concurrency strategy

Node.js employs a single thread with an event-loop to handle these threading issues. Node can manage thousands of concurrent connections in this way without experiencing any of the usual drawbacks of threads. There is virtually minimal context switching and memory overhead per connection.

The event-loop became well-known for the first time thanks to Node.js. Concurrent events are handled by the event-loop using single thread techniques. It makes use of the JavaScript call-back mechanism and is at the core of the Node.js processing model.

How this is implemented in my system is using express.js, a lightweight web framework for building RESTful APIs with javascript, perfect for a startup project. Express.js is a middleware and routing framework that manages a webpage's various routing and operates in between the request and response cycle. Express.js uses a variety of middleware methods to manage the various requests made by the client, such as get, put, post, and delete requests, which these middleware functions can easily handle.

TESTING STRATEGY

Backend testing: To test our backend we must attempt to pass a request to the server to register a new username, or return a 200 response if the username exist, after which we will set an event handler to transform the GUI.

Frontend Testing: I will use the GUI to attempt to login to the user interface using just a username. I will then attempt to search for other users, and start a chat with them by sending a message. I will attempt to do this with multiple user names, and view all logged users in the database.

Alpha Testing: In the presentation, I will attempt a peer-to-peer connection using a deployed version of the web-application with someone from class, and then a group conversation, using different devices through a shared URL.

DEMO

In order to test and verify that our handshake is successful and data is being sent correctly from client to server and reverse, Vozoo will make use of the REST client, avaialable as an extension, to send a request to the server with custom fields and headers, and get a response. This response will let us know that we can now develop our front-end application as the client side is now ready to interact with the server.

We will achieve this by instantiating a request.rest file with the following request:

POST http://localhost:3001/login Content-Type: application/json

```
{
"username": "adam", "secret": "pass1234"
}
```

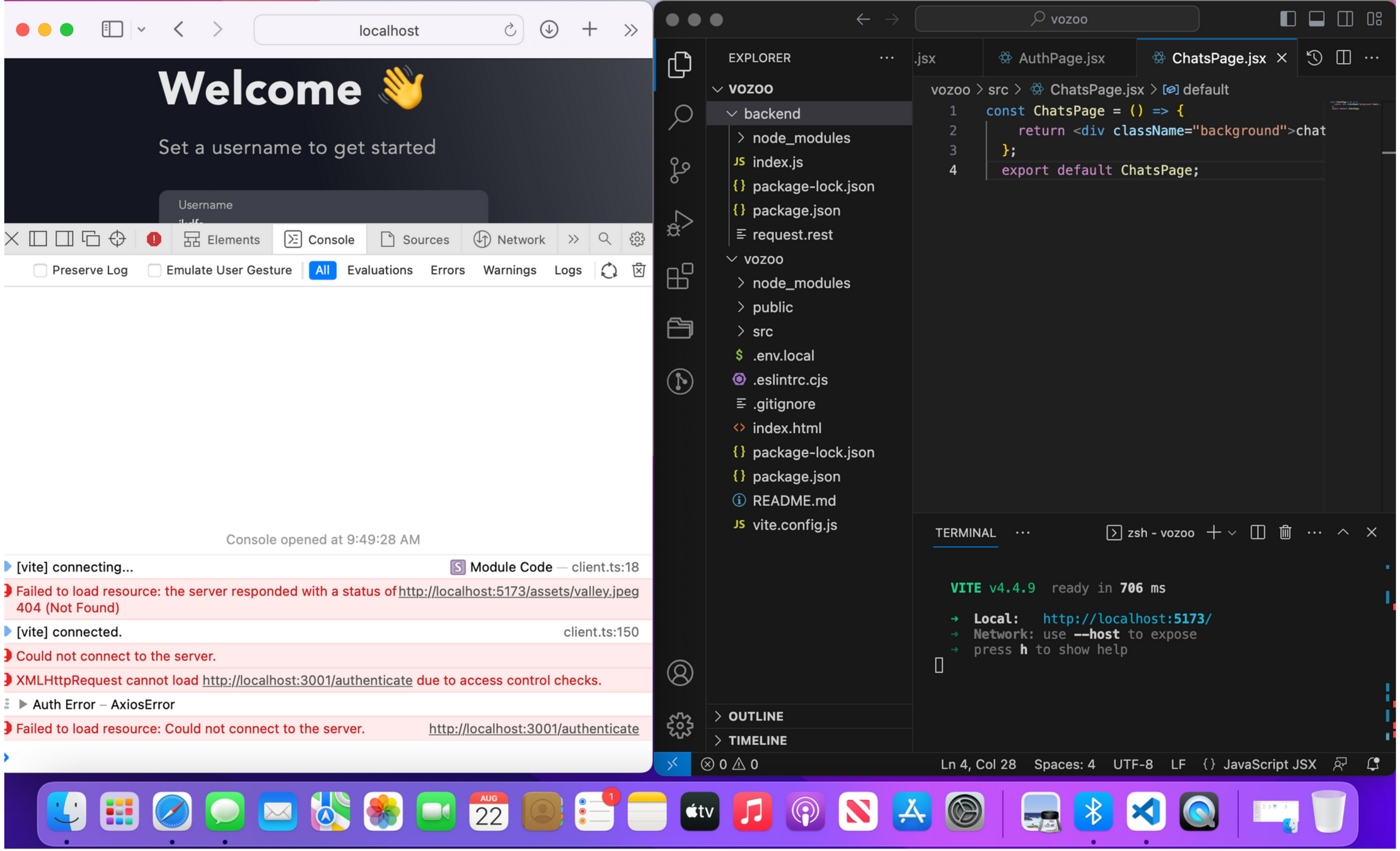

fig-1 Axios Error

errors

An error arose when attempting to pass a request using the REST client, as the network used to connect the server to the application is axis. Axios allows us to make asynchronous requests using JavaScript HTTP Headers. Attempting to debug the code, inspection of the response caught by the error handler stated that the post request had been refused.

This lead me to inspect the configuration of my headers using the cors module, where I then realised i had to alter it. CORS is a mechanism that allows a server to use a combination of HTTP headers to indicate from which domains, other than its own, it receives requests. I use this, axios and express to form my network architecture.

- I had to configure cors with the Access-Control-Allow-Origin - parameter in which origins are allowed to make requests to the server.

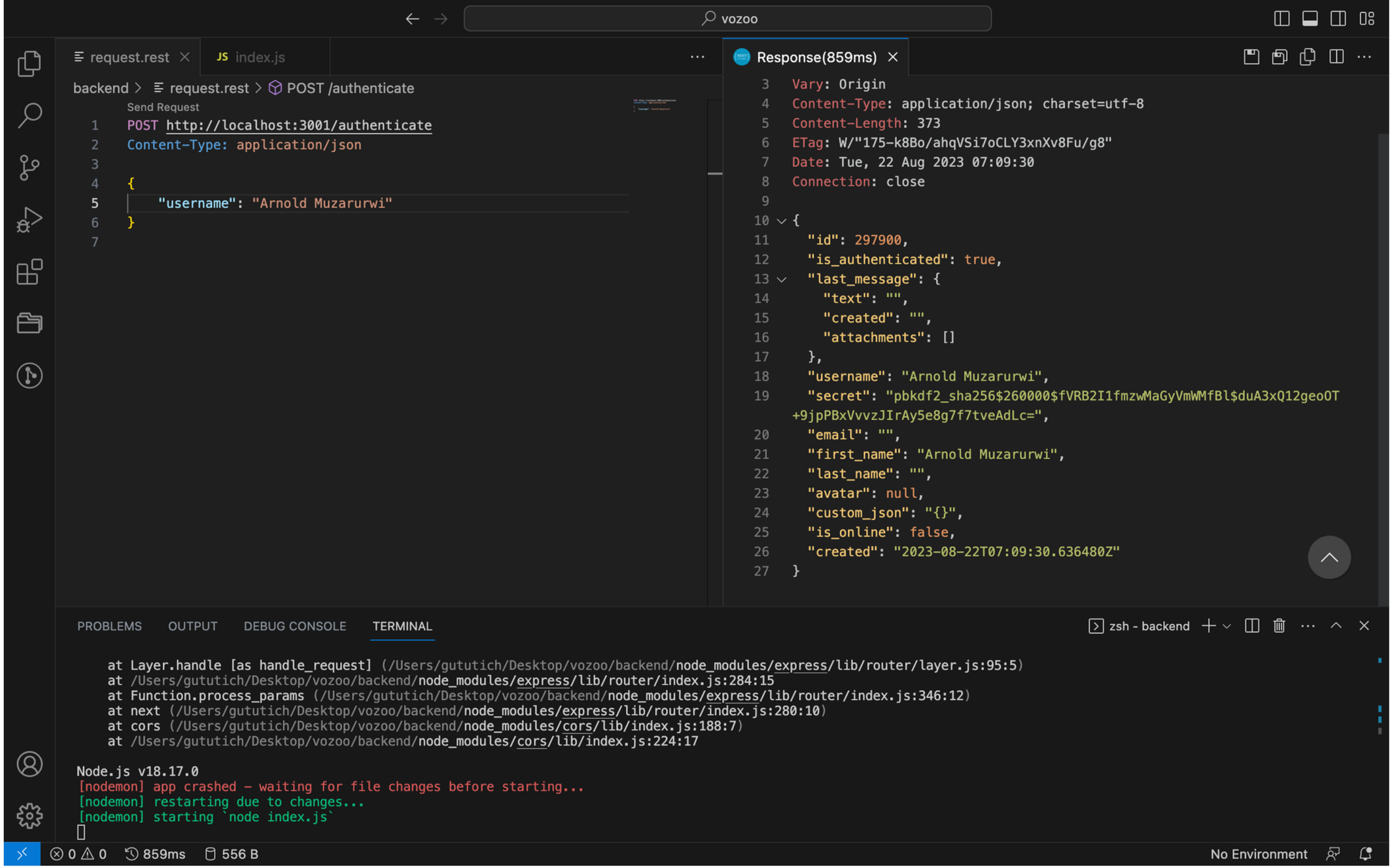

fig-2 REST Client response

correction

As seen in fig-2, the configuration of the cors module allowed us to specifcy the origin and destination of the connection. The information displayed on the right hand panel is the results returned from an HTTP request in JSON format.