# Self-Study

## 1. How to get custom attributes included in client-side validation ?

## Understanding the Issue

In ASP.NET MVC (or ASP.NET Core MVC), custom validation attributes (inheriting from ValidationAttribute) handle server-side validation automatically during model binding. However, for client-side validation with jQuery Unobtrusive Validation (the default in MVC views), the framework relies on specific HTML attributes like data-val="true", data-val-[rule]="Error message", and data-val-[rule]-params to generate JavaScript rules. Custom attributes don't automatically emit these unless you explicitly support client-side validation.

This means your custom attribute works on postback but skips client-side checks, leading to a poor UX where forms submit invalid data to the server.

## Possible Solutions

### 1. Implement IClientValidatable on Your Custom Attribute (Recommended)

This is the cleanest way: Extend your attribute to provide client-side rules directly. The MVC framework will automatically render the necessary data-val-* attributes when generating HTML helpers like @Html.EditorFor.
Step-by-Step:

Make your attribute implement IClientValidatable.
Override GetClientValidationRules to return a ClientValidationRule object, specifying a JavaScript validator function name and any parameters.

**Example Custom Attribute (Server-Side + Client-Side):**

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;

public class UniqueNameAttribute : ValidationAttribute, IClientValidatable
{
    // Server-side logic (as before, e.g., DB check excluding current ID)
    protected override ValidationResult IsValid(object value,
ValidationContext validationContext)
    {
        // Your existing server-side logic here...
        // e.g., query DB for uniqueness
        if (/* duplicate found */) return new ValidationResult("Name must
be unique.");
        return ValidationResult.Success;
    }

    // Client-side support
    public IEnumerable<ModelClientValidationRule>
GetClientValidationRules(ModelMetadata metadata, ControllerContext
context)
    {
        var rule = new ModelClientValidationRule
        {
            ErrorMessage = FormatErrorMessage(metadata.GetDisplayName()),
// "Name must be unique."
            ValidationType = "uniquename" // JS function name (e.g.,
jquery.validate.unobtrusive.adapters.add for this)
        };
        // Optional: Add parameters if needed (e.g., for AJAX endpoint)
        // rule.ValidationParameters["ajaxurl"] = "/api/checkname";

        yield return rule;
    }
}
```

Client-Side JavaScript (Add to Your View or Layout):

You need to register the validator with jQuery Unobtrusive Validation. Place this in a <script> tag or external JS file loaded after jQuery and jquery.validate.unobtrusive.js.

```
javascript(function ($) {
    $.validator.unobtrusive.adapters.add("uniquename", function (options)
{
        options.rules["uniquename"] = true; // Or pass params if using
AJAX
        options.messages["uniquename"] = options.message;
    });

    // Add the actual validation method
    $.validator.addMethod("uniquename", function (value, element) {
        // Simple client-side check (e.g., basic regex or length)
        // For DB uniqueness, use AJAX to call a controller action
        if (value.length < 3) return false; // Placeholder

        // Example AJAX for real uniqueness (async, so use $.Deferred for
validation)
        var def = $.Deferred();
        $.get("/api/validate/name?value=" + encodeURIComponent(value) +
"&id=" + $("#Id").val())
            .done(function (data) {
                def.resolve(data.isUnique); // Assume API returns {
isUnique: true/false }
            })
            .fail(function () {
                def.reject(); // On error, assume invalid
            });
        return def.promise(); // jQuery Validate supports promises in
v1.9+
    });
})(jQuery);
```

Controller Action for AJAX (Optional but Recommended for DB Checks):

```
csharp[HttpGet]
public IActionResult ValidateName(string value, int? id = null)
```

```
{
    // Same logic as server-side, but return JsonResult
    var isUnique = /* DB query excluding id */;
    return Json(new { isUnique });
}
```

In Your View:

No changes needed—@Html.EditorFor(m => m.Name) will now include <input data-val="true" data-val-uniquename="Name must be unique." ... />.
**Pros:** Automatic HTML generation, consistent with built-in attributes.
**Cons:** Requires JS for complex rules (e.g., DB calls aren't truly client-side secure).

## 2. Custom HTML Helper or Tag Helper for Manual Attribute Injection

If IClientValidatable doesn't fit (e.g., very custom rendering), create a helper that adds the data-val-* attributes manually.

**Example (ASP.NET Core Tag Helper):**

```
// In a .cs file or TagHelpers folder
[HtmlTargetElement("input", Attributes = "asp-for")]
public class UniqueNameTagHelper :
Microsoft.AspNetCore.Razor.TagHelpers.TagHelper
{
    public override void Process(TagHelperContext context,
TagHelperOutput output)
    {
        // Base processing...
        if (/* Check if property has [UniqueName] */)
        {
            output.Attributes.Add("data-val", "true");
            output.Attributes.Add("data-val-uniquename", "Name must
```

```
be unique.");
            // Add JS validator as in Solution 1
        }
    }
}
```

Register in _ViewImports.cshtml: @addTagHelper *, YourAssembly.

**Pros:** Fine-grained control.
**Cons:** More boilerplate; doesn't scale well for multiple attributes.

## 3. Pure Client-Side JS Without Attribute Integration

Skip attribute-to-HTML bridging and handle validation entirely in JS (e.g., via form events or libraries like Knockout/Angular).

**Example:**

```
javascript$("#yourForm").on("focusout", "#Name", function () {
    // AJAX call as in Solution 1
    if (!isUnique) {
        // Show error via jQuery Validate's showErrors
    }
});
```

**Pros:** Quick for prototypes.
**Cons:** No integration with MVC's validation pipeline; duplicates effort.

## 4. Disable Client-Side for Custom Attributes (Fallback)

If client-side isn't critical (e.g., for security-sensitive checks like DB uniqueness), rely solely on server-side. Use JS to show a "checking..." spinner on submit.

In View:

```
html@Html.EditorFor(m => m.Name)
<span class="field-validation-valid" data-valmsg-for="Name"
data-valmsg-replace="true"></span>
```

**Pros:** Simple.
**Cons:** Poor UX—no instant feedback.

## Best Practices

- **Prioritize IClientValidatable:** It's the standard for data annotations. Always pair server-side and client-side logic to ensure consistency (e.g., same error messages).
- **Keep Client-Side Light:** For DB-dependent validations (like uniqueness), use AJAX but remember it's not secure—server-side is the gatekeeper. Avoid sending sensitive data in JS.
- **Test Thoroughly:** Validate in both create/edit scenarios (as in your previous query). Use browser dev tools to inspect generated HTML for data-val-* attributes.
- **Bundle JS Properly:** Include jquery.validate and jquery.validate.unobtrusive in your layout. For .NET Core, use LibMan or npm for management.
- **Accessibility & UX:** Add ARIA attributes (e.g., aria-invalid) and clear error styling. Provide fallback messages for JS-disabled users.
- **Performance:** Cache AJAX endpoints if possible; debounce input events to avoid excessive calls.
- **When to Skip Client-Side:** For high-security or infrequent validations, server-only is fine. Document this in your code.
- **Migration Tip:** If upgrading from older MVC, ensure System.ComponentModel.DataAnnotations and Microsoft.AspNetCore.Mvc packages are up-to-date.

This setup should get your custom attribute working seamlessly on both sides.

## 2. Three cases of using custom middleware.

Three common cases where custom middleware is used in ASP.NET Core are:

- **Request Logging and Diagnostics:** Custom middleware can log detailed information about incoming requests and outgoing responses, such as URLs, headers, and processing time. This helps in monitoring, debugging, and auditing request flow throughout the application.
- **Authentication and Authorization:** Middleware can be implemented to check the security credentials of incoming requests, validate tokens or user roles, and either allow the request to proceed or return unauthorized responses early in the pipeline.
- **Exception Handling and Error Responses:** Custom middleware can catch unhandled exceptions occurring in the downstream pipeline, log error details, and generate consistent and user-friendly error responses, improving the robustness and reliability of the application.

These cases use middleware to centralize cross-cutting concerns that apply to all or many requests, providing reusable and maintainable solutions in the request pipeline.

## 3. What are the new features in .net 9 to make operations on Generic type ?

New features in .NET 9 related to operations on generic types include:

- **Allowing ref struct types as generic type parameters:** .NET 9 enables developers to use *ref struct* types (which are stack-only types) as arguments for generic type parameters. This expands the capabilities of generics with more efficient, non-heap-allocated types suitable for high-performance scenarios.
- **New LINQ methods that enhance generic operations:** Methods like *CountBy* and *AggregateBy* allow more efficient grouping and aggregation operations by key without the overhead of creating intermediate collections. These methods can significantly improve performance in generically-typed collections.
- **Introduction of new generic collection types:** For example, *OrderedDictionary<TKey, TValue>* is a new generic type in .NET 9 that provides a more type-safe and flexible alternative to the older non-generic OrderedDictionary.
- **ReadOnlySet<T>:** A new generic read-only collection introduced to provide an immutable equivalent to *ISet<T>*, similar to how *ReadOnlyCollection<T>* and *ReadOnlyDictionary<TKey, TValue>* serve *IList<T>* and *IDictionary<TKey, TValue>* respectively.

These improvements aim to make generic programming more versatile, performant, and type-safe across different scenarios in .NET 9.