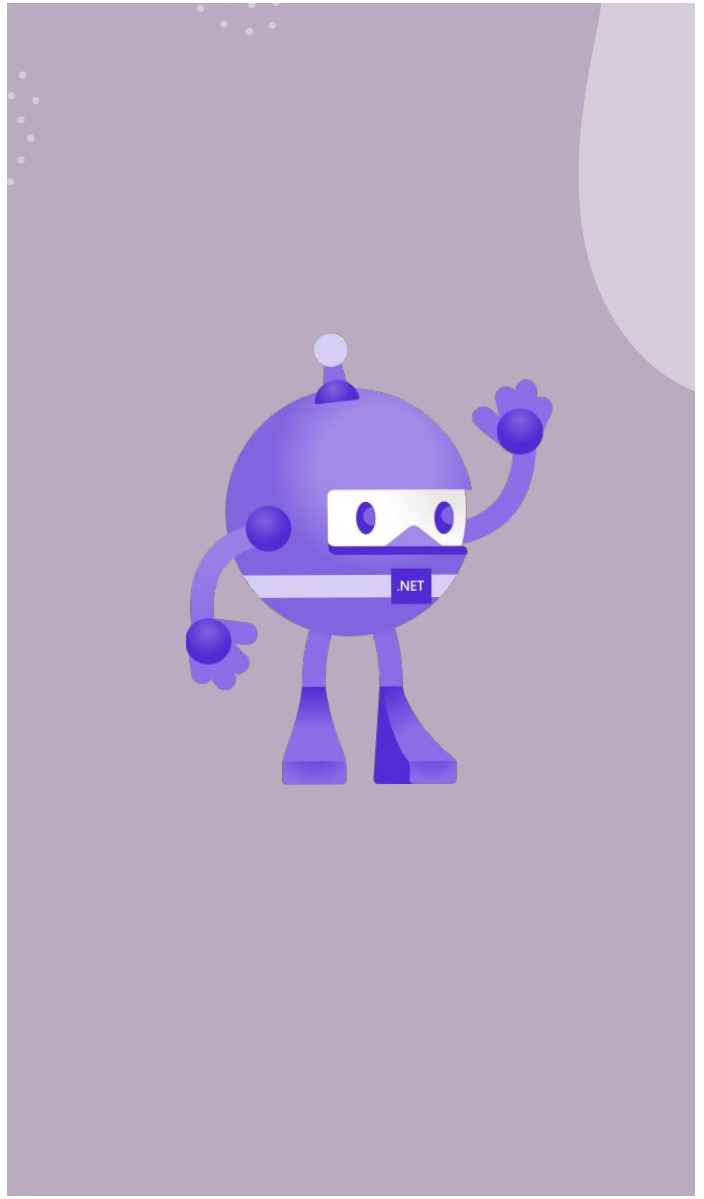
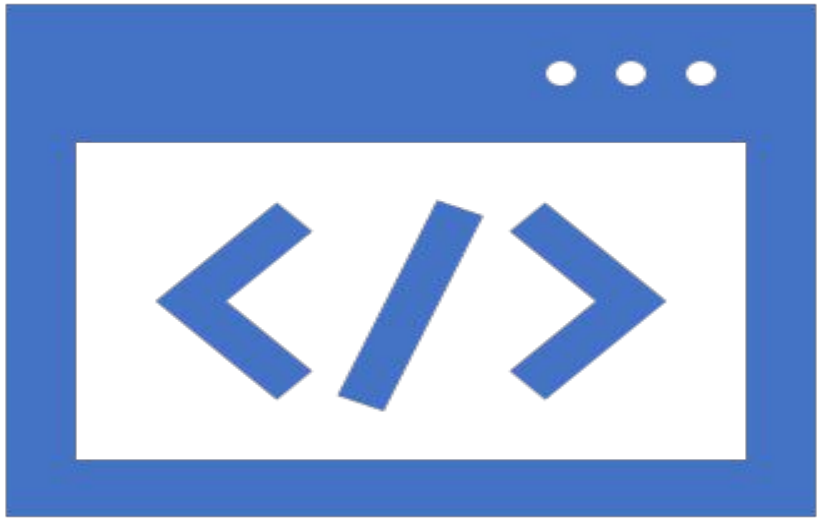
Asp.Net Core – Razor Pages

ASP.NET CORE WEB DEVELOPER PROGRAM @ AITRICH ACADEMY



HTML Helpers

In Razor Pages, HTML Helpers are a set of utility methods that simplify the process of generating HTML markup. They provide a convenient and strongly-typed way to generate HTML elements, form controls, links, and more. HTML Helpers help improve code readability, maintainability, and reduce the likelihood of errors.

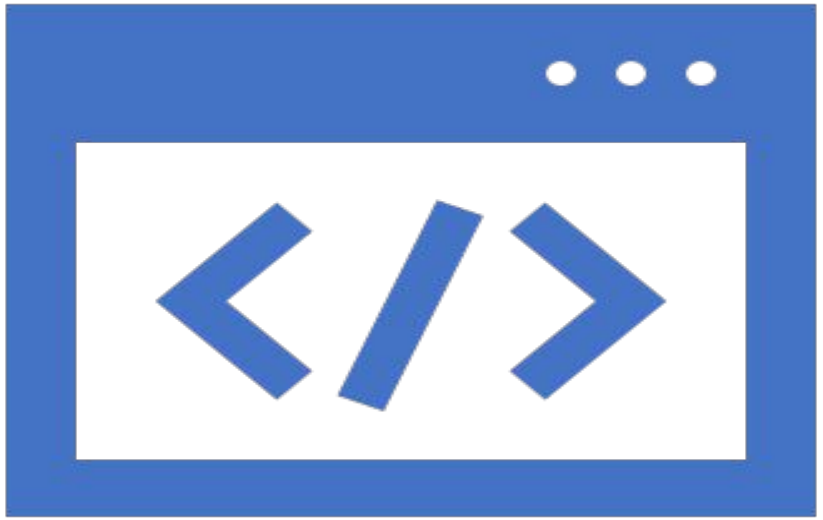


Here are some commonly used HTML Helpers in Razor Pages:

1 . Form Controls

Html.TextBoxFor and Html.PasswordFor:

These helpers generate <input> elements for text input and password input, respectively. They automatically bind to the model property specified



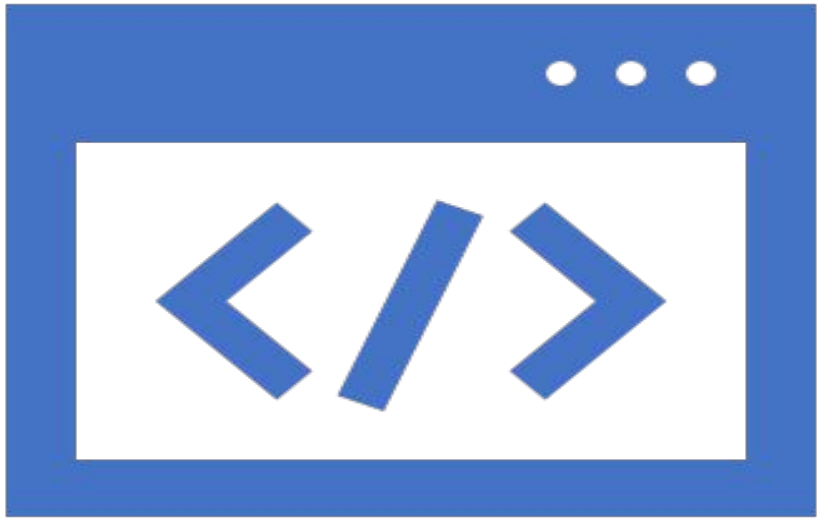
```
@Html.TextBoxFor(m => m.Name)
@Html.PasswordFor(m => m.Password)
```

Html.CheckBoxFor and Html.RadioButtonFor:

These helpers generate checkboxes and radio buttons that bind to a Boolean model property or an enumeration property, respectively.

```
@Html.CheckBoxFor(m => m.IsSubscribed)
@Html.RadioButtonFor(m => m.Gender, "Male")
@Html.RadioButtonFor(m => m.Gender, "Female")
```

```
@Html.DropDownListFor(m => m.CategoryId, Model.Categories, "Select a category")
```



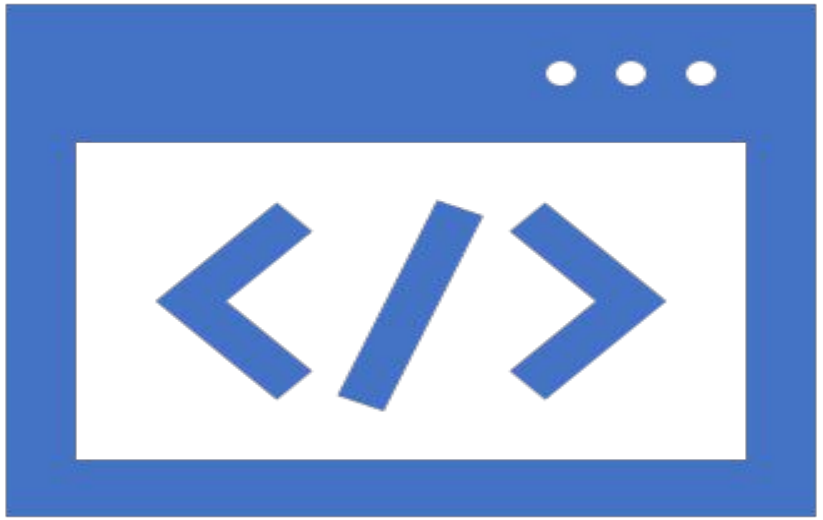
2 . Links and URLs:

Html.ActionLink:
This helper generates an <a> tag for a link to an action method within the same controller or a different controller.

```
@Html.ActionLink("Home", "Index", "Home")
```

Html.RouteLink:
This helper generates a link based on a named route and its parameters

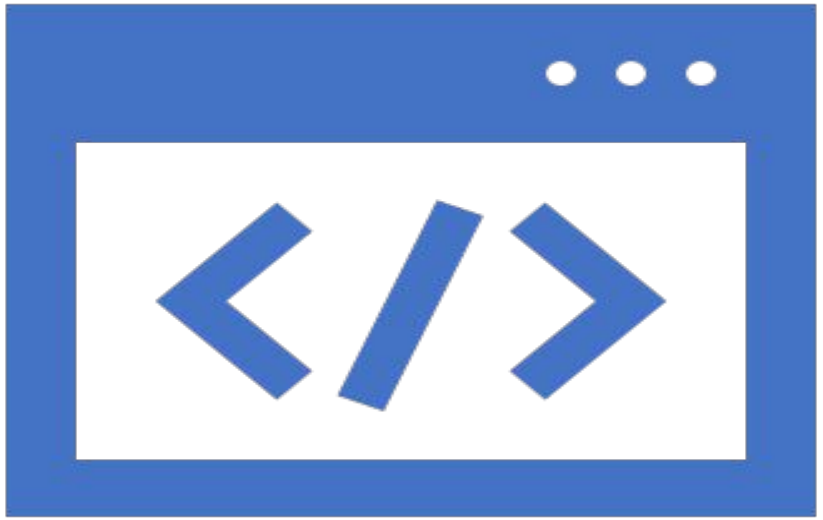
```
@Html.RouteLink("About Us", "AboutRoute")
```



3 . Partial Views

Html.Partial:
This helper renders a partial view within the current view. It can be used to reuse sections of markup.

```
@Html.Partial("_PartialViewName")
```



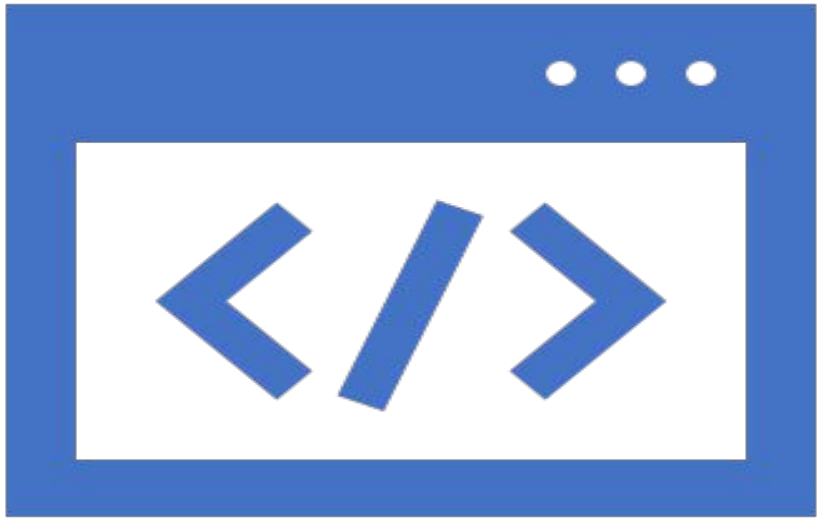
4 . Displaying Data

Html.DisplayFor:
This helper generates HTML markup to display the value of a model property, applying any specified display templates or formatting.

```
@Html.DisplayFor(m => m.Name)
```

Html.EditorFor:
This helper generates an appropriate editor control for a model property, using any specified editor templates.

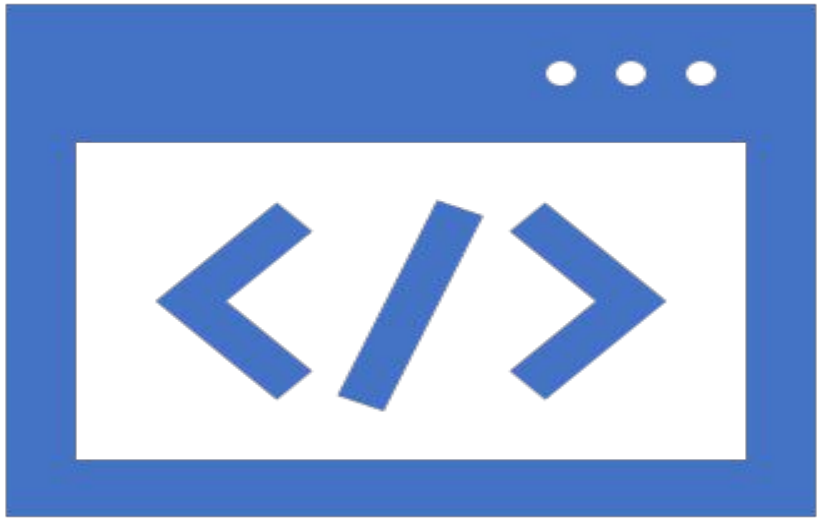
```
@Html.EditorFor(m => m.DateOfBirth)
```



Razor Page Events

In Razor Pages, events are a way to handle user interactions or specific actions that occur during the lifecycle of a page. Events allow you to respond to user input or perform certain operations in response to various triggers. Here are some commonly used events in Razor Pages:

1. Page Events
2. Handler Events

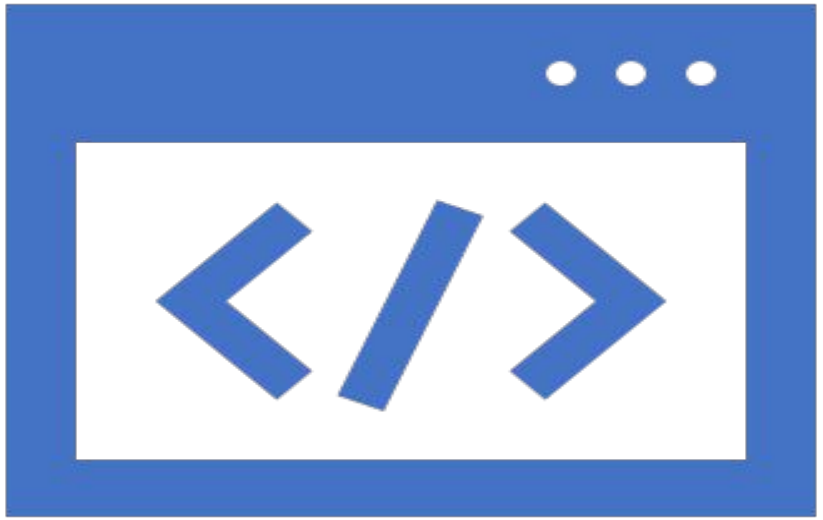


Page Events

OnGet and OnPost: These events are triggered when an HTTP GET or POST request is made to the page. You can override these methods in your Razor Page's code-behind file (PageModel) to handle the corresponding HTTP verb.

```
public IActionResult OnGet()
{
    // Logic for handling HTTP GET request
    return Page();
}

public IActionResult OnPost()
{
    // Logic for handling HTTP POST request
    return Page();
}
```

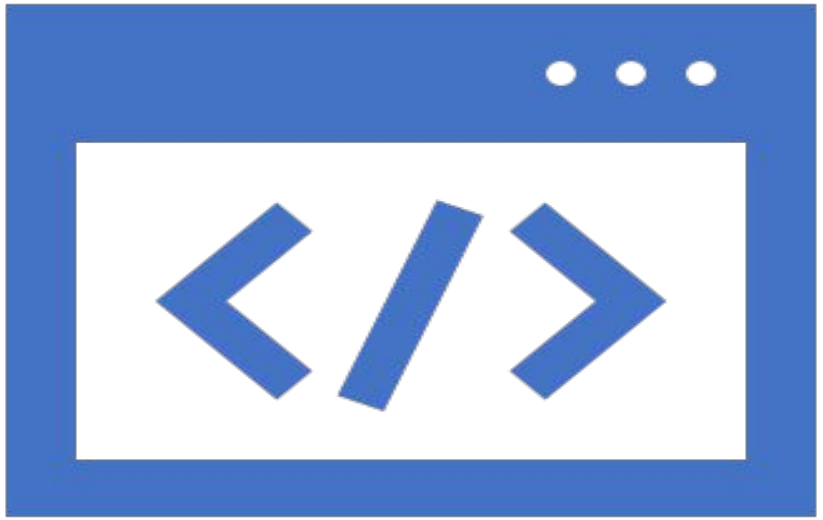


Page Events

OnGetAsync and OnPostAsync: Similar to OnGet and OnPost, these events are asynchronous versions and allow you to perform async operations during the request handling.

```
public async Task<IActionResult> OnGetAsync()
{
    // Asynchronous logic for handling HTTP GET request
    return Page();
}

public async Task<IActionResult> OnPostAsync()
{
    // Asynchronous logic for handling HTTP POST request
    return Page();
}
```

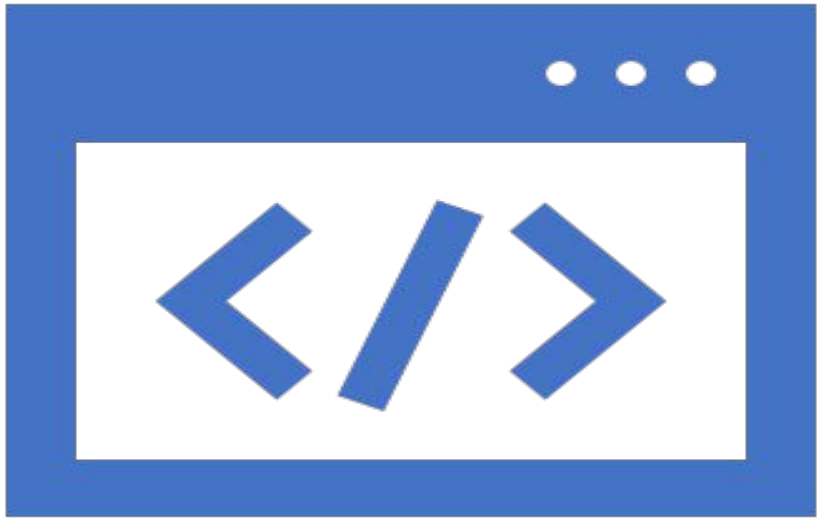


Razor Page Filters

Filters are a powerful feature that allow you to add cross-cutting concerns to your application. They enable you to execute code before or after specific stages in the Razor Pages lifecycle or before or after individual page handler methods.

There are two types of filters in Razor Pages:

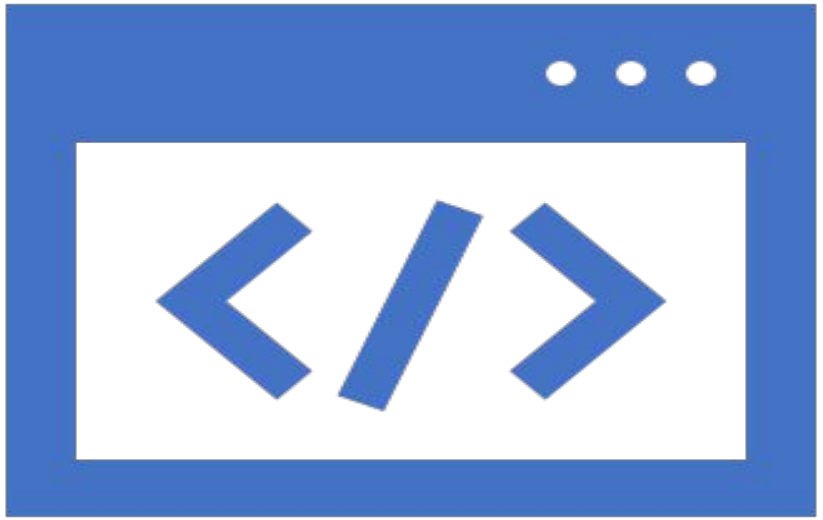
1. Page Filters
2. Handler Filters



Razor Page Filters

Page Filters: These filters are applied at the page level and can be used to perform actions before or after the entire page is executed. They are defined using the PageModel base class and can be used to handle scenarios like authentication, authorization, logging, caching, and so on.

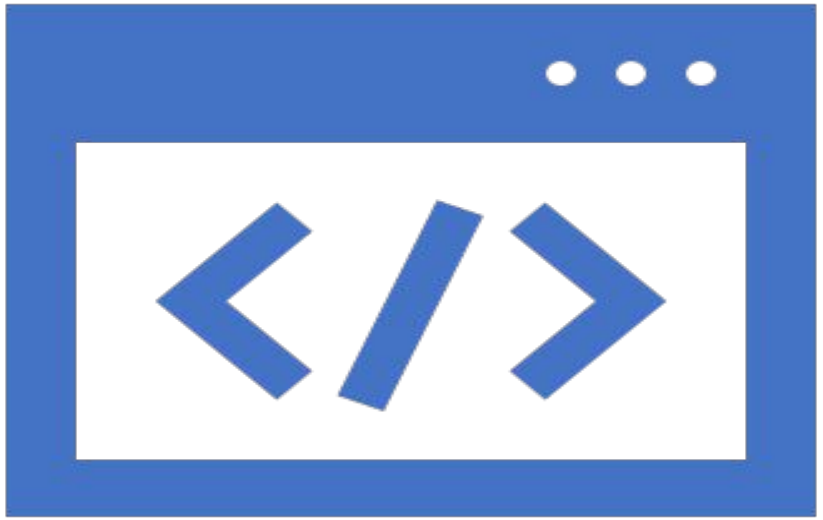
Handler Filters: These filters are applied at the individual page handler method level and allow you to perform actions before or after a specific handler method is executed. They are defined using attributes and can be used to handle scenarios like input validation, model binding, logging, and so on.



Razor Page Validations

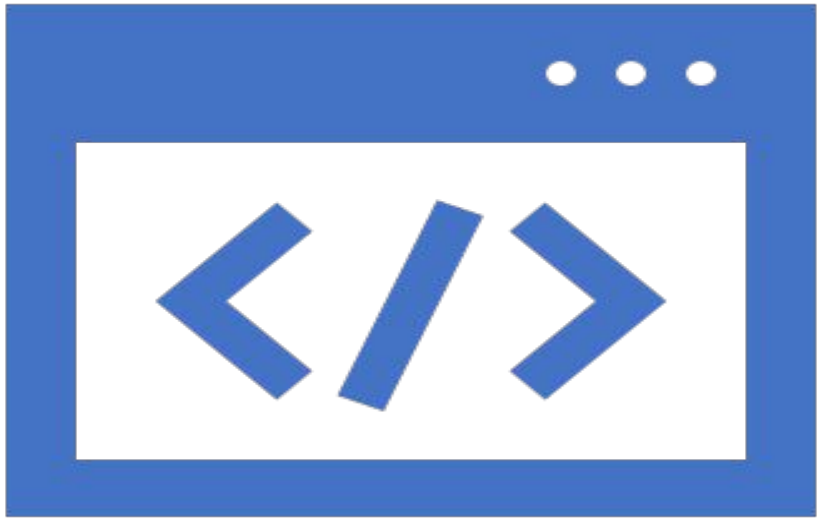
In Razor Pages, you can perform validations on user input to ensure that the data entered by the user meets certain criteria or constraints.

There are various ways to implement validations in Razor Pages, including client-side and server-side validations.



Client Side Validation

Required Fields: You can use the data-val-required and data-val attributes to mark fields as required. Razor Pages automatically generates the necessary JavaScript validation code based on these attributes.



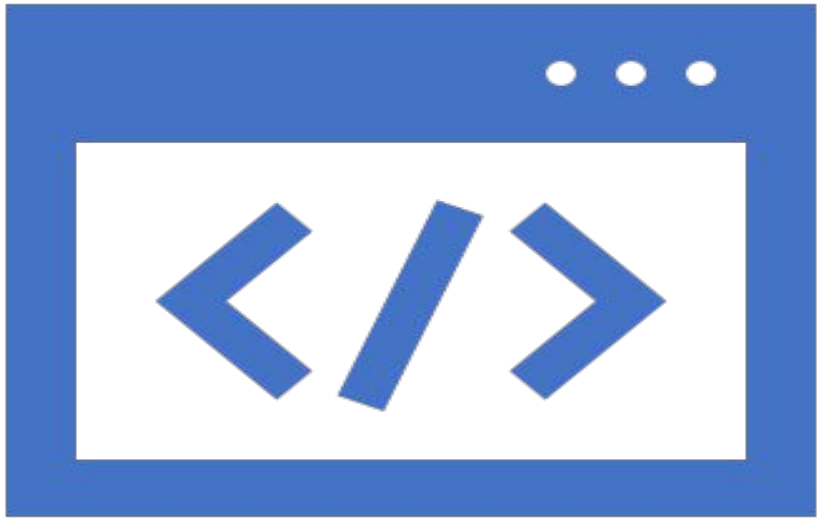
```
<input asp-for="Name" data-val="true" data-val-required="The Name field is required" />
<span asp-validation-for="Name"></span>
```

Client Side Validation

Data Type Validations: You can use data annotations on your model properties to specify the expected data types, such as [Required], [StringLength], [Range], etc.

```
public class MyModel
{
    [Required]
    public string Name { get; set; }

    [Range(18, 99)]
    public int Age { get; set; }
}
```

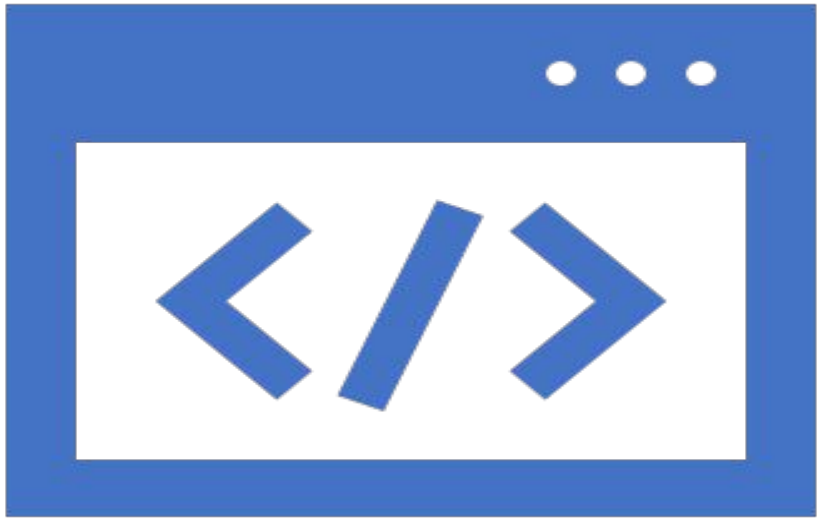


Server Side Validation

ModelState: Razor Pages automatically validates the model state during model binding. You can check for validation errors using the ModelState.IsValid property and display error messages in your Razor Pages.

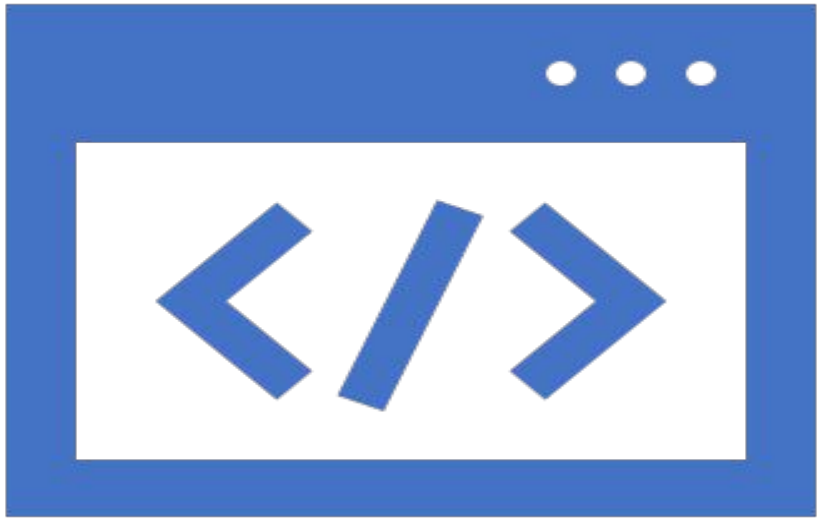
```
public IActionResult OnPost()
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    // Process valid data
    return RedirectToPage("Success");
}
```



Server Side Validation

Custom Validations: You can create custom validation attributes by deriving from the ValidationAttribute class and overriding the IsValid method. This allows you to implement custom validation logic based on your requirements.



```
public class CustomValidationAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value, ValidationContext validationContext)
    {
        // Custom validation logic
        if (/* validation fails */)
        {
            return new ValidationResult("Validation failed");
        }

        return ValidationResult.Success;
    }
}

public class MyModel
{
    [CustomValidation]
    public string SomeProperty { get; set; }
}
```


thank you