Understanding CORS

In this lesson you will learn about Cross-Origin Resource Sharing, also known as CORS, what problem it solves and how it can help us enable the communication between our front-end portal and our microservices.

Cross-Origin Requests

To understand CORS let's first understand what a cross-origin request is. Let's put our Catalog microservice to the side for a moment and let's imagine that the Node.js web server that is hosting our front-end is also the server that hosts the REST APIs to interact with the Mongo database that owns the Catalog data.

In this scenario, the browser initially navigates to the web server url to load the front-end home page, and then, when the Catalog section is requested, the browser makes a GET request to the Catalog REST API to the same web server url, to which the server will reply with the json representation of the catalog data.

The address from which the browser calls the REST API is known as the Origin and is made of the combination of protocol, host and port, which in this case is http://localhost:3000. Notice how the origin in the browser matches exactly the origin of the server that provides the REST API.

Let's now go back to our actual scenario, where our Catalog microservice, hosted in ASP.NET Core's Kestrel webserver, is the one providing the REST API. Now, when the Catalog section is requested in the browser, the browser makes the GET request to the REST API at the origin of the Catalog microservice, which is https://localhost:5001.

Since both the protocol and port of the Catalog microservice are different than the one in the browser, this is known as a cross origin request. The microservice will return the Catalog data just as before, but by now the browser knows that this response comes from a different origin, since it had set the Origin header in the request. Therefore, it rejects the response data with a CORS Error.

This happens because browsers follow what is known as the same-origin policy, which states that a web application can only request resources from the same origin the application was loaded from. Browsers enforce this to prevent malicious web sites from reading confidential information from other web sites.

Unfortunately, this is also preventing our front-end from reaching the Catalog microservice. So, how can we fix this?

Cross-Origin Resource Sharing (CORS)

Here's where CORS comes into place. CORS allows a server to indicate any other origins than its own from which a browser should permit loading of resources. Here's how it works.

Once again, the browser has loaded the front-end page. Now the Catalog section is requested and the GET request is sent to the Catalog microservice. And since Catalog is hosted in a different origin, the browser appends the Origin header.

The difference this time is that the Catalog microservice has been configured with the Access-Control-Allow-Origin header, which indicates all the other origins that are allowed to call the REST API. In this case, this header has been configured with the origin where the front-end is hosted, http://localhost:3000.

Once Catalog sends the items data back to the browser, it appends this header to the response. The browser now compares the value in the Origin header it sent to the value in the received Access-Control-Allow-Origin header. Since they match, the browser now allows the response data to reach the front-end.

This works for simple requests like GETs, but for POST, PUT and other methods things work a bit differently.

CORS Preflighted Requests

When a cross-origin request may perform some sort of write operation to the resources owned by the server, like is the case when performing a POST, PUT or DELETE request, the request needs to be first preflighted. This means that an additional initial request needs to be sent to the server to determine if the actual request is safe to send.

Going back to our scenario, image we now want to create a new item in the Catalog, and therefore our front-end will send a POST request to the REST API. However, since this is a POST of JSON data across origins, the browser automatically sends first a request with the OPTIONS method, this time not just adding the Origin header, but also two other headers called Access-Control-Request-Headers and Access-Control-Request-Method.

Access-Control-Request-Method indicates that when the actual request is sent, it will be a POST and Access-Control-Request-Headers tells the server that the request will come with the content-type header in this case.

The server should have been configured to respond with corresponding headers to indicate what it allows. The headers returned by the server are Access-Control-Allow-Origin, Access-Control-Allow-Headers and Access-Control-Allow-Methods.

If the values returned in these headers match the values in the headers sent in the request, the browser accepts the POST request and submits this original request to the server with the json payload of the new item to create. The Catalog microservice creates the item and returns the expected 201 status code.

As you can see, the key to a successful cross-origin request is in the appropriate CORS configuration on the microservice that exposes the REST API, since only the microservice can declare the allowed origins. But how do we properly configure CORS in ASP.NET Core?

That's the focus of the next lesson where we will add the missing CORS configuration for both the Catalog and Inventory microservices.