12.3.3 Handle CORS Errors

Earlier, Roza discovered that she couldn't simply read in an external JSON dataset into her JavaScript file. Instead, she was required to run a local server. She wants to understand more about the CORS error message she came across, and why she must run a server to bypass it.

When Roza first began to learn to use Plotly, the data used to create charts were contained in the JavaScript file. She was able to open `index.html` directly in her browser and see the graphs. Likewise, when she made a call to the SpaceX API, she was able to open `index.html` directly in her browser.

However, when she attempted to read data from an external JSON file, she encountered an error:

To bypass this CORS error message, we navigated to the directory where `index.html` is located and ran `python -m http.server` in the CLI.

CORS stands for Cross-Origin Resource Sharing. In short, browsers by default do not permit reading of resources from multiple sources. This restriction is in place because of security concerns.

To better understand the issue, we first need to define servers. A **server** is a program or device that performs actions such as processing and sharing data. Our discussion will be limited to servers in the sense of software programs.

A Flask app, for example, is a server program that processes and shares data. Likewise, when we place a call to the SpaceX API, there is a server behind the scenes that processes and shares the requested data. Another example is when a user logs in on a website, the server receives the user's information, compares it against information in its database, and approves or denies the login attempt.

This is called a **request-response model**. The user (also known as the **client**) sends a request to the webpage server. The server, in turn, sends the requested data in response.

Web browsers, for security reasons, heavily restrict reading from, and writing to, local files. If access to local files was allowed, remote sites would be allowed to read and manipulate your private data. Or simply opening a local file with the browser could trigger a malicious script that transmits your data across the internet. This is why we're unable to read a JSON file directly.

However, running a static server, or `python -m http.server` in our case, allows us to skirt this restriction. Python's HTTP server provides a web address for both the JSON and HTML files to avoid these security issues.

Now let's return to the browser error message: `URL scheme must be 'http' or 'https' for CORS request.` This means that the browser can request external data only through the HTTP/HTTPS protocols. In other words, the CORS policy, by default, does not allow data to come in through channels other than through HTTP or HTTPS. Furthermore, the origin of the data must be from a single source, unless specified by CORS.

Here is a concrete example of how CORS works. Suppose that you navigate to a news website, and you are served an ad from adspamnetwork.com. If you happen to be logged into PayPal, and if these browser restrictions weren't in place, the JavaScript code in the ad might make an API call to PayPal and make unauthorized transactions. For this reason, browsers restrict a server from one site (adspamnetwork.com in this case) from making a request to a server from a different site (paypal.com) unless it has been given explicit permission.

How, then, does a website such as ebay.com make API calls to PayPal? The browser generally makes a preflight request to the server, which

verifies whether the browser's origin is allowed to make a request to it. The preflight request also includes other details, such as the types of requests permitted to be made, and the types of files permitted to be transferred. Then a request is made. The code on PayPal's server contains a CORS header that explicitly permits ebay.com to make API requests.

NOTE

> For more information about CORS, see the **MDN documentation (https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS)** .