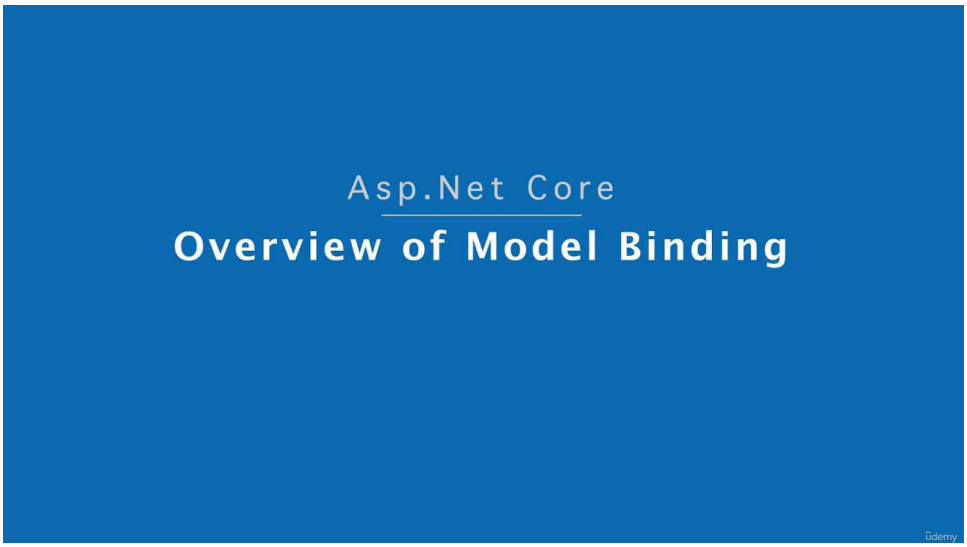
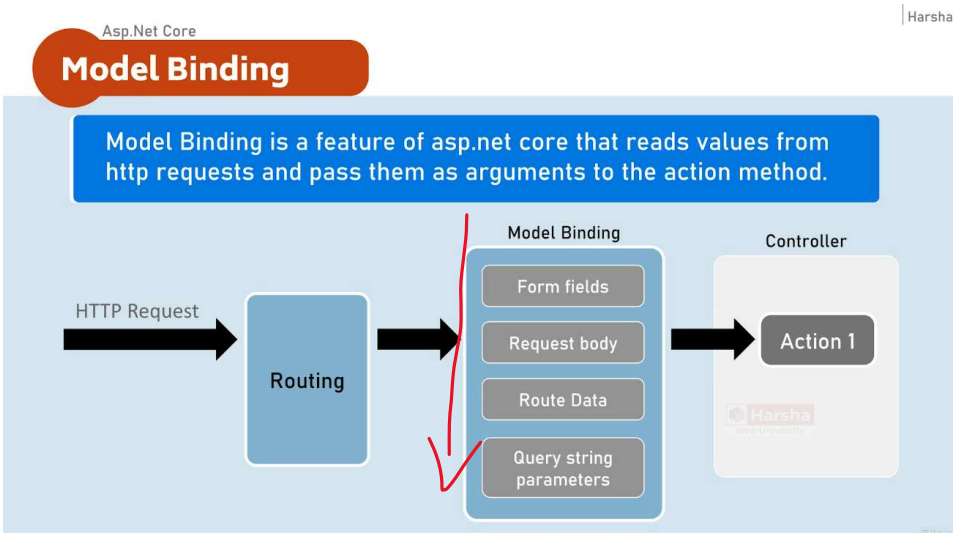
Section 7: Model Binding and Validations [MVC and Web API]

03 January 2025 19:34



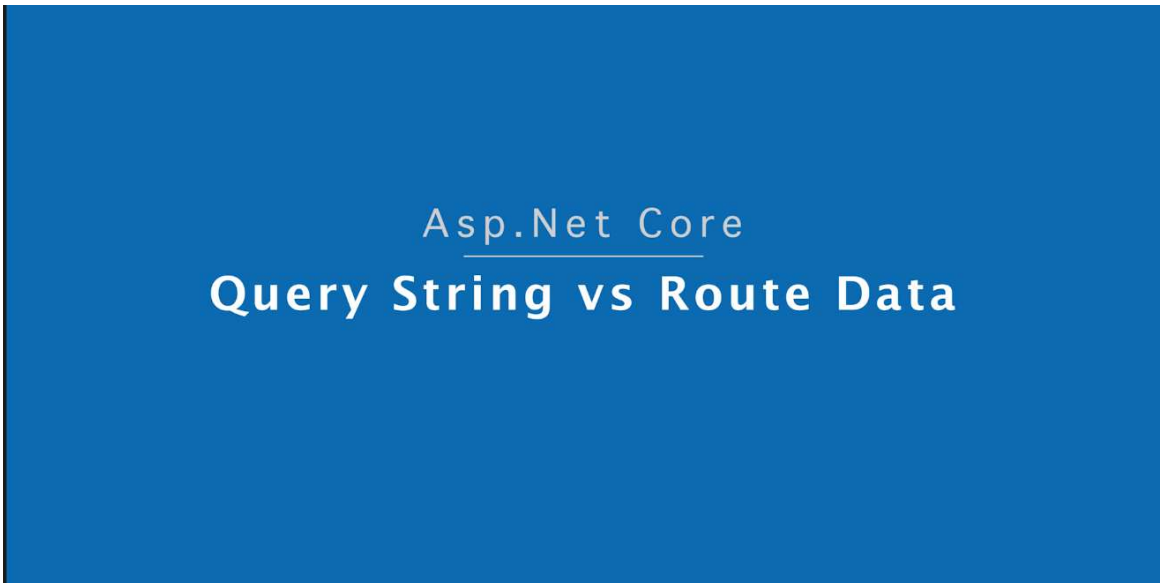


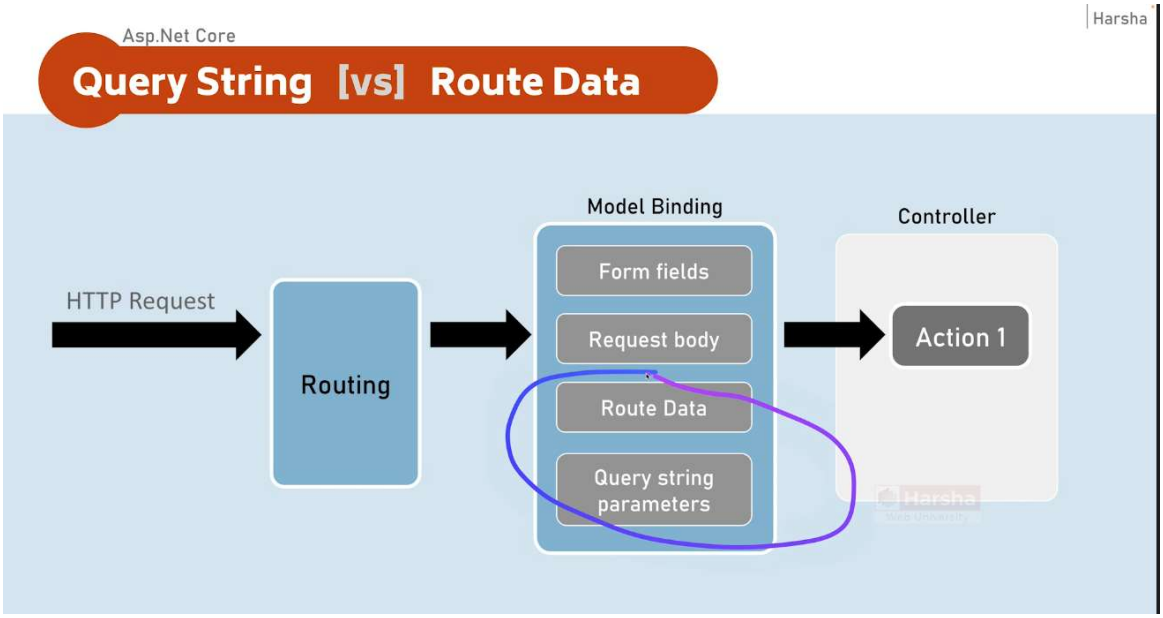
ASP.NET Core web applications require the input data, but from where it fetches the input data that is from the request. The request may send the data in various formats in different ways. For example, the request may contain the input data in the form of request headers, request query string parameters, 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.

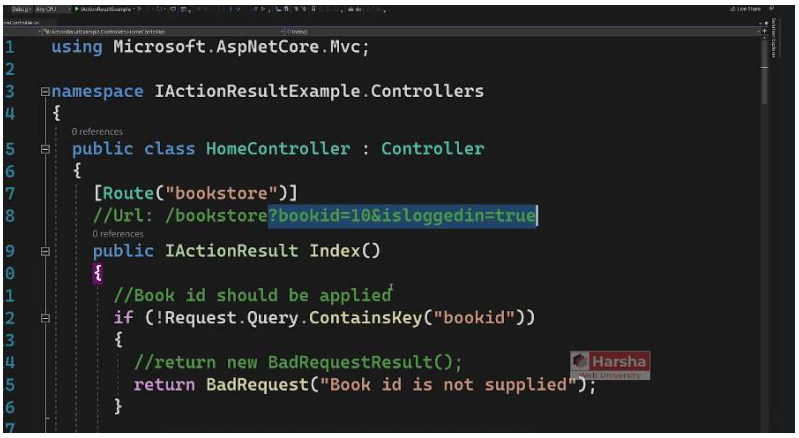




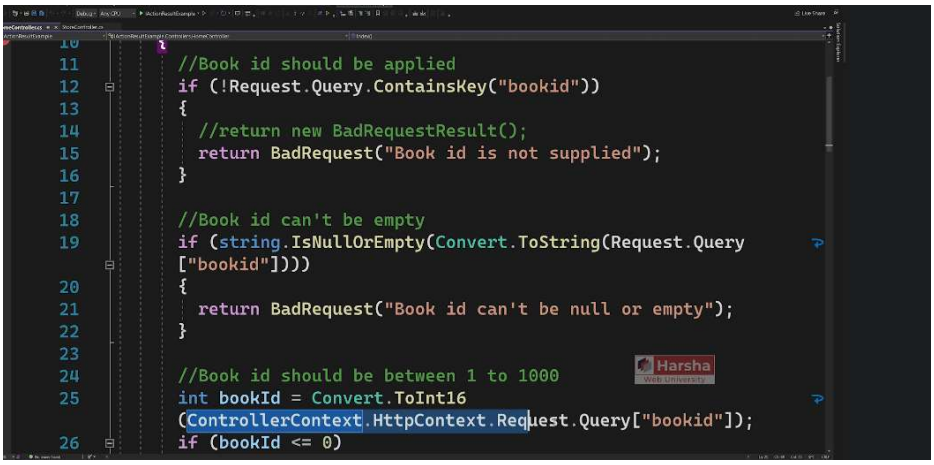
This is the order in which the model binding process picks up the values from the request. Whenever the URL matches a specific route mapped to a particular action method, before executing the action method of the controller, the model binding process occurs. It tries to read the values in the same order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



```
using Microsoft.AspNetCore.Mvc;

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

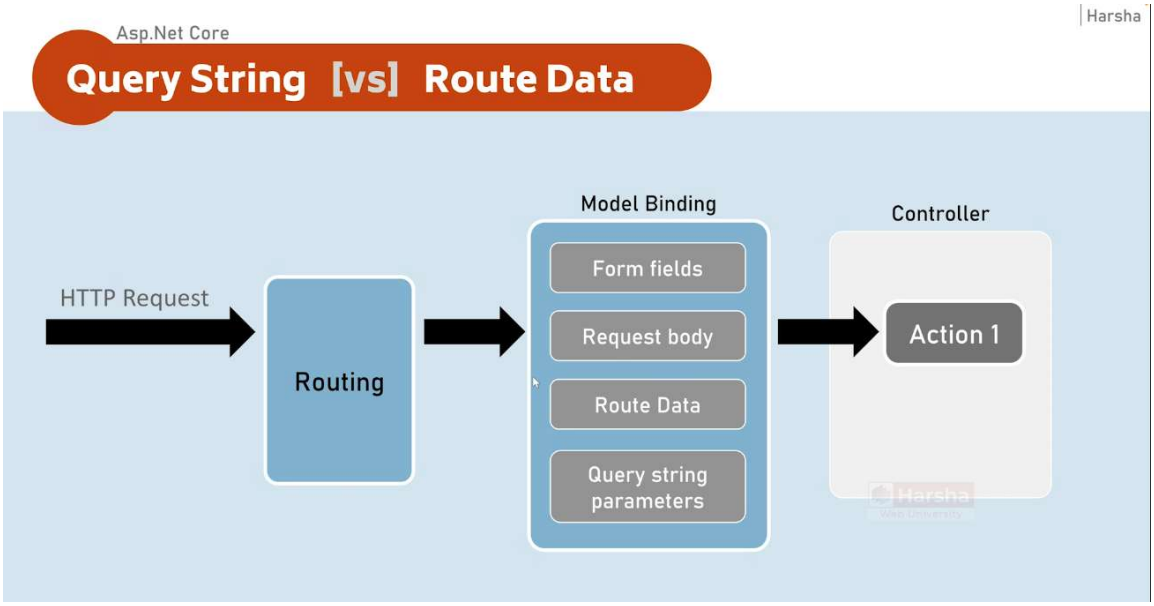


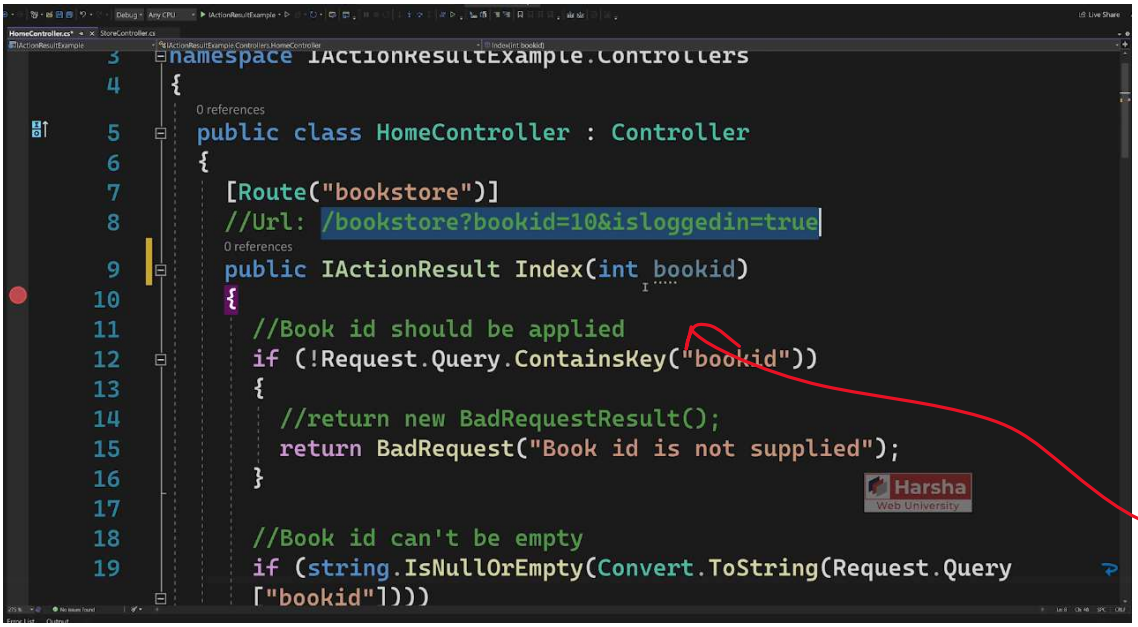
```
            //Book id should be applied
            if (!Request.Query.ContainsKey("bookid"))
            {
                //return new BadRequestResult();
                return BadRequest("Book id is not supplied");
            }

            //Book id can't be empty
            if (string.IsNullOrEmpty(Convert.ToString(Request.Query
            ["bookid"])))
            {
                return BadRequest("Book id can't be null or empty");
            }

            //Book id should be between 1 to 1000
            int bookId = Convert.ToInt16
            (ControllerContext.HttpContext.Request.Query["bookid"]);
            if (bookId <= 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?

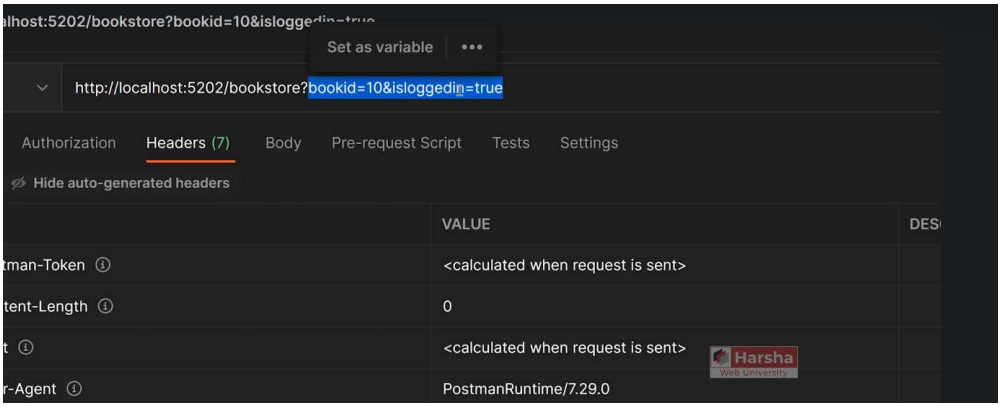
Remember that model binding is not a process you perform manually. Model binding is executed automatically before the action method execution starts. As part of the action method, you can assume that the values are already retrieved by model binding. This means model binding has picked up the values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



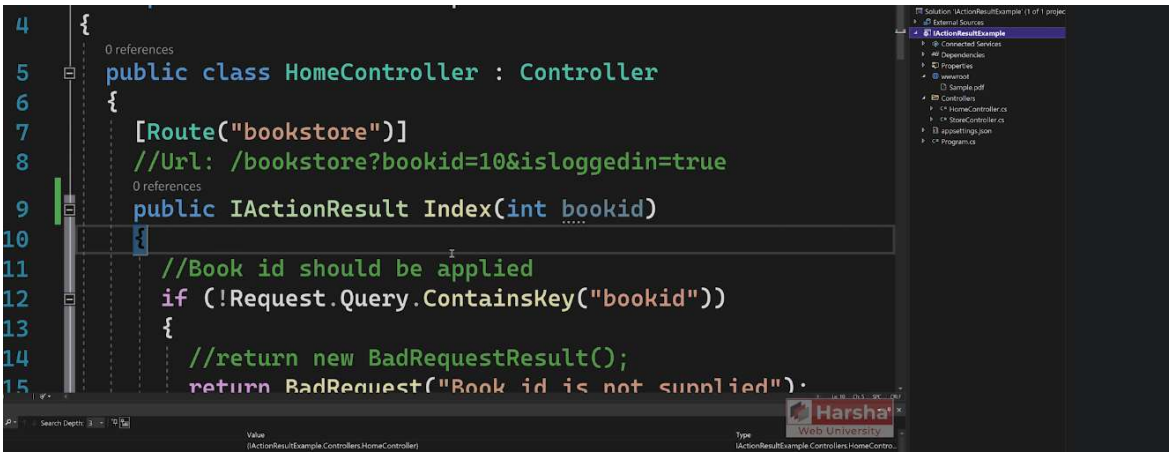


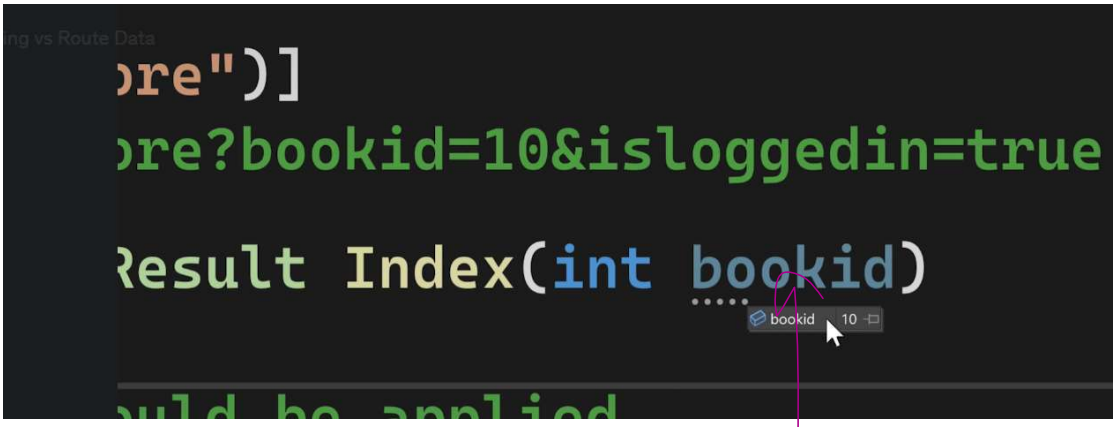
For example: int bookId—something like this. You can assume that ASP.NET Core automatically supplies the values for this parameter, which are retrieved through model binding.
Model binding always executes in the same order. At the fourth position, there is a source called query string parameters, which 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.



so, the breakpoint hits!



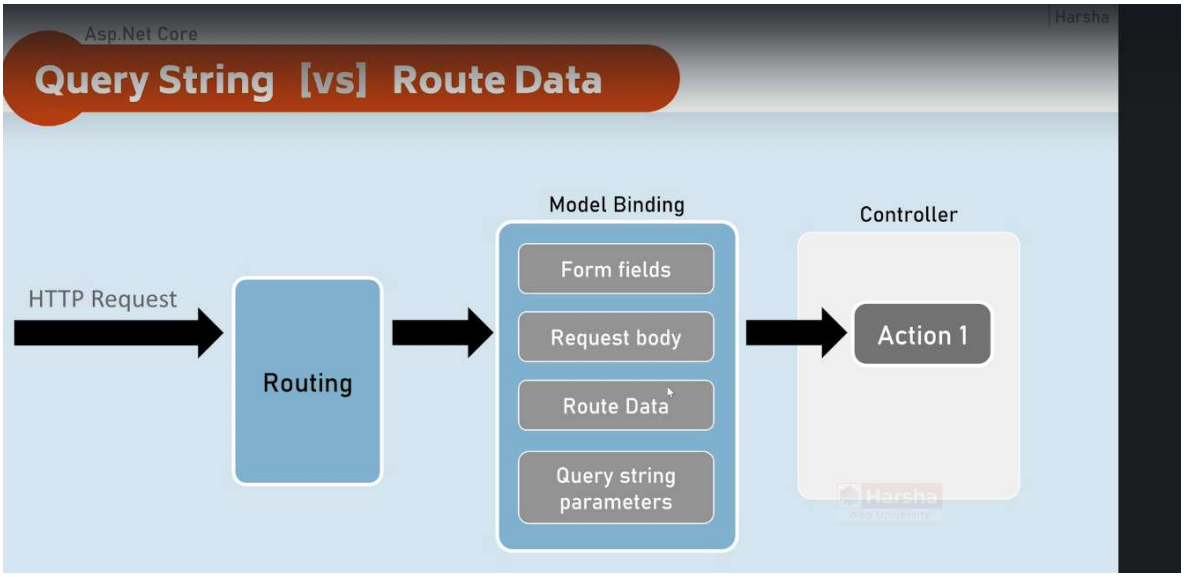


The benefit for the developer is that, everytime you do not have to write this lengthy syntax

```
{
    [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

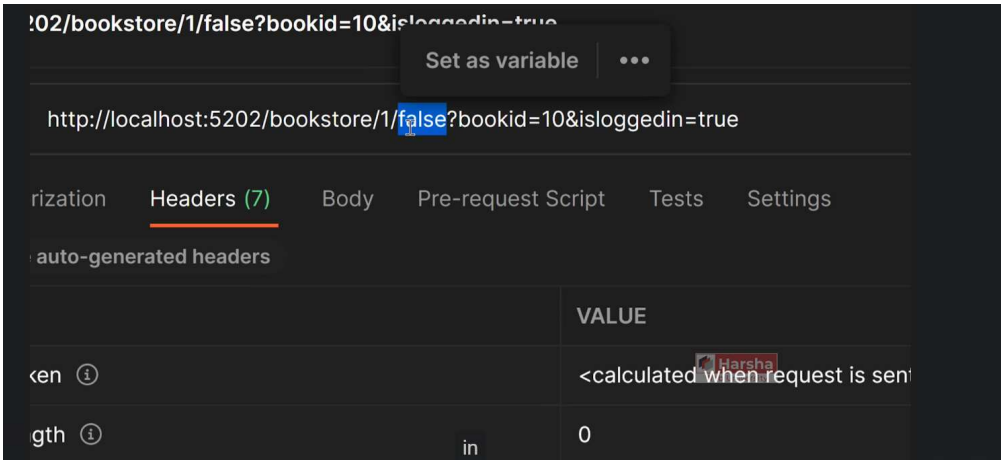
```
public class HomeController : Controller
{
    [Route("bookstore")]
    //Url: /bookstore?bookid=10&isloggedin=true
    0 references
    public IActionResult Index(int bookid)
    {
        //Book id should be applied
        if (!bookid)
        {
            //return new BadRequestResult();
  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

        //Book id can't be empty
```



Route Data has higher priority than Query String.

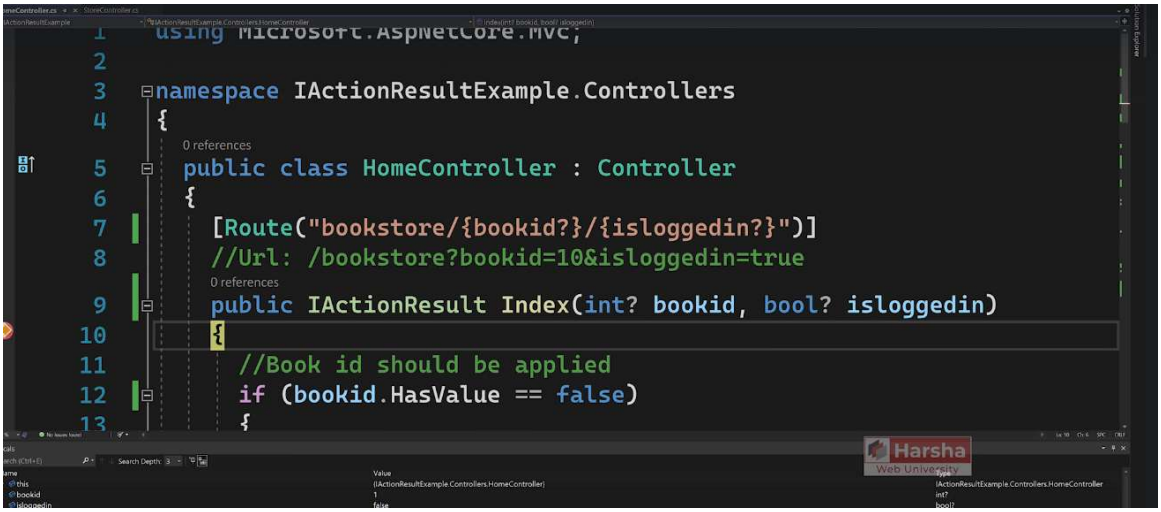
So, if we supply the same data from these two sources, 'route data' will be picked up.

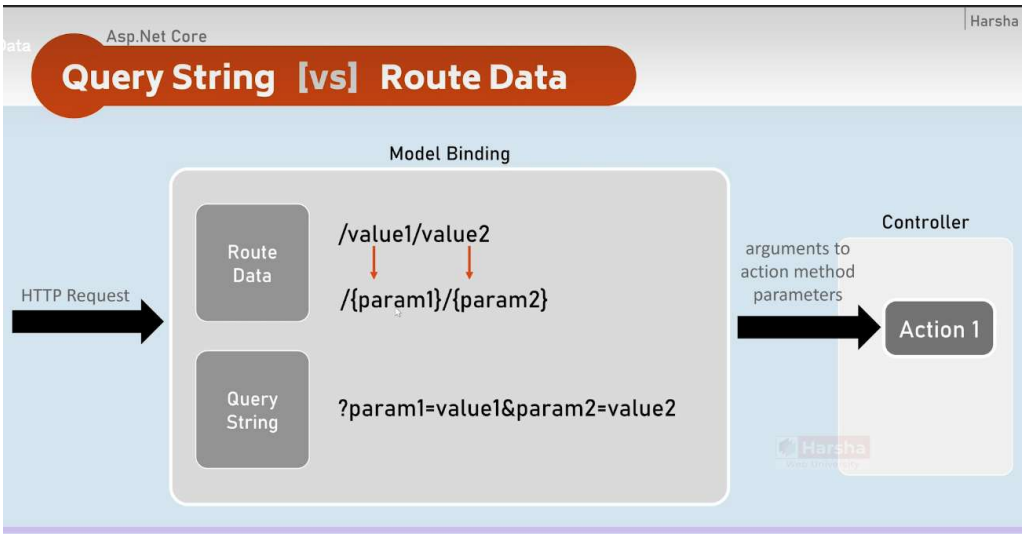
let's prove it!

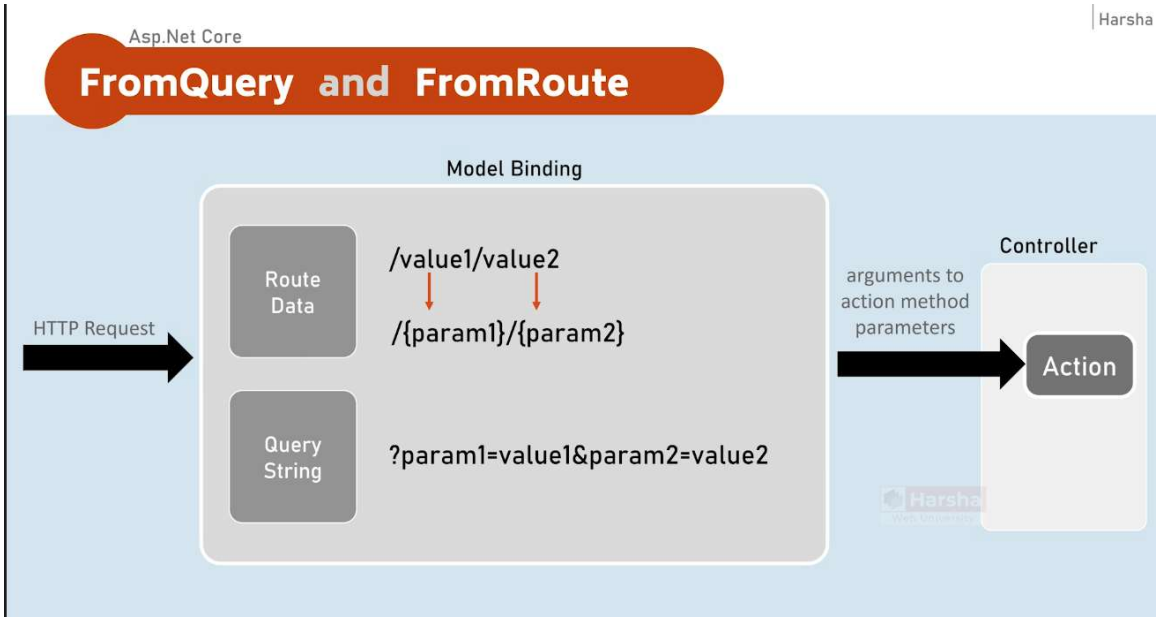


```
namespace IActionResultExample.Controllers
{
    public class HomeController : Controller
    {
        //https://localhost:7119?bookid=123&isloggedin=true
        [Route("bookstore/{bookid?}/{isLoggedIn?}")]
        [Route("/")]
        public IActionResult Index(int? bookid, bool? isLoggedIn)
        {
            //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!

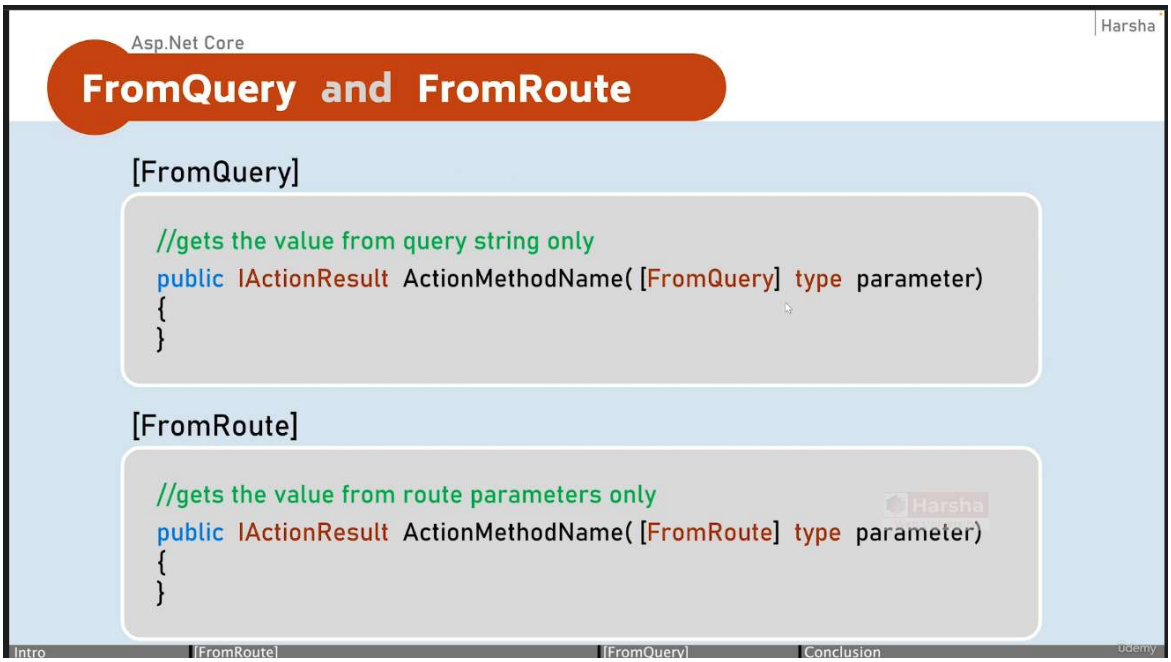




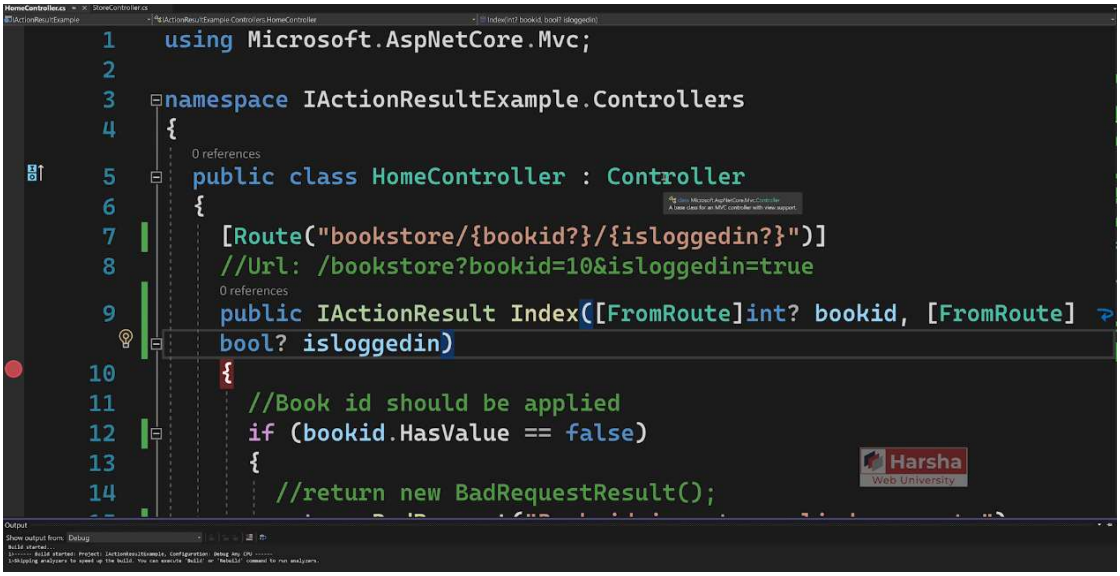


Model binding first picks up the value from the route data and then from the query string. This means whenever you create a parameter for the action method, that parameter will get its value from the route data first. If the value is null or not supplied, it will then try to get the value from the query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

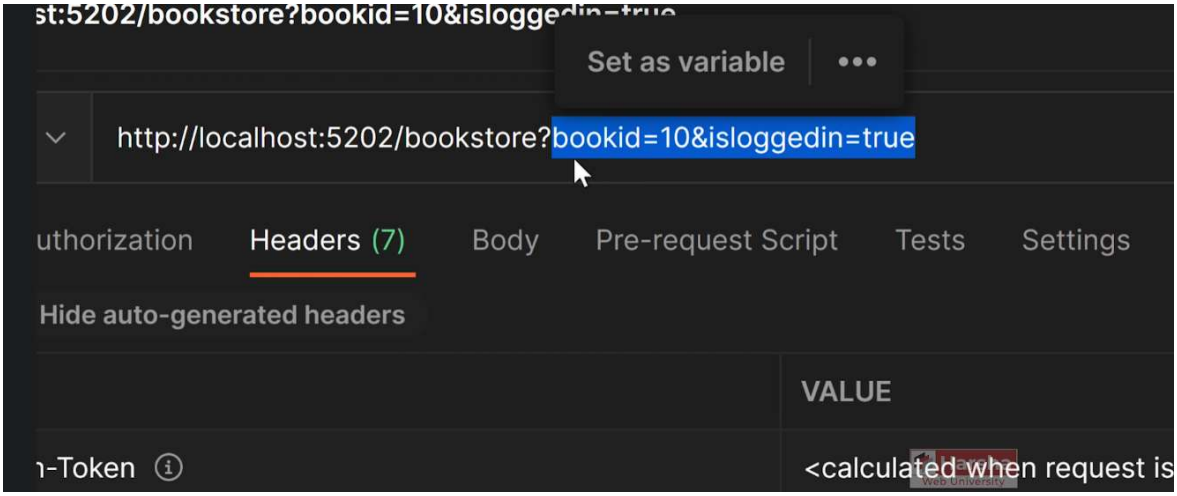
However, you can override this behavior based on your requirement by using FromQuery and FromRoute attributes for that particular parameter.



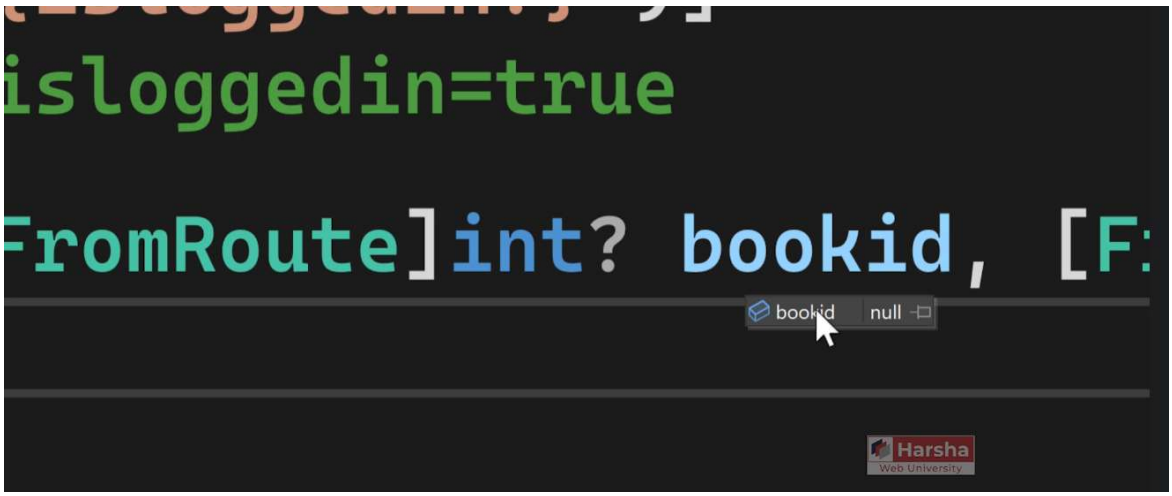
You need to apply FromQuery if you prefer to retrieve the value from the query string. Similarly, you need to use FromRoute to retrieve the value from the route parameter.



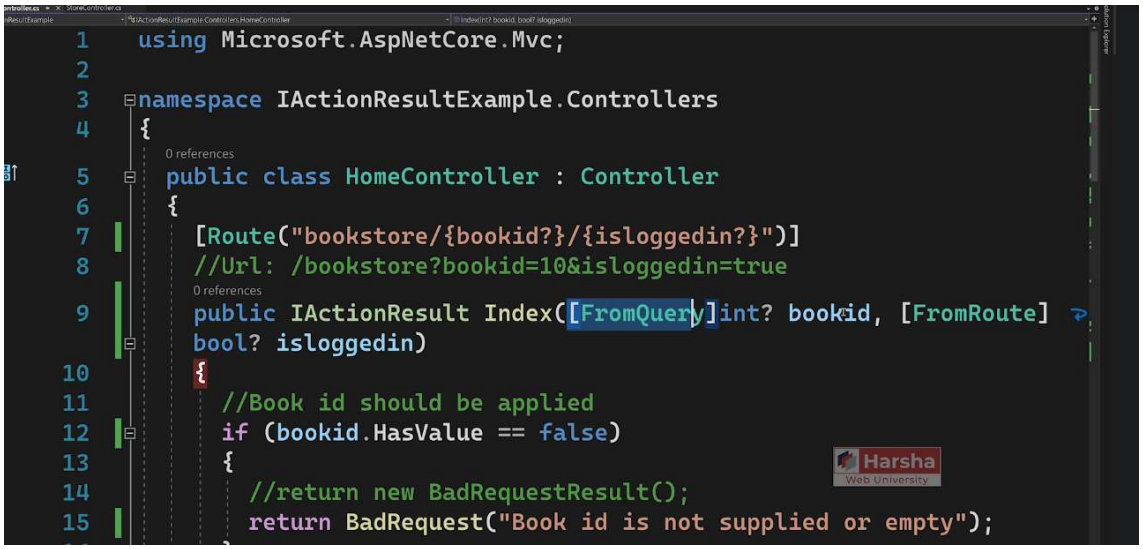
In this case, parameters are present in query string. not route parameters.

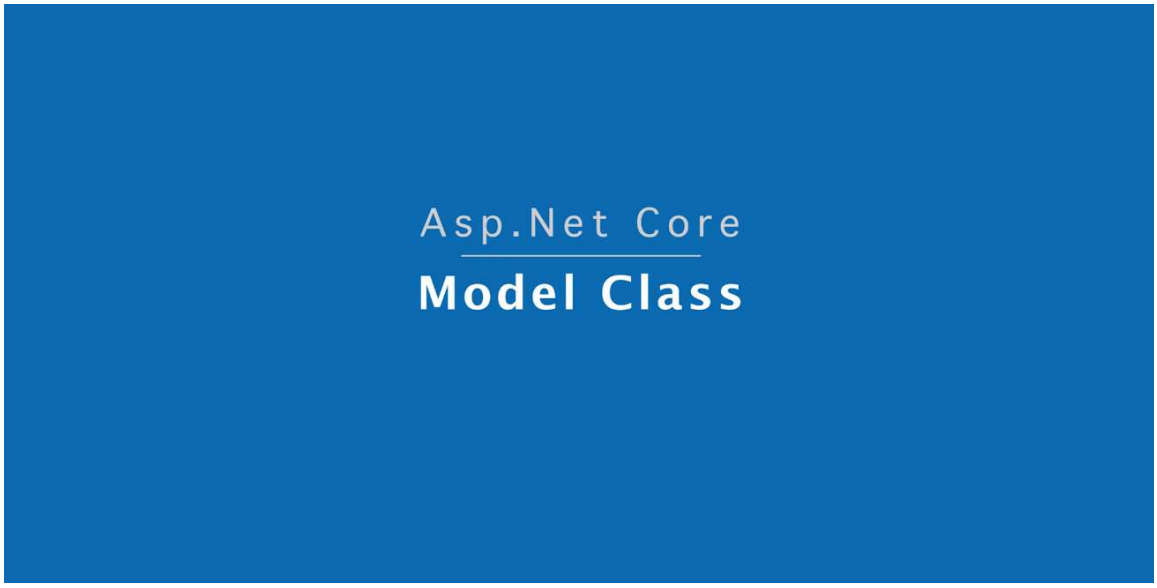


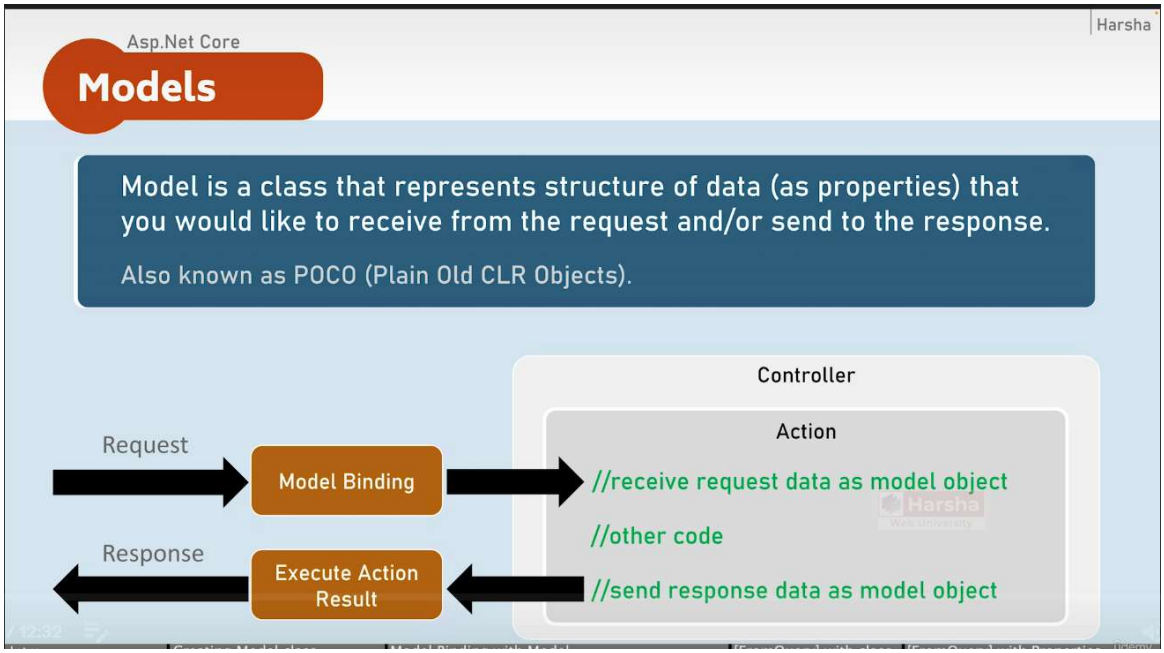
we're getting Null as we set [FromRoute]



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







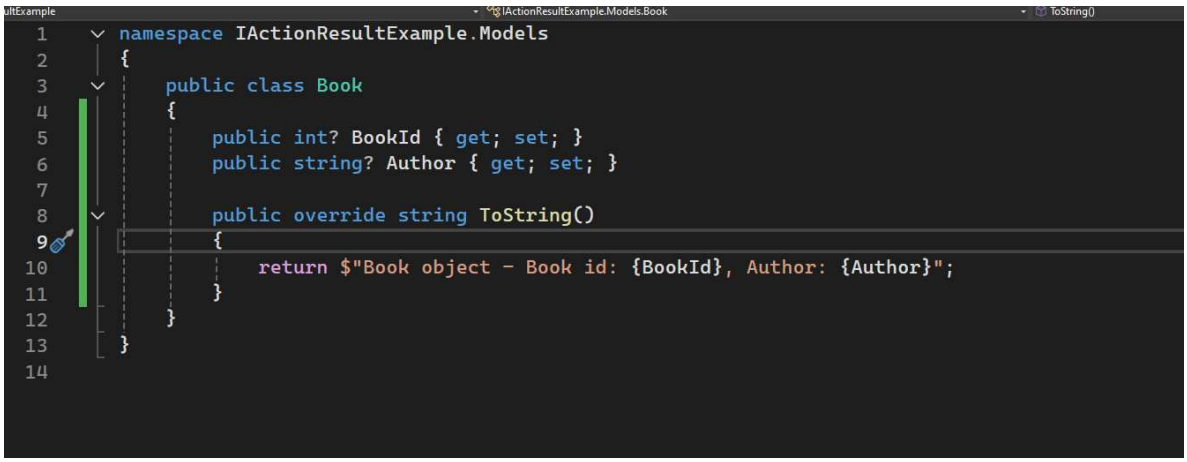
In ASP.NET Core, a model is a class that represents the structure of the data you want to receive from the request and send back as a response to the browser.

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.

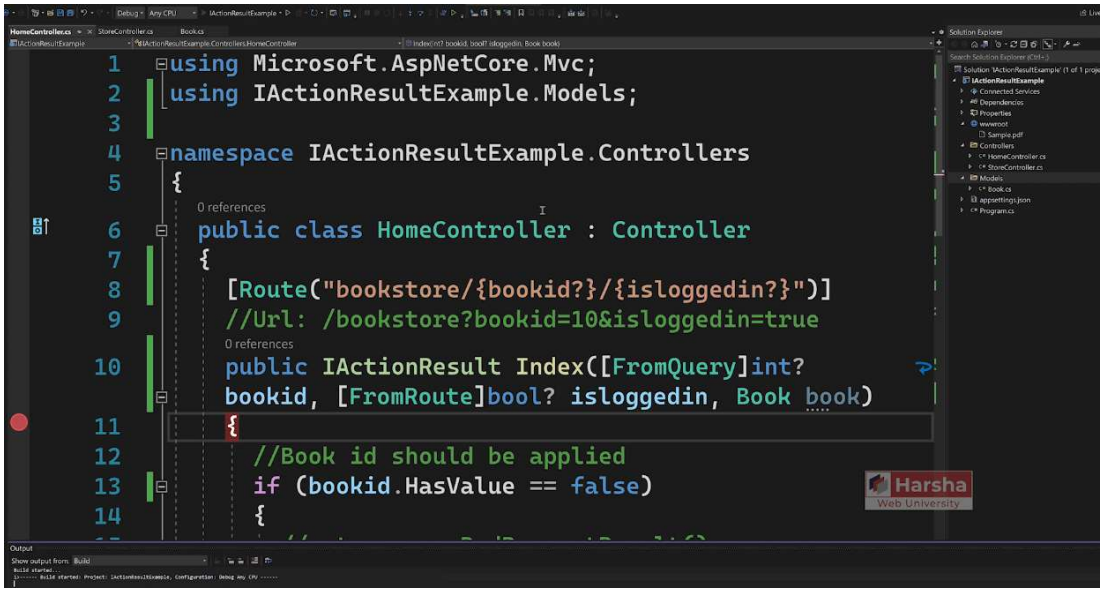
We will focus on returning the model object later. For now, let's focus on how to receive the model object as an argument.

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.

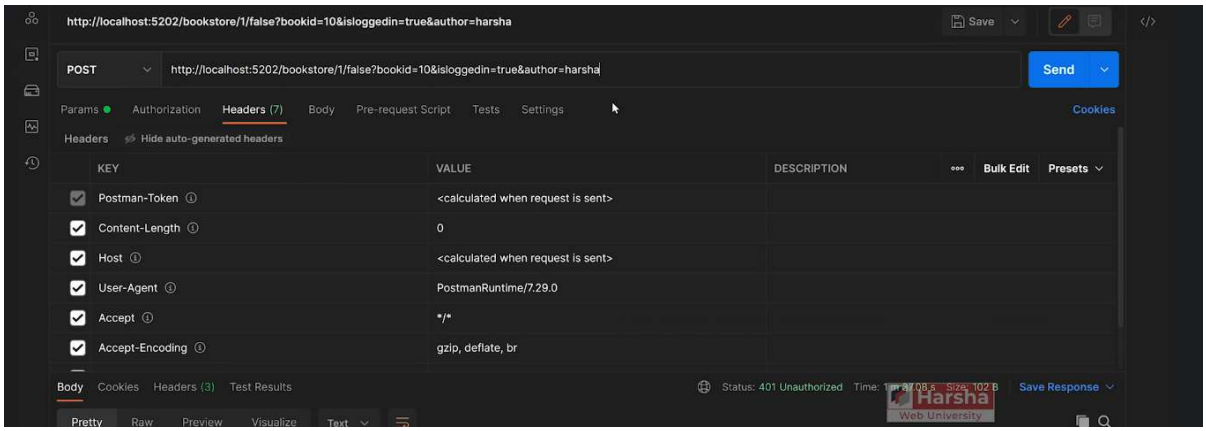


```
namespace IActionResultExample.Models
{
    public class Book
    {
        public int? BookId { get; set; }
        public string? Author { get; set; }

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



```
using Microsoft.AspNetCore.Mvc;
using IActionResultExample.Models;

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

let's run this.



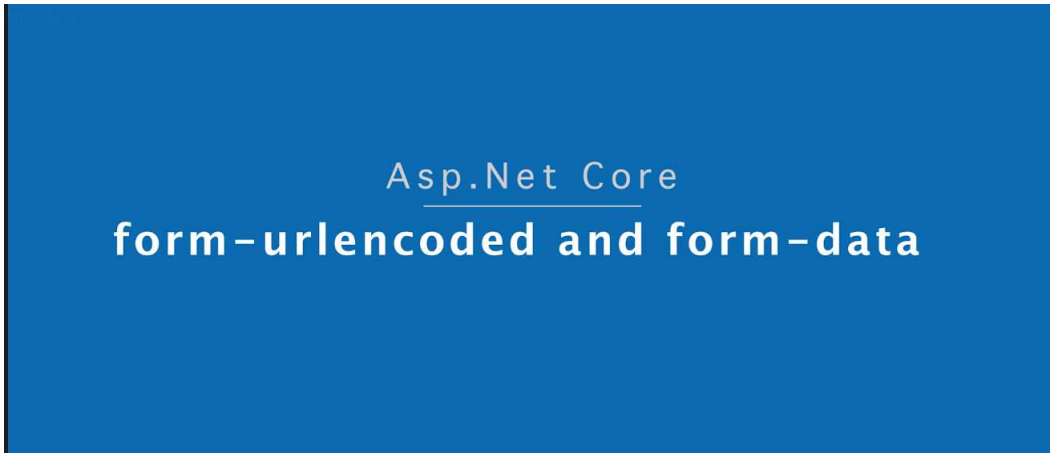
```
public class HomeController : Controller
{
    //https://localhost:7119?bookid=123&isloggedin=true
    [Route("bookstore/{bookid?}/{isLoggedIn?}")]
    [Route("/")]
    public IActionResult Index([FromQuery]int? bookid, [FromRoute]bool? isLoggedIn, Book book)
    {
        //Book id should be applied
        if (bookid.HasValue == false)
        {
            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");
```

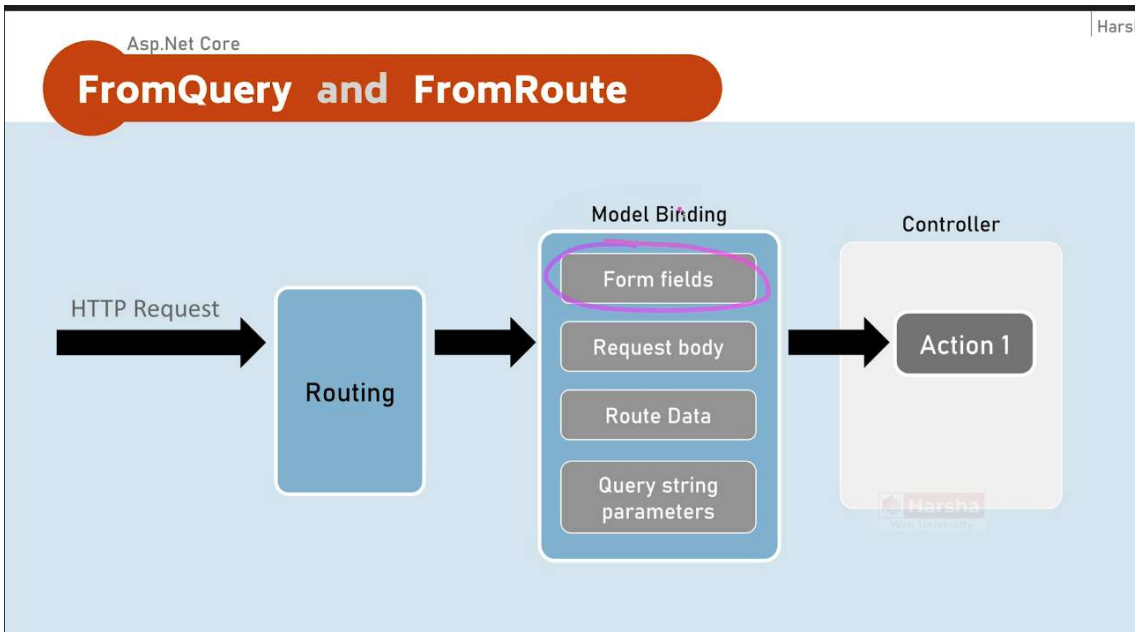
first it will look from route query to fill book object. it is missing. so it gets the data from query string.

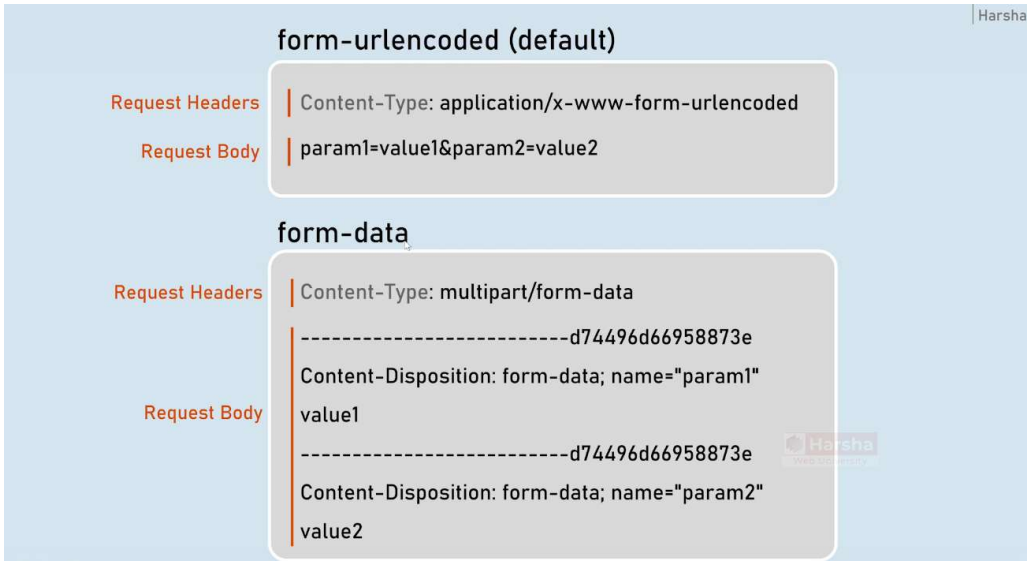
by specifying, [FromQuery] we are saying 'BookId' value will be picked up from the query string. not from the route data.

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

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

        public string? Author { get; set; }

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







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**.

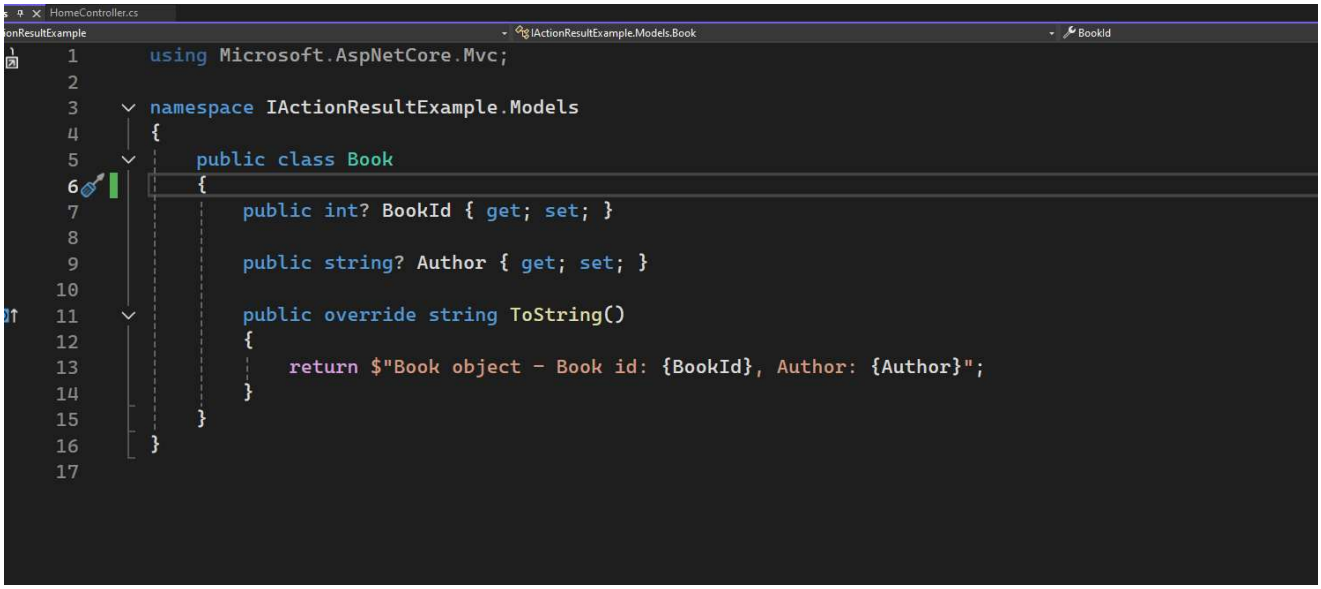
These values are typically added to the request body in one of two formats: either as application/x-www-form-urlencoded or as form data. The first format is simpler, while form data is more complex.

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.

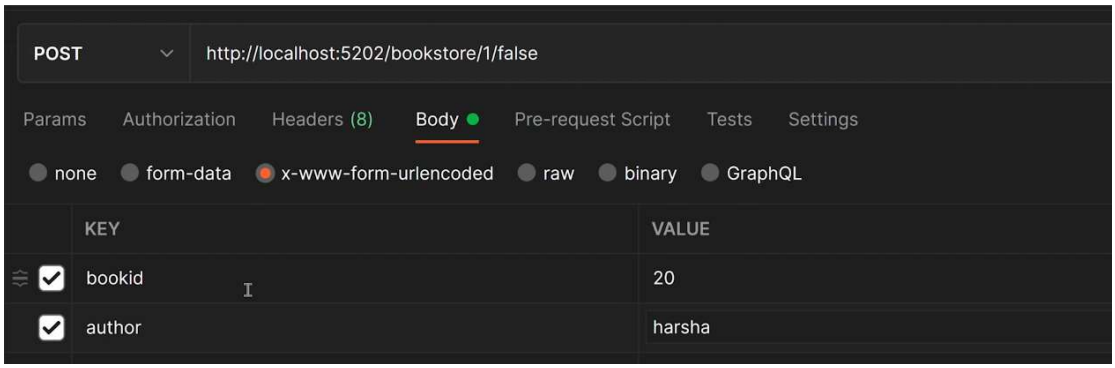
For GET requests, the query string is part of the URL,

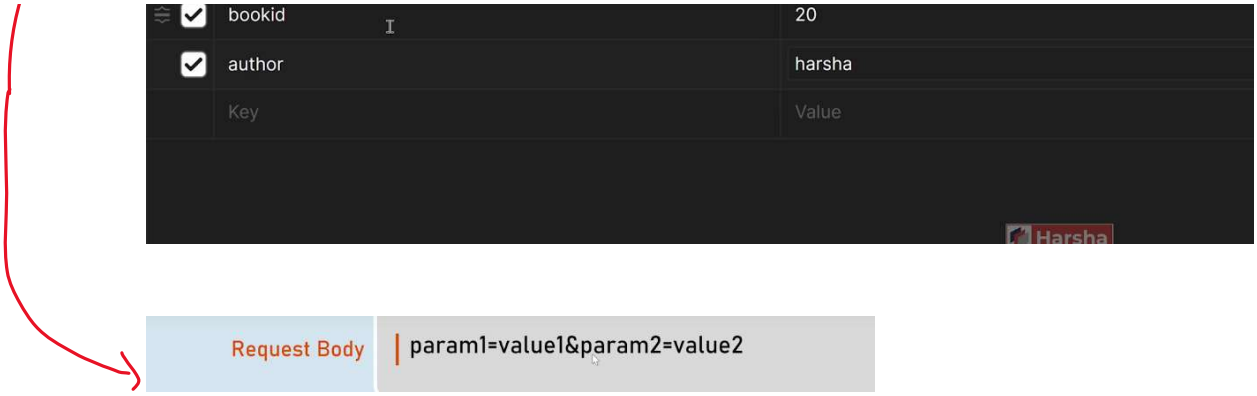
but for POST requests, particularly with application/x-www-form-urlencoded, the same query string is added to the r**equest body.**

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.

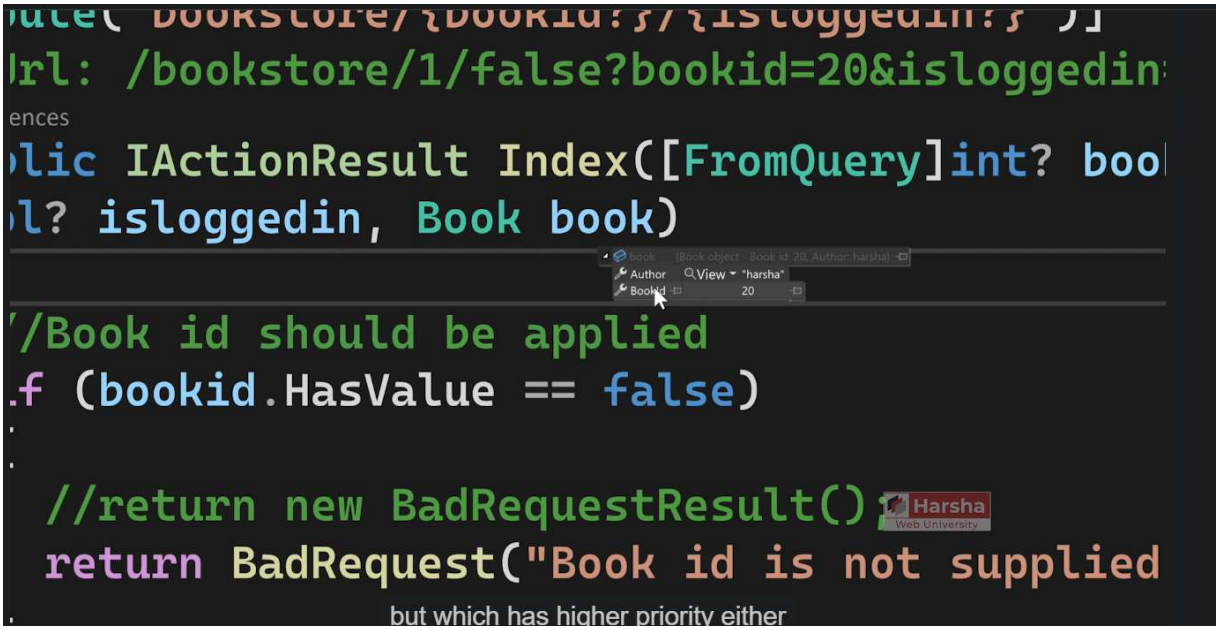


go to postman.
these values are added in the request body in the query string format.



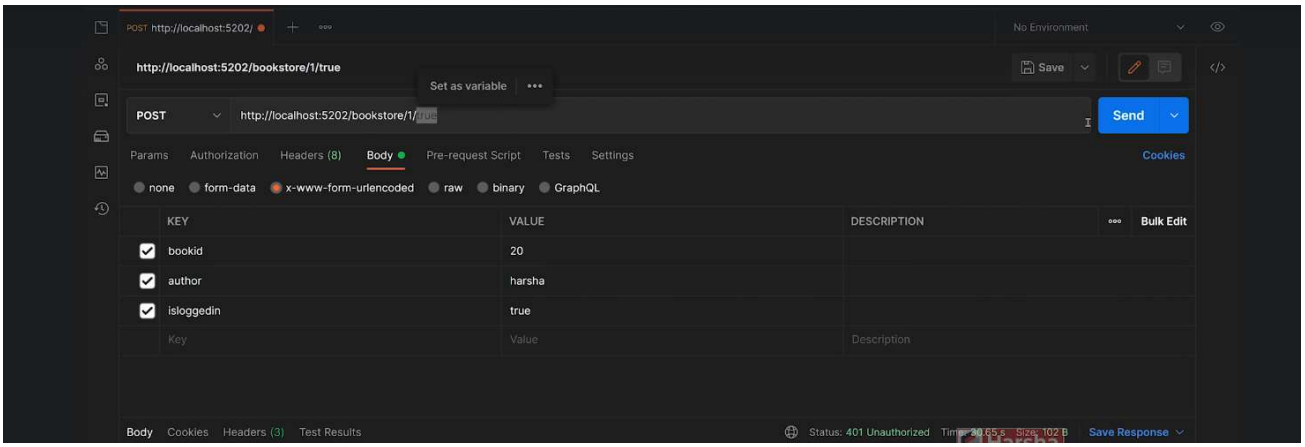


content-type will be set automatically.



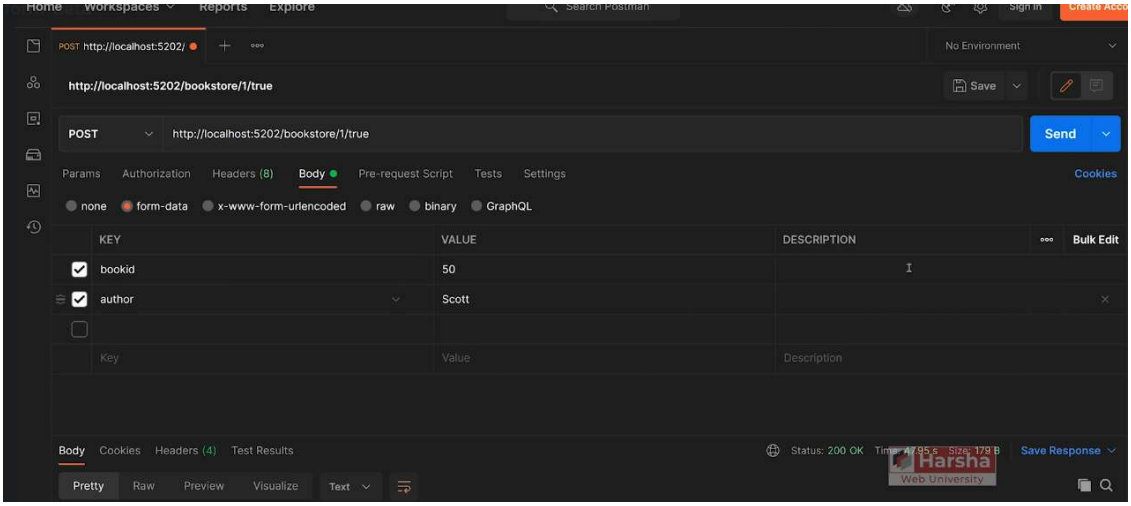
here observe that, BookId is set to 20. We have also pass bookId as 1 from route parameter. But which one has more priority here?

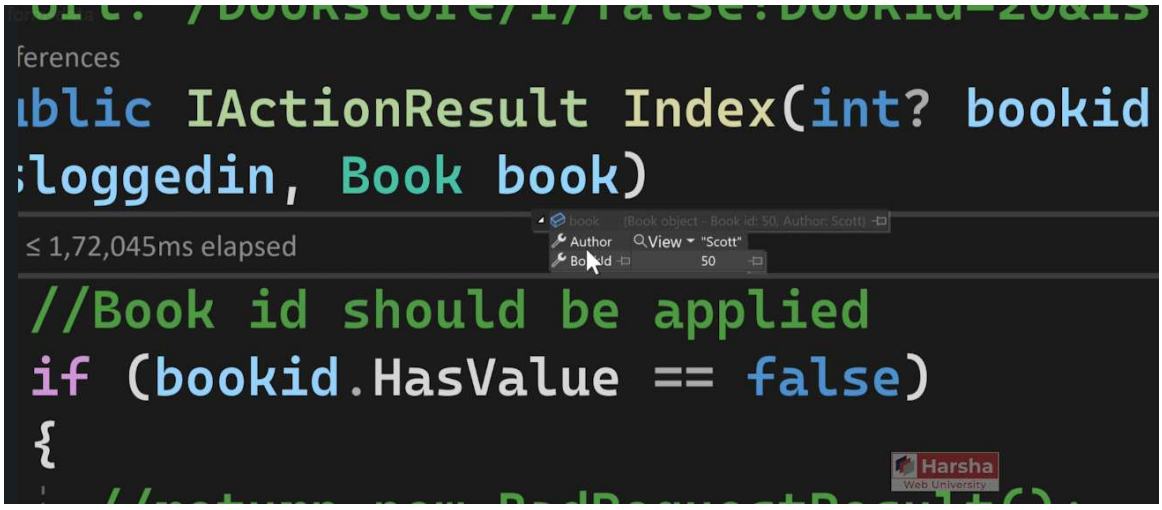
as you can see, ***From data!***



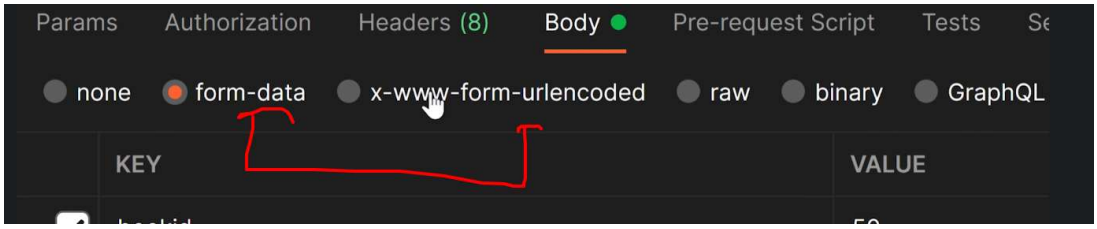
so this was form-url-encoded. now let's explore form-data.

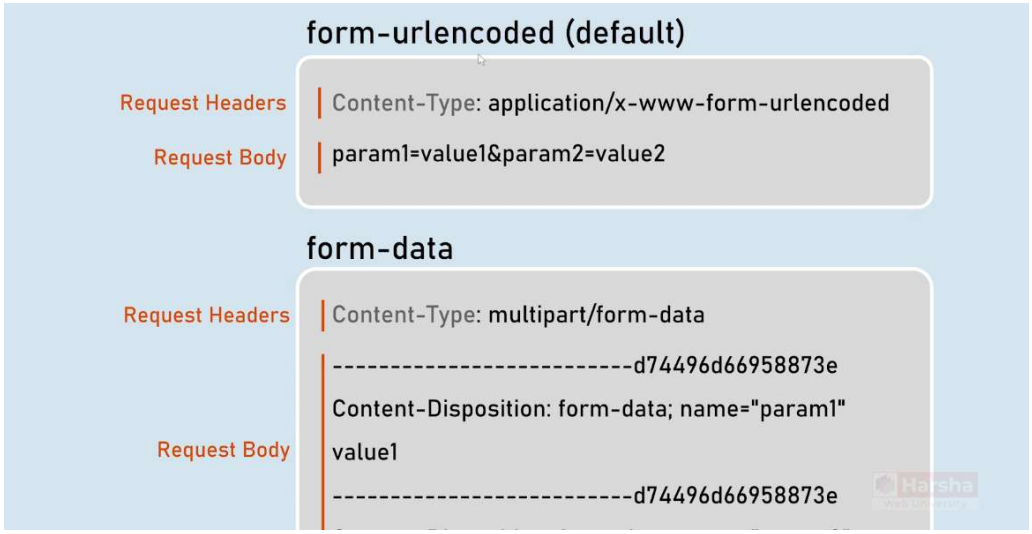
form postman, select form-data then click on 'Send'





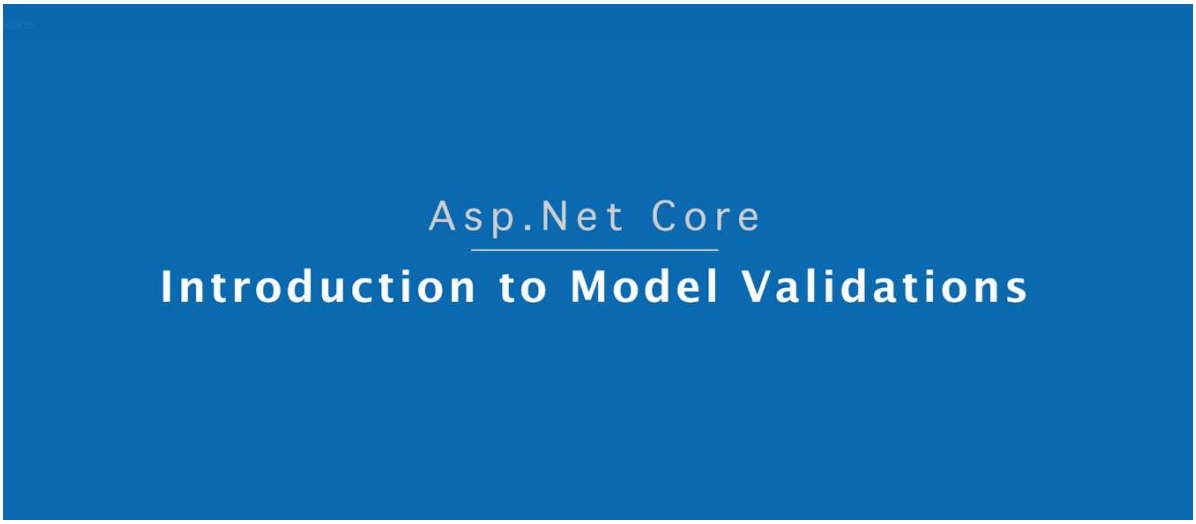
so, what's the difference between both of these?





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

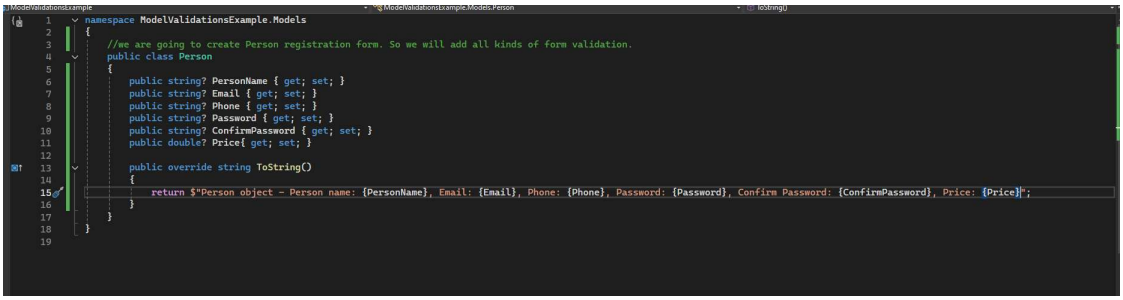
If you want to attach file, then 'form-data' is your only option. lets say you want to upload photo to facebook.



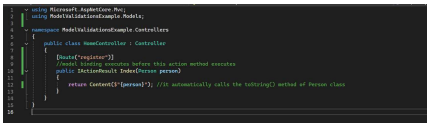
```
15          //return new BadRequestResult();
16          return BadRequest("Book id is not supplied or empty");
17        }
18
19        //Book id can't be less than or equal to 0
20        if (bookid <= 0)
21        {
22          return BadRequest("Book id can't be less than or equal to
            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
```

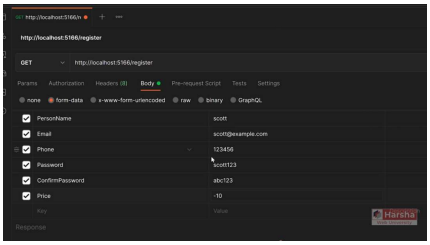
If we validate each and every field in real-world project. it will be very difficult. Check the amount of code that we have written for validations.

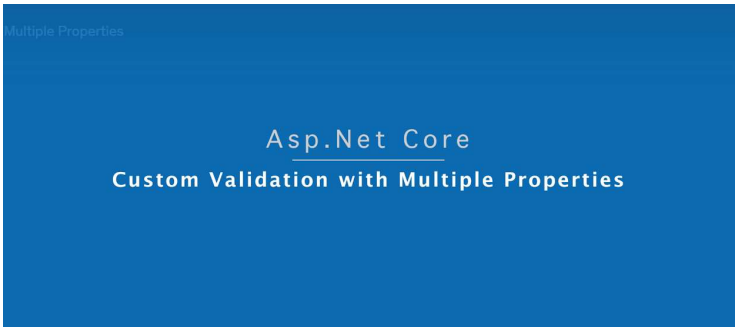
In case of model validations, you will be writing validation rules as attributes just before the properties.

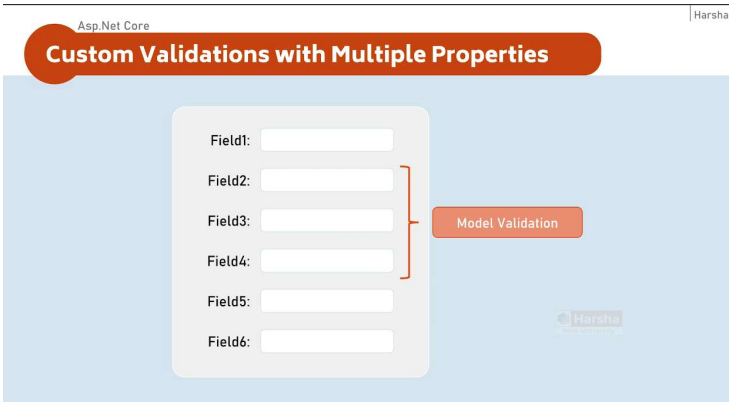


```
1   namespace ModelValidationsExample.Models
2   {
3     //we are going to create Person registration form. So we will add all kinds of form validation.
4     public class Person
5     {
6       public string? PersonName { get; set; }
7       public string? Email { get; set; }
8       public string? Phone { get; set; }
9       public string? Password { get; set; }
10      public string? ConfirmPassword { get; set; }
11      public double? Price{ get; set; }
12
13      public override string ToString()
14      {
15        return $"Person object - Person name: {PersonName}, Email: {Email}, Phone: {Phone}, Password: {Password}, Confirm Password: {ConfirmPassword}, Price: {Price}";
16      }
17    }
18  }
19
```

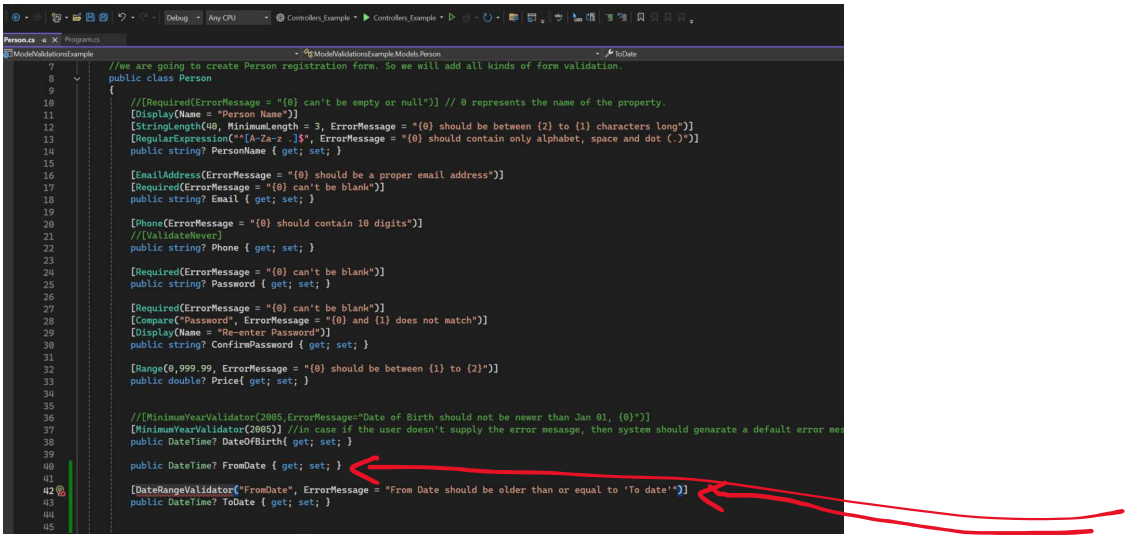






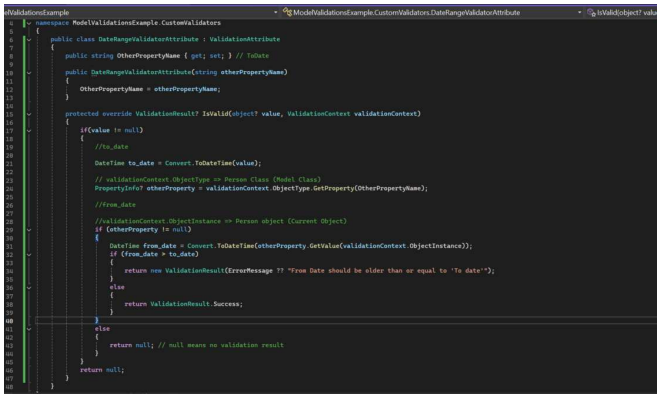


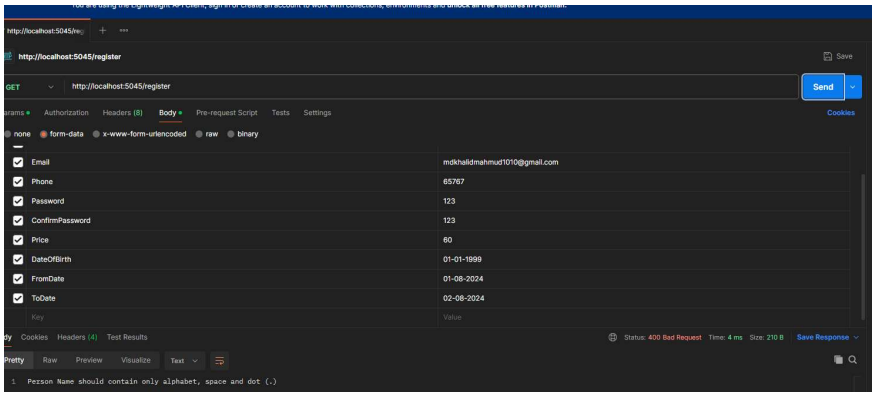
In the last lecture, we have learned how to validate a single property or single field. But sometimes, you have to validate multiple fields. For example, in the registration form, you have more than five text boxes, and you would like to validate two or more properties or fields. Then how do you apply 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.
    [Display(Name = "Person Name")]
    [StringLength(40, 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")]
    [Display(Name = "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 mesasge, then system should genarate a default error mes
    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.





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:

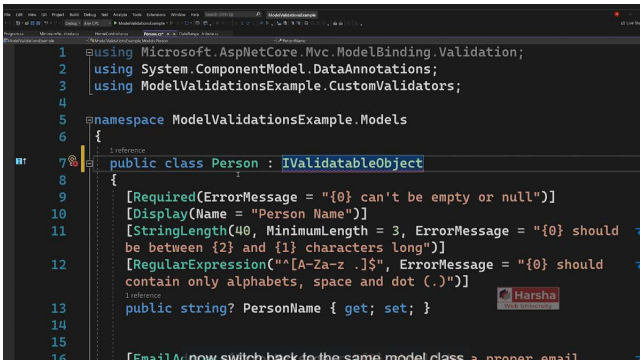
- If **Date of Birth** is provided, **Age** should be null.
- If **Age** is provided, **Date of Birth** should be null.

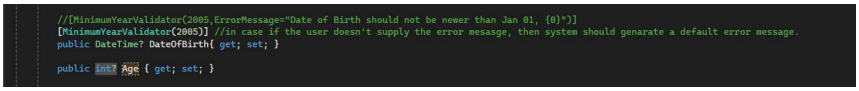
Instead of creating a separate custom validator class, you can implement this logic in the model class itself using the IValidatableObject interface. This approach is ideal when you don't need the validation logic to be reusable or when the requirement is relatively simple.

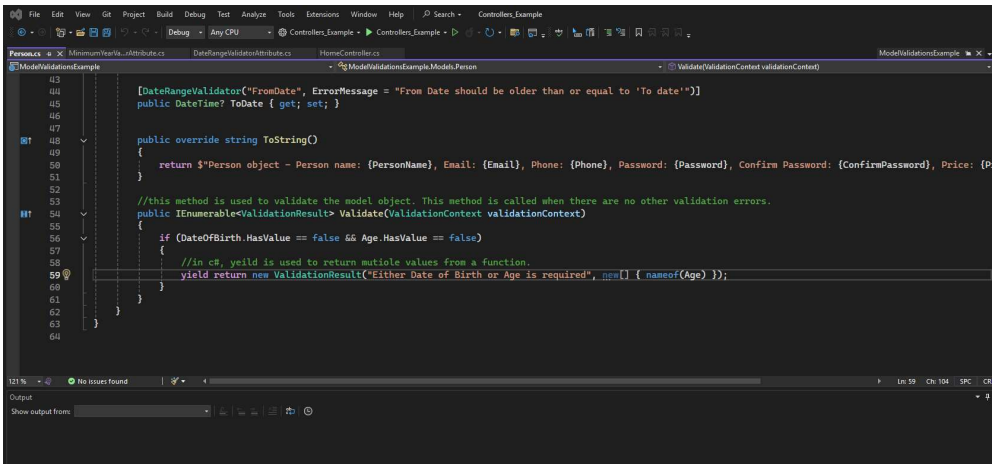
The IValidatableObject interface allows you to encapsulate validation logic within the model class, avoiding the complexity of reusable reflection-based validators. Here's how it works:

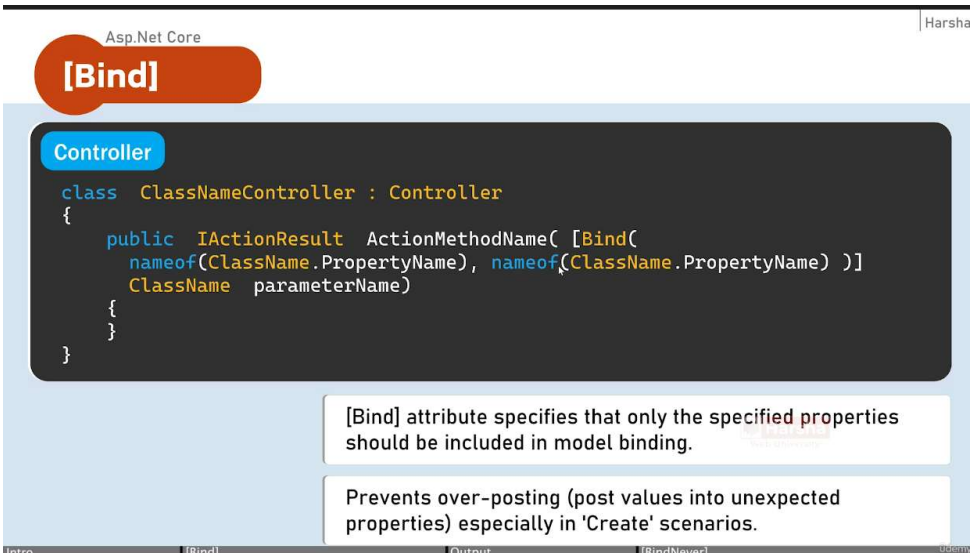
1. Implement the IValidatableObject interface in your model class.
2. Write your custom validation logic in the Validate method.
3. This method will automatically be called during the model validation process.

Let me show you how to implement it.



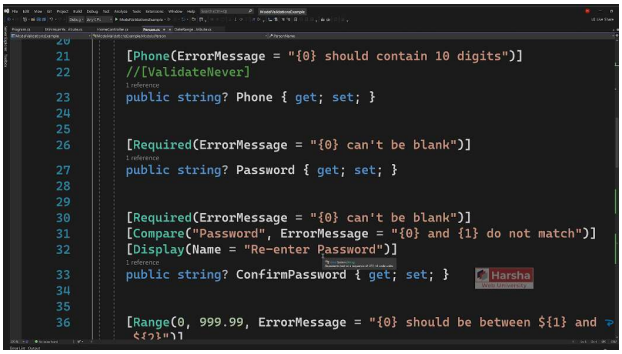




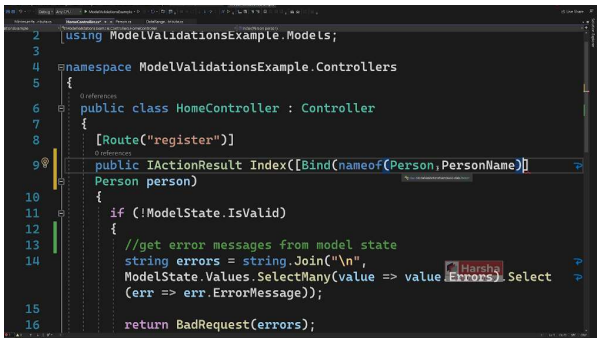


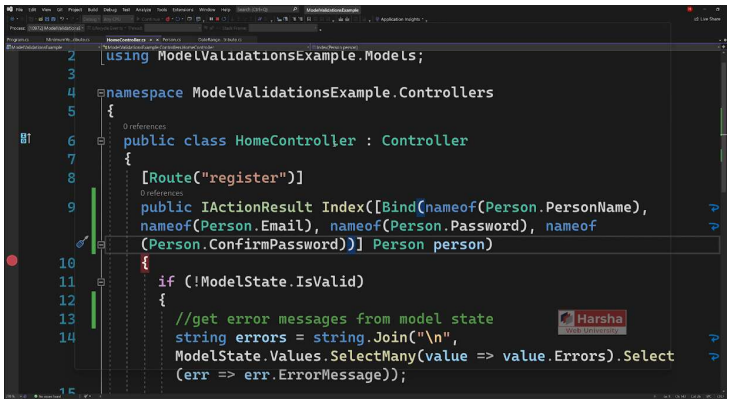
The bind attribute specifies what properties you would like to include in the model binding so that remaining properties will not be binded. See, by default in model binding, all the properties of the model class will be binded. But if you mention bind and specify the list of properties that you want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

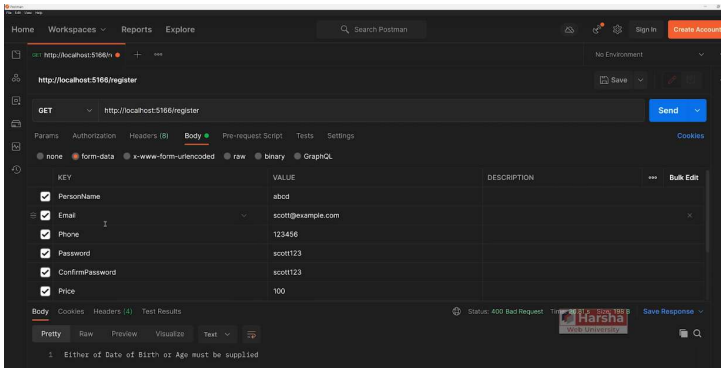


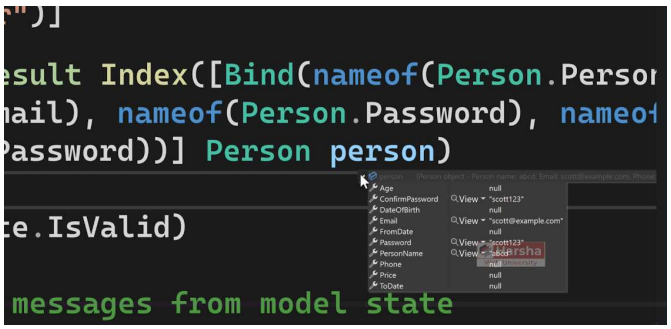
So, in your home controller, in this action method, you can add the bind attribute near this parameter and then specify the list of properties that you want to accept. You can either mention the list of properties with comma separation like this, or, in order to get it dynamically, you can use the nameof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



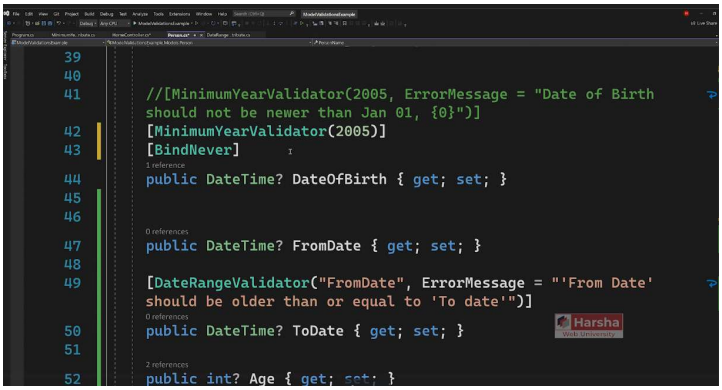


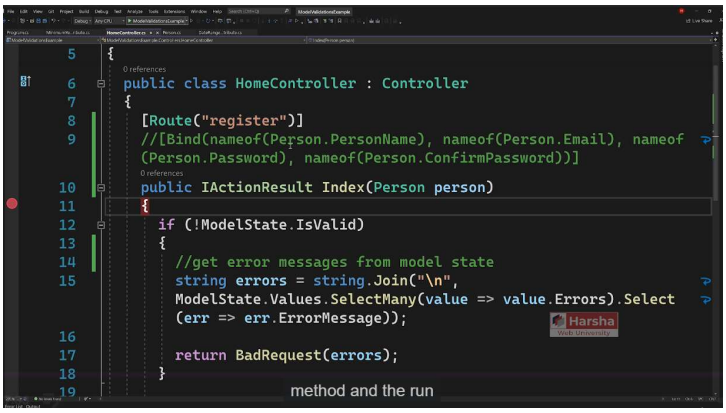
Now, in the postman , we are submitting all the values.

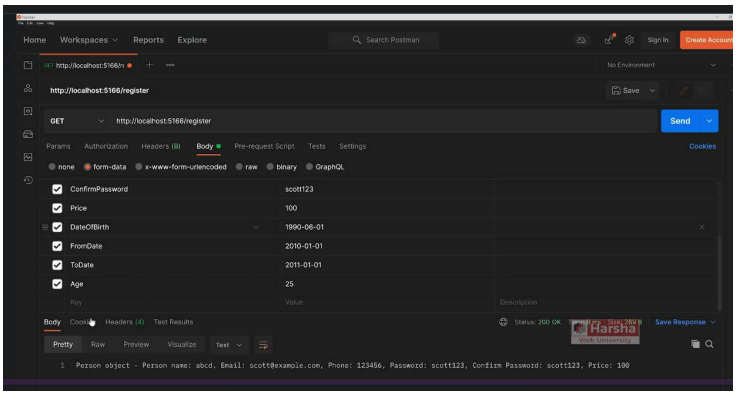




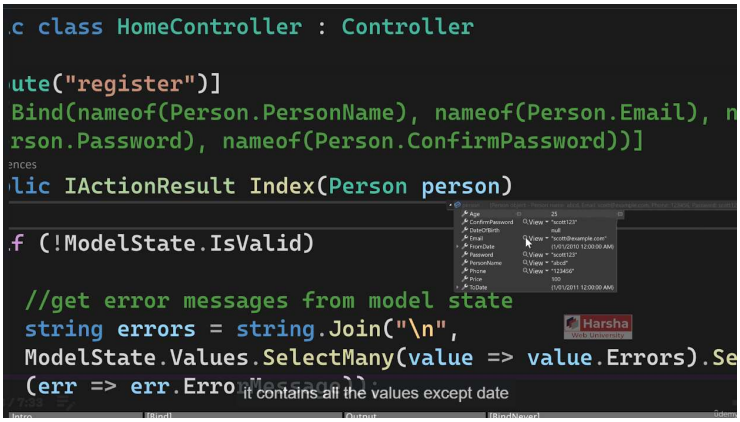
Alternatively, you can skip one or two property. Rest of the field will be binded.

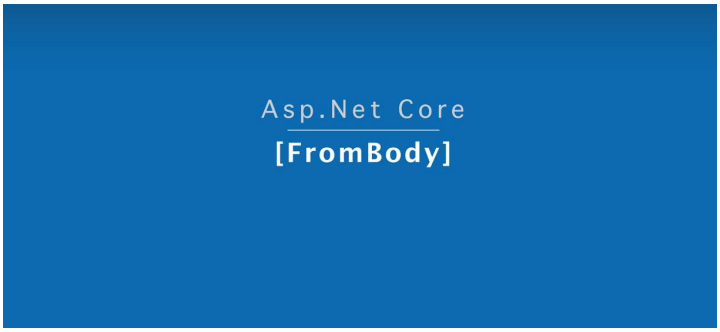


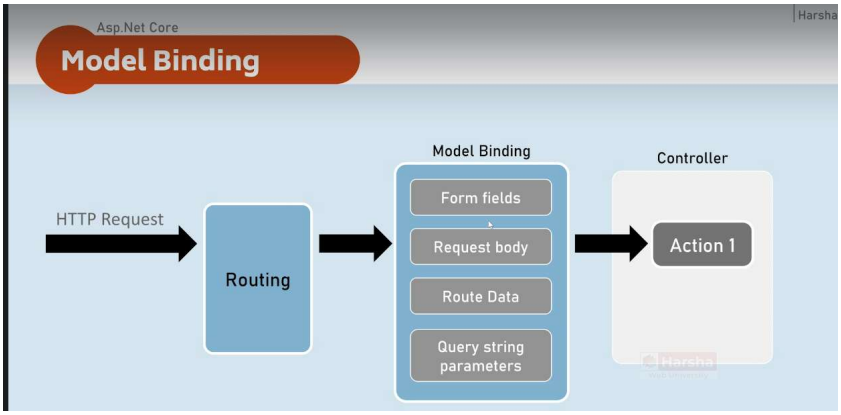




It contains all the value except DateOfBirth.

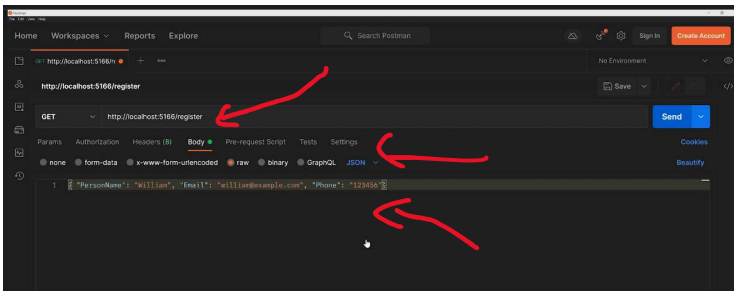




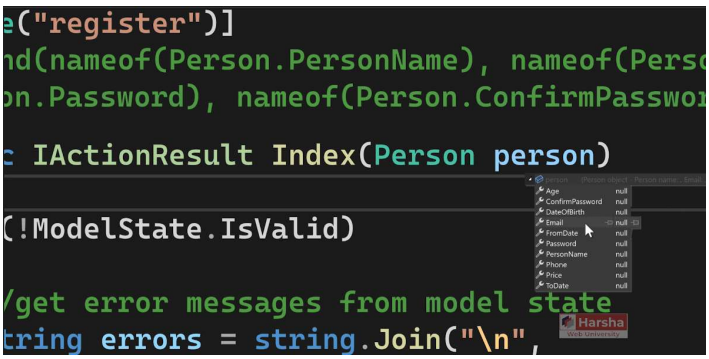


So far, in model binding, we have seen the form fields, route data, and query string parameters. Form fields are of two types: either form URL-encoded or form data. We have already seen that in the earlier lectures in the same section.
But there are some other content types other than form data and form 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

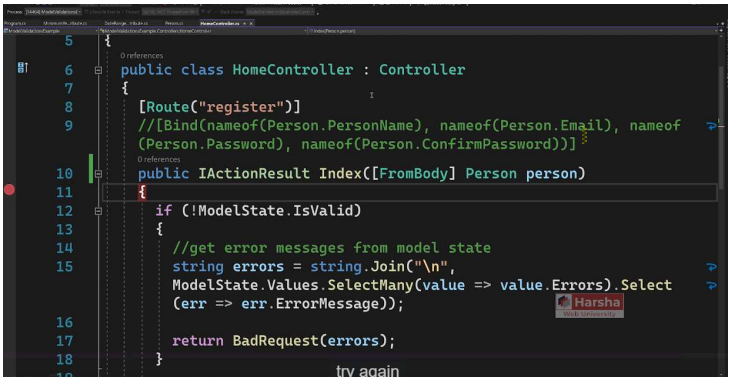
be converted into the model object.

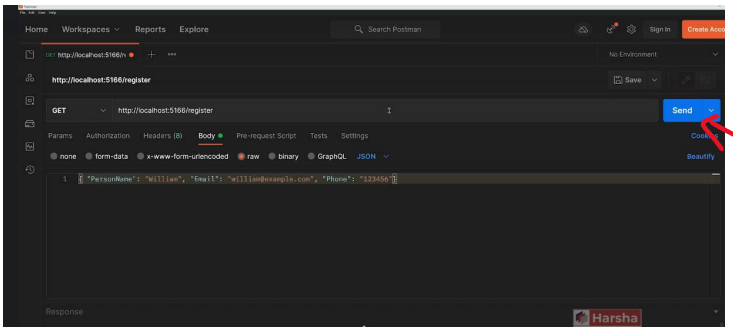


But we have got all the fields null!!! But why?



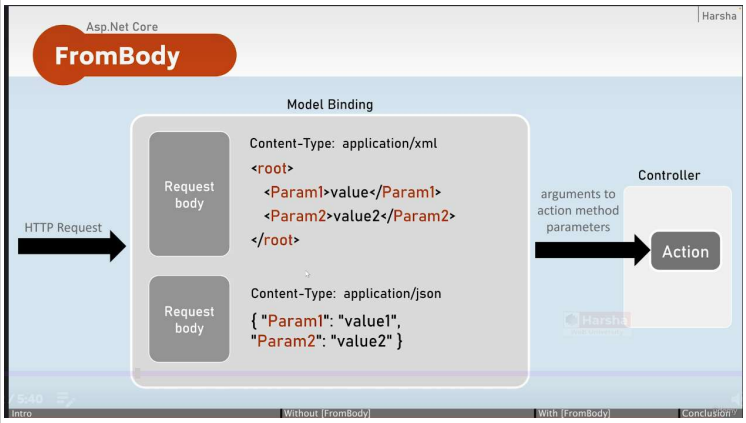
We have to enable **'input-formatters' by** using ***form body*. So that instead of regular model binding, it enables the json input formatter to read the content from the request body and make it as an object of the model class.**

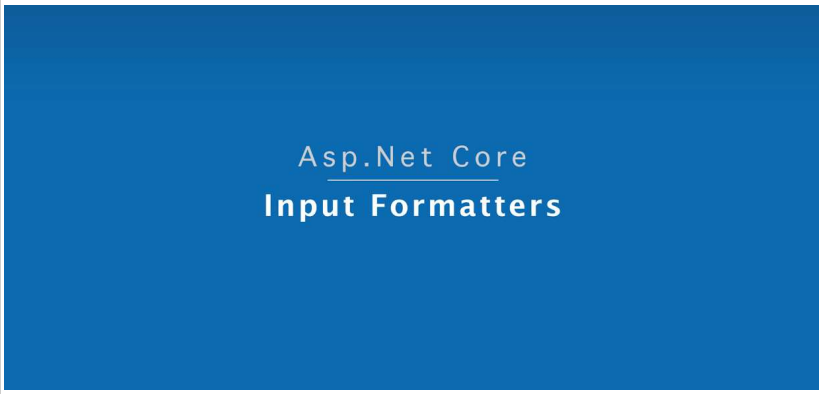


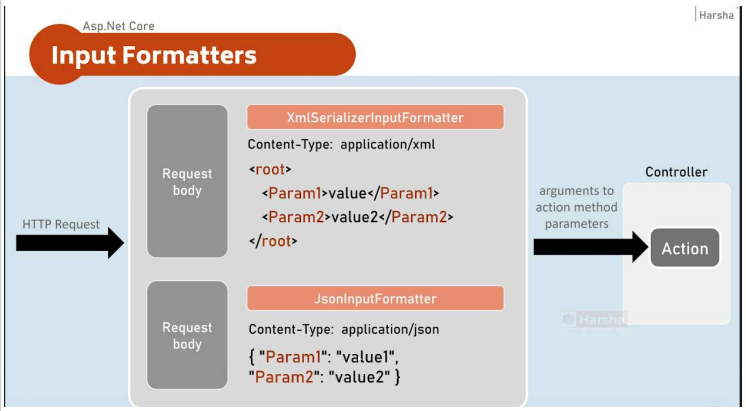


```
ter")]
f(Person.PersonName), nameof(Person.Email),
ord), nameof(Person.ConfirmPassword))]

nResult Index([FromBody] Person person)

tate.IsValid)

or messages from model state
rors = string.Join("\n",
```

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



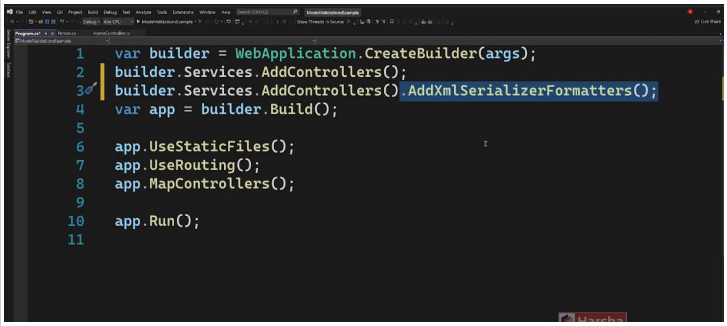




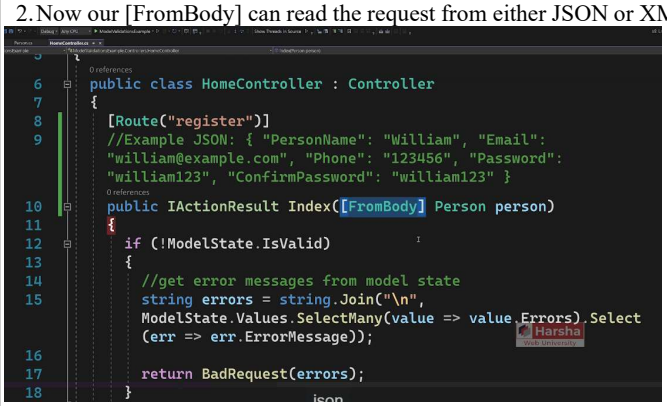
Input formatters are the internal classes in ASP.NET Core used to transform or convert the request body into the

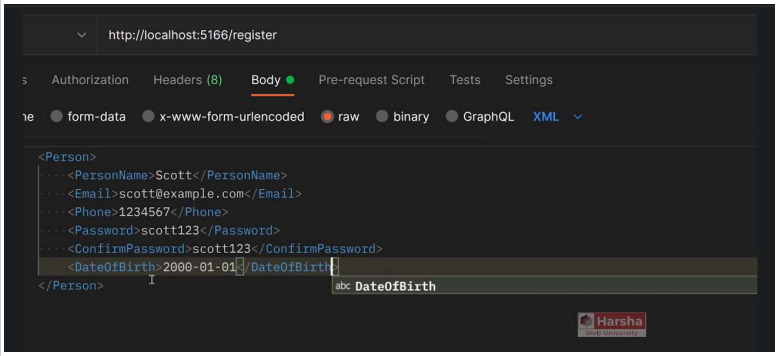
model object.
For example, whenever the request body contains JSON data and the content type is application/json in the request headers, the JSON input formatter will be enabled automatically. It reads the JSON data from the request body and converts it into a model object. For instance, it can read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

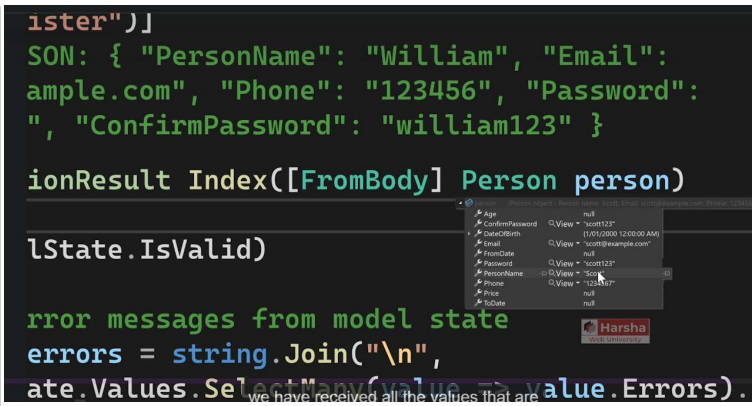
1. We have added XML Serializer Formatter also to the Controllers.

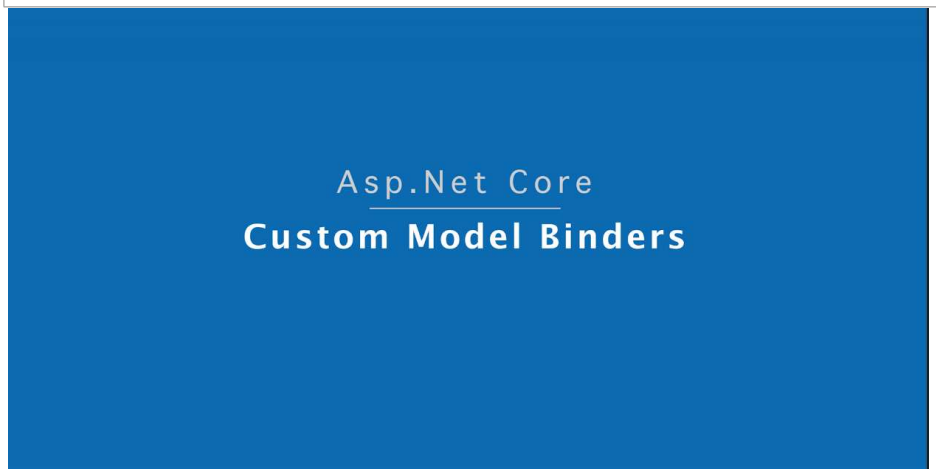


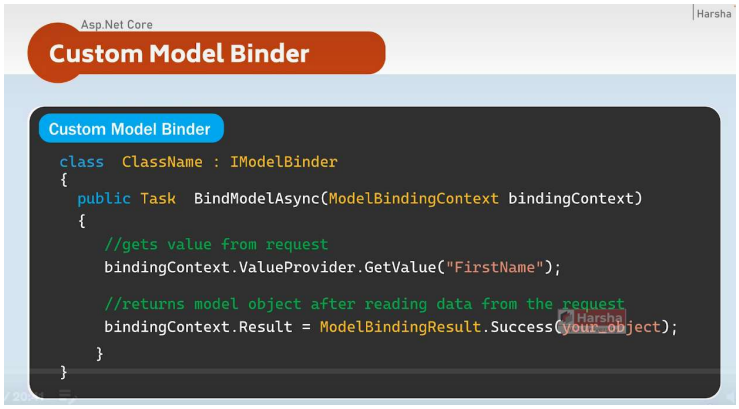
2. Now our [FromBody] can read the request from either JSON or XML











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.

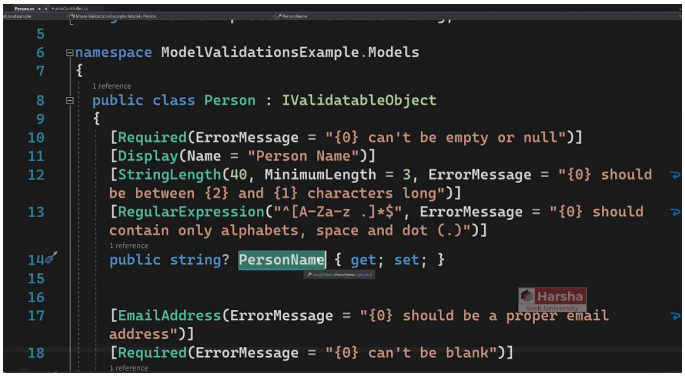
For example, let's say whenever the user has submitted the first name and last name in the request, you would like to concatenate them as a string value, that is, into the person name property. Otherwise, if the user has submitted year, month, and date independently, you have to combine them together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

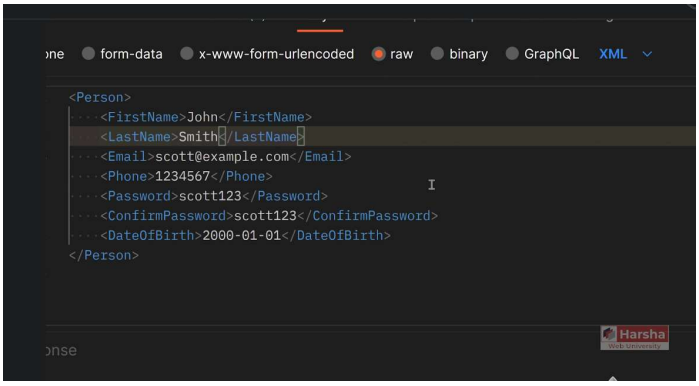
So, for this type of complex operation, you have to create a custom model binder in real-world projects. In other words, whenever you would like to perform a complex manipulation on data types other than the built-in data types, such as custom data types, structures, or enumerations, then you have to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

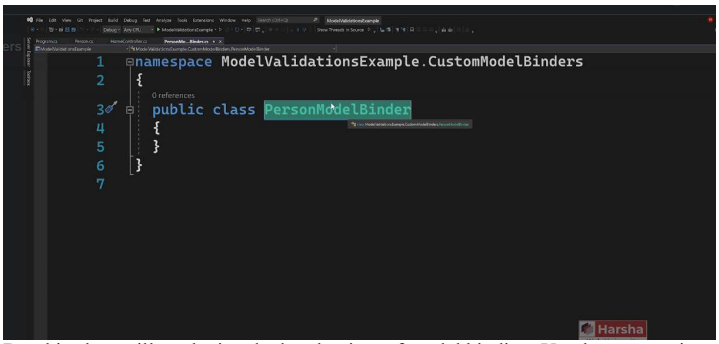
In real-world project, it is rear to use custom-model binding.

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



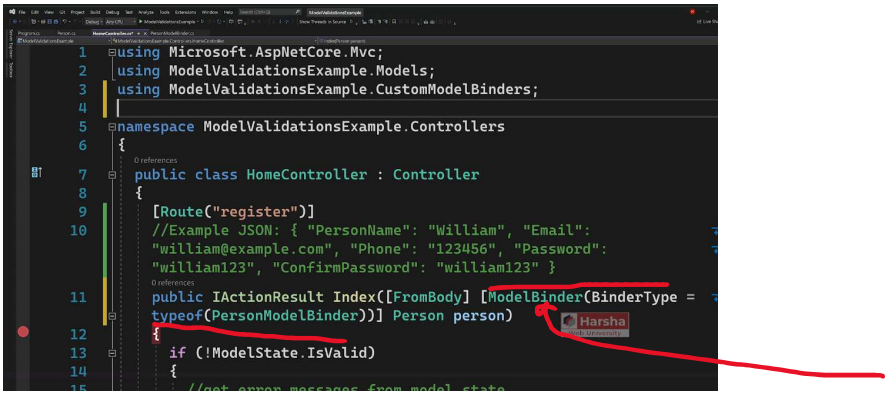


Lets's see how you can solve this problem using custom-model binding.

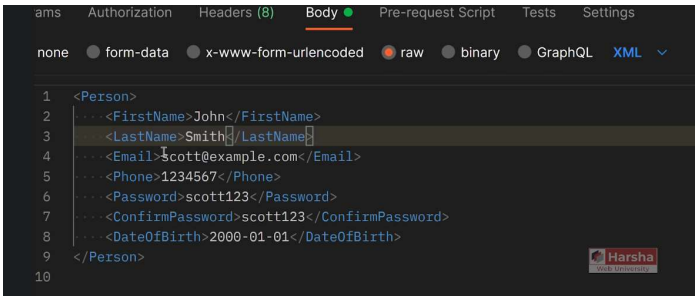


But this class will not be invoked at the time of model binding. You have to set it explicitly.

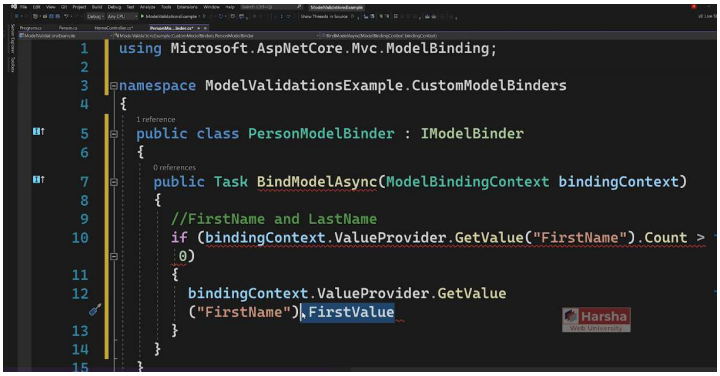
At the home controller, add a new attribute.

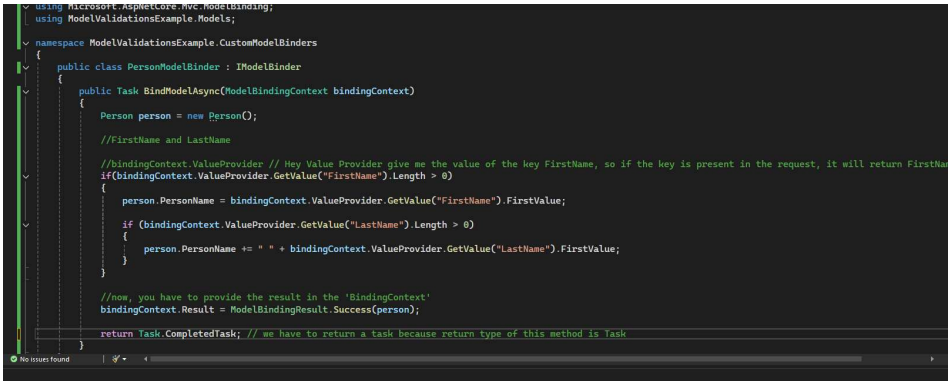


There might be duplicate 'FirstName' and 'LastName' exists in the request.



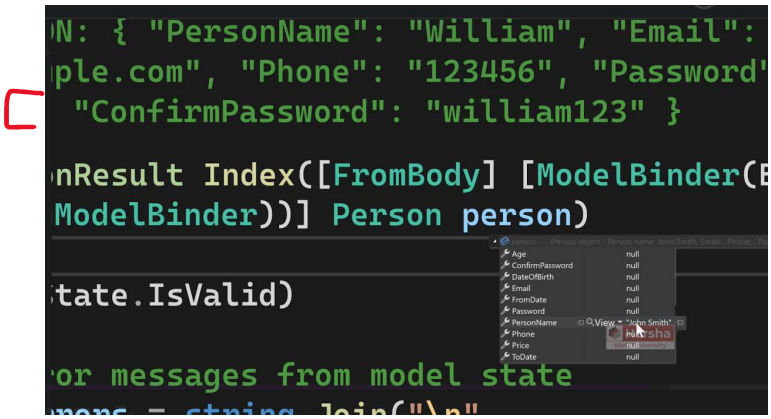
I am saying here is that, if duplicate value is exist, I am only interested in only first value.

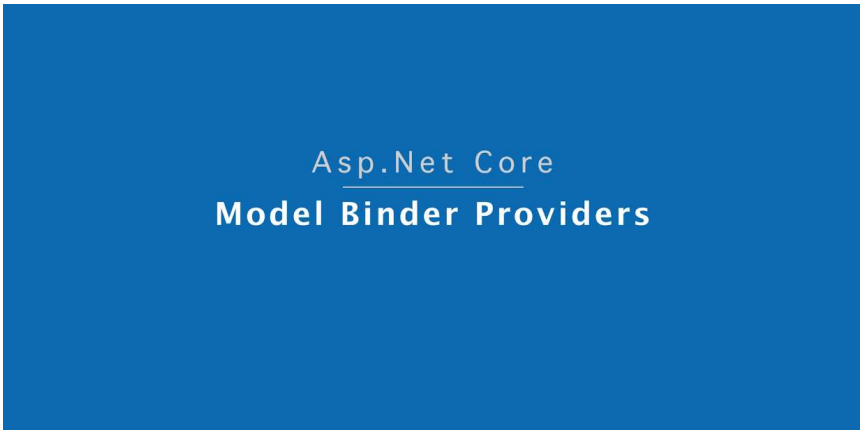




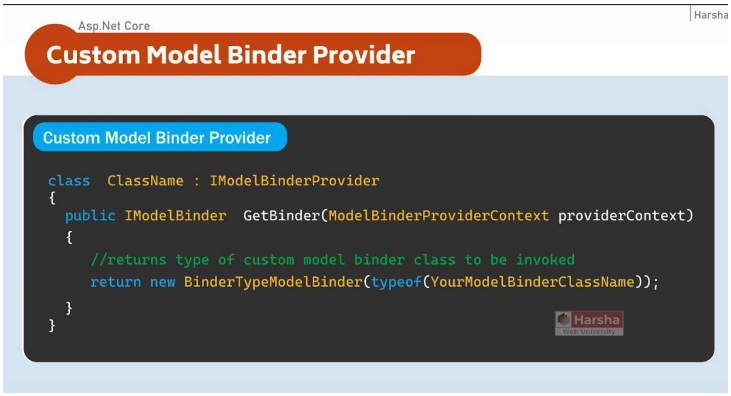
Now run this application. From Postman, click on Send button.

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

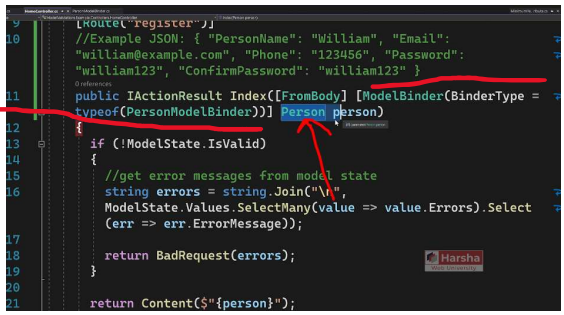


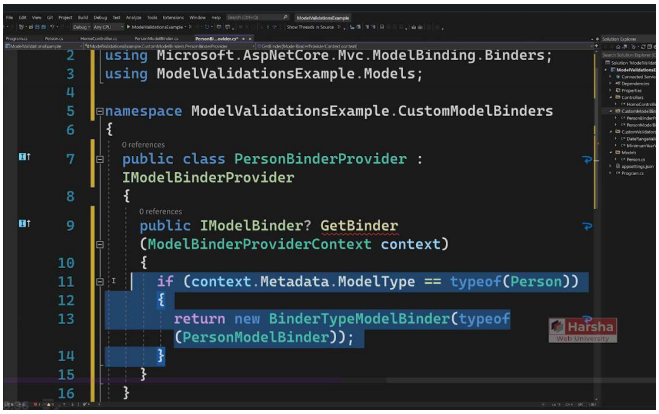


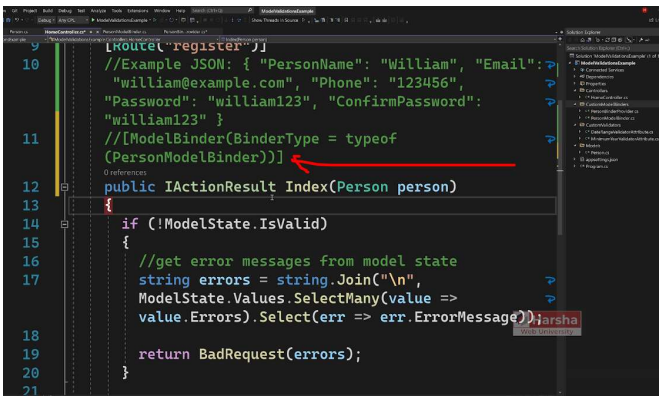
Suppose you would like to use the same custom model binder for all the action methods wherever this type of model class is used. Then you can declare it globally by using a binder provider.
You can create a custom model binder provider like this, and you can return the type of the model binder class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



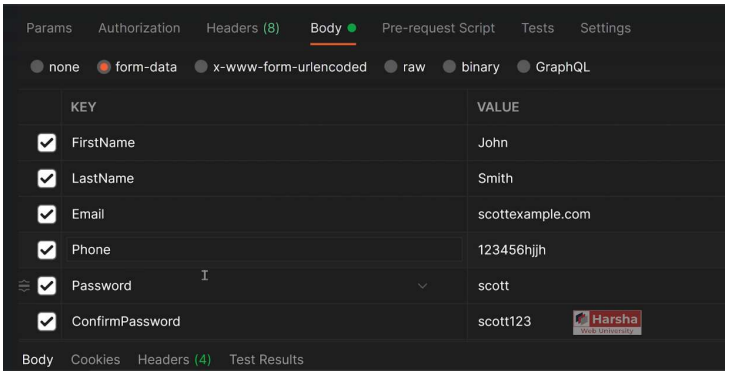
That means any action method that has a parameter of the Person type would automatically use the specified custom model binder. This eliminates the need to explicitly define the model binder in every action method, which is the actual benefit.
Of course, this approach might not be very useful for simple 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.


[Route("register")]
//Example JSON: { "PersonName": "William", "Email":
"william@example.com", "Phone": "123456", "Password":
"william123", "ConfirmPassword": "william123" }
0 references
public IActionResult Index([FromBody] [ModelBinder(BinderType =
typeof(PersonModelBinder))] 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);
}
return Content($"{person}");
Harsha

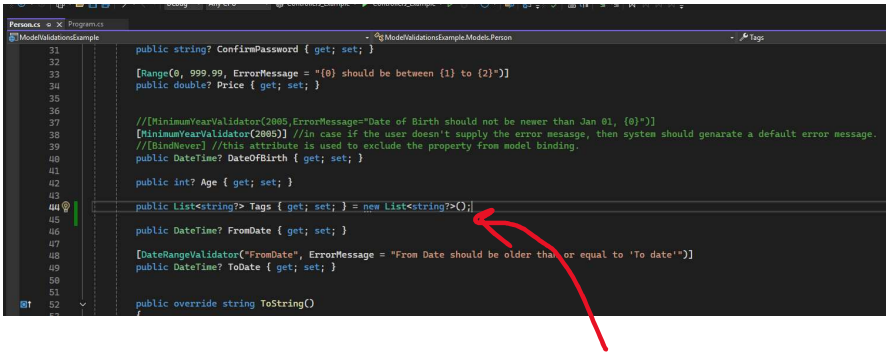

using Microsoft.AspNetCore.Mvc.ModelBinding.Binders;
using ModelValidationsExample.Models;
namespace ModelValidationsExample.CustomModelBinders
{
0 references
public class PersonBinderProvider :
IModelBinderProvider
{
0 references
public IModelBinder? GetBinder
(ModelBinderProviderContext context)
{
if (context.Metadata.ModelType == typeof(Person))
{
return new BinderTypeModelBinder(typeof
(PersonModelBinder));
}
}
}
Harsha


[Route("register")]
//Example JSON: { "PersonName": "William", "Email":
"william@example.com", "Phone": "123456",
"Password": "william123", "ConfirmPassword":
"william123" }
//[ModelBinder(BinderType = typeof
(PersonModelBinder))]
0 references
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);
}
Harsha

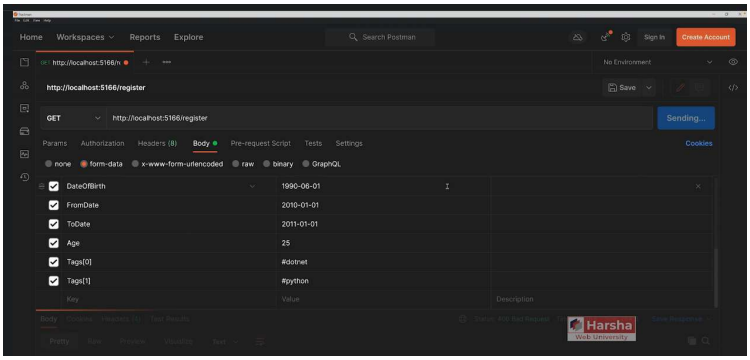

Asp.Net Core
Collection Binding

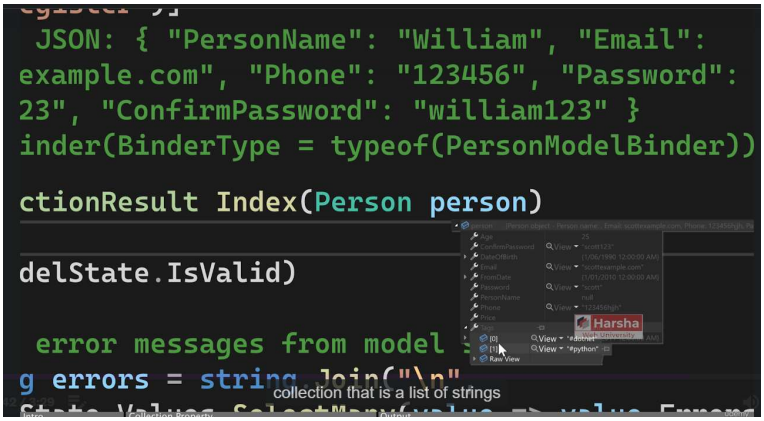


Sometimes, you might want to send more than one value for the same key.
For example, let's say there is a **phone** field, and you want to send more than one phone number. In such a case, how can you receive those values in the model class? Let's try.

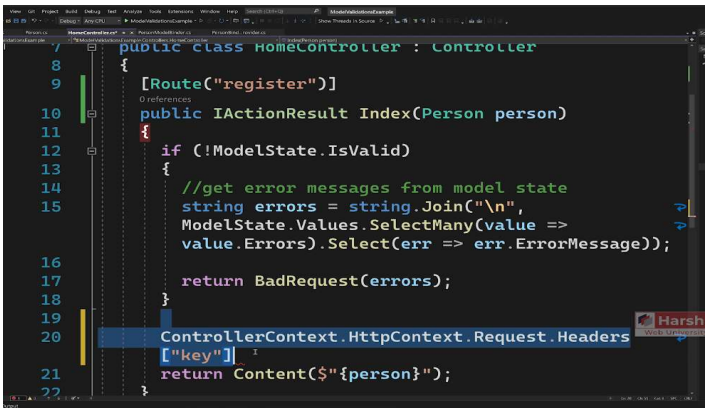


In this way, we can submit more than one value

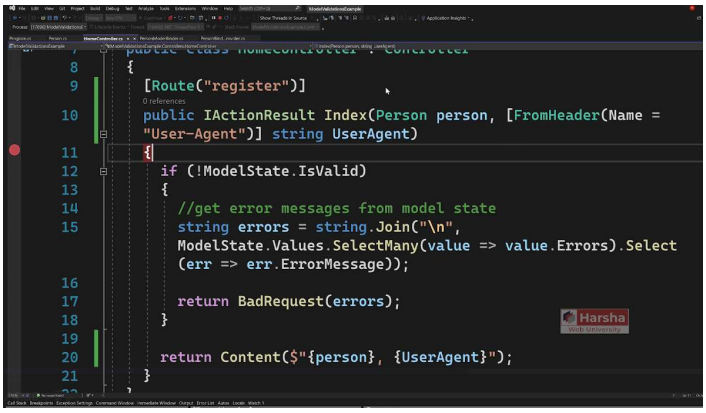




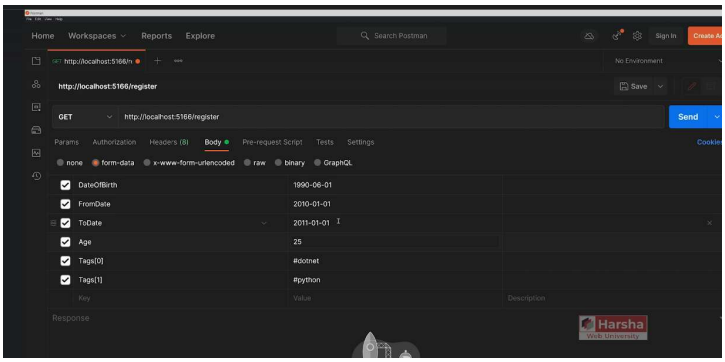
Sometimes, you may want to read the request headers as part of the model binding.
As of now, we can read inputs from the request, such as form fields, request body (JSON or XML), route data, or query strings.
Lastly, you can also receive values from the request headers.

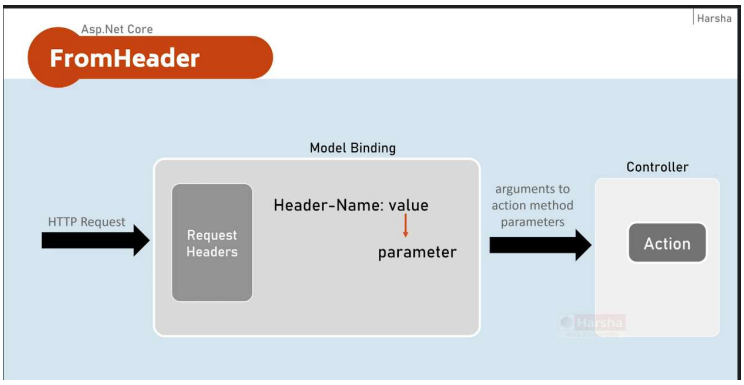


The traditional way to read the request headers is to write:
ControllerContext.HttpContext.Request.Headers
Here, you can specify the header key name to retrieve its value. This is how you can retrieve the value from request headers in traditional code.
But instead of this lengthy approach.



Now run on postman.


Home
Workspaces
Reports
Explore
Search Postman
Sign In
Create A
http://localhost:5166/register
No Environment
Save
GET
http://localhost:5166/register
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
DateOfBirth
1990-06-01
FromDate
2010-01-01
ToDate
2011-01-01
Age
25
Tags[0]
#dotnet
Tags[1]
#python
Key
Value
Description
Response
Harsha
Web University


Harsha
Asp.Net Core
FromHeader
Model Binding
Controller
HTTP Request
Request Headers
Header-Name: value
parameter
arguments to action method parameters
Action