MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
DEC 17, 2018 3:00 AM PST

How to enable CORS in ASP.Net Core

Take advantage of the CORS middleware in ASP.Net Core to bypass the security restrictions of the web browser and allow cross-origin requests

The same-origin policy is a standard security mechanism in web browsers that allows communications between two URLs only if they share the same origin, meaning the same protocol, port, and host. For example, a client or script at http://localhost:6000 will not be able to access a server application at http://localhost:5080 because these two URLs have different port addresses. Security restrictions in your web browser will not allow requests to a server application in another domain.

Here is where CORS (Cross-Origin Resource Sharing) comes to the rescue. CORS is a W3C standard that allows you to get around the default same-origin policy adopted by the browsers. In short, you can use CORS to allow some cross-origin requests while preventing others. In this article we'll examine how CORS can be enabled and configured in ASP.Net Core.

[What is TypeScript? Industrial-strength JavaScript. • See the new features in TypeScript's frequent updates with InfoWorld's TypeScript version feature tracker. | Keep up with hot topics in programming with InfoWorld's App Dev Report newsletter.]

Create an ASP.Net Core Web API project in Visual Studio 2017

First off, let's create an ASP.Net Core Web API project in Visual Studio. If Visual Studio 2017 is up and running in your system, follow the steps outlined below to create an ASP.Net Core Web API project.

1. Launch the Visual Studio 2017 IDE.
2. Click on File > New > Project.
3. Select “ASP.Net Core Web Application (.Net Core)” from the list of templates displayed.
4. Specify a name for the project.
5. Click OK to save the project.
6. A new window “New .Net Core Web Application...” will be displayed.
7. Select “.Net Core” as the runtime and ASP.NET Core 2.1 (or later) from the drop-down list at the top.
8. Select “API” as the project template.
9. Ensure that the check boxes "Enable Docker Support" and “Configure for HTTPS” are unchecked. We won’t be using Docker or HTTPS here.
10. Ensure that “No Authentication” is selected as we won’t be using authentication either.

This will create a new ASP.Net Core project in Visual Studio. We’ll use this project to enable and configure CORS in the sections that follow. Let’s call this project the server application. You can now create another .Net Core Web Appication project in Visual Studio to serve as the client application.

Note that if you try to access the controller methods of the server application from the client application by making AJAX calls, you'll see that the web browser rejects the request. This is because CORS isn't enabled in the server application.

Add CORS to the ASP.Net Core request processing pipeline

To work with CORS in ASP.Net Core, these are the steps we need to follow:

1. Install the CORS middleware.
2. Add the CORS middleware to the pipeline in the ConfigureServices method.
3. Enable CORS in the Configure method.
4. Enable CORS in the controllers, the action methods, or globally.

The Microsoft.AspNetCore.Cors package is the CORS middleware that can be leveraged in ASP.Net Core to enable cross-origin resource sharing. To install this package, click on Tools > NuGet Package Manager > Manage NuGet Packages for Solution. Then search for theMicrosoft.AspNetCore.Cors package in the NuGet package manager and install it. As of this writing, the latest stable version of the Microsoft.AspNetCore.Cors package is 2.2.0.

Next, add the cross-origin resource sharing services to the pipeline. To do this, invoke the AddCors method on the IServiceCollection instance in the ConfigureServices method of the Startup class as shown in the code snippet below.

```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors();
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version_2_2);
        }
```

Configure CORS policy in ASP.Net Core

You can configure CORS policy in various ways in ASP.Net Core. As an example, the following code snippet allows only a specific origin to be accessed.

```
services.AddCors(options =>
            {
                options.AddPolicy("AllowSpecificOrigin",
                    builder => builder.WithOrigins("http://localhost:60571"));
            });
```

Apart from the WithOrigins method, ASP.Net gives us a number of other methods related to other policy options. These include the following:

- AllowAnyOrigin — used to allow access to the resource from any origin
- AllowAnyHeader— used to allow all HTTP headers in the request
- AllowAnyMethod— used to allow any HTTP methods to be accessed
- AllowCredentials — used to pass credentials with the cross-origin request
- WithMethods — used to allow access to specific HTTP methods only
- WithHeaders — used to allow access to specific headers only

If you want to allow more than one origin to access a resource, you can specify the following in the ConfigureServices method.

```
services.AddCors(options =>
            {
                options.AddPolicy("AllowSpecificOrigin",
                    builder => builder.WithOrigins("http://localhost:60571", "http://loc
            });
```

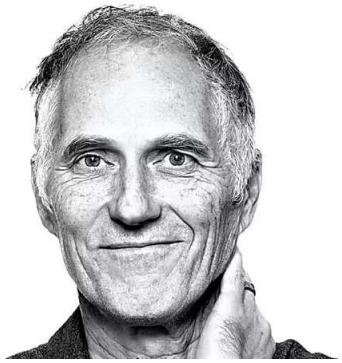
If you want to allow any origin to access a resource, you should use the AllowAnyOrigin method instead of the WithOrigins method. The code snippet given below illustrates how you can allow CORS requests from all origins with any scheme.

```
services.AddCors(options =>
            {
                options.AddPolicy("AllowAllOrigins",
                    builder => builder.AllowAnyOrigin());
            });
```

CORS is a useful mechanism that allows us to flexibly bypass the restrictions of the same-origin policy of web browsers. When we want to allow cross-origin access to our server applications, we can use CORS middleware in ASP.Net Core to do so while taking advantage of a variety of cross-origin access policies.

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

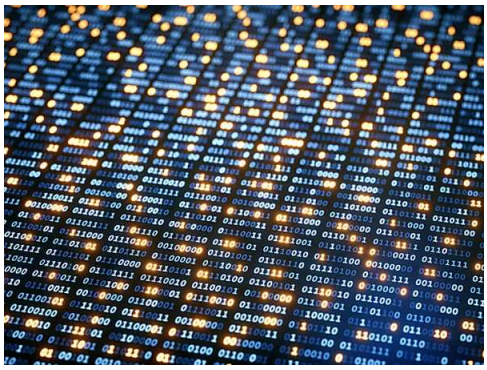
Follow



Tim O'Reilly: the golden age of the programmer is over

Deno Company forms to back Node.js rival

Oracle Database 21c review: The old RDBMS is new again



What is a computational storage drive? Much-needed help for CPUs

Deno 1.9 previews native HTTP/2 server

LLVM 12 arrives with x86, AArch optimizations

3 enterprise AI success stories

Edge computing archetypes are emerging,

Red Hat Enterprise Linux takes aim at edge

When to use Task.WaitAll vs. Task.WhenAll in .NET

The 24 highest paying developer roles in 2020