



FREE eBook

LEARNING asp.net-core

Free unaffiliated eBook created from
Stack Overflow contributors.

#asp.net-
core

Table of Contents

About.....	1
Chapter 1: Getting started with asp.net-core.....	2
Remarks.....	2
Versions.....	2
Examples.....	2
Installation and Setup.....	2
Installing Visual Studio.....	2
Creating an ASP.NET Core MVC Application.....	3
Create a new project from the command line.....	5
Minimal ASP.NET Core Web API with ASP.NET Core MVC.....	5
Controllers.....	6
Conclusion.....	7
Using Visual Studio code to develop Cross platform aspnet core application.....	7
Setup environment variable in ASP.NET Core [Windows].....	11
Chapter 2: Angular2 and .Net Core.....	16
Examples.....	16
Quick tutorial for an Angular 2 Hello World! App with .Net Core in Visual Studio 2015.....	16
Expected errors when generating Angular 2 components in .NET Core project (version 0.8.3).....	41
Chapter 3: ASP.NET Core - Log both Request and Response using Middleware.....	43
Introduction.....	43
Remarks.....	43
Examples.....	43
Logger Middleware.....	43
Chapter 4: Authorization.....	45
Examples.....	45
Simple Authorization.....	45
Chapter 5: Bundling and Minification.....	47
Examples.....	47
Grunt and Gulp.....	47
Bundler and Minifier Extension.....	48

Building Your Bundles.....	48
Minifying Your Bundles.....	49
Automate Your Bundles.....	49
The dotnet bundle Command.....	50
Using BundlerMinifier.Core.....	50
Configuring Your Bundles.....	50
Creating / Updating Bundles.....	51
Automated Bundling.....	51
Available Commands.....	51
Chapter 6: Caching.....	52
Introduction.....	52
Examples.....	52
Using InMemory cache in ASP.NET Core application.....	52
Distributed Caching.....	53
Chapter 7: Configuration.....	54
Introduction.....	54
Syntax.....	54
Examples.....	54
Accessing Configuration using Dependency Injection.....	54
Getting Started.....	54
Work with Environment Variables.....	55
Option model and configuration.....	56
In Memory configuration source.....	56
Chapter 8: Configuring multiple Environments.....	57
Examples.....	57
Having appsettings per Environment.....	57
Get/Check Environment name from code.....	57
Configuring multiple environments.....	58
Render environment specific content in view.....	60
Set environment variable from command line.....	60
Set environment variable from PowerShell.....	60
Using ASPNETCORE_ENVIRONMENT from web.config.....	60

Chapter 9: Cross-Origin Requests (CORS)	62
Remarks	62
Examples	62
Enable CORS for all requests	62
Enable CORS policy for specific controllers	62
More sophisticated CORS policies	63
Enable CORS policy for all controllers	63
Chapter 10: Dependency Injection	65
Introduction	65
Syntax	65
Remarks	65
Examples	66
Register and manually resolve	66
Register dependencies	66
Lifetime control	67
Enumerable dependencies	67
Generic dependencies	67
Retrieve dependencies on a Controller	68
Injecting a dependency into a Controller Action	68
The Options pattern / Injecting options into services	69
Remarks	70
Using scoped services during application startup / Database Seeding	70
Resolve Controllers, ViewComponents and TagHelpers via Dependency Injection	71
Plain Dependency Injection example (Without Startup.cs)	72
Inner workings of Microsoft.Extensions.DependencyInjection	72
IServiceCollection	72
IServiceProvider	73
Result	73
Chapter 11: Error Handling	75
Examples	75
Redirect to custom error page	75

Global Exception Handling in ASP.NET Core	75
Chapter 12: Injecting services into views	77
Syntax	77
Examples	77
The @inject Directive	77
Example Usage	77
Required Configuration	77
Chapter 13: Localization	78
Examples	78
Localization using JSON language resources	78
Set Request culture via url path	86
Middleware Registration	87
Custom Route Constraints	87
Registering the route	87
Chapter 14: Logging	89
Examples	89
Using NLog Logger	89
Add Logger to Controller	89
Using Serilog in ASP.NET core 1.0 application	89
Chapter 15: Middleware	91
Remarks	91
Examples	91
Using the ExceptionHandler middleware to send custom JSON error to Client	91
Middleware to set response ContentType	92
Pass data through the middleware chain	92
Run, Map, Use	93
Chapter 16: Models	95
Examples	95
Model Validation with Validation Attributes	95
Model Validation with Custom Attribute	95
Chapter 17: project.json	97

Introduction.....	97
Examples.....	97
Simple Library project example.....	97
Complete json file:.....	97
Simple startup project.....	100
Chapter 18: Publishing and Deployment.....	101
Examples.....	101
Kestrel. Configuring Listening Address.....	101
Chapter 19: Rate limiting.....	103
Remarks.....	103
Examples.....	103
Rate limiting based on client IP.....	103
Setup.....	103
Defining rate limit rules.....	106
Behavior.....	106
Update rate limits at runtime.....	107
Rate limiting based on client ID.....	108
Setup.....	108
Defining rate limit rules.....	111
Behavior.....	112
Update rate limits at runtime.....	113
Chapter 20: Routing.....	114
Examples.....	114
Basic Routing.....	114
Routing constraints.....	114
Using it on Controllers.....	114
Using it on Actions.....	115
Using it in Default Routes.....	115
Chapter 21: Sending Email in .Net Core apps using MailKit.....	116
Introduction.....	116
Examples.....	116
Installing nuget package.....	116

Simple implementation for sending emails.....	116
Chapter 22: Sessions in ASP.NET Core 1.0.....	118
Introduction.....	118
Examples.....	118
Basic example of handling Session.....	118
Chapter 23: Tag Helpers.....	120
Parameters.....	120
Examples.....	120
Form Tag Helper - Basic example.....	120
Form Tag Helper - With custom route attributes.....	120
Input Tag Helper.....	120
Select Tag Helper.....	121
Custom Tag Helper.....	123
Sample Widget Custom Tag Helper.....	124
Label Tag Helper.....	124
Anchor tag helper.....	125
Chapter 24: View Components.....	126
Examples.....	126
Create a View Component.....	126
Login View Component.....	126
Return from Controller Action.....	127
Chapter 25: Working with JavascriptServices.....	129
Introduction.....	129
Examples.....	129
Enabling webpack-dev-middleware for asp.net-core project.....	129
Prerequisites.....	129
NuGet.....	129
npm.....	129
Configuring.....	129
Add Hot Module Replacement (HMR).....	129
Prerequisites.....	130

Configuration	130
Generating sample single page application with asp.net core.....	130
Credits	132

About

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Chapter 1: Getting started with asp.net-core

Remarks

.NET Core is a general purpose development platform maintained by Microsoft and the .NET community on GitHub. It is cross-platform, supporting Windows, macOS and Linux, and can be used in device, cloud, and embedded/IoT scenarios.

The following characteristics best define .NET Core:

- Flexible deployment: Can be included in your app or installed side-by-side user- or machine-wide.
- Cross-platform: Runs on Windows, macOS and Linux; can be ported to other OSes. The supported Operating Systems (OS), CPUs and application scenarios will grow over time, provided by Microsoft, other companies, and individuals.
- Command-line tools: All product scenarios can be exercised at the command-line.
- Compatible: .NET Core is compatible with .NET Framework, Xamarin and Mono, via the .NET Standard Library.
- Open source: The .NET Core platform is open source, using MIT and Apache 2 licenses. Documentation is licensed under CC-BY. .NET Core is a .NET Foundation project.
- Supported by Microsoft: .NET Core is supported by Microsoft, per .NET Core Support

Versions

Version	Release Notes	Release Date
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 and Setup

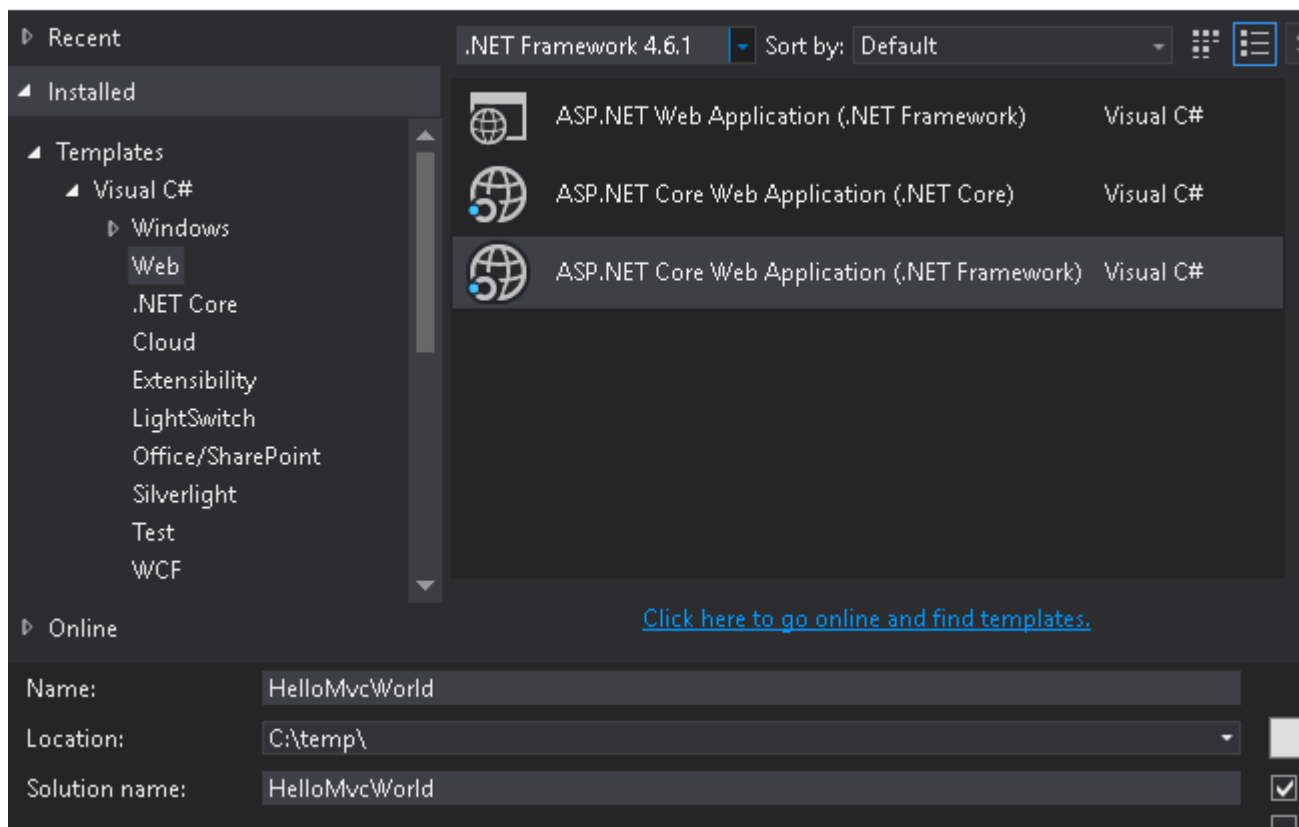
Installing Visual Studio

If you do not have Visual Studio installed, you can [download the free Visual Studio Community Edition here](#). If you already have it installed, you can proceed to the next step.

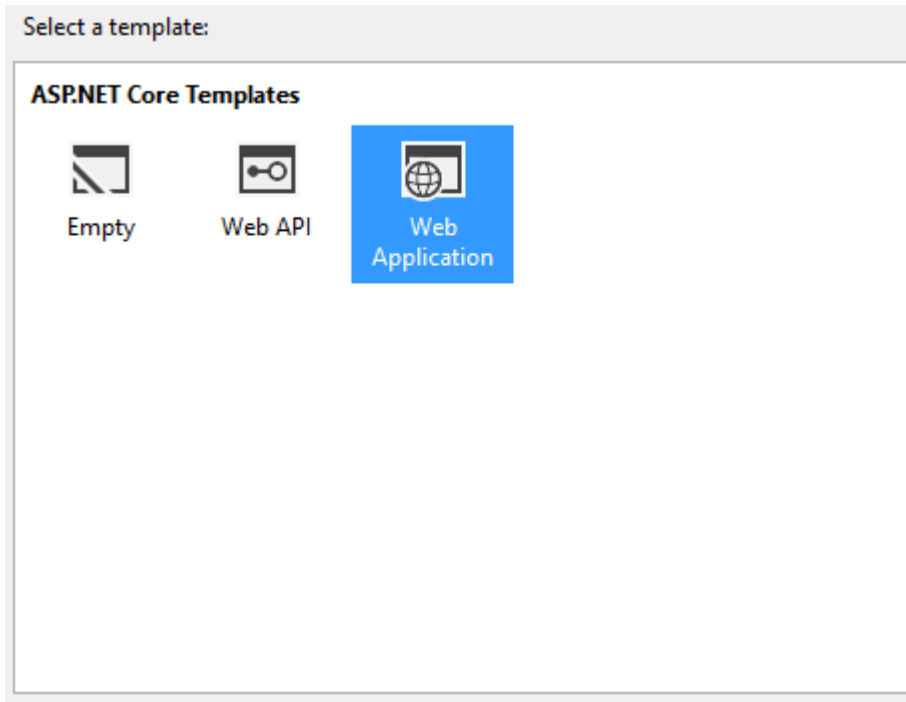
Creating an ASP.NET Core MVC Application.

1. **Open Visual Studio.**
2. **Select File > New Project.**
3. **Select Web** under the language of your choice within the Templates section on the left.
4. **Choose a preferred Project type** within the dialog.
5. **Optional: Choose a .NET Framework you would like to target**
6. **Name your project** and indicate if you want to create a Solution for the project.
7. **Click OK** to create the project.

New Project



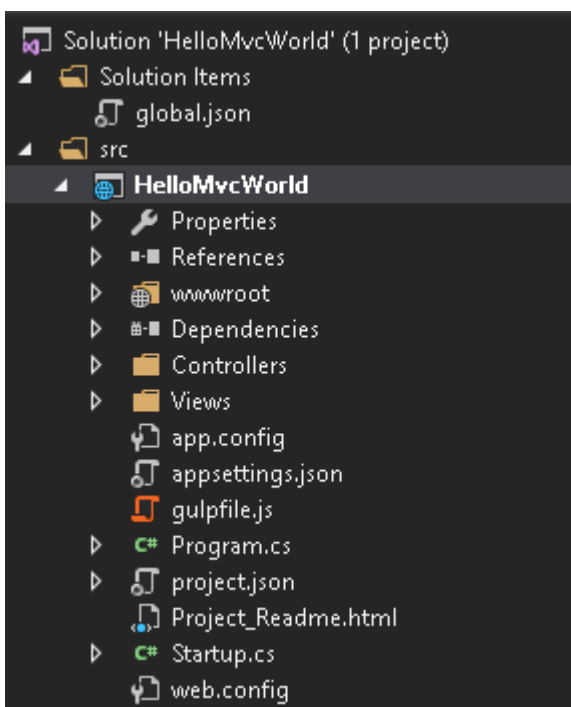
You will be presented with another dialog to select the template you want to use for the project :



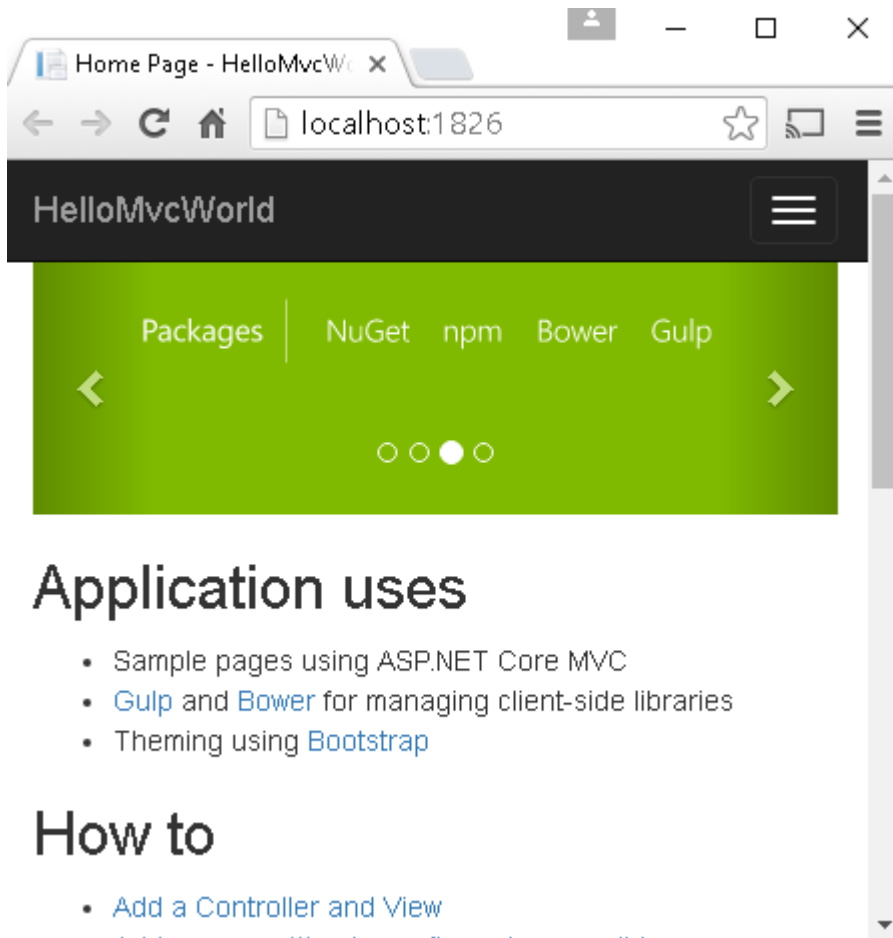
Each of the descriptions are self-explanatory. For this first project, **select Web Application**, which will contain all of the default configurations, authentication, and some existing content.

Since this is an introduction application and doesn't require any security or authentication, you can **change the authentication option to No Authentication** on the right-side of the dialog and **click OK to create the project**.

You should then see the new project within the Solution Explorer :



Press the F5 key to run the application and begin a debugging session, which will launch the application within your default browser :



You can now see that your project is up and running locally and is ready as a starting point for you to build your application.

Create a new project from the command line

It's possible to create a new ASP.NET Core project entirely from the command line using the `dotnet` command.

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

`dotnet new web` scaffolds a new "empty" web project. The `web` parameter tells the `dotnet` tool to use the ASP.NET Core Empty template. Use `dotnet new -all` to show all the available templates currently installed. Other key templates include `console`, `classlib`, `mvc` and `xunit`.

Once the template has been scaffolded out, you can restore the packages required to run the project (`dotnet restore`), and compile and start it (`dotnet run`).

Once the project is running, it will be available on the default port: <http://localhost:5000>

Minimal ASP.NET Core Web API with ASP.NET Core MVC

With ASP.NET Core 1.0, the MVC and Web API framework have been merged into one framework called ASP.NET Core MVC. This is a good thing, since MVC and Web API share a lot

of functionality, yet there always were subtle differences and code duplication.

However, merging these two into framework one also made it more difficult to distinguish one from another. For example, the `Microsoft.AspNet.WebApi` represents the Web API 5.x.x framework, not the new one. But, when you include `Microsoft.AspNetCore.Mvc` (version 1.0.0), you get the full blown package. This will contain *all* the out-of-the-box features the MVC framework offers. Such as Razor, tag helpers and model binding.

When you just want to build a Web API, we don't need all this features. So, how do we build a minimalistic Web API? The answer is: `Microsoft.AspNetCore.Mvc.Core`. In the new world MVC is split up into multiple packages and this package contains just the core components of the MVC framework, such as routing and authorization.

For this example, we're gonna create a minimal MVC API. Including a JSON formatter and CORS. Create an empty ASP.NET Core 1.0 Web Application and add these packages to your `project.json`:

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

Now we can register MVC using `AddMvcCore()` in the startup class:

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

`AddMvcCore` returns an `IMvcCoreBuilder` instance which allows further building. Configuring the middleware is the same as usual:

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

Controllers

The 'old' Web API comes with its own controller base class: `ApiController`. In the new world there is no such thing, only the default `Controller` class. Unfortunately, this is a rather large base class and it's tied to model binding, views and JSON.NET.

Fortunately, in the new framework controller classes don't have to derive from `Controller` to be picked up by the routing mechanism. Just appending the name with `Controller` is enough. This

allows us to build our own controller base class. Let's call it `ApiController`, just for old times sake:

```
/// <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;
}
```

The `[Controller]` attribute indicates that the type or any derived type is considered as a controller by the default controller discovery mechanism. The `[ActionContext]` attribute specifies that the property should be set with the current `ActionContext` when MVC creates the controller. The `ActionContext` provides information about the current request.

ASP.NET Core MVC also offers a `ControllerBase` class which provides a controller base class just without views support. It's still much larger than ours though. Use it if you find it convenient.

Conclusion

We can now build a minimal Web API using the new ASP.NET Core MVC framework. The modular package structure allows us to just pull in the packages we need and create a lean and simple application.

Using Visual Studio code to develop Cross platform aspnet core application

With `AspNetCore` you can develop the application on any platform including Mac, Linux, Windows and Docker.

Installation and SetUp

1. Install visual Studio Code from [here](#)
2. Add `C# extension`
3. Install dot net core sdk. You can install from [here](#)

Now you have all the tools available. To develop the application. Now you need some scaffolding option. For that you should consider using Yeoman. To install Yeoman

1. Install NPM. For this you need Node on your machine. Install from [here](#)

2. Install Yeoman by using NPM

```
npm install -g yo
```

3. Now install the aspnet generator

```
npm install -g generator-aspnet
```

Now we have all the setup on your machine. First let's create a new project with DotNetCore basic command and then create a new project using Yo.

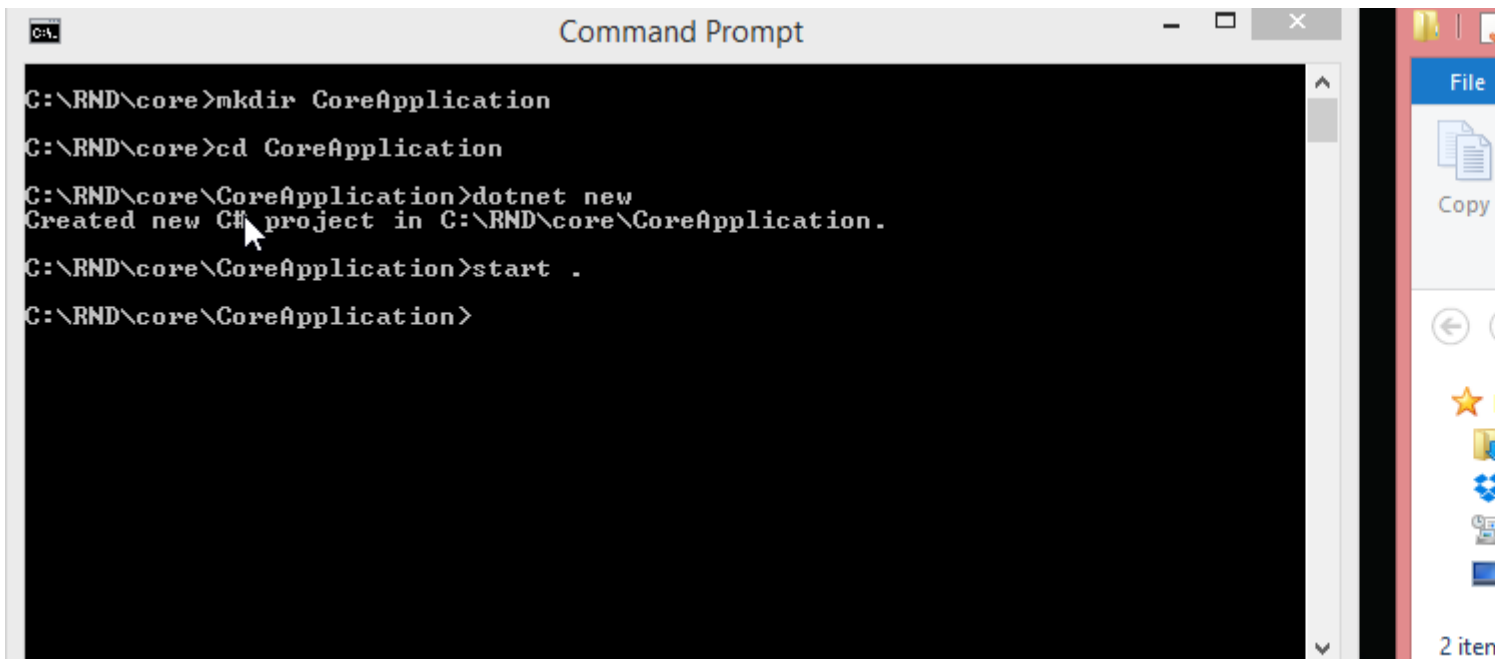
New Project Using Command Line

1. Create a new Project Folder

```
mkdir CoreApplication cd CoreApplication
```

2. Scaffold a very basic dotnet project using default command line option

```
dotnet New
```



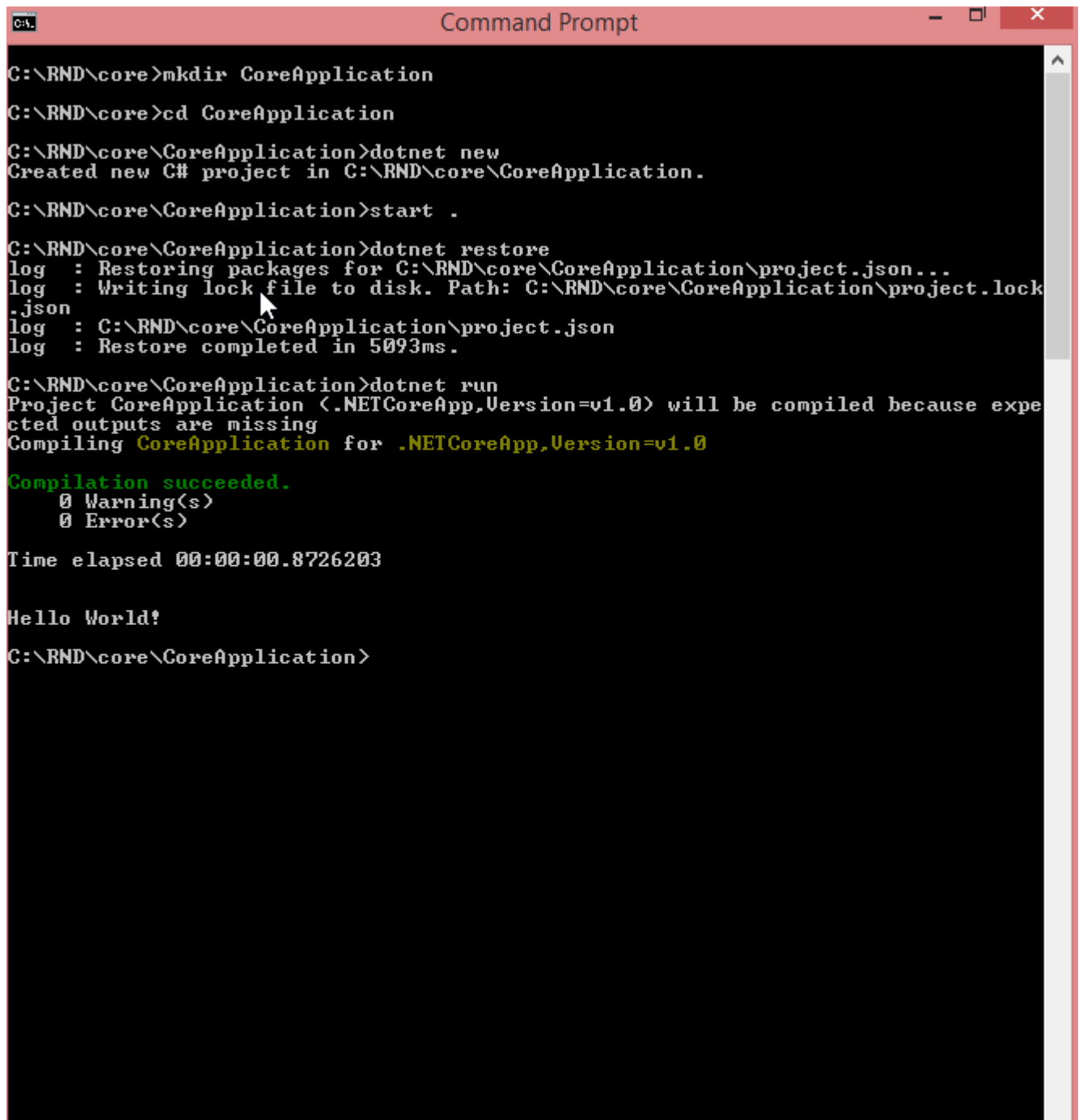
The screenshot shows a Windows Command Prompt window with the following commands and output:

```
C:\RND\core>mkdir CoreApplication
C:\RND\core>cd CoreApplication
C:\RND\core\CoreApplication>dotnet new
Created new C# project in C:\RND\core\CoreApplication.
C:\RND\core\CoreApplication>start .
C:\RND\core\CoreApplication>
```

The window title is "Command Prompt". The background is black with white text. The output of the 'dotnet new' command shows a confirmation message. The 'start .' command is entered but not yet executed.

1. Restore the packages and run the application

```
dotNet restore dotnet run
```

```
C:\RND>mkdir CoreApplication
C:\RND>cd CoreApplication
C:\RND\CoreApplication>dotnet new
Created new C# project in C:\RND\CoreApplication.
C:\RND\CoreApplication>start .
C:\RND\CoreApplication>dotnet restore
log : Restoring packages for C:\RND\CoreApplication\project.json...
log : Writing lock file to disk. Path: C:\RND\CoreApplication\project.lock
.json
log : C:\RND\CoreApplication\project.json
log : Restore completed in 5093ms.
C:\RND\CoreApplication>dotnet run
Project CoreApplication (.NETCoreApp,Version=v1.0) will be compiled because expected outputs are missing
Compiling CoreApplication for .NETCoreApp,Version=v1.0
Compilation succeeded.
    0 Warning(s)
    0 Error(s)
Time elapsed 00:00:00.8726203
Hello World!
C:\RND\CoreApplication>
```

Use Yeoman as Scaffolding Option

Create Project Folder and Run the Yo Command

```
yo aspnet
```

Yeoman will ask some inputs like Project Type, Project Name etc like

```
C:\RND\core>mkdir YoCoreProject
C:\RND\core>cd YoCoreProject
C:\RND\core\YoCoreProject>yo aspnet

  _____
  <--(o)-->
  <  'U'  >
  <  A    >
  <  ~    >
  <  _    >
  <  _o_  >
  <  _y_  >

? _____?
| Welcome to the |
| marvellous ASP.NET Core |
| generator!     |
? _____?

? What type of application do you want to create? (Use arrow keys)
> Empty Web Application
  Console Application
  Web Application
  Web Application Basic [without Membership and Authorization]
  Web API Application
  Class Library
  Unit test project (<xUnit.net>)
```

```
Command Prompt

C:\RND\core\YoCoreProject>yo aspnet

  _____
  <--(o)-->
  <  'U'  >
  <  A    >
  <  ~    >
  <  _    >
  <  _o_  >
  <  _y_  >

? _____?
| Welcome to the |
| marvellous ASP.NET Core |
| generator!     |
? _____?

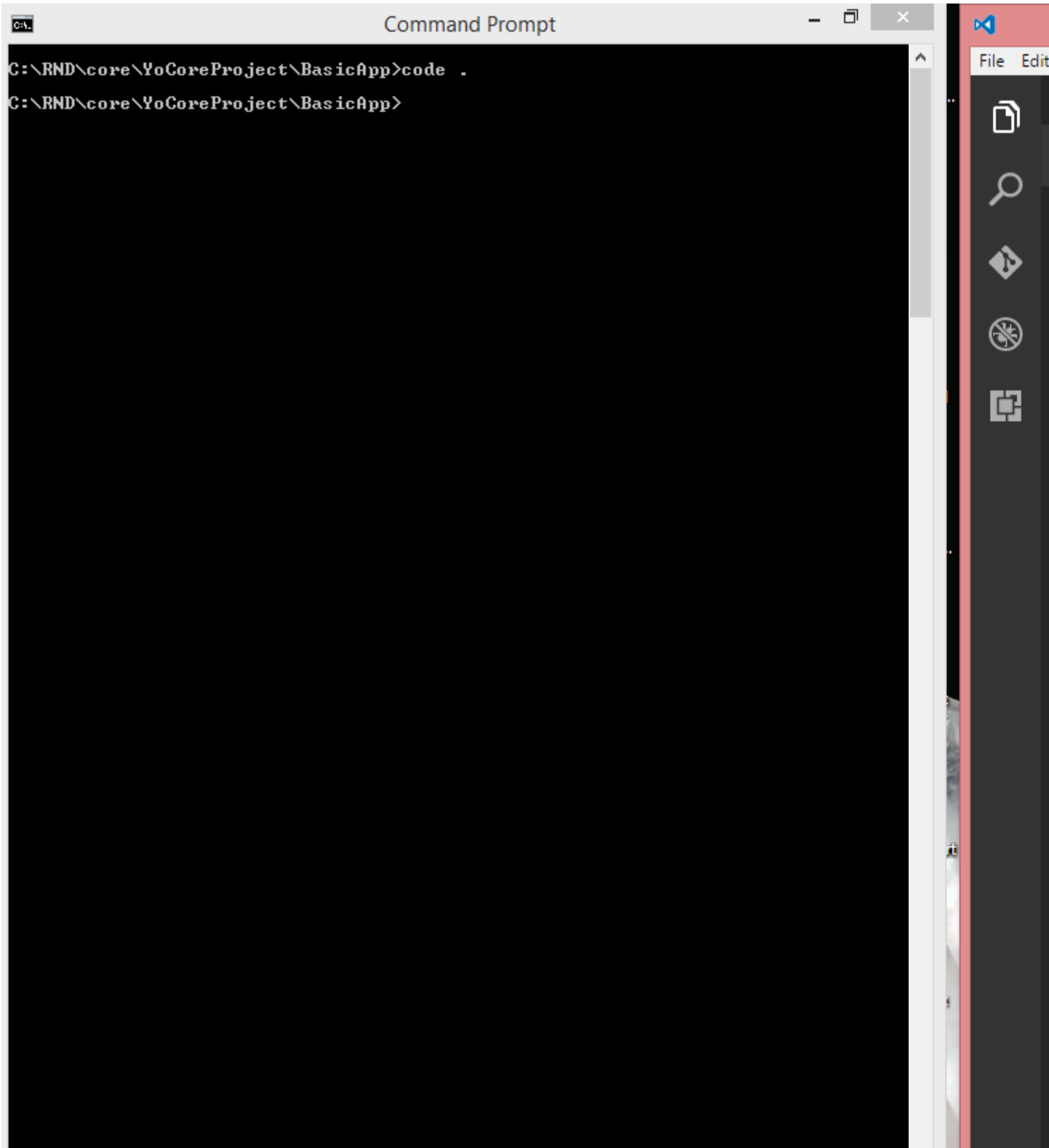
? What type of application do you want to create? Web API Application
? What's the name of your ASP.NET application? BasicApp
create BasicApp\.gitignore
create BasicApp\appsettings.json
create BasicApp\Dockerfile
create BasicApp\Startup.cs
create BasicApp\Program.cs
create BasicApp\project.json
create BasicApp\Properties\launchSettings.json
create BasicApp\Controllers\ValuesController.cs
create BasicApp\web.config
create BasicApp\README.md

Your project is now created, you can use the following commands to get going
cd "BasicApp"
dotnet restore
dotnet build (optional, build will also happen when it's run)
dotnet run
```

Now restore the packages by running dotnet restore command and Run the application

Use VS Code to develop the application

Run the visual studio code like



Now open the files and run the application. You can also search the extension for your help.

Setup environment variable in ASP.NET Core [Windows]

=> [Original Post](#) <=

ASP.NET Core uses the `ASPNETCORE_ENVIRONMENT` environment variable to determine the current environment. By default, if you run your application without setting this value, it will automatically

default to the `Production` environment.

```
> 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.
```

Setting the environment variable in Windows

At the command line

You can easily set an environment variable from a command prompt using the `setx.exe` command included in Windows. You can use it to easily set a user variable:

```
>setx ASPNETCORE_ENVIRONMENT "Development"

SUCCESS: Specified value was saved.
```

Note that the environment variable is not set in the current open window. You will need to open a new command prompt to see the updated environment. It is also possible to set system variables (rather than just user variables) if you open an administrative command prompt and add the `/M` switch:

```
>setx ASPNETCORE_ENVIRONMENT "Development" /M

SUCCESS: Specified value was saved.
```

Using PowerShell Alternatively, you can use PowerShell to set the variable. In PowerShell, as well as the normal user and system variables, you can also create a temporary variable using the `$Env:` command:

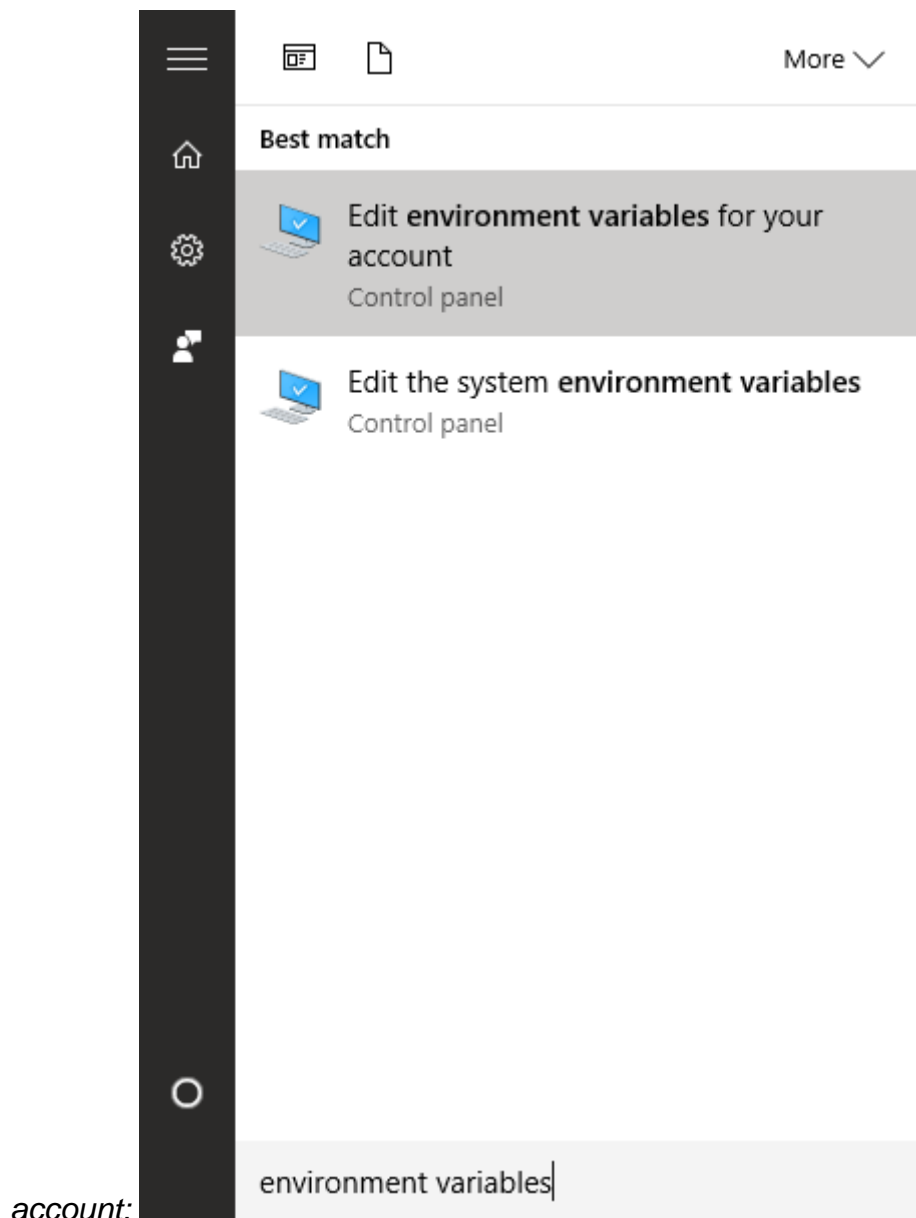
```
$Env:ASPNETCORE_ENVIRONMENT = "Development"
```

The variable created lasts just for the duration of your PowerShell session - once you close the window the environment reverts back to its default value.

Alternatively, you could set the user or system environment variables directly. This method does not change the environment variables in the current session, so you will need to open a new PowerShell window to see your changes. As before, changing the system (Machine) variables will require administrative access

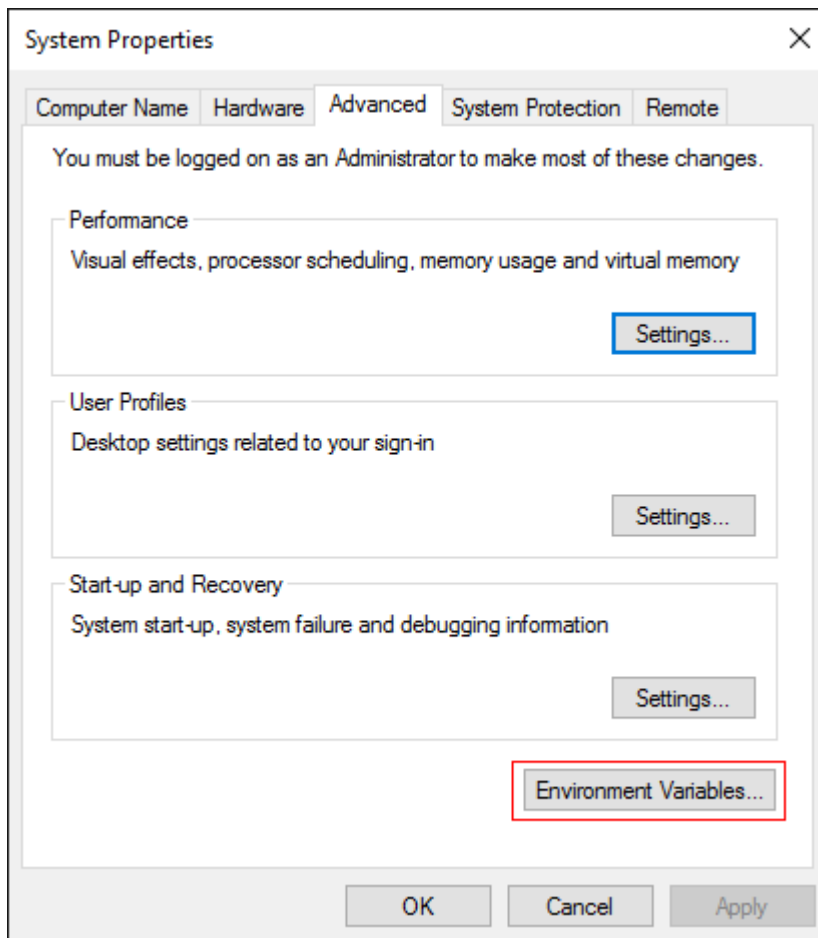
```
[Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Development", "User")
[Environment]::SetEnvironmentVariable("ASPNETCORE_ENVIRONMENT", "Development", "Machine")
```

Using the windows control panel If you're not a fan of the command prompt, you can easily update your variables using your mouse! Click the windows start menu button (or press the Windows key), search for `environment variables`, and choose *Edit environment variables for your*

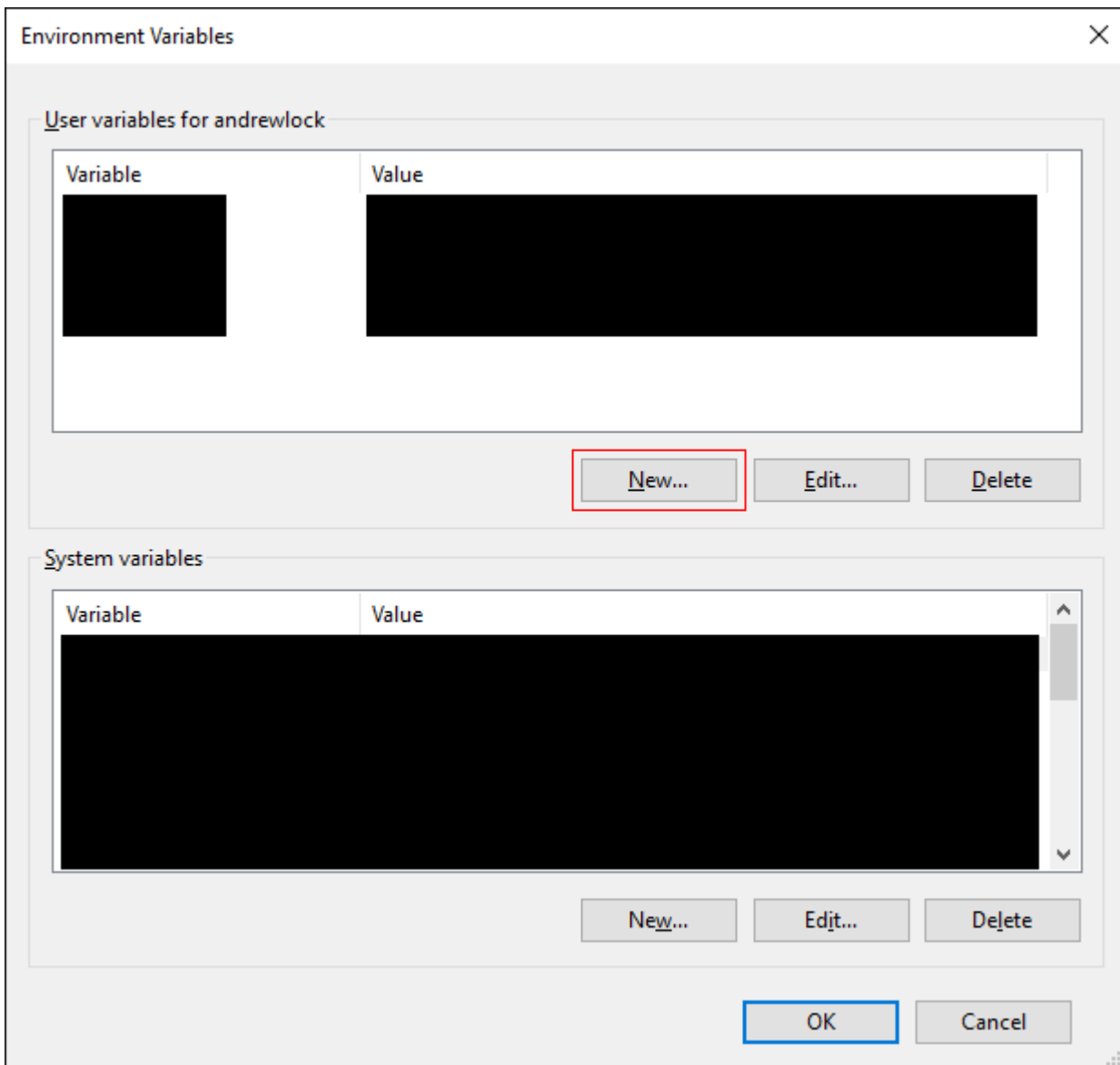


account:

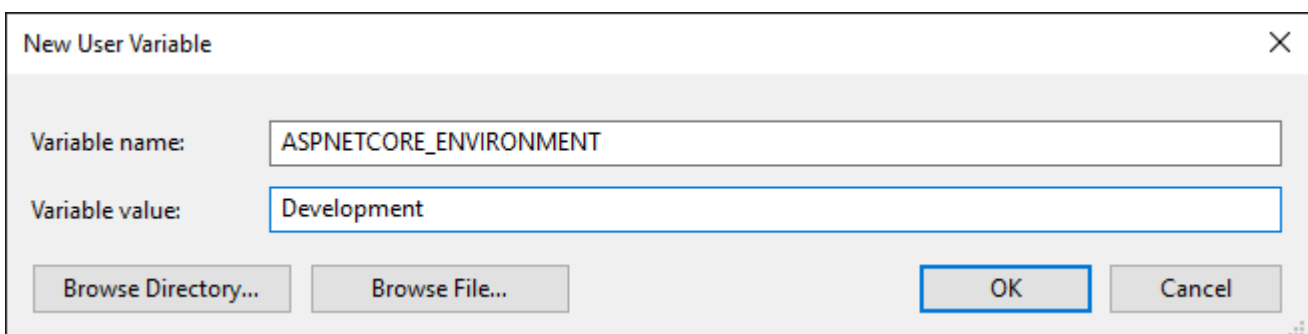
Selecting this option will open the System Properties dialog



Click Environment Variables to view the list of current environment variables on your system.



Assuming you do not already have a variable called `ASPNETCORE_ENVIRONMENT`, click the `New...` button and add a new account environment variable:



Click

OK to save all your changes. You will need to re-open any command windows to ensure the new environment variables are loaded.

Read *Getting started with asp.net-core* online: <https://riptutorial.com/asp-net-core/topic/810/getting-started-with-asp-net-core>

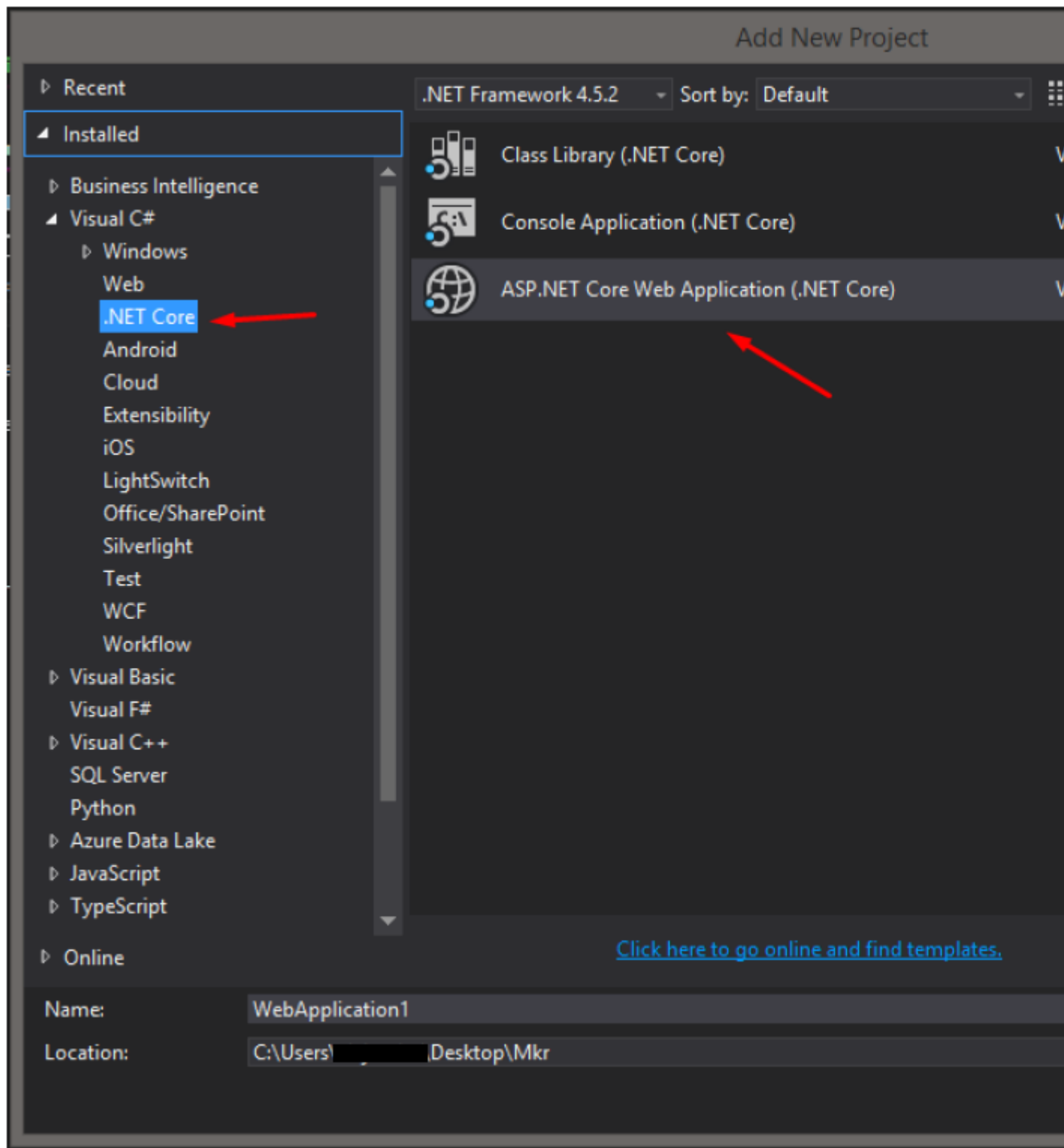
Chapter 2: Angular2 and .Net Core

Examples

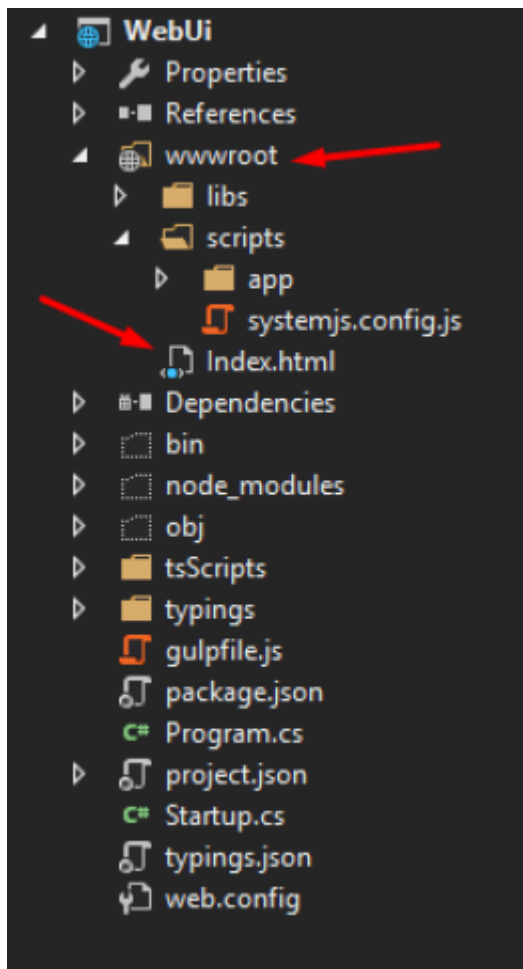
Quick tutorial for an Angular 2 Hello World! App with .Net Core in Visual Studio 2015

Steps:

1. Create Empty .Net Core Web App:

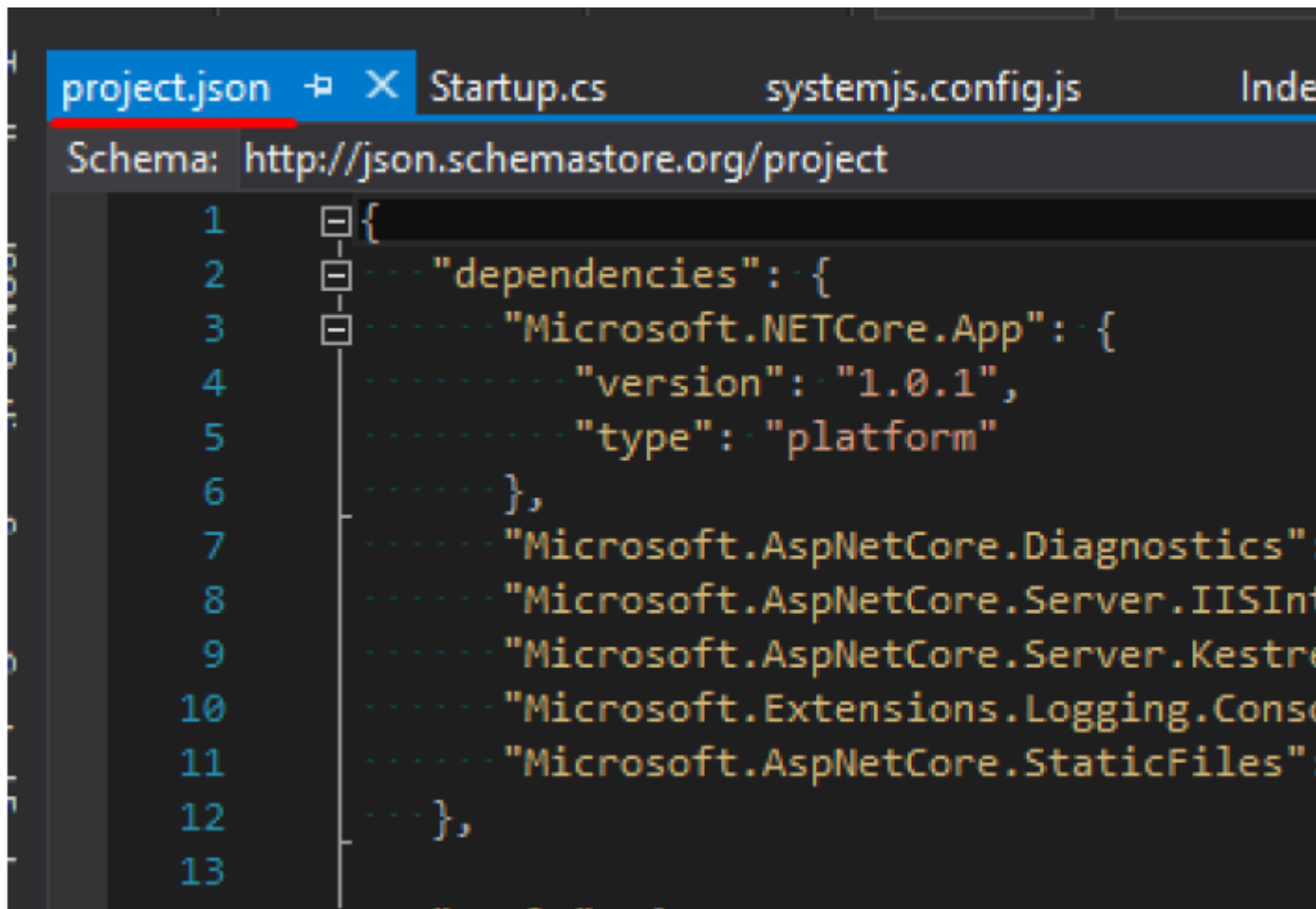


2. Go to wwwroot, and create a normal html page called Index.html:



3. Configure Startup.cs to accept static files (this will require to add "Microsoft.AspNetCore.StaticFiles": "1.0.0" library in the "project.json" file):

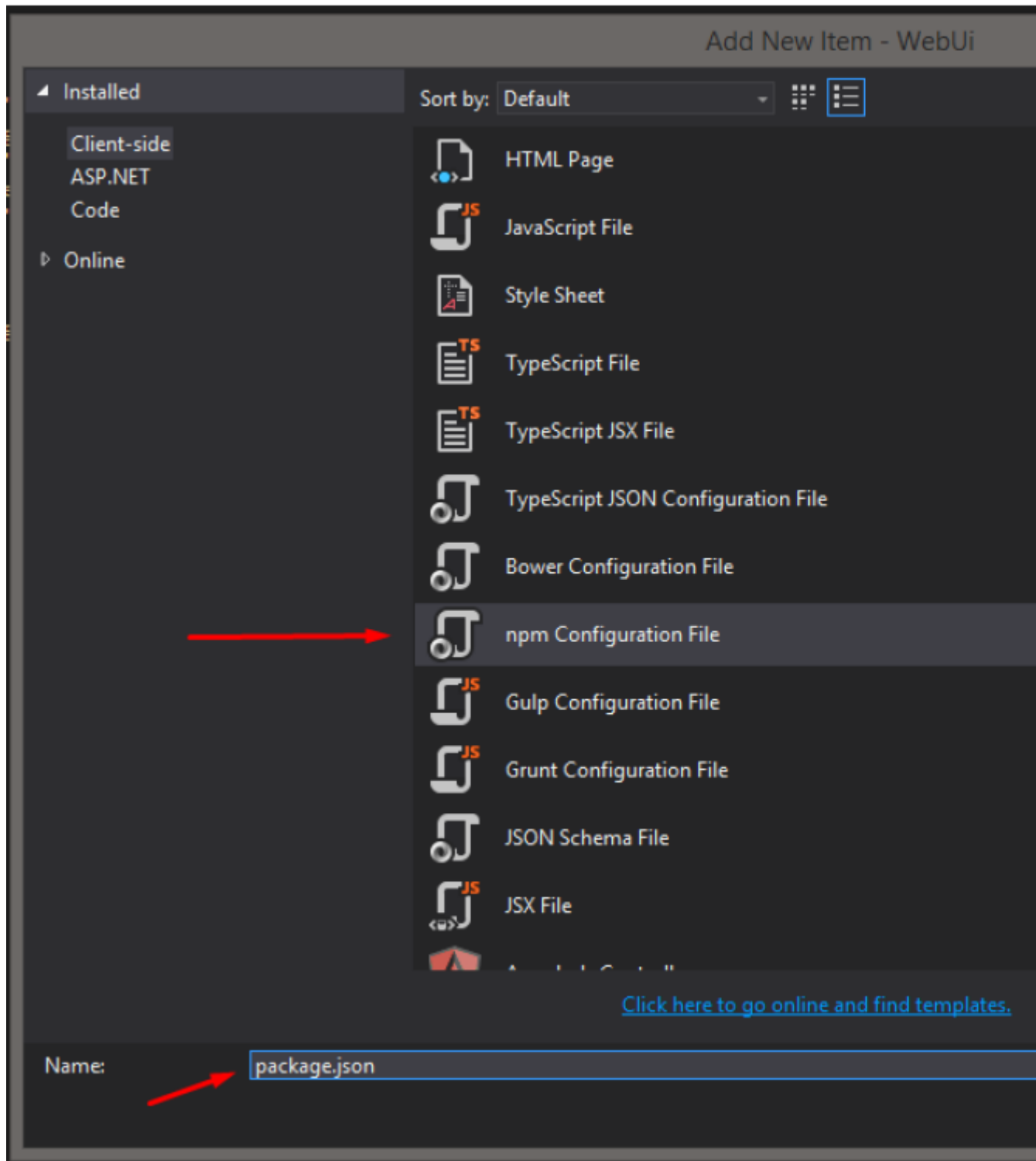
```
// This method gets called by the runtime. Use this method to configure the
- references
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILogger
{
    app.UseDefaultFiles();
    app.UseStaticFiles();
}
```



```
1  {
2    "dependencies": {
3      "Microsoft.NETCore.App": {
4        "version": "1.0.1",
5        "type": "platform"
6      },
7      "Microsoft.AspNetCore.Diagnostics": {
8        "version": "1.0.1",
9        "type": "platform"
10     },
11     "Microsoft.AspNetCore.Server.IISIntegration": {
12       "version": "1.0.1",
13       "type": "platform"
14     },
15     "Microsoft.AspNetCore.Server.Kestrel": {
16       "version": "1.0.1",
17       "type": "platform"
18     },
19     "Microsoft.Extensions.Logging.Console": {
20       "version": "1.0.1",
21       "type": "platform"
22     },
23     "Microsoft.AspNetCore.StaticFiles": {
24       "version": "1.0.1",
25       "type": "platform"
26     }
27   }
28 }
```

4. Add NPN File:

- Right click the WebUi project and add NPN Configuration File (package.json):



- Verify the last versions of the packages:

```

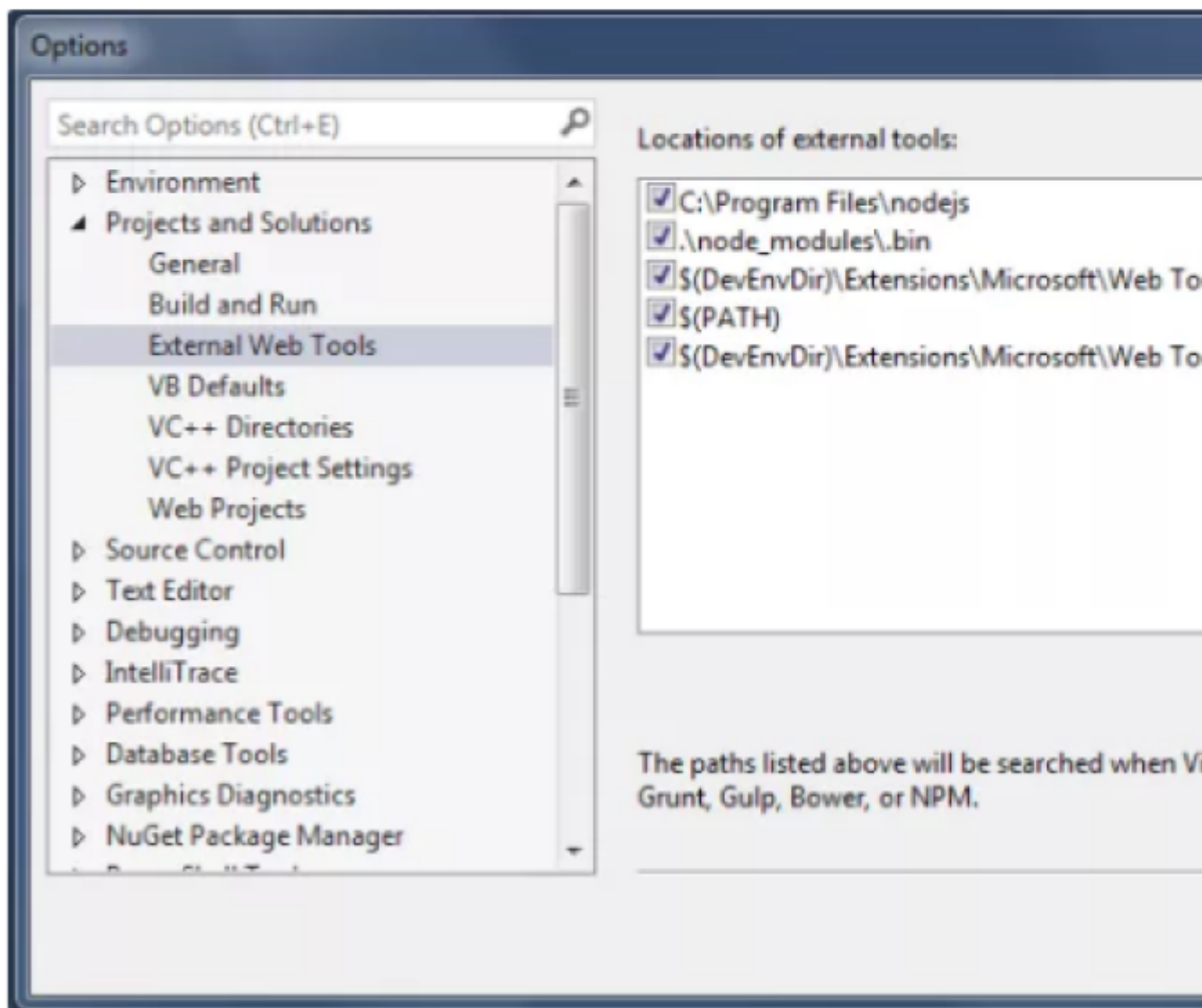
{
  "version": "1.0.0",
  "name": " ",
  "private": true,
  "author": " ",
  "description": " ",
  "scripts": {
    "postinstall": "typings install",
    "typings": "typings"
  },
  "dependencies": {
    "@angular/common": "~2.2.0",
    "@angular/compiler": "~2.2.0",
    "@angular/core": "~2.2.0",
    "@angular/forms": "~2.2.0",
    "@angular/http": "~2.2.0",
    "@angular/platform-browser": "~2.2.0",
    "@angular/platform-browser-dynamic": "~2.2.0",
    "@angular/router": "~3.2.0",
    "core-js": "^2.4.1",
    "reflect-metadata": "^0.1.9",
    "es6-shim": "^0.35.2",
    "rxjs": "5.0.3",
    "systemjs": "0.19.43",
    "zone.js": "^0.7.6",
    "bootstrap": "^3.3.7"
  },
  "devDependencies": {
    "typescript": "^2.1.5",
    "typings": "^2.1.0",
    "gulp": "^3.9.1",
    "gulp-clean": "^0.3.2",
    "gulp-typescript": "^3.1.4"
  }
}

```

Note: If visual studio does not detect the versions of the packages (Check all packages, because some of them does show the version, and some others don't), it might be because the Node version coming in visual studio is not working correctly, so it will probably require to install node js externally and then link that installation with visual studio.

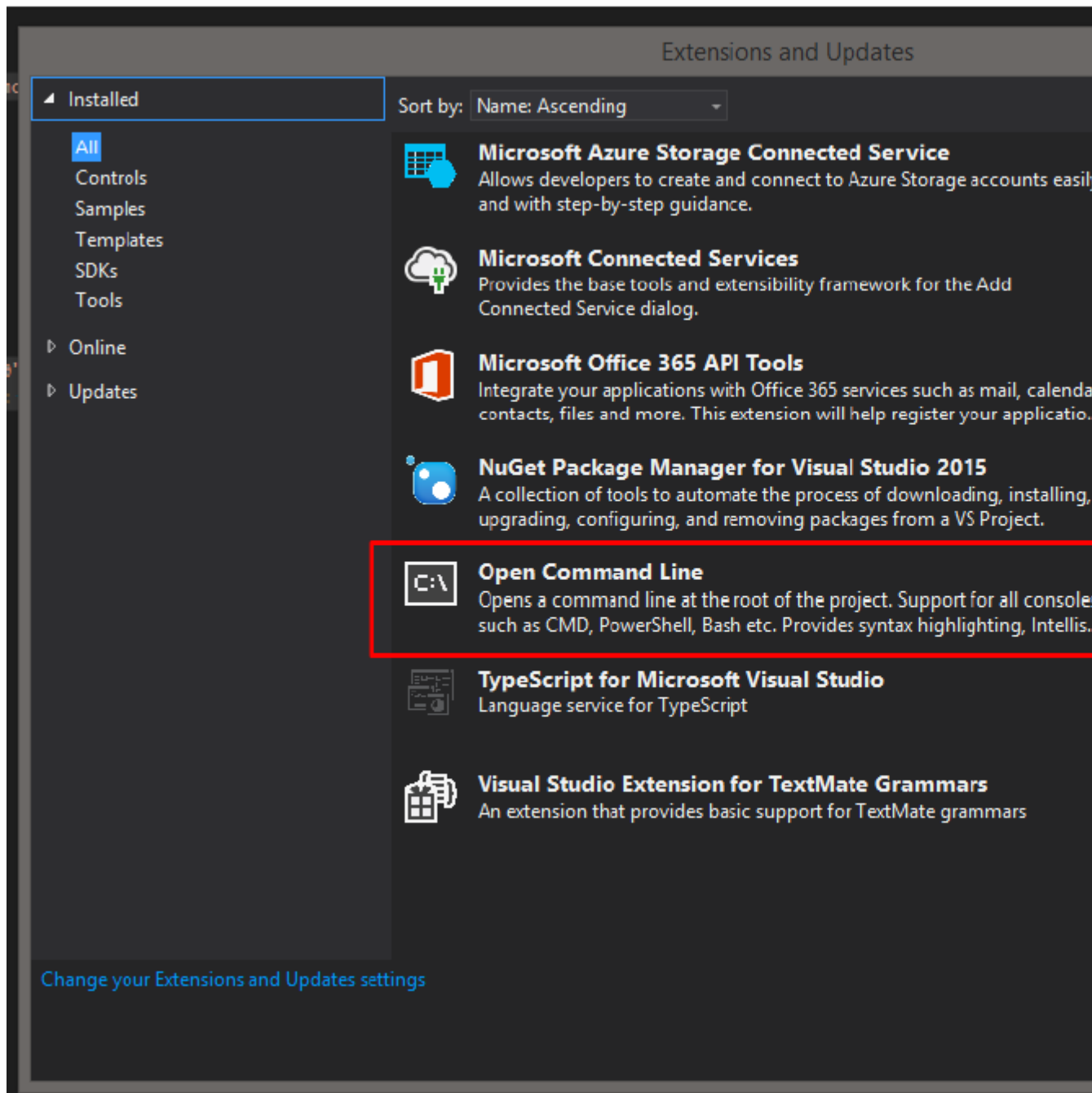
- i. Download and install node js: <https://nodejs.org/es/download/>
- ii. Link the installation with visual studio: <https://ryanhayes.net/synchronize-node-js-install-version-with-visual-studio-2015/>:

1. First, find the Node.js installation you already have and use it. By default, Node.js 0.12.7 installs to "C:\Program Files\nodejs".
2. Once you've got that all copied out to your clipboard, go to Visual Studio 2015.
3. In this dialog, go to **Projects and Solutions > External Web Tools** that manages all of the 3rd party tools used within VS. The screenshot is pointed to.
4. Add an entry at the top to the path to the node.js directory and use that version instead.



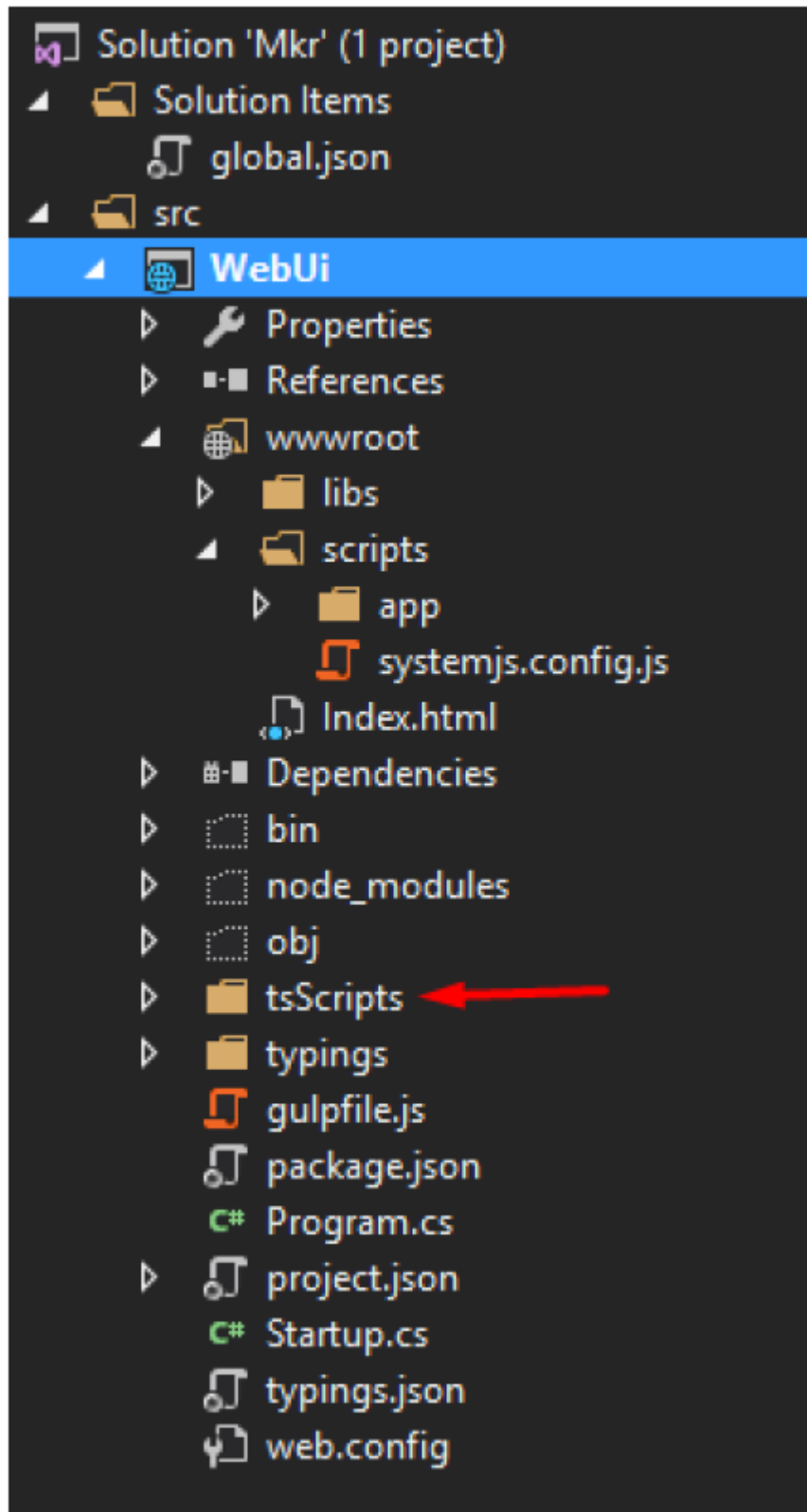
iii. (Optional) after saving the package.json it will install the dependencies in the project, if not, run "npm install" using a command prompt from the same location as the package.json file.

Note: Recommended to install "Open Command Line", an extension that can be added to Visual Studio:

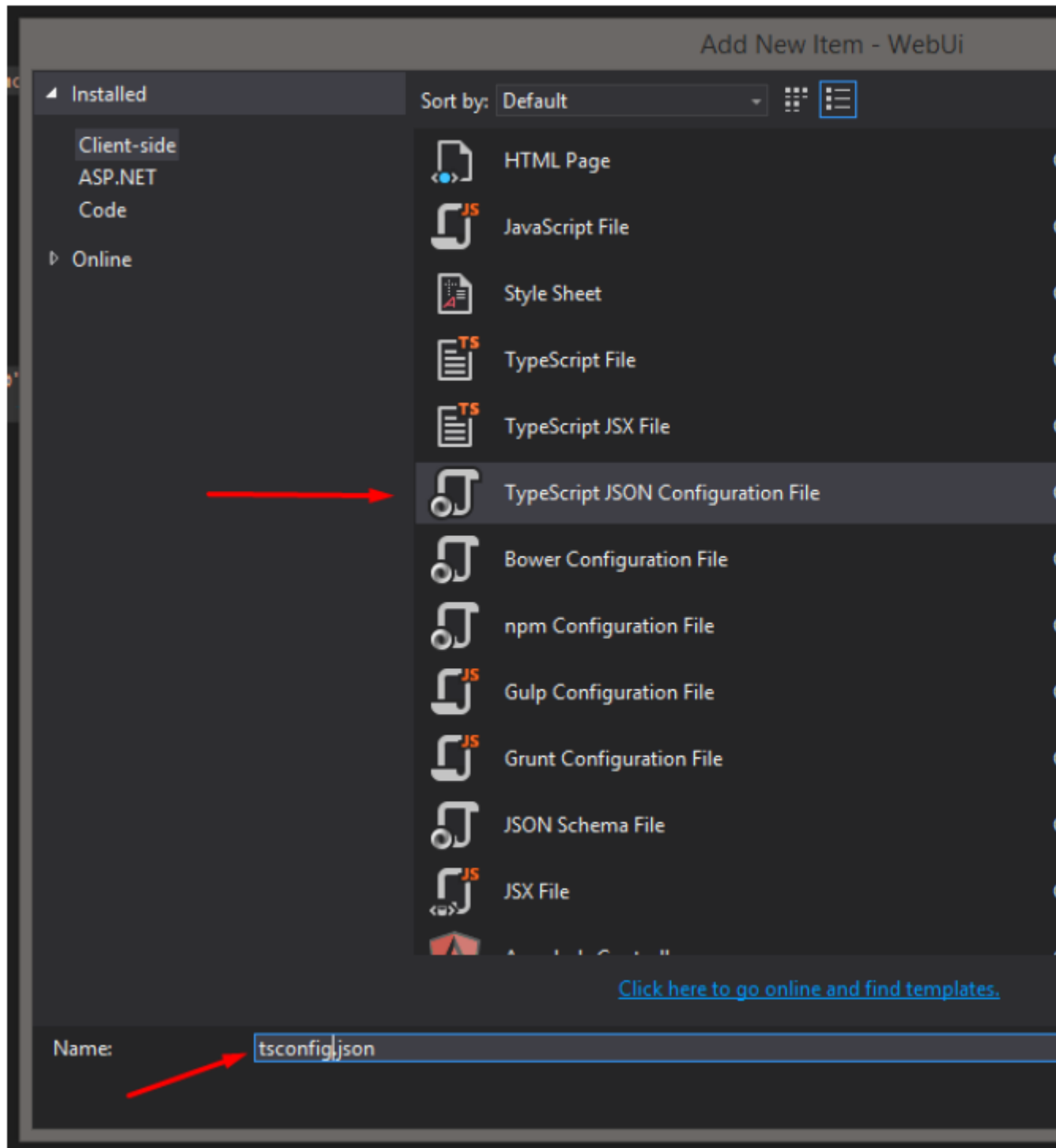


5. Add typescript:

- Create a TypeScript folder inside the WebUi project, just for organization (The TypeScript files won't go to the browser, they will be transpiled into a normal JS file, and this JS file will be the one going to the wwwroot folder using gulp, this will be explained later):



- Inside that folder add "TypeScript JSON Configuration File" (tsconfig.json):



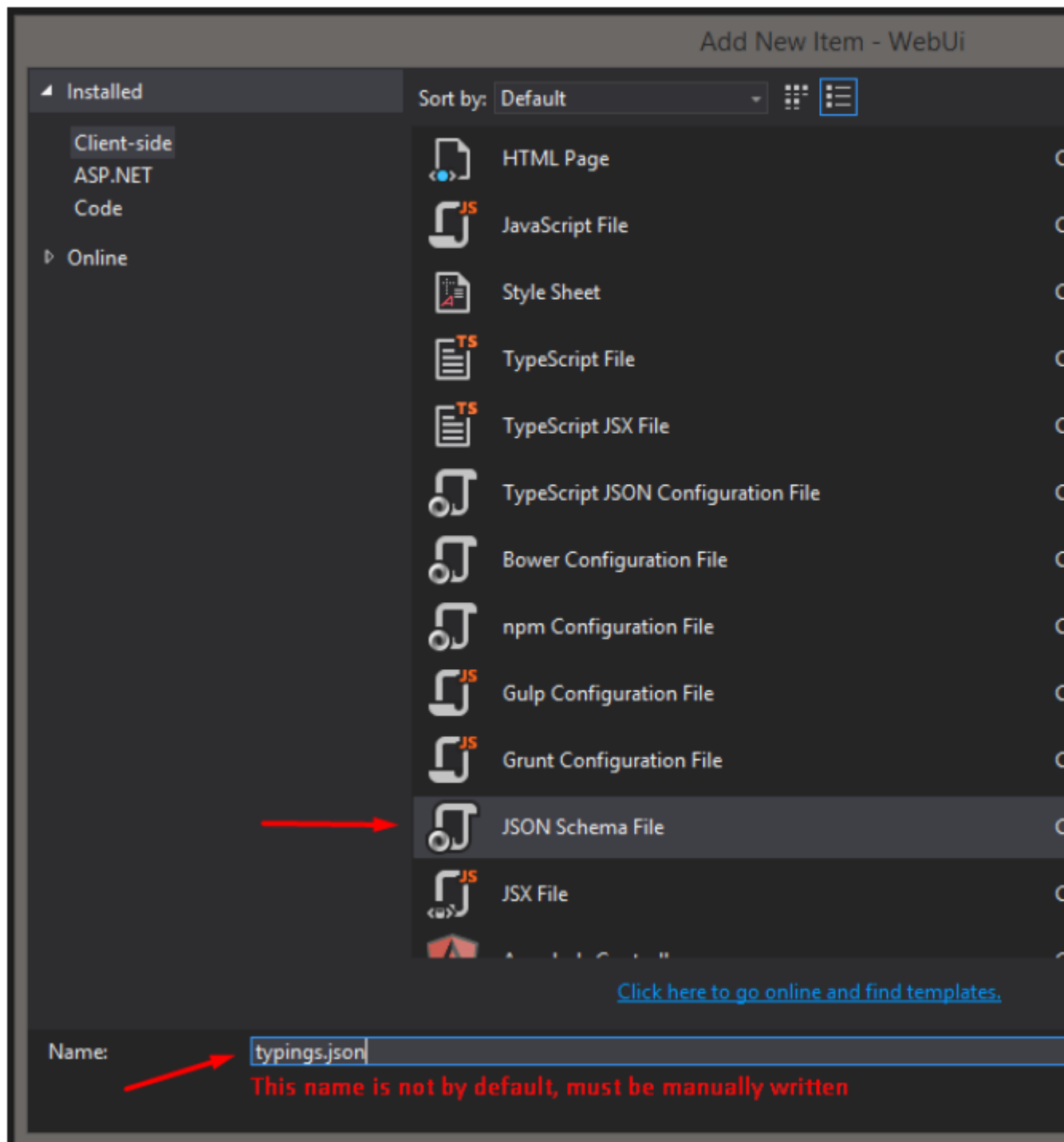
And add the next code:

```

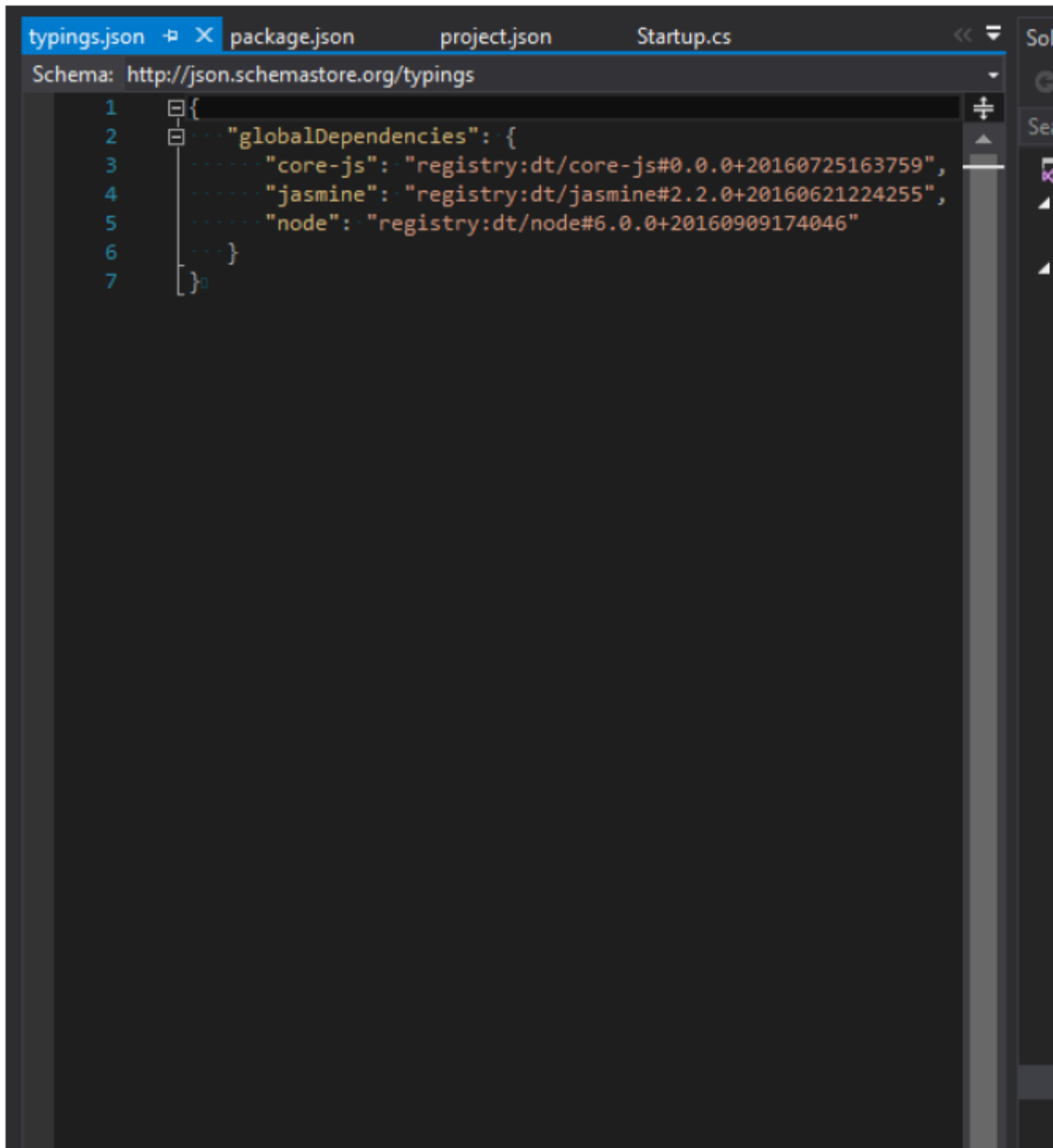
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es6",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "module": "commonjs",
    "outDir": "../wwwroot/scripts/",
    "moduleResolution": "node"
  },
  "exclude": [
    "node_modules",
    "wwwroot",
    "typings/index",
    "typings/index.d.ts"
  ]
}

```

- In the WebUi Project's root, add a new file called typings.json:

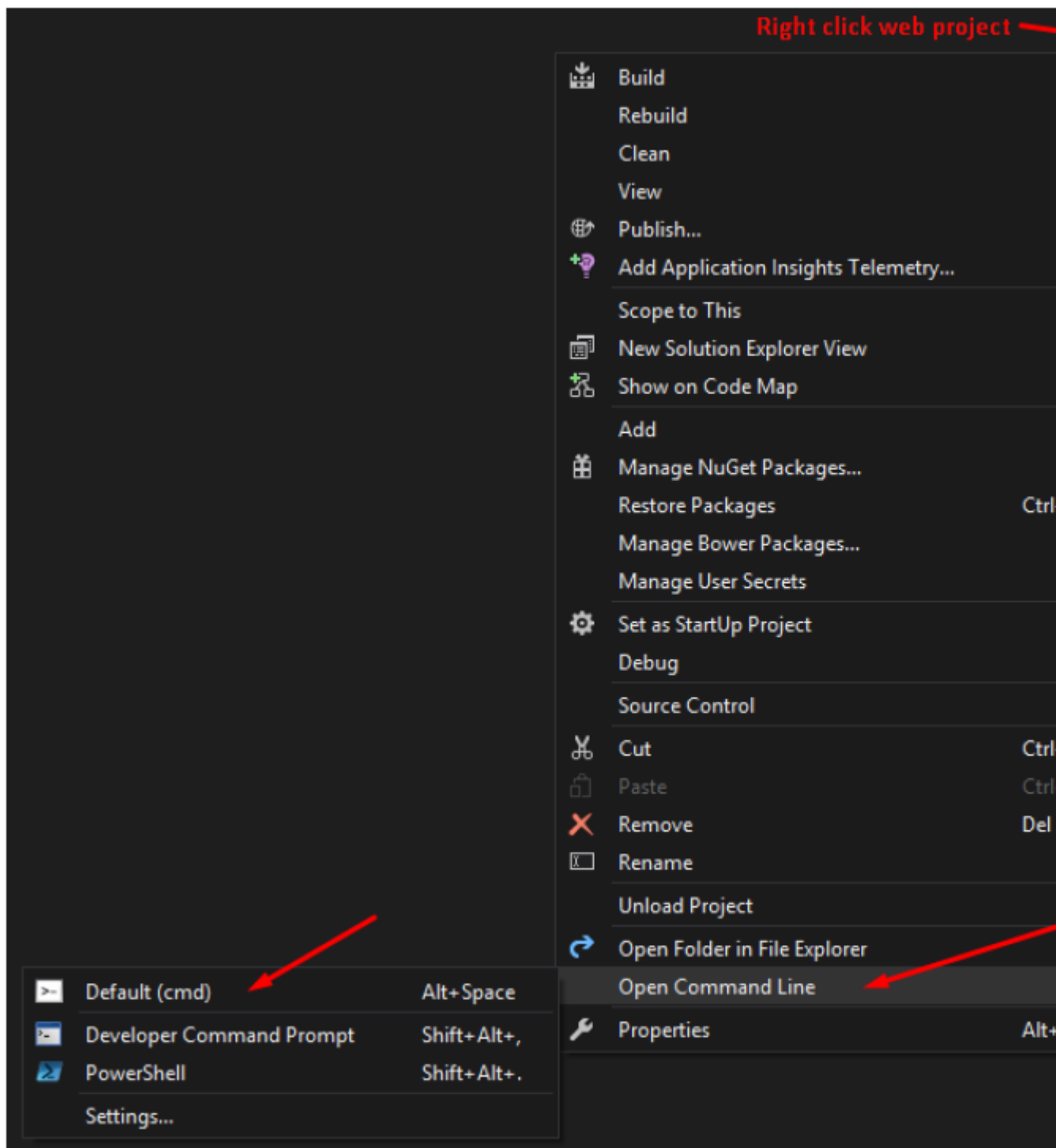


And add the next code:

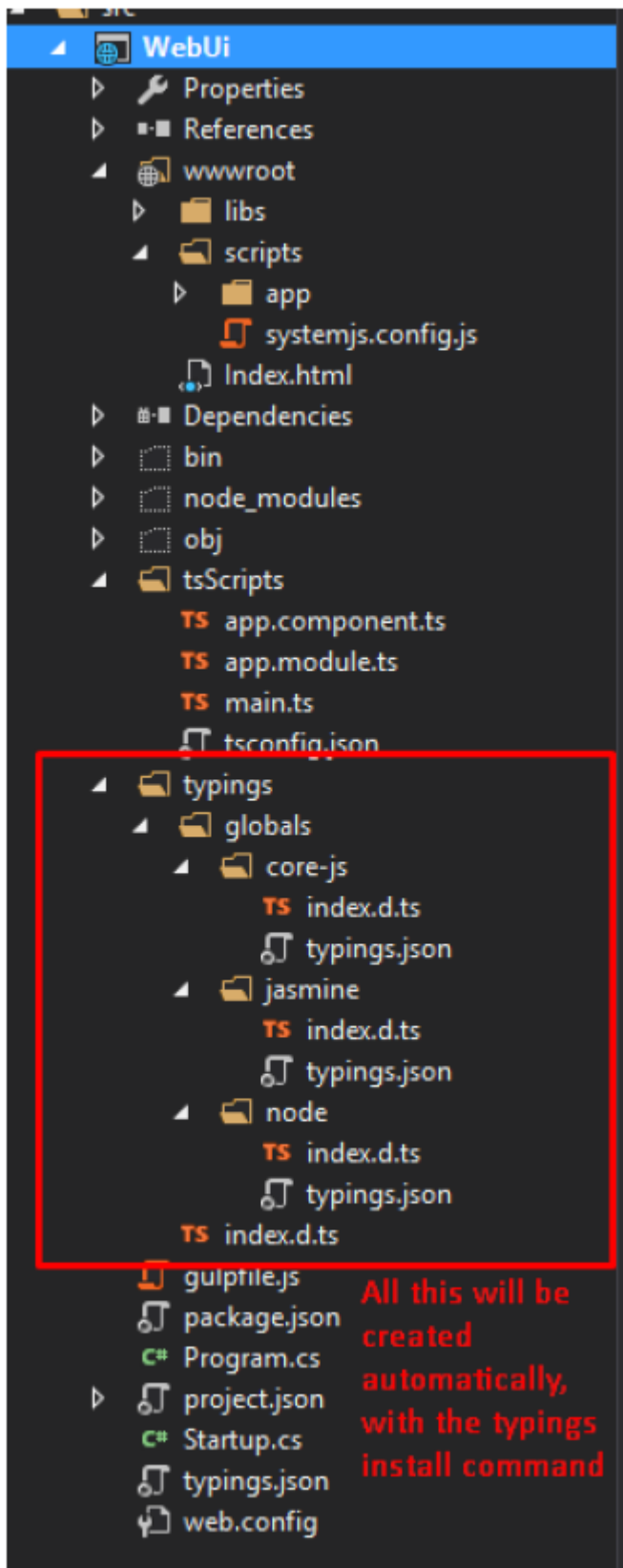


```
1  {
2    "globalDependencies": {
3      "core-js": "registry:dt/core-js#0.0.0+20160725163759",
4      "jasmine": "registry:dt/jasmine#2.2.0+20160621224255",
5      "node": "registry:dt/node#6.0.0+20160909174046"
6    }
7  }
```

- In the Web Project root open a command line and execute "typings install", this will create a typings folder (This requires "Open Command Line" explained as an optional step in the Note inside Step 4, numeral iii).

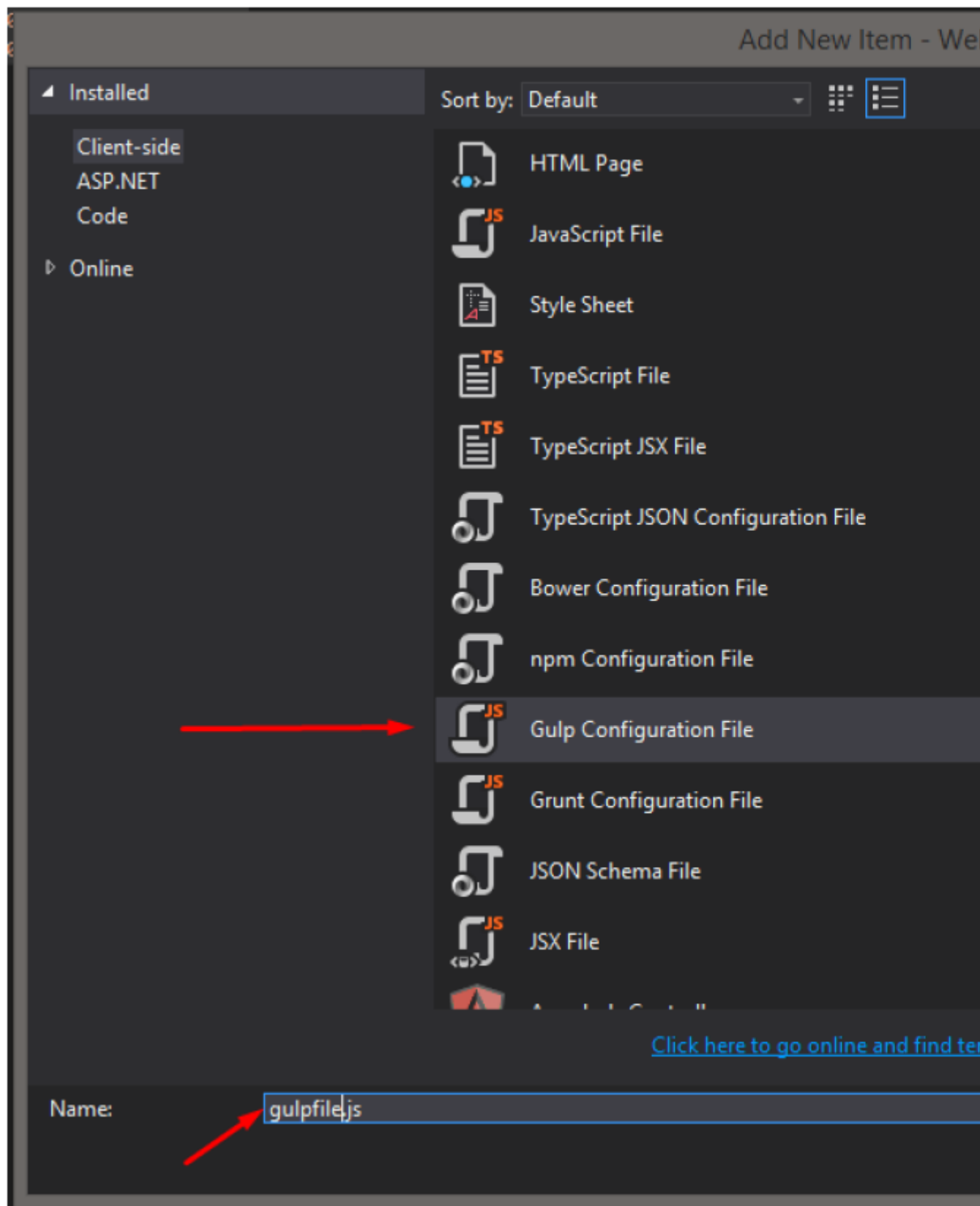


```
j...": "..."
C:\Windows\SYSTEM32\cmd.exe
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.
C:\Users\... \Desktop\Mkr\src\WebUi>typings install
```



6. Add gulp to move files:

- Add "Gulp Configuration File" (gulpfile.js) at the root of the web project:



- Add 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-lib', function () {
  return gulp
    .src([libsDestPath])
    .pipe(clean());
});

gulp.task('clean-app-scripts', function () {
  return gulp
    .src([scriptsDestPath])
    .pipe(clean());
});

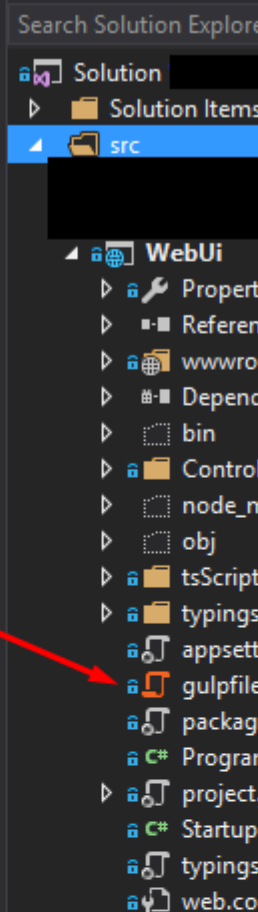
gulp.task("copy-lib", ['clean-lib'], () => {
  gulp
    .src([
      '@angular/**',
      'core-js/client/**',
      'reflect-metadata/**',
      'es6-shim/es6-sh*',
      'rxjs/**',
      'systemjs/dist/system.src.js',
      'zone.js/dist/**',
      'bootstrap/dist/js/bootstrap.*js'
    ], {
      cwd: "node_modules/**"
    })
    .pipe(gulp.dest(libsDestPath));
});

var tsProject = ts.createProject('tsScripts/tsconfig.json', {
  typescript: require('typescript')
});

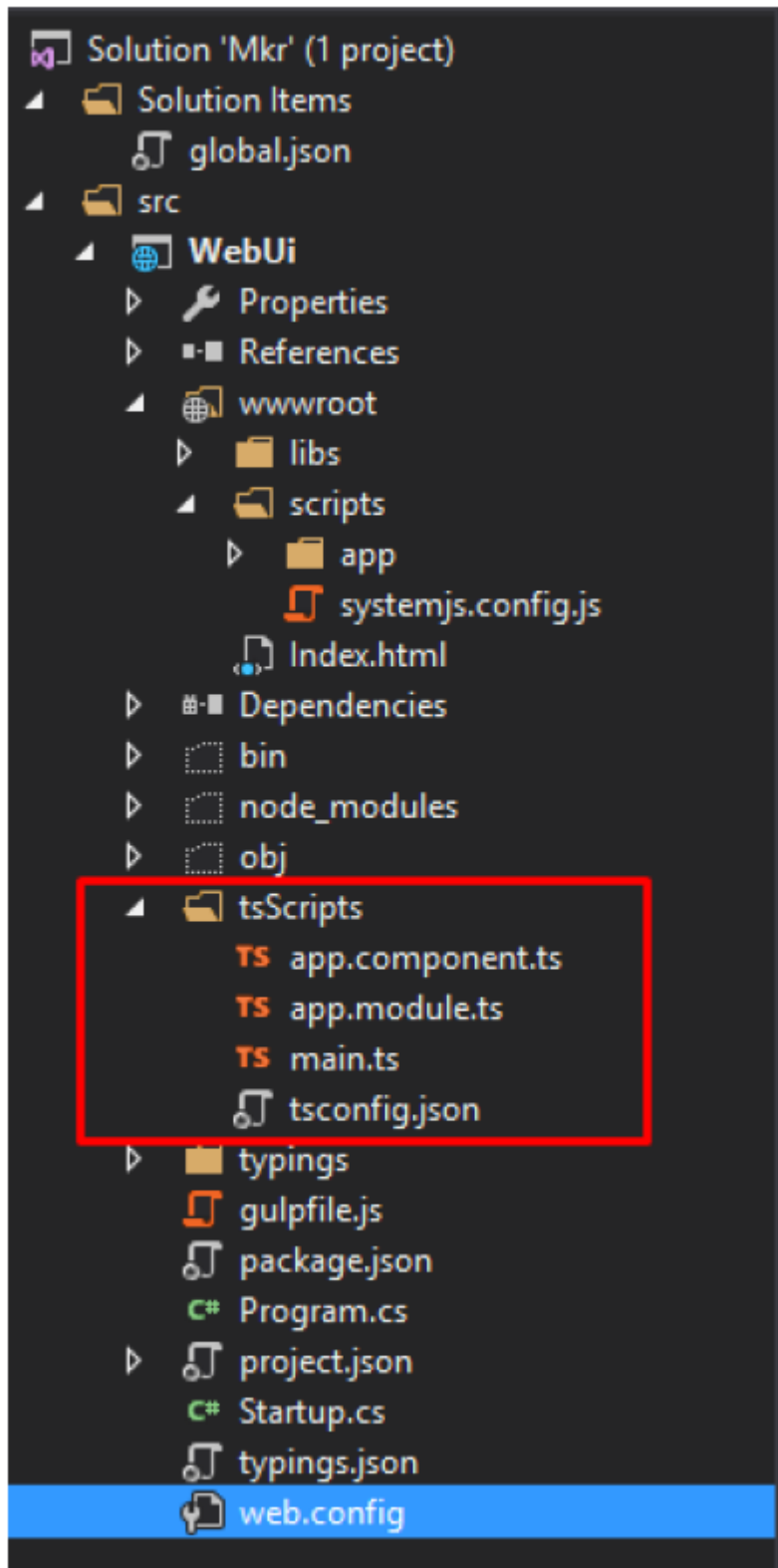
gulp.task('transpile-ts', ['clean-app-scripts'], function (done) {
  var tsResult = gulp
    .src([
      "tsScripts/*.ts"
    ])
    .pipe(tsProject(ts.reporter.fullReporter()));
  return tsResult.js.pipe(gulp.dest(scriptsDestPath));
});

gulp.task('default', ['copy-lib', 'transpile-ts']);

```



7. Add Angular 2 bootstrapping files inside the "tsScripts" folder:



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';

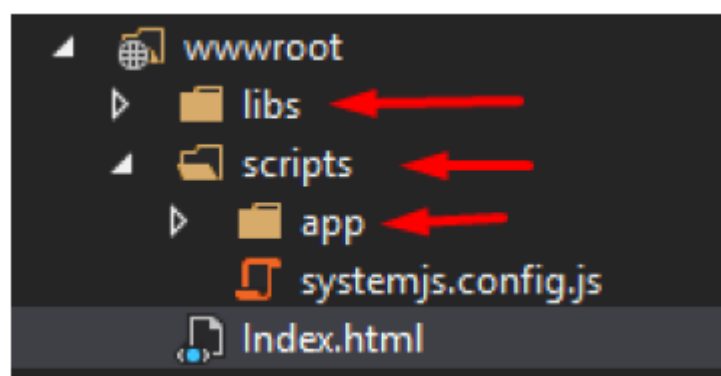
@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

main.ts

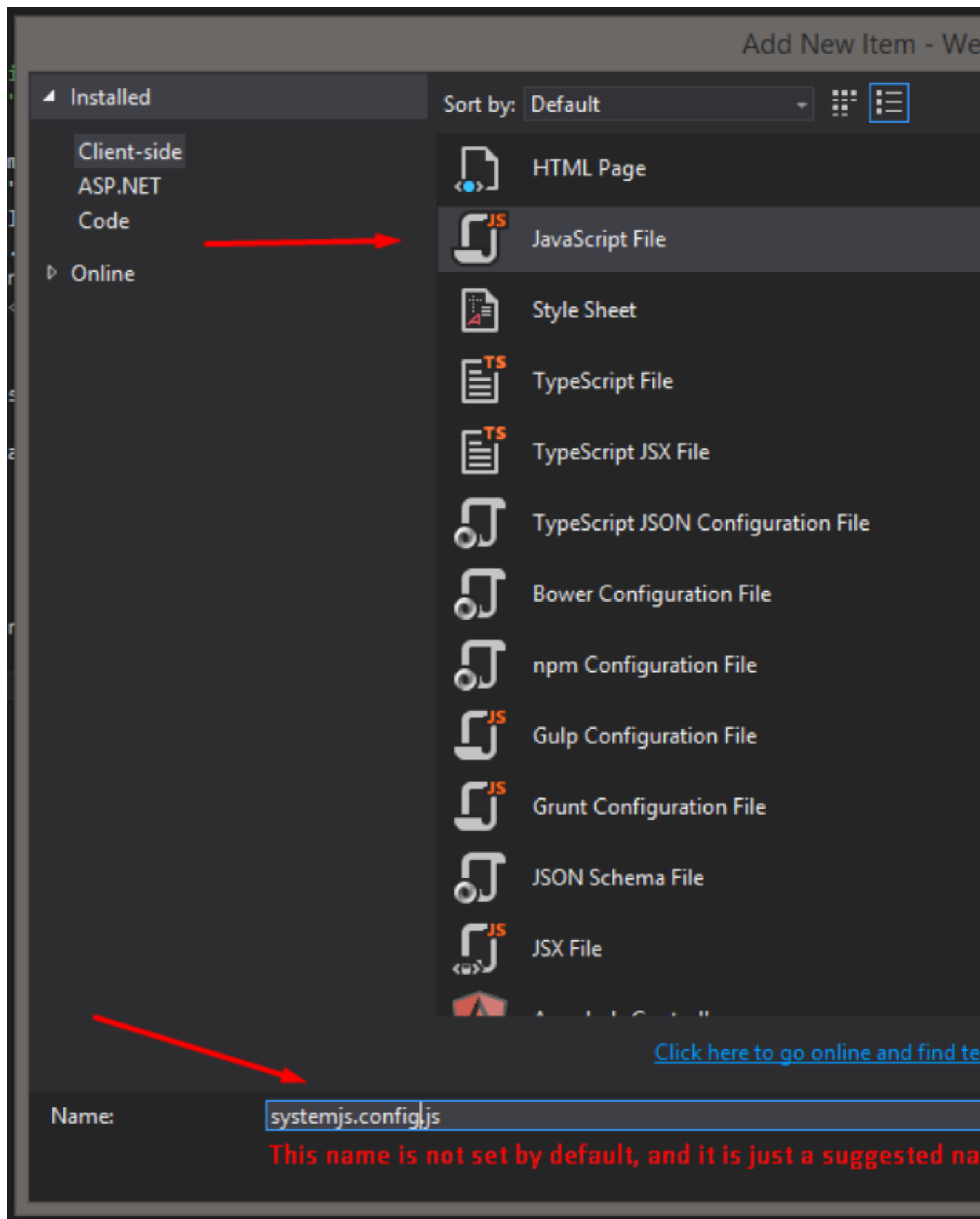
```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

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

8. Inside wwwroot, create the next file structure:



9. Inside the scripts folder (but outside app), add the systemjs.config.js:



And add the next code:

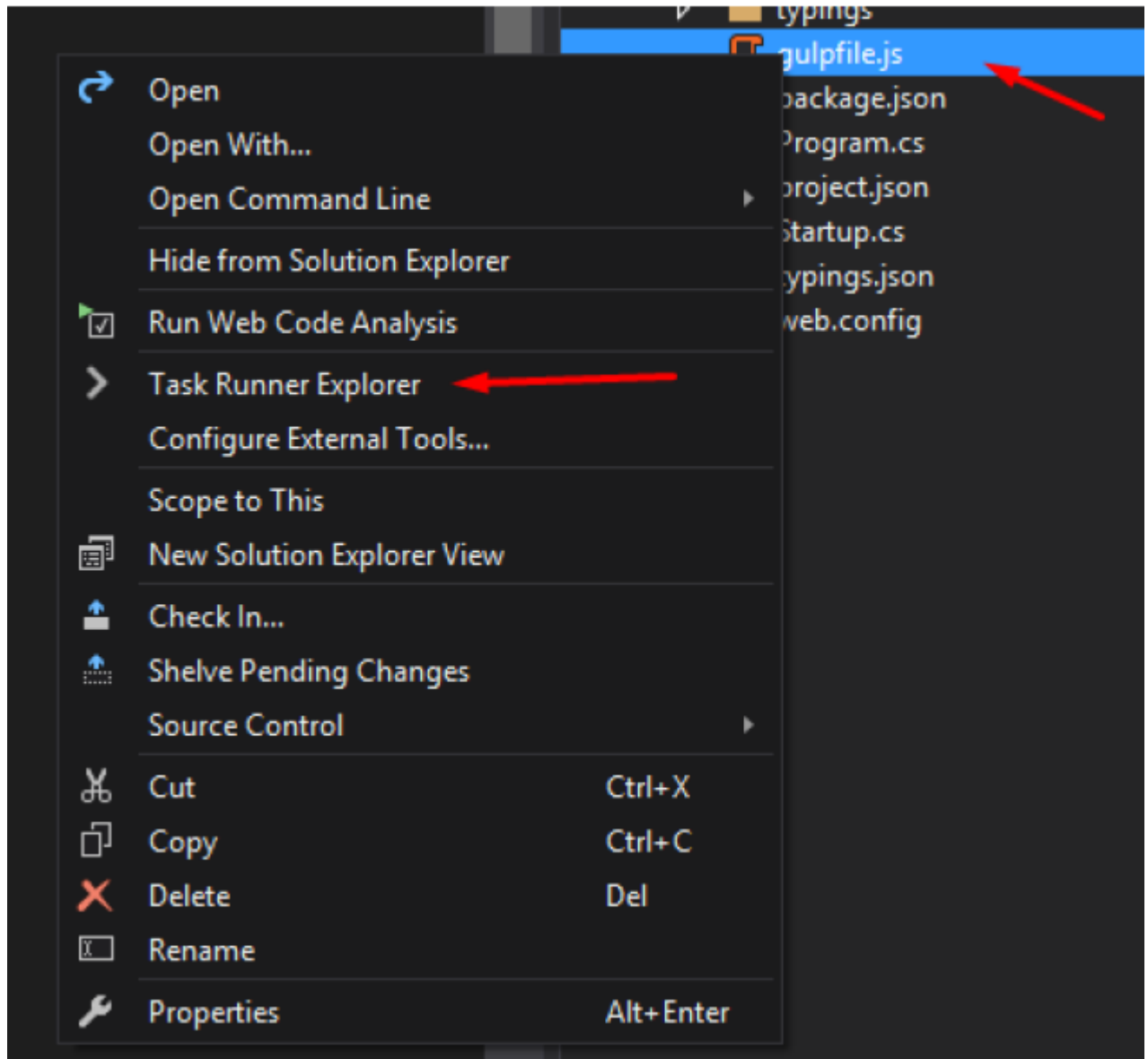
```

1  /**
2   * System configuration for Angular samples
3   * Adjust as necessary for your application needs.
4   */
5  (function (global) {
6    System.config({
7      paths: {
8        // paths serve as alias
9        'npm:': './libs/'
10     },
11     // map tells the System loader where to look for things
12     map: {
13       // our app is within the app folder
14       app: './scripts/app',
15       // angular bundles
16       '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
17       '@angular/common': 'npm:@angular/common/bundles/common.umd',
18       '@angular/compiler': 'npm:@angular/compiler/bundles/compiler',
19       '@angular/platform-browser': 'npm:@angular/platform-browser',
20       '@angular/platform-browser-dynamic': 'npm:@angular/platform-dynamic',
21       '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
22       '@angular/router': 'npm:@angular/router/bundles/router.umd',
23       '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
24       // other libraries
25       'rxjs': 'npm:rxjs'
26     },
27     // packages tells the System loader how to load when no filename
28     packages: {
29       app: {
30         main: './app/main.js',
31         defaultExtension: 'js'
32       },
33       rxjs: {
34         defaultExtension: 'js'
35       }
36     }
37   });
38 })(this);

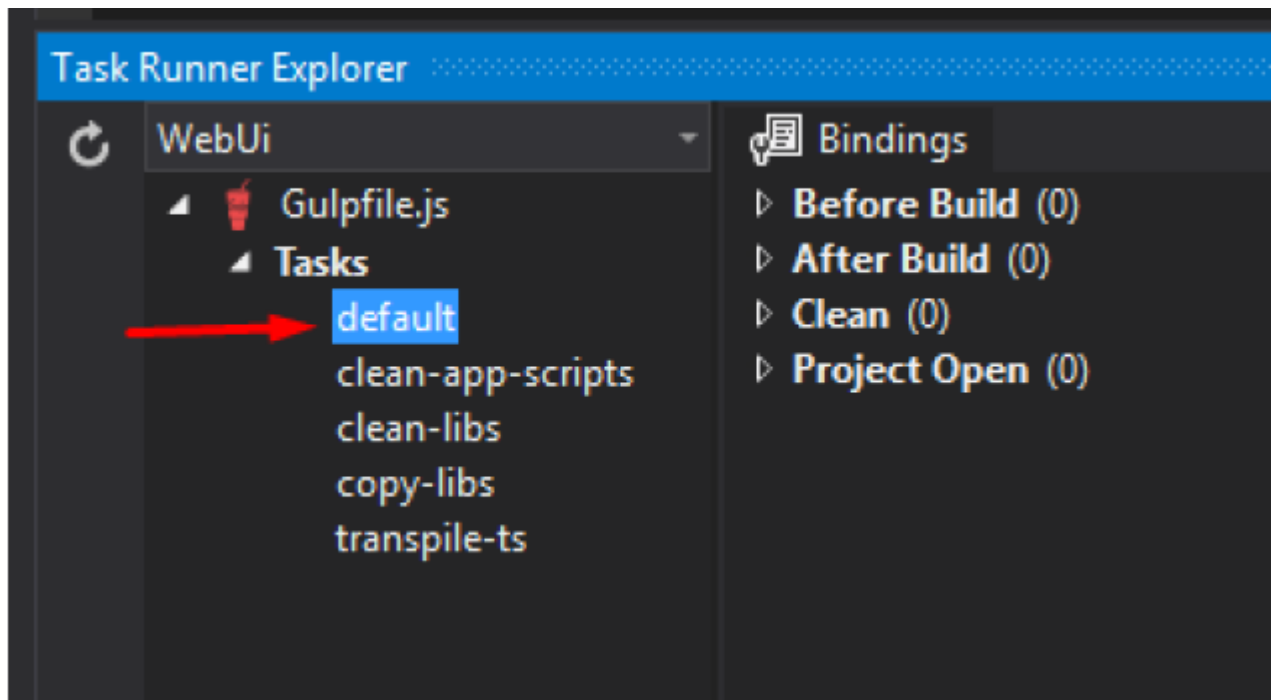
```

10. Execute Gulp Task to generate the scripts in wwwroot.

- Right click gulpfile.js
- Task Runner Explorer



- i. If the tasks are not loaded ("Fail to load. See Output window") Go to output window and take a look at the errors, most of the time are syntax errors in the gulp file.
- Right Click "default" task and "Run" (It will take a while, and the confirmation messages are not very precise, it shows it finished but the process is still running, keep that in mind):



11. Modify Index.html like:

```

1  <!DOCTYPE html>
2  <html>
3  <head>
4      <<meta charset="utf-8" />
5      <<title>[REDACTED]</title>
6
7      <<!--link href="node_modules/bootstrap/dist/css/boot
8      <<link href="app/app.component.css" rel="stylesheet"
9      <<!--1. Load libraries-->
10     <<script src="/libs/core-js/client/shim.min.js"></sc
11     <<script src="/libs/zone.js/dist/zone.js"></script>
12     <<script src="/libs/reflect-metadata/Reflect.js"></s
13     <<script src="/libs/systemjs/dist/system.src.js"></s
14     <<script src="/libs/es6-shim/es6-shim.min.js"></scrip
15     <<script src="/libs/rxjs/bundles/Rx.js"></script>
16
17     <<!--2. Configure SystemJS-->
18     <<script src="/scripts/systemjs.config.js"></script>
19     <<script>
20         <<System.import('/scripts/app/main').catch(function
21         <<console.error(err);
22         <<});
23     <</script>
24 </head>
25 <body>
26     <<my-app>Loading AppComponent content here ...</my-a
27 </body>
28 </html>

```

12. Now run and enjoy.

Notes:

- In case there are compilation errors with typescript, for example "TypeScript Virtual Project", it is an indicator that the TypeScript version for Visual Studio is not updated according to the version we selected in the "package.json", if this happens please install: <https://www.microsoft.com/en-us/download/details.aspx?id=48593>

References:

- Deborah Kurata's "Angular 2: Getting Started" course in Pluralsight:

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

- Angular 2 Official Documentation:

<https://angular.io/>

- Articles by 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/>

Expected errors when generating Angular 2 components in .NET Core project (version 0.8.3)

When generating new Angular 2 components in a .NET Core project, you may run into the following errors (as of version 0.8.3):

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

OR

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

[SOLUTION]

1. Rename app.module.client.ts to app.client.module.ts
2. Open app.client.module.ts: prepend the declaration with 3 dots “...” and wrap the declaration in brackets.

For example: `[...sharedConfig.declarations, <MyComponent>]`

3. Open boot-client.ts: update your import to use the new app.client.module reference.

For example: `import { AppModule } from './app/app.client.module';`

4. Now try to generate the new component: `ng g component my-component`

[EXPLANATION]

Angular CLI looks for a file named app.module.ts in your project, and tries to find a references for the declarations property to import the component. This should be an array (as the sharedConfig.declarations is), but the changes do not get applied

[SOURCES]

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

Read Angular2 and .Net Core online: <https://riptutorial.com/asp-net-core/topic/9352/angular2-and--net-core>

Chapter 3: ASP.NET Core - Log both Request and Response using Middleware

Introduction

For some time I've searched for the best way to log requests and response in an ASP.Net Core. I was developing services and one of the requirements was to record request with its response in one record the the database. So many topics out there but none worked for me. it's either for request only, response only or simply didn't work. When I was able to finally do it, and it had evolved during my project to better error handling and logging exceptions so I thought of sharing.

Remarks

some of the topics that was helpful to me:

- <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/>
- [How to log the HTTP Response Body in ASP.NET Core 1.0](#)

Examples

Logger Middleware

```
using Microsoft.AspNetCore.Http;
using System;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.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.LogToDatabase(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;
        }
    }
}
}
}

```

Read ASP.NET Core - Log both Request and Response using Middleware online:
<https://riptutorial.com/asp-net-core/topic/9510/asp-net-core---log-both-request-and-response-using-middleware>

Chapter 4: Authorization

Examples

Simple Authorization

Authorization in asp.net core is simply `AuthorizeAttribute`

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

    public IActionResult Post()
    {
    }
}
```

This will only allow a logged in user to access these actions.

or use the following to only limit a single action

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

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

If you want to allow all users to access one of the actions you can use `AllowAnonymousAttribute`

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

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

Now `Post` can be accessed by any user. `AllowAnonymous` always comes as a priority to authorize, so

if a controller is set to `AllowAnonymous` then all its actions are public, regardless of if they have an `AuthorizeAttribute` or not.

There is an option to set all controllers to require authorized requests -

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

This is done by adding a default authorization policy to each controller - any `Authorize/AllowAnonymous` Attributes over a controller/action will override these settings.

Read Authorization online: <https://riptutorial.com/asp-net-core/topic/6914/authorization>

Chapter 5: Bundling and Minification

Examples

Grunt and Gulp

In ASP.NET Core apps, you bundle and minify the client-side resources during design-time using third party tools, such as [Gulp](#) and [Grunt](#). By using design-time bundling and minification, the minified files are created prior to the application's deployment. Bundling and minifying before deployment provides the advantage of reduced server load. However, it's important to recognize that design-time bundling and minification increases build complexity and only works with static files.

This is done in ASP.NET Core by configuring Gulp via a `gulpfile.js` file within your project :

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

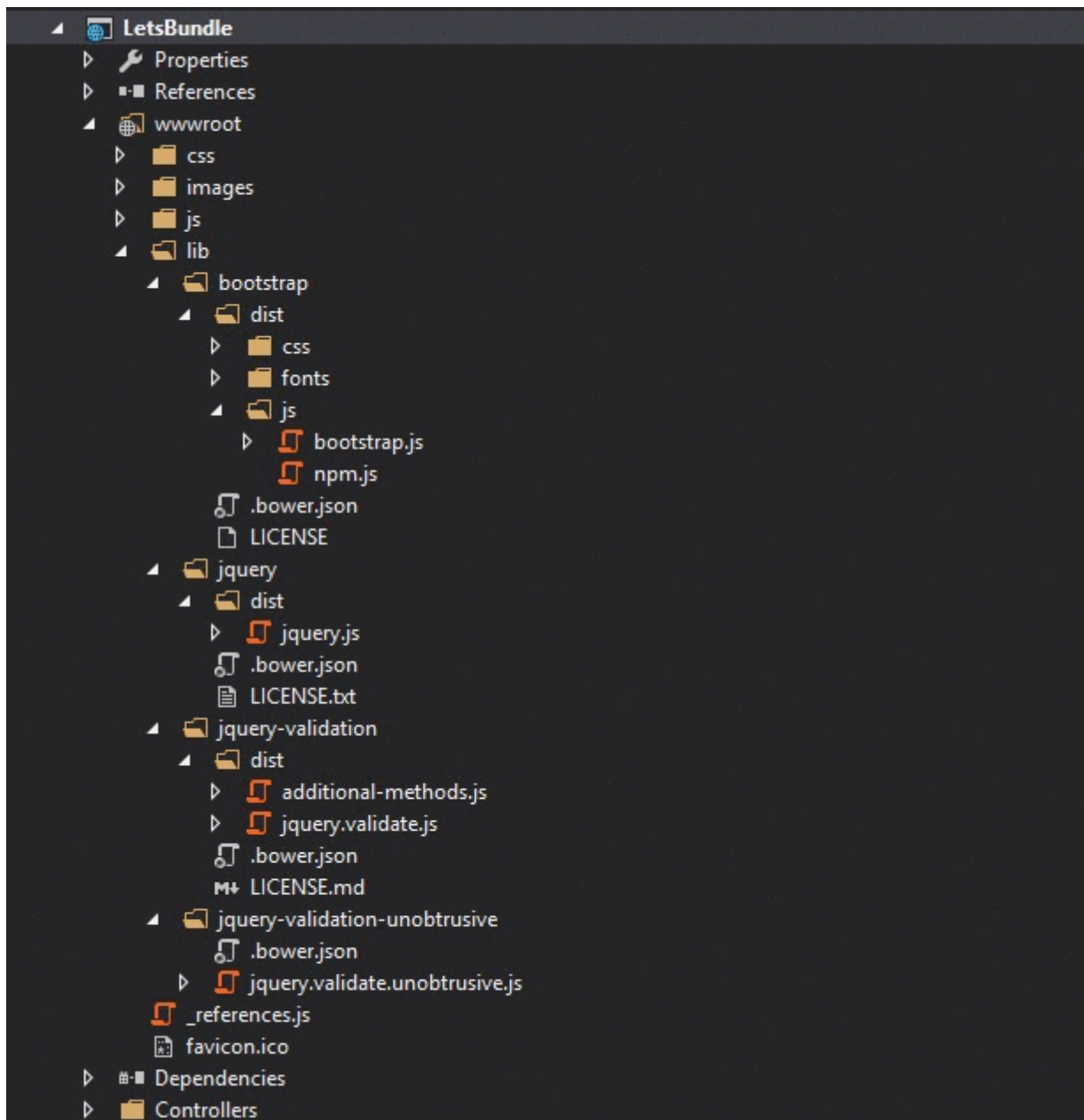
This approach will properly bundle and minify your existing Javascript and CSS files respectively accordingly to the directories and globbing patterns that are used.

Bundler and Minifier Extension

Visual Studio also features an available [Bundler and Minifier Extension](#) that is capable of handling this process for you. The extension allows you to easily select and bundle the files you need without writing a line of code.

Building Your Bundles

After installing the extension, you **select all of the specific files that you want to include within a bundle and use the Bundle and Minify Files option from the extension :**



This will prompt to you name your bundle and choose a location to save it at. You'll then notice a new file within your project called `bundleconfig.json` which looks like the following :

```
[
  {
```



```

"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"
]
}
]

```

NOTE: The order in which the files are selected will determine the order that they appear in within the bundle, so if you have any dependencies, ensure you take that into account.

Minifying Your Bundles

Now the previous step will simply bundle your files, if you want to minify the bundle, then you need to indicate that within the `bundleconfig.json` file. **Simply add a `minify` block like the following to your existing bundle to indicate you want it minified :**

```

[
  {
    "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
    }
  }
]

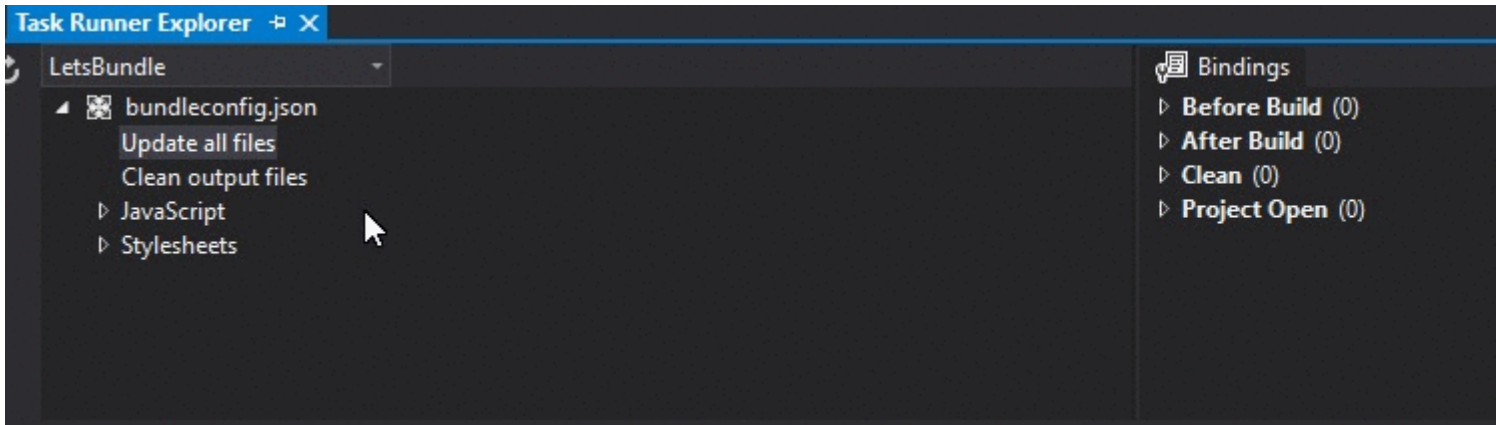
```

Automate Your Bundles

Finally, if you want to automate this process, you can schedule a task to run whenever your application is built to ensure that your bundles reflect any changes within your application.

To do this, you'll need to do the following :

- **Open the Task Runner Explorer** (via Tools > Task Runner Explorer).
- **Right-click on the Update All Files option** below `bundleconfig.json`.
- **Select your preferred binding** from the Bindings context menu.



After doing this, your bundles should be automatically updated at the preferred step that you selected.

The dotnet bundle Command

The ASP.NET Core RTM release introduced `BundlerMinifier.Core`, a new Bundling and Minification tool that can be easily integrated into existing ASP.NET Core applications and doesn't require any external extensions or script files.

Using BundlerMinifier.Core

To use this tool, **simply add a reference to `BundlerMinifier.Core` within the `tools` section of your existing `project.json` file as seen below :**

```
"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"
}
```

Configuring Your Bundles

After adding the tool, you'll need to **add a `bundleconfig.json` file in your project** that will be used to configure the files that you wish to include within your bundles. A minimal configuration can be seen below :

```
[
  {
    "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
  }
}
]

```

Creating / Updating Bundles

After your bundles have been configured, you can bundle and minify your existing files via the following command :

```
dotnet bundle
```

Automated Bundling

The Bundling and Minification process can be automated as part of the build process by adding the `dotnet bundle` command in the precompile section of your existing `project.json` file :

```

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

```

Available Commands

You can see a list of all of the available commands and their descriptions below :

- **dotnet bundle** - Executes the bundle command using the `bundleconfig.json` file to bundle and minify your specified files.
- **dotnet bundle clean** - Clears all of the existing output files from disk.
- **dotnet bundle watch** - Creates a watchers that will automatically run `dotnet bundle` whenever an existing input file from the `bundleconfig.json` configuration to bundle your files.
- **dotnet bundle help** - Displays all available help options and instructions for using the command-line interface.

Read Bundling and Minification online: <https://riptutorial.com/asp-net-core/topic/4051/bundling-and-minification>

Chapter 6: Caching

Introduction

Caching helps in improving performance of an application by maintaining easily accessible copy of the data. *Aspnet Core* comes with two easy to use and testing friendly caching abstractions.

Memory Cache will store data in to local server's memory caching.

Distributed Cache will hold the data cache in a centralized location which is accessible by servers in cluster. It comes with three implementations out of the box : In Memory (for unit testing and local dev), Redis and Sql Server.

Examples

Using InMemory cache in ASP.NET Core application

To use an in memory cache in your ASP.NET application, add the following dependencies to your `project.json` file:

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

add the cache service (from `Microsoft.Extensions.Caching.Memory`) to `ConfigureServices` method in `Startup` class

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

To add items to the cache in our application, we will use `IMemoryCache` which can be injected to any class (for example `Controller`) as shown below.

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

Get will return the value if it exists, but otherwise returns `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))
```

Use the **Set** method to write to the cache. `Set` accepts the key to use to look up the value, the value to be cached, and a set of `MemoryCacheEntryOptions`. The `MemoryCacheEntryOptions` allow you to specify absolute or sliding time-based cache expiration, caching priority, callbacks, and

dependencies. One of the sample below-

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

Distributed Caching

To leverage distributed cache, you'll have to reference one of the available implementations :

- [Redis](#)
- [Sql server](#)

For instance you'll register Redis implementation as follows :

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

Require `IDistributedCache` dependency where you need it:

```
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;
    }
}
```

Read Caching online: <https://riptutorial.com/asp-net-core/topic/8090/caching>

Chapter 7: Configuration

Introduction

Asp.net core provides configuration abstractions. They allow you to load configuration settings from various sources and build a final configuration model which can then be consumed by your application.

Syntax

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

Examples

Accessing Configuration using Dependency Injection

The recommended approach would be to avoid doing so and rather use `IOptions<TOptions>` and `IServiceCollection.Configure<TOptions>`.

That said, this is still pretty straightforward to make `IConfigurationRoot` available application wide.

In the `Startup.cs` constructor you should have the following code to build the configuration,

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

Here `Configuration` is an instance of `IConfigurationRoot`, And add this instance as a Singleton to the service collection in `ConfigureServices` method , `Startup.cs` ,

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

For example, you can now access the configuration in a Controller/Service

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

Getting Started

In this example we will describe what happens when you scaffold a new project.

First thing, the following dependencies will be added to you project (currently `project.json` file) :

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

It will also create a constructor in your `Startup.cs` file which will be in charge of building the configuration using `ConfigurationBuilder` fluent api:

1. It first creates a new `ConfigurationBuilder`.
2. It then sets a base path which will be used to compute absolute path of further files
3. It adds an optional `appsettings.json` to the configuration builder and monitor it's changes
4. It adds an optional environment related `appsettings.environmentName.json` configuration file
5. It then adds environment variables.

```
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();  
}
```

If a same setting is set in several sources, the latest source added will win and its value will be selected.

Configuration can then be consumed using the indexer property. The colon `:` character serve a path delimiter.

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

This will look for a configuration value `ConnectionString` in an `AzureLogger` section.

Work with Environment Variables

You can source configuration from environment variables by calling `.AddEnvironmentVariables()` on you `ConfigurationBuilder`.

It will load environment variables prefixed with `APPSETTING_`. It will then use colon `:` as the key path separator.

This means that : following environment settings :

```
APPSETTING_Security:Authentication:UserName = a_user_name  
APPSETTING_Security:Authentication:Password = a_user_password
```

Will be the equivalent this json :

```
{  
    "Security" : {
```

```

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

```

**** Note that Azure Service will transmit settings as environment variables. Prefix will be set for you transparently. So to do the same in Azure just set two Application Settings in AppSettings blade :**

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

Option model and configuration

When dealing with large configuration sets of value, it might become quite unhandy to load them one by one.

Option model which comes with asp.net offers a convenient way to map a `section` to a dotnet `poco`: For instance, one might hydrate `StorageOptions` directly from a configuration section by adding `Microsoft.Extensions.Options.ConfigurationExtensions` package and calling the `Configure<TOptions>(IConfiguration config)` extension method.

```

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

```

In Memory configuration source

You can also source configuration from an in memory object such as a `Dictionary<string, string>`

```

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

```

This can reveal helpful in integration/unit testing scenarios.

Read Configuration online: <https://riptutorial.com/asp-net-core/topic/8660/configuration>

Chapter 8: Configuring multiple Environments

Examples

Having appsettings per Environment

For each environment you need to create a separate `appsettings.{EnvironmentName}.json` files:

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

Then open `project.json` file and include them into "include" in "publishOptions" section. This lists all the files and folders that will be included when you publish:

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

The final step. In your `Startup` class add:

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

in constructor where you set up configuration sources:

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

Get/Check Environment name from code

All you need is a variable of type `IHostingEnvironment`:

- get environment name:

```
env.EnvironmentName
```

- for predefined `Development`, `Staging`, `Production` environments the best way is to use extension methods from [HostingEnvironmentExtensions](#) class

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

- correctly ignore case (another extension method from [HostingEnvironmentExtensions](#):

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

- case sensitive variant:

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

Configuring multiple environments

This example shows how to configure multiple environments with different Dependency Injection configuration and separate middlewares in one `Startup` class.

Alongside of `public void Configure(IApplicationBuilder app)` and `public void ConfigureServices(IServiceCollection services)` methods one can use `Configure{EnvironmentName}` and `Configure{EnvironmentName}Services` to have environment dependent configuration.

Using this pattern avoids putting too much `if/else` logic within one single method/`Startup` class and keep it clean and separated.

```
public class Startup  
{  
    public void ConfigureServices(IServiceCollection services) { }  
    public void ConfigureStagingServices(IServiceCollection services) { }  
    public void ConfigureProductionServices(IServiceCollection services) { }  
  
    public void Configure(IApplicationBuilder app) { }  
    public void ConfigureStaging(IApplicationBuilder app) { }  
    public void ConfigureProduction(IApplicationBuilder app) { }  
}
```

When a `Configure{Environmentname}` or `Configure{Environmentname}Services` is not found, it will fall back to `Configure` or `ConfigureServices` respectively.

The same semantics also apply to the `Startup` class. `StartupProduction` will be used when the `ASPNETCORE_ENVIRONMENT` variable is set to `Production` and will fall back to `Startup` when it's `Staging` or `Development`.

A complete example:

```
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();
    }
}

```

Render environment specific content in view

You may need to render some content in view, which is specific to some environment only. To achieve this goal you can use Environment [tag helper](#):

```
<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>
```

The Environment tag helper will only render its contents if the current environment matches one of the environments specified using the `names` attribute.

Set environment variable from command line

To set the environment to `Development`

```
SET ASPNETCORE_ENVIRONMENT=Development
```

Now running an Asp.Net Core application will be in the defined environment.

Note

1. There should be no space before and after the equality sign `=`.
2. The command prompt should not be closed before running the application because the settings are not persisted.

Set environment variable from PowerShell

When using PowerShell, you can use `setx.exe` to set environment variables permanently.

1. Start PowerShell
2. Type one of the following:

```
setx ASPNETCORE_ENVIRONMENT "development"
```

```
setx ASPNETCORE_ENVIRONMENT "staging"
```

3. Restart PowerShell

Using ASPNETCORE_ENVIRONMENT from web.config

If you do not want to use `ASPNETCORE_ENVIRONMENT` from environment variables and use it from `web.config` of your application then modify `web.config` like this-

```
<aspNetCore processPath=".\\WebApplication.exe" arguments="" stdoutLogEnabled="false"
stdoutLogFile=".\\logs\\stdout" forwardWindowsAuthToken="false">
    <environmentVariables>
```

```
<environmentVariable name="ASPNETCORE_ENVIRONMENT" value="Development" />
</environmentVariables>
</aspNetCore>
```

Read Configuring multiple Environments online: <https://riptutorial.com/asp-net-core/topic/2292/configuring-multiple-environments>

Chapter 9: Cross-Origin Requests (CORS)

Remarks

Browser security prevents a web page from making AJAX requests to another domain. This restriction is called the same-origin policy, and prevents a malicious site from reading sensitive data from another site. However, sometimes you might want to let other sites make cross-origin requests to your web app.

Cross Origin Resource Sharing (CORS) is a W3C standard that allows a server to relax the same-origin policy. Using CORS, a server can explicitly allow some cross-origin requests while rejecting others. CORS is safer and more flexible than earlier techniques such as JSONP.

Examples

Enable CORS for all requests

Use the `UseCors()` extension method on the `IApplicationBuilder` in the `Configure` method to apply the CORS policy to all requests.

```
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();
}
```

Enable CORS policy for specific controllers

To enable a certain CORS policy for specific controllers you have to build the policy in the `AddCors` extension within the `ConfigureServices` method:

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

```

        .AllowAnyMethod()
        .AllowAnyHeader();
    });

```

This allows you to apply the policy to a controller:

```

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

```

More sophisticated CORS policies

The policy builder allows you to build sophisticated policies.

```

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

```

This policy only allows the origins `http://localhost:5000` and `http://myproductionapp.com` with only the `GET`, `POST` and `HEAD` methods and only accepts the `accept`, `content-type` and `origin` HTTP headers. The `SetPreflightMaxAge` method causes the browsers to cache the result of the preflight request (`OPTIONS`) to be cached for the specified amount of time.

Enable CORS policy for all controllers

To enable a CORS policy across all of your MVC controllers you have to build the policy in the `AddCors` extension within the `ConfigureServices` method and then set the policy on the `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");  
}
```

This CORS policy can be overwritten on a controller or action basis, but this can set the default for the entire application.

Read Cross-Origin Requests (CORS) online: <https://riptutorial.com/asp-net-core/topic/2556/cross-origin-requests--cors->

Chapter 10: Dependency Injection

Introduction

Aspnet core is built with Dependency Injection as one of its key core concepts. It introduces one conforming container abstraction so you can replace the builtin one with a third-party container of your choice.

Syntax

- `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>()`

Remarks

To use generic variants of `IServiceProvider` methods you have to include the following namespace:

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

Examples

Register and manually resolve

The preferred way of describing dependencies is by using constructor injection which follows [Explicit Dependencies Principle](#):

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();
        }
    }
}
```

Register dependencies

Builtin container comes with a set of builtin features :

Lifetime control

```
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**: Created everytime it is resolved
- **AddScoped**: Created once per request
- **AddSingleton**: Lazily created once per application
- **AddSingleton (instance)**: Provides a previously created instance per application

Enumerable dependencies

It is also possible to register enumerable dependencies :

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

You can then consume them as follows :

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

Generic dependencies

You can also register generic dependencies :

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

And then consume it as follows :

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

Retrieve dependencies on a Controller

Once registered a dependency can be retrieved by adding parameters on the Controller constructor.

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

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

Injecting a dependency into a Controller Action

A less known builtin feature is Controller Action injection using the `FromServicesAttribute`.

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

An important note is that the `[FromServices]` **can not** be used as general "Property Injection" or "Method injection" mechanism! It can only be used on method parameters of an controller action or controller constructor (in the constructor it's obsolete though, as ASP.NET Core DI system already uses constructor injection and there are no extra markers required).

It can not be used anywhere outside of a controllers, controller action. Also it is very specific to ASP.NET Core MVC and resides in the `Microsoft.AspNetCore.Mvc.Core` assembly.

Original quote from the ASP.NET Core MVC GitHub issue ([Limit \[FromServices\] to apply only to parameters](#)) regarding this attribute:

@rynnowak:

@Eilon:

The problem with properties is that it appears to many people that it can be applied to any property of any object.

Agreed, we've had a number of issues posted by users with confusion around how this feature should be used. There's really been a fairly large amount of feedback both of the kinds " [FromServices] is weird and I don't like it" and " [FromServices] has confounded me". It feels like a trap, and something that the team would still be answering questions about years from now.

We feel like most valuable scenario for [FromServices] is on method parameter to an action for a service that you only need in that one place.

/cc @danroth27 - docs changes

To anyone in love with the current [FromServices] , I'd strongly recommend looking into a DI system that can do property injection (Autofac, for example).

Notes:

- **Any** services registered with the .NET Core Dependency Injection system can be injected inside an controller's action using the `[FromServices]` attribute.
- Most relevant case is when you need a service only in a single action method and don't want to clutter your controller's constructor with another dependency, which will only be used once.
- Can't be used outside of ASP.NET Core MVC (i.e. pure .NET Framework or .NET Core console applications), because it resides in `Microsoft.AspNetCore.Mvc.Core` assembly.
- For property or method injection you must use one of third-party IoC containers available (Autofac, Unity, etc.).

The Options pattern / Injecting options into services

With ASP.NET Core the Microsoft team also introduced the Options pattern, which allows to have strong typed options and once configured the ability to inject the options into your services.

First we start with a strong typed class, which will hold our configuration.

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

And an entry in the `appsettings.json`.

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

```
}  
}
```

Next we initialize it in the Startup class. There are two ways to do this

1. Load it directly from the `appsettings.json` "mysettings" section

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

2. Do it manually

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

Each hierarchy level of the `appsettings.json` is separated by a `..`. Since `value2` is a property of the `mysettings` object, we access it via `mysettings:value2`.

Finally we can inject the options into our services, using the `IOptions<T>` interface

```
public class MyService : IMyService  
{  
    private readonly MySettings settings;  
  
    public MyService(IOptions<MySettings> mysettings)  
    {  
        this.settings = mySettings.Value;  
    }  
}
```

Remarks

If the `IOptions<T>` isn't configured during the startup, injecting `IOptions<T>` will inject the default instance of `T` class.

Using scoped services during application startup / Database Seeding

Resolving scoped services during application startup can be difficult, because there is no request and hence no scoped service.

Resolving a scoped service during application startup via

`app.ApplicationServices.GetService<AppDbContext>()` can cause issues, because it will be created in the scope of the global container, effectively making it a singleton with the lifetime of the application, which may lead to exceptions like `Cannot access a disposed object in ASP.NET Core` when injecting `DbContext`.

The following pattern solves the issue by first creating a new scope and then resolving the scoped

services from it, then once the work is done, disposing the scoped container.

```
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);
            }
        }
}
```

This is a semi-official way of the Entity Framework core team to seed data during application startup and is reflected in the [MusicStore sample](#) application.

Resolve Controllers, ViewComponents and TagHelpers via Dependency Injection

By default Controllers, ViewComponents and TagHelpers aren't registered and resolved via the dependency injection container. This results in the inability to do i.e. property injection when using a 3rd party Inversion of Control (IoC) container like AutoFac.

In order to make ASP.NET Core MVC resolve these Types via IoC too, one needs to add the following registrations in the `Startup.cs` (taken from the official [ControllersFromService sample](#) on 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>();
}
```

Plain Dependency Injection example (Without Startup.cs)

This shows you how to use [Microsoft.Extensions.DependencyInjection](#) nuget package without the use of the `WebHostBuilder` from kestrel (e.g. when you want to build something else then a `webApp`):

```
internal class Program
{
    public static void Main(string[] args)
    {
        var services = new ServiceCollection(); //Creates the service registry
        services.AddTransient<IMyInterface, MyClass>(); //Add registration of IMyInterface
        //should create a 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 manuallyInstantiate = new ServiceDependingOnIMyInterface(implementation);

        services.AddTransient<ServiceDependingOnIMyInterface>();
        var spWithService = services.BuildServiceProvider(); //Generally its bad practise to
        //rebuild the container because its heavy 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;
    }
}
```

Inner workings of Microsoft.Extensions.DependencyInjection

IServiceCollection

To start building an IOC container with Microsoft's DI nuget package you start with creating an `IServiceCollection`. You can use the already provided Collection: `ServiceCollection`:

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


This `IServiceCollection` is nothing more than an implementation of: `IList<ServiceDescriptor>`, `ICollection<ServiceDescriptor>`, `IEnumerable<ServiceDescriptor>`, `IEnumerable`

All the following methods are only extension methods to add `ServiceDescriptor` instances to the list:

```
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

The serviceprovider is the one 'Compiling' all the registrations so that they can be used quickly, this can be done with `services.BuildServiceProvider()` which is basically an extension method for:

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

Behind the scenes every `ServiceDescriptor` in the `IServiceCollection` gets compiled to a factory method `Func<ServiceProvider, object>` where `object` is the return type and is: the created instance of the `Implementation` type, the `Singleton` or your own defined factory method.

These registrations get added to the `ServiceTable` which is basically a `ConcurrentDictionary` with the key being the `ServiceType` and the value the `Factory` method defined above.

Result

Now we have a `ConcurrentDictionary<Type, Func<ServiceProvider, object>>` which we can use concurrently to ask to create `Services` for us. To show a basic example of how this could have looked.

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

This is not how it works!

This `ConcurrentDictionary` is a property of the `ServiceTable` which is a property of the `ServiceProvider`

Read **Dependency Injection** online: <https://riptutorial.com/asp-net-core/topic/1949/dependency->

injection

Chapter 11: Error Handling

Examples

Redirect to custom error page

ASP.NET Core provides the [status code pages middleware](#), that supports several different extension methods, but we are interesting in `UseStatusCodePages` and

`UseStatusCodePagesWithRedirects`:

- [UseStatusCodePages](#) adds a `StatusCodePages` middleware with the given options that checks for responses with status codes between 400 and 599 that do not have a body. Example of use for redirect:

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

    // your redirect logic
});
```

- [UseStatusCodePagesWithRedirects](#) adds a `StatusCodePages` middleware to the pipeline. Specifies that responses should be handled by redirecting with the given location URL template. This may include a '{0}' placeholder for the status code. URLs starting with '~' will have `PathBase` prepended, where any other URL will be used as is. For example the following will redirect to `~/errors/<error_code>` (for example `~/errors/403` for 403 error):

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

Global Exception Handling in ASP.NET Core

`ExceptionHandler` can be used to handle exceptions globally. You can get all the details of exception object like Stack Trace, Inner exception and others. And then you can show them on screen. You can easily implement like this.

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

```
}  
);
```

You need to put this inside `configure()` of `startup.cs` file.

Read Error Handling online: <https://riptutorial.com/asp-net-core/topic/6581/error-handling>

Chapter 12: Injecting services into views

Syntax

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

Examples

The @inject Directive

ASP.NET Core introduces the concept of dependency injection into Views via the `@inject` directive via the following syntax :

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

Example Usage

Adding this directive into your View will basically generate a property of the given type using the given name within your View using proper dependency injection as demonstrated in the example below :

```
@inject YourWidgetServiceClass WidgetService

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

Required Configuration

Services that use dependency injection are still required to be registered within the `ConfigureServices()` method of the `Startup.cs` file and scoped accordingly :

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

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

Read Injecting services into views online: <https://riptutorial.com/asp-net-core/topic/4284/injecting-services-into-views>

Chapter 13: Localization

Examples

Localization using JSON language resources

In ASP.NET Core there are several different ways we can localize/globalize our app. It's important to pick a way that suits your needs. In this example you'll see how we can make a multilingual ASP.NET Core app that reads language specific strings from `.json` files and store them in memory to provide localization in all sections of the app as well as maintaining a high performance.

The way we do it is by using `Microsoft.EntityFrameworkCore.InMemory` package.

Notes:

1. The namespace for this project is `DigitalShop` that you may change to your projects own namespace
2. Consider creating a new project so that you don't run into weird errors
3. By no means this example show the best practices, So if you think it can be improved please kindly edit it

To begin let's add the following packages to the **existing** `dependencies` section in the `project.json` file:

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

Now let's replace the `Startup.cs` file with: (`using` statements are removed as they can be easily added later)

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 cuturedirection = config["AppOptions:CultureDirection"];

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

    Startup.UiCulture = uiCulture;

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

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

```

In the above code, we first add three `public static` field variables that we will later initialize using the values read from the settings file.

In the constructor for `Startup` class we add a json settings file to the `builder` variable. The first file is required for the app to work, so go ahead and create `appsettings.json` in your project root if it doesn't already exist. Using Visual Studio 2015, this file is created automatically, so just change its content to: (You may omit the `Logging` section if you don't use it)

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
  }
}

```

Going forward, create three folders in your project root:

`Models`, `Services` and `Languages`. In the `Models` folder create another folder named `Localization`.

In the `Services` folder we create a new `.cs` file named `EFLocalization`. The content would be: (Again using statements are not included)

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;
    }
}

```

In the above file we implement the `IStringLocalizerFactory` interface from Entity Framework Core in order to make a custom localizer service. The important part is the constructor of `EFStringLocalizerFactory` where we make a list of all available languages and add it to the database context. Each one of these language files act as a separate database.

Now add each of the following files to the `Models/Localization` folder:

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();
        }
    }
}

```

The above files are just models that will be populated with language resources, cultures and there's also a typical `DbContext` used by EF Core.

The last thing we need to make all of this work is to create the language resource files. The JSON files used to store a key-value pair for different languages available in your app.

In this example our app only has two languages available. English and Persian. For each of the languages we need two files. A JSON file containing key-value pairs and a `.cs` file that contains a class with the same name as JSON file. That class has one method, `GetList` that deserializes the JSON file and returns it. This method is called in the constructor of `EFStringLocalizerFactory` that we created earlier.

So, create these four files in your `Languages` folder:

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": "دی‌دم آش‌وخ"  
  },  
  {  
    "Key": "Hello",  
    "Value": "م‌ال‌س"  
  },  
]
```

We are all done. Now in order to access the language strings (key-value pairs) anywhere in your code (.cs or .cshtml) you can do the following:

in a .cs file (be Controller or not, doesn't matter):

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

in a Razor view file (.cshtml):

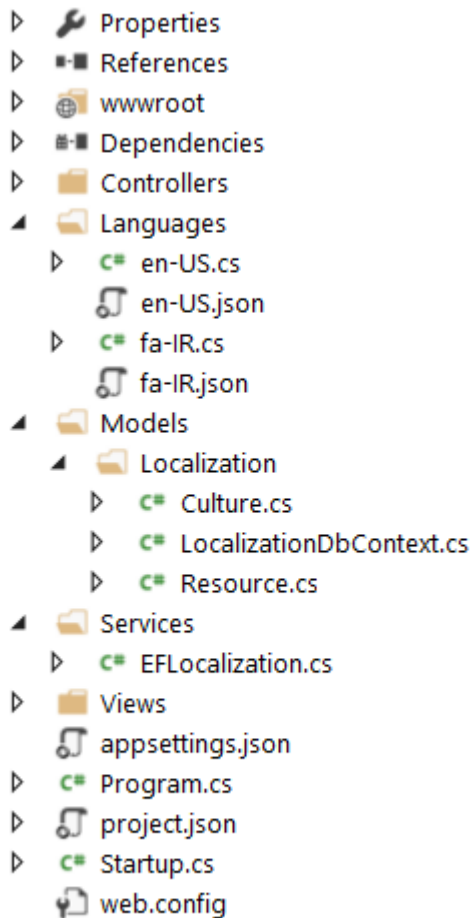
```
<h1>@Startup._e["Welcome"]</h1>
```

Few things to keep in mind:

- If you try to access a `Key` that doesn't exist in the JSON file or loaded, you will just get the key literal (in the above example, trying to access `Startup._e["How are you"]` will return `How are you` no matter the language settings because it doesn't exist)
- If you change a string value in a language .json file, you will need to **RESTART** the app. Otherwise it will just show the default value (key name). **This is specially important when you're running your app without debugging.**
- The `appsettings.json` can be used to store all kinds of settings that your app may need
- Restarting the app is **not necessary** if you just want to change the **language/culture settings from appsettings.json file**. This means that you can have an option in your apps

interface to let users change the language/culture at runtime.

Here's the final project structure:



Set Request culture via url path

By default the built-in Request Localization middleware only supports setting culture via query, cookie or `Accept-Language` header. This example shows how create a middleware which allows to set the culture as part of the path like in `/api/en-US/products`.

This example middleware assumes the locale to be in the second segment of the path.

```
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));
}
}

```

Middleware Registration

```

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);

```

Custom Route Constraints

Adding and creating custom route constraints are shown in the [Route constraints](#) example. Using constraints simplifies the usage of custom route constraints.

Registering the route

Example of registering the routes without using a custom constraints

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

Read Localization online: <https://riptutorial.com/asp-net-core/topic/2869/localization>

Chapter 14: Logging

Examples

Using NLog Logger

[NLog.Extensions.Logging](#) is the official [NLog](#) provider for Microsoft's in .NET Core and ASP.NET Core. [Here](#) and [here](#) are instruction and example respectively.

Add Logger to Controller

Instead of requesting an ILoggerFactory and creating an instance of ILogger explicitly, you can request an ILogger (where T is the class requesting the logger).

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

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

Using Serilog in ASP.NET core 1.0 application

1) In project.json, add below dependencies-

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

2) In Startup.cs, add below lines in constructor-

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

3) In Configure method of Startup class-

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

4) In Controller, create instance of ILogger like this-

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

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

5)Sample logging below-

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

Read Logging online: <https://riptutorial.com/asp-net-core/topic/1946/logging>

Chapter 15: Middleware

Remarks

Middleware is a software component that will determine how to process the request and decide whether to pass it to the next component in the application pipeline. Each middleware has a vary specific role and actions to preform on the request.

Examples

Using the `ExceptionHandler` middleware to send custom JSON error to Client

Define your class that will represent your custom error.

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

    // other fields

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

Then put next `ExceptionHandler` middleware to Configure method. Pay attention that middleware order is important.

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

        var error = context.Features.Get<ExceptionHandlerFeature>();
        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 to set response ContentType

The idea is to use `HttpContext.Response.OnStarting` callback, as this is the last event that is fired before the headers are sent. Add the following to your middleware `Invoke` method.

```
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);
}
```

Pass data through the middleware chain

From [documentation](#):

The ***HttpContext.Items*** collection is the best location to store data that is only needed while processing a given request. Its contents are discarded after each request. It is best used as a means of communicating between components or middleware that operate at different points in time during a request, and have no direct relationship with one another through which to pass parameters or return values.

`HttpContext.Items` is a simple dictionary collection of type `IDictionary<object, object>`. This collection is

- available from the start of an `HttpRequest`
- and is discarded at the end of each request.

You can access it by simply assigning a value to a keyed entry, or by requesting the value for a given key.

For example, some simple Middleware could add something to the `Items` collection:

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

and later in the pipeline, another piece of middleware could access it:

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

Run, Map, Use

Run

Terminates chain. No other middleware method will run after this. Should be placed at the end of any pipeline.

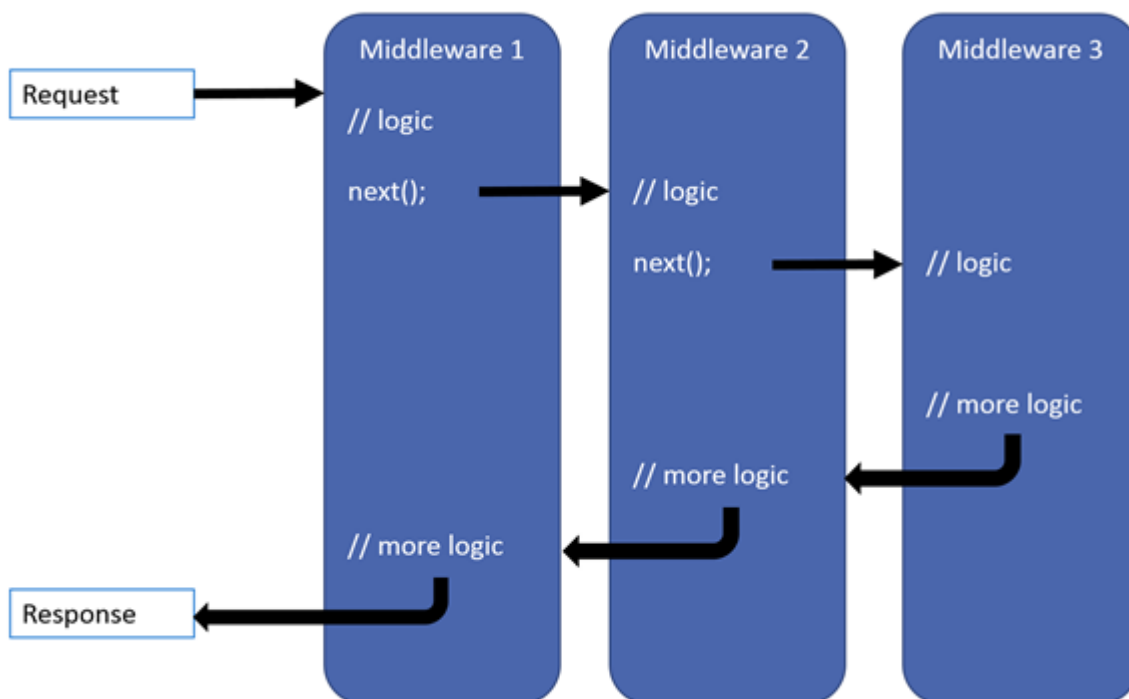
```
app.Run(async context =>
{
    await context.Response.WriteAsync("Hello from " + _environment);
});
```

Use

Performs action before and after next delegate.

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

Illustration of how it works:



MapWhen

Enables branching pipeline. Runs specified middleware if condition is met.

```
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);
}
```

Map

Similar to MapWhen. Runs middleware if path requested by user equals path provided in parameter.

```
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);
}
```

Based on [ASP.net Core Docs](#)

Read Middleware online: <https://riptutorial.com/asp-net-core/topic/1479/middleware>

Chapter 16: Models

Examples

Model Validation with Validation Attributes

Validation attributes can be used to easily configure model validation.

```
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; }
}
```

The built in attributes are:

- [CreditCard]: Validates the property has a credit card format.
- [Compare]: Validates two properties in a model match.
- [EmailAddress]: Validates the property has an email format.
- [Phone]: Validates the property has a telephone format.
- [Range]: Validates the property value falls within the given range.
- [RegularExpression]: Validates that the data matches the specified regular expression.
- [Required]: Makes a property required.
- [StringLength]: Validates that a string property has at most the given maximum length.
- [Url]: Validates the property has a URL format.

Model Validation with Custom Attribute

If the built in attributes are not sufficient to validate your model data, then you can place your validation logic in a class derived from ValidationAttribute. In this example only odd numbers are valid values for a model member.

Custom Validation Attribute

```
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.");  
    }  
}  
}
```

Model Class

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

Read Models online: <https://riptutorial.com/asp-net-core/topic/4625/models>

Chapter 17: project.json

Introduction

Project json is a project configuration file structure, temporarily used by asp.net-core projects, before Microsoft moved back to the csproj files in favor of msbuild.

Examples

Simple Library project example

A library based on NETStandard 1.6 would look like this:

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

Complete json file:

Taken from [microsoft's github page with official documentation](#)

```
{
  "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
}
```

Simple startup project

A simple example of project configuration for a .NetCore 1.1 Console App

```
{
  "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"
    }
  },
}
```

Read project.json online: <https://riptutorial.com/asp-net-core/topic/9364/project-json>

Chapter 18: Publishing and Deployment

Examples

Kestrel. Configuring Listening Address

Using Kestrel you can specify port using next approaches:

1. Defining `ASPNETCORE_URLS` environment variable.

Windows

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

OS X

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

2. Via command line passing `--server.urls` parameter

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

3. Using `UseUrls()` method

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

4. Defining `server.urls` setting in configuration source.

Next sample use `hosting.json` file for example.

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

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

Examples of possible values:

- listen 5000 on any IP4 and IP6 addresses from any interface:

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

or

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

- listen 5000 on every IP4 address:

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

One should be carefully and not use `http://*:5000;http://::5000, http://::5000;http://*:5000, http://*:5000;http://0.0.0.0:5000` or `http://*:5000;http://0.0.0.0:5000` because it will require to register IP6 address :: or IP4 address 0.0.0.0 twice

Add file to `publishOptions` in `project.json`

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

and in entry point for the application call `.UseConfiguration(config)` when creating `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();  
}
```

Read Publishing and Deployment online: <https://riptutorial.com/asp-net-core/topic/2262/publishing-and-deployment>

Chapter 19: Rate limiting

Remarks

[AspNetCoreRateLimit](#) is an open source ASP.NET Core rate limiting solution designed to control the rate of requests that clients can make to a Web API or MVC app based on IP address or client ID.

Examples

Rate limiting based on client IP

With `IpRateLimit` middleware you can set multiple limits for different scenarios like allowing an IP or IP range to make a maximum number of calls in a time interval like per second, 15 minutes, etc. You can define these limits to address all requests made to an API or you can scope the limits to each URL path or HTTP verb and path.

Setup

NuGet install:

```
Install-Package AspNetCoreRateLimit
```

Startup.cs code:

```
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();
}

```

You should register the middleware before any other components except `loggerFactory`.

if you load balance your app you'll need to use `IDistributedCache` with Redis or SQLServer so that all kestrel instances will have the same rate limit store. Instead of the in memory stores you should inject the distributed stores like this:

```

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

```

Configuration and general rules 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
    }
  ]
}

```

If `EnableEndpointRateLimiting` is set to `false` then the limits will apply globally and only rules that have as endpoint `*` will apply. For example if you set a limit of 5 calls per second, any HTTP call to any endpoint will count towards that limit.

If `EnableEndpointRateLimiting` is set to `true` then the limits will apply for each endpoint as in `{HTTP_Verb}{PATH}`. For example if you set a limit of 5 calls per second for `*/api/values` a client can call `GET /api/values` 5 times per second but also 5 times `PUT /api/values`.

If `StackBlockedRequests` is set to `false` rejected calls are not added to the throttle counter. If a client makes 3 requests per second and you've set a limit of one call per second, other limits like per minute or per day counters will only record the first call, the one that wasn't blocked. If you want rejected requests to count towards the other limits, you'll have to set `StackBlockedRequests` to `true`.

The `RealIpHeader` is used to extract the client IP when your Kestrel server is behind a reverse proxy, if your proxy uses a different header than `X-Real-IP` use this option to set it up.

The `ClientIdHeader` is used to extract the client id for white listing, if a client id is present in this header and matches a value specified in `ClientWhitelist` then no rate limits are applied.

Override general rules for specific IPs `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
        }
      ]
    }
  ]
}
```

The IP field supports IP v4 and v6 values and ranges like "192.168.0.0/24", "fe80::/10" or "192.168.0.0-192.168.0.255".

Defining rate limit rules

A rule is composed of an endpoint, a period and a limit.

Endpoint format is `{HTTP_Verb}:{PATH}`, you can target any HTTP verb by using the asterix symbol.

Period format is `{INT}{PERIOD_TYPE}`, you can use one of the following period types: s, m, h, d.

Limit format is `{LONG}`.

Examples:

Rate limit all endpoints to 2 calls per second:

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

If, from the same IP, in the same second, you'll make 3 GET calls to `api/values`, the last call will get blocked. But if in the same second you call `PUT api/values` too, the request will go through because it's a different endpoint. When endpoint rate limiting is enabled each call is rate limited based on `{HTTP_Verb}{PATH}`.

Rate limit calls with any HTTP Verb to `/api/values` to 5 calls per 15 minutes:

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

Rate limit GET call to `/api/values` to 5 calls per hour:

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

If, from the same IP, in one hour, you'll make 6 GET calls to `api/values`, the last call will get blocked. But if in the same hour you call `GET api/values/1` too, the request will go through because it's a different endpoint.

Behavior

When a client make a HTTP call the `IpRateLimitMiddleware` does the following:

- extracts the IP, Client id, HTTP verb and URL from the request object, if you want to implement your own extraction logic you can override the `IpRateLimitMiddleware.SetIdentity`
- searches for the IP, Client id and URL in the white lists, if any matches then no action is taken
- searches in the IP rules for a match, all rules that apply are grouped by period, for each period the most restrictive rule is used
- searches in the General rules for a match, if a general rule that matches has a defined period that is not present in the IP rules then this general rule is also used
- for each matching rule the rate limit counter is incremented, if the counter value is greater than the rule limit then the request gets blocked

If the request gets blocked then the client receives a text response like this:

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

You can customize the response by changing these options `HttpStatusCode` and `QuotaExceededMessage`, if you want to implement your own response you can override the `IpRateLimitMiddleware.ReturnQuotaExceededResponse`. The `Retry-After` header value is expressed in seconds.

If the request doesn't get rate limited then the longest period defined in the matching rules is used to compose the X-Rate-Limit headers, these headers are injected in the response:

```
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
```

By default blocked request are logged using `Microsoft.Extensions.Logging.ILogger`, if you want to implement your own logging you can override the `IpRateLimitMiddleware.LogBlockedRequest`. The default logger emits the following information when a request gets rate limited:

```
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 0HKTLISQQV9D.
```

Update rate limits at runtime

At application startup the IP rate limit rules defined in `appsettings.json` are loaded in cache by either `MemoryCacheClientPolicyStore` or `DistributedCacheIpPolicyStore` depending on what type of cache provider you are using. You can access the Ip policy store inside a controller and modify the IP rules like so:

```
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);
    }
}

```

This way you can store the IP rate limits in a database and push them in cache after each app start.

Rate limiting based on client ID

With ClientRateLimit middleware you can set multiple limits for different scenarios like allowing a Client to make a maximum number of calls in a time interval like per second, 15 minutes, etc. You can define these limits to address all requests made to an API or you can scope the limits to each URL path or HTTP verb and path.

Setup

NuGet install:

```
Install-Package AspNetCoreRateLimit
```

Startup.cs code:

```

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();
}

```

You should register the middleware before any other components except loggerFactory.

if you load balance your app you'll need to use `IDistributedCache` with Redis or SQLServer so that all kestrel instances will have the same rate limit store. Instead of the in memory stores you should inject the distributed stores like this:

```

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

```

Configuration and general rules 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
  }
]
}

```

If `EnableEndpointRateLimiting` is set to `false` then the limits will apply globally and only rules that have as endpoint `*` will apply. For example if you set a limit of 5 calls per second, any HTTP call to any endpoint will count towards that limit.

If `EnableEndpointRateLimiting` is set to `true` then the limits will apply for each endpoint as in `{HTTP_Verb}{PATH}`. For example if you set a limit of 5 calls per second for `*/api/values` a client can call `GET /api/values` 5 times per second but also 5 times `PUT /api/values`.

If `StackBlockedRequests` is set to `false` rejected calls are not added to the throttle counter. If a client makes 3 requests per second and you've set a limit of one call per second, other limits like per minute or per day counters will only record the first call, the one that wasn't blocked. If you want rejected requests to count towards the other limits, you'll have to set `StackBlockedRequests` to `true`.

The `ClientIdHeader` is used to extract the client id, if a client id is present in this header and matches a value specified in `ClientWhitelist` then no rate limits are applied.

Override general rules for specific clients `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
    }
]
}

```

Defining rate limit rules

A rule is composed of an endpoint, a period and a limit.

Endpoint format is `{HTTP_Verb}:{PATH}`, you can target any HTTP verb by using the asterix symbol.

Period format is `{INT}{PERIOD_TYPE}`, you can use one of the following period types: `s`, `m`, `h`, `d`.

Limit format is `{LONG}`.

Examples:

Rate limit all endpoints to 2 calls per second:

```

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

```

If in the same second, a client make 3 GET calls to `api/values`, the last call will get blocked. But if in the same second he calls `PUT api/values` too, the request will go through because it's a different endpoint. When endpoint rate limiting is enabled each call is rate limited based on `{HTTP_Verb}{PATH}`.

Rate limit calls with any HTTP Verb to `/api/values` to 5 calls per 15 minutes:

```

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

```

Rate limit GET call to `/api/values` to 5 calls per hour:

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

If in one hour, a client makes 6 GET calls to `api/values`, the last call will get blocked. But if in the same hour he calls `GET api/values/1` too, the request will go through because it's a different endpoint.

Behavior

When a client make a HTTP call the `ClientRateLimitMiddleware` does the following:

- extracts the Client id, HTTP verb and URL from the request object, if you want to implement your own extraction logic you can override the `ClientRateLimitMiddleware.SetIdentity`
- searches for the Client id and URL in the white lists, if any matches then no action is taken
- searches in the Client rules for a match, all rules that apply are grouped by period, for each period the most restrictive rule is used
- searches in the General rules for a match, if a general rule that matches has a defined period that is not present in the Client rules then this general rule is also used
- for each matching rule the rate limit counter is incremented, if the counter value is greater then the rule limit then the request gets blocked

If the request gets blocked then the client receives a text response like this:

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

You can customize the response by changing these options `HttpStatusCode` and `QuotaExceededMessage`, if you want to implement your own response you can override the `ClientRateLimitMiddleware.ReturnQuotaExceededResponse`. The `Retry-After` header value is expressed in seconds.

If the request doesn't get rate limited then the longest period defined in the matching rules is used to compose the X-Rate-Limit headers, these headers are injected in the response:

```
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
```

By default blocked request are logged using `Microsoft.Extensions.Logging.ILogger`, if you want to implement your own logging you can override the `ClientRateLimitMiddleware.LogBlockedRequest`. The default logger emits the following information when a request gets rate limited:

```
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.
```


Update rate limits at runtime

At application startup the client rate limit rules defined in `appsettings.json` are loaded in cache by either `MemoryCacheClientPolicyStore` or `DistributedCacheClientPolicyStore` depending on what type of cache provider you are using. You can access the client policy store inside a controller and modify the rules like so:

```
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);
    }
}
```

This way you can store the client rate limits in a database and push them in cache after each app start.

Read Rate limiting online: <https://riptutorial.com/asp-net-core/topic/5240/rate-limiting>

Chapter 20: Routing

Examples

Basic Routing

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

This will match requests for `/Home/Index`, `/Home/Index/123` and `/`

Routing constraints

It is possible to create custom routing constraint which can be used inside routes to constraint a parameter to specific values or pattern.

This constrain will match a typical culture/locale pattern, like `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.IsNullOrEmpty(locale))
            return false;

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

Afterwards, the Constraint needs to be registered before it can be used in routes.

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

Now it can be used within routes.

Using it on Controllers

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

Using it on Actions

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

Using it in Default Routes

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

Read Routing online: <https://riptutorial.com/asp-net-core/topic/2863/routing>

Chapter 21: Sending Email in .Net Core apps using MailKit

Introduction

Currently .Net Core does not include support to send emails like `System.Net.Mail` from .Net. [MailKit project](#) (which is available on [nuget](#)) is a nice library for this purpose.

Examples

Installing nuget package

```
Install-Package MailKit
```

Simple implementation for sending 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);
    }
}
}
```

Read Sending Email in .Net Core apps using MailKit online: <https://riptutorial.com/asp-net-core/topic/8831/sending-email-in--net-core-apps-using-mailkit>

Chapter 22: Sessions in ASP.NET Core 1.0

Introduction

Using Sessions in ASP.NET Core 1.0

Examples

Basic example of handling Session

1)First, add dependency in project.json - "Microsoft.AspNetCore.Session": "1.1.0",

2)In startup.cs and add AddSession() and AddDistributedMemoryCache() lines to the ConfigureServices like this-

```
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)Add the UseSession() call in Configure method of startup like this-

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

4)In Controller, Session object can be used like this-

```
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. If you are using cors policy then sometimes it may give errors, after enabling session regarding headers about enabling *AllowCredentials* header and using

WithOrigins header instead of *AllowAllOrigins*.

Read Sessions in ASP.NET Core 1.0 online: <https://riptutorial.com/asp-net-core/topic/8067/sessions-in-asp-net-core-1-0>

Chapter 23: Tag Helpers

Parameters

Name	Info
asp-action	The name of the action method to which the form should be posted to
asp-controller	The name of the controller where the action method specified in asp-action exists
asp-route-*	Custom route values you want to add as querystring to the form action attribute value. Replace * with the querystring name you want

Examples

Form Tag Helper - Basic example

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

Form Tag Helper - With custom route attributes

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

This will generate the below markup

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

Input Tag Helper

Assuming your view is strongly typed to a view model like

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


And you are passing an object of this to the view from your action method.

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

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

</form>
```

This will generate the below markup.

```
<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>
```

If you want the input field to be rendered with a default value, you can set the Name property value of your view model in the action method.

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

Form submission & Model binding

Model binding will work fine if you use `CreateProduct` as your `HttpPost` action method parameter/a parameter named `name`

Select Tag Helper

Assuming your view is strongly typed to a view model like this

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

And in your GET action method, you are creating an object of this view model, setting the `Categories` property and sending to the view

```
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);
}

```

and in your view

```
@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>

```

This will render the below markup(*included only relevant parts of form/fields*)

```

<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>

```

Getting the selected dropdown value in form submission

You can use the same view model as your HttpPost action method parameter

```

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

```

Set an option as the selected one

If you want to set an option as the selected option, you may simply set the `SelectedCategory` property value.

```

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

```

```
}
```

Rendering a Multi select dropdown/ListBox

If you want to render a multi select dropdown, you can simply change your view model property which you use for `asp-for` attribute in your view to an array type.

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

In the view

```
@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>
```

This will generate the SELECT element with `multiple` attribute

```
<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>
```

Custom Tag Helper

You can create your own tag helpers by implementing `ITagHelper` or deriving from the convenience class `TagHelper`.

- The default convention is to target an html tag that matches the name of the helper without the optional `TagHelper` suffix. For example `WidgetTagHelper` will target a `<widget>` tag.
- The `[HtmlTargetElement]` attribute can be used to further control the tag being targetted
- Any public property of the class can be given a value as an attribute in the razor markup. For example a public property `public string Title {get; set;}` can be given a value as `<widget title="my title">`
- By default, tag helpers translates Pascal-cased C# class names and properties for tag helpers into lower kebab case. For example, if you omit using `[HtmlTargetElement]` and the class name is `WidgetBoxTagHelper`, then in Razor you'll write `<widget-box></widget-box>`.
- `Process` and `ProcessAsync` contain the rendering logic. Both receive a **context** parameter with information about the current tag being rendered and an **output** parameter used to

customize the rendered result.

Any assembly containing custom tag helpers needs to be added to the `_ViewImports.cshtml` file (Note it is the assembly being registered, not the namespace):

```
@addTagHelper *, MyAssembly
```

Sample Widget Custom Tag Helper

The following example creates a custom widget tag helper that will target razor markup like:

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

Which will be rendered as:

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

The coded needed to create such a tag helper is the following:

```
[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);
    }
}
```

Label Tag Helper

Label Tag Helper can be used to render `label` for a model property. It replaces method

`Html.LabelFor` in previous versions of MVC.

Let's say you have a model:

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

In the view you can use `label` HTML element and `asp-for` tag helper:

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

This is equivalent to the following code in earlier versions of MVC:

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

Both code snippets above render the same HTML:

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

Anchor tag helper

Anchor tag helper is used generate href attributes to link to a particular controller action or MVC route. Basic example

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

Sometimes, we need to specify additional parameters for the controller action that you are binding to. We can specify values for these parameters by adding attributes with the `asp-route-` prefix.

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

Read Tag Helpers online: <https://riptutorial.com/asp-net-core/topic/2665/tag-helpers>

Chapter 24: View Components

Examples

Create a View Component

View components encapsulate reusable pieces of logic and views. They are defined by:

- A ViewComponent class containing the logic for fetching and preparing the data for the view and deciding which view to render.
- One or more views

Since they contain logic, they are more flexible than partial views while still promoting a good separation of concerns.

A simple custom view component is defined as:

```
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>
```

They can be invoked from any view (or even a controller by returning a ViewComponentResult)

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

Login View Component

The default project template creates a partial view **_LoginPartial.cshtml** which contains a bit of logic for finding out whether the user is logged in or not and find out its user name.

Since a view component might be a better fit (as there is logic involved and even 2 services injected) the following example shows how to convert the LoginPartial into a view component.

View Component class

```
public class LoginViewComponent : ViewComponent
{
    private readonly SignInManager<ApplicationUser> signInManager;
```

```

private readonly UserManager<ApplicationUser> userManager;

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

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

```

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

```

@model WebApplication1.Models.ApplicationUser

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

```

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

```

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

```

Invocation from **_Layout.cshtml**

```

@await Component.InvokeAsync("Login")

```

Return from Controller Action

When inheriting from base `Controller` class provided by the framework, you can use the convenience method `ViewComponent()` to return a view component from the action:

```

public IActionResult GetMyComponent()
{

```

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

If using a POCO class as a controller, you can manually create an instance of the `ViewComponentResult` class. This would be equivalent to the code above:

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

Read View Components online: <https://riptutorial.com/asp-net-core/topic/3248/view-components>

Chapter 25: Working with JavascriptServices

Introduction

According to official documentation:

`JavaScriptServices` is a set of technologies for ASP.NET Core developers. It provides infrastructure that you'll find useful if you use Angular 2 / React / Knockout / etc. on the client, or if you build your client-side resources using Webpack, or otherwise want to execute JavaScript on the server at runtime.

Examples

Enabling webpack-dev-middleware for asp.net-core project

Let's say you use `Webpack` for front end bundling. You can add `webpack-dev-middleware` to serve your statics through tiny and fast server. It allows you to automatically reload your assets when content has changed, serve statics in memory without continuously writing intermediate versions on disk.

Prerequisites

NuGet

Install-Package Microsoft.AspNetCore.SpaServices

npm

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

Configuring

Extend `Configure` method in your `Startup` class

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

Add Hot Module Replacement (HMR)

Hot Module Replacement allows to add, change or delete app module when application is running. Page reloading is not needed in this case.

Prerequisites

In addition to `webpack-dev-middleware` packages:

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

Configuration

Simply update configuration of `UseWebpackDevMiddleware` with new options:

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

You also need to accept hot modules in your app code.

HMR is supported for Angular 2, React, Knockout and Vue.

Generating sample single page application with asp.net core

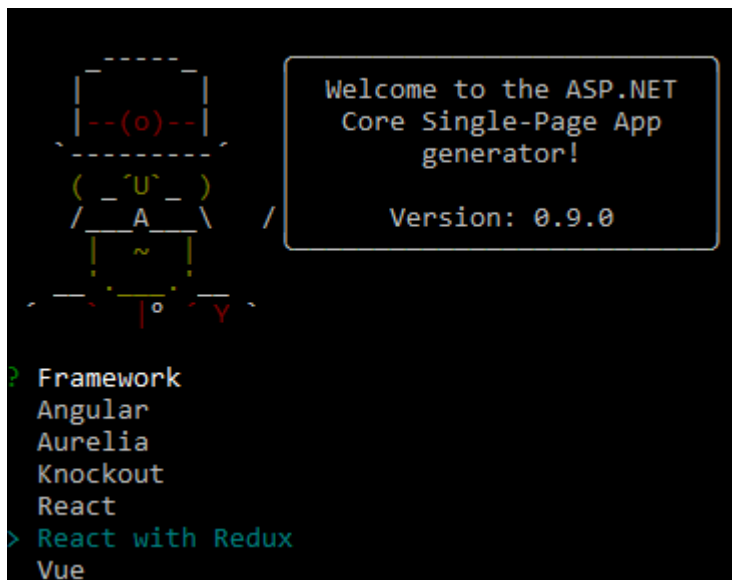
You can use `aspnetcore-spa` generator for `Yeoman` to create brand-new single page application with asp.net core.

This allows you to choose one of the popular front end frameworks and generates project with webpack, dev server, hot module replacement and server-side rendering features.

Just run

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

and choose your favorite framework



Read Working with JavascriptServices online: <https://riptutorial.com/asp-net-core/topic/9621/working-with-javascriptservices>

Credits

S. No	Chapters	Contributors
1	Getting started with asp.net-core	Alex Logan , Alexan , Ashish Rajput , Ashley Medway , Bogdan Stefanjuk , BrunoLM , ChadT , Community , gbellmann , Henk Mollema , Nate Barbettini , Rion Williams , Shog9 , Shyju , Svek , Tseng , VSG24 , Zach Becknell
2	Angular2 and .Net Core	Alejandro Tobón , Sentient Entities
3	ASP.NET Core - Log both Request and Response using Middleware	Gubr
4	Authorization	gilmishal , RamenChef
5	Bundling and Minification	Rion Williams , Zach Becknell
6	Caching	Cyprien Autexier , Sanket
7	Configuration	Cyprien Autexier , Jayantha Lal Sirisena
8	Configuring multiple Environments	dotnetom , Johnny , Robert Paulsen , Sanket , Set , Tseng
9	Cross-Origin Requests (CORS)	Henk Mollema , Sanket , Saqib Rokadia , Tseng
10	Dependency Injection	Alexan , BrunoLM , Cyprien Autexier , Dan Soper , Darren Evans , gilmishal , Gurgen Hakobyan , Jayantha Lal Sirisena , Joel Harkes , maztt , Tseng , Zach Becknell
11	Error Handling	Sanket , Set
12	Injecting services into views	Alex Logan , Rion Williams
13	Localization	Tseng , VSG24 , Zach Becknell
14	Logging	Dmitry , Sanket , Set , Tseng
15	Middleware	Ali , Piotrek , Set , VSG24 , Zach Becknell

16	Models	Alex Logan , Ralf Bönning
17	project.json	Joel Harkes
18	Publishing and Deployment	Set
19	Rate limiting	Stefan P.
20	Routing	ChadT , Tseng
21	Sending Email in .Net Core apps using MailKit	Ankit
22	Sessions in ASP.NET Core 1.0	ravindra , Sanket
23	Tag Helpers	Ali , Daniel J.G. , dotnetom , Shyju , tmg , Zach Becknell
24	View Components	Daniel J.G.
25	Working with JavascriptServices	hmnzr