Softinux Base Documentation Documentation

Release Latest

Benjamin Nolmans, Barbara Post

Jul 16, 2019

Installation

1 SOFTINUX Base Documentation 1

2 Basic Concepts 3

Index 29

CHAPTER 1

SOFTINUX Base Documentation

Warning: The documentation is not complete and therefore undergoes frequent changes.

SOFTINUX Base is a free, open source and cross-platform based on and framework. It runs on Windows, Mac and Linux. It is built using the best and the most modern tools and languages.

It is completely modular and extendable.

Using the features of the underlying ExtCore framework you can easily create your own extensions to extend its functionality.

CHAPTER 2

Basic Concepts

Softinux Base is a framework that looks like a .NET Core web application, but is intended to host mini web applications called extensions. Every extension will plug its content (pages, menu items) as well as security and authentication related items (permissions, roles, links...).

Base manages the common stuff so that the developer can focus on its extension and business logic, just having to provide what we call metadata to know how to display and authorize access to content, and use our version of Authorize attribute.

2.1 Installation

2.1.1 Restore npm packages

After cloning *Base* repository, go to Barebone folder and run `npm ci --save-dev` command so that dependencies packages are installed and settings updated.

Note: You must have to restore web dependencies.

2.1.2 Restore nuGet packages

Restoring the nuGet packages is now an implicit command executed at application build so you don't need to do it manually.

2.1.3 Update database with migration

Go to WebApplication folder and run `dotnet ef database update`.
This will create the database.
See *appsettings.json* ("ConnectionStrings:Default" section) for database path.

The Entity Framework database context is defined in web application's *Startup.cs* (line with `services_.AddDbContext<...`).
We use Sqlite but you can change this easily.

2.1.4 Build the application

Go to the root folder and run `bp.bat` under Windows or `bp.sh` under Linux/Macos. (use -h for help).

Note: You must have to compile and build the application.

2.1.5 Configure the application

The application have some values to configure in `appsettings.json` file.
Theses values are stored into sections:
- Extensions : this is the path to find Extensions. **Important :** see *extensions folder*.
If you wish to change this path, read *what changes to make*.
- ConnectionStrings : the connection configuration to database. See to help you configure.
- Corporate : the name and logo for the application
- RestSeed : identification used to create admin user.

See *configuration section* for a full explanation.

2.1.6 Run the app

Warning: Remove the `SeedDatabase.dll` to avoid any attempt to create a new administrator. See *RestSeed* configuration section.

Go to WebApplication folder and type `dotnet run`.
(If you want, you can also execute from root solution folder with this command `dotnet run --project WebApplication\WebApplication.csproj`).

After that, the application is available on http://localhost:5000/

Note about Visual Studio 2017

If you launched application from Visual Studio, this port will change, being randomly defined, and value is stored in *WebApplication/Properties/launchSettings.json*
You can edit this value in Visual Studio: WebApplication's properties > Debug tab > Web Server Settings/App URL or directly in launchSettings file.
After, the default port used by *dotnet run* is the port defined in *WebApplication/Properties/launchSettings.json*.

Note about Rider 2017.3

Rider 2017.3 cannot execute the PostBuildEvent declared into WebApplication.csproj
You need to execute `./bp.sh copyexts` and `./bp.sh copydeps` after building the solution or project.
Have a look after *Rider useful configuration section*.

2.1.7 Add the administrator user

With Postman (or the program of your choice) make a POST request to this url:
http://localhost:5000/dev/seed/create-user
By command line:

- curl: `curl -i -X POST -H 'Content-Type: application/json' http://localhost:5000/dev/seed/create-user -d {}`
- powershell: `Invoke-WebRequest -Uri http://localhost:5000/dev/seed/create-user -Method POST`

This will create the administrator user with general permissions.

Note: Actually, we creating demo user. The first user is johndoe.

2.1.8 Login with demo user

user: johndoe@softinux.com or johndoe
password: 123_Password
(password is case sensitive)

2.2 Configuration

The configuration is stored into `appsettings.json` file.
Only the following sections are read by *Base*:

- Extensions
- ConnectionStrings
- Corporate
- RestSeed
- SignIn
- LockoutUser
- ValidateUser
- PasswordStrategy
- ConfigureApplicationCookie
- Logging

- Serilog

You can add others sections, but it's up to you to read them.

2.2.1 Extensions

By default, extensions ares stored into `WebApplication/Extensions` folder.
But you can change this if you need. If you make that, you must change the variables into the build script:

- *bp.bat* for windows
- *bp.sh* for *nix system.

You have four variables :

- `netVersion`: folder name defined by .NET Core TargetFramework tag into cs proj file.
- `ext_folder`: extensions folder path.
- `dep_folder`: dependencies folder path.
- `pub_folder`: publish folder path.

```
:: set .NET output folder name (use .NET Core version defined into csproj files)
set netVersion="netcoreapp2.2"
:: Extensions folder
set ext_folder=".\WebApplication\Extensions\"
:: Dependencies folder
set dep_folder=".\WebApplication\bin\Debug\%netVersion%\"
:: Publish folder
set pub_folder=".\WebApplication\bin\Debug\%netVersion%\publish"
```

2.2.2 ConnectionStrings

In this section, you can configure your database connection.
The file come with commented examples of connections strings.

```
"ConnectionStrings": {
    // Please use '/' for directory separator
    "Default": "Data Source=basedb.sqlite"
    // SqlServer
    //"Default": "Data Source=localhost;Initial Catalog=Softinux;
↪MultipleActiveResultSets=True;Persist Security Info=True;User ID=softinux;Password=?
↪"
    // PostgreSql
    //"Default": "Host=localhost;Port=5432;Database=softinux;Pooling=true;User␣
↪ID=softinux;Password=?;"
    // localdb
    //"Default": "Data Source=(localdb)\mssqllocaldb;Database=softinux;Trusted_
↪Connection=True;MultipleActiveResultSets=true"
  }
```

2.2.3 Corporate

Here you can set you Company name and logo.

```
"Corporate": {
  "Name": "SOFTINUX",
  "BrandLogo": "softinux_logo-bg-transparent.png"
}
```

The logo is to be place into : `wwwroot\img`

2.2.4 RestSeed

Here is the **SECRET** configuration for create first user.
The first user is the application administrator.

```
"RestSeed": {
  "UserName": "",
  "UserPassword": "",
  "Id": "",
  "Guid": ""
}
```

You need to set these values.
Id and Guid is used into REST api call to create admin user.

Warning: Is strongly recommended to remove the `SeedDatabase.dll` to avoid any attempt to create a new administrator. This can happen if you change the information in the configuration file and restart the application.

2.2.5 SignIn, LockoutUser, ValidateUser, PasswordStrategy, ConfigureApplicationCookie

These settings are used by .

2.2.6 Logging

This is the standard .NET Core *Logging* configuration.

2.2.7 Serilog

This is the nuGet package configuration. This allows to log to a file.

Here we'll describe what to know about extensions and how to customize things.

2.3 Extension structure

2.3.1 ExtCore concepts

Read ExtCore documentation to learn about extensions and how they are structured into several projects.

2.3.2 Embedded resources

In your .csproj, you'll find this:

```
<ItemGroup>
  <EmbeddedResource Include="Styles\**;Scripts\**\*.min.js;Views\**" />
</ItemGroup>
```

So that your embedded styles, scripts and views are embedded.

You'll also find complementary stuff like this, to be sure that any file used in your project but provided by another project is correctly built as an embedded resource only:

```
<ItemGroup>
  <None Remove="Views\SomeView.cshtml" />
  <None Remove="... path_to_some_file_of_other_project.js" />
</ItemGroup>
<ItemGroup>
  <EmbeddedResource Include="... path_to_some_file_of_other_project.js" />
</ItemGroup>
```

2.3.3 Bundling

Bundling is a convenient way to save bandwith and processor time when dealing with .css files etc. This is not specific to our project but we share our preferred way of doing this, so you would do the same in your extension project:

We use a *bundleconfig.json* file in concerned projects and the .csproj contains something like this:

```
<DotNetCliToolReference Include="BundlerMinifier.Core" Version="2.8.391" />
```

As a side note, embedded resources are bundled first.

2.3.4 Base's common interface

In your extension main project, a class should implement the `Infrastructure.IExtensionMetadata` interface, so that the application knows what the extension provides in matter of display (menu items...).

We usually name it `ExtensionMetadata`.

Menu groups and menu items

Menu groups are ordered by position then alphabetically.
They're not displayed if they contain no menu items. The first occurrence of a menu group defines the associated icon. Menu items (of a menu group) are ordered by position.

General useful properties

`Base.Infrastructure.IExtensionMetadata` and `ExtCore.Infrastructure.IExtensionMetadata` interfaces will require implementation of some properties. We recommend using the following code, using assembly attributes.

```
/// <summary>
/// Gets the current assembly object.
/// </summary>
public Assembly CurrentAssembly => Assembly.GetExecutingAssembly();

/// <summary>
/// Gets the full path with assembly name.
/// </summary>
public string CurrentAssemblyPath => CurrentAssembly.Location;

/// <summary>
/// Gets the name of the extension.
/// </summary>
public string Name => CurrentAssembly.GetName().Name;

/// <summary>
/// Gets the URL of the extension.
/// </summary>
public string Url => Attribute.GetCustomAttribute(CurrentAssembly,␣
↪typeof(AssemblyTitleAttribute)).ToString();

/// <summary>
/// Gets the version of the extension.
/// </summary>
public string Version => Attribute.GetCustomAttribute(CurrentAssembly,␣
↪typeof(AssemblyVersionAttribute)).ToString();

/// <summary>
/// Gets the authors of the extension (separated by commas).
/// </summary>
public string Authors => Attribute.GetCustomAttribute(CurrentAssembly,␣
↪typeof(AssemblyCompanyAttribute)).ToString();

/// <summary>
/// Gets the description of the extension (separated by commas).
/// </summary>
public string Description => Attribute.GetCustomAttribute(CurrentAssembly,␣
↪typeof(AssemblyDescriptionAttribute)).ToString();
```

2.3.5 MVC structure

Controllers

Your controllers should inherit from `Infrastructure.ControllerBase` so that you have access to storage layer (`IStorage`) and optionally logging (`ILoggerFactory`).

Additional configuration to web application

Any implementation of the `ExtCore.Infrastructure.Actions.IConfigureServicesAction` interface allows you to define your injections to the web application services container.
Please use Priority above 1000, the values below are reserved to project.

2.3.6 Utilities

Logging

When you need logging, use `ILoggerFactory` from your controller and instantiate a private logger in your class with:

```
ILogger _logger = _loggerFactory.CreateLogger(GetType().FullName);
```

Then you can adjust log level in app's configuration.

2.3.7 Authentication

Introduction

Our application uses claims to grant access to protected pages.
The `Security.Common` extension manages authenticated access to the application by decorating controllers or controllers' methods.
The `Security` extensions allows to manage authentication data (administration).

Permissions, Scopes and Claims

An extension defines its scope (assembly simple name) so that the Admin, Write and Read permissions are granted by scope. There is also the global scope that is named "Security".
In administration interface you can manage how the permissions are granted.

In your extensions controllers, use `PermissionRequirementAttribute` or `AnyPermissionRequirementAttribute` attribute from `Security.Common.Attributes`.
Then provide the permission level (see `Security.Common.Enums.Permission enumeration`) and scope (extension assembly short name without the version and culture stuff).

A custom claim of type Permission will be created for every scope, its value being the highest permission level.
For example, if the *Write* and *Read* checkboxes are checked for a given scope in administration page, the highest granted permission level is *Write* and the claim will have *Write* value.

You will be able to use it to filter menu items too (work in progress, issue #9).

2.4 Create your extensions

Warning: You cannot place your web application's Extensions folder to another drive. See #2981

You can use Visual Studio 2017, Visual Studio Code or JetBrains Rider to make your own extension. If you decide to use Visual Studio, **be aware that projects are not compatible with Visual Studio 2015**.

2.4.1 What there is to know

Warning: You cannot place your web application's Extensions folder to another drive. See #2981

You can use Visual Studio 2017, Visual Studio Code or JetBrains Rider to make your own extension. If you decide to use Visual Studio, **be aware that projects are not compatible with Visual Studio 2015**.

In this section, we talking of SampleApi. This project is availiable on Github to :

Todo: add git repos for sample app

2.4.2 New Extension with Base source

Use Base solution and add your extension code into it.

Add a new project

Using command-line (easy and cross-platform):

```
$ dotnet new classlib -o <your_new_project> -f netcoreapp2.2
```

Assuming Base's `Infrastructure` framework version is 2.2. Check its .csproj file.

If you don't specify framework version, it will default to `netstandardxxx`, which is not what we expect.

Add project reference to the solution

Go to solution folder and type:

```
$ dotnet sln add <path_to_your_new_project_csproj>
```

Write your code

In your new project, add a reference to Base's `Infrastructure` and also `Security.Common`.
Then create a `ExtensionMetadata` class that implements `Infrastructure.IExtensionMetadata`.

Have a look at *write your extensions*, feel free to open issues for questions.

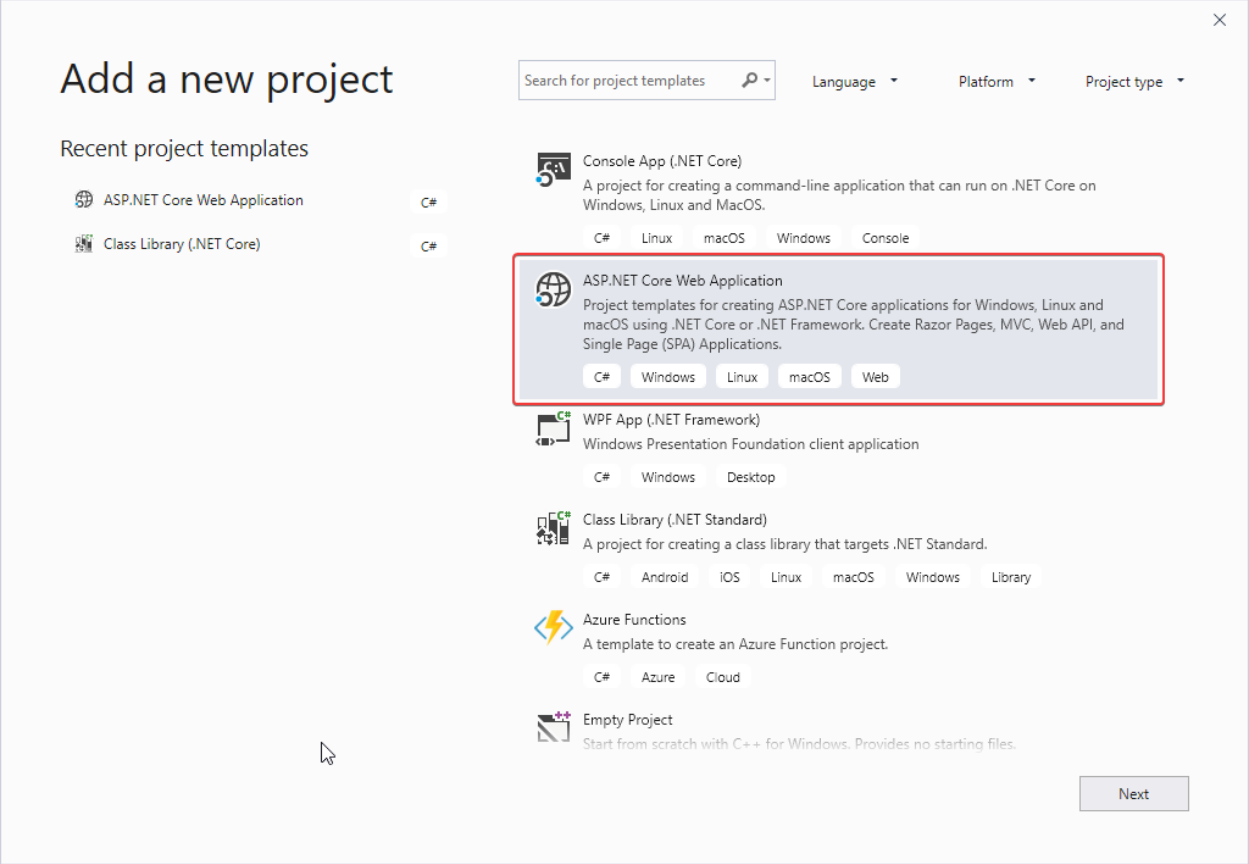
2.4.3 Using Base as dependency

Use your own solution and Base as a dependency. This is an alternative to using Base's solution.

Configure a new project with Visual Studio 2017/2019

Create new solution with a new ASP.NET Core project targeted on framework .NET Core 2.2.

Creation in Visual Studio 2019



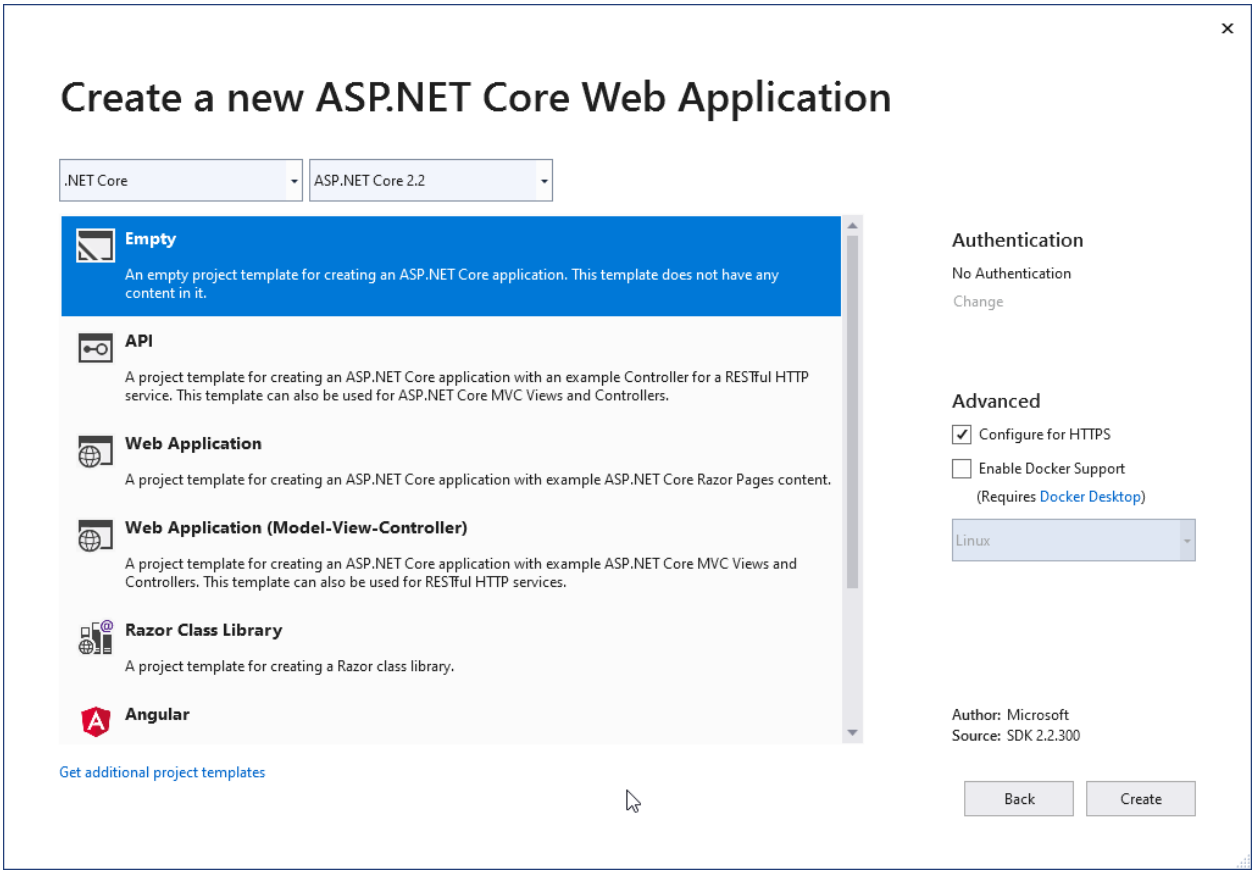
Creation in Visual Studio 2017

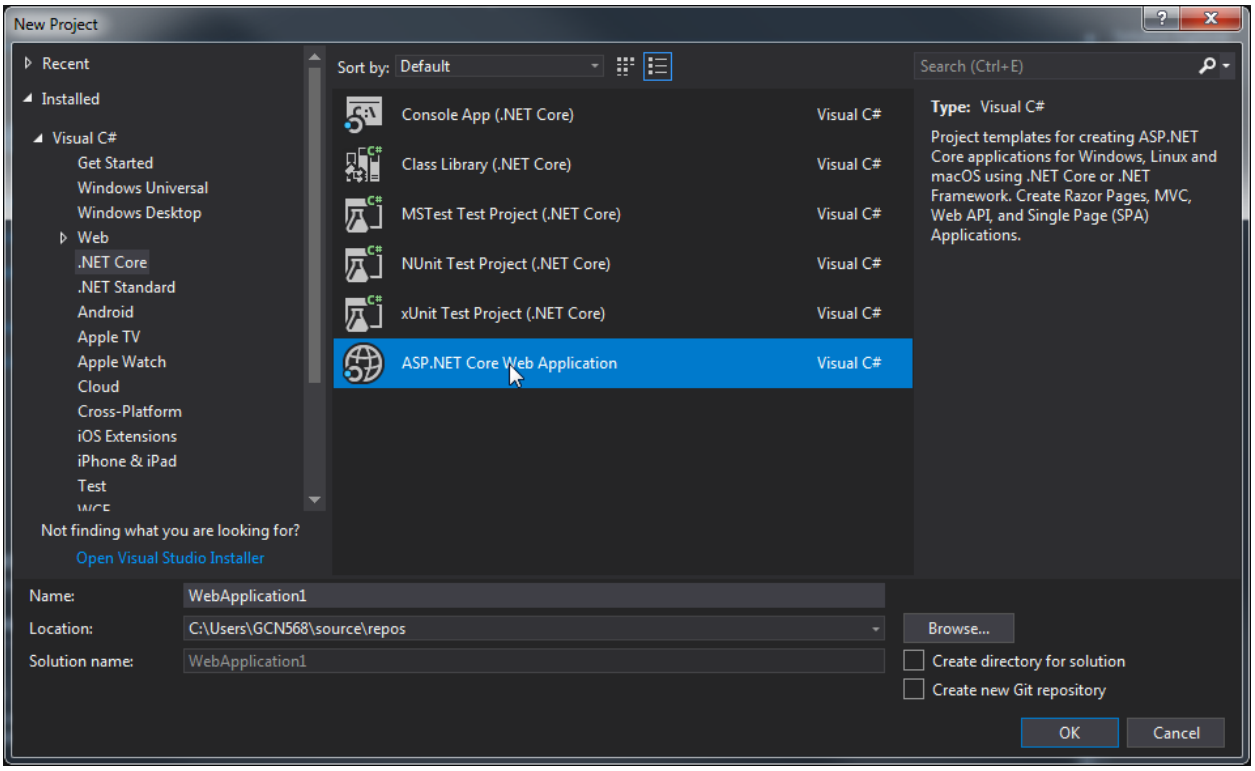
Verification

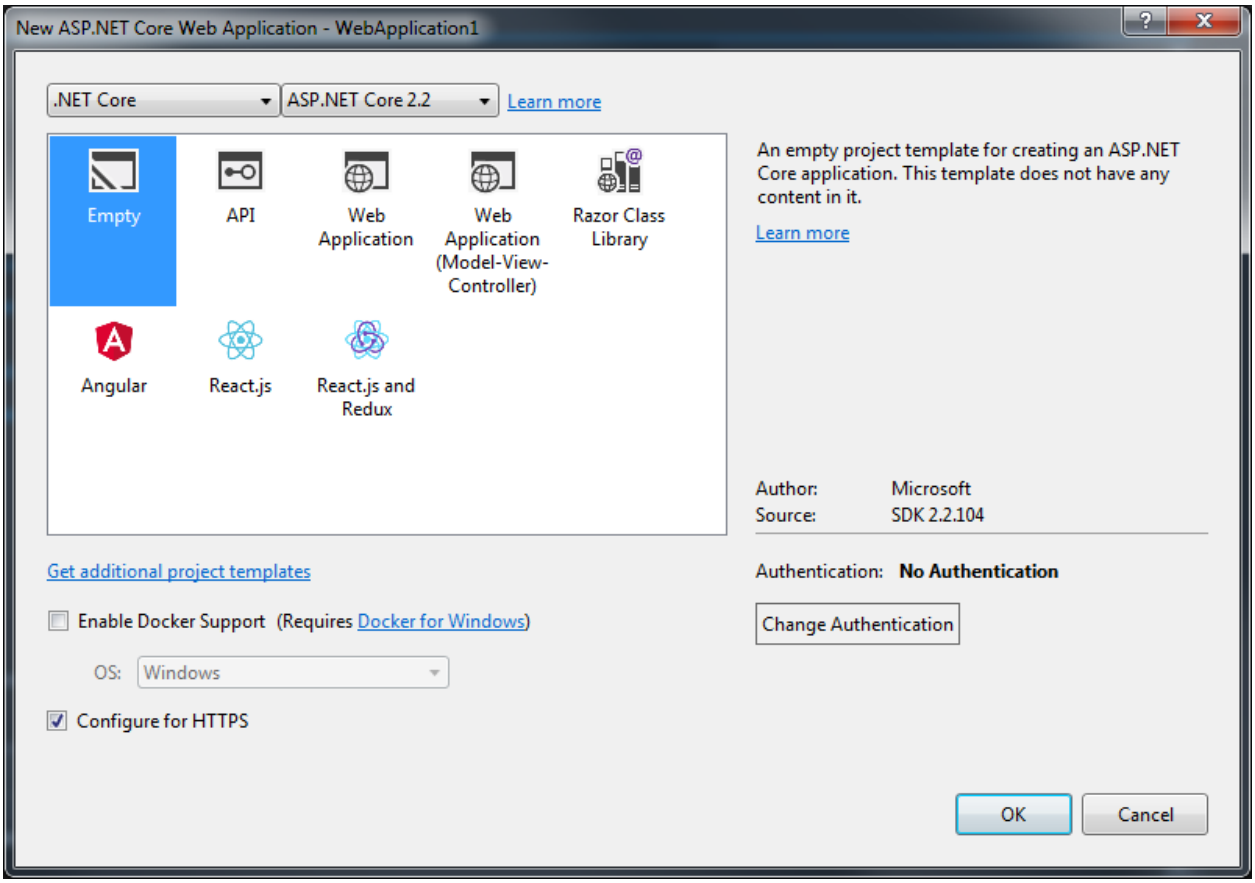
Check if your new project is targeted on framework .NET Core 2.2.

Add references

Add references to the Base and ExtCore (ExtCore is a dependency of Base).


Create a new ASP.NET Core Web Application
.NET Core
ASP.NET Core 2.2
Empty
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
Web Application
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content.
Web Application (Model-View-Controller)
A project template for creating an ASP.NET Core application with example ASP.NET Core MVC Views and Controllers. This template can also be used for RESTful HTTP services.
Razor Class Library
A project template for creating a Razor class library.
Angular
Get additional project templates
Authentication
No Authentication
Change
Advanced
Configure for HTTPS
Enable Docker Support
(Requires Docker Desktop)
Linux
Author: Microsoft
Source: SDK 2.2.300
Back
Create


New Project
Recent
Installed
Visual C#
Get Started
Windows Universal
Windows Desktop
Web
.NET Core
.NET Standard
Android
Apple TV
Apple Watch
Cloud
Cross-Platform
iOS Extensions
iPhone & iPad
Test
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
MSTest Test Project (.NET Core) Visual C#
NUnit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Type: Visual C#
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework. Create Razor Pages, MVC, Web API, and Single Page (SPA) Applications.
Name: WebApplication1
Location: C:\Users\GCN568\source\repos
Solution name: WebApplication1
Browse...
Create directory for solution
Create new Git repository
OK
Cancel


New ASP.NET Core Web Application - WebApplication1
.NET Core
ASP.NET Core 2.2
Learn more
Empty
API
Web Application
Web Application (Model-View-Controller)
Razor Class Library
Angular
React.js
React.js and Redux
An empty project template for creating an ASP.NET Core application. This template does not have any content in it.
Learn more
Author: Microsoft
Source: SDK 2.2.104
Get additional project templates
Authentication: No Authentication
Enable Docker Support (Requires Docker for Windows)
Change Authentication
OS: Windows
Configure for HTTPS
OK
Cancel

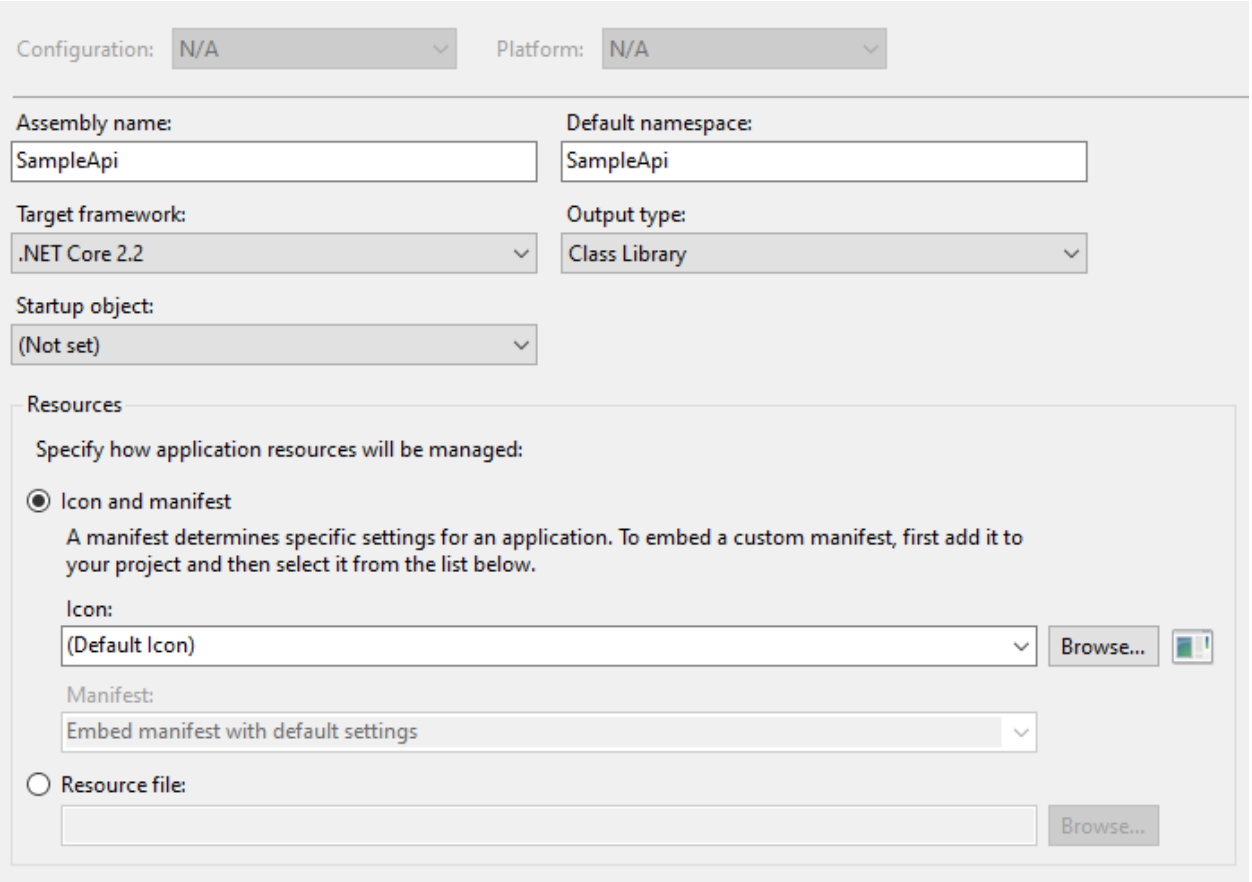
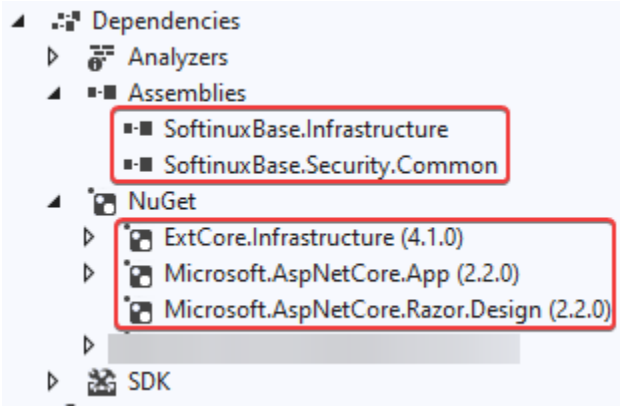
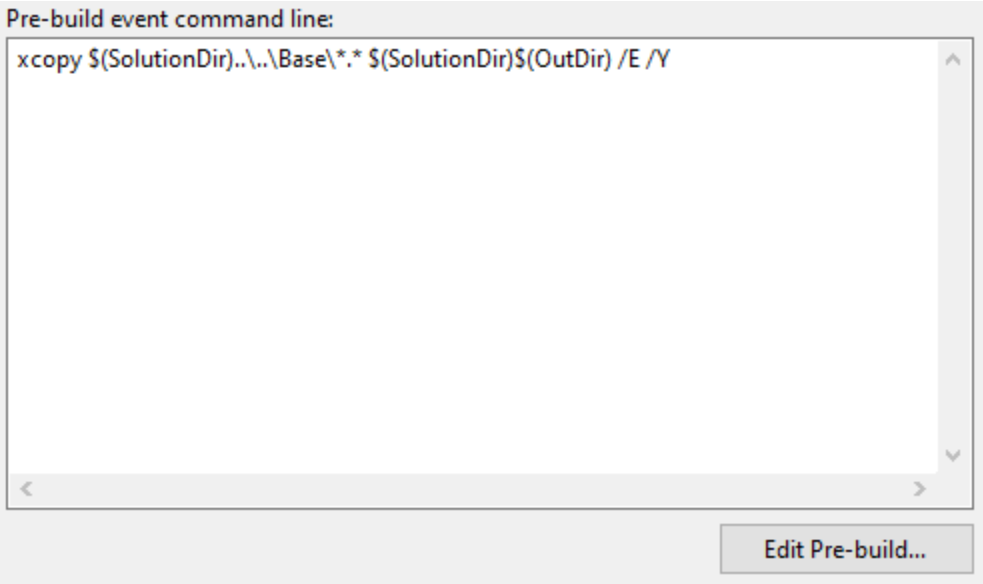


Fig. 1: Project properties, Application tab.



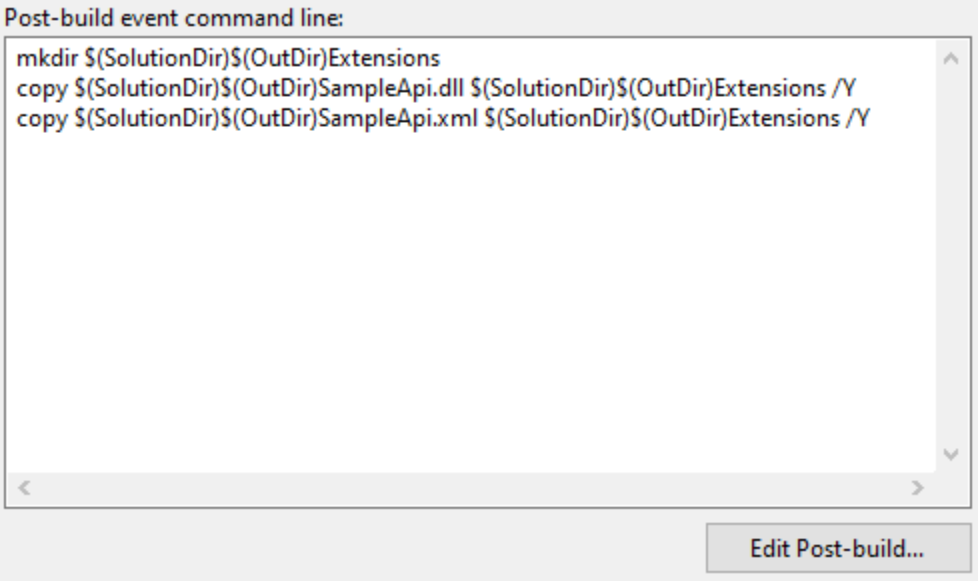
Configure pre-build scripts

Before building, you need to copy all Base dependencies to *$(SolutionDir)$(OutDir)* folder:



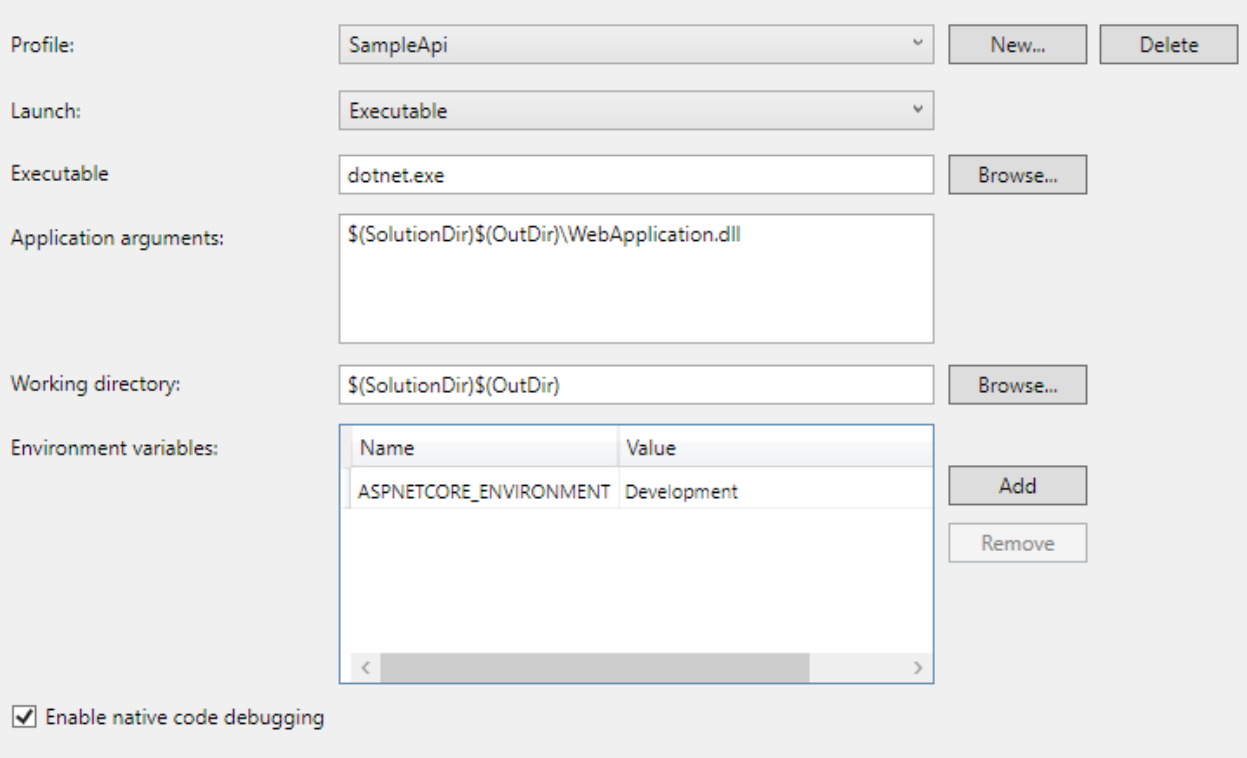
Configure post-build scripts

After building, you need to copy your extension into Base's extensions folder:



Configure debug tab

Most important, configure debugging. Your extension is a partial app and is not directly executed. Here is how to configure your application to enable possibility of debugging.



Now, you can debug your extension into Visual Studio.

With commande line and Visual Studio Code

Create a new project

```
$ dotnet new classlib -o <your_new_project> -f netcoreapp2.2
```

Open your new .csproj file and **adapt it with highlighted lines** as in example:

Listing 1: SampleApi csproj file

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

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
    <ApplicationIcon />
    <OutputType>Library</OutputType>
    <StartupObject />
  </PropertyGroup>

  <PropertyGroup Condition="'$(Configuration)|$(Platform)'=='Debug|AnyCPU'">
    <DocumentationFile>$(BaseOutputPath)bin\$(Configuration)\$(TargetFramework)\
↪$(AssemblyName).xml</DocumentationFile>
    <NoWarn>1701;1702;1591</NoWarn>
  </PropertyGroup>

  <ItemGroup>
```

(continues on next page)

(continued from previous page)

```
17      <EmbeddedResource Include="Styles\**;Scripts\**\*.min.js;Views\**" />
18    </ItemGroup>
19
20    <ItemGroup>
21      <PackageReference Include="ExtCore.Infrastructure" Version="4.1.0" />
22      <PackageReference Include="Microsoft.AspNetCore.App" />
23      <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0"␣
↪PrivateAssets="All" />
24      <PackageReference Include="Swashbuckle.AspNetCore" Version="4.0.1" />
25    </ItemGroup>
26
27    <ItemGroup>
28      <Reference Include="SoftinuxBase.Infrastructure, Version=0.0.1.0, Culture=neutral,
↪ PublicKeyToken=null">
29        <HintPath>..\..\Base\SoftinuxBase.Infrastructure.dll</HintPath>
30      </Reference>
31      <Reference Include="SoftinuxBase.Security.Common, Version=0.0.1.0,␣
↪Culture=neutral, PublicKeyToken=null">
32        <HintPath>..\..\Base\SoftinuxBase.Security.Common.dll</HintPath>
33      </Reference>
34    </ItemGroup>
35
36    <PropertyGroup>
37      <SolutionDir Condition=" '$(SolutionDir)' == '' ">
↪$([MSBuild]::GetDirectoryNameOfFileAbove($(MSBuildThisFileDirectory), SampleApi.
↪sln))</SolutionDir>
38    </PropertyGroup>
39
40    <Target Name="PreBuild" BeforeTargets="PreBuildEvent">
41      <Exec Command="xcopy $(SolutionDir)..\..\Base\*.* $(SolutionDir)$(OutDir) /E /Y" /
↪>
42    </Target>
43
44    <Target Name="PostBuild" AfterTargets="PostBuildEvent">
45      <Exec Command="mkdir $(SolutionDir)$(OutDir)Extensions&#xD;&#xA;copy
↪$(SolutionDir)$(OutDir)SampleApi.dll $(SolutionDir)$(OutDir)Extensions /Y&#xD;&#xA;
↪copy $(SolutionDir)$(OutDir)SampleApi.xml $(SolutionDir)$(OutDir)Extensions /Y" />
46    </Target>
47
48  </Project>
```

Note:

Path in <*HintPath*> are given as examples.
Lines 36 to 38 set the value of the Visual Studio *$(SolutionDir)* macro because *dotnet* doesn't use it.

Visual Studio Code Configuration

Tasks.json

Add lines 26 to 29 and 38 to 40.
Modify line 35 to use `WebApplication.dll` as entry point of application.

Note: The order sequence makes build on every launch.

Listing 2: SampleApi Visual Studio Code tasks file

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "command": "dotnet",
            "type": "process",
            "args": [
                "build",
                "${workspaceFolder}/src/SampleApi/SampleApi.csproj"
            ],
            "problemMatcher": "$tsc"
        },
        {
            "label": "publish",
            "command": "dotnet",
            "type": "process",
            "args": [
                "publish",
                "${workspaceFolder}/src/SampleApi/SampleApi.csproj"
            ],
            "problemMatcher": "$tsc"
        },
        {
            "label": "watch",
            "dependsOrder": "sequence",
            "dependsOn":[
                "build"
            ],
            "command": "dotnet",
            "type": "process",
            "args": [
                "watch",
                "run",
                "${workspaceFolder}/src/SampleApi/bin/Debug/netcoreapp2.2/
↪WebApplication.dll"
            ],
            "problemMatcher": "$tsc",
            "presentation": {
                "reveal": "always",
                "panel": "new"
            }
        }
    ]
}
```

Launch.json

Modify line 13 to use `WebApplication.dll` as the program to execute.
Modify line 15 to specify execution folder.

Listing 3: SampleApi Visual Studio Code launch file

```
{
   // Use IntelliSense to find out which attributes exist for C# debugging
   // Use hover for the description of the existing attributes
   // For further information visit https://github.com/OmniSharp/omnisharp-vscode/blob/master/debugger-launchjson.md
   "version": "0.2.0",
   "configurations": [
        {
            "name": ".NET Core Launch (web)",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            // If you have changed target frameworks, make sure to update the program path.
            "program": "${workspaceFolder}/src/SampleApi/bin/Debug/netcoreapp2.2/WebApplication.dll",
            "args": [],
            "cwd": "${workspaceFolder}/src/SampleApi/bin/Debug/netcoreapp2.2/",
            "stopAtEntry": false,
            // Enable launching a web browser when ASP.NET Core starts. For more information: https://aka.ms/VSCode-CS-LaunchJson-WebBrowser
            "serverReadyAction": {
                "action": "openExternally",
                "pattern": "^\\s*Now listening on:\\s+(https?://\\S+)"
            },
            "env": {
                "ASPNETCORE_ENVIRONMENT": "Development"
            },
            "sourceFileMap": {
                "/Views": "${workspaceFolder}/Views"
            }
        },
        {
            "name": ".NET Core Attach",
            "type": "coreclr",
            "request": "attach",
            "processId": "${command:pickProcess}"
        }
    ]
}
```

2.5 About Entity Framework

By definition, ExtCore uses Entity Framework but provides several projects to define:

- the entities in `YourExtension.Data.Entities`
- the entities mapping in `YourExtension.Data.EntityFramework` (`EntityRegistrar` class)
- the EF provider to actually use, in `YourExtension.Data.EntityFramework.ProviderName`

The `SecurityTest` test project in `Testing/Unit` references the three aforementioned projects related to `Security` extension

and also uses `CommonTest.ApplicationStorageContext` class to indicate the DbContext structure.

2.6 Internals

Implementations of `IConfigureServicesAction` They register services implementations to web application container so that they become available for dependency injection (ExtCore feature).

Security project:

- priority 200: ConfigureAuthentication
- priority 201: AddAuthorizationPolicies

Implementations of `IConfigureAction` They record web application's request pipelines (ExtCore feature).

Security project:

- priority 100: ActivateAuthentication

2.7 Unit testing

2.7.1 Introduction

We use xUnit and its shared context feature. Our base project is in *Testing/Unit/CommonTest*.
It contains the `DatabaseFixture` class, that does several things:

- read configuration files, register services (same principle as web application's Startup)
- expose ExtCore core components such as `IStorage` to test classes
- expose Identity `RoleManager` and `UserManager` to test classes

In addition, to perform an EF migration, an implementation of `IDesignTimeDbContextFactory` has been provided, as CommonTest isn't a console but library project.

The test projects use an identical database to the one web application uses, but empty.

2.7.2 How to setup a test project

When you want to create a migration, be sure that your test project adds references to these projects:

- your extension's entities project (`YourExtension.Data.Entities`)
- your extension's EF project where lives *entities registrar* and repositories implementations (`YourExtension.Data.EntityFramework`)

If you just want to use ExtCore's *repositories* pattern to query DB, reference your extension's repositories project `YourExtension.Data.EntityFramework`.

2.7.3 Running tests

- Perform any necessary migration (at least from *Testing/Unit/CommonTest*, with `dotnet ef database update`).

- If testing with VS Code IDE, we use dotnet-test-explorer extension with some configuration in *.vscode/settings.json* (workspace configuration file).

2.8 How to log

We've integrated Serilog by associating it to the logger factory that ASP.NET Core creates at application startup.

Log level is defined in `appsettings.json` of web application, sections "Logging" and "Serilog".

To log a custom message, inject `Microsoft.Extensions.Logging.ILoggerFactory` into your class constructor.
Then instantiate your logger:

```
Microsoft.Extensions.Logging.ILogger myLogger = _loggerFactory.CreateLogger(GetType().
↪FullName);
```

and log:

```
myLogger.LogInformation("Hello");
```

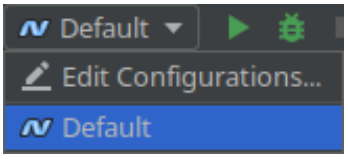
2.9 Configure Rider

Note: This page is for Rider 2018.2 and upper.

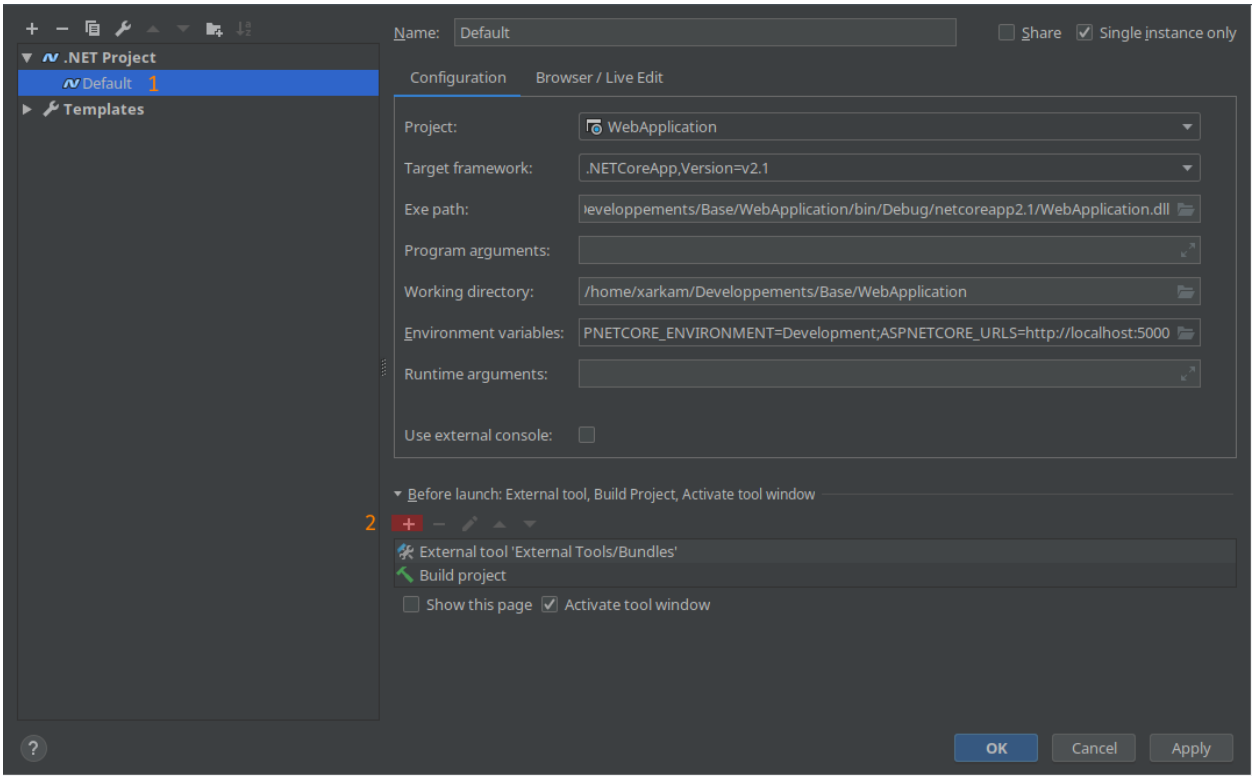
Rider doesn't use all `.sln` tag to build your application.
In this page, we show to configure Rider to build bundles before build the application.

2.9.1 Create an external tool

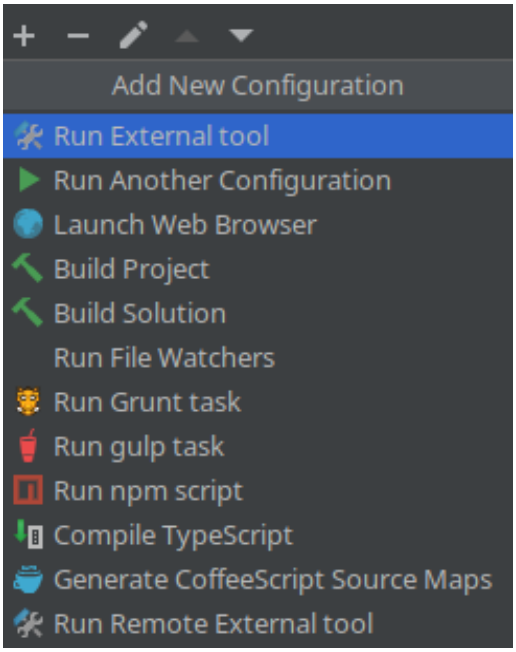
Click on edit configuration



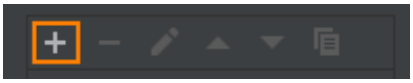
If you have already one configuration, click on it



And click the plus sign in section Before launch (number 2 on picture).
In popup menu, select external tool

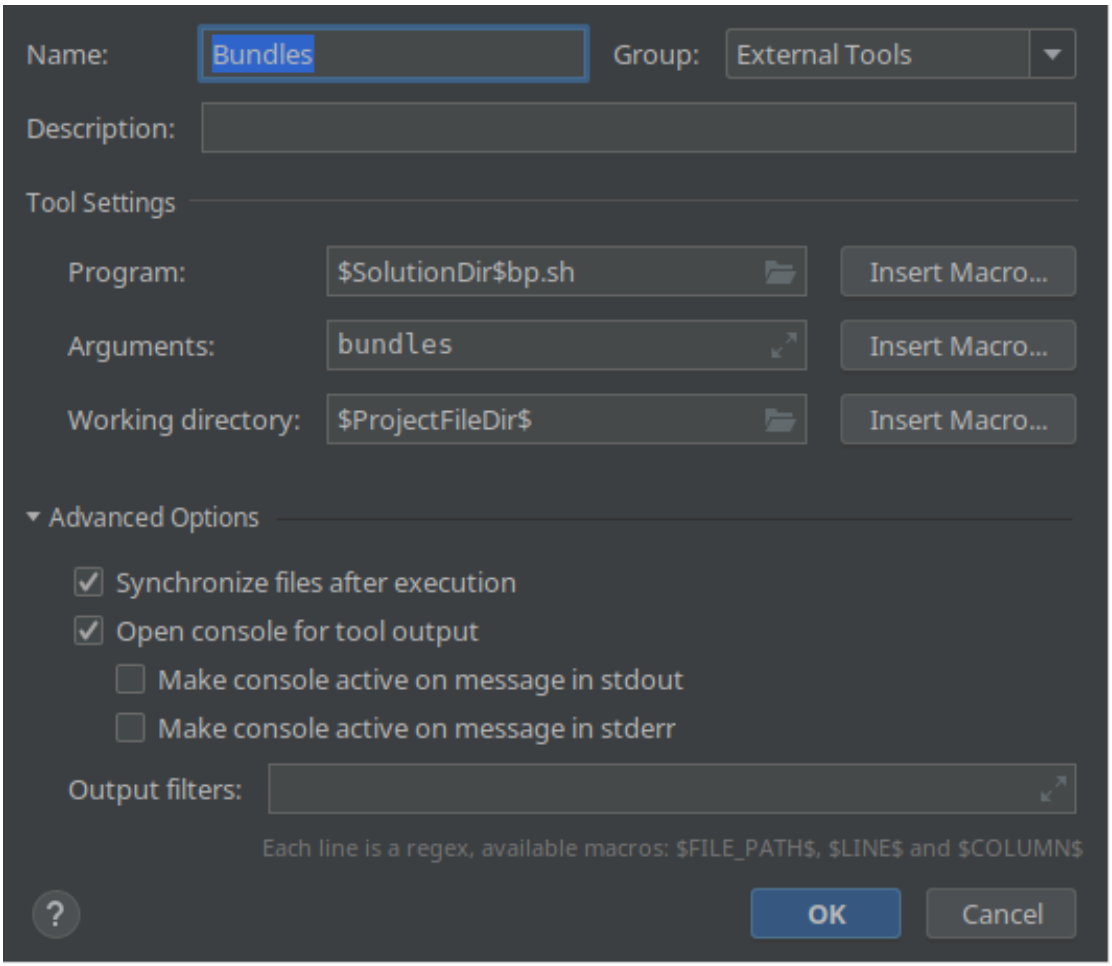


In new window click on plus sign:

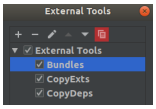


Now, in external tool configuration window:

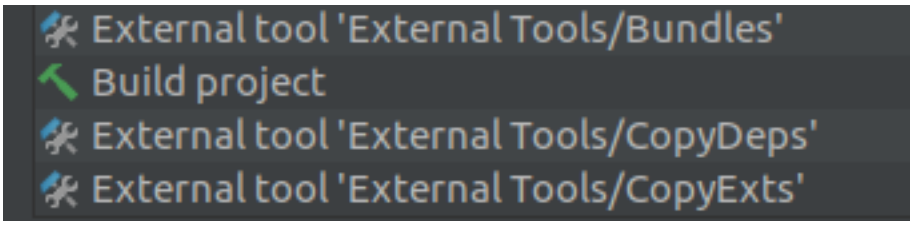
1. enter a name for your new external tool configuration.
2. in program field, enter same text as screen shot. Help yourself with macros.
3. in arguments field enter bundles.
4. working directory is auto completed.
5. click on save.



Once you've configured this external tool, copy it and create the two other ones:



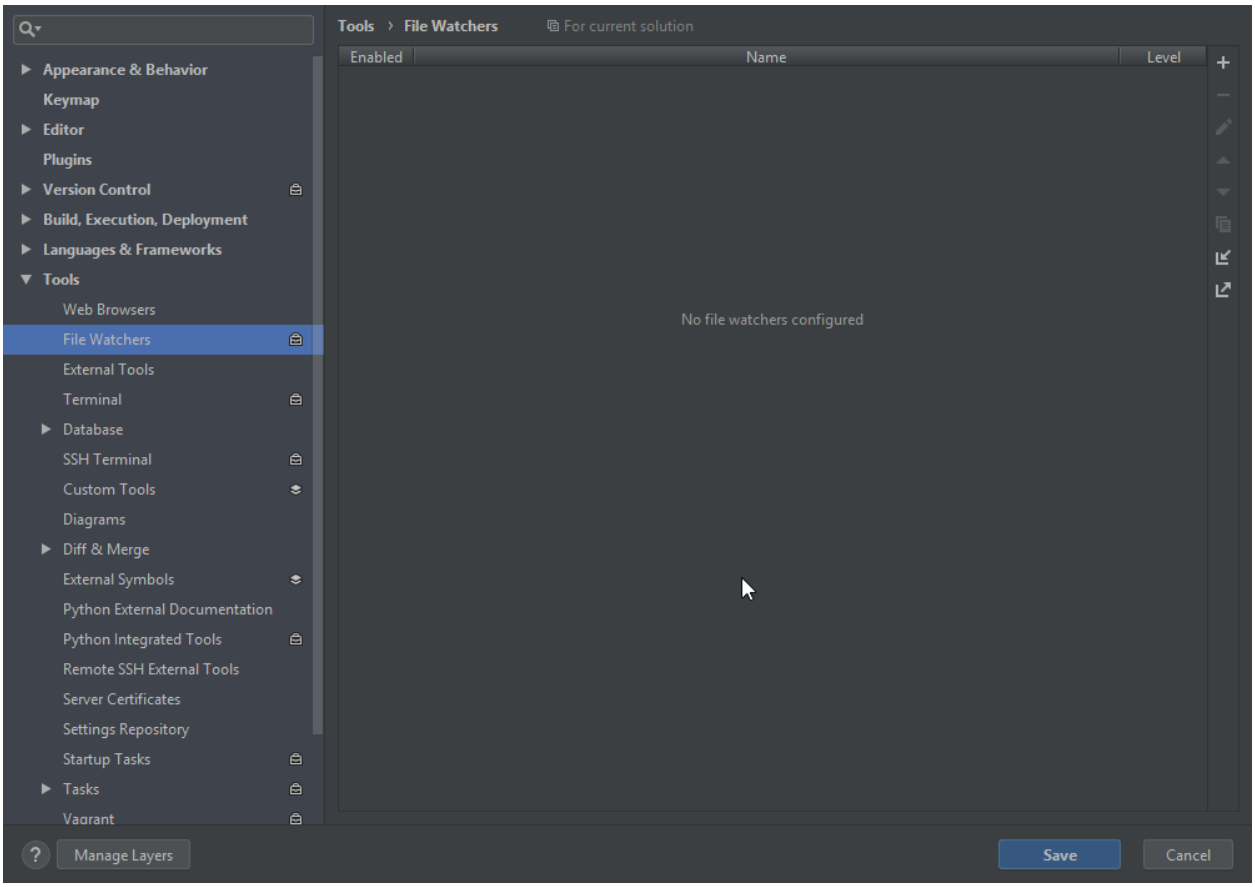
Change argument field to `copyexts` for the second external tool and `copydeps` for the third external tool. Be sure you have the external tools and the project build tasks in this order:



2.9.2 Create an file watcher for javascript minification

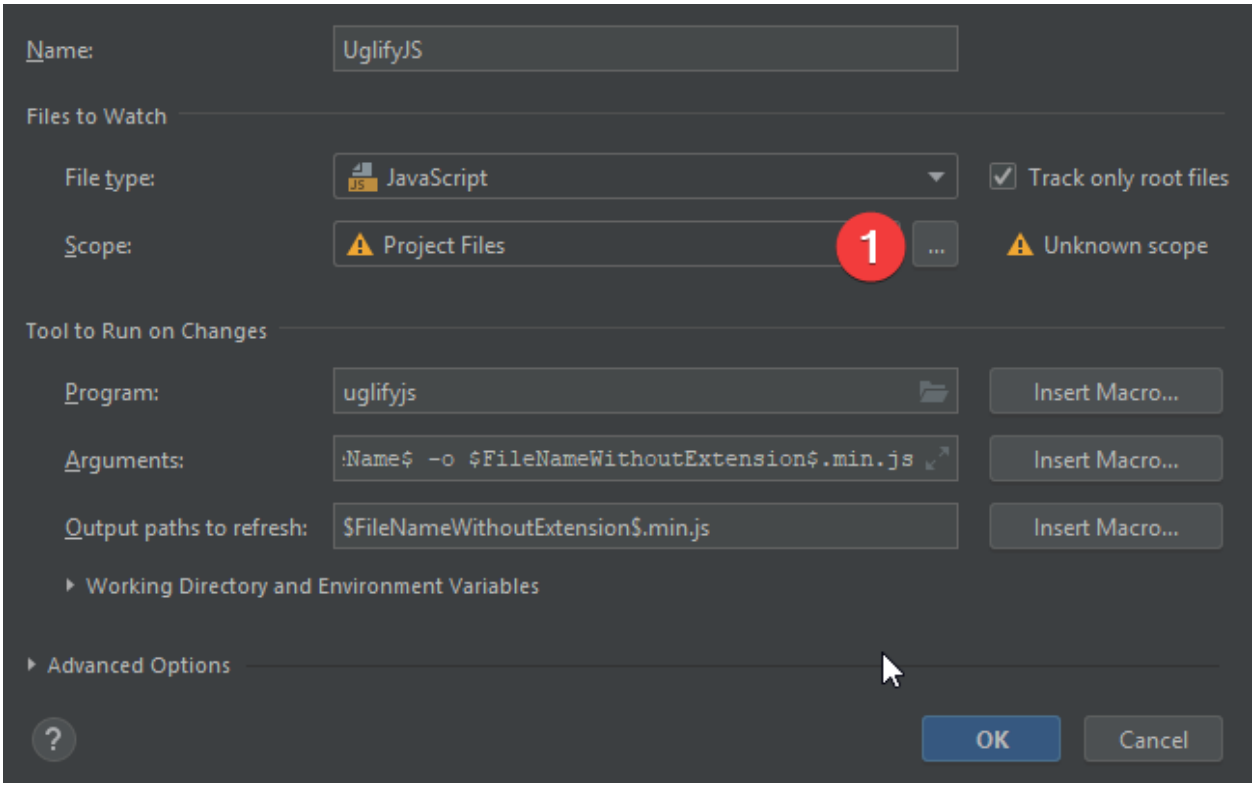
In this example, we use Uglifyjs. You can install with nodejs by `npm install uglify-js -g`.

Goto Settings (`Ctrl` + `Alt` + `S`), section Tools -> File Watchers

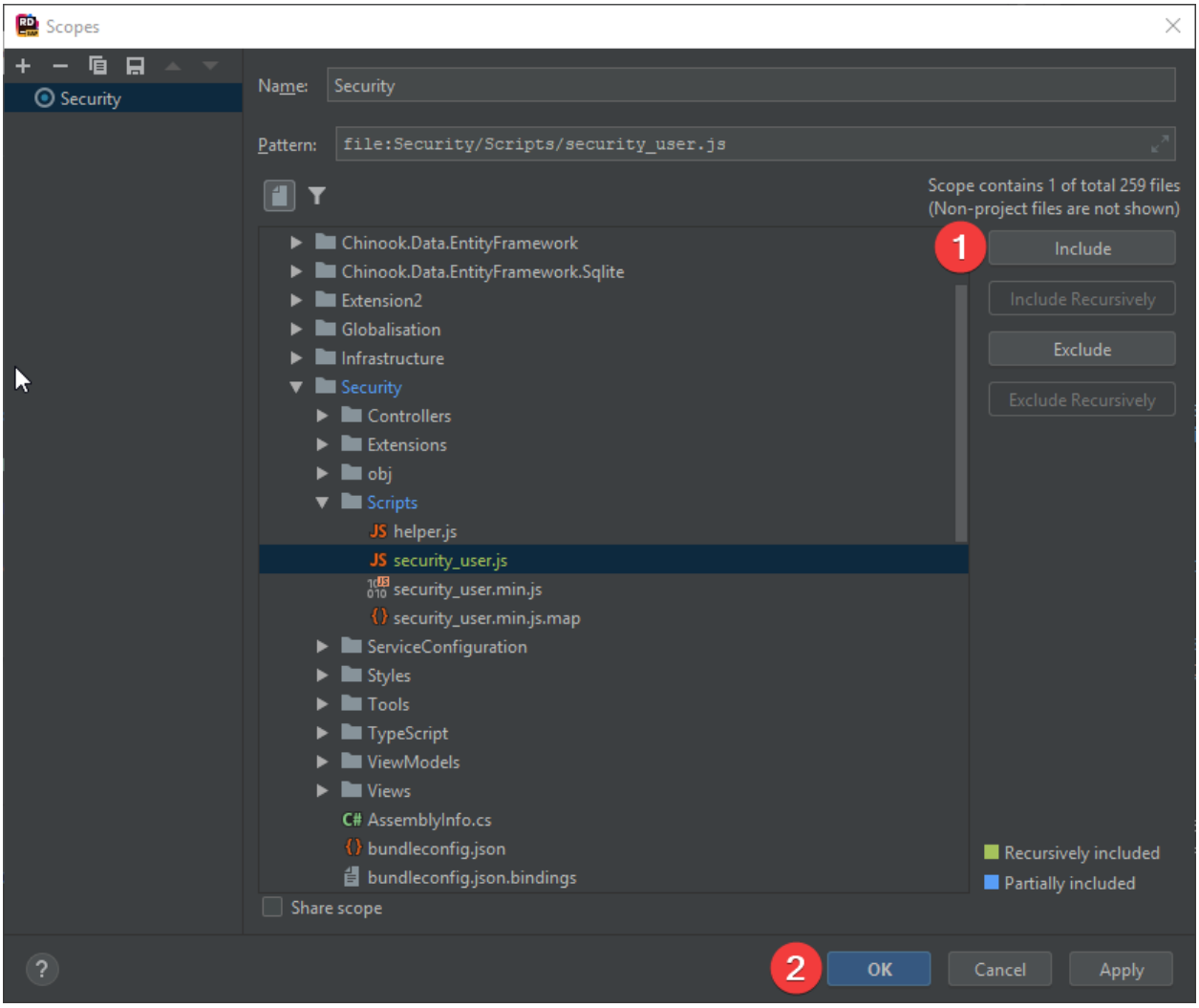


In right of window, click on + sign (or `Ctrl` + `n`) to add a new file watcher.
Select Uglify in list and give a name to your new file watcher.

In Edit Watcher window, click on three dot of Scope field.



In Scope window, select your javascript file and click to add.



Finish by clicking on Ok. Close all settings windows.

2.10 Faq for Linux

Q. I have this message during the build:
`Permission denied for editing the folder '/usr/share/dotnet/sdk/NuGetFallbackFolder'.`
A.: You need to execute `dotnet restore` with root privilege because, the current user ave not right to write into `/usr/share/dotnet/sdk/NuGetFallbackFolder`

Q. Th extension .NET Core Text Explorer cannot find unit Test
A.: The problem is due to `Permission denied for editing the folder '/usr/share/dotnet/sdk/NuGetFallbackFolder'.`
You must declare and set the `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` environment variable to 1 (or true)

2.11 TODO

As mentioned before, all this is work-in-progress.

2.11.1 List of TODOs

The following list is automatically created by the Sphinx TODO plugin. If there is no list, either all TODOs are done (very unlikely), or they are disabled with the option `todo_include_todos = False` in the file `conf.py`.

Todo: add git repos for sample app

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/softinux-base/checkouts/latest/source/implement_your_extension/what_you_need_to_know.rst, line 13.)

Index

D

dep_folder, 6

E

environment variable
    dep_folder, 6
    ext_folder, 6
    netVersion, 6
    pub_folder, 6
ext_folder, 6

N

netVersion, 6

P

pub_folder, 6