

CONFIGURING CORS IN ASP.NET WEB API RISING REQUIREMENT

Tech Stack

C#, ASP.NET

Notes

Cross-Origin Resource Sharing (**CORS**) is a security feature implemented by web browsers. It controls how web applications hosted on one domain can interact with resources hosted on a completely different domain.

When building applications that have both a frontend and a backend, the separate applications often live on different websites with their own addresses. When the frontend makes a request to the backend, the request that's made is called an HTTP request. This request always carries a header called "Origin" which tells where the request came from.

Understanding A CORS error

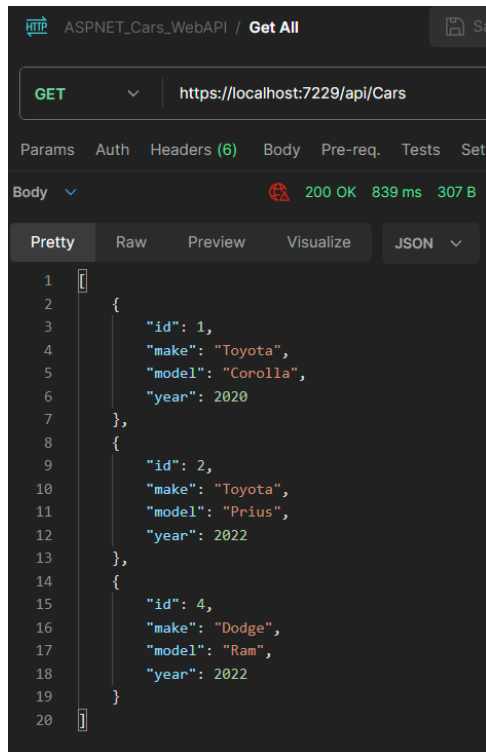
Seeing a CORS error for the first time can be confusing.

The easiest way to experience a **CORS** error is to use a frontend React application to make an axios request to a backend server which does not yet have **CORS** configured.

It is important to understand you will never encounter a CORS error in Postman!

CORS errors only occur when the request originates in a web browser like Chrome or Edge. Postman is not a web browser, so it does not enforce **CORS** restrictions.

Below, we see our **ASP.NET Web API** is running and returning a **200** response along with 3 car objects.



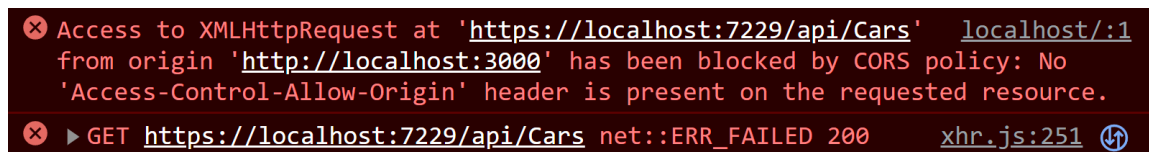
But when I try to make the same API call in my React application, I get an error and do not get back those three cars.

```
function App() {

  const fetchCars = async() =>{
    let response = await axios.get("https://localhost:7229/api/Cars");
    console.log(response.data);
  }

  useEffect(() => {
    fetchCars();
  }, []);

  return (
    <div>
      <h1>Cors Demo</h1>
    </div>
  );
}
```

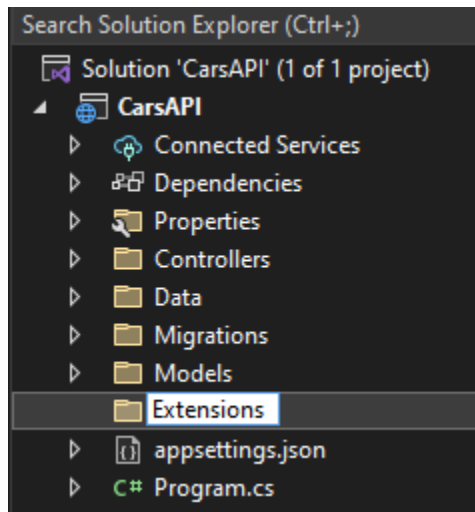


The important thing to remember is that even though you will not see this CORS error until you build out the frontend, no amount of coding or debugging in the

frontend will fix it! We need to make a change to the backend to allow this request to go through.

Configuring CORS in ASP.NET

In Visual Studio, stop your ASP.NET project, and start by creating an **Extensions** directory.

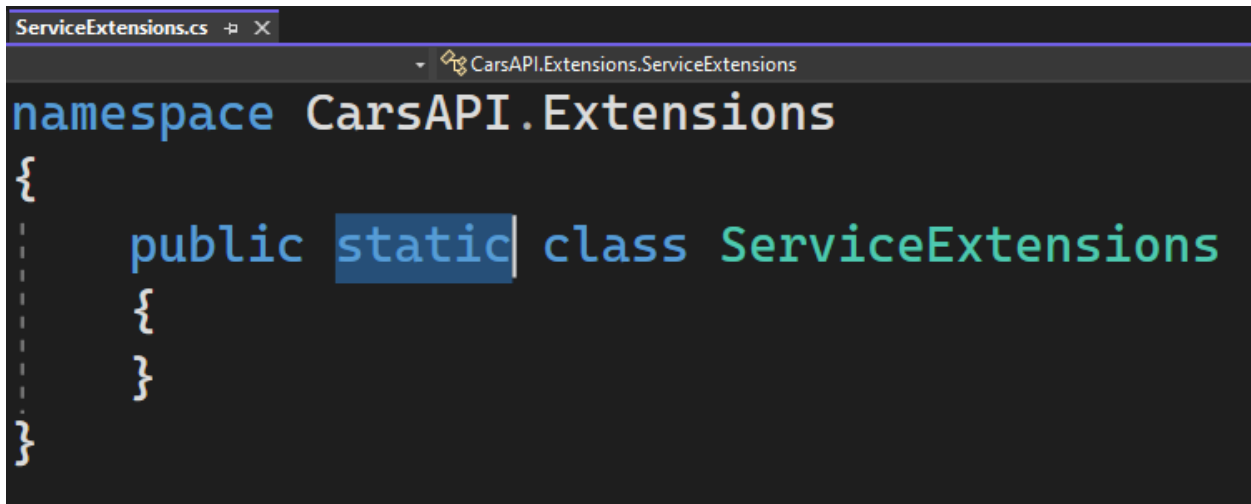


In an ASP.NET Core Web API project, an "**Extensions**" folder typically contains custom extension methods that provide additional functionality or convenience methods for various components of the application. These extensions are not part of the core framework but are created by developers to simplify and modularize their code. This can help with:

- **Modularity:** Extension methods allow you to keep your codebase clean and organized by separating custom functionality from core components.
- **Reusability:** By placing commonly used code as extension methods in this folder, you can easily reuse them across different parts of your application or in other projects.
- **Readability:** Extension methods can enhance the readability of your code by providing descriptive and intuitive method names that encapsulate complex operations.
- **Maintainability:** Changes or updates to specific functionality can be made more efficiently by modifying the extension method in one place rather than scattering the code throughout your application.
- **Encapsulation:** Extension methods allow you to encapsulate logic related to a particular concern, making it easier to manage and test.

Within this folder, create a new class named **ServiceExtensions**.

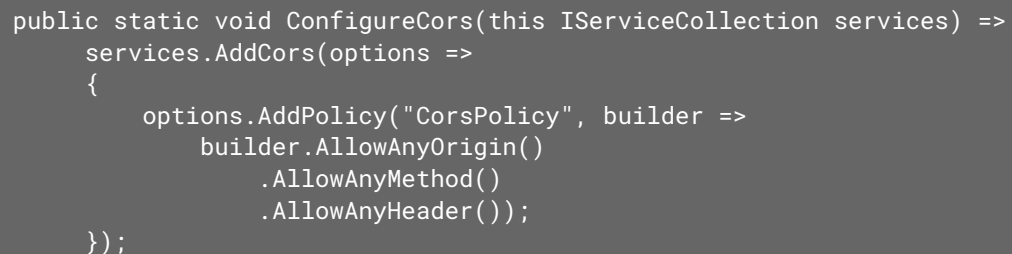
Make the class a **static class** by adding the word **static** to the signature.



```
ServiceExtensions.cs  X
CarsAPI.Extensions.ServiceExtensions

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

Within this class, paste the following method:



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

This method configures CORS settings to allow requests from **any** origin, with **any** HTTP method, and with **any** headers. It's a very permissive **CORS** configuration that may be useful during development but should be adjusted to restrict access more intentionally in a production environment for security reasons. The "=>" syntax is used for defining lambda expressions, which are concise and often used for inline method definitions or expressions in C#.

That's all the ServiceExtensions class needs!

However, we do also now ensure that the application itself knows to use this class.

In the **Program.cs** class, we need to add two lines of code.

First, we will add the service to the container by adding one line **below** the line which says **// Add services to the container**.

```
public class Program
{
    public static void Main(string[] args)
    {
        var builder = WebApplication.CreateBuilder(args);

        // Add services to the container.
        builder.Services.ConfigureCors();
        builder.Services.AddControllers();
    }
}
```

Then, activate the “**CorsPolicy**” we created in in the **ServicesExtensions** class by adding one more line below the line **var app = builder.Build();**

```
30
31 var app = builder.Build();
32
33 app.UseCors("CorsPolicy");
34
```

And that is it! Make sure to save all your files and then restart the the ASP.NET application.

Testing and Conclusion

Once the backend is running again, test the endpoint in **Postman** again. It should return results just as before.

Now finally, return to your **React** application. Restart it with `npm start` if you had stopped it, otherwise refresh the browser.

We now see the error is gone, and replaced by the successful console logging of the **response.data**!

```
▼ (3) [{...}, {...}, {...}] ⓘ App.js:9
  ▶ 0: {id: 1, make: 'Toyota', model: 'Corolla', year: 2020}
  ▶ 1: {id: 2, make: 'Toyota', model: 'Prius', year: 2022}
  ▶ 2: {id: 4, make: 'Dodge', model: 'Ram', year: 2022}
    length: 3
  ▶ [[Prototype]]: Array(0)
```

You have now configured **CORS** for serving content to a **React** application, and showcased the power of encapsulating custom functionality and middleware in your “**Extensions**” folder. Beyond **CORS**, this approach offers the flexibility to extend your application further by including other custom extensions, such as database connection management (e.g., **MySQL**), request/response formatting, authentication, or any other cross-cutting concerns. By organizing and centralizing these extensions, you not only improve code organization but also simplify development, testing, and future updates, making your **WebAPI** more robust and adaptable to evolving requirements.