box></widget-box>` .

- `Process` et `ProcessAsync` contiennent la logique de rendu. Les deux reçoivent un paramètre de **contexte** avec des informations sur la balise en cours de rendu et un paramètre de **sortie** utilisé pour personnaliser le résultat rendu.

Tout assembly contenant des aides de balises personnalisées doit être ajouté au fichier **_ViewImports.cshtml** (notez que c'est l'assembly en cours d'enregistrement, pas l'espace de noms):

```
@addTagHelper *, MyAssembly
```

Exemple de tag personnalisé

L'exemple suivant crée un assistant de balise de widget personnalisé qui ciblera le balisage de rasoir comme:

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

Qui sera rendu comme:

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

Le code nécessaire pour créer un tel assistant est le suivant:

```
[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);
```

```
    }
}
```

Assistant d'étiquette

Label Tag Helper peut être utilisé pour rendre une `label` pour une propriété de modèle. Il remplace la méthode `Html.LabelFor` dans les versions précédentes de MVC.

Disons que vous avez un modèle:

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

Dans la vue, vous pouvez utiliser l'élément HTML `label` et l'assistant de tag `asp-for` :

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

Ceci est équivalent au code suivant dans les versions antérieures de MVC:

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

Les deux extraits de code ci-dessus rendent le même code HTML:

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

Assistant de marquage d'ancre

L'assistant de balise d'ancrage est utilisé pour générer des attributs href afin d'établir un lien avec une action de contrôleur ou un itinéraire MVC particulier. Exemple de base

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

Parfois, vous devez spécifier des paramètres supplémentaires pour l'action du contrôleur à laquelle vous êtes lié. Nous pouvons spécifier des valeurs pour ces paramètres en ajoutant des attributs avec le préfixe asp-route.

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

Lire Tag Helpers en ligne: https://riptutorial.com/fr/asp-net-core/topic/2665/tag-helpers

Chapitre 25: Travailler avec JavascriptServices

Introduction

Selon la documentation officielle:

`JavaScriptServices` est un ensemble de technologies pour les développeurs ASP.NET Core. Il fournit l'infrastructure que vous trouverez utile si vous utilisez Angular 2 / React / Knockout / etc. sur le client, ou si vous créez vos ressources côté client à l'aide de Webpack ou si vous souhaitez exécuter JavaScript sur le serveur à l'exécution.

Examples

Activation de webpack-dev-middleware pour le projet asp.net-core

Disons que vous utilisez `Webpack` pour le regroupement frontal. Vous pouvez ajouter `webpack-dev-middleware` pour servir vos statistiques via un serveur minuscule et rapide. Il vous permet de recharger automatiquement vos actifs lorsque le contenu a changé, de servir des statiques en mémoire sans écrire en continu des versions intermédiaires sur le disque.

Conditions préalables

NuGet

Package d'installation Microsoft.AspNetCore.SpaServices

npm

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

Configuration

Étendre la méthode `Configure` dans votre classe de `Startup`

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

Ajouter le remplacement du module chaud (HMR)

Hot Module Replacement permet d'ajouter, de modifier ou de supprimer un module d'application lorsque l'application est en cours d'exécution. Le rechargement de page n'est pas nécessaire dans ce cas.

Conditions préalables

En plus des `webpack-dev-middleware` :

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

Configuration

`UseWebpackDevMiddleware` simplement à jour la configuration de `UseWebpackDevMiddleware` avec de nouvelles options:

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

Vous devez également accepter les modules chauds dans votre code d'application.

HMR est pris en charge pour Angular 2, React, Knockout et Vue.

Génération d'un exemple d'application d'une page avec asp.net core

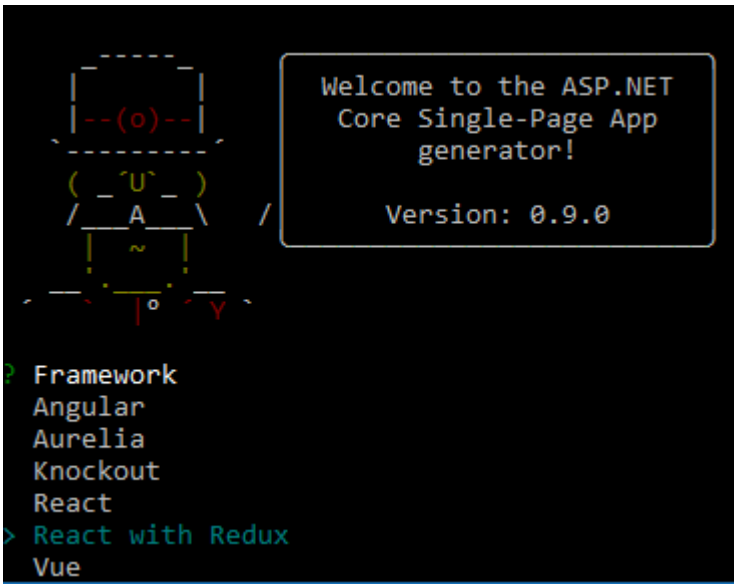
Vous pouvez utiliser le générateur `aspnetcore-spa` pour `Yeoman` pour créer une toute nouvelle application à une page avec asp.net core.

Cela vous permet de choisir l'un des frameworks frontaux les plus populaires et génère des projets avec webpack, dev server, remplacement de module à chaud et fonctionnalités de rendu côté serveur.

Juste courir

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

et choisissez votre cadre préféré



Lire Travailler avec JavascriptServices en ligne: https://riptutorial.com/fr/asp-net-core/topic/9621/travailler-avec-javascriptservices

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec 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	Afficher les composants	Daniel J.G.
3	Angular2 et .Net Core	Alejandro Tobón, Sentient Entities
4	ASP.NET Core - Journal à la fois de la demande et de la réponse à l'aide de middleware	Gubr
5	Autorisation	gilmishal, RamenChef
6	Configuration	Cyprien Autexier, Jayantha Lal Sirisena
7	Configuration de plusieurs environnements	dotnetom, Johnny, Robert Paulsen, Sanket, Set, Tseng
8	Demandes d'origine croisée (CORS)	Henk Mollema, Sanket, Saqib Rokadia, Tseng
9	Des modèles	Alex Logan, Ralf Bönning
10	Enregistrement	Dmitry, Sanket, Set, Tseng
11	Envoi d'e-mails dans les applications .Net Core à l'aide de MailKit	Ankit
12	Injection de dépendance	Alexan, BrunoLM, Cyprien Autexier, Dan Soper, Darren Evans, gilmishal, Gurgen Hakobyan, Jayantha Lal Sirisena, Joel Harkes, mazt, Tseng, Zach Becknell
13	Injection de services	Alex Logan, Rion Williams

	dans des vues	
14	La gestion des erreurs	Sanket, Set
15	Le routage	ChadT, Tseng
16	Limitation de débit	Stefan P.
17	Localisation	Tseng, VSG24, Zach Becknell
18	Middleware	Ali, Piotrek, Set, VSG24, Zach Becknell
19	Mise en cache	Cyprien Autexier, Sanket
20	project.json	Joel Harkes
21	Publication et déploiement	Set
22	Regroupement et Minification	Rion Williams, Zach Becknell
23	Sessions dans ASP.NET Core 1.0	ravindra, Sanket
24	Tag Helpers	Ali, Daniel J.G., dotnetom, Shyju, tmg, Zach Becknell
25	Travailler avec JavascriptServices	hmnzr