DEPLOYMENT IN LOCAL AREA NETOWORK

*In my computer I have the \RetailApp folder shared in a Local Area Network*

*In my computer a MSSQL database server is running also at the \BackEnd*

*Is it possible to share the entire app with other users in the Local Area Network?*

*How can I do that?*

*When I RetailApp.html in VSCode and run Live Server,*

*it has this in the browser http://127.0.0.1:5500/RetailApp.html*

*but this doesn't work if accessed by others from their computers.*

*Folder Structure:*

*\RetailApp*

*\BackEnd*

*\Controllers*

*\DbConnect*

*\Routers*

*\node\_modules*

*.env*

*mainApp.js*

*package-lock.json*

*package.json*

*\css*

*\images*

*\src*

*RetailApp.html*

Yes, it is possible to share the app with other users on your Local Area Network (LAN). However, running the app using Live Server in Visual Studio Code (VSCode) with the http://127.0.0.1:5500/RetailApp.html URL only works on your own machine because 127.0.0.1 refers to the local loopback address (localhost) and is not accessible from other machines.

To make the app accessible from other devices on your network, you need to serve it in a way that can be accessed by others. Here's a step-by-step guide to achieve this:

**Step 1: Serve the Application on a Local Network IP**

Instead of using 127.0.0.1 (which is only accessible from your machine), you can serve the app on an IP address that is accessible to others in the same network. Here's how:

1. **Install Dependencies:** Make sure you have all the necessary dependencies installed in your RetailApp folder. Run:
2. npm install
3. **Modify Your Live Server Settings:** VSCode's Live Server allows you to specify the host address. By default, it uses 127.0.0.1 (localhost). You need to modify it to use the local network IP address of your computer.
   * Find your local IP address:
     + On Windows: Open Command Prompt and run ipconfig. Look for your IPv4 address under the network adapter you're connected to (e.g., 192.168.x.x).
     + On macOS/Linux: Run ifconfig or ip a in the terminal to find your local network IP address.
   * In your VSCode settings, modify Live Server to listen to your local network IP. Go to the **Settings** (Ctrl + ,), search for "Live Server" and look for Live Server > Settings: Host and change the value to your local IP address (e.g., 192.168.x.x).
4. **Start the Server:**
   * Open your project in VSCode, right-click on the RetailApp.html file, and choose **"Open with Live Server"**. The app will now be available at http://192.168.x.x:5500/RetailApp.html (replace 192.168.x.x with your local IP address).
   * Make sure the port 5500 is open or accessible, and ensure your firewall is not blocking the port.

**Step 2: Configure Backend API Access**

Your app seems to be connecting to a backend via \BackEnd. If this backend is running on a local MSSQL server, ensure that the backend is also accessible on the network.

1. **Backend IP Accessibility:** If the backend is running on your local machine and needs to be accessed by other devices, make sure your database server (MSSQL) allows connections from other devices on the network. You may need to:
   * Open the SQL Server port in your firewall (usually port 1433 for MSSQL).
   * Configure the MSSQL server to allow remote connections.
2. **Backend Configuration:**
   * Update the .env or any configuration files in your backend to allow connections from devices in your LAN. For example, if your backend is configured to connect to a database, ensure the connection string is not restricted to localhost, but rather to the machine's local IP.

**Step 3: Test on Other Machines**

Once you have configured both the frontend (Live Server) and backend (MSSQL), test the app from other devices on the same LAN.

* On another machine, open a browser and navigate to http://<your-local-ip>:5500/RetailApp.html. (http://192.168.35.59:5500/RetailApp.html)
* Ensure the backend API is also reachable from other devices by verifying the database and API requests.

**Alternative Option: Use a Node.js Server for More Flexibility**

If you're planning to run a more complex application with better control, you might want to set up an actual Node.js server (e.g., using Express) instead of relying on Live Server. This will give you more flexibility in routing, serving static files, and handling requests across devices. Here’s a simple example:

1. **Install Express:** In your project directory, run:
2. npm install express
3. **Create a Simple Express Server (server.js):** Create a file called server.js in your project root with the following content:
4. const express = require('express');
5. const path = require('path');
6. const app = express();
7. const port = 3000;
8. app.use(express.static(path.join(\_\_dirname, 'RetailApp')));
9. app.get('/', (req, res) => {
10. res.sendFile(path.join(\_\_dirname, 'RetailApp', 'RetailApp.html'));
11. });
12. app.listen(port, '0.0.0.0', () => {
13. console.log(`Server running at http://0.0.0.0:${port}`);
14. });
15. **Start the Server:** Run the server with:
16. node server.js

Now, your app will be served on your local IP address, like http://192.168.x.x:3000/RetailApp.html.

By following these steps, your app should be accessible by other users in your Local Area Network, with both frontend (HTML) and backend (MSSQL) parts working together.

CONNECTION ERROR

In order to handle the error in the frontend when the server is down and prevent the ERR\_CONNECTION\_REFUSED from causing an unhandled failure, you need to implement error handling in the frontend (client-side) while making API requests. You can capture the error using try-catch and send a user-friendly message like "Connection to the server is not established".

Here’s a step-by-step guide for handling the error:

**1. Backend Error Handling:**

Ensure your backend is correctly handling the connection error and sends a clear response to the client. For example, if your backend fails to connect to the database, it can respond with an HTTP error code and a message:

// Backend example

const express = require('express');

const app = express();

// Use your existing connect function

app.get('/data', async (req, res) => {

try {

const pool = await connect(); // Try to connect to the database

const result = await pool.request().query('SELECT \* FROM MyTable');

res.json(result.recordset);

} catch (err) {

console.error('Error connecting to database:', err);

res.status(500).json({ message: 'Unable to connect to the server. Please try again later.' });

}

});

const port = 3000;

app.listen(port, () => {

console.log(`Server is running on port ${port}`);

});

In the example above, if there's a database connection error or any failure in the backend, the server will respond with a 500 status code and a user-friendly message.

**2. Frontend Error Handling:**

In the frontend, you need to catch the error and display a custom message like "Connection to the server is not established" when the server is unavailable.

Here's an example using JavaScript fetch with try-catch block to handle errors and show a user-friendly message.

async function fetchData() {

try {

const response = await fetch('/data'); // Assuming the backend is on the same server

if (!response.ok) {

throw new Error('Server error');

}

const data = await response.json();

console.log('Fetched data:', data);

} catch (error) {

// If the server is down or connection fails

if (error.message === 'Failed to fetch') {

alert('Connection to the server is not established. Please check your connection and try again later.');

} else {

alert('An error occurred while fetching data. Please try again later.');

}

console.error('Error fetching data:', error);

}

}

// Call the function to fetch data

fetchData();

**Explanation:**

1. **Backend:** If the backend fails to connect to the database, it sends a 500 HTTP response with a clear message. You can customize the message to be more specific if needed.
2. **Frontend:** In the fetchData function:
   * It attempts to make a request to the server.
   * If the server is down, the fetch will throw an error with the message "Failed to fetch", which is caught in the catch block.
   * The frontend then displays a custom error message to the user, such as "Connection to the server is not established".

**Extra: Improving the Error Handling**

You could further improve this by checking the error type more carefully (e.g., network errors, server errors), or by showing a retry button for the user to attempt reconnecting. Additionally, using a library like axios provides more robust error handling and response structure.

This way, your frontend is prepared to handle situations where the server is down and give the user a meaningful message instead of a vague or technical error message.