

ASP.NET Core web applications require the input data, but from where it fetches the input data that is from the request. The request may send the data in various formats in different ways. For example, the request may contain the input data in the form of request headers, request query string parameters, or in the request body either in the form of a query string, JSON, or XML. Input parameters may also be sent via route parameters. So, there are multiple sources of request data. It is the responsibility of the server code to retrieve all that input data from the request before starting the process. However, writing manual code for retrieving the values from all these sources is error-prone, time-consuming, and lengthy.

To minimize and simplify the code for retrieving the data from the request parameters, ASP.NET Core has a concept called model binding. The target of model binding is to retrieve the data from various input sources in the request and make it available to the parameters of the controller action methods. This means a controller action method can receive data from the request body, request headers, route data, or any part of the request.

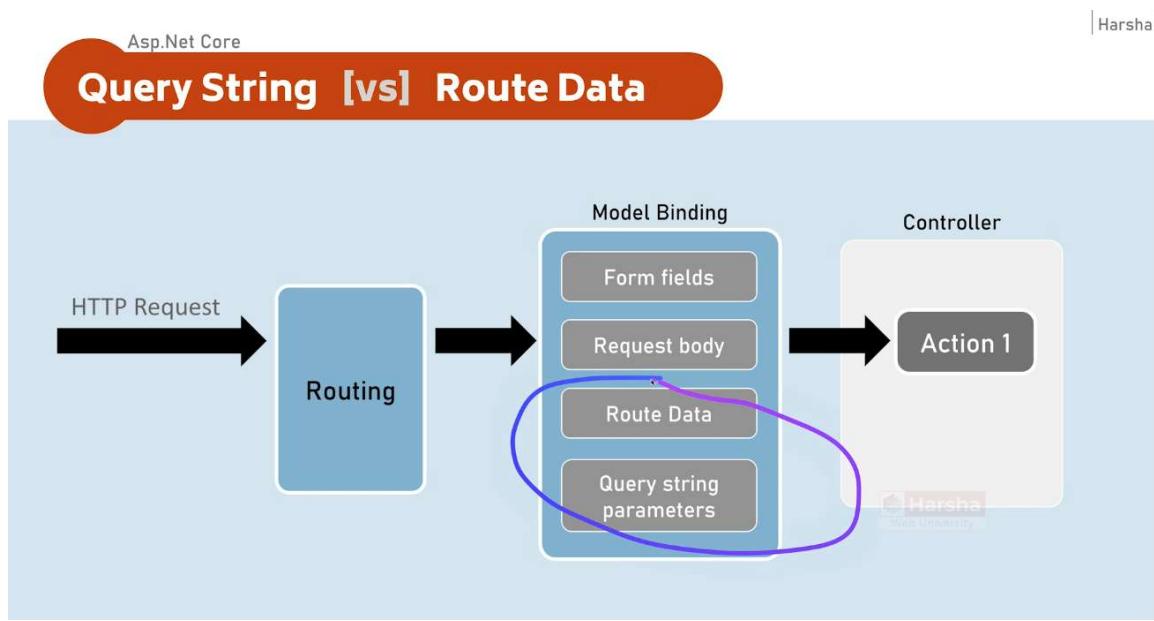
As a developer, you need to know how to handle model data and retrieve data from the request using model binding. Similarly, you can specify the specific source from which you would like to retrieve the data. For example, if you are interested in retrieving the value from the route parameters but not from the request headers, you can specify that in the code. This is what we will learn in this lecture.

By default, the controller parameters retrieve data in a specific order, with higher priority given to the top source and lower priority to the bottom. For instance, if you supply a value `x=10` where the parameter name is `x` and the value is 10, and it is present in all data sources, the top source will be picked while the rest are ignored. However, as a developer, you can specify the specific data source for your parameter. For example, you can declare that a parameter gets its data from route data rather than the request body.

Overall, model binding is a feature or process that happens when a request is received. When routing decides on a specific action method to execute based on the route URL, the model binding process occurs in the background before the execution control reaches the action method. This means that model binding occurs automatically upon each request to fetch data from the request. How to retrieve values processed by model binding will be discussed in further lectures.

Asp.Net Core

Query String vs Route Data



This is the order in which the model binding process picks up the values from the request. Whenever the URL matches a specific route mapped to a particular action method, before executing the action method of the controller, the model binding process occurs. It tries to read the values in the same order from top to bottom.

For now, let's temporarily focus on the last two: route data and query string.

The route data refers to the route parameters that we learned about in the routing section. You can represent the route parameters using curly braces while creating the routes. At runtime, as part of the URL, you can supply values for those route parameters, which are technically called route data in model binding.

First, the model binding process picks up values from the route data. Then it picks up values from the query string. Here, the query string refers to the query string parameters that are part of the URL in a GET request.

```
1 using Microsoft.AspNetCore.Mvc;
2
3 namespace IActionResultExample.Controllers
4 {
5     public class HomeController : Controller
6     {
7         [Route("bookstore")]
8         //Url: /bookstore?bookid=10&isloggedin=true
9         public IActionResult Index()
10        {
11            //Book id should be applied
12            if (!Request.Query.ContainsKey("bookid"))
13            {
14                //return new BadRequestResult();
15                return BadRequest("Book id is not supplied");
16            }
17
18            //Book id can't be empty
19            if (string.IsNullOrEmpty(Convert.ToString(Request.Query["bookid"])))
20            {
21                return BadRequest("Book id can't be null or empty");
22            }
23
24            //Book id should be between 1 to 1000
25            int bookId = Convert.ToInt16
26            (ControllerContext.HttpContext.Request.Query["bookid"]);
        }
    }
}
```

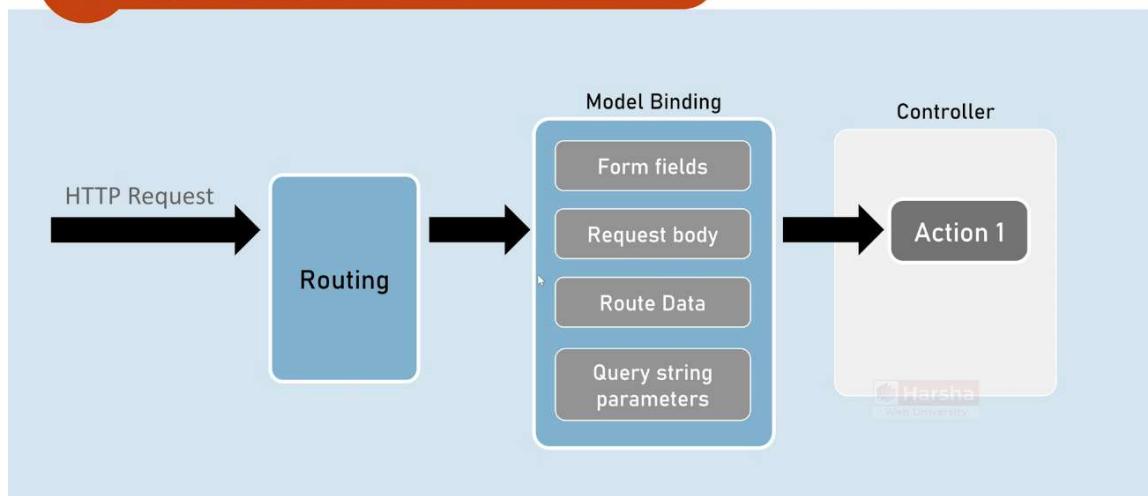
```
11
12     //Book id should be applied
13     if (!Request.Query.ContainsKey("bookid"))
14     {
15         //return new BadRequestResult();
16         return BadRequest("Book id is not supplied");
17     }
18
19     //Book id can't be empty
20     if (string.IsNullOrEmpty(Convert.ToString(Request.Query["bookid"])))
21     {
22         return BadRequest("Book id can't be null or empty");
23     }
24
25     //Book id should be between 1 to 1000
26     int bookId = Convert.ToInt16
        (ControllerContext.HttpContext.Request.Query["bookid"]);
    if (bookId <= 0)
    {
        return BadRequest("Book id should be greater than 0");
    }
}
```

This is the code from the last lecture of the previous section. As part of this code, we are receiving query string parameters in the URL like this. However, every time you want to retrieve values from the request query string, you have to write `request.query` or use `controllerContext.HttpContext.Request`. Isn't this error-prone and too lengthy? Yes, it is.

Is there a shortcut way to get the values easily? Yes, exactly—that is model binding. So how do you retrieve this `bookId` value through model binding?

Remember that model binding is not a process you perform manually. Model binding is executed automatically before the action method execution starts. As part of the action method, you can assume that the values are already retrieved by model binding. This means model binding has picked up the values from the request.

Query String [vs] Route Data



```

namespace ActionResultExample.Controllers
{
    public class HomeController : Controller
    {
        [Route("bookstore")]
        //Url: /bookstore?bookid=10&isloggedin=true
        public IActionResult Index(int bookid)
        {
            //Book id should be applied
            if (!Request.Query.ContainsKey("bookid"))
            {
                //return new BadRequestResult();
                return BadRequest("Book id is not supplied");
            }

            //Book id can't be empty
            if (string.IsNullOrEmpty(Convert.ToString(Request.Query["bookid"])))
        }
    }
}

```

A red circle highlights the line of code containing the route attribute: `[Route("bookstore")]`. A red arrow points from this line to the explanatory text below.

For example: int bookId—something like this. You can assume that ASP.NET Core automatically supplies the values for this parameter, which are retrieved through model binding.

Model binding always executes in the same order. At the fourth position, there is a source called query string parameters, which means this parameter can likely pick up the value from the query string.

So, let's execute this application with this URL.

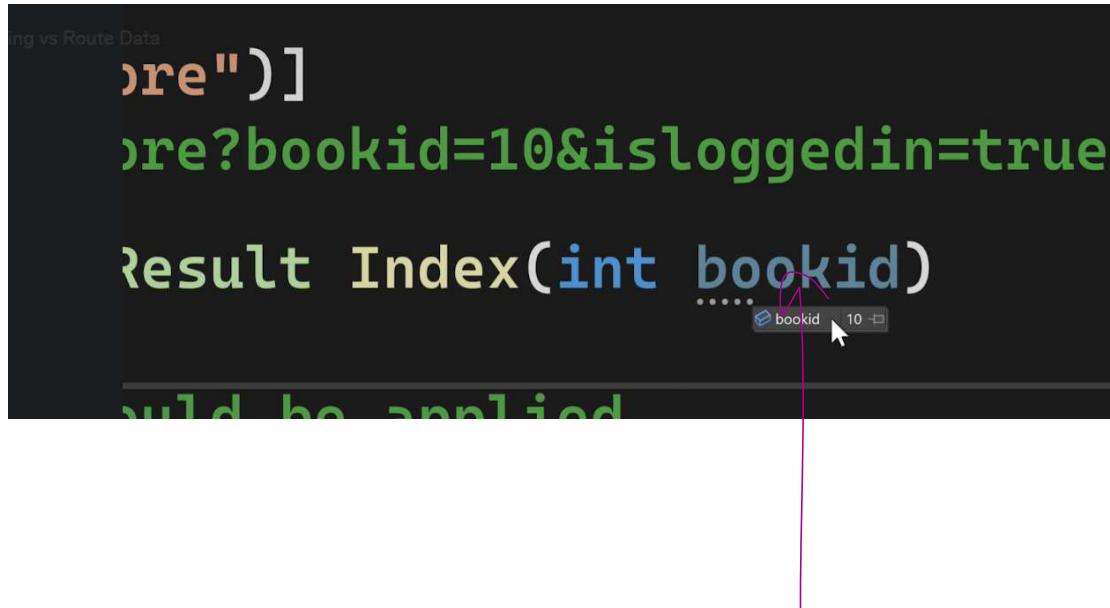
Let's send the request. Our action method can pickup the value.

	Value	Description
Content-Type	<calculated when request is sent>	
Content-Length	0	
User-Agent	PostmanRuntime/7.29.0	

so, the breakpoint hits!

```

4     {
5         public class HomeController : Controller
6     {
7         [Route("bookstore")]
8         //Url: /bookstore?bookid=10&isloggedin=true
9         public IActionResult Index(int bookid)
10        {
11            //Book id should be applied
12            if (!Request.Query.ContainsKey("bookid"))
13            {
14                //return new BadRequestResult();
15                return BadRequest("Book id is not supplied");
16            }
17        }
18    }
19 }
```



The benefit for the developer is that, everytime you do not have to write this lengthy syntax

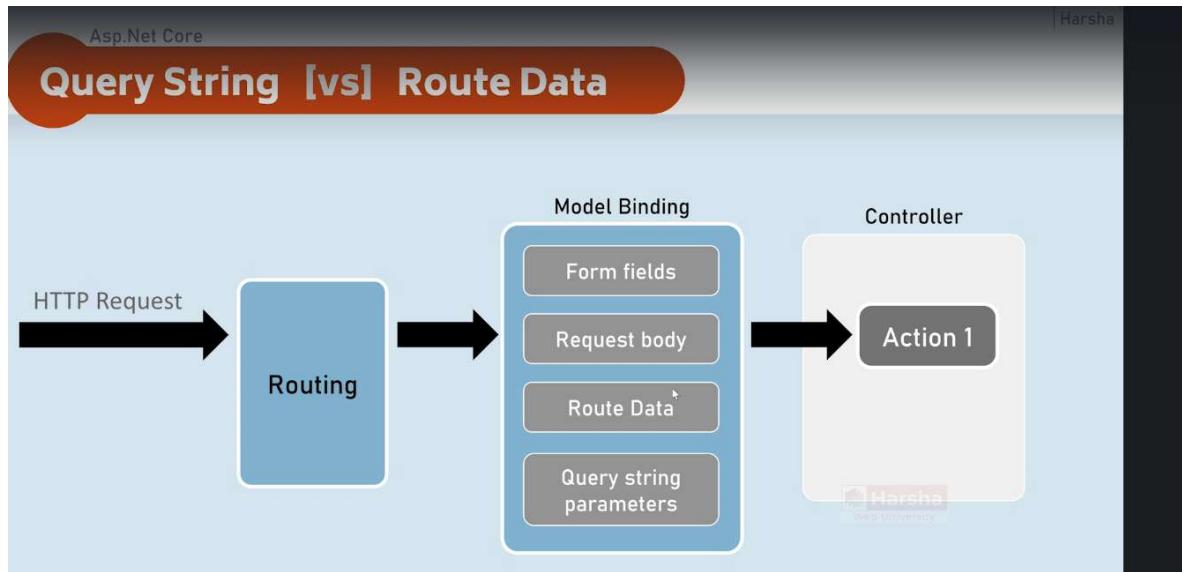
```
{  
    [Route("bookstore")]  
    //Url: /bookstore?bookid=10&isloggedin=true  
    0 references  
    public IActionResult Index(int bookid)  
    {  
        //Book id should be applied  
        if (!Request.Query.ContainsKey("bookid"))  
        {  
            //return new BadRequestResult();  
            return BadRequest("Book id is not supplied");  
        }  
        //Book id can't be empty  
    }  
}
```



you can simply use 'bookId' parameter

```
public class HomeController : Controller  
{  
    [Route("bookstore")]  
    //Url: /bookstore?bookid=10&isloggedin=true  
    0 references  
    public IActionResult Index(int bookId)  
    {  
        //Book id should be applied  
        if (!bookId)  
        {  
            //return new BadRequestResult();  
            return BadRequest("Book id is not supplied");  
        }  
        //Book id can't be empty  
    }  
}
```





Route Data has higher priority than Query String.

So, if we supply the same data from these two sources, 'route data' will be picked up.

let's prove it!

	VALUE
Content-Type	<calculated when request is sent>
User-Agent	0

```

namespace IActionResultExample.Controllers
{
    public class HomeController : Controller
    {
        //https://localhost:7119?bookid=123&isloggedin=true
        [Route("bookstore/{bookid?}/{isloggedin?}")]
        [Route("/")]
        public IActionResult Index(int? bookid, bool? isLoggedIn)
        {
            //Book id should be applied
            if (bookid.HasValue == false)
            {
                return BadRequest("Book id is not supplied or empty");
            }

            //Book id should be between 1 to 1000
            //int bookId = Convert.ToInt16(ControllerContext.HttpContext.Request.Query["bookid"]);
            if(bookid <= 0)
            {
                //Response.StatusCode = 400;
                //return Content("Book id can't be less or equal to zero");
                return BadRequest("Book id can't be less or equal to zero");
            }
        }
}

```

now, observe which value is picked up!

The screenshot shows the Visual Studio code editor with the following code:

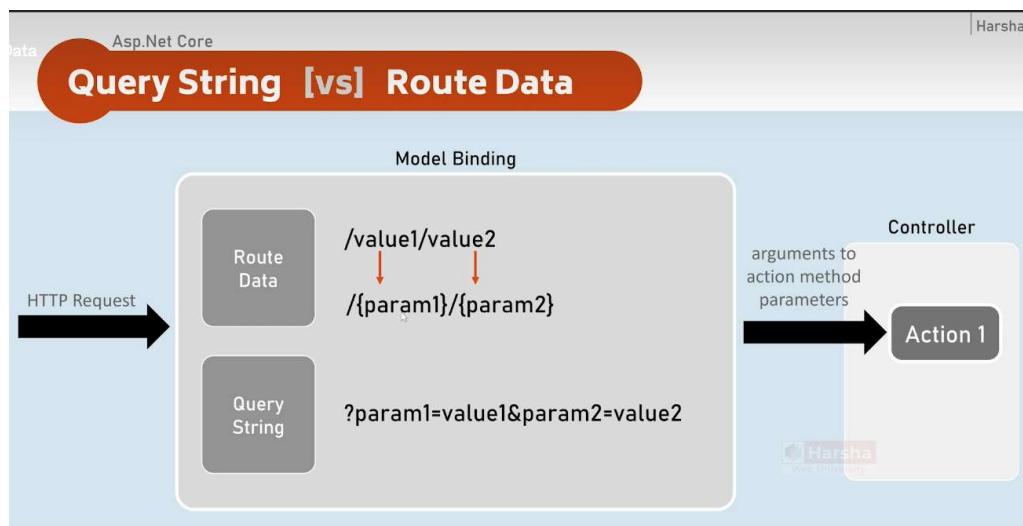
```

using Microsoft.AspNetCore.Mvc;
namespace IActionResultExample.Controllers
{
    public class HomeController : Controller
    {
        [Route("bookstore/{bookid?}/{isloggedin?}")]
        //Url: /bookstore?bookid=10&isloggedin=true
        public IActionResult Index(int? bookid, bool? isloggedin)
        {
            //Book id should be applied
            if (bookid.HasValue == false)
            {

```

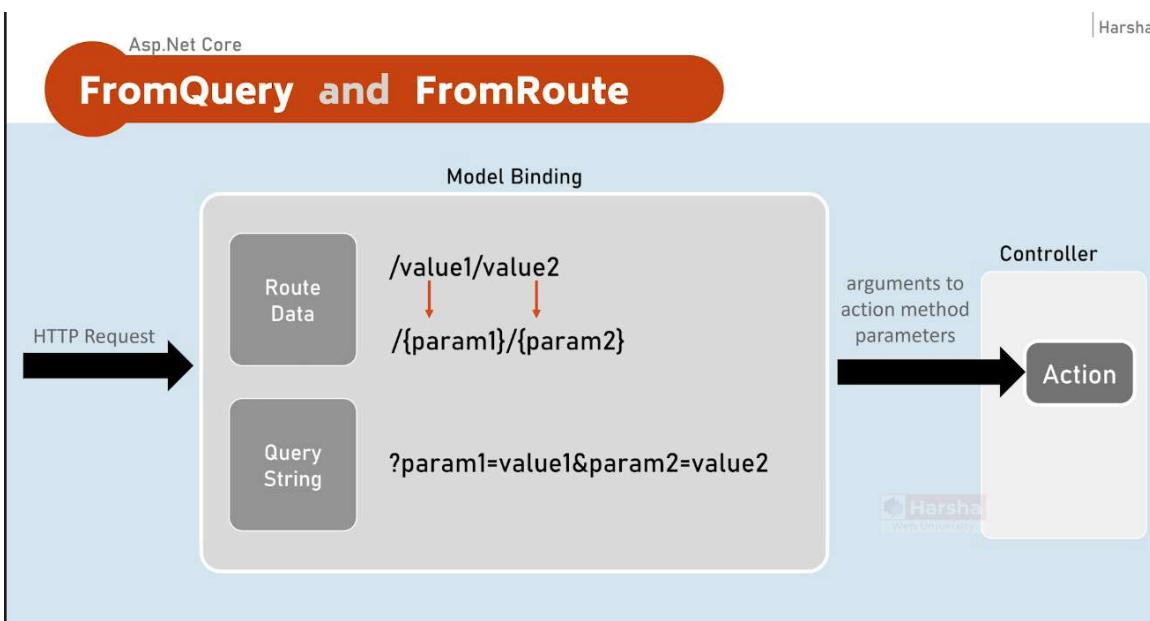
A tooltip from the 'Quick Actions' feature is displayed, showing the value '1' for the 'bookid' parameter. The tooltip text is:

Value
ActionResultExample.Controllers.HomeController
int?
bookid
File



Asp.Net Core

[FromQuery] and [FromRoute]



Model binding first picks up the value from the route data and then from the query string. This means whenever you create a parameter for the action method, that parameter will get its value from the route data first. If the value is null or not supplied, it will then try to get the value from the query string.

However, you can override this behavior based on your requirement by using `FromQuery` and `FromRoute` attributes for that particular parameter.

FromQuery and FromRoute

[FromQuery]

```
//gets the value from query string only
public IActionResult ActionMethodName([FromQuery] type parameter)
{
}
```

[FromRoute]

```
//gets the value from route parameters only
public IActionResult ActionMethodName([FromRoute] type parameter)
{
}
```

Intro

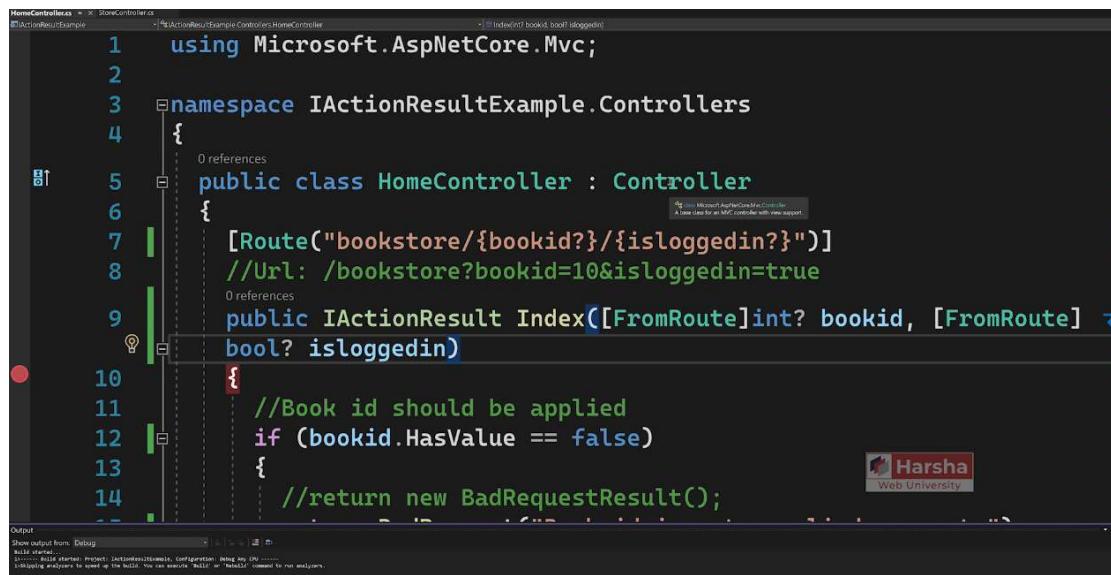
[FromRoute]

[FromQuery]

Conclusion

Udemy

You need to apply **FromQuery** if you prefer to retrieve the value from the query string. Similarly, you need to use **FromRoute** to retrieve the value from the route parameter.



The screenshot shows the `HomeController.cs` file in Visual Studio. The code defines a `HomeController` class that inherits from `Controller`. It contains a `Index` action method with two parameters: `int? bookid` and `bool? isloggedin`. The `bookid` parameter is annotated with `[FromRoute]`, indicating it should be retrieved from the route parameters. The `isloggedin` parameter is also annotated with `[FromRoute]`, indicating it should be retrieved from the query string. The code uses nullable reference types (`int?`, `bool?`) and includes comments explaining the purpose of each parameter.

```
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace IActionResultExample.Controllers
4  {
5      public class HomeController : Controller
6      {
7          [Route("bookstore/{bookid?}/{isloggedin?}")]
8          //Url: /bookstore?bookid=10&isloggedin=true
9          public IActionResult Index([FromRoute] int? bookid, [FromRoute] bool? isloggedin)
10         {
11             //Book id should be applied
12             if (bookid.HasValue == false)
13             {
14                 //return new BadRequestResult();
15             }
16         }
17     }
18 }
```

In this case, parameters are present in query string. not route parameters.

http://localhost:5202/bookstore?bookid=10&isloggedin=true

Set as variable | ...

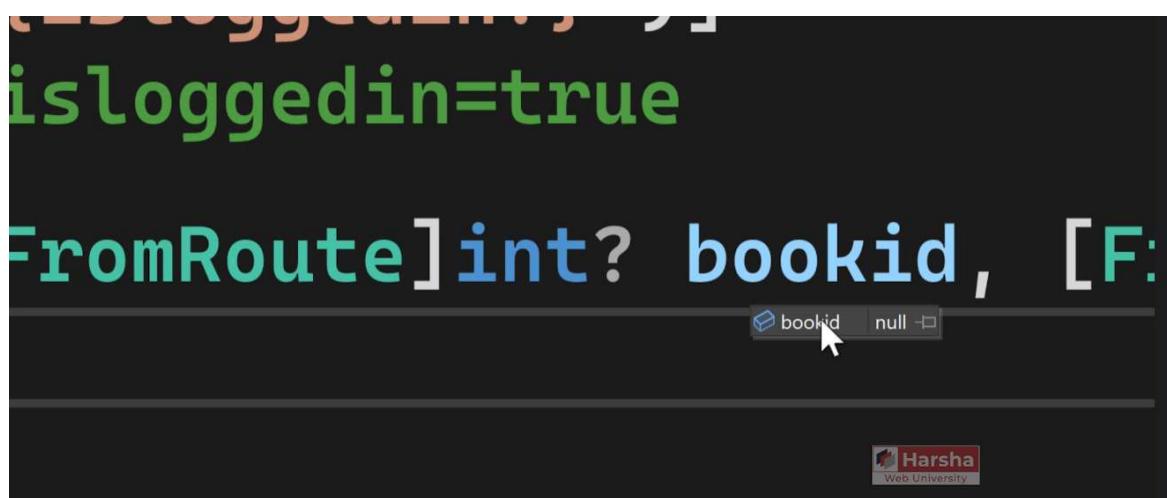
uthorization Headers (7) Body Pre-request Script Tests Settings

Hide auto-generated headers

	VALUE
x-Auth-Token ⓘ	<calculated when request is sent>

200 OK

we're getting Null as we set [FromRoute]



Similarly you can prefer to get the value from query string.

```

1  using Microsoft.AspNetCore.Mvc;
2
3  namespace IActionResultExample.Controllers
4  {
5      public class HomeController : Controller
6      {
7          [Route("bookstore/{bookid?}/{isloggedin?}")]
8          //Url: /bookstore?bookid=10&isloggedin=true
9          public IActionResult Index([FromQuery] int? bookid, [FromRoute] bool? isloggedin)
10         {
11             //Book id should be applied
12             if (bookid.HasValue == false)
13             {
14                 //return new BadRequestResult();
15                 return BadRequest("Book id is not supplied or empty");
16             }
17         }
18     }
19 }

```

bookid null

Asp.Net Core Model Class

Asp.Net Core | Harsha

Models

Model is a class that represents structure of data (as properties) that you would like to receive from the request and/or send to the response.
Also known as POCO (Plain Old CLR Objects).

```
graph LR; Request --> MB[Model Binding]; MB --> A[Action]; A --> EAR[Execute Action Result]; EAR --> Response;
```

Request → Model Binding → Action → Execute Action Result → Response

Controller

Action

//receive request data as model object
//other code
//send response data as model object

12:32

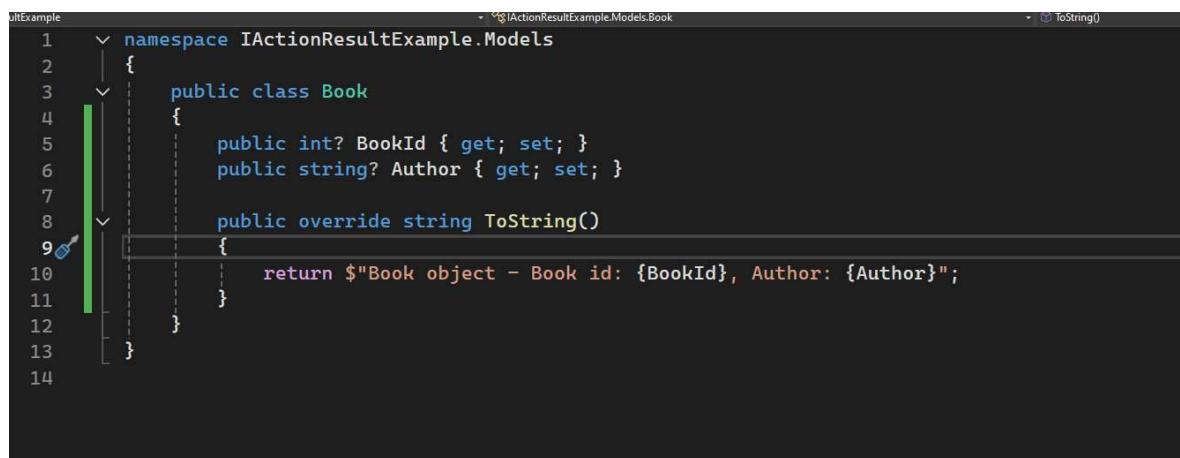
In ASP.NET Core, a model is a class that represents the structure of the data you want to receive from the request and send back as a response to the browser.

This means the action method can receive data in the form of a model object as a parameter, and optionally, you can return the model object as a response.

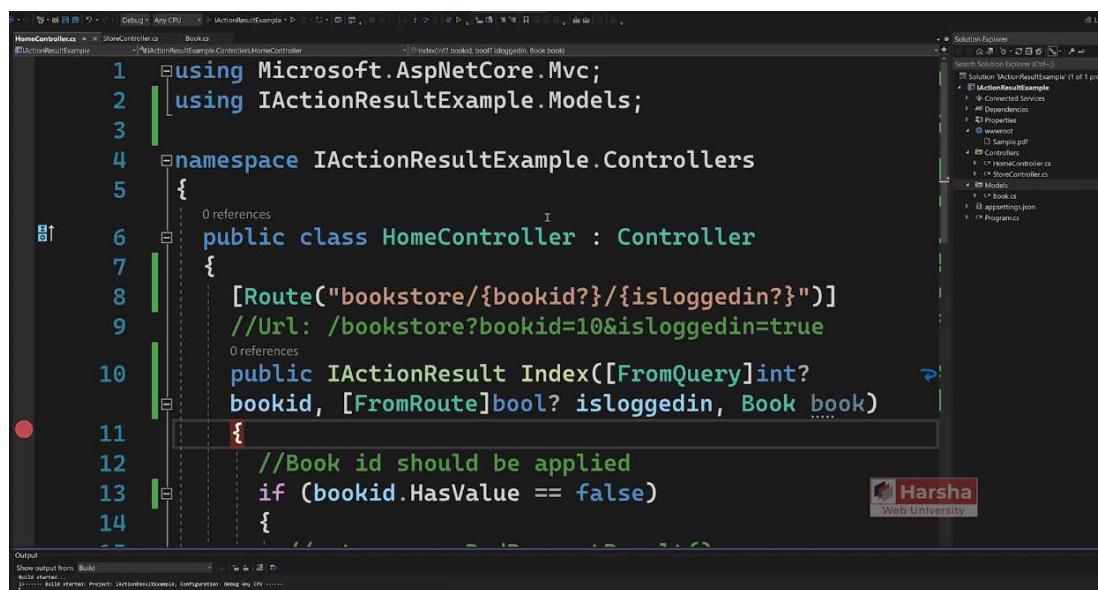
We will focus on returning the model object later. For now, let's focus on how to receive the model object as an argument.

In this case, the role of model binding is to first fetch all the values from the form data, request body, route data, and query string. If the action method has a parameter of the model class type, model binding automatically creates an object of the model class and populates its properties with the values from the request.

The reference to this model object is then received as an argument in the action method. Let me show that.

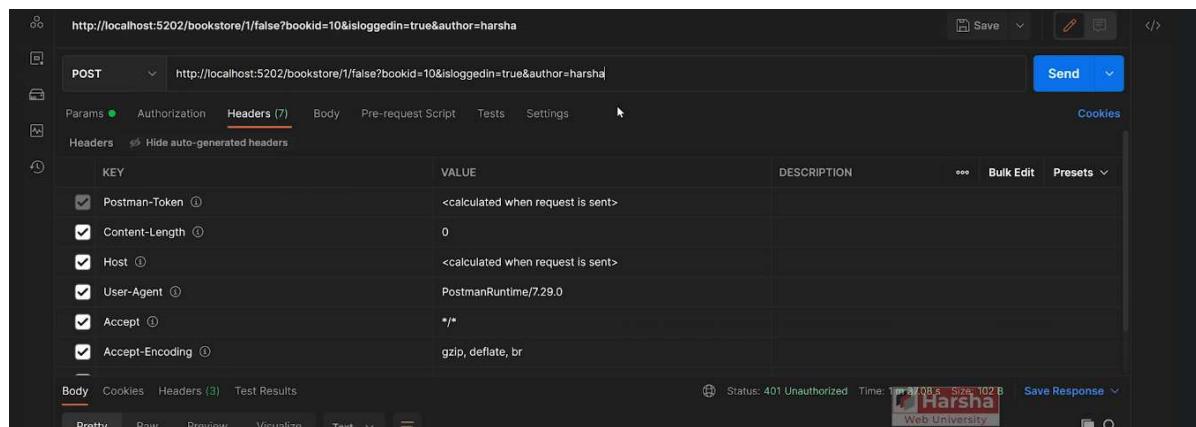


```
1  namespace IActionResultExample.Models
2  {
3      public class Book
4      {
5          public int? BookId { get; set; }
6          public string? Author { get; set; }
7
8          public override string ToString()
9          {
10             return $"Book object - Book id: {BookId}, Author: {Author}";
11         }
12     }
13 }
14
```



```
1  using Microsoft.AspNetCore.Mvc;
2  using IActionResultExample.Models;
3
4  namespace IActionResultExample.Controllers
5  {
6      public class HomeController : Controller
7      {
8          [Route("bookstore/{bookid?}/{isloggedin?}")]
9          //Url: /bookstore?bookid=10&isloggedin=true
10         public IActionResult Index([FromQuery]int?
11             bookid, [FromRoute]bool? isloggedin, Book book)
12         {
13             //Book id should be applied
14             if (bookid.HasValue == false)
15             {
16             }
17         }
18     }
19 }
```

let's run this.



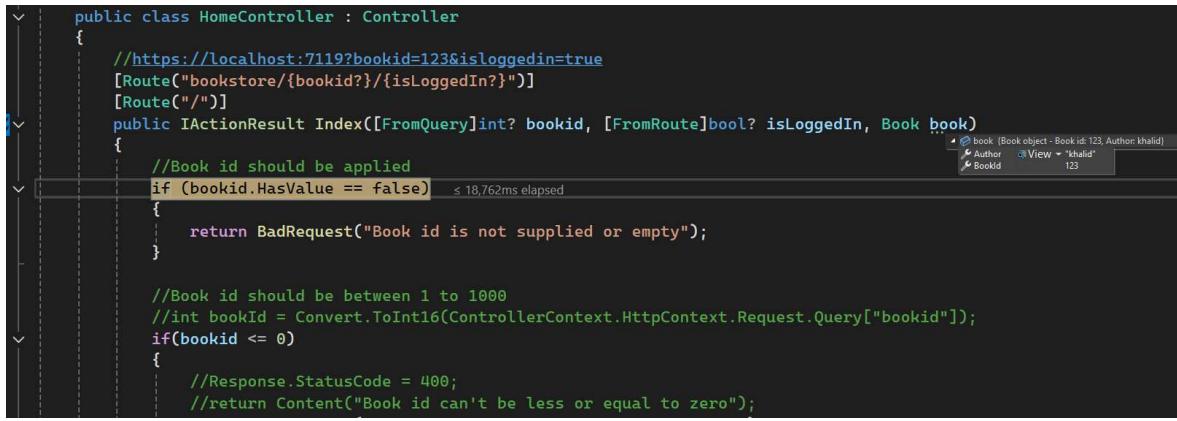
http://localhost:5202/bookstore/?false?bookid=10&isloggedin=true&author=harsha

POST http://localhost:5202/bookstore/?false?bookid=10&isloggedin=true&author=harsha

Headers (7)

KEY	VALUE	DESCRIPTION	Bulk Edit	Presets
Postman-Token	<calculated when request is sent>			
Content-Length	0			
Host	<calculated when request is sent>			
User-Agent	PostmanRuntime/7.29.0			
Accept	*			
Accept-Encoding	gzip, deflate, br			

Status: 401 Unauthorized Time: 1ms | 0.08s Size: 102 B Save Response



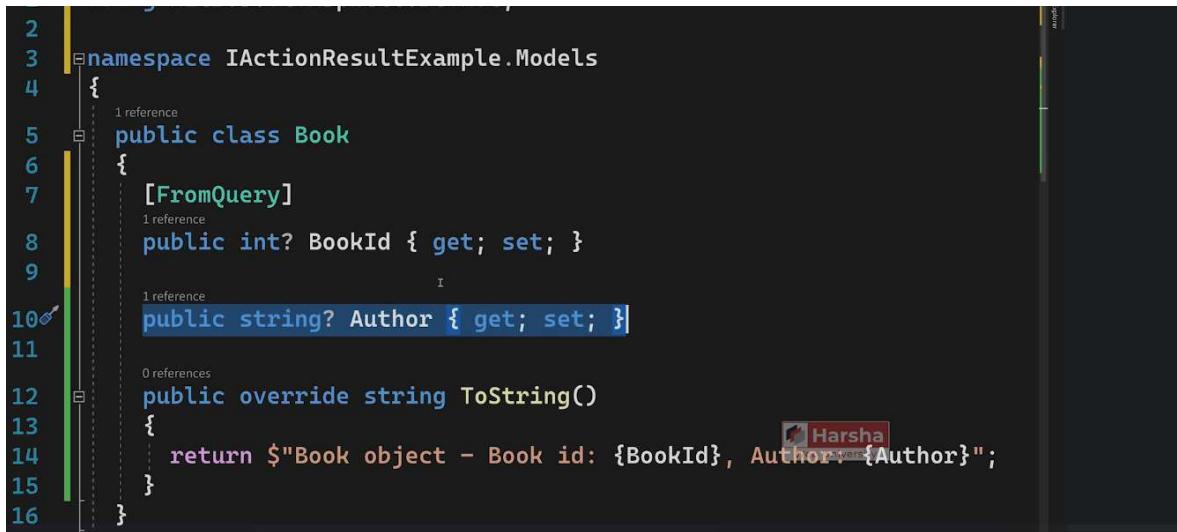
```
public class HomeController : Controller
{
    //https://localhost:7119?bookid=123&isloggedin=true
    [Route("bookstore/{bookid?}/{isLoggedIn?}")]
    [Route("/")]
    public IActionResult Index([FromQuery] int? bookid, [FromRoute] bool? isLoggedIn, Book book)
    {
        //Book id should be applied
        if (bookid.HasValue == false)           18,762ms elapsed
        {
            return BadRequest("Book id is not supplied or empty");
        }

        //Book id should be between 1 to 1000
        //int bookId = Convert.ToInt16(ControllerContext.HttpContext.Request.Query["bookid"]);
        if(bookid <= 0)
        {
            //Response.StatusCode = 400;
            //return Content("Book id can't be less or equal to zero");
        }
    }
}
```

first it will look from route query to fill book object. it is missing. so it gets the data from query string.

by specifying, [FromQuery] we are saying 'BookId' value will be picked up from the query string. not from the route data.

But for 'Author'. we have not set anything. it can be anything.



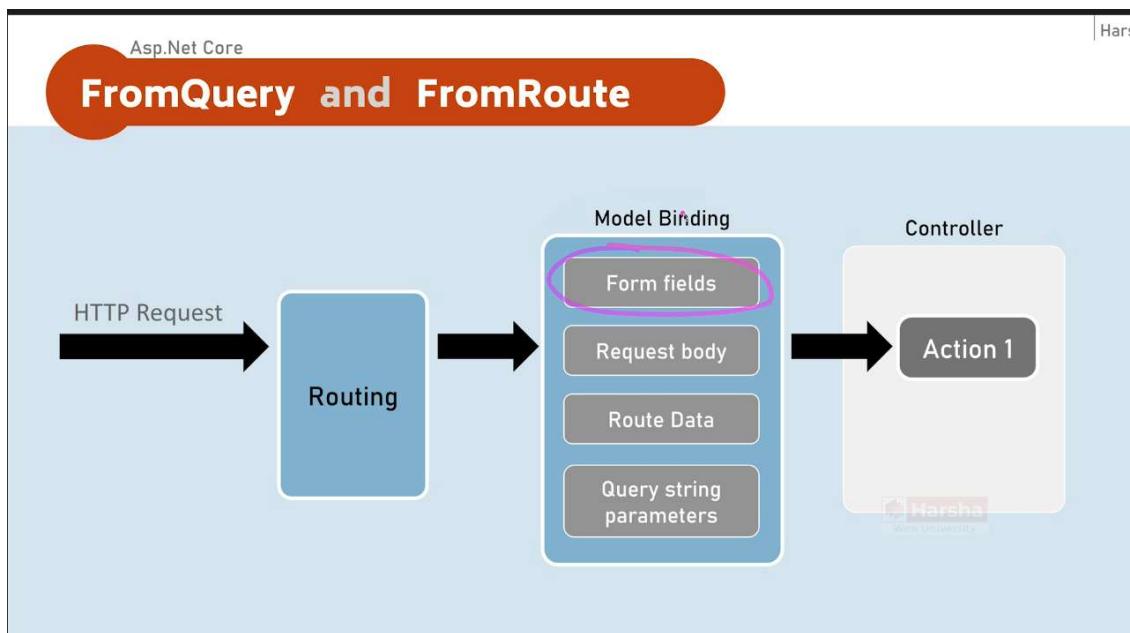
```
namespace IActionResultExample.Models
{
    public class Book
    {
        [FromQuery]
        public int? BookId { get; set; }

        [FromQuery]
        public string? Author { get; set; }

        public override string ToString()
        {
            return $"Book object - Book id: {BookId}, Author: {Author}";
        }
    }
}
```

Asp.Net Core

form-urlencoded and form-data



Harsha

form-urlencoded (default)	
Request Headers	Content-Type: application/x-www-form-urlencoded
Request Body	param1=value1¶m2=value2
form-data	
Request Headers	Content-Type: multipart/form-data
Request Body	-----d74496d66958873e Content-Disposition: form-data; name="param1" value1 -----d74496d66958873e Content-Disposition: form-data; name="param2" value2

In this lecture, we will talk about **form fields**, which have the first priority in the model binding process.

Generally, form fields are used when you click the submit button in an HTML form. For example, in a registration or login form with multiple text boxes like name and date of birth, after entering the values and clicking the submit button, all the entered values are submitted to the server as **form fields**.

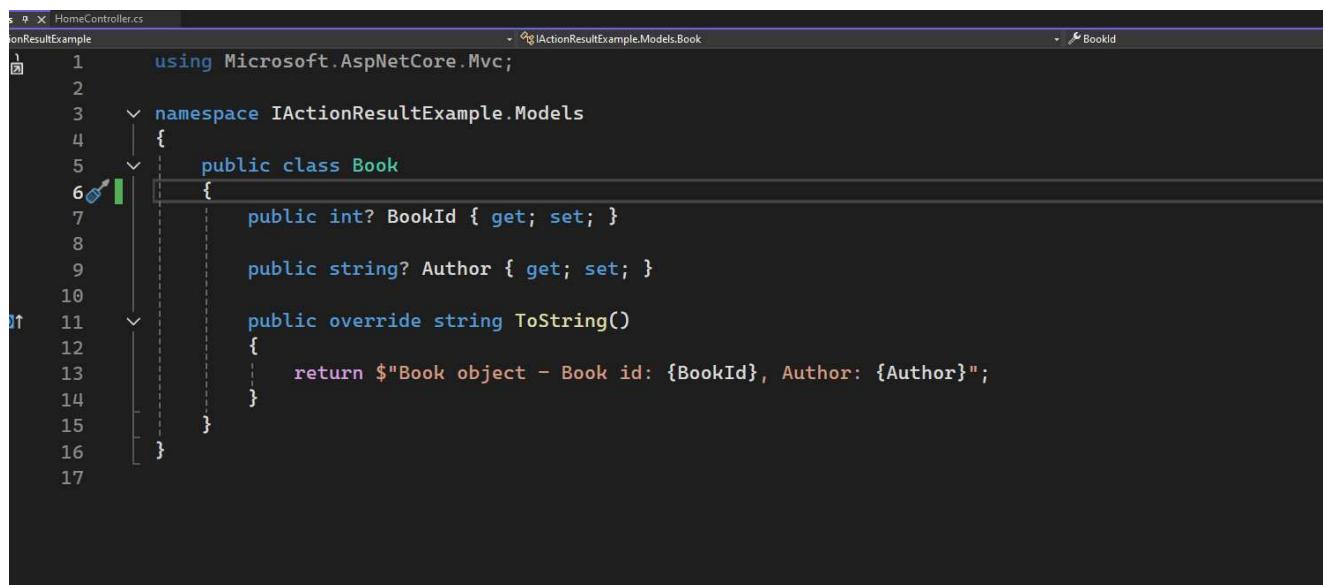
These values are typically added to the request body in one of two formats: either as application/x-www-form-urlencoded or as form data. The first format is simpler, while form data is more complex.

In the case of application/x-www-form-urlencoded, the request headers include a predefined, fixed content type: application/x-www-form-urlencoded. This format is essentially a query string added to the request body instead of the URL.

For GET requests, the query string is part of the URL,

but for POST requests, particularly with application/x-www-form-urlencoded, the same query string is added to the **request body**.

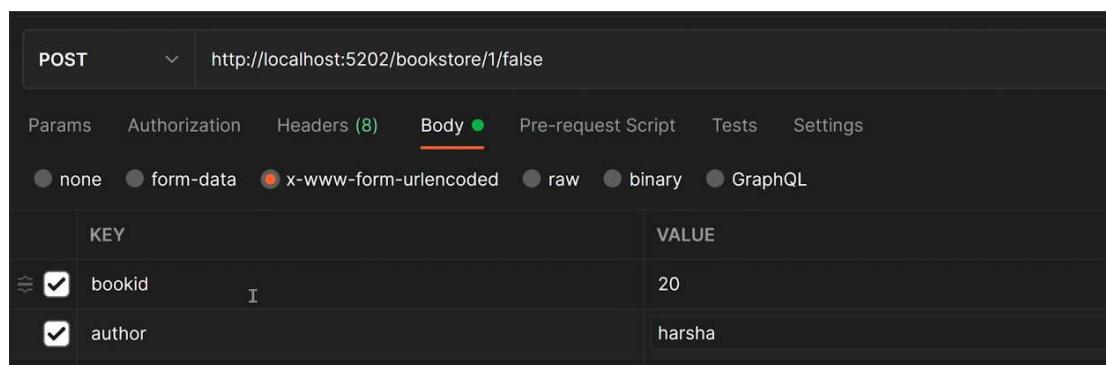
Form data, being more complex, will be discussed later in this lecture. For now, let's explore the simpler application/x-www-form-urlencoded format, which is the default in HTML.



```
HomeController.cs
onResultExample
    ↴ IActionResultExample.Models.Book
    ↴ BookId
1  using Microsoft.AspNetCore.Mvc;
2
3  namespace IActionResultExample.Models
4  {
5      public class Book
6      {
7          public int? BookId { get; set; }
8
9          public string? Author { get; set; }
10
11         public override string ToString()
12         {
13             return $"Book object - Book id: {BookId}, Author: {Author}";
14         }
15     }
16 }
17
```

go to postman.

these values are added in the request body in the query string format.



POST http://localhost:5202/bookstore/1/false

Params Authorization Headers (8) Body Pre-request Script Tests Settings

Body Type: x-www-form-urlencoded

KEY	VALUE
bookid	20
author	harsha



bookid	20
author	harsha
Key	Value

Harsha

Request Body | param1=value1¶m2=value2

content-type will be set automatically.

```
public IActionResult Index([FromQuery] int? bookid, [FromRoute] int? bookId, bool? isloggedin)
{
    if (bookid.HasValue == false)
        return BadRequest("Book id is not supplied");
    else
        return Ok(book);
}
```

The code shows two ways to supply a book ID: via a query parameter (`[FromQuery]`) or via a route parameter (`[FromRoute]`). The `bookid` variable is annotated with `[FromQuery]`, while `bookId` is annotated with `[FromRoute]`. The `book` variable is annotated with `[FromRoute]`.

The output shows the results of the execution:

- Author: harsha
- View: "harsha"
- BookId: 20

The `BookId` value is highlighted in yellow, indicating it was used over the query parameter.

here observe that, BookId is set to 20. We have also pass bookId as 1 from route parameter. But which one has more priority here?

as you can see, **From data!**

POST http://localhost:5202/bookstore/1/true

Body (8) Headers (8)

KEY	VALUE	DESCRIPTION
bookid	20	
author	harsha	
isLoggedIn	true	
Key	Value	Description

Status: 401 Unauthorized Time: 2.65s Size: 102 B Save Response

so this was form-url-encoded. now let's explore form-data.

form postman, select form-data then click on 'Send'

POST http://localhost:5202/bookstore/1/true

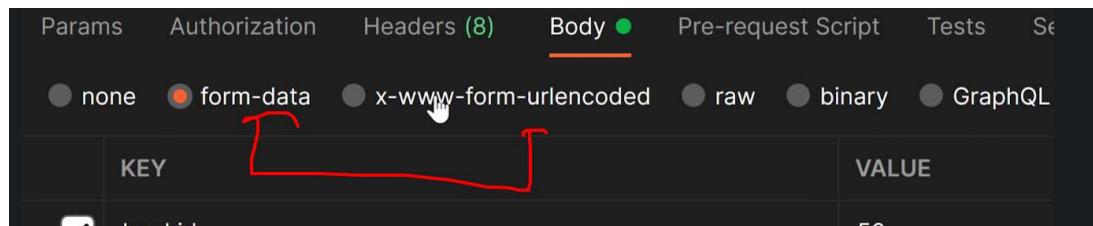
Body (8) Headers (4)

KEY	VALUE	DESCRIPTION
bookid	50	I
author	Scott	x
Key	Value	Description

Status: 200 OK Time: 95ms Size: 179 B Save Response

```
http://localhost:5202/bookstore/1/false. BookId=20&is
ferences
public IActionResult Index(int? bookid
:loggedin, Book book)
≤ 1,72,045ms elapsed
//Book id should be applied
if (bookid.HasValue == false)
{
    //return new BadRequestResult();
}
else
{
    book = _bookService.GetBook(bookid);
    return View(book);
}
```

so, what's the difference between both of these?



The image displays two side-by-side request configurations. The top section is titled 'form-urlencoded (default)' and the bottom section is titled 'form-data'. Both sections have 'Request Headers' and 'Request Body' tabs.

Request Headers:

- For 'form-urlencoded': Content-Type: application/x-www-form-urlencoded
- For 'form-data': Content-Type: multipart/form-data

Request Body:

- For 'form-urlencoded': param1=value1¶m2=value2
- For 'form-data':
 - Content-Disposition: form-data; name="param1"
 - value1

for simple information, let's say 5 or 6, form-urlencoded is more than enough. but for complex data and lengthy form, form-data works better.

If you want to attach file, then 'form-data' is your only option. lets say you want to upload photo to facebook.

Asp.Net Core Introduction to Model Validations

```

15     //return new BadRequestResult();
16     return BadRequest("Book id is not supplied or empty");
17 }
18
19 //Book id can't be less than or equal to 0
20 if (bookid <= 0)
21 {
22     return BadRequest("Book id can't be less than or equal to 0");
23 }
24
25 //Book id should be between 1 to 1000
26 if (bookid > 1000)
27 {
28     return NotFound("Book id can't be greater than 1000");
29 }
30

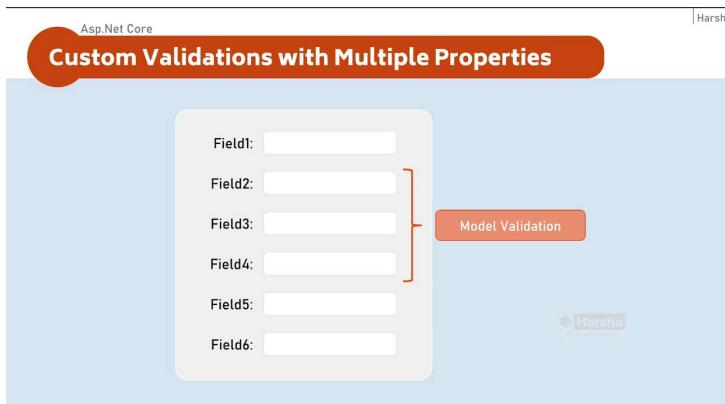
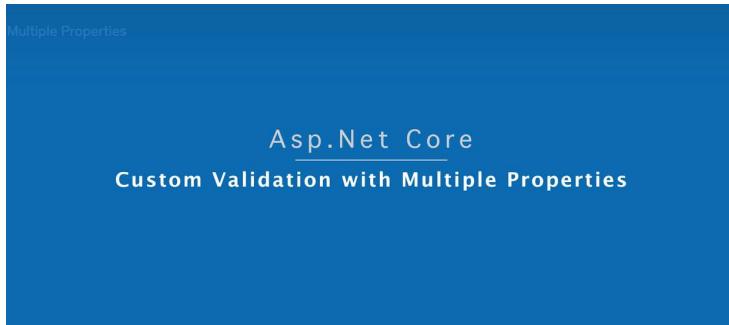
```

If we validate each and every field in real-world project. it will be very difficult. Check the amount of code that we have written for validations.

In case of model validations, you will be writing validation rules as attributes just before the properties.

The diagram illustrates the validation process across three main components:

- Model Validation:** A screenshot of a code editor showing a validation rule applied to a property in a class named `ClassName`. The rule is annotated with `[Attribute] //applies validation rule on this property`.
- Person Class:** A screenshot of a code editor showing the `Person` class definition. It includes several properties: `PersonName`, `Email`, `Phone`, `Password`, `ConfirmPassword`, and `Price`. The validation rule from the first screenshot is applied to the `PersonName` property.
- HomeController:** A screenshot of a code editor showing the `HomeController` class. It contains an `[HttpPost]` action method named `Register` which maps to the `/register` endpoint. The method uses `IActionResult` and `IActionResult<Person>` to handle the response.
- Postman Request:** A screenshot of the Postman application showing a POST request to `http://localhost:51666/register`. The request body is a JSON object containing the validation rules: `PersonName: "scott", Email: "scott@example.com", Phone: "123456", Password: "scott123", ConfirmPassword: "abc123", Price: 10`.



In the last lecture, we have learned how to validate a single property or single field. But sometimes, you have to validate multiple fields. For example, in the registration form, you have more than five text boxes, and you would like to validate two or more properties or fields. Then how do you apply custom validations on multiple fields or multiple properties? Let's try.

```

//we are going to create Person registration form. So we will add all kinds of form validation.
public class Person
{
    //Required[ErrorMessage = "{0} can't be empty or null"] // 0 represents the name of the property.
    [DisplayName("Person Name")]
    [StringLength(10, MinimumLength = 3, ErrorMessage = "{0} should be between {2} to {1} characters long")]
    [RegularExpression(@"[A-Za-z ]*", ErrorMessage = "{0} should contain only alphabet, space and dot(.)")]
    public string? PersonName { get; set; }

    [EmailAddress(ErrorMessage = "{0} should be a proper email address")]
    [Required(ErrorMessage = "{0} can't be blank")]
    public string? Email { get; set; }

    [Phone(ErrorMessage = "{0} should contain 10 digits")]
    //ValidateNever
    public string? Phone { get; set; }

    [Required(ErrorMessage = "{0} can't be blank")]
    public string? Password { get; set; }

    [Required(ErrorMessage = "{0} can't be blank")]
    [Compare("Password", ErrorMessage = "{0} and {1} does not match")]
    [DisplayName("Re-enter Password")]
    public string? ConfirmPassword { get; set; }

    [Range(0,999.99, ErrorMessage = "{0} should be between {1} to {2}")]
    public double? Price{ get; set; }

    //MinimumYearValidator(2005,ErrorMessage="Date of Birth should not be newer than Jan 01, {0}")]
    [MinimumYearValidator(2005)] //in case if the user doesn't supply the error message, then system should generate a default error message
    public DateTime? DateOfBirth{ get; set; }

    public DateTime? FromDate { get; set; }

    [DateRangeValidator("FromDate", ErrorMessage = "From Date should be older than or equal to 'To date'")]
    public DateTime? ToDate { get; set; }
}

```

Once the request is received, of course, routing executes first. Once the action method is detected based on the route, model binding picks up all the values from various data sources in the request, such as form data, route data, and query string parameters, etc. After model binding is completed, it automatically creates a new object of the model class, and that model object is represented as the object instance property here.

```

class DataRangeValidatorAttribute : ValidationAttribute
{
    public string OtherPropertyName { get; set; } // Value
    public DataRangeValidatorAttribute(string otherPropertyName)
    {
        OtherPropertyName = otherPropertyName;
    }
    protected override ValidationResult? IsValid(object? value, ValidationContext validationContext)
    {
        if (value == null)
        {
            //To do
            Datetime to_date = Convert.ToDateTime(value);
            // validationContext.ObjectType == Person Class (Model Class)
            PropertyInfo? otherProperty = validationContext.ObjectType.GetProperty(OtherPropertyName);
            //From Date
            if (otherProperty != null)
            {
                Datetime from_date = Convert.ToDateTime(otherProperty.GetValue(validationContext.ObjectInstance));
                if (from_date > to_date)
                {
                    return new ValidationResult(ErrorMessage ?? "From Date should be older than or equal to 'To date'");
                }
                else
                {
                    return ValidationResult.Success;
                }
            }
            else
            {
                return null; // null means no validation result
            }
        }
        return null;
    }
}

```

To test the custom validation attribute, we can use a tool like Postman. Here's a screenshot of the Postman interface showing a POST request to `http://localhost:5045/register`.

The request body contains the following data:

Key	Value
Email	mdkhaldalmahmud1010@gmail.com
Phone	85767
Password	123
ConfirmPassword	123
Price	60
DateOfBirth	01-01-1899
FromDate	01-08-2024
ToDate	02-08-2024

The response status is `400 Bad Request`, indicating that the validation failed. The error message is: `Person Name should contain only alphabet, space and dot (.)`.



Asp .Net Core
IValidatableObject

Model class

```

class ClassName : IValidatableObject
{
    //model properties here

    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        if (condition)
        {
            yield return new ValidationResult("error message");
        }
    }
}

```

Sometimes, you might need to create a model validation specific to a particular model class, which is not reusable. In such cases, you can implement the `IValidatableObject` interface in your model class. This allows you to write your custom validation logic directly within the same model class instead of writing it externally. Let me show you how.

For example, if your requirement is that **either the Date of Birth or the Age should be submitted** (but not both at the same time), you can handle this validation directly within the model class.

In this scenario:

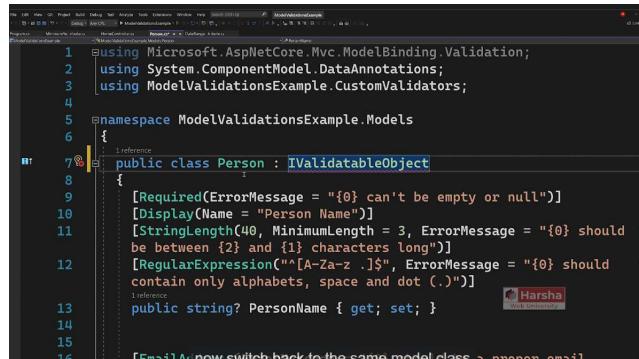
- If **Date of Birth** is provided, **Age** should be null.
- If **Age** is provided, **Date of Birth** should be null.

Instead of creating a separate custom validator class, you can implement this logic in the model class itself using the `IValidatableObject` interface. This approach is ideal when you don't need the validation logic to be reusable or when the requirement is relatively simple.

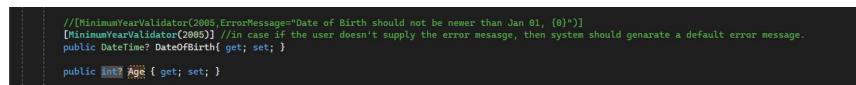
The `IValidatableObject` interface allows you to encapsulate validation logic within the model class, avoiding the complexity of reusable reflection-based validators. Here's how it works:

1. Implement the `IValidatableObject` interface in your model class.
2. Write your custom validation logic in the `Validate` method.
3. This method will automatically be called during the model validation process.

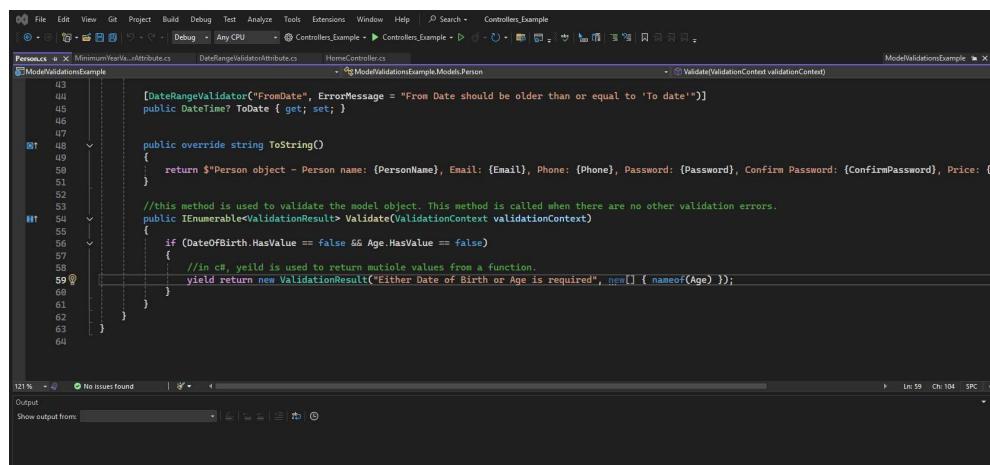
Let me show you how to implement it.



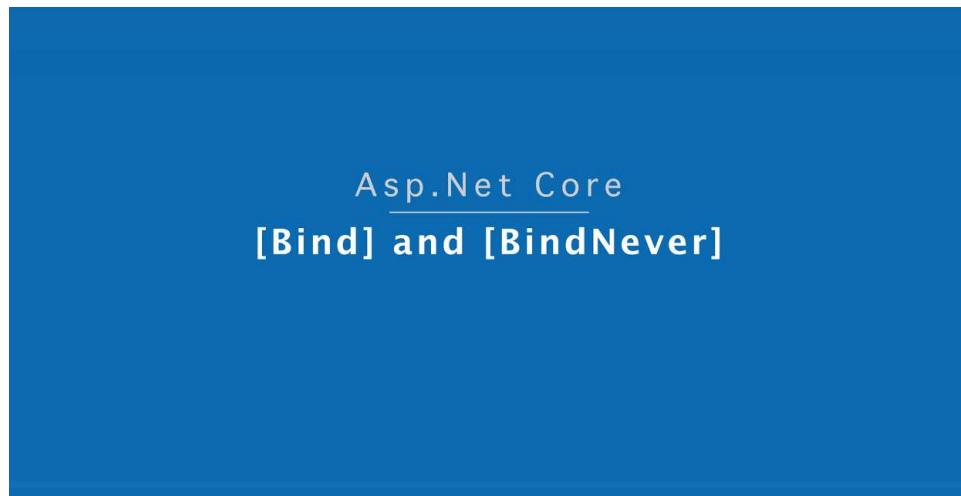
```
1  using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
2  using System.ComponentModel.DataAnnotations;
3  using ModelValidationsExample.CustomValidators;
4
5  namespace ModelValidationsExample.Models
6  {
7      public class Person : IValidatableObject
8      {
9          [Required(ErrorMessage = "{0} can't be empty or null")]
10         [Display(Name = "Person Name")]
11         [StringLength(40, MinimumLength = 3, ErrorMessage = "{0} should be between {2} and {1} characters long")]
12         [RegularExpression("[A-Za-z .]", ErrorMessage = "{0} should contain only alphabets, space and dot(.)")]
13         public string? PersonName { get; set; }
14
15         [Email]
16         public string? Email { get; set; }
17     }
18 }
```



```
1  //#[MinimumYearValidator(2005,ErrorMessage="Date of Birth should not be newer than Jan 01, {0}")]
2  //#[MinimumYearValidator(2005)] //in case if the user doesn't supply the error message, then system should generate a default error message.
3  public DateTime? DateOfBirth { get; set; }
4
5  public int? Age { get; set; }
```



```
1  public override string ToString()
2  {
3      return $"Person object - Person name: {PersonName}, Email: {Email}, Phone: {Phone}, Password: {Password}, Confirm Password: {ConfirmPassword}, Price: {Price}, Date of Birth: {DateOfBirth}, Age: {Age}";
4  }
5
6  //this method is used to validate the model object. This method is called when there are no other validation errors.
7  public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)
8  {
9      if (DateOfBirth.HasValue == false && Age.HasValue == false)
10      {
11          //in c#, yield is used to return multiple values from a function.
12          yield return new ValidationResult("Either Date of Birth or Age is required", new[] { nameof(Age) });
13      }
14  }
15 }
```



[Bind]

Controller

```
class ClassNameController : Controller
{
    public IActionResult ActionMethodName([Bind(
        nameof(ClassName.PropertyName), nameof(ClassName.PropertyName))]
        ClassName parameterName)
    {
    }
}
```

[Bind] attribute specifies that only the specified properties should be included in model binding.

Prevents over-posting (post values into unexpected properties) especially in 'Create' scenarios.

The bind attribute specifies what properties you would like to include in the model binding so that remaining properties will not be binded. See, by default in model binding, all the properties of the model class will be binded. But if you mention bind and specify the list of properties that you want to bind, only these properties will be included in the bind, and the remaining will be skipped.

The benefit of this bind attribute is that you can avoid overposting unwanted properties. For example, in the registration form, you have only 10 fields or properties to be submitted. But some hacker or malicious website has submitted more properties than expected. They may post sensitive or unwanted data. To avoid such issues, we use the bind attribute so that only the specified properties will be submitted, and the remaining properties, which are not mentioned in this bind list, will be skipped even if they are submitted in the request.

Let me show that is an example. Let's say in this person class you would like to allow only person name, email, password, and confirm password. Remaining properties should not be binded. Assume that.

```
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
```

```
[Phone(ErrorMessage = "{0} should contain 10 digits")]
///[ValidateNever]
public string? Phone { get; set; }

[Required(ErrorMessage = "{0} can't be blank")]
public string? Password { get; set; }

[Required(ErrorMessage = "{0} can't be blank")]
[Compare("Password", ErrorMessage = "{0} and {1} do not match")]
[Display(Name = "Re-enter Password")]
public string? ConfirmPassword { get; set; }
```

So, in your home controller, in this action method, you can add the bind attribute near this parameter and then specify the list of properties that you want to accept. You can either mention the list of properties with comma separation like this, or, in order to get it dynamically, you can use the nameof operator.

```

1  using ModelValidationsExample.Models;
2
3  namespace ModelValidationsExample.Controllers
4  {
5      public class HomeController : Controller
6      {
7          [Route("register")]
8          public IActionResult Index([Bind(nameof(Person.PersonName))])
9          Person person
10         {
11             if (!ModelState.IsValid)
12             {
13                 //get error messages from model state
14                 string errors = string.Join("\n",
15                     ModelState.Values.SelectMany(value => value.Errors).Select
16                     (err => err.ErrorMessage));
17
18                 return BadRequest(errors);
19             }
20         }
21     }
22 }

```

```

1  using ModelValidationsExample.Models;
2
3  namespace ModelValidationsExample.Controllers
4  {
5      public class HomeController : Controller
6      {
7          [Route("register")]
8          public IActionResult Index([Bind(nameof(Person.PersonName),
9              nameof(Person.Email), nameof(Person.Password), nameof
10             (Person.ConfirmPassword))] Person person)
11         {
12             if (!ModelState.IsValid)
13             {
14                 //get error messages from model state
15                 string errors = string.Join("\n",
16                     ModelState.Values.SelectMany(value => value.Errors).Select
17                     (err => err.ErrorMessage));
18             }
19         }
20     }
21 }

```

Now, in the postman , we are submitting all the values.

The screenshot shows the Postman interface with a GET request to `http://localhost:5166/register`. The 'Body' tab is selected, showing a form-data structure with the following fields:

KEY	VALUE	DESCRIPTION
PersonName	abcd	
Email	scott@example.com	
Phone	123456	
Password	scott123	
ConfirmPassword	scott123	
Price	100	

```

1  result Index([Bind(nameof(Person.Person
2  mail), nameof(Person.Password), nameof
3  Password))] Person person)
4
5  if (!ModelState.IsValid)
6
7  messages from model state

```

Alternatively, you can skip one or two property. Rest of the field will be binded.

```

39
40
41 // [MinimumYearValidator(2005, ErrorMessage = "Date of Birth
42 should not be newer than Jan 01, {0}")]
43 [MinimumYearValidator(2005)]
44 [BindNever]
45
46
47 public DateTime? DateOfBirth { get; set; }
48
49 [DateRangeValidator("FromDate", ErrorMessage = "'From Date'
should be older than or equal to 'To Date'")]
50
51 public DateTime? ToDate { get; set; }
52
53 public int? Age { get; set; }

```

```

5
6 public class HomeController : Controller
7 {
8     [Route("register")]
9     // [Bind(nameof(Person.PersonName), nameof(Person.Email), nameof(
10 Person.Password), nameof(Person.ConfirmPassword))]
11
12     public IActionResult Index(Person person)
13     {
14         if (!ModelState.IsValid)
15         {
16             // get error messages from model state
17             string errors = string.Join("\n",
18             ModelState.Values.SelectMany(value => value.Errors).Select
19             (err => err.ErrorMessage));

```

The screenshot shows a POST request to `http://localhost:5166/register`. The body is set to `x-www-form-urlencoded` and contains the following parameters:

- `ConfirmPassword`: scottt23
- `Price`: 100
- `DateOfBirth`: 1990-06-01
- `FromDate`: 2010-01-01
- `ToDate`: 2011-01-01
- `Age`: 25

The response status is 200 OK, and the response body is: `Person object - Person name: abcd, Email: scott@example.com, Phone: 123456, Password: scottt23, Confirm Password: scottt23, Price: 100`.

It contains all the value except DateOfBirth.

```

c class HomeController : Controller
{
    [Route("register")]
    Bind(nameof(Person.PersonName), nameof(Person.Email), nameof(Person.Password), nameof(Person.ConfirmPassword))]
    public IActionResult Index(Person person)
    {
        if (!ModelState.IsValid)
        {
            // get error messages from model state
            string errors = string.Join("\n",
            ModelState.Values.SelectMany(value => value.Errors).Select
            (err => err.ErrorMessage));

```

A tooltip is visible over the `ModelState.IsValid` check, stating: `it contains all the values except date`.

Asp.Net Core | Harsha

[BindNever]

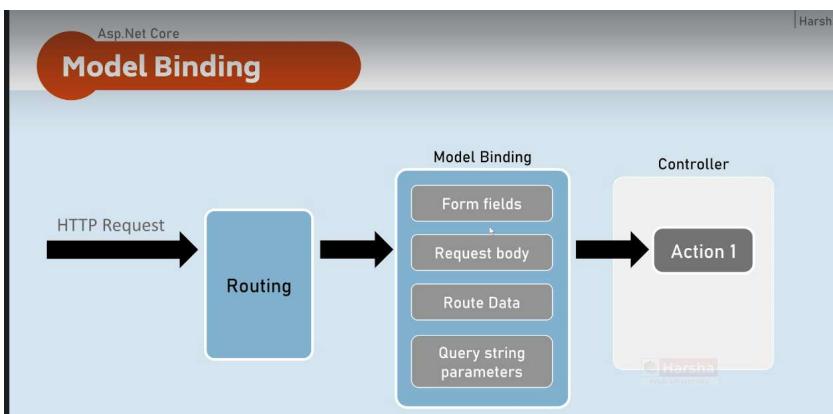
```

Model
class ModelClassName
{
    [BindNever]
    public type PropertyName { get; set; }
}

```

[BindNever] attribute specifies that the specified property should NOT be included in model binding.

Asp.Net Core
[FromBody]



Asp.Net Core | Harsha

FromBody

[FromBody]

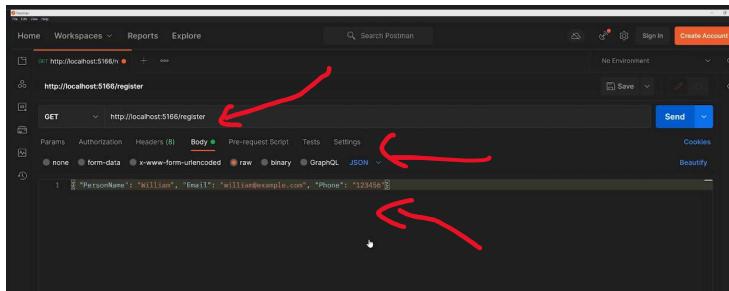
```

//enables the input formatters to read data from request body (as JSON or XML or custom) only
public IActionResult ActionMethodName([FromBody] type parameter)
{
}

```

So far, in model binding, we have seen the form fields, route data, and query string parameters. Form fields are of two types: either form URL-encoded or form data. We have already seen that in the earlier lectures in the same section. But there are some other content types other than form data and form URL-encoded. For example, you have application/json, application/csv (comma-separated values), or application/xml. For such types of content types, you have to consider using the **request body**. Of course, the content types such as form URL-encoded and form data are also sent as a part of the request body, but those are not called request body model binding. They are treated as form fields because, by default, HTML forms use either form URL-encoded or multipart form data. So, those are called form fields. Here, request body model binding refers to application/json, application/xml, or a custom format. When you want to receive JSON data or XML data, mainly, you have to use the FromBody attribute for your model parameter in the action method. This enables the input formatters to parse the request body into the specified model object. For example, JSON data or XML data can

be converted into the model object.

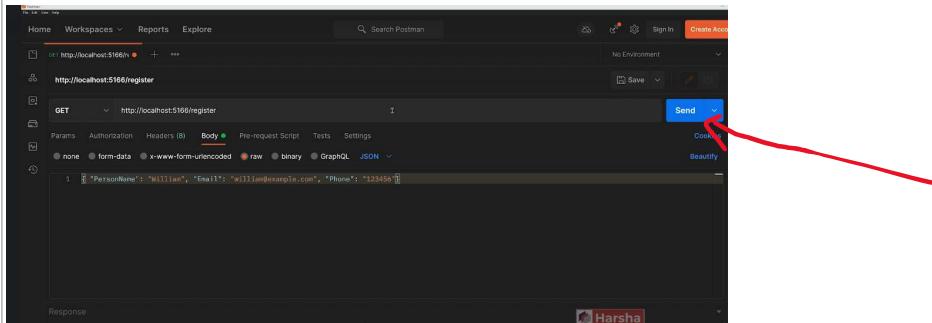


But we have got all the fields null!!! But why?

```
[Bind(nameof(Person.PersonName), nameof(Person.Password), nameof(Person.ConfirmPassword))]  
public IActionResult Index(Person person)  
{  
    if (!ModelState.IsValid)  
  
        //get error messages from model state  
        string errors = string.Join("\n",  
            ModelState.Values.SelectMany(value => value.Errors).Select  
            (err => err.ErrorMessage));  
  
    return BadRequest(errors);  
}
```

We have to enable 'input-formatters' by using *form body*. So that instead of regular model binding, it enables the json input formatter to read the content from the request body and make it as an object of the model class.

```
public class HomeController : Controller  
{  
    [Route("register")]  
    //Bind(nameof(Person.PersonName), nameof(Person.Email), nameof(  
    //Person.Password), nameof(Person.ConfirmPassword))  
    public IActionResult Index([FromBody] Person person)  
    {  
        if (!ModelState.IsValid)  
        {  
            //get error messages from model state  
            string errors = string.Join("\n",  
                ModelState.Values.SelectMany(value => value.Errors).Select  
                (err => err.ErrorMessage));  
  
            return BadRequest(errors);  
        }  
    }  
}
```



```

ter")]
f(Person.PersonName), nameof(Person.Email),
ord), nameof(Person.ConfirmPassword))]

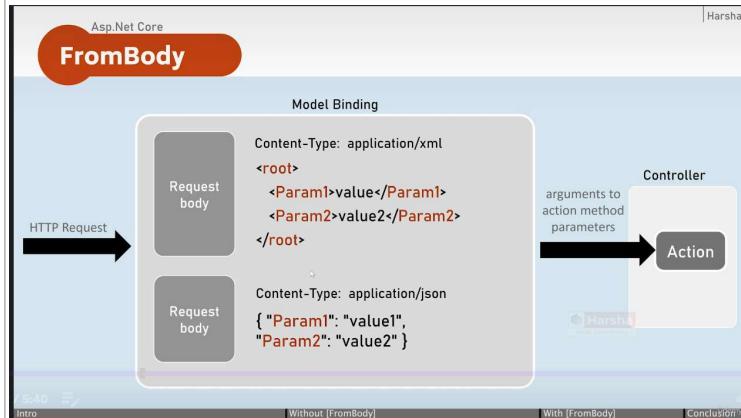
nResult Index([FromBody] Person person)

tate.IsValid)

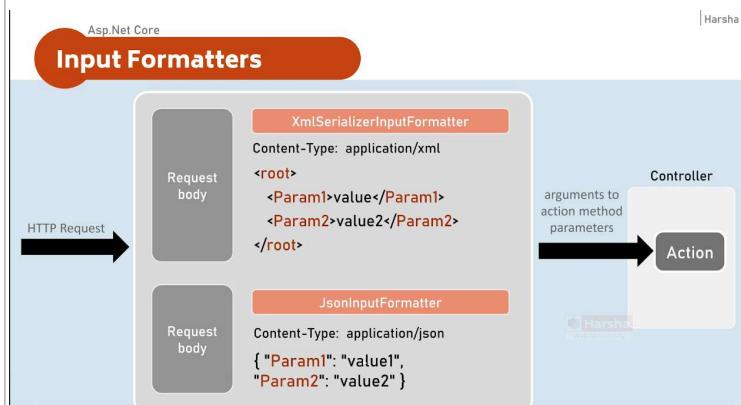
or messages from model state
rors = string.Join("\n",
e.Values.SelectMany(value => value.Errors))

```

But for some older application, you may need to send the data as XML instead of JSON.



Asp.Net Core Input Formatters



Input formatters are the internal classes in ASP.NET Core used to transform or convert the request body into the

model object.

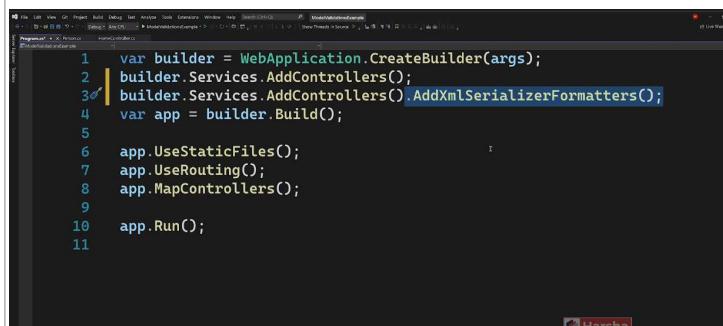
For example, whenever the request body contains JSON data and the content type is application/json in the request headers, the JSON input formatter will be enabled automatically. It reads the JSON data from the request body and converts it into a model object. For instance, it can read the JSON data and convert it into an object of the Person model class in our example.

Similarly, if the request body contains XML data and the content type is application/xml in the request headers, the XML serializer input formatter will be enabled automatically. It reads the XML data and converts it into a model object.

By default, ASP.NET Core controllers have only one input formatter, which is the JSON input formatter. That's why, without manually adding the JSON input formatter to the controller, it can read JSON data from the request body by default.

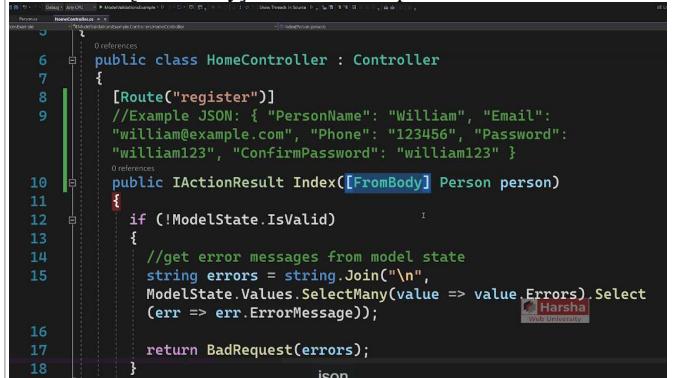
However, if you want to work with XML (i.e., read XML data from the request body), you must manually add the XML serializer input formatter to the controllers. Let us see how to do this in this lecture.

1. We have added XML Serializer Formatter also to the Controllers.

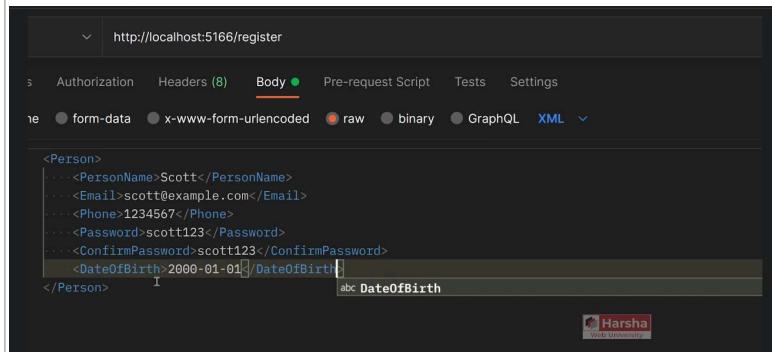


```
1 var builder = WebApplication.CreateBuilder(args);
2 builder.Services.AddControllers();
3 builder.Services.AddControllers().AddXmlSerializerFormatters();
4 var app = builder.Build();
5
6 app.UseStaticFiles();
7 app.UseRouting();
8 app.MapControllers();
9
10 app.Run();
11
```

2. Now our [FromBody] can read the request from either JSON or XML



```
6 public class HomeController : Controller
7 {
8     [Route("register")]
9     //Example JSON: { "PersonName": "William", "Email":
10     //william@example.com", "Phone": "123456", "Password":
11     "william123", "ConfirmPassword": "william123" }
12     public IActionResult Index([FromBody] Person person)
13     {
14         if (!ModelState.IsValid)
15         {
16             //get error messages from model state
17             string errors = string.Join("\n",
18                 ModelState.Values.SelectMany(value => value.Errors).Select
19                 (err => err.ErrorMessage));
20
21             return BadRequest(errors);
22         }
23     }
24 }
```



The screenshot shows a Postman request to `http://localhost:5166/register`. The `Body` tab is selected, and the `XML` radio button is chosen. The XML payload is:

```
<Person>
  <PersonName>Scott</PersonName>
  <Email>scott@example.com</Email>
  <Phone>1234567</Phone>
  <Password>scott123</Password>
  <ConfirmPassword>scott123</ConfirmPassword>
  <DateOfBirth>2000-01-01</DateOfBirth>
</Person>
```

```

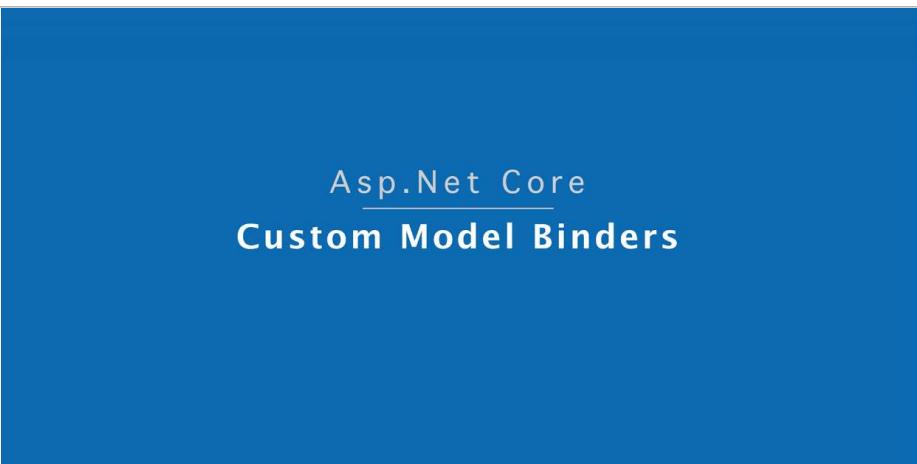
    ister")]
    SON: { "PersonName": "William", "Email": "example.com", "Phone": "123456", "Password": "", "ConfirmPassword": "william123" }

    ionResult Index([FromBody] Person person)

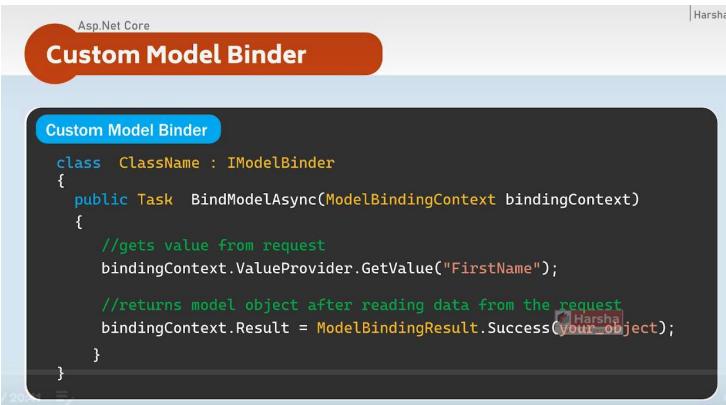
    lState.IsValid)

    rror messages from model state
    errors = string.Join("\n",
    ate.Values.SelectMany(o => o.Value.Errors).

```



Asp.Net Core Custom Model Binders



Whenever you would like to execute some complex logic while model binding, you have to create a custom model binder. This means instead of using the default model binding functionality, you are going to create your own model binding, and you can reuse the same across multiple places in the entire project.

For example, let's say whenever the user has submitted the first name and last name in the request, you would like to concatenate them as a string value, that is, into the person name property. Otherwise, if the user has submitted year, month, and date independently, you have to combine them together to form a date value.

So, for this type of complex operation, you have to create a custom model binder in real-world projects. In other words, whenever you would like to perform a complex manipulation on data types other than the built-in data types, such as custom data types, structures, or enumerations, then you have to create a custom model binder.

For another example, let's say there is a string value submitted as a part of the request, like bank account type = savings account. Based on that string value, you have to generate the corresponding enumeration value (enum value). So, for this type of operation, the default model binding doesn't work. You have to create your own, and that is called a custom model binder.

In this case, you have to create your custom model binder class that implements the `IModelBinder` interface. This means it is recognized as a custom model binder. But this interface forces you to create the `BindModelAsync` method in which you can write your actual model binding functionality.

To get a specific value, you can use the `ValueProvider.GetValue` method of the binding context, and you can assign them to the respective property of your model class. This means you have to create your own object of the model class manually and assign the value to each property one by one using this value provider. This process is almost like the default functionality in inbuilt model binding but done programmatically here.

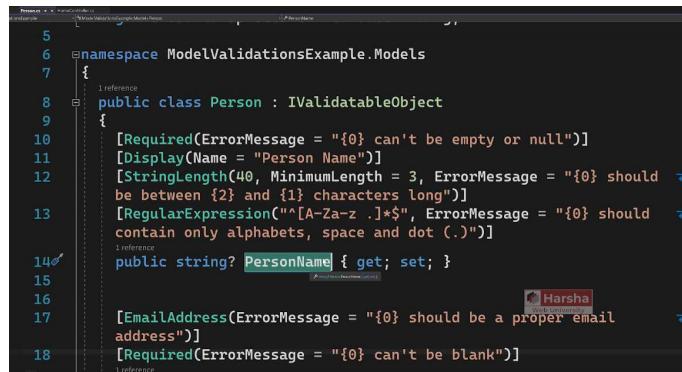
The benefit is you have complete control to decide what value should be assigned to what property instead of just depending on the property name.

For another example, let's say there is an address, and the user submits street name and city name independently. Suppose the user submits the street and city names with a comma separator. For example, ABC Street, PQR City. Based on the comma separator, you have to split it into multiple pieces. The first value should be assigned to the `Street` property, and the second value to the `City` property.

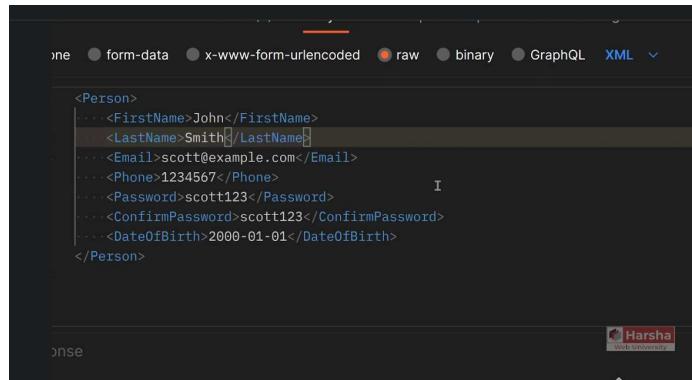
For this type of complex or custom logic, custom model binders are best.

In real-world project, it is rare to use custom-model binding.

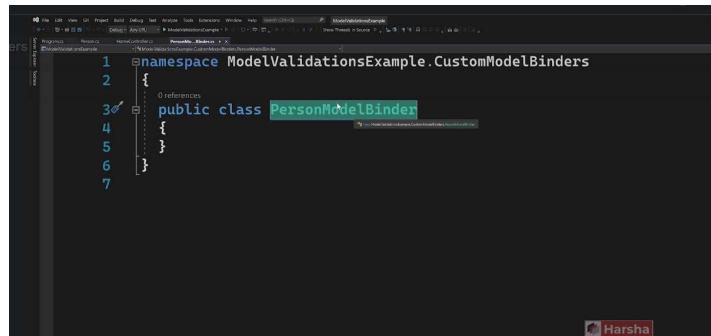
Here in our project, let's say user submits `firstName` and `lastName`, both of these should be joined and push to '`PersonName`'.



```
5
6  namespace ModelValidationsExample.Models
7  {
8      public class Person : IValidatableObject
9      {
10          [Required(ErrorMessage = "{0} can't be empty or null")]
11          [Display(Name = "Person Name")]
12          [StringLength(40, MinimumLength = 3, ErrorMessage = "{0} should be between {2} and {1} characters long")]
13          [RegularExpression("[a-zA-Z .]*$", ErrorMessage = "{0} should contain only alphabets, space and dot (. )")]
14          public string? PersonName { get; set; }
15
16
17          [EmailAddress(ErrorMessage = "{0} should be a proper email address")]
18          [Required(ErrorMessage = "{0} can't be blank")]
19      }
20  }
```



Lets's see how you can solve this problem using custom-model binding.



```
1  namespace ModelValidationsExample.CustomModelBinders
2  {
3      public class PersonModelBinder
4      {
5      }
6  }
```

But this class will not be invoked at the time of model binding. You have to set it explicitly.

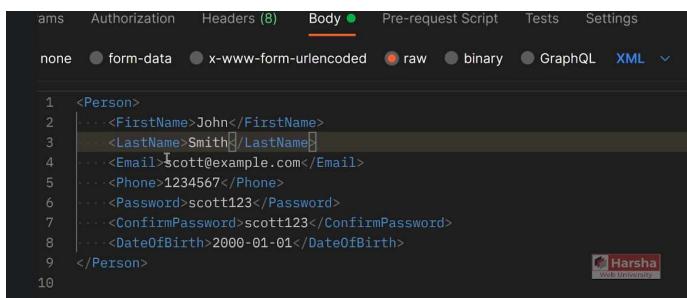
At the home controller, add a new attribute.

```

1 using Microsoft.AspNetCore.Mvc;
2 using ModelValidationsExample.Models;
3 using ModelValidationsExample.CustomModelBinders;
4
5 namespace ModelValidationsExample.Controllers
6 {
7     public class HomeController : Controller
8     {
9         [Route("register")]
10        //Example JSON: { "PersonName": "William", "Email": "william@example.com", "Phone": "123456", "Password": "william123", "ConfirmPassword": "william123" }
11        public IActionResult Index([FromBody] [ModelBinder(BinderType = typeof(PersonModelBinder))] Person person)
12        {
13            if (!ModelState.IsValid)
14            {
15                //get error messages from model state

```

There might be duplicate 'FirstName' and 'LastName' exists in the request.



I am saying here is that, if duplicate value is exist, I am only interested in only first value.

```

1 using Microsoft.AspNetCore.Mvc.ModelBinding;
2
3 namespace ModelValidationsExample.CustomModelBinders
4 {
5     public class PersonModelBinder : IModelBinder
6     {
7         public Task BindModelAsync(ModelBindingContext bindingContext)
8         {
9             //FirstName and LastName
10            if (bindingContext.ValueProvider.GetValue("FirstName").Count > 0)
11            {
12                bindingContext.ValueProvider.GetValue("FirstName").FirstValue
13            }
14        }
15    }

```

```

1 using Microsoft.AspNetCore.Mvc.ModelBinding;
2 using ModelValidationsExample.Models;
3
4 namespace ModelValidationsExample.CustomModelBinders
5 {
6     public class PersonModelBinder : IModelBinder
7     {
8         public Task BindModelAsync(ModelBindingContext bindingContext)
9         {
10             Person person = new Person();
11             //FirstName and LastName
12             //bindingContext.ValueProvider // Hey Value Provider give me the value of the key FirstName, so if the key is present in the request, it will return FirstName
13             if(bindingContext.ValueProvider.GetValue("FirstName").Length > 0)
14             {
15                 person.PersonName = bindingContext.ValueProvider.GetValue("FirstName").FirstValue;
16
17                 if (bindingContext.ValueProvider.GetValue("LastName").Length > 0)
18                 {
19                     person.PersonName += " " + bindingContext.ValueProvider.GetValue("LastName").FirstValue;
20                 }
21             }
22
23             //now, you have to provide the result in the 'BindingContext'
24             bindingContext.Result = ModelBindingResult.Success(person);
25
26             return Task.CompletedTask; // we have to return a task because return type of this method is Task
27         }
28     }
29 }

```

Now run this application. From Postman, click on Send button.

The remaining values are NULL since we have not assigned them in custom model binder.c

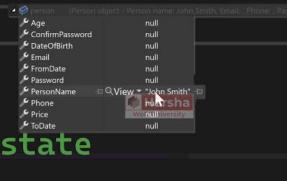
```

    "PersonName": "William", "Email": "william@example.com", "Phone": "123456", "Password": "ConfirmPassword": "william123" }

    [BindResult Index([FromBody] [ModelBinder(BinderType = typeof(PersonModelBinder))]) Person person)

    if (ModelState.IsValid)
    {
        // or messages from model state
        string errors = ModelState

```



Asp.Net Core Model Binder Providers

Suppose you would like to use the same custom model binder for all the action methods wherever this type of model class is used. Then you can declare it globally by using a binder provider.

You can create a custom model binder provider like this, and you can return the type of the model binder class that you have already created. This will enable it globally for the entire project, wherever the particular type of model class is used as an action method parameter.

Asp.Net Core Harsha

Custom Model Binder Provider

```

class ClassName : IModelBinderProvider
{
    public IModelBinder GetBinder(ModelBinderProviderContext providerContext)
    {
        //returns type of custom model binder class to be invoked
        return new BinderTypeModelBinder(typeof(YourModelBinderClassName));
    }
}

```

That means any action method that has a parameter of the Person type would automatically use the specified custom model binder. This eliminates the need to explicitly define the model binder in every action method, which is the actual benefit. Of course, this approach might not be very useful for simple or smaller projects. However, for larger projects, the benefit is significant as you don't have to repeatedly specify the model binder for every action method.

```
9 [Route("register")]
10 //Example JSON: { "PersonName": "William", "Email":
11 //                "william@example.com", "Phone": "123456", "Password":
12 //                "william123", "ConfirmPassword": "william123" }
13
14 public IActionResult Index([FromBody] [ModelBinder(BinderType =
15 typeof(PersonModelBinder))] Person person)
16 {
17     if (!ModelState.IsValid)
18     {
19         //get error messages from model state
20         string errors = string.Join("\n",
21             ModelState.Values.SelectMany(value => value.Errors).Select
22             (err => err.ErrorMessage));
23
24         return BadRequest(errors);
25     }
26
27     return Content($"{person}");
28 }
```

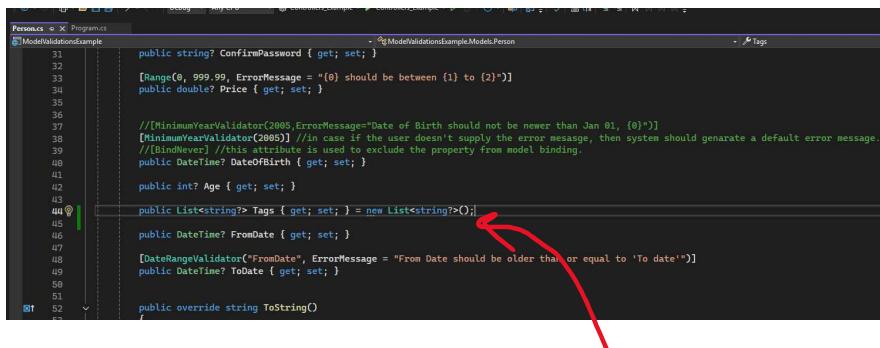
```
2 using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
3 using ModelValidationsExample.Models;
4
5 namespace ModelValidationsExample.CustomModelBinders
6 {
7     public class PersonBinderProvider : IModelBinderProvider
8     {
9         public IModelBinder? GetBinder(
10             (ModelBinderProviderContext context)
11         {
12             if (context.Metadata.ModelType == typeof(Person))
13             {
14                 return new BinderTypeModelBinder(typeof(
15                     PersonModelBinder));
16             }
17         }
18     }
19 }
```

```
9 [Route("register")]
10 //Example JSON: { "PersonName": "William", "Email":
11 //                "william@example.com", "Phone": "123456",
12 //                "Password": "william123", "ConfirmPassword": "william123" }
13
14 [ModelBinder(BinderType = typeof(
15     PersonModelBinder))]
16 public IActionResult Index(Person person)
17 {
18     if (!ModelState.IsValid)
19     {
20         //get error messages from model state
21         string errors = string.Join("\n",
22             ModelState.Values.SelectMany(value => value.Errors).Select
23             (err => err.ErrorMessage));
24
25         return BadRequest(errors);
26     }
27
28     return Content($"{person}");
29 }
```



Postman interface showing a 'Body' tab with 'form-data' selected. The 'Tags' key has two values: 'one' and 'two'. Other fields include FirstName (John), LastName (Smith), Email (scott@example.com), Phone (1234567890), Password (scott), and ConfirmPassword (scott123).

Sometimes, you might want to send more than one value for the same key. For example, let's say there is a **phone** field, and you want to send more than one phone number. In such a case, how can you receive those values in the model class? Let's try.



In this way, we can submit more than one value

Postman interface showing a 'Body' tab with 'form-data' selected. The 'Tags' key has two values: 'one' and 'two'. Other fields include DateOfBirth (1990-06-01), FromDate (2010-01-01), ToDate (2011-01-01), Age (25), and a file named '#dotnet.py'.

Browser developer tools Network tab showing a JSON response:

```

JSON: { "PersonName": "William", "Email": "example.com", "Phone": "123456", "Password": "23", "ConfirmPassword": "william123" }

```

Code snippet showing binder logic:

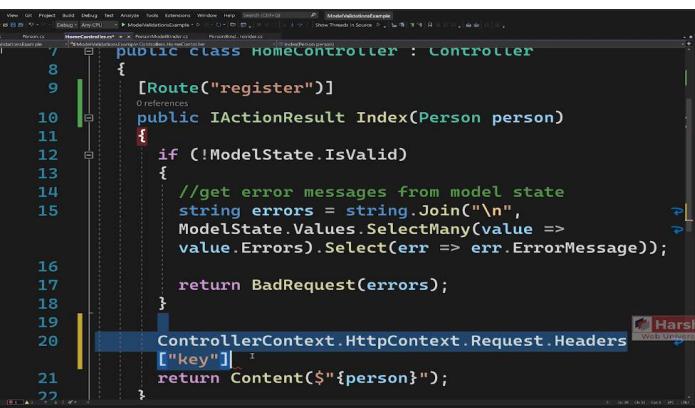
```

if (BinderType = typeof(PersonModelBinder))
    ActionResult Index(Person person)
else if (ModelState.IsValid)
    error messages from model
    string errors = string.Join("\n", ModelState.Values.SelectMany(v => v.Errors));
    collection that is a list of strings

```

Asp.Net Core [FromHeader]

Sometimes, you may want to read the request headers as part of the model binding.
As of now, we can read inputs from the request, such as form fields, request body (JSON or XML), route data, or query strings.
Lastly, you can also receive values from the request headers.



A screenshot of the Visual Studio IDE showing the code for a HomeController. The code uses the traditional approach to read request headers:

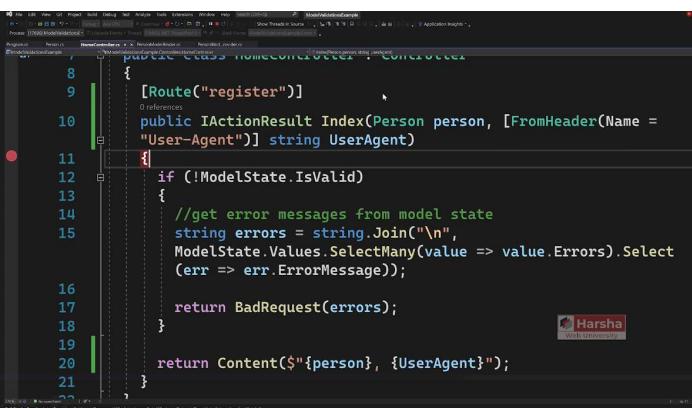
```
public class HomeController : Controller
{
    [Route("register")]
    public IActionResult Index(Person person)
    {
        if (!ModelState.IsValid)
        {
            //get error messages from model state
            string errors = string.Join("\n",
                ModelState.Values.SelectMany(value =>
                    value.Errors).Select(err => err.ErrorMessage));
        }
        return BadRequest(errors);
    }
    ControllerContext.HttpContext.Request.Headers["key"]
    return Content($"{person}");
}
```

The traditional way to read the request headers is to write:

ControllerContext.HttpContext.Request.Headers

Here, you can specify the header key name to retrieve its value. This is how you can retrieve the value from request headers in traditional code.

But instead of this lengthy approach.



A screenshot of the Visual Studio IDE showing the same HomeController code, but using the [FromHeader] attribute to read the User-Agent header:

```
public class HomeController : Controller
{
    [Route("register")]
    public IActionResult Index(Person person, [FromHeader(Name =
    "User-Agent")] string userAgent)
    {
        if (!ModelState.IsValid)
        {
            //get error messages from model state
            string errors = string.Join("\n",
                ModelState.Values.SelectMany(value => value.Errors).Select
                (err => err.ErrorMessage));
        }
        return BadRequest(errors);
    }
    return Content($"{person}, {userAgent}");
}
```

Now run on postman.

The screenshot shows the Postman application interface. A GET request is being made to `http://localhost:5166/register`. The 'Body' tab is selected, showing the following parameters:

- DateOfBirth: 1990-06-01
- FromDate: 2010-01-01
- ToDate: 2011-01-01
- Age: 25
- Tags[0]: #dotnet
- Tags[1]: #python

