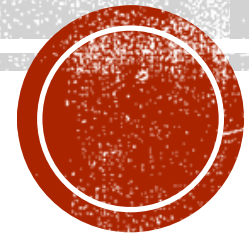


BINDING AND VALIDATING REQUESTS WITH RAZOR PAGES

Kamal Beydoun

Lebanese University – Faculty of Sciences I

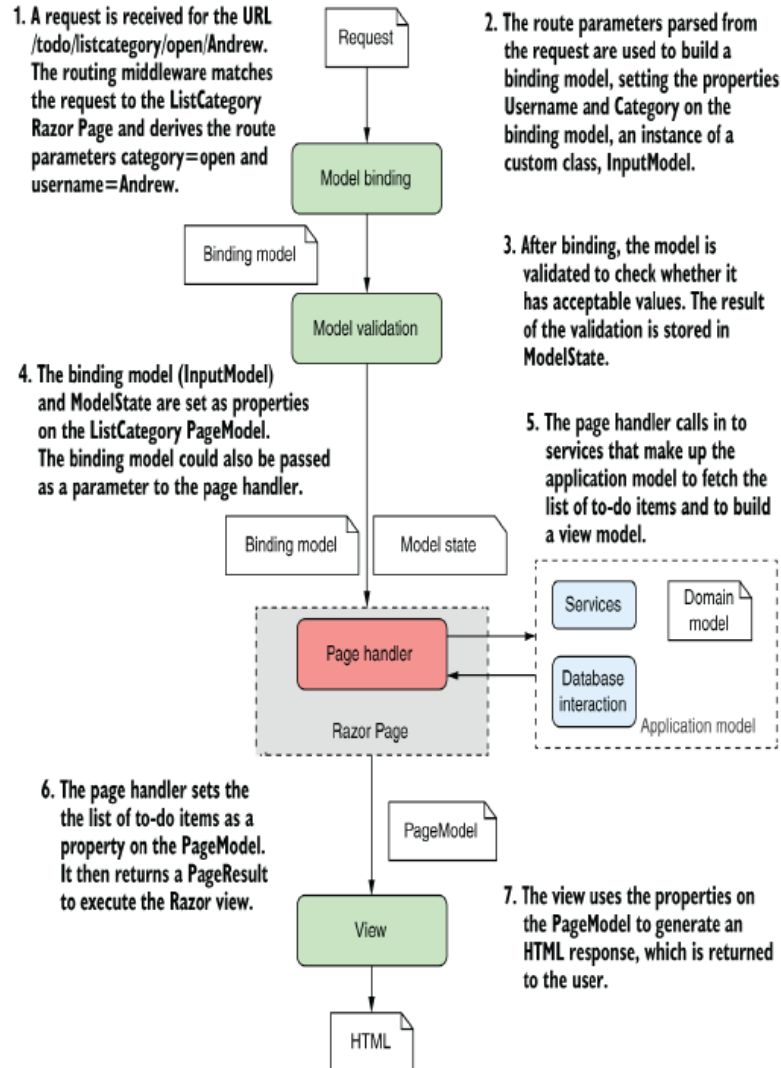
Kamal.beydoun@ul.edu.lb



UNDERSTANDING THE MODELS IN RAZOR PAGES AND MVC

- The classic MVC design pattern has three independent components:
 - **Model**—The data to display and the methods for updating this data
 - **View**—Displays a representation of data that makes up the model
 - **Controller**—Calls methods on the model and selects a view
- ASP.NET Core has multiple models, which takes the single-responsibility principle (SRP) one step further than some views of MVC.

UNDERSTANDING THE MODELS IN RAZOR PAGES AND MVC



UNDERSTANDING THE MODELS IN RAZOR PAGES AND MVC

- ASP.NET Core Razor Pages uses several models, most of which are plain old CLR objects (POCOs), and the application model, which is more of a concept around a collection of services.
- Each of the models in ASP.NET Core is responsible for handling a **different aspect** of the overall request

UNDERSTANDING THE MODELS IN RAZOR PAGES AND MVC — BINDING MODEL

- The binding model is all the information that's provided by the user when making a request, as well as additional contextual data.
 - route parameters parsed from the URL,
 - the query string, and
 - form or JavaScript Object Notation (JSON) data in the request body
- Binding models in Razor Pages are typically defined by creating a **public property** on the page's PageModel and decorating it with the **[BindProperty]** attribute.
 - They can also be passed to a page handler as **parameters**.

UNDERSTANDING THE MODELS IN RAZOR PAGES AND MVC — APPLICATION MODEL

- The application model isn't really an ASP.NET Core model at all. It's typically a whole group of different services and classes.
 - **Anything needed** to perform some sort of business action in your application.

UNDERSTANDING THE MODELS IN RAZOR PAGES AND MVC — PAGE MODEL

- The **PageModel** of a Razor Page serves two main functions:
 - it acts as the controller for the application by **exposing** page handler methods
 - it acts as the view model for a Razor view.
- All the **data required** for the view to generate a response is **exposed** on the PageModel.

FROM REQUEST TO BINDING MODEL: MAKING THE REQUEST USEFUL

- The process of **extracting** values from the request and creating C# objects from them is called **model binding**.
- Any publicly settable properties on your Razor Page's PageModel that are decorated with the **[BindProperty]** attribute are **created** from the **incoming request** using model binding

```
public class IndexModel: PageModel
{
    [BindProperty]                                ❶
    public string Category { get; set; }          ❶

    [BindProperty(SupportsGet = true)]            ❷
    public string Username { get; set; }          ❷

    public void OnGet()
    {
    }

    public void OnPost(ProductModel model)        ❸
    {
    }
}
```

- ❶ Properties decorated with **[BindProperty]** take part in model binding.
- ❷ Properties are not model-bound for GET requests unless you use **SupportsGet**.
- ❸ Parameters to page handlers are also model-bound when that handler is selected.

Which part is the binding model?

Listing 16.1 shows a Razor Page that uses multiple binding models: the Category property, the Username property, and the ProductModel parameter (in the OnPost handler) are all model-bound.

Using multiple models in this way is fine, but I prefer to use an approach that keeps all the model binding in a single, nested class, which I often call InputModel. With this approach, the Razor Page in listing 16.1 could be written as follows:

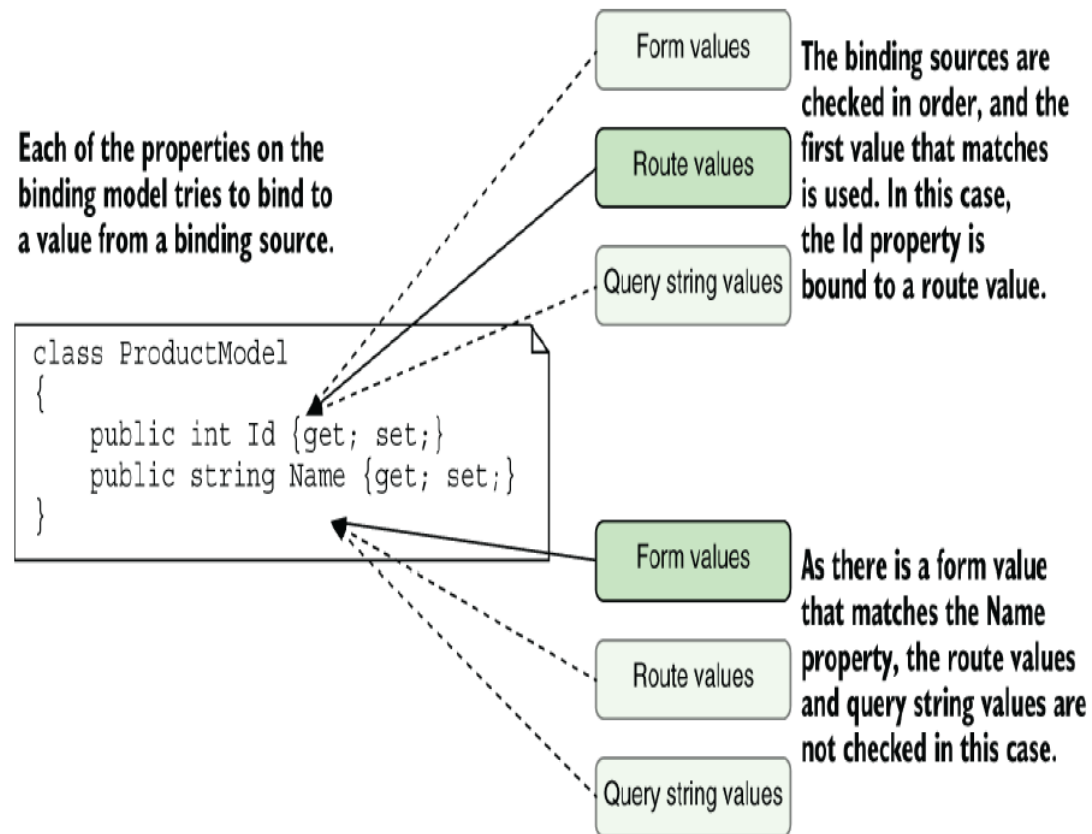
```
public class IndexModel: PageModel
{
    [BindProperty]
    public InputModel Input { get; set; }
    public void OnGet()
    {
    }

    public class InputModel
    {
        public string Category { get; set; }
        public string Username { get; set; }
        public ProductModel Model { get; set; }
    }
}
```

FROM REQUEST TO BINDING MODEL: MAKING THE REQUEST USEFUL

- By default, ASP.NET Core uses three different binding sources when creating your binding models in Razor Pages. It takes the first value it finds (if any) that **matches the name** of the binding model.
- **Form values**—Sent in the body of an HTTP request when a form is sent to the server using a POST.
- **Route values**—Obtained from URL segments or through **default** values after matching a route.
- **Query string values**—Passed at the end of the URL, not used during routing.

FROM REQUEST TO BINDING MODEL: MAKING THE REQUEST USEFUL



NOTE

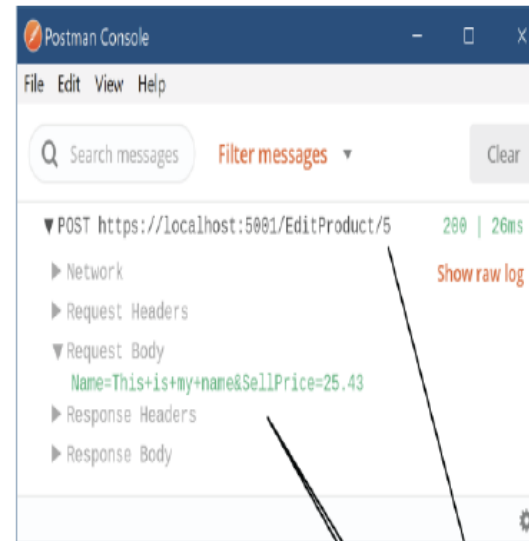
- In Razor Pages, **different properties** of a complex model can be model-bound to **different sources**.
 - This differs from minimal APIs, where the whole object would be bound from a **single source**, and “**partial**” binding is **not** possible.
- Razor Pages also bind to **form bodies** by default, while minimal APIs cannot.

PAGEMODEL PROPERTIES OR PAGE HANDLER PARAMETERS?

- Ebook

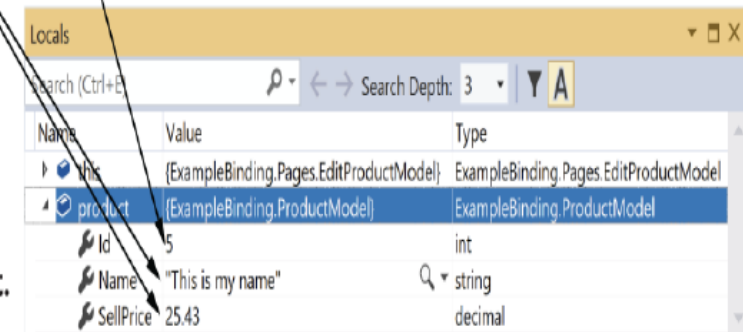
EXAMPLE

```
public void OnPost(ProductModel product)
```



An HTTP request is received and is directed to the `EditProduct` Razor Page by routing. A property of type `ProductModel` is decorated with a `[BindProperty]` attribute.

The model binder builds a new `ProductModel` object using values taken from the request. This includes route values from the URL, as well as data sent in the body of the request.



BINDING SIMPLE TYPES

Listing 16.2 A Razor Page accepting a simple parameter

```
public class CalculateSquareModel : PageModel
{
    public void OnGet(int number)           ❶
    {
        Square = number * number;          ❷
    }

    public int Square { get; set; }         ❸
}
```

@page "{**number**}"

- ❶ The method parameter is the binding model.
- ❷ A more complex example would do this work in an external service, in the application model.
- ❸ The result is exposed as a property and is used by the view to generate a response.

BINDING SIMPLE TYPES

If **none** of the binding sources contains the required value, the binding model is set to a **new, default instance** of the type instead.

- For **value types**, the value will be `default(T)`. For an `int` parameter this would be `0`, and for a `bool` it would be `false`.
- For **reference types**, the type is created using the default (parameter less) constructor. For custom types like `ProductModel`, that will create a new object.
- For **nullable types** like `int?` or `bool?`, the value will be **null**.
- For **string types**, the value will be **null**.

BINDING SIMPLE TYPES

```
public class ConvertModel : PageModel
{
    public void OnGet(
        string currencyIn,
        string currencyOut,
        int qty
    )
    {
        /* method implementation */
    }
}
```

@page "{currencyIn}/{currencyOut}"

URL (route values)	HTTP body data (form values)	Parameter values bound
/GBP/USD		currencyIn=GBP currencyOut=USD qty=0
/GBP/USD?currencyIn=CAD	QTY=50	currencyIn=GBP currencyOut=USD qty=50
/GBP/USD?qty=100	qty=50	currencyIn=GBP currencyOut=USD qty=50
/GBP/USD?qty=100	currencyIn=CAD& currencyOut=EUR&qty=50	currencyIn=CAD currencyOut=EUR qty=50

The default model binder isn't case-sensitive

BINDING SIMPLE TYPES

- The values stored in binding sources **are all strings**.
- The model binder will convert pretty much any primitive .NET type such as int, float, decimal (and string obviously), any custom type that has a **TryParse** plus anything that has a `TypeConverter`.

BINDING COMPLEX TYPES - SIMPLIFYING METHOD PARAMETERS BY BINDING TO COMPLEX OBJECTS

```
public IActionResult OnPost(  
    string firstName, string lastName,  
    string phoneNumber, string email)
```

Four parameters might not seem that bad right now, but what happens when the **requirements** change and you need to collect other details?

```
public IActionResult OnPost(UserBindingModel user)|
```

```
public class UserBindingModel  
{  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public string Email { get; set; }  
    public string PhoneNumber { get; set; }  
}
```

```
public IActionResult OnPost()
```

```
[BindProperty]  
public UserBindingModel User { get; set; }
```

You can bind only properties that are public and settable.

BINDING COLLECTIONS AND DICTIONARIES

ListBinding	ListBinding
<h3>Select currencies</h3> <div><div>GBP</div><div>USD</div><div>CAD</div><div>EUR</div></div> <div>Filter</div>	<h3>Exchange rates</h3> <div>USD 1.22</div> <div>CAD 1.64</div>
© 2020 - ListBinding - Privacy	© 2020 - ListBinding - Privacy

```
public void OnPost(List<string> currencies);|
```

BINDING COLLECTIONS AND DICTIONARIES

You could then POST data to this method by providing values in several different formats:

- **currencies[index]**—Where `currencies` is the name of the parameter to bind and `index` is the index of the item to bind, such as `currencies[0]=GBP¤cies[1]=USD`.
- **[index]**—If you're binding to a single list (as in this example), you can omit the name of the parameter, such as `[0]=GBP&[1]=USD`.
- **currencies**—Alternatively, you can omit the index and send `currencies` as the key for every value, such as `currencies=GBP¤cies=USD`.

BINDING FILE UPLOADS WITH IFORMFILE

- Razor Pages supports users uploading files by exposing the **IFormFile** and **IFormFileCollection** interfaces.
- You can use these interfaces as your binding model, either as a method parameter to your page handler or using the **[BindProperty]** approach,

```
public void OnPost(IFormFile file);
```

```
public void OnPost(IEnumerable<IFormFile> file);
```


CHOOSING A BINDING SOURCE

- The most common scenarios are when you want to bind a method parameter to a **request header value** or when the **body of a request contains JSON-formatted data** that you want to bind to a parameter.

```
public class PhotosModel: PageModel
{
    public void OnPost(
        [FromHeader] string userId,      ❶
        [FromBody] List<Photo> photos)  ❷
    {
        /* method implementation */
    }
}
```

- ❶ The `userId` is bound from an HTTP header in the request.
- ❷ The list of photo objects is bound to the body of the request, typically in JSON format.

CHOOSING A BINDING SOURCE

You can use a few different attributes to override the defaults and to specify a binding source for each binding model (or each property on the binding model)

- **[FromHeader]**—Bind to a header value.
- **[FromQuery]**—Bind to a query string value.
- **[FromRoute]**—Bind to route parameters.
- **[FromForm]**—Bind to **form** data posted in the **body** of the request. This attribute is not available in minimal APIs.
- **[FromBody]**—Bind to the request's body content.

Only one value may be decorated with the **[FromBody]** attribute. Also, as form data is sent in the body of a request, the **[FromBody]** and **[FromForm]** attributes are effectively mutually exclusive.

CHOOSING A BINDING SOURCE

- **[BindNever]**—The model binder will skip this parameter completely. You can use this attribute to prevent mass assignment,
- **[BindRequired]**—If the parameter was not provided or was empty, the binder will add a **validation error**.
- **[FromServices]**—This is used to indicate the parameter should be provided using dependency injection (DI).
 - This attribute isn't required in most cases, as .NET 7 is smart enough to know that a parameter is a service registered in DI, but you can be explicit if you prefer.

VALIDATING BINDING MODELS - VALIDATION IN RAZOR PAGES

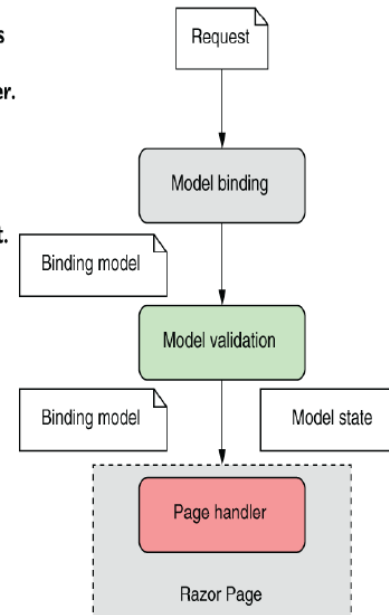
- Minimal APIs don't have any direct support for validation in the framework; you have to layer it on top using **filters and additional packages**.
- In Razor Pages, validation is built in. Validation occurs automatically after model binding but before the page handler executes.

1. A request is received for the URL /checkout/saveuser, and the routing middleware selects the SaveUser Razor Page endpoint in the Checkout folder.

2. The framework builds a UserBindingModel from the details provided in the request.

3. The UserBindingModel is validated according to the DataAnnotation attributes on its properties.

4. The UserBindingModel and validation ModelState are set on the SaveUser Razor Page, and the page handler is executed.



VALIDATING BINDING MODELS - VALIDATION IN RAZOR PAGES

```
public class UserBindingModel
{
    [Required]
    [StringLength(100)]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

    [Required]
    [StringLength(100)]
    [Display(Name = "Last name")]
    public string LastName { get; set; }

    [Required]
    [EmailAddress]
    public string Email { get; set; }

    [Phone]
    [Display(Name = "Phone number")]
    public string PhoneNumber { get; set; }
}
```

①

②

③

④

⑤

For validation requirements that don't lend themselves to attributes, such as when the validity of one property depends on the value of another, you can implement **InvalidatableObject**

VALIDATING ON THE SERVER FOR SAFETY

- Validation of the binding model occurs before the page handler executes, but note that **the handler always executes**, whether the validation **failed or succeeded**.
 - It's the responsibility of the page handler to **check** the result of the validation.
- The Razor Pages framework stores the output of the validation attempt in a property on the PageModel called ModelState.
 - This property is a **ModelStateDictionary** object

VALIDATING ON THE SERVER FOR SAFETY

```
public class CheckoutModel : PageModel ❶
{
    [BindProperty] ❷
    public UserBindingModel Input { get; set; } ❷

    public IActionResult OnPost() ❸
    {
        if (!ModelState.IsValid) ❹
        {
            return Page(); ❺
        }

        /* Save to the database, update user, return success */ ❻
        return RedirectToPage("Success");
    }
}
```

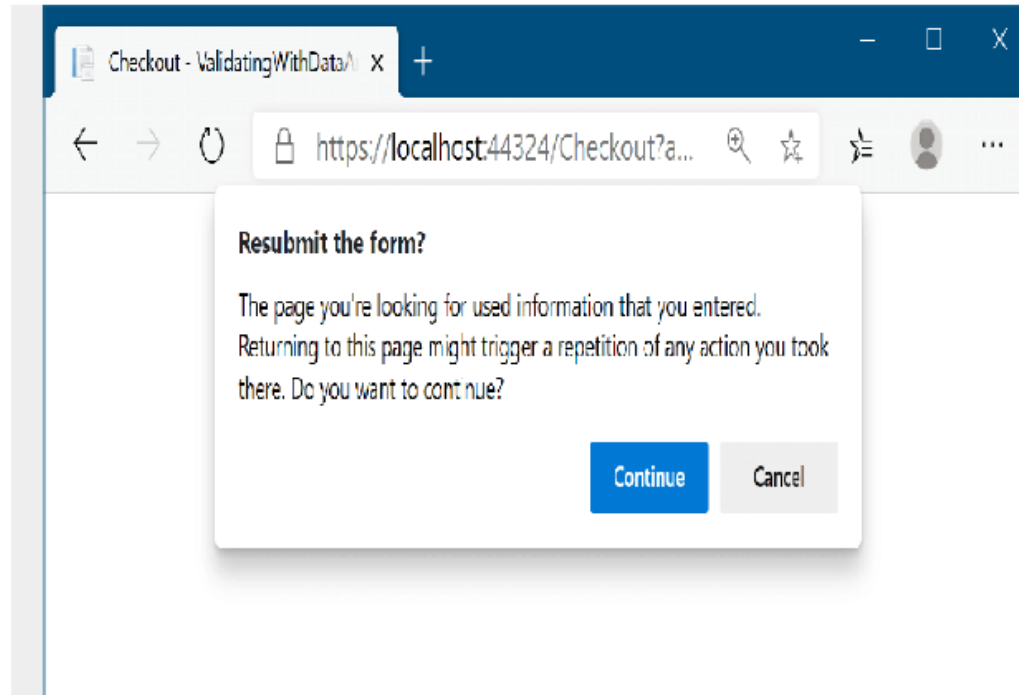
- ❶ The ModelState property is available on the PageModel base class.
- ❷ The Input property contains the model-bound data.
- ❸ The binding model is validated before the page handler is executed.
- ❹ If there were validation errors, IsValid will be false.
- ❺ Validation failed, so redisplay the form with errors and finish the method early.
- ❻ Validation passed, so it's safe to use the data provided in the model.

The screenshot shows a web form titled "Checkout" with the instruction "Enter the values and click Submit". It contains four input fields: "Your name" (filled with "Andrew"), "Last name" (empty), "Email" (filled with "Not an email"), and "Phone number" (filled with "Not a number"). Each of the three empty or invalid fields has a red error message below it: "The Last name field is required.", "The Email field is not a valid e-mail address.", and "The Phone number field is not a valid phone number." respectively. A blue "Submit" button is at the bottom. The footer text reads "© 2020 - ValidatingWithDataAnnotations - Privacy".

- You can also add extra validation errors to the collection, **such as business rule validation errors** that come from a different system.
- You can add **errors to ModelState** by calling **AddModelError()**, which will be displayed to users on the form alongside the DataAnnotation attribute errors.

POST-REDIRECT-GET

■ Ebook



Refreshing a browser window after a POST causes a warning message to be shown to the user

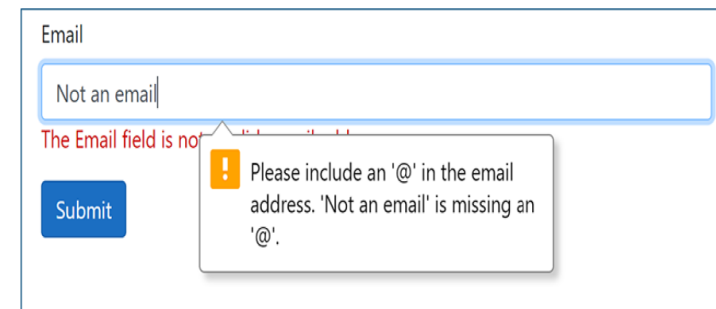
The POST-REDIRECT-GET pattern says that in response to a successful POST, you should return a REDIRECT response to a new URL, which will be followed by the browser making a GET to the new URL. If the user refreshes their browser now, they'll be refreshing the final GET call to the new URL. No additional POST is made, so no additional payments or side effects should occur.

This pattern is easy to achieve in ASP.NET Core applications using the pattern shown in listing 16.7. By returning a `RedirectToPageResult` after a successful POST, your application will be safe if the user refreshes the page in their browser.

VALIDATING ON THE CLIENT FOR USER EXPERIENCE

You can add client-side validation to your application in a few different ways.

- HTML5 has several built-in validation behaviors that many browsers use.
- Run JavaScript on the page and checking the values the user entered before submitting the form. This is the most common approach used in Razor Pages.



The screenshot shows a web form with a label "Email" above a text input field. The input field contains the text "Not an email". Below the input field, there is a red error message that reads "The Email field is not valid". A blue "Submit" button is located below the error message. A tooltip with an orange exclamation mark icon is displayed, containing the text: "Please include an '@' in the email address. 'Not an email' is missing an '@'."

ORGANIZING YOUR BINDING MODELS IN RAZOR PAGES

- Ebook — Very Important