CLEAVE Formatting

Great question! When using **Cleave.js**, you **do not** need to modify your numeric input's type to text yourself. Cleave.js will take care of that for you when you initialize the formatting on the input element, regardless of whether the input type is text, number, or even tel. It works by directly applying the format to the value as the user types.

**So, do you still need to modify type="number" to type="text" when using Cleave.js?**

No, you **don't** need to change the type to text manually. Cleave.js works perfectly fine with type="number". However, in some cases (if you want more flexibility with the formatting), people prefer to set the input type to text because it gives you more control over the input behavior, but it’s not strictly necessary.

Here’s how Cleave.js works with type="number" or type="text":

* **If you use type="number"**: Cleave.js will still apply the formatting as you type, but depending on the browser, it may still show some input restrictions (like not allowing commas or a formatted number).
* **If you use type="text"**: This lets Cleave.js work more freely with the input, ensuring that you have full control over the formatting and won't encounter any browser-imposed numeric input limitations.

In most cases, people go with type="text" for numeric inputs when using Cleave.js, but technically **both** will work.

**Cleave.js Parameters/Options**

Cleave.js has a lot of configuration options that let you tailor the formatting to your needs. Here are the most common ones:

**1. numeral**

* Enables formatting for **numbers** (i.e., adding commas, decimal points).
* **Example**: Formatting numbers with commas and decimals.

new Cleave('#SalesRec\_ItemPrce', {

numeral: true

});

**2. numeralThousandsGroupStyle**

* Determines how the thousands are grouped (e.g., thousand, lakh, or wan for different locales).
* **Example**: Using commas to group thousands.

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

numeralThousandsGroupStyle: 'thousand' // "thousand" for commas

});

**3. prefix**

* Adds a prefix (e.g., $, €, or ₹) to the input field.
* **Example**: Prefixing the input with $ for currency input.

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

prefix: '$'

});

**4. delimiter**

* Defines a custom character to separate groups (e.g., a space or period).
* **Example**: Using a period (.) as a thousand separator.

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

delimiter: '.'

});

**5. blocks**

* Specifies the number of characters for each block of the input. You can define multiple blocks for different types of inputs.
* **Example**: Splitting input into blocks (e.g., groups of 4 digits).

new Cleave('#SalesRec\_ItemPrce', {

blocks: [4, 4, 4],

delimiter: '-'

});

**6. rawValueTrimPrefix**

* Removes the prefix from the raw value (this option is useful if you want to store the raw numeric value without the formatting).

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

prefix: '$',

rawValueTrimPrefix: true // Removes the '$' from the stored value

});

**7. integerOnly**

* Ensures only integer values are allowed (disables decimal points).
* **Example**: For quantity inputs or whole numbers.

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

integerOnly: true

});

**8. decimalMark**

* Allows you to define which character will be used as the decimal point (commas or periods).
* **Example**: Using a comma as the decimal separator (common in European locales).

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

decimalMark: ',' // Use ',' as the decimal separator

});

**9. prefix and suffix**

* Adds a prefix or suffix to the input value (e.g., for currency or percentage).
* **Example**: Adding a % symbol at the end of the value.

new Cleave('#SalesRec\_ItemPrce', {

numeral: true,

suffix: '%'

});

**10. uppercase**

* Forces the input value to be in uppercase.
* **Example**: For text inputs where you want to enforce uppercase (not typically used for numeric inputs).

new Cleave('#SalesRec\_ItemPrce', {

uppercase: true

});

**11. date**

* Specifically for date formatting. Cleave.js supports formatting dates in many styles (e.g., DD/MM/YYYY or MM/DD/YYYY).
* **Example**: Formatting a date.

new Cleave('#dateInput', {

date: true,

datePattern: ['d', 'm', 'Y']

});

**12. time**

* Specifically for time formatting. Cleave.js supports time input with options like hours, minutes, and seconds.
* **Example**: Formatting time as HH:MM.

new Cleave('#timeInput', {

time: true,

timePattern: ['h', 'm']

});

**Example Using Cleave.js with Multiple Options**

Here’s a full example of how to use **Cleave.js** to format a price input with a currency symbol, commas for thousands, and two decimal places:

<label for="SalesRec\_ItemPrce">Unit Item Price</label>

<input type="text" id="SalesRec\_ItemPrce" name="ItemPrce">

<script src="https://cdn.jsdelivr.net/npm/cleave.js@1.6.0/dist/cleave.min.js"></script>

<script>

new Cleave('#SalesRec\_ItemPrce', {

numeral: true, // Enable numeral formatting

numeralThousandsGroupStyle: 'thousand', // Group thousands with commas

prefix: '$', // Prefix the value with $

decimalMark: '.', // Use a period as the decimal mark

suffix: ' USD', // Optional: Append ' USD' as the suffix

rawValueTrimPrefix: true // Removes the '$' when accessing the raw value

});

</script>

**Conclusion**

* **Cleave.js** is very powerful for handling numeric inputs with formatting (commas, decimals, currency, etc.), and you don’t need to change the type to text unless you need complete freedom to control how the value is entered.
* The library handles the formatting for you, and you can customize it further with various configuration options.
* If you need to go beyond basic formatting and work with multiple input types (like dates or times), Cleave.js has great flexibility and can help you manage those inputs easily.

Let me know if you'd like to dive deeper into any specific features or options!

try {

const response = await fetch('http://localhost:3000/sales/editSalesDetail', {

method: 'PUT',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

cRecordId: cRecordId,

cItemCode: cItemCode,

nQuantity: nQuantity,

nItemPrce: nItemPrce,

nDiscRate: nDiscRate,

nAmount\_\_: nAmount\_\_,

nLandCost: nLandCost

})

});

// Check if the response is okay (status code in the range 200-299)

if (!response.ok) {

// Handle different status codes

if (response.status === 404) {

throw new Error("Resource not found (404).");

} else if (response.status === 500) {

throw new Error("Server error (500). Please try again later.");

} else if (response.status === 400) {

throw new Error("Bad request (400). Please check your input.");

} else if (response.status === 401) {

throw new Error("Unauthorized (401). Please log in.");

} else if (response.status === 403) {

throw new Error("Forbidden (403). You don't have permission to perform this action.");

} else {

// Handle other errors generically

throw new Error(`HTTP error! Status: ${response.status}`);

}

}

const updatedItem = await response.json();

if (updatedItem) {

itemsDtl[index] = updatedItem;

updateItemTable(true);

showNotification('Sales Item record updated successfully!');

updateSalesTotals(cCtrlNum\_);

}

} catch (error) {

console.error("Error processing editSalesDetail:", error);

displayErrorMsg(error, 'Error processing editSalesDetail');

} finally {

document.getElementById('loadingIndicator').style.display = 'none';

}

In your fetch request, you're already using the response.ok to check for errors, but let's break down how you can handle different HTTP status codes, as well as give an overview of common status codes (other than 404 and 500) you might encounter.

**1. Handling Various Status Codes in fetch:**

fetch doesn’t automatically reject the promise on HTTP errors (like 404, 500, etc.). Instead, it resolves the promise with the response, and it’s up to you to inspect the response and handle it.

Here’s how you can capture the status code and handle different scenarios based on the status code:

try {

const response = await fetch('http://localhost:3000/sales/editSalesDetail', {

method: 'PUT',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({

cRecordId: cRecordId,

cItemCode: cItemCode,

nQuantity: nQuantity,

nItemPrce: nItemPrce,

nDiscRate: nDiscRate,

nAmount\_\_: nAmount\_\_,

nLandCost: nLandCost

})

});

// Check if the response is okay (status code in the range 200-299)

if (!response.ok) {

// Handle different status codes

if (response.status === 404) {

throw new Error("Resource not found (404).");

} else if (response.status === 500) {

throw new Error("Server error (500). Please try again later.");

} else if (response.status === 400) {

throw new Error("Bad request (400). Please check your input.");

} else if (response.status === 401) {

throw new Error("Unauthorized (401). Please log in.");

} else if (response.status === 403) {

throw new Error("Forbidden (403). You don't have permission to perform this action.");

} else {

// Handle other errors generically

throw new Error(`HTTP error! Status: ${response.status}`);

}

}

const updatedItem = await response.json();

if (updatedItem) {

itemsDtl[index] = updatedItem;

updateItemTable(true);

showNotification('Sales Item record updated successfully!');

updateSalesTotals(cCtrlNum\_);

}

} catch (error) {

console.error("Error processing editSalesDetail:", error);

displayErrorMsg(error, 'Error processing editSalesDetail');

} finally {

document.getElementById('loadingIndicator').style.display = 'none';

}

**2. Explanation of Common Status Codes:**

Here’s a quick guide to some common HTTP status codes and how you might want to handle them:

* **200 OK**: The request was successful, and the server returned the expected data.
* **201 Created**: The request was successful and a new resource was created (commonly used for POST requests).
* **400 Bad Request**: The server couldn't understand the request due to invalid syntax or missing data.
* **401 Unauthorized**: The client must authenticate to get the requested response (e.g., token or login required).
* **403 Forbidden**: The server understands the request, but the client doesn't have permission to access the resource.
* **404 Not Found**: The server can't find the requested resource. Often this means a URL or route doesn’t exist.
* **500 Internal Server Error**: A generic error when the server encounters an unexpected condition that prevents it from fulfilling the request.
* **502 Bad Gateway**: The server received an invalid response from the upstream server.
* **503 Service Unavailable**: The server is not ready to handle the request. Commonly, this happens when the server is down for maintenance.
* **504 Gateway Timeout**: The server didn’t get a response from an upstream server in time.

**3. Improvement on Error Handling:**

In the code above:

* **Specific handling of different status codes**: Instead of a generic error message, you can customize the message based on the specific HTTP error status.
* **Graceful user feedback**: For example, you could display a user-friendly message like "We are sorry, but the resource was not found" for a 404 error, or "Our servers are currently down, please try again later" for a 500 error.

**4. Display the Error Message to the User:**

In your catch block, you're already logging the error and calling displayErrorMsg. You could adjust this function to show the user-friendly message based on the error status code.

function displayErrorMsg(error, context) {

const errorMessage = error.message || 'An unexpected error occurred.';

// Example of showing the error message in a user-friendly way

const errorElement = document.createElement('div');

errorElement.classList.add('error-message');

errorElement.textContent = `${context}: ${errorMessage}`;

document.body.appendChild(errorElement); // Append the error message somewhere visible

}

This way, the user will see an appropriate error message based on the status code that was returned from the API.

**Summary:**

1. You **check response.ok** to see if the request was successful.
2. If the response is not successful, **throw a custom error** based on the status code.
3. **Handle various HTTP status codes** (like 404, 500, etc.) to give specific feedback to the user.
4. **Display friendly error messages** to the user based on the error context.

This pattern ensures you handle errors appropriately while providing a good user experience. Let me know if you need further clarification! 😊