.NET Core Service Configuration

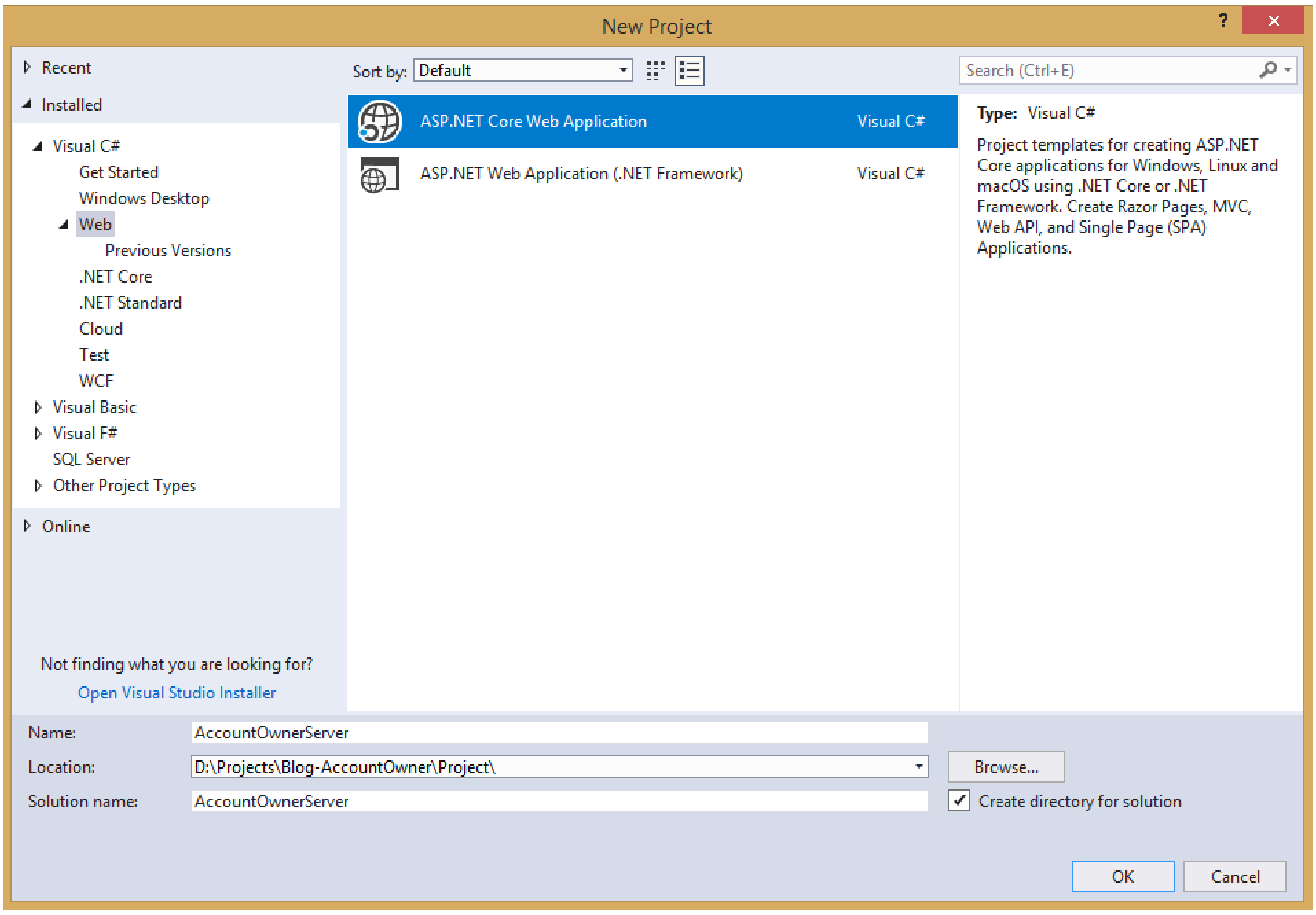
.NET Core configuration differs greatly from standard .NET projects. We don't have a web.config file but instead, we use a built-in Configuration framework.

Therefore having a good understanding of how to configure your project and how to configure the services you will use is a must.

This lesson will explain how to use the configuration methods in the Startup class, how to register different services and how to use extension methods to help you achieve this.

Creating a New Project and Modifying LaunchSettings.json File

After we have finished creating and populating the database, we are going to create a Visual Studio project for the server part of our application. Open Visual Studio and create a new ASP.NET Core Web Application with the name `AccountOwnerServer`:

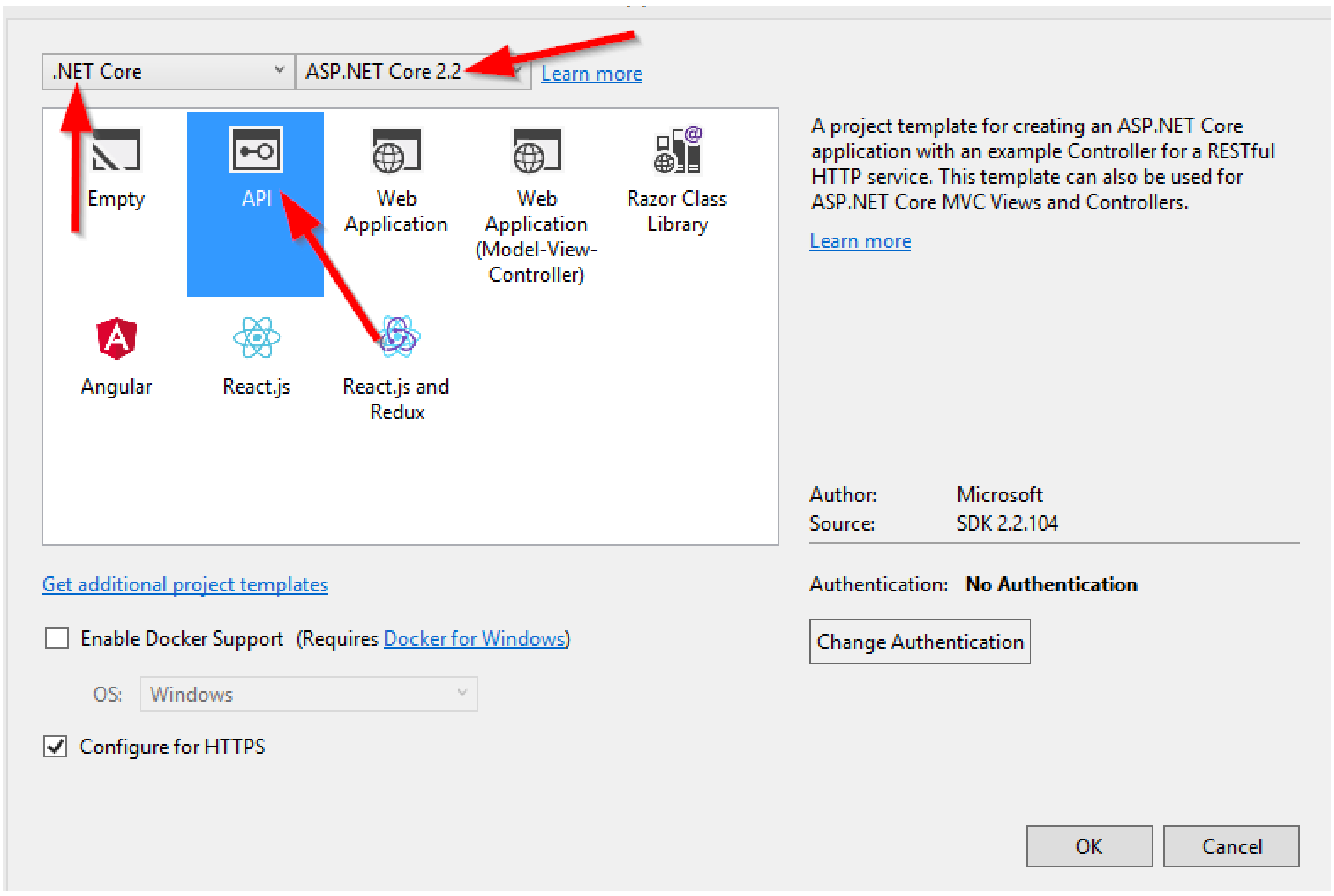


Type: Visual C#

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: AccountOwnerServer
Location: D:\Projects\Blog-AccountOwner\Project\
Solution name: AccountOwnerServer
[✓] Create directory for solution

In the next window choose Web API and from the left drop-down list choose .NET Core. Also, from the right drop-down choose ASP.NET Core 2.2. After all that, just click the OK button and the project will load.



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.

Learn more

Author: Microsoft
Source: SDK 2.2.104

Authentication: **No Authentication**

Change Authentication

☐ Enable Docker Support (Requires Docker for Windows)

OS: Windows

☑ Configure for HTTPS

After the project has been created, we are going to modify the `launchSettings.json` file, which is quite important file for the .NET Core configuration. Let's change the `applicationUrl` property and the `launchBrowser` property to `false`, to prevent a browser from launching when the project starts.

In the Solution Explorer expand the Properties and double click on the `launchSettings.json` file. Let's modify that file:

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": false,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "AccountOwnerServer": {
      "commandName": "Project",
      "launchBrowser": false,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

Program.cs and Startup.cs Explanations

In `Program.cs` you will find this piece of code:

```
public static void Main(string[] args)
{
    CreateWebHostBuilder(args).Build().Run();
}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .UseStartup<Startup>();
```

Notice this part of the code: `UseStartup<Startup>()`. The `Startup` class is mandatory for .NET Core; it's where we configure embedded or custom services that our application needs. When you open the `Startup` class, you can see the constructor and the two methods, the content of which we will change during application development. In the method `ConfigureServices`, you will do exactly that, configure your services. Furthermore, in the method `Configure` you are going to add different middleware components to the application's request pipeline.

All of our configuration code could be written inside the `ConfigureServices` method, but large applications could potentially contain many services. As a result, it could make this method unreadable and hard to maintain. Therefore we will create extension methods for each configuration and place the configuration code inside those methods.

Extension Methods and CORS Configuration

An extension method is inherently a static method. They play a great role in .NET Core configuration because they increase readability of our code. What makes extension methods different from other static methods is that they accept `this` as the first parameter, and `this` represents the data type of the object that uses that extension method. An extension method must be inside a static class. This kind of method extends the behavior of the types in .NET.

So, let's start writing some code.

Create a new folder `Extensions` in the project and create a new class inside that folder, with the name `ServiceExtensions`. We are going to make that class static and inside that class, we will add our service extension methods:

```
namespace AccountOwnerServer.Extensions
{
    public static class ServiceExtensions
    {
    }
}
```

First, we need to configure CORS in our application. CORS (Cross-Origin Resource Sharing) is a mechanism that gives rights to the user to access resources from the server on a different domain. Because our client-side app will run on a different domain than the server's domain, configuring CORS is mandatory. So add this code to the `ServiceExtensions` class:

```
public static void ConfigureCors(this IServiceCollection services)
{
      services.AddCors(options =>
      {
          options.AddPolicy("CorsPolicy",
              builder => builder.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader()
              .AllowCredentials());
      });
}
```

We are using the basic settings for adding CORS policy because for this project allowing any origin, method, and header is quite enough. But you can be more restrictive with those settings if you want. Instead of the `AllowAnyOrigin()` method which allows requests from any source, you could use `WithOrigins("http://www.something.com")` which will allow requests just from the specified source. Also, instead of `AllowAnyMethod()` that allows all HTTP methods, you can use `WithMethods("POST", "GET")` that will allow only specified HTTP methods. Furthermore, you can make the same changes for the `AllowAnyHeader()` method by using, for example, the `WithHeaders("accept", "content-type")` method to allow only specified headers.

IIS Configuration as Part of .NET Core Configuration

Additionally, you need to configure an IIS integration which will help us with IIS deployment. Add the following code to the `ServiceExtensions` class:

```
public static void ConfigureIISIntegration(this IServiceCollection services)
{
      services.Configure<IISOptions>(options =>
      {

      });
}
```

We do not initialize any of the properties inside the options because we are fine with the default values.

In the `Startup` class we will change the `ConfigureServices` and `Configure` methods:

```
public void ConfigureServices(IServiceCollection services)
{
      services.ConfigureCors();

      services.ConfigureIISIntegration();

      services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
}

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
        app.UseHsts();
    }

    app.UseHttpsRedirection();

    app.UseCors("CorsPolicy");

    app.UseForwardedHeaders(new ForwardedHeadersOptions
    {
        ForwardedHeaders = ForwardedHeaders.All
    });

    app.UseStaticFiles();

    app.UseMvc();
}
```

In the `ConfigureServices` method, CORS and IIS configuration have been added. Furthermore, CORS configuration has been added to the application's pipeline inside the `Configuration` method. But as you may notice, there is a little more code.

- `app.UseForwardedHeaders` will forward proxy headers to the current request. This will help us with Linux deployment.
- `app.UseStaticFiles()` enables using static files for the request. If we don't set a path to the static files, it will use a `wwwroot` folder in our solution explorer by default.

Conclusion

Concerning .NET Core configuration, you know how to modify the `launchSettings.json` file, the purpose of the `Program` and the `Startup` classes, how to configure services and finally how to create and use extension methods.